Algorithms: CSE 202 — Homework 2 Solutions

Problem 1: Nesting Boxes (CLRS)

A d-dimensional box with dimensions $(x_1, x_2, \ldots, x_d)$ *nests* within another box with dimensions $(y_1, y_2, \ldots, y_d)$ if there exists a permutation π on $\{1, 2, \ldots, d\}$ such that $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \ldots, x_{\pi(d)} < y_d$.

1. Argue that the nesting relation is transitive.
2. Describe an efficient method to determine whether or not one d-dimensional box nests inside another.
3. Suppose that you are given a set of n d-dimensional boxes $\{B_1, B_2, \ldots, B_n\}$. Describe an efficient algorithm to determine the longest sequence $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k} \rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \ldots, k-1$. Express the running time of your algorithm in terms of n and d.

Solution: Nesting Boxes

1. Let A, B and C be d-dimensional boxes with dimensions $(a_, \ldots, a_d)$, $(b_1, \ldots, b_d)$ and $(c_1, \ldots, c_d)$ respectively. We use the same symbol to denote a box as well as its dimensions. To prove that the nesting is transitive, we need to show that if A can be nested in B and B can be nested in C, then A can be nested in C. Let π_A be a permutation that nests A in B, that is, $a_{\pi_A(1)} < b_1, \ldots, a_{\pi_A(d)} < b_d$. Let π_B be a permutation that nests B in C, that is, $b_{\pi_B(1)} < c_1, \ldots, b_{\pi_B(d)} < c_d$. Let $\pi'_A = \pi_A \circ \pi_B$. In other words, π' is the permutation obtained by applying π_B followed by π_A. We argue that π' lets us nest A inside C. To see that $a_{\pi'(i)} < c_i$ for $1 \leq i \leq d$, we observe $b_{\pi_B(i)} < c_i$ and $a_{\pi_A(\pi_B(i))} < b_{\pi_B(i)}$.

2. **Algorithm description:** Let A be the d-dimensional box which is to be nested within another d-dimensional box B. Sort the dimensions of A and B. If the length of every dimension in the sorted list of A is smaller than the length of the corresponding dimension in the sorted list of B, then A can be nested within B.

Correctness proof:

Claim 1. *If the sorted permutation of both A and B does not satisfy the nesting requirement for each dimension, no other permutation can satisfy the sorting requirement.*

Proof. Let $(a_1, \ldots, a_d)$ be the dimensions of the box A such that $a_1 \leq \cdots \leq a_d$. Let $(b_1, \ldots, b_d)$ be the dimensions of the box B such that $b_1 \leq \cdots \leq b_d$. If $a_i > b_i$ for some $1 \leq i \leq d$, then $a_j > b_k$ for all $i \leq j \leq d$ and $k \leq i$. Hence any permutation that satisfies the nesting requirement must map the $d - i + 1$ elements $a_i, \ldots, a_d$ bijectively to the $d - i$ elements $b_{i+1}, \ldots, b_d$, which is not possible □

Time complexity: Sorting the d dimensions takes $\Theta(d \log d)$. An additional d comparisons are made after sorting. Hence overall time complexity is $\Theta(d \log d)$.

3. **Algorithm description:** We represent the problem as that of finding the longest path in a graph G. Each box is represented by a vertex in G. In addition, G contains the vertices S and T. We use B_i to represent the box and the corresponding vertex. The edges of G are determined as follows.

For every pair of vertices B_i and B_j, join vertex B_i to vertex B_j with a directed edge if box B_i nests in box B_j. There is a directed edge from S to each of the other vertices. Every vertex $u \neq T$ is connected to T by a

directed edge from u to T. The graph G is a directed acyclic graph since a box cannot be nested inside itself. The longest path from S to T will correspond to the longest sequence of nesting boxes from $\{B_1, B_2, \ldots, B_n\}$.

We define the following subproblems for each vertex v in G.

- Let $D(v)$ be the length of the longest path that terminates at vertex v.
- Let $P(v)$ be the vertex immediately preceding v in the longest path that terminates at v.

We use the following recursive formulation to compute the functions $D(.)$ and $P(.)$.

For the unique source vertex S, we define $D(S) = 0$ and $P(S) =$ **null**. For all other nodes v, we define $D(v) = \max_{u \text{ is an immediate predecessor of } v \text{ in } G} D(u) + 1$ and $P(v) = u$, the predecessor vertex for which the maximum is achieved in the recursive definition of $D(v)$.

The recursive formulation is well-defined as long as we can compute the functions $D(v)$ and $P(v)$ for each v after the predecessors of v have been computed. Since the vertices can be topologically ordered, we conclude that the recursive formulation is well-defined.

We prove that $D(v)$ is the length of the longest path that terminates at v by induction on the index of v in the topological order. Since S is the unique source vertex, it is the first vertex in the topological order. The length of the longest path that terminates at S is zero. Indeed $D(S) = 0$ which establishes the correctness for the base case.

Let v be any vertex other than the source vertex. Assume that $D(u)$ is indeed the length of the longest path for all vertices u preceding v in the topological order. We will prove that $D(v)$ is the length of the longest path that terminates at v. Consider the longest path L that terminates at v. Since v is not a source vertex, there must be at least one predecessor vertex for v. Let u be the vertex that immediately precedes v in L. By inductional hypothesis, $D(u)$ is the length of the longest path that terminates at u so the length of L is $1 + D(u)$. Since $D(v)$ is the maximum of $1 + D(u')$ over all immediate predecessors u' of v, we conclude that $D(v)$ is indeeed the length of the longest path that terminates at v.

$D(T)$ is the length of the longest path in G. We can trace back the path from T to S to get the longest path in G from S to D.

Pseudocode :

```
Construct graph G = (V, E)
for each u in V do
    dist[u] ← −∞
    prev[u] ← nil
end for

dist[S] ← 0
Topologically sort V

for each v in V do
    for each u which is a predecessor of u do
        if D[v] < D[u] + 1 then
            D[v] = D[u] + 1
            P[v] = u
        end if
    end for
end for
current = P[T]
while current ≠ S do
    print current
    current ← P[current]
end while
```

Time Complexity: The time for sorting the dimensions for each of the n nodes is $\Theta(nd \log d)$. The time for constructing the graph is $\Theta(n^2 d)$ since it requires n^2 comparison of d dimensions each. The algorithm to

find the longest path takes $\Theta(|V| + |E|)$. Since there are a total of n nodes and $\Theta(n^2)$ edges in G, finding the longest path takes $\Theta(n^2)$. Hence the overall complexity is $\Theta(n^2 d + nd \log d)$.

Problem 2: Business plan

Consider the following problem. You are designing the business plan for a start-up company. You have identified n possible projects for your company, and for, $1 \leq i \leq n$, let $c_i > 0$ be the minimum capital required to start the project i and $p_i > 0$ be the profit after the project is completed. You also know your initial capital $C_0 > 0$. You want to perform at most k, $1 \leq k \leq n$, projects before the IPO and want to maximize your total capital at the IPO. Your company cannot perform the same project twice.

In other words, you want to pick a list of up to k distinct projects, $i_1, \ldots, i_{k'}$ with $k' \leq k$. Your *accumulated capital* after completing the project i_j will be $C_j = C_0 + \sum_{h=1}^{h=j} p_{i_h}$. The sequence must satisfy the constraint that you have sufficient capital to start the project i_{j+1} after completing the first j projects, i.e., $C_j \geq c_{i_{j+1}}$ for each $j = 0, \ldots, k' - 1$. You want to maximize the final amount of capital, $C_{k'}$.

Solution: Business plan

Algorithm: We select, start and complete projects in a sequence. The next project is selected after the previous project has been completed. Assume the projects $s_1, s_2, \ldots, s_j$ have already been selected, started and completed. Let C_j denote the available capital after the project s_j has been completed and before the next project is selected. A project l is *eligible* at time j if it has not been selected so far and its capital requirement $c_l \leq C_j$. If $j \leq k - 1$, select the next project using the following greedy strategy. Among all eligible projects, select the project with the maximum profit. If there is no such project, stop the selection process.

We maintain a heap of all eligible projects ordered by profit so that the project with the maximum profit is at the top of the heap. After completing a project, if the heap is not empty, we select an eligible project with the maximum profit, remove it from the heap and continue. Observe that C_j is increasing with j as profits are nonnegative. As projects get completed, additional projects may become eligible in which case we insert them into the heap. A project never leaves the heap unless it is selected for execution. The heap can be managed with $O(n \lg n)$ time.

Let S^g be the sequence of projects selected by the greedy strategy. We show that S^g is an optimal sequence of projects. We use induction on the number of projects to prove the correctness. In particular, we show that replacing any other choice with the greedy choice in the sequence will produce at least as a good a solution as the optimal solution.

For a feasible sequence of projects $T = t_1, \ldots, t_l$, let value$(T) = C_0 + p_{t_1} + \cdots + p_{t_l}$ denote the capital after all the projects in the sequence have been completed. For an empty sequence T, we define its value as value$(T) = C_0$.

Claim 2. *The greedy strategy produces an optimal solution on every instance.*

Proof. We obtain a contradiction by assuming that there is an instance on which the greedy strategy fails to produce an optimal solution.

Let n be the smallest integer such that there exists an instance of size n for which the greedy strategy does not produce an optimal solution. We argue that $n \geq 2$. If $n = 1$ there is only one feasible solution. If the initial capital is large enough, we select and execute the project. Otherwise, no project is selected. Hence the greedy solution is optimal if there is only one project.

Let I be an instance of size n such that the greedy algorithm fails to produce an optimal solution on I. We are given $k, C_0, c_1, \ldots, c_n, p_1, \ldots, p_n$. Let $S^g = s_1, s_2, \ldots, s_{k'}$ be the greedy solution for I for some $0 \leq k' \leq k$ where s_i is the i-th project selected by the algorithm. If $k' = 0$, S^g is an empty sequence. Observe that the sequence S^g satisfies the following property: for all $1 \leq j < l$, s_{j+1} is the project with the largest profit among all eligible projects at the completion of the first j projects in S^g.

Let $S_1^g = s_2, \ldots, s_{k'}$.

Let $T = t_1, \ldots, t_{k''}$ be an optimal sequence of projects. Without loss of generality, assume that the projects in T are ordered so that project t_{j+1} is the project with the maximum profit among all projects which are eligible at the time the first j projects in the sequence T are completed. Let $T_1 = t_2, \ldots, t_{k''}$.

Case I — $s_1 = t_1$**:** Consider the instance I' of size $n-1$ obtained from I by deleting the project s_1 and setting its initial capital to $C_1 = C_0 + p_{s_1}$ and the number of projects to be performed to $k-1$. Observe that the greedy strategy produces the solution S_1^g on I' (explain why), which by assumption, is an optimal solution. Also, we have that T_1 is a feasible solution for I' and value(T) = value(T_1) (explain why). We get

$$\begin{aligned} \text{value}(S^g) &= C_0 + p_{s_1} + p_{s_2} + \cdots + p_{s'_k} \\ &= \text{value}(S_1^g) \\ &\geq \text{value}(T_1) \\ &= \text{value}(T) \end{aligned}$$

which shows that S^g is indeed an optimal solution which leads to a contradiction.

Case II — $s_1 \neq t_1$**:**

Let $T' = s_1, t_2, \ldots, t_{k''}$ if s_1 is not in the sequence T. Otherwise, let T' be the sequence obtained from T by swapping t_1 with s_1. In either case, T' is a feasible solution for I (explain why). Moreover value(T') $\geq$ value(T) since $p_{s_1} \geq p_{t_1}$. This implies that T' is an optimal solution.

Now we are in the situation where the first choices of S^g and the optimal solution T' coincide which leads to a contradiction to the fact that S^g is the smallest instance on which the greedy strategy fails.

□

Complexity; The algorithms runs in time $O(n \lg n)$ since the total time required to manage the heap is $O(n \lg n)$.

Problem 3: Worker partnerships

Consider the following problem: After millions of years of warfare between the continents of Ur and Nena, the proletariat of the continents decided that it is best to work together to advance the civilization. Each continent has selected a set of n workers to team with the workers from the other continent. Each team will consist of two workers, one from each continent, so they can work together on a project for the benefit of both the continents. However, it is not an easy matter for workers from different continents to work together productively. Each worker has a degree of adaptability to work with a worker from the other continent. Let p_i for $1 \leq i \leq n$ be the degree of adaptability of the i-th worker from the continent Ur. Let q_i for $1 \leq i \leq n$ be the degree of adaptability for the i-th worker from the continent Nena. If worker i from Ur and worker j from Nena formed a pair, then the productivity of the pair is $p_i q_j$, the product of their degrees of adaptability. Our goal is to form as many pairs of workers as possible where one worker is chosen from each set so that for each pair the product of their degrees of adaptability is at least x.

Each worker can only be in at most one team. Each team has exactly two workers, one from each continent. We want to maximize the number of teams.

Solution: Matching workers

1 High Level Description

Let $P = p_1, p_2, \ldots, p_n$ be the list of degrees of adaptability of workers $1, 2, \ldots, n$ from Ur. Let $Q = q_1, q_2, \ldots, q_n$ be the list of degrees of adaptability of workers $1, 2, \ldots, n$ from Nina.

Sort the list P in non-decreasing order to get $P' = p'_1, p'_2, \ldots, p'_n$. Sort the list Q in non-increasing order to get $Q' = q'_1, q'_2, \ldots, q'_n$. Let p be the first elements in P' and be the first element in Q'. If the pq is greater

than or equal to x, assign the corresponding workers to a team and move to the next item in each list. If the product is less than x, move to the next item in P'. When one of the lists is empty, terminate the algorithm and return the assigned teams. This is the maximum number of teams possible.

2 Correctness

Claim 1: The greedy strategy produces an optimal solution for every instance.

Proof; Since the number of workers in each country is the same, we define the size of an instance to be the number of workers. We obtain a contradiction by assuming that there is an instance on which the greedy strategy fails to produce an optimal solution. Let n be the smallest integer such that there exists an instance of size n for which the greedy strategy does not produce an optimal solution. We argue that $n \geq 2$. If $n = 1$ there is only one possible solution. If the product of the degrees of adaptabilities of the workers from each country is greater than or equal to the threshold x, we assign them to a team. Otherwise, no team is formed.

Let I be an instance of size $n \geq 2$ such that the greedy algorithm fails to produce an optimal solution on I. Let the optimal solution to the problem be $O = (p_1^o, q_1^o), \ldots, (p_k^o, q_k^o)$. Let the solution given by the greedy algorithm be $G = (p_1^g, q_1^g), \ldots, (p_{k'}^g, q_{k'}^g)$ and $k > k^{'}$.

Without loss of generality, assume that the G and O are ordered in non-increasing order based on the degree of adaptability of workers from Nina . Observe that each team (p_i^g, q_i^g) in G satisfies the following property: q_i^g is paired with a worker from Ur such that the product of the degrees of adaptabilities is the minimum among all products of degrees of adaptabilities involving q_i^g and a worker from Ur where the product is greater than or equal to the threshold x.

2.1 Case 1

$q_1^o = q_1^g$:

2.1.1 Case 1a

$p_1^o = p_1^g$: Consider the instance I' of size $n - 1$ that is obtained by deleting the workers corresponding to q_1^o and p_1^o from I. Let $G^{'} = (p_2^g, q_2^g), \ldots, (p_{k'}^g, q_{k'}^g)$ and $O^{'} = (p_2^o, q_2^o), \ldots, (p_k^o, q_k^o)$. Notice that G' is indeed the greedy solution for the instance I'. Also observe that O' is a feasible solution to I' since all the workers involved in O' are part of the instance I' and the product of degrees of adaptabilities of each pair is greater or equal to the threshold x. By induction, G' is optimal from which we get $k = k'$ which in turn shows that G is an optimal solution to I.

2.1.2 Case 1b

$p_1^o < p_1^g$: This case cannot happen since p_1^g is the minimum among the degrees of adaptabilities of workers in Ur who can be paired with q_1^g to form a team.

2.1.3 Case 1c

$p_1^o > p_1^g$: In this case, we modify the optimal solution O to get O'. There are two cases. If p_1^g does not appear in O, we replace p_1^o with p_1^g in the first pair of O. It is easy of see that O' is feasible and it has the same size as O. If p_1^g appears elsewhere in O, we swap p_1^o with p_1^g. The resulting O' has the same size as that of O. O' is also feasible since $p_1^o > p_1^g$. Now this case is reduced to Case 1a.

2.2 Case 2a

$q_1^o \neq q_1^g$

2.2.1 Case 2a

$q_1^o > q_1^g$: Since the lists G and O are ordered in descending order of the degrees of adaptability of workers from Nina, if q_1^g appears in the first team in G, then we can conclude that there is no compatible worker from Ur for q_1^o. This is a contradiction as the optimal algorithm claims to have found one such worker.

2.2.2 Case 2b

$q_1^o < q_1^g$: This means that q_1^g does not appear in O. We can replace q_1^o with q_1^g in the optimal solution which reduces this case to Case 1.

All these cases together imply that that the greedy algorithm works correctly for instances of size n. Thus we have no counterexamples for the correctness of the greedy algorithm.

Problem 4: Shortest wireless path sequence (KT 6.14)

A large collection of mobile wireless devices can naturally form a network in which the devices are the nodes, and two devices x and y are connected by an edge if they are able to directly communicate with each other (e.g., by a short-range radio link). Such a network of wireless devices is a highly dynamic object, in which edges can appear and disappear over time as the devices move around. For instance, an edge (x, y) might disappear as x and y move far apart from each other and lose the ability to communicate directly.

In a network that changes over time, it is natural to look for efficient ways of *maintaining* a path between certain designated nodes. There are two opposing concerns in maintaining such a path: we want paths that are short, but we also do not want to have to change the path frequently as the network structure changes. (That is, we'd like a single path to continue working, if possible, even as the network gains and loses edges.) Here is a way we might model this problem.

Suppose we have a set of mobile nodes V, and at a particular point in time there is a set E_0 of edges among these nodes. As the nodes move, the set of edges changes from E_0 to E_1, then to E_2, then to E_3, and so on, to an edge set E_b. For $i = 0, 1, 2, \ldots, b$, let G_i denote the graph (V, E_i). So if we were to watch the structure of the network on the nodes V as a "time lapse", it would look precisely like the sequence of graphs $G_0, G_1, G_2, \ldots, G_{b-1}, G_b$. We will assume that each of these graphs G_i is connected.

Now consider two particular nodes $s, t \in V$. For an s-t path P in one of the graphs G_i, we define the *length* of P to be simply the number of edges in P, and we denote this $\ell(P)$. Our goal is to produce a sequence of paths $P_0, P_1, \ldots, P_b$ so that for each i, P_i is an s-t path in G_i. We want the paths to be relatively short. We also do not want there to be too many *changes*–points at which the identity of the path switches. Formally, we define $changes(P_0, P_1, \ldots, P_b)$ to be the number of indices i $(0 \leq i \leq b-1)$ for which $P_i \neq P_{i+1}$.

Fix a constant $K > 0$. We define the cost of the sequence of paths $P_0, P_1, \ldots, P_b$ to be

$$cost(P_0, P_1, \ldots, P_b) = \sum_{i=0}^{b} \ell(P_i) + K \cdot changes(P_0, P_1, \ldots, P_b).$$

1. Suppose it is possible to choose a single path P that is an s-t path in each of the graphs $G_0, G_1, \ldots, G_b$. Give a polynomial-time algorithm to find the shortest such path.

2. Give a polynomial-time algorithm to find a sequence of paths $P_0, P_1, \ldots, P_b$ of minimum cost, where P_i is an s-t path in G_i for $i = 0, 1, \ldots, b$.

Solution: Shortest wireless path sequence (KT 6.14)

a. Let $G_i = (V, E_i), i = 0, \ldots, b$, be a sequence of graphs and let $s, t \in V$. If there is a single path P from s to t in each of the G_i, then to find a shortest one we can perform breadth first search (BFS) on $G = (V, \bigcap_{i=0}^{b} E_i)$. Computing G can be done in time $O((|V| + \sum |E_i|)$ and BFS takes time $O(|E|+|V|)$ where $|E| = \bigcap_{i=0}^{b} E_i$. Hence the total runtime is bounded by $O((|V| + \sum |E_i|) = O(b \cdot n^2)$

b. To find a min cost sequence $P_0, \ldots, P_b$, notice that in any optimal sequence, if i is the *last breakpoint*, i.e., the last index where the path changes, then P_b (with no further breakpoints) is the shortest path for the graph sequence $G_i, \ldots, G_b$; and $P_0, \ldots, P_{i-1}$ is the optimal sequence of paths for $G_0, \ldots, G_{i-1}$ (with potentially additional breakpoints). We turn this into an algorithm below.

Let $D_{i,j}$ be the length of a shortest path from s to t in $(V, \bigcap_{k=i}^{j} E_k)$. We can compute $D_{i,j}$ for each i, j as follows.

for $i \leftarrow 0, \ldots, b$ **do**
  $E' \leftarrow E_i$ // E' will store $\cap_{k=i}^{j} E_k$
  $D_{i,i} \leftarrow$ BFS distance from s to t in (V, E')
  for $j \leftarrow i+1, \ldots, b$ **do**
    $E'' \leftarrow \varnothing$
    for each $e \in E_j$ **do**
      if $e \in E'$, add e to E''
    end for
    $E' \leftarrow E''$
    $D_{i,j} \leftarrow$ BFS distance from s to t in (V, E')
  end for
end for

This takes time $O(b^2 \sum(|E_i| + |V|)) = O(b^3 n^2)$.

Next, for $0 \leq i \leq b$, let C_i be the cost of an optimal sequence for $G_0, \ldots, G_i$. Let B_i be the index position of the latest breakpoint. The cost of the optimal sequence of paths C_j is then either the cost of the of shortest path with no breakpoints $D_{0,j}$ or the cost of the shortest sequence of paths up to index B_j, followed by a path with no more breakpoints.

More formally, for $j \in [0, b]$,

$$C_j = \min_{i \in [0,j]} C_{i-1} + (j - i + 1) D_{i,j} + K.$$

$$B_j = \arg\min_{i \in [0,j]} C_{i-1} + (j - i + 1) D_{i,j} + K.$$

In the base case, $C_0 = D_{0,0}$ and $B_0 = 0$. We define $C_{-1} = -K$ such that $C_j = j * D_{0,j}$ if $B_j = 0$, i.e. in the case that the optimal sequence of paths does not contain a breakpoint.

Correctness follows immediately from our assertion that the optimal sequence of paths either contains no breakpoints, in which case we pick the shortest path in the intersection of the graphs, or there must be a last breakpoint B_j. Then the cost of the shortest sequence of paths consists of the cost of the shortest sequence of paths from indices 0 to B_{j-1}, plus the cost of the shortest path from with no breakpoints from indices B_j to j times the number of indices in that range, plus the cost of one breakpoint.

Finally, the total runtime consists of computing first all values $D_{i,j}$ in time $O(b^3 n^2)$ and then computing all values C_j. Since C_j is computed from $C_0, \ldots, C_{j-1}$ in time $O(b)$, the total time of the algorithm is dominated by the time to compute all values $D_{i,j}$, i.e. $O(b^3 n^2)$.