Algorithms 2007 | Tutorial questions | The University of Sydney
2016 semester 2 | Tutorial 12 | School of IT

Pre-tutorial questions

Do you know the basic concepts of this week's lecture content? These questions are only to test yourself. They will not be explicitly discussed in the tutorial, and no solutions will be given to them.

1. What does it mean in practice that a problem is NP-hard or NP-complete?
2. What is a c-approximation algorithm?
3. If we can solve the Vertex Cover problem for trees in polynomial time why doesn't this contradict the fact that Vertex Cover problem is NP-complete?

Tutorial

Problem 1

Consider the following simple greedy algorithm that processes the items in arbitrary order. For each item, it attempts to place the item in the first bin that can accommodate the item. If no bin is found, it opens a new bin and puts the item within the new bin. Prove that the greedy algorithm is a 2-approximation algorithm for the bin-packing problem.

Solution: Two simple proofs for this.

Proof 1: First note that it is impossible for 2 bins to be at most half full because such a possibility implies that at some point, exactly one bin was at most half full and a new one was opened to accommodate an item of size at most $V/2$. But since the first one has at least a space of $V/2$, the algorithm will not open a new bin for any item whose size is at most $V/2$. Only after the bin fills with more than $V/2$ or if an item with a size larger than $V/2$ arrives, the algorithm may open a new bin.

Thus if we have B bins, at least $B-1$ bins are more than half full. Therefore $\sum_{i=1}^{n} a_i > \frac{B-1}{2}V \sum_{i=1}^{n} a_i > \frac{B-1}{2}V$. Because $\frac{\sum_{i=1}^{n} a_i}{V} \frac{\sum_{i=1}^{n} a_i}{V}$ is a lower bound of the optimum value OPT, we get that $B-1 < 2OPT$ and therefore $B \leq 2OPT$.

Proof 2: Suppose greedy algorithm returns more than $2OPT$ bins. If we take any two successive bins, together they must contain at least V of items (otherwise only one bin would be sufficient). Since we have at least OPT pairs plus one extra bin, we have together more than $OPTV$ items, a contradiction.

Problem 2

Consider the following algorithm for the Euclidean travelling salesman problem. As input we are given a set P of n points in the plane. Compute a minimum spanning tree of P. Double all the edges such that a tour T is formed. Prove that T is at most twice a long as an optimal solution for the Euclidean travelling salesman problem.

Solution: Let H_{opt} denote the optimal tour. Observe that a TSP tour with one edge removed is a spanning tree (not necessarily MST).
It implies that the weight of the MST T is a lower bound on the cost of an optimal tour, that is $c(T) < c(H_{opt})$.
A walk along the MST extended tour, denoted W, traverse every edge of T exactly twice, hence, $c(W) = 2c(T)$ which means $c(W)2c(H_{opt}$ and we have $c(W)/c(H_{opt}) \leq 2$.

Problem 3

As input, Knapsack takes a set of n items, each with value v_i and weight w_i, and a knapsack with weight bound W (for simplicity we assume that all elements have $w_i < W$). Find a subset I of the items that maximizes the value $\sum_{i \in I} v_i$ subject to the constraint $\sum_{i \in I} w_i \leq W$.

Knapsack is known to be NP-hard. Using dynamic programming, we can get an exact solution for knapsack in time $O(nW)$. Unfortunately, this is not polynomial in the size of its representation since W could be very large. Thus, we go back to our toolbox of approximation algorithms.

1. Lets try a basic greedy algorithm. Intuitively, we want the items with the most bang for the buck.

Algorithm 1 GreedyKnapsack

```
1: procedure GreedyKnapsack($\{v]\}, \{w\}, W$)
2:     Sort items in non-increasing order of $v_i/w_i$.
3:     Greedily pick items in the above order as long as adding an item
       to the collection does not exceed the capacity of the knapsack.
4: end procedure
```

Prove that this algorithm can be arbitrarily bad.

Solution: Consider two items. The first item has weight 1 and value 2. The second item has weight W and value B. The greedy algorithm will only pick the first item.

2. We make the following small adjustment to our greedy algorithm:

Algorithm 2 GreedyKnapsack Redux

```
1: procedure GreedyKnapsack($\{v]\}, \{w\}, W$)
2:     Sort items in non-increasing order of $v_i/w_i$.
3:     Greedily pick items in the above order until we hit an item $i$ that
       is too big to fit.
4:     Select either $\{1, 2, \ldots, i-1\}$ or item $\{i\}$.
5: end procedure
```

At first glance, this seems to be a clumsy hack in order to deal with the above counterexample. However, it turns out that this algorithm has far better performance. Prove that this algorithm is a 2-approximation algorithm.

Solution: Let k be the index of the first item that is not accepted by our algorithm. We employed a greedy algorithm. Therefore we can say that if our solution is suboptimal, we must have some leftover space WS, where S is the solution constructed by our algorithm. We know that $v_1 + v_2 + \ldots + v_k \geq OPT$. In fact, imagine for a second that our algorithm was able to take a fraction of an item then $v_1 + v_2 + \ldots + \alpha \cdot v_k \geq OPT$ where $\alpha = \frac{WS}{w_k}$. Therefore, either $v_1 + v_2 + \ldots + v_{k-1} \geq \frac{1}{2}OPT$ or $v_k \geq \alpha \cdot v_k \geq \frac{1}{2}OPT$.

Solution: Let us root the tree at some vertex r, and let us denote by T_u the subtree made up of u and its descendants. We use the following DP states:

$$M[u,i] = \begin{cases} \texttt{True} & \text{if there is constrained } k\text{-colouring } \chi \text{ of } T_u \text{ such that } \chi(u) = i \\ \texttt{False} & \text{otherwise} \end{cases}$$

First, we can rule out states that give u an invalid color, we can immediately set $M[u,i] =$ `False` if $i \notin \phi(u)$. If $i \in \phi(u)$ we use the following recurrence

$$M[u,i] = \bigwedge_{v \text{ child of } u} \left(\bigvee_{\substack{j \in [1,k] \\ j \neq i}} M[v,j] \right)$$

In other words, if for every child v of u we can find a valid colouring of T_v such that v is not coloured i then $M[u,i]$ should be set to `True`.
State $M[u,i]$ takes $O(\deg(u)k)$ time to fill so the total time spent filling the DP table is

$$\sum_{u,i} O(\deg(u)k) = O(mk^2) = O(nk^2).$$

Problem 4
Consider the following problem defined on a set of clients C and a set of facilities F. Each facility $i \in F$ has associated an opening cost f_i and for each client-facility pair (j,i) there is a cost $d_{j,i}$ associated with client j being assigned to facility i. (You can think of $d_{j,i}$ as the distance from j to i.) The objective of the problem is to open a subset of the facilities so that clients can be assigned to a close-by facility.

Given F, C $f : F \to \mathbb{Z}^+$ and $d : C \times F \to \mathbb{Z}^+$, the *facility location* problem is to choose a set $X \subset F$ minimizing

$$\sum_{i \in X} f_i + \sum_{j \in C} \min_{i \in X} d_{j,i}.$$

1. Show that the facility location problem NP-hard
2. Give an $H_{|C|}$ approximation algorithm based on the greedy algorithm for set cover.

Solution: The problem is NP-hard because it contains minimum set cover as a special case. Let $(U, \mathcal{S})$ be an instance of minimum set cover where U is a ground set and $\mathcal{S}$ is a collection of subsets of U; the objective is to a minimum number of sets $\mathcal{C} \subseteq \mathcal{S}$ whose union equals U. The set U plays the roles of the clients and the set $\mathcal{S}$ plays the role of the facilities, which we set to have an opening cost of 1. For an element $u \in U$ and $S \in \mathcal{S}$ we define $d_{u,S} = 0$ if $u \in S$ and $d_{u,S} = \infty$ otherwise. It is easy to check that any solution to the facility location problem with finite cost is a valid set cover and that cost of the facility location solution is the cardinality of the associated cover.
It turns out that the reduction goes in the other direction as well. The goal is to cover the clients. For each facility i and each subset X of the clients, we create a subset with weight $f_i + \sum_{j \in X} d_{j,i}$. A solution to the set cover induces a solution to the facility location that is no more expensive than the cover. On the other hand, every solution to the facility location problem induces a cover with the same weight. Therefore, by approximating the minimum weight set cover we can approximate the facility location problem.
The is one obstacle, however. The greedy algorithm cannot be run directly on the set cover instance because the number of sets is exponential on the number of clients. To work around this, we run the greedy algorithm implicitly. To find the most cost effective set in one iteration of the greedy algorithm we do the following for each facility i. First, we sort the remaining uncovered clients in increasing order of distance to i: $d_{j_1,i} \leq d_{j_2,i} \leq \cdots$. Second, we find the index k minimizing $\frac{f_i + \sum_{\ell=1}^{k} d_{j_\ell,i}}{k}$; the set $(i, \{j_1, \ldots, j_k\})$ is the most cost effective set among all sets induced by i. Thus, the most cost effective set can be computed in polynomial time even though there are an exponential number of set to choose from.

Problem 5
Consider the following local search algorithm for vertex cover with the objective of minimizing the size of the cover. The algorithm start with the complete set of vertices. Let S be a vertex cover. There are two types of local moves while at S:

1. Remove one vertex from S
2. Remove two vertices from S and add one to it.

The only restriction is to maintain feasibility. Show that this algorithm can get stuck in local optima that are much larger than the global optimum.

Solution: Consider a complete bipartite graph with two vertices on the left an $n - 2$ vertices on the right. An optimal vertex cover is formed by the two left vertices; however, the right hand side vertices form a locally optimal vertex cover. Therefore, the algorithm can get stuck in arbitrarily bad solutions.