Homework 5 Solutions

CSE6140, Fall 2017

November 2017

Problem 1

Solution:

Algorithm Description: Let us denote Atlanta as stop d_0 and New York as stop d_{n+1}, so that $\{d_1, \ldots, d_n\}$ represent the distances of intermediate gas stations between the origin and destination. Assume without loss of generality that we start with an empty tank of gas in Atlanta, such that d_0 is our first fueling stop in all cases. For our greedy strategy, we fill the tank to capacity each time we stop for gas. We start the trip from Atlanta with a full tank of gas and travel to the farthest gas station along the route within K miles of Atlanta. Once we have refilled the tank, we again travel to the next farthest gas station within K miles of the previous stop. Repeat this process until we reach New York.

Representation of Solutions: Let us represent a solution obtained using our greedy strategy as a set of x indices $\mathcal{G} = \{g_1, \ldots, g_x\}$ representing which gas stations we stop at, with $g_i \in \{1, \ldots, n\}$. Similarly, we can represent an optimal solution as $\mathcal{O} = \{o_1, \ldots, o_y\}$, where again the indices $o_j \in \{1, \ldots, n\}$ represent the gas stations at which to stop.

Proof by Greedy Choice + Optimal Substructure Properties

Greedy Choice Property: We propose that we can replace the first stop o_1 from the optimal solution with the first stop from the greedy solution g_1 to construct a new solution $\mathcal{S} = \{g_1, o_2, \ldots, o_y\}$. We know that $d_{g_1} \geq d_{o_1}$, since the greedy strategy leaves Atlanta with at least as much gas as the optimal strategy does (e.g. if optimal strategy $\mathcal{O}$ does not fill the tank to capacity in Atlanta), and since station $d_{g_1+1} > K$ miles from Atlanta and is thus unreachable from Atlanta without refueling. We also know that $d_{o_2} - d_{o_1} \leq K$, since the second stop in the optimal solution must be reachable from the first stop. Together, this gives $d_{o_2} - d_{g_1} \leq d_{o_2} - d_{o_1} \leq K$. This shows that the greedy choice can be made in place of the first choice in $\mathcal{O}$, keeping the number of stops the same, and that this does not hurt the feasibility of the resulting solution $\mathcal{S}$.

Optimal Substructure Property

Let P be the problem of selecting refueling stations between d_0 and d_n, for which we have an optimal solution $\mathcal{S}$ consisting of a set of locations to stop at for gas. The first stop in $\mathcal{S}$ is at station g_1 at distance d_{g_1}. Given that we fill the gas tank fully at station d_{g_1}, the remaining subproblem is to select refueling stations between $\{d_{g_1+1}, \ldots, d_n\}$, starting at location d_{g_1} instead of Atlanta. We denote this subproblem as P'. Suppose we have a solution $\mathcal{S}'$ to this subproblem. Then, the solution $\mathcal{S} = g_1 \cup \mathcal{S}'$, and $|\mathcal{S}| = 1 + |\mathcal{S}'|$. If $\mathcal{S}'$ is *not* an optimal solution for P', then there must exist another solution to P', $\mathcal{S}''$ such that $|\mathcal{S}''| < |\mathcal{S}'|$. In this case, we use this to construct a solution for P using $|\mathcal{S}''| + 1 < |\mathcal{S}|$ stops, which contradicts the optimality of $\mathcal{S}$. Hence, the optimal solution to P must include the optimal solution to P' within it.

Proof by Exchange Argument

Assume that our greedy is not optimal. That is, it produces a solution that has more stops than strictly necessary. Let the sequence of stops of our greedy approach be $g_1, \cdots, g_x$. Since this is not optimal, an optimal sequence of stops must contain less stops. Let $o_1, \cdots, o_y$ be such an optimal solution, where $y < x$. Let k be the largest index for which: $g_1, \cdots, g_k = o_1, \cdots, o_k$. Consider the stop $k+1$. We know that $g_{k+1}! = o_{k+1}$. Since our greedy approach selected the $k+1$ gas station as the farthest away from $g_k(=o_k)$, but within $\mathcal{K}$ miles of g_k, o_k must be closer to g_k than g_{k+1}. Note, if o_k were even farther away from g_k, your car would have run out of gas and hence the optimal solution wouldn't even be a solution. Hence, we can replace o_{k+1} with g_{k+1} in the optimal solution, without affecting neither the size nor the correctness of the optimal solution.
Now, we repeat the same exchange procedure for each subsequent stop from $k+2$ to y, transforming the optimal solution into $o_1, \cdots, o_k, g_{k+1}, \cdots, g_y = g_1, \cdots, g_k, g_{k+1}, \cdots, g_y$. Remember that $y < x$. Since the optimal solution is valid solution, it means that stop o_y is within $\mathcal{K}$ miles from New York and hence you wouldn't need to stop for gas anymore. But by our exchange argument above g_y is either equal to o_y or even closer to New York than o_y is. Thus, our greedy algorithm would have stopped after g_y without producing the additional stops $g_{y+1}, \cdots, g_x$. This contradicts our assumption that $y < x$, and this in turn contradicts our original assumption that our greedy method is not optimal. Hence, our greedy method is in fact optimal as claimed.

Pseudo code: O(n) runtime The greedy algorithm we use is to go as far as possible before stopping for gas. Let c_i be the city with distance d_i from Atlanta

---

S={}
last = 0
for $i = 1 \rightarrow n$ **do**
  if $(d_i - last > K)$ **then**
    $S = S \cup \{c_{i-1}\}$
    $last = t_{i-1}$
  end if
end for

Problem 2

Part 1

Define subproblems: Let D_x be the number of ways to write x as the sum of $1, 3, 4$
Optimal Substructure
This problem has optimal substructure property. Since the last integer in the expression of x can either be 1,3 or 4, we are left with the subproblem of expressing $x-1, x-3$ and $x-4$ as a combination of 1, 3 and 4's. Let D_x be the optimal solution (i.e. be the total number of ways to express x). So D_x should be the combination of maximum number of ways of expressing $x-1, x-3$ and $x-4$. Suppose the solutions to these subproblems are themselves not optimal. Then, if we have the actual optimal solution to these subproblems, we can substitute them into the solution for the overall problem (total number of ways of expressing x) and obtain even better global optimal solution. However, this contradicts the original claim that we started out with an optimal solution D_x in the first place. Therefore, the problem exhibits optimal substructure.

Recurrence is then

- Consider one possible solution $i = p_1 + p_2 + \cdots + p_m$ for subproblem with number i, $0 \leq i \leq x$
- If $p_m = 1$, the rest of the terms must sum to $i - 1$
- Thus, the number of sums that end with $p_m = 1$ is equal to D_{i-1}
- Take other cases into account ($p_m = 3, p_m = 4$)

$$D_i = D_{i-1} + D_{i-3} + D_{i-4} \tag{1}$$

Solve the base cases:

- $D_0 = 1$
- $D_i = 0\,(\forall i \leq 0)$
- Alternatively, we can set: $D_0 = D_1 = D_2 = 1$ and $D_3 = 2$

Where to find the solution: D_x.

Pseudo code: O(x) runtime

Initialize $D[0] = D[1] = D[2] = 1; D[3] = 2$
for $(i = 4; i \leq x; i++)$ **do**
    D[i] = D[i-1]+D[i-3] + D[i-4]
end for
return $D[x]$

Part 2

We are given an array of values $v_1, v_2, \cdots, v_n$ where v_i represents the value of gift i. Assume both players play intelligently and to win (i.e., given a choice between selecting gifts of values 5 and 7, they would choose the gift with value 7).

Optimal Substructure (3 pts)

Let $OPT(i, j)$ be the maximum value Alan can obtain if it is his turn for the sequence of gifts $v_i, v_{i+1}, \cdots, v_{j-1}, v_j$. We have two cases: Alan picks 1) gift i or 2) gift j.

1) Alan picks gift i. Now Charlie can pick gift $i + 1$ or j (whichever has maximum value). Suppose Charlie picks gift i. This leaves the subproblem $OPT(i + 1, j)$. Alternatively if Charlie picks gift j, this leaves the subproblem $OPT(i + 1, j - 1)$.

2) Alan picks gift j. Now Charlie can pick gift i or $j - 1$. If Charlie picks gift i, this creates the subproblem $OPT(i + 1, j - 1)$. If Charlie picks gift $j - 1$ this creates the subproblem $OPT(i, j - 2)$.

To maximize his total value, Alan will select the choice that leads to the maximum global optimum $OPT(i, j)$. First note we have overlapping subproblems: the subproblem $OPT(i + 1, j - 1)$ appears in both cases. Suppose we are given the global optimal solution $OPT(i, j)$. We have the resultant subproblems as characterized above, namely $OPT(i+1, j)$, $OPT(i+1, j-1)$, and $OPT(i, j - 2)$. To claim this problem has optimal substructure, we must have optimal solutions to the subproblems within the optimal solution to the global problem. Suppose the

solutions to those subproblems themselves are not optimal. If we have the actual optimal solution to those subproblems, we can substitute them into the solution for the overall problem $OPT(i, j)$ and obtain an even better global optimal solution. However, this contradicts the original claim that we started out with an optimal solution to $OPT(i, j)$ in the first place. Therefore, this problem exhibits optimal substructure.

Recurrence (4 pts)

Again, let $OPT(i, j)$ be the maximum value Alan can obtain if it is his turn for the sequence of gifts $v_i, v_{i+1}, \cdots, v_{j-1}, v_j$. We have two cases:

1) Alan picks gift i. So Charlie is left with the option of selecting either gift $i + 1$ or gift j (whichever has maximum value). If Charlie selects gift $i + 1$, Alan will have the choice of selecting gift $i + 2$ or j. If Charlie selects gift j, Alan will have the choice of selecting gift $i + 1$ or $j - 1$. Since we can assume Charlie will pick whichever gift is most optimal for him, Alan will therefore obtain total value of $v_i + min(OPT(i + 2, j), OPT(i + 1, j - 1))$.

2) Alan picks gift j. So Charlie is left with the option of selecting either gift i or gift $j - 1$ (whichever has maximum value). Using similar logic as above, Alan will obtain total value of $v_j + min(OPT(i + 1, j - 1), OPT(i, j - 2))$.

Therefore,

$$OPT(i, j) = max(v_i + min(OPT(i + 2, j), OPT(i + 1, j - 1)), \\ v_j + min(OPT(i + 1, j - 1), OPT(i, j - 2)))$$

Base cases: $OPT(i, i) = v_i$ and $OPT(i, i + 1) = max(v_i, v_{i+1})$.

Algorithm (2 pts)

In the algorithm below $M(i, j)$ represents the optimal value obtainable for Alan for gifts with values $v_i, v_{i+1}, \cdots, v_j$. The global optimal is in $M(1, n)$.

Algorithm 1 Bottom up algorithm to find maximum total value obtainable for Alan.

```
1: procedure MAX_VALUE(i,j)
2:     M is n × n array of zeros
3:     for i = 1 to n do
4:         M[i, i] = v_i
5:         M[i, i + 1] = max(v_i, v_{i+1})
6:     end for
7:     for k = 2 to n − 1 do
8:         for i = 1 to n − k do
9:             j = i + k
10:            M[i, j] = max(v_i + min(M[i+2, j], M[i+1, j−1]), v_j + min(M[i+1, j−1], M[i, j−2]))
11:        end for
12:    end for
13: return M[1, n]
14: end procedure
```

Complexity Analysis (1 pt)

Time complexity is $\mathcal{O}(n^2)$ and space complexity is $\mathcal{O}(n^2)$

Problem 3

(a)

Recall that in Ford-Fulkerson algorithm, the algorithm terminates when there is no augmenting path P in G_f. After the capacity of a single edge $(u, v) \in E$ is increased by 1, it could be possible that there is a new augmenting path P, and the algorithm should be reactivated until no more paths can be found. Since only a single edge changes, we just require a one-time check.

---

for each $e \in E$ **do**
  $f(e) \leftarrow$ previous max flow assignment
end for
update $c(e)$ to $c'(e)$
if there is an $s - t$ path P in G_f **then**
  $f \leftarrow Augment(f, c, P)$
  update G_f
end if
return f

The Augment function is the same as in the lecture notes. Time complexity analysis:

- Finding an $s - t$ path P takes $O(|V| + |E|)$ using BFS or DFS.
- Updating the flow with augmenting path is $O(|V|)$ since P is a simple path with at most $(|V| - 1)$ edges.
- Finding new residual graph can be done in $O(|E|)$ time to update each edge's residual capacity. So the net time complexity to update the max flow is $O(|V| + |E|)$.

Proof of correctness: This algorithm is essentially the same as the Ford-Fulkerson algorithm if we consider the capacities are already updated at the beginning and the results of the old/previous max-flow is a partial solution step during the Ford-Fulkerson algorithm. So its proof of correctness is the same as that of the Ford-Fulkerson algorithm. Since the capacity only increases by 1, the new max-flow f' must satisfy $V(f) \leq v(f') \leq v(f) + 1$, i.e. we need at most one iteration step of Ford-Fulkerson algorithm.

(b)

Unlike (a), we cannot directly reactivate Ford-Fulkerson algorithm because we are decreasing the capacity of a single edge $e = (u, v)$ instead. Notice that if the new capacity is greater than the flow going through the edge, $c'(e) \geq f(e)$, then there is no need to update the maximum flow. Otherwise, we need to apply Ford-Fulkerson algorithm inversely to decrease a flow with amount of 1 in order to satisfy the new capacity constraint. In order to find the reverse path, we can take advantage of the residual graph, as the reverse edges hold the flow going through the network. Notice that there could be an alternative augmenting path, so Ford-Fulkerson must be executed again for at most one step. See the following pseudo code for more details.

Where Decrease is the reverse operation of Augment.

Time complexity analysis:

- Finding a reverse $t - s$ path P in G_f through $e(u, v)$ takes $O(|V| + |E|)$ using BFS or DFS
- Changing the flow with augmenting/decreasing path is $O(|V|)$ since P is a simple path with at most $(|V| - 1)$ edges;

---

if $c'(e) \geq f(e)$ **then**
  return f
end if
find a reverse $t - s$ path in G_f through $e = (u, v)$ using BFS or DFS from t to v and u to s
$f \leftarrow Decrease(f, P)$
update capacity of edge: $c'(e) = c(e) - 1$
if there is an $s - t$ path P in G_f **then**
  $f \leftarrow Augment(f, c, P)$
  update G_f
end if
return f

- Finding new residual graph can be done in $O(|E|)$ time to update each edge's residual capacity.

So the net time complexity to update the max flow is $O(|V| + |E|)$.

Proof of correctness:

- If $c'(e) \geq f(e)$, the original max-flow f is still a valid flow. Decreasing capacity cannot create a new augmenting path, so there are still no augmenting paths. Then, according to the Augmenting path theorem, the original max-flow f is still a max-flow.
- Otherwise $c'(e) < f(e)$. The original max-flow f is no more valid, and this edge $e = (u, v)$ becomes a bottleneck edge. Then, the Decrease procedure would create a new valid flow f', with a value $v(f') = v(f) - 1$. We can consider f' as a partial solution step during the Ford-Fulkerson algorithm, and then, the proof of correctness comes from that of the Ford-Fulkerson algorithm. Since $v(f') = v(f) - 1$ and the value of the new max-flow is at most $v(f)$, we need at most one additional iteration to find an additional augmenting path.

Problem 4

Part 1 & 2

True, True
One can divide all capacities by 2. On the modified graph there is an integral maximal flow. If we multiply this by 2 we get an even maximal flow on the original graph.
Alternatively, one can replace "integral" by "even" in the proof that Ford-Fulkerson is correct for integral edge weights.
Also, sum of even numbers is even.

Part 3 & 4

False, False
Counter-example, see for instance Figure 1.

Problem 5

(Part 1) We assume the adjacency-list representation of the graph. We use two arrays S and T indexed by V to mark whether a vertex can be a source or a target (respectively). Initially,

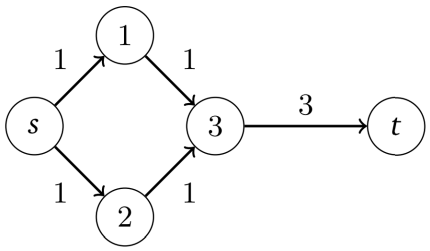

Figure 1: Vertices $1, 2, 3$ with edge capacities shown.

we mark each vertex as a potential source and a potential target. For each (directed) edge $(u, v) \in E$, we unmark u in the target array T, and unmark v in the source array S. After all the edges are considered, those vertices that are still marked in S are the sources, and those vertices that are still marked in T are the targets.

(Part 2) Assume that a DAG does not contain a source. This means that for every vertex v, there is (at least) an edge $(u, v) \in E$. Let $n = |V|$. We start with any arbitrary vertex v_0, and obtain a sequence of vertices $v_1, v_2, v_3, \cdots, v_n$ such that $(v_i, v_{i-1}) \in E$ forall $i = 1, 2, 3, \cdots, n$. Since G contains only n vertices, there must be a repetition in $v_0, v_1, v_2, \cdots, v_n$. Let $v_i = v_j$ with $0 \leq i < j \leq n$. By construction, $v_j, v_{j-1}, v_{j-2}, \cdots, v_{i+1}, v_i$ is a cycle in G, a contradiction. Analogously, the existence of a target in G can be proved.

(Part 3) Let $s_1, s_2, \ldots, s_k$ be all the sources and $t_1, t_2, \ldots, t_l$ be all the targets in G (these can be identified in $O(|V| + |E|)$ time by Part(1)). We convert G to a new DAG G' whose vertex set contains two additional vertices s and t. We add the edges (s, s_i) for all $i = 1, 2, \ldots, k$, and also the edges (t_j, t) for all $j = 1, 2, \ldots, l$. G' is a DAG with a unique source s and a unique target t. Moreover, the count of all (s_i, t_j) paths (for all i, j) in G is the same as the count of all (s, t) paths in G'. The size of G' continues to remain $O(|V| + |E|)$.

We make a topological sorting of the vertices in G'. This can be done in $O(|V| + |E|)$ time. Let the listing be $s = v_0, v_1, v_2, \ldots, v_n$, $t = v_{n+1}$. We use an array C indexed by the vertices in G' to store the count of paths from s to the vertices.

Initialize $C[v_0] = 1$ and $C[v_i] = 0$ for all $i = 1, 2, \ldots, n + 1$
for $i = 0, 1, 2, \ldots, n$ **do**
  For all edges (v_i, v_j) in G', set $C[v_j] = C[v_j] + C[v_i]$
end for
return $C[v_{n+1}]$

Since there are no back edges (that is, edges (v_i, v_j) with $i > j$), the for loop does not miss a path from s to t. With the adjacency list representation of G', this phase can again be finished in $O(|V| + |E|)$ time.

The introduction of the new vertices s, t could have been avoided. In that case, we start by setting $C[s_i] = 1$ for all the sources s_i in G. At the end, we return $C[t_1] + C[t_2] + \cdots + C[t_l]$. However, a topological sorting of G is necessary for the correctness of this algorithm.