Prof. Athina Markopoulou
Winter 2018

Design and Analysis of Algorithms
EECS215

Homework 5 - Solution Sketch

Topic: NP and NP-Completeness

Posted on Fri 03/16/2018.

1. *(30 Points)* **The Path Selection Problem. (Ch.8, Pr.9)**

   Consider the path selection problem stated below:

   > Given a directed graph $G = (V, E)$, a set of paths $P_1, P_2, ...P_c$, and an integer $k > 0$, is it possible to select at least k out of the c paths so that no two of the selected paths share any nodes?

   (a) *(20 points)* Prove that the path selection problem is NP-complete.

   (b) *(10 points)* Is this a decision or an optimization problem? In which of the six broad categories of NP-complete problems does this problem belong?

   Answer:

   (a) The *path selection* problem is in NP because given a set of k out of c paths, we can check in polynomial time that no two of them have any nodes in common (considering set differences between every pairs of paths).

   The *path selection* problem is NP-complete because other NP-complete problems reduce to it in polynomial time. A pair of paths has a "conflict" if the two paths have a node in common. This intuition indicates that we should look for an NP-complete problem within the family of "packing problems", *e.g.,* the independent set problem or the set packing problem.

   The easiest reduction is probably from the set packing problem (see your book p.458 for the definition of the set packing problem). Let us consider an arbitrary instance of the set packing problem (*i.e.,* a set of elements U, a set of subsets of U $S_1, S_2, ...S_c$ and a number k) and ask the question:

   > `P1:` Does there exist a collection of at least k subsets so that no two of them intersect?

   We now need to construct a specific instance of the *path selection problem* (`P2`) so that the answer to `P1` is YES if and only if the answer to `P2` is yes. Construct a graph with nodes corresponding to the elements in U. Then take every subset $S_i, i = 1, ...c$ and draw a path on G connecting the nodes corresponding to the elements. (Question: does the order in which we consider the elements in S_m matter?). Then argue that the answer to `P1` is YES if and only if the answer to `P2` on the constructed graph is YES.

   (b) It is a decision problem and belongs to the packing problems, as explained above.

2. *(10 Points)* **Interval Scheduling.** Consider the decision version of the problem:

   Given a collection of intervals in the timeline, and a bound k, does the collection contain a subset of non-overlapping intervals of size at least k?

   For each statement below, state whether it is "True", "False", or "Unknown", and briefly justify your answer.

   (a) *(2 Points)* Interval Scheduling is in P.

   (b) *(2 Points)* Interval Scheduling is in NP.

   (c) *(2 Points)* Interval Scheduling is NP-complete.

   (d) *(2 Points)* Interval Scheduling $\leq_p$ Vertex Cover.

   (e) *(2 Points)* Independent Set $\leq_p$ Interval Scheduling.

   Answer:

   (a) **True**, because the optimization version of Interval Scheduling can be solved in polynomial time by a Greedy algorithm (Chapter 4.1), therefore the decision version is also in P.

   (b) **True**, because every problem in P are also in NP. Alternatively define certificate (set of k intervals) and polynomial time certifier (pairwise check for conflict in $O(k^2)$ time).

   (c) **Unknown**, depends on whether $P = NP$, which is unknown. This would mean there exist an NP-complete problem that reduces to Interval Scheduling, which is unknown.

   (d) **True**, Vertex cover is NP-complete and by definition every problem in NP can be reduced to it.

   (e) **Unknown**, depends on whether $P = NP$, which is unknown. Independent Set is NP-complete; this reduction would mean that $P = NP$.

3. *(35 Points)* **Vertex Cover on Trees.**

   For a graph $G = (V, E)$ with $|V| = n$ nodes and $|E| = m$ edges, a *vertex cover* is defined as a subset of nodes $S \subset V$ s.t. that all edges e are covered (*i.e.,* each edge is touched by at least one node in S). Let us restrict our attention specifically to graphs that are trees.

   (a) *(15 Points)* Design a greedy algorithm that finds the minimum size vertex cover on a tree; or argue that such an algorithm does not exist.

   (b) *(15 Points)* Design a dynamic programming algorithm that finds a minimum size vertex cover on a tree in $O(m+n)$ time. *(Note: justify why the running time of your algorithm is indeed $O(m+n)$.)*

   (c) *(5 Points)* Is the polynomial time algorithm in (b) in contradiction with the fact that the vertex cover is a well-known NP-complete problem? Please explain.

   Answer:

(a) In 10.2, we saw a greedy algorithm for finding an independent set on trees. We also know that the vertex cover (of size at most k) is the complement of the independent set (of size at least k).

Alternatively, you can define a greedy algorithm directly for the Vertex Cover problem. We can make the following key observation: if v is a leaf and u is its parent, there exists a minimum vertex cover that contains u. The proof is similar, i.e., by exchange argument: the edge (u, v) needs to be covered by at least one node, and u is a better option than v, because it covers edge (u, v) plus potentially more edges. Based on this key observation, we can design a greedy algorithm:

- find a leaf v;
- include its parent u to the vertex cover;
- delete u, v and all the edges incident to them;
- repeat until there are no edges left;

(b) *Sketch:* In section 10.2, there is a DP algorithm for the weighted independent set on trees. You can use the same algorithm for the unweighted version, considering that all nodes have weight 1; in that case, the weight of the chosen independent set is simply its cardinality. *(Note that the DP approach is an overkill in the unweighted version of the problem, which can be solved by a greedy algorithm. Nevertheless, it is a valid DP algorithm that solves the problem optimally.)* You can then take the complement of the independent set problem and find the vertex cover. The number of subproblems is exactly the number of vertices n and with a little care (visiting nodes in postorder) the algorithm can be made linear $O(m + n)$.

Alternatively, you can design a DP algorithm directly for the vertex cover problem, as we did in the previous question. Section 10.1, e.g., slides 7-9, also provide ideas for building a recursion.

(c) No, it is not a contradiction. The problem is NP-complete, but this question considers only a special type of input, i.e., trees as opposed to general graphs.

4. *(25 Points)* **The Traveling Salesman Problem.**

Let's consider the optimization version of the traveling salesman problem (TSP) : We are given a graph $G(V, E)$ with $n = |V|$ nodes. Each edge of the graph has an associated distance d_{ij}. The goal is to find a *tour*, *i.e.,* a cycle that passes through every node exactly once, with minimum total distance.

(a) *(6 Points)* State the decision version of the TSP problem. Show that the decision version of TSP is in NP.

(b) *(6 Points)* How would you solve the TSP problem using a brute force approach? What would be the running time of that approach?

(c) *(13 Points)* Develop a dynamic programming approach that solves this problem and report the running time of your algorithm.

Answer:

(a) The decision version of TSP is in Section 8.5, p.479 in your textbook:

> We are given a graph $G(V, E)$ with $n = |V|$ nodes and a number k. Each edge has an associated distance d_{ij}. Is there a tour with length at most k?

This is in NP, because we can efficiently check whether a sequence of nodes is a YES answer to the decision problem. A valid solution must have $n+1$ nodes, all nodes should be visited once except the first/last node. This can be done *e.g.,* by reading every node, increasing the corresponding counter, and then check the values of the counters at the end. While reading the nodes in the claimed solution, we can also incrementally compute the length of the tour.

(b) The brute force approach would consider all possible permutations of n cities which is in polynomial factor of $O(n!)$.

(c) A faster approach would be to use DP to intelligently enumerate less paths. We will define subproblems which correspond to the initial portion of the tour. Suppose we have started at node 1 as required, have visited a few nodes and are now in node j. Let us define the following subproblem:

> For a subset of nodes $S \subset \{1, 2, ...n\}$ that include 1 and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot both start and end at node 1. Now let us express $C(S, j)$ in terms of smaller subproblems. The second to last city has to be some $i \in S$ so that the overall path length is the shortest possible:

$$C(S, j) = min_{i \in S: i \neq j} C(S - \{j\}, i) + d_{ij}.$$

The subproblems are ordered by S. There are at most $2^n \cdot n$ subproblems and each of them takes linear time to solve. The total running time is therefore $O(n^2 \cdot 2^n)$. This is better than the brute force, but still exponential, which is no surprise since this is an NP-hard problem.