# CS 124 Homework 2: Spring 2024

**Your name:** Anusha Murali

**Collaborators:** None

**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.

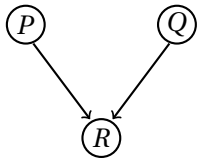
For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential-time algorithms (unless specifically 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.

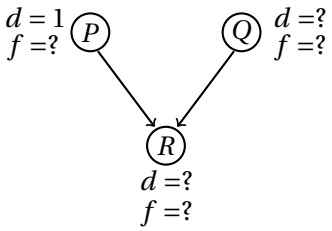
**Solution**

Consider three tasks, $P, Q$, and $R$, where $P$ and $Q$ must be completed before $R$. Following directed graph for this task assignment is a counterexample showing that topological sort by ordering in increasing order does not work.

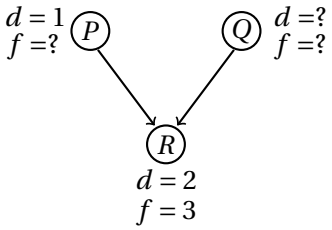



Let $d$ denote the time when the node was discovered (i.e: $d$ is the preorder time) during a DFS search and $f$ denote the time when the DFS search was finished on that node ((i.e: $f$ is the postorder time).

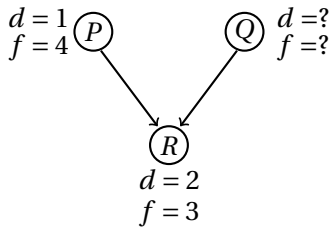
Let us say that the DFS search begins at $P$. So, we have the following:



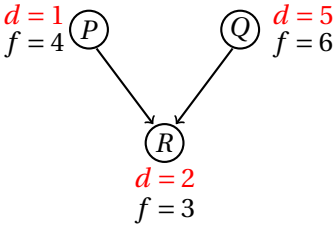

We follow the directed edge to discover node $R$ at $t = 2$. Since $R$ doesn't have any outgoing edges, we mark it as finished at $t = 3$. So, we have the following:




We now backtrack to $P$, which is the parent of $R$. Since $P$ doesn't have any more children, we mark $P$ as finished at $t = 4$. So, we have the following:




Restarting DFS, we now discover node $Q$ at time $t = 5$. Since $Q$'s only child, $R$, has already been explored, we mark $Q$ as finished at time $t = 6$. So, we have the following final discovered/finished timings for our DFS traversal:



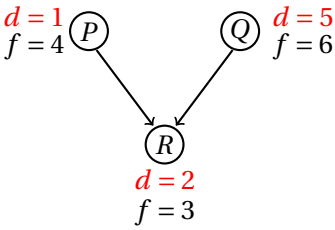

Therefore, the increasing topological order using the preorder time (shown in red) is $1, 2, 5$, which corresponds to completing the tasks in the order of $P, R$ and $Q$. According to the constraints in the above directed graph, both $P$ and $Q$ must be completed before $R$. However, using the increasing topological order using the preorder time shows that task $R$ will be completed before $Q$, which is invalid.

Therefore, a topological sort by ordering in increasing order according to the preorder time does not work. □

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

**Solution**

The counterexample that we used in Part (a) can be used to show that a topological sort by decreasing preorder time does not work.




The preorder times are in red in the above DFS traversal diagram. The decreasing preorder times are $d = 5, d = 2$, and $d = 1$, which corresponds to completing the tasks in the order of $Q, R$, and $P$. However, according to the constraints in the above directed graph, both $P$ and $Q$ must be completed before $R$.

Therefore, a topological sort by ordering in decreasing order according to the preorder time 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.)

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

   **Solution**

   We want to find if there is a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that

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

   Representing each currency as a vertex, and the exchange rate between two currencies as a directed edge between the vertices, we can construct a directed graph to represent our problem. Since our requirement for the “risk-free currency exchange" is that we need to finally convert back to the original currency, we are looking for a cycle in our directed graph. Therefore, if we can transform our problem involving the products of exchange rates into a summation problem on edge lengths, we could use a suitable shortest path finding algorithm for solving it.

   Taking logarithm on both sides (in base 2), we obtain,

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

   where each term on the LHS can represent an edge between two vertices (currencies). In order to formulate the above problem as a shortest path finding problem, we can negate each path weight. So, we obtain,

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

   Therefore, in order to solve our “risk-free currency exchange" problem, we can use the Bellman-Ford algorithm, which can detect any negative cycles.

   Hence, our directed graph can be constructed as follows:

   (a) Vertices: $N$ Currencies.

   (b) Edge length between vertex $i$ and $j$: Negative of the logarithm of the currency exchange between currency $i$ and currency $j$, if one exists.

We will run the single-source Bellman-Ford algorithm on the above directed graph with the initial currency as the source node. With minor modifications, we call our algorithm, **RiskFreeCurrencyFound()**, which will return True, if there is a negative cycle in the graph and False otherwise. We note that a cycle does not necessarily have to involve the source node, but can be present anywhere in the path originating from the source node.

The pseudocode of the algorithm, **RiskFreeCurrencyFound(G, s)** is listed below. It uses three helper functions, namely,

(a) **Initialize()** : Initializes all the $|V|$ vertices as being infinite distance from the source node. The distance on the source node is initialized to 0, as source node is at 0 distance from itself.

(b) **Relax()**: This function relaxes the distance values at each node by comparing the current distance value to the sum of the distance value at the predecessor node and the edge weight between the predecessor and itself.

(c) **Check-for-Negative-Cycle()**: Once the Bellman-Ford algorithm has iterated through all the edges $|V|-1$ times and relaxed all the node distances, one more check on any two adjacent vertices $u, v$ that results in $v.d > u.d + w(u,v)$ indicates that distance is decreasing. This indicates the presence of a negative cycle. As soon as we encounter the first such negative cycle, **Check-for-Negative-Cycle()** returns True.

**Algorithm 1** Initialize(G, s)

```
for each vertex v ∈ G.V do
    v.d = ∞                          {Initialize all distance to the source node (initial currency) to ∞}
v.s = 0                              {Initialize the distance to the source node to 0}
```

**Algorithm 2** Relax($u, v, w$)

```
if v.d > u.d + w(u,v) then
    v.d = u.d + w(u,v)               {If a smaller distance found, update v.d}
```

**Algorithm 3** Check-for-Negative-Cycle(G)

```
for each edge (u,v) ∈ G.E do
    if v.d > u.d + w(u,v) then
        return True                  {Return True as soon as a negative cycle in G is found}
    return False
```

**Algorithm 4** RiskFreeCurrencyFound(G, s)

```
1: Initialize(G, s)
2: for i = 1 to |G.V| − 1 do
3:     for each edge (u,v) ∈ G.E do
4:         Relax(u,v,w)
5: return Check-for-Negative-Cycle(G)   {Return True if there is at least once negative cycle in G}
```

**Proof of the algorithm**

We claim that **RiskFreeCurrencyFound()** returns True if there is a sequence of risk free currency exchanges found, and False otherwise. In other words, we claim that our algorithm finds if there is a negative cycle reachable from the source $s$, then for some edge $(u, v)$, $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.

Suppose $v_0 \to v_1 \to \cdots \to v_k$ is a negative cycle reachable from $s$, where $v_0 = v_k$. This means,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.$$

We prove this using contradiction. So, assume that $d_{n-1}(v_i) \leq d_{n-1}(v_{i-1}) + w(v_{i-1}, v_i)$ for $1 \leq i \leq k$. When we set up this inequality for each vertex on the cycle and sum them up, we obtain,

$$\sum_{i=1}^{k} d_{n-1}(v_i) \leq \sum_{i=1}^{k} d_{n-1}(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Since $v_0 = v_k$, the first two terms are the same. Therefore,, we cancel them out and obtain,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0,$$

which contradicts our initial assumption that $v_0 \to v_1 \to \cdots \to v_k$ is a negative cycle. This implies that if there is a negative cycle, **Check-for-Negative-Cycle()** called on line number 5 of our algorithm **RiskFreeCurrencyFound()** returns True.

The vertices in our graph corresponds to currencies and the weights of the edges in our graph correspond to the negative logarithms of the currency exchange rate. For example, if the vertex $u$ represents one currency (say, US dollar) and the vertex $v$ represents another currency (say, Japanese Yen), the edge $(u, v)$ has a weight $w = -\log(r_{u,v})$, which is the negative logarithm of the exchange rate between the currency $u$ and currency $v$. Therefore, when **RiskFreeCurrencyFound()** returns True, since we have already updated the distances in all the vertices in line numbers 2 - 5 of **RiskFreeCurrencyFound()**, it must be that there is a negative cycle in our graph. Therefore,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0,$$

which implies

$$\begin{aligned} \sum_{i=1}^{k} \log(r_{i-1,i}) &> 0 \\ \log(r_{i_1,i_2}) + \log(r_{i_2,i_3}) + \ldots + \log(r_{i_{k-1},i_k}) + \log(r_{i_k,i_1}) &> 0 \\ r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} &> 1, \end{aligned}$$

confirming that the presence of a risk free exchange results in a negative cycle. □

**Runtime of the algorithm**

On line number 1 of **RiskFreeCurrencyFound()**, we call **Initialize()** to set the initial distances of all the vertices. Therefore, this call requires $O(|V|)$ runtime.

On line numbers 2 and 3, we loop through the edges $|V|-1$ times, each time evaluating the condition for relaxing and updating the distance, $v.d$. Therefore, these two lines require $(|V|-1)O(|E|) = O(|V|\,|E|)$ runtime.

Finally on line number 5, we call **Check-for-Negative-Cycles()**, which in the worst case will iterate through every edge in the graph. Therefore, this requires $O(|E|)$ runtime.

Hence, the runtime of our algorithm to detect a negative cycle is $O(|V|+|V|\,|E|+|E|) = O(|V|\,|E|)$, where $|V|$ is the number of vertices (number of currencies) and $|E|$ is the number of edges (number of distinct currency exchanges). □

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 $G$.

   **Solution**

   In this version of the game, $c = k = 1$. So there is only one pebble, which, honoring the rule of the game, can move to an immediately adjacent vertex in the connected graph. We will use a modified version of the Breadth First Search (BFS) on the undirected connected graph $G$. The problem is asking for a winning strategy that traverses each of the $m$ edges, in both directions, for a total of exactly $2m$ moves.

**Algorithm 5** BFS_Pebble1($G, s$)

```
1:  Q.empty()                                   {Initialize an empty queue}
2:  Explored[·] = False                         {Mark all n vertices in G as not explored}
3:  Traversed[·] = null                         {Initially the traversed edges list is empty}
4:  Q.append(s)                                 {Push s into the queue}
5:  Explored[s] = True
6:  while Q is not empty do
7:      v = Q.pop()
8:      for all edges from v to w in G.adjacentEdges(v) do
9:          Traversed.append((v, w))            {Add edge (v, w) to list as it is now traversed}
10:         if Explored[w] == False then
11:             Q.append(w)                     {Push vertex w into the queue}
12:             Explored[w] = True
13: return Traversed[·]                         {Return the traversed edges in the order they were traversed}
```

Consider an arbitrary edge $e = (u, v)$ in $G$. If we travel from vertex $u$ to $v$ and then again from $v$ to $u$, we would have travelled the edge $e$ twice. In our version of the BFS, for every vertex $v$, we traverse all of its edges $(v, w)$ to reach the vertex $w$. If vertex $w$ is not already explored, then we push it to the queue and mark that vertex as “explored". A vertex that is already marked as “explored" will not be pushed to the queue again. Since our algorithm is traversing every edge emanating from a vertex, for any two adjacent vertices $u$ and $v$, both edges $(u, v)$ and $(v, u)$ will be traversed twice. This is because on line numbers 8 and 9, we are traversing the edges emanating from a vertex, instead of visiting a vertex as in the traditional BFS traversal. This is

the main difference between our traversal algorithm and the traditional BFS algorithm. The pseudocode of the algorithm is given above. The starting vertex $s$ can be any one of the $n$ vertices in $G$. Initially, the list **Traversed[]** is empty and as each edge $(w, w)$ is visited, it is added to this list (line number 9). Since an edge $(v, w)$ is visited twice, once from $v$ and again from $w$, when the algorithm terminates this list will contain $2m$ entries (two entries for each edge in both directions) in the graph. Note that we could also confirm this using a **print** statement on line number 9, instead.

**Correctness of BFS traversal**: We claim that the algorithm **BFS_Pebble1()** returns a winning strategy by outputting the $2m$ edges in the order they were traversed. We can prove this using contradiction. We claim that every edge in the connected graph $G$ is eventually traversed twice. Suppose this claim is not correct. In other words, there exists an unvisited vertex $w$ whose neighbor $v$ was visited. When $v$ was visited, the edge $(v, w)$ was traversed (in line number 9) and its neighbor $w$ was appended to the queue (line number 11) to be processed. The order of edge traversal is preserved as we append the traversed edge to the **Traversed[]** list immediately upon its traversal (line number 9). When this neighbor $w$ is later popped from the queue (line number 7) and its edges are processed, the previously traversed edge $(v, w)$ will be now traversed in the opposite direction - i.e: the edge $(w, v)$ will be now traversed. This is true for all the neighbors $w$ of $v$ (line number 8). The algorithm runs until the queue becomes empty (line number 6). Since $w$ is a neighbor of $v$ and $w$ was pushed into the queue, it must be eventually popped and visited before the algorithm terminates. It follows that every edge in the connected graph $G$ is eventually traversed in both directions. Since there are $m$ edges, there will be exactly $2m$ such traversals, which correspond to the entries in the **Traversed[]** list. □

**Runtime**: Using an adjacency list representation, the above **BFS_Pebble1** algorithm runs in $O(m + n)$ time. This is because, every vertex and every edge (in both directions) will be explored by the above algorithm once.

(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 $O(n^{2k})$.

**Solution**: We solve this problem by viewing the configuration of the graph $G$ after each move. Let us define a “state", which is essentially a configuration of the graph $G$. The state contains information on the $n$ vertices of $G$ that have pebbles on them and the number of those pebbles on each of those $n$ vertices. Specifically, a state is an ordered set (ordered by the vertex numbers), such that each member of the set is an ordered pair with the first element being the vertex number and the second element being the number of pebbles on it.

Two states $S_i$ and $S_j$ are adjacent or neighboring, if moving one pebble from one vertex to another vertex in the first configuration $S_i$ could lead to the second configuration $S_j$ , and vice versa. We note that each move is subject to the constraints defined by $c, k, n$ in the problem statement.

Going from one state to another would mean updating the number of pebbles on the appro-

priate vertices. The start-state is the one specified in the problem statement: the $k$ pebbles are placed on vertices $0, 1, \ldots, k-1$. And a target-state is one where the target vertex $t$ has a pebble on it. The problem statement describes the target state $t$ as a special designated vertex, but it is possible to have multiple such target states.

Our goal is to perform a Breadth First Search (BFS) on the graph of states induced on $G$, with the aforementioned start-state and target-states. Note that this graph of states may be disconnected due to the constraints imposed by the problem based on $c, k, n$. In other words, if all the target-states lie in some components separate from that of the start-state, then there is no winning strategy. Otherwise, there exists at least one winning strategy.

The pseudocode of the algorithm is given below.

**Algorithm 6** BFS_Pebble2(G, s)

```
1:  D ← []                                                  {D is a hashmap}
2:  s ← {(v_0, 1), (v_1, 1), ..., (v_{k-1}, 1)}             {s is the start-state}
3:  Q.append(s)                                             {Push s into the queue}
4:  while Q is not empty do
5:      u = Q.pop()                                         {Pop the state in the front}
6:      if (t, ℓ) ∈ u, where ℓ > 0 then
7:          return True.                                    {Winning strategy found}
8:      for each neighboring state v of u do
9:          if D[v] =False then
10:             Q.append(v)                                 {v ∉ D}
11:             D[v] = True
12: return False                                            {No winning strategy found}
```

As can be seen from the algorithm above, we only create the state-graph as needed. In other words, we don't create the entire state-graph at the beginning, but only create its nodes as we progress with the algorithm. If we found a winning strategy by moving to the target vertex $t$ (line number 7), we will immediately exit the algorithm by returning True. We also note that the above algorithm could be modified slightly to return the actual winning strategy, if needed.

**Correctness**:

As far as the correctness is concerned, we need to prove two claims, namely,

(1) that our construction is correct, i.e., there exists a target-state that lies in the same component as that of the start-state if and only if there exists a winning strategy, and

(2) that our algorithm correctly finds a winning strategy, given part (1) is true.

Since Part (2) follows from the correctness of BFS, we only need to prove Part (1).

If there exists a winning strategy, there must be at least one final configuration that one could reach from the starting configuration through a series of configurations or moves, with respect to the constraints specified in the problem. Our state-graph is created, such that we connect two states when there is a legitimate move possible between the two states, so the sequence of moves from the starting state to the final state must exist in our graph (via strong induction on the states in the path sequence). For the other direction, if a path exists in the

state graph (which was constructed based on legitimate moves), that path would just correspond to an actual strategy, otherwise at least one connection in the graph must be illegitimate. This proves Part (1).

**Runtime**: In order to determine the running time of the algorithm, we study each iteration of the While-loop on line number 4. We note that there are a maximum of $N = n^k$ (actually $\left\lceil \frac{n^k}{k!} \right\rceil$) states possible in the state-graph for $G$. This is because each pebble has at most $n$ choices to be on, and there are $k$ *identical* pebbles. Therefore, there can be at most $N$ iterations of the algorithm (by the correctness of BFS). In each iteration, we first pop out the state $u$ in the front, and check in at most $O(k)$ time if the target vertex $t$ is in there or not. If not, we then examine its neighbors. For that, we first determine the set of vertices $S$ in $G$ that already have at least $c$ adjacent vertices with pebbles on them in that configuration, which could potentially be $n$. Checking for that could take $O(n^2)$ time. For each vertex $a$ in $u$, we remove one pebble from $a$ and put it on every vertex in $S$ to create $|S|$ new states (which could potentially be $n$). Therefore, we generate a total of $kn$ possible new states. Then we check if each state is in the map in $O(1)$ time before adding it to $Q$. This process takes $O(k^2 n)$ time. Hence, the total running time is

$$O\left((k^2 n + n^2) \cdot n^k\right) = O\left((k^2 + n) n^{k+1}\right), \text{ which is at most } O(n^{2k}) \text{ as desired.} \quad \square$$

(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!

In Part (b), we presented an algorithm which has a runtime of $O\left((k^2 + n) n^{k+1}\right)$. The exponent of $n$ could be improved further by a more careful implementation, but for $n, k$ larger than some absolute constants, we note that this is in $o(n^{2k})$.

[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.

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:

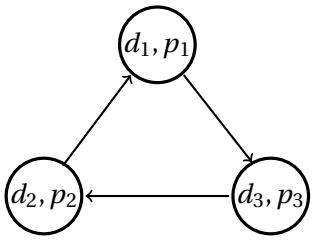
$$(p_1, d_1), (p_2, d_2), (p_3, d_3), (p_4, d_4), (p_5, d_5).$$

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.

**Solution**

Create a directed graph where each vertex is a patient-donor pair $(p_i, d_i)$. For example, following is a such graph on three patient-donor pairs. The direction of the edges show which donor is donating to which patient. In this example, $d_1$ is donating to $p_3$, $d_3$ is donating to $p_2$ and $d_2$ is donating to $p_1$.




**Algorithm**: Run DFS on the above graph starting at any vertex until it finds a back edge or terminates without finding one. If it finds a back edge, then we have detected a cycle, which means there is at least one patient-donor pair that can successfully exchange their kidneys. Since our graph can contain multiple connected components, we need to ensure that we run DFS on all the connected components until we find a cycle on one of them. This is achieved by the function **CanDonationsOccur(G)** given below. This function returns true if a set of donations can occur, and returns false otherwise. The function **CanDonationsOccur()** in turn calls **IsAcyclicDFS(G, v)**, which runs the actual DFS recursively on a given connected component. (Please note that **IsAcyclicDFS()** was based on the algorithms from Erickson's Algorithm, Section 6.2: Detecting Cycles, Page 231).

**Algorithm 7** CanDonationsOccur(G)

```
for all vertices v do
    v.status ← NEW
for all vertices v do
    if v.status = NEW then
        if IsAcyclicDFS(G, v) = False then
            return True                                  {There is a cycle in G}
return False
```

**Algorithm 8** IsAcyclicDFS($G, v$)

```
1: v.status ← ACTIVE
2: for each edge v → w do
3:     if w.status = ACTIVE then
4:         return False                                  {Found a back edge}
5:     else if w.status = NEW then
6:         if IsAcyclicDFS(w) = False then
7:             return False
8:     v.status ← FINISHED
9: return True                                           {No cycles found}
```

**Correctness**: The algorithm **CanDonationsOccur()** returns true if and only if a cycle exists in one of the connected components of $G$. If a cycle exists, when **IsAcyclicDFS()** is called on some vertex $v$, one of its descendants would have a back edge pointing at $v$. We note that since **IsAcyclicDFS()** marks each vertex as ACTIVE on line number 1, if one of its descendants is found to be ACTIVE (line number 3), then we have found a back edge - so there is a cycle. On the other hand, if **CanDonationsOccur()** returns true, then **IsAcyclicDFS()** should have returned false. This happens on line number 4 or on a recursive call on line number 6 upon encountering a vertex $w$ with the status = ACTIVE, implying there was a cycle in $G$. The presence of cycle means that there was a back edge in $G$. □

**Runtime**: Using an adjacency list, the DFS algorithm to detect a cycle takes a runtime of $O(V+E)$, where $V$ and $E$ are the number of vertices and number of edges in the graph respectively. In addition, we need $O(V)$ space to maintain an array to remember the visited vertices. □

(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.

**Solution**

We pair the altruistic donor $d_0$ with every patient, $p_i, 1 \le i \le n$, one at a time and identify the highest value donation sequence. Since the problem does not say that the donation value $v_{ij}$

cannot be negative, we assume that one or more edge weights can be negative. Therefore, we use the Bellman-Ford algorithm for our purpose. We also know that, starting with the altruistic donor $d_0$, there will be no cycle in the highest value donation sequence (because the last donor-patient pair in the sequence will not be donating to the altruistic donor). Hence our problem can be solved by finding the highest value donation sequence (that may possibly include some negative donation values) in a directed acyclic graph.

Bellman-Ford algorithm can be used to solve the single-source shortest path from the altruistic donor vertex to every other vertex in $G$. Since we are looking for the highest value donation sequence, we can negate the donation values representing the edge weight and find the shortest path - which will actually be the highest value donation sequence.

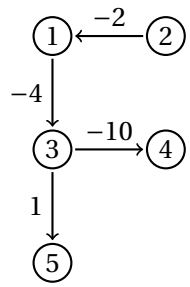
We assume that there are $n$ donor-patient pairs, which are labeled $(d_1, p_1), (d_2, p_2), \ldots, (d_n, p_n)$. As in the case of Part (a), each of these $n$ donor-patient pairs corresponds to a vertex in our directed graph, $G$. The graph $G$ contains one more vertex, which corresponds to the altruistic donor. Since the altruistic donor is not specifically associated with any donor, we label it $(d_0, -)$. It is given that (without the help of the altruistic donor) no set of donations can occur. Therefore, there are no cycles in our graph, and consequently $G$ is a DAG with $n+1$ vertices. The number of edges $m$ is determined by the output of the function $c(i, j)$, as stated in the problem. We note that $c(0, j) = 1$ for every patient, $p_j, 1 \leq j \leq n$, as the altruistic donor could potentially donate her kidney to any one of the $n$ patients. When the altruistic donor donates her kidney to a patient $p_j$, we assume that the donation is compatible.

The **high-level pseudocode** of our approach is given below, which is described in detail later.

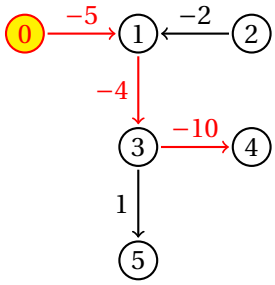
**Algorithm 9** High level pseudocode

```
for each patient p_i do
    Add vertex (d_0, −) to G with a directed edge to p_i      {Donate altruistic donor's (d_0) kidney to p_i}
    Negate the donation values and add them as edge weights
    Run single source shortest distance Bellman-Ford on G and save output to d[·]
    Save the maximum shortest distance sequence of p_i in highval_seq_array[i]
    Delete vertex (d_0, −) from G
return The sequence with the largest value from highval_seq_array[·]
```

The following example is used to illustrate the pseudocode. Assume that we have 5 donor-patient pairs $(d_1, p_1), (d_2, p_2), (d_3, p_3), (d_4, p_4), (d_5, p_5)$. We label them as $1, 2, 3, 4$ and 5. The $c(i, j)$ values are $c(1,3) = c(2,1) = c(3,4) = c(3,5) = 1$ and all other $c(i, j) = 0$. The respective donation values are $v_{2,1} = 2, v_{1,3} = 4, v_{3,4} = 10$ and $v_{3,5} = -1$. **We use the negative of the donation values as edge weights**, so we can find the maximum value sequence by using the shortest path algorithm. Hence our DAG for the above 5 donor-patient pairs is as follows:

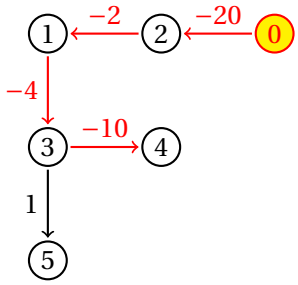



We run our algorithm 5 times, each time adding the vertex corresponding to the altruistic donor $(d_0, -)$ with a directed edge to the respective patient. Let $v_{0,1} = 5, v_{0,2} = 20, v_{0,3} = -3, v_{0,4} = 7$ and $v_{0,5} = 3$ be the donation values from the altruistic donor to the respective patients. Our modified Bellman-Ford algorithm returns the maximum value sequence for each of the 5 patients. I have shown below the modified DAGs and the resulting highest value sequences (in red) for $p_1$ and $p_2$ (i.e for vertices 1 and 2):




Maximum value sequence for $p_1$ =

$d_0 \rightarrow (d_1, p_1) \rightarrow (d_3, p_3) \rightarrow (d_4, p_4)$




Maximum value sequence for $p_2$ =

$d_0 \rightarrow (d_2, p_2) \rightarrow (d_1, p_1) \rightarrow (d_3, p_3) \rightarrow (d_4, p_4)$

The Bellman-Ford's single-source shortest path algorithm returns the path with the smallest path cost. However, since we used the negatives of the donation values as the edge weights, the output of the Bellman-Ford algorithm would correspond to the highest value sequence. We finally compare the values of the sequences for all 5 runs, and select the one that has the highest absolute value.

For the above example, when the algorithm is run on all 5 patients, we find that when the altruistic donor gives her kidney to patient 2, we find the maximum value for the donor sequence, which is $(d_0 \rightarrow d_2 \rightarrow d_1 \rightarrow d_3 \rightarrow d_4)$, yielding a total value of $|-20-2-4-10| = 36$.

The pseudocode of the more detailed algorithm, which is based on Bellman-Ford's single source shortest path algorithm, is presented below.

Following is the pseudocode for the Bellman-Ford's single source shortest path algorithm. It uses the **Update()** function below to update the edge weights.

**Algorithm 10** Update $(u, v)$

**if** $d[v] > d[u] + \text{weight}((u, v))$ **then**
  $d[v] = d[u] + \text{weight}((u, v))$ {weight is the negative of the donation value}
  $\pi[v] = u$ {$\pi$ contains the previous vertex}

**Algorithm 11** Bellman-Ford ($G = (V, E), s \in V$)

**for** all $v \in V$ **do**
  $d[v] \leftarrow \infty$; $d[s] = 0$; $\pi[s] \leftarrow null$ {Initialize}
**for** $i = 1$ to $n$ **do**
  **for** every $(u, v) \in E$ **do**
    Update $(u, v)$
**return** $d[\cdots], \pi[\cdots]$

Our distance array is based on the weight of the edge, which is the **negative** value of the donation from the donor associated with vertex $u$ to the patient associated with vertex $v$.

Suppose the altruistic donor is donating to patient $p_i$. The following function, taking $d[\cdot]$ and $\pi[\cdot]$ of patient $p_i$ (which were obtained by running the above Bellman-Ford's single source shortest path algorithm) as inputs, returns the maximum value sequence for this DAG.

**Algorithm 12** Find_Max_Value_Sequence ($d[\cdots], \pi[\cdots]$)

maxval = $-\infty$
max_value_sequence = null
**for** $i = 1$ to $n$ **do**
  curr_seq_value = $\left|\sum_{u,v} \text{weight}(d[\cdot])\right|$. {Sum the edge weights of this path sequence}
  **if** curr_seq_value > maxval **then**
    maxval = curr_seq_value
    max_value_sequence = $d[i]$
**return** Maximum $d[i]$ {Return the largest shortest distance for $p_i$}

Finally, the pseudocode of the main function, which iteratively tries the altruistic donor to each of the $n$ patients, is shown below.

**Algorithm 13** Find_Highest_Value_Sequence

1: Build $G$ {Vertices: $n$ donor-patient $(d_i, p_i)$ pairs, Edge weights: negative donation values}
2: **for** $i = 1$ to $n$ **do**
3:  Add vertex $(d_0, -)$ to $G$ with a directed edge to $p_i$ {Donate altruistic donor's $(d_0)$ kidney to $p_i$}
4:  $d_i[\cdot]$, $\pi_i[\cdot]$ =Bellman-Ford ($G$) {Run Bellman-Ford single-source shortest path algo}
5:  **highval_seq_array[i]** = Find_Max_Value_Sequence ($d_i[\cdot], \pi_i[\cdot]$) {Find the max sequence for $p_i$}
6:  Delete vertex $(d_0, -)$ from $G$ {Delete altruistic donor, so that we can rebuild $G$ for next patient}
7: **return** Highest value sequence among the $n$ sequences of **highval_seq_array[]**

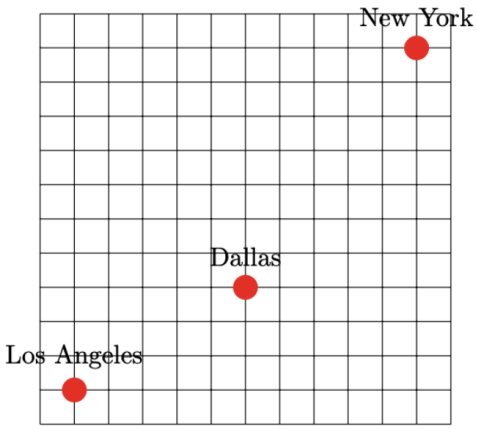
We first build a directed graph $G$ using the $n$ donor-patient $(d_i, p_i)$ pairs as vertices and the corresponding donation values as edge weights. We then add the vertex for the altruistic donor, $(d_0, -)$, to $G$, each time connecting it to a new patient, $p_i, 1 \leq i \leq n$, using a directed edge, where the edge weight is the **negative** of the donation value of the altruistic donor to the respective patient. We then run Bellman-Ford's single-source shortest path algorithm to find the least cost path from vertex $(d_0, -)$ to each of the $n$ patients. The path sequence containing the maximum donation value for patient $p_i$ is saved in **highval_seq_array[i]**. We then delete $(d_0, -)$ from $G$ and repeat the process for the next patient, until the altruistic donor is tried with all $n$ patients.

The highest value sequence among the $n$ sequences of **highval_seq_array[]** is the (donor, patient) sequence that we are looking for. It can be obtained using a function similar to **Find_Max_Value_Sequence()** above.

**Correctness**: By the construction of the problem, we know our graph is a DAG. The reason is as follows. Without the altruistic donor, no set of donations can occur. So we initially start with a DAG. When the altruistic donor is introduced as source vertex $(d_0, -)$, since the altruistic donor does not require a kidney donation, there are no directed edges pointing from a patient to the source vertex $(d_0, -)$. Therefore, our graph is still a DAG. The Bellman-Ford single-source shortest path finding algorithm is a well known algorithm for finding the shortest path from a given source vertex to every other vertex in a DAG, even in the presence of negative edge weights. Therefore, when a DAG and a source vertex are input to the Bellman-Ford single-source shortest path algorithm, it will output the least-cost path sequences from the source vertex to every other vertex. Hence, all what is left to show is that when we negate the edge weights, the shortest path returned by the Bellman-Ford algorithm returns the highest value sequence. Let $x_1, x2, \ldots, x_k \in \mathbb{R}$, with $x_1 < x_2 < \cdots < x_{k-1} < x_k$, where $x_i$ is the cost of a path in $G$. Therefore, $-x_1 > -x_2 > \cdots > -x_{k-1} > -x_k$. Hence, the shortest path returned by the Bellman-Ford algorithm, which used the negatives of the actual donation values as the edge weights, in fact is the highest value sequence. □

**Runtime**: The main algorithm, **Find_Highest_Value_Sequence()** calls (on line number 4) Bellman-Ford single-source shortest path algorithm $n$ times, once for each patient. The runtime of the Bellman-Ford algorithm is $O(mn)$ as we call update on each of the $m$ edges $n-1$ times. We assume the cost of negating the edge weights, adding the altruistic donor vertex (on line number 3) and deleting it again (on line number 6) take constant time. So, for the $n$ iterations they take $O(n)$ time. Therefore, the overall runtime of the main algorithm is $O(mn^2)$. □

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 $(x_\ell, y_\ell)$, 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 we can represent the United States as a $n \times n$ grid, there are $n^2$ coordinates in the grid. Iron-Man 1 can be in any one of the $n^2$ coordinates (excluding the positions of all obstacles) and Iron-Man 2 can be in any one of the $n^2$ coordinates (excluding the positions of all obstacles).

Let $M = \sum(x_{o_i}, y_{o_i})$ the total number of positions of all the obstacles in the $n \times n$ grid.

Therefore, there are a total of $(n^2 - M) \times (n^2 - M) = n^4 - 2Mn^2 + M^2$ possible positions for the pair of Iron-Men in the grid. Since $M < n$, the upper bound on the number of possible positions for the pair of Iron-Men is $O(n^4)$.

The above upper bound is tight. We can show this with an example. We ignore the obstacles in the following argument, however the argument does not change even in the presence of

obstacles. Consider the $13 \times 13$ grid provided in the problem, where IM1 is at (1, 1) in LA and IM2 is at (6, 4) in Dallas. In order to move IM2 down by 1 unit to (6,3), we could move IM1 left 4 times and then right 3 times. This would bring IM2 down to $y$ coordinate 0 and then up to $y$ coordinate 3. Therefore, by using the fact that going against the boundary of the grid an Iron Man cannot actually move, while the other one can, we can see that IM2 can move to all $n^2$ positions (ignoring the obstacles), while IM1 is still in LA. This is true for all $n^2$ positions of IM1. Therefore, when one of the Iron-Men is stuck against the boundary of the grid, the possible positions for the second Iron-Man is $n^2$, leading to the worst-case lower bound of $\Omega(n^4)$. Hence we see that the best asymptotic upper bound for the number of possible positions for the pair of Iron-Men 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. 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.

**Solution** We construct a graph $G'$ consisting of $O(n^4)$ vertices, each of which represents the combined pair of locations of both Iron-Men in the original $n \times n$ grid (after removing the positions of the obstacles, $(x_{o_i}, y_{o_l})$) as they move along the $n \times n$ grid. For example, $v_s = \{(x_\ell, y_\ell), (x_d, y_d)\}$ is a vertex in graph $G'$, which represents the initial positions of IM1 (Los Angeles) and IM2 (Dallas). Similarly, the vertex $v_d = \{(x_{ny}, y_{ny}), (x_{ny}, y_{ny})\}$ represents the final positions of IM1 (New York City) and IM2 ( New York City).

Construction of the edges in $G'$ is as follows. Suppose IM1 is moving from position $u_1 = (x_1, y_1)$ to an adjacent position $u_2 = (x_2, y_2)$, while IM2 is moving from position $v_1 = (x'_1, y'_1)$ to an adjacent position $v_2 = (x'_2, y'_2)$. We note that each Iron Man move in only one unit in one direction (either $x$ or $y$ direction). So, we construct an edge from vertex $(u_1, v_1)$ to vertex $(u_2, v_2)$) in $G'$ with the edge weight being the sum of the weights of those 2 edges in the original $n \times n$ grid.

We also note that the construction of $G'$ consists of a pre-processing step of removing the positions of all obstacles $(x_{o_i}, y_{o_i})$ from the original $n \times n$ grid, before generating $G'$, as neither Iron Man can traverse through an obstacle. Since there are $n^2$ grid points in the original grid, this pre-processing step would require $O(n^2)$ runtime.

The weight of the edges in $G'$ correspond to the fuel costs for the respective moves on the $n \times n$ grid and are non-negative. The computation of the fuel cost between two adjacent vertices in $G'$ costs $O(1)$ time as it only involves addition operation on the fuel costs on 2 edges. Hence we require an $O(n^4)$ time for the computation of fuel costs in $G'$, and this computation will be done only after the removal of all the positions of the obstacles $(x_{o_i}, y_{o_l})$. We note that we don't create the entire $G'$ at the beginning, but only create its nodes and edges as we progress with the algorithm.

Therefore, we have now transformed the given problem of moving both IM1 from Los Angeles and IM2 from Dallas to New York City on the original $n \times n$ grid into a problem of finding the shortest path between the vertices, $v_s = \{(x_\ell, y_\ell), (x_d, y_d)\}$ and $v_d = \{(x_{ny}, y_{ny}), (x_{ny}, y_{ny})\}$ in the weighted graph $G'$ with $n^4$ vertices.

The graph $G'$ contains edges with non-negative weights. Hence we can use Djikstra's algorithm to find the shortest path to $v_d = \{(x_{ny}, y_{ny}), (x_{ny}, y_{ny})\}$ from $v_s = \{(x_\ell, y_\ell), (x_d, y_d)\}$.

**Algorithm 14** Djikstra ($G' = (V, E)$, length: $E \to \mathbb{R}^+$, $s \in V$)

```
1: for all v ∈ V do
2:     d[v] ← ∞
3: d[s] = 0; Prev[s] ← null; Insert(H, s, 0)                    {start at s}
4: while H ≠ ∅ do
5:     u ← deleteMin(H)
6:     for (u, v) ∈ E do
7:         if d[v] > d[u] + length((u, v)) then
8:             d[v] = d[u] + length((u, v))
9:             Prev[v] = u
10:            Change(H, v, d[v])
11: return d[···], Prev[···]
```

**Correctness**: Since there are $n^4$ possible positions of the (IM1, IM2) pair for each of their $n^2$ possible locations on the original $n \times n$ grid, it follows that there is a one-to-one mapping between the two Iron-Men on the original $n \times n$ grid and $G'$. Hence both Iron-Men traveling from a vertex $u = ((x_1, y_1), (x_1', y_1'))$ to another vertex $v = ((x_2, y_2), (x_2', y_2'))$ on graph $G'$ implies that IM1 is traveling from vertex $(x_1, y_1)$ to vertex $(x_2, y_2)$ and that IM2 is traveling from vertex $(x_1', y_1')$ to vertex $(x_2', y_2')$ on the original $n \times n$ grid. Hence the cost of IM1 and IM2 traveling to New York City ($x_{ny}, y_{ny}$ from their respective initial locations (i.e: Los Angeles and Dallas) is the same as the path cost between $v_s = \{(x_\ell, y_\ell), (x_d, y_d)\}$ and $v_d = \{(x_{ny}, y_{ny}), (x_{ny}, y_{ny})\}$ in graph $G'$ with $n^4$ vertices. The proof that the algorithm finds the least-cost path on the weighted graph $G'$ follows from the correctness of Djikstra's single-source shortest path finding algorithm.

**Runtime**: When Djikstra's algorithm uses a linked list, the total cost of $n$ deleteMin and $m$ insert operations is $O(n^2)$ on a graph of $n$ vertices. Since our transformation resulted in graph $G'$ with $n^4$ vertices, the cost, using a linked list, would be $O(n^8)$.

However, binary heaps can do inserts in $O(\log n)$ time and deleteMin in $O(\log n)$ time. This leads to a total runtime for Djikstra of $O((m+n)\log n)$ on a graph with $n$ vertices and $m$ edges. Since our transformed graph $G'$ has $O(n^4)$ vertices and $O(n^4)$ edges, it leads to a runtime of $O((n^4 + n^4)\log n^4 = \boxed{O(n^4 \log n)}$ as desired. Hence an implementation of Djikstra's algorithm using either binary heaps (or Fibonacci heap) will enable us to obtain a runtime of $O(n^4 \log n)$. □

(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 $\beta$ and a second directed edge from $\bar{\beta}$ to $\alpha$. How can this be interpreted?)