CS124 Lecture 19

We have defined the class of **NP**-complete problems, which have the property that if there is a polynomial time algorithm for any one of these problems, there is a polynomial time algorithm for all of them. Unfortunately, nobody has found an algorithm for any **NP**-complete problem, and it is widely believed that it is impossible to do so.

This might seem like a big hint that we should just give up, and go back to solving problems where we can find a polynomial time solution. Unfortunately, **NP**-complete problems show up all the time in the real world, and people want solutions to these problems. What can we do?

NP-completeness results often arise because we want an exact answer. If we relax the problem so that we only have to return a good answer, then we might be able to develop a polynomial time algorithm. For example, we have seen that a greedy algorithm provides an approximate answer for the SET COVER problem—approximate in the sense that we can give some guarantee on how far its answer is from optimal.

Often when we talk about an approximation algorithm, we give an *approximation ratio*. The approximation ratio gives the ratio between our solution and the actual solution. The goal is to obtain an approximation ratio as close to 1 as possible. If the problem involves a minimization, the approximation ratio will be greater than 1; if it involves a maximization, the approximation ratio will be less than 1.

Vertex Cover Approximations

In the Vertex Cover problem, we wish to find a set of vertices of minimal size such that every edge is adjacent to some vertex in the cover. That is, given an undirected graph $G = (V, E)$, we wish to find $U \subseteq V$ such that every edge $e \in E$ has an endpoint in U. We have seen that Vertex Cover is **NP**-complete.

A natural greedy algorithm for Vertex Cover is to repeatedly choose a vertex with the highest degree, and put it into the cover. When we put the vertex in the cover, we remove the vertex and all its adjacent edges from the graph, and continue. Unfortunately, in this case the greedy algorithm gives us a rather poor approximation, as can be seen with the Figure 19.1.

In the example, all edges are connected to the base level; there are $m/2$ vertices at the next level, $m/3$ vertices at the next level, and so on. Each vertex at the base level is connected to one vertex at each other level, and the connections are spread as evenly as possible at each level. A greedy algorithm could always choose a topmost vertex, whereas the optimal cover consists of the bottom vertices. This example shows that, in general, the greedy

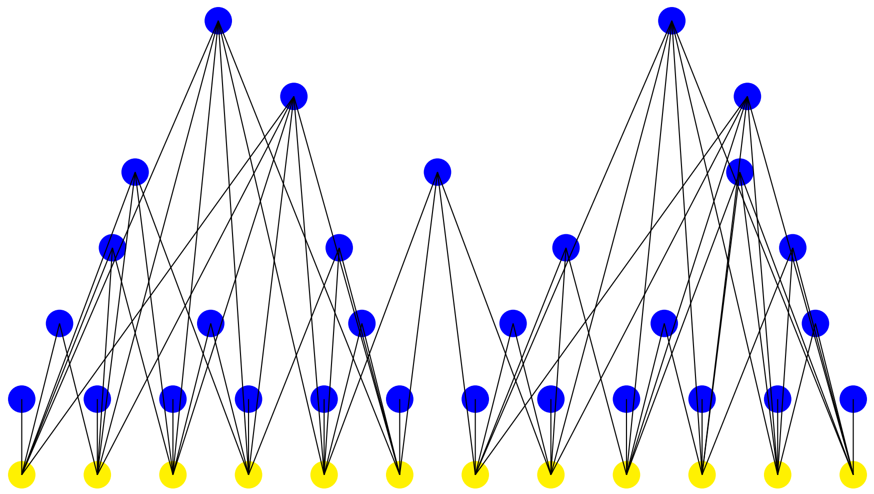

Figure 19.1: A bad greedy example. Yellow vertices are an optimal cover; blue ones are picked by the greedy algorithm.

approach could be off by a factor of $\Omega(\log n)$, where n is the number of vertices.

A better algorithm for vertex cover is the following: repeatedly choose an edge, and throw *both* of its endpoints into the cover. Throw the vertices and its adjacent edges out of the graph, and continue.

It is easy to show that this second algorithm uses at most twice as many vertices as the optimal vertex cover. This is because each edge that gets chosen during the course of the algorithm must have one of its endpoints in the cover; hence we have merely always thrown two vertices in where we might have gotten away with throwing in 1.

Somewhat surprisingly, this simple algorithm is still the best known approximation algorithm for the vertex cover problem. That is, no algorithm has been proven to do better than within a factor of 2.

Maximum Cut Approximation

We will provide both a randomized and a deterministic approximation algorithm for the MAX CUT problem. The MAX CUT problem is to divide the vertices in a graph into two disjoint sets so that the numbers of edges between vertices in different sets is maximized. This problem is **NP**-hard. (Note that the corresponding MIN CUT problem can be solved in polynomial time, which we'll see later in the course.)

The randomized version of the algorithm is as follows: we divide the vertices into two sets, HEADS and TAILS. We decide where each vertex goes by flipping a (fair) coin.

What is the probability an edge crosses between the sets of the cut? This will happen only if its two endpoints lie on different sides, which happens 1/2 of the time. (There are 4 possibilities for the two endpoints – HH,HT,TT,TH – and two of these put the vertices on different sides.) So, on average, we expect 1/2 the edges in the graph to cross the cut. Since the most we could have is for all the edges to cross the cut, this random assignment will, on average, be within a factor of 2 of optimal.

We now examine a deterministic algorithm with the same "approximation ratio". (In fact, the two algorithms are intrinsically related– but this is not so easy to see!) We will split the vertices into sets S_1 and S_2. Start with all vertices on one side of the cut. Now, if you can switch a vertex to a different side so that it increases the number of edges across the cut, do so. Repeat this action until the cut can no longer be improved by this simple switch.

We switch vertices at most $|E|$ times (since each time, the number of edges across the cut increases). Moreover, when the process finishes we are within a factor of 2 of the optimal, as we shall now show. In fact, when the process finishes, at least $|E|/2$ edges lie in the cut.

We can count the edges in the cut in the following way: consider any vertex $v \in S_1$. For every vertex w in S_2 that it is connected to by an edge, we add $1/2$ to a running sum. We do the same for each vertex in S_2. Note that each edge crossing the cut contributes 1 to the sum– 1/2 for each vertex of the edge.

Hence the cut C satisfies

$$C = \frac{1}{2}\left(\sum_{v \in S_1} |\{w : (v,w) \in E, w \in S_2\}| + \sum_{v \in S_2} |\{w : (v,w) \in E, w \in S_1\}|\right).$$

Since we are using the local search algorithm, at least half the edges from any vertex v must lie in the set opposite from v; otherwise, we could switch what side vertex v is on, and improve the cut! Hence, if vertex v has degree $\delta(v)$, then

$$\begin{aligned}
C &= \frac{1}{2}\left(\sum_{v \in S_1} |\{w : (v,w) \in E, w \in S_2\}| + \sum_{v \in S_2} |\{w : (v,w) \in E, w \in S_1\}|\right) \\
&\geq \frac{1}{2}\left(\sum_{v \in S_1} \frac{\delta(v)}{2} + \sum_{v \in S_2} \frac{\delta(v)}{2}\right) \\
&= \frac{1}{4}\sum_{v \in V} \delta(v) \\
&= \frac{1}{2}|E|,
\end{aligned}$$

where the last equality follows from the fact that if we sum the degree of all vertices, we obtain twice the number of edges, since we have counted each edge twice.

In practice, this algorithm often does better than just getting a cut within a factor of 2.

Euclidean Travelling Salesperson Problem

In the Euclidean Travelling Salesman Problem, we are given n points (cities) in the $x-y$ plane, and we seek the tour (cycle) of minimum length that travels through all the cities. This problem is NP-complete (showing this is somewhat difficult).

Our approximation algorithm involves the following steps:

1. Find a minimum spanning tree T for the points.
2. Create a *pseudo tour* by walking around the tree. The pseudo tour may visit some vertices twice.
3. Remove repeats from the tour by *short-cutting* through the repeated vertices. (See Figure 19.2.)

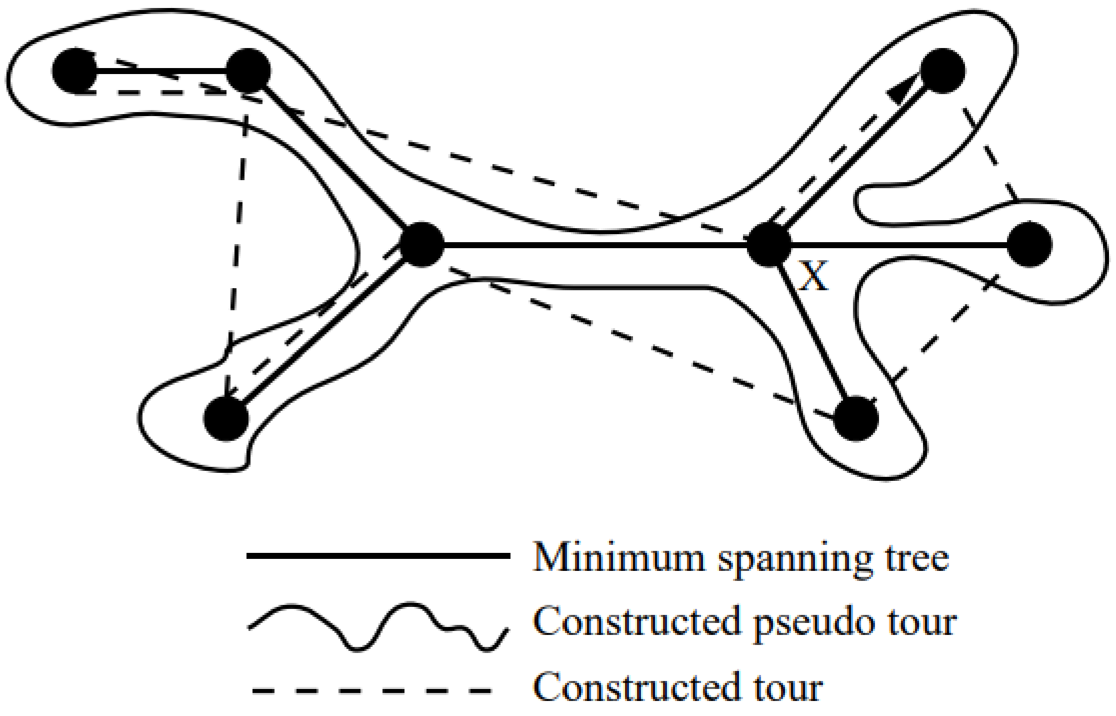

Figure 19.2: Building an approximate tour. Start at X, move in the direction shown, short-cutting repeated vertices.

We now show the following inequalities:

$$\begin{aligned} \text{length of tour} &\leq \text{length of pseudo tour} \\ &\leq 2(\text{size of T}) \\ &\leq 2(\text{length of optimal tour}) \end{aligned}$$

Short-cutting edges can only decrease the length of the tour, so the tour given by the algorithm is at most the length of the pseudo tour. The length of our pseudo tour is at most twice the size of the spanning tree, since this pseudo tour consists of walking through each edge of the tree at most twice. Finally, the length of the optimal tour is at least the size of the minimum spanning tree, since any tour contains a spanning tree (plus an edge!).

Using a similar idea, one can come up with an approximation algorithm that returns a tour that is within a factor of $3/2$ of the optimal. Also, note that this algorithm will work in any setting where short-cutting is effective. More specifically, it will work for any instance of the travelling salesperson problem that satisfies the *triangle inequality* for distances: that is, if $d(x,y)$ represents the distance between vertices x and y, and $d(x,z) \leq d(x,y) + d(y,z)$ for all x, y and z.

MAX-SAT: Applying Randomness

Consider the MAX-SAT problem: given a SAT formula in conjunctive normal form (that is, an OR of clauses, each of which is an AND of literals, each of which is a variable or its negation), what's the most clauses that can be satisfied? What happens if we do the simplest random thing we can think of– we decide whether each variable should be TRUE or FALSE by flipping a coin.

Theorem 19.1 *On average, at least half the clauses will be satisfied if we just flip a coin to decide the value of each variable. Moreover, if each clause has k literals, then on average* $1 - 2^{-k}$ *clauses will be satisfied.*

The proof is simple. Look at each clause. If it has k literals in it, then each literal could make the clause TRUE with probability 1/2. So the probability the clause is not satisfied is $1 - 2^{-k}$, where k is the number of literals in the clause.

Linear Programming Relaxation

The next approach we describe, linear programming relaxation, can often be used as a good heuristic, and in some cases it leads to approximation algorithms with provable guarantees. Again, we will use the MAX-SAT problem as an example of how to use this technique.

A *linear program* has the following form: there is an *objective function* that one seeks to optimize, along with *constraints* on the variables. The objective function and the constraints are all *linear* in the variables; that is, all equations have no powers of the variables, nor are the variables multiplied together. For instance, the following is a linear program:

$$\begin{array}{rcl} \multicolumn{3}{l}{\max 100x_1 + 600x_2 + 1400x_3} \\ x_1 & \leq & 200 \\ x_2 & \leq & 300 \\ x_1 + x_2 + x_3 & \leq & 400 \\ x_2 + 3x_3 & \leq & 600 \\ x_1, x_2, x_3 & \geq & 0 \end{array}$$

If the variable values are required to be integers, it's called an *integer linear program* or more commonly just *linear program*. There are polynomial time algorithms for solving linear programs, but integer linear programs are NP-complete.

Many **NP**-complete problems can be easily described by a natural Integer Programming problem. (Of course, all **NP**-complete problems can be transformed into some Integer Programming problem, since Integer Programming is **NP**-complete; but what we mean here is in many cases the transformation is quite natural.) Even though we cannot solve the related Integer Program, *if we pretend it is a linear program*, then we can solve it. This idea is known as *relaxation*, since we are relaxing the constraints on the solution; we are no longer requiring that we get a solution where the variables take on integer values.

If we are extremely lucky, we might find a solution of the linear program where all the variables are integers, in which case we will have solved our original problem. Usually, we will not. In this case we will have to try to somehow take the linear programming solution, and modify it into a solution where all the variables take on integer values. *Randomized Rounding* is one technique for doing this.

MAX-SAT

We may formulate MAX-SAT as an integer programming problem in a straightforward way. Suppose the formula contains variables $x_1, x_2, \ldots, x_n$ which must be set to TRUE or FALSE, and clauses $C_1, C_2, \ldots, C_m$. For each variable x_i we associate a variable y_i which should be 1 if the variable is TRUE, and 0 if it is FALSE. For each clause C_j we have a variable z_j which should be 1 if the clause is satisfied and 0 otherwise.

We wish to maximize the number of satisfied clauses s, or

$$\sum_{j=1}^{m} z_j.$$

The constraints include that that $0 \leq y_i, z_j \leq 1$; since this is an integer program, this forces all these variables to be either 0 or 1. Finally, we need a constraint for each clause saying that its associated variable z_j can be 1 if and only if the clause is actually satisfied. If the clause C_j is $(x_2 \vee \overline{x_4} \vee x_6 \vee \overline{x_8})$, for example, then we need the restriction:

$$y_2 + y_6 + (1 - y_4) + (1 - y_8) \geq z_j.$$

This forces z_j to be 0 unless the clause can be satisfied. In general, we replace x_i by y_i, $\overline{x_i}$ by $1 - y_i$, $\vee$ by $+$, and set the whole thing $\geq z_j$ to get the appropriate constraint.

When we solve the linear program, we will get a solution that might have $y_1 = 0.7$ and $z_1 = 0.6$, for instance. This initially appears to make no sense, since a variable cannot be 0.7 TRUE. But we can still use these values in a reasonable way. If $y_1 = 0.7$, it suggests that we would prefer to set the variable x_1 to TRUE (1). In fact, we could try just rounding each variable up or down to 0 or 1, and use that as a solution! This would be one way to turn the non-integer solution into an integer solution. Unfortunately, there are problems with this method. For example, suppose we have the clause $C1 = (x_1 \vee x_2 \vee x_3)$, and $y_1 = y_2 = y_3 = 0.4$. Then by simple rounding, this clause will not be TRUE, even though it "seems satisfied" to our linear program (that is, $z_1 = 1$). If we have a lot of these clauses, regular rounding might perform very poorly.

It turns out that there an interpretation for 0.7 that suggests a better way than simple rounding. We think of the 0.7 as *a probability*. That is, we interpret $y_1 = 0.7$ as meaning that x_1 would like to be true with probability 0.7. So we take each variable x_i, and independently we set it to 1 with the probability given by y_i (and with probability $1 - y_i$ we set x_i to 0). This process is known as *randomized rounding*. One reason randomized rounding is useful is it allows us *to prove* that the expected number of clauses we satisfy using this rounding is a within a constant factor of the true optimum.

First, note that whatever the maximum number of clauses s we can satisfy is, the value found by the linear program, or $\sum_{j=1}^{m} z_j$, is at least as big as s. This is because the linear program could achieve a value of at least s simply by using as the values for y_i the truth assignment that make satisfying s clauses possible.

Now consider a clause with k variables; for convenience, suppose the clause is just $C_1 = (x_1 \vee x_2 \ldots \vee x_k)$. Suppose that when we solve the linear program, we find $z_1 = \beta$. Then we claim that the probability that this clause is satisfied after the rounding is at least $(1 - 1/e)\beta$. This can be checked (using a bit of sophisticated math), but it follows by noting (with experiments) that the worst possibility is that $y_1 = y_2 \ldots = y_k = \beta/k$. In this case, each x_1 is FALSE with probability $(1 - \beta/k)$, and so C_1 ends up being unsatisfied with probability $(1 - \beta/k)^k$. Hence the probability it is satisfied is at least (again using some math) $1 - (1 - \beta/k)^k \geq (1 - 1/e)\beta$.

Hence the ith clause is satisfied with probability at least $(1 - 1/e)z_i$, so the expected number of satisfied clauses after randomized rounding is at least $(1-1/\mathrm{e})\sum_{j=1}^{m} z_j$. This is within a factor of $(1-1/\mathrm{e})$ of our upper bound on the maximum number of satisfiable clauses, $\sum_{j=1}^{m} z_j$. Hence we expected to get within a constant factor of the maximum.

Combining the Two

Surprisingly, by combining the simple coin flipping algorithm with the randomized rounding algorithm, we can get an even better algorithm. The idea is that the coin flipping algorithm does best on long clauses, since each literal in the clause makes it more likely the clause gets set to TRUE. On the other hand, randomized rounding does best on short clauses; the probability the clause is satisfied $(1-(1-\beta/k)^k)$ decreases with k. It turns out that if we try both algorithms, and take the better result, on average we will satisfy $3/4$ of the clauses.

We also point out that there are even more sophisticated approximation algorithms for MAX-SAT, with better approximation ratios. However, these algorithms point out some very interesting and useful general techniques.