Lecture 25: NP completeness

Our second example is VERTEX-COVER (VC). A vertex cover is a subset of nodes of a graph G such that all edges in G have at least one node in the subset. The VC problem is: does G have a VC of size k or fewer? It is again easy to see VC is in NP. We can directly reduce CLIQUE to VC, as done in the textbook. Since we have seen before VC and IS can be reduced to each other, we will then know VC is also NP complete.

Now HAM-CYCLE. This reduction is more complex. In class, I briefly explained the most important part of the reduction. Read the textbook for more details.

We now move to numerical problems (recall Knapsack problem). We start with the SUBSET-SUM problem. Given finite set S of positive integers, and an integer t, we ask whether there exists a subset of integers of S that sum to t. Clearly, the SUBSET-SUM is in NP. We now show VERTEX-COVER is polynomial-time reducible to SUBSET-SUM.

In VC, we have a graph G and an integer k, and the question is: does G have a vertex cover of size k or smaller? To use a black box of SUBSET-SUM, we will construct integers S, and t from G s.t. G has a VC of size k or smaller i f subset of S add up to t. Let nodes in graph labeled by 1, 2, 3, . . . , n. We will create one integer a_i per node v_i and one integer b_{i,j} for edge (v_i, v_j). Then selected a_i correspond to a vertex cover (denoted C), and selected b_{i,j} are those with precisely one node in the vertex cover. We define integers in a matrix, where a row corresponds to an integer. Integers are treated as base-4. We add a special column as the first column (the most significant bit), which is 1 if the row is for a_i, and 0 for b_{i,j}. Then we have one column for each edge (u, v). For integer a_i, it has value 1 if node v_i is one of the two nodes for the edge corresponding to the column. b_{i,j} has precisely one 1 at the column corresponding to the edge (v_i, v_j). Finally, t = k * 4^|E| + 2 \sum_{i=0}^{|E|-1} 4^i.

To see why it works, we show two things. First, given a vertex cover C, we select the following integers. We pick all corresponding a_i for v_i \in C, and b_{i,j} if exactly one of v_i, v_j is in C. The selected k integers (corresponding to a_i) have MSB of 1, which add to k*4^|E|. For the rest of edges, we have exactly two 1 selected for that column. Check this out if you do not understand it yet. On the other hand, suppose we have a subset of integers summing to t. We note that there is carry in all columns (except the MSB). It follows easily that the corresponding rows (for the selected a_i) must form a vertex cover. Check it if you do not yet understand.

I have also shown that a formulation of the Minesweeper is NP complete. Please refer to the posted slides for more information.

Lecture 26: Algorithms for NP completeness problems

Often we must work on NP-complete problems, even when polynomial-time algorithms are unlikely obtainable. We can take several approaches.

- Efficient algorithm for special case of the problem.
- Fast algorithm when the parameter of the problem is favorable.
- Faster (although not polynomial time) algorithm than brute force algorithm.

Now, some NP complete problems become easier when we restrict the problem instances on special cases. As an example, we consider the problem of finding independent set on a tree. We first note that if there is an isolated node u (i.e. has no edge incident on u), we can add u to the independent set. A key property of a tree is that there exists a leaf u (only connects to a node v). For INDEPENDENT-SET problem, we prefer u. To see this, we consider an independent set C. if neither u nor v is in C, we can simply add u and this will lead to a (larger) independent set (why the enlarged set remains an independent set?). On the other hand, if v \in C, we can have another independent set C’ = (C - {v}) \cup \{u\} which remains an independent set (why?). Thus, we know for a given tree T(V, E) and u is a leaf, then there exists an IS includes u. This gives a greedy algorithm: each time, we look for a leaf u (which must exist when there is no cycle), include u in the independent set, and remove u and v (where there is an edge (u, v)). We repeat until the graph is empty. Note: what we got becomes
a forest (not a single tree but several disjoint trees). But what we proved still work: there still exists a leaf in a set of trees.

Sometimes, certain instances (with some range of parameters) of hard problems are easy to solve. Here, what parameters you can use depends on the problem. Here is one example: vertex cover. Recall that we want to find a subset of \( k \) nodes (fewer) covering all edges. The brute-force is enumerating all subsets of \( k \) nodes, which lead to \( O(n^k) \) algorithm. Here, \( n \) is the number of vertices in the graph. We want to have a faster algorithm. A simple observation is that each edge \((u,v)\) needs to covered and thus either \( u \) or \( v \) needs to be selected; since we are not sure whether to pick \( u \) or \( v \), we just try these two choices and continue; the algorithm is efficient if \( k \) is small since there are only \( 2^k \) choices we need to try. A recursive algorithm VERTEX-COVER(G,C) is like this. G: graph, C: the set of picked vertex cover. If the size of C is k, and G is not empty, then return empty set (which means we fail to find a vertex cover of \( k \) nodes or smaller). If G is empty, return C. Otherwise, G is not empty (which means there are edges). Pick any edge \((u,v)\) of G and try two options: first select \( u \) (then remove all edges incident to \( u \)) to form \( G' \). Recursively call VERTEX-COVER(G',C+\{u\}) and if it succeeds and returns \( C' \), return \( C' \). Similarly, if VERTEX-COVER(G',C+\{v\}) succeeds and returns \( C' \), return \( C' \). Here, \( G' \) refers to the graph where edges incident to the selected node are removed. Time analysis: note that the height of tree is at most \( k \), and so the size of tree is \( O(2^k) \). Updating graph takes \( O(n) \) time each step. So total run time \( O(n2^k) \). When \( k \) is small, this algorithm is much better than the brute-force algorithm.

We now revisit the traveling salesman problem, where we will give an exponential time algorithm. Recall the problem is given a complete weighted graph and needs a tour of all nodes exactly once with min-cost. The naive algorithm would take \( O(n!) \) time. We are going to use dynamic programming on this problem. First let us start tour at node 1 (and will return to node 1). Here is the DP subproblem: \( OPT[S, i] \) for all subset \( S \) of 2,3,...,n and some \( i \in S \), which is equal to the minimum total cost of the part of simple path from city 1 and visit nodes in \( S \) in some order and end at city \( i \). Here, \( S \) is non-empty. First, \( OPT[\{i\}, i] = dist(1, i) \) (where \( dist(i, j) \) is the distance between node i and node j). Then, \( OPT[S, i] = min_{j \in S-\{i\}} OPT[S-\{i\}, j] + dist(i, j) \). Suppose all \( OPT[S, i] \) is computed, the final solution is: \( min_i OPT[\{2, 3, \ldots, n\}, i] + dist(1, i) \). How to compute \( OPT[S, i] \)? In the increasing size of \( S \): start with \( S \) of size 1, then size 2, and so on. This works since in the recurrence \( OPT[S, i] \) only depends on subsets of smaller cardinality. Time: \( O(n \cdot 2^n) \) is the DP table size, and each table cell takes \( O(n) \) time. So total time is \( O(n^2 2^n) \). Although this is still an exponential time algorithm, it is much faster than the naive algorithm.