Rigorous Analysis of Heuristics for NP-hard Problems

Uriel Feige
Weizmann Institute, Israel
uriel.feige@weizmann.ac.il

January 21, 2005

Abstract

The known NP-hardness results imply that for many combinatorial optimization problems there are no efficient algorithms that find an optimal solution, or even a near optimal solution, on every instance. A heuristic for an NP-hard problem is a polynomial time algorithm that produces optimal or near optimal solutions on some input instances, but may fail on others. The study of heuristics involves both an algorithmic issue (the design of the heuristic algorithm) and a conceptual challenge, namely, how does one evaluate the quality of a heuristic. Current methods for evaluating heuristics include experimental evidence, hand waving arguments, and rigorous analysis of the performance of the heuristic on some wide (in a sense that depends on the context) classes of inputs. This talk is concerned with the latter method. On the conceptual side, several frameworks that have been used in order to model the classes of inputs of interest (including random models, semi-random models, smoothed analysis) will be discussed. On the algorithmic side, several algorithmic techniques and principles of analysis that are often useful in these frameworks will be presented.

1 Introduction

Given a computational problem, it is desirable to have algorithms that produce *optimal* results, are *efficient* (polynomial time), and work on *every* input instance. For many combinatorial problems, this goal is too ambitious, as shown by the theory of NP-completeness. Hence one should set

goals that are more modest. Approaches that are tried and have firm theoretical foundations include approximation algorithms (relax the optimality requirement) and fixed parameter tractability (refine the efficiency requirement). We shall discuss a different approach, that of *heuristics*, that relaxes the universality requirement. Here we define a heuristic to be a polynomial time algorithm that produces optimal results on *typical* inputs. The notion of a *typical* input is rather fuzzy, and a major conceptual challenge of the study of heuristics is to give this notion a rigorous meaning.

Some of the research goals of the study of heuristics are the following:

- Explain the apparent success of known heuristics.
- Come up with good heuristic ideas.
- Match heuristics to problems.
- Investigate fundamental limitations of the heuristic methodology.

If we wish to perform a mathematically rigorous study of heuristics, we may want to ask our selves how does one *prove* that a certain heuristic is good, and likewise, how does one *prove* that a certain heuristic is bad.

Here we use the following approach. One should first provide a rigorous definition of what the concept of *typical input* means. Given such a definition (for example, suppose that in some context, a typical graph can be assumed to be a planar graph), one is no longer dealing with the fuzzy notion of heuristics, but with the familiar notion of worst case analysis of algorithms. It will often be the case that we shall model a typical input as an input chosen at random from some well defined distribution. We remark that also in this case (of *average* case analysis), we will typically be performing worst case analysis. The reason for this is that usually analysis of algorithm in random models breaks down into two parts. One first establishes that random inputs are likely to have a certain property P (e.g., random graphs are likely to have very strong expansion properties), and then one shows an algorithm that work on every input that has property P.

2 Some theoretical frameworks

We sketch some theoretical frameworks that have been suggested in order to model typical inputs.

2.1 Random inputs

A good example is the $G_{n,p}$ model of random graphs.

An interesting algorithmic result [7] in this model is that there is a polynomial time algorithm that with high probability finds Hamiltonian a cycle in a random graph, even when p is so small such that the minimum degree in the graph is 2. (When the minimum degree is below 2, then certainly the graph does not have a Hamiltonian cycle.)

On the other hand, there are other NP-hard problems (such as max-clique) for which no polynomial time algorithm is known to produce optimal results on a random graph.

2.2 Planted solution models

When the random model seems too difficult, it may be useful to consider a *planted solution* model. For example, one can plant a clique of large size k in a random graph, and ask how large k can be so that a polynomial time algorithm can detect it. It is known that $k = \Omega(\sqrt{n})$ suffices [2].

2.3 Semi-random models, monotone adversaries

Given a specific random model (or planted solution model), there is danger that algorithms designed for the model will suffer from "over-fitting", and would not work under a slightly different model. To add robustness to algorithms, one may consider semi-random models, first suggested by Blum and Spencer [6].

Here is an example of what the author considers to be over-fitting. When $k \gg \sqrt{n \log n}$, the vertices of a planted k-clique almost surely are those of highest degree in an otherwise random graph. An algorithm may select the k highest degree vertices and check if they form a clique.

A specific version of semi-random models is that of the *monotone adversary* [10]. For example, in the planted clique model, the monotone adversary is allowed to remove arbitrarily many non-clique edges. The degree based algorithm no longer works. Still, more sophisticated algorithms based on semidefinite programming do work, up to $k = \Omega(\sqrt{n})$ [11].

2.4 Smoothed analysis

This model was advocated by Spielman and Teng [18]. The idea is to take an arbitrary input, but then to make a random perturbation to the input. This

may capture well situations where in a typical numerical input, the low order bits are random. Such a model was used in order to offer an explanation for the practical success of the simplex algorithm [18]. For NP-hard problems, it was shown that problems that have fully polynomial time approximation schemes are typically solved in polynomial time on smoothed instances [3].

2.5 Stable inputs

In some applications (such as clustering), the interesting inputs are those that are *stable* in the sense that a small perturbation in the input does not change the combinatorial solution. Bilu and Linial [5] define the notion of stable inputs, and present algorithms that solve NP-hard cut problems whenever the input instance is (highly) stable.

2.6 A comparison

In smoothed analysis, one first picks an arbitrary (worst case) instance. This instance defines a certain region in instance-space (all input instances that can be derived by small perturbations from the original instance). Then, a random input is chosen in this region.

In the monotone adversary model, first an instance is chosen at random, which then defines a region (all instances reachable from the original instance by monotone changes). Thereafter, an arbitrary (worst case) input is chosen in this region.

Hence in a sense, the difference between smoothed analysis and monotone adversary is mainly in the order of quantifiers (forall followed by random versus random followed by forall). In this respect, the monotone adversary model is more difficult.

For stable inputs, the regions in instance-space are determined by the combinatorial solution, rather than by the instance representation. A worst case region is picked, and within it, a worst case input, provided that it is far from the boundary of the region.

3 Algorithmic techniques

Common techniques for designing heuristics in some of the models presented above include detecting statistical irregularities induced by an optimal solution, the use of approximation algorithms, and "hill climbing" once a near optimal solution is found, using the fact that in many of these models near

optimal solutions are necessarily of small Hamming distance from the optimal solution.

3.1 Random 3SAT

We will use the well known problem of 3SAT to demonstrate some of the past rigorous work on analysis of heuristics. We shall consider a random 3CNF input formula f with n variables and m clauses, with $m \gg n$. The expected number of satisfying assignments for f is $(1-1/2^3)^m \cdot 2^n$, implying that when $m \gg n$ the formula is unlikely to be satisfiable.

We shall consider two tasks. One is to search for a satisfying assignment when the formula happens to be satisfiable. The other is to prove non-satisfiability for non-satisfiable formulas (refutation). We remark that for worst case analysis, refutation and search are strongly related (when a search procedure stops without finding a satisfying assignment, this serves as a refutation). For heuristics, we shall see that search and refutation may require very different algorithms.

3.2 Searching for a solution

There are algorithms that appear to very quickly find satisfying assignments in random formulas [8], and it would be very interesting to support this empirical finding by rigorous analysis. We are not able to do so at the moment. Here we present some results that can be proved rigorously.

When $m \gg n \log n$, then if the formula happens to be satisfiable, the satisfying assignment is likely to be unique. It then can be shown that the distribution on random satisfiable formulas can be approximated by the following distribution in the planted solution model.

Pick at random an assignment a to the variables. Choose clauses at random, discarding clauses not satisfied by a, until m clauses are reached. When $m \gg n \log n$, it is likely that a is the unique satisfying assignment.

The planted solution a induces some easily detectable statistical properties. For every variable x, in every clause C that contained x and was discarded, the polarity of x in C disagreed with its polarity in a. Set x according to the polarity that agrees with the majority of the occurrences of x in f. When $m \gg n \log n$, it is likely that this algorithm recovers a.

We now consider the more difficult case of $m = d \cdot n$ for some large constant d. In this case the distribution generated by the planted model is no longer known to be statistically close to that of random satisfiable

formulas. The reason is that the planted model favors formulas with many satisfying assignments. We shall present an algorithm that works in the planted model. It is not known whether this algorithm also extends to the model of random satisfiable formulas.

Given the formula f, start with an initial assignment $a(0)$ that is simply the majority vote assignment (breaking ties arbitrarily). A linear fraction of the variables (exponentially small in d) are likely to be set in disagreement with a, and a linear fraction of clauses are likely not to be satisfied by $a(0)$. We now move to a satisfying assignment by a "hill climbing" procedure. The procedure described here is taken from [13], and its analysis is based on [1, 14]. The procedure itself is a considerable simplification of the procedures described in [1, 14]. This simplification was achieved by following the methodology of considering semi-random inputs (a monotone adversary is allowed to add arbitrary clauses in which all three literals are set in agreement with a), which forces one to make algorithms more robust, and often helps clean away aspects of the algorithm that rely on too detailed statistical properties of the input distribution.

The hill climbing algorithm works it iterations. In each iteration, a local search is performed in order to improve the current assignment. Let $a(j)$ denote the assignment at iteration j, and let $T(j)$ be the set of clauses satisfied by $a(j)$.

Pick an arbitrary clause C not satisfied by $a(j)$. Find the assignment closest (in Hamming distance) to $a(j)$ that satisfies the sub-formula $T(j) \cup C$. Increment j and repeat.

The algorithm obviously finds a satisfying assignment. The only question is how fast.

To analyse the complexity of a single iteration, we let $h(j)$ denote the Hamming distance between $a(j)$ and $a(j+1)$. Since $a(j+1)$ can be reached from $a(j)$ by iteratively flipping variables in currently arbitrary unsatisfied clauses in $T(j) \cup C$, it follows that the time per iteration is at most $3 \cdot 2^{h-1} n^{O(1)}$, which is polynomial when $h = O(\log n)$.

The main technical lemma is that with high probability, in all iterations, $h < O(\log n)$. Hence the algorithm works in polynomial time. The proof of this lemma shows that with high probability, f has a *core* with properties as defined below, and that the algorithm works on *every* formula that has such a core.

A variable x for which $a(0) = a$ is a core variable if flipping it makes at least one clause in $T(0)$ not satisfied, and every assignment in which x is flipped that satisfies $T(0)$ requires flipping a linear number of other variables.

Probabilistic analysis shows that removing the core variables and simplifying the random formula f, it is likely to decompose into small sub-formulas of size $O(\log n)$, on disjoint sets of (non-core) variables.

In any formula (and initial assignment $a(0)$) that has such a core one has the following property. An iteration can be completed by $O(\log n)$ flips of non-core variables. Moreover, as long as $h = O(\log n)$, no core variable will accidently be flipped, and hence the above property is maintained in all iterations.

3.3 Refutation

If a formula is not satisfiable, the heuristic presented above takes exponential time to detect this. Hence we need a different heuristic for refutation.

A general approach for refutation may use approximation algorithms. When $m \gg n$, every assignment satisfies roughly $7m/n$ clauses of a random formula. An algorithm for approximating max-3SAT within a ratio better than $7/8$ would refute most dense 3SAT formulas. Unfortunately, approximating max-3SAT (in the worst case) beyond $7/8$ is NP-hard [16].

Turning the above algorithm around, we may ask what are the consequences of the hypothesis that there is no polynomial time algorithm for refuting dense random 3CNF formulas. This would imply that one cannot approximate max-3SAT within a ratio better than $7/8$, which is a known (but very difficult) NP-hardness result. Many other hardness of approximation results would follow [9], some of which are currently not known to have NP-hardness analogues. The above hypothesis (regardless of its correctness) seems to be a good rule of thumb for conjecturing hardness of approximation results. Many of its predictions (with weaker constants) can be proved assuming that NP does not have sub-exponential algorithms [17].

So how does one refute random 3CNF formulas? When $m > n^2$ one can do the following. There are roughly $3n$ clauses containing the variable x_1. It suffices to refute the sub-formula f_1 containing these clauses. Substitute $x_1 = 0$ and simplify f_1 to a 2SAT formula. This is a random formula with roughly $3n/2$ clauses, and hence is unlikely to be satisfiable. 2SAT can be decided in polynomial time. Repeating the above argument with $x_1 = 1$ refutes f_1.

As m gets smaller, refutation gets harder. The best algorithms known for refuting random 3SAT [12] require $m > cn^{3/2}$ (where experimentation shows that one can take $c = 2.5$). These algorithms are based on pair-wise statistical irregularities and eigenvalue computations. This approach was

initiated in [15] for 4SAT. We sketch this approach.

Consider a random 4SAT formula f with $m \gg n^2$ clauses. In a satisfying assignment a, at least half the variables are negative. (A complimentary argument handles the case that at least half of the variables are positive in a.) Let S be the set of variables negative in a. observe that there cannot be a positive clause in f whose four variables are in S. Construct a graph G on $N = \binom{n}{2}$ vertices, in which every pair of variables is a vertex, and every positive clause $(x_i \vee x_j \vee x_k \vee x_l)$ contributes an edge $([x_i x_j], [x_k x_l])$. If f is satisfiable then S induces an independent set of size $N/4$. Hence to refute f it suffices to show that G has no independent set of size $N/4$. But when f is random, the graph G is random, and the condition $m \gg n^2$ implies that G has a large linear number of edges. Such graphs do not have large independent sets. Moreover, this can be certified efficiently by eigenvalue techniques (or by semidefinite programming, using the theta function of Lovasz).

In combination with some additional ideas, the above approach extends to refuting random 3SAT formulas with $m > cn^{3/2}$ clauses for large enough c [12]. It is not know how to refute random 3SAT formulas with less than $n^{3/2}$ clauses. In particular, it is known that when m is much smaller than $n^{3/2}$, resolution would take exponential time [4], and that certain semidefinite programming approaches (reducing 3SAT to independent set on a graph with $7m$ vertices, and computing the theta function of the resulting graph) would not work.

4 Summary

We presented some rigorous models in which one can study heuristics. We presented some algorithmic results in these models (the presentation was biased towards algorithms that the author is more familiar with). There are also hardness results for some of these models, showing that under certain settings of the parameters of the model, no heuristic will work. This is beyond the scope of this presentation, but see [10] for example.

Two points that we wish to make is that in principle, it is possible to study heuristics in a mathematically rigorous way, and that once this is done, the design of heuristics may require quite sophisticated algorithmic ideas and supporting mathematical analysis. But perhaps the main point is that the rigorous study of heuristics is still a young and wide open research area.

Acknowledgements

This research of the author is partly supported by a grant from the German Israeli Foundation (GIF). This manuscript was prepared when the author was visiting Microsoft Research at Redmond, Washington.

References

[1] Noga Alon, Nabil Kahale: A Spectral Technique for Coloring Random 3-Colorable Graphs. SIAM J. Comput. 26(6): 1733-1748 (1997).

[2] Noga Alon, Michael Krivelevich, Benny Sudakov: Finding a Large Hidden Clique in a Random Graph. SODA 1998: 594-598.

[3] Rene Beier, Berthold Voecking: Typical properties of winners and losers in discrete optimization. STOC 2004: 343-352.

[4] Eli Ben-Sasson, Avi Wigderson: Short proofs are narrow - resolution made simple. J. ACM 48(2): 149-169 (2001).

[5] Yonatan Bilu, Nati Linial: Are stable instances easy? Manuscript (2004).

[6] Avrim Blum, Joel Spencer: Coloring Random and Semi-Random k-Colorable Graphs. J. Algorithms 19(2): 204-234 (1995).

[7] Bela Bollobas, Trevor I. Fenner, Alan M. Frieze: An algorithm for finding Hamilton cycles in a random graph. Combinatorica 7(4): 327-341 (1987).

[8] A. Braunstein, M. Mezard, R. Zecchina: Survey propagation: an algorithm for satisfiability. Manuscript (2002).

[9] Uriel Feige: Relations between average case complexity and approximation complexity. STOC 2002: 534-543.

[10] Uriel Feige, Joe Kilian: Heuristics for Semirandom Graph Problems. J. Comput. Syst. Sci. 63(4): 639-671 (2001).

[11] Uriel Feige, Robert Krauthgamer: Finding and certifying a large hidden clique in a semirandom graph. Random Struct. Algorithms 16(2): 195-208 (2000).

[12] Uriel Feige, Eran Ofek: Easily Refutable Subformulas of Large Random 3CNF Formulas. ICALP 2004: 519-530.

[13] Uriel Feige, Dan Vilenchik: A Local Search Algorithm for 3SAT. Technical Report MCS 04-07, Computer Science and Applied Mathematics, The Weizmann Institute of Science (2004).

[14] Abraham Flaxman: A spectral technique for random satisfiable 3CNF formulas. SODA 2003: 357-363.

[15] Andreas Goerdt, Michael Krivelevich: Efficient Recognition of Random Unsatisfiable k-SAT Instances by Spectral Methods. STACS 2001: 294-304.

[16] Johan Hastad: Some optimal inapproximability results. J. ACM 48(4): 798-859 (2001).

[17] Subhash Khot: Ruling Out PTAS for Graph Min-Bisection, Densest Subgraph and Bipartite Clique. FOCS 2004: 136-145.

[18] Daniel A. Spielman, Shang-Hua Teng: Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. STOC 2001: 296-305.