James V. Jahan — Homework 2 — CSE 202: BOO

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.

Problem 1.1:

Assume there exist 3 d-dimensional boxes

$x = (x_1, x_2, \ldots, x_d)$

$y = (y_1, y_2, \ldots, y_d)$

$z = (z_1, z_2, \ldots, z_d)$

Where x nests in y and y nests in z. For this to be the case, there needs to exist a permutation for both x and y.

- Denote the permutation on x as π
- Denote the permutation on y as γ

According to the nesting definitions then

$x_{\pi(1)} < y_1, \; x_{\pi(2)} < y_2 \ldots x_{\pi(d)} < y_d$

$y_{\gamma(1)} < z_1, \; y_{\gamma(2)} < z_2 \ldots y_{\gamma(d)} < z_d$

The existence of π and γ then means that the following holds

$x_{\pi(\gamma(1))} < z_1$, $x_{\pi(\gamma(2))} < z_2$, ..., $x_{\pi(\gamma(d))} < z_d$

- Conceptually this can be abstracted as if x "fits" in y and y "fits" in z, after some dimension reorientation for both x and y, we can perform both dimension reorientations on x to "fit" in z.

Thus if x nests in y, y nests in z, by the existence of the permutations π and γ respectively, then x also nests in z by chaining these permutations. This can also be Conceptualized as the existence of a permutation on x that necessitates a nesting in z, given x nests in y, and y nests in z.

Problem 1.2:

Definitions and Problem Formulation:

Define 2 d-dimensional boxes

$x = (x_1, x_2, \ldots, x_d)$

$y = (y_1, y_2, \ldots, y_d)$

Determine if x can nest in y, or y can nest in x.

Nesting box - The box which can nest another box.

Nested box - The box for which $\exists \pi$, s.t it nests in another box.

Algorithm:

Sort x and y by their d dimensions in non-increasing order (i.e., $sorted(x) = x'$ where $x'_1 \geq x'_2 \geq x'_3 \ldots \geq x'_d$). For the assessment of a nesting relationship evaluate if x'_1 (i.e., $sorted(x)_1$) > y'_1). If this is the case then we are checking whether y can nest in x (y'_1 is smaller than x'_1). If $x'_1 < y'_1$, check if x can nest in y. If $x'_1 == y'_1$, conclude no nesting exists between x and y, as the problem denotes nesting as a strict $<$ for all dimensions.

For the postulated nested box (i.e, either x' or y' according to the largest element check above) evaluate whether every element in the sorted postulated nested box is $<$ its corresponding element in the sorted postulated nesting box. If so then return TRUE, one d-dimensional box nests in the other. Otherwise return FALSE, no such valid nesting exists for the given pair of boxes. Mathematically this can be represented as:

if Sorted postulated nested box dimension $i <$ Sorted postulated nesting box dimension i

$1 \leq i \leq d$

Return TRUE

else

Return FALSE

Proof of Correctness:

For a box to nest in another, there must exist a permutation π s.t. each dimension of the nesting box $> \pi($ each dimension of the nested box), as per the problem definition. We prove here that the above algorithm returns the existence of a π such that one box can nest in one another.

Base Case: $d=1$

In the case of $d=1$ a nesting relationship exists if $x_1 < y_1$ or $y_1 < x_1$. The above algorithm accounts for this case as the sorting of a single element is equivalent to the element.

Inductive Hypothesis: For $i>1$, assume $\forall n \leq i$, for $n < d$, the algorithm outputs the existence of a nesting relationship for the first i-dimensions of the 2 boxes.

Inductive Step:

Observe that if the inductive hypothesis holds then $x'_{i+1} \leq x'_i$ and $y'_{i+1} \leq y'_i$, based on the sorting employed by the algorithm. Thus there cannot exist a $y_{i+1} > y'_i \rightarrow y'_i$ that invalidates the previous comparisons and relationship. Thus the existence of a nested box at $i+1$ is dependent on the solution at i and whether $y'_{i+1} < x'_{i+1}$ or $x'_{i+1} < y'_{i+1}$ as identified by the algorithm.

Time Complexity:

This algorithm requires sorting 2 d-dimensional arrays (one for each box) giving a time complexity of $2 \cdot O(d \log d)$. Comparisons of each element as specified by the algorithm take $O(d)$. The limiting factor of this approach then is the sorting giving the overall algorithm a run time of $O(d \log d)$.

Problem 1.3:

Definitions and Problem Formulation:

Given n-dimensional boxes, identify the longest sequence of nesting boxes.

Key Idea:

Boxes can be represented as vertices, and the existence of a nesting relationship can be represented as a directed edge. Given the construction of this graph, the problem translates to finding the longest path in a directed acyclic graph (DAG).

Algorithm:

Sort each n d-dimensional boxes in the same manner as the algorithm given in Problem 1.2 (i.e, non-increasing order). Then create a graph with the vertices, V, equivalent to the boxes B (i.e, $V = \{V_1 = B_1, V_2 = B_2, \ldots, V_n = B_n\}$). Note there are no edges in this graph currently. After this for all pairs of n-boxes evaluate whether a nesting relationship exists. If one does exist, create a directed edge from the nested box to the nesting box. These edges all will have a weight of 1. Now given the construction of this graph G, we need to find the longest path, which requires a topological sorting.

Topologically sort G by selecting all vertices of in-degree = 0 at each point needed in the topological sorting.

→ This assumes topological sorting through the set, stack method, where an element is only added to the stack after its children have all been explored, or a vertex has no children.

In the topological sorting, assign each of the vertices in G with an in-degree = 0 with a value of 1. All other vertices are assigned a value of $-\infty$. Proceeding with this topological ordering we can employ dynamic programming to set the longest path, and thus the longest sequence of nesting boxes. Specifically, iterate through the vertices in the order established by the topological sorting populating the value of a vertex v with

$$DP[v] = \max(DP[v], DP[u] + 1)$$

where u is the vertex which has a directed edge into v. After iterating through each vertex v in the topological sorting, and updating DP[v] for every adjacent vertex v from u, the longest sequences of boxes is the max of DP. The sequence of boxes can be obtained by also storing the incoming edge that yielded DP[v] and then backtracking through DP.

Proof of Correctness:

First note that G is a DAG. This can be shown through the transitive property in Problem 1.1. If a cycle were to exist in G, this would mean for some set of nesting boxes: B_i nests B_j, nests B_k nests ... nests B_p, there exists a backwards nesting (e.g. B_k nests in B_i). However this creates a contradiction in violation of the transitive relationship of nesting. Thus G must be a DAG.

In Problem 1.2 we showed the correctness of nesting relationships, so that leaves proving that the dynamic programming part of this algorithm does indeed yield the longest sequence of nesting boxes:

Base Case: $n = 1$

In this case, B_1 has an in-degree = 0, is the first element in the topological sorting and has the maximum value $DP[B_1] = 1$. Thus our algorithm returns the desired output of 1, $\langle B_1 \rangle$.

Note this can also be generalized to n-boxes w/ no nesting relationships, as all will have a DP value = 1 and any such selection is the longest nested sequence.

Inductive Hypothesis: For $i > 1$, $\forall\ n \leq i < n$, assume the above algorithm determines the largest sequence of nesting boxes.

Inductive Step:

Proceeding from the inductive hypothesis (IH), proceed to i+1. As our algorithm topologically sorted G, we know that V_{i+1} can only have 3 cases:

Case1: V_{i+1} is a node of in-degree 0 (as would occur when re-starting selection during topological sort).
→ In this case $DP[V_{i+1}] = 1$ and is equivalent or < the best sequence obtained at i.

Case2: V_{i+1} results in an equivalent maximum nesting box sequence, in which case the solution at step i is still valid. However the algorithm will catch this as a valid maximum as well.

Case3: V_{i+1} adds to the longest nesting box sequence. In this case we know that the order we proceed through G and update DP, that $DP[V_{i+1}] = \max(DP[V_i] + 1)$ which is the case. We cannot increase the value at i+1 by more than 1 as all edges are weighted = 1.

Thus given the construction of G, the topological sorting, & incrementing DP according to the maximal number of boxes it is capable of nesting, the maximum of the DP object is indeed the longest sequences of nesting boxes $\{B_{i_1}, B_{i_2}, \ldots, B_{i_k}\}$

Time Complexity:

This algorithm contains multiple steps with the following time complexities:

1) Sorting n d-dimensional boxes $\rightarrow O(nd \log d)$
2) Creating the graph by comparing all pairs of boxes of d-dimension $\rightarrow O(n^2 d)$
3) Creating a topological ordering $\rightarrow O(n)$
4) Iterating through the topological ordering to populate DP $\rightarrow O(n^2)$.

Based on these 4 key components, the overall time complexity for this algorithm is $O(nd \cdot \max(n, \log d))$.

Problem 2: Classes and rooms

You are given a list of classes C and a list of classrooms R. Each class c has a positive enrollment $E(c)$ and each room r has a positive integer capacity $S(r)$. You want to assign each class a room in a way that minimizes the total sizes (capacities) of rooms used. However, the capacity of the room assigned to a class must be at least the enrollment of the class. You cannot assign two classes to the same room. Design an efficient algorithm for assigning classes to rooms and prove the correctness of your algorithm.

Key Idea:

We can never fit a class in a room with a capacity < Class enrollment. Thus all rooms with a capacity < the class with the lowest enrollment can be omitted from consideration. This, coupled with the minimization goal + the constraint that no two classes can share the same room sets up for a Greedy Algorithmic approach.

Definitions:

$C_s = \{c_1, c_2, \ldots, c_n\}$ where $c_1 \leq c_2 \leq c_3 \leq \ldots \leq c_n$

$R_s = \{r_1, r_2, \ldots, r_n\}$ where $S(r_1) \leq S(r_2) \leq \ldots \leq S(r_m)$

Both C_s and R_s are sorted classes and rooms by enrollment and capacity respectively.

GS = Greedy solution employed by algorithm

OS = Proposed optimal solution used for comparison in proof that is assumed to perform better than GS

Algorithm:

Sort classes and rooms as outlined above. Given C_s and R_s are sorted in non-decreasing order by enrollment and capacity respectively, iterate through C_s from 1 to n and employ the following match procedure w/ R_s. If for any C_i $S(r_j)$ is $< C_i$, remove r_j from R_s, which makes r_{j+1} the new first element of R_s. Continue this removal procedure until the first element in R_s has an enrollment $\geq C_i$. At this point, assign class i to this room + remove both C_i from C_s and the first room in R_s (i.e., the room C_i matched w/). If desired update the total enrollment-capacity difference as the difference between the capacity and enrollment for the given assignment. Continue this procedure until either all classes have been assigned to a room, or there are no rooms left in R_s, in which case return an error stating that a valid assignment of all classes to rooms is not possible.

Time Complexity:

This algorithm contains multiple steps with the following time complexities:

1) Sorting rooms by capacity $\Rightarrow O(n \log n)$
2) Sorting classes by enrollment $\Rightarrow O(n \log n)$
3) Class-room assignment $\Rightarrow O(n)$

Class-room assignment is $O(n)$ as either need to remove $m-n$ rooms and match n rooms. If $m > n$, then this amounts to $m-n+n$ operations $= m$. If $m \leq n$ then there can be at most m operations.

Given these complexities, the overall time complexity amounts to the time-limiting step of sorting giving:

$$O(\max(m \log m, n \log n))$$

Proof of Correctness:

Let $GS = \{g_1, g_2, \ldots, g_k\}$ be the output of the above algorithm, where $k \leq n$ depending on if a valid assignment exists for all classes. Denote that each g_i corresponds to the room to which a class corresponds in C_s (i.e., the assigned rooms correspond to the mapping of the sorted classes by enrollment.

Let $OS = \{o_1, o_2, \ldots, o_n\}$ be the output of the optimal solution. Denote that each o_i corresponds to the room to which a class corresponds in C_s.

Thus $o_i \neq g_i \; \forall \; i \leq k$ correspond to the same class for the rooms $o_i \neq g_i$.

Case 1: Room $g_1 == O_1$

In this case we can create an equivalent or better solution $OS' = \{g_1, O_2, \ldots O_n\}$ by exchanging g_1 for O_1.

Case 2: Room $g_1 \neq O_1$

Subcase 1: $g_1 \notin OS$

In this case since g_1 is not in OS, again we can exchange O_1 for g_1 in OS' as based on the greedy heuristic of our algorithm g_1 is the smallest valid room to which C_1 can be assigned. Note that if $S(O_1) < S(g_1)$ this violates the constraints of the problem since $S(O_1) < C_1$ and thus would be invalid. Thus at this step either OS has violated the problem constraints, or $OS' = \{g_1, O_2, \ldots O_n\}$ by swapping g_1 for O_1 where $g_1 \leq O_1$.

Subcase 2: $g_1 \in OS$

If g_1 exists in OS, note then the following must be true: Class C_1 is assigned to room O_1 and class C_p is assigned to room g_1. By the organization of OS and GS being based on C_s, we know that $C_p \geq C_1$. Since $O_1 < C_1$ (see reasoning above) violates the problem constraints, $O_1 \geq C_1$. In this case we can move O_1 to position p in OS' and $g_1 = O_p$ to position 1 in OS' without changing the minimum cost of OS' from OS. This follows by the following:

Cost at position $1 = S(O_1) - C_1$

Cost at position $p = S(g_1) - C_p$

$$S(g_1) = S(O_1) - Z \quad \text{where } Z \geq 0$$

Swapping O_1 and g_1 gives:

Swapped cost at Position $1 = S(O_1) - Z - C_1$

$= \text{Cost}_{\text{pos } 1} - Z$

Swapped cost at Position $2 = S(g_1) + Z - C_p$

$= \text{Cost}_{\text{pos } p} + Z$

Swapping is valid as $g_1 \leq O_1$ and $C_p \geq C_1$

and C_p can be assigned to g_1

and C_1 can be assigne to O_1

Since the cost of each pairing contributes equally to the total overall cost, the z's cancel out leading to an equivalent optimal solution: $OS' = \{g_1, o_2, \ldots o_p^*, \ldots, o_n\}$ where $o_p^* = o_1$.

Using the above logic cases we can, for every k element in OS create an equivalent or better solution OS' by swapping g_i with o_i. Thus this shows for $k = n$ GS is an optimal solution.

Note however the case when $k < n$. In this instance OS' would differ from GS by $n-k$ elements s.t. OS' would have the following not found in GS: $\{o_{k+1}, o_{k+2}, \ldots, o_n\}$. For this instance to occur $\{o_{k+1}, o_{k+2}, \ldots, o_n\}$ would have to be selected from discarded rooms omitted by GS as they had $s(r_i) < c_i$. Specifically note that GS would only have a $k < n$ when the allotment of rooms could not fit the enrollment of all classes. In such a case OS violates the constraints of the problem. Thus OS must be bounded by k the maximum number of valid room-class assignments.

Thus given the proof above, can conclude the correctness of GS.

Problem 3: 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, $C_{k'}$.

Key Idea:

Current capital at any given time point (or project point) is a constraint. Consider a similar problem for which we can select $K \leq n$ projects in which we want to maximize profit (w/ no capital constraints). In such a scenario, one could pick the K projects with the largest profits to maximize returns. Now returning to the given problem, we can abstract this idea with the capital constraints. Specifically, we can employ a greedy approach, where we select the project w/ the greatest profit w/i the capital constraints. This ensures at step $t+1$, we maximize our capital available. We follow the constraint that no project can be completed more than once.

Definitions:

GS = Greedy solution employed by algorithm

OS = Proposed optimal solution used for comparison in proof that is assumed to perform better than GS

$C_t \Rightarrow$ Capital after the t^{th} project is selected where $t \leq K \leq n$

Algorithm and Implementation:

Using the Key Idea here, the algorithm can be succinctly summarized as: At any step t (where $t-1$ projects have been completed), select the project with the greatest projected profit that also has a cost $\leq$ current capital. After completing the project update the current capital as

$C_t = C_{t-1}$ + Profit of the selected project at step t.

Note that to follow the constraints of the problem, no project will be selected more than once. That is once a project has been selected it will be removed from consideration at all subsequent steps.

To efficiently implement this, first sort projects by their costs/capital requirements. Next construct a max heap with priority according to projected profit, for all projects with costs $\leq C_0$. In the process of adding elements to the max heap, remove them from the sorted list of projects by cost (to ensure no project is duplicated).

At step t for a non-empty max heap, select and remove the top (i.e., highest priority/profit) element. Update C_t according to the rules above. Next add any elements from the list of remaining projects with capital requirements $> C_{t-1}$ but $\leq C_t$ to the heap.

Continue this process until the heap is empty and no valid projects remain (i.e, either we have performed all k projects or all remaining projects have a capital requirement $> C_t$). Return the capital at this point as the maximal capital.

Time Complexity:

This algorithm has multiple steps each with their own time complexities:

1) Sorting projects by cost $\Rightarrow O(n \log n)$
2) Max heap construction, removal, and insertion $\Rightarrow$ $O(n \log n)$
3) Determining which projects should be added to the heap $\Rightarrow$ $O(n)$

Given these step-level time complexities, can state that the time complexity of this algorithm is $O(n \log n)$.

Proof of Correctness:

Let $GS = \{g_1, g_2, \ldots, g_{k'}\}$ be the output of the above algorithm where $k' \leq k \leq n$ and g_i corresponds to the projects selected by the approach at each step $t=i$. Specifically, projects are ordered here in the order they were selected by the greedy algorithm.

Let $OS = \{O_1, O_2, \ldots, O_m\}$ be the output of the optimal solution, where $k' \leq m \leq k \leq n$ and each O_i ordered according to the project with the greatest allowable profit at step i, where allowable is defined as a project with cost $<$ current capital. With this ordering we create a parallel/comparable ordering with GS.

Base Case(s): $n = 1$ and all projects have a capital requirement $> C_0$.

In the case $n = 1$, GS will evaluate if this project has a cost $< C_0$ and if so return $C_1 = C_0 + P_1$, the maximum capital possible. If not it will return C_0. Note this aligns w/ OS here as $OS == GS$ in the case of $n = 1$ (we can only select upto 1 project). In the case all projects have a cost $> C_0$, both GS and OS are equivalent $GS = OS = \{\}$ and both return C_0. Thus for these cases GS is optimal.

Optimality for $n > 2$:

For this situation note there must be at least 1 viable project by the cost constraint of the problem, otherwise we revert to the base case.

Case 1: $O_1 == g_1$

In this case we can create an equivalent or superior solution $OS' = \{g_1, O_2, O_3 \ldots, O_m\}$ by exchanging g_1 for O_1.

Case 2: $O_1 \neq g_1$

Subcase 1: $g_1 \notin OS$

In this case note we can exchange g_1 for O_1 as by the greedy heuristic of our algorithm and orderings of GS and OS $g_1 \geq O_1$. Thus again we can exchange g_1 for O_1 in $OS' = \{g_1, O_2, O_3, \ldots, O_m\}$.

Subcase 2: $g_1 \in OS$

If g_1 is in OS at position P, observe that we can swap $g_1 = O_P$ with O_1 in OS':

$$OS' = \{g_1, O_2, O_3, \ldots O_P^* = O_1, \ldots, O_m\}$$

Note OS' equivalence with OS in terms of maximized capital → the same projects are selected, we have simply changed the order.

Using the above logic cases we can, for every k' element in OS create an equivalent or better solution OS' by swapping g_i with O_i. Thus this shows for $k' = m$ GS is an optimal solution.

Finally, need to examine the case where $m > k'$ (i.e., OS returns more projects than GS). Observe that this cannot happen. Specifically after following the swapping and exchange arguments above, $OS' = GS + \{O_{k'+1}, O_{k'+2}, \ldots, O_m\}$. In order for this to be the case $\{O_{k'+1}, O_{k'+2}, \ldots, O_m\}$ would need to be viable projects at step $t = k'$. However if they were, they would be in GS and thus if $m > k'$ we denote a contradiction by violating the capital constraints of the problem. Thus $m = k'$.

Thus given the proof above, can conclude the correctness of GS for maximizing capital according to the problem constraints.

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 $i = 0, 1, \ldots, b$.

Problem 4.1:

Definitions and Problem Formulation:

$\mathcal{G} = \{G_1, G_2, G_3, \ldots, G_b\}$

Given that we know $\exists$ at least one common path P that is an s-t path for each graph in $\mathcal{G}$, give a polynomial time algorithm to find the shortest such path.

Key Idea:

The problem states "suppose it is possible to choose a single path P that is an s-t path in each of the graphs $G_0, G_1, \ldots, G_b$" Thus we know that all graphs in $\mathcal{G}$ share all paths that are common (since we are finding the shortest common path). Therefore, we can construct a new graph $G_{int}: (V, E_{int})$ where E_{int} is the intersection of edges of each graph in $\mathcal{G}$.

Algorithm:

Create an intersection graph $G_{int}: (V, E_{int})$, where E_{int} is the intersection of edges of each graph in $\mathcal{G}$:

$$E_{int} = E_0 \cap E_1 \cap E_2 \cdots \cap E_b$$

On this intersection graph run breadth-first search (BFS) from node s to obtain the shortest s-t path that is in each graph in $\mathcal{G}$, storing the edge that updated a node's s distance. Having this, can backtrack from t to s to set the specific set of edges in shortest path.

Proof of Correctness:

For this algorithm to be correct, we need to show 2 key points 1) G_{int} is connected b/w s + t 2) G_{int} can be used to find the shortest such s-t path P as specified by the problem. We know both of these are true by the problem definition of 4.1. That is the problem states "suppose it is possible to choose a single

Path P that is an s-t path in each of the graphs $G_0, G_1, \ldots, G_b$". Thus any path P $\exists$ for all G_i in $\mathcal{G}$ is in G_{int}. Moreover the Path P's existence also specifies a connection in G_{int} b/w s-t. Thus the problem specifications prove the correctness of the algorithm. Finally, claim that BFS can be used to find the shortest path to t from s in a graph.

Time Complexity:

This algorithm has 2 separate steps:

1) Creating $G_{int} \Rightarrow O(b \cdot n^2)$
   - There are b graphs to intersect each w/ a number of edges E. Define the maximum number of edges any G_i can contain as n (which in the worst case could be $||V||^2$). While checking elements in E_i are in another E_j is $O(n)$, creating the set of edges in an efficient manner for intersection is also $O(n)$, leading to intersection time of $O(n^2)$.
2) Running BFS $\Rightarrow O(||V|| + ||E||) \Rightarrow O(n)$

Thus can claim this algorithm has a time complexity of $O(b \cdot n^2)$, where n in the worst case is $||V||^2$.

Problem 4.2:

Definitions and Problem Formulation:

Give a polynomial-time algorithm to find a sequence of paths $P_0 \to P_b$ of minimum cost, where P_i is an s-t path in G_i.

$$\text{Cost}(P_1 \to P_{b+1}) = \sum_{i=1}^{b+1} \ell(P_i) + K \cdot \Delta(P_1 \to P_{b+1})$$

- Let Δ represent changes as given by the problem
- **Note:** Changed graph formulation from $0 \to b$, to $1 \to b+1$; this re-indexing is employed in algorithm below.

Key Idea:

This is a twist on a dynamic programming, multi-way choice problem. Specifically it is very similar to segmented least squares. Here if can designate a set end point $j \leq n$, can find an $i \leq j$ that minimizes the cost b/w i & j. This cost can then be conceptualized at the minimal i as:

$(j - i + 1) \cdot$ Shortest path common between i & j +

K + Optimal Path up to and including $i-1$

↓ cost of changing paths ↓ minimum cost

This conceptualization sets the stage for a dynamic programming algorithm.

Algorithm:

Calculate and store the shortest s-t path for each graph in $\mathcal{G}$ (i.e. $G_1, G_2, \ldots, G_{b+1}$), using BFS. If no s-t path exists in any G_i, return ∞ as there is no s-t path possible at ~~that~~ time point. Assuming then, all G_i have at least 1 valid s-t path, and have computed the shortest path as stated create $\mathcal{G}_{int} = \{G_{i,j} \; ; 1 \leq i < j \leq b+1\}$ where $G_{i,j}$ is the intersection of all graphs from i to j. On each intersection graph in $\mathcal{G}_{int}$, run BFS to obtain the shortest s-t path length, storing these values as well (if for a $G_{i,j}$ no path exists, store that path length as ∞). For the path lengths computed and stored for all G_i and $G_{i,j}$ in $\mathcal{G}$ and $\mathcal{G}_{int}$, denote these values as $e_{i,j} \Rightarrow$ The s-t path length for graphs G_i through G_j with $1 \leq i \leq j \leq b+1$.

With processing complete, proceed to dynamic programming. Define a dynamic programming array DP of size b+2 and populate it according to the following formula:

$$DP[j] = \min_{1 \leq i \leq j} \left((j-i+1) \cdot e_{i,j} + \underbrace{K}_{\text{if } j \neq 1 \rightarrow \text{otherwise } +0} + DP[i-1] \right)$$

This defines the recurrence formula which D[0] is defined to be $D[0] = 0$. Iterate from 1 to b, populating DP according to this recurrence relationship. DP[b+1] will give the

minimum cost. To get the sequence of paths, we can backtrack through DP. Specifically from $j = b+1$, find the i that minimizes $(j-i+1)e_{i,j} + K + DP[i-1]$ and output the path of intersection specified by $e_{i,j}$ (note each BFS stores the paths, or can backtrack through the paths as required). Repeat this process for $j = i-1$ until $j == 0$ in which case the optimal sequence of paths has been output.

Proof of Correctness:

Base Cases: $\mathcal{G} = \{\}$ or $\mathcal{G} = \{G_1\}$ (no graphs or 1 graph where $b=0$)

In the case of $\mathcal{G} = \{\}$ ($b=-1$ empty), $DP[0]$ is still initialized and the minimum cost is 0, by the definition of the algorithm. Backtracking yields the termination condition resulting in no output.

In the case of a single graph, this reverts to BFS on G_1. Thus the only element $e_{i,j}$ is $e_{1,1} = BFS(G_1)$. Running the algorithm yields

$$DP[j] = \min_{1 \le i \le j} \left((j-i+1)e_{i,j} + K_{j \neq 1} + DP[i-1] \right)$$

$$DP[j=1] = (1-1+1)\,BFS(G_1) + 0 + 0$$

$$= BFS(G_1)$$

($j==1$ here, don't track the first change from an empty set of graphs to a non-empty set)

where $BFS(G_1)$ is the length of the shortest s-t path in G_1.

Inductive Hypothesis: For $j > 1$, assume $\forall\ 1 \le j \le bH$, $OP[j]$ returns the minimum cost (and implicitly the sequence of paths)

Inductive Step:

$$OP[j] = \min_{1 \le i \le j} \left((j - i + 1) e_{i,j} + K_{j \neq 1} + OP[i-1] \right)$$

$$\Downarrow$$

$$OP[j+1] = \min_{1 \le i \le j+1} \left(((j+1) - i + 1) e_{i,j+1} + K_{j \neq 1} + OP[i-1] \right)$$

Note that if $OP[j]$ holds by the IH, then it is dependent on its previously computed elements in OP. Moreover, it can never look ahead (past $j+1$) here as the range of the minimum is capped by the current "end" point $j+1$. If the best path is the extension of the final path in $OP[j]$, we note that i is uncharged between cases j and $j+1$, with the cost adjusted by the extension of another graph $G_{j+1} \Rightarrow$ i.e. we have an extra graph so the cost scales by the # of graphs w/ a consistent path from $j - i + 1$ to $j - i + 2$.

On the other hand, note the case when either G_{j+1} cannot extend the best final path in j and/or the best path in G_{j+1} reduces the cost by changing the s-t path. In this case we either need to find the best common path for a new i (as this cannot be the same if no common path exists), or pay the penalty $K + BFS(G_{j+1}) +$ the cost at the previous iteration. By selecting the minimum ($OP[i-1]$)

the algorithm ensures the cost minimization (and thus the sequence of paths), by checking all "left" precomputed costs (OP[i-1]) with the $e_{i,j} \cdot (j=j+1) - i + 1$ precomputed "right" path lengths.

Time Complexity:

This algorithm contains multiple steps with differing time complexities:

1) BFS on all graphs in $\mathcal{G} \Rightarrow O(b \cdot (|V| + |E|))$

$\Rightarrow O(n \cdot b)$

2) Creating $\mathcal{G}_\cap$ for all pairs of graphs:

$\Rightarrow O((n^2 b) \cdot b^2) \Rightarrow O(n^2 b^3)$

($n^2 b$: Time of a graph intersection; b^2: all pairs of graphs)

3) BFS on all graphs in $\mathcal{G}_{int} \Rightarrow O(n \cdot b^2)$

4) Running the OP approach $\Rightarrow O(b^2)$

(need to check all pairs in the worst case)

Given these complexities, observe Step 2 to be the time limiting step yielding the polynomial time complexity of $O(n^2 b^3)$.

Problem 5: Untangling signal superposition (KT 6.19)

You're consulting for a group of people (who would prefer not to be mentioned here by name) whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: "untangling" a superposition of two known signals. Specifically, they're picturing a situation in which each of two ships is emitting a short sequence of 0s and 1s over and over, and they want to make sure that the signal they're hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

This describes the whole problem; we can make it a little more explicit as follows. Given a string x consisting of 0s and 1s, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k. So $x' = 10110110110$ is a repetition of $x = 101$.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of x and y, since characters $1, 2, 5, 7, 8, 9$ form 101101–a repetition of x–and the remaining characters $3, 4, 6$ form 000–a repetition of y.

In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we're listening to: We want to make sure s "unravels" into simple repetitions of x and y. Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y.

Definitions and Problem Formulation:

S = String of length n consisting of 0s or 1s

x = pattern of 0s + 1s

y = another pattern of 0s + 1s

x^n = A repeating of pattern x n / length(x) times

⇒ Note: Change of definition here. This notation is used in downstream algorithm and analysis.

y^n = A repeating of pattern y n / length(y) times

Example for clarity of notation:

S = 10110111101 x = 1100 y = 10101

n = 10

both x^n and y^n are both of length n

x^n = 1100111001

y^n = 1010110101

Determine if s is an interleaving of patterns x and y efficiently. More explicitly can s, be decomposed into an s' + s'', repititions of x + y respectively.

KeyIdea:

If we build up x into $x^\wedge$ + y into $y^\wedge$, we can treat each element in $x^\wedge$ and $y^\wedge$ as a potential match to a position in s. If we can ever "align" s with $x^\wedge + y^\wedge$ s.t. the number of matches is equal to n, then s is an interleaving of x + y. If not then s is not an interleaving of x + y.

Algorithm:

Create a dynamic programming matrix DP of dimensions $(n+1)^2$. Populate DP[0,0]=0. Building on the KeyIdea above, conceptualize $y^\wedge$ as the rows from 1 to n. Similarly Conceptualize $x^\wedge$ as the columns from 1 to n. Define the following recurrence relation for populating DP.

$$DP[i,j] = \max(DP[i,j-1] + x^\wedge[j-1] == S[i+j-1],$$
$$DP[i-1,j] + y^\wedge[i-1] == S[i+j-1])$$

Populate the first row DP[0, 1:n] and column DP[1:n, 0] according to whether $x^\wedge[j-1]$ or $y^\wedge[i-1]$ matches S[j] or

$S[i]$ respectively. Note the above is a valid instance of the recurrence relationship where $x'[-1]$ and $y'[-1]$ can be conceptualized as $-\infty$ as these are undefined.

After populating the first row and column, populate the remainder of DP in either a row or column manner. If any element in DP == n, return TRUE → s is an interleaving of x and y. Otherwise return FALSE → s is not an interleaving of x.

→ Similarly could wait until all DP was populated and check if max DP == n.

If the interleaving of s' + s'' are desired, we can backtrack from where DP == n, building s' from right to left when take an element from x' and s'' from right to left when take an element from y'.

Implementation Intricacy: Given DP is $(n+1)^2$ and S of length n, it appears we will exceed the bounds of S, for $i+j-1 \geq n$. We note that in this case the boolean sum condition (e.g., $x'[j-1] == S[i+j-1]$ and $y'[i-1] == S[i+j-1]$) will always equate to FALSE or 0 to prevent maximum and access issues. If the populate all of DP approach + then backtrack to construct s' + s'' approach is taken,

s' + s'' should only be constructed from right to left (as specified above) during instances of decreases from the maximum, otherwise the length of s' + s'' > n (and would not be a valid pattern repeats that compose S).

Proof of Correctness:

Claim: For any prefix of s of length i that is a valid interleaving of x + y, the subsequent prefix of s of length i+1 is also a valid interleaving if there exists a valid next instance of x or y that matches s[i+1]. If no valid interleaving exists for a prefix of s, no valid interleaving exists for the subsequent prefix.

- Instance here is defined to be the next valid element of x or y that yields an interleaving.

Base Cases: length(s) == 0 or length(s) == 1

If the signal s is empty, the above algorithm returns TRUE as DP[0,0] == length(S) (this is a design choice of the algorithm in this corner case → treat a lack of signal as an interleaving of 2 patterns).

If the length(S) == 1, then s is an interleaving of x or y if s[0] == x[0] (which is equal to x^[0]) or s[0] == y[0] (which equates to y^[0]). By the formulation of the above algorithm, DP[1,0] and/or DP[0,1] will be 1 if y[0] and/or x[0] match with s[0]. As 1 is the length of S, we return TRUE. If neither match, we exceed the bounds of S

And DP[1,1] cannot be incremented according to Implementation Intricacy procedure outlined above. In this case the max(DP) == 0 < length(S) == 1 and S is not an interleaving of x and y.

Inductive Hypothesis: For $m > 1$, assume $\forall k \leq m < n$, where $m = i+j$, max(DP) identifies whether a prefix of S of length m is a valid interleaving of x + y.

Inductive Step:

Note that for the IH to be true, then max(DP) for a prefix of S of length m, must equal m if there is a valid interleaving, or be $< m$ if there isn't. As explained in the Claim above, if a valid interleaving of a prefix of S does not exist (i.e. max(DP) $< m$) then even if there is a matching with either x' or y' at m+1, the max of DP at m+1 can be at most $m < m+1$ and thus is not a valid interleaving.

If max(DP) == m, thus identifying a valid interleaving of a m-length prefix of S, then as specified by the above Claim, if the next valid instance of x or y matches w/ S[m+1] then a valid interleaving exists (for any maximum of DP == m). Otherwise if no match exists for any max(DP) from $m \rightarrow m+1$, then the max(DP) at m+1 is $m < m+1$ and thus no valid interleaving exists. Note the construction of x' + y' allows for the identification of the next valid instance of x or y for comparison with S[m+1].

Time Complexity:

Constructing $x^n + y^n$ should take linear time, but populating DP will require populating DP which given the dimensions will take $O(n^2)$ time. Therefore the time complexity of this algorithm is $O(n^2)$.