CS 124 Homework 2: Spring 2022

Your name: Jamin Liu

Collaborators: Elizabeth Li

No. of late days used on previous psets: 1
No. of late days used after including this pset: 3

Homework is due Wednesday at midnight ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignme nt must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style may count in your grades. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan in your results to turn them in.

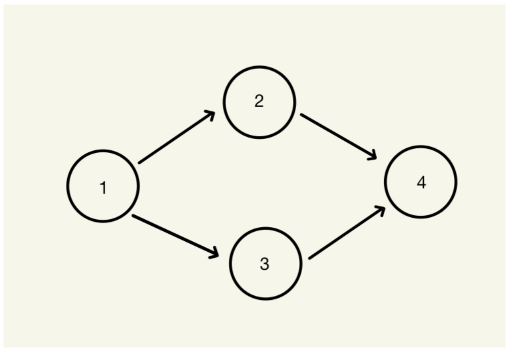
You can collaborate with other students that are currently enrolled in this course in brainstorming and thinking through approaches to solutions but you should write the solutions on your own: you must wait one hour after any collaboration or use of notes from collaboration before any writing in your own solutions that you will submit.

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or not credit. Again, try to make your answers clear and concise.

1. (a) **(7 points)** We saw in lecture that we can find a topological sort of a directed acyclic graph by running DFS and ordering according to the postorder time (that is, we add a vertex to the sorted list *after* we visit its out-neighbors). Suppose we try to build a topological sort by ordering in increasing order according to the preorder, and not the postorder, time. Give a counterexample to show this doesn't work, and explain why it's a counterexample.

Solution:

Consider the following graph:

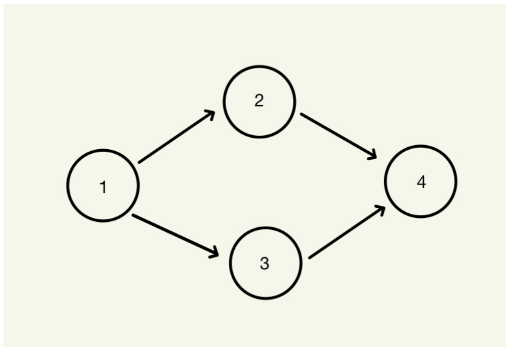


Ordered in increasing order according to preorder: 1 2 4 3
Ordered in decreasing order according to postorder: 1 3 2 4

Following DFS, by ordering in increasing order according to preorder in the above graph, we violate the definition of a topological sort. By definition, a topological sort is valid if for every directed edge from u to v, the task u must be completed before v may be started. However, by ordering by increasing preorder, this condition is violated, specifically, along the directed edge from 3 to 4. This is because we visit 4 before 3 is visited, while a topological sort would expect 3 to be visited before 4. Thus, in this counterexample, ordering in increasing order according to preorder does not produce a valid topological sort. Comparatively, if we order by decreasing postorder following DFS, we satisfy the condition as for every directed edge, the parent node is visited before the child node.

(b) **(7 points)** Same as above, but we try to sort by decreasing preorder time. **Solution:**

Consider the following graph:



Ordered in decreasing order according to preorder: 3 4 2 1
Ordered in decreasing order according to postorder: 1 3 2 4

Following DFS, by ordering in decreasing order according to preorder in the above graph, we violate the definition of a topological sort. By definition, a topological sort is valid if for every directed edge from u to v, the task u must be completed before v may be started. However, by ordering by decreasing preorder, this condition is violated, specifically, along the directed edge from 2 to 4. This is because we visit 4 before 2 is visited, while a topological sort would expect 2 to be visited before 4. This ordering similarly violates this condition for the directed edges from 1 to 3 and from 1 to 2. Thus, in this counterexample, ordering in decreasing order according to preorder does not produce a valid topological sort. Comparatively, if we order by decreasing postorder following DFS, we satisfy the condition as for every directed edge, the parent node is visited before the child node.

2. **(20 points)** News from Cambridge: every night, snow falls and covers all the sidewalks. Every morning, the city's lone snow shoveler, Pat, is tasked with clearing all the sidewalks of snow. Proper snow-shoveling technique requires that sidewalks on opposite sides of the same street be shoveled in opposite directions. (Every street has sidewalks on both sides.) Give an algorithm to find a snow-shoveling path for Pat that doesn't require any more walking than necessary—at most once per sidewalk. (If you have to assume anything about the layout of the city of Cambridge, make it clear!) (Your algorithm should work for any city, not just the Cambridge in which Harvard is.)

Solution:

Let us represent Cambridge as an undirected graph G with each edge representing a street with two sidewalks opposite one another and each vertex representing street intersections / ends of streets. By representing Cambridge as an undirected graph in this manner, we are guaranteed that Pat may traverse any undirected edge in both directions. That is, Pat may traverse the edge (shovel the sidewalk) from vertex 1 to vertex 2 or traverse that same edge (shovel the opposite sidewalk) from vertex 2 to vertex 1. We may assume the graph is connected, as the end of street or a street intersection cannot exist without a street and our edges are undirected. Also note that this representation works for any city and not just Cambridge. As we traverse our graph, we will keep track of edges visited to ensure we traverse each edge only once in each direction. Our algorithm is as follows:

1. Construct the undirected graph G as described above with each edge representing a street with two sidewalks opposite one another and each vertex representing street intersections / ends of streets.

2. We will have Pat shovel the city according to a DFS algorithm as seen below; however, with an additional condition.

```
def search(v):
    vertex v
    explored(v) := 1
    previsit(v)
    for (v,w) | E
        if explored(w) = 0 then search(w)
    postvisit(v)
    end search

def DFS (G(V,E))
    graph G(V,E)
    for each v | V do
        explored(v) := 0
    for first v in V do
        if explored(v) = 0 then search(v)
    end DFS
```

We interpret the DFS algorithm for Pat's shoveling as follows. Pat starts at a intersection (vertex) v in Cambridge having not visited any other streets (edges) or intersections. Using the helper function $SEARCH$, Pat notes that v has been explored and looks at all streets coming out of v to different intersections. For a intersection w, if w has not been explored, Pat shovels the street from v to w, specifically the sidewalk (v, w) and calls $SEARCH$ on w, continuing this pattern as deeply as he can. When Pat finds himself at an intersection where all connected intersections have been already been searched, he will call $POSTVISIT$ an return along the path which he came. For example, once Pat has explored all intersections connected to w, he will call $POSTVISIT$ and turn around and shovel the sidewalk (w, v) going the opposite direction on the path he came from. This will continue until Pat returns to the intersection he initially started at.

However, in addition to our DFS algorithm, if Pat finds an unvisited edge that leads to a vertex that has already been visited (back edge), he will traverse down the edge and then return back (shovel the sidewalk on one side, turn around and shovel the sidewalk on the other side going back).

Proof of Correctness:

In order to prove our algorithm is correct, we want to prove that every sidewalk is shoveled exactly once. This is the same as proving that all edges will be visited exactly once in each direction. Let us note that since our graph is connected, we may start from any vertex. We can deconstruct these into two different types of edges:

1. **Tree Edges:** Tree edges are traversed in the one direction for everytime $SEARCH$ is called or when Pat goes further into the DFS. Leveraging results from lecture, we know that every vertex will be visited by DFS, or that $SEARCH$ will be called on every vertex. Since $SEARCH$ is only called on a vertex if it is not explored and we set a vertex to explored once we call search on it, we know that we call $SEARCH$ at most one time on any vertex. As such, we know that $SEARCH$ is called once on every vertex, meaning every tree edge is traversed in one direction exactly once. These tree edges are also traversed in the opposite

direction for every call of $POSTVISIT$. Since $POSTVISIT$ is called once for every call of $SEARCH$, and $SEARCH$ is called exactly once on every vertex, we have that $SEARCH$ and $POSTVIST$ are called exactly once for each vertex. Thus, all tree edges are traversed in the both directions exactly once by our algorithm.

2. **Back Edges:** In the context of our problem, back edges are unexplored edges leading back to vertices that have already been explored. According to our algorithm, if Pat encounters a back edge, he will traverse the edge and immediately traverse it back in the opposite direction. Once Pat returns to the vertex he originally found the back edge from, the algorithm will have returned to the same position to continue running DFS and the back edge will not be traversed again. As such, all back edges are traversed in the both directions exactly once by our algorithm.

Since we have an undirected graph, there are no cross edges, as stated in lecture.

Thus, for all types of edges we have shown all edges will be visited exactly once in each direction, meaning all sidewalks will be shoveled exactly once, so our algorithm is correct.

Running Time:

To construct our graph of intersections (vertices) and streets (edges), our algorithm takes time $O(|E| + |V|)$. Next, running DFS, from lecture we have that DFS has time complexity $O(|E|+|V|)$. As such, the runtime complexity of this algorithm is $O(|E|+|V|)$ which is linear.

3. **(0 points, optional)**[1] This exercise is based on the 2SAT problem. The input to 2SAT is a logical expression of a specific form: it is the conjunction (AND) of a set of clauses, where each clause is the disjunction (OR) of two literals. (A literal is either a Boolean variable or the negation of a Boolean variable.) For example, the following expression is an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \overline{x_3}) \wedge (x_4 \vee \overline{x_1}).$$

A solution to an instance of a 2SAT formula is an assignment of the variables to the values T (true) and F (false) so that all the clauses are satisfied– that is, there is at least one true literal in each clause. For example, the assignment $x_1 = T, x_2 = F, x_3 = F, x_4 = T$ satisfies the 2SAT formula above.

Derive an algorithm that either finds a solution to a 2SAT formula, or returns that no solution exists. Carefully give a complete description of the entire algorithm and the running time.

(Hint: Reduce to an appropriate problem. It may help to consider the following directed graph, given a formula I in 2SAT: the nodes of the graph are all the variables appearing in I, and their negations. For each clause $(\alpha \vee \beta)$ in I, we add a directed edge from $\overline{\alpha}$ to β and a second directed edge from $\overline{\beta}$ to α. How can this be interpreted?)

[1]We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.

4. **(15 points)** The *risk-free currency exchange problem* offers a risk-free way to make money. Suppose we have currencies $c_1, \ldots, c_n$. (For example, c_1 might be dollars, c_2 rubles, c_3 yen, etc.) For various pairs of distinct currencies c_i and c_j (but not necessarily every pair!) there is an exchange rate $r_{i,j}$ such that you can exchange one unit of c_i for $r_{i,j}$ units of c_j. (Note that even if there is an exchange rate $r_{i,j}$, so it is possible to turn currency i into currency j by an exchange, the reverse might not be true— that is, there might not be an exchange rate $r_{j,i}$.) Now if, because of exchange rate strangeness, $r_{i,j} \cdot r_{j,i} > 1$, then you can make money simply by trading units of currency i into units of currency j and back again. (At least, if there are no exchange costs.) This almost never happens, but occasionally (because the updates for exchange rates do not happen quickly enough) for very short periods of time exchange traders can find a sequence of trades that can make risk-free money. That is, if there is a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdots r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$, then trading one unit of c_{i_1} into c_{i_2} and trading that into c_{i_3} and so on back to c_{i_1} will yield a profit.

   Design an efficient algorithm to detect if a risk-free currency exchange exists. (You need not actually find it.)

Solution:

A system of currencies can be represented as a directed graph G with directed edges as the exchange rates and vertices as currencies. To detect if a risk-free currency exchange exists, we want to detect a cycle of currency exchanges that yield a profit. As such, we seek a cycle of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdots r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. To do this, we implement the following algorithm:

1. Construct a graph G as described above with currencies as vertices $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ and directed edges (i, j) between c_i and c_j for every exchange rate $r_{i,j}$.

2. For all directed edges in G, assign every edge (i, j) with the weight $-\log(r_{i,j})$.

3. Run the Bellman-Ford algorithm as discussed in lecture and seen below on the graph for $n-1$ rounds of the outer loop.

```
def Bellman-Ford(G, length: E(G) → ℝ, s ∈ V(G)):
    dist = {s: 0, other vertices: ∞}
    prev = {s: null}
    for i in range(n-1):
        for (v,w) in E:
            if dist[w] > dist[v] + length(v,w):
                        prev[w] = v
                        dist[w] = dist[v] + length(v,w)
```

4. Run the Bellman-From algorithm for an nth round of the outer loop. If this round yields an update to *dist*, then return true (a negative cycle indicating a risk-free currency exchange has been detected), otherwise return false.

Proof of Correctness:

First, we will show that detecting a negative cycle $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ where every directed edge from c_i to c_j has the weight $-\log(r_{i,j})$ is the same as detecting a risk-free currency exchange. From the problem statement, we know that detecting a risk-free currency exchange is the same as finding a cycle of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. As such, we wish to show that a cycle of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$ is the same a negative cycle $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ where every directed edge from c_i to c_j has the weight $-\log(r_{i,j})$. In order to have a negative cycle, the sum of weights for the distinct edges along the path of the cycle must be negative. That is, $\sum(-\log(r_{i,j})) < 0$. As such, for some cycle of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$, we have:

$$\begin{aligned} r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} &> 1 \\ \prod r_{i,j} &> 1 \\ \log(\prod r_{i,j}) &> \log(1) \\ \sum(\log(r_{i,j})) &> 0 \\ \sum(-\log(r_{i,j})) &< 0 \end{aligned}$$

Thus, we have shown that detecting a negative cycle $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ where every directed edge from c_i to c_j has the weight $-\log(r_{i,j})$ is the same as detecting a risk-free currency exchange.

From a logical standpoint, we take the log of each exchange rate so that multiplication is preserved when the Bellman-Ford algorithm sums each weight. We multiply by -1 as the Bellman-Ford algorithm attempts to identify the shortest path, or the lowest sum of weights.

Next, we must show that we correctly identify a negative cycle. From lecture, we may assume the Bellman-Ford algorithm is correct. Moreover, from lecture, we have that for the nth round of running the Bellman-Ford algorithm, there is an update if and only if there exists a negative cycle. As such, detecting an update is synonymous with detecting a negative cycle which is synonymous with detecting a risk-free currency exchange. Thus, our algorithm correctly detects risk-free currency exchanges.

Running Time:

For the graph G representing a system of currencies with set of edges E and set of vertices V. To construct the graph, the algorithm takes linear time $O(|E| + |V|)$ and similarly to construct the weights as negative logarithms of exchange rates, the algorithm takes linear time $O(|E| + |V|)$. Next, the Bellman-Ford algorithm is run n times. From lecture, we know that the outer loop runs $|V|$ times and the inner loop runs $|E|$ times, giving us runtime $O(|E| \cdot |V|)$. Thus, the runtime complexity of this algorithm is $O(|E||V|)$.

5. **(20 points)** Suppose that you are given a directed graph $G = (V, E)$ along with weights on the edges (you can assume that they are all positive). You are also given a vertex s and a tree T connecting the graph G that is claimed to be the tree of shortest paths from s that you would get using Dijkstra's algorithm. Can you check that T is correct in linear time?

Solution:

Consider the following algorithm for checking the correctness of T:

1. First, we initialize a dictionary D of length $|V|$ for each vertex and it's corresponding distance from s with $s : 0$ initialized as the first element. Traverse T and find the length of the shortest paths for each vertex from s according to T and record these in D for $O(1)$ lookup time. We can do this by iterating through T with a DFS and using the formula $D[w] = D[v] + \mathit{length}(v, w)$ for two subsequent nodes v and w.

2. Run a singular round of Bellman-Ford's algorithm as seen below on G using D as *dist*:

```
def Bellman-Ford(G, length: E(G) → ℝ, s ∈ V(G)):
    dist = D
    prev = {s: null}
    for (v,w) in E:
        if dist[w] > dist[v] + length(v,w):
                    prev[w] = v
                    dist[w] = dist[v] + length(v,w)
```

If this round yields an update to *dist*, then return false (an update implies that T is not the tree of shortest paths), otherwise return true.

Proof of Correctness:

For this algorithm, we will assume from results in lecture that the Bellman-Ford algorithm is correct. For completeness, we want to show that if T is correct, then the algorithm returns true. For soundness. we want to show that if T is incorrect, the algorithm will return false.

First, let us take the case that T is correct. If T is correct, then it is the tree of shortest paths from s. As such, we know the lengths of shortest paths recorded when running DFS are indeed the shortest. Next, we run the Bellman-Ford algorithm and set *dist* to our dictionary of recorded shortest paths. The algorithm will check for all edges (v, w) in E, if $\mathit{dist}[w] > \mathit{dist}[v] + \mathit{length}(v, w)$. If this condition is met, *dist* is updated with the new shortest path $\mathit{dist}[v] + \mathit{length}(v, w)$. However, since we know that *dist* already contains all the shortest paths, this condition will never be met for any edge (v, w). As such, *dist* is not updated and our algorithm will correctly return true.

Next let us take the case that T is incorrect. If T is incorrect, then it is not the tree of shortest paths from s. As such, we know that at least one of the lengths of shortest paths recorded when running DFS is not actually the shortest path to a vertex. Next, we run the Bellman-Ford algorithm and set $dist$ to our dictionary of recorded shortest paths. Since we know that at least one of the recorded shortest paths is not actually the shortest, for some edge (v, w), the condition $dist[w] > dist[v] + length(v, w)$ will be true. As such, $dist[w]$ will be updated with the new shortest path $dist[v] + length(v, w)$. Since $dist$ is updated, our algorithm will correctly return false.

By demonstrating that if T is correct, our algorithm returns true and if T is incorrect, our algorithm returns false, we have proved that our algorithm is correct.

Running Time:

Using DFS to traverse the tree to get the lengths of shortest paths from s to each vertex takes $O(|V| + |E|)$ time as we visit every edge and vertex. To record the length of each shortest path in our dictionary is $O(V)$ as we do this for each vertex. Running a singular round of Bellman-Ford's algorithm requires us to iterate through all edges, so this takes $O(|E|)$. Thus, the run time complexity of this algorithm is $O(|V| + |E|)$ which is linear.

6. Patients who require a kidney transplant but do not have a compatible donor can enter a kidney exchange. In this exchange, patient-donor pairs $p_i - d_i$ may be able to donate to each other: there's an input function c such that for each pair (i, j) of patient-donor pairs, either $c(i, j) = 1$ and d_i can donate a kidney to p_j, or $c(i, j) = 0$ and d_i can't donate a kidney to p_j. As an example, suppose that we have patient-donor pairs:

$$p_1 - d_1, p_2 - d_2, p_3 - d_3, p_4 - d_4, p_5 - d_5,$$

that $c(3, 2) = c(3, 1) = c(2, 1) = c(1, 3) = 1$, and that for all other inputs c is 0. That is, in this example, d_3 can donate to p_2 or p_1, d_2 can donate to p_1, and d_1 can then donate to p_3 in the original $p_3 - d_3$ pair. Then a set of these donations can simultaneously occur: all those donations except $d_3 \to p_1$ could happen simultaneously, with $p_4 - d_4$ and $p_5 - d_5$ not participating. For every donor that donates a kidney, their respective patient must also receive a kidney, so if instead $c(1, 3) = 0$, no donations could occur: d_3 will refuse to donate a kidney to p_2 because p_3 won't get a kidney.

   (a) **(5 points)** Give an algorithm that determines whether or not there exists any nonempty set of donations that can occur.

   Solution:

   We may represent a group of patient donors as a directed graph G. Each vertex will represent a patient-donor pair $p_i - d_i$. A directed edge (i, j) from vertex v_i to v_j exists if $c(i, j) = 1$ or if d_i can donate a kidney to p_j. If $c(i, j) = 0$ then no directed edge exists. Consider the following algorithm to determine whether or not there exists any nonempty set of donations that can occur:

1. Construct a graph G as described above with patient-donor pairs as vertices and directed edges (i, j) from vertex v_i to v_j for every $c(i, j) = 1$ or if d_i can donate a kidney to p_j.

2. Run DFS on G and check whether or not any back edges exist based on preorder/postorder hierarchy. From lecture, we know there exists a back edge (u, v) if $\textit{postorder}(u) < \textit{postorder}(v)$, so we check this condition for all edges in G. If there are exists a back edge then return true (there exists a non-empty set of donations that can occur), otherwise return false.

Proof of Correctness:

For this algorithm, we will assume from results in lecture that DFS is correct. For completeness, we want to show that if there exists a non-empty set of donations that can occur, then the algorithm returns true. For soundness. we want to show that if there does not exist a non-empty set of donations that can occur, the algorithm will return false.

If a non-empty set of donations can occur, we know by the construction of our graph and definition of a valid kidney exchange, that there must exist a cycle in G. If there exists a cycle in G, then a run of DFS in G must yield a back edge (u, v) such that $\textit{postorder}(u) < \textit{postorder}(v)$, as proven in lecture. If a back edge is detected, our algorithm will correctly return true to indicate that a non-empty set of donations can occur.

If a non-empty set of donations cannot occur, we know by the construction of our graph and definition of a valid kidney exchange, that there cannot exist a cycle in G. If there does not exist a cycle in G, then a run of DFS in G will not yield a postorder hierarchy such that $\textit{postorder}(u) < \textit{postorder}(v)$ for any edge (u, v). As such, there does not exist a back edge in our graph G. Accordingly, our algorithm will correctly return false to indicate that a non-empty set of donations cannot occur.

Running Time:

To construct our graph of patient-donors, our algorithm takes time $O(|E| + |V|)$. Next, running DFS, from lecture we have that DFS has time complexity $O(|E| + |V|)$. Checking the post-order hierarchy to see if each edge is a back edge takes $O(|E|)$. As such, the runtime complexity of this algorithm is $O(|E| + |V|)$ which is linear.

(b) **(20 points)** Suppose that no set of donations can occur in the previous part, but we add of an altruistic donor, d_0. This altruistic donor is not bound to a patient, and is unconditionally willing to donate a kidney. Additionally, for each donation from d_i to p_j, consider that there is some value v_{ij} associated with that donation. Give an algorithm that returns the highest value donation sequence. For partial credit, you can consider the cases where 1) every donation has the same value or 2) donations have possibly-distinct but only positive values.

Solution:

We will construct the graph in a similar manner as done in part (a), adding a vertex v_o corresponding our altruistic donor, d_0 and adding directed edges from v_0 to all compatible patients. Next for every edge (i, j), we add the weight v_{ij} for the value of the donation.

We know that there are no valid donation sequences before d_0 is added by the problem statement, meaning there are no pre-existing cycles present in the graph and that v_0 must be present in any donation sequence. Moreover v_0 only has edges directed outwards and thus, cannot be part of a cycle. As such, the addition of v_0 does not introduce any new cycles meaning our graph is a DAG. As such, to determine the sequence with the maximum value, consider the following algorithm:

1, Run DFS on G starting with v_0 and construct a topological following decreasing postorder of vertices.

2. Next we run a single iteration of a modified version of Dijkstra's algorithm which identifies largest distances to a vertex rather than least. The algorithm is as follows:

```
def Dijkstra(G, length: E(G) → ℝ, s ∈ V(G)):
    vertices_to_explore = [s: 0]
    dist = {s: 0, other vertices: −∞}
    prev = {s: null}
    while vertices_to_explore != []:
                v = deletemin(vertices_to_explore)
            for (v,w) in E:
                        if dist[w] < dist[v] + length(v,w):
                                prev[w] = v
                                dist[w] = dist[v] + length(v,w)
                                insert(w,dist[w],vertices_to_explore)
```

After running this algorithm, we find the vertex with largest associated distance from v_0 in *dist* and trace it's path through *prev* to identify our highest value donation sequence.

Proof of Correctness:

To prove the correctness of our algorithm, we proceed by strong induction. We want to demonstrate that at any given time when looping through the vertices in Dijkstra's, the values in our dictionary *dist* are the largest distances from the source vertex for all vertices visited.

Base Case (—V— = 1): When we only have one vertex, namely the donor is v_0. Our algorithm will return 0 because the distance from v_0 to itself is 0.

Inductive Hypothesis: Suppose for all nk, for the kth vertex in topological sort ordering, the values in our dictionary *dist* are the largest distances from the source vertex for all vertices visited.

Inductive Step: We wish to prove that for the $k+1$th vertex in topological sort ordering, the values in our dictionary *dist* are the largest distances from the source vertex for all vertices visited. We know by our inductive hypothesis that this is true up till the kth vertex. As such, for the ancestors in the topological sort of the v_{k+1}, *dist* holds the largest distances from the source vertex, more specifically, this means *dist* holds the largest distances from the source vertex for all parents of the v_{k+1}. Next, following our algorithm, the distance from the source vertex for the v_{k+1} is initialized to $-\infty$, we will calculate the maximum distance from the source to the v_{k+1} by checking for all parents of v_{k+1} if $dist[v_{k+1}] < dist[v_p] + length(v_p, v_{k+1})$, where v_p is a parent vertex of v_{k+1}. If this condition is met, *dist* and *prev* are updated with the new largest distance and previous vertex in this largest distance path for v_{k+1}. As such, for the $k+1$th vertex in the topological sort ordering, the values in our dictionary *dist* are the largest distances from the source vertex for all vertices visited.

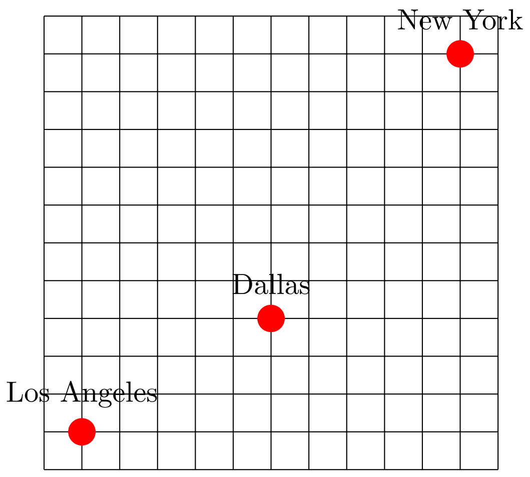
Thus, through strong induction, we have proven our algorithm is indeed correct.

Running Time:

Constructing our graph takes $O(|V| + |E|)$, From lecture, we have that running DFS on a graph also takes time $O(|V| + |E|)$ Running the maximum version of Dijkstra's algorithm can be done in $O(|E| \log |V|)$ time under the assumption of a Fibonacci-heap implementation. Finding the vertex with greatest path length and tracking its path can be done in $O(|V| + |E|)$. As such, the runtime of our algorithm is $O(|V| + |E|)$, which is linear.

7. Tony Stark has been thinking about how he can be more effective as Iron-Man and he's finally figured it out: two Iron-Men! He has two Iron-Man suits and he can control each remotely. Unfortunately, he's been having trouble getting the technology exactly right, so every time he makes a move in one suit, the other suit follows with a different move. Precisely, if Iron-Man 1 moves right, Iron-Man 2 moves up; if IM1 moves left, IM2 moves down; if IM1 moves up, IM2 moves left; and if IM1 moves down, then IM2 moves right. To slow him down, Thanos dropped one suit in Los Angeles and the other in Dallas. Tony needs your help getting both his suits back to Stark Industries in New York.

   Because of COVID-related travel restrictions, the Iron-Men cannot leave the United States. For the sake of this problem, assume that the United States can be modelled as an n by n grid, as below (climate change has shaved off the East and West coasts).

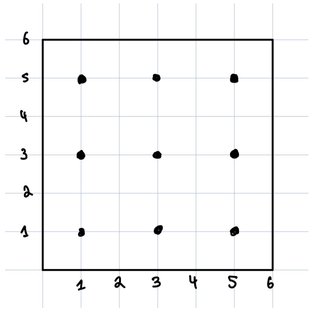


If an Iron-Man tries to move off the grid or into an obstacle, it merely stays in place. Additionally, each step has a cost that depends on the robot's location. For example, moving left from (0, 1) might cost 1 fuel but moving left from (10, 15) might require jumping over someone's backyard pool and thus might cost 3 fuels. Once a robot reaches Stark Industries, it powers down and costs 0 fuels even as its counterpart continues to move. You are given the positions of Los Angeles (x_ℓ, y_ℓ), Dallas (x_d, y_d), and New York (x_{ny}, y_{ny}), the positions of all obstacles (x_{o_i}, y_{o_i}), and the cost of every possible move from every possible location.

(a) **(10 points)** Give and explain an asymptotic upper bound on how many possible positions there are for the pair of Iron-Men, and explain why no better asymptotic upper bound is possible.

Solution:

Since there are n^2 positions in the grid, and each Iron-Man can occupy any of these positions the asymptotic upper bound on how many possible permutation there are for the pair of Iron-Men is $O(n^4)$. To show why no better asymptotic upper bound is possible, consider the following situation:



In this arrangement of obstacles shown above, imagine a situation in which one IM is locked vertically between obstacles while the other is free to move horizontally, or vice versa where one IM is locked horizontally between obstacles while the other is free to move vertically. There are $\Theta(n^2)$ such positions in which an IM can be locked while the other IM is free to move along $\Theta(n)$ rows and $\Theta(n)$ columns. Thus, in this case, we have $\Theta(n^4)$ positions as a lower bound. More specifically, locking an IM while the other is free to move vertically or horizontally gives us the bound of the possible positioins for the two IMs as $\Theta((n^2 -^2 /4\rfloor)^2) = \Theta((3n^2/4)^2) = \Theta(9n^4/16) = \Theta(n^4)$. As such, there is not a better asymptotic upper bound than the one proposed above.

(b) **(20 points)** Give an algorithm to find the cheapest sequence of {L, R, U, D} moves (that is, the one that requires you to buy the smallest amount of robot fuel) that will bring both Iron-Men home to New York.

Hint: Try to represent the position of the two Iron-Men as a single vertex in some graph. For full credit, it suffices to find an $O(n^8)$ algorithm, but an $O(n^4 \log n)$ algorithm may be eligible for an exceptional score.

Solution:

We may represent the $\Theta(n^4)$ pairs of positions for which the Iron Men can occupy as a graph G where pairs of locations $((x_i, y_i), (x_j y_j))$ representing the locations each Iron-Man are represented by vertices. Any possible move from a pair of locations $((x_i, y_i), (x_j y_j))$ to another $((x_i, y_i)', (x_j, y_j)')$ we represent with a directed edge whose weight is with the fuel for the corresponding move. To find the cheapest sequence of {L, R, U, D} moves, consider the following algorithm:

1. Construct the graph G as described above where pairs of locations $((x_i, y_i), (x_j y_j))$ representing the locations each Iron-Man are represented by vertices. Any possible move from a pair of locations $((x_i, y_i), (x_j y_j))$ to another $((x_i, y_i)', (x_j, y_j)')$ we represent with a directed edge whose weight is with the fuel for the corresponding move.

2.Run Dijkstra's Algorithm on G, using the vertex $((xl, yl), (xd, yd))$ as the source. The algorithm is as follows:

```
def Dijkstra(G, length: E(G) → ℝ, s ∈ V(G)):
    vertices_to_explore = [s: 0]
    dist = {s: 0, other vertices: ∞}
    prev = {s: null}
    while vertices_to_explore != []:
            v = deletemin(vertices_to_explore)
        for (v,w) in E:
            if dist[w] > dist[v] + length(v,w):
                prev[w] = v
                dist[w] = dist[v] + length(v,w)
                insert(w,dist[w],vertices_to_explore)
```

Dijkstra's identifies the shortest paths between the source vertex and all other vertices, so we may find the shortest path from $((x_l, y_l), (x_d, y_d)))$ to $((x_ny, y_ny), (x_ny, y_ny))$ in *dist*. This shortest path from Dijkstra's is to the cheapest sequence of moves that brings both Iron-Men to New York since each edge is weighted with the cost of fuel. Also note that our algorithm assumes IM1 is in Los Angeles and IM2 is in Dallas but our algorithm would function the same if IM1 is in Dallas and IM2 is in Los Angeles, we would simply use $((x_d, y_d), (x_l, y_l))$ instead.

Proof of Correctness:

We wish to show that our algorithm identifies the cheapest sequence of moves to bring the Iron-Men to New York. We will assume from results from lecture that Dijkstra's algorithm is correct. First, if there exists a shortest path from $((x_l, y_l), (x_d, y_d))$ to $((x_ny, y_ny), (x_ny, y_ny))$, we are guaranteed that Dijkstra's will find it, per our assumption. Next, we know that if our algorithm identifies a shortest path from the source vertex to $((x_ny, y_ny), (x_ny, y_ny))$, we know by the construction of our path that this is the cheapest sequence of moves to bring each Iron-Man to New York as each edge is weighted with the cost of fuel for the corresponding move.

Running Time

First, creating our graph takes time $O(|V| + |E|)$ as we create all vertices and edges. From part (a) we know that there are an upper bound of n^4 vertices. Let us note the for every vertex we can move in four directions (up, down, left, right), so there are a maximum of four directed edges coming out from each vertex, giving an upper bound of $4n^4$ edges. Thus, the overall construction of our graph takes time $O(n^4)$. In terms of the runtime of Dijkstra's, as stated in lecture, Dijkstra runs in $O(|V| \log |V| + |E|)$ if using a Fibonacci heap. Assuming this method of implementation, the overall runtime of algorithm is then is then $O(n^4 \log n)$.