CS 170 Fall 2017

Efficient Algorithms and Intractable Problems
Prasad Raghavendra and Sanjam Garg

Homework 3

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.
- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.
- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Wednesday, September 20, at 4:59pm

0. Who did you work with?

List all your collaborators on this homework. If you have no collaborators, please list “none”.

Solution: N/A

1. (★★ level) Can you win OneSeventy?

Congratulations! UnElectronic Arts has hired you for Summer 2018. As it turns out, they've been working on a game called OneSeventy for the past 25 years, and the game has a ton of quests. **For a player to win, the player must finish all the quests.** There are a total of N quests in the game. Here's how the game works: The player can *arbitrarily* pick one of the N quests to start from. Once the player completes a quest, they unlock some more quests. The player can then choose one of the unlocked quests and complete it, and so on.

So, for instance, let's say that this game had only 4 quests, A,B,C,D.

Let's say that after you complete

- quest A, you unlock quests $[B,D]$.
- quest B, you unlock quests $[C,D]$.
- quest C, you unlock nothing, $[]$.
- quest D, you unlock quest $[C]$.

Is this game winnable? Yes, because of the following scenario:

The player picks quest A to start with. At the end of quest A, their unlocked list contains $[B,D]$. Say that they choose to do quest B, then their unlocked list will contain $[C,D]$. Say that they choose to complete quest C, then their unlocked list will contain quest $[D]$. Finally, they finish quest D.

Note that if the player had started with quest C instead of quest A, they would have lost this game, because they wouldn't have unlocked any quests and would be stuck. *But the game is still winnable, because there is **a** starting quest which makes the player win.*

OneSeventy has N quests, enumerated as $\{n_1, n_2, \ldots, n_N\}$. Each quest n_i unlocks k_i quests. So in the example, if $n_1 = A$, then $k_1 = 2$. Let $K = \sum_{i=1}^{n} k_i$.

UnElectronic Arts' concern is: is their game with N quests winnable? You are provided with information about the quests in a similar manner as the example above. Design an algorithm to find out whether or not the game is winnable. The algorithm should run asymptotically faster than $O(NK)$. You must provide the main idea and runtime for your algorithm.

Solution: Represent the quests as nodes in a graph, and an edge (i,j) exists if a player is allowed to do Quest$_j$ after Quest$_i$.

For the game to be winnable, there must exist at least one vertex from which it is possible to reach every other vertex in the graph. In other words, for the game to be winnable, this graph must have only one source strongly connected component, because two source strongly connected components are not reachable from other other.

So the algorithm is to run DFS from any node, and find the one with the highest post value. This node has to be in a source strongly connect component. Next, run DFS from this node, and check whether every vertex is reachable from it or not. If yes, the game is winnable, and otherwise it is not.

This is essentially just running DFS twice, and so it runs in $O(N+K)$ time.

2. (★★ level) Dijkstra's Durability

For the following questions, answer yes or no and provide justification. Consider a directed graph $G = (V, E)$ with positive edge lengths, and two vertices s and t. We want to find the shortest path from s to t in G, so we run Dijkstra's algorithm. Now:

(a) We also want to find the shortest path from s to another node t'. Do we need to run Dijkstra's algorithm again?

(b) Suppose we add a positive number k to all edge lengths. Is the shortest path from s to t always the same as before?

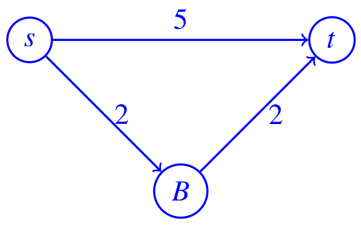
(c) Suppose we subtract a positive number k from all edge lengths. Is the shortest path from s to t always the same as before?

(d) Suppose we multiply all edge lengths by a positive number k. Is the shortest path from s to t always the same as before?

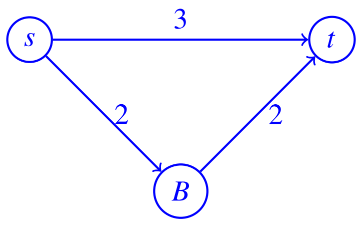
Solution:

(a) No, Dijkstra's algorithm finds distances from the source to all other nodes.

(b) No, consider this graph as a counterexample, adding $k = 2$:



(c) No, consider this graph as a counterexample, subtracting $k = 2$:



(d) Yes, let us compare two paths, p_1 and p_2. Let $C(p_i)$ denote the total cost of path p_i. The total cost of a path is just a summation over the lengths of all of the edges in the path, i.e. for p_1 the total cost is $C(p_1) = \sum_{(u,v)\in p_1} w_{uv}$. When we multiply all edges by k, this means we have $\sum_{(u,v)\in p_i} kw_{uv} = k\sum_{(u,v)\in p_i} w_{uv} = kC(p_i)$. Thus if our original relation was $C(p_1) < C(p_2)$, $C(p_1) = C(p_2)$, or $C(p_1) > C(p_2)$, we now have $kC(p_1) < kC(p_2)$, $kC(p_1) = kC(p_2)$, or $kC(p_1) > kC(p_2)$, respectively. Thus the inequality or equality is preserved through the multiplications.

3. (★★★ level) Graph Basics

(a) An undirected graph G is called *bipartite* if we can separate its vertices into two subsets A and B such that every edge in G must cross between A and B. Show that a graph is bipartite if and only if it has no odd cycles.

Hint: Consider a spanning tree of the graph, which is a subset of the graph's edges which allow it to be a tree on all of its vertices.

(b) A directed acyclic graph G is *semiconnected* if for any vertices A and B there is either a path from A to B or a path from B to A. Show that G is semiconnected if and only if there is a directed path that visits all of the vertices of G.

Solution:

(a) Without loss of generality, assume that the graph is connected. Otherwise, apply the following proof to each connected component.

First, assume a graph has no odd cycles. Do a BFS from some root vertex r. Color a vertex red if it has odd distance from r and black if it has an even distance to r. We now show that the red and black vertices form a bipartition of the graph. Suppose, for the sake of contradiction, that there are two vertices u and v that are both red and are adjacent to each other. Let x be the least common ancestor of u and v in the tree. Since u and v are both red, the xu and xv paths both have even lengths or both have odd lengths. In particular, concatenating the paths shows that there is an even-length path that connects u to v. Adding the uv edge to this path creates an odd cycle, which is a contradiction.

On the other hand, a bipartite graph can never have an odd cycle. Suppose there is an odd cycle. We can try to color the odd cycle. WLOG, assume, the first vertex of the coloring is blue. This uniquely determines the color of the next vertex, and so on. So the odd vertices on this path are blue and the even are red. The last vertex cannot be colored either red or blue, since it is neighbors with the first vertex which is also odd and the previous vertex which is even. Hence, no odd cycle can exist.

(b) First, we show that the existence of a directed path p that visits all vertices implies that G is semiconnected. For any two vertices A and B, just consider the subpath of p. If A appears before B in p, then this subpath will go from A to B. Otherwise, it will go from B to A. In either case, A and B are semiconnected for all pairs of vertices (A,B) in G. This completes this direction.

Now, we show that if the DAG G is semiconnected, then there is a directed path that visits all of the vertices. Consider a topological ordering $v_1, v_2, \dots, v_n$ of the vertices in G. For any pair of consecutive vertices v_i, v_{i+1}, we know that there is a path from v_i to v_{i+1} or from v_{i+1} to v_i by semiconnectedness. But topological orderings do not have any edges from later vertices to earlier vertices. Therefore, there is a path from v_i to v_{i+1} in G. This path cannot visit any other vertices in G because the path cannot travel from later vertices to earlier vertices in the topological ordering. Therefore, the path from v_i to v_{i+1} must be a single edge from v_i to v_{i+1}. This edge exists for any consecutive pair of vertices in the topological ordering, so there is a path from v_1 to v_n that visits all vertices of G.

4. (★★★ level) Count Shortest Paths

Given a directed graph $G = (V, E)$ with positive edge lengths, and two vertices s and t. The shortest paths from s and t are not always unique: there could be two or more different paths with the same minimum length. Design an algorithm to find the number of shortest paths from s to t. Your algorithm should be as efficient as possible.

(Please turn in a four part solution to this problem.)

Hint: Try to count the number of shortest paths from s to every other vertex.

Solution:

Main Idea: This can be done by slightly modifying Dijkstra's algorithm. For every vertex $v \in V$, let $\mathsf{cnt}(v)$ be the number of shortest paths from s to v. When $\mathsf{dist}(v)$ gets updated by $\mathsf{dist}(u) + l(u,v)$, $\mathsf{cnt}(v)$ should also be updated by $\mathsf{cnt}(u)$. The tricky part is when $\mathsf{dist}(v) = \mathsf{dist}(u) + l(u,v)$, $\mathsf{dist}(v)$ is not updated, but $\mathsf{cnt}(v)$ should be added by $\mathsf{cnt}(u)$.

Pseudocode: Slightly modify Dijkstra's algorithm: First, $\mathsf{cnt}(v)$ is initialized to be all 0 in the initialization loop, except that $\mathsf{cnt}(s) = 1$. Second, the main loop is modified as follows:

```
while $H$ is not empty do
    $u = \mathsf{deletemin}(H)$
    for all edges $(u,v) \in E$ do
        if $\mathsf{dist}(v) > \mathsf{dist}(u) + l(u,v)$ then
            $\mathsf{dist}(v) = \mathsf{dist}(u) + l(u,v)$
            $\mathsf{cnt}(v) = \mathsf{cnt}(u)$
            $\mathsf{decreasekey}(H, v)$
        if $\mathsf{dist}(v) = \mathsf{dist}(u) + l(u,v)$ then
            $\mathsf{cnt}(v) += \mathsf{cnt}(u)$
```

Proof of Correctness: By Djikstra's proof of correctness, this algorithm will identify the shortest path from the source s to all the other vertices. For the number of shortest paths, we consider some vertex v. If there are multiple shortest paths, consider the final edge of each shortest path. Assume there are a_1 shortest paths sharing the same final edge (u_1, v), a_2 shortest paths sharing the same final edge (u_2, v), ... , a_k shortest paths sharing the same final edge (u_k, v). Notice that a_i is exactly the number of shortest paths from s to u_i, namely $\mathsf{cnt}(u_i)$. Hence the total number of shortest paths from s to v is $\mathsf{cnt}(v) = \sum_{i=1}^{k} \mathsf{cnt}(u_i)$. Since all the vertices $\{u_1, u_2, \ldots, u_k\}$ are deleted from H before v, all the $\mathsf{cnt}(u_i)$ values are added to $\mathsf{cnt}(v)$ (and every value gets added exactly once). By an inductive argument we conclude that all the $\mathsf{cnt}(\cdot)$ values are correct.

Runtime Analysis: The running time is the same as Djikstra's algorithm $O((|V| + |E|)\log|V|)$ when the heap is implemented as a binary heap.

5. (★★★★★ level) Premium Member

There is a set V of cities connected by flights operated by Algorithmic Airline (AA), and each flight has a cost. Let $H \subseteq V$ be a set of AA hubs, and AA would like to encourage its members to choose flights landing at these hubs. If an AA member has landed in these hubs for a total of k times, he/she will become a premium member. Prof. Garg just became an AA member and will start traveling. He is currently in Berkeley and the only way he will take to travel between cities is taking AA flights. What is the minimum cost for him to become a premium member?

Formally, given a directed graph $G = (V, E)$ with positive edge lengths, a non-empty set $H \subseteq V$, a positive integer k, and a starting vertex s. Design an algorithm with running time polynomial in $|V|$, $|E|$, and k to find the shortest path starting at s and containing at least k vertices in H (with cycles permitted). If there doesn't exist such a path, your algorithm should output ∞.

(Please turn in a four part solution to this problem.)

Hint: Try to construct a new graph, and try to solve the case $k = 2$.

Solution:

Main Idea: We will make $k+1$ copies of G. Now connect consecutive copies at all of the AA hubs (vertices in H). Think of this as $k+1$ layers. We start at the first layer, and the AA hubs are the only way to go to the next layer. To be more concrete, each edge going into some $v \in H$ becomes a directed edge to the copy of v in the next layer. Thus, we can only reach the next floor by landing in a AA hub, so when we arrive at the i-th layer, we have landed in exactly $(i-1)$ AA hubs. To solve our problem we merely need to use Dijkstra's algorithm to find the shortest path from the first to the last layer.

Pseudocode:

```
procedure ($G$, $H$, $k$, $s$)
    Construct graph $G^\star$ as $k+1$ identical copies of vertices and egdes in $G$
    Let $G_i$ denote the $i$-th copy of $G$ in $G^\star$
    for every $G_i$ from $G_1$ to $G_k$ do
        for each edge $(u, v)$ in $G_i$ s.t. $v \in H$ do
            Remove $(u, v)$ and add an edge $(u, v')$ where $v'$ indicates the vertex in $G_{i+1}$ corresponding to $v$
    Run Dijkstra's algorithm on $G^\star$ starting at $s$ of $G_1$
    Output the minimum of the shortest path that terminates at any vertex in $G_{k+1}$
```

Proof of Correctness: Notice that the only edges between G_i and G_{i+1} for any i terminate in some $v \in H$. And there is no edge from G_i to G_j where $j < i$. By induction we can prove that any path from s to some vertex w in G_i must pass through exactly $(i-1)$ vertices in H. Hence a path passes through k AA hubs if and only if the terminal vertex is in G_{k+1}. We wish to find the shortest such path, so we can simply run Dijkstra's algorithm on $G^\star$.

Runtime Analysis: Let $V^\star$ be the set of vertices in $G^\star$ and $E^\star$ be its set of edges. Dijksta's algorithm runs in $O((|V^\star| + |E^\star|)\log|V^\star|)$ time. $|V^\star| = \Theta(k|V|)$ and $|E^\star| = \Theta(k|E|)$. Therefore, the running time of our algorithm is in $O(k(|V| + |E|)\log(k|V|))$.

Note: A slight variation would be to first compute shortest paths between every pair of AA hubs by using $|H|$ calls to Dijkstra's. Then use multiple copies of the graph which has only vertices from H and all possible edges between them with lengths equal to the shortest paths in G. This gives a running time of $O\left(|H|(|V| + |E|)\log|V| + k|H|^2\log(k|H|)\right) \in O\left(|V|(|V| + |E|)\log|V| + k|V|^2\log(k|V|)\right)$.

6. (★★★★★ level) Alternate Universe Theory

It is the year 3050, and humankind is divided into three – the magicians, the ordinary, and you. The ordinary and the magicians don't interact with each other, but you are special. You can interact and use the transport systems of both, the ordinary and the magicians. Here is the idea: the magicians can transport you across different universes. Sometimes, they charge you money for this service, but sometimes they give you money instead. A magician can only teleport you one-way. Also, it so happens that if a magician teleports you from u to v, then there is **no way for you to go back** from v to u no matter how you travel. In other words, if there is an edge from u to v (where u, v are in different universes), then there is no path of edges that leads from v to u.

Now, at each separate universe, exist the ordinary folk and taxi-stops. They drive taxis, charge you some money and transport you from one taxi-stop to another. Taxis drive both-ways, so if you get to stop a from b, you can just take a taxi back from b to a. Some of these taxi-stops are **also** magician-stops, where you can find magicians to get teleported by.

In summary, taxis drive from taxi-stop to taxi-stop. Some taxi-stops may also be magician-stops. Magicians are found at magician-stops, and they teleport you from one universe's magician-stop to another universe's taxi-stop.

Formally, our graph looks like $G = (V, T \cup U)$ where V is the set of all stops (taxi or magician), T is the set of edges where taxis can drive (between taxi-stops) and U is the set of universe-teleportation edges, i.e., if $(x, y) \in U$, then there is a magician that will take you from universe x to universe y.

An edge in T is specified as an **undirected** edge (v_1, v_2, M_{v_1,v_2}), which means that there is a taxi that will take you from v_1 to v_2 and charge you M_{v_1,v_2}. Note that the cost of going from $(v_2)to(v_1)$ need not be the same as M_{v_1,v_2}.

An edge in U is specified as a **directed** edge (u_1, u_2, M_{u_1,u_2}). Remember that magicians could give you money to use their services.

Your goal is to find the cheapest path from some $s_1 \in V$ to some $s_2 \in V$ in time $O((|V| + |E|)\log|V|)$ where $E = T \cup U$. A four part solution is required for this problem.

(Please turn in a four part solution to this problem.)

Hint: Think about the fact that if there is an edge from u to v, where u, v are in different universes, then there is no path back from v to u. What does this mean about each universe?

Hint 2: If you are stuck, try thinking about the following questions (no credit for answering).

- *Let there be a directed graph, where there are some negative edges. These negative edges are all of the form (s, t_i) for some vertex s, i.e., all these edges go out of s. If Dijkstra's algorithm is run starting at s, will it work correctly?*
- *Now consider the set up from the previous hint, i.e., all the negative edges leave some vertex s. Say that your starting node is v_1. Find an algorithm to compute the shortest path from v_1 to v_2. The runtime of this efficient algorithm should be the same as that of good old Dijkstra's.*

Solution:

Hint answers:

- Yes, if there is no negative cycle, then Dijkstra's algorithm would work.
  Proof: Consider the proof of Dijkstra's algorithm. The proof depended on the fact that if we know the shortest paths for a subset $S \subseteq V$ of vertices, and if (u, v) is an edge going out of S such that v has the

minimum estimate of distance from s among the vertices in $V \backslash S$, then the shortest path to v consists of the (known) path to u and the edge (u,v). We can argue that this still holds even if the edges going out of the vertex s are allowed to be negative. Let (u,v) be the edge out of S as described above. For the sake of contradiction, assume that the path claimed above is not the shortest path to v. Then there must be some other path from s to v which is shorter. Since $s \in S$ and $v \notin S$, there must be some edge (i,j) in this path such that $i \in S$ and $j \notin S$. But then, the distance from s to j along this path must be greater than that the estimate of v, since v had the minimum estimate. Also, the edges on the path between j and v must all have non-negative weights since the only negative edges are the ones out of s. Hence, the distance along this path from s to v must be greater than the estimate of v, which leads to a contradiction.

- A shortest path from v_1 to v_2 may or may not pass through s.

  To find a shortest path from v_1 to v_2 that does not pass through s, we delete s from the graph and run Dijkstra's algorithm from v_1. Let d'_{v_1,v_2} be the length of the shortest path found.

  For the other case we need to

  - Find a shortest path from v_1 to s. This can be done by running Dijkstra's algorithm from v_1. Since all the negative edges leave s and there is no negative cycle, Dijkstra's algorithm will correctly find a shortest path from v_1 to s. Let this distance be $d_{v_1,s}$.
  - Find a shortest path from s to v_2. From Part (a) we know that this can be done by running Dijkstra's algorithm from s. Let this distance be d_{s,v_2}.

  The shortest distance is then given by $d_{v_1,v_2} = \min(d'_{v_1,v_2}, d_{v_1,s} + d_{s,v_2})$. A path corresponding to the shortest distance can be found in a similar way as Dijkstra's algorithm. The algorithm has same asymptotic running time as Dijkstra's algorithm since it runs Dijkstra's algorithm three times and deleting the vertex s from the graph can be done in linear time.

Solution 1:
Main Idea: We will modify Dijkstra's algorithm to solve this problem. Dijkstra keeps a set A, and $dist(u)$ for all $u \notin A$. A contains nodes whose shortest path from s_1 are already computed. For $u \notin A$, $dist(u)$ stores the shortest $s_1 \to u$ path among the paths that start at s_1, and use only nodes in A before ending at u. At each iteration, Dijkstra adds the u with the smallest $dist(u)$ to A, and update the $dist(v)$ for all v where $(u,v) \in E$.

In particular, we will change the order of how the nodes are added to A. We first recognize the SCCs of our graph treating taxi-routes as bi-directional edges. If two nodes are connected by a taxi-route, then they must be in the same SCC, and the additional constraint about magicians ensures that no universe edge will be inside an SCC. Thus the edges across different SCCs are exactly the universe routes.

We first linearize the SCCs to get a topological order of them, and for each node u in a SCC, let $r(u)$ be the index of the SCC in the topological ordering. It is easy to see that if $r(u) = r(v)$, then any path from u to v must be all taxi-routes, and if $r(u) > r(v)$, there won't be any path from u to v.

The modification we make to Dijkstra is that on each iteration, we include the u with smallest $r(u)$, and among the nodes with the same $r(u)$, we include the one with the smallest $dist(u)$.

Pseudocode:

```
procedure TopologicalDijkstra's($V$, $T$, $U$, $s_1$, $s_2$)
```

Find and linearize the meta-graph (SCCs) of G, extract a valid topological sort.

Create priority queue of nodes sorted first on the position in this topological sort $r(\cdot)$, then sorted by distance $dist(\cdot)$.

Now run Dijkstra's to find the shortest path from s_1 to s_2.

Proof of Correctness:
We will prove by induction that when we add the node u to the visited set A, $dist(u)$ is the true shortest path distance from s_1 to u.

Base Case: We have the shortest path from the source s_1 to itself.

Inductive Hypothesis: All vertices v in A have $dist(v)$ equal to the length of the shortest path from s_1 to v, and all vertices v in V have $dist(v)$ equal to the length of the shortest path from s_1 to v that uses only nodes in A as intermediate nodes.

Inductive Step: Let u denote the node to be returned by the priority queue (sorted by topological ordering $r(u)$, then distance $dist(u)$). Let u_{prev} denote the previous node in the shortest path to u.

Case: The edge (u_{prev}, u) is a taxi-edge OR a non-negative cost universe edge. The same proof for adding a new node to the visited set in Dijkstra's holds. Therefore, our inductive hypothesis holds after we add u to the visited set.

Case: The edge (u_{prev}, u) is a negative universe edge. Assume for contradiction there is some other path that we have not yet considered, but is actually shorter than $s_1 \to \cdots \to u_{prev} \to u$. Because we sort our priority queue by topological sorting first, the other path must also consider some universe edge to get to the new SCC $r(u)$; call this path $s_1 \to \cdots \to v_{prev} \to v \to \cdots \to u$, and the corresponding universe edge $(v_{prev} \to v)$.

We know that all of the edges in the new path $s_1 \to \cdots \to v_{prev} \to v \to \cdots \to u$ after taking the flight $v_{prev} \to v$ are taxi-routes and therefore must be non-negative; $s \to \cdots \to v_{prev} \to v$ must be shorter than the old path $s_1 \to \cdots \to u_{prev} \to u$, and therefore has already been visited.

Because $s_1 \to \cdots \to v_{prev} \to v \to \cdots \to u$ is assumed to be the true shortest path to u, all of the intermediate nodes have been added to the visited set A, and their true distances have been computed due to our inductive hypothesis, regardless of the presence of negative universe edges. Because we are considering visiting the SCC $r(u)$, all of its parent SCCs have been visited fully. The path $v \to \cdots \to u$ is contained exclusively in this SCC, and has already been visited because all of the intermediate nodes are of lower distance than $s_1 \to \cdots \to u_{prev} \to u$. All of this occurred before adding u to the visited set A through $s \to \cdots \to u_{prev} \to u$, which contradicts our assumption that we hadn't yet visited this path; therefore, the path $s_1 \to \cdots \to u_{prev} \to u$ is the true shortest path for $s_1 \to \cdots \to u$, and we can safely add node u to the visited set A.

Furthermore, for any node w not in A, $dist(w)$ is equal to the length of the shortest path from s_1 to w that uses only nodes in A as intermediate nodes. If the shortest according path $s_1 \to w$ does not contain the new visited node u, adding u to the visited set A does not change $dist(w)$ and therefore remains optimal using nodes only from A. Otherwise, the path must be of the form $s_1 \to \cdots \to u \to w$ (otherwise it would have already been explored in a previous step). This new path will correctly reflect $dist(w)$ using only nodes from

A, since we have the shortest path for $s_1 \to \cdots \to u$ from the previous paragraphs, and $u \to w$ from visiting u.

Running time analysis:
The running time is the same as Dijkstra's, since the preprocessing (i.e. finding SCCs, linearization) is linear time, and the rest is the same complexity as Dijkstra's.

Note:
There is an alternate formulation of this solution that visits each SCC individually and completely in topological order with its own (smaller) instance of Dijkstra's, and adds a dummy source node to each SCC, and an edge from the source to each of the vertices with an incoming universe edge. If (u,v) is the universe edge, then the weight of (s_1,v) is $dist[u] + weight(u,v)$.

Solution 2:

Main idea: We will propagate negative weights through the graph until we reach a sink (SCC), at which point they can be safely removed, leaving us with a graph with only positive weights, so we can run a simple Dijkstra's algorithm.

Pseudocode:

```
procedure DELETMAGIC($V$, $T$, $U$, $s_1$, $s_2$)
    Find and linearize the meta-graph (SCCs) of G.
    Consider SCCs in topological order.
    When considering the $i$th SCC, if there are any negative incoming edges, let $w$ be the minimum weight of incoming edges, and increase all incoming edges' weights by $|w|$ and decrease all outgoing edges' weights by $|w|$.
    Now run Dijkstra's to find the shortest path from $s_1$ to $s_2$.
```

Running time analysis:

The linearization and changing of edge weights are linear in $(|V| + |T| + |U|)$. Afterwards we run Dijkstra's algorithm, which dominates the overall running time, giving us $O((\log|V|)(|V| + |T| + |U|)$.

Proof of correctness: For any SCC that is not a sink or source, any path taking an incoming edge of this SCC must also include one of its outgoing edges, so the path has no net change since we added the same amount to the incoming edge that we subtracted from the outgoing edge. For source SCCs, no change is done since there are no incoming edges. For sink SCCs, we add the same $|w|$ to all incoming edges, so each incoming path has the same cost as it would have before, plus $|w|$. Thus the shortest path in G is still the the shortest path in our graph. Note that the only sink we may be interested in is one containing our destination v_2, so it does not matter that total costs of paths to different sinks may be altered by different amounts.

7. (??? level) (Optional) Redemption for Homework 2

Submit your *redemption file* for Homework 2 on Gradescope. If you looked at the solutions and took notes on what you learned or what you got wrong, include them in the redemption file.

Solution: N/A