CS 170
Spring 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:

- *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 material. 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 grade.

Due Tuesday, February 14, at 11:59am

1. (★★ level) Dijkstra's Durability

For the following claims, answer yes or no and provide justification. Consider an arbitrary graph G with positive edge weights. Shortest path means least cost path.

(a) Suppose we want to know the distance from a node s to a node t_1, so we run Dijkstra's algorithm. Now suppose we also want to know the distance from s to another node t_2. Do we need to run the algorithm again?

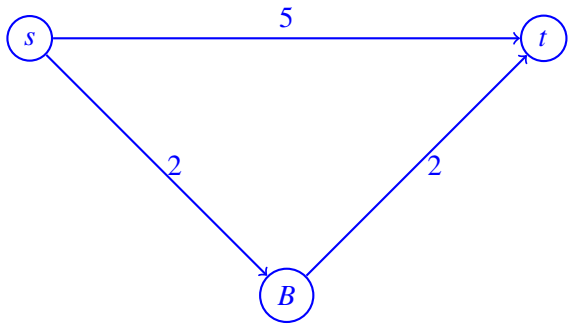
(b) Suppose we know the shortest path from a node s to another node t. Is the shortest path (the sequence of nodes, not the total cost) always the same if we add k to all edge weights, where k is an arbitrary positive number?

(c) Now answer the same question, except we multiply by k instead of adding k.

Solution:

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

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



(c) 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 weight 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.

2. (★★★★★ level) Biconnected Components

Let $G = (V, E)$ be a connected undirected graph. For any two edges $e, e' \in E$, we say that $e \sim e'$ if either $e = e'$ or there is a (simple) cycle containing both e and e'. We say a set of edges C is a *biconnected component* if there is some edge $e \in E$ such that $C = \{e' \in E | e \sim e'\}$.

(a) Show that two distinct biconnected components cannot have any edges in common. (Note: That this is equivalent to showing that if $e' \sim e_1$ and $e' \sim e_2$, then $e_1 \sim e_2$.)

(b) Associate with each biconnected component all the vertices that are endpoints of its edges. Show that the vertex sets corresponding to two different biconnected components are either disjoint or intersect in a single vertex. Such a vertex is a *separating vertex*.

Note that a separating vertex, if removed, would disconnect the graph. We will now walk you through how you can use DFS to identify the biconnected components and separating vertices of a graph in linear time. Consider a DFS tree of G.

(c) Show that the root of the DFS tree is a separating vertex if and only if it has more than one child in the tree.

(d) Show that a non-root vertex v of the DFS tree is a separating vertex if and only if it has a child v' none of whose descendants (including itself) has a backedge to a proper ancestor of v.

For each vertex u define $\texttt{pre}(u)$ to be the pre-visit time of u. For two vertices u and w, w is a **backcestor** of u iff the following two conditions both hold:

1. w is an ancestor of u
2. $\exists$ (back) edge (u,w) OR $\exists$ (back) edge (v,w), where v is some descendant of u in the DFS tree.

Define $\texttt{low}(u)$ to be the minimum possible value of $\texttt{pre}(w)$, where w is a backcestor of u.

Another way of stating the result of (d) is that a non-root vertex u is a separating vertex if and only if $\texttt{pre}(u) \leq \texttt{low}(v)$ for some child v of u.

(e) Give an algorithm that computes all separating vertices and biconnected components of a graph in linear time. (Hint: Think of how to compute `low` in linear time with DFS. Use `low` to identify separating vertices and run an additional DFS with an extra stack of edges to remove biconnected components one at a time.)

No need for a full 4-part solution. Just give a clear description of the algorithm, in plain English or psuedocode, and running time analysis.

As an example, part (a) is done for you.

a) Let B and C be two distinct biconnected components defined by $B = \{e' | e_1 \sim e'\}$ and $C = \{e' | e_2 \sim e'\}$, where e_1 and e_2 are distinct edges. Suppose B and C share edge f in common. Then $f \sim e_1$ and $f \sim e_2$. But this implies $e_1 \sim e_2$.

   To see why, note that is clearly true if $f = e_1$ or $f = e_2$. Otherwise, there is a simple cycle containing f and e_1 and a simple cycle containing f and e_2. The union of these two simple cycles contains a simple cycle that has both e_1 and e_2 as edges. (Convince yourself why. Drawing some visuals may help.)

   By a similar argument, if $e' \sim e_1$ then $e' \sim e_2$ because $e_1 \sim e_2$. Likewise, if $e' \sim e_2$ then $e' \sim e_1$. Thus, $B = C$, contradicting the fact that they are distinct sets.

Solution:

b) We argue that two biconnected components must share at most one vertex. For the sake of contradiction, assume that two components C_1 and C_2 share two vertices u and v. Note that there must be a path from u to v in both C_1 and C_2, since in each component, there is a simple cycle containing one edge incident on u and one edge incident on v. The union of these two paths gives a cycle containing some edges from C_1 and some from C_2. However, this is a contradiction as this would imply that an edge in C_1 is related to an edge in C_2.

   (Note: The name *separating vertex* hints that if we delete such a vertex, then we will disconnect the biconnected components it is common to. Indeed, let u be a shared vertex. Let (u, v_1) and (u, v_2) be the edges corresponding to u in the two components. Then we claim that removing u disconnects v_1 and v_2. If not, then there must be a path between v_1 and v_2, which does not pass through u. However, this

path, together with (u, v_1) and (u, v_2), gives a simple cycle containing one edge from each component which is a contradiction.)

c) Let v be the root of the tree. If u is the only child of v in the tree and (v, w) is some edge in the graph then $(v, u) \sim (v, w)$ because either $u = w$ or (v, u), P, (w, v) forms a simple cycle, where P is the path from u to w in the DFS tree. Thus, v is not contained in more than one biconnected component. If v has two children u_1 and u_2, then if $(v, u_1) \sim (v, u_2)$ there would be a simple path from u_1 to u_2 which cannot happen because there cannot be cross edges in an undirected graph. Thus, v is part of two distinct biconnected components.

We can also cite the disconnecting property of separating vertices. If the root has only one child, then it is effectively a leaf and removing the root still leaves the tree connected. The DFS from the first child explores every vertex reachable through a path not passing through the root. Also, since the graph is undirected, there can be no edges from the subtree of the first child to that of any other child. Hence, removing the root disconnects the tree if it has more than one children.

(Note to readers: Either approach above should be able to earn full points for part (c).)

d) Let v be a non-root vertex with predecessor u. Suppose v' is a child of v that does not have any descendants with backedges to ancestors of v. If $(u, v) \sim (v, v')$, then we have a simple cycle with ancestors of v, followed by v, followed by descendants of v'. Since this cycle closes, we would have a descendant of v' connected to an ancestor of v.

If no such v' existed, then any child v' of v would have some descendant w that had a backedge to an ancestor of v. We could take (v, v'), followed by a path from v' to w, followed by the backedge, and then back to (u, v) to get a simple cycle showing $(u, v) \sim (v, v')$. Thus, v is only part of one biconnected component.

Again, another way of looking at this is with the disconnecting view of separating vertices. If every child v' has some descendant with a backedge to an ancestor of v, each child can reach the entire tree above v so the graph is still connected after removing v. If there is a child v' such that none of its descendants have a backedge to an ancestor of v, then in the graph after removing v, there is no path between an ancestor of v and v' (note that there cannot be any cross edges since the graph is undirected).

(Note to readers: Likewise for (d), either approach should be able to earn full points.)

e) We first show how to compute the entire array of `low` values in linear time. While exploring each vertex u, we look at all the edges of the form (u, v) and can store at u, the lowest $\texttt{pre}(v)$ value for all neighbors of u. Then, $\texttt{low}(u)$ is given by the minimum of this value, $\texttt{pre}(u)$, and the `low` values of all the children of u. Since each child can pass its `low` value to the parent when its popped off the stack, the entire array can be computed in a single pass of DFS.

A non-root node u is a separating vertex iff $\texttt{pre}(u) \leq \texttt{low}(v)$ for any child v of u. This can be checked while computing the array. Also, if u is a separating vertex and v is a child such that $\texttt{pre}(u) \leq \texttt{low}(v)$, then the entire subtree with v as the root must be in different biconnected components than the ancestors or other children of u. However, this subtree itself may have many biconnected components as it might have other separating vertices.

Hence, we perform a DFS pushing all the edges we see on a stack. Also, when we explore a child v of a separating vertex u such that the above condition is met, we push an extra mark on the stack (to mark the subtree rooted at v). When DFS returns to v, i.e. when v is popped off the stack (of vertices), we can also pop the subtree of v from the stack of edges (pop everything till the mark). If the subtree had multiple biconnected components, they would be already popped off before the DFS returned to v.

3. (★★★ level) Alternative Factory

You have the unfortunate predicament that you frequently find yourself inexplicably trapped within factories,

a different one every time. Each factory consists of many small platforms, connected by a network of conveyor belts. Being an amateur surveyor, you can accurately calculate the time it takes to ride each conveyor belt with a quick glance. Some platforms have a button on them. Whenever you press one of these buttons, ALL conveyor belts in the factory reverse direction. A factory has one exit; you want to get there as quickly as possible. Give an efficient ($O((|V|+|E|)\log|V|)$ time) algorithm to find the fastest way out of the factory.

You only need to give a main idea and running time analysis. Be sure your main idea clearly and fully describes your algorithm.

For convenience, you can treat the factory as a graph, with the platforms being vertexes and the conveyor belts being directed edges. You know before running your algorithm the platform s at which you start, the location of the exit t, and the location of all buttons (think of this as a list of button locations). You can determine the time it takes to traverse the conveyor belt between any two platforms u and v via the function $\ell(u,v)$.

Notes in case you want to pick at the details:
-It takes a negligible amount of time to get from one end of a platform to the other.
-The time it takes to press a button is also negligible.
-You can't run up a conveyor belt the wrong way.
-You may assume $\ell(u,v) = \ell(v,u)$.
-Be warned that you should throw out your sense of spatial reasoning. You have seen all kinds of setups of conveyor belt configurations, sometimes with thousands of conveyor belts connected to a single platform.

(Hint: Try to construct a new graph that encodes the state of the conveyor belts. Perhaps your new graph will have twice as many vertexes as the original.)

Solution:

(i) **Main idea** To represent the state of reversed conveyor belts, we will make a copy of the graph, G^R, with all edges' directions reversed. We connect the original G to G^R with a zero-cost, bidirectional edge at each button location. Then we can simply run Dijkstra's algorithm on the combined graph.

Another possible solution would be to have the two copies of the graph intersect at the vertexes with buttons, so the buttons are shared nodes rather than edges.

(ii) **Running time analysis** Constructing G^R as well as the edges for the buttons, takes $\Theta(|V|+|E|)$ time. The combined graph has $2|V|$ vertexes and at most $2|E|+2|V|$ edges, so running Dijkstra's algorithm will take $O((|V|+|E|)\log|V|)$ time, which means the overall running time is $O((|V|+|E|)\log|V|)$.

4. (★★★ level) Shortest Path Oracle

Given a connected, directed graph $G=(V,E)$ with positive edge weights, and vertices s and t, you want to find the shortest path from s to t in a graph in $O(|E|+|V|)$ time. Normally, this wouldn't be possible; however, you have consulted the Order Oracle, who gave you a list L of the V vertices in increasing order of the shortest-path distance from s.

In other words, $L[1]$ is s, $L[2]$ is the closest other vertex to s, $L[|V|]$ is the farthest vertex from s, etc. Design an algorithm SHORTESTPATH(G,s,t,L) which computes the shortest $s-t$ path in $O(|E|+|V|)$ time.

Solution:

Main Idea: We will modify Dijkstra's algorithm to solve this problem. The oracle tells us the order in which we need to look at vertexes, so instead of using a priority queue we can just use L, and we won't have to update it.

Note: Similar results can be obtained by any of the following methods:

- Modify shortest path in a DAG and iterating through L instead of linearizing.
- Modify Bellman-Ford and *only* update distances for each edge out of $L[i]$, for each iteration of the inner loop (with outer loop over L at iteration i).
- Modify BFS and explore nodes with a queue in the order of L. Note that if you did this you also needed to track the distance from s to t in some way.

Pseudocode:

procedure SHORTESTPATH($G(V,E), s, t, L$)
  for all $u \in V$ **do**:
    dist(u) = ∞
    prev(u) = *nil*
  dist(s) = 0
  for $u \in L$ **do**:
    for all edges $(u,v) \in E$ **do**:
      if dist(v) > dist(u) + $\ell(u,v)$ **then**:
        dist(v) = dist(u) + $\ell(u,v)$
        prev(v) = u

Construct and output shortest path by following *prev* pointers, starting from t until reaching s, then reversing the order.

Proof of Correctness:
The proof of correctness is very similar to that of Dijkstra's. We have two loop invariants. (1) dist(u) for all $u \in V$ is the cost of the shortest path from s to u using only nodes which have already been visited. (2) prev(u) is the node preceding u in that same path to u.

Invariant (1) can be proved by inducting on the index in L. The base case $L[0]$ is simple since this will be the start vertex s. The inductive step follows from noting that the shortest path can only use nodes that occur earlier in L, by the definition of L. Further, the algorithm has considered all edges out of these nodes, so dist will be appropriately updated. Invariant (2) follows from the same line of reasoning, noting that we also update prev whenever we update dist.

Running time analysis:
We do not have to update a priority queue as we would in Dijksta's algorithm, so the analysis is more akin to BFS. The running time is dominated by the two for loops, but we note that we are looking at each vertex and each edge exactly once. For each edge, we do a constant amount of work, so the total running time is $\Theta(|V|+|E|)$.

5. (★★★★★ level) Archipelago Adventure

As a tourist visiting an island chain, you can drive around by car, but to get between islands you have to take the ferry. (more formally, our graph is $G = (V, R \cup F)$, where V is our set of locations (vertexes), R is the set of roads, and F is the set of ferry routes). A road is specified as an **undirected** edge $(u, v, C_{u,v})$ that connects the two points u, v with a **non-negative** (time) cost $C_{u,v}$. A ferry is specified as a **directed** edge $(u, v, C_{u,v})$ which connects point u to point v on another island (one way) with cost $C_{u,v}$. Some of the ferries are actually magical fairies so their costs **can possibly be negative!** Each ferry only runs one way and the routes are such that if there is a ferry from u to v, then there is no sequence of roads and ferries that lead back from v to u.
Assume we have the the adjacency list representation of roads and fairies (two separate data structures).
You are finished with your vacation so you want to find the shortest (least cost) route from your current

location s to the airport t. Design an efficient algorithm to compute the shortest path from s to t (given V, R, F, s and t as inputs). Your running time should be $O((|V|+|E|)\log|V|)$, where $E = R \cup F$.

Hint: Notice that the directed (ferry) edges are between different islands, each of which forms a separate (strongly) connected component.

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

- *Consider a directed graph with no negative cycles in which the only negative edges are those that leave some node s; all other edges are positive. Will Dijkstra's algorithm, started at s, work correctly on such a graph? Find a counterexample or prove your answer.*
- *Now consider the case in which the only negative edges are those that leave v, a vertex different from the start node s. Given two vertices s,t, find an efficient algorithm to compute the shortest path from s to t (should have the same asymptotic running time as Dijkstra's algorithm).*

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 $V \setminus S$, 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 s to t may or may not pass through v.
  To find a shortest path from s to t that does not pass through v, we delete v from the graph and run Dijkstra's algorithm from s. Let $d'_{s,t}$ be the length of the shortest path found.
  For the other case we need to
  - Find a shortest path from s to v. This can be done by running Dijkstra's algorithm from s. Since all the negative edges leave v and there is no negative cycle, Dijkstra's algorithm will correctly find a shortest path from s to v. Let this distance be $d_{s,v}$.
  - Find a shortest path from v to t. From Part (a) we know that this can be done by running Dijkstra's algorithm from v. Let this distance be $d_{v,t}$.

  The shortest distance is then given by $d_{s,t} = \min(d'_{s,t}, d_{s,v} + d_{v,t})$. 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 v 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 are already computed. For $u \notin A$, $dist(u)$ stores

the shortest $s \to u$ path among the paths that start at s, 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 roads as bi-directional edges. If two nodes are connected by roads, then they must be in the same SCC, and the additional constraint about ferries ensures that no ferry edge will be inside an SCC. Thus the edges across different SCCs are exactly the ferry 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 roads, 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, R, F, s, t)
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 to t.

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 to u.

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

Inductive Hypothesis: All vertices v in A have $dist(v)$ equal to the length of the shortest path from s to v, and all vertices v in V have $dist(v)$ equal to the length of the shortest path from s 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 road edge OR a non-negative ferry 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 ferry edge. Assume for contradiction there is some other path that we have not yet considered, but is actually shorter than $s \to \cdots \to u_{prev} \to u$. Because we sort our priority queue by topological sorting first, the other path must also consider some ferry edge to get to the new SCC $r(u)$; call this path $s \to \cdots \to v_{prev} \to v \to \cdots \to u$, and the corresponding ferry edge $(v_{prev} \to v)$.

We know that all of the edges in the new path $s \to \cdots \to v_{prev} \to v \to \cdots \to u$ after taking the flight

$v_{prev} \to v$ are roads and therefore must be non-negative; $s \to \cdots \to v_{prev} \to v$ must be shorter than the old path $s \to \cdots \to u_{prev} \to u$, and therefore has already been visited.

Because $s \to \cdots \to v_{prev} \to v \to \cdots \to u$ 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 ferry 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 $s \to \cdots \to u_{prev} \to u$. 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 $s \to \cdots \to u_{prev} \to u$ is the true shortest path for $s \to \cdots \to u$, 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 to w that uses only nodes in A as intermediate nodes. If the shortest according path $s \to w$ 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 $s \to \cdots \to u \to w$ (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 $s \to \cdots \to u$ 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 ferry edge. If (u,v) is the ferry edge, then the weight of (s,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, R, F, s, t)
Find and linearize the meta-graph (SCCs) of G.
Consider SCCs in topological order.
When considering the ith 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 to t.

Running time analysis:

The linearization and changing of edge weights are linear in $(|V| + |R| + |F|)$. Afterwards we run Dijkstra's algorithm, which dominates the overall running time, giving us $O((\log|V|)(|V| + |R| + |F|)$.

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 t, so it does not matter that total costs of paths to different sinks may be altered by different amounts.