Introduction to Complexity Theory
Lecture Notes for Second Semester

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, ISRAEL.
Email: oded@wisdom.weizmann.ac.il

July 1999

©Copyright 1999 by Oded Goldreich.
Permission to make copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that new copies bear this notice and the full citation on the first page. Abstracting with credit is permitted.

II

Preface

Complexity Theory is a central field of Theoretical Computer Science, with a remarkable list of celebrated achievements as well as a very vibrant present research activity. The field is concerned with the study of the *intrinsic* complexity of computational tasks, and this study tend to aim at *generality*: It focuses on natural computational resources, and the effect of limiting those on the *class of problems* that can be solved.

These lecture notes were taken by students attending my year-long introductory course on Complexity Theory, given in 1998–99 at the Weizmann Institute of Science. The course was aimed at exposing the students to the basic results and research directions in the field. The focus was on concepts and ideas, and complex technical proofs were avoided. Specific topics included:

- Revisiting NP and NPC (with emphasis on search vs decision);
- Complexity classes defined by one resource-bound – hierarchies, gaps, etc;
- Non-deterministic Space complexity (with emphasis on NL);
- Randomized Computations (e.g., ZPP, RP and BPP);
- Non-uniform complexity (e.g., P/poly, and lower bounds on restricted circuit classes);
- The Polynomial-time Hierarchy;
- The counting class #P, approximate-#P and uniqueSAT;
- Probabilistic proof systems (i.e., IP, PCP and ZK);
- Pseudorandomness (generators and derandomization);
- Time versus Space (in Turing Machines);
- Circuit-depth versus TM-space (e.g., AC, NC, SC);
- Average-case complexity;

It was assumed that students have taken a course in computability, and hence are familiar with Turing Machines.

Most of the presented material is quite independent of the specific (reasonable) model of computation, but some (e.g., Lectures 5, 16, and 19–20) depends heavily on the locality of computation of Turing machines.

State of these notes

These notes are neither complete nor fully proofread, let alone being far from uniformly well-written (although the notes of some lectures are quite good). Still, I do believe that these notes suggest a good outline for an *introduction to complexity theory* course.

Using these notes

A total of 26 lectures were given, 13 in each semester. In general, the pace was rather slow, as most students were first year graduates and their background was quite mixed. In case the student body is *uniformly more advanced* one should be able to cover much more in one semester. Some concrete comments for the teacher follow

- Lectures 1 and 2 revisit the P vs NP question and NP-completeness. The emphasis is on presenting NP in terms of search problems, on the fact that the mere existence of NP-complete sets is interesting (and easily demonstratable), and on reductions applicable also in the domain of search problems (i.e., Levin reductions). A good undergraduate computability course should cover this material, but unfortunately this is often not the case. Thus, I suggest to give Lectures 1 and 2 if and only if the previous courses taken by the students failed to cover this material.

- There is something anal in much of Lectures 3 and 5. One may prefer to shortly discuss the material of these lectures (without providing proofs) rather than spend 4 hours on them. (Note that many statements in the course are given without proof, so this will not be an exception.)

- One should be able to merge Lectures 13 and 14 into a single lecture (or at most a lecture and a half). I failed to do so due to inessential reasons. Alternatively, may merge Lectures 13–15 into two lectures.

- Lectures 21–23 were devoted to communication complexity, and circuit depth lower bounds derived via communication complexity. Unfortunately, this sample fails to touch upon other important directions in circuit complexity (e.g., size lower bound for AC0 circuits). I would recommend to try to correct this deficiency.

- Lecture 25 was devoted to Computational Learning Theory. This area, traditionally associated with "algorithms", does have a clear "complexity" flavour.

- Lecture 26 was spent discussing the (limited in our opinion) meaningfulness of relativization results. The dilemma of whether to discuss something negative or just ignore it is never easy.

- Many interesting results were not covered. In many cases this is due to the trade-off between their conceptual importance as weighted against their technical difficulty.

Bibliographic Notes

There are several books which cover small parts of the material. These include:

1. Garey, M.R., and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

2. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998. Copies have been placed in the faculty's library.

3. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

4. M. Sipser. *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

However, the presentation of material in these lecture notes does not necessarily follow these sources.

Each lecture is planned to include bibliographic notes, but this intension has been only partially fulfilled so far.

Acknowledgments

I am most grateful to the students who have attended the course and partipiated in the project of preparing the lecture notes. So thanks to *Sergey Benditkis, Reshef Eilon, Michael Elkin, Amiel Ferman, Dana Fisman, Danny Harnik, Tzvika Hartman, Tal Hassner, Hillel Kugler, Oded Lachish, Moshe Lewenstein, Yehuda Lindell, Yoad Lustig, Ronen Mizrahi, Leia Passoni, Guy Peer, Nir Piterman, Ely Porate, Yoav Rodeh, Alon Rosen, Vered Rosen, Noam Sadot, Il'ya Safro, Tamar Seeman, Ekaterina Sedletsky, Reuben Sumner, Yael Tauman, Boris Temkin, Erez Waisbard,* and *Gera Weiss.*

I am grateful to Ran Raz and Dana Ron who gave guess lectures during the course: Ran gave Lectures 21–23 (on communication complexity and circuit complexity), and Dana gave Lecture 25 (on computational learning theory).

Thanks also to *Paul Beame*, *Ruediger Reischuk* and *Avi Wigderson* who have answered some questions I've had while preparing this course.

Lecture Summaries

Lecture 1: The P vs NP Question. We review the fundamental question of computer science, known as the $\mathcal{P}$ versus $\mathcal{NP}$ question: Given a problem whose solution can be verified efficiently (i.e., in polynomial time), is there necessarily an efficient method to actually *find* such a solution? Loosely speaking, the first condition (i.e., efficient verification) is captured in the definition of $\mathcal{NP}$, and the second in that of $\mathcal{P}$. The actual correspondence relies on the notion of *self-reducibility*, which relates the complexity of determining whether a solution exists to the complexity of actually finding one.

Notes taken by Eilon Reshef.

Lecture 2: NP-completeness and Self Reducibility. We prove that any relation defining an NP-complete language is self-reducible. This will be done using the SAT self-reducibility (proved in Lecture 1), and the fact that SAT is NP-Hard under Levin Reductions. The latter are Karp Reductions augmented by efficient transformations of NP-witnesses from the original instance to the reduced one, and vice versa. Along the way, we give a simple proof of the existence of NP-Complete languages (by proving that Bounded Halting is NP-Complete).

Notes taken by Nir Piterman and Dana Fisman.

Lecture 3: More on NP and some on DTIME. In the first part of this lecture we discuss two properties of the complexity classes P, NP and NPC: The property is that NP contains problems which are neither NP-complete nor in P (provided NP $\neq$ P), and the second one is that NP-relations have optimal search algorithms. In the second part we define new complexity classes based on exact time bounds, and consider some relations between them. We point out the sensitivity of these classes to the specific model of computation (e.g., one-tape versus two-tape Turing machines).

Notes taken by Michael Elkin and Ekaterina Sedletsky.

Lecture 4: Space Complexity. We define "nice" complexity bounds; these are bounds which can be computed within the resources they supposedly bound (e.g., we focus on time-constructible and space-constructible bounds). We define space complexity using an adequate model of computation in which one is not allowed to use the area occupied by the input for computation. Before dismissing sub-logarithmic space, we present two results regarding it (contrasting sub-loglog space with loglog space). We show that for "nice" complexity bounds, there is a hierarchy of complexity classes – the more resources one has the more tasks one can perform. One the other hand, we mention that this increase in power may not happen if the complexity bounds are not "nice".

Notes taken by Leia Passoni and Reuben Sumner.

Lecture 5: Non-Deterministic Space. We recall two basic facts about deterministic space complexity, and then define non-deterministic space complexity. Three alternative models for measuring non-deterministic space complexity are introduced: the standard non-deterministic model, the online model and the offline model. The equivalence between the non-deterministic and online models and their exponential relation to the offline model are proved. We then turn to investigate the relation between the non-deterministic and deterministic space complexity (i.e., Savitch's Theorem).

Notes taken by Yoad Lustig and Tal Hassner.

Lecture 6: Non-Deterministic Logarithmic Space We further discuss composition lemmas underlying previous lectures. Then we study the complexity class $\mathcal{NL}$ (the set of languages decidable within Non-Deterministic Logarithmic Space): We show that directed graph connectivity is complete for $\mathcal{NL}$. Finally, we prove that $\mathcal{NL} = \text{co}\mathcal{NL}$ (i.e., $\mathcal{NL}$ class is closed under complementation).

Notes taken by Amiel Ferman and Noam Sadot.

Lecture 7: Randomized Computations We extend the notion of efficient computation by allowing algorithms (Turing machines) to toss coins. We study the classes of languages that arise from various natural definitions of acceptance by such machines. We focus on probabilistic polynomial-time machines with one-sided, two-sided and zero error probability (defining the classes $\mathcal{RP}$ (and co$\mathcal{RP}$), $\mathcal{BPP}$ and $\mathcal{ZPP}$). We also consider probabilistic machines that uses logarithmic spaces (i.e., the class $\mathcal{RL}$).

Notes taken by Erez Waisbard and Gera Weiss.

Lecture 8: Non-Uniform Polynomial Time ($\mathcal{P}$/Poly). We introduce the notion of non-uniform polynomial-time and the corresponding complexity class $\mathcal{P}$/poly. In this (somewhat fictitious) computational model, Turing machines are provided an external advice string to aid them in their computation (on strings of certain length). The *non-uniformity* is expressed in the fact that an arbitrary advice string may be defined for every different length of input. We show that $\mathcal{P}$/poly "upper bounds" the notion of efficient computation (as $\mathcal{BPP} \subseteq \mathcal{P}$/poly), yet this upper bound is not tight (as $\mathcal{P}$/poly contains non-recursive languages). The effect of introducing *uniformity* is discussed, and shown to collapse $\mathcal{P}$/poly to $\mathcal{P}$. Finally, we relate the $\mathcal{P}$/poly versus $\mathcal{NP}$ question to the question of whether NP-completeness via Cook-reductions is more powerful that NP-completeness via Karp-reductions. This is done by showing, on one hand, that $\mathcal{NP}$ is Cook-reducible to a sparse set iff $\mathcal{NP} \subset \mathcal{P}$/poly, and on the other hand that $\mathcal{NP}$ is Karp-reducible to a sparse set iff $\mathcal{NP} = \mathcal{P}$.

Notes taken by Moshe Lewenstein, Yehuda Lindell and Tamar Seeman.

Lecture 9: The Polynomial Hierarchy (PH). We define a hierarchy of complexity classes extending $\mathcal{NP}$ and contained in PSPACE. This is done in two ways, shown equivalent: The first by generalizing the notion of Cook reductions, and the second by generalizing the definition of $\mathcal{NP}$. We then relate this hierarchy to complexity classes discussed in previous lectures such as $\mathcal{BPP}$ and $\mathcal{P}$/Poly: We show that $\mathcal{BPP}$ is in PH, and that if $\mathcal{NP} \subseteq \mathcal{P}$/poly then PH collapses to is second level.

Notes taken by Ronen Mizrahi.

Lecture 10: The counting class $\#\mathcal{P}$. The class $\mathcal{NP}$ captures the difficulty of determining whether a given input has a solution with respect to some (tractable) relation. A potentially harder question, captured by the class $\#\mathcal{P}$, refers to determining the number of such solutions. We first define the complexity class $\#\mathcal{P}$, and classify it with respect to other complexity classes. We then prove the existence of $\#\mathcal{P}$-complete problems, and mention some natural ones. Then we try to study the relation between $\#\mathcal{P}$ and $\mathcal{NP}$ more exactly, by showing we can probabilistically approximate $\#\mathcal{P}$ using an oracle in $\mathcal{NP}$. Finally, we refine this result by restricting the oracle to a weak form of SAT (called $uniqueSAT$).

Notes taken by Oded Lachish, Yoav Rodeh and Yael Tauman.

Lecture 11: Interactive Proof Systems. We introduce the notion of interactive proof systems and the complexity class IP, emphasizing the role of randomness and interaction in this model. The concept is demonstrated by giving an interactive proof system for Graph Non-Isomorphism. We discuss the power of the class IP, and prove that co$\mathcal{NP} \subseteq \mathcal{IP}$. We discuss issues regarding the number of rounds in a proof system, and variants of the model such as public-coin systems (a.k.a. Arthur-Merlin games).

Notes taken by Danny Harnik, Tzvika Hartman and Hillel Kugler.

Lecture 12: Probabilistically Checkable Proof (PCP). We introduce the notion of Probabilistically Checkable Proof (PCP) systems. We discuss some complexity measures involved, and describe the class of languages captured by corresponding PCP systems. We then demonstrate the alternative view of $\mathcal{NP}$ emerging from the PCP Characterization Theorem, and use it in order to prove non-approximability results for the problems $max3SAT$ and $maxCLIQUE$.

Notes taken by Alon Rosen and Vered Rosen.

Lecture 13: Pseudorandom Generators. Pseudorandom generators are defined as efficient deterministic algorithms which stretch short random seeds into longer pseudorandom sequences. The latter are indistiguishable from truely random sequences by any efficient observer. We show that, for efficiently sampleable distributions, computational indistiguishability is preserved under multiple samples. We related pseudorandom generators and one-way functions, and show how to increase the stretching of pseudorandom generators. The notes are augmented by an essay of Oded.

Notes taken by Sergey Benditkis, Il'ya Safro and Boris Temkin.

Lecture 14: Pseudorandomness and Computational Difficulty . We continue our discussion of pseudorandomness and show a connection between pseudorandomness and computational difficulty. Specifically, we show how the difficulty of inverting one-way functions may be utilized to obtain a pseudorandom generator. Finally, we state and prove that a hard-to-predict bit (called a hard-core) may be extracted from any one-way function. The hard-core is fundamental in our construction of a generator.

Notes taken by Moshe Lewenstein and Yehuda Lindell.

Lecture 15: Derandomization of BPP. We present an efficient deterministic simulation of randomized algorithms. This process, called derandomization, introduce new notions of pseudorandom generators. We extend the definition of pseudorandom generators and show how to construct a generator that can be used for derandomization. The new construction differ from the generator that constructed in the previous lecture in it's running time (it will run slower, but fast enough for the simulation). The benefit is that it is relying on a seemingly weaker assumption.

Notes taken by Erez Waisbard and Gera Weiss.

Lecture 16: Derandomizing Space-Bounded Computations. We consider derandomization of space-bounded computations. We show that $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^2 n)$, namely, any bounded-probability Logspace algorithm can be deterministically emulated in $O(\log^2 n)$ space. We further show that $\mathcal{BPL} \subseteq \mathcal{SC}$, namely, any such algorithm can be deterministically emulated in $O(\log^2 n)$ space and (simultaneously) in polynomial time.

Notes taken by Eilon Reshef.

Lecture 17: Zero-Knowledge Proof Systems. We introduce the notion of zero-knowledge interactive proof system, and consider an example of such a system (Graph Isomorphism). We define perfect, statistical and computational zero-knowledge, and present a method for constructing zero-knowledge proofs for NP languages, which makes essential use of bit commitment schemes. We mention that zero-knowledge is preserved under sequential composition, but is not preserved under the parallel repetition.

Notes taken by Michael Elkin and Ekaterina Sedletsky.

Lecture 18: NP in PCP[poly,O(1)]. The main result in this lecture is $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$. In the course of the proof we introduce an $\mathcal{NPC}$ language "Quadratic Equations", and show it to be in $\mathcal{PCP}(poly, O(1))$. The argument proceeds in two stages: First assuming properties of the proof (oracle), and then testing these properties. An intermediate result that of independent interest is an efficient probabilistic algorithm that distinguishes between linear and far-from-linear functions.

Notes taken by Yoad Lustig and Tal Hassner.

Lecture 19: Dtime(t) contained in Dspace(t/log t). We prove that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$. That is, we show how to simulate any given deterministic multi-tape Turing Machine (TM) of time complexity t, using a deterministic TM of space complexity $t/\log t$. A main ingrediant in the simulation is the analysis of a pebble game on directed bounded-degree graphs.

Notes taken by Tamar Seeman and Reuben Sumner.

Lecture 20: Circuit Depth and Space Complexity. We study some of the relations between Boolean circuits and Turing machines. We define the complexity classes $\mathcal{NC}$ and $\mathcal{AC}$, compare their computational power, and point out the possible connection between uniform-$\mathcal{NC}$ and "efficient" parallel computation. We conclude the discussion by establishing a strong connection between space complexity and depth of circuits with bounded fan-in.

Notes taken by Alon Rosen and Vered Rosen.

Lecture 21: Communication Complexity. We consider Communication Complexity – the analysis of the amount of information that needs to be communicated betwen two parties which wish to reach a common computational goal. We start with some basic definitions, considering both deterministic and probabilistic models for the problem, and annotating our discussion with a few examples. Next we present a couple of tools for proving lower bounds on the complexity of communication problems. We conclude by proving a linear lower bound on the communication complexity of probabilistic protocols for computing the inner product of two vectors, where initially each party holds one vector.

Notes taken by Amiel Ferman and Noam Sadot.

Lecture 22: Circuit Depth and Communication Complexity. The main result presented in this lecture is a (tight) nontrivial lower bound on the monotone circuit depth of s-t-Connectivity. This is proved via a series of reductions, the first of which is of significant importance: A connection between circuit depth and communication complexity. We then get a communication game and proceed to reduce it to other such games, until reaching a game called FORK. We conclude that a lower bound on the communication complexity of FORK, to be given in the next lecture, will yield an analogous lower bound on the monotone circuit depth of s-t-Connectivity.

Notes taken by Yoav Rodeh and Yael Tauman.

Lecture 23: Depth Lower Bound for Monotone Circuits (cont.). We analyze the FORK game, introduced in the previous lecture. We give tight lower and upper bounds on the communication needed in a protocol solving FORK. This completes the proof of the lower bound on the depth of monotone circuits computing the function st-Connectivity.

Notes taken by Dana Fisman and Nir Piterman.

Lecture 24: Average-Case Complexity. We introduce a theory of average-case complexity which refers to computational problems coupled with probability distributions. We start by defining and discussing the classes of P-computable and P-samplable distributions. We then define the class DistNP (which consists of NP problems coupled with P-computable distributions), and discuss the notion of average polynomial-time (which is unfortunately more subtle than it may seem). Finally, we define and discuss reductions between distributional problems. We conclude by proving the existence of a complete problem for DistNP.

Notes taken by Tzvika Hartman and Hillel Kugler.

Lecture 25: Computational Learning Theory. We define a model of automoatic learning called probably approximately correct (PAC) learning. We define efficient PAC learning, and present several efficient PAC learning algorithms. We prove the Occam's Razor Theorem, which reduces the PAC learning problem to the problem of finding a succinct representation for the values of a large number of given labeled examples.

Notes taken by Oded Lachish and Eli Porat.

XIV

Lecture 26: Relativization. In this lecture we deal with relativization of complexity classes. In particular, we discuss the role of relativization with respect to the $\mathcal{P}$ vs. $\mathcal{NP}$ question; that is, we shall see that for some oracle A, $\mathcal{P}^A = \mathcal{NP}^A$, whereas for another A (actually for almost all other A's) $\mathcal{P}^A \neq \mathcal{NP}^A$. However, it also holds that $\mathcal{IP}^A \neq \mathcal{PSPACE}^A$ for a random A, whereas $\mathcal{IP} = \mathcal{PSPACE}$

Notes taken by Leia Passoni.

Contents

Preface **III**

Acknowledgments **VII**

Lecture Summaries **IX**

14 Pseudorandomness and Computational Difficulty **189**
14.1 Introduction . . . 189
14.2 Definitions . . . 190
14.3 A Pseudorandom Generator based on a 1-1 One-Way Function . . . 192
14.4 A Hard-Core for Any One-Way Function . . . 194
Bibliographic Notes . . . 197

15 Derandomization of BPP **199**
15.1 Introduction . . . 199
15.2 New notion of Pseudorandom generator . . . 201
15.3 Construction of non-iterative pseudorandom generator . . . 202
15.3.1 Parameters . . . 203
15.3.2 Tool 1: An unpredictable predicate . . . 203
15.3.3 Tool 2: A design . . . 204
15.3.4 The construction itself . . . 205
15.4 Constructions of a design . . . 208
15.4.1 First construction: using $GF(l)$ arithmetic . . . 208
15.4.2 Second construction: greedy algorithm . . . 209
Bibliographic Notes . . . 211

16 Derandomizing Space-Bounded Computations **213**
16.1 Introduction . . . 213
16.2 The Model . . . 213
16.3 Execution Graphs . . . 214
16.4 Universal Hash Functions . . . 216
16.5 Construction Overview . . . 217
16.6 The Pseudorandom Generator . . . 217
16.7 Analysis . . . 219
16.8 Extensions and Related Results . . . 223
16.8.1 $\mathcal{BPL} \subseteq \mathcal{SC}$ . . . 223
16.8.2 Further Results . . . 224

Bibliographic Notes . . . 224

17 Zero-Knowledge Proof Systems **225**
17.1 Definitions and Discussions . . . 225
17.2 Graph Isomorphism is in Zero-Knowledge . . . 230
17.3 Zero-Knowledge Proofs for NP . . . 235
17.3.1 Zero-Knowledge NP-proof systems . . . 235
17.3.2 NP $\subseteq$ ZK (overview) . . . 236
17.3.3 Digital implementation . . . 240
17.4 Various comments . . . 244
17.4.1 Remark about parallel repetition . . . 244
17.4.2 Remark about randomness in zero-knowledge proofs . . . 245
Bibliographic Notes . . . 245

18 NP in PCP[poly,O(1)] **247**
18.1 Introduction . . . 247
18.2 Quadratic Equations . . . 248
18.3 The main strategy and a tactical maneuver . . . 249
18.4 Testing satisfiability assuming a nice oracle . . . 251
18.5 Distinguishing a nice oracle from a very ugly one . . . 253
18.5.1 Tests of linearity . . . 253
18.5.2 Assuming linear π testing π's coefficients structure . . . 255
18.5.3 Gluing it all together . . . 258
Bibliographic Notes . . . 258
Appendix A: Linear functions are far apart . . . 259
Appendix B: The linearity test for functions far from linear . . . 260

19 Dtime vs Dspace **263**
19.1 Introduction . . . 263
19.2 Main Result . . . 264
19.3 Additional Proofs . . . 267
19.3.1 Proof of Lemma 19.2.1 (Canonical Computation Lemma) . . . 267
19.3.2 Proof of Theorem 19.4 (Pebble Game Theorem) . . . 268
Bibliographic Notes . . . 270

20 Circuit Depth and Space Complexity **271**
20.1 Boolean Circuits . . . 271
20.1.1 The Definition . . . 271
20.1.2 Some Observations . . . 272
20.1.3 Families of Circuits . . . 272
20.2 Small-depth circuits . . . 273
20.2.1 The Classes NC and AC . . . 274
20.2.2 Sketch of the proof of $AC^0 \subset NC^1$ . . . 275
20.2.3 NC and Parallel Computation . . . 277
20.3 On Circuit Depth and Space Complexity . . . 278
Bibliographic Notes . . . 281

21 Communication Complexity **283**
21.1 Introduction . . . 283
21.2 Basic model and some examples . . . 283
21.3 Deterministic versus Probabilistic Complexity . . . 284
21.4 Equality revisited and the Input Matrix . . . 285
21.5 Rank Lower Bound . . . 288
21.6 Inner-Product lower bound . . . 289
Bibliographic Notes . . . 292

22 Monotone Circuit Depth and Communication Complexity **293**
22.1 Introduction . . . 293
22.1.1 Hard Functions Exist . . . 294
22.1.2 Bounded Depth Circuits . . . 295
22.2 Monotone Circuits . . . 295
22.3 Communication Complexity and Circuit Depth . . . 297
22.4 The Monotone Case . . . 299
22.4.1 The Analogous Game and Connection . . . 300
22.4.2 An Equivalent Restricted Game . . . 301
22.5 Two More Games . . . 303
Bibliographic Notes . . . 305

23 The FORK Game **307**
23.1 Introduction . . . 307
23.2 The FORK game – recalling the definition . . . 308
23.3 An upper bound for the FORK game . . . 308
23.4 A lower bound for the FORK game . . . 309
23.4.1 Definitions . . . 309
23.4.2 Reducing the density . . . 310
23.4.3 Reducing the length . . . 311
23.4.4 Applying the lemmas to get the lower bound . . . 314
Bibliographic Notes . . . 314

24 Average Case Complexity **315**
24.1 Introduction . . . 315
24.2 Definitions . . . 316
24.2.1 Distributions . . . 316
24.2.2 Distributional Problems . . . 316
24.2.3 Distributional Classes . . . 316
24.2.4 Distributional-NP . . . 318
24.2.5 Average Polynomial Time . . . 318
24.2.6 Reductions . . . 319
24.3 DistNP-completeness . . . 319
Bibliographic Notes . . . 322
Appendix A : Failure of a naive formulation . . . 322
Appendix B : Proof Sketch of Proposition 24.2.4 . . . 323

25 Computational Learning Theory **327**
25.1 Towards a definition of Computational learning . . . 327
25.2 Probably Approximately Correct (PAC) Learning . . . 329
25.3 Occam's Razor . . . 332
25.4 Generalized definition of PAC learning algorithm . . . 336
25.4.1 Reductions among learning tasks . . . 336
25.4.2 Generalized forms of Occam's Razor . . . 337
25.5 The (VC) Vapnik-Chervonenkis Dimension . . . 338
25.5.1 An example: VC dimension of axis aligned rectangles . . . 339
25.5.2 General bounds . . . 340
Bibliographic Notes . . . 342
Appendix: Filling-up gaps for the proof of Claim 25.2.1 . . . 342

26 Relativization **343**
26.1 Relativization of Complexity Classes . . . 343
26.2 The $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ question Relativized . . . 344
26.3 Relativization with a Random Oracle . . . 348
26.4 Conclusions . . . 351
Bibliographic Notes . . . 352

Lecture 14

Pseudorandomness and Computational Difficulty

Notes taken by Moshe Lewenstein and Yehuda Lindell

Summary: In this lecture we continue our discussion of pseudorandomness and show a connection between pseudorandomness and computational difficulty. More specifically we show how the difficulty of inverting one-way functions may be utilized to obtain a pseudorandom generator. Finally, we state and prove that a hard-to-predict bit (called a hard-core) may be extracted from any one-way function. The hard-core is fundamental in our construction of a generator.

14.1 Introduction

The main theme of this lecture is the utilization of one-way functions in order to construct a pseudorandom generator. Intuitively, a one-way function is a function that is *easy* to compute and *hard* to invert. Generally, "easy" refers to polynomial time and "hard" to the fact that success in the *average case* requires more than polynomial time (for any polynomial). It is critical that the difficulty be in the average case and *not* in the worst case, as with $\mathcal{NP}$-Complete problems. This will become clear later.

How can one-way functions help us construct a pseudorandom generator? The answer lies in the property of unpredictability. This concept will be formalized in the coming lectures but for now we will discuss it informally. Assume that we have a string s and we begin to scan it in some given (computable) order. If at some stage we can predict the next bit with probability significantly greater than one half, then the string is clearly not random (because for a random string, each bit is chosen *independently* with probability exactly one half). On the other hand, it can be shown that if we *cannot* predict any "next" bit during our scan with success significantly more than $\frac{1}{2}$, then the string is pseudorandom.

In this light, the use of computationally difficult problems becomes clear. We rely on the difficulty of inverting a one-way function f. More specifically we show that there exists a function $b : \{0,1\}^n \rightarrow \{0,1\}$ such that given x it is easy to compute $b(x)$ yet given only $f(x)$ it is difficult. This function is formally called a hard-core of f. Now, although given $f(x)$, $b(x)$ is fully and deterministically defined, we have no way of finding or predicting its value. Therefore, the computational difficulty of finding $b(x)$ provides us with an unpredictable bit which forms the basis of our generator.

14.2 Definitions

We now recall the concepts necessary for this lecture.

Definition 14.1 Pseudorandom generators: *G is a* pseudorandom generator *if:*

1. *G operates in (deterministic) polynomial time*
2. *For every s, $|G(s)| > |s|$ (w.l.o.g., assume that $\exists l(\cdot)$ such that $\forall s \in \{0,1\}^n \ |G(s)| = l(n)$).*
3. *$\{G(U_n)\}$ and $\{U_{l(n)}\}$ are probabilistic polynomial time indistinguishable (where U_n is the uniform distribution over $\{0,1\}^n$).*

Definition 14.2 One-way functions: *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a length-preserving function (i.e. $\forall x \ |f(x)| = |x|$). Then f is* one-way *if:*

1. *f is "easy" to compute. Formally, there exists a polynomial time algorithm A such that $\forall x$ $A(x) = f(x)$.*
2. *f is "hard" to invert in the average case. Formally, for every probabilistic polynomial time algorithm A, for every polynomial $p(\cdot)$, and for all sufficiently large n's,*

$$Prob[A(f(U_n)) \in f^{-1}f(U_n)] < \frac{1}{p(n)}$$

The above definition refers to a length-preserving function. This is a simplifying assumption we will use throughout this lecture, but it is generally not necessary as long as $|x| = poly(|f(x)|)$. The requirement that the lengths of x and $f(x)$ be polynomially related is necessary to ensure that the difficulty involved in inverting $f(x)$ is not because x is too long. Since the inverting algorithm must work in time polynomial in $|f(x)|$, if $|f(x)|$ is logarithmic in $|x|$, no algorithm can even write x. In this case there is no computational difficult in *inverting* f and the one-wayness is due to the above technicality. Assuming that f is length-preserving avoids this problem.

As we will see, there is no requirement that f be length-preserving in the hard-core theorem stated and proved in section 4. However, the exact construction of the pseudorandom generator in section 3 relies heavily upon the length-preserving property of the one-way function and the assumption that it is $1-1$. Other constructions exist for the more general case but are more complex.

Although the definition of a one-way function guarantees that it is difficult to find the entire string x given $f(x)$, it may be very easy to obtain some of the bits of x. For example, assuming that f is one-way, consider the function $f'(\sigma \cdot x) = \sigma \cdot f(x)$, where $\sigma \in \{0,1\}$. It is easy to see that f' is also one-way. This is rigorously shown by assuming that f' is *not* one-way and showing how f can be inverted using an algorithm which inverts f'. So we see that despite the fact that f' is one-way, the first bit of the input is completely revealed.

Therefore, in order to obtain unpredictability as desired, we need a specific bit based on x, which is provably hidden given $f(x)$. This bit is called a *hard-core* of f.

Reducibility Arguments: The above-mentioned method of proof (showing that $f'(\sigma \cdot x) = \sigma \cdot f(x)$ is one-way) is called a reduction. It is worthwhile discussing this technique as it will form the basis of many proofs that we will see. The context in which it appears is when we take a certain primitive

and construct a new primitive based on it. For example, we will need it to prove our construction of a pseudorandom generator from a one-way function. Although it may be intuitively clear that a new construction is "correct" this sort of intuition may be misleading and so requires a sound proof. We do this by showing that if the newly constructed primitive is not secure, then an algorithm defying it may be used to defy the original primitive as well. More concretely, if our generator is distinguishable, then a distinguishing algorithm for it may be used to invert the one-way function used in the construction. We will see this type of argument many times in the coming lectures.

Definition 14.3 Hard-Core: *The function* $b : \{0,1\}^* \to \{0,1\}$ *is a* hard-core *of* f *if:*

1. b *is easy to compute (in the same sense as above),*
2. *For every probabilistic polynomial time algorithm* A*, for every polynomial* $p(\cdot)$*, and for all sufficiently large* n*'s,*

$$Prob[A(f(U_n)) = b(U_n)] < \frac{1}{2} + \frac{1}{p(n)}$$

Note that it is trivial to predict b with probability $\frac{1}{2}$ by simply guessing. The above definition requires that we cannot do significantly better than this guess. A hard-core of a one-way function plays an important role in cryptographic applications (as all information is hidden). However, in our case, we will use the computational difficulty involved as the basis for our construction of a pseudorandom generator. This can be seen in the following way. Given $f(x)$, $b(x)$ is fully-defined, yet completely unpredictable to any polynomially bound algorithm. This extra bit of unpredictable information is what will supply the "stretch" effect necessary for the generator.

The following claim shows that a hard-core can only exist for a one-way function. This is intuitively obvious, since if f is $1-1$ but not one-way then we can invert it and compute b. This is formally shown below.

Claim 14.2.1 *If* f *is 1-1 and polynomial time computable, and* b *is a hard-core of* f*, then* f *is one-way.*

Proof: By contradiction, assume that b is a hard-core of f, yet f is not one-way. We now show how an algorithm inverting f can be used to predict $b(x)$ from $f(x)$. Note once again the reduction technique we use.

f is not one-way, therefore there exists a probabilistic polynomial time algorithm A and a polynomial $p(\cdot)$ such that for infinitely many n's, $Prob[A(f(x)) = x] \geq \frac{1}{p(n)}$ $(|x| = n)$.

We now construct an algorithm A' for predicting b from f:

Input: y

- $x' \leftarrow A(y)$ (attempt to invert y, using A).
- If $f(x') = y$ then output $b(x')$
- Otherwise, output 0 or 1 with probability 1/2.

We now calculate the success probability of A'.

$$Prob[A'(f(x)) = b(x)] \;=\; Prob[A(f(x)) = x] \cdot 1 + Prob[A(f(x)) \neq x] \cdot \frac{1}{2}$$

$$\geq \frac{1}{p(n)} \cdot 1 \; + \; (1 - \frac{1}{p(n)}) \cdot \frac{1}{2} \quad = \quad \frac{1}{2} \; + \; \frac{1}{2p(n)}$$

The success probability of A' is greater than or equal to $\frac{1}{2} + \frac{1}{2p(n)}$ for infinitely many n's, thereby contradicting the fact that b is a hard-core of f. ■

Comment: If f is not 1-1, then the above claim is false. Consider f such that $f(x) = 0^{|x|}$. Clearly $b(x) =$ "the 1st bit of x" is a hard-core. However this is because of information theoretic considerations rather than computational bounds. The function f defined here is *trivially* inverted by taking any arbitrary string of length $|x|$ as the preimage. However, $b(x)$ may clearly not be guessed with any probability better than $\frac{1}{2}$.

14.3 A Pseudorandom Generator based on a 1-1 One-Way Function

In this section we show a construction of a pseudorandom generator given a length-preserving 1-1 one-way function. The construction is based on a hard-core of the one-way function. In the next section, we show how to generically construct a hard-core of any one-way function.

We note that constructions of a pseudorandom generator exist based on *any* one-way function (not necessarily length-preserving and 1-1). However, the constructions and proofs in the more general case are long and complicated and we therefore bring this case only.

Theorem 14.4 *Let f be a length-preserving, 1-1 one-way function and let b be a hard-core of f. Then $G(s) = f(s)b(s)$ is a pseudorandom generator (stretching the input by one bit).*

Proof: We first note that as f is length-preserving and 1-1, $f(U_n)$ is distributed uniformly over $\{0,1\}^n$ and is therefore fully random. It remains to show that for $s \in_R \{0,1\}^n$, the combination of $f(s)$ and $b(s)$ together remains indistinguishable from U_{n+1}. Intuitively, as we cannot predict $b(s)$ from $f(s)$, the string looks random to any computationally bound algorithm.

We will now show how a successful distinguishing algorithm may be used to predict b from f. This then proves that no such distinguishing algorithm exists (because b is a hard-core of f). By contradiction, assume that there exists an algorithm A and a polynomial $p(\cdot)$ such that for infinitely many n's

$$|Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(U_{n+1}) = 1]| \geq \frac{1}{p(n)}$$

As $f(U_n)$ is distributed uniformly, this is equivalent to A successfully distinguishing between $\{f(U_n)b(U_n)\}$ and $\{f(U_n)U_1\}$. It follows that A can distinguish between $X_1 = \{f(U_n)b(U_n)\}$ and $X_2 = \{f(U_n)\overline{b(U_n)}\}$.

Let X be the distribution created by uniformly choosing $\sigma \in_R \{1,2\}$ and then sampling from X_σ. Clearly X is identically distributed to $\{f(U_n)U_1\}$. Now:

$$Prob[A(X) = 1] = \frac{1}{2} \cdot Prob[A(X_1) = 1] \; + \; \frac{1}{2} \cdot Prob[A(X_2) = 1]$$

$$\Rightarrow Pr[A(X_2) = 1] = 2 \cdot Prob[A(X) = 1] \; - \; Prob[A(X_1) = 1]$$

Therefore:

$$| \; Prob[A(f(U_n)b(U_n)) = 1] \; - \; Prob[A(f(U_n)\overline{b(U_n)}) = 1] \; |$$

$$= | \ Prob[A(X_1) = 1] \ - \ Prob[A(X_2) = 1] \ |$$
$$= | \ Prob[A(X_1) = 1] - 2 \cdot Prob[A(X) = 1] + Prob[A(X_1) = 1] \ |$$
$$= | \ 2 \cdot Prob[A(X_1) = 1] - 2 \cdot Prob[A(X) = 1] \ |$$
$$= 2 \cdot | \ Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(U_{n+1}) = 1] \ | \geq \frac{2}{p(n)}$$

Assume, without loss of generality, that for infinitely many n's it holds that:

$$Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(f(U_n)\overline{b(U_n)}) = 1] \geq \frac{2}{p(n)}$$

Otherwise we simply reverse the terms here and make the appropriate changes in the algorithm and the remainder of the proof (i.e. we change step 2 of the algorithm below to: If $A(y \cdot \sigma) = 0$ then output σ).

We now construct A' to predict $b(U_n)$ from $f(U_n)$. Intuitively, A' adds a random guess to the end of it's input y (where $y = f(x)$ for some x) and attempts to see if it guessed $b(f^{-1}(y))$ correctly based on A's response to this guess. The algorithm follows:

Input: y

1. Uniformly choose $\sigma \in_R \{0,1\}$
2. If $A(y \cdot \sigma) = 1$ then output σ
3. Otherwise, output $\overline{\sigma}$

We now calculate the probability that A' successfully computes $b(f^{-1}(y))$. As σ is uniformly chosen from $\{0,1\}$ we have:

$$Prob[A'(f(U_n)) = b(U_n)] = \frac{1}{2} \cdot Prob[A'(f(U_n)) = b(U_n) \mid \sigma = b(U_n)]$$
$$+\frac{1}{2} \cdot Prob[A'(f(U_n)) = b(U_n) \mid \sigma = \overline{b(U_n)}]$$

Now, by the algorithm (see steps 2 and 3 respectively) we have:

$$Prob[A'(f(U_n)) = b(U_n) \mid \sigma = b(U_n)] = Prob[A(f(U_n)b(U_n)) = 1]$$

and

$$Prob[A'(f(U_n)) = b(U_n) \mid \sigma = \overline{b(U_n)}] = Prob[A(f(U_n)\overline{b(U_n)}) = 0]$$
$$= 1 - Prob[A(f(U_n)\overline{b(U_n)}) = 1]$$

By our contradiction hypothesis:

$$Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(f(U_n)\overline{b(U_n)}) = 1] \geq \frac{2}{p(n)}$$

Therefore:

$Prob[A'(f(U_n)) = b(U_n)]$

$$= \frac{1}{2}\cdot Prob[A(f(U_n)b(U_n)) = 1] + \frac{1}{2}\cdot(1 - Prob[A(f(U_n)\overline{b(U_n)}) = 1])$$
$$= \frac{1}{2} + \frac{1}{2}\cdot(Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(f(U_n)\overline{b(U_n)}) = 1]) \;\geq\; \frac{1}{2} + \frac{1}{p(n)}$$

which is in contradiction to the fact that b is a hard-core of f and cannot be predicted with non-negligible advantage over $\frac{1}{2}$. ■

We remind the reader that in the previous lecture we proved that a generator stretching the seed by even a single bit can be deterministically converted into a generator stretching the seed by any polynomial length. Therefore, it should not bother us that the above construction seems rather weak with respect to its "stretching capability".

At this stage it should be clear why it is crucial that the one-way function be hard to invert in the average case and not just in the worst case. If the function is invertible in the average case, then it is easy to distinguish between $\{f(U_n)b(U_n)\}$ and $\{U_{n+1}\}$ most of the time. This would clearly *not* be satisfactory for a pseudorandom generator.

14.4 A Hard-Core for Any One-Way Function

In this section we present a construction of a hard-core from any one-way function. Here there is no necessity that f be 1-1 or even length-preserving. As we have seen in the previous section, the existence of a hard-core is essential in our construction of a pseudorandom generator.

Theorem 14.5 *Let $f_0 : \{0,1\}^* \to \{0,1\}^*$ be a one-way function. Define $f(x,r) = (f_0(x), r)$ where $|x| = |r|$. Then $b(x,r) = \sum_{i=1}^{n} x_i r_i \bmod 2$ is a hard-core of f.*

Note that since f_0 is one-way, f is clearly one-way (trivially, any algorithm inverting f can be used to invert f_0).

Proof: Assume by contradiction, that there exists a probabilistic polynomial time algorithm A and a polynomial $p(\cdot)$ such that for infinitely many n's

$$Prob_{x,r}[A(f(x,r)) = b(x,r)] \geq \frac{1}{2} + \frac{1}{p(n)}$$

where the probabilities are taken, uniformly and independently, over $x \in_R \{0,1\}^n$, $r \in_R \{0,1\}^n$ and coin tosses of A (if any).

The following claim shows that there are a significant number of x's for which A succeeds with non-negligible probability. We will then show how to invert f for these x's.

Claim 14.4.1 *Let $S_n \subseteq \{0,1\}^n$ be the set of all x's where $Prob_r[A(f(x,r)) = b(x,r)] \geq \frac{1}{2} + \frac{1}{2p(n)}$. Then, $Prob_x[x \in S_n] > \frac{1}{2p(n)}$.*

Proof: By a simple averaging argument. Assume by contradiction that $Prob_x[x \in S_n] \leq \frac{1}{2p(n)}$. Then:

$$Prob_{x,r}[A(f(x,r)) = b(x,r)] = Prob_{x,r}[A(f(x,r)) = b(x,r) \mid x \in S_n]\cdot Prob[x \in S_n]$$
$$+ Prob_{x,r}[A(f(x,r)) = b(x,r) \mid x \notin S_n]\cdot Prob[x \notin S_n]$$

Now, trivially both $Prob_{x,r}[A(f(x,r)) = b(x,r) \mid x \in S_n] \leq 1$ and $Prob[x \notin S_n] \leq 1$. Furthermore, by the contradiction hypothesis $Prob[x \in S_n] \leq \frac{1}{2p(n)}$. Finally, based on the definition of S_n, $Prob_{x,r}[A(f(x,r)) = b(x,r) \mid x \notin S_n] < \frac{1}{2} + \frac{1}{2p(n)}$. Putting all these together:

$$Prob_{x,r}[A(f(x,r)) = b(x,r)] < 1 \cdot \frac{1}{2p(n)} + (\frac{1}{2} + \frac{1}{2p(n)}) \cdot 1 = \frac{1}{2} + \frac{1}{p(n)}$$

which contradicts the assumption regarding the success probability of A. ■

It suffices to show that we can retrieve x from $f(x)$ for $x \in S_n$ with probability $\frac{1}{poly(n)}$ (because then we can invert any $f(x)$ with probability $\geq \frac{1}{2p(n)poly(n)} = \frac{1}{poly(n)}$). So, assume that for x: $Prob_r[A(f(x,r)) = b(x,r)] \geq \frac{1}{2} + \frac{1}{2p(n)}$ (as in the claim).

Denote $B_x(r) = A(f(x,r)$. Now x is fixed and $B_x(r)$ is a black-box returning $b(x,r)$ with probability $\geq \frac{1}{2} + \frac{1}{2p(n)}$. We use calls to B_x to retrieve x given $f(x)$. We also denote $\epsilon = \frac{1}{2p(n)}$.

As an exercise, assume that $Prob_r[B_x(r) = b(x,r)] > \frac{3}{4} + \epsilon$. There is no reason to assume that this is true but it will help us with the proof later on. Using this assumption, for every $i = 1, ..., n$ we show how to recover x_i (that is, the i'th bit of x).

Input: $y = f(x)$

1. Uniformly select $r \in_R \{0,1\}^n$
2. Compute $B_x(r) \oplus B_x(r \oplus e^i)$ where $e^i = (0^{i-1}10^{n-i})$ (1 in the i'th coordinate and 0 everywhere else).

Now, $Prob[B_x(r) \neq b(x,r)] < \frac{1}{4} - \epsilon$ by the hypothesis. Therefore, $Prob[B_x(r \oplus e^i) \neq b(x, r \oplus e^i)] < \frac{1}{4} - \epsilon$ (because $r \oplus e^i$ is also uniformly distributed). So, the probability that $B_x(r) = b(x,r)$ and $B_x(r \oplus e^i) = b(x, r \oplus e^i)$ is greater than $\frac{1}{2} + 2\epsilon$ (by summing the errors).
In this case: $B_x(r) \oplus B_x(r \oplus e^i) = b(x,r) \oplus b(x, r \oplus e^i)$. However,

$$b(x,r) \oplus b(x, r \oplus e^i) = \sum_{j=1}^{n} x_j r_j + \sum_{j=1}^{n} x_j (r_j + e^i_j) \ mod\ 2$$

$$= \sum_{j=1}^{n} x_j r_j + x_i + \sum_{j=1}^{n} x_j r_j \ mod\ 2 = x_i$$

So, if we repeat this $O(\frac{n}{\epsilon^2})$ times and take a majority vote, we obtain a correct answer with a very high probability (in the order of $1 - \frac{1}{2^n}$). This is true (but with probability $\geq 1 - \frac{1}{2n}$) even if the different executions are only pairwise independent (this can be derived using Chebyshev's inequality and is important later on). We do the same for all i's and in this way succeed in inverting $f(x)$ with high probability. Note that we can use the same set of random strings r for each i (the only difference is in obtaining $b(x, r \oplus e^i)$ each time).

This inversion algorithm worked based on the unreasonable assumption that $Prob_r[B_x(r) = b(x,r)] > \frac{3}{4} + \epsilon$. This was necessary as we needed to query B_x on two different points and therefore we had to sum the error probabilities. When the probability of success is only ϵ above $\frac{1}{2}$, the resulting error probability is far too high.

In order to solve this problem, remember that we are really interested in calculating $b(x, e^i) = x_i$. However, we cannot query $B_x(e^i)$ because we have no information on B_x's behaviour at any given

point (the known probabilities are for $B_x(r)$ where r is uniformly distributed). Therefore we queried B_x at r and $r \oplus e^i$ (2 random points) and inferred x_i from the result.

We now show how to compute $b(x,r) \oplus b(x, r \oplus e^i)$ for $O(\frac{n}{\epsilon^2})$ pairwise independent r's. Based on what we have seen above, this is enough to invert $f(x)$. Our strategy is based on obtaining values of $b(x,r)$ for "free" (that is, without calling B_x). We do this by guessing what the value should be, but in such a way that the probability of being correct is non-negligible.

Let $l = \log_2(m+1)$ where $m = O(\frac{n}{\epsilon^2})$.
Let $s^1, ..., s^l \in_R \{0,1\}^n$ be l uniformly chosen n-bit strings.
Then for every non-empty $I \subseteq \{1, ..., l\}$, define: $r^I = \bigoplus_{i \in I} s^i$. Each r^I is an n-bit string constructed by xoring together the strings s^i indexed by I.

Each r^I is uniformly distributed as it is the xor of random strings. Moreover each pair is independent since for $I \neq J$, $\exists s^i$ s.t. w.l.o.g. $s^i \in I$ and $s^i \notin J$. Therefore, r^I given r^J is uniformly distributed based on the distribution of s^i.

Now, let us uniformly choose $\sigma^1, ..., \sigma^l \in \{0,1\}$. Assume that we were very lucky and that for every i, $\sigma^i = b(x, s^i)$ (in other words, we guessed $b(x, s^i)$ correctly every time). Note that this lucky event happens with the non-neglible probability of $\frac{1}{2^l} = \frac{\epsilon}{n^2} = \frac{1}{poly(n)}$. The following claim shows that in this lucky case we achieve our aim.

Claim 14.4.2 *Let $\tau^I = \bigoplus_{i \in I} \sigma^i$. Then, if for every i, $\sigma^i = b(x, s^i)$ then for every I, $\tau^I = b(x, r^I)$.*

Proof:

$$b(x, r^I) = b(x, \sum_{i \in I} s^i) = \sum_{j=1}^{n} x_j \cdot \sum_{i \in I} s^i_j = \sum_{i \in I} \sum_{j=1}^{n} x_j \cdot s^i_j = \sum_{i \in I} b(x, s^i) = \sum_{i \in I} \sigma^i = \tau^I$$

where all sums are mod 2. ∎

The above claim shows that by correctly guessing $\log m$ values of σ^i we are able to derive the value of $b(x, \cdot)$ at m pairwise independent points. This is because under the assumption that we guessed the σ^i's correctly, each τ^I is exactly $b(x, r^I)$ where the r^I's are uniformly distributed and pairwise independent.

Note that since there are $2^l - 1 = m$ different subsets I, we have the necessary number of different points in order to obtain x_i. The algorithm uses these points in order to extract x_i instead of uniformly chosen r's. (An alternative strategy is not to guess the σ^i's but to try every possible combination of them. Since $2^l = m + 1$ which is polynomial, we can do this in the time available.)

The Actual Algorithm:

Input: y

1. Uniformly choose $s^1, ..., s^l \in_R \{0,1\}^n$ and $\sigma^1, ..., \sigma^l \in \{0,1\}$.
2. For every non-empty $I \subseteq \{1, ..., l\}$, define: $r^I = \bigoplus_{i \in I} s^i$ and compute $\tau^I = \bigoplus_{i \in I} \sigma^i$.
3. For every $i \in \{1, ..., n\}$ and non-empty $I \subseteq \{1, ..., l\}$ do
   - $v_i^I = \tau^i \oplus B_x(r^I \oplus e^i)$
   - Guess x_i = majority$_I\{v_i^I\}$

In order to calculate the probability that the above algorithm succeeds in inverting $f(x)$, assume that for every i, we guessed σ^i such that $\sigma^i = b(x, s^i)$ (in step 1). The probability of this event occurring is $\frac{1}{2^l} = \frac{1}{m+1}$ for $m = O(\frac{n}{\epsilon^2})$ and $\epsilon = \frac{1}{2p(n)}$. In other words, the probability that we are lucky is $\frac{1}{poly(n)}$.

Since we know that for every I, $B_x(r^I \oplus e^i)$ is correctly computed with a probability greater that $\frac{1}{2} + 2\epsilon$, it follows that $b(x, r^I) \oplus B_x(r^I \oplus e^i)$ is also correct with the same probability (as the σ^i's are correct). As previously mentioned, due to the pairwise independence of the events, the probability that we succeed in inverting $f(x)$ in this case is at least $1 - \frac{1}{2n}$ (this is proved using Chebyshev's inequality).

It is easy to see that the probability that we succeed in guessing all σ^i's correctly *and* then proceed to successfully invert f is the product of the above two probabilities, that is $\frac{1}{poly(n)}$.

We therefore conclude that the probability of successfully inverting f is non-negligible. This is in contradiction to the one-wayness of f. Therefore, b as defined is a hard-core of f. ■

Summary of the proof.

We began by assuming the existence of an algorithm A which predicts $b(x, r)$ from $f(x, r)$ with non-negligible success probability (over all choices of x, r and coin tosses of A). We then showed that there is a non-negligible set of x's for which A succeeds and that we proceed to attempt to invert $f(x)$ for these x's only. This enabled us to set x and focus only on the randomness in r.

In the next stage we described how we can obtain x, under the assumption that both $b(x, r)$ and $b(x, r \oplus e^i)$ can be computed correctly with probability $\geq \frac{1}{2} + \frac{1}{poly(n)}$. This was easily shown as $b(x, r) \oplus b(x, r \oplus e^i) = x_i$ and by repeating we can achieve a small enough error so that the probability that we succeed in obtaining x_i for all i is at least $\frac{1}{poly(n)}$.

Finally we showed how to achieve the previous assumption. This involved realizing that pairwise independence for different r's is enough and then showing how $poly(n)$ pairwise independent n-bit strings can be obtained using only $log(poly(n))$ n-bit random strings $s^1, ..., s^l$.

Based on this, we guess the value of $b(x, s^i)$ for each i. The critical point here is that the probability of guessing correctly for all i is $\frac{1}{poly(n)}$ and that the value of $b(x, \cdot)$ for all $poly(n)$ pairwise independent n-bit strings can be immediately derived from these guesses.

In short, we succeeded (with non-negligible probability) in guessing $b(x, \cdot)$ correctly for a polynomial number of pairwise independent strings. These strings are used as r for $b(x, r)$ in the inversion algorithm described in the middle stage. We compute $b(x, r \oplus e^i)$ using A. Assuming that we guessed correctly, we achieve the necessary success probability for computing both $b(x, r)$ and $b(x, r \oplus e^i)$. As we guess once for the entire inversion algorithm and this is independent of all that follows, we are able to extract x from $f(x)$ with overall probability of $\frac{1}{poly(n)}$.

This proves that no such algorithm A exists.

Bibliographic Notes

This lecture is based mainly on [2] (see also Appendix C in [1]).

1. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

2. O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st STOC*, pages 25–32, 1989.

Lecture 15

Derandomization of BPP

Notes taken by Erez Waisbard and Gera Weiss

Summary: In this lecture we present an efficient deterministic simulation of randomized algorithms. This process, called derandomization, introduce new notions of pseudorandom generators. We extend the definition of pseudorandom generators and show how to construct a generator that can be used for derandomization. The new construction differ from the generator that we constructed in the previous lecture in it's running time (it will run slower, but fast enough for the simulation), but most importantly in it's assumptions. We are not assuming the existence of one-way function but we make another assumption which may be weaker than that.

15.1 Introduction

Randomness plays a key role in many algorithms. However random sources are not always available and we would like to minimize the usage of randomness. A naive way to remove the "randomness element" from algorithm is simply to go over all possible coin tosses it uses and act according to the majority. This however we can't do for BPP in polynomial time with respect to the size of the input. If we could shrink the amount of randomness the algorithm consumes (to logarithmic) then we could imply the naive idea in polynomial time. A way to use small random source to create much more randomness was introduced in the previous lecture - the pseudorandom generator. Pseudorandom generator G stretch out random seed into a polynomial-long pseudorandom sequence that can't be efficiently (in polynomial time) distinguished from a truly random sequence.

$$G : \{0,1\}^k \rightarrow \{0,1\}^{poly(k)}$$

For convenience we reproduce here the formal definition we gave in the previous lecture for pseudorandom generator.

Definition 15.1 *A deterministic polynomial-time algorithm G is called a pseudorandom generator if there exist a stretching function $l : N \rightarrow N$, so that for any probabilistic polynomial-time algorithm D, for any positive polynomial p, and for all sufficiently large k's*

$$|Prob[D(G(U_k)) = 1] - Prob[D(U_{l(k)}) = 1]| < \frac{1}{p(k)}$$

Where U_n denotes the uniform distribution over $\{0,1\}^n$ and the probability is taken over U_k (resp., $U_{l(k)}$) as well as over the coin tosses of D.

Suppose we have such a pseudorandom generator then for every $\epsilon > 0$ we can shrink the amount of randomness used by an algorithm A, deciding a language in BPP, from $poly(n)$ to n^ϵ (where n is the length of the input). The shrinking of randomness will not cause significance change in the behavior of the algorithm, meaning that it is infeasible to find a long enough input on which the algorithm which uses less randomness will decide differently than the original one. The problem is that the above doesn't indicate that there are no inputs for which A will act differently when using the pseudorandom source, but that they are hard to find. Thus in order to derandomize BPP we will need stronger notion of pseudorandom generator. We will need a pseudorandom generator which can fool any small (polynomial size) circuit (and hence any polynomial time algorithm).

Definition 15.2 *A deterministic polynomial-time algorithm G is called a non-uniformly strong pseudorandom generator if there exist a stretching function $l : N \to N$, so that for any family $\{C_k\}_{k \in L}$ of polynomial-size circuits, for any positive polynomial p, and for all sufficiently large k's*

$$|Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_{l(k)}) = 1]| < \frac{1}{p(k)}$$

Theorem 15.3 *If such G exist which is robust against polynomial-size circuit then*

$$\forall \epsilon > 0 \quad BPP \subseteq Dtime(2^{n^\epsilon})$$

Proof: $L \in BPP$ implies that the algorithm A for deciding L doesn't only take an input x with size n but also uses randomness R with size l (when we write for short $A(x)$ we really mean $A(x, R)$). The relation between the size of the input and the size of the randomness is $l = poly(n)$. Let us construct a new algorithm A' which will use less randomness than A but will act similar to A' for almost all inputs.

$$A'(x, s) \stackrel{\text{def}}{=} A(x, G(s))$$

Where $s \in \{0,1\}^{n^\epsilon}$.
A' uses little randomness and we claim that A' only differ from A for finitely many x's.

Proposition 15.1.1 *For all but finitely many x's*

$$|Prob[A(x) = 1] - Prob[A'(x) = 1]| < \frac{1}{6}$$

Proof: The idea of the proof is that if there were infinitely many x's for which A and A' differ, then we could distinguish G's output from a random string, in contradiction to the assumption that G is a pseudorandom generator.

In order to contradict Definition 15.2 it suffices to present a family $\{C_k\}$ of small circuits for which

$$|Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_{l(k)}) = 1]| \geq \frac{1}{6}$$

Suppose towards contradiction that for infinitely many x's A and A' behave differently, i.e

$$Prob[A(x) = 1] - Prob[A'(x) = 1] \geq \frac{1}{6}$$

Then we incorporate these inputs and A into a family of small circuits as follows

$$x \to C_x(\alpha) \stackrel{\text{def}}{=} A(x, \alpha)$$

This will enable us to distinguish the output of the generator from a uniformly distributed source. The circuit C_k will be one of $\{C_x : A(x)$ uses k coin tosses$\}$. Note that if there are infinitely many x's on which A and A' differ then there are infinitely many sizes of x's on which they differ. The amount of randomness used by the algorithm is polynomial with respect to the size of the input.

The idea behind this construction is that

$$C_k(G(U_k)) \equiv A'(x) \quad and \quad C_k(U_{l(k)})) \equiv A(x)$$

Hence we have a family of circuits such that

$$|Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_{l(k)}) = 1]| \geq \frac{1}{6}$$

Which is a contradiction to the definition of pseudorandom generator. ■

Saying that algorithm A decides BPP means that if $x \in L$ the probability that A will say $'YES'$ is greater than $\frac{2}{3}$ and if $x \notin L$ the probability that A will say $'YES'$ is smaller than $\frac{1}{3}$. By the above proposition, for all but finitely many x's $|Prob[A(x) = 1] - Prob[A'(x) = 1]| < 1/6$. Thus, for all but finitely many x's

$$x \in L \Rightarrow prob\ [A(x, U_n) = 1] \geq \frac{2}{3} \Rightarrow prob\ [A'(x, s) = 1] > \frac{1}{2}$$

$$x \notin L \Rightarrow prob\ [A(x, U_n) = 1] \leq \frac{1}{3} \Rightarrow prob\ [A'(x, s) = 1] < \frac{1}{2}$$

Now we define the algorithm A'' which incorporate these finitely many inputs, and for all other inputs it loops over all possible $s \in \{0,1\}^{n^\epsilon}$ (seeds of G) and decides by majority.

Algorithm A'': On input x proceed as follows.

```
if x is one of those finitely x's
    return the known answer
else
    for all s ∈ {0,1}^{n^ε}
       run A'(x, s)
    return the majority of A' answers
```

Clearly this A'' deterministicly decides L and run in time $2^{n^\epsilon} \cdot poly(n)$ as required. ■

15.2 New notion of Pseudorandom generator

The time needed for A'' to decide if an input x is in L or not was exponential in the length of it's seed s. For simulation purposes if the random seed is logarithmic in the size of the input, running the pseudorandom generator exponential time in the length of the seed is really running

it polynomial time with respect to the length of the input x. Thus the time needed for simulating a randomized algorithm which run in polynomial time remains polynomial even if we allow the pseudorandom generator to run exponential time (with logarithmic size seed). In general, for the purpose of derandomizing BPP we may allow the generator to use exponential time. There seems to be no point insisting that the generator will use only polynomial time in the length of the seed when the running time of the simulating algorithm has an exponential factor in the size of the seed.

Motivated as above, we now introduce a new notion of pseudorandom generator with the following properties.

1. Indistinguishable by a polynomial-size circuit.
2. Can run in exponential time ($2^{O(k)}$ on k-bit seed).

The efficiency of the generator which was one of the key aspects is relaxed in the new notion. This new notion may seem a little unfair at first since we only give polynomial time to the distinguisher and allow the generator to run exponential time. It even suggests that if we give the seed as an extra input, polynomial size circuit wouldn't be able to take advantage of that because it wouldn't be able to evaluate the generator on the given seed. The relaxation allow us to construct pseudorandom generators under seemingly weaker conditions than the ones required for polynomial time pseudorandom generators (the assumption about existence of one-way functions).

15.3 Construction of non-iterative pseudorandom generator

The difference between the definition of pseudorandom generator that we introduced in this lecture and the definition of pseudorandom that we had before (the one usually used for cryptographic purposes) is that we allow the generator to run in time exponential in it's input size.

This difference enables us to construct a random generator under possibly weaker conditions without damaging the derandomization process. In this section we will demonstrate how to construct such a generator using "unpredictable predicate" and a structure called "design" (we will give a precise definition later). In the construction we use two main tools which we assume to exist. We assume the existence of such a predicate and the existence of a design, but later we will show how to construct such a design hence the only real assumption that we make is the existence of a predicate.

In the previous class we proved that pseudorandom generators exist if and only if one-way permutation exist.
We will show (in section 15.3.2 below) that this assumption is not stronger than the previous assumption, i.e the existence of one-way permutation implies the existence of a predicate (but not necessarily the opposite way). So it seems better to use the new assumption which may be true even if there exist no one-way function.

The previous construction uses a one-way permutation $f : \{0,1\}^l \to \{0,1\}^l$ in the following way:

> From a random string $S_0 = S$ (the seed), compute a sequence $\{S_i\}$ by $S_{i+1} = f(S_i)$. The random bits are then extracted from this sequence using a hard-core predicate. We proved that a small circuit that is not fooled by this bit sequence can be used to

demonstrate that f is not a one-way permutation because it can be used to compute f^{-1}.

The construction that we will give here has a different nature. Instead of creating the bits sequentially we will generate them in parallel. Unpredictable predicates supply an easy way to construct one additional random looking bit from the seed. The problem is to generate more bits. We will do it by invoking the predicate on nearly disjoint subsets of the bits in the seed. This will give us bits which are nearly independent (such that no polynomial-size circuit will notice that they has some dependence).

15.3.1 Parameters

- k - Length of seed.
- m - Length of output (a polynomial in k).
- We want the generator to produce an output that is indistinguishable by any polynomial-size circuit (in k, or equivalently in m).

15.3.2 Tool 1: An unpredictable predicate

The first tool that we will need in order to construct a pseudorandom generator is a predicate that can not be approximated by polynomial sized circuits.

Definition 15.4 *We say that an exp(l)-computable predicate $b : \{0,1\}^l \to \{0,1\}$ is unpredictable by small circuits (or simply unpredictable) if for every polynomial $p(\cdot)$, for all sufficiently large l's and for every circuit C of size $p(l)$:*

$$Prob[C(U_l) = b(U_l)] < \frac{1}{2} + \frac{1}{p(l)}$$

This definition require that small circuits attempting to compute b have only negligible fraction of advantage over unbiased coin toss. This is a real assumption, because we don't know of a way to construct such a predicate (unlike the next tool that we will show how to construct later).

To evaluate the strength of our construction we prove, in the next claim, that the existence of unpredictable predicate is guaranteed if one-way permutation exist. The other way is not necessarily true, so it is possible that our construction can be applied (if someone will find a provable unpredictable predicate) while other constructions that depend on the existence of one-way permutations are not applicable (if someone will prove that such functions do not exist).

Claim 15.3.1 *If f_0 is a one-way permutation and b_0 is a hard-core of f_0, then $b(x) \stackrel{\text{def}}{=} b_0\left(f_0^{-1}(x)\right)$ is an unpredictable predicate.*

Proof:
We begin by noting that b is computable in exponential time because it takes exponential time to invert f and together with the computation of b_0, the total time is no more than exponential in the size of x.

The second property that we need to show is that it is impossible to predict b. In order to prove this property we use the variable $y \stackrel{\text{def}}{=} f_0^{-1}(x)$ to get:

$$b\left(f_0(y)\right) = b_0(y)$$

Assume towards contradiction that b is predictable. This means that there exist an algorithm A and a polynomial $p(\cdot)$ such that for infinite number of n's :

$$Prob\left[A(U_n) = b(U_n)\right] \geq \frac{1}{2} + \frac{1}{p(n)}$$

A is a polynomial-size algorithm which can predict b with a noticeable bias. But f is a permutation so we may write:

$$Prob\left[A\left(f_0(U_n)\right) = b\left(f_0(U_n)\right)\right] \geq \frac{1}{2} + \frac{1}{p(n)}$$

Hence, from the definition of b we get:

$$Prob\left[A\left(f_0(U_n)\right) = b_0(U_n)\right] \geq \frac{1}{2} + \frac{1}{p(n)}$$

which is a contradiction to the definition of b_0 as a hard core. ■

Recall that we demonstrated a generic hard-core predicate (the inner product *mod* 2) that is applicable to an arbitrary one-way permutation, hence essentially the last claim is only assuming the existence of one-way permutation (because the hard-core predicate can always be found). And we succeeded to show that the existence of unpredictable predicate may only be a weaker assumption than the existence of a one-way permutation. We did not prove that it is really a weaker assumption but we did show that it is not stronger. It may be the case that both assumptions are the same but we don't know of any proof for such a claim.

The assumption that we use to construct a generator is the existence of a "hard" predicate. The word "hard" means that the predicate can not be approximated by small circuits. The hardness of a predicate is measured by two parameters:

- The size of the circuit.
- The closeness of approximation.

In this notes we use polynomial-size circuits and polynomial approximation. Other papers demonstrated that similar process can be carried out with different conditions on this hardness parameters. In particular, the closeness of approximation parameter is greatly relaxed.

15.3.3 Tool 2: A design

The task of generating a single bit from a random bits seems easy if you have the predicate that we assumed to exist in the previous section because the output of the predicate must look random to every polynomial-size circuit. The problem is how can we generate more than one bit. We will do this using a collection of nearly disjoint sets to get random bits that are almost mutually independent (almost means indistinguishable by a polynomial-size circuits). To formalize this idea we introduce the notion of a design:

Definition 15.5 *A collection of m subsets $\{I_1, I_2, ..., I_m\}$ of $\{1, ..., k\}$ is a (k,m,l)-design (or simply design) if the following three properties exist:*

1. *For every $i \in \{1, ..., m\}$,*
$$|I_i| = l$$
2. *For every $i \neq j \in \{1, ..., m\}$,*
$$|I_i \cap I_j| = o(\log\ k)$$
3. *The collection is constructible in exp(k)-time.*

In our application the set $\{1, ..., k\}$ is all bit locations in the seed, and the subsets $\{I_1, I_2, ..., I_m\}$ correspond to different subsequences extracted from the seed. For example, one may look at a ten bit seed
$$S = \langle 1010010110 \rangle$$
The subset $I = \{1, 5, 7\} \subset \{1, ..., 10\}$ correspond to the first, the fifth and the seventh bits in the seeds which are, in this example :
$$S[I] = \langle 100 \rangle$$
In general, for $S = \langle \sigma_1 \sigma_2 \cdots \sigma_k \rangle$, and $I = \{i_1, ..., i_l\} \subset \{1, ..., k\}$ we use the notation:
$$S[I] \stackrel{\text{def}}{=} \langle \sigma_{i_1} \sigma_{i_2} \cdots \sigma_{i_k} \rangle$$

15.3.4 The construction itself

We now want to construct a pseudorandom generator with a polynomial stretching function. The size of the seed will be k and the size of the output will be $m = m(k)$ which is a polynomial in k. Suppose we have a design $\{I_1, ..., I_m\}$ which consist of m subsets of the set $\{1, ..., k\}$. If these subset where completely disjoint then it is obvious (from Definition 15.4) that for every unpredictable predicate b, the sequence $\langle b(S[I_1])\ b(S[I_2]) \cdots b(S[I_m]) \rangle$ is unpredictable. Since unpredictability implies pseudorandomness we get pseudorandom generator. We will show that this is also true when the intersection of any two subsets is logarithmic in k and m (i.e, this is a design). Our generator will blow up seeds by applying the unpredictable predicate to every subsequence corresponding to subsets in the design.

Proposition 15.3.2 *Let $b : \{0,1\}^{\sqrt{k}} \to \{0,1\}$ be unpredictable predicate, and $\{I_1, ..., I_m\}$ be a $(k, m, \sqrt{k})$ - design, then the following function is a pseudorandom generator (as defined in section 15.2):*
$$G(S) \stackrel{\text{def}}{=} \langle b(S[I_1])\ b(S[I_2]) \cdots b(S[I_m]) \rangle$$

Proof:
Based on the definition of unpredictable predicate and the definition of a design it follows that it take no more than exponentially many steps to evaluate G.

We will show now that no small circuit can distinguish the output of G from a random sequence. Suppose towards contradiction that there exist a family of polynomial-size circuits $\{C_k\}_{k \in \mathsf{N}}$ and a polynomial $p(\cdot)$ such that for infinitely many k's
$$|Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_m) = 1]| > \frac{1}{p(k)}$$

(This is the negation of G being a pseudorandom generator because the definition of pseudorandom generator demands that for every sufficiently large k's we have $|\cdots| < \frac{1}{p(k)}$ which imply that there are only finitely many k's with $|\cdots| > \frac{1}{p(k)}$)

We will assume that this expression is positive and remove the absolute value, because if we have infinitely many k's such that the above is true, we have that half of them have the same sign. We may take the sign as we want since we can always take other sequence of circuits with the reversed sign. So without loss of generality we assume that for infinitely many k's

$$Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_m) = 1] > \frac{1}{p(k)}$$

For any $0 \leq i \leq m$ we define a "hybrid" distribution H_k^i on $\{0,1\}^m$ as follows: the first i bits are chosen to be the first i bits of $G(U_k)$ and the other $m-i$ bits are chosen uniformly at random :

$$H_k^i \stackrel{\text{def}}{=} G(U_k)_{[1,...,i]} \circ U_{m-i}$$

In this notation we get $H_k^0 = U_m$ and $H_k^m = G(U_k)$. If we look at the function :

$$f_k(i) \stackrel{\text{def}}{=} Prob[C_k(H_k^i) = 1]$$

We get that, since $f_k(m) - f_k(0) > \frac{1}{p(k)}$, there must be some $0 \leq i_k \leq m$ such that:

$$f_k(i_k + 1) - f_k(i_k) > \frac{1}{m} \cdot \frac{1}{p(k)}$$

So we know that there exist a circuit which behaves significantly different if the i_k'th bit is taken randomly or from the generator (the difference is greater than one over the polynomial $p'(k) \stackrel{\text{def}}{=} m \cdot k$). That is:

$$Prob[C_k(H_k^{i_k+1}) = 1] - Prob[C_k(H_k^{i_k}) = 1] > \frac{1}{p'(k)}$$

We will use this circuit to construct another circuit which will be able to guess the next bit with some bias. On i_k bits of input and $m - i_k$ bits of random, the new circuit C_k' will behave as follows:

$$C_k'(\langle y_1, ..., y_{i_k} \rangle, \langle R_{i_k+1}, ..., R_m \rangle) \stackrel{\text{def}}{=} \begin{cases} R_{i_k+1} & if\ C_k(\langle y_1, ..., y_{i_k}, R_{i_k+1}, ..., R_m \rangle) = 1 \\ 1 - R_{i_k+1} & otherwise \end{cases}$$

where $\langle y_1, ..., y_m \rangle \stackrel{\text{def}}{=} G(U_k)$.

The idea behind this construction is that we know that the the probability that C_k will return 1 is significantly different whether R_{i_k+1} equals y_{i_k+1} or not. This is true because when R_{i_k+1} equals y_{i_k+1} we see that C_k is getting $H_k^{i_k+1}$ as input and otherwise it get $H_k^{i_k}$. The consequence of this fact is that we can use the answer of C_k to distinguish these two cases.

To analyze the behavior of C_k' more formally we look at two events (or Boolean variables):

$$A \stackrel{\text{def}}{=} (C_k(\langle y_1, ..., y_{i_k}, R_{i_k+1}, ..., R_m \rangle) = 1)$$
$$B \stackrel{\text{def}}{=} (R_{i_k+1} = y_{i_k+1})$$

Notice that C'_k will return y_{i_k+1} in two distinct scenarios. The first scenario is when $R_{i_k+1} = y_{i_k+1}$ and C_k returns 1, and the second scenario is when $R_{i_k+1} \neq y_{i_k+1}$ and C_k returns 0. Using the above notation we get that:

$$Prob[C'_k = y_{i_k+1}] = Prob[B] \cdot Prob[A|B] + Prob[B^c] \cdot Prob[A^c|B^c]$$

Since

- $Prob[A] = f(i_k) = Prob[C_k(H_k^{i_k}) = 1]$
- $Prob[A|B] = f(i_k + 1) = Prob[C_k(H_k^{i_k+1}) = 1]$
- $Prob[B] = Prob[B^c] = \frac{1}{2}$
- $Prob[A] = Prob[B] \cdot Prob[A|B] + Prob[B^c] \cdot Prob[A|B^c]$

we get that:

$$\begin{aligned}
Prob[C'_k = y_{i_k+1}] &= Prob[B] \cdot Prob[A|B] + Prob[B^c] \cdot Prob[A^c|B^c] \\
&= Prob[B] \cdot Prob[A|B] + Prob[B^c] - Prob[B^c] \cdot Prob[A|B^c] \\
&= Prob[B] \cdot Prob[A|B] + Prob[B^c] - (Prob[A] - Prob[B] \cdot Prob[A|B]) \\
&= \frac{1}{2} + Prob[A|B] - Prob[A] \\
&= \frac{1}{2} + f(i_k + 1) - f(i_k) \\
&> \frac{1}{2} + \frac{1}{p'(k)}
\end{aligned}$$

Hence the conclusion is:

$$[Prob[C'_k(G(U_k)_{[1,\ldots,i_k])} = G(U_k)_{i_k+1}] > \frac{1}{2} + \frac{1}{p'(k)}$$

(We use subscript notation to take partial bits of a given bit sequence. In this particular case $G(U_k)_{[1,\ldots,i_k])}$ is the first i_k bits of $G(U_k)$ and $G(U_k)_{i_k+1}$ is the $i_k + 1$'s bit of $G(U_k)$.)

That is, C'_k can guess the i_k'th bit of $G(U_k)$ with $\epsilon \stackrel{\text{def}}{=} \frac{1}{p'(k)}$ advantage over unbiased coin toss.

Let us now extend C' to also get complete seed as input (in addition to $S[I_1], S[I_2], ..., S[I_{i_k}]$)

$$C''_k(S \circ G(S)_{[1,\ldots,i_k]}) \stackrel{\text{def}}{=} C'_k(G(S)_{[1,\ldots,i_k]})$$

Since $G(U_k)_{i_k+1}$ is defined to be $b(S[I_{i_k+1}])$, we have:

$$Prob_s[C''_k(S \circ G(S)_{[1,\ldots,i_k]}) = b(S[I_{i_k+1}])] > \frac{1}{2} + \epsilon$$

We claim that there exist $\alpha \in \{0,1\}^{k-|I_{i_k+1}|}$ such that:

$$Prob_s[C''_k(S \circ G(S)_{[1,\ldots,i_k]}) = b(S[I_{i_k+1}]) \mid S[\overline{I_{i_k+1}}] = \alpha] > \frac{1}{2} + \epsilon$$

This claim is true because if we look at a random selection of S as two independent selections of $S[I_{i+1}]$ and $S[\overline{I_{i+1}}]$ we see that the average over the second selection is greater than $\frac{1}{2}+\epsilon$ so there must be an element with weight greater than that.

Now we come to the key argument in this proof. By incorporating C_k'' with α to a new circuit, we get a circuit that can approximate the value of $b(S[I_{i_k+1}])$ but it need the "help" of $b(S[I_1]), b(S[I_2]), ..., b(S[I_{i_k}])$. We will show now that we can do without this "help" when $\{I_1, ..., I_m\}$ is a design (note that we didn't use the fact that this is a design until now).

To prove that it is possible to build a circuit that doesn't use the "help", we need to show that there exist a polynomial-size circuit that get only $S[I_{i_k+1}]$ and can approximate $b(S[I_{i_k+1}])$. To do this we use the fact that all the bits in $S[I_1], S[I_2], ..., S[I_{i_k}]$ depend only on small fraction of $S[I_{i_k+1}]$ so circuits for computing these bits are relatively small and we can incorporated them in a circuit that include all possible values of these bits. Details follows.

To elaborate the last paragraph, recall that the intersection of any two subsets in a design is at most $O(log\ k)$, hence given $S[\overline{I_{i_k+1}}] = \alpha$ we know that for every $i \leq i_k$ the bits in I_i are fixed except for $O(log\ k)$ bits that may be in $S[I_{i_k+1}]$, and are given as part of the input. Since there are at most $O(log\ k)$ such bits, there exist a polynomial-size circuit that can "precompute" the value of b for every combination of these bits.

The first part of this circuit is a collection of tables, one for every I_i. Each table is indexed by all possible values of the "free bits" of an I_i (i.e., these in I_{i_k+1}). The entry for every such value (of $S[I_{i_k+1} \cap I_i]$) contains the corresponding $b(S[I_i])$ (under $S[\overline{I_{i_k+1}}] = \alpha$).

The second part of this circuit just implements a "selector"; that is, it uses the bits of $S[I_{i_k+1}$ in order to obtain the appropriate values of $b(S[I_1]), b(S[I_2]), ..., b(S[I_{i_k}])$ from the corresponding tables mentioned above.

Since we have polynomially many entries in every table and polynomially many tables we get that this is a polynomial-size circuit.

The conclusion is that we got a circuit that can approximate b with a non negligible advantage over unbiased coin toss. There exist infinitely many such circuits for infinitely many k's, which contradict the assumption of b being unpredictable predicate. ■

15.4 Constructions of a design

In this section we describe how to construct a design that can be used for the generator that we introduced in the preceding section. We need to construct m different subsets of the set $\{1, ..., k\}$ each has size l with small intersections.

15.4.1 First construction: using $GF(l)$ arithmetic

Assume without loss of generality that l is a prime factor and let $k = l^2$ (if l is not a prime factor pick the smallest power of 2 which is greater than l).

For the field $F \stackrel{\text{def}}{=} GF(l)$, we get that the Cartesian product $F \times F$ contains $k = l^2$ elements which we identify, with the k elements of $\{1, ..., k\}$. Every number in $\{1, ..., k\}$ is assigned to a pair in $F \times F$ (in a one-to-one correspondence). In the forgoing discussion we will interchange the pair and it's representative in $\{1, ..., k\}$ freely.
For every polynomial $p(\cdot)$ of degree d over F introduce the subset:

$$I_p \stackrel{\text{def}}{=} \{\langle e, p(e)\rangle : e \in F\} \subseteq F \times F$$

We get that:

1. The size of each set is $|I_p| = |F| = l$.
2. For every two polynomials $p \neq q$ the sets I_p and I_q intersects in at most d points (that is, $|I_p \cap I_q| \leq d$). This is true since:

   $$|\{\langle e, p(e)\rangle : e \in F\} \cap \{\langle e, q(e)\rangle : e \in F\}| = |\{\langle e : p(e) = q(e)\}|$$

   But the polynomial $p(e) - q(e)$ has degree smaller or equal to d so it can only have d zeroes (due to the Fundamental Theorem of Algebra).
3. There are $|F|^{d+1} = l^{d+1}$ polynomials over $GF(l)$ so for every polynomial $P(\cdot)$ we can find d such that the number of sets is greater than $P(l)$.
4. This structure is constructible in exponential (actually polynomial) in k time because all we need to do is simple arithmetic in $GF(l)$.

The conclusion is that we have a design (see Definition 15.5) that can be applied in the construction of pseudorandom generator that we gave above. This design remove the second assumption that we made about the existence of a design, so we get (as promised) that the only assumption needed in order to allow derandomization of BPP is that there exist an unpredictable predicate.

15.4.2 Second construction: greedy algorithm

In this subsection we introduce another construction of a design. We call this algorithm greedy because it just scans all the subsets of the needed size until it find one that doesn't overlap all previously selected subsets too much. This algorithm is "simpler" than the one we gave in the previous subsection but we need to show that it work correctly.
Consider the following parameters:

- $k = l^2$
- $m = poly(k)$
- We want that for all i to have $|I_i| = l$ and for all $i \neq j$, $|I_i \cap I_j| = O(log\ k)$

For these parameters we give a simple algorithm that scans all subsets one by one to find the next set to include in the design:

for $i = 1$ to m

  for all $I \subset [k]$, $|I| = l$ do

    $flag \leftarrow FALSE$

    for $j = 1$ to $i - 1$

if $|I_i \cap I_j| > log\ k$ then $flag \leftarrow TRUE$

if $flag = TRUE$ then $I_i \leftarrow I$

This algorithm run in an exponential time because there are 2^l subsets of size l and we scan them m times. Since $m \cdot 2^l < 2^k$ we get that even if we had to scan all the subsets in every round we could finish the process in time exponential in k.
To prove that the algorithm works it is enough to show that if we have $I_1, I_2, ..., I_{i-1}$ such that

1. $i \leq m$
2. For every $j < i : |I_j| = l$
3. For every $j_1, j_2 < i : |I_{j_1} \cap I_{j_2}| \leq log\ m + 2$

Then there exist another set $I_i \subset [k]$ such that $|I_i| = l$ and for every $j < i : |I_j \cap I_i| < 2 + log\ m$. We prove this claim using the Probabilistic Method. That is, we will show that most choices of I_i will do. In fact we show the following

Claim 15.4.1 *Let $I_1, I_2, ..., I_{i-1} \subset [k]$ each of size l. Then there exists an l-set, I, such that for all j's, it is the case that $|I_j \cap I| \leq \log\ m + 2$.*

Proof: We first consider the probability that a uniformly chosen l-set has a large intersection with a fixed l-set, denoted S. Actually, it will be easier to analyze the intersection of S with a set R selected at random so that for every $i \in [k]$

$$Prob[i \in R] = \frac{2}{l}$$

That is, the size of R is a random variable, with expectation $2k/l$ (which equals $2l$). We will show that with very high probability the intersection of R with S is not too big, and that very high probability $|R| \geq l$ (and so we can find an appropriate l-set). Details follow.
Let s_i be the i'th element in S sorted in any order (e.g., the natural increasing order). Consider the sequence $\{X_i\}_{i=1}^{l}$ of random variables defined as

$$X_i \stackrel{\text{def}}{=} \begin{cases} 1 & if\ s_i \in R \\ 0 & otherwise \end{cases}$$

Since these are independent Boolean random variables with $Prob[X_i = 1] = \frac{2}{l}$ for each i, we can use Chernoff's bound to get:

$$\begin{aligned} Prob\,[|S \cap R| > 2 + log\ m] &= Prob\left[\sum_{i=1}^{l} X_i > 2 + log\ m\right] \\ &= Prob\left[\frac{\sum_{i=1}^{l} X_i}{l} > \frac{2}{l} + \frac{log\ m}{l}\right] \\ &< Prob\left[\left|\frac{\sum_{i=1}^{l} X_i}{l} - \frac{2}{l}\right| > \frac{log\ m}{l}\right] \\ &< 2 \cdot e^{-log^2\ m} \ll 1/2m \end{aligned}$$

It follows that for R selected as above, the probability that there exists an I_j so that $|R \cap I_j| > 2 + \log m$ is bounded above by $\frac{i-1}{2m} < \frac{1}{2}$. The only problem is that such an R is not necessarily of size l. We shall show that with high probability the size of R is at least l, and so it contains a subset which will do (as I_i).
Consider the sequence $\{Y_i\}_{i=1}^k$ defined as

$$Y_i \stackrel{\text{def}}{=} \begin{cases} 1 & if\ i \in R \\ 0 & otherwise \end{cases}$$

Then, applying Chernoff's Bound we get:

$$\begin{aligned} Prob[|R| < l] &\leq Prob[|\tfrac{1}{k}\textstyle\sum_{i=1}^k Y_i - \tfrac{2}{l}| < \tfrac{1}{l}] \\ &< 2 \cdot e^{-2} \ll \tfrac{1}{2} \end{aligned}$$

Thus, for R selected as above, the probability that either $|R| < l$ or there exists an I_j so that $|R \cap I_j| > 2 + \log m$ is *strictly smaller than 1.* Therefore , there exists a set R such that $|R| \geq l$ and yet, for every $j < i$, we have $|R \cap I_j| \leq 2 + \log m$. Any l-subset of R qualifies as the set asserted by the claim. ■

We stress that this discussion about a randomly selected set is not part of the algorithm. The algorithm itself is totally deterministic. The randomness is just in our discussion – it serves as a tool to show that the algorithm will always find what it is looking for in every step.

Bibliographic Notes

This lecture is based on [4]. Further derandomization results, building on [4], can be found in [1], [2] and [3]. Specifically, in the latter paper a "full derandomization of BPP" is provided under the assumption that there exists a language $L \in \mathcal{E}$ having almost-everywhere exponential circuit complexity. That is: *Let* $\mathcal{E} \stackrel{\text{def}}{=} \cup_c \text{Dtime}(t_c)$, *with* $t_c(n) = 2^{cn}$. *Suppose that there exists a language* $L \in \mathcal{E}$ *and a constant* $\epsilon > 0$ *such that, for all but finitely many* n*'s, any circuit* C_n *which correctly decides* L *on* $\{0,1\}^n$ *has size at least* $2^{\epsilon n}$. *Then,* $\mathcal{BPP} = \mathcal{P}$.

1. L. Babai, L. Fortnow, N. Nisan and A. Wigderson. BPP has Subexponential Time Simulations unless EXPTIME has Publishable Proofs. *Complexity Theory*, Vol. 3, pages 307–318, 1993.

2. R. Impagliazzo. Hard-core Distributions for Somewhat Hard Problems. In *36th FOCS*, pages 538–545, 1995.

3. R. Impagliazzo and A. Wigderson. P=BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *29th STOC*, pages 220–229, 1997.

4. N. Nisan and A. Wigderson. Hardness vs Randomness. *JCSS*, Vol. 49, No. 2, pages 149–167, 1994.

Lecture 16

Derandomizing Space-Bounded Computations

Notes taken by Eilon Reshef

Summary: This lecture considers derandomization of space-bounded computations. We show that $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^2 n)$, namely, any bounded-probability Logspace algorithm can be deterministically emulated in $O(\log^2 n)$ space. We show that $\mathcal{BPL} \subseteq \mathcal{SC}$, namely, any such algorithm can be deterministically emulated in $O(\log^2 n)$ space and (simultaneously) in polynomial time.

16.1 Introduction

This lecture considers derandomization of space-bounded computations. Whereas current techniques for derandomizing time-bounded computations rely upon unproven complexity assumptions, the derandomization technique illustrated in this lecture stands out in its ability to derive its results exploiting only the pure combinatorial structure of space-bounded Turing machines.

As in previous lectures, the construction yields a pseudorandom generator that "fools" Turing machines of the class at hand, when in our case the pseudorandom generator generates a sequence of bits that looks truly random to any space-bounded machine.

16.2 The Model

We consider probabilistic Turing machines along the lines of the online model discussed in Lecture 5. A probabilistic Turing machine M has four tapes:

1. A read-only bidirectional input tape.
2. A read-only unidirectional random tape.
3. A read-write bidirectional work tape.
4. A write-only unidirectional output tape.

We consider $\mathcal{BPSPACE}(\cdot)$, the family of *bounded probability* space-bounded complexity classes. These classes are the natural two-sided error counterparts of the single-sided error classes $\mathcal{RSPACE}(\cdot)$, defined in Lecture 7 (Definition **??**).

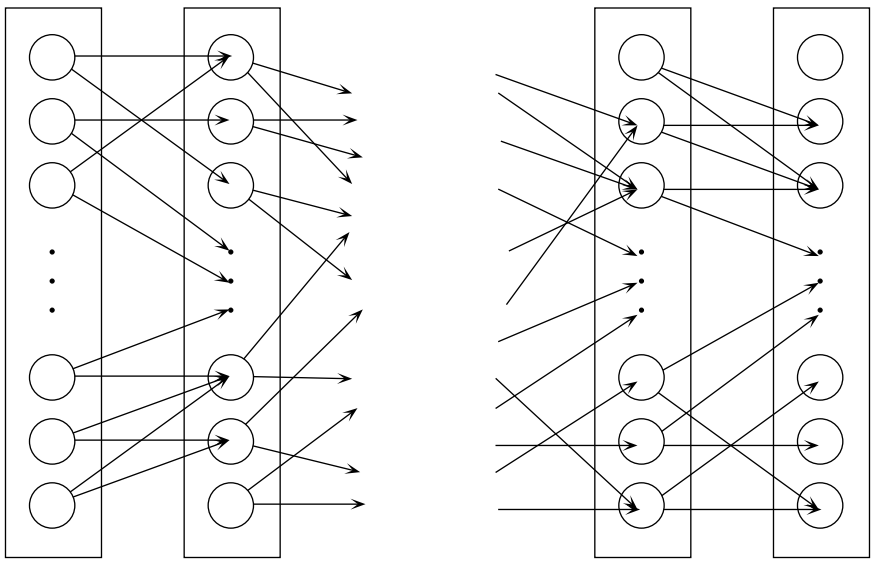

Figure 16.1: An Execution Graph of a Bounded-Space Turing Machine

Formally,

Definition 16.1 *For any function $s(\cdot) : \mathsf{N} \to \mathsf{N}$, the complexity class $\mathcal{BPSPACE}(s(\cdot))$ is the set of all languages L for which there exists a randomized Turing machine M such that on an input x*

1. *M uses at most $s(|x|)$ space.*
2. *The running time of M is bounded by $\exp(s(|x|))$.*
3. *$x \in L \Rightarrow \Pr[M(x) = 1] \geq 2/3$.*
4. *$x \notin L \Rightarrow \Pr[M(x) = 1] < 1/3$.*

Recall that condition (2) is important, as otherwise $\mathcal{NSPACE}(\cdot) = \mathcal{RSPACE}(\cdot) \subseteq \mathcal{BPSPACE}(\cdot)$. As usual, we are interested in the cases where $s(\cdot) \geq \log(\cdot)$. In particular, our techniques derandomize the complexity class $\mathcal{BPL}$, defined as

Definition 16.2 $\mathcal{BPL} \triangleq \mathcal{BPSPACE}(\log)$

Throughout the rest of the discussion we assume that the problems at hand are decision problems, that all functions are space-constructible, and that all logarithms are of base 2.

16.3 Execution Graphs

We represent the set of possible executions of a $\mathcal{BPSPACE}(\cdot)$ Turing machine M on an input x, $|x| = n$, as a layered directed graph $G_{M,x}$ (see Figure 16.1).

A vertex in the i-th layer of $G_{M,x}$ corresponds to a possible configuration of M after it has consumed exactly i random bits, i.e., when the head reading from the random tape is on the i-th cell. Thus, the i-th layer of $G_{M,x}$ corresponds to the set of all such possible configurations. $G_{M,x}$

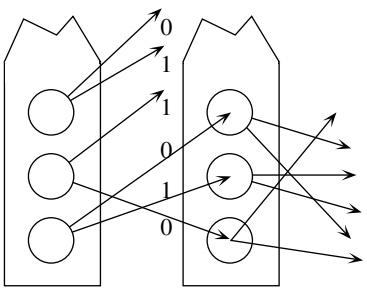

Figure 16.2: Edges in the Execution Graph $G_{M,x}$

contains an edge from a configuration vertex in the i-th layer to a configuration vertex in the $(i+1)$-th layer if there is a possible transition between the two configurations.

Formally, $G_{M,x} = (V_{M,x}, E_{M,x})$ is defined as follows. For each $i = 1, \ldots, D$, with $D \leq \exp(s(n))$, let the i-th layer $V^i_{M,x}$ be the set of all possible configurations of M given that M has consumed exactly i random bits. The vertex set $V_{M,x}$ is a union of all layers $V^i_{M,x}$. For each vertex $v \in V^i_{M,x}$, the edge set $E_{M,x}$ contains two outgoing directed edges to vertices $f_0(v), f_1(v)$ in $V^{i+1}_{M,x}$, where $f_0(v)$ (resp. $f_1(v)$) corresponds to the configuration M reaches following the sequence of transitions carried out after reading a "0" (resp. "1") bit from the random tape, and until the next bit is read (see Figure 16.2). For the convenience of the exposition below, assume that each of the two edges is labeled by its corresponding bit, i.e., "0" or "1".

Note that when the location of the head on the random tape is fixed, a configuration of M is fully determined by the contents of the work tape and by the positions of the heads on the work tape and on the input tape. Thus,

$$|V^i_{M,x}| \leq 2^{s(n)} \cdot s(n) \cdot n \leq \exp(s(n)).$$

The initial configuration of M corresponds to a designated vertex $v_0 \in V^0_{M,x}$. Similarly, the set of final vertices $V^D_{M,x}$ is partitioned into the set of accepting configurations V_{ACC} and the set of rejecting configurations V_{REJ}.

A *random walk* on $G_{M,x}$ is a sequence of steps, emanating from v_0, and proceeding along the directed edges of $G_{M,x}$, where in each step the next edge to be traversed is selected uniformly at random. Under this interpretation, the probability that a machine M accepts x is exactly the probability that a random walk emanating from v_0 reaches a vertex in V_{ACC}.

In contrast, a *guided walk* on $G_{M,x}$ with a *guide* R is a sequence of steps, emanating from v_0, and proceeding along the directed edges of $G_{M,x}$, where in the i-th step the next edge to be traversed is determined by the i-th bit of the guide R, i.e., the edge taken is the one whose label is equal to the value of the i-th bit of R.

Let $ACC(G_{M,x}, R)$ denote the event that a guided walk on $G_{M,x}$ with a guide R reaches a vertex in V_{ACC}. In this view,

$$\Pr[M \text{ accepts } x] = \Pr_{R \in_R \{0,1\}^D}[ACC(G_{M,x}, R)],$$

when R is selected uniformly at random from the set $\{0,1\}^D$.

Thus, a language L is in $\mathcal{BPL}$ if there exists a Turing machine M with a space bound of $s(n) = O(\log(n))$ such that for $D(n) = \exp(s(n)) = \mathrm{poly}(n)$,

- Whenever $x \in L$, $\Pr_{R \in_R \{0,1\}^{D(n)}}[ACC(G_{M,x}, R)] \geq 2/3$.
- Whenever $x \notin L$, $\Pr_{R \in_R \{0,1\}^{D(n)}}[ACC(G_{M,x}, R)] < 1/3$.

A (D, W)-*graph* G is a graph that corresponds to the execution of some $s(\cdot)$-space-bounded Turing machine on an input x, with a "depth" (number of layers) of D, $D \leq \exp(s(|x|))$ and a "width" (number of vertices in each layer) of W, $W \leq \exp(s(|x|))$. In the sequel, we present a derandomization method that "fools" any (D, W)-graph by replacing the random guide R with a pseudorandom guide R'.

16.4 Universal Hash Functions

The construction below is based upon a universal family of hash functions,

$$\mathcal{H}_\ell \;=\; \{h : \{0,1\}^\ell \to \{0,1\}^\ell\}.$$

Recall the following definition:

Definition 16.3 *A family of functions $\mathcal{H} = \{h : A \to B\}$ is called a* universal family of hash functions *if for every x_1 and x_2 in A, $x_1 \neq x_2$, $\Pr_{h \in_R \mathcal{H}}[h(x_1) = y_1 \text{ and } h(x_2) = y_2] = \left(\frac{1}{|B|}\right)^2$.*

Note that in our case the family $\mathcal{H}_\ell$ is degenerate, since the functions in $\mathcal{H}_\ell$ map ℓ-bit strings to ℓ-bit strings, and thus do not have any "shrinking" behavior whatsoever.

The construction requires that the functions h in $\mathcal{H}_\ell$ have a succinct representation, i.e., $|\langle h \rangle| = 2\ell$. An example of such a family is the set of all linear functions over $GF(2^\ell)$, namely

$$\mathcal{H}_\ell \;\triangleq\; \{h_{a,b} \;\mid\; a, b \in GF(2^\ell)\},$$

where

$$h_{a,b}(x) \;\triangleq\; ax + b,$$

with $GF(2^\ell)$ arithmetic.

Clearly, $|\langle h_{a,b} \rangle| = 2\ell$, and $h_{a,b}$ can be computed in space $O(\ell)$.

For the purpose of our construction, a hash function h is *well-behaved* with respect to two sets A and B, if it "extends well" to the two sets, i.e.,

$$\left|\Pr_{x \in_R \{0,1\}^\ell}[x \in A \text{ and } h(x) \in B] - \rho(A) \cdot \rho(B)\right| \leq 2^{-\ell/5},$$

where for any set $S \subseteq \{0,1\}^\ell$, we denote by $\rho(S)$ the probability that a random element x hits the set S, namely,

$$\rho(S) \;\triangleq\; \frac{|S|}{2^\ell} \;=\; \Pr_{x \in \{0,1\}^\ell}[x \in S].$$

The following proposition asserts that for any two sets A and B, almost all of the functions in $\mathcal{H}_\ell$ are well-behaved in the above sense with respect to A and B. Formally,

Proposition 16.4.1 *For every universal family $\mathcal{H}_\ell$ of hash functions, and for every two sets $A, B \subseteq \{0,1\}^\ell$, all but a $2^{-\ell/5}$ fraction of the $h \in \mathcal{H}_\ell$ satisfy*

$$\left|\Pr_{x \in \{0,1\}^\ell}[x \in A \text{ and } h(x) \in B] - \rho(A) \cdot \rho(B)\right| \leq 2^{-\ell/5}.$$

16.5 Construction Overview

We now turn to consider an arbitrary (D, W)-graph G representing the possible executions of an $s(\cdot)$-space-bounded Turing machine M on an input x, where $|x| = n$. We construct a pseudorandom generator $H : \{0,1\}^k \to \{0,1\}^D$ that emulates a truly random guide on G.

Formally,

Definition 16.4 *A function $H : \{0,1\}^k \to \{0,1\}^D$ is a* (D, W)-pseudorandom generator *if for every (D, W)-graph G,*

$$\left|\Pr_{R\in\{0,1\}^D}[ACC(G, R)] - \Pr_{R'\in\{0,1\}^k}[ACC(G, H(R'))]\right| \leq 1/10.$$

In Section 16.6, we prove the following theorem:

Theorem 16.5 *There exists a (D, W)-pseudorandom generator $H(\cdot)$ with $k(n) = O(\log D \cdot \log W)$. Further, $H(\cdot)$ is computable in space linear in its input.*

In particular,

Corollary 16.6 *There exists a (D, W)-pseudorandom generator $H(\cdot)$ with the following parameters:*

- $s(n) = \Theta(\log n)$.
- $D(n) \leq \text{poly}(n)$.
- $W(n) \leq \text{poly}(n)$.
- $k(n) = O(\log^2 n)$.

By trying all possible assignments for the seed of $H(\cdot)$, it follows that

Corollary 16.7 $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^2 n)$.

In fact, as will be evident from the construction below, the pseudorandom generator H operates in space $O(\log n)$. However, the space complexity of the derandomization algorithm is dominated by the space used for writing down the seed for $H(\cdot)$.

Note that this result is not very striking, since the same result is known to hold for the single-sided error class $\mathcal{RL}$, as $\mathcal{RL} \subseteq \mathcal{NL} \subseteq \mathcal{DSPACE}(\log^2 n)$. However, as shown below, the construction can be extended to yield more striking results.

16.6 The Pseudorandom Generator

In this section, we formally describe a (D, W)-pseudorandom generator as defined in Theorem 16.5. Without loss of generality, we assume $D \leq W$. The pseudorandom generator H is based on the universal family of hash functions $\mathcal{H}_\ell$, and extends strings of length $O(\ell^2)$ to strings of length $D \leq \exp(\ell)$, for $\ell = \Theta(\log |W|)$. The pseudorandom strings cannot be distinguished from truly random strings by any (D, W)-graph.

The input to H is interpreted as the tuple

$$I = (r, \langle h_1\rangle, \langle h_2\rangle, \ldots, \langle h_{\ell'}\rangle),$$

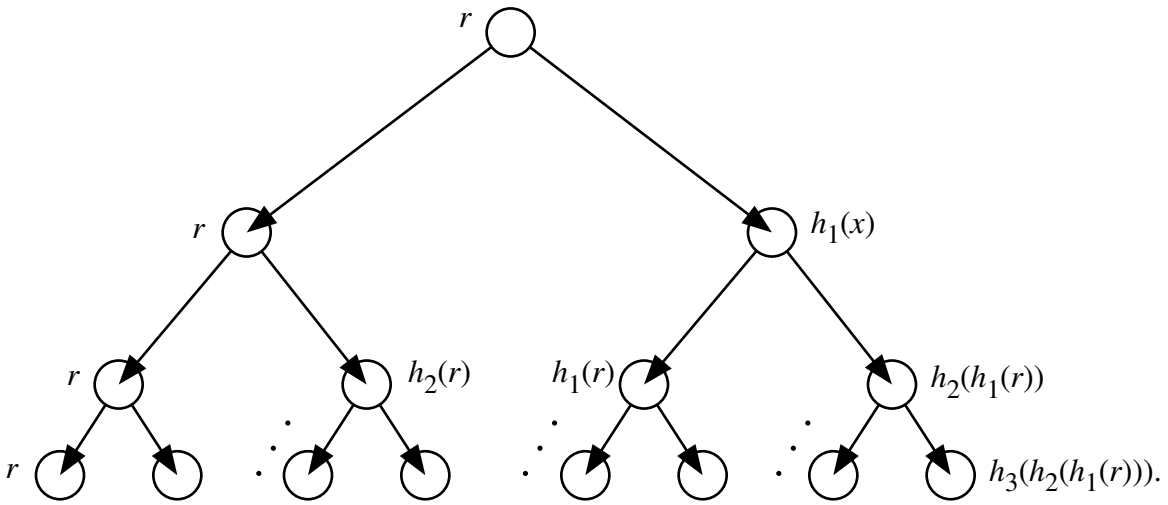

Figure 16.3: The Computation Tree T of $H(\cdot)$

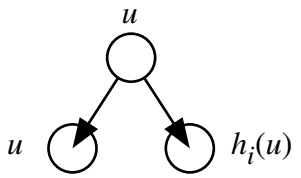

Figure 16.4: A Node in the Computation Tree of H

where $|r| = \ell$, $h_1, \ldots, h_{\ell'}$ are functions in $\mathcal{H}_\ell$, and $\ell' = \log(D/\ell)$. It can be easily observed that the length of the input I is indeed bounded by $O(\ell^2)$.

Now, given an input I, it may be most convenient to follow the computation of the pseudo-random generator H by considering a computation over a complete binary tree T of depth ℓ' (see Figure 16.3). The computation assigns a value to each node of the tree as follows. First, set the value of the root of the tree to be r. Next, given that a node located at depth $i-1$ has a value of u, set the value of its left child to u, and the value of its right child to $h_i(u)$ (see Figure 16.4). Finally, $H(\cdot)$ returns the concatenation of the binary values of the leaves of the tree, left to right.

More formally,

$$H(I) \stackrel{\triangle}{=} \alpha_0 \cdot \alpha_1 \ldots \cdot \alpha_{2^{\ell'}-1},$$

where α_j is defined such that if the binary representation of j is $\tau_1 \cdots \tau_{\ell'}$, then

$$\alpha_j \stackrel{\triangle}{=} h_{\ell'}^{\tau_{\ell'}} \circ \cdots \circ h_1^{\tau_1}(r),$$

where $h_i^1(z) = h_i(z)$ and $h_i^0(z) = z$ for every z.

Yet another way of describing $H(\cdot)$ is recursively as

$$H(r, \langle h_i \rangle, \ldots, \langle h_{\ell'} \rangle) \;=\; H(r, \langle h_{i+1} \rangle, \ldots, \langle h_{\ell'} \rangle) \cdot H(h_i(r), \langle h_{i+1} \rangle, \ldots, \langle h_{\ell'} \rangle),$$

where $H(z) \;=\; z$.

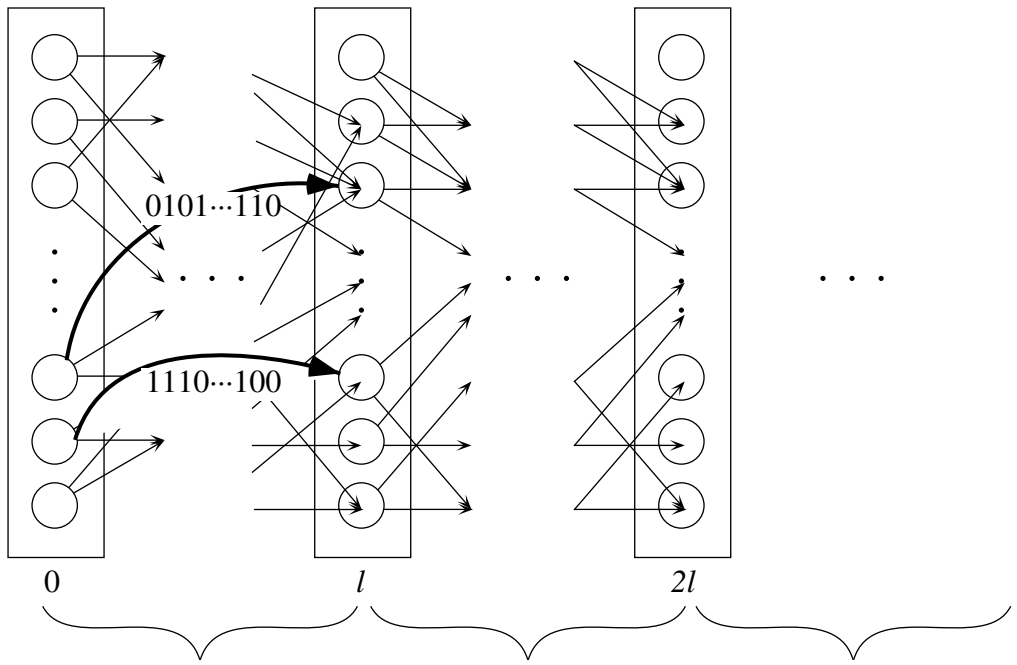

Figure 16.5: Contracting ℓ Layers

Note that the output of $H(\cdot)$ is composed of exactly $2^{\ell'} = D/\ell$ blocks, each of length ℓ, and hence the length of the output $H(I)$ is indeed D.

16.7 Analysis

It remains to see that $H(\cdot)$ is indeed a (D, W)-pseudorandom generator.

Theorem 16.8 (Theorem 16.5 rephrased) *$H(\cdot)$ is a (D, W)-pseudorandom generator.*

Proof: Consider a (D, W) graph $G_{M,x}$ that corresponds to the possible executions of a prospective distinguisher M on an input x. We show that $H(\cdot)$ is a (D, W)-pseudorandom generator by showing that a guided walk on $G_{M,x}$ using the guide $H(I)$ behaves "similarly" to a truly random walk, where $I = (z, \langle h_1 \rangle, \ldots, \langle h_{\ell'} \rangle)$ is drawn uniformly as a seed of H.

In an initial step, prune layers from $G_{M,x}$, so that only each ℓ-th layer remains, contracting edges as necessary (see Figure 16.5). Formally, construct a layered multigraph G_0 whose vertex set is the union of the vertex sets $V^0_{M,x}, V^{\ell}_{M,x}, V^{2\ell}_{M,x}, \ldots, V^{D}_{M,x}$, and whose edges correspond to directed paths of length ℓ in $G_{M,x}$. Label each edge in the multigraph G_0 by an ℓ-bit string which is the concatenation of the labels of the edges along the corresponding directed path in $G_{M,x}$. Thus, every multiedge e in G_0 corresponds to a subset of $\{0,1\}^{\ell}$. Clearly, a random walk on $G_{M,x}$ is equivalent to a random walk on G_0, when at each step an ℓ-bit string R is drawn uniformly, and the edge traversed is the edge whose label is R.

The analysis associates $H(\cdot)$ with a sequence of coarsenings. At each such coarsening, a new hash function h_i, uniformly drawn from $\mathcal{H}_{\ell}$, is used to decrease the number of truly random bits needed for the random walk by a factor of 2. After ℓ' such coarsenings, the only truly random bits

required for the walk are the ℓ random bits of r, with the additional bits used to encode the hash functions $h_1, \ldots, h_{\ell'}$.

We begin by presenting the first coarsening step. In this step, the random guide $R = (R_1, R_2, \ldots, R_{D/\ell})$ is replaced by a "semi-random" guide $R' = (R_1, h_{\ell'}(R_1), R_3, h_{\ell'}(R_3), \ldots, R_{D/\ell-1}, h_{\ell'}(R_{D/\ell-1}))$, where $h_{\ell'}$ and the R_i's are drawn uniformly at random. Below we show that the semi-random guide behaves "almost like" the truly random guide, i.e., that for some ϵ,

$$\left|\Pr_R[ACC(G_0, R)] - \Pr_{R'}[ACC(G_0, R')]\right| < \epsilon.$$

We begin with a technical preprocessing step, removing from G_0 edges whose traversal probability is very small. Formally, let E_{light} denote the set of all "light" edges, i.e., edges (u, v) for which $\Pr_{R_i \in_R \{0,1\}^\ell}[u \to v] < 1/W^2$. Create a graph G_0' whose vertex set is the same as G_0's, but containing only edges in $E \setminus E_{\text{light}}$. We first show that the removal of the light edges have a negligible effect. Formally,

Lemma 16.7.1 *For $\epsilon_1 = 2/W$,*

1. $|\Pr_R[ACC(G_0', R)] - \Pr_R[ACC(G_0, R)]| < \epsilon_1$.
2. $|\Pr_{R'}[ACC(G_0', R')] - \Pr_{R'}[ACC(G_0, R')]| < \epsilon_1$.

Proof: For the first part, the probability that a random walk R uses an edge in E_{light} is at most $\epsilon_1 = D \cdot (1/W^2) \leq 1/W$, and hence

$$\left|\Pr_R[ACC(G_0, R)] - \Pr_R[ACC(G_0'), R)]\right| < \epsilon_1.$$

For the second part, consider two consecutive steps guided by R' along the edges of G_0. The probability that the first step of R' traverses a light edge is bounded by $1/W^2$. By Proposition 16.4.1 with respect to the sets $\{0,1\}^\ell$ and the set of light edges available for the second step, for all but a fraction of $2^{-\ell/5}$ of the hash functions $h_{\ell'}$, the probability that the second step of R' traverses a light edge is bounded by $1/W^2 + 2^{-\ell/5} \leq 2/W^2$, assuming that the constant for ℓ is large enough. Hence, except for a fraction of $(D/2) \cdot 2^{-\ell/5} < \epsilon_1/2$ of the hash functions, the overall probability that R' traverses a light edge is bounded by $D \cdot (2/W^2) < \epsilon_1/2$. Thus, the overall probability of hitting a light edge is bounded by ϵ_1. ■

It thus remains to show that the semi-random guide R' behaves well with respect to the pruned graph G_0'. Formally, for some ϵ_2 specified below, we show that

Lemma 16.7.2 $|\Pr_R[ACC(G_0', R)] - \Pr_{R'}[ACC(G_0', R')]| < \epsilon_2$.

Proof:

Consider three consecutive layers of G_0', say $V_{M,x}^0$, $V_{M,x}^\ell$, $V_{M,x}^{2\ell}$ (see Figure 16.6), and fix a triplet of vertices $u \in V_{M,x}^0$, $v \in V_{M,x}^\ell$, and $w \in V_{M,x}^{2\ell}$ for which the edges (u, v) and (v, w) are in the edge set of G_0'. Let $E_{u,v}$ denote the set of edges in G_0' connecting u to v, and let $E_{v,w}$ denote the set of edges in G_0' connecting v to w.

The probability that a random walk emanating from u visits v and reaches w can be written as

$$P_{u-v-w} \;=\; \Pr_{R_1, R_2 \in \{0,1\}^\ell}\left[R_1 \in E_{u,v} \textbf{ and } R_2 \in E_{v,w}\right].$$

Since the graph G_0' was constructed such that $\Pr_{R_i \in_R \{0,1\}^\ell}[u \to v] \geq 1/W^2$ and $\Pr_{R_i \in_R \{0,1\}^\ell}[v \to w] \geq 1/W^2$, it follows that $P_{u-v-w} \geq 1/W^4$.

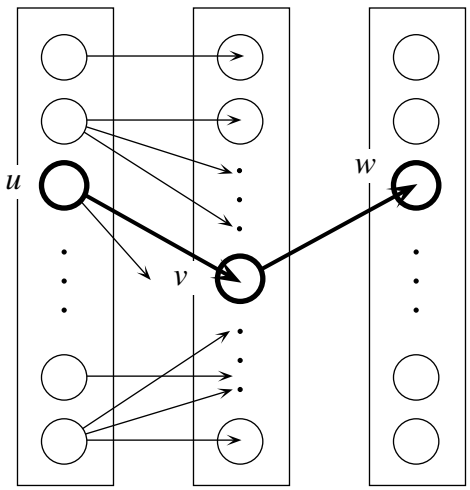

Figure 16.6: Three Consecutive Layers of G'_0

Now, the crux of the construction is that the above random walk can be replaced by a "semi-random" walk, namely, a walk whose first ℓ steps are determined by a random guide R_1, and whose last ℓ steps are determined by $h_{\ell'}(R_1)$, for some function $h_{\ell'}$ in $\mathcal{H}_\ell$. Given $h_{\ell'}$, the probability that a semi-random walk emanating from u reaches w via v is $P^{h_{\ell'}}_{u-v-w}$, where

$$P^{h}_{u-v-w} \triangleq \Pr_{R_1 \in \{0,1\}^\ell} [R_1 \in E_{u,v} \textbf{ and } h(R_1) \in E_{v,w}]$$

However, Proposition 16.4.1 applied with respect to the sets $E_{u,v}$ and $E_{v,w}$ asserts that except for a fraction of $2^{-\ell/5}$ of the hash functions $h_{\ell'}$,

$$\left| P^{h_{\ell'}}_{u-v-w} - P_{u-v-w} \right| \leq 2^{-\ell/5}. \tag{16.1}$$

Thus, except for a fraction of at most $\epsilon_3 = W^4 \cdot 2^{-\ell/5}$ of the hash functions, Equation (16.1) holds for *every* triplet of vertices u, v and w in every triplet of consecutive layers, i.e.,

$$\forall\ u, v, w \quad \left| P^{h_{\ell'}}_{u-v-w} - P_{u-v-w} \right| \leq 2^{-\ell/5}. \tag{16.2}$$

Next, fix a hash function $h_{\ell'}$ which satisfies Equation (16.2). The overall probability that a truly random walk starting from u reaches w can be written as

$$P_{u-w} \ = \ \sum_v P_{u-v-w},$$

whereas the probability that a semi-random walk starting from u reaches w is

$$P^{h_{\ell'}}_{u-w} \ = \ \sum_v P^{h_{\ell'}}_{u-v-w}.$$

Consequently, with a suitable selection of constants, and since $P_{u-w} \geq 1/W^4$,

$$|P^{h_{\ell'}}_{u-w} - P_{u-w}| \leq W \cdot 2^{-\ell/5} \leq W^5 \cdot 2^{-\ell/5} \cdot P_{u-w} \leq 2^{-\ell/10} \cdot P_{u-w} = \epsilon_4 \cdot P_{u-w},$$

for $\epsilon_4 = 2^{-\ell/10}$.

However, once the hash function $h_{\ell'}$ is fixed, every two-hop walk $u - v - w$ depends only on the corresponding ℓ-bit R_i, and hence for every "accepting path", i.e., a path $P \in \{0,1\}^D$ leading from the initial vertex v_0 to an accepting vertex,

$$\left|\Pr[R' = P] - \Pr[R = P]\right| \leq \epsilon_4 \cdot \Pr[R = P].$$

Since the probability of accepting is a sum over all accepting paths,

$$\left|\Pr_{R'}[ACC(G'_0, R')] - \Pr_R[ACC(G'_0, R)]\right| \leq \epsilon_4 \cdot \Pr_R[ACC(G'_0, R)] \leq \epsilon_4. \tag{16.3}$$

Finally, consider the two events $ACC(G'_0, R)$ and $ACC(G'_0, R')$, where the hash function $h_{\ell'}$ is drawn uniformly at random. The probability that R' hits a "bad" hash function $h_{\ell'}$ is bounded by ϵ_3. Otherwise, Equation (16.3) holds, and thus the lemma holds for $\epsilon_2 = \epsilon_3 + \epsilon_4$. ■

By the above two lemmas, it follows that the semi-random guide R' behaves "well' in the original graph G_0, i.e.,

Corollary 16.9 $|\Pr_R[ACC(G_0, R)] - \Pr_{R'}[ACC(G_0, R')]| < \epsilon$, *where* $\epsilon = 2 \cdot \epsilon_1 + \epsilon_2$. ■

Now, to perform another coarsening step, construct a new multigraph G_1 by contracting each pair of adjacent edge sets as follows:

- The vertex set of G_1 is the union of the vertices in the even layers of G.
- Create an edge for every two-hop path taken by the semi-random walk. Formally, for every two adjacent edges (u, v) and (v, w) labeled σ and $h_{\ell'}(\sigma)$ respectively, create an edge (u, w) in G_1, and label it σ.

Reapply Lemma 16.7.1 and Lemma 16.7.2 on G_1, this time using a new hash function $h_{\ell'-1}$, yielding a new multigraph G_2, and so on. It thus follows that at each step,

$$\left|\Pr[ACC(G_{i+1}, R^{(i+1)})] - \Pr[ACC(G_i, R^{(i)})]\right| < \epsilon.$$

After ℓ' iterations, the resulting graph $G_{\ell'}$ is a bipartite graph, on which a truly random guide r is applied. Since the above analysis corresponds to the behavior of the pseudorandom generator $H(\cdot)$,

$$\left|\Pr_{I \in_R \{0,1\}^{|I|}}[ACC(G, H(I))] - \Pr_{R \in_R \{0,1\}^D}[ACC(G, R)]\right| \quad \leq \quad \ell' \cdot \epsilon \leq 1/10,$$

where $I = (r, \langle h_1 \rangle, \langle h_2 \rangle, \ldots, \langle h_{\ell'} \rangle)$, which concludes the proof. ■

Remark: The analysis requires that the constant for $\ell = \Theta(\log n)$ be large enough. In the underlying computational model, this ensure that the machine M cannot retain even a description of a single function h_i. Otherwise, M could examine the first four blocks of the pseudorandom sequence, i.e., $z, h_{\ell'}(z), z', h_{\ell'}(z')$, and fully determine $h_{\ell'}$ by solving two linear equations with two variables. This would make it possible for M to distinguish between a truly random sequence R and the pseudorandom sequence $H(I)$.

16.8 Extensions and Related Results

16.8.1 $\mathcal{BPL} \subseteq \mathcal{SC}$

Whereas Corollary 16.7 asserts that $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^2 n)$, the running time of the straightforward derandomized algorithm is $\Omega(\exp(\log^2(n)))$, and in particular is not polynomial in n.

In this section we consider the complexity class $\mathcal{TS}(t(\cdot), s(\cdot))$, which denotes the set of all languages that can be recognized by a Turing machine whose running time is bounded by $t(\cdot)$ and whose space is (simultaneously) bounded by $s(\cdot)$. In particular, we consider $\mathcal{SC}$ (a.k.a., "Steve's Class"),

Definition 16.10 $\mathcal{SC} \triangleq \mathcal{TS}(\text{poly}(n), \text{polylog}(n))$.

We state the following theorem:

Theorem 16.11 $\mathcal{BPL} \subseteq \mathcal{SC}$.

Proof Sketch: Consider a language $L \in \mathcal{BPL}$ and a corresponding Turing machine M for which $L = L(M)$. Now, instead of trying all $O(\log^2(n))$ possible values of the input $I = (r, \langle h_1 \rangle, \ldots, \langle h_{\ell'} \rangle)$ of H to determine whether an input x is in L, perform the following steps:

- "Magically" find a sequence of "good" functions $h_1, \ldots, h_{\ell'}$.
- Emulate M only over all possibilities of r. Since $|r| = \ell$, the emulation can be carried out in time exponential in ℓ, and hence in polynomial time.

To find a sequence of "good" functions, incrementally fix $h_{\ell'}$, $h_{\ell'-1}$, ..., h_1. To fix a single h_j, assume all functions $h_{\ell'}, \ldots, h_{j+1}$ were already fixed and stored in memory. Consider all functions h in $\mathcal{H}_\ell$, and test each such h to determine whether Lemma 16.7.1 and Equation (16.2) hold. Since the existence of a "good" function h_j is asserted by Theorem 16.8, it remains to verify that finding such a function can be carried out in time exponential in ℓ (and hence in polynomial time) and in logarithmic space.

To see that this is the case, recall that the pseudorandom generator $H(\cdot)$ can be written recursively as

$$H(r, \langle h_i \rangle, \ldots, \langle h_{\ell'} \rangle) \;=\; H(r, \langle h_{i+1} \rangle, \ldots, \langle h_{\ell'} \rangle) \cdot H(h_i(r), \langle h_{i+1} \rangle, \ldots, \langle h_{\ell'} \rangle),$$

where $H(z) = z$. Consequently, once the functions $h_{\ell'}, \ldots, h_{j+1}$ are fixed, every single probability P_{u-v-w} can be computed directly simply by exhaustively considering all possible random guides R (of length 2ℓ). Similarly, given a candidate hash function $h_{\ell'}$, every single probability $P^{h_{\ell'}}_{u-v-w}$ can also be computed directly by exhaustively considering all semi-random guides R'. Hence, to determine whether Equation (16.2) holds, simply compare P_{u-v-w} and $P^{h_{\ell'}}_{u-v-w}$ for every triplet of vertices u, v and w in adjacent layers. Testing whether Lemma 16.7.1 holds can be carried out in a similar manner. Clearly, each such test can be carried out in time exponential in ℓ, and since the number of candidate functions $h_{\ell'}$ is also exponential in ℓ, the overall running time is exponential in ℓ. Further, testing a single function h can be carried out in space linear in ℓ, and hence the overall space complexity is dominated by the space needed to store the functions $h_{\ell'}, \cdots, h_1$, i.e., by $O(\ell^2)$. ■

Remark: The edge set of the i-th multigraph G_i depends upon the hash functions drawn in previous steps. Thus, although "almost all" functions h in $\mathcal{H}_\ell$ would satisfy Equation (16.2), one cannot consider fixing a function h_j before committing on the functions $h_{\ell'}, \ldots, h_{j+1}$.

16.8.2 Further Results

Below we state, without a proof, two related results:

Theorem 16.12 $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^{1.5} n)$.

Theorem 16.13 (Informal) *Every random computation that can be carried out in polynomial time and in linear space can also carried out in polynomial time and linear space, but using only a linear amount of randomness.*

Bibliographic Notes

The main result presented in this lecture is due to Noam Nisan: The generator itself was presented and analyzed in [1], and the $\mathcal{SC}$ derandomization was later given in [2].

Theorems 16.12 and 16.13 are due to [4] and [3], respectively.

1. N. Nisan. Pseudorandom Generators for Space Bounded Computation. *Combinatorica*, Vol. 12 (4), pages 449–461, 1992.

2. N. Nisan. $\mathcal{RL} \subseteq \mathcal{SC}$. *Journal of Computational Complexity*, Vol. 4, pages 1-11, 1994.

3. N. Nisan and D. Zuckerman. Randomness is Linear in Space. To appear in *JCSS*. Preliminary version in *25th STOC*, pages 235–244, 1993.

4. M. Saks and S. Zhou. $RSPACE(S) \subseteq DSPACE(S^{3/2})$. In 36th *FOCS*, pages 344–353, 1995.

Lecture 17

Zero-Knowledge Proof Systems

Notes taken by Michael Elkin and Ekaterina Sedletsky

Summary: In this lecture we introduce the notion of zero-knowledge interactive proof system, and consider an example of such a system (Graph Isomorphism). We define perfect, statistical and computational zero-knowledge and present a method for constructing zero-knowledge proofs for NP languages, which makes essential use of bit commitment schemes. Presenting a zero-knowledge proof system for an NP-complete language, we obtain zero-knowledge proof systems for every language in NP. We consider a zero-knowledge proof system for one NP-complete language, specifically Graph 3-Colorability. We mention that zero-knowledge is preserved under sequential composition, but is not preserved under the parallel repetition.

Oded's Note: *For alterantive presentations of this topic we refer the reader to either Section 2.3 in [2] (approx. 4 pages), or the 30-page paper [3], or the first 4–5 sections in Chapter 4 of [1] (over 50 pages).*

17.1 Definitions and Discussions

Zero-knowledge (ZK) is quite central to cryptography, but it is also interesting in context of this course of the complexity theory. Loosely speaking, zero-knowledge proof systems have the remarkable property of being convincing and yielding nothing beyond the validity of the assertion.

We say that the proof is zero-knowledge if the verifier does not get from it anything that he can not compute by himself, when it assumes that the assertion is true.

Traditional proof carries with it something which is beyond the original purpose. The purpose of the proof is to convince somebody, but typically the details of proof give the verifier *more* than merely conviction in the validity of the assertion and it is not clear whether it is essential or not.

But there is an extreme case, in which the prover gives the verifier nothing beyond being convinced that the assertion is true. If the verifier assumed a-priori that the assertion is true, then actually the prover supplied no new information.

The basic paradigm of zero-knowledge interactive proof system is that whatever can be efficiently obtained by interacting with a prover, could also be computed without interaction, just by assuming that the assertion is true and conducting some efficient computation.

Recall that in the definition of interactive proof system we have considered properties of the verifier, whereas no requirements on the prover were imposed. In zero-knowledge definition we talk

about some *feature* of the prescribed prover, which captures prover's robustness against attempts to gain knowledge by interacting with it. Verifier's properties are required to ensure that we have a proof system. A straightforward way of capturing the informal discussion follows.

Definition 17.1 *Let A and B be a pair of interactive Turing machines, and suppose that all possible interactions of A and B on each common input terminate in a finite number of steps. Then $\langle A, B\rangle(x)$ is the random variable representing the (local) output of B when interacting with machine A on common input x, when the random-input to each machine is uniformly and independently chosen.*

Definition 17.2 *Let (P, V) be an interactive proof system for some language L. We say that (P, V), actually P, is* zero-knowledge *if for every probabilistic polynomial time interactive machine V^* there exists an (*ordinary*) probabilistic polynomial time machine M^* so that for every $x \in L$ holds*

$$\{\langle P, V^*\rangle(x)\}_{x\in L} = \{M^*(x)\}_{x\in L},$$

where the equality "=" between the ensembles of distributions can be interpreted in one of three ways that we will discuss later.

Machine M^* is called a simulator *for the interaction of V^* with P.*

We stress that we require that *for every* V^* interacting with P, not merely for V, there exists a simulator M^*. This simulator, although not having access to the interactive machine P, is able to simulate the interaction of V^* with P. This fact is taken as evidence to the claim that V^* did not gain any knowledge from P (since the same output could have been generated without any access to P).

V^* is an interactive machine, potentially something more sophisticated than V, which is the prescribed verifier. What V^* is interested in, is to extract from the prover more information than the prover is willing to tell. The prover wants to convince the verifier in the fact that $x \in L$, but the verifier is interested to get *more* information. Any efficient way, which the verifier may try to do it, is captured by such interacting process or strategy V^*.

We are talking here about probability ensembles, but unlike our definitions of the pseudo-randomness, now probability ensembles are defined using index which is not a natural number, but rather less trivial. We have to modify the formalism slightly and to enable indexing by any countable set. It is important to understand that for every x we have two distributions $\langle P, V^*\rangle(x)$ and $M^*(x)$.

¿From now on the distribution ensembles are indexed by strings $x \in L$, so that each distribution contains only strings of length polynomial in the length of the index.

The question is when these two probability ensembles are equal or close.

There are three natural notions:

Definition 17.3 *Let (P, V) be an interactive proof system for some language L. We say that (P, V), actually P, is* perfect zero-knowledge(PZK) *if for every probabilistic polynomial time interactive machine V^* there exists an (*ordinary*) probabilistic polynomial time machine M^* so that for every $x \in L$ the distributions $\{\langle P, V^*\rangle(x)\}_{x\in L}$ and $\{M^*(x)\}_{x\in L}$ are identical, i.e.*

$$\{\langle P, V^*\rangle(x)\}_{x\in L} \equiv \{M^*(x)\}_{x\in L}.$$

We emphasize that a distribution may have several different ways of being generated. For example, consider the uniform distribution over n bits. The normal way of generating such a distribution would be to toss n coins and to write down their outcome. Another way of doing it would be to toss $2*n$ coins, to ignore the coins in the even positions, and only to output the values of the coins in odd positions. These are two different ways of producing the same distribution.

Back to our definition, the two distributions $\{\langle P, V^*\rangle(x)\}_{x\in L}$ and $\{M^*(x)\}_{x\in L}$ have totally different ways of being produced. The first one is produced by interaction of the two machines and the second is produced by a traditional probabilistic polynomial time machine.

Consider a "honest" verifier. The two distributions are supposed to be exactly the same when x is in L. As we have seen in the example of uniform distribution, the two generated in different ways distributions may be identical, and our definition, indeed, requires them to be identical. The verifier has several other parameters, except of random coins and partial message history, and we do not fix them. The probability that the verifier accepts x, when $x \in L$, depends on these parameters. However, by completeness requirement of interactive proof system, this probability should be close to 1 for any possible values of the parameters.

When x is in L, M^* accepts with very high probability, but when x is not in L, it is not required that distribution $\{M^*(x)\}$ would be similar to $\{\langle P, V^*\rangle(x)\}$, and it may not happen. So machine M^*, which is called a simulator, simulates the interaction, assuming that x is in L. When x is not in L, it can do total rubbish.

To emphasize this point consider the following example.

Example 1.1: Consider a verifier V that satisfies the definition of interactive proof system, and so when $x \in L$, V will accept with very high (close to 1) probability. When $x \notin L$, V will accept with very low (close to 0) probability. Such $\langle P, V\rangle$ can be simulated by a trivial machine M which always accepts. When $x \in L$ the distributions $\langle P, V\rangle(x)$ and $M(x)$ are very close (the first is very close to 1 and the second is identically 1), whereas when $x \notin L$ the two distributions are totally different (the first one is close to 1 and the second one is identically 0). This behavior should not surprise, because if we would be able to simulate $\langle P, V\rangle$ by some non-interactive probabilistic polynomial time M it would follow that $IP \subseteq BPP$, whereas the definition we have introduced is much more general.

Example 1.2: Before we introduce an example of zero-knowledge interactive proof system, let us first describe an interactive proof system that "hides" some information from the verifier. Although, the system is not zero-knowledge, the verifier can not determine any particular bit of the "real" proof with probability bigger than 1/2.

Consider an NP relation R' that satisfies $(x, w') \in R$ if and only if $(x, \overline{w'}) \in R$. Such a relation can be created from any NP relation R by the following modification

$$R' \underline{\underline{\triangle}} \{(x, 0w) : (x, w) \in R\} \cup \{(x, 1w) : (x, \overline{w}) \in R\}.$$

Now we are suggesting the following interactive proof system by which the prover just sends to the verifier a witness w'.

$$\underline{Prover} \quad x \quad \underline{Verifier}$$
$$\xrightarrow[w']{\qquad\qquad}$$

Given the input x, the prover selects uniformly at random either w' or $\overline{w'}$, and sends one of them to the verifier. Both w' and $\overline{w'}$ are witnesses for x and so it is a valid interactive proof system. But if the verifier is interested to get some individual bit of the witness w', he has no way to do it using the data that he obtained from the prover. Indeed, each individual bit that the verifier receives from the prover is distributed uniformly and the verifier could produce such distribution by itself without interaction with the prover. However, we observe that the verifier gets some info about the witness, since it knows that he received either the witness or its complement.

Except of perfect zero-knowledge we define several more liberal notions of the same class. One of them requires that the distributions will be statistically close. By statistically close we mean that the variation distance between them is negligible as the function of the length of input x.

Definition 17.4 *The distribution ensembles* $\{A_x\}_{x\in L}$ *and* $\{B_x\}_{x\in L}$ *are* statistically close *or have* negligible variation distance *if for every polynomial* $p(\cdot)$ *there exists integer* N *such that for every* $x \in L$ *with* $|x| \geq N$ *holds*

$$\sum_{\alpha} |Prob[A_x = \alpha] - Prob[B_x = \alpha]| \leq \frac{1}{p(|x|)}.$$

Definition 17.5 *Let* (P, V) *be an interactive proof system for some language* L*. We say that* (P, V)*, actually* P*, is* statistical zero-knowledge *proof system (SZK) or* almost perfect *interactive proof system if for every probabilistic polynomial time verifier* V^* *there exists non-interactive probabilistic polynomial time machine* M^* *such that the ensembles* $\{\langle P, V^*\rangle(x)\}_{x\in L}$ *and* $\{M^*(x)\}_{x\in L}$ *are statistically close.*

Even more liberal notion of zero-knowledge is computational zero-knowledge interactive proof system.

Definition 17.6 *Two ensembles* $\{A_x\}_{x\in L}$ *and* $\{B_x\}_{x\in L}$ *are computationally indistinguishable if for every probabilistic polynomial time distinguisher* D *and for every polynomial* $p(\cdot)$ *there exists integer* N *such that for every* $x \in L$ *with* $|x| \geq N$ *holds*

$$|Prob[D(x, A_x) = 1] - Prob[D(x, B_x) = 1]| \leq \frac{1}{p(|x|)}.$$

The probabilistic polynomial time distinguisher D is given an index of the distribution in the ensemble. This is a general notion of indistinguishability of ensembles indexed by strings.

Definition 17.7 *Let* (P, V) *be an interactive proof system for some language* L*. We say that* (P, V)*, actually* P*, is computational* zero-knowledge *proof system (CZK) if for every probabilistic polynomial time verifier* V^* *there exists non-interactive probabilistic polynomial time machine* M^* *such that the ensembles* $\{\langle P, V^*\rangle(x)\}_{x\in L}$ *and* $\{M^*(x)\}_{x\in L}$ *are computationally indistinguishable.*

We should be careful about the order of quantifying in this definition. First we have to determine the verifier V^* and the simulating machine M^*, and then we should check whether the distributions are indistinguishable by "trying all possible" probabilistic polynomial time distinguishers D.

Typically, when we say zero-knowledge, then we mean computational zero-knowledge. This is the most liberal notion, but from the point of view of cryptography applications it is good enough, because, basically, it says that the non-interactive machine M^* can simulate the interactive system $\langle P, V^* \rangle$ in such a way, that "no one" can distinguish between them. Essentially, the generated distributions are close in the same sense as distribution generated by pseudo-random generator is close to the uniform distribution. The idea is that if the machine M^* is able to generate by itself, without interaction, the distribution, that is very close in computational sense to the distribution generated by V^*, that interacts with P, then V^* gains nothing from interaction with P.

Observe that zero-knowledge interactive proof system definition imposes three requirements. Completeness and soundness requirements follows from interactive proof system definition, and zero-knowledge definition imposes the additional condition.

The completeness condition fixes both the prover and the verifier, and states that when the both parties follow the prescribed protocol (both parties are "honest") and $x \in L$, then the verifier accepts with high probability. Observe that if either prover or verifier are "dishonest" the condition may not hold. Indeed, we can not quantify this condition over all verifiers, since some verifier may always reject and then of course the probability of accepting will be zero. On the other hand, if the prover is "dishonest", or in other words, there is another prover instead of P, then it may send some rubbish instead of witness in the NP case or instead of following the prescribed protocol in the general case, and the verifier will accept only with low probability, hence the condition will not hold.

The soundness condition fixes only the verifier and protects his interests. It says, that the verifier doesn't need to trust the prover to be honest. Soundness condition quantifies over all possible provers and says, that no matter how the prover behaves he has very small probability of convincing the verifier to accept a wrong statement. Of course, it is correct only for the fixed verifier V and not for a general one, since we may think about a verifier that always accepts. For such a verifier the probability to accept a wrong statement is 1, hence the soundness condition does not hold for him.

And the zero-knowledge condition protects the prover. It tells the prover: "You are entering the protocol which enables you to convince the other party, even if the other party does not trust you and does not believe you. You can be convinced, that you do not really give the other party more than you intended to (i.e. that the statement is true, nothing beyond that)."

The zero-knowledge condition fixes the prover and quantifies over all verifiers. It says that for any verifier, sophisticated and dishonest as much as he may be, he can not gain from the prover (which obeys the protocol, again bad prover may send all the information to the verifier and then, of course, zero-knowledge condition will not hold) more that the verifier could gain without interacting with the prover.

We finish the section by proving that BPP is contained in PZK. Indeed, any probabilistic polynomial time algorithm may be viewed as an interactive proof system without prover. In such a system no verifier can gain knowledge from the prover, since there is no prover. Thus the system is zero-knowledge. We formalize these considerations in the following claim.

Proposition 17.1.1 $BPP \subseteq PZK$.

Proof: Consider an interactive proof system in which the verifier V is just the probabilistic polynomial time algorithm, that decides the language L. Such V exists, since $L \in BPP$. The

prover P is just a deterministic machine, that never sends data to the verifier. $< P, V >$ is an interactive proof system, since V will accept with probability $\geq \frac{2}{3}$, when $x \in L$, and will accept with probability $\leq \frac{1}{3}$, when $x \notin L$, since $L \in BPP$, and V decides L. Hence, the completeness and soundness conditions hold. Clearly, it is perfect zero-knowledge since for every V^*, the distribution that $< P, V^* >$ generates is identical to the distribution generated by V^* itself. ∎

17.2 Graph Isomorphism is in Zero-Knowledge

Let $ISO \underline{\underline{\triangle}} \{(\langle G_1 \rangle, \langle G_2 \rangle) \mid G_1 \cong G_2\}$.

We assume that $ISO \notin BPP$, since otherwise $ISO \in ZK$, by Claim 1.1. So we are interested in showing zero-knowledge interactive proof systems for languages that are not in BPP, or at least are conjectured not to be in BPP.

Next, we introduce an interactive protocol proving that two graphs are isomorphic. The trivial interactive proof would be that the prover will send to the verifier the isomorphism, but this gives more information that the mere fact that the two graphs are isomorphic.

Instead of sending the isomorphism, which is not a good idea, we will use the following construction.

Construction 2.1 (Perfect Zero-Knowledge Proof for Graph Isomorphism)

- Common Input: A pair of two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Let ϕ be an isomorphism between the input graphs, namely ϕ is a 1-1 and onto mapping of the vertex set V_1 to the vertex set V_2 so that $(u, v) \in E_1$ if only if $(\phi(u), \phi(v)) \in E_2$. Suppose that $|V_1| = |V_2| = n$ and the vertices of the both graphs are the numbers from 1 to n.

- Prover's first Step (P1): The prover selects a random isomorphic copy of G_2, and sends it to the verifier. Namely, the prover selects at random, with uniform probability distribution, a permutation π from the set of permutations over the vertex set V_1, and constructs a graph with vertex set V_1 and edge set

$$F \stackrel{def}{=} \{(\pi(u), \pi(v)) : (u, v) \in E_1\}$$

  The prover sends $H = (V_1, F)$ to the verifier.

- Motivating Remark: If the input graphs are isomorphic, as the prover claims, then the graph sent in step P1 is isomorphic to both input graphs. However, if the input graphs are not isomorphic then no graph can be isomorphic to both of them.

- Verifier's first Step (V1): Upon receiving a graph, $G' = (V', E')$, from the prover, the verifier asks the prover to show an isomorphism between G' and one of the input graph, chosen at random by the verifier. Namely, the verifier uniformly selects $\sigma \in \{1, 2\}$, and sends it to the prover (who is supposed to answer with an isomorphism between G_σ and G').

- Prover's second Step (P2): If the message, σ, received from the verifier equals 1 then the prover sends π to the verifier. Otherwise (i.e., $\sigma \neq 1$), the prover sends $\pi \circ \phi$ (i.e., the composition of π on ϕ) to the verifier. (Remark: the prover treats any $\sigma \neq 1$ as $\sigma = 2$.)

- Verifier's second Step (V2): If the message, denoted ψ, received from the prover is an isomorphism between G_σ and G' then the verifier outputs 1, otherwise it outputs 0.

For the schematic representation of the protocol, see the Diagram 2.1.

The verifier program presented above is easily implemented in probabilistic polynomial time. In case the prover is given an isomorphism between the input graphs as auxiliary input, also the prover's program can be implemented in probabilistic polynomial time. We now show that the above pair of interactive machines constitutes an almost perfect zero-knowledge interactive proof system for the language ISO.

Diagram 2.1

Prover		*Verifier*
$\pi \in_R Sym([n])$		
$H \longleftarrow \pi G_1$		
	$\xrightarrow{H}$	
		$\sigma \in_R \{1,2\}$
	$\xleftarrow{\sigma}$	
if $\sigma = 1$ then send $\psi = \pi$, otherwise $\psi = \pi \circ \phi^{-1}$		
	$\xrightarrow{\psi}$	
		Accept if and only if $H = \psi(G_\sigma)$

Theorem 17.8 *The construction above is an almost perfect zero-knowledge interactive proof system.*

Proof:

1. Completeness.

If the two graphs are isomorphic, we claim that the verifier will always accept. Indeed, if $\sigma = 1$, then the verifier comes to the end of the protocol with $H = \pi(G_1)$ and permutation $\psi = \pi$. The verifier checks whether

$$H = \psi(G_\sigma). \tag{17.1}$$

We observe that $\pi(G_1) = H$, and $\psi(G_\sigma) = \psi(G_1) = \pi(G_1)$, implying (17.1), hence the verifier accepts.

If $\sigma = 2$, then the verifier comes to the end of the protocol with $H = \pi(G_1)$ and permutation $\psi = \pi \circ \phi$. Again he verifies whether (l24:eq) holds. We observe that $\pi(G_1) = H$ and $\psi(G_\sigma) = \pi\phi(G_2) = \pi(G_1)$, again implying (l24:eq), hence the verifier accepts in both cases with probability 1.

The intuition is very simple. If the two graphs are isomorphic and the prover created the isomorphic copy to one of them, he should have no problem in showing isomorphism to each of the graphs.

2. Soundness.

Let $G_1 \ncong G_2$. Consider any prover P^*. If it sends to V a graph H, which is not isomorphic neither to G_1 nor to G_2, then this prover will have no way later to present an isomorphism from G_σ (no matter whether $\sigma = 1$ or 2) to H, *since there is no such isomorphism.* So, in this case, the probability of P^* to convince the verifier that $(\langle G_1 \rangle, \langle G_2 \rangle) \in ISO$, is zero. Suppose P^* sends an H that is isomorphic either to G_1 or to G_2. Without loss of generality, assume $H \cong G_1$. Then if the verifier randomly selected $\sigma = 1$, then P^* will be able to show the isomorphism between H and $G_\sigma = G_1$. Otherwise, if the verifier randomly selected $\sigma = 2$, then there is no isomorphism between $H \cong G_1$ and $G_\sigma = G_2$ (as otherwise G_1 and G_2 would be isomorphic), hence P^* will not be able to find one, despite his unlimited computational power. Hence, in this case, P^* will have probability of exactly $\frac{1}{2}$ to convince the verifier that $(\langle G_1 \rangle, \langle G_2 \rangle) \in ISO$ and we have shown that it is his optimum strategy. So

$$Prob[< P^*, V > (\langle G_1 \rangle, \langle G_2 \rangle) = accept \mid G_1 \ncong G_2] \leq \frac{1}{2}$$

for every prover P^*. By executing this protocol twice sequentially, we obtain

$$Prob[< P^*, V > (\langle G_1 \rangle, \langle G_2 \rangle) = accept \mid G_1 \ncong G_2] \leq \frac{1}{4},$$

hence, satisfying the soundness condition.

3. Zero-knowledge.

There is no other way to prove that the protocol is zero-knowledge, except of building a simulator and proving that it really generates the same distribution.

Simulator M^***.** By definition of zero-knowledge we have to show that the distributions are the same when we are given an input from the language. On input $x \stackrel{def}{=} (\langle G_1 \rangle, \langle G_2 \rangle)$, simulator M^* proceeds as follows:

1. Setting the random-tape of V^*: *Let $q(\cdot)$ denote a polynomial bounding the running time of V^*. The simulator M^* starts by uniformly selecting a string $r \in \{0,1\}^{q(|x|)}$, to be used as the contents of the random-tape of V^*.*

2. Simulating the prover's first step (P1): *The simulator M^* selects at random,with uniform probability distribution, a "bit" $\tau = \{1, 2\}$ and a permutation ψ from the set of permutations over the vertex set V_τ. It then constructs an isomorphic copy G'' of the graph G_τ, i.e. $G'' = \psi(G_\tau)$.*

3. Simulating the verifier's first step (V1): *The simulator M^* initiates the execution of V^* by placing x on V^*'s common-input-tape, placing r (selected in step (1) above) on V^*'s random-tape, and placing G'' (constructed in step (2) above) on V^*'s incoming message-tape. After executing a polynomial number of steps of V^*, the simulator can read the outgoing message of V^*, denoted σ. Let us assume, without loss of generality, that the message sent by V^* is either 1 or 2. Indeed, if V^* sends $\sigma \notin \{1, 2\}$, then the prover has nothing to do with it and we may augment the prover P with a rule to ignore $\sigma \notin \{1, 2\}$ and just to wait for a "valid" σ. This would be very easy to simulate.*

4. Simulating the prover's second step (P2): *If $\sigma = \tau$ then the simulator halts with output (x, r, G'', ψ).*

5. Failure of the simulation: *Otherwise (i.e. $\sigma \neq \tau$), the simulator halts with output $\perp$.*

As could be seen from (4) we output the full view of the verifier. We stress that the simulator "has no way to know" whether V^* will ask to see an isomorphism to G_1 or to G_2.

This description of the simulator machine may confuse. Indeed, the definition of zero-knowledge considers the distributions of two random variables $\langle P, V^* \rangle (x)$ and $M^*(x)$, the outputs of $\langle P, V^* \rangle$ and M^* respectively. On the other hand, here M^* returns its whole view, that consists of all the data it possesses, specifically, x, r, G'' and ψ. This inconsistency can be treated by considering $\langle P, V^* \rangle (x)$ and $M^*(x)$ in the zero-knowledge definition as the views of V^* and M^* respectively, and by showing that this approach is equivalent to our definition.

Definition 17.9 *Let (P, V) be an interactive proof system for some language L. We say that (P, V), actually P, is* perfect zero-knowledge(PZK) by view *if for every probabilistic polynomial time interactive machine V^* there exists an (*ordinary*) probabilistic polynomial time machine M^* so that for every $x \in L$ holds*

$$\left\{ view_{(P,V^*)}(x) \right\}_{x \in L} \equiv \left\{ M^*(x) \right\}_{x \in L},$$

where $view_{(P,V^*)}(x)$ is the final view of V^* after running $\langle P, V^* \rangle$ on input x and $M^*(x)$ is, as usual, the output of M^* after running on input x.

Claim 17.2.1 *An interactive proof system is perfect zero-knowledge if and only if it is perfect zero-knowledge by view.*

Proof: One direction is straightforward. Suppose there is a probabilistic polynomial time machine M^*, which for every input $x \in L$ outputs $M^*(x)$, that is distributed identically to the view of V^* at the end of execution of $\langle P, V^* \rangle$ on x. We observe that the last step of V^*, i.e. printing the output, is done without interaction with the prover. Note also that M^* may, instead of printing the output, write it down on its work-tape. Then M^* has on its work-tape the final view of V^*. Hence, it is capable to perform the last step of V^* and output the result and so the modified $M^*(x)$ is identical to $\langle P, V^* \rangle$, completing the proof of this direction.

In the opposite direction, we suppose that for every V^* there is a non-interactive probabilistic polynomial time machine M^*, which prints the same output, when it runs on x (for every $x \in L$), as V^* when $\langle P, V^* \rangle$ machine runs on x.

Consider some particular V^*. We need to show that there is a machine that for every $x \in L$ prints at the end of its execution on x the output identical to the view V^* at the end of execution of $\langle P, V^* \rangle$ on x. To see it, consider a verifier V^{**}, that behaves exactly like V^*, but outputs its whole view (i.e., it emulates V^* except that at the end it outputs the view of V^*). There is a machine M^{**}, such that its output $M^{**}(x)$ is distributed identically to the output of V^{**}, i.e. to the view of V^*. Thus M^{**} is the required machine. It completes the proof of the second direction, establishing the equivalency of the definitions. ■

Recall that the Definitions 17.3 and 17.9 both require that for every probabilistic polynomial time interactive machine V^* there exists an (*ordinary*) probabilistic polynomial time machine M^*

with certain properties. Observe that in the proof of the non-trivial (i.e., second) direction of the above claim we use the fact that for every V^{**} (constructed out of V^*) there is a corresponding simulator. We stress that this was not used in the first direction in which we did not modify V^* (but rather M^*).

Statistical zero-knowledge by view and computational zero-knowledge by view are defined analogously. Similar claims about equivalency between statistical zero-knowledge and statistical zero-knowledge by view, and between computational zero-knowledge and computational zero-knowledge by view can be proved using the same argument as in the proof of Claim 17.2.1.

We claim that, when two graphs are isomorphic, then H gives no information on τ, because we can draw a correspondence between the possible mappings that can generate H from G_1 and the possible mappings that can generate H from G_2 by the isomorphism between the two graphs. It follows that

Claim 17.2.2 *Let $x = (\langle G_1 \rangle, \langle G_2 \rangle) \in ISO$. Then for every string r, graph H, and permutation ψ, it holds that*

$$Prob\left[view_{(P,V^*)}(x) = (x, r, H, \psi)\right] = Prob\left[M^*(x) = (x, r, H, \psi) \mid (M^*(x) \neq \perp)\right].$$

Proof: Let $m^*(x)$ describe $M^*(x)$ conditioned on its not being $\perp$. We first observe that both $m^*(x)$ and $\text{view}_{(P,V^*)}(x)$ are distributed over quadruples of the form $(x, r, \cdot, \cdot)$, with uniformly distributed $r \in \{0,1\}^{q(|x|)}$, for some polynomial $q(\cdot)$. Let $v(x, r)$ be a random variable describing the last two elements of $\text{view}_{(P,V^*)}(x)$ conditioned on the second element equals r. Similarly, let $\mu(x, r)$ describe the last two elements of $m^*(x)$ (conditioned on the second element equals r). Clearly, it suffices to show that $v(x, r)$ and $\mu(x, r)$ are identically distributed, for every x and r. Observe that once r is fixed the message σ sent by V^* on common input x, random-tape r, and incoming message H, is uniquely defined. Let us denote this message by $v^*(x, r, H)$. We need to show that both $v(x, r)$ and $\mu(x, r)$ are uniformly distributed over the set

$$C_{x,r} \stackrel{def}{=} \{(H, \psi) : H = \psi(G_{v^*(x,r,H)})\}.$$

The proof is slightly non-trivial because it relates (at least implicitly) to the automorphism group of the graph G_2 (i.e., the set of permutations π for which $\pi(G_2)$ is identical, not just isomorphic, to G_2). For simplicity, consider the special case in which the automorphism group of G_2 consists of merely the identity permutation (i.e., $G_2{=}\pi(G_2)$ if and only if π is the identity permutation). In this case, $(H, \psi) \in C_{x,r}$ if and only if H is isomorphic to both G_1 and G_2 and ψ is the isomorphism between H and $G_{v^*(x,r,H)}$. Hence, $C_{x,r}$ contains exactly $|V|!$ pairs, each containing a different graph H as the first element, proving the claim in the special case.

For the proof of the general case we refer the reader to [3] (or to [1]). ■

Recall that to prove perfect zero-knowledge we need to show that $\text{view}_{(P,V^*)}(x)$ and $M^*(x)$ are identically distributed. Here, it is not the case. Although, $\text{view}_{(P,V^*)}(x)$ and $M^*(x)|(M^*(x) \neq \perp)$ are identically distributed, when $M^*(x) = \perp$ the distributions are totally different. A common way to overcome this difficulty is to change the definition of perfect zero-knowledge, at least a bit. Suppose we allow the simulator to output a special symbol which we call "failure", but we require that this special symbol is outputted with probability at most 1/2. In this case the construction

would satisfy the definition and we could conclude that it is a perfect zero-knowledge proof (under the changed definition).

But recall that Theorem 17.8 states that the construction is almost perfect zero-knowledge. This is indeed true, without any change of the definitions, if the simulator reruns steps (2)-(4) of the construction $|x|$ times. If, at least once, at step (4) σ is equal to τ, then output (x, r, G'', ψ). If at all $|x|$ trials $\sigma \neq \tau$, then output rubbish. In such a case the simulation will not be perfect, but will be statistically close, because the statistical difference will be $2^{-|x|}$.

It remains to show that the running time of the simulator is polynomial in $|x|$. In the case when we run it $|x|$ times, it is obvious, concluding our proof that the interactive proof system is almost perfect zero-knowledge. In the case when we change the definition of perfect zero-knowledge in the described above way, we are done by one iteration, hence the running time is against polynomial

Another possibility is to allow the simulator to run expected polynomial time, rather than strict polynomial time, in such a case the interpretation would be to rerun steps (2)-(4), until the output is not $\perp$. Every time we try, we have a success probability of exactly $1/2$. Hence, the expected number of trials is 2.

This concludes the proof of the Theorem 17.8. ■

These definitions, one allowing the failure probability and another allowing an expected polynomial time, are not known to be equivalent. Certainly, if we have a simulator which with probability at most $1/2$ outputs the failure, then it can be always converted to one which runs in expected polynomial time. But the opposite direction is not known and is not clear.

17.3 Zero-Knowledge Proofs for NP

17.3.1 Zero-Knowledge NP-proof systems

We want to show why it was essential to introduce the interactive proofs in order to discuss zero-knowledge (in a non-trivial way). One can also define zero-knowledge for NP-proofs, but by the following claim such proofs exist only for BPP (and are thus "useless").

Proposition 17.3.1 *Let L be a language that admits zero-knowledge NP-proof system. Then $L \in BPP$.*

Proof: For the purpose of the proof we use only the fact that an "honest" verifier V, which outputs its whole view, can be simulated by a probabilistic polynomial time machine M.

(One may think that in view of Claim 17.2.1 there is no point to specify that V outputs its whole view. It is not correct. In the proof of the claim we used the fact that if an interactive proof system is zero-knowledge, then *for any* verifier V^* there is a simulator M^* with certain properties. Once we fix a verifier, such kind of argument no longer works. It is the reason that we specify the output of V.)

Let L be the language that the NP proof system decides and R_L its NP relation. Let $x \in L$. The view of the verifier when interacting with a prover on input x will be the input itself and the message, call it w, that he received. Since we consider an honest prover, the following holds

$$view_{(V,P)}(x) = (x, w) \in R_L.$$

Simulator M simulates this view on input x. We show that $(x, M(x)) \in R_L$ with high probability, specifically

$$Prob[(x, M(x)) \in R_L] \geq \frac{2}{3}. \tag{17.2}$$

Also, we will show that for $x \notin L$

$$Prob[(x, M(x)) \in R_L] = 0 < \frac{1}{3}, \tag{17.3}$$

hence, M is the probabilistic polynomial time algorithm for L, implying $L \in BPP$.

First, discuss the case of $x \in L$ and suppose for contradiction that

$$Prob[(x, M(x)) \in R_L] < \frac{2}{3}. \tag{17.4}$$

Then we claim that there is a deterministic polynomial time distinguisher D, that distinguishes between the two distributions $\langle P, V\rangle(x)$ and $M(x)$ with non-negligible probability. Consider $D(\alpha)$ which is defined as 1 if $\alpha \in R_L$, and 0 otherwise, where α is a pair of the form (x, w). Obviously, for $x \in L$,

$$Prob[D(x, \langle P, V\rangle(x)) = 1] = 1$$

since P is a "honest" prover, i.e. a prover that supplies witness for $x \in L$.

Also, from (4) follows that

$$Prob[D(x, M(x)) = 1] < \frac{2}{3}.$$

So there is a non-negligible gap between the two cases. It contradicts the assumption, that the distribution of $M(x)$ is polynomially indistinguishable from the distribution of $\langle P, V\rangle(x)$.

On the other hand, when $x \notin L$, then by the definition of NP, there is no witness y, such that $(x, y) \in R_L$. Particularly, $(x, M(x)) \notin R_L$. In other words,

$$Prob[(x, M(x)) \in R_L] = 0.$$

Concluding the proof of (2) and (3), hence $L \in BPP$. ■

17.3.2 NP $\subseteq$ ZK (overview)

Now, we are going to show that NP has a zero-knowledge interactive proof ($NP \subseteq ZK$).

We will do it assuming, that we have some magic boxes, called commitment schemes, and later we will describe their implementation.

Commitment schemes are used to enable a party to commit itself to a value while keeping it secret. In a latter stage the commitment is "opened" and it is guaranteed that the "opening" can yield only a single value determined in the committing phase. Commitment schemes are the digital analogue of non-transparent sealed envelopes. Nobody can look inside the envelopes and know the value. By putting a note in such an envelope a party commits itself to the contents of the note while keeping it secret.

We present a zero-knowledge proof system for one NP-complete language, specifically *Graph 3-Coloring*.

The language *Graph 3-Coloring*, denoted $G3C$, consists of all simple graphs (i.e., no parallel edges or self-loops) that can be *vertex-colored* using 3 colors so that no two adjacent vertices are given the same color. Formally, a graph $G = (V, E)$, is $3 - colorable$, if there exists a mapping $\phi : V \longmapsto \{1, 2, 3\}$, so that $\phi(u) \neq \phi(v)$ for every $(u, v) \in E$.

In general, if we want to build a zero-knowledge interactive proof system for some other NP language, we may just use standard reduction and run our protocol on the reduced instance of the graph colorability. Thus, if we can show a zero-knowledge proof for one $NP-$complete language, then we can show for all. Basically it is correct, although inaccurate. For more details see [3].

One non-zero-knowledge proof is to send the coloring to the verifier, which would check it, and this would be a valid interactive proof system, but, of course, not a zero-knowledge one.

The instructions that we give to the prover can actually be implemented in polynomial time, if we give the prover some auxiliary input, specifically the 3-coloring of the graph, which is the NP-witness that the graph is in $G3C$. Let ψ be a 3-coloring of G. The prover selects at random some permutation. But this time it is not a permutation of the vertices, but rather permutation of the colors

$$\pi \in_R Sym([3]).$$

Then it sets $\phi(v) \underline{\underline{\triangle}} \pi(\psi(v))$, for each $v \in V$ and puts each of these permuted colors in a separate locked box (of the type that was discussed before), and the boxes have marks, so that both the prover and verifier know the number of each box.

$$\boxed{\phi(1)}^1, \boxed{\phi(2)}^2, ..., \boxed{\phi(i)}^i, ..., \boxed{\phi(n)}^n$$

The ϕ-color of vertex i is sent in the box number i. The verifier can not see the contents of the boxes, but the prover claims, that he has sent a legal coloring of the graph, which is a permutation of the original one.

The verifier selects an edge $e = (u, v)$ at random, and sends it to the prover.

$$\underline{P} \xleftarrow{\qquad e = (u, v) \qquad e \in_R Edge(G) \qquad} \underline{V}$$

Basically, it asks to inspect the colors of vertices u and v. It expects to see two different colors, and if they are not, then the verifier knows that something is wrong.

The prover sends back the key to box number u and the key to box number v.

$$\underline{P} \xrightarrow{\qquad key_u \quad and \quad key_v \qquad} \underline{V}$$

The verifier uses the keys to open these two boxes (other boxes remain locked), and looks inside. If he finds two different colors from the set $\{1, 2, 3\}$, then it accepts. Otherwise, he rejects. This completes the description of interactive proof system for $G3C$, that we call $G3C$ protocol.

Even more drastically then before, this will be a very weak interactive proof. In order to make any sense we have to repeat it a lot of times. Every time we repeat it, a prover selects a new permutation π, so crucially the color that the verifier sees in one iteration have nothing to do with the colors, that he sees in other iterations.

Of course, if the prover would always use the same colors, then the verifier would know the original coloring by just asking sufficient number of times, a different edge each time. So it is crucial, that in every iteration the coloring is totally random. We observe that the randomness

of the coloring may follow also from a random choice of the original coloring ϕ and not only from the randomness of the permutation π, which has very small sample space (only 3!=6). A computationally unbounded prover will have no problem with randomly selecting a 3-coloring for a graph. On the other hand, a polynomial time bounded prover has to be feeded with all the 3-colorings as auxiliary inputs, in order to be able to select randomly at uniform an original coloring ϕ.

Observe that zero-knowledge property (when augmenting the definition a bit) is preserved under sequential composition. The error probabilities also decrease if we apply the protocol in parallel. But in general, it is not known that zero-knowledge is preserved under parallel composition. In particular, the protocol which is derived by running $G3C$ protocol in parallel many times is in some sense not zero-knowledge, or at least probably is not zero-knowledge.

Proposition 17.3.2 *$G3C$ protocol is zero-knowledge interactive proof system.*

Proof:

1. Completeness.

If the graph is 3-Colorable, and both the prover and the verifier follow the protocol, then the verifier will always accept. Indeed, since ϕ is a legal coloring, for every edge $e = (u, v)$ holds $\phi(u) \neq \phi(v)$. Hence, for $G \in G3C$

$$Prob[\langle P, V\rangle(\langle G\rangle) = accept] = 1,$$

and we are done.

2. Soundness.

Let ϕ be a color-assignment of graph G that the prover uses in his trial to convince the verifier that the graph is 3-colorable. If $G \notin G3C$, then by definition of 3-Colorability there exists an edge

$$e_0 = (u, v) \in Edges(G),$$

such that either $\phi(u) = \phi(v)$ or $\phi(u) \notin \{1, 2, 3\}$. Without loss of generality, suppose $\phi(x) \in \{1, 2, 3\}$ for all x's. If the verifier asked to open e_0 then by commitment property (specifically, by unambiguity requirement of a commitment scheme, to be discuss it in the next subsection) he will reveal that $\phi(u) = \phi(v)$ and thus he will reject. I.e.

$$\text{Prob}\left[\begin{array}{c}\text{a randomly selected edge}\\ e = (w, z) \text{ satisfies } \phi(w) \neq \phi(z)\end{array}\right] \leq \frac{|Edges(G)\backslash\{e_0\}|}{|Edges(G)|} = 1 - \frac{1}{|Edges(G)|}.$$

Hence, for any prover P^*, for $G \notin G3C$

$$Prob[\langle P^*, V\rangle(G) = accept] \leq 1 - \frac{1}{|Edges(G)|}.$$

By repeating the protocol sequentially, sufficiently many times, specifically $\lceil \log_{(\frac{|Edges(G)-1|}{|Edges(G)|})} \frac{1}{3} \rceil$, we reduce this probability to

$$Prob[\langle P^*, V\rangle(G) = accept|\ G \notin G3C] < \frac{1}{3}$$

for any prover P^*.

3. Zero-Knowledge.

In order to show that $G3C$ protocol is a zero-knowledge proof system, we devise for every verifier V^* the following simulating non-interactive machine M^*, that for every $G \in G3C$ generates a distribution $m^*(\langle G \rangle) = M^*(\langle G \rangle) | \, (M^*(G) \neq \perp)$ identical to the distribution $\langle P, V^* \rangle (\langle G \rangle)$.

Description of M^*

(1) Fix the random coins r of V^*.

(2) Select at random an edge $e' = (u', v')$,

$$e' = (u', v') \in_R Edges(G).$$

(3) V^* sends to the M^* (in fact, it is inter-routine communication, since V^* is incorporated in M^*) boxes that are filled in the following way. All the boxes are filled with garbage, except the two boxes of u' and v', which are filled with two different, randomly selected numbers between 1 and 3, namely

$$c \neq d \in_R \{1, 2, 3\}.$$

Put c in u' box, d in v' box.

(4) If V^* chooses e', M^* sends V^* the keys and the simulation is completed. If the verifier sends a different edge, we rerun everything from step (1).

Totally, rerun everything at most $|Edges(G)|$ times. If at least once V^* selected e' in that iteration, print the output of V^*. Otherwise, output $\perp$.

V^* has no way to know what boxes are filled with garbage and what boxes are filled with something meaningful, since the boxes are "non-transparent" (see below about secrecy requirement of a commitment scheme). Thus the probability, that in each iteration V^* will select e' is $\frac{1}{|Edges(G)|}$. By doing $|Edges(G)|$ iterations we reach a constant probability (of about e^{-1}) of generating distribution exactly identical to $\langle P, V^* \rangle (\langle G \rangle)$.

Consider the distribution of

$$m^*(\langle G \rangle) = M^*(\langle G \rangle) \mid \, (M^*(\langle G \rangle) \neq \perp).$$

For every $G \in G3C$, if M^* did not output $\perp$, then one of the iterations was successful. In this case V^* has selected e', which is a randomly selected edge from the graph G. M^* prints the output of V^* on e'. Thus, the distribution of $m^*(\langle G \rangle)$ is identical to the distribution of the output of V^* when it is given a randomly selected edge from the graph G. Now consider the distribution of $\langle P, V^* \rangle (\langle G \rangle)$. The prescribed prover P is fixed and by the construction it supplies to V^* a randomly selected edge from the graph G. Thus the distribution of $\langle P, V^* \rangle (\langle G \rangle)$ is also identical to the distribution of the output of V^* when it is given a randomly selected edge from the graph. Hence,

$$m^*(\langle G \rangle) = \langle P, V^* \rangle (\langle G \rangle).$$

So if the boxes are perfectly sealed, then $G3C \in PZK$. ■

17.3.3 Digital implementation

In reality we use *commitment scheme* instead of these "sealed envelopes", that we discussed previously. In general, a *commitment scheme* is an efficient *two-phase* two-party protocol through which one party, called the *sender* (S), can commit itself to a *value* so the following two conflicting requirements are satisfied.

1. *Secrecy*: At the end of the first phase, the other party, called the *receiver* (R), does not gain any knowledge of the sender's value. This requirement has to be satisfied for any polynomial-time receiver.

2. *Unambiguity*: Given the transcript of the interaction in the first phase, there exists at most one value which the receiver may later (i.e., in the second phase) accept as legal "opening" of the commitment. This requirement has to be satisfied even if the sender tries to cheat (no matter what strategy it employs).

In addition, we require that the protocol is *viable* in the sense that if both parties follow it then, at the end of the second phase, the receiver gets the value committed to by the sender.

Denote by $S(s, \sigma)$ the message that the sender sends to the receiver when he wants to commit himself to a bit σ and his random coins are s. The secrecy requirement states that, for random s, the distributions of the random variables $S(s, 0)$ and $S(s, 1)$ are indistinguishable by polynomial-size circuits.

Let view of S (R), denoted as $View(S)$ ($View(R)$), be the collection of all the information known to the sender (receiver). Denote by r the random coins of R and let $\overline{m}$ be the sequence of messages that R received from S. Then $View(R) = (r, \overline{m})$. In case of single-round commit, $\overline{m} = S(s, \sigma)$. When the sender S wants to commit itself to a bit σ and his random coins sequence is s, then $View(S) = (s, \sigma)$. The unambiguity requirement states that for all but a negligible fraction of r's, there is no such $\overline{m}$ for which there exist two sequences of random coin tosses of S, s and s' such that

$$View(S) = (s, 0) \ \ and \ \ View(R) = (r, \overline{m})$$

and

$$View(S) = (s', 1) \ \ and \ \ View(R) = (r, \overline{m}).$$

The intuition of this formalism is that if such s and s' would exist, then the receiver's view would not be monosemantic. Instead, it would enable to sophisticated sender to claim that he committed either to zero or to one, and the receiver would not be able to prove that the sender is cheating.

If we think about the analogy of sealed envelopes, we send them and we believe that their contents is already determined. However, if we do not open them, they look the same whatever the contents is.

In the rest of this section we will need commitment schemes with a seemingly stronger secrecy requirement that defined above. Specifically, instead of requiring secrecy with respect to all polynomial-time machines, we will require secrecy with respect to all (not necessarily uniform) families of polynomial-size circuits. Assuming the existence of non-uniformly one-way functions commitment schemes with nonuniform secrecy can be constructed, following the same constructions used in the uniform case.

Proposition 17.3.3 *The interactive proof system for G3C that uses bit commitment schemes instead of the "magic boxes" for sending colors is still zero-knowledge.*

Proof:

1. Completeness.

If the graph is 3-Colorable, the prover (the sender) will have no problem to convince the verifier (the receiver) by sending the "right keys" to the commitment schemes, that contain the colors of the endpoints of the edge that the verifier asked to inspect. More formally, by sending the "right keys" we mean performing the reveal phase of the commitment scheme. It takes the following form:

1. The prover sends to the verifier the bit σ (the contents of the sealed envelope) and its random coins, s, that he has used when he committed to the bit (we call it the *commit phase*).

2. The verifier checks that σ and s and his own random coins r indeed yield messages that the verifier has received in the commit phase. This verification is done by running the segment of the prover's program in which it committed to the bit σ and the verifier's program. Both programs are now run by the verifier with fixed coins and take polynomial time. Observe that the verifier could not run the whole prover's program, since it need not to be polynomial. As previously, the probability of accepting a 3-Colorable graph is 1.

2. Soundness.

The unambiguity requirement of the commitment scheme definition ensures that the soundness is satisfied too. We will have a very small increase in the probability to accept a non-3-Colorable graph, relatively to the case when we used magic boxes. As in the proof of Claim 3.2 let G be a non-3-Colorable graph and ϕ the assignment of colors that the prover uses. If the interactive proof system is based on magic boxes, then the probability to accept G is exactly equal to the probability of selecting a properly colored by ϕ edge from G while selecting one edge uniformly at random. As we have seen in the proof of Claim 3.2, this probability (further denoted p_0) is bounded by $\frac{|Edges(G)|-1}{|Edges(G)|}$. Intuitively, p_0 is the probability that the verifier asked to inspect an edge that is properly colored by ϕ, although ϕ is not a proper coloring.

If the proof system is based on commitment schemes rather than on magic boxes, then except of p_0 there is a probability that the verifier asked to inspect a non-properly colored edge, but the prover has succeeded to cheat him. It may happen only if the verifier's random coins r belong to the fraction of all possible random coins of the verifier, for which there are random coins of the prover, which enable him to pretend that he committed both to 0 and to 1. Unambiguity requirement of the commitment scheme ensures that this fraction is negligible, and hence the probability (further denoted p_1) that r belongs to this fraction is negligible too. Thus we can bound p_1 by $\frac{1}{2*|Edges(G)|}$. So, the total probability to accept a non-3-Colorable graph is bounded by

$$p_0 + p_1 \leq \frac{|Edges(G)| - 1}{|Edges(G)|} + \frac{1}{2 * |Edges(G)|} = \frac{|Edges(G)| - 1/2}{|Edges(G)|}.$$

By repeating the protocol sufficiently many times we can make this probability smaller than $\frac{1}{3}$, thus satisfying the soundness property.

3. Zero-Knowledge.

The zero-knowledge, that will be guaranteed now, is a computational zero-knowledge.

To show this we prove that M^* outputs $\perp$ with probability at most $\frac{1}{2}$, and that, conditioned on not outputting $\perp$, the simulator's output is computationally indistinguishable from the verifier's view in a "real interaction with the prover".

Claim 17.3.4 *For every sufficiently large graph, $G = (V, E)$, the probability that $M^*(\langle G \rangle) = \perp$ is bounded above by $\frac{1}{2}$.*

Proof: As above, n will denote the cardinality of the vertex set of G. Let $e_1, e_2, ..., e_n$ be the contents of the n "sealed envelopes" that the prover/the simulator sends to the verifier. Let $s_1, s_2, ..., s_n$ be the random coins that the prover used in the commit phase while committing to $e_1, e_2, ..., e_n$. Let us denote by $p_{u,v}(G, r, (e_1, e_2, ..., e_n))$ the probability, taken over all the choices of the $s_1, s_2, ..., s_n \in \{0,1\}^n$, that V^*, on input G, random coins r, and prover message $(C_{s_1}(e_1), ..., C_{s_n}(e_n))$, replies with the message (u, v). We assume, for simplicity, that V^* always answers with an edge of G (since otherwise its message is anyhow treated as if it were an edge of G). We claim that for every sufficiently large graph, $G = (V, E)$, every $r \in \{0,1\}^{q(n)}$, every edge $(u, v) \in E$, and every two sequences $\alpha, \beta \in \{1, 2, 3\}^n$, it holds that

$$|p_{u,v}(G, r, \alpha) - p_{u,v}(G, r, \beta)| \leq \frac{1}{2\,|E|}$$

This is proven using the non-uniform secrecy of the commitment scheme.
For further details we refer the reader to [3] (or to [1]). ■

Claim 17.3.5 *The ensemble consisting of the output of M^* on input $G = (V, E) \in G3C$, conditioned on it not being $\perp$, is computationally indistinguishable from the ensemble $view_{(P,V^*)}(\langle G \rangle)_{G \in G3C}$. Namely, for every probabilistic polynomial-time algorithm, A, every polynomial $p(\cdot)$, and every sufficiently large graph $G = (V, E)$,*

$$\left| Pr(A(m^*(\langle G \rangle)) = 1) - Pr(A(view_{(P,V^*)}(\langle G \rangle)) = 1) \right| < \frac{1}{p(|V|)}$$

We stress that these ensembles are very different (i.e., the statistical distance between them is very close to the maximum possible), and yet they are computationally indistinguishable. Actually, we can prove that these ensembles are indistinguishable also by (non-uniform) families of polynomial-size circuits. In first glance it seems that Claim 17.3.5 follows easily from the secrecy property of the commitment scheme. Indeed, Claim 17.3.5 is proven using the secrecy property of the commitment scheme, yet the proof is more complex than one anticipates at first glance. The difficulty lies in the fact that the above ensembles consist not only of commitments to values, but also of an opening of some of the values. Furthermore, the choice of which commitments are to be opened depends on the entire sequence of commitments.

Proof: The proof can be found in [3] (or to [1]). ■

This completes the proof of computational zero-knowledge and of Proposition 17.3.3. ■

Such a bit commitment scheme is not difficult to implement. We observe that the size of the intersection between the space of commitments to 0 and the space of commitments to 1 is a negligible fraction of the size of the union of the two spaces, since a larger intersection would violate the unambiguity requirement. Hence, the space of commitments to 0 and the space of commitments to 1 almost do not intersect. On the other hand, commitments to 0 should be indistinguishable from commitments to 1. Using mechanisms of one-way functions and hard-core bits, we can satisfy the seemingly conflicting requirements.

Consider the following construction.

Let $f : \{0,1\}^n \longrightarrow \{0,1\}^n$ be one-way permutation, and $b : \{0,1\}^n \longrightarrow \{0,1\}$ be its hard-core bit, where n is a security parameter. To commit itself to a bit $v \in \{0,1\}$, the sender S selects uniformly at random $s \in_R \{0,1\}^n$ and sends $(f(s), b(s) \oplus v)$ to the receiver R. R stores this pair as $\alpha = f(s)$ and $\sigma = b(s) \oplus v$.

In the second phase (i.e. when sending the "keys"), S reveals its random coins s. R calculates $v = \sigma \oplus b(s)$, and accepts v if $\alpha = f(s)$. Otherwise, R rejects, since if $\alpha \neq f(s)$, then the sender tries to cheat.

Proposition 17.3.6 *The protocol is a bit commitment scheme.*

Proof:

Secrecy: For every receiver R^* consider the distribution ensembles $\langle S(0), R^* \rangle (1^n)$ and $\langle S(1), R^* \rangle (1^n)$. Observe that

$$\langle S(0), R^* \rangle (1^n) = (f(s), b(s))$$

and

$$\langle S(1), R^* \rangle (1^n) = (f(s), \overline{b(s)}).$$

By definition of hard-core bit $b(\cdot)$ of a one-way function $f(\cdot)$, for every probabilistic polynomial time algorithm A, every polynomial $p(\cdot)$ and for every sufficiently large random string s

$$Pr[A(f(s)) = b(s)] < \frac{1}{2} + \frac{1}{p(|s|)}.$$

In other words, the bit $b(s)$ is *unpredictable* by probabilistic polynomial time algorithm given $f(s)$. Thus the distributions $(f(s), b(s))$ and $(f(s), \overline{b(s)})$ are probabilistic polynomial time indistinguishable. (For the proof of equivalency between indistinguishability and unpredictability see the previous lecture.)

Hence for any probabilistic polynomial time distinguisher D

$$\left| Prob[D(f(s), b(s)) = 1] - Prob[D(f(s), \overline{b(s)} = 1]) \right| < \frac{1}{p(|s|)}$$

proving the secrecy.

Unambiguity: We claim that there is no r for which $(r, \overline{m})$ is ambiguous, where $\overline{m}$ is the sequence of messages that S sent to R. Suppose,that $(r, \overline{m})$ is a possible 0-commitment, i.e. there exist a string s such that, $\overline{m}$ describes the messages received by R, where R uses local coins r and interacts with S, which uses local coins s and has input $v = 0$ and security parameter n. Also, suppose for contradiction that $(r, \overline{m})$ is a possible 1-commitment.

Then there exists s_1 such that $View(S) = (s_1, 0, 1^n)$, $View(R) = (r, \overline{m})$.

And there exists s_2 such that $View(S) = (s_2, 1, 1^n)$, $View(R) = (r, \overline{m})$.

But then $\overline{m}$=$(f(s_1), b(s_1))$=$(f(s_2), \overline{b(s_2)})$. I.e. $f(s_1) = f(s_2)$, implying $s_1 = s_2$ since $f(\cdot)$ is a permutation. But then $b(s_1) = b(s_2)$, contradicting $b(s_1) = \overline{b(s_2)}$. Hence, the assumption that there exists ambiguous $(r, \overline{m})$ leads to the contradiction. Therefore, the unambiguity requirement is satisfied, implying that the protocol is a one-bit commitment scheme. ■

17.4 Various comments

17.4.1 Remark about parallel repetition

Recall that the definition of zero-knowledge requires that for every V^* there exist a probabilistic polynomial time simulator M^* such that for every $x \in L$ the distributions of $\langle P, V^* \rangle (x)$, $M^*(x)$ are identical/statistically close/computationally indistinguishable.

A particular case of this concept is a blackbox zero-knowledge.

Definition 17.10 (Blackbox Zero-Knowledge): *Let (P, V) be an interactive proof system for some language L. We say that (P, V) is* perfect/statistical/computational blackbox zero-knowledge *if there exists an oracle machine M, such that for every probabilistic polynomial time verifier V^* and for every $x \in L$, the distributions of $\langle P, V^* \rangle (x)$, $M^{V^*}(x)$ are identical/statistically close/computationally indistinguishable.*

Recall that M^{V^*} is an oracle Turing machine M with access to oracle V^*. The following theorem is given without proof.

Theorem 17.11 *If there is an interactive proof system (P, V)* (with negligible error probability) *for language L that satisfies*

- *The prescribed verifier V, sends the outcome of each coin it tosses (i.e. the interactive proof system is of public-coin type).*
- *The interactive proof system consists of constant number of rounds.*
- *The protocol is blackbox zero-knowledge.*

Then $L \in BPP$.

Observe that $G3C$ protocol is blackbox zero-knowledge, but its failure probability is $1 - \frac{1}{|E|}$, hence it is not negligible. Blackbox zero-knowledge is preserved under sequential composition. Thus by repeating $G3C$ polynomially many times we obtain a blackbox zero-knowledge protocol with negligible error probability, but the number of rounds of this protocol is no more constant, thus violating the second condition of the theorem.

Blackbox zero-knowledge is not preserved under parallel composition. Indeed, there exist zero-knowledge proofs that when repeated twice in parallel do yield knowledge. Repeating G3C protocol polynomially many times in parallel clearly satisfies all conditions of the Theorem 17.11 except blackbox zero-knowledge. Thus, unless $NP \subseteq BPP$, this parallel protocol is not black-box zero-knowledge.

More generally, our unability to construct an interactive proof system that satisfies all the conditions of Theorem 17.11 $G3C$ does not surprise, since if we could construct such an interactive proof system, then by Theorem 17.11, $G3C$ would belong to BPP and since $G3C$ is NP-complete it would imply $NP \subseteq BPP$.

> Oded's Note: *All known zero-knowledge proof systems are proven to be so via a black-box argument, and it seems hard to conceive an alternative. Thus practially speaking, Theorem 17.11 may be understood as saying that zero-knowledge proofs (with negligible error) for languages outside BPP should either use non-constant number of rounds or use "private coins" (i.e., not be of public-coin type).*

17.4.2 Remark about randomness in zero-knowledge proofs

In interactive proofs it was important that the verifier took random steps. But as we mentioned, the prover could be deterministic. The only advantage of probabilistic prover in an interactive proof system is that it may be more efficient.

On the other hand, the prover in a zero-knowledge proof system has to be randomized, not because it has not enough power to do things deterministically, but rather because a deterministic prover will not be able to satisfy the zero-knowledge requirement. In $G3C$ example, suppose that the prover selected the permutation π in a very complicated and secret way, but deterministically. Then a simple verifier will just exhaust all the edges, when repeating the protocol several times. Hence, such a protocol would not be a zero-knowledge. In general, we may prove that if a language has a zero-knowledge proof in which either prover or verifier is deterministic, then the language is in BPP.

Bibliographic Notes

For the comprehensive discussion of zero-knowledge and commitment scheme see Chapter 4 in [1]. Theorem 17.11 appear in [4]

1. Oded Goldreich, *Foundations of Cryptography – fragments of a book*. February 1995. Revised version, January 1998. Both versions are available from `http://theory.lcs.mit.edu/~oded/frag.html`.

2. Oded Goldreich, *Modern Cryptography, Probabilistic proofs and Pseudorandomness*, Springer Verlag, Berlin, 1998.

3. O. Goldreich, S. Micali, A. Wigderson, *Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. JACM*, Vol. 38, No. 1, pages 691–729, 1991. Preliminary version in *27th FOCS*, 1986.

4. O. Goldreich and H. Krawczyk, *On the Composition of Zero-Knowledge Proof Systems*, *SIAM Journal on Computing*, Vol.25, No. 1, February 1996, pages 169-192.

Lecture 18

NP in PCP[poly,O(1)]

Notes taken by Tal Hassner and Yoad Lustig

Summary: The major result in this lecture is $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$. In the course of the proof we introduce an $\mathcal{NPC}$ language "Quadratic Equations", and show it to be in $\mathcal{PCP}(poly, O(1))$, in two stages : first assuming properties of the proof (oracle) and then testing these properties. An intermediate result that might invoke independent interest is an efficient probabilistic algorithm that distinguishes between linear and far-from-linear functions.

18.1 Introduction

In this lecture we revisit $\mathcal{PCP}$ in hope of giving more flavor of the area by presenting the result $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$. Recall that last semester we were shown without proof "the $\mathcal{PCP}$ theorem" $\mathcal{NP} = \mathcal{PCP}(log(n), O(1))$ (Corollary 12.3 in Lecture 12). (Clearly $\mathcal{PCP}(log(n), O(1)) \subseteq \mathcal{NP}$ as given only logarithmic randomness the verifier can only use its randomness to choose from a polynomial selection of queries to the oracle, therefor the answers to all the possible queries can be encoded in a polynomial size witness (for a more detailed proof see proposition 3.1 in lecture 12).

Two of the intermediate results on the way to proving the $\mathcal{PCP}$ theorem are $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$ and $\mathcal{NP} \subseteq \mathcal{PCP}(log, polylog)$. In this lecture we will only see a proof of $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$.

We recall the definition of a PCP proof system for a language L. A Probabilistically Checkable Proof system for a language L is a probabilistic polynomial-time oracle machine M (called verifier) satisfying:

1. Completeness : For every $x \in L$ there exists an oracle π_x such that : $\Pr[M^{\pi_x}(x) \text{ Accepts}] = 1$.
2. Soundness : For every $x \notin L$ and for every oracle π : $\Pr[M^{\pi}(x) \text{ Accepts}] \leq \frac{1}{2}$.

In addition to limiting the verifier to polynomial time we are interested in two additional complexity measures namely, the *randomness* M uses, and the *number of queries* M may perform. We denote by $\mathcal{PCP}(f(\cdot), g(\cdot))$ the class of languages for which there exists a PCP proof system utilizing at most $f(|x|)$ coin tosses and $g(|x|)$ queries for every input x.

We should note that an easy result is that $\mathcal{PCP}(0, poly) \subseteq \mathcal{NP}$. In $\mathcal{PCP}$ we allow the verifier two advantages over an $\mathcal{NP}$ verifier. The first is probabilistic soundness as opposed to perfect one, (which is not relevant in this case as the verifier is deterministic). The second major advantage is that the $\mathcal{PCP}$ "proof" (oracle) is not limited to polynomial length but is given as an oracle. One may look at the oracle as a proof of exponential length for which the verifier machine has random

access (reading a bit from the memory address $addr$ equivalent to performing a query about $addr$). However the verifier is limited to polynomial time and therefor actually reads only polynomial number of bits. In case the verifier is completely deterministic the bits it will read are known in advance and therefor there is no need to write down the rest of the bits i.e. a polynomial witness (as in $\mathcal{NP}$) will suffice.

18.2 Quadratic Equations

Back to our goal of proving $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$, it is easy to see that for any $L \in \mathcal{NPC}$ proving $L \in \mathcal{PCP}(poly, O(1))$ will suffice, since given any language L_1 in $\mathcal{NP}$ we can decide it by reducing every instance x to it's corresponding instance x_L and using our $\mathcal{PCP}(poly, O(1))$ proof system to decide whether x_L is in L (iff $x \in L_1$).
In this section we introduce an $\mathcal{NPC}$ language that we will find convenient to work with.

Definition 18.1 (Quadratic Equations): *The language* Quadratic Equations *denoted QE consists of all satisfiable sets of quadratic equations over $GF(2)$.*

Since in GF(2) $x = x^2$ for all x we assume with out loss of generality all the summands are either of degree 2 or constants.
QE formulated as a problem :

Problem: QE
Input: A sequence of quadratic equations $\{\sum_{i,j=1}^{n} c_{i,j}^{(k)} x_i x_j = c^{(k)}\}_{k=1}^{m}$.
Task: Is there an assignment to $x_1 \dots x_n$ satisfying all equations ?

Clearly QE is in $\mathcal{NP}$ as a satisfying assignment is an easily verifiable witness. To see that it is $\mathcal{NP} - hard$ we will see a reduction from $3 - SAT$.

Given an instance of $3 - SAT$, $\bigwedge_{i=1}^{m} (l_{i1} \vee l_{i2} \vee l_{i3})$ where each l_{ij} is a literal i.e. either an atomic proposition p or it's negation $\neg p$. We will associate with each clause $C_i = (l_{i1} \vee l_{i2} \vee l_{i3})$ (for example $(p_4 \vee \neg p_7 \vee p_9)$) a cubic equation in the following way:

With each atomic proposition p we associate a variable x_p. Now, looking at C_i if l_{ij} is an atomic proposition p then we set the corresponding factor y_{ij} to be $(1 - x_p)$ otherwise l_{ij} is the negation of an atomic proposition $\neg p$ in which case we set the corresponding factor y_{ij} to be x_p . Clearly, the factor y_{ij} equals 0 iff the literal l_{ij} is true, and the expression $y_{i1}y_{i2}y_{i3}$ equals zero iff the disjunction $l_{i1} \vee l_{i2} \vee l_{i3}$ is true (In our example the clause $(p_4 \vee \neg p_7 \vee p_9)$ gets transformed to the equation "$(1 - x_4)x_7(1 - x_9) = 0$"). Therefor the set of equations $\{y_{i1}y_{i2}y_{i3} = 0\}_{i=1}^{m}$ is satisfiable iff our $3 - CNF$ is satisfiable.

We have reduced $3 - SAT$ to a satisfiability of a set of cubic equations but still have to reduce it to quadratic ones (formally we have to open parenthesis to get our equations to the normal form this can be easily done).

What we need is a way to transform a set of cubic equations into a set of quadratic ones (such that one set is satisfiable iff the other is). The latter can be done as follows:
For each pair of variables x_i, x_j in the cubic system introduce a new variable z_{ij} and a new equation $z_{ij} = x_i x_j$, now go through all the equations and whenever we find a summand of degree 3 ($x_i x_j x_k$) replace it by a summand of degree 2 ($z_{ij} x_k$). Our new equation system is composed of two parts, the first part introduces the new variables ($\{z_{ij} = x_i x_j\}_{i,j=1}^{n}$ quadratic in the size of the original

input), and the second part is the transformed set of equations all of degree 2 (linear in the size of the input)
In our example:
$(1-x_4)x_7(1-x_9)=0 \iff x_7 - x_4x_7 - x_7x_9 + x_4x_7x_9 = 0$ may be replaced by:
$x_4x_7 = z_{4\,7}$ and $x_7 - x_4x_7 - x_7x_9 + z_{4\,7}x_9 = 0$.
The last technical step is to go over the equation set and replace every summand of degree 1 (i.e. x_i) by its square (i.e. x_i^2). Since in GF(2) for all a it holds that $a = a^2$ this is purely a technical transformation that does not change in any way the satisfiability of the equations.
Clearly the new set of equations is satisfiable iff the original set is.

Since the entire procedure of reducing a $3-CNF$ to a set of quadratic equations can be done in polynomial time we have reduced $3-SAT$ to QE.

Note that the trick used to reduce the degree of summands in the polynoms can be iterated to reduce the degree of higher degree summands, however the new equations of the kind $z_{ij} = x_ix_j$ are of degree 2, and therefor such a trick cannot be used to reduce the degree of equations to degree 1. In fact degree 1 equations are linear equations for which we know an efficient procedure of deciding satisfiability (Gaussian Elimination). Thus there can be no way of reducing a set of quadratic equations to a set of linear ones in polynomial time, unless $\mathcal{P} = \mathcal{NP}$.

18.3 The main strategy and a tactical maneuver

The task standing before us is finding a $\mathcal{PCP}(poly, O(1))$ proof system for QE. Intuitively this means finding a way in which someone can present a proof for satisfiability of an equation set (the oracle), which might be very long but one may verify its correctness (at least with high probability) taking only a constant number of "glimpses" at the proof regardless of the size of the equation system.

The existence of such a proof system seems counter-intuitive at first since in the proof systems we are familiar with, any mistake however small in any place of the proof cause the entire proof to be invalid. (However existence of such proof systems is exactly what the "PCP Theorem" asserts.)

In this section we try to develop an intuition of how such a proof system can exist by outlining the main ideas of the proof that $QE \in \mathcal{PCP}(poly, O(1))$. We will deal with the "toy example" of proving that one linear expression can get the value 0 over GF(2) (of course since such a question is decidable in polynomial time there is a trivial proof system in which the oracle gives no information at all (always answers 1), but we will not make use of that triviality).

To develop an intuition we adopt a convention that the proof of validity for an equation system (the oracle in the $\mathcal{PCP}$ setting) is written by an adversary who tries to cheat us into accepting non-satisfiable equations. The goal of our system is to overcome such tries to deceive us (while still enabling the existence of proofs for satisfiable proof systems).

Suppose first that we can restrict our adversary to writing only proofs of certain kinds, i.e. having special properties. For example we may restrict the proofs to encode assignments in the following way :
Fix some encoding of linear expressions in the variables $x_1, \ldots, x_n$ into natural numbers (denote by C(ex) the natural number encoding the expression 'ex'). A reasonable encoding for the linear expression $\sum_{i=1}^{n} \lambda_i x_i$ would be simply the number whose binary expansion is the sequence of bits $\lambda_1\lambda_2 \ldots \lambda_n$).

Suppose the adversary is restricted to writing proofs in which he encodes assignments. To encode an assignment $x_1 = a_0, \ldots, x_n = a_n$, the adversary evaluates all the linear expressions over

$x_1, \ldots, , x_n$ with that specific assignment, and writes the value $ex(a_0, \ldots, a_n)$ at the place $C(ex)$ in the proof. In the $\mathcal{PCP}$ setting this means the oracle π answers $ex(a_0, \ldots, a_n)$ on the query $C(ex)$. For example $\pi(C(x_1 + x_3))$ would have the value $a_1 + a_3$. If we can trust the adversary to comply to such a restriction, all we have to do given an expression ex, is to calculate $C(ex)$ and query the oracle π.

The problem of course is that we cannot restrict the adversary. What we can do is check whether the proof given to us is of the kind we assume it is. To do that we will use properties of such proofs. For example in our case we may try to use the linearity of adding linear expressions: for every two expressions e_1, e_2 it holds that $(e_1 + e_2)(a_0, \ldots, a_n) = e_1(a_0, \ldots, a_n) + e_2(a_0, \ldots, a_n)$. Therefor if the proof is of the kind we assume it to be, then the corresponding values in the proof will also respect addition, i.e. the oracle π must satisfy $\pi(C(e_1 + e_2)) = \pi(C(e_1)) + \pi(C(e_2))$.

In general we look for a characteristic of a special 'kind' of proofs. We want that assuming a proof is of that 'kind', one would be able to check its validity using only $O(1)$ queries, and that checking whether a proof is of that special 'kind' will also be feasible using $O(1)$ queries.

The main strategy is to divide the verification of a proof to two parts :

1. Check whether the proof is of a 'good kind' (the oracle answers have a special property), otherwise reject.

2. Assuming that the proof is of the 'good kind', determine its validity, (for example is the proof encoding a satisfying assignment to the equation system)

Up to this point we have developed the general strategy which we intend to use in our proof. Our characteristic of a 'good' proof would be that it encodes an assignment to the variables in a special way. In Section 4 we will define exactly when is a proof of the 'good kind', and see how 'good' proofs' validity can be checked using $O(1)$ queries. In Section 5 we will see how to check whether a proof is of the 'good kind' in $O(1)$ queries.

The bad news is that our strategy as stated above is infeasible if taken literally. We search for a characteristic of proofs that will enable us to distinguish between 'good' proofs and 'bad' proofs in $O(1)$ queries, but suppose we take a 'good' proof and switch just one bit. The probability that such a 'flawed' proof can be distinguished from the original proof in $O(1)$ queries is negligible. (The probability of even querying about that specific bit is the number of queries over the size of the proof, but the size of the proof is all the queries one may pose to an oracle which is exponential in our case).

On the other hand this problem should not be critical, if the adversary changes only one bit, it seems he doesn't have a very good chance of deceiving us, since we probably won't read that bit anyway and therefor it would look to us as if we were given a proof of the 'good' kind.

It seems that even if distinguishing between 'good' proofs and 'bad' ones is infeasible, it may suffice to distinguish between 'good' proofs and proofs which are 'very different' from 'good' proofs. We should try to find a "kind" of proofs for which we can verify that a given proof "looks similar" to. A proof will be "similar" to "good" if with high probability sampling from it will yield the same results as sampling from some really "good proof".

Since we only sample the proof given to us we can only check that the proof behaves "on average" as a proof with the property we want. This poses a problem in the other part of the proof checking process. When we think of it, the adversary does not really have to cheat in many places to deceive us, just in the few critical ones (e.g. the location $C(ex)$ where ex is the expression we try to evaluate). (Remember our adversary is a fictitious all powerful entity - it knows all about our algorithm. Our only edge is that the adversary must decide on a proof before we choose our random moves).

For example in our "toy system" we decide whether the equation is satisfiable based on the single bit $C(ex)$ (which one can calculate from the expression ex) so the adversary only has to change that bit to deceive us. In general during the verification process we do not ask random queries but ones that have an intent, usually this means our queries have a structure i.e. they belong to a small subset of the possible queries. So the adversary can gain a lot by cheating only on that small subset while "on the average" the proof will still look as of the good kind.

Here comes our tactical maneuver: What we would like is to enjoy both worlds, ask random queries and yet get answers to the queries that interest us (which are from a small subset). It turns out that sometimes this can be done, what we need is another property of the proofs. Suppose that we want to evaluate a linear function L on a point x. We can choose a random shift r and evaluate $L(x+r)$ and $L(r)$ we can now evaluate $L(x) = L(x+r) - L(r)$. If r is uniformly distributed so is $x+r$ (also they are dependent), so we have reduced evaluating L at a specific point (i.e. x) into evaluating L at two random points (i.e. r and $x+r$).

In general we want to calculate the value of a function f in a point x by applying f only to random points, we need some method of calculating $f(x)$ from our point x the shift r and the value of the function on the random points $f(x+r)$ and $f(r)$. If on a random point the chance of a mistake is small enough (i.e. p) then the chance of a mistake in one of the two points $x+r, r$ is at most $2p$ which means we get the value on x with probability greater or equal then $1-2p$. If one can apply such a trick to "good" proofs , we can ask the oracle only random queries, that way neutralizing an adversary attempt to cheat us at "important places". This is called *self-correction* since we can "correct" the value of a function (at an important point) using the same function.

18.4 Testing satisfiability assuming a nice oracle

This section corresponds to the second part of our strategy i.e. we are going to check satisfiability while assuming the oracle answers are nice enough.

The property we are going to assume is as follows:
The oracle π fixes some assignment $x_1 = a_1, \ldots, x_n = a_n$ and when given a sequence of coefficients $\{b_{ij}\}_{i,j=1}^{n}$ answers $\sum_{i,j=1}^{n} b_{ij} a_i a_j$. We assume that the testing phase (to be described in Section 6) ensures us that the oracle π has this property "on the average" i.e. on at most a fraction of 0.01 of the queries we might get an arbitrary answer. The idea is that the oracle encodes a satisfying assignment (if the equations are satisfiable) and that our task is verifying that the assignment encoded by the oracle is indeed satisfying.

Note that we may get the assignment a_i to any specific variable x_i by setting $b_{ii} = 1$ and all the other coefficients $b_{kl} = 0$; But we cannot just find the entire assignment (and evaluate the equations) since that would take a linear number of queries.

We have to check whether the set $\{ \sum_{i,j=1}^{n} c_{i,j}^{(k)} x_i x_j = c^{(k)} \}_{k=1}^{m}$ is satisfiable using only a constant number of queries, the tools at our disposal allow us to evaluate any quadratic equation with the assignment the oracle encodes (also we might have to use our self correction trick to ensure the oracle does not cheat on the questions we are likely to ask. The self correction will work since the assignment is fixed and therefor $\sum_{i,j=1}^{n} b_{ij} a_i a_j$ is just a linear function of the b_{ij}s).

The naive approach of checking every equation in our set will not work for the obvious reason that the number of queries using this approach equals the number of equations which might be linear in the size of the input. We must find a way to "check many equations at once". Our trick

will be random summations i.e. we will toss a coin for each equation and sum the equations for which the result of the toss is 1. If all the equations are satisfied clearly the sum of the equations will be satisfied (we sum both sides of the equations).

Random summation test

1 Choose a random subset S of the equations $\{\sum_{i,j=1}^{n} c_{i,j}^{(k)} x_i x_j = c^{(k)}\}_{k=1}^{m}$

2 Sum the linear expressions at the left hand side of the equations in S and denote $ex_{rsum} = \sum_{k \in S} (\sum_{i,j=1}^{n} c_{i,j}^{(k)} x_i x_j)$.
After rearranging the summands present ex_{rsum} in normal form.

3 Sum the constants at the right hand side of the equations in S and denote $c_{rsum} = \sum_{k \in S} c^{(k)}$.
(Note that if an assignment $a_1, \ldots, a_n$ is satisfying then $ex_{rsum}(a_1, \ldots, a_n) = c_{rsum}$).

4 Query about ex_{rsum} using self correction technique and compare the answer to c_{rsum}. Accept if they are equal and reject otherwise.

What is the probability of discovering that not all the equations were satisfied ? If only one equation is not satisfied we would discover that with probability $\frac{1}{2}$ since if we include that equation in the sum (set S), the left hand side of the sum is not equal to the right hand side no matter which other equations we sum. But what happens if we have more then one unsatisfied equation?
In this case two wrongs make a right since if we sum two unsatisfied equations the left hand side will be equal to the right hand side (this is the wonderful world of GF(2)).

To see that in any case the probability of discovering the existence of an unsatisfied equation is $\frac{1}{2}$, consider the last toss of coin for an unsatisfied equation (i.e. for all the other unsatisfied equations it was already decided whether to include them in the sum or not). If the sum up to now is satisfied then with probability $\frac{1}{2}$ we will include that equation in the sum and therefor have an unsatisfied sum at the end, and if the sum up to now is not satisfied then with probability $\frac{1}{2}$ we will not include that last equation in the sum and remain with an unsatisfied sum at the end.

We have seen that if an assignment does not satisfy all the equations, then with probability $\frac{1}{2}$ it fails the random sum test. All that is left to completely analyze the test (assuming the oracle was tested and found nice) is a little book keeping.

First we must note that the fact the oracle passed the "oracle test" and found nice, does not assure us that all its answers are reliable, there is 0.01 probability of getting an arbitrary answer. In fact since our queries have a structure (they are all partial sums of the original set of equations), we must use self correction which means asking two queries for every query we really want to ask (of the random shift and of the shifted point of interest). During a random summation test we need to ask only one query (of the random sum), using self correction that becomes two queries (of the shifted point and of the random shift) and therefor we have a probability of 0.02 that our result will be arbitrary. Assuming the answers we got were not arbitrary, the probability of detecting an unsatisfying assignment is 0.5 . Therefor the overall probability of detecting a bad assignment is $0.98 * 0.5 = 0.49$ making the probability of failure 0.51. Applying the test twice and rejecting if one of the trials fail we get failure probability of $0.51^2 < 0.3$.

The result of our "book keeping" is that if we can test that the oracle has our property with failure probability smaller then 0.2, then the probability of the entire procedure to fail is smaller than 0.5, and we satisfy the $\mathcal{PCP}$ satisfiability requirement. (In our case the completeness requirement is easy, just note that an oracle encoding a satisfying assignment always pass).

18.5 Distinguishing a nice oracle from a very ugly one

In this section our objective is to present a method of testing whether the oracle behaves "typically well". We shall say that an oracle π behaves "well" if it encodes an assignment in the way defined above, and that π behaves "typicly well" if it agrees on more then 99% of the queries with a function that encodes an assignment that way. Our test should detect an oracle that does not behave "typicly well" with probability greater then 0.8 .

As described in the strategy our approach is to search some characterizing properties of the "good oracles". What does a "good oracle" look like ? Assume π is a good oracle corresponding to an assignment $x_1 = a_1, \ldots, x_n = a_n$. We may look at the oracle as a function $\pi : \{0,1\}^{n^2} \to \{0,1\}$ denoting its argument $\underline{b} = (b_{11}, \ldots, b_{1n}, b_{21}, \ldots, b_{nn})^\top$. Then, since $a_1 \ldots, a_n$ are fixed, π is a linear function of $\underline{b}$ i.e. $\pi(\underline{b}) = \sum_{i,j=1}^{n} \lambda_{ij} b_{ij}$. Furthermore π's coefficients λ_{ij} have a special structure - there exists a sequence of constants $\{a_i\}_{i=1}^n$ s.t. $\lambda_{ij} = a_i a_j$. It turns out that these properties characterize the "good oracles", since if π^* is a linear function for which there exists a sequence of constants $\{a_i^*\}_{i=1}^n$ s.t. π^*'s coefficients are $\lambda_{ij}^* = a_i^* a_j^*$, then π^* is encoding the assignment $x_1 = a_1^*, \ldots, x_n = a_n^*$.

Our "oracle test" is composed of two stages: testing the linearity of π and testing that the linearity coefficients λ_{ij}'s have the correct structure.

18.5.1 Tests of linearity

In order to devise a test and prove its correctness we begin with some formal definitions.

Definition 18.2 (linearity of a function)
A function $f : \{0,1\}^m \to \{0,1\}$ *is called* linear *if there exist constants* $a_1^f, \ldots, a_m^f$ *s.t. for all* $\underline{\sigma} = (\sigma_1, \ldots, \sigma_m)^\top \in \{0,1\}^m$ *it holds that* $f(\underline{\sigma}) = \sum_{i=1}^{m} a_i^f \sigma_i$.

Claim 18.5.1 (alternative definition of linearity)
A function $f : \{0,1\}^m \to \{0,1\}$ *is* linear *iff for every two vectors* $\underline{\sigma^1}, \underline{\sigma^2} \in \{0,1\}^m$ *and every* $\lambda_1, \lambda_2 \in \{0,1\}$ *it holds that:* $\lambda_1 f(\underline{\sigma^1}) + \lambda_2 f(\underline{\sigma^2}) = f(\lambda_1 \underline{\sigma^1} + \lambda_2 \underline{\sigma^2})$.

Proof: Suppose first that f is linear by the definition i.e. there exists constants $a_1^f, \ldots, a_m^f$ s.t. for all $\underline{\sigma} = (\sigma_1, \ldots, \sigma_m)^\top$, $f(\underline{\sigma}) = \sum_{i=1}^{m} a_i^f \sigma_i$. Then for every $\underline{\sigma^1}, \underline{\sigma^2}$ it holds that $\lambda_1 f(\underline{\sigma^1}) + \lambda_2 f(\underline{\sigma^2}) = \sum_{i=1}^{m} a_i^f \lambda_1 \sigma_i^1 + \sum_{i=1}^{m} a_i^f \lambda_2 \sigma_i^2 = \sum_{i=1}^{m} a_i^f (\lambda_1 \sigma_i^1 + \lambda_2 \sigma_i^2) = f(\lambda_1 \underline{\sigma^1} + \lambda_2 \underline{\sigma^2})$.

Suppose now that the claim holds i.e. for every two vectors $\sigma^1, \sigma^2 \in \{0,1\}^m$ $\lambda_1 f(\sigma^1) + \lambda_2 f(\sigma^2) = f(\lambda_1 \underline{\sigma^1} + \lambda_2 \underline{\sigma^2})$. Denote by a_i^f the value $f(\underline{e^i})$ where $\underline{e^i}$ is the i^{th} element in the standard basis i.e. all $\underline{e^i}$'s coordinates but the i^{th} are 0 and the i^{th} coordinate is 1.
Every $\underline{\sigma} \in \{0,1\}^m$ can be expressed as $\underline{\sigma} = \sum_{i=1}^{m} \sigma_i \underline{e^i}$.

Then, $f(\underline{\sigma}) = f(\sum_{i=1}^{m} \sigma_i \underline{e^i})$ and by the claim we get :
$f(\sum_{i=1}^{m} \sigma_i \underline{e^i}) \;=\; \sum_{i=1}^{m} \sigma_i f(\underline{e^i}) \;=\; \sum_{i=1}^{m} \sigma_i a_i^f$ ■

Definition 18.3 (distance from linearity)
Two functions $f, g : \{0,1\}^m \to \{0,1\}$ *are said to be* at distance at least δ *(or* δ far*) if :*
$\Pr_{\sigma \in_R \{0,1\}^m}[f(\sigma) \neq g(\sigma)] \geq \delta$
Two functions $f, g : \{0,1\}^m \to \{0,1\}$ *are said to be* at distance at most δ *(or* δ close*) if :*
$\Pr_{\sigma \in_R \{0,1\}^m}[f(\sigma) \neq g(\sigma)] \leq \delta$
Two functions $f, g : \{0,1\}^m \to \{0,1\}$ *are said to be* at distance δ *if :*
$\Pr_{\sigma \in_R \{0,1\}^m}[f(\sigma) \neq g(\sigma)] = \delta$
A function $f : \{0,1\}^m \to \{0,1\}$ *is said to be* at distance at most δ from linear *if there exists some linear function* $L : \{0,1\}^m \to \{0,1\}$ *s.t.* f *is at distance at most* δ *from* L.
In a similar fashion we define distance at least ϵ from linear *and* distance ϵ from linear.

Notice that since there are only finitely many linear functions $L : \{0,1\}^m \to \{0,1\}$ for every function f there is a closest linear function (not necessarily unique) and the distance from linearity is well defined.

We define now a verification algorithm $A^{(\cdot)}(\cdot)$ that accepts as input a distance parameter ϵ and oracle access to a function f (therefor we actually run $A^f(\epsilon)$), and tries to distinguish between f's which are at distance at least ϵ from linear and linear functions. A iterates a "basic procedure" $T^{(\cdot)}$ that detects functions which are "bad" (ϵ-far from linear) with small constant probability. Using enough iterations ensures the detection of "bad" f's with the desired probability.

Basic procedure T^f:

1 Select at random $\underline{a}, \underline{b} \in_R \{0,1\}^n$.

2 Check whether $f(\underline{a}) + f(\underline{b}) = f(\underline{a} + \underline{b})$, if not reject.

Linearity Tester $A^f(\epsilon)$:

1 Repeat for $\lceil \frac{36}{\epsilon} \rceil$ times the basic procedure $T^{(f)}$

2 Reject if $T^{(f)}$ rejects even once, accept otherwise.

Theorem 18.4 (Linearity testing): *If* f *is linear* T^f *always accepts, and if* f *is at distance at least* ϵ *from linear then* $\Pr[T^f \;\; rejects] \geq p_{rej} \stackrel{\text{def}}{=} \frac{\epsilon}{4}$.

Proof: Clearly if f is linear T will accept. If f is not linear we deal separately with functions close to linear and with functions relatively far from linear.

Denote by δ the distance of f from linearity and let L be a linear function at distance δ from f.

Denote by $G = \{\underline{a} | f(\underline{a}) = L(\underline{a})\}$ the set of "good" inputs on which the functions agree. Clearly $|G| = (1-\delta)2^m$. We shall try to bound from below the probability that an iteration of A rejects.

Since the value of a linear function of every two from the three points $\underline{a}, \underline{b}, \underline{a}+\underline{b}$ fixes the value of third, it is easy to see that the algorithm will reject if two of these points are in G and the third is not in G. Therefor the probability that one iteration rejects is greater or equal to $\Pr[\underline{a} \in G, \underline{b} \in G, (\underline{a}+\underline{b}) \notin G] + \Pr[\underline{a} \in G, \underline{b} \notin G, (\underline{a}+\underline{b}) \in G] + \Pr[\underline{a} \notin G, \underline{b} \in G, \underline{a}+\underline{b} \in G]$.

The three events are clearly disjoint. What might be less obvious is that they are symmetric. It is easy to see $\Pr[\underline{a} \notin G, \underline{b} \in G, (\underline{a}+\underline{b}) \in G] = \Pr[\underline{a} \in G, \underline{b} \notin G, (\underline{a}+\underline{b}) \in G]$. Notice also that instead of choosing $\underline{b}$ at random we may choose $(\underline{a}+\underline{b})$ at random and then $\underline{a}+(\underline{a}+\underline{b}) = \underline{b}$, so the third event is also symmetric and therefor $\Pr[\underline{a} \in G, \underline{b} \in G, (\underline{a}+\underline{b}) \notin G] = \Pr[\underline{a} \notin G, \underline{b} \in G, (\underline{a}+\underline{b}) \in G]$.

Thus the probability of rejection is greater or equal to $3\Pr[\underline{a} \notin G, \underline{b} \in G, (\underline{a}+\underline{b}) \in G]$.
The latter can be presented as $3\Pr[\underline{a} \notin G] \cdot \Pr[\underline{b} \in G, (\underline{a}+\underline{b}) \in G | \underline{a} \notin G]$. By definition of δ it holds that $\Pr[\underline{a} \notin G] = \delta$. It is also clear that
$\Pr[\underline{b} \in G, (\underline{a}+\underline{b}) \in G | \underline{a} \notin G] = 1 - \Pr[\underline{b} \notin G \text{ or } (\underline{a}+\underline{b}) \notin G | \underline{a} \notin G]$.
Therefor the probability of rejection is greater or equal to $3\delta(1 - \Pr[\underline{b} \notin G \text{ or } (\underline{a}+\underline{b}) \notin G | \underline{a} \notin G])$.
Using the symmetry of $\underline{b}$ and $\underline{a}+\underline{b}$ explained above and union bound the probability of rejection is greater or equal to $3\delta(1 - 2\Pr[\underline{b} \notin G | \underline{a} \notin G])$. Since $\underline{a}$ and $\underline{b}$ were chosen independently the probability of rejection is greater or equal to $3\delta(1 - 2\Pr[\underline{b} \notin G]) = 3\delta(1-2\delta)$.

Notice that the analysis above is good for small δ; i.e., for functions which are quite close to linear functions. However the function $3\delta(1-\delta)$ drops to 0 at $\frac{1}{2}$, therefor the analysis is not good enough for functions which are quite far from linear functions. The obvious reason for the failure of the analysis is that if the function is far enough from a linear function then the probability that two of the three points will fall inside the "good" set of points (whose f value is identical to their value under the closest linear function) is small. Thus, we need an alternative analysis for the case of big δ. Specifically, we show that if f is at distance greater or equal $\frac{3}{8}$ then the probability of rejection is at least $\frac{1}{8}$. As the proof is rather long and technical it is given in Appendix B.

Combining both cases, of functions relatively close to linear and functions relatively far from linear we get the desired result. That is, let δ be the distance of f from linear. Note that $\delta \leq \frac{1}{2}$, (since for every function f the expected distance of f from a random linear function is $\frac{1}{2}$). In case $\delta > \frac{3}{8}$ we have a rejection probability of at least $\frac{1}{8} \geq \frac{\delta}{4}$. Otherwise, $\delta \leq \frac{3}{8}$, and we have a rejection probability of at least $3\delta(1-2\delta) > 3\delta(1-2\cdot\frac{3}{8}) = \frac{3}{4}\delta$. ■

Corollary 18.5 *If f is linear then the verification algorithm $A^{(f)}(\epsilon)$ always accepts it. If f is ϵ far from linear then $A^{(f)}(\epsilon)$ rejects it with probability larger then 0.99*

Proof: If f is linear it always passes the basic procedure T. Suppose f is ϵ far from linear then by Theorem 4 the probability that one iteration of the basic procedure T^f rejects is bigger then $\frac{\epsilon}{4}$. Therefor the probability that f passes all the iterations A invokes is smaller then $(1-\frac{3}{4}\delta)^{36\cdot\frac{1}{\epsilon}}$. By the inequality $1-x \leq e^{-x}$ the probability that A accepts is smaller then $e^{-\frac{3}{4}\epsilon\cdot\frac{36}{\epsilon}} = e^{-9} < 0.01$.

(To prove the $1-x \leq e^{-x}$ inequality notice equality holds for 0, and differentiate both sides to see that the right hand side drops slower then the left hand side). ■

18.5.2 Assuming linear π testing π's coefficients structure

If π passes the linearity tester we assume that there exists a linear function $L_\pi : \{0,1\}^{n^2} \to \{0,1\}$ at distance smaller or equal 0.01 from π. For the rest of the discussion L_π stands for some such

fixed linear function.

For π to be nice it is not enough that it is close to a linear L_π, but L_π must also be of the special structure that encodes an assignment. We saw that this means that there exists a sequence $\{a_i\}_{i=1}^n$ s.t. $L_\pi(\underline{b}) = \sum_{i,j=1}^{n} a_i a_j b_{ij}$ in other words L_π's linearity coefficients $\{\lambda_{ij}\}_{i,j=1}^n$ have the special structure which is $\lambda_{ij} = a_i a_j$.

L_π is a linear function from $\{0,1\}^{n^2}$ to $\{0,1\}$ and therefor its natural representation is as a row vector of length n^2 ($1 \times n^2$ matrix). However if we rearrange L_π's n^2 coefficients to an $n \times n$ matrix form the constraint on L_π can be phrased in a very elegant form, namely there exists a vector $\underline{a} = (a_1, \ldots, a_n)^\top$ s.t. $(\lambda_{ij}) = \underline{a}\,\underline{a}^\top$. (Notice this is not a scalar product but an outer product and the result is an $n \times n$ matrix). Notice that $(\underline{a}\,\underline{a}^\top)_{ij} = a_i a_j$ which is exactly what we want of λ_{ij}.

So what has to be checked is that (λ_{ij}) really has this structure i.e. there exists a vector $\underline{a}$ s.t. $(\lambda_{ij}) = \underline{a}\,\underline{a}^\top$. How can this be done ?

Notice that if (λ_{ij}) has this special structure then the vector $\underline{a}$ is exactly the diagonal of the matrix (λ_{ij}). This is the case since $\lambda_{ii} = a_i a_i = a_i^2$ however in GF(2) $x^2 = x$ for every x, leading us to $\lambda_{ii} = a_i$.

The last observation means that we can find out every coefficient a_i in $\underline{a}$ simply by querying with $b_{ii} = 1$ and the rest of the $b_{kl} = 0$ (we will have to use self correction as always). This seems to lead us to a reasonable test. First obtain $\underline{a}$ by querying its coefficients a_i, then construct $\underline{a}\,\underline{a}^\top$. What has to be done now is to check whether (λ_{ij}) is indeed equal to the matrix we have constructed $\underline{a}\,\underline{a}^\top$. A natural approach for checking this equality will be sampling: we have access to (λ_{ij}) as a function since we can query π, we can try to simulate $\underline{a}\,\underline{a}^\top$ as a function and compare their results on random inputs. The problem with this natural approach is that querying about every coefficient a_i would cost a linear number of queries and we can only afford a constant number of queries.

It seems we will have to get by without explicitly constructing the entire $\underline{a}$. As the "next best thing" we may try to "capture" $\underline{a}$ by a random summation of its coefficients. A random sum of $\underline{a}$ coefficients defined by a random string $\underline{r} \in_R \{0,1\}^n$ is the sum of the coefficients a_i for which $r_i = 1$ i.e. $\sum_{i=1}^{n} r_i a_i$. This is the scalar product of $\underline{a}$ with $\underline{r}$ denoted $< \underline{a}, \underline{r} >$. The random sum is of course a function of the random string $\underline{r}$ so we introduce the notation $A(\cdot) =< \underline{a}, \cdot >$ where $A(\cdot)$ is the random sum as a function of $\underline{r}$.

Just as we can find any coefficient a_i of $\underline{a}$ by querying π with the appropriate bit on the diagonal b_{ii} turned on, we can find the result of a sum of coefficients. To get the result of a sum of the coefficients $i_1, \ldots, i_k$ all we have to do is query with $b_{i_1 i_1} = 1, \ldots, b_{i_k i_k} = 1$ and all the other bits 0. The result will be $\sum_{l=1}^{k} x_{i_l}^2 = \sum_{l=1}^{k} x_{i_l}$ which is what we want. (As always we have to use self correction).

How is all this going to help us to compare the matrices $\underline{a}\,\underline{a}^\top$ to (λ_{ij}) ?
Most of us immediately identify an $n \times n$ matrix M with its corresponding linear function from $\{0,1\}^n$ to $\{0,1\}^n$. However such a matrix can also stand for a bilinear function
$f_M : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, (where $f_M(\underline{x}, \underline{y}) = \underline{x}^\top M \underline{y}$).

For matrices of the form $\underline{a}\,\underline{a}^\top$ the operation becomes very simple :
$f_{(\underline{a}\,\underline{a}^\top)}(\underline{x}, \underline{y}) = \underline{x}^\top (\underline{a}\,\underline{a}^\top)\underline{y} = (\underline{x}^\top \underline{a})(\underline{a}^\top \underline{y}) =< \underline{x}, \underline{a} >< \underline{a}, \underline{y} >= A(\underline{x})A(\underline{y})$
(The same result can also be developed in the more technical way of opening the summations).

Our access to $A(\cdot)$ enables us to evaluate the bilinear function represented by $\underline{a}\,\underline{a}^\top$. We can also evaluate the bilinear function represented by (λ_{ij}) since :

$f_{(\lambda_{ij})}(\underline{x},\underline{y}) = x^\top(\lambda_{ij})y = \sum_{i,j=1}^{n} \lambda_{ij}x_iy_j$.
So in order to evaluate $f_{(\lambda_{ij})}(\underline{x},\underline{y})$ we only have to feed $b_{ij} = x_iy_j$ into π. Once again we will use our self correcting technique, this time the structure of the queries does not stand out as in the the case of $A(\cdot)$. Nonetheless, the distribution of queries is not uniform (for example there is a skew towards 0s as it is enough that one of the coordinates x_i or y_j is 0 so that b_{ij} will be 0).

It seems reasonable to test whether $(\lambda_{ij}) = \underline{a}\,\underline{a}^\top$ by testing whether the bilinear functions represented by the two matrices are the same, and do that by sampling. The idea is to sample random vectors $\underline{x},\underline{y} \in \{0,1\}^n$ and check if the functions agree on their value.

Structure test for (λ_{ij}):

1 Select at random $\underline{x},\underline{y} \in_R \{0,1\}^n$.
2 Evaluate $A(\underline{x})$ and $A(\underline{y})$ by querying with coefficients corresponding to a matrix U where $\underline{x}$ (resp. $\underline{y}$) is the diagonal i.e. $U_{ii} = x_i$ and the rest of the coefficients are 0s. (The queries should be presented using self correction).
3 Compute $f_{(\underline{a}\,\underline{a}^\top)}(\underline{x},\underline{y}) = A(\underline{x}) \cdot A(\underline{y})$
4 Query $f_{(\lambda_{ij})}(\underline{x},\underline{y})$ by querying π with $\{x_iy_j\}_{i,j=1}^n$ as the query bits. (Again self correction must be used).
5 Accept if $f_{(\underline{a}\,\underline{a}^\top)}(\underline{x},\underline{y}) = f_{(\lambda_{ij})}(\underline{x},\underline{y})$, Reject otherwise.

We are left to prove that if the matrices (λ_{ij}) and $\underline{a}\,\underline{a}^\top$ differ then if we sample we will get different results of the bilinear functions with reasonably high probability.

Given two different matrices M, N we want to bound from below the probability that for two random vectors $\underline{x},\underline{y}$ the bilinear functions $f_M(\underline{x},\underline{y})$ and $f_N(\underline{x},\underline{y})$ agree.

The question is when $\underline{x}^\top M\underline{y} = \underline{x}^\top N\underline{y}$?
Clearly this is equivalent to $\underline{x}^\top(M-N)\underline{y} = 0$.
Suppose we choose $\underline{y}$ first, if $(M-N)\underline{y} = \underline{0}$ then whatever $\underline{x}$ we choose the result $\underline{x}^\top\underline{0}$ will always be 0 and does not depend on $\underline{x}$. If on the other hand $(M-N)\underline{y} \neq \underline{0}$ then it might be the case that $\underline{x}^\top((M-N)\underline{y}) = 0$ depending on the choice of $\underline{x}$. We will analyze the probabilities for these two events separately.

To analyze the probability of choosing a $\underline{y}$ s.t. $(M-N)\underline{y} = \underline{0}$, denote the column dimension of $(M-N)$ by d. There exist d linearly independent columns $i_1,\ldots,i_d$ of $(M-N)$ assume without loss of generality these are the last d columns $n-d+1,\ldots,n$. Also assume the choice of $\underline{y}$ is made by tossing coins for its coordinates' values one by one and that the coins for the last coordinates $n-d+1,\ldots,n$ are tossed last. Lets look at the situation just before the last d coins are tossed (the rest of the coins have already been chosen).

The value of $(M-N)\underline{y}$ will be the sum of columns corresponding to coordinates of value 1, however at our stage not all the coordinates values have been chosen. At this stage we may look of at the last d columns' "contribution" to the final sum $((M-N)\underline{y})$ as a random variable. Denote by $(M-N)_{\downarrow j}$ the j^{th} column in $(M-N)$ and by $\underline{v_{rand}}$ the random variable $\sum_{k=n-d+1}^{n} y_k \cdot (M-N)_{\downarrow k}$. (The sum of columns corresponding to coordinates of value 1 out of the last d coordinates). The rest of the coordinates "contribution" has already been set. Denote by $\underline{v_{set}}$ the vector $\sum_{k=1}^{n-d} y_k \cdot (M-N)_{\downarrow k}$ (The contribution of the rest of the coordinates).

Clearly $(M-N)\underline{y} = \underline{v_{set}} + \underline{v_{rand}}$ i.e. $(M-N)\underline{y} = \underline{0}$ iff $\underline{v_{set}} = -\underline{v_{rand}}$. The question is can this happen and at what probability ? First note that this can happen - otherwise $\underline{v_{set}}$ is independent

of columns $i_1, \ldots, i_d$ which means the columns dimension is bigger then d. Second notice that since columns $i_1, \ldots, i_d$ are independent there is only one linear combination of them that equals $\underline{v_{set}}$. This means that there is only one result of the coin tosses for coordinates $i_1, \ldots, i_d$ that will make $\underline{v_{rand}}$ equal $\underline{v_{set}}$, i.e. the probability is $2^{-d} = 2^{-dim(M-N)}$.

Assuming $(M-N)\underline{y} \neq \underline{0}$ what is the probability that $\underline{x}^\top((M-N)\underline{y}) = 0$? The question is given a fixed vector that is not $\underline{0}$ what is the probability that its inner product with a randomly chosen vector is 0 ? By exactly the same reasoning used above (and in the random sum argument in Section 4), the probability is $\frac{1}{2}$. (Toss coins for the random vector coordinates one by one and look at the coin tossed for a coordinate which is not 0 in $(M-N)\underline{y}$, there is always a result of the coin flip that would bring the inner product to 0).

Overall the two vectors have the same image if either $(M-N)\underline{y} = 0$ (with probability bounded by $\frac{1}{2}$) or if $\underline{x}^\top((M-N)\underline{y}) = 0$ (with probability $\frac{1}{2}$). Therefor the probability that the two vectors differ is at least $\frac{1}{4}$.

To finish the probability "book keeping" we must take into account the probability of getting wrong answers from the oracle. The test itself has a failure probability of 0.75 . The test uses three queries ($A(\underline{x}), A(\underline{y})$ and $f_{(\lambda_{ij})}(\underline{x}, \underline{y})$) but using self correction each of those becomes two queries. We get a probability of 0.06 of getting an arbitrary answer leading us to probability of 0.81 for the test to fail. Repeating the test 10 independent times we get a failure probability of $0.81^{10} \leq \frac{1}{8}$ and the probability of failure in either the linearity testing or structure test is bounded by $0.125 + 0.01 \leq 0.15$.

The goal of detecting "bad oracles" with probability greater than 0.8 is achieved.

18.5.3 Gluing it all together

Basicly our verification of a proof consists of three stages:

1. linearity testing : a phase in which if the oracle π is 0.01 far from linear we have a chance of 0.99 to catch it (and reject).
   To implement this stage we use the linearity tester $A^\pi(0.01)$. (Distance parameter set to 0.01).

2. structure test : assuming the oracle π is close to linear we test that it encodes an assignment using the "structure test". To boost up probability we repeat the test 10 independent times. At the end of this stage we detect bad oracles with probability greater than 0.8

3. satisfiability test : assuming "good oracle" we use the "random summation" test to verify that the assignment encoded by the oracle is indeed satisfying. (This test is repeated twice).

Bibliographic Notes

This lecture is mostly based on [1], where $\mathcal{NP} \subseteq \mathcal{PCP}[\text{poly}, O(1)]$ is proven (as a step towards proving $\mathcal{NP} \subseteq \mathcal{PCP}[\log, O(1)]$). The linearity tester is due to [4], but the main analysis presented here follows [3]. For further studies of this linearity tester see [2].

1. S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof Verification and Intractability of Approximation Problems. *JACM*, Vol. 45, pages 501–555, 1998. Preliminary version in *33rd FOCS*, 1992.

2. M. Bellare, D. Coppersmith, J. Hastad, M. Kiwi and M. Sudan. Linearity testing in characteristic two. *IEEE Transactions on Information Theory*, Vol. 42, No. 6, November 1996, pages 1781–1795.

3. M. Bellare, S. Goldwasser, C. Lund and A. Russell. Efficient probabilistically checkable proofs and applications to approximation. In *25th STOC*, pages 294–304, 1993.

4. M. Blum, M. Luby and R. Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. *JCSS*, Vol. 47, No. 3, pages 549–595, 1993. Preliminary version in *22nd STOC*, 1990.

Appendix A: Linear functions are far apart

In this section we intend to prove that linear functions are far apart. In addition to the natural interest such a result may invoke, we hope the result may shed some light as to why linear functions are good candidates for $\mathcal{PCP}$ proof systems.

When looking for a $\mathcal{PCP}$ proof system it is clear that the proofs must be robust in the sense that given a good proof π a small change effecting a constant number of bits must yield another good proof π_1, this is so since the probability of the verifier's to detect such a small change is negligible. This means that π_1 must carry "the same information" as π (at least with respect to the information the verifier can "read" from it). This formulation has a very strong scent of error correcting codes, we do know that error correcting codes have this property. In the case of a $\mathcal{PCP}$ proof system for a language in $\mathcal{NP}$ we even have a natural candidate for the information to be coded namely the witness.

One should note that our discussion is just a plausibility argument as there are major differences between a $\mathcal{PCP}$ verifier and an error correcting decoder. On one hand the $\mathcal{PCP}$ verifier does not have to decode at all. The verifier's task is to be convinced that there exists an information word with some desired properties (the witness). On the other hand the verifier is significantly weaker (computationally) then an efficient error correcting decoder, since it may only look at part of the code word (proof).

If the reader is convinced that the error correcting approach is a good one to begin with, note that in addition to being an error correcting code linear functions have another desired property : we can use self correction in a natural way since $L(x) = L(x+r) - L(r)$.

Proposition 18.5.2 *If $f, g : \{0,1\}^m \to \{0,1\}$ are both linear and $f \neq g$ then the distance of f from g is exactly $\frac{1}{2}$*

Proof: Note that $(f-g)$ is also linear, clearly $f(\underline{x}) \neq g(\underline{x})$ iff $(f-g)(\underline{x}) \neq 0$. All we have left to prove is that for every linear function $h \neq 0$ it holds that $\Pr_{\underline{x}}[h(\underline{x}) \neq 0] = \frac{1}{2}$.
Denote $h(\underline{x}) = \sum_{i=1}^{m} a_i^h x_i$. Since $h \neq 0$ there exists an a_i^h that does not equal 0. Assume the bits of $\underline{x}$ are chosen one by one (x_1 is chosen before x_2 and so on), denote by l the last i for which a_i^h is not 0 (i.e. $a_l = 1$ and for all $j > l$ $a_j = 0$). Clearly $h(\underline{x}) = (\sum_{i=1}^{l-1} a_i^h x_i) + x_l$. If $\sum_{i=1}^{l-1} a_i^h x_i = 0$ then with probability $\frac{1}{2}$ we choose $x_l = 1$ and we get $h(\underline{x}) = 1$. If on the other hand $\sum_{i=1}^{l-1} a_i^h x_i = 1$ then with probability $\frac{1}{2}$ we choose $x_l = 0$ and again we get $h(\underline{x}) = 1$. ■

Corollary 18.6 *If a function $f : \{0,1\}^m \to \{0,1\}$ is at distance less than $\frac{1}{4}$ from linear then there exists a* unique *linear function L_f s.t. f is at distance less then $\frac{1}{4}$ from L_f.*

Proof: Otherwise by triangle inequality we would get two linear functions at distance smaller than $\frac{1}{2}$ ■

Appendix B: The linearity test for functions far from linear

Recall that our objective is to bound the probability of failure of one iteration of the basic procedure T in case of functions far from linear. The result we need is that given a function $f : \{0,1\}^m \to \{0,1\}$ which is at distance at least $\frac{3}{8}$ from linear the probability that $f(x+y) \neq f(x)+f(y)$ for randomly chosen x, y is bigger than some constant c. Denote by η the probability of failing one iteration of T (i.e. $\eta \stackrel{\text{def}}{=} \Pr_{x,y}[f(x+y) \neq f(x)+f(y)]$).

Intuitively if the probability of choosing x and y s.t. $f(x)+f(y) = f(x+y)$, is very big than there must be some linear function L which agree with f on "a lot" of points so f cannot be "too far" from linear. The problem is that it is not clear how to find this L or even those points on which L and f agree.

Since a linear function behavior on the entire space $\{0,1\}^n$ is fixed by its behavior on any n independent vectors, we should somehow see how "most of the points would like f to behave". Formalizing this intuition is not as hard as it may seem. Given a point a we would like to find how "most of the points want f to behave on a", the natural way to define that, is that the point x "would like" $f(a)$ to take a value that would make $f(a) = f(x+a) - f(x)$ true.

Definition 18.7 $(L_f(\cdot))$:
For every $a \in \{0,1\}^n$ define $L_f(a)$ to be either 0 or 1 s.t. $\Pr_{x \in_R \{0,1\}^n}[L_f(a) = f(x+a) - f(x)] \geq \frac{1}{2}$ (In cases were $L_f(a)$ might be either 0 or 1 (probability $\frac{1}{2}$) define $L_f(a)$ arbitrarily.)

What we would like to see now is that L_f is indeed linear, and that L_f is reasonably close to f (depending on η the probability to fail the basic procedure T). We would indeed prove those claims but before embarking on that task we have to prove a technical result.

By definition $\Pr_{x,a}[f(x)+f(x+a) = L_f(a)] \geq 1-\eta$ therefor by an averaging argument $L_f(a)$ behaves nice "on the average a" (i.e. for most a's $\Pr_x[f(x)+f(x+a) = L_f(a)] \geq 1-2\eta$). However there might have been a few bad points b on which $\Pr_x[f(x)+f(x+b) = L_f(b)] = \frac{1}{2}$. We would like to show L_f behaves nicely on all the points.

Claim 18.5.3 *For all points $a \in \{0,1\}^n$ it holds that: $\Pr_{x \in_R \{0,1\}^n}[f(x)+f(x+a) = L_f(a)] \geq 1-2\eta$.*

Proof: Look at two points x, y chosen at random, what is the probability that "they vote the same for $L_f(a)$" i.e. $f(x)+f(x+a) = f(y)+f(y+a)$?
Clearly if we denote $p = \Pr_x[f(x)+f(x+a) = L_f(a)]$ then
$\Pr_{x,y}[f(x)+f(x+a) = f(y)+f(y+a)] = p^2 + (1-p)^2$ (x and y might both go with $L_f(a)$ or against it). From Another perspective:

$$\begin{aligned}
&\Pr_{x,y}[f(x)+f(x+a) = f(y)+f(y+a)] = \\
&= \Pr_{x,y}[f(x)+f(x+a)+f(y)+f(y+a) = 0] = \\
&= \Pr_{x,y}[f(x)+f(y+a)+f(x+y+a)+f(y)+f(x+a)+f(x+y+a) = 0] \leq \\
&\leq \Pr_{x,y}[(f(x)+f(y+a)+f(x+y+a) = 0) \wedge (f(y)+f(x+a)+f(x+y+a) = 0)] \; =
\end{aligned}$$

$= 1 - \Pr_{x,y}[(f(x)+f(y+a)+f(x+y+a) \neq 0) \vee (f(y)+f(x+a)+f(x+y+a) \neq 0)] \leq$
$\leq 1 - 2\eta$

The last inequality is true since all $x, y, a+x, a+y$ are uniformly distributed (also dependent), from the definition of η and using the union bound. We got that $p^2 + (1-p)^2 \geq 1 - 2\eta$. Simple manipulation brings us to :
$1 - 2p + 2p^2 \geq 1 - 2\eta \iff p(p-1) \geq -\eta \iff p(1-p) \leq \eta$.
Note that since $L_f(a)$ value (0 or 1) was defined to maximize $p = \Pr[f(x) + f(x+a) = L_f(a)]$, p is always bigger or equal to $\frac{1}{2}$. So $\frac{1}{2}(1-p) \leq p(1-p) \leq \eta$ and from that $p \geq 1 - 2\eta$ follows. ■

Claim 18.5.4 *If the failure probability η is smaller then $\frac{1}{6}$ then the function $L_f(\cdot)$ is linear.*

Proof: Given any $a, b \in \{0,1\}^n$ we have to prove $L_f(a+b) = L_f(a) + L_f(b)$. The strategy here is simple, we will prove (by the probabilistic method) the existence of "good intermediate" points in the sense that L_f behaves nicely on these points and this forces L_f to be linear.
Suppose there exists x, y s.t. the following events happen simultaneously :

E1 : $L_f(a+b) = f(a+b+x+y) + f(x+y)$.
E2 : $L_f(b) = f(a+b+x+y) + f(a+x+y)$.
E3 : $L_f(a) = f(a+x+y) + f(x+y)$.

Then :

$$L_f(a+b) = f(a+b+x+y) + f(x+y) = L_f(b) + f(a+x+y) + f(x+y) = L_f(b) + L_f(a)$$

.

To prove the existence of these points choose x, y at random. By Claim 7.1 the probability that each of the events E1, E2, E3 will not happen is smaller or equal 2η, so the probability that one of these will not happen is smaller or equal to 6η. Since $\eta < \frac{1}{6}$ the probability that some of those events do not happen is smaller then 1, i.e. there exists x and y for which the events E1, E2, E3 happen and therefor L_f is linear with regard to a, b. ■

Claim 18.5.5 *The function $L_f(\cdot)$ is at most at distance 3η from f.*

Proof: Basicly we show that since L_f is defined as "how f should behave if it wants to be linear" then if $f(a) \neq L_f(a)$ then "f is not linear on a" since f is close to linear it must be close to L_f.

Denote by p the probability that f agrees with L_f i.e. $\Pr_x[f(x) = L_f(x)] = p$. (Notice that the distance between f and L_f equals $1-p$ by definition).

Choose two vectors x, y and denote by E the event "$f(x+y) = L_f(x+y)$" and by F
the event "$f(x) + f(y) = f(x+y)$".
$\Pr_{x,y}[F \wedge E^c] = \Pr_{x,y}[(f(x)+f(y) = f(x+y)) \wedge (f(x+y) \neq L_f(x+y))] =$
$= \Pr_{x,y}[(f(x+y+y)+f(y) = f(x+y)) \wedge (f(x+y) \neq L_f(x+y))]$

Clearly whenever $f(x+y+y) + f(y) = f(x+y)$ and $f(x+y) \neq L_f(x+y)$ it holds that $f(x+y+y) + f(y) \neq L_f(x+y)$ therefor :
$\Pr_{x,y}[F \wedge E^c] \leq \Pr_{x,y}[f((x+y)+y) + f(y) \neq L_f(x+y)] \leq 2\eta$
Where the last inequality is by claim 7.1

Now notice $\Pr[F] = (1-\eta)$ and $\Pr[E] = p$ by definition.
Looking at $\Pr[F]$ from another perspective:

$$\Pr[F] = \Pr[F \wedge E] + \Pr[F \wedge E^c] \leq \Pr[E] + \Pr[F \wedge E^c] \leq p + 2\eta$$

Comparing the two evaluations of $\Pr[F]$ we get $1-\eta \leq p+2\eta$ i.e. $p \geq 1-3\eta$. Since the distance between f and L_f is $1-p$ our result is that f's distance from L_f is smaller or equal to 3η ■

To conclude if η, the probability to fail T, is smaller then $\frac{1}{8}$ then f is at distance at most $\frac{3}{8}$ from linear and this means :

Corollary 18.8 *If f is at distance bigger than $\frac{3}{8}$ from linear then the probability to fail one iteration of the basic procedure T is bigger or equal to $\frac{1}{8}$.*

Oded's Note: *The above analysis is not the best possible. One may show that if f is at distance bigger than $1/4$ from linear then T rejects with probability at least $2/9$.*

Lecture 19

Dtime vs Dspace

Notes taken by Tamar Seeman and Reuben Sumner

Summary: In this lecture, we prove that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$. That is, we show how to simulate any given deterministic multi-tape Turing Machine (TM) of time complexity t, using a deterministic TM of space complexity $t/\log t$. A main ingrediant in the simulation is the analysis of a pebble game on directed bounded-degree graphs.

19.1 Introduction

We begin by defining Dtime(t) and Dspace(t).

Definition 19.1 (Dtime): *$Dtime(t(\cdot))$ is the set of languages L for which there exists a multi-tape deterministic TM M which decides whether or not $x \in L$ in less than $t(|x|)$ Turing machine steps.*

Definition 19.2 (Dspace): *$Dspace(s(\cdot))$ is the set of languages L for which there exists a multi-tape deterministic TM which decides whether or not $x \in L$ while never going to the right of cell $s(|x|)$ on any of its tapes.*

Since any Turing machine step can move the head(s) by at most one position, it is immediately clear that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot))$. Furthermore NP is easily in $Dspace(p(\cdot))$ for some polynomial p, but is not believed to be in $Dtime(q(\cdot))$ for any polynomial q. Thus it seems that space is much more powerful than time. In this lecture we will further refine this intuition by proving that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$. It follows that $Dtime(t(\cdot))$ is a strict subset of $Dspace(t(\cdot))$, since it has already been shown that $Dspace(o(t))$ is a strict subset of $Dspace(t)$.

Note: The multi-tape TM consists of one read-only, bi-directional input tape, one optional write-only output tape (in the case of a machine to decide a language we will not include such a tape and determine acceptance based on the exact state that the Turing machine terminates in) together with a fixed number of bi-directional read/write work tapes. The number of work tapes is irrelevant for $Dspace()$, though, since a TM M with k work tapes can be simulated by a TM M' with just a single work tape and the same amount of space. This is done by transforming k work tapes into one work tape with k tracks. This transformation then simulates the work of the original Turing machine by using polynomially more time, but the same amount of space. However, in the case of $Dtime()$ the number of tapes *does* matter.

19.2 Main Result

Theorem 19.3 *If $t(\cdot)$ is constructible in space $t(\cdot)/\log t(\cdot)$ and $t(\cdot)$ is at least linear, then $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$.*

In order to make the proof of this theorem as readable as possible we state some results (without proof) in place they are needed so that their motivation is clear, but we prove them only later so as not to disturb the flow of the proof.

Proof: Let L be a language accepted by a TM M in $Dtime(t(\cdot))$. Let x be an input and $t = t(|x|)$. Divide each tape into $t^{1/3}$ blocks of $t^{2/3}$ cells. (This ensures that $(\#\text{ blocks})^2 \ll t$, a necessary feature.) Similarly, partition time into periods of $t^{2/3}$ steps. During a particular period, M visits at most two blocks of space on each tape, since the number of steps M can take does not exceed the size of a single block.

Lemma 19.2.1 (Canonical Computation Lemma): *Without loss of generality such a machine moves between block boundaries only on the very last move of a time period. This holds with at most a linear increase in the amount of time used, and a constant number of additional tapes.*

The proof is postponed to the next section. Therefore without loss of generality we assume that in each time period our machine stays within the same block on each work tape.

Our goal now is to compute the final state of the machine, which will indicate whether or not our input is in the language. We therefore have to somehow construct the blocks that we are in at the final time period while never storing the complete state of the work tapes of the machine since these would potentially exceed our space bound. To do so, we establish a dependency graph between different states and find a sufficiently efficient method of evaluating the graph node corresponding to the final machine state.

We introduce some notation to describe the computation of the machine.

$h_i(j)$ is the block location of the i^{th} tape head during the j^{th} period. $h_i(j) \in \left\{1, \ldots, t^{1/3}\right\}$.

$h(j)$ is the vector $h_1(j), h_2(j), \ldots, h_k(j)$ for a k-tape machine.

$c_i(j)$ is the content of block $h_i(j)$ on the i^{th} tape, together with the Turing machine state at the end of the period and head position on tape i. $c_i(j) \in \{0,1\}^{t^{2/3}} \times O(1) \times \{0,1\}^{O(\log t)}$. [1]

$c(j)$ is the vector $c_1(j), c_2(j), \ldots, c_k(j)$.

$l_i(j)$ is the last period where we visited block $h_i(j)$ on tape i. This is $\max\{j' < j$ such that $h_i(j') = h_i(j)\}$.

In order to compute $c(j)$ we will need to know:

- $c(j-1)$ to determine what state to start our computation in.
- $c_1(l_1(j)), \ldots, c_k(l_k(j))$ so that we know the starting content of the blocks we are working with.

[1] We assume that $\Sigma = \{0,1\}$, where Σ is the tape alphabet

Figure 19.1: An interesting pebbling problem

It is immediately clear then that we need at most $k+1$ blocks from past stages of computation to compute $c(j)$.

Define a directed graph $G = (V, E)$ where $V = \{1, \ldots, t^{1/3}\}$ and $E = \{(j-1, j)|j > 1\} \cup \{(l_i(j), j)\}$. So vertex i represents knowledge of $c(i)$. There is an edge $(j-1) \rightarrow j$ since $c(j)$ depends on $c(j-1)$. Similarly there is an edge $l_i(j) \rightarrow j$ since $c(j)$ depends on $c_i(l_i(j))$ for all i. Hence this graph represents exactly the dependency relationship that we have just described.

Consider the example in Figure 19.1. The most obvious way to reach node 6 would be

- calculate state 1 from the starting state
- calculate state 2 from state 1 and erase state 1
- calculate state 3 from state 2 and keep state 2
- calculate state 4 from state 3 and keep state 3 as well as still keeping state 2
- calculate state 5 from states 3 and 4 and erase states 3 and 4
- calculate state 6 from states 2 and 5 and erase states 2 and 5

Notice that when calculating state 5 we had in memory states 2,3 and 4, a total of three prior states. We can do better.

- calculate state 1 from the starting state
- calculate state 2 from state 1 and erase state 1
- calculate state 3 from state 2 and *erase* state 2
- calculate state 4 from state 3 and keep state 3
- calculate state 5 from states 3 and 4 and then erase both states 3 and 4
- calculate state 1 from the starting state (for the second time!)
- calculate state 2 from state 1 and erase state 1
- calculate state 6 from states 2 and 5 and then erase both

This second calculation required two more steps to compute but now we never needed to remember more than two previous states, rather than three. It is exactly this tradeoff of time versus space which enables us to achieve our goal.

In general on a directed acyclic graph of bounded degree we can play a pebble game. The rules of the game are simple:

1. A pebble may be placed on any node all of whose parents have pebbles.
2. A pebble may be removed from any node.
3. A pebble may be placed on any node having no parents. This is a special case of the first rule.

The goal of the pebble game is to pebble a given target node while minimizing the number of pebbles used at any one time.

This game corresponds to calculations of our Turing machine. A pebble on vertex j in the game corresponds to a saved value for $c(j)$. When the game tells us to pebble node j then we can recover the contents of $c(j)$ as follows:

1. Load contents of $c_1(l_1(j)), \ldots, c_k(l_k(j))$ from storage. Since there was an edge from $l_i(j) \to j$ in the graph, Rule 1 guarantees that node $l_i(j)$ has a pebble on it. Since $l_i(j)$ has a pebble in the pebble game, then in our computation we have $c(l_i(j)$ and therefore $c_i(l_i(j))$ in storage.
2. Load the starting state and head position for period j from $c(j-1)$, again guaranteed to be available by the pebble game.
3. Based on all the above, it is easy to reconstruct $c(j)$.

Note that in order to determine the answer of the original Turing machine, it suffices to reconstruct $c(t^{1/3})$. Our aim is to do so using $O(t/\log t)$ space, which can be reduced to pebbling the target node $t^{1/3}$ using $t^{1/3}/\log t$ pebbles. We first state the following general result regarding pebbling:

Theorem 19.4 (Pebble Game Theorem): *For any fixed d, any directed acyclic graph $G = (V, E)$ with in-degree bound d, and any $v \in V$, we can pebble v while never using more than $O(|V|/\log|V|)$ pebbles simultaneously.*

The proof is postponed to the next section. Notice, however, that the theorem does not state anything about how efficient it is to compute this pebbling.

Using the above result, we simulate the machine M on input x as follows.

1. Compute $t = t(|x|)$
2. Loop over all possible guesses of $h(1), \ldots, h(t^{1/3})$. For each one do:
   (a) Compute and store the dependency graph.
   (b) Execute a pebbling strategy to reach node $t^{1/3}$ on the graph as follows:
      i. Determine next pebble move
      ii. if the move is "Remove i" then erase $c(i)$
      iii. if the move is "Pebble i" then set the machine to the initial state for period i by loading the contents of the work tape from storage. Calculate $c(i)$. Verify that in step 2 we did guess $h(i+1)$ correctly. If not, then abort the whole calculation so far and return to the next guess in step 2. Otherwise save $c(i)$ for future use.
   (c) Having just executed "Pebble $t^{1/3}$", terminate and accept or reject according to the original Turing machine.

We need to show that all of the above computation can be performed within our space bound.

- Step 1 can be performed within our space bound by our hypothesis regarding the (space constructability of the) function $t(\cdot)$.
- In step 2a we store a graph $G = (V, E)$ where $|V| = t^{1/3}$ and $|E| \leq |V| \cdot (k+1)$. A simple list representation of the graph will therefore take $(k+1) \cdot t^{1/3} \cdot \log_2(t^{1/3}) \ll t/\log t$ space.

	$Rev(b_i(j-2))$	$Rev(b_i(j-1))$	$Rev(b_i(j))$	
$\ldots$	$b_i(j-1)$	$b_i(j)$	$b_i(j+1)$	$\ldots$
	$Rev(b_i(j))$	$Rev(b_i(j+1))$	$Rev(b_i(j+2))$	

Figure 19.2: The new work tape

- Step 2(b)ii actually frees space rather than using space.
- Step 2(b)iii needs space only for k blocks copied from storage, and space to store the result. Since our pebble game guarantees that we will never need more than $t^{1/3}/\log t^{1/3}$ pebbles, we will never need more than $(t^{1/3}/\log t^{1/3}) \cdot (k+1) \cdot t^{2/3} = O(t/\log t)$ space, our space bound.

So aside from step 2(b)i all of step 2b can be calculated within our space bound.

We need to describe now how to perform step 2(b)i. Consider a non-deterministic algorithm. We set a counter to be the maximum number of possible pebble moves to reach our target. Since it never makes sense to return to an earlier pebbling configuration of the graph, we can bound the number of steps by $2^{t^{1/3}}$. Such a counter will only require $t^{1/3}$ space, which is within our bound. We now non-deterministically make our choice of valid pebbling move and decrement the counter. We accept if we pebble the target vertex, reject if the counter reaches its limit first. The dominant space required by the routine is therefore the space required to store a working copy of the graph, $O(t^{1/3} \cdot \log t)$.

In Lecture 5 we proved that $Nspace(s) \subseteq Dspace(s^2)$. Therefore, the above non-deterministic subprogram which runs in $Nspace(t^{1/3} \cdot \log t)$ can be converted to a deterministic subprogram running in $Dspace(t^{2/3} \cdot \log^2 t \ll t/\log t)$. ■

19.3 Additional Proofs

We now prove two results stated without proof in the previous section.

19.3.1 Proof of Lemma 19.2.1 (Canonical Computation Lemma)

Our new machine M' which simulates the operation of M works as follows. Firstly each work tape is replaced by a three-track work tape. Three additional work tapes are added:

- one uses a unary alphabet and is used as a counter
- the second is a k-track "copy" tape
- the last is a k-track binary "position" tape

After calculating the block bound $t^{2/3}$, store it in unary on the counter tape. Now mark the work tapes into blocks of length $t^{2/3}$ by putting a special end-of-block marker where needed (each block will thus be one cell larger to accommodate the end of cell marker). Let $b_i(j)$ and $b'_i(j)$ denote the contents of the j^{th} block of tape i in the original machine and new machine respectively. Then $b'_i(j)$ is the tuple $(Reverse(b_i(j-1)), b_i(j), Reverse(b_i(j+1)))$. Here $Reverse$ means reversing the order of the tape cells. When simulating the computation we start by working on the middle track. If we see the end-of-block marker when going left, then we start using the first track instead and reverse the directions of each head move (on this track). Similarly, if we go off the end of the block

while going right we switch to using the last track and again reverse the direction of head moves. Throughout we keep moving the head on the counter tape until it indicates that we have reached the end of the period.

At the end of the period we have to do some housekeeping. First we save the state, either within the state itself or on a special tape somewhere; either way this is only a constant amount of information. We store the head position within each block on the "position" tape as follows. Scan left along all work tapes not moving past the beginning of the block. For all work tapes not at the beginning of the block write a 1 in the corresponding track of the "position" tape, and for all others write a 0. Continue in this way until every head position is at the start of block for all the work tapes; the number of 1's on track i of the "position" tape will thus equal the head position, within the block, on work tape i. This takes one additional time period.

Consider Figure 19.2. At this point only the center block is up to date; the values for $b_i(j-1)$ and $b_i(j)$ in the left block are 'stale' since we may have modified them in our last time period. Similarly the values for $b_i(j)$ and $b_i(j+1)$ in the right block are also 'stale'. We update these stale values by scanning back and forth over all three cells and using the "copy" tape as a temporary area to store the contents of one block (all three tracks) at a time. Altogether this process requires about 6 time periods.

Finally we return the heads to their correct positions, but this time if they were working on the first track of the working block we place them in the corresponding cell in the previous block, and if they were in the last track then we place them in the next block. This may require an additional time period or two. Altogether this simulation of a single time period has cost us a little bit of extra space (more work tapes) and a constant factor increase of about 14 in the number of periods. ■

19.3.2 Proof of Theorem 19.4 (Pebble Game Theorem)

Denote by $R_d(p)$ the minimum number of edges in a di-graph of in-degree bound d which requires p pebbles (that is, at least one node in the graph needs p pebbles). We will first show that $R_d(p) = \Omega(p \log p)$ and then that this fact implies our theorem.

Consider a graph $G = (V, E)$ with the minimal possible number of edges requiring p pebbles. Let V_1 be the set of vertices which can be pebbled using at most $p/2$ pebbles. Let G_1 be the subgraph induced by V_1 having edge set $E_1 = E \cap (V_1 \times V_1)$. Let $V_2 = V - V_1$ and let G_2 be the subgraph induced by V_2 and having edge set $E_2 = E \cap (V_2 \times V_2)$. Let $A = E - E_1 - E_2$ be all edges between the two subgraphs. Notice that there cannot be any edge from V_2 to V_1 since then the vertex at the head of the edge (in V_1) would require at least as many pebbles as the vertex at its tail, which is more than $p/2$.

There exists $v \in V_2$ which requires $p/2 - d$ pebbles in G_2

Assume not. Then we show that we can pebble any node in G with fewer than p pebbles, contradicting the hypothesis regarding G. For any node $v \in V_2$ we have assumed that in G_2 we can pebble it with $< p/2 - d$ pebbles. Invoke the strategy to pebble v in the graph G_2 on the original graph G. Whenever we want to pebble a vertex $u \in G_2$ that has a parent in G_1, bring a pebble to the parent in G_1, using $\leq p/2$ pebbles. Since some u might have as many as d parents in G_1, we actually use as many as $d - 1 + p/2$ pebbles when pebbling the d^{th} parent in G_1 (using $d-1$ pebbles to cover the first $d-1$ parents and $p/2$ to do the actual pebbling). Once all the parents in G_1 are pebbled, pebble u and then lift the pebbles on all the vertices in G_1. Thus we can pebble

our target v using at most $(p/2-d)+(p/2+d-1)=p-1$ pebbles. Since we can also pebble any $v \in V_1$ with at most $p/2 < p$ we can pebble any $v \in V$ with fewer than p pebbles, a contradiction to our original choice of G.

If v with in-degree k requires m pebbles then it has a parent needing $m-k$

We show that in general, for any $G=(V,E)$ where G is an acyclic di-graph, any node $v \in V$ requiring m pebbles, with an in-degree of k, has a parent u requiring at least $m-k+1$ pebbles. Suppose to the contrary that each parent needs $m-k$ or fewer pebbles. Then simply bring a pebble to each of them in arbitrary order, each time removing all pebbles not on parents of v. When bringing a pebble to the i^{th} parent we have $i-1$ pebbles covering the other parents and use at most $m-k$ pebbles for the pebbling itself. Thus at any time we use at most $m-k+i-1$ pebbles. Over all parents then, the maximum number of pebbles that we use is $m-k+k-1=m-1$ which may occur only when pebbling the k^{th} parent. Now, however, we have k pebbles on the parents of v and we can pebble v itself having used at most $m-1$ pebbles, a contradiction.

There exists $u \in G_1$ which requires $p/2-d$ pebbles (in both G and G_1)

Consider any $v \in G_2$. Repeatedly replace v by a parent in G_2 until you can go no further. Since G is acyclic this is guaranteed to stop. When it stops, since the new v requires, by virtue of being in G_2, more than $p/2 > 0$ pebbles, v has at least one parent in G_1 (and no parents in G_2). Let $m > p/2$ be the number of pebbles needed to pebble v and let k be the in-degree of v in the original graph G. By the above claim v has a parent which requires $m-k+1 > p/2-k+1$ pebbles. Futhermore, since $k \leq d$ we see that v has a parent requiring at least $p/2-d+1$ pebbles. Therefore (as v has no parents in G_2) there is a vertex $u \in G_1$ requiring at least $p/2-d+1 > p/2-d$ pebbles.

$|E_2|+|A| \geq R_d(p/2-d)+\frac{p}{4d}$

Since G_2 requires at least $p/2-d$ pebbles, $|E_2| \geq R_d(p/2-d)$. If $|A| \geq \frac{p}{4} \geq \frac{p}{4d}$ then we are done.

Otherwise $|A| < \frac{p}{4}$. We will ignore $|A|$ and show that $|E_2| \geq R_d(p/2-d)+\frac{p}{4d}$. Pebble each of the $< p/4$ vertices in V_1 with children in V_2 in succession. Independently pebbling each would require at most $p/2$ pebbles. By pebbling one at a time we can pebble them all using at most $p/4-1+p/2=3p/4-1$ pebbles. When done we are left with less than $p/4$ pebbles on the graph, leaving more than $3p/4$ pebbles free. Since we know that there must exist a $v \in G_2$ requiring p pebbles in G then it must require at least $3p/4$ pebbles in G_2. Consider a vertex v with out-degree of 0 which requires $3p/4$ pebbles in G_2, and remove it. As proven in section 19.3.2 the resulting graph must still require at least $3p/4-d$ pebbles. Repeating this process i times, we have a graph requiring $3p/4-i \cdot d$ pebbles with at least i fewer edges. So for $i=\frac{p}{4d}$ times, we have a graph requiring at least $3p/4-p/4=p/2$ pebbles with $\frac{p}{4d}$ fewer edges. This subgraph will have at least $R_d(p/2) \geq R_d(p/2-d)$ edges. Together with the $\frac{p}{4d}$ that we removed we see that $|E_2|+|A| \geq |E_2| \geq R_d(p/2-d)+\frac{p}{4d}$ as required.

Putting it together

So $|E|=|E_1|+|E_2|+|A|$. By Section 19.3.2 we have $|E_1| \geq R_d(p/2-d)$. By Section 19.3.2 we have $|E_2|+|A| \geq R_d(p/2-d)+\frac{p}{4d}$. Therefore $R_d(p)=|E| \geq 2R_d(p/2-d)+\frac{p}{4d}$, where the equality is due to the hypothesis that G has minimum number of edges among grapohs requiring p pebbles.

To solve the recurrence notice that

$$R_d(p) \geq 2R_d(\frac{p}{2} - d) + \frac{p}{4d} \geq 2R_d(\frac{p}{2} - 2d) + \frac{p}{4d}$$

For any i we get

$$R_d(\frac{p}{2^i} - 2d) \geq 2R_d(\frac{\frac{p}{2^i} - 2d}{2} - d) + \frac{\frac{p}{2^i} - 2d}{4d} = 2R_d(\frac{p}{2^{i+1}} - 2d) + \frac{\frac{p}{2^i} - 2d}{4d}$$

So

$$R_d(p - 2d) \geq 2^i R_d(\frac{p}{2^i} - 2d) + \sum_{j=0}^{i-1} 2^j \frac{\left(\frac{p}{2^j} - 2d\right)}{4d} = 2^i R_d(\frac{p}{2^i} - 2d) + \sum_{j=0}^{i-1} \frac{p - 2^{j+1}d}{4d}$$

Setting $i = \log_2(p/2d)$ we get

$$\begin{aligned} R_d(p) \geq R_d(p-2d) &\geq 2^i R_d(0) + \sum_{j=0}^{\log_2(p/2d)-1} \frac{p - 2^{j+1}d}{4d} \\ &\geq \sum_{j=0}^{\log_2(p/2d)-1} \frac{p - 2^{\log_2(p/2d)}d}{4d} \\ &= \log_2(p/2d)\frac{p - \frac{p}{2d}d}{4d} \\ &= \log_2(p/2d)\frac{p/2}{4d} \\ &= \Omega(p \log p) \end{aligned}$$

So, for some constant $c > 0$, $R_d(p) \geq c \cdot p \log p$.

Now consider our original question of how many pebbles one needs to pebble a graph of n vertices. With $p = kn/\log n$ pebbles, we can certainly pebble all graphs with less than $c \cdot p \log p$ edges. Now,

$$\begin{aligned} cp \log p &= c\frac{kn}{\log n} \log \frac{kn}{\log n} \\ &= c\frac{kn}{\log n}(\log k + \log n - \log\log n) \\ &> ckn\left(1 - \frac{\log\log n}{\log n}\right) \\ &> ckn/2 \end{aligned}$$

for all sufficiently large n such that $\frac{\log\log n}{\log n} < 1/2$. Letting $k = \frac{2d}{c}$, we can pebble all graphs with less than $\frac{cn}{2} \cdot \frac{2d}{c} = dn$ edges, using $p = kn/logn$ pebbles. Since any graph on n vertices, with in-degree bound of d, has less than dn edges, it may be pebbled by $O(dn/\log n)$ pebbles. Since d is a constant, the theorem follows. ■

Bibliographic Notes

This lecture is based on [1].

1. J.E. Hopcroft, W. Paul, and L. Valiant. On time versus space. *J. of ACM*, Vol. 24, No. 2, pages 332–337, 1977.

Lecture 20

Circuit Depth and Space Complexity

Notes taken by Vered Rosen and Alon Rosen

Summary: In this lecture we study some of the relations between Boolean circuits and Turing machines. We define the complexity classes NC and AC, compare their computational power, and point out the possible connection between $uniform\text{-}NC$ and "efficient" parallel computation. We conclude the discussion by establishing a strong connection between space complexity and depth of circuits with bounded fan-in.

20.1 Boolean Circuits

Loosely speaking, a *Boolean Circuit* is a directed acyclic graph with three types of labeled vertices: *inputs*, *outputs*, and *gates*. The inputs are sources in the graph (i.e. vertices with in-degree 0), and are labeled with either Boolean variables or constant Boolean values. The outputs are sinks in the graph (i.e. vertices with out-degree 0). The gates are vertices with in-degree $k > 0$, which are labeled with Boolean functions on k inputs. We refer to the in-degree of a vertex as its *fan-in* and its out-degree as its *fan-out*. A general definition of Boolean circuits would allow the labeling of gates with arbitrary Boolean functions. We restrict our attention to circuits whose gates are labeled with the boolean functions AND, OR, and NOT (denoted $\bigwedge, \bigvee, \neg$ respectively).

20.1.1 The Definition

Definition 20.1 (Boolean Circuit): *A* **Boolean Circuit** *is a directed acyclic graph with labeled vertices:*

- *The* input *vertices, labeled with a variable x_i or with a constant (0 or 1), and have fan-in 0.*
- *The* gate *vertices, have fan-in $k > 0$ and are labeled with one of the boolean functions $\bigwedge, \bigvee, \neg$ on k inputs (in the case that the label is $\neg$, the fan-in k is restricted to be 1).*
- *The* output *vertices, labeled 'output', and have fan-out 0.*

Given an assignment $\sigma \in \{0,1\}^m$ to the variables $x_1, ..., x_m$, $C(\sigma)$ will denote the value of the circuit's output. The value is defined in the natural manner, by setting the value of each vertex according to the boolean operation it is labeled with. For example, if a vertex is labeled $\bigwedge$ and the vertices with a directed edge to it have values a and b, then the vertex has value $a \bigwedge b$.

We denote by $size(C)$ the number of gates in a circuit C, and by $depth(C)$ the maximum distance from an input to an output (i.e. the longest directed path in the graph).

A circuit is said to have *bounded* fan-in if there exists an a-priori upper bound on the fan-in of its AND and OR gates (NOT gates, must have fan-in 1 anyway). If there is no a-priori bound on the fan-in (other than the size of the circuit), the circuit is said to have *unbounded* fan-in.

20.1.2 Some Observations

- We have already seen how to construct a circuit which simulates the run of a Turing machine M on some input $x \in \{0,1\}^n$ (see the proof of Cook's Theorem , Lecture 2). Using this construction the resulting circuit will be of size quadratic in $T_M(n)$ (the running time of M on input length n), and of depth bounded by $T_M(n)$.

- Circuits may be organized into disjoint layers of gates, where each layer consists of gates having equal distance from the input vertices. Once presented this way, a circuit may be thought of as capturing a certain notion of parallel computation. We could associate each path starting from an input vertex with a computation performed by a single processor. Note that all such computations can be performed concurrently. Viewed in this way, the circuit's depth corresponds to parallel time, whereas the size corresponds to the total amount of parallel work.

- Any bounded fan-in circuit can be transformed into a circuit whose gates have fan-in 2 while paying only a constant factor in its depth and size. A gate with constant fan-in c, can be converted into a binary tree of gates of the same type which computes the same function as the original gate. Since c is a constant so will be the tree's size and depth. We can therefore assume without loss of generality that all gates in bounded fan-in circuits have fan-in 2. Note that the same transformation in the case of unbounded fan-in will increase the depth by a factor logarithmic in the size of the circuit.

- Any circuit can be modified in such a way that all negations (i.e. $\neg$ gates) in it appear only in the input layer. Using De-Morgan's laws (that is, $\neg \bigvee x_i = \bigwedge \neg x_i$), each $\bigvee$ gate followed by a negation can be transformed into an $\bigwedge$ gate whose inputs are negations of the original inputs (the same argument applies symmetrically for $\bigwedge$ gates). This way, we can propagate all negations in the circuit towards the input layer without changing the value of the circuit's output, and without changing its depth. Thus, without loss of generality, all "internal" gates in a Boolean circuit are labeled with $\bigwedge$ or $\bigvee$. When measuring the depth of the circuit, negations will be counted as a single layer.

- In an unbounded fan-in circuit, two consecutive $\bigwedge$ (resp. $\bigvee$) gates having identical labels can be merged into a single gate with the same label without changing the value of the circuit's output. We can therefore assume that unbounded fan-in circuits are of the special form where all $\bigwedge$ and $\bigvee$ gates are organized into alternating layers with edges only between adjacent layers.

  Note, however, that the above argument does not necessarily work for bounded fan-in circuits since the merging operation might cause a blow-up in the fan-in of the resulting gates which will exceed the specified bound.

20.1.3 Families of Circuits

Eventhough a circuit may have arbitrarily many output vertices we will focus on circuits which have only one output vertex (unless otherwise specified). Such circuits can be used in a natural way

to define a language (subset of $\{0,1\}^*$). Since we are interested in deciding instances of arbitrary size, we consider *families* of Boolean circuits with one different circuit for each input size.

Definition 20.2 (Family of Circuits): *A language $L \subseteq \{0,1\}^*$ is said to be decided by a* family of circuits, *$\{C_n\}$, when C_n takes n variables as inputs, if and only if for every n, $C_n(x) = \chi_L(x)$ for all $x \in \{0,1\}^n$.*

Given a family of circuits, we might be interested in measuring the growth-rate of the size and depth of its circuits (as a function of n). This may be useful when trying to connect circuit complexity with some other abstract model of computation.

Definition 20.3 (Depth and Size of Family) *Let D and S be sets of integer functions ($N \to N$), we say that a family of circuits $\{C_n\}$, has depth D and size S if for all n, $depth(C_n) \leq d(n)$ and $size(C_n) \leq s(n)$ for some $d(\cdot) \in D$ and $s(\cdot) \in S$.*

If we wish to correlate the size and depth of a family $\{C_n\}$ that decides a language L with its Turing machine complexity, it is necessary to introduce some notion of *uniformity*. Otherwise, we could construct a family of circuits which decides a non-recursive language (see Lecture 8). The notion of uniformity which we will make use of is *logspace uniformity*. Informally, we require that a description of a circuit can be obtained by invoking a Turing machine using space which is logarithmic in the length of its output (i.e. the circuit's description). A description of a circuit (denoted $desc(C_n)$), is a list of its gates, where for each gate we specify its type and its list of predecessors. Note that the length of $desc(C_n)$ is at most quadratic in $size(C_n)$.

Definition 20.4 (logspace uniformity): *A family of circuits, $\{C_n\}$, is called* logspace uniform *if there exists a deterministic Turing machine, M, such that for every n, $M(1^n) = desc(C_n)$ while using space which is logarithmic in the length of $desc(C_n)$.*

The reason we require M to work in space which is logarithmic in its *output* length (rather than its input length) lies in the fact that $size(C_n)$ (and therefore the length of $desc(C_n)$) might be super-polynomial in n. The problem is that the description of a circuit with super-polynomial size, *cannot* be produced by Turing machines working with space which is logarithmic in n (the input size), and therefore such a circuit (one with super-polynomial size) would have been overlooked by a definition using the input length as parameter. Note that if we restrict ourselves to circuits with polynomial size, then the above remark does not apply, and it is sufficient to require that M is a deterministic logspace Turing machine.

Based on the above definitions, the class $\mathcal{P}/poly$ is the class of all languages for which there exists a family of circuits having polynomial depth and polynomial size (see Lecture 8). We have already seen how to use the transformation from Turing machines into circuits in order to prove that $uniform\text{-}\mathcal{P}/poly$ contains (and in fact equals) $\mathcal{P}$. We note that the above transformation can be performed in logarithmic space as well. This means that $logspace\text{-}uniform\text{-}\mathcal{P}/poly$ also equals $\mathcal{P}$. In the sequel, we choose logspace uniformity to be our preferred notion of uniformity.

20.2 Small-depth circuits

In this section we consider polynomial size circuits whose depth is considerably smaller than n, the number of inputs. By depth considerably smaller than n we refer to *poly-logarithmic* depth, that is, bounded by $O(\log^k n)$ for some $k \geq 0$. We are interested in separating the cases of unbounded and bounded fan-in, specifically, we will investigate the relation between two main complexity classes. As we will see, these classes will eventually turn out to be subsets of $\mathcal{P}$ which capture the notion of what is "efficiently" computable by parallel algorithms.

20.2.1 The Classes NC and AC

The complexity class NC is defined as the class of languages that can be decided by families of bounded fan-in circuits with polynomial size, and poly-logarithmic depth (in the number of inputs to the circuits). The actual definition of NC introduces an hierarchy of classes decided by circuit families with increasing depth.

Definition 20.5 (NC): *For $k \geq 0$, define NC^k to be the class of languages that can be decided by families of circuits with* bounded *fan-in, polynomial size, and depth $O(\log^k n)$. Define NC to be the union of all NC^k's (i.e. $NC \stackrel{\text{def}}{=} \bigcup_k NC^k$).*

A natural question would be to ask what happens to the computational power of the above circuits if we remove the bound on the fan-in. For instance, it is easy to see that the decision problem "is the input string $\sigma \in \{0,1\}^n$ made up of all 1's?" (the AND function) can be solved by a depth-1 unbounded fan-in circuit, whereas in a bounded fan-in circuit this problem would require depth at least $\Omega(\log n)$, just so all the input bits could *effect* the output gate. The classes based on the unbounded fan-in model (namely AC) are defined analogously to the classes in the NC hierarchy.

Definition 20.6 (AC): *For $k \geq 0$, define AC^k to be the class of languages that can be decided by families of circuits with* unbounded *fan-in, polynomial size, and depth $O(\log^k n)$. Define AC to be the union of all AC^k's (i.e. $AC \stackrel{\text{def}}{=} \bigcup_k AC^k$).*

As we will see in the sequel, AC equals NC. Note however, that such a result does not necessarily rule out a separation between the computational power of circuits with bounded and respectively unbounded fan-in. We have already seen that the class NC^0 of languages decided by constant depth circuits with bounded fan-in, is strictly contained in AC^0, its unbounded fan-in version. We are therefore motivated to look deeper into the NC^k and AC^k hierarchies and try to gain a better understanding of the relation between their corresponding levels.

Theorem 20.7 *For all $k \geq 0$,*

$$NC^k \subseteq AC^k \subseteq NC^{k+1}$$

Proof: The first inclusion is trivial, we turn directly to prove the second inclusion. Since any gate in an unbounded fan-in circuit of size $\text{poly}(n)$ can have fan-in at most $\text{poly}(n)$, each such gate can be converted into a tree of gates of the same type with fan-in 2, such that the output gate of the tree computes the same function as the original gate (since this transformation can be performed in logarithmic space, the claim will hold both for the uniform and the nonuniform settings). The resulting depth of the tree will be $\log(\text{poly}(n)) = O(\log n)$. By applying this transformation to each gate in an unbounded fan-in circuit of depth $O(\log^k n)$ we obtain a bounded fan-in circuit deciding the same language as the original one. The above transformation will cost us only a logarithmic factor in the depth and a polynomial factor in the size (i.e. the depth of the resulting circuit will be $O(\log^{k+1} n)$, and the size will remain polynomial in n). Thus, any language in AC^k is also in NC^{k+1}. ■

Corollary 20.8 $AC = NC$

In light of Theorem 20.7, it might be interesting to ask how far does the NC (resp. AC) hierarchy extend. Is it infinite, or does it collapse to some level? Even if a collapse seems unlikely, at present no argument is known to rule out this option. One thing which **is** known at present, is that AC^0 is strictly contained in NC^1.

Theorem 20.9 *$AC^0 \subset NC^1$ (strictly contained).*

Note that *uniform-NC* is contained in $\mathcal{P}$. Theorem 20.9 implies that *uniform-AC^0* is *strictly* contained in $\mathcal{P}$. An interesting open question is to establish what is the exact relationship between *uniform-NC* and $\mathcal{P}$, it is not currently known whether both classes are equal or not (analogously to the $\mathcal{P}$ vs. $\mathcal{NP}$ problem). As a matter of fact, it is not even known how to separate *uniform-NC^1* from $\mathcal{NP}$.

20.2.2 Sketch of the proof of $AC^0 \subset NC^1$

We prove the Theorem by showing that the decision problem "does the input string have an even number of 1's?" can be solved in NC^1 but **cannot** be solved in AC^0. In this context it will be more convenient to view circuits as computing functions rather than deciding languages.

Definition 20.10 (Parity): *Let $x \in \{0,1\}^n$, the function Parity is defined as:*

$$Parity(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \sum_{i=1}^{n} x_i \ (mod\ 2)$$

Claim 20.2.1 *$Parity \in NC_1$.*

Proof: We construct a circuit computing *Parity*, using a binary tree of logarithmic depth where each gate is a *xor* gate. Each xor gate can then be replaced by the combination of 3 legal gates: $a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$. This transformation increases the depth of the circuit by a factor of 2, and the size of the circuit by a factor of 3. Consequently, *Parity* is computed by circuits of logarithmic depth and polynomial size, and is thus in NC^1. ■

Claim 20.2.2 *$Parity \notin AC_0$.*

In order to prove claim 20.2.2 we show that every constant depth circuit computing *Parity* must have sub-exponential size (and therefore *Parity* cannot belong to AC^0), more precisely:

Theorem 20.11 *For every constant d, a circuit computing Parity on n inputs with depth d, must have size $exp(\Omega(n^{\frac{1}{d-1}}))$.*

Proof: (sketch): The Theorem is proven by induction on d and proceeds as follows:

1. Prove that parity circuits of depth 2 must be of large size.
2. Prove that depth d parity circuits of small size can be converted to depth $d-1$ parity circuits of small size (thus contradicting the induction hypothesis).

The first step is relatively easy, whereas the second step (which is the heart of the Theorem) is by far more complicated. We will therefore give only the outline of it.

Base case ($d = 2$): Without loss of generality, we assume that the circuit given to us is an OR of ANDs (in case it is an AND of ORs the following arguments apply symmetrically). Then, any AND gate evaluating to "1" determines the value of the circuit. Consider now an AND gate (any gate in the intermediate layer). Note that all the variables $x_1, \ldots, x_n$ must be connected to that gate. Otherwise, assume there exists an i such that there is no edge going from x_i (and from $\neg x_i$) into the gate: Then, take an assignment to the variables going into the gate, causing it to evaluate to "1" (we assume the gate is not degenerate). Under this assignment, the circuit will output "1", regardless of the value of x_i, which is impossible.
Due to that, we can associate each AND gate with a certain assignment to the n variables (determined by the literals connected to that gate). We can say that this gate will evaluate to "1" iff the variables will have this assignment.
This argument shows, that there must be at least 2^{n-1} AND gates. Otherwise, there exists an assignment σ to $x_1, \ldots, x_n$, such that $Parity(\sigma) = 1$, of which there is no AND gate identified with σ. This means that the circuit evaluated on the assignment σ will output "0", in contradiction.

The induction step: The basic idea of this step lies in a lemma proven by Hastad. The lemma states that given a depth 2 circuit, say an AND of ORs, then if one gives random values to a randomly selected subset of the variables, then it is possible to write the resulting induced circuit as an OR of relatively few ANDs with very high probability. We now outline how use this lemma in order to carry on the induction step:

Given a circuit of depth d computing parity, we assign random values to some of its inputs (only a large randomly chosen subset of the inputs are preset, the rest stay as variables). Consequently, we obtain a simplified circuit that works on fewer variables. This circuit will still compute parity (or the negation of parity) of the remaining variables.

By virtue of the lemma, it is possible to interchange two adjacent levels (the ones closest to the input layer) of ANDs and ORs. Then, by merging the two now adjacent levels with the same connective, we decrease the depth of the circuit to $d-1$. This is done without increasing significantly the size of the circuit.

Let us now make formal what we mean by randomly fixing some variables.

Definition 20.12 (random restriction): *A* random restriction *with a parameter ϵ to the variables $x_1, \ldots, x_n$, treats every variable x_i (independently) in the following way:*

$$x_i = \begin{cases} w.p.\ \frac{1-\epsilon}{2} & set\ x_i \leftarrow 0 \\ w.p.\ \frac{1-\epsilon}{2} & set\ x_i \leftarrow 1 \\ w.p.\ \epsilon & leave\ it\ \text{“}alive\text{”} \end{cases}$$

Observe that the expected number of variables remaining is $m = n\epsilon$. Obviously, the smaller ϵ is the more we can simplify our circuit, but on the other hand we have fewer remaining variables.

In order to contradict the induction hypothesis, we would like the size of the transformed circuit (after doing a random restriction and decreasing its depth by 1) to be smaller than $exp(o(m^{\frac{1}{d-2}}))$. It will suffice to require that $n^{\frac{1}{d-1}} < m^{\frac{1}{d-2}}$, or alternatively, $m > n \cdot n^{\frac{-1}{d-1}}$. Thus, a wise choice of the parameter ϵ, would be $\epsilon = n^{\frac{-1}{d-1}}$. ∎

20.2.3 NC and Parallel Computation

We now turn to briefly examine the connection between the complexity class $uniform\text{-}NC$ and parallel computation. In particular, we consider the connection to the parallel random-access machine (PRAM), which can be viewed as a parallel version of the RAM. A RAM is a computing device (processor), consisting of a program, a finite set of registers, and an infinite memory (whose cells are of the same size as the registers). The program is a finite sequence of instructions which are executed sequentially, where at the execution of each instruction the processor reads and writes values to some of its registers or memory cells, as required by the instruction.

The PRAM consists of several independent sequential processors (i.e. RAMs), each having its own set of private registers. In addition there is an infinite shared memory accessible by all processors. In one unit time (of a clock common to all processors), each processor executes a single instruction, during which it can read and write to its private set of registers, or to the shared memory cells. PRAMs can be classified according to restrictions on global memory access. An Exclusive-Read Exclusive-Write (EREW) PRAM forbids simultaneous access to the same memory cell by different processors. A Concurrent-Read Exclusive-Write (CREW) PRAM allows simultaneous reads but no simultaneous writes, and a Concurrent-Read Concurrent-Write (CRCW) PRAM allows simultaneous reads and writes (in this case one has to define how concurrent writes are treated). Despite this variety of different PRAM models, it turns out that they do not differ very widely in their computational power.

In designing algorithms for parallel machines we obviously want to minimize the time required to perform the concurrent computation. In particular, we would like our parallel algorithms to be *dramatically* faster than our sequential ones. Improvements in the running time would be considered dramatic if we could achieve an exponential drop in the time required to solve a problem, say from polynomial to logarithmic (or at least poly-logarithmic). Denote by $\text{PRAM}(t(\cdot), p(\cdot))$ the class of languages decidable by a PRAM machine working in parallel time $t(\cdot)$ and using $p(\cdot)$ processors, then we have the following:

Theorem 20.13 $uniform\text{-}NC = \text{PRAM}(\text{polylog},\text{poly})$

Hence, the complexity class $uniform\text{-}NC$ captures the notion of what is "efficiently" computable by PRAM machines (just as the class $\mathcal{P}$ captures the notion of what is "efficiently" computable by RAM machines, which are equivalent to Turing machines). Note however, that the PRAM cannot be considered a physically realizable model, since, as the number of processors and the size of the global memory scales up, it quickly becomes impossible to provide a constant-length data path from any processor to any memory cell. Nevertheless, the PRAM has proved to be an extremely useful vehicle for studying the logical structure of parallel computation. Algorithms developed for other, more realistic models, are often based on algorithms originally designed for the PRAM. Moreover, a transformation from the PRAM model into some other more realistic model will cost us only a logarithmic factor in the parallel complexity.

Finally, we would like to note that the analogy of NC to parallel computation has some aspects missing. First of all, it ignores the issue of the communication between processors. As a matter of fact, it seems that in practice this is the overshadowing problem to make large scale parallel computation efficient (note that the PRAM model implicitly overcomes the communication issue since two processors can communicate in $O(1)$ just by writing a message on the memory). Another aspect missing is the fact that the division of NC into subclasses based on running times seems to obscure the real bottleneck for parallel computing, which is the number of processors required. An algorithm that requires n processors and $log^2 n$ running time is likely to be far more useful than

one that requires n^2 processors and takes $\log n$, but the latter is in NC^1, the more restrictive (and hence presumably better) class.

20.3 On Circuit Depth and Space Complexity

In this section we point out a strong connection between depth of circuits with **bounded** fan-in and space complexity. It turns out that circuit depth and space complexity are *polynomially related*. In particular, we are able to prove that L (and in fact NL) falls within NC. For the sake of generality, we introduce two families of complexity classes, which can be thought of as a generalized version of NC. From now on, we assume that all circuits are uniform.

Definition 20.14 ($\mathcal{DEPTH}/\mathcal{SIZE}$): *Let d, s be integer functions. Define $\mathcal{DEPTH}/\mathcal{SIZE}(d(\cdot), s(\cdot))$ to be the class of all languages that can be decided by a uniform family of* bounded *fan-in circuits with depth $d(\cdot)$ and size $s(\cdot)$.*

In particular, if we denote by poly the set of all integer functions bounded by a polynomial and by polylog the set of all integer functions bounded by a poly-logarithmic function (e.g. $f \in$ polylog iff $f(n) = O(\log^k n)$ for some $k \geq 0$), then using the above notation, the class NC can be viewed as $\mathcal{DEPTH}/\mathcal{SIZE}(\text{polylog}, \text{poly})$.

Definition 20.15 ($\mathcal{DEPTH}$): *Let d be an integer function. Define $\mathcal{DEPTH}(d(\cdot))$ to be the class of all languages that can be decided by a uniform family of* bounded *fan-in circuits with depth $d(\cdot)$.*

Clearly, $NC \subseteq \mathcal{DEPTH}(\text{polylog})$. Note that the size of circuits deciding the languages in $\mathcal{DEPTH}(d(\cdot))$ is not limited, except for the natural upper bound of $2^{d(\cdot)}$ (due to the bounded fan-in)[1]. However, circuits deciding languages in NC are of polynomial size. Therefore, the class $\mathcal{DEPTH}(\text{polylog})$ contains languages which potentially do not belong to NC. We are now ready to connect circuit depth and space complexity.

Theorem 20.16 *For every integer function $s(\cdot)$ which is at least logarithmic,*

$$\mathcal{NSPACE}(s) \subseteq \mathcal{DEPTH}/\mathcal{SIZE}(O(s^2), 2^{O(s)})$$

Proof: Given a non-deterministic $s(n)$-space Turing machine M, we construct a uniform family of circuits, $\{C_n\}$, of depth $O(s^2)$ and size $2^{O(s)}$ such that for every $x \in \{0,1\}^*$, $C_{|x|}(x) = M(x)$.

Consider the configuration graph, $G_{M,x}$, of M on input x (see Lecture 6). Recall that the vertices of the graph are all the possible configurations of M on input x, and a directed edge leads from one configuration to another if and only if they are possible consecutive configurations of a computation on x. In order to decide whether M accepts x it is enough to check whether there exists a directed path in $G_{M,x}$ leading from the initial configuration vertex to the accepting configuration vertex. The problem of deciding, given a graph and two of its vertices, whether there exists a directed path between them, is called the directed connectivity problem (denoted $CONN$, see also Lecture 6). It turns out that $CONN$ can be decided by circuits with poly-logarithmic depth. More precisely:

Claim 20.3.1 $CONN \in NC^2$

[1] Thus, an alternative way to define $\mathcal{DEPTH}(d(\cdot))$ would be $\mathcal{DEPTH}/\mathcal{SIZE}(d(\cdot), 2^{d(\cdot)})$.

Proof: Let G be a directed graph on n vertices and let A be the adjacency matrix corresponding to it; that is, A is a boolean matrix of size $n \times n$, and $A_{i,j} = 1$ iff there is a directed edge from vertex v_i to vertex v_j in G. Now, let B be $A + I$, i.e. add to A all the self loops. Consider now the boolean product of B with itself, defined as

$$B^2_{i,j} = \bigvee_{k=1}^{n} (B_{i,k} \wedge B_{k,j}) \tag{20.1}$$

The resulting matrix will satisfy that $B^2_{i,j} = 1$ if and only if there is a directed path of length 2 or less from v_i to v_j in G. Similarly, B^4's entries will denote the existence of paths in G of length up to 4, and so on. Using $\log n$ boolean multiplications we can compute the matrix B^n, which is the adjacency matrix of the transitive closure of A - containing the answers to all the possible connectivity questions (for every pair of vertices in G). Squaring a matrix of size $n \times n$ can be done in AC^0 (see Equation 20.1) and therefore in NC^1. Hence, computing B^n can be done via repeated squaring in NC^2. ■

The circuit we build is a composition of two circuits. The first circuit takes as input some $x \in \{0,1\}^n$, a description of M (which is of a constant length) and outputs the adjacency matrix of $G_{M,x}$. The second circuit takes as input the adjacency matrix of $G_{M,x}$ and decides whether there exists a directed path from the initial configuration vertex to the accepting configuration vertex (i.e. decides $CONN$ on $G_{M,x}$). We start by constructing the first circuit. Given x and the description of M we generate all the possible configurations of M on x (there are $2^{O(s)}$ such configurations, each is represented by $O(s)$ bits). Then, for each pair of configurations we can decide if there should be a directed edge between them (i.e. whether these are possible consecutive configurations). This is done basically by comparing the contents of the work tape in the two configurations, and requires depth $O(\log s)$ (note that the size of the resulting circuit will be $2^{O(s)}$). As for the second circuit, since $G_{M,x}$ is of size $2^{O(s)}$, we have (by Claim 20.3.1) that $CONN$ can be decided on $G_{M,x}$ in depth $O(s^2)$ and size $2^{O(s)}$. Overall, we obtain a circuit C_n of depth $O(s^2)$ and of size $2^{O(s)}$, such that $C_n(x) = M(x)$. ■

Corollary 20.17 $NL \subseteq NC^2$

Proof: Take $s(n) = \log n$, and get that NL can be decided by circuits of polynomial size and of depth $O(\log^2 n)$. ■

Using the fact that $\mathcal{DEPTH}/\mathcal{SIZE}(O(s^2), 2^{O(s)})$ is contained in $\mathcal{DEPTH}(O(s^2))$, we can conclude:

Corollary 20.18 *For every integer function $s(\cdot)$ which is at least logarithmic,*

$$\mathcal{NSPACE}(s) \subseteq \mathcal{DEPTH}(O(s^2))$$

We are now ready to establish a reverse connection between circuit depth and space complexity. This is done by proving a result which is close to being a converse to Corollary 20.18 (given that for every function $s(\cdot)$, $\mathcal{DSPACE}(s) \subseteq \mathcal{NSPACE}(s)$).

Theorem 20.19 *For every function $d(\cdot)$ which is at least logarithmic,*

$$\mathcal{DEPTH}(d) \subseteq \mathcal{DSPACE}(d)$$

Proof: Given a uniform family of circuits $\{C_n\}$ of depth $d(n)$, we construct a deterministic $d(n)$-space Turing machine M such that for every $x \in \{0,1\}^*$, $M(x) = C_{|x|}(x)$. Our algorithm will be the composition of two algorithms, each using $d(n)$ space. The following lemma states that the above composition will give us a $d(n)$-space algorithm, as required (this is a special case of the Composition lemma - the search version, see Lecture 6).

Lemma 20.3.2 *Let M_1 and M_2 be two $s(n)$-space Turing machines. Then, there exists an $s(n)$-space Turing machine M, that on input x outputs $M_2(M_1(x))$.*

Our algorithm is (given input $x \in \{0,1\}^n$):

1. Obtain a description of C_n.
2. Evaluate $C_n(x)$.

A description of a circuit is a list of its gates, where for each gate we specify its type and its list of predecessors. Note that the length of the description of C_n might be exponential in $d(n)$ (since the number of gates in C_n might be exponential in $d(n)$), therefore we must use Lemma 20.3.2. The following claims establish the Theorem:

Claim 20.3.3 *A description of C_n can be generated using $O(d(n))$ space.*

Claim 20.3.4 *Circuit evaluation for bounded fan-in circuits can be solved in space=O(circuit depth).*

Proof: (of Claim 20.3.3) By the uniformity of $\{C_n\}$, there exists a deterministic machine M such that $M(1^n) = desc(C_n)$, while using $\log(|desc(C_n)|)$ space. Since $|desc(C_n)| \leq 2^{O(d(n))}$, we get that M uses $O(d(n))$ space, as required. ■

Proof: (of Claim 20.3.4) Given a circuit C of depth d and an input x, we want to compute $C(x)$. Our implementation will be recursive. A natural approach would be to use the following algorithm:

Denote by $VALUE(C_x, v)$ the value of vertex v in the circuit C when assigning x to its inputs, where v is encoded in binary (i.e. using $\log(size(C)) = O(d)$ bits). Note that $VALUE(C_x,$'*output*'$)$ is equal to the desired value, $C(x)$. The following procedure obtains $VALUE(C_x, v)$:

1. If v is a leaf then return the value assigned to it.
   Otherwise, let u and w be v's predecessors and **op** be v's label.
2. Compute recursively $\sigma \leftarrow VALUE(C_x, u)$ and $\tau \leftarrow VALUE(C_x, w)$.
3. Return σ **op** τ.

Notice that Step 2 in the algorithm requires two recursion calls. Since we are going only one level down each recursion call, one may hastily conclude that the space consumed by the algorithm is $2^{O(d)}$. Remember however, that we are dealing with space, this means that the space consumed in the first recursion call can be reused by the second one, and therefore the actual space consumption is $O(d^2)$ (since there are $O(d)$ levels in the recursion, and on each level we need to memorize the vertex name which is of length $O(d)$).

This is still not good enough, remember that our goal is to design an algorithm working in space $O(d)$. This will be done by representing the vertices of C in a different manner: Each vertex will be specified by a path reaching it from the output vertex. The output vertex will be represented

by the empty string ϵ. Its left predecessor will be represented by "0", and its right predecessor by "1", the left predecessor of "1" will be represented by "10", and so on. Since there might be several paths reaching a vertex, it might have multiple names assigned to it, but this will not bother us.

Consequently, each vertex is represented by a bit string of length $O(d)$. Moreover, obtaining from a vertex name its predecessor's or successor's name is done simply by concatenation of a bit or deletion of the last bit. The following procedure computes $VALUE(C_x, \overline{path})$ in $O(d)$ space:

1. Check whether $\overline{path}$ defines a leaf. If it does, then return the value assigned to it. Otherwise, let **op** be the label of the corresponding vertex.
2. Compute recursively $\sigma \leftarrow VALUE(C_x, \overline{path} \circ 0)$ and $\tau \leftarrow VALUE(C_x, \overline{path} \circ 1)$.
3. Return σ **op** τ.

When computing this procedure, C_x will be written on the input tape and $\overline{path}$ will be written on the work tape. At each level of the recursion, $\overline{path}$ determines precisely all the previous recursion levels, so the space consumption of this algorithm is $O(d)$, as required. ■

■

Corollary 20.20 $NC^1 \subseteq L \subseteq NL \subseteq NC^2$

Proof: The first inclusion follows from Theorem 20.19, the second inclusion is trivial and the third inclusion is just Corollary 20.17. ■

Bibliographic Notes

This lecture is mostly based on [1]. For wider perspective see Cook's old survey [2].

1. A. Borodin. On relating time and space to size and depth. *SIAM J. on Computing*, Vol. 6, No. 4, pages 733–744, 1977.
2. S.A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, Vol. 64, pages 2–22, 1985.

Lecture 21

Communication Complexity

Lecture given by Ran Raz
Notes taken by Amiel Ferman and Noam Sadot

Summary: This lecture deals with Communication Complexity, which is the analysis of the amount of information that needs to be communicated by two parties which are interested in reaching a common computational goal. We start with some basic definitions and simple examples. We continue to consider both deterministic and probabilistic models for the problem, and then we develop a combinatorial tool to help us with the proofs of lower bounds for communication problems. We conclude by proving a probabilistic linear communication complexity lower bound for the problem of computing the inner product of two vectors where initially each party holds one vector.

21.1 Introduction

The communication problem arises when two or more parties (i.e. processes, systems etcetera) need to carry out a task which could not be carried out alone by each of them because of lack of information. Thus, in order to achieve some common goal, defined as a function of their inputs, the parties need to communicate. Often, the formulation of a problem as a communication problem serves merely as a convenient abstraction; for example, a task that needs to share information between diefferent parts of the same CPU could be formulated as such. Communication complexity is concerned with analaysing the amount of information that must be communicated between the different parties in order to correctly perform the intended task.

21.2 Basic model and some examples

In order to investigate the general problem of communication we state a few simplified assumptions on our model:

1. There are only two parties (called player 1 and player 2)
2. Each party has unlimited computing power and we are only concerned with the communication complexity
3. The task is a computation of a predefined function of the input

As we shall see, this model is rich enough to study some non-trivial and interesting aspects of communication complexity.

The input domains of player 1 and player 2 are the (finite) sets X and Y respectively. The two players start with inputs $x \in X$ and $y \in Y$, and their task is to compute some predefined function $f(x, y)$. At each step, the communication protocol specifies which bit is sent by one of the players (alternately), and this is based on information communicated so far as well as on the initial inputs of the players.

Let us see a few examples:

1. Equality Function (denoted EQ):

   The function $f(x, y)$ is defined as:

   $f(x, y) = 1$ if $x = y$

   $f(x, y) = 0$ if $x \neq y$

   That is, the two players are interested to know wheather their initial inputs are equal.

2. Disjointness:

   The inputs are subsets: $x, y \subseteq \{1, \ldots, n\}$

   $f(x, y) = 1$ iff $x \cap y \neq \emptyset$

3. Inner Product (denoted IP):

   The inputs are: $x, y \in \{0, 1\}^n$

   $f(x, y) = \sum_i^n x_i \cdot y_i \bmod 2$

21.3 Deterministic versus Probabilistic Complexity

We begin with some definitions:

Definition 21.1 *A deterministic protocol P with domain $X \times Y$ and with range Z (where X and Y are the input domains of player 1 and player 2 respectively and Z is the domain of the function f) is defined as a deterministic algorithm in which at each step it specifies a bit to be sent from one of the players to the other. The output of the algorithm, denoted $P(x, y)$ (on inputs x and y), is the output of each of the players at the end of the protocol and it is required that:*

$$\forall x \in X, y \in Y \qquad : \qquad P(x, y) = f(x, y)$$

Definition 21.2 *The communication complexity of a deterministic protocol is the worst case number of bits sent by the protocol for some inputs.*

Definition 21.3 *The communication complexity of a function f is the minimum complexity of all deterministic protocols which compute f. This is denoted by $CC(f)$.*

A natural relaxation of the above defined deterministic protocol would be to allow each player to toss coins during his computation. This means that each player has an access to a random string and the protocol that is carried out depends on this string. The way to formulate this is to determine a distribution Π from which the random strings each player uses are sampled uniformly, once the strings are chosen, the protocol that is carried out by each of the players is completely

deterministic. We consider the Monte-Carlo model, that is, the protocol should be correct for a large fraction of the strings in Π.

Note that the above description of a randomized protocol implicitly allows for two kinds of possibilities: one in which the string that is initially sampled from Π is common to both players, and the other is that each player initially samples his own private string so that the string sampled by one player is not visible to the other player. These two possibilites are called respectively the public and the private model. How are these two models related ? First of all, it is clear that any private protocol can be simulated as a public protocol: the strings sampled privately by each user are concatenated and serve as the public string. It turns out that a weaker reduction exists in the other direction: any public protocol can be simulated as a private protocol with a small increase in the error and an additive of $O(\log n)$ bits of communication; the idea of the proof is to show that any public protocol can be transformed to a protocol which uses the same amount of communication bits but only $O(\log n + \log \delta^{-1})$ random bits with an increase of δ in the error. Next, each player can sample a string of that length and send it to the other player thus causing an increase of $O(\log n + \log \delta^{-1})$ in the communication complexity and of δ in the error. In view of these results we shall confine ourselves to the public model.

Definition 21.4 *A randomized protocol P is defined as an algorithm which initially samples uniformly a string from some distribution Π and then carries on exactly as in the deterministic case. The sampled string is common to both player, i.e. - this is the public model. It is required that an ϵ - error protocol will satisfy:*

$$\forall x \in X, y \in Y \qquad Pr_{r \in_R \Pi}[P(x,y) = f(x,y)] \geq 1 - \epsilon$$

Note that in a randomized protocol, the number of bits communicated may vary for the same input (due to different random strings). Hence the communication complexity is defined with respect to the strings sampled from Π. One can define the communication complexity of a protocol, viewed as a random variable (with respect to the distribution Π, and the worst possible input), in the average case. However we prefer the somewhat stronger worst-case behaviour:

Definition 21.5 *The communication complexity of a randomized protocol P on input (x,y) is the maximum number of bits communicated in the protocol for any choice of initial random strings of each player. The communication complexity of a randomized protocol P is the maximum communication complexity of P over all possible inputs (x,y)*

Definition 21.6 *The communication complexity of a function f computed with error probability ϵ, denoted $CC_\epsilon(f)$ is the minimum communication complexity of a protocol P which computes f with error probability ϵ.*

In Lecture 3 we have actually considered a private case of the above definition, one in which there is no error; i.e., $CC_0(f)$.

21.4 Equality revisited and the Input Matrix

Recall the Equality Function defined in Section 2, in which both players wish to know wheather their inputs (which are n-bit strings) are equal (i.e. whether $x = y$). Let us first present a randomized protocol which computes EQ with a constant error probability and a constant communication complexity:

Protocol for player i (i = 1,2) ($input_1 = x$ and $input_2 = y$):

1. sample uniformly an n-bit string r (this r is common to both players - public model)
2. compute $< input_i, r >_2$ (the inner product of $input_i$ and r mod 2)
3. send the product computed and receive the product computed by the other player (single bit)
4. if the two bits are equal then output 1 else output 0

If the inputs are equal, i.e. $x = y$, then clearly $< x, r >_2 = < y, r >_2$ for all r-s and thus each player will receive and send the same bit and will decide 1. However, if $x \neq y$, then for a string r sampled uniformly we have that $< x, r >_2 = < y, r >_2$ with probability exactly one half. Thus, the error probability of a single iteration of the above protocol is exactly one half. Since at each iteration we sample a random string r independantly from other iterations we get that after carrying out the protocol for exactly C times the error probability is exactly 2^{-C}. Furthermore, since the number of bits communicated in each iteration is constant (exactly two bits), we get that after C iterations of the above protocol, the communication complexity is $O(C)$. Hence, if C is a constant, we get both a constant error probability and a constant communication complexity (2^{-c} and $O(1)$ respectively for a constant c). However, if we choose C to be equal to $log(n)$ then the error probability and the communication complexity will be, respectively, $\frac{1}{n}$ and $O(log(n))$.

We now present an alternative protocol for solving EQ that also achieves an error probabilty of $\frac{1}{n}$ and communication complexity of $O(log(n))$. Interestingly, this protocol is (already) in the private model:

We present both (n-bit strings) inputs as the coefficients of polynomials over $GF(p)$ where p is an arbitrary fixed prime between n^2 and $2n^2$ (results in number theory guarantee the existence of such a prime). So, both inputs may be viewed as:

input of player 1:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i \bmod p$$

input of player 2:

$$B(x) = \sum_{i=0}^{n-1} b_i \cdot x^i \bmod p$$

Protocol for player 1 (For player 2: just reverse A and B)

1. choose uniformly a number t in $GF(p)$
2. compute $A(t)$
3. send both t and $A(t)$ to the other player
4. receive s and $B(s)$ from other player
5. if $A(s) = S$ then decide 1 else decide 0

Clearly, if the inputs are equal then so are the polynomials and thus necessarily $A(t) = B(t)$ for every $t \in GF(p)$. If however, $A \neq B$, then these polynomials have at most $n - 1$ points on which they agree (i.e. t-s for which $A(t) = B(t)$) since their difference is a polynomial of degree

$n-1$ which can have at most $n-1$ roots. So the probability of error in this case is $\frac{n-1}{p} \leq \frac{n}{n^2} = \frac{1}{n}$. Notice that since t and $B(t)$ are $O(lgn)$ bits long, we may conclude that $CC_{\frac{1}{n}}(EQ) = O(lgn)$.

Proofs of lower bounds which relate to certain families of algorithms usually necessitate a formalization that could express in a non-trivial way the underlying structure. In our case, a combinatorial view proves to be effective: (Recall that the input domains of both parties are denoted X and Y) we may view the protocol as a process which in each of its steps partitions the input space $X \times Y$ into disjoint sets such that at step t each set includes exactly all input pairs which according to their first t bits cause the protocol to "act" the same (i.e., communicate exactly the same messages during the algorithm). Intuitively, each set at the end of this partioning process is comprised of exactly all pairs of inputs that "cause" the protocol to reach the same conclusion (i.e., compute the same output).

A nice way to visualize this is to use a matrix: each row corresponds to a $y \in Y$ and each column corresponds to a $x \in X$. The value of the matrix in position (i,j) is simply $f(i,j)$, where f is the function both parties need to compute. This matrix is called the Input Matrix. Since the Input Matrix is just another way to describe the function f,we may choose to talk about the communication complexity of an Input Matrix A - dentoed $CC(A)$ instead of the communication complexity of the corresponding function f. For example, the matrix corresponding to the Equality Function is the indentity matrix (since the output of the Equality Function must be 1 iff the inputs are of the form (i,i) for each input pair). The above mentioned partioning process can now be viewed as a partioning of the matrix into sets of matrix elements. It turns out that these sets have a special structure, namely rectangles. Formally, we define

Definition 21.7 *A* rectangle *in $X \times Y$ is a subset $R \subseteq X \times Y$ such that $R = A \times B$ for some $A \subseteq X$ and $B \subseteq Y$.* (Note that elements of the rectangle, as defined above, need not be adjacent in the input matrix.)

However, in order to relate our discussion to this definition we need an alternative characterization of rectangles given in the next proposition:

Proposition 21.4.1 *$R \subseteq X \times Y$ is a rectangle iff $(x_1,y_1) \in R$ and $(x_2,y_2) \in R \Rightarrow (x_1,y_2) \in R$*

Proof: $\Rightarrow$ If $R = A \times B$ is a rectangle then from $(x_1,y_1) \in R$ we get that $x_1 \in A$ and from $(x_2,y_2) \in B$ we get that $y_2 \in B$ and so we get that $(x_1,y_2) \in A \times B = R$.

$\Leftarrow$ We define the sets $A = \{x | \exists y \ s.t. \ (x,y) \in R\}$ and $B = \{y | \exists x \ s.t. \ (x,y) \in R\}$. On the one hand it is clear that $R \subseteq A \times B$ (directly from A and B's definition). On the other hand, suppose $(x,y) \in A \times B$. Then since $x \in A$ there is a y' such that $(x,y') \in R$ and similiarly there is an x' such that $(x',y) \in R$ from this, according to the assumption we have that $(x,y) \in R$. ■

We shall now show that the sets of matrix elements partitioned by the protocol in the sense described above actually form a partition of the matrix into rectangles: Suppose both pairs of inputs (x_1,y_1) and (x_2,y_2) cause the protocol to exchange the same sequence of messages. Since the first player (with input x_1) cannot distinguish at each step between (x_1,y_1) and (x_1,y_2) (he computes a function of x_1 and the messages so far in any case) then he will communicate the same message to player 2 in both cases. Similarly, player 2 cannot distinguish at each step between (x_2,y_2) and (x_1,y_2) and will act the same in both cases. We showed that if the protocol acts the same on inputs (x_1,y_1) and (x_2,y_2) then it will act the same on input (x_1,y_2), which, using proposition 21.4.1 establishes the fact that the set of inputs on which the proctocol behaves the same is a rectangle.

Since the communication is the same during the protocol for the pair of inputs (x_1, y_1) and (x_2, y_2) (and for the pairs of inputs in the rectangle defined by them, as was explained in the last paragraph) then the protcol's output must be the same for these pairs, and this implies that the value of the f function must be the same too. Thus, a deterministic protocol partitions the Input Matrix into rectangles whose elements are identical, that is, the protocol computes the same output for each pair of inputs in the rectangle. We say that a deterministic protocol partitions the Input Matrix into rectangles of monochromatic rectangles (where color is identified with the input matrix value). Since at each step the protocol partitions the Input Matrix into two (usually not equal in size) parts we have the following:

Fact 21.4.2 *A deterministic protocol P of communication complexity k partitions the Input Matrix into at most 2^k monochromatic rectangles*

Recalling the fact that the Input Matrix of a protocol to the equality problem is the identity matrix, then since the smallest monochromatic rectangle that contains each entry of 1 in the matrix is the singleton matrix which contains exactly one element and since the matrix is of size $2^n \times 2^n$ (for inputs of size n), we get that every protocol for the equality problem must have partitioned the Input Matrix into at least $2^n + 1$ monochromatic rectangles (2^n for the 1's and at least 1 for the zeros). Thus, from Fact 21.4.2 and from the trivial protocol for solving EQ in which player 1 sends its input to the player 2 and player 2 sends to player 1 the bit 1 iff the inputs are equal ($n+1$ bits of communication), we get the following corollary:

Corollary 21.8 $CC(EQ) = n + 1$

21.5 Rank Lower Bound

Using the notion of an Input Matrix developed in the previous section, we now state and prove a useful theorem regarding the lower bound of communication complexity:

Theorem 21.9 *Let A be an Input Matrix for a certain function f, then $CC(A) \geq log_2(r_A)$ where r_A is the rank of A over any fixed field F*

Proof: The proof is by induction on $CC(A)$.

Induction Base: If $CC(A) = 0$ then this means that both sides were able to compute the function f without any communication. This means that for every pair of inputs, both sides compute a constant function which could be either $f(x, y) = 1$ or $f(x, y) = 0$ for all x and y. This implies that A must be the all 0-s or the all 1-s matrix, and so, by definition $r_A \in 0, 1$. Thus, indeed, $CC(A) \geq log_2(r_A)$ as required.

Induction Step: Suppose the claim is true for $CC(A) \leq n - 1$ and we shall prove the claim for $CC(A) = n$. Consider the first bit sent: This bit actually partitions A into two matrices A_0 and A_1 such that the rest of the protocol can be seen as a protcol that relates to only one of these matrices. Since the maximal communication complexity needed for both matrices cannot surpass $n - 1$ (otherwise $CC(A)$ could not have been equal to n), we get the following equation:

$$CC(A) \geq 1 + Max\{CC(A_0), CC(A_1)\} \geq 1 + Max\{log_2(r_{A_0}), log_2(r_{A_1})\} \tag{21.1}$$

where the second inequality is by the induction hypothesis. Now, since $r_A \leq r_{A_0} + r_{A_1}$, we have that $r_A \leq 2 \cdot Max\{r_{A_0}, r_{A_1}\}$. Put differently, we have $Max\{log_2(r_{A_0}), log_2(r_{A_1})\} \geq log_2(r_A) - 1$. Combining this with Eq. (21.1) we get that $CC(A) \geq 1 + log_2(r_A) - 1 = log_2(r_A)$. ∎

Applying this theorem to the Input Matrix of the equality problem (the identity matrix), we easily get the lower bound $CC(EQ) \geq n$. A linear lower bound for the deterministic communication complexity of the Inner Product problem can also be achieved by applying this theorem. In the next section we'll see a linear lower bound on the randomized communication complexity of the Inner Product function.

21.6 Inner-Product lower bound

Recalling Inner-Product problem from section 21.2, we prove the following result:

Theorem 21.10 $CC_\epsilon(IP) = \Omega(n)$

To simplify the proof, and the mathematical techniques needed for the proof, we will assume that $0 < \epsilon < (\frac{1}{4} - \tau)$, for arbitarary small $\tau > 0$.

To prove the above theorem, we assume that there is a probabilistic communication protocol P in the public coin model using random string R that uses less than $n\epsilon$ communication bits, and we will show a contradiction. By definition, we know that

$$\Pr_R[P_R(x,y) = f(x,y)] \geq 1 - \epsilon$$

for every string x and y. Since it is true for each pair of strings, the following property is true:

$$\Pr_{(x,y),R}[P_R(x,y) = f(x,y)] \geq 1 - \epsilon$$

in which the probability measure over (x,y) is taken as the uniform probability over all such pairs. Changing the order of the probability measures, we obtain:

$$\Pr_{R,(x,y)}[P_R(x,y) = f(x,y)] \geq 1 - \epsilon$$

And since $\epsilon < 1$, we conclude that there is a fixed random string r that for the deterministic protocol induced by P_r, the following property exists:

$$\Pr_{(x,y)}[P_r(x,y) = f(x,y)] \geq 1 - \epsilon$$

By this method, we produce a deterministic protocol on which we can work now and prove lower bound. In contrast to the previous section, the protocol P_r should work well only for most of the inputs, but not necessarily for all of them. Proving lower bound on this deterministic protocol P_r will immediately give the lower bound of the original randomized protocol P.

The method for proving the lower bound is to show that there are not "big" enough rectangles that are not balanced in the input matrix after $n\epsilon$ time, and hence to conclude that we need more than that time. The following definition will be helpful:

Definition 21.11 *A rectangle $U \times V \subset \{0,1\}^n \times \{0,1\}^n$ is* big *if its size satisfies $|U \times V| \geq 2^{2n(1-\epsilon)}$. Otherwise, the rectangle is* small.

Note that there must be at least one big rectangle in the above matrix. Otherwise, using Fact 21.4.2 and the fact that we have at most $2^{n\epsilon}$ rectangles (for at most $n\epsilon$ communiaction), we infer that the size of the entire matrix is not more than $2^{n\epsilon} \cdot 2^{2n(1-\epsilon)} = 2^{2n(1-\frac{\epsilon}{2})} < 2^{2n}$ which leads to a contradiction.

Claim 21.6.1 *If P_r works in $n\epsilon$ time then there exists a big rectangle $U \times V \subset \{0,1\}^n \times \{0,1\}^n$ such that $f(x,y)$ is the same for at least $1-2\epsilon$ fraction of the elements in the rectangle $U \times V$.*

Proof: To prove the claim, we recall that $\Pr_{(x,y)}[P_r(x,y) = f(x,y)] \geq 1-\epsilon$. In other words, at most ϵ fraction of the elements in the matrix do not satisfy $P_r(x,y) = f(x,y)$. In addition, after at most $n\epsilon$ time, we will have a partition of the matrix into at most $2^{n\epsilon}$ rectangles. By Definition 21.11, each small rectangle has size less than $2^{2n(1-\epsilon)}$, and so the number of the elements that belong to small rectangles is less than $2^{n\epsilon} \cdot 2^{2n(1-\epsilon)} = 2^{2n(1-\frac{\epsilon}{2})} < 2^{2n-1}$, and so big rectangles contain more than half of the matrix elements. Thus, if all big rectangles have more than 2ϵ error, the total error due only to these rectangles would be more than ϵ, which leads to a contradiction. The claim follows. ■

By using this claim, we fix a big rectangle R that satisfies the conditions of the previous claim. Without loss of generality, we can assume that the majority of this rectangle is 0. If it is not 0, we just switch every element in the matrix.

Let us denote by B_n the $2^n \times 2^n$ input matrix for Inner Product problem, which looks like that:

$$B_n = \left\{ x \cdot y \bmod 2 \right\}_{(x,y) \in \{0,1\}^n \times \{0,1\}^n}$$

The inner elements are scalar products on the field $GF(2)$. The matrix B_n contains two types of elements: zeroes and ones. By switching each 0 into 1 and each 1 into -1, we get a new matrix H_n (which is a version of Hadamard matrix) that looks like that:

$$H_n = \left\{ (-1)^{x \cdot y} \right\}_{(x,y) \in \{0,1\}^n \times \{0,1\}^n}$$

This matrix has the following property:

Claim 21.6.2 *H_n is an orthogonal matrix over the reals.*

Proof: We will prove that each two rows in the matrix H_n are orthogonal. Let r_x be a row corresponds to a string x and r_z be a row corresponds to a string $z \neq x$. The scalar product between these two rows is

$$\sum_{y \in \{0,1\}^n} (-1)^{x \cdot y} \cdot (-1)^{z \cdot y} =$$

$$\sum_{y \in \{0,1\}^n} (-1)^{\Delta \cdot y}$$

where $\Delta = x \oplus z$. Since $x \neq z$ there is an index $j \in \{1, \ldots, n\}$ such that $\Delta_j = 1$, then the previous expression is equal

$$\sum_{y \in \{0,1\}^n} (-1)^{\sum_{i \neq j} y_i \Delta_i + y_j} \tag{21.2}$$

Let us denote by $y' = y_1 \ldots y_{j-1} y_{j+1} \ldots y_n$, then we can write (21.2) as:

$$\sum_{y'} \sum_{y_j} (-1)^{\sum_{i \neq j} y_i \Delta_i + y_j} =$$

$$\sum_{y'} (-1)^{\sum_{i \neq j} y_i \Delta_i} \sum_{y_j} (-1)^{y_j}$$

Clearly,

$$\sum_{y_j \in \{0,1\}} (-1)^{y_j} = -1 + 1 = 0$$

which proves the claim. ■

We have 2^n rows and columns in the matrix H_n. Let us enumerate the rows by r_i, for $i = 0, 1, \ldots, 2^n - 1$. Then by the previous claim, we have the following properties, where here $\cdot$ denotes inner product over the reals:

1. $r_i \cdot r_j = 0$ for $i \neq j$
2. $r_i \cdot r_i = \|r_i\|^2 = 2^n$ for $i = 0, 1, \ldots, 2^n - 1$. This follows easily from the fact that the absolute value of each element in H_n is 1.

Thus, the rows in the matrix define an orthogonal base over the reals.

The following definition will be helpful in the construction of the proof of Theorem 21.10:

Definition 21.12 (discrepency): *The* discrepency *of a rectangle $U \times V$ is defined as*

$$D(U \times V) = \left| \sum_{(x,y) \in U \times V} (-1)^{f(x,y)} \right|$$

Let R_0 be a big rectangle (of small error) as guaranteed by Claim 21.6.1. Suppose without loss of generality that R_0 has a majority of zeros (i.e., at least $1 - 2\epsilon$ fraction of 0's). Recall that the size of R_0 is at least $2^{2n(1-\epsilon)}$. Thus, R_0 has a big discrepency; that is,

$$D(R_0) \;\geq\; (1 - 2\epsilon - 2\epsilon) \cdot 2^{2n(1-\epsilon)} \;=\; (1 - 4\epsilon) \cdot 2^{2n(1-\epsilon)} \tag{21.3}$$

On the other hand, we have an upper bound on the discrepency of any rectangle (an in particular of R_0):

Lemma 21.6.3 *The discrepency of any rectangle R is bounded from above by $2^{-\frac{n}{2}} \cdot 2^{2n}$*

Proof: Let us denote $R = U \times V$. The matrix H_n has the property that each bit b in B_n changes into $(-1)^b$ in H_n. Let us consider the following charactaristic vector $I_U : \{0,1\}^n \to \{0,1\}^n$ that is defined in the following way:

$$I_U(x) = \begin{cases} 1 & \text{if } x \in U \\ 0 & \text{otherwise} \end{cases}$$

Observe that $I_U \cdot r_j$ is exactly the number of 1's minus the number (-1)'s in r_j, so

$$\begin{aligned} D(U \times V) &= \left| \sum_{j \in V} I_U \cdot r_j \right| \leq \sum_{j \in V} |I_U \cdot r_j| \\ &\leq \sum_{j \in \{0,1\}^n} |I_U \cdot r_j| \end{aligned}$$

where both inequalities are trivial. Using Cauchy-Schwartz inequality (for the second line), we obtain

$$\begin{aligned} D(R) &\leq \sum_{j \in \{0,1\}^n} 1 \cdot |I_U \cdot r_j| \\ &\leq \sqrt{2^n \cdot \sum_{j \in \{0,1\}^n} |I_U \cdot r_j|^2} \end{aligned} \tag{21.4}$$

Recalling that H_n is an orthogonal matrix, and the norm of each row is $\sqrt{2^n}$, we denote $\hat{r_j} = \frac{1}{\sqrt{2^n}} r_j$ which define an orthonormal base. With this notation, Eq. (21.4) can be written as:

$$\begin{aligned}\sqrt{2^n \cdot \sum_{j\in\{0,1\}^n} |I_U \cdot \sqrt{2^n}\hat{r_j}|^2} &= \sqrt{2^n \cdot 2^n \cdot \sum_{j\in\{0,1\}^n} |I_U \cdot \hat{r_j}|^2} \\ &= 2^n \cdot \sqrt{\sum_{j\in\{0,1\}^n} |I_U \cdot \hat{r_j}|^2}\end{aligned}$$

Since $\{\hat{r_j}\}_{j=0,1,\ldots,2^n-1}$ is an orthonormal base, the square root above is merely the norm of I_U (as the norm is invariant over all orthonormal bases). However, looking at the "standard" (point-wise) base, we have that the norm of I_U is $\sqrt{|U|} \leq \sqrt{2^n}$ (since each element in the vector I_U is 0 or 1). To conclude, we got that:

$$D(R) \leq 2^n \cdot \sqrt{2^n} = 2^{\frac{3n}{2}} \tag{21.5}$$

which proves the lemma. ■

We now derive a contradiction by contrasting the upper and lower bounds provided for R_0. By Eq. (21.3), we got that:

$$D(R_0) \geq (1 - 4\epsilon) \cdot 2^{2n(1-\epsilon)}$$

which is greater than $2^{3n/2}$ for any $0 < \epsilon < \frac{1}{4}$ and all sufficiently large n's (since for such ϵ the exponent is strictly bigger than $3n/2$ which for sufficiently big n's compensates for the small positive factor $1 - 4\epsilon$). In contrast, Lemma 21.6.3 applies also to R_0 and implies that $D(R_0) \leq 2^{3n/2}$, in contradiction to the above bound (of $D(R_0) > 2^{3n/2}$).

To conclude, we showed a contradiction to our initial hypothesis that the comminication complexity is lower than ϵn. Theorem 21.10 thus follows.

Bibliographic Notes

For further discussion of Communication Complexity see the textbook [2]. Specifically, this lecture corresponds to Chapters 1 and 3.

Communication Complexity was first defined and studied by Yao [4], who also introduced the "rectangle-based" proof technique. The rank lower bound was suggested in [3]. The lower bound on the communication complexity of the Inner Product function is due to [1].

1. B. Chor and O. Goldreich. Unbiased Bits From Sources of Weak Randomness and Probabilistic Communication Complexity. *SIAM J. Comp.*, Vol. 17, No. 2, April 1988, pp. 230–261.

2. E. Kushilevitz and N. Nisan. *Communication Complexity*, Cambridge University Press, 1996.

3. K. Mehlhorn and E. Schmidt. Las-Vegas is better than Determinism in VLSI and Distributed Computing. In *Proc. of 14th STOC*, pp. 330-337, 1982.

4. A.C. Yao. Some Complexity Questions Related to Distributive Computing. In *Proc. of 11th STOC*, pp. 209-213, 1979.

Lecture 22

Monotone Circuit Depth and Communication Complexity

Lecture given by Ran Raz
Notes taken by Yael Tauman and Yoav Rodeh

Summary: One of the main goals of studying circuit complexity is to prove lower bounds on the size and depth of circuits computing specific functions. Since studying the general model gave few results, we will concentrate on monotone circuits. The main result is a tight nontrivial bound on the monotone circuit depth of *st-Connectivity*. This is proved via a series of reductions, the first of which is of significant importance: A connection between circuit depth and communication complexity. We then get a communication game and proceed to reduce into other such games, until reaching the game $FORK$, and the conclusion that proving a lower bound on its communication complexity will give a matching lower bound on the monotone circuit depth of *st-Connectivity*.

22.1 Introduction

Turing machines are abstract models used to capture our concept of computation. However, we tend to miss some complexity properties of functions when examining them from the Turing machine point of view. One such central property of a function, is how efficiently it can be run in parallel. This property is best observed when we use the circuit model — by the depth of the circuit realizing the function. Another motivation for prefering the circuit model to the Turing machine model, is the hope that using advanced combinatorial methods will more easily give lower bounds to the size of circuits, and hence to the running time of Turing machines.

Recall that we need only examine circuits made up of $NOT(\neg)$ $OR(\vee)$ and $AND(\wedge)$ gates, any other gate can be simulated with constant blowup in the size and depth of the circuit. We may also assume all the NOT gates are at the leaf level because using De-Morgan rewrite rules, we do not increase the depth of the circuit at all, and may increase its size by a constant factor of 2 at most. In this lecture we will only discuss bounded fan-in circuits, and therefore may assume all gates to be of fan-in 2 (except NOT).

As always, our goal is to find (or at least prove the existence of) hard functions. In the context of circuit complexity we measure hardness by two parameters:

Definition 22.1 (Depth, Size): *Given* $f : \{0,1\}^n \to \{0,1\}$*, we define:*

1. *$Depth(f) \stackrel{\text{def}}{=}$ The minimum depth of a circuit computing f, where the depth of a circuit is the maximum distance from an input leaf to the output (when the circuit is viewed as a directed acyclic graph).*

2. *$Size(f) \stackrel{\text{def}}{=}$ The minimum size of a circuit computing f, where the size of a circuit is the number of gates it contains.*

Note that these quantities do not necessarily correlate: A circuit that computes f and has size $Size(f)$ and depth $Depth(f)$ may not exist. In other words, it is possible that every circuit of minimal size does not achieve minimal depth.

We will first prove the existence of hard functions:

22.1.1 Hard Functions Exist

There are no explicit families of functions that are proved to need large size circuits, but using counting arguments we can easily prove the existence of functions with size that is exponential in the size of their input.

Proposition 22.1.1 *For large enough n, there exists a function $f : \{0,1\}^n \to \{0,1\}$ s.t. $Size(f) > \frac{2^n}{n^2}$.*

Proof: First easy observation is that the number of functions $\{f \mid f : \{0,1\}^n \to \{0,1\}\}$ is exactly 2^{2^n}, since each such function can be represented as a $\{0,1\}$ vector of length 2^n.

We will now upper bound the number of circuits of size s. The way we approach the problem is by adding one gate at a time, starting from the inputs. At first we have $2n$ inputs — the variables and their negations. Each gate we add, is either an OR or an AND gate, and its two inputs can be chosen from any of the original inputs or from the outputs of the gates we already have. Therefore, for the first gate we have $\binom{2n}{2}$ choices for the the inputs and another choice between OR and AND. For the second gate we have exactly the same, except now the number of inputs to choose from is increased by one. Thus, the number of circuits of size s is bounded by:

$$\prod_{i=0}^{s-1} 2 \cdot \binom{2n+i}{2} < \prod_{i=0}^{s-1} 2 \cdot \frac{(2n+i)^2}{2} < \prod_{i=0}^{s-1} (2n+s)^2 = (2n+s)^{2s} = 2^{2s \cdot \log(2n+s)}$$

We wish to prove that the number of circuits of size $s = \frac{2^n}{n^2}$ is strictly less than the number of functions on n variables, and hence prove that there are functions that need circuits of size larger than s. For this we need to prove:

$$\begin{gathered} 2^{2s\log(2n+s)} < 2^{2^n} \\ \Updownarrow \\ 2s\log(2n+s) < 2^n \end{gathered}$$

Which is obviously true for $s < \frac{2^n}{n^2}$, since for large enough n:

$$2s\log(2n+s) < 2 \cdot \frac{2^n}{n^2}\log(2^n) = 2^n \cdot (\frac{2}{n}) < 2^n$$

■

If we examine the proof carefully, we can see that actually most functions need a large circuit. Thus it would seem that it should be easy to find such a hard function. However, to the shame of all

involved, the best known lower bounds for size and depth of "explicitly given" functions (actually families of functions) are:

$$\begin{aligned} Size &\geq 4n \\ Depth &\geq 3\log(n) \end{aligned}$$

We therefore focus on weaker models of computation:

22.1.2 Bounded Depth Circuits

The first model we consider is that of bounded depth circuits. There are two deviations from the standard model. The first is that we artificially bound the depth of the circuit, and only consider the size of the circuit as a parameter for complexity. This immediately implies the other difference from the standard model: We do not bound the fan-in of gates. This is because otherwise, if we bound the depth to be a constant d, we automatically bound the size to be less than 2^d which is also a constant. This makes the model uninteresting, therefore we allow unbounded fan-in. Notice that any function can be computed by a depth 2 circuit (not counting NOT's) by transforming the function's truth table into an OR of many AND's. However, this construction gives exponential size circuits. Several results were reached for this model (see Lecture 20), but we will focus on a different model in this lecture.

22.2 Monotone Circuits

Monotone circuits is the model we consider next, and throughout the rest of this lecture. Monotone circuits are defined in the same way as usual circuits except we do not allow the usage of NOT gates.

It seems intuitive that monotone circuits cannot calculate any function, because there is no way to simulate a NOT gate using AND and OR gates. We will formulate and prove a characterization of the functions that can be computed using monotone circuits:

Definition 22.2 (Monotone Function): *$f : \{0,1\}^n \to \{0,1\}$ is a* monotone function *if for every $x, y \in \{0,1\}^n$, $x \geq y$ implies $f(x) \geq f(y)$. Where the partial order $\geq$ on $\{0,1\}^n$ is the hamming order, i.e., $(x_1, \ldots, x_n) \geq (y_1, \ldots, y_n)$ if and only if for every $1 \leq i \leq n$ we have $x_i \geq y_i$.*

Remark: The hamming partial order can be thought of as the containment order between sets, where a vector $x \in \{0,1\}^n$ corresponds to the set $S_x = \{i \mid x_i = 1\}$. Then: $x \leq y$ if and only if $S_x \subseteq S_y$.

An example of a monotone function is $CLIQUE_{n,k} : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$. The domain of the function $CLIQUE_{n,k}$ is the set of graphs on n vertices $\{1, \ldots n\}$. A graph is represented by assignments to the $\binom{n}{2}$ variables $x_{i,j}$, where for every pair $i, j \in \{1, \ldots n\}$, $x_{i,j} = 1$ iff (i, j) is an edge in the graph.

$CLIQUE_{n,k}$ is 1 on a graph if and only if the graph has a clique of size k. Clearly, $CLIQUE_{n,k}$ is a monotone function, because when our ordering is interpreted as the containment ordering between the sets of edges in a graph, then if a graph G contains a clique of size k, any other graph containing the edges of G will also contain the same clique.

Theorem 22.3 *A function $f : \{0,1\}^n \to \{0,1\}$ is monotone if and only if it can be computed by a monotone circuit.*

Proof:

- ($\Longrightarrow$) We will build a monotone circuit that computes f: For every α s.t. $f(\alpha) = 1$ we define:

$$\phi_\alpha(x) = \bigwedge_{\alpha_i = 1} x_i$$

We also define:

$$\phi(x) = \bigvee_{f(\alpha)=1} \phi_\alpha(x)$$

It is clear that ϕ can be realized as a monotone circuit. Now we claim that $\phi = f$.

1. For every α s.t. $f(\alpha) = 1$, we have $\phi_\alpha(\alpha) = 1$ and therefore $\phi(\alpha) = 1$.
2. If $\phi(x) = 1$, then there is an α s.t., $\phi_\alpha(x) = 1$ and thereby $f(\alpha) = 1$. The fact that $\phi_\alpha(x) = 1$ means that $x \geq \alpha$ by the definition of ϕ_α. Now, from the monotonicity of f we conclude that $f(x) \geq f(\alpha) = 1$, meaning $f(x) = 1$.

- ($\Longleftarrow$) The functions AND and OR and the projection function $p_i(x_1, \ldots, x_n) = x_i$ are all monotone. We will now show that composition of monotone functions forms a monotone function, and therefore conclude that every monotone circuit computes a monotone function.

Let $g : \{0,1\}^n \to \{0,1\}$ be a monotone function. Let $f_1, \ldots, f_n : \{0,1\}^N \to \{0,1\}$ be also monotone. We claim that $G : \{0,1\}^N \to \{0,1\}$ define by:

$$G(x) = g(f_1(x), \ldots, f_n(x))$$

is also monotone.

If $x \geq y$ then from the monotonicity of $f_1, \ldots, f_n$, we have that for all i: $f_i(x) \geq f_i(y)$. In other words:

$$(f_1(x), \ldots f_n(x)) \geq (f_1(y), \ldots f_n(y))$$

Now, from the monotonicity of g, we have:

$$G(x) = g(f_1(x), \ldots, f_n(x)) \geq g(f_1(y), \ldots, f_n(y)) = G(y)$$

■

We make analogous definitions for complexity in monotone circuits:

Definition 22.4 ($Mon\text{-}Size$, $Mon\text{-}Depth$): *Given a monotone function $f : \{0,1\}^n \to \{0,1\}$, we define:*

1. $Mon\text{-}Size(f) \stackrel{\text{def}}{=}$ *The minimum size of a monotone circuit computing f.*
2. $Mon\text{-}Depth(f) \stackrel{\text{def}}{=}$ *The minimum depth of a monotone circuit computing f.*

Obviously for every monotone function f, $Mon\text{-}Size(f) \geq Size(f)$, and $Mon\text{-}Depth(f) \geq Depth(f)$. In fact there are functions for which these inequalities are strict. We will not prove this result here.

Unlike the general circuit model, several lower bounds were proved for the monotone case. For example, it is known that for large enough n and specific k (depending on n):

$$\begin{aligned} Mon\text{-}Size(CLIQUE_{n,k}) &= \Omega(2^{\frac{1}{3}n}) \\ Mon\text{-}Depth(CLIQUE_{n,k}) &= \Omega(n) \end{aligned}$$

¿From now on we shall concentrate on proving a lower bound on *st-Connectivity*:

Definition 22.5 (*st-Connectivity*): *Given a directed graph G on n nodes, two of which are marked as s and t, st-Connectivity$(G) = 1$ if and only if there is a directed path from s to t in G.*

Obviously *st-Connectivity* is a monotone function since if we add edges we cannot disconnect an existing path from s to t.

Theorem 22.6 $Mon\text{-}Depth(st\text{-}Connectivity) = \Theta(\log^2(n))$

In a previous lecture, we proved that *st-Connectivity* is in NC_2. This we proved by constructing a circuit that performs $O(\log(n))$ successive boolean matrix multiplications. Notice that the operation of multiplying boolean matrices is a monotone operation (it uses only AND and OR gates). Therefore, the circuit constructed for *st-Connectivity* is actually monotone. If we define $Mon\text{-}NC_i$ to be the natural monotone analog of NC_i, then *st-Connectivity* is in $Mon\text{-}NC_2$. Also, from the above theorem *st-Connectivity* is not in $Mon\text{-}NC_1$. This gives us:

Corollary 22.7 $Mon\text{-}NC_1 \neq Mon\text{-}NC_2$

An analogous result in the non-monotone case is believed to be true, yet no proof is known.

We will proceed by reducing the question of monotone depth to a question in communication complexity.

22.3 Communication Complexity and Circuit Depth

There is an interesting connection between circuit depth and communication complexity which will assist us when proving our main theorem. Since the connection itself is interesting, we will prove it for general circuits. First some definitions:

Definition 22.8 *Given $f : \{0,1\}^n \to \{0,1\}$ we define a communication game G_f:*

- *Player 1 gets $x \in \{0,1\}^n$, s.t. $f(x) = 1$.*
- *Player 2 gets $y \in \{0,1\}^n$, s.t. $f(y) = 0$.*

Their goal is to find a coordinate i s.t. $x_i \neq y_i$.

Notice that this game is not exactly a communication game in the sense we defined in the previous lecture, since the two players do not compute a function, but rather a relation.

We denote the communication complexity of a game G by $CC(G)$. The connection between our complexity measures is:

Lemma 22.3.1 $CC(G_f) = Depth(f)$

Proof:

1. First we'll show $CC(G_f) \leq Depth(f)$. Given a circuit C that calculates f, we will describe a protocol for the game G_f. The proof will proceed by induction on the depth of the circuit C.
   - **base case:** $Depth(f) = 0$. In this case, f is simply the function x_i or $\neg x_i$, for some i. Therefore there is no need for communication, since i is a coordinate in which x and y always differ.

- **Induction step:** We look at the top gate of C: Assume $C = C_1 \wedge C_2$, then

$$Depth(C) = 1 + \max\{Depth(C_1), Depth(C_2)\}$$
$$\Downarrow$$
$$Depth(C_1), Depth(C_2) \leq Depth(C) - 1$$

Denote by f_1 and f_2 the functions that C_1 and C_2 calculate respectively. By the induction hypothesis:

$$CC(G_{f_1}), CC(G_{f_2}) \leq Depth(C) - 1$$

We know that $f(x) = 1$ and $f(y) = 0$, therefore:

$$f_1(x) = f_2(x) = 1$$
$$f_1(y) = 0 \text{ or } f_2(y) = 0$$

Now, as the first step in the protocol, player 2 sends a bit specifying which of the functions f_1 or f_2 is zero on y. Assume player 2 sent 1. In this case they both know:

$$f_1(y) = 0$$
$$f_1(x) = 1$$

And now the game has turned into the game G_{f_1}. This we can solve (using our induction hypothesis) with communication complexity $CC(G_{f_1}) \leq Depth(f_1)$. If player 2 sent 2 we would use the protocol for G_{f_2}. We needed just one more bit of communication. Therefore our protocol will have communication complexity of:

$$\begin{aligned} CC(G_f) &\leq 1 + \max\{CC(G_{f_1}), CC(G_{f_2})\} \\ &\leq 1 + \max\{Depth(f_1), Depth(f_2)\} \\ &= 1 + (Depth(f) - 1) \qquad = Depth(f) \end{aligned}$$

We proved this for the case where $C = C_1 \wedge C_2$. The case where $C = C_1 \vee C_2$ is proved in the same way, expect player 1 is the one to send the first bit (indicating if $f_1(x) = 1$ or $f_2(x) = 1$).

2. Now we'll show the other direction: $CC(G_f) \geq Depth(f)$. For this we'll define a more general sort of communication game based on two non-intersecting sets: $A, B \subseteq \{0,1\}^n$:

- Player 1 gets $x \in A$
- Player 2 gets $y \in B$
- Their goal is to find a coordinate i s.t. $x_i \neq y_i$.

We'll denote this game by $G_{A,B}$. Using the new definition G_f equals $G_{f^{-1}(1),f^{-1}(0)}$. We will prove the following claim:

Claim 22.3.2 *If $CC(G_{A,B}) = d$ then there is a function $f : \{0,1\}^n \to \{0,1\}$ that satisfies:*

- $f(A) = 1$ *(i.e., $f(x) = 1$ for every $x \in A$).*
- $f(B) = 0$
- $Depth(f) \leq d$

In the case of G_f, the function we get by the claim must be f itself, and we get that it satisfies $Depth(f) \leq CC(G_f)$, proving our lemma.

Proof: *(claim)* By induction on $d = CC(G_{A,B})$

- **Base case:** $d = 0$, meaning there is no communication, so there is a coordinate i in which all of A is different then all of B, and so the function $f(\alpha) = \alpha_i$ or the function $f(\alpha) = \neg\alpha_i$ will satisfy the requirements depending on whether the coordinate i is 1 or 0 in A.
- **Induction step:** We have a protocol for the game $G_{A,B}$ of communication complexity d. First assume player 1 sends the first bit in the protocol. This bit partitions the set A into two disjoint sets $A = A_0 \cup A_1$, or in other words, this bit turns our game into one of the following games (depending on the bit sent): $G_{A_0,B}$ or $G_{A_1,B}$. Each one of these has communication complexity of at most $d-1$ simply by continuing the protocol of $G_{A,B}$ after the first bit is already sent. Now, by the induction hypothesis we have two functions f_0 and f_1 that satisfy:
  - $f_0(A_0) = 1$ and $f_1(A_1) = 1$.
  - $f_0(B) = f_1(B) = 0$
  - $Depth(f_0), Depth(f_1) \leq d - 1$

  We define $f = f_0 \vee f_1$. Then:
  - $f(A) = f_0(A) \vee f_1(A) = 1$, because f_0 is 1 on A_0 and f_1 is 1 on A_1.
  - $f(B) = f_0(B) \vee f_1(B) = 0$
  - $Depth(f) = 1 + \max\{Depth(f_0), Depth(f_1)\} \leq d$

  So f is exactly what we wanted.

  If player 2 sends the first bit, he partitions B into two disjoint sets $B = B_0 \cup B_1$, and turns the game into G_{A,B_0} or G_{A,B_1}. By the induction hypothesis we have two functions corresponding to the two games, g_0 and g_1, so that:

$$g_0(A) = g_1(A) = 1$$
$$g_0(B_0) = g_1(B_1) = 0$$

  We define $g \stackrel{\text{def}}{=} g_0 \wedge g_1$. This g satisfies:
  - $g(A) = g_0(A) \wedge g_1(A) = 1$.
  - $g(B) = g_0(B) \wedge g_1(B) = 0$ (because g_0 is 0 on B_0, and g_1 is 0 on B_1).

■

■

22.4 The Monotone Case

Let us remember that our goal was to prove tight bounds on the monotone depth of *st-Connectivity*. Therefore we will define an analogue game for monotone functions, that will give us a lemma of the same flavor as the last.

22.4.1 The Analogous Game and Connection

Definition 22.9 (Monotone game): *Given a monotone $f : \{0,1\}^n \to \{0,1\}$ we define a communication game M_f:*

- *Player 1 gets $x \in \{0,1\}^n$, s.t. $f(x) = 1$.*
- *Player 2 gets $y \in \{0,1\}^n$, s.t. $f(y) = 0$.*

Their goal is to find a coordinate i s.t. $x_i > y_i$, i.e. $x_i = 1$ and $y_i = 0$ We denote this kind of a game a monotone game.

The game is exactly the same as G_f, except f is monotone, and the goal is more specific; i.e., the goal is to a find a coordinate i where not only $x_i \neq y_i$ but also $x_i > y_i$. Notice that the goal is always achievable, because if there is no such i, then y is at least as large as x in every coordinate. This means that $y \geq x$, but this contradicts the fact that f is monotone and $f(x) = 1$, $f(y) = 0$.

Our corresponding lemma for the monotone case is:

Lemma 22.4.1

$$CC(M_f) = Mon\text{-}Depth(f)$$

Proof: The proof is similar to the non-monotone case:

1. When building the protocol from a given circuit:
   - **Base case:** since f is monotone, if the depth is 0, we have that $f(\alpha) = \alpha_i$ and therefore it must be the case that $x_i = 1$ and $y_i = 0$. Hence, again there is no need for communication, and the answer is i (after all $x_i > y_i$).
   - **Induction step:** In the induction step, the top gate separates our circuit into two sub-circuits. The protocol then uses one communication bit to decide which of the two games corresponding to the two sub-circuits to solve. Since the sub-circuits are monotone, by the induction hypothesis they each have a protocol to solve their matching monotone game. This solves the monotone game corresponding to the whole circuit, since the sub-games are monotone, and therefore the coordinate i found satisfies $x_i > y_i$.
2. When building the circuit from a given protocol:
   - **Base case:** if there is no communication, both players already know a coordinate i in which $x_i > y_i$, hence our circuit would simply be $f(\alpha) = \alpha_i$, which is monotone and of depth 0.
   - **Induction step:** Each communication bit splits our game into two sub-games of smaller communication. Notice that if the original game was a monotone game, so are the two sub-games. By the induction hypothesis, the circuits for these games are monotone. Now, since we only add AND and OR gates, the circuit built is monotone.

■

22.4.2 An Equivalent Restricted Game

Let us define a more restricted game than the one in Definition 22.9, that will be easier to work with. First some definitions regarding monotone functions:

Definition 22.10 (minterm, maxterm): *Let $f : \{0,1\}^n \to \{0,1\}$ be a monotone function.*

- *A* minterm *of f is $x \in \{0,1\}^n$ s.t. $f(x) = 1$ and for every $x' < x$ we have $f(x') = 0$.*
- *A* maxterm *of f is $y \in \{0,1\}^n$ s.t. $f(y) = 0$ and for every $y' > y$ we have $f(y') = 1$.*

For example, for *st-Connectivity*:

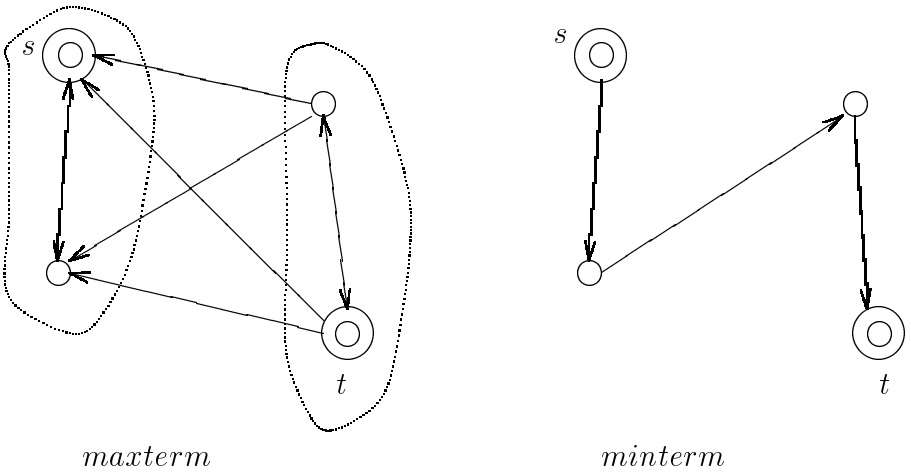

Figure 22.1: A maxterm and minterm example for *st-Connectivity*.

- The set of minterms is the set of graphs that contain only a simple (does not intersect itself) directed path from s to t:

  1. If a graph G is a minterm, then it must contain a simple path from s to t, and it cannot contain any other edge. This is because *st-Connectivity*$(G) = 1$, therefore there is a simple path P from s to t in G. G cannot contain any other edges, because then $P < G$ (in the edge containment order), but *st-Connectivity*$(P) = 1$, contradicting the fact that G is a minterm.
  2. Every G that is a simple path from s to t is a minterm, because *st-Connectivity*$(G) = 1$ and every edge we drop will disconnect s from t, therefore it is minimal.

- The set of maxterms for *st-Connectivity* is the set of graphs G s.t. G's set of vertices can be partitioned into two disjoint parts S and T that satisfy:

  1. $s \in S$ and $t \in T$.
  2. G contains all possible directed edges except those from S to T

  This is indeed the set of maxterms for *st-Connectivity*:

  1. If G is a maxterm then let S be the set of vertices that are reachable from s in G. Set T to be all other vertices. $t \in T$ because one cannot reach t from s in G, since

st-Connectivity$(G) = 0$. Also, G must contain all edges except those from S to T, otherwise we can add the missing edges and still leave t unconnected from s. There are no edges from S to T by the definition of S as the connected component reachable from s.

2. If G satisfies both criteria, then every path starting from s in G will remain in S and therefore will not reach t so *st-Connectivity*$(G) = 0$. Every edge we add to G will connect S to T and since S and T are strongly connected it will create a path between s and t.

Another way to view a maxterm of *st-Connectivity*, is that the partition is defined by a coloring of the vertices by two colors 0 and 1, where s is colored 0 and t is colored 1. The set of vertices colored 0 is S, and those colored 1 are T.

We will now use maxterms and minterms to define a new communication game:

Definition 22.11 ($\hat{M}_f$): *Given a monotone $f : \{0,1\}^n \to \{0,1\}$ we define a communication game $\hat{M}_f$:*

- *Player 1 gets $x \in \{0,1\}^n$, s.t. x is a minterm of f (in particular $f(x) = 1$).*
- *Player 2 gets $y \in \{0,1\}^n$, s.t. y is a maxterm of f (in particular $f(y) = 0$).*

Their goal is to find a coordinate i s.t. $x_i > y_i$, i.e. $x_i = 1$ and $y_i = 0$.

Notice that $\hat{M}_f$ is a restriction of M_f to a smaller set of inputs, therefore the protocol that will solve M_f will also solve $\hat{M}_f$. Hence $CC(\hat{M}_f) \leq CC(M_f)$. In fact, the communication complexity of the two games is exactly the same:

Proposition 22.4.2 $CC(\hat{M}_f) = CC(M_f)$

Proof: What is left to prove is that: $CC(\hat{M}_f) \geq CC(M_f)$. Given a protocol for $\hat{M}_f$ we construct a protocol for M_f of the same communication complexity.

1. Player 1 has x s.t. $f(x) = 1$. He now finds a minimal x' s.t. $x' \leq x$ but $f(x') = 1$. This is done by successively changing coordinates in x from 1 to 0, while checking that $f(x')$ still equals 1. This way, eventually, he will get x' that is a minterm.

2. In the same manner player 2 finds a maxterm $y' \geq y$.

The players now proceed according to the protocol for $\hat{M}_f$ on inputs x' and y'. Since x' is a minterm, and y' is a maxterm, the protocol will give a coordinate i in which:

$$\begin{array}{lll} x'_i = 1 & \Longrightarrow & x_i = 1 \quad \text{because } x' \leq x \\ y'_i = 0 & \Longrightarrow & y_i = 0 \quad \text{because } y' \geq y \end{array}$$

The communication complexity is exactly the same, since we used the same protocol except for a preprocessing stage that does not cost us in communication bits. ∎

Combining our last results we get:

Corollary 22.12 *Given a monotone function $f : \{0,1\}^n \to \{0,1\}$:*

$$Mon\text{-}Depth(f) = CC(\hat{M}_f)$$

22.5 Two More Games

As we have seen, when examining bounds on the monotone depth of *st-Connectivity*, we need only examine the communication complexity of the following game denoted KW (for Karchmer and Wigderson) which is simply a different formulation of $\hat{M}_{st\text{-}Connectivity}$:

Given n nodes and two special nodes s and t,

- Player 1 gets a directed path from s to t.
- Player 2 gets a coloring C of the nodes by 0 and 1, s.t. $C(s) = 0$ and $C(t) = 1$.
- The goal is to find an edge (v, w) on player 1's path s.t. $C(v) = 0$ and $C(w) = 1$.

First we will use this formulation to show an $O(\log^2(n))$ upper bound on $Mon\text{-}Depth(st\text{-}Connectivity)$ using a protocol for KW with communication complexity $O(\log^2(n))$:

Proposition 22.5.1 $CC(KW) = O(\log^2(n))$

Proof: The protocol will simulate binary search on the input path of player 1. In each step, we reduce the length of the path by a factor of 2, while keeping the invariant that the color of the first vertex in the path is 0, and the color of the last is 1. This is of course true in the beginning since $C(s) = 0$ and $C(t) = 1$.

The base case, is that the path has only one edge, and in this case we are done, since our invariant guarantees that this edge is colored as we want. Now, player 1 sends player 2 this edge. The communication cost is $O(\log(n))$.

If the path is longer, player 1 asks player 2 the color of the middle vertex in the path. This costs $\log(n) + 1$ bits of communication — the name of the middle vertex sent from player 1 to player 2 takes $\log(n)$ bits, and player 2's answer costs one more bit. If the color is 1, the first half of the path satisfies our invariant, since the first vertex is colored 0, and now the last will be colored 1. If the color is 0, we take the second half of the path. In any case, we cut the length of the path by 2 with communication cost $O(\log(n))$.

Since the length of the original path is at most n, we need $O(\log(n))$ steps until we reach a path of length 1. All in all, we have communication complexity of $O(\log^2(n))$. ■

We will now direct our efforts towards proving a lower bound of $\Omega(\log^2(n))$ for $Depth(st\text{-}Connectivity)$ via a lower bound for KW. For this we will continue to yet another reduction into a different communication game called $FORK$:

Definition 22.13 ($FORK$): *Given $n = l \cdot w$ vertices and three special vertices s, t_1, and t_2, where the n vertices are partitioned into l layers $L_1, \ldots, L_l$, and each layer contains w vertices:*

- *Player 1 gets a sequence of vertices $(x_0, x_1, \ldots, x_l, x_{l+1})$, where for all $1 \leq i \leq l$: $x_i \in L_i$, and $x_0 = s$, $x_{l+1} = t_1$.*
- *Player 2 gets a sequence of vertices $(y_0, y_1, \ldots, y_l, y_{l+1})$, where for all $1 \leq i \leq l$: $y_i \in L_i$, and $y_0 = s$, $y_{l+1} = t_2$.*
- *Their goal is to find an i such that $x_i = y_i$ and $x_{i+1} \neq y_{i+1}$.*

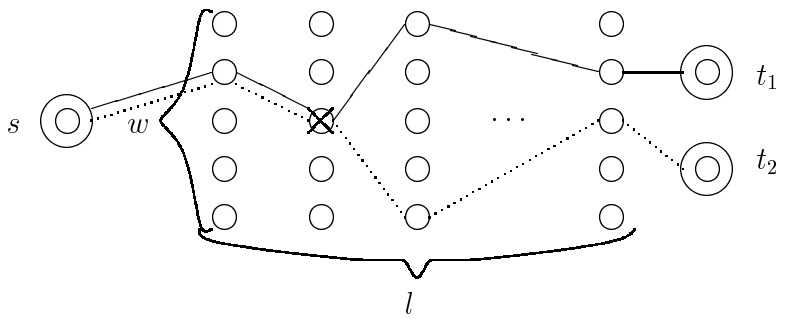

Figure 22.2: Player 1's sequence is solid, player 2's is dotted, and the fork point is marked with an x.

Obviously, such an i always exists, since the sequences start at the same vertex (s), and end in different vertices ($t_1 \neq t_2$), therefore there must be a fork point.

Note: The sequences the players get can be thought of as an element in $\{1, \ldots, w\}^l$. Since the start vertex is set to be s, and the end vertices for both players are also set to be t_1 and t_2 (depending on the player).

This game is somewhat easier to deal with then KW, because of the symmetry between the players. We will show that this new game needs no more communication than KW , and therefore, proving a lower bound on its communication complexity suffices.

Proposition 22.5.2 $CC(FORK) \leq CC(KW)$

Proof: Assuming we have a protocol for KW, we will show a protocol for $FORK$ which uses the same amount of communication. Actually, as in the proof of Proposition 22.4.2, all that the players have to do is some preprocessing, and the protocol itself does not change.

Recall that in the game KW player 1 has a directed path between two special vertices s and t, that goes through a set of regular vertices. Player 2 has a coloring of all vertices by 0 and 1, where s is colored 0, and t is colored 1.

To use the protocol for KW, the players need to turn their instance of $FORK$ into an instance of KW.

- We define the vertex s in $FORK$ to be s in KW.
- We define the vertex t_1 to be t.
- All other vertices are regular vertices.
- The path of player 1 remains exactly the same — it is indeed between s and $t(= t_1)$.
- The coloring of player 2 is: a vertex is colored 0 if and only if it is in his input path of vertices — note that s is colored 0, since it is the first vertex in his sequence, and $t(= t_1)$ is colored 1 because it is not on this path (which goes from s to t_2).

After this preprocessing we use the protocol for KW to get an edge (u, v) that is on player 1's path, where u is colored 0, and v is colored 1. This means, that u is on player 2's path, because it is colored 0, and v is not, because it is colored 1.

Hence, u is exactly the kind of fork point we were looking for, since it's in both players path and its successor is different in the two paths. ■

In the next lecture we will prove that

$$CC(FORK) = \Omega(\log(l)\log(w))$$

Setting $l = w = \sqrt{n}$, our main theorem follows:

$$CC(st\text{-}Connectivity) = \Theta(\log^2(n))$$

Bibliographic Notes

This lecture is based mainly on [2]. Specifically, the connection between circuit depth and the corresponding communication complexity problem was made there. The current analysis of the communication complexity problem corresponding to s-t-Connectivity, via reduction to the game FORK, is due to [1] and is simpler than the original analysis (as in [2]).

1. M. Gringi and M. Sipser. Monotone Separation of Logarithmic Space from Logarithmic Depth. *JCSS*, Vol. 50, pages 433–437, 1995.

2. M. Karchmer and A. Wigderson. Monotone Circuits for Connectivity Require Super-Logarithmic Depth. *SIAM J. on Disc. Math.*, Vol. 3, No. 2, pages 255–265, 1990.

Lecture 23

The FORK Game

Lecture given by Ran Raz
Notes taken by Dana Fisman and Nir Piterman

Summary: We analyze the game FORK that was introduced in the previous lecture. We give tight lower and upper bounds on the communication needed in a protocol solving FORK. This completes the proof of the lower bound on the depth of monotone circuits computing the function st-Connectivity.

23.1 Introduction

We have seen in the previous lecture the connection between circuit depth and communication complexity: We compared the depth of a circuit computing a function f to the number of bits transferred between two players playing a communication game G_f. We saw that given a communication protocol solving G_f using communication of c-bits, we can construc t a circuit computing f whose depth is c. On the other hand, given a c-depth circuit computing f we can plan a communication protocol for G_f which uses c bits. Thus, we established that the best communication protocol for solving G_f uses t he same amount of communication bits as the depth of the best circuit computing f. tion protocol. The plan was to use communication complexity to prove upper and lower bounds on the depths of circuits. Failing to reach satisfactory results using this model, the weaker model of monotone functions and *monotone* circuits was introduced. Since a mono tone circuit is in particular a circuit but not the other way around, proving lower bounds on monotone circuits does not prove the same lower bounds in the unrestricted model. All the same, our goal was to show a (tight) lower bound for the depth of monot one circuits computing st-Connectivity. The plan was to achieve the result by conducting a series of reductions. The first used the connection between circuit depth and communication complexity to reduce the question to a communication protocol. Then, various reductions between several kinds of games led us to the FORK game (see Definition 23.1). Based on a lower bound for the communication needed in FORK, the lower bound for st-Connectivity of $\Omega(log^2 n)$ was proven. The purpose of this lecture is to prove the lower bound of the FORK game. We will give a complete analysis of the FORK game showing that the communication between the two players is $\Theta(log_2 w \cdot log_2 l)$, thus supplying the missing link in the proof of the lower bound on the depth of st-Connectivity.

23.2 The FORK game – recalling the definition

FORK is a game between two players. Each player gets a path from a predefined set of possible paths. Both paths start at the same point but have different end points. Hence, at least one fork point in which the two paths separate, must exist. The players' goal is to find such a fork point. More formally, we recall the definition of the FORK game given in previous lecture.

Definition 23.1 (FORK): *Given $n = l \cdot w$ vertices and three special vertices s, t_1, and t_2, where the n vertices are divided into l layers $l_1, \ldots, l_l$, and each layer contains w vertices:*

- Player I *gets a sequence of vertices $(x_1, x_2, \ldots, x_l)$, where for all $1 \leq i \leq l$: $x_i \in l_i$. For simplicity of notation we assume that* Player I *is given two more coordinates: $x_0 = s$, $x_{l+1} = t_1$.*
- Player II *gets a sequence of vertices $(y_1, y_2, \ldots, y_l)$, where for all $1 \leq i \leq l$: $y_i \in l_i$. Again we consider $y_0 = s$, $y_{l+1} = t_2$.*
- *Their goal is to find a coordinate i such that $x_i = y_i$ and $x_{i+1} \neq y_{i+1}$.*

In order to stress the fact that the inputs are elements of $[w]^l$, where $[w] = \{1, ..., w\}$, we slightly modified the definition, excluding the constant points s, t_1 and t_2 from the input sequences given to the players.

The following theorem states the main result of this lecture, giving the bounds on the communication complexity of solving the FORK game.

Theorem 23.2 *The communication complexity of the* FORK *game is $\Theta(log_2 w \cdot log_2 l)$.*

We first show an upper bound on the communication complexity of the game. The upper bound is given here only for the completeness of the discussion concerning the FORK problem.

23.3 An upper bound for the FORK game

The proof given here is very similar to the proof of the upper bound for the KW game (see previous lecture). We basically perform a binary search on the path to find the FORK point.

Proposition 23.3.1 *The communication complexity of the* FORK *game is $O(log_2 w \cdot log_2 l)$.*

Proof: For the sake of simplicity, the following notation is introduced. We denote $F_{a,b}$ as the FORK game where the inputs are of the form $\overline{x} = (x_a, x_{a+1}, ..., x_b)$ and $\overline{y} = (y_a, y_{a+1}, ..., y_b)$. Like in the general FORK gam e, we consider $\overline{x}$ and $\overline{y}$ as having two more coordinates. An $(a-1)$-coordinate such that $x_{a-1} = y_{a-1} = s$ is the origin point of the paths. A $(b+1)$-coordinate such that $x_{b+1} = t_1$ and $y_{b+1} = t_2$ are the endpoints of the paths.

First notice that if the length of the paths is only one, i.e., $a = b$, then the problem can be solved using $log_2 w$ bits, obeying the following protocol:

- Player I sends its input (this requires $log_2 w$ bits).
- Player II replies with 1 if he has the same coordinate and with 0 otherwise (one bit).

If they have the same coordinate, the fork point is found in that coordinate and then the paths separate to t_1 and t_2. Otherwise the fork is the point of origin $s = x_{a-1} = y_{a-1}$.

If the length of the paths is larger than 1, i.e., $a < b$, then the problem can be reduced to half using $log_2 w$ bits. (For simplicity, we assume that $\frac{a+b}{2}$ is an integer):

- Player I sends its middle layer node: $x_{\frac{(a+b)}{2}}$ (this requires $log_2 w$ bits).
- Player II checks if they have the same middle node: $y_{\frac{a+b}{2}} = x_{\frac{a+b}{2}}$.

  If so he sends 1 otherwise he sends 0 (one bit).

If the players have the same middle point (i.e., Player I sent 1), then there has to be a fork point between the middle and the end. Thus the game is reduced to $F_{\frac{a+b}{2}+1,b}$. On the other hand, if the middle points differ, then there has to be a fork point between the start and the middle. Thus the game is reduced to $F_{a,\frac{a+b}{2}-1}$. Note that there is no point in including the middle layer itself in the range of the reduced game. The reason being that in the first case, the mutual point in layer $\frac{a+b}{2}$ is actually the new origin point, while in the second case, the two points in layer $\frac{a+b}{2}$ are actually the new end points.

Therefore FORK (i.e. $F_{1,l}$ using this notation) is solved in $log_2 l$ iterations of the protocol, requiring transmission of $O(log_2 w \cdot log_2 l)$ bits. ■

Our goal now is to show that this upper bound is tight.

23.4 A lower bound for the FORK game

In order to show the lower bound we consider games that work only for a subset of the possible inputs. We perform some inductive process in which we establish the connection between games with different sets. Two kinds of transformations are considered:

1. Given a protocol that works for some set, we devise a protocol that works for smaller density sets with one less bit of communication.
2. Given a protocol that works for some set, we convert it into a protocol that works for sets of higher density but shorter paths using the same amount of communication.

When adapting the given protocol into a new one, some heavy computations are involved. However, recall that the only parameter considered during the application of the protocol is the number of bits transmitted. Thus, any computation done in the preparati on of the protocol and any computation done locally by either side is not taken into account.

23.4.1 Definitions

We first define a subgame of FORK that works only on a subset of the possible inputs. Given a subset $S \subseteq [w]^l$ we consider the game FORK^S where Player I gets an input $\overline{x} \in S$ and Player II gets an input $\overline{e}rliney \in S$ and they have the same goal of finding the fork point between the two paths.

We will work only with subsets of $[w]^j$, where the power j (the length of the path) will change from time to time. We call w the width of the game. Throughout the discussion the width w of the game will be constant and we frequently ignore i t.

As explained above, the density of the set will be an important parameter in the transformations:

Definition 23.3 (density): *The* density *of a set* $S \subseteq [w]^l$ *is defined as* $\alpha(S) = \frac{|S|}{w^l}$.

Given a protocol solving a partial FORK game, we ask ourselves what is the density of the sets that this protocol works on. We are interested in the minimal protocol that will work for some set of density α:

Definition 23.4 *($(\alpha, l) - protocol$): A communication protocol will be called an* (α, l)-protocol *if it works for some set S of paths of length l with density $\alpha(S) = \alpha$.*

Definition 23.5 *($CC(\alpha, l)$): Let $CC(\alpha, l)$ denote the smallest communication complexity of an $(\alpha, l) - protocol$.*

Using this terminology, since there is just one set of density 1, a protocol for FORK is just a $(1, l) - protocol$ and so we are interested in $CC(1, l)$.

23.4.2 Reducing the density

The following lemma enables the first kind of transformation discussed above. Given a protocol that works for a set of a certain density we adapt it to a protocol that works for a subset of half that density. The new protocol will work with one less bit o f communication than the original protocol.

Lemma 23.4.1 *If there exists an $(\alpha, l) - protocol$ which works with c bits and $c > 0$, then there is also an $(\frac{\alpha}{2}, l) - protocol$ that works with $c - 1$ bits.*

By the above lemma, the best protocol for an α density set will require at least one more bit than the best protocol for an $\frac{\alpha}{2}$ density set. Thus, we have

Corollary 23.6 *If $CC(\alpha, l)$ is non-zero then $CC(\alpha, l)$ is greater than $CC(\frac{\alpha}{2}, l)$ by at least one;*

$$CC(\alpha, l) \geq CC(\tfrac{\alpha}{2}, l) + 1$$

Proof: The proof can be generalized for any communication protocol. As explained in detail in Lecture 21, we use the fact that any communication protocol can be viewed as the two parties observing a matrix. The matrix' rows correspond to the possible inputs of Player I and its columns correspond to the possible inputs of Player II. The entries of the matrix are the desired results of the protocol. Passing one bit between the two players partitions this matrix into two rectangles (horizontally if the first bit is sent by Player I and vertically if it is sent by Player II).

Let P be a non-zero communication $(\alpha, l) - protocol$ for the set S. Assume without loss of generality that Player I sends the first bit in the protocol P. Let $S_0 \subseteq S$ be those inputs in S for which Player I sends 0 as the f irst bit, and similarly let $S_1 \subseteq S$ be those inputs in S for which Player I sends 1 as the first bit. Since $S_0 \cup S_1 = S$, either S_0 or S_1 contains at least half the inputs in S. Assuming that this is S_0 then $\alpha(S_0) \geq \frac{\alpha(S)}{2}$. Let P' be a protocol that works like P but without sending the first bit, while the players assume that the value of this bit is 0. Obviously P' works if Player I gets inputs from S_0 and Player II gets inputs fr om S. In particular it works when both inputs are from the subset S_0. We conclude that P' is an $(\frac{\alpha}{2}, l) - protocol$ that works with one bit less than P.
The case where S_1 is larger than S_0 is identical. ■

In order to apply Lemma 23.4.1 the protocol must cause at least one bit to be sent by some player. Therefore, we would like to identify the sets for which a protocol consumes some communication. We show that for large enough sets, whose density is more than $\frac{1}{w}$, any protocol indeed uses non-zero communication.

Lemma 23.4.2 *Any protocol for a set whose density is larger than $\frac{1}{w}$ requires communication of at least one bit.*

Corollary 23.7 *For every l and every $\alpha > \frac{1}{w}$, $CC(\alpha, l) > 0$.*

Proof: We will show that if P is a protocol that works with no communication at all and solves FORK^S then the density of S is at most $\frac{1}{w}$.

Since no information is passed between the players, any of the players must establish his result depending only on his input. It could be the case where both players get the same path $\overline{x} = \overline{y}$. In this case the only fork point is in the last layer $x_l = y_l$ (with $x_{l+1} = t_1 \neq t_2 = y_{l+1}$). So Player I must always give the last layer as the fork point. If some $\overline{y} \in S$ has a different point in the last layer (i.e. $y_l \neq x_l$), the protocol will give a wrong answer. Hence for all the paths in S the last point in the path has to be j for some j between 1 and w. So the set S is in fact a subset of $[w]^{l-1} \times \{j\}$ and therefore its density is not greater than $\frac{w^{l-1}}{w^l} = \frac{1}{w}$. ■

Note that using these two lemmas we can get a lower bound of $\Omega(log_2 w)$ bits for FORK. As long as the density is greater than $\frac{1}{w}$, Lemma 23.4.2 guarantees that we can apply Lemma 23.4.1. Repeating Lemma 23.4.1 less than $log_2 w$ times we know that the density has not decreased beyond $\frac{1}{w}$ and so Lemma 23.4.1 can be applied again.

$$CC(1, l) \geq CC(\tfrac{1}{2}, l) + 1 \geq CC(\tfrac{1}{4}, l) + 2 \geq ... \geq CC(\tfrac{1}{w}, l) + log_2 w$$

Considering our aim is to use this bound in connection with boolean circuits, the bound $CC(1, l) = \Omega(log_2 w)$ is insignificant. This is the case, since in order to read an input of length w we must use a circuit of at least $log_2 w$ depth anyhow.

23.4.3 Reducing the length

In order to reach the desired bound we need to manipulate the length of the paths as well. The main tool will be the following 'amplification' lemma that allows us, using an $(\alpha, l) - protocol$, to construct another protocol that works on a set of short er paths (of length $\frac{l}{2}$) but whose density is larger than α.

Lemma 23.4.3 *Let $\alpha \geq \frac{12}{w}$. If there exists an $(\alpha, l) - protocol$ for FORK^S, that uses c bits of communication, then there is also an $(\frac{\sqrt{\alpha}}{2}, \frac{l}{2}) - protocol$ that uses the same number of bits.*

For α's in the range $\frac{12}{w} < \alpha < \frac{1}{4}$ using this lemma increases the density of the set. Since:

$$\tfrac{12}{w} < \alpha < \tfrac{1}{4} \implies \tfrac{\sqrt{\alpha}}{2} > \alpha$$

The proof of the lemma uses the following technical claim:

Claim 23.4.4 *Consider an $n \times n$ matrix of 0-1. Denote by α the fraction of one-entries in the matrix and by α_i the fraction of one-entries in the i^{th} row. We say that a row i is* dense *if $\alpha_i \geq \frac{\alpha}{2}$. One of the following two cases hold:*

1. *there is some row i with $\alpha_i \geq \sqrt{\frac{\alpha}{2}}$*
2. *the number of dense rows is at least $\sqrt{\frac{\alpha}{2}} \cdot n$*

Proof: Intuitively, the claim says that either one of the rows is 'very dense' or there are a lot of dense rows.

Suppose by contradiction that the two cases do not hold. Let us calculate the density of one-entries in the entire matrix. Since Case 2 does not hold there are less than $\sqrt{\frac{\alpha}{2}} \cdot n$ dense rows. Since

Case 1 does not hold each of them has less than $\sqrt{\frac{\alpha}{2}} \cdot n$ one-entries. Hence the fraction of one-entries in all the dense rows is less than $\sqrt{\frac{\alpha}{2}} \cdot \sqrt{\frac{\alpha}{2}} = \frac{\alpha}{2}$. Non-dense rows contain less than $\frac{\alpha}{2} \cdot n$ one-entries and there are at most n non-dense rows, hence the fraction of one-entries in the non-dense rows is less than $\frac{\alpha}{2}$. Thus, the total fraction of one-entries is less than $\frac{\alpha}{2} + \frac{\alpha}{2} = \alpha$, contradicting the assumed fraction of one-entries in the matrix. ■

Proof: (Of Lemma 23.4.3)
Given an $(\alpha, l) - protocol$ we would like to show an $(\frac{\sqrt{\alpha}}{2}, \frac{l}{2}) - protocol$.
Let P be an $(\alpha, l) - protocol$. Assume that it works for the set S of paths in $[w]^l$ whose density is $\alpha(S) = \alpha$. We view the paths in S as two concatenated paths of half the length. Given $(s_1, ..., s_l)$ a path in S we denote $(s_1, ..., s_l)$ by $a \circ b$ where $a = (s_1, ..., s_{\frac{l}{2}}),\ b = (s_{\frac{l}{2}+1}, ..., s_l)$.

For any $a \in [w]^{\frac{l}{2}}$ we denote by $Suffix(a)$ the set of possible suffixes b for a, that form a path in S:

$$Suffix(a) = \{b \in [w]^{\frac{l}{2}} \mid a \circ b \in S\}$$

Consider a matrix whose rows and columns correspond to paths in $[w]^{\frac{l}{2}}$. An entry of the matrix (a, b) is 1 if the path $a \circ b$ is in S and 0 otherwise. Thus, the fraction of one-entries in the matrix is α (the density of S). Applying Claim 23.4.4 to the matrix, we get that this matrix satisfies either (1) or (2). Either there exists a prefix of a path in S that has a lot of suffixes, or there exist many prefixes of paths in S that have quite a lot of suffixes. In the first case we use the set of suffixes as the new set for which we build a new protocol, while in the second case we use the set of 'heavy' prefixes as the new set. In both cases we adapt the protocol P to work for the new set of half length paths. Details follow:

1. In case there is a prefix of a path a with at least $\sqrt{\frac{\alpha}{2}} \cdot w^{\frac{l}{2}}$ suffixes, we let $S' = Suffix(a)$ be the set of suffixes of that prefix, and the new protocol P' will work as following:

   Player I gets an input x in S', he concatenates it to a forming the path $a \circ x$ in S. In a similar way Player II forms the path $a \circ y$. Now they have paths in S and can apply the protocol P to find the fork point. Since the paths coincide in their first halves, the fork point must be in the second half.

   Note that if the first coordinate in x and y is different, the fork in the whole path can be found in the last coordinate of a. This is the case where for S' the fork was found in the point of origin of the paths, s.

2. In case there are many (i.e. at least $\sqrt{\frac{\alpha}{2}} \cdot w^{\frac{l}{2}}$) prefixes of paths with at least $\frac{\alpha}{2}$ suffixes we take S' to be the set of all the prefixes of 'dense' paths; that is, $S' = \{x : |Suffix(x)| \geq \frac{\alpha}{2} \cdot w^{\frac{l}{2}}\}$. We have $|S'| \geq \sqrt{\frac{\alpha}{2}} \cdot w^{\frac{l}{2}}$. For each possible input x in S' we will try and build two suffixes $b_1(x)$ and $b_2(x)$ such that for any two inputs x and y the suffixes $b_1(x)$ and $b_2(y)$ will not coincide. In this case a fork found between $x \circ b_1(x)$ and $y \circ b_2(y)$ must be in the first half of the path, since the second half is ensured not to coincide in any point.

   We suggest a method for building the suffixes (for simplicity we assume that w is even):

   For each layer $\frac{l}{2} + 1, ..., l$ we color half the nodes in the layer in orange and the other half in purple. If for every x, the suffix $b_1(x)$ will be colored orange and the suffix $b_2(x)$ will be colored purple the goal will be fulfilled. We wi ll show that such a coloring exists by showing that the probability for such a coloring over all random colorings is positive.

Claim 23.4.5 *There exists a coloring of all nodes such that for most $a \in S'$ there are suffixes $b_1(a)$ and $b_2(a)$ such that:*

- *$a \circ b_1(a) \in S$ and $a \circ b_2(a) \in S$*
- *all nodes in $b_1(a)$ are colored orange*
- *all nodes in $b_2(a)$ are colored purple*

We propose the following way to color the vertices in layers $\frac{l}{2}+1, ..., l$:

- Choose randomly $\frac{w}{2}$ paths $r_1, ..., r_{\frac{w}{2}}$ in $[w]^{\frac{l}{2}}$ and color all the vertices appearing in them in orange.
- In layers where the paths covered less than $\frac{w}{2}$ of the vertices choose randomly orange vertices to achieve $\frac{w}{2}$ vertices colored in orange.
- Color the rest of the vertices in purple.

By symmetry it is apparent that this method for coloring produces the same distribution as uniformly choosing $\frac{w}{2}$ nodes out of the w possible nodes in each layer (coloring them in orange and the rest in purple).

We shall show that for any a in S' there is a high probability that two such suffixes (i.e., one colored orange and one colored purple) can be found. Define S'' as the set of all prefixes a in S' that have two such suffixes. The expected size of S'' will be close to the size of S'. Therefore exists a coloring that induces such a set S'', whose size is very close to the size of S'.

For a path $a \in S'$, since the density of suffixes of a is at least $\frac{\alpha}{2}$, the probability that for some i, the random path r_i (chosen in the above process) is not a suffix of a is at most $(1-\frac{\alpha}{2})$. Since these paths are chosen independently and $\alpha > \frac{12}{w}$, the probability that none of the $\frac{w}{2}$ paths is a suffix of a is at most 0.05. This is because

$$Prob[r_i \notin Suffix(a)] \leq (1-\tfrac{\alpha}{2})$$

$$Prob[\forall i,\ r_i \notin Suffix(a)] \leq (1-\tfrac{\alpha}{2})^{\frac{w}{2}} < (1-\tfrac{6}{w})^{\frac{w}{2}} \approx e^{-3} \approx 0.049$$

There is no difference between the probability that the first suffix (i.e. the one colored orange) does not exist and the probability that the second suffix (i.e. the one colored purple) does not exist. Hence, we can use union bound to determine that the probability of either $b_1(a)$ or $b_2(a)$ not to exist, is smaller than 0.1.

Therefore, the expected size of S'' is at least 0.9 of the size of S'. Formally:

Define for every element a in S' a random variable $X_a = \begin{cases} 1 & \text{if } a \text{ has two such suffixes} \\ 0 & \text{otherwise} \end{cases}$

Obviously $|S''| = \Sigma_{a \in S'} X_a$ hence:

$$E(|S''|) = E(\Sigma_{a \in S'} X_a) = \Sigma_{a \in S'} E(X_a) = |S'| \cdot E(X_a) \geq 0.9 \cdot |S'|$$

So we have a set S'' of density at least $0.9 \cdot \sqrt{\frac{\alpha}{2}} \cdot w^{\frac{l}{2}} > \frac{\sqrt{\alpha}}{2} \cdot w^{\frac{l}{2}}$ such that for any path a of length $\frac{l}{2}$ in S'' we have two suffixes $b_1(a)$ and $b_2(a)$ colored orange and purple, respectively.

Player I will get an input x in $[w]^{\frac{l}{2}}$ and concatenate $b_1(x)$ to it, creating $x \circ b_1(x)$, a path in S. Player II, getting y, will create in a similar way $y \circ b_2(y)$. The two players will run the protocol P and find the desired fork point.

Note that if the last coordinate in x and y is equal, the fork can be found there. This is legitimate in S'' because the end points of the paths t_1 and t_2 are different.

We have seen that in both cases we were able to find subsets of $[w]^{\frac{l}{2}}$ of density at least $\frac{\sqrt{\alpha}}{2}$ for which we adapted the protocol P. Therefore we have proven:

$$CC(\alpha, l) \geq CC(\tfrac{\sqrt{\alpha}}{2}, \tfrac{l}{2})$$

■

23.4.4 Applying the lemmas to get the lower bound

We show that any protocol for solving the FORK game uses at least $\Omega(log_2 w \cdot log_2 l)$ bits of communication by employing Lemmas 23.4.1 and 23.4.3. From Lemma 23.4.2 we know that during all the transformat ions, we do not find a zero-bit protocol. For the simplicity of calculations we assume that l is a power of 2.

We start by reducing FORK to any subset of density $\frac{2}{\sqrt{w}}$.

$$CC(1, l) \geq CC(\tfrac{2}{\sqrt{w}}, l)$$

Using Lemma 23.4.1 by a series of $\Omega(log_2 w)$ transformations (all allowed by Lemma 23.4.2) we get:

$$CC(\tfrac{2}{\sqrt{w}}, l) \geq CC(\tfrac{16}{w}, l) + \Omega(log_2 w)$$

Using Lemma 23.4.3, we have

$$CC(\frac{16}{w}, l) \geq CC(\frac{2}{\sqrt{w}}, \frac{l}{2})$$

and so

$$CC(\frac{2}{\sqrt{w}}, l) \geq CC(\frac{2}{\sqrt{w}}, \frac{l}{2}) + \omega(log_2 w)$$

Iterate the last two steps $log_2 l$ times, getting

$$CC(1, l) = \Omega(log_2 l \cdot log_2 w)$$

We conclude that the communication complexity of the FORK game is $\Theta(log_2 l \cdot log_2 w)$.

Bibliographic Notes

This lecture is based on [1].

1. M. Gringi and M. Sipser. Monotone Separation of Logarithmic Space from Logarithmic Depth. *JCSS*, Vol. 50, pages 433–437, 1995.

Lecture 24

Average Case Complexity

Notes taken by Tzvika Hartman and Hillel Kugler

Summary: We introduce a theory of average case complexity. We define the notion of a distribution function and the classes P-computable and P-samplable of distributions. We prove that P-computable $\subset$ P-samplable (strict containment). The class DistNP, which is the distributional analogue of NP, is defined. We introduce the definition of polynomial on the average and discuss the weaknesses of an alternative definition. The notion of reductions between distributional problems is presented. Finally, we prove the existence of a problem that is complete for DistNP.

24.1 Introduction

Traditionally, in theoretical computer science, the emphasis of the research is on the worst case complexity of problems. However, one may think that the more natural (and practical) way to measure the complexity of a problem is by considering its average case complexity. Many important problems were found to be NP-complete and there is little hope to solve them efficiently in the worst case. In these cases it would be useful if we could develop algorithms that solve them efficiently on the average. This is the main motivation for the theory of average case complexity.

When discussing the average case complexity of a problem we must specify the distribution from which the instances of the problem are taken. It is possible that the same problem is efficiently solvable on the average with respect to one distribution, but hard on the average with respect to another. One may think that it is enough to consider the most natural distribution - the uniform one. However, in many cases it is more realistic to assume settings in which some instances are more probable than others (e.g. some graph problems are most interesting on dense graphs, hence, the uniform distribution is not relevant).

It is interesting to compare the average case complexity theory with cryptography theory, since they deal with similar issues. One difference between the two theories is that in cryptography we deal with problems that are difficult on the average, while in the average case theory we try to find problems that are easy on the average. Another difference is that in cryptography we need problems for which it is possible to generate efficiently instance-solution pairs such that solving the problem given only the instance is hard. In contrast, this property is not required in average case complexity theory.

24.2 Definitions

24.2.1 Distributions

We now introduce the notion of a distribution function. We assume a canonical order of the binary strings (e.g. the standard lexicographic order). The notation $x < y$ means that the string x precedes y in this order and $x - 1$ denotes the immediate predecessor of x.

Definition 24.1 (Probability Distribution Function) A *distribution function* $\mu : \{0,1\}^* \to [0,1]$ is a non-negative and non-decreasing function (i.e., $\mu(0) \geq 0, \mu(x) \leq \mu(y)$ for each $x < y$) from strings to the unit interval $[0,1]$ which converges to one (i.e., $lim_{x\to\infty}\mu(x) = 1$). The *density function* associated with the distribution function μ is denoted by μ' and defined by $\mu'(0) = \mu(0)$ and $\mu'(x) = \mu(x) - \mu(x-1)$ for every $x > 0$.

Clearly, $\mu(x) = \sum_{y \leq x} \mu'(y)$. Notice that we defined a single distribution on all inputs of all sizes, rather than ensembles of finite distributions (each ranging over fixed length strings). This makes the definition robust under different representations. An important example is the uniform distribution function defined by $\mu'_u \stackrel{\text{def}}{=} \frac{1}{|x|^2} \cdot 2^{-|x|}$. This density function converges to some constant different than 1. A minor modification, defining $\mu'_u \stackrel{\text{def}}{=} \frac{1}{|x|\cdot(|x+1|)} \cdot 2^{-|x|}$, settles this problem:

$$\sum_{x\in\{0,1\}^*} \mu'_u(x) = \sum_{x\in\{0,1\}^*} \frac{1}{|x| \cdot |x| + 1} \cdot 2^{-|x|} = \sum_{n\in N} \sum_{x\in\{0,1\}^n} (\frac{1}{n} - \frac{1}{n+1}) \cdot 2^{-n}$$

$$= \sum_{n\in N} (\frac{1}{n} - \frac{1}{n+1}) = 1$$

We will use a notation for the probability mass of a string relative to all strings of equal size: $\mu'_n(x) \stackrel{\text{def}}{=} \frac{\mu'(x)}{\sum_{|y|=|x|} \mu'(y)}$.

24.2.2 Distributional Problems

Average case complexity is meaningful only if we associate a problem with a specific distribution of its instances. We will consider only decision problems. Similar formulations for search problems can be easily derived.

Definition 24.2 (Distributional Problem) A *distributional decision problem* is a pair (D, μ), where $D : \{0,1\}^* \to \{0,1\}$ and $\mu : \{0,1\}^* \to [0,1]$ is a distribution function.

24.2.3 Distributional Classes

Before defining classes of distributional problems we should consider classes of distributions. It is important to restrict the distributions, otherwise, the whole theory collapses to the worst case complexity theory (by choosing distributions that put all the probability mass on the worst cases). We will consider only "simple" distributions in a computational sense.

Definition 24.3 (P-samplable) A distribution μ is in the class *P-samplable* if there is a probabilistic Turing machine that gets no input and outputs a binary string x with probability $\mu'(x)$, while running in time polynomial in $|x|$.

Definition 24.4 (P-computable) A distribution μ is in the class *P-computable* if there is a deterministic polynomial time Turing machine that on input x outputs the binary expansion of $\mu(x)$.

Interesting distributions must put noticeable probability mass on long strings (i.e., at least $\frac{1}{poly(n)}$ on strings of length n). Consider to the contrary the density function $\mu'(x) \stackrel{\text{def}}{=} 2^{-3|x|}$. An algorithm of exponential running time, $t(x) = 2^{|x|}$, will be considered to have constant on the average running time with respect to this distribution (since $\sum_x \mu'(x) \cdot t(|x|) = \sum_n 2^{-n} = 1$). Intuitively, this distribution does not make sense since usually the long instances are the difficult ones. By assigning negligible probability to these long instances, we can artificially make the average running time of the algorithm small, even though the algorithm is not efficient. Consider, for example, an extreme case in which all instances of size greater than some constant have zero probability. In this case, every algorithm has average constant running time.

We now show that the uniform distribution is P-computable. For every x,

$$\mu_u(x) = \sum_{y \leq x} \mu'_u(y) = \sum_{\substack{y \leq x \\ |y|<|x|}} \mu'_u(y) + \sum_{\substack{y \leq x \\ |y|=|x|}} \mu'_u(y) = \left(1 - \frac{1}{|x|}\right) + \left(N_x \cdot \frac{1}{|x|(|x|+1)} \cdot 2^{-|x|}\right)$$

where $N_x \stackrel{\text{def}}{=} |\{y \in \{0,1\}^{|x|} : y \leq x\}| = 1 + \sum_{i=1}^{|x|} 2^{i-1} \cdot x_i$, where $x = x_n \cdots x_1$. Obviously, this expression can be computed in time polynomial in the length of x.

Proposition 24.2.1 *P-computable $\subset$ P-samplable (strict containment assuming $\#P \neq P$)*

Proof: We prove the proposition in two steps:

1. **Claim 24.2.2** *For every distribution μ, if $\mu \in$ P-computable then $\mu \in$ P-samplable.*

   Proof: Let μ be a distribution that is P-computable. We describe an algorithm that samples strings according to the distribution μ, assuming that we can compute μ in polynomial time. Intuitively, we can view the algorithm as picking a random real number r in $[0,1)$ and checking which string $x \in \{0,1\}^*$ satisfies $\mu(x-1) < r \leq \mu(x)$. Unfortunately, this is not possible since r has infinite precision. To overcome this problem, we select randomly in each step one bit of the expansion (of r) and stop when we are guaranteed that there is a unique x satisfying the requirement above (i.e., every possible extension will yield the same x).

   The algorithm is an iterative procedure with a stopping condition. In each iteration we select one bit. We view the bits selected so far as the binary expansion of a truncated real number, denoted by t. Now we find the smallest n such that $\mu(1^n) \geq t$. By performing a binary search over all binary strings of length n we find the greatest $x \in \{0,1\}^n$ such that $\mu(x) < t$. At this point we check if $\mu(x+1) \geq t + 2^{-l}$, where l is the length of the binary expansion of t. If so, we halt and output $x+1$. Otherwise, we continue to the next iteration. Obviously, this can be implemented in polynomial time. ■

2. **Claim 24.2.3** *There exists a distribution which is P-samplable but not P-computable, under the assumption $\#P \neq P$.*

   Proof: Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be an NP relation (we assume for simplicity that R contains only pairs of strings with the equal lengths). We define a distribution function:

$$\mu'(x \cdot \sigma \cdot y) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } R(x,y) \neq \sigma \\ \frac{1}{|x|^2 \cdot |y|^2 \cdot 2^{|x|} \cdot 2^{|y|}} & \text{otherwise} \end{cases}$$

for every $x, y \in \{0,1\}^*$ of equal length and $\sigma \in \{0,1\}$. The fact that this distribution function converges to a constant can be easily verified using the fact that for every x, y exactly one of the possible values of σ gives a non-zero probability.

We now show that μ is P-samplable. The algorithm samples uniformly a string of length $2n$, denoted $x \cdot y$, where $|x| = |y| = n$ (recall that the uniform distribution is P-computable and thus P-samplable). Next, σ is defined as $R(x, y)$, which is an NP-relation and therefore can be computed efficiently. The algorithm outputs $x \cdot \sigma \cdot y$.

The next step is to show that if μ is P-computable then #P problems can be solved in polynomial time. The number of NP witnesses for x is given by the following expression:

$$\frac{\mu(x \cdot 1 \cdot 1^n) - \mu(x \cdot 0 \cdot 1^n)}{\frac{1}{n^4 \cdot 2^{2n}}} = (\mu(x \cdot 1 \cdot 1^n) - \mu(x \cdot 0 \cdot 1^n)) \cdot n^4 \cdot 2^{2n}$$

The numerator is the probability mass of all strings that end with an NPwitness for x. By normalizing we get the actual number of witnesses, thus solving the #P problem. ■

The proposition follows directly from the two claims. ■

In the sequel we will focus on the P-computable class.

24.2.4 Distributional-NP

We now define the average-case analogue to the class NP:

Definition 24.5 (The class DistNP) A distributional problem (D, μ) belongs to the class *DistNP* if D is an NP-predicate and μ is P-computable. *DistNP* is also denoted $\langle \mathcal{NP}, \mathcal{P} - computable \rangle$.

The class $\langle \mathcal{NP}, \mathcal{P} - samplable \rangle$ is defined similarly.

24.2.5 Average Polynomial Time

The following definition may seem obscure at first glance. In the appendix we discuss the weaknesses of alternative naive formulation.

Definition 24.6 (Polynomial on the Average) A problem D is *polynomial on the average* with respect to a distribution μ if there exists an algorithm A that solves D in time $t_A(\cdot)$ and there exists a constant $\epsilon > 0$ such that

$$\sum_{x \in \{0,1\}^*} \mu'(x) \cdot \frac{t_A(x)^\epsilon}{|x|} < \infty$$

A necessary property that a valid definition should have is that a function that is polynomial in the worst case should be polynomial on the average. Assume that x^d bounds the running-time of the problem and let $\epsilon = \frac{1}{d+1}$. This function is polynomial on the average (with respect to any μ) according to Definition 6:

$$\sum_{x \in \{0,1\}^*} \mu'(x) \cdot \frac{t_A(x)^\epsilon}{|x|} \leq \sum_{x \in \{0,1\}^*} \mu'(x) \cdot \frac{|x|^{\frac{d}{d+1}}}{|x|} \leq \sum_{x \in \{0,1\}^*} \mu'(x) = 1 < \infty$$

We will now try to give some intuition for Definition 6. A natural definition for the notion of a function $f(\cdot)$ that is "constant on the average" with respect to the distribution μ is requiring

$$\sum_{x\in\{0,1\}^*} \mu'(x)\cdot f(x) < \infty$$

Using this definition, $g(\cdot)$ is called "linear on the average" if $g(x) = O(f(x)\cdot|x|)$ where $f(\cdot)$ is constant on the average. This implies

$$\sum_{x\in\{0,1\}^*} \mu'(x)\frac{g(x)}{|x|} < \infty$$

A natural extension of this definition for the case of polynomial on the average yields Definition 6.

24.2.6 Reductions

We now introduce the definition of a reduction of one distributional problem to another. In the worst case reductions, the two requirements are efficiency and validity. In the distributional case we also require that the reduction "preserve" the probability distribution. The purpose of the last requirement is to ensure that the reduction does not map very likely instances of the first problem to rare instances of the second problem. Otherwise, having a polynomial time on the average algorithm for the second distributional problem does not necessary yield such an algorithm for the first distributional problem. This requirement is captured by the domination condition.

Definition 24.7 (Average Case Reduction) We say that the distributional problem (D_1,μ_1) reduces to (D_2,μ_2) (denote $(D_1,\mu_1) \propto (D_2,\mu_2)$) if there exists a polynomial time computable function f such that

1. *validity*: $x\in D_1$ iff $f(x)\in D_2$.
2. *domination*: There exists a constant $c>0$ such that for every $y\in\{0,1\}^*$,

$$\sum_{x\in f^{-1}(y)} \mu_1'(x) \le |y|^c\cdot\mu_2'(y)$$

The following proposition shows that the reduction defined above is adequate:

Proposition 24.2.4 *If $(D_1,\mu_1)\propto(D_2,\mu_2)$ and (D_2,μ_2) is polynomial on the average then so is (D_1,μ_1).*

See proof in Appendix B.

24.3 DistNP-completeness

In this section we state two theorems regarding DistNP-complete problems and prove the first one.

Theorem 24.8 *There exists a DistNP-complete problem.*

Theorem 24.9 *Every problem complete for DistNP is also complete for $\langle \mathcal{NP}, \mathcal{P}-sampable\rangle$.*

Proof: (of first theorem) We first define a distributional version of the *Bounded Halting* problem (for a discussion on the *Bounded Halting* problem see Lecture 2.3). We then show that it is in DistNP-complete.

Definition 24.10 (Distributional Bounded Halting)

1. *Decision:* $BH(M, x, 1^k) = 1$ iff there exists a computation of the non-deterministic machine M on input x which halts within k steps.

2. *Distribution:*
$$\mu'_{BH}(M, x, 1^k) \stackrel{\text{def}}{=} \frac{1}{|M|^2 \cdot 2^{|M|}} \cdot \frac{1}{|x|^2 \cdot 2^{|x|}} \cdot \frac{1}{k^2}$$

Proving completeness results for distributional problems is more complicated than usual. The difficulty is that we have to reduce all DistNP problems with different distributions to one single distributional problem with a specific distribution. In the worst case version we used the reduction $x \to (M_D, x, 1^{P_D(|x|)})$, where D is the NP problem we want to reduce and M_D is the non-deterministic machine that solves D in time $P_D(n)$ on inputs of length n (see Lecture 2.3). A first attempt is to use exactly this reduction. This reduction is valid for every DistNP problem, but for some distributions it violates the domination condition. Consider for example distributional problems in which the distribution of (infinitely many) strings is much higher than the distribution assigned to them by the uniform distribution. In such cases, the standard reduction maps an instance x having probability mass $\mu'(x) \gg 2^{-|x|}$ to a triple $(M_D, x, 1^{P_D(|x|)})$ with much lighter probability mass (recall that $\mu'_{BH}(M_D, x, 1^{P_D(|x|)}) < 2^{-|x|}$). Thus the domination condition is not satisfied.

The essence of the problem is that μ'_{BH} gives low probability to long strings, whereas an arbitrary distribution can give them high probability. To overcome this problem, we will map long strings with high probability to short strings, which get high probability in μ'_{BH}. We will use an encoding of strings which maps a string x into a code of length bounded above by $\log_2 \frac{1}{\mu'(x)}$. We will use the following technical coding lemma:

Lemma 24.3.1 (Coding Lemma): *Let μ be a polynomial-time computable distribution function (i.e., $\mu \in \mathcal{P}-$computable). Then there exists a coding function C_μ satisfying the following three conditions:*

1. Efficient encoding: *The function C_μ is computable in polynomial time.*

2. Unique decoding: *The function C_μ is one-to-one.*

3. Compression: *For every x*
$$|C_\mu(x)| \leq 1 + min\{|x|, \log_2 \frac{1}{\mu'(x)}\}$$

Proof: The function C_μ is defined as follows:

$$C_\mu(x) \stackrel{\text{def}}{=} \begin{cases} 0 \cdot x & \text{if } \mu'(x) \leq 2^{-|x|} \\ 1 \cdot z & \text{otherwise} \end{cases}$$

where z is the longest common prefix of the binary expansions of $\mu(x-1)$ and $\mu(x)$ (e.g., if $\mu(1010) = 0.10000$ and $\mu(1011) = 0.10101$ then $C_\mu(1011) = 110$). The intuition behind this

definition is that we want to find uniquely a point with certain precision in the interval between $\mu(x-1)$ and $\mu(x)$. As the length of the interval grows, the precision needed is lower. Recall that the length of the interval is exactly the value of $\mu'(x)$, implying the short encoding of high probability strings.

We now show that $C_\mu(x)$ satisfies the conditions of the lemma:

1. *Efficient encoding:* The efficiency of the encoding follows from the fact that μ is a polynomial time computable function.

2. *Unique decoding:* In the first case, in which $C_\mu(x) = 0 \cdot x$, the unique decoding is obvious. In the second case, in which $C_\mu(x) = 1 \cdot z$, since the intervals are disjoint and $\mu(x-1) < 0.z1 \leq \mu(x)$, every z determines uniquely the encoded x, and the unique decoding conditions follows.

3. *Compression:* In the first case, in which $\mu'(x) \leq 2^{-|x|}$, $|C_\mu(x)| = 1 + |x| \leq 1 + \log_2 \frac{1}{\mu'(x)}$. In the second case, in which $\mu'(x) > 2^{-|x|}$, let $l = |z|$ and $z_1 \cdots z_l$ be the binary representation of z. Then,

$$\mu'(x) = \mu(x) - \mu(x-1) \leq \left(\sum_{i=1}^{l} 2^{-i} z_i + \sum_{i=l+1}^{poly(|x|)} 2^{-i} \right) - \sum_{i=1}^{l} 2^{-i} z_i < 2^{-|z|}$$

   So $|z| \leq \log_2 \frac{1}{\mu'(x)}$, and the compression condition follows.

■

Now we use the Coding Lemma to complete the proof of the theorem. We define the following reduction of (D, μ) to (BH, μ_{BH}):

$$x \to (M_{D,\mu}, C_\mu(x), 1^{P_{D,\mu}(|x|)})$$

where $M_{D,\mu}$ is a non-deterministic machine that on input y guesses non-deterministically x such that $C_\mu(x) = y$ (notice that the non-determinism allows us not to require efficient decoding), and then runs M_D on x. The polynomial $P_{D,\mu}(n)$ is defined as $P_D(n) + P_C(n) + n$, where $P_D(n)$ is a polynomial bounding the running time of M_D on acceptable inputs of length n and $P_C(n)$ is a polynomial bounding the running time of the encoding algorithm.

It remains to show that this reduction satisfies the three requirements.

1. *Efficiency:* The description of $M_{D,\mu}$ is of fixed length and by the coding lemma C_μ is computable by polynomial time. Therefore, the reduction is efficient.

2. *Validity:* By construction of $M_{D,\mu}$ it follows that $D(x) = 1$ if and only if there exists a computation of machine $M_{D,\mu}$ that on input $C_\mu(x)$ halts with output 1 within $P_{D,\mu}(|x|)$ steps.

3. *Domination:* Notice that it suffices to consider instances of *Bounded Halting* which have a preimage under the reduction. Since the coding is one-to-one, each such image has a unique preimage. By the definition of μ_{BH},

$$\mu'_{BH}(M_{D,\mu}, C_\mu(x), 1^{P_{D,\mu}(|x|)}) = c \cdot \frac{1}{P_{D,\mu}(|x|)^2} \cdot \frac{1}{|C_\mu(x)|^2 \cdot 2^{|c_\mu(x)|}}$$

where $c = \frac{1}{|M_{D,\mu}|^2 \cdot 2^{|M_{D,\mu}|}}$ is a constant independent of x. By the compression requirement of the coding lemma,

$$\mu'(x) \leq 2 \cdot 2^{-|C_\mu(x)|}$$

Hence,

$$\mu'_{BH}(M_{D,\mu}, C_\mu(x), 1^{P_{D,\mu}(|x|)}) \geq c \cdot \frac{1}{P_{D,\mu}(|x|)^2} \cdot \frac{1}{|C_\mu(x)|^2} \cdot \frac{\mu'(x)}{2} > \frac{c}{2 \cdot P_{D,\mu}(|x|)^2 \cdot |C_\mu(x)|^2} \cdot \mu'(x)$$

Therefore, the reduction satisfies the requirements and the distributional version of the *Bounded Halting* problem is DistNP-complete.

■

Bibliographic Notes

The theory of average-case complexity was initiated by Levin [4]. Theorem 24.9 is due to [3]. The lecture was based on the exposition in [2]. For further investigations see [1].

1. S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the Theory of Average Case Complexity. *Journal of Computer and system Sciences*, Vol. 44, No. 2, April 1992, pp. 193–219.

2. O. Goldreich. Notes on Levin's Theory of Average-Case Complexity. *ECCC*, TR97-058, Dec. 1997.

3. R. Impagliazzo and L.A. Levin. No Better Ways to Generate Hard NP Instances than Picking Uniformly at Random. In *Proc. of the 31st FOCS*, 1990, pp. 812–821.

4. L.A. Levin. Average Case Complete Problems. *SIAM Jour. of Computing*, Vol. 15, pages 285–286, 1986.

Appendix A : Failure of a naive formulation

We now discuss an alternative definition of the notion of polynomial on the average which seems more natural:

Definition 24.11 (Naive Formulation of Polynomial on the Average) A problem D is *polynomial on the average* with respect to a distribution μ if there exists an algorithm A that solves D in time $t_A(\cdot)$ and there exists a constant $c > 0$ such that for every n

$$\sum_{x \in \{0,1\}^n} \mu'_n(x) \cdot t_A(x) < n^c$$

There are three main problems with this naive definition:

1. This definition is very dependent on the particular encoding of the problem instance. In this definition, the average is taken over all instances of equal length. Changing the encoding of the problem instances does not preserve the partition of instances according to their length and hence does not preserve the average running time of the algorithm.

2. This definition is not robust under functional composition of algorithms. Namely, if distributional problem A can be solved in average polynomial time given access to an oracle for distributional problem B and B can be solved in average polynomial time, then it does not follow that A can be solved in average polynomial time.

3. This definition is not machine independent, i.e., an algorithm can be polynomial on the average in one reasonable computational model, but hard on the average in another (e.g. the simulation of a two-tape Turing machine on a one-tape Turing machine).

The two last problems stem from the fact that the definition is not closed under application of polynomials. For example, consider a function $t(x)$ defined as follows:

$$t(x) \stackrel{\text{def}}{=} \begin{cases} 2^n & \text{if } x = 1^n \\ n^2 & \text{otherwise} \end{cases}$$

This function is clearly polynomial on the average with respect to the uniform distribution (i.e., for every $x \in \{0,1\}^n, \mu'_n(x) = 2^{-n}$).

This is true since

$$\sum_{x \in \{0,1\}^n} \mu'_n(x) \cdot t(x) = 2^{-n} \cdot 2^n + (1 - 2^{-n}) \cdot n^2 < n^2 + 1$$

On the other hand,

$$\sum_{x \in \{0,1\}^n} \mu'_n(x) \cdot t^2(x) = 2^{-n} \cdot 2^{2n} + (1 - 2^{-n}) \cdot n^4 > 2^n$$

which implies that the function $t^2(x)$ is not polynomial on the average. This problem does not occur in Definition 6 since if $t(x)$ is polynomial on the average with the constant ϵ then $t^2(x)$ is polynomial on the average with the constant $\frac{\epsilon}{2}$.

Appendix B : Proof Sketch of Proposition 24.2.4

In this appendix we sketch the proof of proposition 2.4. We first restate the proposition:

Proposition 2.4 : If $(D_1, \mu_1) \propto (D_2, \mu_2)$ and (D_2, μ_2) is polynomial on the average then so is (D_1, μ_1).

The formal proof of the proposition has many technical details, so it will be only sketched. As a warm-up, we will first prove the proposition under some simplifying assumptions. To our belief, this gives intuition to the full proof. The simplifying assumptions, regarding the definition of a reduction, are:

1. There exists a constant c_1 such that for every x, $|f(x)| \leq c_1 \cdot |x|$, where f is the reduction function.

2. Strong domination condition: There exists a constant c_2 such that for every $y \in \{0,1\}^*$,

$$\sum_{x \in f^{-1}(y)} \mu'_1(x) \leq c_2 \cdot \mu'_2(y)$$

Proof: (*of simplified version*) The distributional problem (D_2, μ_2) is polynomial on the average implies that there exists an algorithm A_2 and a constant $\epsilon_2 > 0$ such that

$$(1) \qquad \sum_{x \in \{0,1\}^*} \mu_2'(x) \cdot \frac{t_{A_2}(x)^{\epsilon_2}}{|x|} < \infty$$

We need to prove that (D_1, μ_1) is polynomial on the average, i.e. that there exists an algorithm A_1 and a constant $\epsilon_1 > 0$ such that

$$(2) \qquad \sum_{x \in \{0,1\}^*} \mu_1'(x) \cdot \frac{t_{A_1}(x)^{\epsilon_1}}{|x|} < \infty$$

The algorithm A_1, when given x, applies the reduction f on x and then applies A_2 on $f(x)$. Therefore, $t_{A_1}(x) = t_f(x) + t_{A_2}(f(x))$, where t_f denotes the time required to compute f. For the sake of simplicity we ignore $t_f(x)$ and assume $t_{A_1}(x) = t_{A_2}(f(x))$. Taking $\epsilon_1 \stackrel{\text{def}}{=} \epsilon_2$ we obtain:

$$\begin{aligned}
\sum_{x \in \{0,1\}^*} \frac{\mu_1'(x) \cdot t_{A_1}(x)^{\epsilon_1}}{|x|} &= \sum_{y \in \{0,1\}^*} \sum_{x \in f^{-1}(y)} \frac{\mu_1'(x) \cdot t_{A_2}(y)^{\epsilon_1}}{|x|} \\
&\leq c_1 \cdot \sum_{y \in \{0,1\}^*} \frac{t_{A_2}(y)^{\epsilon_1}}{|y|} \sum_{x \in f^{-1}(y)} \mu_1'(x) \\
&\leq c_1 \cdot c_2 \cdot \sum_{y \in \{0,1\}^*} \frac{t_{A_2}(y)^{\epsilon_2} \cdot \mu_2'(y)}{|y|} < \infty
\end{aligned}$$

The first inequality uses Assumption 1, and the second uses the simplified domination condition (Assumption 2). ∎

Sketch of full proof: First we explain how to deal with the technical problem that arises when considering the running time of the reduction, $t_f(x)$, in the running time of algorithm A_1. This technical problem arises many times in the proof and is solved as described below. The fact that for every $\epsilon \leq 1$, $(a+b)^\epsilon \leq a^\epsilon + b^\epsilon$ is used to bound expression (2) by two sums. The first sum contains the factor $t_f(x)$ and can be easily bounded by choosing the appropriate ϵ. The second sum converges as shown in the proof of the simplified version.

In order to prove that expression (2) converges we partition the sum in expression (2) into two sums and show separately that each sum converges. Formally, $y \in \{0,1\}^*$ is called a *bad* y if $\mu_2'(y) \cdot t_{A_2}(y)^{\epsilon_2} \geq |y|$ and is called a *good* y otherwise. We partition the x's according to the "goodness" of their images under f. This induces the following partition into two sums.
Denote

$$B = \sum_{bad\ y} \sum_{x \in f^{-1}(y)} \frac{\mu_1'(x) \cdot t_{A_2}(y)^{\epsilon_1}}{|x|} \qquad G = \sum_{good\ y} \sum_{x \in f^{-1}(y)} \frac{\mu_1'(x) \cdot t_{A_2}(y)^{\epsilon_1}}{|x|}$$

Then

$$\sum_{x \in \{0,1\}^*} \frac{\mu_1'(x) \cdot t_{A_1}(x)^{\epsilon_1}}{|x|} = \sum_{y} \sum_{x \in f^{-1}(y)} \frac{\mu_1'(x) \cdot t_{A_2}(y)^{\epsilon_1}}{|x|} = B + G$$

The intuition behind this partition is that each bad y contributes large weight (at least one) to the sum in expression (1). The fact that expression (1) converges implies that there is a finite number of bad y's. This is used to show that B converges. For the second sum we again partition it into two sums, G_1 and G_2. The first sum, G_1, consists of good y's for which $t_{A_2}(y)$ is bounded by $p(|x|)$ for every $x \in f^{-1}(x)$ and some polynomial p (that depends on ϵ_2 and the domination constant c). This sum can be bounded by choosing an ϵ_1 so that $t_{A_2}(y)^{\epsilon_1} < |x|$ for any $x \in f^{-1}(y)$. That is,

$$\sum_{y:t_{A_2}(y)<\min_{x\in f^{-1}(y)}\{p(|x|)\}} \sum_{x\in f^{-1}(y)} \frac{\mu_1'(x)\cdot t_{A_2}(y)^{\epsilon_1}}{|x|} < \sum_x \mu_1'(x) = 1$$

The second sum, G_2, consists of the rest of the good y's. For each y in this sum, we have $t_{A_2}(y) \geq q(y) \stackrel{\text{def}}{=} \min_{x\in f^{-1}(y)}\{p(|x|)\}$. Note that q grows at least as a power of p (depending on the relation of $|f(x)|$ to $|x|$); that is for some $\epsilon > 0$ we have $q(y) \geq p(|y|^\epsilon)$. By a suitable choice of p and ϵ_1, we have

$$|y|^c \cdot t_{A_2}(y)^{\epsilon_1} \;=\; \frac{|y|^c \cdot t_{A_2}(y)^{\epsilon_2}}{p(|y|^\epsilon)^{\epsilon_2-\epsilon_1}} \;<\; \frac{t_{A_2}(y)^{\epsilon_2}}{|y|}$$

Thus,

$$\begin{aligned}\sum_{y:t_{A_2}(y)\geq q(y)} \sum_{x\in f^{-1}(y)} \frac{\mu_1'(x)\cdot t_{A_2}(y)^{\epsilon_1}}{|x|} &\leq \sum_{y:t_{A_2}(y)\geq q(y)} |y|^c \mu_2'(y)\cdot t_{A_2}(y)^{\epsilon_1} \\ &\leq \sum_{y:t_{A_2}(y)\geq q(y)} \frac{\mu_2'(y)\cdot t_{A_2}(y)^{\epsilon_2}}{|y|}\end{aligned}$$

which converges being a partial sum of (1).

Finally, we choose ϵ_1 to be the minimum of all the ϵ_1's used in the proof to obtain the convergence of the original sum. ■

Lecture 25

Computational Learning Theory

Lecture given by Dana Ron
Notes taken by Oded Lachish and Eli Porat

Summary: We define a model of learning known as probably approximately correct (PAC) learning. We define efficient PAC learning, and present several efficient PAC learning algorithms. We prove the Occam's Razor Theorem, which reduces the PAC learning problem to the problem of finding a succinct representation for the values of a large number of given labeled examples.

25.1 Towards a definition of Computational learning

Learning is a notion we are familiar with from every day life. When embarking on the task of importing this notion into computer science, the first natural step is to open a dictionary and find an exact meaning for it. A natural meaning that can be found is "gaining knowledge through experience". With this meaning in mind, we set on the task of defining a formal computer science model for learning. In order to get a clue to what the model should look like, it is worthwhile to examine a real life setting.

Learning to diagnose a new disease. A medical doctor learns the symptoms for diagnosing a new disease by drawing a number of files, from a file archive. Each file contains a list of the patient's parameters such as weight, body temperature, age, etc., and a *label* indicating the diagnosis; that is, whether the person has this specific disease. The number of examples the doctor has drawn depends on how accurate he wanted his list of symptoms to be. Using these files he concludes a list of symptoms for diagnosing the new disease. In order to check the accuracy of symptoms he concluded, he draws a labeled file, uses the list of symptoms to reach a diagnosis, and finally whether the diagnosis obtained by him matches the true diagnosis provided in the file (i.e., the label of the file).

Using this setting we can phrase the process of learning as follows: In order to learn an unknown subject (disease) from a family of subjects (family of diseases) the learner (doctor) receives a measure of accuracy, with this measure in mind he draws a number of labeled examples (files) from the set of all possible examples (archive of files), labeled with respect to subject. Using the examples he reaches a rule (list of symptoms) for labeling examples. He checks his rule accuracy, by drawing an example, and comparing its label to the label computed by the rule for this example.

But something is still lacking in the above model. We demonstate this by using the above setting (of a doctor learning to diagnose a new disease): Can a doctor learn a new disease that occurs in a certain group of people, when he does not see a file of a person from this specific group? Of course we cannot hope for this to happen. Thus, the missing component in the above model is that the doctor's ability to diagnose will be tested against the same distribution of examples (files) based on which he has learned to diagnose. Thus, if in the learning stage the doctor didn't draw any file of a person from a specific group, then chances are that he won't be asked to diagnose such a person later (i.e., the the accuracy of symptoms he concluded will not be checked for such a person). We stress that we do not require the learner (doctor) to have any knowledge regarding the distribution of examples.

The components of the learning process: We call the object that conducts the learning process, the *learner*. In this lecture the learner is an algorithm. The objects on which the learning is done are called *instances*. The specifics of the instances do not interest us. We are only interested in an abstract representation of the parameters characterizing it. We represent an instance by a vector defined as follows: a value is assigned to each parameter of the instance, this vector is the representation of these values.

- $X = \bigcup_{n \geq 1} X_n$ Instance space – the set of all possible values of instance representation.
- $D : X_n \longrightarrow [0,1]$ Underlying distribution – the (unknown) probability distribution over example space.

where n is the number of parameters characterizing an instance. In this lecture the instance space will be either $\{0,1\}^n$ or R^n. The learning is done with respect to the distribution D, from which we obtain independent samples. We denote by $x \sim D$ an example x drawn from X_n according to distribution D.

The subject and aim of our learning is a *labeling* of the instance space, where by labeling we mean a correlation between the instance space and the a set values. In this lecture we use the set $\{0,1\}$ (or any set isomorphic to it) as possible labeling values. Each instance is labeled by such a value, and so the labeling (of all instances) is a function from X to $\{0,1\}$. Such a labeling is called a *concept*. We use the following notations

- $F = \bigcup_{n \geq 1} F_n$ concept class – family of concepts we may need to learn. This family is a subset of the set of all Boolean functions defined over X.
- $f \in F_n$ target concept – the concept which is the subject of the learning.

where n appears in notations since a concept is a function over the instance space X_n.

The rule that is the output of the learner is also a labeling of the instance space, therefore it is also a Boolean function. We call this function the *hypothesis*.

The final component missing is a measure of accuracy of the hypothesis. We actually consider the complementary measure; that is the error of the hypothesis. The latter is merely the probability that for an instance drawn according to D, which is the very distribution used in the learning process, the hypothesis agrees with the target concept. That is,

Definition 25.1 (hypothesis error): *$err_{f,D}(h)$ is the probability that the hypothesis h disagrees with the target function f on an instance x drawn from X_n according to D. That is,*

$$err_{f,D}(h) \stackrel{\text{def}}{=} \Pr_{x \sim D}[h(x) \neq f(x)]$$

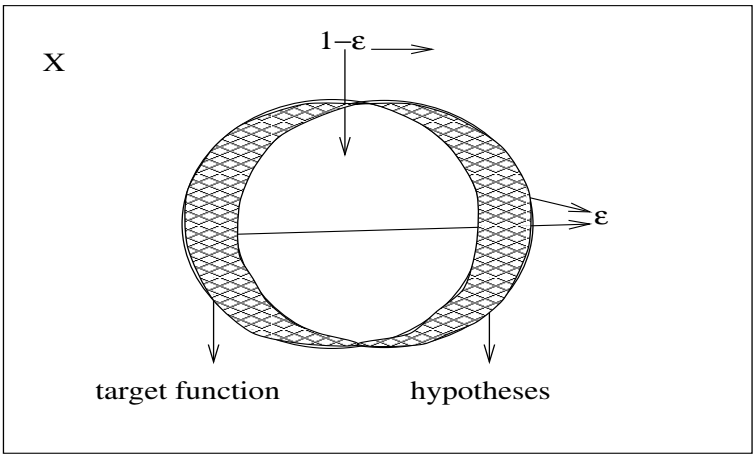

Figure 25.1: The target function and hypothesis with their difference shaded

25.2 Probably Approximately Correct (PAC) Learning

In this model the algorithm is only given partial information on the target concept, that is, a "small" number of examples drawn independently by a given distribution. Since the examples are chosen independently by a distribution, it could be the case that the algorithm was given the same example all the time. Therefore it is unconceivable to expect the algorithm to give an hypothesis h which is fully equivalent to the target function f. A realistic view to the problem would be to expect with a given probability that the algorithm will supply an hypothesis h that is an approximation of the target function f with respect to the underlying distribution. The following definitions capture these qualities:

Definition 25.2 (PAC learning algorithm): *An algorithm A is called a PAC* learning algorithm for a concept class *F if the following holds: For every n, for every function $f \in F_n$, for every distribution $D : X_n \longrightarrow [0,1]$, for every error parameter $0 < \epsilon < 1$ and for every confidence parameter $0 < \delta < 1$, given parameters n, ϵ, δ and a set of f-labeled examples, $\{< x^i, f(x^i) >\}$, where x^i is drawn independently under the distribution D, the algorithm outputs a hypothesis h such that:*

$$\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$$

where the probability is taken over the choice of examples, as well as over the internal coin tosses of the algorithm. Note $h = A(n, \epsilon, \delta, \{< x^i, f(x^i) >\})$.

Once we established a formal model, a natural step would be to inquire upon following questions:

1. What is an efficient PAC learning algorithm?
2. How many examples do we need for PAC learning a concept from a concept class?

In order to deal with these questions we define complexity measures for the PAC learning model. A natural complexity measure is the number of examples needed:

Definition 25.3 (sample complexity): *The* sample complexity *of a PAC algorithm is the number of examples it utilizes, as a function of n, ϵ and δ.*

The classical measure of running time of an algorithm is also applicable for this model. The running time complexity is necessarily larger then the sample complexity. In some cases it appears to be substaintially larger.

> Oded's Note: *That is, under reasonable complexity assumptions (e.g., existence of one-way functions), there are concept classes that can be PAC learned using* $\mathrm{poly}(n, \frac{1}{\epsilon}, \log \frac{1}{\delta})$ *examples, but cannot be PAC learned in* $\mathrm{poly}(n, \frac{1}{\epsilon}, \frac{1}{\delta})$*-time. (In analogy to other contexts, we should expect complexities to be logarithmic in* $1/\delta$*.)*

Using these complexity measures we can answer the first question. We say that a PAC learning algorithm is efficient if it runs in time polynomial in n, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and $size(f)$, where the size of $f \in F_n$ is usually polynomial in n. In the rest of this lecture, we focus on the second question (i.e., consider only the sample complexity of PAC algorithms).

Learning axis-aligned rectangles over $[0,1]^2$. We wish to design an efficient PAC learning algorithm for axis-aligned rectangles over $[0,1]^2$.

Let us first explicitly cast the problem according to the PAC learning model.

- An instance is a point in $[0,1]^2$, we represent it by its x,y axis coordinates, that is $< x, y >$.
- The instance space is $X = \{\text{all } < x, y > \text{ representations of points in } [0,1]^2\}$.
- A concept is an axis-aligned rectangle in $[0,1]^2$, represented by four points $< x_{min}, y_{min}, x_{min}, y_{max} >$ such that:

  $$x_{min}, x_{max}, y_{min}, y_{max} \in [0,1]$$

  $$x_{min} \leq x_{max} \text{ and } y_{min} \leq y_{max}$$

  An instance $< x, y >$ is in the axis aligned rectangle f represented by $< x_{min}, y_{min}, x_{min}, y_{max} >$ if:

  $$x_{min} \leq x \leq x_{max} \text{ and } y_{min} \leq y \leq y_{max}$$

  If $< x, y >$ is in this f, we label it by a "+" (i.e., $f(< x, y >) = +$), otherwise we label it by a "-".
- The concept class is $F = \{\text{all axis-aligned rectangle in } [0,1]^2 \}$. The target concept is an axis aligned rectangle $f \in F$.
- Finally, as usual, the underlying distribution is denoted D, the error parameter is $0 < \epsilon < 1$ and the confidence parameter is $0 < \delta < 1$.

For the sake of clarity we will refer to the target concept by target rectangle, and to axis-aligned rectangles as rectangles. We use the following notations:

- $W_D(g)$ is the weight assigned by distribution D to the rectangle g. That is,

  $$W_D(g) \stackrel{\text{def}}{=} \Pr_{s \sim D}[g(s) = +]$$

- S is a random variable representing the set of example points. Each element in S is drawn according to D. We let S^+ denote the subset of S labeled "+".

We stress that all probabilities are taken over the choice of examples.

Claim 25.2.1 *The following algorithm is an efficient PAC learning algorithm for axis-aligned rectangles over* $[0,1]^2$.

If S^+ *is empty then output the empty rectangle as hypothesis.*
Otherwise, output the rectangle represented by $< \tilde{x}_{min}, \tilde{y}_{min}, \tilde{x}_{max}, \tilde{y}_{max} >$, *where*
$\tilde{x}_{min}$ = *the minimal x-axis coordinate in* S^+.
$\tilde{y}_{min}$ = *the minimal y-axis coordinate in* S^+.
$\tilde{x}_{max}$ = *the maximal x-axis coordinate in* S^+.
$\tilde{y}_{max}$ = *the maximal y-axis coordinate in* S^+.

Proof: Let f be the target rectangle and g be the hypothesis output by the algorith. Note that the hypothesis rectangle is always contained in the target rectangle (since the borders of the former are determined by positive examples).

We partition the proof into two cases:

1. $W_D(f) > \epsilon$
2. $W_D(f) \leq \epsilon$

We start with the proof of the first case. Given a target rectangle let us draw the following auxiliary construction: We cover the upper side of the target rectangle with a line, we push this line towards the opposite side, until we get a rectangle A_1, such that $W_D(A_1) = \frac{\epsilon}{4}$.

> Oded's Note: *We assume that an adequate stopping point exists; that is, that the distribution is such that some rectangle has weight smaller than* $\epsilon/4$ *whereas a slightly bigger rectangle has weight* $\epsilon/4$. *Clearly, an approximation of this assumption is good enough, but even such an approximation is not guaranteed to exist. This issue is dealt with in an appendex.*

We repeat this process for all other sides of the residual target rectangle (see Fig. 25.2). We get rectangle A_2,A_3,A_4 such that $W_D(A_2) = W_D(A_3) = W_D(A_4) = \frac{\epsilon}{4}$. (It should be stressed, that we assumed that this process can be done, which is not necessarily the case. We will deal with this problem in appendix.) Let us look at the part of the f, that is not covered by A_1,A_2,A_3 and A_4, it is a rectangle we call it B. According to auxiliary construction:

$$\begin{aligned} W_D(f) - W_D(B) &= W_D(A_1) + W_D(A_2) + W_D(A_3) + W_D(A_4) \\ &= 4 \cdot \frac{\epsilon}{4} = \epsilon \end{aligned}$$

If hypothesis rectangle h contains B (i.e., $h \supseteq B$) then (using $h \subseteq f$)

$$\begin{aligned} err_{f,D}(h) &= \Pr_{s \sim D}[s \in f \text{ and } s \notin h] \\ &\leq \Pr_{s \sim D}[s \in f \text{ and } s \notin B] \\ &= W_D(f) - W_D(B) = \epsilon \end{aligned}$$

Furthermore, according to algorithm, if there is an example in each of the rectangle A_1, A_2, A_3 and A_4, then $B \subseteq h$. Thus, we merely bound the number of examples, denoted m, such that the probability for this event not to occur, is less then δ. For any $i \in \{1,2,3,4\}$, the probability that no example resides in A_i is $(1 - W_D(A_i))^m = (1 - \frac{\epsilon}{4})^m$. Thus,

$$\begin{aligned} \Pr[err_{f,D}(h) > \epsilon] &\leq \Pr[\exists i \text{ no sample in } A_i] \\ &\leq 4 \cdot \left(1 - \frac{\epsilon}{4}\right)^m \end{aligned}$$

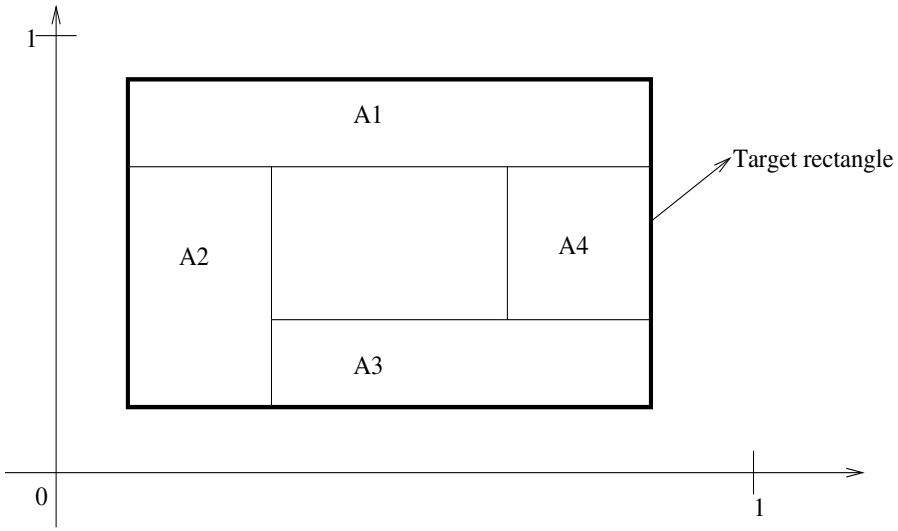

Figure 25.2: Target rectangle in bold, and rectangle A_1, A_2, A_3, A_4 each of weight $\frac{\epsilon}{4}$

So we need to set m so that $4 \cdot (1 - (\epsilon/4))^m \leq \delta$. Using the inequality $1 - \frac{\epsilon}{4} \leq e^{-\frac{\epsilon}{4}}$, the condition simplifies to $4(e^{-\frac{\epsilon}{4}})^m \leq \delta$, which solves to $m \geq (4/\epsilon) \cdot \ln(4/\delta)$.

We now turn to the second case, where $W_D(f) \leq \epsilon$: Using $h \subseteq f$, we have (for every sequence of examples used by the algorithm):

$$\begin{aligned} err_{f,D}(h) &= \Pr_{s\sim D}[s \in f \text{ and } s \notin h] \\ &\leq \Pr_{s\sim D}[s \in f] \;=\; W_D(f) \;\leq\; \epsilon \end{aligned}$$

Thus, in this case, $\Pr[err_{f,D}(h) \leq \epsilon] = 1$.

Note all operations in algorithm depend linearly on the size of the sample, which in turn depends linearly on $\frac{1}{\epsilon}$ and $ln\frac{4}{\delta}$. Therefore the algorithm is an efficient PAC learning algorithm. ■

25.3 Occam's Razor

Occam's Razor is based on a principle stated by William of Occam. We interpret Occam's principle as follows: learning can be achieved by finding a succinct representation for the labels of large number of examples. By a succinct representation, we mean that the size of the representation is sublinear in the number of examples. This reduces the problem of learning to the problem of finding a hypothesis, consistent with every given example. We say an algorithm is an *Occam's Algorithm*, if it outputs a succinct hypothesis consistent with every given example.

Theorem 25.4 (Occam's Razor – basic version): *Let F_n be a finite concept class. Let A be an algorithm such that for every n, for every $f \in F_n$ and for every number of examples labeled by f, the algorithm outputs an hypothesis $h \in F_n$, that is consistent with all the given examples. Then for any distribution D, for any error parameter $0 < \epsilon < 1$ and for any confidence parameter $0 < \delta < 1$ if the number of examples drawn independently by D is larger or equal $\frac{1}{\epsilon}(\log \frac{1}{\delta} + \log |F_n|)$, then*

$$\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$$

where h is the hypothesis output by the algorithm, and the probability is taken over the choice of examples as well as over the internal coin tosses of the algorithm.

Proof: We use the following notations:

- $S =< a_j, f(a_j) >_{j=1...m}$ is the set of labeled examples.
- $C_F(S)$ is the set of hypotheses in F that are consistent on all examples in S. That is,

$$C_F(S) \stackrel{\text{def}}{=} \{h \in F : h(a_i) = f(a_i) \text{for } i = 1, ..., m \}$$

- $B_{D,\epsilon}(f)$ is the set of hypotheses in F that have a probability of error larger than ϵ. That is,

$$B_{D,\epsilon}(f) \stackrel{\text{def}}{=} \{h \in F : err_{f,D}(h) > \epsilon\}$$

Note that the algorithm always outputs an hypothesis $h \in C_F(S)$. We will upper bound the probability that this hypothesis is in $B_{D,\epsilon}(f)$. Actually, we will upper bound the probability that any hypothesis in $C_F(S)$ remains in $B_{D,\epsilon}(f)$.

$$\Pr_S[\text{there exists } h \in B_{D,\epsilon}(f) \text{ such that } h \in C_F(S)] \leq \sum_{h \in B_{D,\epsilon}(f)} \Pr_S[h \in C_F(S)] \tag{25.1}$$

Proposition 25.3.1 *For any $h \in B_{D,\epsilon}(f)$, $\Pr_S[h \in C_F(S)] \leq \frac{\delta}{|F|}$*

Proof: For any hypothesis in $B_{D,\epsilon}$ the probability that the hypothesis will be in $C_F(S)$ is the probability that all given example points landed in the agreement area. By using definition of $B_{D,\epsilon}(f)$ the probability that a single example point landed in the agreement area is at most $1 - \epsilon$. Therefore for any fixed $h \in B_{D,\epsilon}(f)$:

$$\Pr_S[h \in C_F(S)] \leq (1 - \epsilon)^{|S|} \leq e^{-\epsilon|S|}$$

Using the assumption on $|S|$:

$$\epsilon|S| \geq \log(\tfrac{1}{\delta}|F|)$$

Thus,

$$e^{-\epsilon|S|} \leq \tfrac{\delta}{|F|}$$

and the proposition follows. ■

Combining Eq. (25.1) and Proposition 25.3.1, the theorem follows. ■

It is easy to observe, that an algorithm satisfying this theorem, is a PAC learning algorithm. Also note that since the hypothesis is taken from the concept class it is necessarily succinct.

Learning monomials. The concept class F_n of monomials is a family of boolean expressions, over literals corresponding to n variables $x_1, x_2, ..., x_n$, defined as follows:

$$F_n = \{f : f = l_1 \wedge l_2 \wedge ... \wedge l_t\}$$

Where for each $1 \leq i \leq t$

$$l_i \in \{x_j, \overline{x_j}\}_{j=1}^{n}$$

We seek an efficient PAC learning algorithm for the concept class of monomials. A instance $a \in X_n$ is interpreted as an assignment to the variables $x_1, x_2, ..., x_n$. We us the notation a_i for the value of the i'th bit of a.
We denote the set of given examples by:

$$S = \{< a^j, f(a^j) >\}_{j=1,...,m}$$

where each $a^j = a_1^j, ..., a_n^j$.

We call an example *negative* if it is labeled by 0, otherwise we call it *positive*. Before stating the algorithm, let us see what information can we conclude about the target concept, from a given example. A positive example consists of an assignment to the target concept, such that every the literal in it evaluates to 1. A negative example consists of an assignment to the target concept, such that there exists a literal in it that evaluates to 0. Thus negative examples convey much less information then positive examples, and the information they convey is not trivial to use. Our algorithm uses only positive examples.

Claim 25.3.2 *Given at least* $\frac{1}{\epsilon}(2n + \log\frac{1}{\delta})$ *examples, the following algorithm is a PAC learning algorithm.*

1. *initialize* $h = x_1 \wedge \overline{x_1} \wedge x_2 \wedge \overline{x_2} \wedge ... \wedge x_n \wedge \overline{x_n}$
2. *for* $j = 1...m$, *if* $f(a^j) =$ '$+$' *then do*
   - *for each* $i \in \{1...n\}$ *if* $a_i^j = 1$ *remove* $\overline{x_i}$ *from* h
   - *for each* $i \in \{1...n\}$ *if* $a_i^j = 0$ *remove* x_i *from* h

Proof: We use the following notations: By $S = \{< a^j, f(a^j) >\}_{j=1,...,m}$ we denote the set of all samples. By h^j we denote the expression h after j iterations; that is, after processing the examples $a^1, ..., a^j$.

We first show that the final hypothesis h is consistent with all examples. Using induction on j, we show that h^j is consistent with the first j examples and that h^j includes all literals in f (i.e., $h^j \supseteq f$). The induction basis ($j = 0$) holds trivially (as h^0 is as initialized in Step 1). In the induction step, suppose that $h^j \supseteq f$ is consistent with the first j examples, and consider what happens in the $j + 1^{\text{st}}$ iteration. We consider two cases

Case 1: the $j + 1^{\text{st}}$ example is positive. In this case $h^{j+1} \subseteq h^j$ and $h^{j+1} \supseteq f$ (since the only literals omitted from h^j are those that cannot be in f).

- Using $h^{j+1} \subseteq h^j$ (i.e., $h^j(a) = 1$ implies $h^{j+1}(a) = 1$), we have $h^{j+1}(a^i) = h^j(a^i) = f(a^i)$ for every $i = 1, ..., j$ satisfying $f(a^i) = 1$.
- Using $h^{j+1} \supseteq f$ (i.e., $f(a) = 0$ implies $h^{j+1}(a) = 0$), we have $h^{j+1}(a^i) = f(a^i)$ for every $i = 1, ..., j$ satisfying $f(a^i) = 0$.
- Finally, by the operation of the current interaction, $h^{j+1} \subseteq \wedge_{i=1}^{n} l_i$, where $l_i = x_i$ if $a_i^{j+1} = 1$, and $l_i = \overline{x}_i$ otherwise. Thus, $h^{j+1}(a^{j+1}) = 1$.

It follows that $h^{j+1}(a^i) = f(a^i)$ holds for $i = 1, ..., j, j + 1$.

Case 2: the $j + 1^{\text{st}}$ example is negative. In this case $h^{j+1} = h^j$ and so $h^{j+1} \supseteq f$. Also, since $f(a^{j+1}) = 0$, it follows that $h^{j+1}(a^{j+1}) = 0$. Thus, $h^{j+1}(a^i) = f(a^i)$ holds for $i = 1, ..., j, j + 1$.

Let us compute the cardinality of F_n. The cardinality of F_n is bounded by 2^{2n}, since each literal can either not appear in the monomial or appear in monomial, and the number of literals is $2n$. Thus, $\log_2 |F_n| \leq 2n$, and the claim follows by applying Occam's Razor. ■

Is this version of Occam's Razor powerful enough? The following example shows that this current version is somewhat limited.

Learning a 3-term DNF. The concept class F_n of 3-term DNF's is a family of boolean expressions, over variables $x_1, x_2, ..., x_n$, defined as follows:

$$F_n = \{f : f = M1 \wedge M2 \wedge M3 \text{ where } M1, M2, M3 \text{ are monomials over } x_1, x_2, ..., x_n\}$$

The learning task seems similar to the previous task of learning monomials. However, the problem of finding a consistent 3-term DNF seems intractable. Specifically:

Claim 25.3.3 *The problem of finding a 3-term DNF that is consistent with a given set of examples is $\mathcal{NP}$-complete.*

The proof is via a reduction to 3-colorability and is omitted. Actually, the difficulty is not due to Occam algorithms only. It rather holds with respect to any PAC learning algorithm that always outputs hypotheses in the concept class (in our case a 3-term DNF). Recall that in our definition of PAC algorithms we did not insist that when learning a target concept from F_n the algorithm must output a hypothesis in F_n. However, we did make this condition when defining an Occam algorithm.

Claim 25.3.4 *If $\mathcal{NP}$ is not contained in $\mathcal{BPP}$ then no probabilistic polynomial-time that outputs a hypothesis in 3-term DNF can learn the class of 3-term DNF formulae.*

Proof: We show a randomized polynomial-time reduction of the problem of finding a 3-term DNF that is consistent with a given set of examples to the problem of PAC learning 3-term DNF's via such hypotheses.

Let L be a PAC learning algorithm of the latter form, and suppose we are given a set of m instances, denoted S, labeled by some 3-term DNF, denoted f. We invoke algorithm L on input parameters $\epsilon = 1/2m$ and $\delta = 1/3$, and feed it with a sequence of (labeled) examples uniformly distributed in S. (This sequence may contain repetirtions.) Thus, L is running with distribution D which is uniform on S, and outputs a 3-term DNF hypothesis h satisfying

$$\Pr\left[err_{f,D}(h) \geq \frac{1}{2m}\right] \leq \frac{1}{3}$$

Since each $s \in S$ has probability mass $1/m$, it follows that

$$\Pr[\exists s \in S \text{ s.t. } h(s) \neq f(s)] \leq \frac{1}{3}$$

Thus, with probability $2/3$ the hypothesis h is consistent with f on S. Invoking Claim 25.3.3, the current claim follows. ■

Discussion: What the last claim says is that if we insist that the learning algorithm outputs a hypothesis in the concept class being learned (as we do in case of Occam's Razor) then we cannot learn 3-term DNF formulae. In the next section, we shall see that the latter class can be learned if we allow the hypothesis class to be different.

25.4 Generalized definition of PAC learning algorithm

Oded's Note: *This section was drastically revised by me.*

In accordance with the above discussion we define explicitly the notion of learning one concept class with a possibly different class of hypotheses, which typically is a superset of the concept class.

Definition 25.5 (PAC learning, revisited): *Let $F = \cup_n F_n$ and $H = \cup_n H_n$ so that $F_n \subseteq H_n$ are classes of functions mapping X_n to $\{0,1\}$. We say that algorithm A PAC* learns the concept class *F* using the hypothesis class *H if the following holds: For every n, for every function $f \in F_n$, for every distribution $D : X_n \longrightarrow [0,1]$, for every error parameter $0 < \epsilon < 1$ and for every confidence parameter $0 < \delta < 1$, given parameters n, ϵ, δ and a set of f-labeled examples, $\{< x^i, f(x^i) >\}$, where x^i is drawn independently under the distribution D, the algorithm outputs a hypothesis $h \in H_n$ such that:*

$$\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$$

where the probability is taken over the choice of examples, as well as over the internal coin tosses of the algorithm.

That is, the only change relative to Definition 25.2 is the condition that the output hypothesis h belongs to H_n. In case $H = F$ we say that the algorithm is a proper learning algorithm for F.

In contrast to the negative results in the previous section, we show that the class 3-term DNF can be efficiently learned using the hypothesis class 3CNF. This statement is proven via a reduction of this learning task to the task of learning monomials (already solved efficiently above). To present this reduction, we first define what we mean in general by reduction among learning tasks.

25.4.1 Reductions among learning tasks

Reductions are a powerful tool common in computer science models. It is only natural to define such notion for the model of PAC learning.

Definition 25.6 *We say that the concept class F over the instance space X, is PAC-reducible to the concept class F' over the instance X' if for some polynomial p*

- *there exists a polynomial-time computable mapping G from X to X' such that for every n and for every $x \in X_n$, $\quad G(x) \in X_{p(n)}$.*
- *there exists a polynomial $q()$ such that for every concept $f \in F_n$, there exists a concept $f' \in F'_{p(n)}$ such that $size(c') \leq q(size(c))$ and for every $x \in X_n$, $\ f(x) = f'(G(x))$.*

Note that the second item does not require an efficient transformation between f and f'. In fact, for proper learning (by efficient algorithms), an efficient transformation from F' to F must be required.

Theorem 25.7 *Let F and F' be concept classes. If F is PAC-reducible to F', and F' is efficiently PAC learnable, then F is efficiently PAC learnable.*

Proof: Given an efficient PAC learning algorithm L'. We use L' to learn a an unknown target concept $f \in F_n$ as follows: Given an example labeled $< x, f(x) >$, where $x \in X_n$, we compute an example labeled $< G(x), f(x) >$, where $G(x) \in X'_n$, and supply it to L'. The original examples are chosen by a distribution D on X, and so the reduced examples $G(x)$ (computed by us) are

drawn by distribution D' on X' induced by ditribution D. Also, let f' be the function associated by Item 2 to f (i.e., $f'(G(x)) = f(x)$). Algorithm L' will output an hypothesis h' such that:

$$\Pr[err_{f',D'}(h') \leq \epsilon] \geq 1 - \delta$$

where the probability depends on a sample of D'. We take the composition of h' and G to be our hypothesis h for f (i.e., $h(x) = h'(G(x))$).

We need to evaluate $\Pr[err_{f,D}(h) \leq \epsilon]$, where $h = h' \circ G$ and h' is the output of L'. We first observe that

$$\begin{aligned} err_{f,D}(h) &= \Pr_{x \sim D}[h(x) \neq f(x)] \\ &= \Pr_{x \sim D}[h'(G(x)) \neq f'(G(x))] \\ &= \Pr_{x' \sim D'}[h'(x') \neq f'(x')] \\ &= err_{f',D'}(h') \end{aligned}$$

and so $\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$, as required. ■

Learning a 3-term DNF, revisited. Recall that, assumping $\mathcal{NP}$ is not contained in $\mathcal{BPP}$, it is infeasible to *properly* learn 3-term DNF's (i.e., learn this class by hypotheses in it). In contrast, we now show that it is feasible to learn 3-term DNF's by 3CNF hypotheses. Actually, we show that it is feasible to (properly) learn the class of 3CNF (which contain via a easy reduction all 3-term DNF's).[1] This is shown by reducting the learning of 3-CNF's to the learning of monomials.

Claim 25.4.1 *Learning 3-CNF's is reducible to learning monomials. Furthermore, there exists an efficient algorithm for* properly *learning 3-CNF.*

Proof: We define the following transformation G from 3-CNF instance space X_n, to the monomial instance space X'_k, where $k = 8 \cdot \binom{n^3}{3}$. For each of the possible k clauses, we associate a distinct variable, and the transformation from 3CNF to monomials just maps the set of clauses in the 3CNF into the set of variables. (Indeed, we reduce to learning monotone monomials.) The transformation G does the analogous thing; that is, it maps truth assignments to the n 3NCF variables onto truth assignments to the k monomial varaibles in the natural way (i.e., a monomial variable representing a possible clause is assigned the value to which this clause evaluates under the assignment to the 3CNF variables). This transformation satisfies the conditions of Definition 25.6.

To show the furthermore-part, observe that the hypothesis constructed by the reduction-algorithm (given in the proof of Theorem 25.7) can be readily put in 3CNF with respect to the space X_n. ■

25.4.2 Generalized forms of Occam's Razor

We have seen that it is worthwhile to use a hypothesis from a wider class of functions (than merely from the concept class). A staightfoward generalization of Occam's Razor to a case where the algorithm outputs a hypothesis from a predetermined hypothesis class follows.

Theorem 25.8 (Occam's Razor – generalization to predetermined hypothesis class): *Let $F_n \subseteq H_n$ be finite concept classes. Let A be an algorithm such that for every n, for every $f \in F_n$ and for every*

[1] Applying the distributional law to a 3-term DNF over n variables, we obtain a 3CNF with at most $(2n)^3$ clauses.

number of examples labeled by f, the algorithm outputs an hypothesis $h \in H_n$, that is consistent with all the given examples. Then for any distribution D, for any error parameter $0 < \epsilon < 1$ and for any confidence parameter $0 < \delta < 1$ if the number of examples drawn independently by D is larger or equal $\frac{1}{\epsilon}(\log \frac{1}{\delta} + \log |H_n|)$, then

$$\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$$

The proof is by a straightforward adaptation of the proof given to Theorem 25.4 (which is indeed a special case obtained by setting $H_n = F_n$).

A wider generalization is obtained by not determining a priori the hypothesis class. In such case, we need some other way to bound the hypothesis from merely recording all examples. This is done by requiring that the hypothesis's length is strictly shorter than the number of examples seen. This leads to the following formulation.

Definition 25.9 *Let α and $\beta < 1$ be constants. We say that L is an (α, β)-*Occam's algorithm *for F if given m examples, L outputs a hypothesis h such that h is consistent with every given example and*

$$size(h) \leq (n \cdot size(f))^{\alpha} m^{\beta}$$

Theorem 25.10 *For every α and $\beta < 1$, an (α, β)-Occam's algorithm can be turned into a PAC learning algorithm by running it on $O(\epsilon^{-1} \cdot (\text{poly}(n \cdot size(f)) + \log(1/\delta))$ examples.*

Proof: Firstly, we generalize Theorem 25.8 to a setting in which the Occam algorithm may uses different hypothesis classes for different number of examples. Specifically, suppose that when seeing m examples labelled by $f \in F_n$, the algorithm outputs a hypothesis in $H_{n,m}$. Then such an algorithm is a PAC learner provided

$$m \geq \epsilon^{-1} \cdot (\log(1/\delta) + \log |H_{n,m}|)\,,$$

where ϵ and δ are the error and confidence parameters (given to the PAC version).

Now, the conditions of the current theorem provide such a "dynamic" hypothesis class, $H_{n,m}$, and the upper bound on the length of hypothesis guarantees that

$$\log |H_{n,m}| \leq (n \cdot size(f))^{\alpha} m^{\beta}$$

So, since $\beta < 1$, for sufficiently large $m = \text{poly}(n \cdot size(f))/\epsilon$, we have $\epsilon m \ll (n \cdot size(f))^{\alpha} m^{\beta}$. The theorem follows. ■

25.5 The (VC) Vapnik-Chervonenkis Dimension

In all versions of Occam's Razor discussed above, we assumed that the hypothesis class is finite. In general this is not necessarily the case. For example, the natural hypothesis class for learning axis-aligned rectangles over $[0, 1]^2$ is the infinite set of all possible axis-aligned rectangles. Therefore we can not apply Occam's Razor to this problem. We would like a tool with similar flavor for the case of infinite hypothesis classes. The first step for achieving this goal, is finding a parameter of finite value that characterizing also infinite classes.

Definition 25.11 (shattering a set): *A finite subset S of instance space X_n is* shattered by a family of functions *F_n if for every $S' \subseteq S$ there exists a function $f \in F_n$ such that*

$$f(x) = \begin{cases} 1 & x \in S' \\ 0 & x \in S \backslash S' \end{cases}$$

That is for $S = \{x^1, ..., x^m\}$ and for every $\alpha \in \{0,1\}^m$ there exists a function $f \in F$, such that for any $i = \{1...m\}$, $f(x^i) = \alpha^i$, where α^i is the i^{th} bit of α.

Definition 25.12 (VC dimention): *The* VC dimention of a set of functions *F, denoted VC-dim(F), is the maximal integer d such that there exists a set S of cardinality d that is shattered by F.*

25.5.1 An example: VC dimension of axis aligned rectangles

The following example demonstrates the computation of VC dimension of a family of functions.

Proposition 25.5.1 *VC-dim(axis-aligned rectangles) = 4*

Proof: We start by exhibiting a set of four points in $[0,1]^2$ that can be shattered. For every labeling of the set of four points exhibited in Fig. 25.3, we can find a rectangle that induces such labeling.

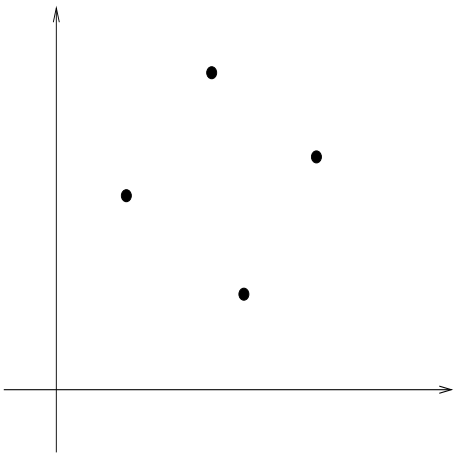

Figure 25.3: Four point that in $[0,1]^2$, that can be shattered

1. In case all points are labeled "+", we take the rectangle $[0,1]^2$ itself.
2. In case all points are labeled "-", we take the empty rectangle.
3. For the case three points are labeled "-" and one point "+", take a "small" rectangle that covers only the point labeled "+".
4. For the case three points are labeled "+" and one point"-", we take rectangles as can be seen in Fig. 25.4.
5. For the case two points are labeled "-" and two points "+", we take rectangles as can be seen in Fig. 25.5. (The figure shows only 4 out of the 6 subcases; the other two are easier.)

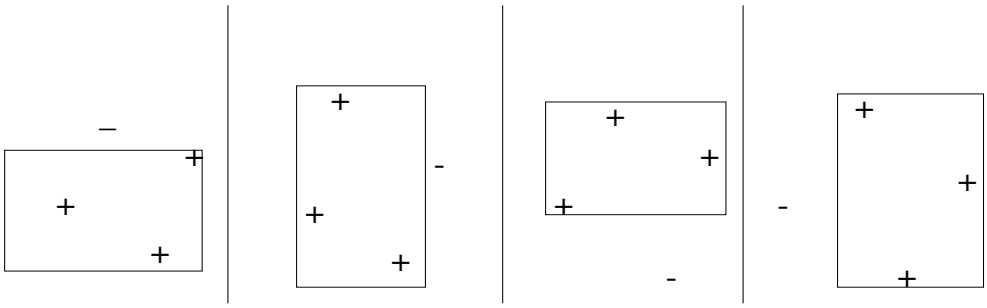

Figure 25.4: rectangle covering the three points labeled "+"

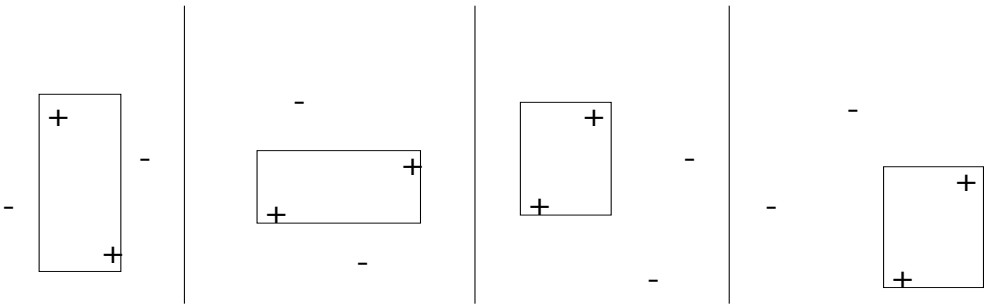

Figure 25.5: rectangle covering the two points labeled "+"

It remains to prove that for every set of five points in $[0,1]^2$, there exists is a labeling, such that it can not be induced by any rectangle in $[0,1]^2$.

By the proof of Claim 25.2.1 (the learning algorithm for axis aligned rectangles), for every set of five points in $[0,1]^2$, at most four of them determine the minimal rectangle that contains the whole set. Then no rectangle is consistent with the labeling that assigns these 4 boundary points '+' and assigns the remaining point (which must reside inside any rectangle coving these points) a '-'. ■

25.5.2 General bounds

VC dimension is linked to the sample complexity of PAC learning via the following two theorems:

Theorem 25.13 (upper bound on sample complexity)*: Let H, F be function classes such that $F \subseteq H$. Let A be a algorithm that given a labeled sample always outputs a hypothesis $h \in H$ consistent with the sample. Then there exists a constant c_0, such that for any target function $f \in F$, for any underlying distribution D, any error parameter $0 < \epsilon < 1$ and any confidence parameter $0 < \delta < 1$, if the number of examples is greater or equal:*

$$\frac{c_0}{\epsilon} \cdot \left(VC\text{-}dim(H) \cdot \log \frac{1}{\epsilon} + \log \frac{1}{\delta} \right)$$

then with probability greater then $1 - \delta$ we get

$$err_{D,f}(h) \leq \epsilon$$

where h is the hypothesis output by A.

Theorem 25.14 (lower bound on sample complexity)*: Any PAC learning algorithm for learning a concept class F such that $d = VC\text{-}dim(F)$ requires $m = \Omega(\frac{1}{\epsilon} \cdot (\log \frac{1}{\delta} + d))$ examples.*

We note that there is a gap of factor $\log \frac{1}{\epsilon}$ between the bounds. The proof of the these theorems is complex. Instead we prove a slightly weaker lower bound theorem:

Theorem 25.15 (lower bound, slightly weaker form): *Any PAC learning algorithm for learning a concept class F such that $d = VC\text{-}dim(F)$ requires requires $\Omega(\frac{d}{\epsilon})$ examples in the worst case.*

Oded's Note: *The proof was revised by me.*

Proof: Since the VC dimension is d, there exists a set of d points shattered by F. We denote this set by $S = \{e^1, ..., e^d\}$. Since S is shattered by F, for each possible labeling of S there exists a function $f \in F$ which is consistent with this labeling. Let us denote by $f_\alpha \in F$ a function consistent with the labeling $\alpha = \alpha_1 \cdots \alpha_d \in \{0,1\}^d$; i.e., $f_\alpha(e^i) = \alpha_i$ for each $i = 1, ..., d$. Let $F' = \{f_\alpha : \alpha \in \{0,1\}^d\} \subseteq F$.

We consider error parameter $\epsilon \leq \frac{1}{8}$, and an arbitrary confidence parameter $0 < \delta < \frac{1}{2}$.

We start by proving a bound of $\Omega(d)$. Towards this goal, we define the underlying distribution D to be uniform over S; that is, every point in S is assigned probability $\frac{1}{d}$, and all other instances are assigned zero probability.

Let us assume, in contrary to the claimed bound, that $\frac{d}{2}$ examples suffice. Under this assumption in the best case, $\frac{d}{2}$ different examples where drawn. We denote this set by S_1, and let $S_2 \stackrel{\text{def}}{=} S \setminus S_1$. Intutively, the algforithm only got the labeling of S_1 and so there is no way it can distinguish between the $2^{d-|S_1|}$ functions in F' consistent with S_1.

Formally, we consider a target function f chosen uniformly in F'. The algorithm then obtain a sample S_1 labeled by f, and outputs a hypothesis h. We are interested in the behavior of the random variable $err_{f,D}(h)$. Recall that this random varaible is defined over the probability space defined by the uniform choice of $f \in F'$, the uniform choice of $d/2$ points in S (yielding S_1), and additional coin tosses the learning algorithm may do. Let us reverse the "natural" order of the randomization, and consider what happens when S_1 is selected first, and $f \in F'$ is selected next. Furthermore, we select f in two phases: first we select at random the value of f on S_1, and we postpone for later the random choice of the value of f on S_2. (Together, the values of f on S_1 and S_2 will determine a unique $f \in F'$, which will be uniformly distributed.) However, the learning algorithm is oblivious of the latter choices (i.e., of the values of f on S_2), and so the hypothesis h is stochastically independent of the latter. So the full order of events we consider is:

1. d points are selected uniformly in S, determining S_1 and S_2.
2. one assigns uniformly Boolean values to the points in S_1 and presents these to the learning algorithm.
3. the learning algorithm outputs a hypothesis h.
4. one assigns uniformly Boolean values to the points in S_2, thus determining a unique function $f \in F'$.

Since f is determined on S_2 only after h is fixed, and its values on S_2 are uniformly distributed in $\{0,1\}$, the expected value of $err_{f,D}(h)$ is at least

$$\frac{1}{2} \cdot \frac{|S_2|}{|S|} \geq \frac{1}{2} \cdot \frac{d/2}{d} = \frac{1}{4}$$

Thus, there exists $f \in F' \subseteq F$ so that with probability at least $1/2$,

$$err_{f,D}(h) \geq \frac{1}{8}$$

This establishes a lower bound of $d/2$ on the sample complexity, for any $\epsilon \leq 1/8$ (and $\delta \leq 1/2$).

In order to prove the stronger bound, stated in the theorem, we modify the distribution D as follows. Fixing an arbitrary element $e^1 \in S$, we let D assign it probability $1 - 8\epsilon$, and assign each other element of S probability $8\epsilon/(d-1)$. (Again, only points in S are assigned non-zero probability.) If we take a sample of m points from D, we expect only $8\epsilon m$ points to be different than e^1. Thus, if $m < (d-1)/(20\epsilon)$ then with very high probability only $10\epsilon m < (d-1)/2$ points will be different than e^1. Applying an argument as above, we conclude that our error with respect to D is expected to be the probability assigned to points not seen by the algorithm times one half; that is,

$$\left(\frac{d-1}{2} \cdot \frac{8\epsilon}{d-1}\right) \cdot \frac{1}{2} = 2 \cdot \epsilon$$

Thus, there exists $f \in F' \subseteq F$ so that with probability at least $1/2$, $err_{f,D}(h) \geq \epsilon$. This establishes a lower bound of $(d-1)/(20\epsilon)$ on the sample complexity, for any $\epsilon \leq 1/8$ (and $\delta \leq 1/2$). ■

Bibliographic Notes

The PAC learning model was introduced by Valiant [3]. Occam's Razor Theorem is due to [1]. The interested reader is referred to a textbook by Kearns and Vazirani [2].

1. A. Blumer, A. Ehrenfeucht, D. Haussler, and M.K. Warmuth. Occam's Razor. *Information Processing Letters*, Vol. 24 (6), pages 377–380, April 1987.

2. M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

3. L.G. Valiant. A Theory of the Learnable. *Communications of the ACM*, Vol. 27 (11), pages 1134–1142, November 1984.

Appendix: Filling-up gaps for the proof of Claim 25.2.1

Oded's Note: *The gap left open in the proof was the assumption that we can slides the border of the A_i's so that the weight of each of them is exactly $\epsilon/4$. Firstly, we repeat the comment by which it is not essential to have these rectanges have weight exactly $\epsilon/4$, and it suffices to have each of the A_i's have weight $\Theta(\epsilon)$. But still it may be that there is a probability mass of $\Omega(\epsilon)$ residing on one single axis-allinged line. This problem can be resolved is several ways. For example, one may pertub all points at random, and argue that the performance of the algorithm cannot be improved (a proof is indeed called for!). Alternatively, one may argue seperately for these patalogical cases of single lines having probability mass of $\Omega(\epsilon)$.*

Lecture 26

Relativization

Notes taken by Leia Passoni

Summary: In this lecture we deal with relativization of complexity classes. In particular, we discuss the role of relativization with respect to the $\mathcal{P} \stackrel{?}{=} NP$ question; that is, we shall see that for some oracle A, $\mathcal{P}^A = \mathcal{NP}^A$ whereas for another A $\mathcal{P}^A \neq \mathcal{NP}^A$. However, it also holds that $\mathcal{IP}^A \neq \mathcal{PSPACE}^A$ for a random A, whereas $\mathcal{IP} = \mathcal{PSPACE}$

Oded's Note: *The study of relativization is motivated by the belief that relativized results indicate limitations of proof techniques which may be applied in the real (unrelativized) world. In the conclusion section we explain why we do not share this belief. In a nutshell, whereas it is useful to refer to proof techniques when discussing and unifying known results, it is misleading to refer to a "proof technique" as if it were a domain with well-defined boundaries (and speculate on which results are beyond such boundaries).*

In contrast, we see benefit in an attempt to define frameworks for proving certain results, and discuss properties of proofs within such well-defined frameworks (e.g., proofs of certain well-defined properties cannot prove a particular result or results of certain well-defined form). In fact, our original intention was to present such a framework, called Natural Proofs, in this lecture. This intention was abandoned since we did not see circuit size lower bounds in the course, and such proofs are the context of Natural Proofs.

26.1 Relativization of Complexity Classes

We have already mentioned the use of polynomial time machines that have access to some oracle in order to define the Polynomial Hierarchy (PH).

Given any two compexity classes C_1 and C_2, were C_2 is the class of oracles, it is not always possible to have a natural notion of what is the class $C_1^{C_2}$ – it is not necessarily the case that for every complexity class C_1 we can define its *relativization* to C_2.

There are some conditions under which such a relativization can be done. One of those conditions seems to be that the complexity class C_1 has to be defined in terms of some type of machines – that is, for any language L in C_1 there exists a machine M_L such that $L = L(M_L)$. Furthermore, the definition of M_L has to be extendable to the definition of *oracle machine*.

Oracle Machines. We consider three types of oracle machines:

Definition 26.1 *A (deterministic/nondeterministic/probabilistic) polynomial time* oracle machine *is a machine with a distinguished work tape – the query tape, where the machine may make oracle queries. These queries are answered by a function called the* oracle.

Some notations: $M^f(x)$ denotes a computation of M on input x when given access to oracle f. When the machine writes on the oracle tape a query q (and invokes the oracle), then the tape's contents q is replaced by $f(q)$.

We also write $M^A(x)$, where the oracle A is a language. In this case M is given access to the characteristic function of the language, χ_A.

26.2 The $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ question Relativized

Trying to solve the $\mathcal{P} \neq \mathcal{NP}$ conjecture, the corresponding relativized conjecture has been investigated, that is, determining whether $\mathcal{P}^A \neq \mathcal{NP}^A$ or $\mathcal{P}^A = \mathcal{NP}^A$ for some oracle A. Thus, we ask which of the following two possibilities holds:

- There exists an oracle A such that $\mathcal{P}^A = \mathcal{NP}^A$
- There exists an oracle A such that $\mathcal{P}^A \neq \mathcal{NP}^A$

Recall that

$$\mathcal{P}^A \stackrel{\text{def}}{=} \{L : \exists \ \text{deterministic} \ \ \text{poly} - \text{time} \ \ M : L(M^A) = L\}$$

$\mathcal{P}^A$ is then a class of languages: from a countable set of machines M we get a countable set of oracle machines M^A that run in polynomial time, each having oracle access to A.

Note that if $A = \emptyset$ or $A = \{0,1\}^*$, then $\mathcal{P}^A = \mathcal{P}$. In fact, if the answers given to queries to the oracle are all "yes" or all "no", then any language in $\mathcal{P}^A$ can be accepted by a deterministic polynomial time machine, just committing the oracle questins and aswers.
Also, $A \in \mathcal{P} \Rightarrow \mathcal{P}^A = \mathcal{P}$: in this case a P machine can simulate the access to the oracle in polynomial time.
Therefore, it is worthwhile to use oracles machines when the oracle A defines a language which is "complex".

In the same way $\mathcal{NP}^A$ can be defined:

$$\mathcal{NP}^A \stackrel{\text{def}}{=} \{L : \exists \ \text{nondeterministic} \ \ \text{poly} - \text{time} \ \ M : L(M^A) = L\}$$

It could seem that if $\exists A : \mathcal{P}^A \neq \mathcal{NP}^A$ then $\mathcal{P} \neq \mathcal{NP}$, but this is not the case. Of course it would be correct to assert that if $\forall A \ \ \mathcal{P}^A \neq \mathcal{NP}^A$ then $\mathcal{P} \neq \mathcal{NP}$, because in this case it would be possible to consider a trivial oracle like $A = \emptyset$ or $A = \{0,1\}^*$.

Going back to the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ question, a possible reason for investigating its relativized version is the hope that a result obtained for a class of oracles would shed light on the unrelativized question. The two following theorems show that this hope fails.

Theorem 26.2 *An oracle A exists such that $\mathcal{P}^A = \mathcal{NP}^A$*

Proof: It is obvious that $\forall A \;\; \mathcal{P}^A \subseteq \mathcal{NP}^A$. It is left to show that $\exists A : \mathcal{NP}^A \subseteq \mathcal{P}^A$.

The idea of the proof is to choose as oracle A a $\mathcal{PSPACE}$-complete language, that is, a language so "powerful" that no more advantage is left for a nondeterministic machine over a deterministic one if they both have access to oracle A.

We first show that $\mathcal{NP}^{\mathcal{PSPACE}} \subseteq \mathcal{PSPACE}$. Let A be a $\mathcal{PSPACE}$ language and let $L \in \mathcal{NP}^A$. Then there exists a nondeterministic poly-time machine M such that $L = L(M^A)$. Equivalently, there exists a deterministic poly-time machine $\bar{M}$ such that $\forall x \in L, \exists y$ and polynomial $p_1 : |y| \leq p_1(|x|)$ and such that $\bar{M}^A(x, y)$ accepts. It is possible then to derive a deterministic machine M' such that $M'(x)$ accepts using at most $p_1(|x|)$ cells of space. Machine M' cycles throughout all possible y's of length $\leq p_1(|x|)$, simulating $\bar{M}^A(x, y)$ for each such y. In this simulation, when $\bar{M}$ queries the oracle on a word w, machine M' simulates the oracle by determining whether $w \in A$. Note that $|w| \leq p_2(|x|)$, where $p_2(|x|)$ is the polynomial bound on the number of steps in which $\bar{M}$ accepts the input x. Since $A \in \mathcal{PSPACE}$ it follows that determining $w \in A$ can be done in space $poly(|w|) = poly(|x|)$. So, M' can decide $L \in \mathcal{NP}^A$ using in all $p(|x|)$ space cells for some polynomial p. We conclude that

$$\text{for any } A \in \mathcal{PSPACE}, \quad \mathcal{NP}^A \subseteq \mathcal{PSPACE}. \tag{26.1}$$

We now show that for some $A \in \mathcal{PSPACE}$ (specifically, any $\mathcal{PSPACE}$-complete A will do), $\mathcal{PSPACE} \subseteq \mathcal{P}^A$. Consider a $\mathcal{PSPACE}$-complete language, A. For any $\bar{L} \in \mathcal{PSPACE}$, consider the Cook-reduction of $\bar{L}$ to A; that is, a polynomial-time oracle machine R that given access to oracle A decides $\bar{L}$. But this means that $\bar{L} = L(R^A) \in \mathcal{P}^A$ (which also equals $\mathcal{P}^{\mathcal{PSPACE}}$). Thus,

$$\text{for some } A \in \mathcal{PSPACE}, \quad \mathcal{PSPACE} \subseteq \mathcal{P}^A. \tag{26.2}$$

Combining Equations (26.1) and (26.2), we conclude that there exists an oracle A such that

$$\mathcal{NP}^A \subseteq \mathcal{PSPACE} \subseteq \mathcal{P}^A$$

and the theorem follows. ■

In spite of the result of Theorem 26.2, the oracle A can be chosen in such a way that the two relativized classes $\mathcal{P}^A$ and $\mathcal{NP}^A$ are different.

Theorem 26.3 *An oracle A exists such that $\mathcal{P}^A \neq \mathcal{NP}^A$*

Proof: Since $\mathcal{P}^A \subseteq \mathcal{NP}^A$, we want to find a language L such that $L \in \mathcal{NP}^A \setminus \mathcal{P}^A$, that is, a language that separates the two classes.
In this case the idea-technique for getting the separation is to define the language using the oracle A. Thus, for every oracle A we define

$$L_A \stackrel{\text{def}}{=} \{1^n : \exists w \in \{0,1\}^n : w \in A\}$$

What is peculiar about language L_A is that it describes whether $A \cap \{0,1\}^n$ is a nonempty set. It can be seen that $L_A \in \mathcal{NP}^A$, by showing a nondeterministic machine with access to A that accepts L_A:

On input $x \in \{0,1\}^n$:

1. if $x \neq 1^n$, then reject

2. otherwise guess $w \in \{0,1\}^n$
3. accept iff $w \in A$

If $x = 1^n \in L_A$, there exists a computation of the machine that accepts – the computation that succeeds in guessing a proper w which is checked to be in A by an oracle query to A. Otherwise, if $1^n \notin L_A$, there is no $w \in \{0,1\}^n$ so that $w \in A$, and therefore the machine can never accept; so in this case, no matter what guess it makes, the machine always rejects.

Now we have to show the existence of an oracle A so that no deterministic poly-time machine can accept L_A. We first show the following

Claim 26.2.1 *For every deterministic polynomial time M, there exists A so that $L_A \neq L(M^A)$*

Proof: We are given machine M and $p(\cdot)$, the polynomial upper bound on the running time of M and on the number of possible queries M can ask to the oracle.

We then choose n_1 such that $n_1 = min\{m : 2^m > p(m)\}$, and we consider the computation of M^A on input 1^{n_1}. The purpose is to define A in such a way that M will err in its decision whether $1^{n_1} \in A$ or not.

Definition of A: we consider three cases regarding the length $n = |q|$ of a query q: $n < n_1$, $n = n_1$, $n > n_1$.

- If $n < n_1$, we can for instance assume we have already answered "0" to all 2^n possible queries during computations with inputs of length less than n_1. In this case, on input 1^{n_1} we answer consistently "0".

- If $n = n_1$, then we use the fact $2^n > p(n)$, which implies that not all n-bit strings are queried. To all (up to $p(n)$) queries on input 1^{n_1} answer "0". What about the queries which M didn't ask? We will define A in such a way that

  - If $M^A(1^{n_1})$ accepts then let $A \cup \{0,1\}^{n_1} = \emptyset$. So we don't put in A any word of length n_1, i.e. we answer "0" to all remaining queries;
  - If $M^A(1^{n_1})$ rejects then let $A \cup \{0,1\}^{n_1} = \{\bar{w}\}$, where $\bar{w}$ is one of the words of length n_1 that have not been queried by M.

  We stress that $M^A(1^{n_1})$ does not depend on whether $\bar{w} \in A$ (since $\bar{w}$ is not queried in this compuation), but $1^n \in L_A$ does depend on whether $\bar{w} \in A$.

- If $n > n_1$. Like before, answer "0". But in this case we are committing the oracle answers on longer strings.

  Anyway, this is not a problem because any $n > n_1$ has to be $n \leq p(n_1)$, otherwise M wouldn't have the time to write the query. Thus, given n_1 there is a finite number of possible lengths of queries $n : n > n_1$, so we can carry on at this stage the construction for these longer n's. (This extra discussion is only important for the extension below.)

Oracle A is thus defined in such a way that, on input 1^{n_1} machine M^A cannot decide whether 1^{n_1} is in L_A or not. ■

Anyway, Claim 26.2.1 is not enough to prove Theorem 26.3, since the oracle A in the claim is possibly different for any M. We need to state a stronger claim. That is

$$\exists A : \forall \text{ deterministic polynomial-time } M, \ L_A \neq L(M^A)$$

This claim is provable by extending the same argument used in Claim 26.2.1. The idea is as follows: first we enumerate all deterministic poly-time machines

$$\begin{array}{cccccc} M_1, & M_2, & \ldots, & M_i, & \ldots \\ \downarrow & \downarrow & & \downarrow & \\ n_1, & n_2, & \ldots, & n_i, & \ldots \end{array}$$

and to each machine M_i we associate a "large enough" integer n_i. The oracle A is built in stages in such a way that, at any step we deal with a portion $A(i-1) \subseteq \{0,1\}^{n_i - 1}$ of the oracle. Then $A = \cup_{i>0} A(i)$ and $M_i^A(1^{n_i}) = M_i^{A(i-1)}(1^{n_i})$. The oracle A is such that, at any step i, M_i^A errs in decision on input 1^{n_i}.

Define $A(i) = \{$words that have been placed in A after step $i\}$. We let $A(0) = \emptyset$ and $n_0 = 0$. Consider machine M_1 and its polynomial bound $p_1(\cdot)$. Choose n_1 such that

$$n_1 = min\{m : 2^m > p_1(m)\}$$

At this point $A = A(0) = \emptyset$. To build $A(1)$ we follow the same procedure used in Claim 26.2.1; in particular

- if $M_1^A(1^{n_1})$ accepts then $A(1) = A(0)$;
- if $M_1^A(1^{n_1})$ rejects then $A(1) = A(0) \cup \{w_{n_1}\}$, where w_{n_1} is one of the words of length n_1 that have not been queried by M_1.

At stage i, consider machine M_i and $p_i(\cdot)$. Choose n_i such that

$$n_i = min\{m : 2^m > p_i(m) \text{ and } (\forall j < i),\ m > p_j(n_j)\}$$

Currently A equals $A(i-1)$; so when on input 1^{n_i} machine M_i asks queries of length less than n_i, the anwers have to be consistent with what has previously defined to be in the oracle. Like before, we then build $A(i)$ such that

- if $M_i^A(1^{n_i})$ accepts then $A(i) = A(i-1)$;
- if $M_i^A(1^{n_i})$ rejects then $A(i) = A(i-1) \cup \{w_{n_i}\}$, where w_{n_i} is one of the words of length n_i that have not been queried by M_i.

The word w_{n_i} possibly added to A at this stage is one of those not queried by M_i on input 1^{n_i}; moreover, words possibly added in successive stages have length greater than $p_j(n_j)$ for every $j < i$. This ensures that $M_j^{A(i)}(1^{n_j}) = M_j^{A(j-1)}(1^{n_j})$ for every $j < i$.

The oracle A is then built in such a way that no polynomial time machine M with oracle A exists such that $L(M^A) = L_A$ – every M_i^A fails deciding on input 1^{n_i}.

It is important to note that this construction works only on a specific portion of the oracle A at every stage, keeping all other portions of A untouched. ■

Considering the results of Theorem 26.2 and Theorem 26.3, we can understand that the investigation of the relativized $\mathcal{P} \neq \mathcal{NP}$ conjecture cannot possibly help in proving or disproving the unrelativized one. What instead should be taken from these results is the fact that the $\mathcal{P} \neq \mathcal{NP}$ conjecture should be investigated with *proof techniques that do not relativize*, that is, proof thechniques that cannot be extended to oracles – so the simulation and diagonalization thechniques used in the above theorems do not seem to be adequate for investigating the conjecture.

26.3 Relativization with a Random Oracle

Theorem 26.3 can be somehow generalized; in fact it is possible to prove that

$$Prob_A[\mathcal{P}^A \neq \mathcal{NP}^A] = 1 \tag{26.3}$$

The probabilily is taken over all oracles A; Eq. (26.3) means that for almost all oracles A, $\mathcal{P} \neq \mathcal{NP}$ holds. Since $\mathcal{P}^A \subseteq \mathcal{BPP}^A$ for all A, Eq. (26.3) is a consequence of the following

Theorem 26.4 *For a random oracle A, $\mathcal{NP}^A \not\subseteq \mathcal{BPP}^A$, that is, $Prob_A[\mathcal{NP}^A \not\subseteq \mathcal{BPP}^A] = 1$*

Proof: We have to find a language L such that $L \in \mathcal{NP}^A$ but $L \notin BPP^A$ with probability 1. Again, the idea is to define L_A depending on the oracle A:

$$L_A \stackrel{\text{def}}{=} \{1^n : \exists w \in \{0,1\}^n : \forall u \in C_n, wu \in A\}$$

Here C_n is defined to be any canonical set of n strings of length n. For instance, we can take C_n to be the set of all strings of Hamming weight 1: $C_n \stackrel{\text{def}}{=} \{10\ldots0, 010\ldots0, \ldots, 0\ldots01\}$.

> Oded's Note: *The idea behind the use of C_n is to associate with each $w \in \{0,1\}^n$ a coin with odds 2^{-n} of being 1 (and being zero otherwise). If our oracle were to be chosen so that each string $w \in \{0,1\}^n$ is in it with probability 2^{-n} then the extra complication of using C_n would not have been needed. However, the oracle is chosen so that each string is equally likely to be or not be in it, and so we need to implement a coin with odds 2^{-n} of being 1 by using an unbiased coin. That's exactly what the above construction does. The aim of all this is to get to a situation that $1^n \notin L_A$ and "$1^n \in L_A$ via a single witness w" are about as likely, and are also quite likely.*

Again, it can be seen that $L_A \in \mathcal{NP}^A$ by showing a nondeterministic machine with access to A that generates L_A:

On input $x \in \{0,1\}^n$:

1. if $x \neq 1^n$, then reject
2. otherwise guess $w \in \{0,1\}^n$
3. accept iff $\forall u \in C_n, wu \in A$

Here the machine makes n queries to A and accepts iff all of them are anwered "yes" by the oracle.

Now for a randomly chosen A it can be shown that

$$Prob_A[1^n \notin L_A] = Prob_A[\nexists w : \forall u \in C_n, wu \in A] = 1/\mathrm{e} \;\; = \text{ constant} \tag{26.4}$$

In fact, if w and u are fixed, then $Prob_A[wu \in A] = 1/2$ because A is chosen at random, thus every query has probability $1/2$ to be in A or not.

If only w is fixed then $Prob_A[\forall u \in C_n, wu \in A] = 2^{-n}$ since all n events $[wu \in A]$ are independent.

So for this fixed w, $Prob_A[\neg\forall u \in C_n, wu \in A] = 1 - Prob_A[\forall u \in C_n, wu \notin A] = 1 - 2^{-n}$.

Again, there are 2^n possible w and all events are independent, thus $Prob_A[1^n \notin L_A] = Prob_A[\forall w, \neg\forall u \in C_n, wu \in A] = (1-2^{-n})^{2^n} = 1/\mathrm{e} = \text{constant}$.

In a similar way it can also be shown that

$$Prob_A[\exists! w \in \{0,1\}^n : \forall u \in C_n, wu \in A] \approx 1/\mathrm{e} \;\; = \;\; \text{constant} \tag{26.5}$$

Language L_A was then defined in such a way that

- with constant probability, $1^n \notin A$
- with constant probability, there exists a unique witness for $1^n \in A$ i.e. $\exists! w : \forall u \in C_n, wu \in A$

Intuitively, there is therefore a very small probability for any probabilistic polynomial time machine to come across the unique w so that $\forall u \in C_n, wu \in A$. In fact the following can be proven:

Claim 26.3.1 *Let M be a probabilistic polynomial time oracle machine. Then, for sufficiently large n*

$$Prob_A[M^A(1^n) = \chi_{L_A}(1^n)] < 0.8$$

Remarks: What we actually want to achieve is $Prob_A[L(M^A) = L_A] = 0$; by Claim 3.1 we can make the probability that M^A decides correctly on all inputs 1^n (i.e. for all n) vanish exponentially with the number of n's we consider.
Moreover, we want Claim 3.1 to hold for any possible M, that is $Prob_A[\mathcal{BPP}^A \ni L_A] = 0$. This extension can be carried out similarly to what was done in Theorem 26.3.

Proof: We choose n large enough so that the running time of $M^A(1^n)$ which equals $poly(n) \ll 2^n$. Now consider the case in which on input 1^n machine M makes queries of length $2n$ (other cases can be carried as it has been done in Theorem 26.3). We evaluate

$$\begin{aligned}
Prob_A[M^A(1^n) = \chi_{L_A}(1^n)] &= P_{\text{no}} + P_{\text{uni}} + P_{\text{rest}} \text{ , where} \\
P_{\text{no}} &\stackrel{\text{def}}{=} Prob_A[\chi_{L_A}(1^n) = 0] \cdot Prob_A[M^A(1^n) = 0 | \chi_{L_A}(1^n) = 0] \\
P_{\text{uni}} &\stackrel{\text{def}}{=} Prob_A[\exists! w : \forall u \in C_n, wu \in A] \\
&\qquad \cdot Prob_A[M^A(1^n) = 1 | \exists! w : \forall u \in C_n, wu \in A] \\
P_{\text{rest}} &\stackrel{\text{def}}{=} Prob_A[\chi_{L_A}(1^n) = 1 \text{ and } \exists w' \neq w : \forall u \in C_n,\, wu \in A \,\&\, w'u \in A]
\end{aligned}$$

(That is, P_{no} represents the probability that M is correct on an input not in L_A, P_{uni} the probability M is correct on an input with a unique witness, and P_{rest} the that M is correct on an input with several witnesses.) Looking at the various probabilities, we have:

1. By Eq. (26.4), $Prob_A[\chi_{L_A}(1^n) = 0] = 1/\mathrm{e}$,
2. By Eq. (26.5), $Prob_A[\exists! w : \forall u \in C_n, wu \in A] \approx 1/\mathrm{e}$.

So

$$P_{\text{rest}} \;\leq\; Prob_A[\exists w' \neq w : \forall u \in C_n,\, wu \in A \,\&\, w'u \in A] \approx 1 - 2/\mathrm{e} \tag{26.6}$$

It is possible to see that the difference between the two probabilities

a) $p_1 \stackrel{\text{def}}{=} Prob_A[M^A(1^n) = 1 | \chi_{L_A}(1^n) = 0]$

b) $p_2 \stackrel{\text{def}}{=} Prob_A[M^A(1^n) = 1 | \exists! w : \forall u \in C_n, wu \in A]$

is very small. In fact, we can define the following oracle:

$A^{(w_0)} \stackrel{\text{def}}{=}$
- randomly choose A so that $1^n \notin L_A$ meaning $\forall w \in \{0,1\}^n \ \ \exists u \in C_n : wu \notin A$
- only on w_0 modify A so that $\forall u \in C_n, w_0 u \in A$

Then it can be seen that the probability – taken over a random A and a random selceted w of length n

b') $Prob_{A, w \in \{0,1\}^n}[M^{A^{(w)}}(1^n) = 1] = p_2$

In fact, consider the process of selecting A at random such that there exists no w for which $\forall u \in C_n, wu \in A$. Then select w_0 at random and modify A so that only in w_0 $\forall u \in C_n, w_0 u \in A$. This process is the same as selecting a random A conditioned to the existence of a unique w such that $\forall u \in C_n, wu \in A$. So the set of oracles $A^{(w)}$ considered in **b')** has the same distribution as the set of random oracles conditioned to the existence of a unique w for which $\forall u \in C_n, wu \in A$, that is considered in **b)**.

Probabilities **a)** and **b')** are then very close; in fact in **b')**, when the machine asks the first query, there is a chance only of 2^{-n} for the machine to come across the unique w such that $\forall u \in C_n, wu \in A$. (On all other w's the two oracles behave the same.) Asking the second query, there is a chance only of $1/(2^n - 1)$ for the machine to get different answers to the query, and so on. Since the total number of possible queries in n is $p(n)$ for some polynomial p, the difference $p_1 - p_2$ is approximately $\frac{p(n)}{2^n} \ll 1$ for the appropriate n that has been chosen.
The fact that these two probabilities are so close means that there is a very small chance that the machine can notice the modification that has been introduced.

Thus, since $Prob_A[M^A(1^n) = 0 | \chi_{L_A}(1^n) = 0] = 1 - Prob_A[M^A(1^n) = 1 | \chi_{L_A}(1^n) = 0] = 1 - p_1$, we see that

$$\begin{aligned} Prob_A[M^A(1^n) = \chi_{L_A}(1^n)] &= P_{\text{no}} + P_{\text{uni}} + P_{\text{rest}} \\ &\leq Prob_A[\chi_{L_A}(1^n) = 0] \cdot (1 - p_1) \\ &\quad + Prob_A[\exists! w : \forall u \in C_n, wu \in A] \cdot p_2 \ + \ (1 - 2\mathrm{e}^{-1}) \\ &\approx \mathrm{e}^{-1} \cdot (1 - p_1) + \mathrm{e}^{-1} \cdot p_2 + 1 - 2\mathrm{e}^{-1} \\ &= \mathrm{e}^{-1} \cdot (p_2 - p_1) + 1 - \mathrm{e}^{-1} \end{aligned}$$

Since the term $\mathrm{e}^{-1}(p_2 - p_1) < p(n)/2^n$ is negligible, we get

$$Prob_A[M^A(1^n) = \chi_{L_A}(1^n)] \leq 1 - \mathrm{e}^{-1} < 0.8$$

and the claim follows. ■

This also completes the proof of Theorem 26.4. ■

The reason for investigating how $\mathcal{P}$ and $\mathcal{NP}$ behave if relativized to a random oracle is the following. The statements in Theorems 26.2 and 26.3 only mean that both $\mathcal{P} = \mathcal{NP}$ and $\mathcal{P} \neq \mathcal{NP}$ are

supported by *at least* one oracle. This cannot possibly give evidence towards, say, $\mathcal{P} \neq \mathcal{NP}$. But then, what Theorem 26.4 says is that, considering all possible oracles, only a negligible fraction of them supports $\mathcal{P} = \mathcal{NP}$. So it seem plausible that such an assertion supports the $\mathcal{P} \neq \mathcal{NP}$ conjecture. The underlying reasoning, if valid, should extend to any pair of complexity classes. It is known as the *Random Oracle Hypothesis* and asserts that two complexity classes are different if and only if they differ under relativization to a random oracle. Unfortunately, the Random Oracle Hypothesis turns out to be false. For example, the following can be proven:

Theorem 26.5 *For a random oracle A, $\mathrm{co}\mathcal{NP}^A \not\subseteq \mathcal{IP}^A$, that is, $Prob_A[\mathrm{co}\mathcal{NP}^A \not\subseteq \mathcal{IP}^A] = 1$*

Since it is known that $\mathrm{co}\mathcal{NP} \subseteq \mathcal{IP}$, Theorem 26.5 indicates that the Random Oracle Hypothesis fails. Thus, the random-relativization separation result of Theorem 26.4 cannot say anything about separation of the unrelativized classes.

26.4 Conclusions

> Oded's Note: *This section was written by me. Although my main motivation for giving this lecture was to present these views, and although I did express them in class, little of them found their way to the above notes. (Unfortunately, in general, students tend to focus on technical material and attach less importance to conceptual discussions. At times even high-level discussions of proof ideas are gives less attention than some very low-level details.)*

The study of relativized complexity classes was initiated with the hope that it may shed some light on the unrelativized classes. Our own opinion is that this hope is highly unjustified.

Relativized results as predictors of non-relativized results. The most naive hope regarding relativized complexity classes is that the relations which hold in *some* relativized world are exactly those holding in the real (unrelativized) world. As shown above, this cannot possibly be true since different oracles may lead to different relativized results. That is, there may be a relativized world (i.e., an oracle) in which some relation holds and another in which the relation does not hold; whereas the same situation can not possibly hold in the (single) real world.

Conflicting relativizations cannot occur when one considers relativization results which hold for almost all orcales (i.e., relative to a random oracle). A new hope postulated by the "Random Oracle Conjecture" was that relations which hold in *almost all* relativized world are exactly those holding in the real (unrelativized) world. However, this conjecture has been refuted; we have mentioned above one such refutation (i.e., $\mathrm{co}\mathcal{NP} \subseteq \mathcal{IP}$ and yet $\mathrm{co}\mathcal{NP}^A \not\subseteq \mathcal{IP}^A$ for almost all oracles A).

We mention that not only that we have results which contradict the above naive hopes, but also the way these results are proven seems to indicate that the structure of computation of an oracle machine has little to do with the computation of an analogous regular machine. Add to this our initial comment by which not any class can be relativized (i.e., relativization requires that the class be defined in terms of some type of machines and that these machine extend naturally – at least in a syntaxtic sense – to orcale machines). We stress again that such extensions, even when they seem syntaxtically natural, may totally disrupt the "semantics" associated with the bare machine.

Our conclusion is that relativized results are poor predictors of non-relativized results. The advocates of relativization are certainly aware of the above, and their current advocacy is based on the belief that relativized results indicate limitations of proof techniques which may be applied in the real (unrelativized) world. That is –

Relativized results as indicating limitations of proof techniques. The thesis (of the above advocates) is that a relativized result asserting some relation indicates that the contrary relation *cannot be proven* in the real (unrelativized) world *via* "a proof which relativizes". This thesis is based on the hidden assumption that there is a natural (generic) way of extending a real (unrelativized) proof into a version which refers to the corresponding oracle classes. The thesis suggests that one should consider the question of whether such a natural extention is indeed valid in the current case. Suppose that some result fails relative to some oracle, then the thesis asserts that any proof for which the natural relativization remains valid will fail to prove the real (unrelativized) result (since it would have proven the same result also relative to any oracle, whereas the relativized result asserts the existence of an oracle for which the result does not hold).

Our objection to the above thesis is based on the objection to the assumption that such a natural (generic) way of extending a real (unrelativized) proof into a version which refers to the "corresponding oracle classes" exists. Indeed certain known proofs (e.g., the time hierarchy) extend naturally to the relativized world, but we don't see a generic procedure of transforming real (unrelativized) proofs into relativized ones. (Recall our warning that not all complexity classes have a relativized version.) Without such generic procedure of transforming unrelativized proofs into relativized ones, it is not clear what is suggested by the above thesis. Furthermore, one can transform any proof which is known to "relativize" into a proof (of the same statement) that does not "relativize" (via the process witnessing the former "relativization"). Of course this transformation can be undone by a new transformation, but the latter can not be considered generic in any natural sense.

What is left is a suggestion to use relativized results as a tool for testing the viability of certain approaches for proving real (unrelativized) results. That is –

Relativized results as a debugging tool. Suppose one has a vague idea on how to prove some result regarding the real world. Further suppose that one is not sure whether this idea can be carried out. Then the suggestion is to ask whether the same idea seems to apply also to the relativized world and whether a contrary relativization result is known. The thesis is that if the answer to both questions is possitive then one should give-up hope that the idea can work (in the real world).

We object even to this minimalistic suggestion. If one has a vague idea which may or may not work in the real work, possibly depending on unspecified details, then why should one believe in the validity of statements inferred from applying a vague procedure (i.e., "relativization") to this vague idea? Indeed, in some cases the (relativized) indication obtained (as suggested above) may save time which could have been wasted pursuing an unfruitful idea, but in some cases it may cause to abandon a fruitful idea. We see no reason to believe that the probability of the first event is greater than that of the second, and point out that the pay-off (benefit/damage) seems to be much greater in the second.

Bibliographic Notes

The study of relativized complexity classes was initiated in [1], which shows that the P vs NP question has contradicting relativizations. This came as a surprise since in recursive function theory, the origin of the relativization paradigm, all standard results do hold in all relativized worlds. Thus, relativization was taken as an indication that the main technique of recursive function theory –

that is diagonalization – can not settle questions such as P vs NP.[1] The "curse of contradicting relativizations" was eliminated in [2] that considers the measure of oracles (or relativized worlds) for which certain relations hold. The authors also postulated the Random Oracle Hypothesis that essentially states that structural relationships which hold in almost all oracle worlds also hold in the unrelativized case (i.e., the real world). Several refutations of the Random Oracle Hypothesis are known: For example, co$\mathcal{NP} \subseteq \mathcal{IP}$ (see Lecture 11 or [4]), whereas this fails in almost all oracle worlds (see [3]).

A different approach towards investigation of the limitation of certain proofs was taken in [5]. The latter work defines a framework for proving circuit size lower bounds and certain features, called *natural*, of proofs within this framework. It then shows that the existence of natural proofs contradicts the existence of corresponding pseudorandom functions, which in turn implies that the existence of pseudorandom functions cannot be proven via natural proofs.[2]

1. T. Baker, J. Gill, and R. Solovay. Relativizations of the P =? NP question. *SIAM Journal on Computing*, Vol. 4 (4), pages 431–442, December 1975.

2. C. Bennett and J. Gill. Relative to a random oracle A, $\mathcal{P}^A \neq \mathcal{NP}^A \neq \text{co}\mathcal{NP}^A$ with probability 1. *SIAM Journal on Computing*, Vol. 10 (1), pages 96–113, February 1981.

3. R. Chang, B. Chor, O. Goldreich, J. Hartmanis, J. Håstad, D. Ranjan, and P. Rohatgi. The Random Oracle Hypothesis is False. *JCSS*, Vol. 49 (1), pages 24–39, 1994.

4. C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *JACM*, Vol. 39 (4), pages 859–868, 1992.

5. A.R. Razborov and S. Rudich. Natural Proofs. *JCSS*, Vol. 55 (1), pages 24–35, 1997.

[1] We consider the latter sentence to be on the verge on non-sense: It is not clear what is meant by saying that a proof technique cannot settle a certain question. In contrast, saying that a proof with certain well-defined properties cannot settle questions of certain properties may be clear and useful. That is, whereas the notion of a proof technique (i.e., a proof which uses a certain technique) is undefined, one may define properties of proofs. An inspiring case in which the latter was done is the formulation and study of Natural Proofs [5].

[2] Our original intention was to discuss also Natural Proofs in the lecture. However, since the students did not see circuit size lower bounds in this course, the former plan was abandoned.