CS 170
Fall 2017

Efficient Algorithms and Intractable Problems
Prasad Raghavendra and Sanjam Garg

Homework 4

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, October 4, 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) Kruskal and Prim

Answer the following questions. Justify your answer with a short proof or counterexample. You can assume all edges have distinct costs.

(a) We create a subgraph of G with the first k edges chosen by Kruskal's algorithm. Can there be a strictly cheaper acyclic subgraph of G with k edges?

(b) We create a subgraph of G with the first k edges chosen by Prim's algorithm (starting from some root node r). Can there be a strictly cheaper connected subgraph of G containing the node r along with k other nodes?

Solution:

(a) No, there cannot be a cheaper acyclic subgraph. We will prove this by contradiction as follows.
Call the sorted edges in the proposed cheaper subgraph $e'_1, \ldots e'_k$. Likewise, the sorted edges in the Kruskal solution are $e_1, \ldots, e_k$. We show that for each corresponding pair (e'_j, e_j), e'_j is heavier or equally as heavy as e_j, which leads us to conclude that there is no lighter subgraph than the one output by Kruskal's.

Consider a particular pair: let A be the set of edges in G cheaper than e_j. If e'_j is cheaper than e_j, then e'_i for $i \in [1, j]$ must be cheaper than e_j. This amounts to saying that e'_i for $i \in [1, j]$ forms an acyclic subgraph on the graph $G_A = (V, A)$ (V is the set of original vertices), since e'_i are part of a tree.

However, we know that G_A consists of $|V| - (j-1)$ connected components (including singleton CC's, aka nodes) because Kruskal's picked $j-1$ edges in A and these formed a spanning forest on G_A. Thus, there can't be a spanning forest of more than $j-1$ edges (we can see this is true if we consider each CC in G_A separately, and apply the argument that a tree on a graph (CC) has exactly $V-1$ edges (let V be number of nodes in CC)). This means there can't be a forest of more than $j-1$ edges, and it is impossible for e'_i for $i \in [1, j]$ to form an acyclic subgraph. Thus, e'_j is not cheaper than e_j.

(b) There there can be a cheaper connected subgraph. A counterexample is as follows.
Consider a path of 7 nodes with r at the middle (i.e. r is the 4-th node), and the edge costs are $1, 2, 9, 6, 7, 8$ from the left to the right respectively. For $k = 3$, Prim's algorithm will pick the three edges to the right of r, with total cost 21, yet the subgraph containing r and the three nodes to its left has cost 12.

2. (★★★ level) Uniqueness of Minimum Spanning Tree

In this problem, we will study the conditions that make a graph have a unique MST and devise an efficient algorithm to recognize such graphs. For all parts of this problem, assume that the input graph is connected.

(a) In general, a graph need not have a unique MST. Give an example of a graph with multiple MSTs.

(b) Recall the *Cut Property*: Suppose edges X are part of at least one MST of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across this partition. Then, $X \cup \{e\}$ is part of some MST.

Prove that, if we additionally assume that e is the unique lightest edge across the partition, then not only *some* MST containing X contains e but *every* MST containing X also contains e. This is called the *Unique Cut Property*.

(c) For any graph G, let E' be the set of all edges e such that e is a (not necessarily unique) lightest edge across some partition. Show that G has a unique MST if and only if E' forms a spanning tree of G.

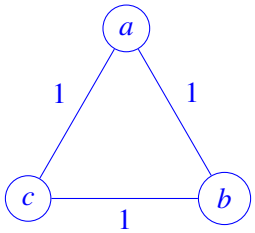
Hint: you may use part (b).

(d) Design an efficient algorithm to decide whether a graph $G = (V, E)$ has a unique MST. Your algorithm's running time should be at most $O((|V| + |E|) \log |V|)$.

(Please turn in a four part solution to this problem, but, in the "Proof of Correctness" section, you are allowed to provide just a short justification of the algorithm instead of a full formal proof.)

Solution:

(a) There are many answers to this question. An example is a triangle where each edge weights the same:



(b) The proof is basically the same as that of the normal Cut Property.

Proof: Let X be a set of edges that are part of at least one MST of $G = (V, E)$ and let S be any subset of nodes such that X does not cross between S and $V - S$. Suppose for the sake of contradiction that e is the unique minimum edge across this cut but e is not contained in some MST T that contains X.

Consider $T \cup \{e\}$. Since e is not part of the spanning tree T, $T \cup \{e\}$ has a simple cycle that contains e. Since e crosses the cut $(S, V - S)$, there must be another edge e' in the cycle that also crosses the cut. Since e is the unique minimum edge across the cut S and $V - S$, e' must has strictly more weight than e. As a result, $T \cup \{e\} - \{e'\}$ is a spanning tree that is strictly lighter than T, which is a contradiction since the latter is a MST. □

(c) **Proof**: First, observe that E' always connects every vertex in V because it contains at least one edge from each cut.

Now, for the forward direction, suppose for the sake of contradiction that G has a unique MST T but E' does not form a spanning tree of G. Since E' always connects every vertex in V, there must be an edge e' that is in E' but not in the MST T. From the Cut Property e' must be in some MST, which contradicts the assumption that T is a unique MST of G.

For the other direction, suppose that E' is a spanning tree of G. Consider any edge $e \in E'$. Since E' is a spanning tree, removing e from E' must partition the vertices into two connected components S and $V - S$. However, since, by definition of E', every edge with lightest weight across $(S, V-S)$ is included in E', e must be the unique lightest edge across the partition. From the Unique Cut Property, we have that e must be in every MST of G. In other words, every MST of G must contain every edge from E'. However, since E' is already a MST, we can conclude that it must be the only MST in G. □

(d) There are two solutions to the problem.

Solution 1:

Main Idea:

This solution uses the intuitions developed from the earlier parts of the problem. Specifically, we will find the subset E' defined in the previous part and check whether E' is a spanning tree of G. The only tricky part is to figure out whether an edge e should be in E' (i.e. whether e is one of the lightest edge across some cut $(S, V-S)$). It is not hard to see that e is included in E' if and only if, when we look at the subgraph induced by all edges strictly lighter than e, the two endpoints of e are in different connected components. Note here that, we already have the connected components ready in our Kruskal's algorithm! The only thing we need to be careful is that, in the original Kruskal's implementation, when we consider an edge e, it is possible that we already considered some other edge e' whose weight is the same as e; in this case, e' might have already been added to the solution. However, for the purpose of this problem, we do not want to union the two endpoints of e' yet. This can be easily implemented by, instead of immediately union the two endpoints of each edge e' that we want to add to the tree, we add e' to a list and only union its endpoints when we see an edge e whose weight is strictly more than e'.

Pseudocode:

In the implementation below, $E_{\text{to-add}}$ is the list of edges that we have held off the unions of their endpoints and $\ell_{\text{to-add}}$ is the weight of these edges. We only union the endpoints of these edges only when we see an edge whose weight is strictly larger than $\ell_{\text{to-add}}$.

```
1:  function UNIQUE-MST(G = (V, E))
2:      for all u ∈ V do
3:          MAKESET(u)
4:      Let E' = {}.
5:      Let E_to-add = {} and ℓ_to-add = −∞.
6:      Sort the edges E by weight.
7:      for all edges {u, v} ∈ E, in increasing order of weight do
8:          if ℓ_to-add ≠ ℓ(u, v) then
9:              for all edges {u', v'} ∈ E_to-add do
10:                 UNION(u', v')
11:             Let ℓ_to-add = ℓ(u, v) and E_to-add = ∅.
12:         if FIND(u)≠FIND(v) then
13:             Add (u, v) to both E' and E_to-add
14:     if |E'| = |V| − 1 then
15:         return true
16:     else
17:         return false
```

Proof Sketch of Correctness:

As stated in the main idea, the key claim is the following: an edge e is in E' if and only if the endpoints of e are in different connected components in the subgraph induced by the edges that are strictly lighter

than e.

To see that this claim is true, let us first consider the forward direction. Suppose that e is in E'. Then, it must be one of the lightest edges across some cut $(S, V-S)$. Hence, if we look at the subgraph induced by the edges strictly lighter than e, then there is no edge across this cut, meaning that the two end points of e must be in different components.

For the backward direction, suppose that an edge e has endpoints in different connected components in the subgraph induced by the edges that are strictly lighter than e. Let S be the connected component of one endpoint. Consider the cut $(S, V-S)$. There is no edge strictly lighter than e across this cut, and e crosses the cut. Hence, e must be one of the lightest edges across $(S, V-S)$, which by definition means that $e \in E'$.

Once we have proved this claim, observe that, on line 12 when $\text{FIND}(u), \text{FIND}(v)$ are called, the disjoint set data structure represents the subgraph induced by edges strictly lighter than $\{u, v\}$. As a result, by our claim above, $\text{FIND}(u) = \text{FIND}(v)$ if and only if $\{u, v\}$ is in E'. In other words, we have computed E' correctly. Finally, note that, since E' must connect all vertices, we can check whether it is a spanning tree by just checking that $|E'| = |V| - 1$, which is indeed what our algorithm does.

Runtime:

Same as runtime of Kruskal's algorithm, i.e. $O((|V| + |E|) \log |V|)$.

Solution 2:

Main Idea:

In this solution, we first find a MST T of $G = (V, E)$. Then, we will create a new graph $G' = (V', E')$ where the vertex set and the edge set is the same as those of G. The only difference between G and G' is that we will assign the weights in such a way that finding a MST of G' is equivalent to finding a MST of G that shares as few edges with T as possible. This can be done as follows: for an edge $e \in E$ with weight w, we assign the weight of this edge in G' to be $(w, 1)$ if the edge is included in T and $(w, 0)$ otherwise. Here the sum of two tuples is the coordinate-wise sum and tuples are ordered by the first coordinate first and, if there is a tie, use the second coordinate to break the tie.

At this point, we can just find an MST on G'. If it is the same as that in G, then we know that the MST is unique. Otherwise, it is not unique.

Pseudocode:

1: **function** UNIQUE-MST($G = (V, E)$)
2: Find a MST T of G using Kruskal's or Prim's algorithm.
3: Create a new graph $G' = (V', E')$ where the vertex set and the edge set are the same as those of G. For an edge $e \in E$ of weight w in G, we assign it weight $(w, 0)$ if $e \in T$ and $(w, 1)$ otherwise.
4: Find a MST T' of G' using Kruskal's or Prim's algorithm.
5: **if** $T = T'$ **then**
6: **return** true
7: **else**
8: **return** false

Proof Sketch of Correctness:

Since the vertex set and the edge set of G' is the same as those of G, an edge set is a spanning tree of G' if and only if it is a spanning tree of G. Moreover, if a spanning tree T^* has total weight W in G, the total weight of T^* in G' will be (W, P) where P is the number of edges T^* shares with T.

As a result, finding a MST T' in G' is equivalent to finding a MST in G where ties are broken by the number of edges that T' shared with T. Hence, a MST T' in G' is equal to T if and only if T is the unique MST of G.

Runtime:
Same as runtime of finding MST, i.e. $O((|V|+|E|)\log|V|)$.

3. (★★★★ level) A Wizardry Party

Unlike magicians, wizards are not known to transport people across universes. Instead, they are known for wearing (often pointy) hats, sporting long white beards, and just being old and grumpy in general. Nevertheless, they still like to socialize and they regularly organize parties. Although you are a Muggle (i.e. No-Maj), you would like to attend these cool parties too. There is one problem regarding this though: these wizards talk a lot and, if you are to attend their parties, you have to blend in well during conversations.

One topic they converse about often is their ages. Not totally unlike us humans, wizards consider asking for someone's age or saying it out loud a bit rude. Instead, wizards ask about their relative ages, i.e., how much older is wizard x_i compared to wizard y_i? During the party, you sometimes overhear other wizards' questions and answers. While other times, some wizards ask you questions. In order to not look too suspicious, you would like to try your best to use the queries you overheard to answer questions directed to you.

More specifically, at each time step i, you are given one of the following as input:

- Three integers x_i, y_i and d_i: this means that you overhear that wizard x_i is older than wizard y_i by d_i years. Note that d_i can be negative if wizard y_i is older than wizard x_i.
- Two integers x_i and y_i: this means that you are asked how much older wizard x_i is compared to wizard y_i. Your answer should be of the following two forms:
  - If you can determine the answer from the conversations you overheard so far, then output a single integer corresponding to the number of years wizard x_i is older than wizard y_i.
  - Otherwise, output "I don't know".

Design an efficient algorithm to solve the problem. To receive full points, your algorithm should run in time at most $O(m \log n)$ where m is the number of questions and n is the number of wizards.

(Please turn in a four part solution to this problem, but, in the "Proof of Correctness" section, you are allowed to provide just a short justification of the algorithm instead of a full formal proof.)

Example Input:
1 2 1
1 3
2 3 -3
1 3
Example Output:
"I don't know"
-2
Explanation: First, you overhear that wizard 1 is one year older than wizard 2. Then, you are asked how much older wizard 1 is compared to wizard 3; at this point, you do not have enough information to answer the question yet, so you just output "I don't know". Next, you learn that wizard 2 is three years younger than wizard 3. You are then asked the same question again. This time you have enough information to deduce that wizard 1 is two years younger than wizard 2 and hence you output -2.

Solution: Main Idea:
Let us view the overheard information as graphs where each vertex corresponds to a wizard and there is an edge between x and y if and only if we have overheard the age difference between wizard x and wizard y. The main observation is that we can determine the age difference of two wizards only when there is a path between them. Hence, using disjoint set data structure covered in class, we can already determine whether we should output "I don't know" or the age difference in $O(m \log^* n)$ time. To figure out the age differences, we simply augment the age difference between every vertex and its parent to the disjoint set data structure.

Pseudocode:

In the implementation below, we use `diff` to denote the age difference between x and its parent $\pi(x)$; in particular, $\texttt{diff}(x)$ is equal to the number of years $\pi(x)$ is older than x. FIND(x) is also adjusted so that it not only returns the root but also the age different between x and its root. Similarly, `Union` now takes an additional parameter d, indicating the number of years that x is older than y. Finally, `Union` and `Query` are the functions that should be called when we overheard an information and when we are asked a question respectively.

function MAKESET(x)
    $\pi(x) = x$
    $\texttt{rank}(x) = 0$
    $\texttt{diff}(x) = 0$
function FIND(x)
    if $\pi(x) \neq x$ **then**
        $(y, d) =$ FIND($\pi(x)$)
        Let $\pi(x) = y$ and $\texttt{diff}(x) = d + \texttt{diff}(x)$.
    return $(\pi(x), \texttt{diff}(x))$
function UNION(x, y, d)
    $(r_x, d_x) =$ FIND(x)
    $(r_y, d_y) =$ FIND(y)
    if $r_x \neq r_y$ **then**
        if $\texttt{rank}(x) > \texttt{rank}(y)$ **then**
            $\pi(r_y) = r_x$
            $\texttt{diff}(r_y) = d_x + d - d_y$
        else
            $\pi(r_x) = r_y$
            $\texttt{diff}(r_x) = d_y - d - d_x$
            if $\texttt{rank}(x) = \texttt{rank}(y)$ **then**
                $\texttt{rank}(y) = \texttt{rank}(y) + 1$
function QUERY(x, y)
    (r_x, d_x) = FIND(x)
    (r_y, d_y) = FIND(y)
    if $r_x = r_y$ **then**
        return $d_y - d_x$
    else
        return "I don't know"

Proof Sketch of Correctness:

By simple arithmetics, it is not hard to verify that, for every vertex x, $\texttt{diff}(x)$ is always the number of years x's parent $\pi(x)$ is older than x. Hence, the second argument return by $\texttt{Find}(x)$ is always the age difference between x and its root. As a result, the difference $d_y - d_x$ returned by $\texttt{Query}(x, y)$ is indeed the number of years wizard x is older than wizard y.

Runtime:

Same as runtime of disjoint set data structure with union-by-rank and path compression, which is $O(m \log^* n)$. We also accept a solution that uses only union-by-rank heuristic, which results in a running time of $O(m \log n)$.

4. (★★★ level) Arbitrage

Shortest-path algorithms can also be applied to currency trading. Suppose we have n currencies $C = \{c_1, c_2, \ldots, c_n\}$: e.g., dollars, Euros, bitcoins, dogecoins, etc. For any pair i, j of currencies, there is an exchange rate $r_{i,j}$: you can buy $r_{i,j}$ units of currency c_j at the price of one unit of currency c_i. Assume that $r_{i,i} = 1$ and $r_{i,j} \geq 0$ for all i, j.

(a) The Foreign Exchange Market Organization (FEMO) has hired Oski, a CS170 alumnus, to make sure that it is not possible to generate a profit through a cycle of exchanges; that is, for any currency $i \in C$, it is not possible to start with one unit of currency i, perform a series of exchanges, and end with more than one unit of currency i. (That is called *arbitrage*.) Give an efficient algorithm for the following problem: given a set of exchange rates $r_{i,j}$ and two specific currencies s, t, find the most advantageous sequence of currency exchanges for converting currency s into currency t. We recommend that you represent the currencies and rates by a graph whose edge lengths are real numbers.

(b) In the economic downturn of 2016, the FEMO had to downsize and let Oski go, and the currencies are changing rapidly, unfettered and unregulated. As a responsible citizen and in light of what you saw in lecture this week, this makes you very concerned: it may now be possible to find currencies $c_{i_1}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \times r_{i_2,i_3} \times \cdots \times r_{i_{k-1},i_k} \times r_{i_k,i_1} > 1$. This means that by starting with one unit of currency c_{i_1} and then successively converting it to currencies $c_{i_2}, c_{i_3}, \ldots, c_{i_k}$ and finally back to c_{i_1}, you would end up with more than one unit of currency c_{i_1}. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for profit.

You decide to step up to help out the World Bank. Given an efficient algorithm for detecting the presence of such an anomaly. You may use the same graph representation as for part (a).

Solution:

(a) **Main Idea:**
We represent the currencies as the vertex set V of a complete directed graph G and the exchange rates as the edges E in the graph. Finding the best exchange rate from s to t corresponds to finding the path with the largest product of exchange rates. To turn this into a shortest path problem, we weigh the edges with the negative log of each exchange rate. Since edges can be negative, we use Bellman-Ford to help us find this shortest path.

Pseudocode:

```
1: function BestConversion(s, t)
2:     G ← Complete directed graph, c_i as vertices, edge lengths l = {−log(r_{i,j}) | (i, j) ∈ E}.
3:     dist, prev ← BellmanFord(G, l, s)
4:     return Best rate: e^{−dist[t]}, Conversion Path: Follow pointers from t to s in prev
```

Proof of Correctness:
To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph, taking s as origin. The correctness of the entire algorithm follows from the proof of correctness of Bellman-Ford.

Runtime:
Same as runtime of Bellman-Ford, $O(|V|^3)$ since the graph is complete.

(b) **Main Idea:**
Just iterate the updating procedure once more after $|V|$ rounds. If any distance is updated, a negative

cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

Pseudocode: This algorithm takes in the same graph constructed in the previous part.

```
1: function HASARBITRAGE(G)
2:     dist, prev ← BELLMANFORD(G, l, s)
3:     dist* ← Update all edges one more time
4:     return True if for some v, dist[v] > dist*[v]
```

Proof of Correctness:
Same as the proof for the modification of Bellman-Ford to find negative edges.

Runtime:
Same as Bellman-Ford, $O(|V|^3)$.

Note:
Both questions can be also solved with a variation of Bellman-Ford's algorithm that works for multiplication and maximizing instead of addition and minimizing.

5. (★★★★ level) Dogs and Strangers

You're walking home from work, with your city modeled as a directed graph. You start at vertex s with vertex t as your destination. On each road (directed edge), you may encounter a dog, generous stranger, or no one at all. Meeting a dog means you give him one or more pieces of your bacon. A generous stranger will give you a certain number of bacon pieces. And meeting no one means you keep your current amount of bacon. Each road always has the same person/dog who takes or gives the same amount of bacon each time you walk down that road. You leave your starting location without any bacon. At no point on your journey home are you allowed to have negative pieces bacon (meaning you would owe a dog bacon), and you are allowed to go down the same road multiple times. You can assume that every vertex is reachable from s and you can reach t from every vertex. Devise an efficient algorithm to determine whether there exists a path for you to get home or if no such path exists.

Solution: Main ideas:

1. Use a variant of Bellman-Ford;
2. At each update, verify that the value of the path never becomes negative;
3. If a reachable (from s, via a path that is never negative) positive cycle is found, we are guaranteed to have some legal path from s to t.

Pseudocode:

($w(e)$ is the weight of each edge (can be negative))

procedure BACON($G = (V,E)$; $w : E \to \mathbb{R}$; $s,t \in V$) ▷ Initialize:
  for $v \in V$ **do**
    $Value[v] \leftarrow -\infty$
  $Value[s] \leftarrow 0$
  ▷ Main Loop:
  for $i = 1 \ldots |V| - 1$ **do**
    for $(u \to v) \in E$ **do**
      if $Value[v] < Value[u] + w(e)$ AND $Value[u] \geq 0$ **then** ▷ Idea # 2
        $Value[v] \leftarrow Value[u] + w(e)$
  if $Value[t] \geq 0$ **then return** True
  ▷ Find positive cycles
  for $(u \to v) \in E$ **do**
    if $Value[v] < Value[u] + w(e)$ AND $Value[u] \geq 0$ **then return** True
  return False

Proof:

Claims 1 and 5 below guarantee that if there is an always-nonnegative path from s to t, the algorithm returns true. Claims 2-4 guarantee that if the algorithm returns true, then there is a path.

Note that in the Main Loop the algorithm tries to find the path with the highest total weight (which is different from the length (number edges) of the path) of length $|V| - 1$ or less. The loop after that looks for paths that have more than $|V| - 1$ edges in order to check for cycles.

Claim 1: If there is an always-nonnegative path from s to t of length at most $|V|-1$, the algorithm discovers it in the Main Loop.

Proof of Claim 1: By induction on path length.

Induction hypothesis: If there is an always-nonnegative path from s to v (for any $v \in V$) of length ℓ and total weight $\bar{w}$, we will have $Value[v] \geq \bar{w}$ by the ℓ-th iteration of the Main Loop.

Base case: $\ell = 0$, $v = s$.

Induction step: Let u be the last vertex before v in the path. By the induction hypothesis, after the $(\ell-1)$-th, $Value[u] \geq \bar{w} - w(u \to v)$, and since the path is always-nonnegative, $Value[u] \geq 0$. Therefore after we tried to perform the ℓ-th update on edge $(u \to v)$, we have that $Value[v] \geq \bar{w}$.

Claim 2: If $Value[t] \geq 0$, then there is an always-nonnegative path from s to t.

Proof of Claim 2: By induction on the Main Loop's updates.

Induction hypothesis: At any point during the run of the algorithm and for every $v \in V$, if $Value[v] \geq 0$, then there is an always-nonnegative path from s to v of total weight $Value[v]$.

Base case: Before the loop begins, only $Value[s] = 0$.

Induction step: Suppose the induction hypothesis is true before the inner loop considers edge $(u \to v)$. Only $Value[v]$ may change. If $Value[v]$ changes to $Value[u] + w(u \to v)$, then by the induction hypothesis, there exists an always-nonnegative path from s to u of total weight at least $Value[u]$. Furthermore, since $Value[u] \geq 0$, we can continue the same path from u to v and it remains always-nonnegative.

Claim 3: If the shortest (fewest number of edges) always-nonnegative path from s to t has length at least $|V|$, then the graph has a positive cycle that is reachable from s via an always-nonnegative path. By stating this positive cycle is reachable from s via an always-nonnegative path, this means one full traversal of the cycle can be done without the total weight of the path becoming negative.

Proof of Claim 3: If the always-nonnegative path has length $\geq |V|$ it must contain a cycle, and this path must already traverse this cycle at least once because of its length. If the cycle is not positive, we can remove it and get an always-nonnegative path from s to t that has fewer edges and a higher total weight.

Claim 4: If the graph has a positive cycle that is reachable from s via an always-nonnegative path (meaning the always-nonnegative path can traverse the cycle once), then there is an always-nonnegative path from s to t.

Proof of Claim 4: By the premise, there is an always-nonnegative path from s to the cycle and through the cycle once (since that is how the algorithm finds these positive cycles); traverse the cycle an arbitrary number of times to reach an arbitrarily high positive value; then traverse any simple path from the cycle to t.

Claim 5: If the graph has a positive cycle that is reachable from s, the algorithm detects it. Only positive cycles that can be traversed once by an always-nonnegative path from s are detected.

Proof of Claim 5: The path to the cycle and the first time we go through the cycle until we close a loop are an always-nonnegative path of length at most $|V|$ from s to some vertex v. Therefore, by the time we exit the Main Loop, for every vertex v in the cycle, we have $Value[v] \geq 0$. Therefore one of the values on the cycle must be updated in the $|V|$-th iteration.

Running time: Like Bellman-Ford, $O(|V| \cdot |E|)$.

6. (??? **level**) **(Optional) Redemption for Homework 3**

Submit your *redemption file* for Homework 3 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