1 Lecture 2, October 28, 1999

A very useful algorithmic technique is that of dynamic programming. We shall illustrate the use of dynamic programming on fixed parameter versions of NP-hard optimization problems on graphs. We start with a short review of the notion of NP-completeness.

1.1 NP-completeness

The class P is that of all problems solvable in polynomial time (or languages recognizable in polynomial time). The class NP (nondeterministic polynomial time) stands for those problems solvable in polynomial time by a nondeterministic Turing machine. The notion of nondeterminism is not required in the current course, and hence we shall use the more intuitive notion of NP as that of problems whose solutions it is easy to verify.

Let $R_L(.,.)$ be a two input relation with the following properties:

1. For some constant c, $R_L(x, w)$ is satisfied only if $|w| \leq |x|^c$, where $|.|$ denotes length (number of bits in binary string).
2. There is a polynomial time algorithm (in $|x|$) that tests for every x, w whether $R_L(x, w)$ holds.

Consider the language $L \subset \{0,1\}^*$ defined by $x \in L$ iff for some "witness" w, $R_L(x, w)$ holds. Then this is an NP language. Moreover, for every $L' \in NP$ one can define a relation $R_{L'}$ as above such that membership in L' is equivalent to having a witness for $R_{L'}$.

As a cannonocal example of an NP-language consider 3SAT. The input strings x are interpreted via some natural encoding system as representing a 3CNF (conjunctive normal form) formula, that is, a collection of clauses where each clause is the *logical or* of up to three literals, where a literal is either a variable or its negation. A 3CNF formula is *satisfiable* iff there exists an assignment to its variables that satisfies all its clauses simultaneously. 3SAT is the language of all satisfiable 3CNF formulas. 3SAT is in NP, because it is easy to check in polynomial time whether an assignment w to the variables satisfies the formula x.

We say that a language L is NP-hard if it is at least as hard as every NP-language in the following sense. For every NP-language L', there is some polynomial time computable function f such that for every input x, $f(x) \in L$ iff $x \in L'$. Hence if there is an efficient algorithm for determining membership in L, it can be used for efficiently determining membership in every NP language. Conversely, if some language in NP does not have polynoial time algorithms, than no NP-hard language is recognizable in polynomial time.

3SAT is known to be NP-hard, via a cannonical reduction from computations of nondeterministic Turing machines. Having one language L' that is NP-hard, proving a new language L NP-hard becomes conceptually simpler, as it suffices to show a reduction from L' to L. Because of transitivity, this then implies that all NP languages reduce to L.

As an example of a reduction, let us show that k-clique is NP-hard. The input for k-clique is a graph $G(V, E)$ and a parameter k. The instance belongs to the language if there is $S \subset V$ of cardinality $|S| = k$ such that the subgraph induced on S is a clique (for $u, v \in S$, $(u, v) \in E$).

We show that 3SAT can be reduced to k-clique in polynomial time. Let x be a 3CNF formula with variables $y_1, \ldots, y_n$ and clause $C_1, \ldots, C_m$. For simplicity we assume that each clause contains three distinct variables (it is easy to extend our reduction to handle smaller clauses). Based on x we create a graph G on $7m$ vertices and set $k = m$. Each clause C_i of x gives rise to a cluster C_i of seven vertices in G. These vertices are labeled by the seven different assignments to the literals of the clause C_i that satisfy this clause. Two vertices in G are connected by an edge if they are consistent, meaning that either they share no variable, or if they do share a variable, the assignment implied by the labels of the two vertices is consistent on the variables that they share (it does not happen that one label assigns 0 to a variable and the other label assigns 1 to the same variable). Observe that the clusters form independent sets (there are no edges within clusters).

If x is satisfiable, then G has a clique of size k. Consider an arbitrary satisfying assignment w for x. It satisfies every clause. In G, from each cluster, select the vertex labeled by an assignment that does not contradict w. These vertices form a clique of size $m = k$.

Likewise, if G has a clique of size k, then this clique must contain exactly one vertex from each cluster. This vertex is labeled by some assignment. All assignments are mutually consistent (by the edge relationship) and hence can be combined to one assignment that satisfies all clauses.

Observe that k-clique is not only NP-hard, but also belongs to NP. The languages that are both in NP and NP-hard are called NP-complete. All NP-complete languages are equivalent in the following sense: either all of them have polynomial time algorithms, or none of them do.

Our treatment of NP-completeness above was very sketchy, as this subject belongs to a computational complexity course rather than an algorithms course. In particular, there are other notions of reductions that can be used (such as Turing reductions, and even randomized reductions). For a more detailed treatment of NP-completeness see the book by Garey and Johnson.

1.2 Parameterized NP-complete optimization graph problems

Often NP-hard optimization problems have a parameter associated with them. As an example, k-clique has the parameter k – the size of the clique we are looking for. Often, when this parameter is small, the problem is no longer NP-hard. For k-clique, we can determine by exhaustive search whether a graph has a k-clique in time roughly $\binom{n}{k}\binom{k}{2}$, which is polynomial for every fixed k. (Later in the course we shall see a different algorithm for finding k-cliques which has a slightly better asymptotic running time.)

For some problems, a parameter of small value does not appear to make the problem easier. Consider for example **k-coloring**: can the input graph be legally colored with k-colors? This problem is NP-complete for every $k \geq 3$. (The problem is polynomially solvable for $k = 2$, as two colorability is just bipartiteness.)

We shall see examples of three parametrized NP-hard optimization graph problems where dynamic programming gives relatively efficient algorithms.

1.3 Bisection

Here we consider **k-bisection**: does the input graph have a bisection (cutting the graph into two parts of equal number of vertices) with only k edges joining the two parts?

As a special case, let us first consider 0-bisection: does the input graph have a bisection with no edges joining the two parts?

The following algorithm solves 0-bisection. First, decompose the graph into connected components. (This can be done easily in polynomial time using any graph traversal algorithm, such as breadth first search or depth first search.) Let $C_1, \ldots, C_l$ denote the connected components, and let $n_1, \ldots, n_l$ denote their respective sizes. If the graph has a 0-bisection, then every connected component must lie entirely on one side of the cut. Hence the problem reduces to a *subset sum* problem: find coefficients $a_i \in \{0,1\}$ such that $\sum_{i=1}^{l} a_i n_i = n/2$. In general, subset sum is NP-hard, when complexity is measured as a function of l (the number of items). However, if is solvable in polynomial time via dynamic programming if complexity is measured in terms of n (the desired sum) and items have positive integer sizes. Hence in our case, the problem is solvable in polynomial time (see below) and we can decide 0-bisection in polynomial time.

This algorithm extends to k bisection for every fixed k as follows. We show here how to find a bisection of at most k-edges, but the approach presented here can be extended also to find a bisection of exactly k edges. Guess the k edges of the bisection. (There are only $\binom{m}{k}$ possibilities to choose from, where m is the number of edges, and hence we can exhaustively try all options when k is constant.) Remove these edges. Solve 0-bisection of the remaining graph. Put back the k-edges to obtain a k-bisection.

Remark: The complexity of the algorithm above is proportional to $\binom{m}{k}$, which might be as bad as $\Omega(n^{2k})$. Later in the course we will see how to replace the dependence on m by a dependence on n, having running time proportional to $\binom{nk}{k}$, which is $O(n^k)$ for fixed k.

We now explain how dynamic programming solves the subset sum problem. Recall that we need to find coefficients $a_i \in \{0,1\}$ such that $\sum_{i=1}^{l} a_i n_i = n/2$. We make a table T with l columns numbered 1 to l and $n/2+1$ rows numbered 0 to $n/2$. Our algorithm will set $T_{c,r}$ to 1 if there are $\{0,1\}$ coefficients satisfying $\sum_{i=1}^{r} a_i n_i = c$, and to 0 otherwise. We are interested in $T_{n/2,l}$ which is the answer to the subset sum problem. We fill the table column by column. For the first column, $T_{0,1} = 1$, $T_{n_1,1} = 1$, and all the rest of the entries are 0. Column $i+1$ can easily be completed by considering column i. $T_{i+1,j} = 1$ iff either $T_{i,j} = 1$ or $T_{i,j-n_{i+1}} = 1$.

Finding that $T_{n/2,l} = 1$, an actual value for the a_i's satisfying $\sum_{i=1}^{l} a_i n_i = n/2$ can be found by traversing T column by column backwards from $T_{n/2,l}$, each time to a 1-entry in the previous column that implied a 1 in the current entry.

(To be continued in Lecture 3. Handout 2 is common to Lectures 2 and 3, and corrects error in Handout 1.)

References:

- Garey and Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.