CS 124 Homework 2: Spring 2024

Collaborators: Ryan Jiang, Rowan Wang, Ossimi Ziv, Kevin Liu, Denny Cao, Charlie Chen

No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

Homework is due Wednesday Feb 14 at 11:59pm ET. Note that although this is *two* weeks from the date on which this is assigned, your first programming assignment will also be released on Feb 7, so you should budget your time appropriately. You are allowed up to **twelve** late days, but the number of late days you take on each assignment must be a nonnegative integer at most **two**.

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 your papers to turn them in.

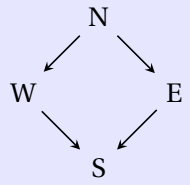
Collaboration Policy: You may collaborate on this (and all problem sets) only with other students currently enrolled in the class, and of course you may talk to the Teaching Staff or use Ed. You may also consult the recommended books for the course and course notes linked from the timetable. You may not use Generative AI or large language models, or search the web for solutions, or post the questions on chat forums. Furthermore, you must follow the "one-hour rule" on collaboration. You may not write anything that you will submit within one hour of collaborating with other students or using notes from such sources. That is, whatever you submit must first have been in your head alone, or notes produced by you alone, for an hour. Subject to that, you can collaborate with other students, e.g. in brainstorming and thinking through approaches to problem-solving.

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 no credit. Again, try to make your answers clear and concise.

Problems

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.

Answer: We will consider the following graph as a counter example:



If we run DFS starting at node N, we get node N has preorder 1 and postorder 8, node W has preorder 2 and postorder 5, node S has preorder 3 and postorder 4, and node E has preorder 6 and postorder 7. Thus, if we attempt to build a topological sort by ordering by increasing preorder, we get N, W, S, E. However, this is not a valid topological sorting since the directed edge (E,S) exists but S comes before E in the ordering. Therefore, this graph is a counterexample which shows that ordering by increasing preorder does not work.

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

Answer: We will consider the following graph as a counter example:

$$X \longrightarrow Y$$

If we run DFS starting at node X, we get node X has preorder 1 and postorder 4 and node Y has preorder 2 and postorder 3. Thus, if we attempt to build a topological sort by ordering by decreasing preorder, we get Y, X. However, this is not a valid topological sorting since the directed edge (X,Y) exists but Y comes before X in the ordering. Therefore, this graph is a counterexample which shows that ordering by decreasing preorder does not work.

2. **(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 $r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$, 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.)

Algorithm: To detect if a risk-free currency exchange program exists, we must see there exists a sequence $r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1}$ exists 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 true.

We will define our algorithm to detect a risk-free currency exchange as such: Construct a directed graph $G = (V, E)$ such that $V = \{c_1, \ldots, c_n\} \cup \{c_0\}$ and E be the set of directed edges such that for all currency exchange rates $r_{i,j}$, edge $(c_i, c_j) \in E$. Let us define an edge length function $l(c_i, c_j) = -\log(r_{i,j})$. Finally, let c_0 be defined as a vertex such that edges $(c_0, c_i) \in E$ and $r_{c_0,c_i} = 0$ for all $c_i \in V$ (and thus $l(c_0, c_i) = 0$ for all $c_i \in V$).

Note that the following descriptions borrow heavily from the Lecture 5 notes.
Then, we will run the Bellman-Ford algorithm on G starting from c_0. We will store the distance $d[...]$ as d_1. Next, we will run the inner loop of the Bellman-Ford algorithm one more time, storing the new distance as d_2. Finally, if $d_1 \neq d_2$, then return True. Otherwise, return False.

Correctness: We will now prove that our algorithm returns True if and only if there exists a risk-free currency exchange program.

Proof. Suppose there exists a risk-free currency exchange program. Then, we are given from the problem description that there must exist a sequence of exchange rates 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 true. Let us transform this inequality by taking the log of both sides and multiplying by -1. This yields:

$$\log(r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1}) > 0$$

$$(\log(r_{i_1,i_2})) + (\log(r_{i_2,i_3})) \ldots + (\log(r_{i_{k-1},i_k})) + (\log(r_{i_k,i_1})) > 0$$

$$(-\log(r_{i_1,i_2})) + (-\log(r_{i_2,i_3})) \ldots + (-\log(r_{i_{k-1},i_k})) + (-\log(r_{i_k,i_1})) < 0$$

Therefore, if the condition $(-\log(r_{i_1,i_2})) + (-\log(r_{i_2,i_3})) \ldots + (-\log(r_{i_{k-1},i_k})) + (-\log(r_{i_k,i_1})) < 0$ is true, then we know 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 true and there exists a risk-free currency exchange program. Note that this is equivalent to finding if there exists a negative cycle in G since we defined $l = -\log(r_{i,j})$ for all $(c_i, c_j) \in E$. Since we run the Bellman-Ford algorithm starting from c_0, which is by definition connected to every other vertex in G, we know that our algorithm reaches every component of G. In addition, c_0 will not be included in any cycles of G since c has no in-neighbors by definition. Therefore, by the correctness of the Bellman-Ford algorithm for detecting negative cycles, our algorithm will return True if there exists a risk-free currency exchange program.

Now suppose that our algorithm returns True. Then, by the correctness of the Bellman-Ford algorithm for detecting negative cycles, we know that there exists a negative cycle in G. Thus, by our definition of length function l, we know for some sequence of exchange rates:

$$(-\log(r_{i_1,i_2})) + (-\log(r_{i_2,i_3})) \ldots + (-\log(r_{i_{k-1},i_k})) + (-\log(r_{i_k,i_1})) < 0$$

$$-\log(r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1}) < 0$$

Multiplying by -1 and taking the exponent on both sides, we have:

$$r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$$

By the problem description, if there exists a sequence of exchange rates 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 true, then there exists a risk-free currency exchange program. Thus, if our algorithm returns True, then there exists a risk-free currency exchange program.

Hence, our algorithm returns True if and only if there exists a risk-free currency exchange program. □

Runtime:

Proof. Let $m = |E|$ and $n = |V|$. Thus, our construction of graph $G = (V, E)$ takes $O(n+m)$ time to create n vertices and m edges. Next, our edge length function takes the negative log of all $r_{i,j}$ to construct our edge lengths for G. Thus, assuming that taking the negative log is $O(1)$ running time, this step in our algorithm takes $O(m)$ running time. Next, our algorithm runs the Bellman-Ford algorithm on G, plus another additional iteration of the inner loop through all the edges. This means the outer-loop of our modified Bellman-Ford runs n times (as opposed to $n-1$ times) and the inner loop takes $O(m)$ time to iterate through all edges of G. Therefore, our algorithm solves the risk-free currency exchange problem in $O(n+m+m+mn) = O(mn)$ time. □

(Note: You may find the question above more accessible after the Monday (Feb. 5) lecture.)

3. In this problem we consider two versions of "Pebbling Games", a class of "solitaire" games (played by one player). In both versions, the input to the game includes an undirected graph G with n vertices and m edges, and positive integer parameters c and $k \leq n$. At the beginning of the game, the player is given k pebbles which are placed on vertices $0, 1, \ldots, k-1$. At any given moment of time, the k pebbles are located at some k (not necessarily distinct) vertices of the graph. In each move, the player can move any pebble to a vertex v, provided that prior to this move, at least c vertices adjacent to v have a pebble. (The pebble moved does not have to be one of those that start out adjacent to v.)

   (a) **(20 points)** In version 1 of the game, we have $c = k = 1$ and so the unique pebble effectively moves across an edge of the graph. In this version, the player wins if there is a strategy that traverses every edge (in both directions) in exactly $2m$ moves, where m is the number of edges of G. Give a winning strategy (i.e., an algorithm that outputs an order in which the edges are traversed) for the player for every connected graph G.

   Algorithm: To give a winning strategy for every connected graph G, we will implement a modified version of DFS called TET (traverse edges twice) as such:

   Define a set called Explored to keep track of which vertices our algorithm has visited. Define a list called Walk to keep track of the order in which edges are traversed.

Next, we will define functions previsit and postvisit such that previsit(u,v) adds edge (u, v) to Walk and postvisit(u,v) adds edge (v, u) to Walk. Finally, we define a sub-function called Search(u). First, add vertex u to Explored. For all edges $(u, v) \in E$, we check if v is not in Explored. If not, we call previsit(u,v), recursively call Search on v, and then call postvisit(u,v). Else, if v is already in Explored, then call previsit(u,v) and then postvisit(u,v).

Thus, we define TET(s) as such: call Search(s) on an arbitrary starting vertex $s \in V$ and return Walk.

Correctness: We will now prove that TET traverses every edge exactly once in both directions and returns a Walk that is well-ordered.

Proof. Since we are given that G is a connected undirected graph, it follows by definition that every vertex $v \in G$ is reachable from vertex s. Furthermore, by Lecture Notes 4, we know that a undirected graph G cannot have any cross edges. Therefore, G can only contain tree edges and back edges, which we will consider as separate cases:

We will show that TET traverses a back edge $(u, v) \in E$ exactly once in both directions if and only if (u, v) not in Walk. First, we assume $(u, v) \in E$ not in Walk. By definition of a back edge, we know that there exists some path $\{v, ..., u\}$ in G. Furthermore, we know that TET must have already visited every vertex in the path $\{v, ..., u\}$, otherwise (u, v) would not be a back edge. It follows that v has already been visited by TET so v is already in Explored. Therefore, by the definition of our algorithm, we call previsit(u,v) and postvisit(u,v), which adds the edges (u, v) and (v, u) to Walk. Thus, given our assumption that $(u, v) \in E$ not in Walk, this means we have traversed the back edge (u, v) exactly once in both directions. Now we assume that TET traverses a back edge (u, v) exactly once in both directions. Suppose for the sake of contradiction that (u, v) is already in Walk. Thus, by our definition of Search, u must be in Explored. However, if u is already in Explored, then Search(u) would not be called again by the definition of our algorithm. Thus, TET would not traverse (u, v) in the first place, which is a contradiction. Hence, if (u, v) not in Walk, TET traverses a back edge $(u, v) \in E$ exactly once in both directions. Therefore, we have proved that TET traverses a back edge $(u, v) \in E$ exactly once in both directions if and only if (u, v) not in Walk.

We will now show that TET traverses a tree edge $(u, v) \in E$ exactly once in both directions if and only if $(u, v) \in E$ is not in Walk. First, we assume $(u, v) \in E$ not in Walk. Therefore, we know by definition of a tree edge that vertex v has not been explored by our algorithm before, or equivalently, is not already in the set Explored. Hence, by our definition of TET, we call previsit(u,v), Search(v), and postvisit(u,v). This ensures we add both edges (u, v) and (v, u) to Walk through previsit and postvisit. Given our assumption that $(u, v) \in E$ not in Walk, this means we have traversed the tree edge (u, v) exactly once in both directions. Now we assume that TET traverses a tree edge

(u,v) once in both directions. Suppose for the sake of contradiction that (u,v) is already in Walk. Thus, by our definition of Search, u must be in Explored. However, if u is already in Explored, then Search(u) would not be called again by the definition of our algorithm. Thus, TET would not traverse (u,v) in the first place, which is a contradiction. Hence, if (u,v) not in Walk, TET traverses a tree edge $(u,v) \in E$ exactly once in both directions. Therefore, we have proved that TET traverses a tree edge $(u,v) \in E$ exactly once in both directions if and only if (u,v) not in Walk.

Finally, we will prove that the sequence of edges given by TET is well ordered with induction. Consider the base case when $|V| = 2$ in connected graph G. Thus, the length of Walk is 2 and is well ordered since G is undirected. For our inductive hypothesis, assume that the length of Walk is n and the last edge in Walk is (u,v). We will consider three cases: when v is a leaf (there are no more edges to unexplored vertecies), when v has a tree edge to w, and when v has a back edge to w. When v is a leaf, Search(v) terminates without doing anything and Search(u) appends (v,u) to Walk. When v has a tree edge (v,w), Search(v) appends the edge (v,w) to Walk. When v has a back edge (v,w), Search(v) appends the edge (v,w) first to Walk. Since every case first appends an edge starting at v to Walk, we know that the well-orderness of Walk when $|Walk| = n$ implies the well-orderness of Walk when $|Walk| = n+1$. Therefore, by induction, TET gives a sequence of edges which is well-ordered.

As such, since we have proved that TET traverses an edge exactly once in either direction if and only if the edge does not exist in Walk, we have proved that TET outputs a well-ordered sequence of edges which traverses every edge in both directions in $2m$ moves. □

Runtime:

Proof. Let $m = |E|$ and $n = |V|$. Since our algorithm calls Search on every vertex only once, this takes $O(n)$ time. In addition, since our algorithm adds a total of $2m$ edges to Walk, under the assumption that appending to a list is $O(1)$ time, this takes $O(2m)$ time. Therefore, the runtime of TET is $O(n+2m)$ which is $O(n+m)$. □

(b) **(20 points)** In version 2 of the game, c and k are arbitrary and there is a special designated target vertex t such that the player wins if they can place a pebble on t in any finite number of moves. Give an algorithm to determine if a given graph G has a winning strategy. For full credit, your algorithm should run in time at most $O(n^{2k})$.

Algorithm: We define our algorithm to determine if a given graph G has a winning strategy as such:

Construct a directed graph $G' = (V', E')$ such that each V' is the set of all unique game states of G given c and k. That is, each vertex $v \in V$ represents a unique state of graph G where the k pebbles are placed at some k (not necessarily distinct) vertices

of G.Furthermore, we define the set of edges E' such that an directed edge $(v_i, v_j) \in E'$ if v_j can be reached in a single move from v_i (note that this is a directed edge since you it may not be possible to move from one game state to another and back again). Thus, if a player can validly move one pebble (ie. a move to u is valid if and only if u has c adjacent vertices with a pebble) to get from the game state of v_i to v_j. Finally, when we define a vertex $v_i \in V'$, we mark the vertex if it is a "winning" game state (ie. has a pebble at t).

Next, we run BFS on G' starting from the initial game state v_0 (the initial placement of all k pebbles on G). If the path contains at least one "winning" marked vertex, return True. Otherwise, return False.

Correctness: We will now prove that our algorithm returns True if and only if there exists a winning strategy.

Proof. To find the existence of a winning strategy, we must find a sequence of moves between "states" of G such that there is a sequence from the initial state of G to a state with a pebble at t. Recall that we defined our graph G' such that each vertex $v \in V$ represents a unique state of graph G and each edge $(v_i, v_j) \in E'$ if v_j can be reached in a single move from v_i. Thus, by the definition of G', finding the existence of a winning strategy equates to finding if there exists a path from initial state v_0 to a winning state v_w.

Now suppose our algorithm returns True. Since our algorithm runs BFS on G' starting at v_0, this implies that there exists a path from v_0 to a winning vertex v_w by the correctness of BFS. As we have shown above, this must mean that there exists a winning strategy. Now suppose there exists a winning strategy in the pebble game. By the correctness of graph G' above, this must mean that there exists a path from initial game state v_0 to a winning vertex v_w. Therefore, we know by the correctness of BFS that our algorithm will find the path from v_0 to v_w and return True.

Thus, we have proved that our algorithm returns True if and only if there exists a winning strategy. □

Runtime:

Proof. In our construction of G', we must construct V' and E'. Since each of the k pebbles in G can be at any of the n vertices (since there can be more than one pebble at a vertex), there is a total of n^k possible game states. Thus, construction of V' takes $O(n^k)$ time. In addition, since it is possible that all game states are reachable from each other, we must perform $n^k(n^k - 1)$ checks to see if an edge exists between any two vertices in V'. Assuming the check takes constant $O(1)$ time, this takes $O(n^{2k})$ time. Finally, since our algorithm runs BFS on G', this takes $O(|V| + |E|)$ time. Therefore, our algorithms total runtime is $O(n^k + n^{2k} + n^k + n^{2k})$ which is $O(n^{2k})$. □

(c) **(0 points, optional)**[1] Find an algorithm for version 2 of the game which runs in time $o(n^{2k})$—the faster, the better!

Answer:

4. It sometimes happens that a patient who requires a kidney transplant has someone (e.g. a friend or family member) willing to donate a kidney, but the donor's kidney is incompatible with the patient for medical reasons. In such cases, pairs of a patient and 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 a given function c such that for each pair (i, j) of patient-donor pairs, either $c(i, j) = 1$, meaning that d_i can donate a kidney to p_j, or $c(i, j) = 0$, meaning that d_i can't donate a kidney to p_j. As an example, suppose that we have five patient-donor pairs:

$$(p_1, d_1), (p_2, d_2), (p_3, d_3), (p_4, d_4), (p_5, d_5).$$

Suppose also 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: e.g. d_3 gives a kidney to p_2, d_2 gives a kidney to p_1, and d_1 gives a kidney to p_3. (In this example, (p_4, d_4) and (p_5, d_5) don't participate). 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 a set of donations can occur.

Algorithm: We will define our algorithm which determines whether or not a set of donations can occur as such:

Construct a directed graph $G = (V, E)$ such that all patient-donor pairs $(p_i, d_i) \in V$. Furthermore, for all pairs of patient-donor pairs (i, j) such that $c(i, j) = 1$, we construct a directed edge $(v_i, v_j) \in E$ (v_i denotes the patient-donor pair (p_i, d_i)). That is, if d_i can donate a kidney to p_j, then there exists edge $(v_i, v_j) \in E$.

Then, run DFS from an arbitrary vertex and obtain the preorder and postorder numbers of all $v \in V$. Next, iterate through the $(v_i, v_j) \in E$ and check if postorder(v_i) < postorder(v_j). If any postorder(v_i) < postorder(v_j), return True. Otherwise, return False.

Correctness: We will now prove that our algorithm returns True if and only if a set of donations can occur.

Proof. Suppose that a set of donations can occur. Thus, we know that there exists a sequence of patient donor pairs such that $c(1,2) = c(2,3) = ... = c(i, i+1) = c(i+1, 1) = 1$. That is, there exists a sequence of patient-donor pairs such that for every donor who donates to a patient, their respective patient also receives a transplant. Note that by

[1] This question will not be used for grades, but try it if you're interested. It may be used for recommendations or TF hiring.

our definition of our graph $G = (V, E)$, this condition is equivalent to G containing a cycle. By the definition of a cycle, G contains a cycle if and only if there exists a back edge $(u, v) \in E$. In addition, by Claim 2 from Lecture Notes 4, there exists a $(u, v) \in E$ such that postorder(u) < postorder(v) if and only if (u, v) is a back edge. Recall that our algorithm runs DFS on G and iterates through all $(u, v) \in E$ while checking the condition of Claim 2. Thus, by the correctness of Claim 2 proved in Lecture Notes 4 and the correctness of DFS, our algorithm will return True if there exists a cycle in G.

Now suppose that our algorithm returns True. This implies that there exists a directed edge in $(u, v) \in E$ such that postorder(u) < postorder(v). By the correctness of Claim 2 of Lecture Notes 4 and DFS, we know that there must exist a cycle in G. Thus, there exists a series of vertices $v_1 ... v_j$ such that $(v_1, v_2) ... (v_{j-1}, v_j) \in E$. Note that by our construction of G, this is equivalent to the statement that there exists a sequence of patient-donor pairs such that for every donor who donates to a patient. Therefore, if ouyr algorithm returns True, then there is a sequence of patient-donor pairs such that a set of donations can occur. □

Runtime:

Proof. Let $n = |V|$ and $m = |E|$. Therefore, construction of G takes $O(n + m)$ runtime. Furthermore, our algorithm runs DFS on G which takes $O(n + m)$ time (by Lecture Notes 4), with the assumption that postvisit and previsit take $O(1)$ time. Finally, we loop through all the edges and to check the postorder of the two endpoints of each edge. This step in our algorithm takes $O(m)$ time since we loop through all edges $(u, v) \in E$. Therefore, our algorithm takes a total of $O((n+m)+(n+m)+m)$ time which is $O(n + m)$. □

(b) **(20 points)** Suppose that no set of donations can occur in the previous part, but we add 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.

Algorithm: We will define our algorithm to find the highest value donation sequence as such:

Construct a directed graph $G = (V, E)$ such that $V = \{(p_1, d_1), ...(p_i, d_i)\} \cup \{d_0\}$, where (p_i, d_i) denotes the i-th patient-donor pair. For all pairs of patient-donor pairs (i, j) such that $c(i, j) = 1$, we construct a directed edge $(q_i, q_j) \in E$ (q_i denotes the patient-donor pair (p_i, d_i)). That is, if d_i can donate a kidney to p_j, then there exists edge $(q_i, q_j) \in E$. Moreover, let us construct edges $(d_0, q_i) \in E$ for all $q_i \in V \setminus \{d_0\}$, since d_0 represents an unconditionally willing donor. Finally, let us define a length function such that for all edges $(q_i, q_j) \in E$, $l(q_i, q_j) = -v_{ij}$.

Then, run Bellman-Ford on G starting at vertex d_0. Next, return the shortest path given by Bellman-Ford which starts from d_0.

Correctness: We will now prove that our algorithm returns the highest possible value donation sequence.

Proof. Let the sequence $S = \{(p_0, d_0), ..., (p_i, d_i)\}$ be the sequence of valid donations such that the sum of all values $v_{i,i+1}$ of donations between $((p_i, d_i), (p_{i+1}, d_{i+1})) \in S$ is maximized. Note that maximizing the sum:

$$\sum_{i=0}^{|S|-1} v_{i,i+1}$$

Is equivalent to minimizing the sum:

$$\sum_{i=0}^{|S|-1} -v_{i,i+1}$$

Recall that our graph G is defined such that $V = \{(p_1, d_1), ...(p_i, d_i)\} \cup \{d_0\}$, $(q_i, q_j) \in E$ if donor i can donate to patient j, and edge length is $l(q_i, q_j) = -v_{ij}$. Therefore, if we find a sequence of vertices $\{q_0, ..., q_i\} \in V$ with the shortest path of G, then we have found a sequence of valid donations such that the sum of all values $v_{i,i+1}$ of donations between $((p_i, d_i), (p_{i+1}, d_{i+1})) \in S$ is maximized.

Next, we will prove that a path in G is a valid donation sequence if and only if it begins at d_0. In the problem description, we assume that no set of donations can occur before we added donor d_0. Therefore, any valid sequence of donations in G must include d_0, otherwise there would be a possible donation sequence before we added d_0. In the other direction, every path starting at d_0 is a valid donation sequence since every patient-donor pair receives a kidney except d_0, who is not bound by a patient. In addition, it follows from part (a) that no cycles exist in G prior to adding d_0. Thus, since d_0 is not bound to a patient, it follows that d_0 must have only out-neighbors and there are no cycles in G even with d_0 added as a vertex. Therefore, d_0 must be the starting vertex for any path it is a part of, including the shortest path in G.

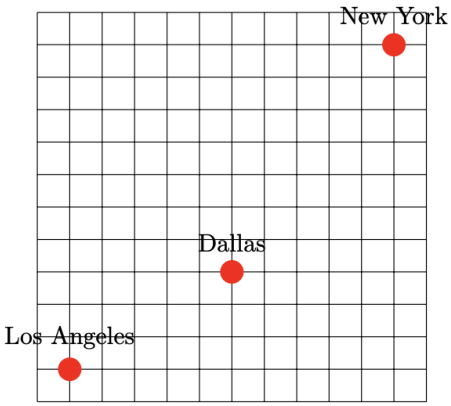
Next, our algorithm runs Bellman-Ford starting at d_0 and returns the shortest path. Since we proved that a path in G is a valid donation sequence if and only if it begins at d_0 in the previous part, it follows by the correctness of Bellman-Ford and paragraphs 1 and 2 that our algorithm outputs the highest possible value donation sequence. □

Runtime:

Proof. Let $n = |V|$ and $m = |E|$. Our construction of graph $G = (V, E)$ takes $O(n + m)$ time because we create n different nodes and m edges between nodes. Next, our algorithm's edge length function multiplies all $v_{i,j}$ by -1 to construct our edge lengths for

G. Thus, assuming that multiplication is $O(1)$ running time, this step in our algorithm takes $O(m)$ running time. Finally, our algorithm runs Bellman-Ford on G starting at d_0. This step in our algorithm takes $O(mn)$ time from Lecture Notes 5. Thus, our algorithm takes a total of $O(n+m+m+mn)$ time which is $O(mn)$. □

5. 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. Assume that the United States can be modeled as an n by n grid, as below.



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.

Answer: $O(n^4)$
For an asymptotic upper bound to be the best upper bound possible, we must find a bound B such that the B is an asymptotically tight bound around the number of possible positions there are for the pair of Iron-men. Thus, we will prove that $B = O(n^4)$ is the best possible upper bound.

Proof. Let us first prove the statement $B = O(n^4)$. Since each Ironman IM1 or IM2 can occupy n^2 possible spaces on an $n \times n$ board, then theoretically, the maximum permutations of locations of IM1 and IM2 is $n^2 * n^2 = n^4$. Therefore, B can be at most n^4, or equivalently $B = O(n^4)$.

Now we will prove that there is no other better bound for B. We define the position of IM1 on the n by n grid as the coordinates (x_1, y_1). Similarly, we define the position of IM2 on the n by n grid as the coordinates (x_2, y_2). Now, let $A = (x_1, y_2)$ and $B = (x_2, y_1)$. Since we are given that x-directional moves (L,R) from IM1 cause IM2 to move only in the y-direction (U,D) and y-directional moves (U,D) from IM1 cause IM2 to move only in the y-direction (L,R), points A and B move independently of each other since changes in x_1 or y_2 cause no change in x_2 or y_1. Next, we will consider a $n \times n$ grid devoid of obstacles in which IM1 starts at $(0,0)$ and IM2 starts at $(0,n)$.

Therefore, A must be located at $(0,n)$ and B must be located at $(0,0)$. Now consider moving IM1 1 time to the left. Since IM1 is at x coordinate 0, it will not move. However, this corresponds to the y coordinate of IM2 to go down 1. Thus, A will end at $(0,n-1)$. Now we move IM1 to the right 1 time. This will cause A to move diagonally up to the right 1 times, ending at $(1,n)$. If we then move IM1 left 2 times (which causes IM2 to go down 2 times), it follows by the same logic that A ends at $(0,n-2)$. Then, we move IM1 to the right 2 times, causing IM2 to go up 2 times, causing A to go diagonally up-right to $(2,n)$. By repeating this process from $1...n$, we are essentially having A traverse through the diagonals of the upper left triangle. Thus, it follows that A can reach any point of the grids top left triangle through diagonal traversals. Note that this is exactly $\frac{n(n+1)}{2}$ positions.

By similar logic, B is located at $(0,0)$. Now consider moving IM1 1 time downwards. Since IM1 is at y coordinate 0, it will not move. However, this corresponds to the x coordinate of IM2 to go right 1 time. Thus, A will end at $(1,0)$. Now we move IM1 up 1 time. This will cause A to move diagonally up to the left 1 times, ending at $(0,1)$. If we then move IM1 down 2 times (which causes IM2 to go right 2 times), it follows by the same logic that A ends at $(2,0)$. Then, we move IM1 up 2 times, causing IM2 to go left 2 times, causing B to go diagonally up-right to $(0,2)$. By repeating this process from $1...n$, we are essentially having B traverse through the diagonals of the lower right triangle. Thus, it follows that B can reach any point of the grids lower right triangle through diagonal traversals. Note that this is also exactly $\frac{n(n+1)}{2}$ positions.

Since we have shown that A and B move independently of one another, this means the total possible positions of A and B is $\frac{n(n+1)}{2} * \frac{n(n+1)}{2}$, which is $O(n^4)$. Moreover, as each position of A and B identify a unique position of IM1 and IM2, then we have shown that the best asymptotic upper bound for the positions of IM1 and IM2 is $O(n^4)$. □

(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.

Algorithm: Let us define our algorithm for finding the cheapest sequence of moves that will bring both Iron-Men home to New York as such:

Construct a graph $G = (V, E)$ such that V is the set of all possible game states of our Ironman problem. That is, every $v \in V$ represents a unique permutation of the locations of IM1 and IM2. Furthermore, let us define E such that edge $(v_i, v_j) \in E$ if it is possible to get to game state represented by v_j from the game state represented by v_i in one move. Therefore, edge e denotes a single {L, R, U, D} move. Finally, let edge length function $l(v_i, v_j) = f_{i,j}$ where $f_{i,j}$ is the total fuel cost of 1 move of IM1 and IM2 from game state v_i to game state v_j (however, IM1 or IM2 will not always cost fuel to move between game states if one of them hits an obstacle or reaches New York).

Next, run Dijkstras algorithm on G starting from the initial game state v_0. We return the shortest path from v_0 to v_w where v_w denotes the game state with both Iron-men in New York.

Correctness: We will now prove that our algorithm returns the cheapest sequence of {L, R, U, D} moves that will bring both Iron-men home to New York.

Proof. Recall that we defined $f_{i,j}$ as the total fuel cost of IM1 and IM2 for one {L, R, U, D}. Thus, finding the cheapest sequence of moves that bring both Iron-men home is equivalent to minmizing the sum:

$$\sum_{i=0}^{w-1} f_{i,i+1}$$

Now recall that we defined our graph $G = (V, E)$ such that V is the set of all unique game states, edge $(v_i, v_j) \in E$ if we can reach game state v_j from game state v_i in one move, and $l(v_i, v_j) = f_{i,j}$. Thus, it is clear that minimizing the sum of total fuel costs is equivalent to finding the shortest path from the initial game state v_0 to the end game state v_w.

Since our algorithm runs Dijkstras on G starting at v_0, it follows from the correctness of G and Dijkstras that our algorithm will return a shortest path from v_0 to v_w that represents the cheapest sequence of moves that will bring both Iron-men home (since every edge e denotes a single {L, R, U, D} move). □

Runtime:

Proof. First, recall that in part (a) we proved that $O(n^4)$ is the best asymptotic upper bound for the possible positions of the two Iron-men. Therefore, when we construct of $G = (V, E)$, we make $O(n^4)$ vertices for $O(n^4)$ possible unique game states. Furthermore, if we naively consider every possible pair of $\binom{n^4}{2}$ game states to determine if there is an edge between the pair, the construction of E takes $O(n^8)$ (with constant runtime for checking if an edge can exist). Finally, our algorithm runs Dijkstras on G. Since every $O(n^4)$ vertex can have at most 4 edges (because there are at most 4

possible moves in every game state), our runtime for Dijkstras is $O(n^4 \log(n^4) + 4n^4)$ with the Fibonacci heap implementation. Thus, the total runtime for our algorithm is $O(n^8 + n^4 + n^4 \log(n^4) + 4n^4)$ which is $O(n^8)$. □

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 $O\left(n^8\right)$ algorithm, but an $O\left(n^4 \log n\right)$ algorithm may be eligible for an exceptional score.

(Note: You may find the question above more accessible after the Monday (Feb. 5) lecture.)

6. **(0 points, optional)** This problem 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:

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

A satisfying assignment 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 satisfying assignment to a 2SAT formula, or returns that no satisfying assignment 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 $\bar{\alpha}$ to β and a second directed edge from $\bar{\beta}$ to α. How can this be interpreted?)