Homework 4. Due May 14

CS180: Algorithms and Complexity
Spring 2018

GUIDELINES:

- Upload your assignments to Gradescope by 5:59 PM.
- Follow the instructions mentioned on the course webpage for uploading to Gradescope very carefully (including starting each problem on a new page and matching the pages with the assignments); this makes it easy and smooth for everyone. As the guidelines are simple enough, bad uploads will not be graded.
- You may use results proved in class without proofs as long as you state them clearly.
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly. Homework is a stepping stone for exams; keep in mind that reasonable partial credit will be awarded and trying the problems will help you a lot for the exams.

1. Problem 5.1 from [DPV]. You should break ties between edges of same weight (if needed) in lexicographic order. [.75 points]

   (In second part of part(c), "For each edge in this sequence, give a cut that justifies its addition." means to find a cut for which the edge being added is an edge of minimum weight crossing the cut.)

2. You are given a connected graph G with n vertices and m edges, and a minimum spanning tree T of the graph. Suppose one of the edge weights $c(e)$ for an edge $e \in T$ is updated. Give an algorithm that runs in time $O(m)$ to test if T still remains the minimum spanning tree of the graph. You may assume that all edge weights are distinct both before and after the update. Explain why your algorithm runs in $O(m)$ time and is correct. [.75 points]

   (Hint: Consider the cut obtained by deleting e from T. Note that as you are only allowed $O(m)$ time, you cannot recompute a MST in the new graph from scratch. Also, make sure you specify your algorithm fully.)

   Solution. MSTCHECKING(G, T, u, v, weight)

   (a) $NODES[l] \leftarrow 0$ for every nodes l.
   (b) Remove $e = \{u, v\}$ in T.
   (c) Run DFS from u on T, set $NODES[l] = 1$ for every nodes l visited during DFS.

(d) Run DFS from v on T, set $NODES[l] = -1$ for every nodes l visited during DFS.

(e) For every edge $e' = \{u', v'\}$ in G:

i. if $NODES[u'] + NODES[v'] = 0$ and $w(e') < weight$ Return False.

(f) Return True

Correctness:
When we delete e from T, the nodes of T becomes 2 sets s_1 and s_2. Let s_1 be the cut, so the MST contains the least weight edge crossing the cut, if the updated weight edge e is still minimal, then T still remains the MST.
Complexity:
Every steps from a to e in our algorithm runs in $O(m)$ time, so the complexity is $O(m)$.

3. When their respective sport is not in season, UCLA's student-athletes are very involved in their community, helping people and spreading goodwill for the school. Unfortunately, NCAA regulations limit each student-athlete to at most one community service project per quarter, so the athletic department is not always able to help every deserving charity. For the upcoming quarter, we have S student-athletes who want to volunteer their time, and B buses to help get them between campus and the location of their volunteering. There are F projects under consideration; project i requires s_i student-athletes and b_i buses to accomplish, and will generate $g_i > 0$ units of goodwill for the university.

Use dynamic programming to produce an algorithm to determine which projects the athletic department should undertake to maximize goodwill generated. For full-credit, your algorithm should run in time $O(SBF)$ but you don't have to prove its correctness or analyze the time complexity. [.75 points]

(Hint: This is similar to the knapsack problem we did in class with the caveat that we have two 'constraints': number of students and number of buses (just as the weight was a constraint in knapsack). Can you use this to choose the right subproblems? Also, note that you have to determine the best set of projects and not just the best value. For this part, you can design an algorithm similar to the one we used for knapsack based on working back from the recurrence you get.)

Solution. This problem resembles the classic knapsack problem where each 'item' in the 'knapsack' is a project. In this case, each project has two components to its 'weight': the number of student athletes it needs, s_i, and the number of buses b_i. The value of the knapsack is simply the sum of the goodwill values g_i corresponding to the projects being used in the 'knapsack'. We can define our problem as follows:

Let $goodwill_i(s, b)$ be the goodwill obtained having considered using projects $i = 0, 1, ..., F$ where $i = 0$ represents having considered doing none of the projects. Each $goodwill_i$ depends on all previous $goodwill$ values from 0 through i. We can define $goodwill_i$ in terms of $goodwill_{i-1}$ since each $goodwill$ value incorporates the sum of goodwills over range of projects up until i. If one decides to use a project i and the previous $i-1$ projects have been decided on, then the change in the overall goodwill value from $i-1$ is just the gain in goodwill via project i, g_i, plus the net gain from all the other projects having been considered, $goodwill_{i-1}$.

The other possibility for setting each *goodwill* value is that the previous project $i-1$ may have NOT been used, so then we have to account for no cost to the number of student athletes and the number of buses.

These two possibilities are incorporated as follows:
$goodwill_i(s,b) = max(goodwill_{i-1}(s-s_i, b-b_i) + g_i, goodwill_{i-1}(s,b))$

For initialization, we set all $goodwill_0(s,b) = 0$ for all integers s, b since there is no way these values can have goodwill since no projects are considered. Then we loop through successive $goodwill_i(j,k)$ for $i = 1, ...F$ and for each i, we apply the update formula above for all $0 \leq j < S, 0 \leq k < B$ (anytime j or k is negative when using the update formula, we treat that $goodwill_i$ value to be $-\infty$ since that scenario is disallowed).

The algorithm can therefore be written as:

(a) Set $goodwill_0(s,b) = 0$, $0 \leq s \leq S$, $0 \leq b \leq B$

(b) for $i = 1, \cdots, F$:
  for $s = 1, \cdots, S$:
    for $b = 1, \cdots, B$:
      Set $goodwill_i(s,b) = max(goodwill_{i-1}(s-s_i, b-b_i) + g_i, goodwill_{i-1}(s,b))$

The required result is then $goodwill_F(S,B)$.

The correctness of the algorithm follows easily by induction on i. The base-case of $i = 0$ is correct as we've hardcoded it into the algorithm. Further, if $goodwill_j(s,b)$ is computed correctly for all triples such that $j < i$, then $goodwill_i(s,b)$ would be computed correctly as we are computing the entry by the correct recurrence.

Time-complexity: The algorithm takes time $O(B \cdot S \cdot F)$ as we have to iterate through all possible values for all possible values of $1 \leq i \leq F$, $0 \leq j < S$ and $0 \leq k < B$.

4. You have a knapsack of total weight capacity W and there are n items with weights $w_1, \ldots, w_n$ respectively. Give an algorithm to compute the number of different subsets that you can pack safely into the knapsack. In other words, given integers $w_1, \ldots, w_n, W$ as input, give an algorithm to compute the number of different subsets $S \subseteq [n]$ such that $\sum_{i \in S} w_i \leq W$. For full-credit, your algorithm should run in time $O(nW)$ but you don't have to prove its correctness or analyze the time complexity. [.75 points]

   (Hint: This is similar to the knapsack problem, but instead of looking for the optimal subset you are looking for the total number of *feasible* subsets. Nevertheless, you can use build on the same basic approach.)

Solution. We will solve the problem by a case-analysis as we did for other dynamic programming problems in class. For $1 \leq i \leq n$ and $0 \leq w \leq W$, let $C(i,w)$ be the number of feasible combinations of the first i items $1, 2, \ldots, i$ that fit safely into a knapsack of capacity w. We adopt the convention that $C(i,w) = 0$ if w is negative.

Now, suppose we have computed $C(i,w')$ for all $1 \leq w' \leq W$ and would like to compute $C(i+1,w)$ for some capacity w. Then, we have just two options for the $i+1$'th item: either we take the item or not. Further, if we include the item $i+1$, then the *available* capacity of

the knapsack drops to $w - w_{i+1}$; so the number of feasible configurations of the remaining items would be $C(i, w - w_{i+1})$. Hence, we have the recurrence:

$$C(i+1, w) = C(i, w) + C(i, w - w_{i+1})$$

The above relation immediately gives us the following algorithm for computing C:

(a) for $0 \leq w \leq W$, let $C(1, w) = 1$ if $w_1 > w$ (the only feasible solution is the empty set) and $C(1, w) = 2$ if $w_1 \leq w$.

(b) for $i = 1, \ldots, n-1$:
  for $w = 0, \cdots, W$:
    compute $C(i+1, w) = C(i, w) + C(i, w - w_{i+1})$.

Our final desired answer is $C(n, W)$.

The correctness of the algorithm follows easily by induction. The base-case of $i = 1$ is correct as we've hardcoded it into the algorithm. Further, if $C(i, w)$ are correctly computed for all $0 \leq w \leq W$, then $C(i+1, w)$ would all be correctly computed as we use the correct recurrence relation.

Time-complexity: The algorithm has $n \cdot W$ iterations, with O(1) work per iteration. Therefore, the total run-time is $O(nW)$.