U.C. Berkeley — CS170: Intro to CS Theory Handout N24
Professor Luca Trevisan December 4, 2001

Notes for Lecture 24

1 Some NP-complete Numerical Problems

1.1 Subset Sum

The **Subset Sum** problem is defined as follows:

- Given a sequence of integers $a_1, \ldots, a_n$ and a parameter k,
- Decide whether there is a subset of the integers whose sum is exactly k. Formally, decide whether there is a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = k$.

Subset Sum is a true *decision problem*, not an optimization problem forced to become a decision problem. It is easy to see that Subset Sum is in NP.

We prove that Subset Sum is NP-complete by reduction from Vertex Cover. We have to proceed as follows:

- Start from a graph G and a parameter k.
- Create a sequence of integers and a parameter k'.
- Prove that the graph has vertex cover with k vertices iff there is a subset of the integers that sum to k'.

Let then $G = (V, E)$ be our input graph with n vertices, and let us assume for simplicity that $V = \{1, \ldots, n\}$, and let k be the parameter of the vertex cover problem.

We define integers $a_1, \ldots, a_n$, one for every vertex; and also integers $b_{(i,j)}$, one for every edge $(i, j) \in E$; and finally a parameter k'. We will define the integers a_i and $b_{(i,j)}$ so that if we have a subset of the a_i and the $b_{(i,j)}$ that sums to k', then: the subset of the a_i corresponds to a vertex cover C in the graph; and the subset of the $b_{(i,j)}$ corresponds to the edges in the graph such that exactly one of their endpoints is in C. Furthermore the construction will force C to be of size k.

How do we define the integers in the subset sum instance so that the above properties hold? We represent the integers in a matrix. Each integer is a row, and the row should be seen as the base-4 representation of the integer, with $|E| + 1$ digits.

The first column of the matrix (the "most significant digit" of each integer) is a special one. It contains 1 for the a_is and 0 for the $b_{(i,j)}$s.

Then there is a column (or digit) for every edge. The column (i, j) has a 1 in a_i, a_j and $b_{(i,j)}$, and all 0s elsewhere.

The parameter k' is defined as

$$k' := k \cdot 4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^i$$

This completes the description of the reduction. Let us now proceed to analyze it.

From Covers to Subsets Suppose there is a vertex cover C of size k in G. Then we choose all the integers a_i such that $i \in C$ and all the integers $b_{(i,j)}$ such that exactly one of i and j is in C. Then, when we sum these integers, doing the operation in base 4, we have a 2 in all digits except for the most significant one. In the most significant digit, we are summing one $|C| = k$ times. The sum of the integers is thus k'.

From Subsets to Covers Suppose we find a subset $C \subseteq V$ and $E' \subseteq E$ such that

$$\sum_{i \in C} a_i + \sum_{(i,j) \in E'} b_{(i,j)} = k'$$

First note that we never have a carry in the $|E|$ less significant digits: operations are in base 4 and there are at most 3 ones in every column. Since the $b_{(i,j)}$ can contribute at most one 1 in every column, and k' has a 2 in all the $|E|$ less significant digits, it means that for every edge (i,j) C must contain either i or j. So C is a cover. Every a_i is at least $4^{|E|}$, and k' gives a quotient of k when divided by $4^{|E|}$. So C cannot contain more than k elements.

1.2 Partition

The **Partition** problem is defined as follows:

- Given a sequence of integers $a_1, \ldots, a_n$.
- Determine whether there is a partition of the integers into two subsets such the sum of the elements in one subset is equal to the sum of the elements in the other.

  Formally, determine whether there exists $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = (\sum_{i=1}^n a_i)/2$.

Clearly, Partition is a special case of Subset Sum. We will prove that Partition is is NP-hard by reduction from Subset Sum.[1]

Given an instance of Subset Sum we have to construct an instance of Partition. Let the instance of Subset Sum have items of size $a_1, \ldots, a_n$ and a parameter k, and let $A = \sum_{i=1}^n a_i$.

Consider the instance of Partition $a_1, \ldots, a_n, b, c$ where $b = 2A - k$ and $c = A + k$.

Then the total size of the items of the Partition instance is $4A$ and we are looking for the existence of a subset of $a_1, \ldots, a_n, b, c$ that sums to $2A$.

It is easy to prove that the partition exists if and only if there exists $I \subseteq \{1, \ldots, n\}$ such that $\sum_i a_i = k$.

1.3 Bin Packing

The **Bin Packing** problem is one of the most studied optimization problems in Computer Science and Operation Research, possibly the second most studied after TSP. It is defined as follows:

- Given items of size $a_1, \ldots, a_n$, and given unlimited supply of bins of size B, we want to pack the items into the bins so as to use the minimum possible number of bins.

[1] The reduction goes in the non-trivial direction!

You can think of bins/items as being CDs and MP3 files; breaks and commercials; bandwidth and packets, and so on.

The decision version of the problem is:

- Given items of size $a_1, \ldots, a_n$, given bin size B, and parameter k,
- Determine whether it is possible to pack all the items in k bins of size B.

Clearly the problem is in NP. We prove that it is NP-hard by reduction from Partition.

Given items of size $a_1, \ldots, a_n$, make an instance of Bin Packing with items of the same size and bins of size $(\sum_i a_i)/2$. Let $k = 2$.

There is a solution for Bin Packing that uses 2 bins if and only if there is a solution for the Partition problem.

2 Approximating NP-Complete Problems

What does it mean to approximate the solution of an NP-complete problem, when we have so far been considering only questions with yes/no answers? Actually, many of the problems we have looked at are most naturally stated as *optimization problems*, which we can approximate. For example, in **TSP** the yes/no question is whether there is a cycle visiting each node once of length at most K; the optimization problem is to find the *shortest* cycle; and the approximation problem is to find a *short* cycle, i.e. one whose length is as short as one can afford to compute. Similarly, for **Vertex Cover** the yes/no question is whether there is subset of vertices touching each edge, of cardinality at most k; the optimization problem is to find the vertex cover with the *fewest* vertices; and the approximation problem is to find a vertex cover with *few* vertices, i.e. one whose cardinality is as small as one can afford to compute.

To measure how good an approximation we compute, we need a measure of error. Suppose $c > 00$ is an approximation to the exact value $c^* > 0$. (For Vertex Cover, c is the size of a vertex cover found by an approximate algorithm, and c^* is the true minimum vertex cover size.) Then the *ratio bound* (or *performance ratio* or *approximation ratio*) of c is ρ where

$$\max\left\{\frac{c}{c^*}, \frac{c^*}{c}\right\} \le \rho \ .$$

Note that $\rho \ge 1$, and $\rho = 1$ exactly when $c = c^*$. If the size of the problem whose answer is c^* is n, we may also write $\rho(n)$ to indicate that the error may depend on n.

For example, $\rho(n) = 2$, means that the computed result c never differs from c^* by more than a factor of 2.

We will characterize our approximation algorithms by the value of ρ that they guarantee. It turns out that some NP-complete problems let us compute approximations for small ρ in polynomial time, but for other NP-complete problems computing any approximation with bounded ρ is still NP-complete.

2.1 Approximating Vertex Cover

Recall that a *vertex cover* C is a subset $C \subset V$ of the set of vertices of a graph $G = (V, E)$ with the property that every edge in E has a vertex in C as an end point. The vertex cover optimization problem is to find the vertex cover of smallest cardinality c^*. It turns out that the following simple greedy algorithm never finds a cover more than twice too large, i.e. $c \leq 2c^*$, or $\rho = 2$.

```
$C = \emptyset$ // initialize an empty vertex cover
for all $(u, v) \in E$
    if $u \notin C$ and $v \notin C$ then $C := C \cup \{u, v\}$
return $C$
```

The set C can be represented using a Boolean vector of size $|V|$, so that checking membership in C can be done in constant time. The running time is obviously $O(|E| + |V|)$, since we use $O(|V|)$ time to initialize the representation of C and then we do a constant amount of work for every edge.

Theorem 1
The above algorithm satisfies $\rho \leq 2$, i.e. if C is the cover computed by the algorithm and C^* is an optimal cover, then $|C| \leq 2|C^*|$.

Proof: First of all, it should be clear that the algorithm returns a vertex cover. Every time an edge is found that is not covered by the current set C, then both endpoints are added to C, thus guaranteeing that the edge is covered.

For the approximation, let M be the set of edges (u, v) such that when (u, v) is considered in the `for` loop, the vertices u, v are added to C. By construction, $|C| = 2|M|$. Furthermore, M is a matching, and so each edge of M must be covered by a distinct vertex; even if C^* is an optimum cover, we must have $|C^*| \geq |M|$, and so $|C| \leq 2|C^*|$. □

Let us now consider another way of achieving an approximation within a factor of 2. Given a graph $G = (V, E)$, write a linear program having a variable x_v for each vertex v, and structured as follows:

$$\begin{array}{lll}
\min & \sum_v x_v & \\
\text{s.t.} & & \\
& x_u + x_v \geq 1 & \text{(for each } (u, v) \in E) \\
& x_v \geq 0 & \text{(for each } v \in V)
\end{array}$$

Now, let C be a vertex cover, and consider the solution defined by setting $x_v = 0$ if $v \notin C$, and $x_v = 1$ if $v \in C$. Then the solution is feasible, because all values are ≥ 0, and each (u, v) is covered by C, so that $x_u + x_v$ is equal to either one or two (and so is at least one). Furthermore, the cost of the solution is equal to the cardinality of C. This implies that the cost of the optimum of the linear program is at most the cost of an optimum solution for the vertex cover problem.

Let's now solve optimally the linear program, and get the optimal values x_v^*. By the above reasoning, $\sum_x x_v^* \leq c^*$. Let us then define a vertex cover C, as follows: a vertex v belongs to C if and only if $x_v^* \geq 1/2$. We observe that:

- C is a valid vertex cover, because for each edge (u, v) we have $x^*_u + x^*_v \geq 1$, so at least one of x^*_u or x^*_v must be at least $1/2$, and so at least one of u or v belongs to C.
- $\sum_v x^*_v \geq |C|/2$, because every vertex in C contributes at least $1/2$ to the sum, and the vertices not in C contribute at least 0. Equivalently, $|C| \leq \sum_v x^*_v \leq 2c^*$ where c^* is the cardinality of an optimum vertex cover.

So, again, we have a 2-approximate algorithm.

No algorithm achieving an approximation better than 2 is known. If $\mathbf{P}\neq\mathbf{NP}$, then there is no polynomial time algorithm that achieves an approximation better than $7/6$ (proved by Håstad in 1997). A few months ago, a proof has been announced that, if $\mathbf{P}\neq\mathbf{NP}$, then there is no polynomial time algorithm that achieves an approximation better than $4/3$ (the result is still unpublished).

The greedy algorithm might have been independently discovered by several people and was never published. It appears to have been first mentioned in a 1974 manuscript. The linear programming method is from the late 1970s and it also applies to the *weighted* version of the problem, where every vertex has a positive weight and we want to find the vertex cover of minimum total weight.

The LP-based algorithm was one of the first applications of the following general methodology: formulate a linear program that "includes" all solutions to an NP-complete problem, plus fractional solutions; solve the linear program optimally; "round" the, possibly fractional, optimum LP solution into a solution for the NP-complete problem; argue that the rounding process creates a solution whose cost is not too far away from the cost of the LP optimum, and thus not too far away from the cost of the optimum solution of the NP-complete problem. The same methodology has been applied to several other problems, with considerable success.

2.2 Approximating TSP

In the Travelling Salesman Problem (TSP), the input is an undirected graph $G = (V, E)$ and weights, or distances, $d(u, v)$ for each edge (u, v). (Often, the definition of the problem requires the graph to be complete, i.e., E to contain all possible edges.) We want to find an order in which to visit all vertices exactly once, so that the total distance that we have to travel is as small as possible. The vertices of the graph are also called "cities." The idea is that there is a salesman that has to go through a set of cities, he knows the distances between cities, and he wants to visit every city and travel the smallest total distance. This problem turns out to be extremely hard to approximate in this general formulation, mostly because the formulation is too general.

A more interesting formulation requires the graph to be complete and the distances to satisfy the "triangle inequality" that says that for every three cities u, v, w we have $d(u, w) \leq d(u, v) + d(v, w)$. This version of the problem is called "metric" TSP, or ΔTSP, where the letter Δ represents the "triangle" inequality.[2]

[2]The metric TSP is equivalent to the problem where we are given a graph $G = (V, E)$, not necessarily complete, and weights, or distances, $d(u, v)$ for each edge (u, v), and we want to find an order in which to visit each vertex *at least* once, so that the total distance that we have to travel is as small as possible. (Notice that we may come back to the same vertex more than once.) You could try to prove the equivalence as an exercise.

Consider the following algorithm for metric TSP: find a minimum spanning tree T in G, then define a cycle that starts at the root of the tree, and then moves along the tree in the order in which a DFS would visit the tree. In this way, we start and end at the root, we visit each vertex at least once (possibly, several times) and we pass through each edge of the minimum spanning tree exactly twice, so that the total length of our (non-simple) cycle is $2 \cdot cost(T)$.

What about the fact that we are passing through some vertices multiple times? We can straighten our tour in the following way: suppose a section of the tour is $u \to v \to w$, and v is a vertex visited by the tour some other time. Then we can change that section to $u \to w$, and by the triangle inequality we have only improved the tour. This action reduced the number of repetitions of vertices, so eventually we get a tour with no repetitions, and whose cost is only smaller than $2 \cdot cost(T)$.

Finally, take an optimal tour C, and remove an edge from it. Then what we get is a spanning tree, whose cost must be at least $cost(T)$. Therefore, the cost the optimal tour is at least the cost of the minimum spanning tree; our algorithm finds a tour of cost at most twice the cost of the minimum spanning tree and so the solution is 2-approximate.

This algorithm has never been published and is of uncertain attribution. An algorithm achieving a 3/2-approximation was published in 1976, based on similar principles. There has been no improvement since then, but there are reasons to believe that a 4/3-approximation is possible.

In 2000 it has been shown that, if **P**$\neq$**NP**, then there is no polynomial time algorithm achieving an approximation of about 1.01 for metric TSP.

If points are in the plane, and the distance is the standard geometric distance, then any approximation ratio bigger than one can be achieved in polynomial time (the algorithm is by Arora, 1996). This remains true even in geometries with more than two dimensions, but not if the number of dimensions grows at least logarithmically with the number of cities.

2.3 Approximating Max Clique

Håstad proved in 1996 that there can be no algorithm for Max Clique achieving an approximation ratio $\rho(n) = n^{.99}$, unless **NP** has polynomial time probabilistic algorithms (which is almost but not quite the same as **P**=**NP**). In fact, any constant smaller than one can be replaced in the exponent, and the result is still true.

Notice that the algorithm that returns a single vertex is an n-approximation: a vertex is certainly a clique of size 1, and no clique can be bigger than n.

We will see a $(n/\log n)$-approximation. In light of Håstad's result we cannot hope for much more.[3]

Given a graph $G = (V, E)$, the algorithm divides the set of vertices in $k = n/logn$ blocks $B_1, \ldots, B_k$, each block containing $\log n$ vertices. (It's not important how the blocks are chosen, as long as they all have size $\log n$.) Then, for each block B_i, the algorithm finds the largest subset $K_i \subseteq B_i$ that is a clique in G. This can be done in time $O(n(\log n)^2)$

[3]There is reason to believe that the right result is that a $n/2^{\sqrt{\log n}}$-approximation is possible in polynomial time, but not better. So far, the best known approximation algorithm has roughly $\rho(n) = n/(\log n)^2$, while a result by Khot shows that, under strong assumption, $\rho(n) = n/2^{(\log n)^{1-o(1)}}$ is not achievable in polynomial time.

time, because there are n possible subset, and checking that a set is a clique can be done in quadratic time (in the size of the set). Finally, the algorithm returns the largest of the cliques found in this way.

Let K^* be the largest clique in the graph. Clearly, there must be one block B_i that contains at least $|K^*|/k$ vertices of K^*, and when the algorithm considers B_i it will find a clique with at least $|K^*|/k$ vertices. Possibly, the algorithm might find a bigger clique in other blocks. In any case, the size of the clique given in output by the algorithm is at least $1/k = \log n/n$ times the size of K^*.