CS 580 — Algorithm Design, Analysis, And Implementation

Fall 2021 — Vassilis Zikas — HW 3

Due November Nov. 5 at 11:59 p.m.

1. (30 points) Sara is playing a game called *Button Mash* with her boyfriend Charles and her sister Elizabeth. In Button Mash, all players are trying to cooperatively achieve the highest total score after n rounds. Each round, one of the players (or no players possibly), can choose to push a button to earn points. Sara, Elizabeth, and Charles are each given an array (J, E, and C respectively) of length n containing integers (positive or negative) where the ith element denotes the amount of points they would earn if that person pushed the button on the ith round. For example, $J[5]$ would be the amount of points that the team would earn on the 5th round if Sara pushed the button. There are two restrictions to the button pushing:

   - At most one player can push the button each round.
   - No player can push the button in two consecutive rounds.

   Design and analyze an algorithm that returns the maximum amount of points that the team can earn along with an array of length n denoting who pushed the button each round to obtain that score.

   Answer: Create a 4 length n arrays OPT_J, OPT_E, OPT_C, and OPT_N. $OPT_J[i]$ represents the maximum amount of points the team can earn by the ith round where Sara pushes the button on the ith round. Likewise, OPT_E and OPT_C are similar but for Elizabeth and Charles respectively. $OPT_N[i]$ represents the maximum amount of points the team can earn by the ith round if nobody pushes the button on the ith round. We can specify the values of these arrays using the following recurrences:

$$OPT_J[i] = \begin{cases} J[1] & \text{if } i = 1 \\ J[i] + \max\{OPT_E[i-1], OPT_C[i-1], OPT_N[i-1]\} & \text{otherwise} \end{cases}$$

$$OPT_E[i] = \begin{cases} E[1] & \text{if } i = 1 \\ E[i] + \max\{OPT_J[i-1], OPT_C[i-1], OPT_N[i-1]\} & \text{otherwise} \end{cases}$$

$$OPT_C[i] = \begin{cases} C[1] & \text{if } i = 1 \\ C[i] + \max\{OPT_J[i-1], OPT_E[i-1], OPT_N[i-1]\} & \text{otherwise} \end{cases}$$

$$OPT_N[i] = \begin{cases} 0 & \text{if } i = 1 \\ \max\{OPT_J[i-1], OPT_E[i-1], OPT_C[i-1], OPT_N[i-1]\} & \text{otherwise} \end{cases}$$

   The final solution is

$$\max\{OPT_J[n], OPT_E[n], OPT_C[n], OPT_N[n]\}.$$

   Let S denote the length n array of button pushes. We use backtracking to fill S.

   (a) Set d be the final decision made (e.g. $d \leftarrow J$ if $\max\{OPT_J[n], OPT_E[n], OPT_C[n], OPT_N[n]\} = OPT_J[n]$).

   (b) Set $S[n] \leftarrow d$.

(c) Define

$$d[i] = \begin{cases} J[i] & \text{if } d = J \\ E[i] & \text{if } d = E \\ C[i] & \text{if } d = C \\ 0 & \text{if} d = N \end{cases}$$

From $i = n-1$ to 1, check:

i. If $OPT_d[i+1] - d[i] = OPT_J[i]$, then set $d \leftarrow J$.
ii. If $OPT_d[i+1] - d[i] = OPT_E[i]$, then set $d \leftarrow E$.
iii. If $OPT_d[i+1] - d[i] = OPT_C[i]$, then set $d \leftarrow C$.
iv. If $OPT_d[i+1] - d[i] = OPT_N[i]$, then set $d \leftarrow N$.
v. Set $S[i] \leftarrow d$

(d) Return S.

Proof of correctness: Let us first look at the base cases. When $i = 1$ and Sara pushes the button, the maximum points they can earn is simply $J[1]$. When no one pushes the button in the first round, the maximum number of points they can earn is 0 i.e. $OPT_N[0] = 0$.
When Sara pushes the button at round i, the increase in points is given by $J[i]$. If Sara pushes the button at round i, in the previous round she could not have pushed the button. Thus, in the previous round either Elizabeth or Charles pushed or no one did. Thus, we choose the maximum of all three possibilities i.e. maximum score in rounds $i-1$ if Charles pushed, or Elizabeth or no one pushed.
Similar reasoning holds for Elizabeth and Charles.
For the last case when no one pushes the button on round i, in the previous round any of the three could have pushed the button or no one did. We choose the maximum of all 4 possibilities i.e. maximum score in rounds $i-1$ if any of the three picked or if no one did.
Thus, we have covered all the possible cases.

2. (30 points) Let $G : (V, E)$ be an undirected graph, let $s, t, v \in V$ be three distinct vertices in the graph. Design and analyze an efficient algorithm to find a *path* from s to t that goes though v. Recall that a path cannot repeat vertices. **Answer:** First we will show how to find vertex disjoint paths between vertices s and t in an unweighted undirected graph $G : V, E$ using max flow algorithms. To do this, we first show how to find edge disjoint paths in an undirected graph. Then, we use this to find vertex disjoint paths in the graph by adding capacity of 1 on each vertex.

We first construct a directed graph $G' : V', E'$ as follows. First, vertex set $V' = V$. For each edge $e : (u, v) \in E$, we add the two directed edges of unit capacity $(u, v), (v, u) \in E'$. It can be verified that edge disjoint paths in G can be extracted by the subgraph of directed edges saturated by max flow algorithms in G'.

Next, to get vertex disjoint paths, we can think of the vertices as having capacity 1 i.e. only one unit of flow can pass through any vertex. This would imply that the vertex can be involved in at most one path that contributes to the max flow. Therefore, to compute vertex disjoint paths we assign capacity 1 to each vertex in the graph G'. To account for non negative vertex capacities in the directed graph $G' : V', E'$, we transform the graph to get $D' : V'', E''$ as follows. We split each vertex $v \in V'$ into two nodes $v_1, v_2 \in V''$ and add the edge $(v_1, v_2) \in E''$ with capacity 1. The incoming edges $(h, v) \in E'$ become edges of the form $(h_2, v_1) \in E''$ in D' with capacity 1. The outgoing edges $(v, h) \in E'$ become outgoing edges of the form (v_2, h_1) in D' with capacity 1.

Finally, we solve the question. We add a new node u to the graph and also create two new edges (u, s) and (u, t). Now, we need to find the maximum number of vertex-disjoint paths between u and v. Note that u has degree 2 and so there will be at most two vertex-disjoint paths between u and v. The time taken to transform the graph as explained above and run the max flow algorithm is $O(|V|+|E|)$. It is not difficult to show that there will be exactly two vertex-disjoint paths between u and v if and only if there is a path from s to t that goes through v.

3. There may be multiple LP formulations of a problem, for example max flow. Answer the following questions:

   (a) (40 = 20 + 20 points) In lecture, we formulated the *max flow problem* as a linear program. Write another way of modeling the max flow as a linear program by defining variables for each s-t path. What is an interpretation of this LP – explain in sentences, as was done in class for the Chicago vs. Detroit Pizza example? In particular, explain what each variable, the objective and each constraint means.

   (b) What is the dual of the above LP? What is an interpretation of the LP – explain in sentences, as was done in class for the Chicago vs. Detroit Pizza example? In particular, explain what each variable, the objective and each constraint means.

   Answer:

   (a) Let P denote the set of all s-t paths in the given graph. LP for the max flow is

   $$\max \textstyle\sum_{p\in P} x_p \text{ subject to } \sum_{p:e\in P} x_p \leq c_e, \text{ for all e} \in E, \text{ and } x \geq 0.$$

   We have a variable x_p for each s-t path in P. Consider all the paths through an edge e, the sum of variables for each of these paths should be less than the capacity of the edge since we are packing s-t paths subject to edge cost constraints. The objective is to maximize the number of paths we pack in total.

   (b) Dual of the LP is

   $$\min \textstyle\sum_{e\in E} c_e y_e \text{ subject to } \sum_{e\in p} y_e \geq 1 \text{ for all } p \in P, \text{ and } y \geq 0$$

   We have a variable for each edge y_e. We are covering all s-t paths in the given graph such that we pick at least one (total) edge from each path. The objective is to minimize the edges (fractional) we need to pick to cover all the paths.

4. **(*) 2nd Bonus Problem.** (30 points) Jane has been employed to repair a 2-lane road. Specifically, the 2-lane road is modeled as a 2 dimensional array of length n and width 2. Jane is given the location of the patches that are damaged (see the example below). Jane's objective is to repair the road in minimum cost. She has to work with the following constraints:

   - She can only cover rectangular patches i.e. she cannot have L shaped patches.
   - She needs to cover all the bad parts.
   - She can only use P rectangular patches in total.

   Her cost is directly proportional to the area she covers. Give an efficient algorithm for Jane to repair the road. You need to give the optimal area and the optimum repair patches.

   In the example below, the optimal way of repairing the road using P = 2 patches (green-filled area) is given as under. The optimal area is 8 units.

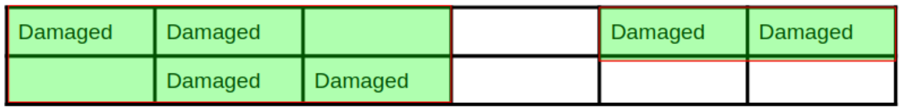

Figure 1: Optimal road patch for P = 2

Answer: Let $OPT(m,n,p)$ be the optimal area for covering m cells of the first row and n cells of the bottom row using p patches.
Let the first row be denoted by M and second row with N.
(Note: Arrays are assumed to be 1-indexed).
The recurrence relation is given by:

$$OPT(m,n,p) = \left\{ \begin{array}{lr} 0 & \text{If } p=0 \text{ and no damaged parts remain} \\ \infty & \text{If } p=0 \text{ and damaged parts remain} \\ 0 & \text{If } m=n=0 \\ \min\{b\} & \text{If } m=0 \\ \min\{a\} & \text{If } n=0 \\ min\{a,b,c\} & \text{Otherwise} \end{array} \right\}$$

$$a = \left\{ \begin{array}{lr} \min_{1\le k\le m}\{OPT(m-k,n,p-1)+C_{p_M}\} & \text{If } \; M[m] = Damaged \\ OPT(m-1,n,p) & \text{Otherwise} \end{array} \right\}$$

$$b = \left\{ \begin{array}{lr} \min_{1\le k\le n}\{OPT(m,n-k,p-1)+C_{p_N}\} & \text{If } \; N[n] = Damaged \\ OPT(m,n-1,p) & \text{Otherwise} \end{array} \right\}$$

$$c = \left\{ \begin{array}{lr} \infty & \text{If } \; m \neq n \\ OPT(n-1,n-1,p) & \text{If } \; M[m] \neq Damaged \text{ and } N[n] \neq Damaged \\ \min_{1\le k\le n}\{OPT(m-k,n-k,p-1)+C_{P_{MN}}\} & \text{Otherwise} \end{array} \right\}$$

where C_{p_M} gives the area of a patch that extends only in either row M covering all damaged patches from m till k. We can similarly define C_{p_N}. $C_{p_{MN}}$ gives the area of a patch that extends in both arrays M and N covering all damaged patches from $m=n$ till k (i.e. in this case the rectangles are allowed to have width 2).

The quantity a includes the case when the last cell of the first row i.e. $M[m]$ is damaged and the last cell of the second row i.e. $N[n]$ is not damaged. In this case a patch must cover $M[m]$. By iterating over all possible lengths of the patches in the first row, we find the optimal choice. The case when $M[m]$ is not damaged is straight forward i.e. we can ignore $M[m]$. Similar interpretation holds for quantity b. a and b combined handle the cases of rectangular patches of width 1.

c considers the case of rectangular patches of width 2. If $m \neq n$, then we cannot have a rectangular patch of width 2 covering both $M[m]$ and $N[n]$ (the patch should not cover anything beyond positions m and n in rows M and N respectively). Next when both $M[m]$ and $N[n]$ are not damaged, we can ignore the current column. In the last case, when either or both of $M[m]$ and $N[n]$ are damaged, we find the best rectangular patch by iterating over all possible lengths.

Therefore, we have covered all the cases. The base cases are right. If we assume that the smaller subproblems are calculated correctly (induction hypothesis), then by the arguments above (inductive step) our algorithm is correct. The time complexity is $O(n^3P)$ and space complexity is $O(n^2P)$.