HW10 Solutions

Data Structure and Algorithm (New York University)

EL9343 Homework 10

Due: Nov. 23rd 8:00 a.m.

1. Let $G = (V, E)$ be a DAG. There must be a vertex t that has no outgoing edge[1]. Fix this node t, and for every node $v \in V$, let $P(v)$ be the number of distinct paths from v to t. Define $P(v) = 0$ if no such path exists and define $P(t) = 1$ for convenience. Give a polynomial time algorithm to compute $P(v)$ for every node v. Please also analyze the time complexity of your algorithm.
   Solution:
   Run a topological sort on G. Let $label(v)$ be the value assigned to v by the sort and $N(v)$ be the neighbor-set of v. Then,

$$P(v) = \begin{cases} 0 & label(v) > label(t), \\ 1 & v = t, \\ \sum_{v' \in N(v)} P(v') & \text{otherwise} \end{cases}$$

   Thus, we can compute the $P(v)$ in decreasing order of $label(v)$, since all $v' \in N(v)$ has a larger label than v.
   The time complexity of the algorithm is $O(|V| + |E|)$. The topological sort takes $O(|V| + |E|)$ time. The above calculation of $P(v)$ iterates over all the nodes, and every edge is considered exactly once when finding the neighbors, thus the time is at most $O(|V| + |E|)$. This gives the overall time complexity.
   Note (Proof for at least one vertex has no outgoing edge):
   Pick an arbitrary vertex v and follow an arbitrary path "away from" v until you reach a vertex that has no outgoing edges. Since there are no cycles, the path will never visit any vertex twice and hence the above process must terminate after some number of steps, proving the existence of t (otherwise, all nodes have outgoing edges, such process will never terminate).

2. (**Maximum Independent Set in Trees**[2]) In an undirected graph $G = (V, E)$, an *independent set* S is a subset of the vertex set V that contains no edge inside it, i.e. S is an *independent set* on $G \Leftrightarrow S \subseteq V, \forall u, v \in S \rightarrow \{u, v\} \notin E$.
   Given a rooted tree $T(V, E)$ with root node r, find an independent set of the maximum size. Briefly describe why your algorithm is correct.
   Solution:
   Define:

$$IS(T) = \text{size of maximum independent set in } T$$
$$IS_{with\ root}(T) = \text{size of maximum independent set in } T \text{ that includes the root}$$
$$IS_{without\ root}(T) = \text{size of maximum independent set in } T \text{ that excludes the root}$$

   So the maximum independent set has size $IS(T)$, where T is the given rooted tree. If the root is included in the set, then all the sub-trees must excluded their roots (otherwise not an independent set). If the root is excluded, then each sub-tree can include or exclude its root, whichever has the larger size. As the result, the recursive formula of the problem is,

$$IS_{without\ root}(T) = \sum_{T' \text{ is sub-tree of } T} IS(T')$$
$$IS_{with\ root}(T) = 1 + \sum_{T' \text{ is sub-tree of } T} IS_{without\ root}(T')$$
$$IS(T) = \max\{IS_{with\ root}(T), IS_{without\ root}(T)\}$$

3. Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value.
   (a) Give an greedy algorithm to find an optimal solution to this variant of the knapsack problem;

[1] You don't need to prove this statement in the question, but you can try to do it yourself. Please don't include the proof in your submission. We will post the proof in the solution for you to check.

[2] Finding maximum-sized independent sets in general graphs is NP-complete. So we will focus on tree cases.

(b) Prove that your algorithm is correct (by showing greedy-choice property and optimal substructure property).

Solution:

(a) First we sort the items by the weight, and then we take one item each time until we are unable to take more. This gives the optimal solution.

(b) Greedy-choice property:
Suppose that exists an optimal solution that the item j is taken and the item $i < j$ is not taken, while we have $W_j > W_i$ and $V_j < V_i$. Then, we can take item j out of the knapsack and put item i in the knapsack to get a higher value solution. That is contradiction to the original solution being optimal.
Optimal substructure property:
If item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W - W_j$ that can be taken from the $n - 1$ items other than j.

4. Suppose you have an unrestricted supply of pennies, nickels, dimes, and quarters. You wish to give your friend n cents using a minimum number of coins.

(a) Describe a greedy strategy to solve this problem;

(b) Provide the pseudo-code and write down its time complexity;

(c) Prove the correctness of your algorithm.

Solution:

(a) Use as many quarters as possible, then dimes, then nickels, and finally pennies.

(b) The pseudo-code is as follows,

GREEDY-CHANGE(n)
$n_1 = n \mod 25$ (mod here means to get the remainder)
$a = (n - n_1)/25$
$n_2 = n \mod 10$
$b = (n_1 - n_2)/10$
$n_3 = n \mod 5$
$c = (n_2 - n_3)/5$
$d = n_3$
return a, b, c, d

The time complexity is $O(1)$.

(c) Greedy property:
First observe that no optimal solution can have ≥ 5 pennies (use nickel instead), ≥ 2 nickels (use dime instead), ≥ 3 dimes (use a quarter and a nickel instead).
Next, we are going to prove the greedy property using a case by case analysis[3].

i. $1 \leq n < 5$, this is clearly true that we have to use just pennies.

ii. $5 \leq n < 10$, we pick a nickel in this case, as the alternative is to use ≥ 5 pennies.

iii. $10 \leq n < 25$, we pick a dime in this case. Otherwise we have to use ≥ 5 pennies or ≥ 2 nickels (or both), none of which can be optimal.

iv. $n \geq 25$. From the observation, the maximum cents we can generate with our constraints on the number of pennies, nickels and dimes is $(4 * 1 + 1 * 5 + 2 * 10) = 29$. So for any $n > 29$, we have to use quarters.
Also, for $25 \leq n \leq 29$, notice that we have to use $n - 25$ pennies, and to get 25 cents the optimal way is to pick a quarter.

Therefore, with any number of n, we can get the optimal solution following greedy choice.
Optimal substructure property:

[3]This kind of proof doesn't work in all cases, but I find it the best way in this one. An alternative (and more general) way to prove this is to assume an optimal solution that is different from the one generated by the algorithm, and then to find some contradiction.

If a coin of value v_i is removed from an optimal solution with number of coins as m, the remaining solution is optimal that uses $m-1$ coins to give $n-v_i$ cents.