CS 170 Fall 2014 — Algorithms — David Wagner — Soln 6

1. (20 pts.) Fundamental concepts

Decide whether each of the following claims is correct or not. If it is correct, write "True" and then provide a short proof (one or two sentences should be enough). If it is incorrect, write "False" and provide a counterexample (please make your counterexample as small as possible, for the readers' sake).

(a) Let G be a dag and suppose the vertices $v_1, \ldots, v_n$ are in topologically sorted order. If there is an edge (v_i, v_j) in G, then we are guaranteed that $i < j$.

Solution:

True. This is the definition of a topologically sorted order. In other words, all edges in the graph are from an earlier vertex to a vertex later in the order.

(b) Let G be a dag and suppose the vertices $v_1, \ldots, v_n$ are in topologically sorted order. If there is a path from v_i to v_j in G, then we are guaranteed that $i < j$.

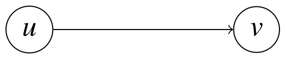
Solution:

True. Consider a path $v_{i_1} \to \cdots \to v_{i_k}$. From part (a) we conclude that $i_j < i_{j+1}$ for $j = 1, \ldots, k-1$, so by transitivity, $i_1 < i_k$.

(c) Let $G = (V, E)$ be a directed graph. If there is an edge (u, v) in G, then u and v must be in the same strongly connected component.

Solution:

False. Consider the following dag:



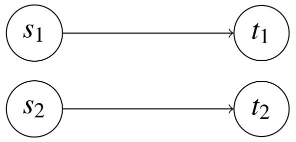
There is an edge between u and v, but they are not in the same strongly connected component.

(d) Let $G = (V, E)$ be a directed graph. If $s \in V$ is a source and $t \in V$ is a sink, then there is a path from s to t.

Solution:

False. The simplest possible example is just two vertices s and t and no edges. A source is defined to be a vertex with no incoming edges and a sink is a vertex with no outgoing edges. An isolated vertex is simultaneously both a source and a sink.

A slightly more complicated example is

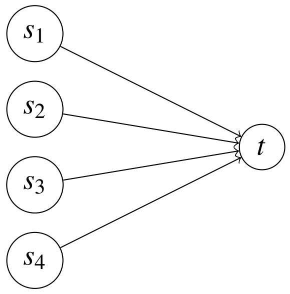


s_1 is a source, and t_2 is a sink, but there is no path connecting them.

(e) Let $G = (V,E)$ be a directed graph where every vertex has at most three outgoing edges. Then every vertex has at most three incoming edges.

Solution:

False, consider the following directed graph:



All vertices have at most one outgoing edge, but t has four incoming edges.

(f) Let $G = (V,E)$ be a dag. Then there are at most $|V|^2$ possible ways to order the vertices so they are in topologically sorted order.

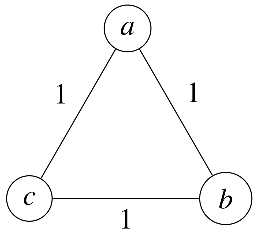
Solution:

False. Consider $G = (V,\emptyset)$, that is, a graph without edges. Any ordering of the vertices is a topological order. This gives us a total of $|V|!$ topological orders.

(g) Let G be a connected undirected graph with positive lengths on all the edges. Let s be a fixed vertex. Let $d(s,v)$ denote the distance from vertex s to vertex v, i.e., the length of the shortest path from s to v. If we choose the vertex v that makes $d(s,v)$ as small as possible, subject to the requirement that $v \neq s$, then every edge on the path from s to v must be part of every minimum spanning tree of G.

Solution:

False. Consider the following counterexample:



Take $s = a$. Both $v = b$ and $v = c$ minimize $d(a,v)$, but neither edge is part of every MST: for instance, (a,b) and (b,c) form a minimum spanning tree that does not contain (a,c).

(h) Let G be a connected undirected graph with positive lengths on all the edges, where no two edges have the same length. Let s be a fixed vertex. Let $d(s,v)$ denote the distance from vertex s to vertex v, i.e., the length of the shortest path from s to v. If we choose the vertex v that makes $d(s,v)$ as small as possible, subject to the requirement that $v \neq s$, then every edge on the path from s to v must be part of every minimum spanning tree of G.

Solution:

True. First let's analyze the the definition. We need to find a vertex v that minimizes $d(s,v)$. Because the edge weights are positive, v has to be a neighbor a s. Or in other words, part (h) claims the lightest edge that is incident on s has to part of every minimum MST. Let us call this edge e. Assume, for contradiction, that e is not part of some MST T. Let us add e to T. This creates a cycle, which goes through e to s and exit s through another edge e'. We now remove e'. Removing an edge from a cycle keeps the graph connected and a tree. This creates a tree which is lighter than T which contradicts our assumptions that T is a MST.

2. (20 pts.) BFS?

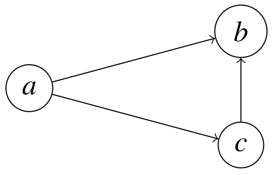
We previously used DFS and `post` times for topological sorting. What if we use BFS? In particular, consider the following minor modification to BFS:

```
BFS(G, s):
1.  For each $u \in V$:
2.      Set $\texttt{dist}(u) := \infty$.
3.  Set $\texttt{dist}(s) := 0$.
4.  Set $\texttt{clock} := 0$.
5.  Initialize $Q$ to a queue containing just $s$.
6.  While $Q$ is not empty:
7.      $u :=$ eject($Q$) (pop the front element off $Q$, and call it $u$)
8.      For each edge $(u,v) \in E$:
9.          If $\texttt{dist}(v) = \infty$:
10.             inject($Q,v$) (append $v$ to the end of $Q$)
11.             Set $\texttt{dist}(v) := \texttt{dist}(u) + 1$.
12.     Set $\texttt{post}(u) := \texttt{clock}$, and $\texttt{clock} := \texttt{clock} + 1$.
```

Let G be a directed acyclic graph. Suppose there is a source $s \in V$ such that every vertex in G is reachable from s. Suppose we run BFS(G, s), and then sort the vertices according to ~~decreasing~~ increasing `post`-value. Is this guaranteed to produce a valid topological sorting of the vertices? Either prove that the resulting algorithm is correct (i.e., that it is guaranteed to output a valid linearization of G), or show a small counterexample (a dag where the algorithm fails).

Solution:

No, this algorithm does not always produce a topologically sorted list of vertices. For example, consider the following graph:



If the algorithm starts with a and pushes b first into the queue, then the resulting order will be a,b,c. This order is not a topologically sorted order because the edge (c,b) goes from the last vertex to an earlier vertex in the order.

Comment: This problem illustrates why we spent so much time on depth-first search. For some problems—such as graph reachability—any graph traversal order works fine. But for some purposes, a depth-first search has special properties that are necessary to solve the problem. For instance, topological sorting and finding strongly connected components rely heavily on the special properties of a depth-first traversal order, and the standard algorithms don't work if you substitute some other traversal order for depth-first search.

Why does DFS work for topological sorting, when other methods such as BFS don't? When you use DFS, post-order has some lovely properties: because of the first-in-last-out properties of a stack, DFS won't finish processing a vertex until it finishes everything reachable from it. Therefore, intuitively, the post value for a vertex gives you some information about what other vertices could be reachable from it. Roughly, the higher the post value, the more nodes might be reachable from it. In some vague sense, it's this sort of property that makes reverse post-order a valid topological order, when you use DFS. In contrast, post-order doesn't have these lovely properties when you use BFS (because BFS doesn't use a stack), so post-order from a BFS traversal doesn't tell you very much about reachability.

So do we ever need anything *other* than depth-first search? Why did we study breadth-first search? Well, there are other problems where a depth-first traversal order is not helpful, but where some other traversal order enables elegant solutions to the problem. For instance, shortest paths in a graph where all edges are of unit length can be elegantly solved using breadth-first search, but good luck trying to compute shortest paths using any other traversal order.

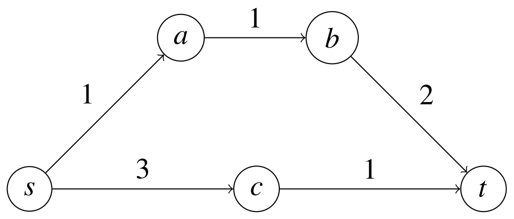
3. (25 pts.) Travel planning

You are given a set of n cities and an $n \times n$ matrix M, where $M(i,j)$ is the cost of the cheapest direct flight from city i to city j. All such costs are non-negative. You live in city A and want to find the cheapest and most convenient route to city B, but the direct flight might not be the best option. As far as you're concerned, the best route must have the following properties: the sum of the costs of the flights in the route should be as small as possible, but if there are several possible routings with the same total cost, then you want the one that minimizes the total number of flights. Design an efficient algorithm to solve this problem. Your algorithm should run in $O(n^2 \lg n)$ time.

(If you want, you can assume that the best route will require fewer than, say, 1000 flights, and that the cost of each flight is an integer number of dollars.)

A comment about all the solutions. In all the solution we build a complete graph whose vertices are the cities and the weight of the edge between city i and city j is given by $M(i,j)$. We will use the terms *edge* and *direct flight*, *path* and *route*, *weight* and *cost*, *vertex* and *city*, *shortest*, *cheapest* interchangeably.

Note that using Dijkstra's algorithm alone does not solve this problem. Consider the following graph. Dijkstra's algorithm will return the path $s \to a \to b \to t$, but the one with the smallest number of edges is just $s \to c \to t$.



Solution #1:

Main idea. We'll create a new matrix M' by adding a small penalty to the cost of each direct flight so that routes with more flights will cost more than routes with fewer flights. The penalty should be small enough that the cost of a cheapest route M' is still less than the cost of the next cheapest route in M.

Pseudocode.

```
1. Construct a new matrix $M'$ by $M'(i,j) = M(i,j) + 10^{-3}$.
2. Run Dijkstra's algorithm from A on $M'$.
3. Return the shortest path from A to B.
```

Correctness. The cost of a route in M' is the same as the cost in M plus $k \cdot 10^{-3}$, where k is the number of flights on the route. Flight costs integers in M. This means that the cost of any non-cheapest route is higher by at least 1 than the shortest path.

Since the number of flights on the best route is less than 1000, the cost in M' of the best route in M is increased by less than 1. This means that the shortest path in M' is the shortest path in M with the least number of flights.

Running Time. Creating M' takes $O(n^2)$, running Dijkstra takes $O((|V| + |E|) \lg |V|) = O(n^2 \lg n)$ and returning the path takes $O(|V| + |E|) = O(n^2)$. This gives a total of $O(n^2 \lg n)$.

Solution #2:

Main idea. We can modify Dijkstra's algorithm, using a slightly different definition of the distance between two vertices: the distance from s to v is a *pair*, namely, the pair (c,d), where c is the lowest possible cost of all routes from s to v, and d counts the minimum number of direct flights needed to get from s to v via an itinerary of cost at most c. In other words, (c,d) is the cost of the "best" route from s to v, with "best" defined as in the problem statement. The operations in Dijkstra's algorithm that compare and add distances then need to be modified accordingly.

Pseudocode.

```
1. Create a new matrix $M'$ by $M'(i,j) = (M(i,j),1)$. Or in other words, the new weight is the pair of the
   original weight and a number 1 indicating 1 direct flight.
2. Run Dijkstra's algorithm with the modified notion of distance as described next, from A on $M'$.
3. Return the shortest path from $A$ to $B$.
```

Detailed explanation. The difficulty here is that, not only do we want to find the shortest path but, if there are ties, we want to use the fewest number of hops. Fortunately, we can *still* use Dijkstra's algorithm, we just need to redefine our notion of "distance". Here's one easy way to accomplish this. We will think of the distance of a path as a pair (c,d), where c is the cost of the whole trip, and d is the number of hops in the trip. Now, we will define a comparison function $<$ as follows:

$$(c_1,d_1) < (c_2,d_2) := \begin{cases} \text{if } c_1 < c_2, & \text{return } \texttt{true} \\ \text{if } c_1 = c_2 \text{ and } d_1 < d_2, & \text{return } \texttt{true} \\ \text{otherwise}, & \text{return } \texttt{false} \end{cases}$$

We also replace the constants 0 and ∞ which we use to initialize `dist` to the pairs $(0,0)$ and (∞,∞), respectively.

Finally, if we have a path to u with distance (c,d) and we add the edge (u,v), then the cost of this longer path is obviously $(c + \ell(u,v), d+1)$, where $\ell(u,v)$ is the length of the edge between u and v (i.e., the cost of a flight from city u to v). Now that we have a new "algebra" to add and compare distances, we may simply use Dijkstra's algorithm as before but with our redefined operations. (Note that we will need to make sure to use a priority queue where the DeleteMin operation finds the element that is smallest under our new comparison operator.) And, because our new notion of distance favors fewer hops among equal-cost paths, intuitively it makes sense that Dijkstra's algorithm should find the shortest path with the fewest number of hops.

Correctness. This algorithm can be proven correct by mimicking the proof of correctness of Dijkstra's algorithm, as proven in the textbook. This gets a bit tedious.

One key step is to prove the following lemma:

Lemma 1 *Suppose $s \rightsquigarrow u$ is a path with distance (c,d). If we add an edge $u \to v$ to the end of this path, then the new path $s \rightsquigarrow v$ will be longer than the original; i.e., if the distance of $s \rightsquigarrow v$ is (c',d'), then $(c,d) < (c',d')$.*

Proof: Suppose the edge $u \to v$ is of dollar cost c''. Then $(c',d') = (c+c'', d+1)$. Dollar costs are positive, so $c < c+c'' = c'$, so $(c,d) < (c',d')$ by the definition of $<$. $\square$

I will leave it to you to verify that the rest of the standard proof goes through, with slight modifications, once this lemma is established. You would have needed to provide some argument in your solution about this.

Running time. We have a graph with n nodes and an edge for every city, so $|E| = n(n-1) = O(n^2)$. The running time is at most a constant factor larger than Dijkstra's algorithm, i.e., $O((|V|+|E|)\log|V|)$ (assuming you use a binary heap for the priority queue). So, for this particular graph the running time will be $O(n^2 \log n)$ as desired.

Solution #3:

Main idea. Run Dijkstra from the source to compute shortest paths and then remove edges so that the paths left are the shortest paths. At this point a simple BFS can find the shortest path with the least number of edges.

Pseudocode.

```
MinEdgeDijkstra($G = (V,E)$, $\ell$, $s$, $t$):
1.  dist := Dijkstra($G$, $\ell$, $s$).
2.  For each edge $(u,v) \in E$:
3.      Delete $e$ from $G$ if dist$(u) + \ell(u,v) >$ dist$(v)$.
4.  Run BFS on the modified graph.
5.  Return the min-edge path from $s$ to $t$ by following the BFS prev pointers.
```

Proof of correctness. Line 1 computes the shortest distance dist$(\cdot)$ from s to every other vertex. In lines 2-3 we break all non-shortest paths so that the paths from s to t are exactly the shortest paths from s to t[1]. Then we run BFS to find the shortest path with the smallest number of edges and return it.

Running time. This algorithm runs in time $O(n^2 \lg n)$.

Justification of running time. Dijkstra's algorithm takes $O((|V|+|E|\log|V|) = O(n^2 \lg n)$ time . Iterating over the edges take $O(|E|) = O(n^2)$, running BFS is also $O(|V|+|E|) = (n^2)$ and returning the shortest BFS path also takes $O(n^2)$ time. The total is $O(n^2 \lg n)$.

[1]For details, see see the official solution of problem 2 in homework 5.

Solution #4:

Main idea. Modify Dijkstra's algorithm to keep track of the smallest number of edges found in the shortest path.

Pseudocode. We record for each vertex the number of edges in the shortest path with the least number of edges. In lines 1-2 we initialize it similarly to how `dist`is initialized.

When each edge is processed, if it improves `dist` (lines 3-4), then the number of edges in the shortest path handled so far is set to `nedges`. If the current edge is part of a path that has the same distance as a previously handle path, then we set `nedges`to be the minimum of the two.

The pseudocode for the modification is given in Figure 1.

```
     for all $v \in V$:
         $\texttt{dist}(v) = \infty$
1.       $\texttt{nedges}(v) = \infty$
     $\texttt{dist}(s) = 0$
2.   $\texttt{nedges}(s) = 0$

     $H = \texttt{make-queue}(V)$
     while $H$ is not empty:
         $u = \texttt{delete-min}(H)$

         for all edges $(u,v) \in E$:

3.           if $\texttt{dist}(v) > \ell(u,v) + \texttt{dist}(u)$:
4.               $\texttt{nedges}(v) = \texttt{nedges}(u) + 1$
5.           else if $\texttt{dist}(v) > \ell(u,v) + \texttt{dist}(u)$:
6.               $\texttt{nedges}(v) = \min(\texttt{nedges}(v), \texttt{nedges}(u) + 1)$

             if $\texttt{dist}(v) > \ell(u,v) + \texttt{dist}(u)$:
                 $\texttt{dist}(v) = \texttt{dist}(u) + \ell(u,v)$
                 $\texttt{decrease-key}(H, v)$
```

Figure 1: Modified Dijkstra's Algorithm for solution #4. The numbered lines are the lines added to the regular Dijkstra's algorithm.

Finally, we trace back the min-edge shortest path by choosing an incoming edge with both $\texttt{dist}(u) = \texttt{dist}(v) + l(u,v)$ and $\texttt{nedges}(u) = \texttt{nedges}(v) + 1$.

Proof of correctness. Because this algorithm does not alter the computation of `dist`or which vertices are pushed into the heap, the modification does not change `dist`.

We will prove the following invariant. This is similar to the proof of correctness for Dijkstra's algorithm.

Invariant 1 *Let $R = V \setminus H$, that is the set of vertices already deleted from the queue. At the end of each iteration of the while loop, the following condition holds: for every node $u \in R$, the value $\texttt{nedges}(u)$ is the smallest number of edges in the shortest path from s to u.*

Proof: By induction over the number of iterations. The base case is trivial $S = \{s\}$ at the end of the first iteration. Assume it is true for the first n iterations and consider the $n+1$ iteration. When u is removed from the queue, all of its incoming edges from vertices whose distances from s are smaller have already been handled and therefore `nedges`(v) is the 1 more than the minimum `nedges`(u) for any incoming edge (u,v) that is part of a shortest path to v. $\Box$

Running time. This algorithm runs in time $O((|V| + |E| \log |V|) = O(n^2 \lg n)$. The modification just adds constant time for each edge and for each vertex.

Comments: It is much easier to prove answer 1 and 3, because we don't need to re-prove Dijkstra's algorithm correct; we can just use it as a subroutine that's already known to be correct.

Answer 3 is also easy because did all the heavy work in showing how to break non-shortest paths in problem 2 of homework 5. It is also more general than answer 1, because it does not depend on the extra assumptions that edge weights are integers and that the best path has less than a thousand edges.

Answer 2 is harder to prove correct, because we need to re-prove the correctness of Dijkstra's algorithm. However, Answer 2 has the advantage of being more generic than 1 and also shows Dijkstra's algorithm actually works correctly with any notion of distance that is additive, totally ordered, and satisfies Lemma 1.

Answer 1 can be made a bit more general, by replacing the penalty 10^{-3} with $1/|V|$. Any shortest path from s to v must visit at most $|V|$ vertices (no vertex can be visited twice, because this would create a cycle, and then we could just bypass the cycle and get a shorter path), so uses at most $|V| - 1$ edges. This eliminates the need to assume that the shortest path involves at most 1000 flights.

All answers can be sped up to run in $O(n^2)$ time, by using a slightly different data structure for the priority queue. In fact, it suffices to use an unsorted list of elements, with a pointer from each vertex v to the element associated with v. Then Insert and DecreaseKey can be done in $O(1)$ time, and DeleteMin can be implemented in $O(|V|) = O(n)$ time by scanning the entire list to find the minimum. Since Dijkstra's algorithm uses $O(|V|)$ Insert and DeleteMin calls and $O(|E|)$ DecreaseKey calls, the total running time is $O(n^2)$ with this data structure. For very dense graphs, this data structure leads to a small speed improvement; but for sparse graphs, the standard priority queue data structure based upon a binary heap is best.

4. (25 pts.) Road network design

There is a network of roads $G = (V,E)$ connecting a set of cities V. Each road $e \in E$ has an associated (non-negative) length $\ell(e)$. We can build *one* new road, and there are a bunch of candidates for where this road might go. As a designer for the public works department, you are asked to determine which of these candidates, if added to the existing network G, would result in the maximum decrease in the driving distance between two fixed cities s and t. Design an $O((|V| + |E|) \log |V|)$ time algorithm for this problem.

In particular, the problem is:

Input: an undirected graph G with non-negative edge lengths $\ell : E \to \mathbb{R}_{\geq 0}$, and two vertices s,t, and a list $(a_1,b_1), \ldots, (a_n,b_n)$ of pairs of vertices

Output: the index i such that adding the edge (a_i,b_i) reduces the distance from s to t as much as possible

Solution #1: If the distance between s and t decreases with the addition of $e' = (u,v)$, the new shortest path from s to t will be either (1) the concatenation of the shortest path from s to u, the edge (u,v) and the shortest path from v to t, or (2) concatenation of the shortest path from s to v, the edge (u,v) and the shortest path from u to t. The length of this path will be

$$\min(d(s,u) + \ell(u,v) + d(v,t), d(s,v) + \ell(u,v) + d(u,t)),$$

where $d(x,y)$ denotes the distance from x to y in the original graph G (without adding any new edge). We can compute the length of this path by running Dijkstra's algorithm once from s and once from t. With all the shortest path distances from s and t, we can evaluate in constant time the expression shown above for the length of the shortest path from s to t going through e' for any $e' \in E'$. The shortest of these paths will give us the best edge to add and its length will tell us what improvement the addition brings, if any.

The running time of this algorithm is $\Theta(T(|V|,|E|)+|E'|)$ where $T(|V|,|E|)$ is the running time of Dijkstra's algorithm. When $|E'| = O(|V|+|E|)$, the total running time is $O((|V|+|E|)\lg|V|)$ using a binary heap.

Solution #2: Construct a directed graph $\vec{G} = (\vec{V},\vec{E})$ from the undirected graph $G = (V,E)$ and the potential edges E'. For every vertex $v \in V$, create two copies $v_0, v_1 \in \vec{V}$. For every (undirected) edge $(u,v) \in E$, create corresponding (directed) edges (u_0,v_0), (v_0,u_0), $(u_1,v_1),(v_1,u_1) \in \vec{E}$ having same weight. Finally, for every (undirected) potential edge $(u,v) \in E'$, create (directed) edges $(u_0,v_1),(v_0,u_1) \in \vec{E}$ having the same weight, to point from the 0-copy to the 1-copy.

Now the shortest path from s_0 to t_1 in $\vec{G}$ corresponds to the shortest path from s to t in G, using a potential edge *exactly once*. Dijkstra's algorithm then solves the problem immediately. The running time is $\Theta(T(|V|,|E|+|E'|))$. When $|E'| = O(|V|+|E|)$, the running time is $O((|V|+|E|)\lg|V|)$ using a binary heap.

5. (10 pts.) MSTs for directed graphs

Kruskal's algorithm takes as input an *undirected* graph that's connected, and finds a connected subgraph of minimum weight. But suppose we have a *directed* graph that's strongly connected, and we want to find a strongly connected subgraph of minimum weight. Will Kruskal's algorithm work for this task?

In particular, consider the following algorithm:

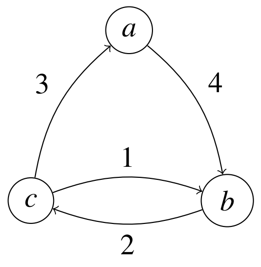
```
ModifiedKruskal($G = (V,E)$, $w$):
1.  Set $X := \{\}$ (the empty set).
2.  Sort the edges by weight.
3.  For each edge $(u,v) \in E$, in increasing order of weight:
4.      If there is a path from $u$ to $v$ using only edges in $X$, do nothing, else add $(u,v)$ to $X$.
5.  Return $X$.
```

Suppose we are given a *directed* graph $G = (V,E)$ that is strongly connected, with weights $w : E \to \mathbb{R}$ on the edges. Suppose we run ModifiedKruskal(G, w) and get the result X. Define the graph G^* by $G^* = (V,X)$; in other words, G^* has the edges returned by the above algorithm. Are we guaranteed that G^* has the minimal possible total weight, out of all possible graphs $G' = (V,E')$ such that $E' \subseteq E$ and G' is strongly connected? Either prove that the answer is yes, or give a small counterexample to show that the answer is no.

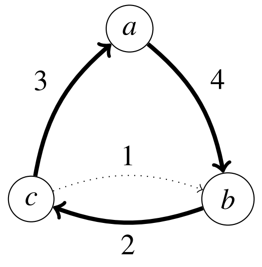
Solution:

No, it will not work. Consider the weighted graph G shown below. The correct minimum cost spanning subgraph is G_C, with cost 9. ModifiedKruskal adds the edges in increasing weight. The first edge added is (c,b), from which there is no recove, since (c,b) is not required for connectivity; ultimately it outputs the graph G_k shown below with cost 10, which is not minimal.

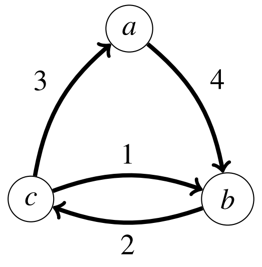
Example graph G with the correct min cost graph G_C and the graph produced by ModifiedKruskal G_K.



(a) Weighted graph G



(b) Weighted graph G_C



(c) Weighted graph G_K

Comment: If we make a similar modification to Prim's algorithm, it too fails. ModifiedPrim would add edge (c,b) if vertex c is processed first, and thus fail in a similar way.

This hints that if we want to find a minimal strongly connected subgraph of a directed graph, we really need some fundamentally new approach, not just a tiny tweak to Prim's or Kruskal's.

6. **(0 pts.) Crazy hard optional problem: Two vertex-disjoint paths**

(This is an *optional* bonus challenge problem. Only solve it if you want an extra challenge.)

Find a polynomial-time algorithm to solve the following problem:

Input: A dag G, and vertices s_1, s_2, t_1, t_2.

Question: Does there exist a path from s_1 to t_1 and a path from s_2 to t_2, such that no vertex appears in both paths?

Your algorithm should have running time $O(|V|^c)$ for some constant c (e.g., $c = 4$ or something like that).

Don't submit a solution. We won't grade it. Health warning: This problem may be brain-meltingly hard.

Solution: Linearize the graph, and label each vertex with the index in which it appears in the topologically sorted list (e.g., the first vertex in the list is labelled 1, and so on).

Make two copies of the graph, and imagine the following solitaire game. Initially there's a marker in the first copy of the graph, placed on the vertex s_1, and there's a marker in the second copy of the graph, placed on the vertex s_2. At each turn, you are allowed to move *one* of the two markers along an edge, as long as the move obeys both of the following rules:

1. Only the marker on the "lower-numbered" vertex is allowed to move. For instance, if the first marker is sitting on a vertex labeled 7 and the second marker is sitting on a vertex labeled 5, you must move the second one (not the first).
2. You are not allowed to move a marker onto a copy of the same vertex that the other marker is sitting on.

Your goal is to get the marker in the first graph to t_1 and the marker in the second graph to t_2.

Can you win this solitaire game? We will prove that you can win, if and only if the answer to the original question is "yes."

Can we build an algorithm to check whether it's possible to win the game? Yes, we can, by constructing a new graph $G^* = (V^*, E^*)$ whose vertices represent the location of both markers, and where edges represent how the markers can move in a single turn of the solitaire game. We can then use depth-first search in G^* to determine whether the vertex $\langle t_1, t_2 \rangle$ is reachable from the vertex $\langle s_1, s_2 \rangle$. The running time is then $\Theta(|V^*| + |E^*|) = \Theta(|V^2| + |V||E|) = \Theta(|V||E|)$.

Details. Let's make this more precise. Define $l(v)$ to be the label on vertex v. In other words, if the topological sort returns $v_1, v_2, \ldots, v_n$, then we'll define $l(v_1) = 1$, $l(v_2) = 2$, and so on.

Define the graph G^* as follows. Its vertex set is

$$V^* = \{\langle v, w \rangle : v, w \in V, v \neq w\},$$

so each vertex of G^* is a pair of distinct vertices from G. Its edge set is

$$\begin{aligned} E^* = &\{(\langle v, w \rangle, \langle v', w \rangle) : l(v) < l(w), (v, v') \in E\} \cup \\ &\{(\langle v, w \rangle, \langle v, w' \rangle) : l(v) > l(w), (w, w') \in E\}. \end{aligned}$$

We can see how this corresponds to the solitaire game introduced informally above. We run DFS in G^*, starting from the vertex $\langle s_1, s_2 \rangle \in V^*$. If $\langle t_1, t_2 \rangle$ is reachable, then we answer "yes" to the original question, otherwise we answer "no." All that remains is to prove that this algorithm is correct. We do this next.

Proof of correctness.

Lemma 2 *Suppose there exists a path $s_1 = a_1 \to a_2 \to \cdots \to a_m = t_1$ and a path $s_2 = b_1 \to b_2 \to \cdots \to b_n = t_2$ in G, such that no vertex appears in both paths. Then there exists a path from $\langle s_1, s_2 \rangle$ to $\langle t_1, t_2 \rangle$ in G^*.*

Proof: Define $v_1, \ldots, v_{n+m}$ and $w_1, \ldots, w_{n+m}$ iteratively, as follows. First, $v_1 = a_1$ and $w_1 = b_1$. Also, if $l(v_i) < l(w_i)$, then define v_{i+1} to be the next vertex that appears in the path $a_1 \rightsquigarrow a_m$ immediately after v_i, and define $w_{i+1} = w_i$. Alternatively, if $l(v_i) > l(w_i)$, then symmetrically define $v_{i+1} = v_i$ and define w_{i+1} to be the next vertex that appears in the path $b_1 \rightsquigarrow a_n$ immediately after w_i.

Note that this sequence is well-defined. Because the paths $a_1 \rightsquigarrow a_m$ and $b_1 \rightsquigarrow a_n$ have no vertex in common, we are guaranteed $v_i \neq w_i$ for all i, and consequently $l(v_i) \neq l(w_i)$ (this can be proven by induction on i). Also, $v_{n+m} = a_m$ and $w_{n+m} = b_n$, so this path ends at $\langle t_1, t_2 \rangle$, as claimed. □

Lemma 3 *Suppose that there exists a path from $\langle s_1, s_2 \rangle$ to $\langle t_1, t_2 \rangle$ in G^*. Then there exists a path $s_1 \rightsquigarrow t_1$ and a path $s_2 \rightsquigarrow t_2$ in G, such that no vertex appears in both paths.*

Proof: Let the path in G^* be

$$\langle s_1, s_2 \rangle = \langle v_1, w_1 \rangle \to \langle v_2, w_2 \rangle \to \cdots \to \langle v_q, w_q \rangle = \langle t_1, t_2 \rangle.$$

Define $a_1, \ldots, a_m$ to be the sequence of vertices output by the following algorithm:

```
1. For $i := 1, 2, \ldots, q$:
2.     If $i = 1$ or $v_i \neq v_{i-1}$: output $v_i$.
```

In other words, $a_1, \ldots, a_m$ is the result of removing all repeated vertices from the sequence $v_1, \ldots, v_q$. Similarly, define $b_1, \ldots, b_n$ to be the result of removing all repeated vertices from the sequence $w_1, \ldots, w_q$.

Obviously, $a_1 = v_1 = s_1$ and $b_1 = w_1 = s_2$. Since each edge $\langle v_i, w_i \rangle \to \langle v_{i+1}, w_{i+1} \rangle$ changes either the first component or second component (but not both), we find that $m + n = q$ and therefore $a_m = v_k = t_1$ and $b_n = w_k = t_2$, so this yields a path from s_1 to t_1 in G and a path from s_2 to t_2 in G.

Moreover, we can prove that these two paths are vertex-disjoint. Suppose there is some vertex x that is visited by both paths. We will show that this implies a contradiction. Let i be the smallest index where either $v_i = x$ or $w_i = x$; such an i must exist. Suppose without loss of generality it is $v_i = x$ (the other case is symmetrical). Then we know $w_i \neq x$, since by the definition of V^*, $\langle x, x \rangle \notin V^*$.

Since the labels on the vertices of any path in G are strictly increasing, we know that the labels $l(v_1), \ldots, l(v_{i-1})$ are all $< l(x)$. For a similar reason, $l(w_i) < l(x)$. Let j be the last index such that $l(w_j) < l(x)$. We see that $j \geq i$, and

$$l(w_1) \leq \cdots \leq l(w_j) < l(x) = l(v_i).$$

Therefore, none of $w_1, w_2, \ldots, w_j$ are equal to x.

Also, the same equation tells us that at states $\langle v_i, w_i \rangle, \ldots, \langle v_j, w_j \rangle$, the second marker is the one that moves, so in fact $v_i = \cdots = v_j$. Now consider the transition $\langle v_j, w_j \rangle \to \langle v_{j+1}, w_{j+1} \rangle$. We've seen that $l(w_j) < l(x) = l(v_j)$, so the second marker is the one that moves, and $v_{j+1} = v_j$. Moreover, since j was the last index such that $l(w_j) < l(x)$, we must have $l(w_{j+1}) > l(x)$ (it cannot be equal, because that would violate the second rule of the solitaire game, or equivalently, because that would take you to $\langle x, x \rangle$, which is not an element of V^*). Now since the levels of the vertices on a path are strictly increasing, it follows that all of $w_{j+1}, w_{j+2}, \ldots, w_q$ have labels larger than $l(x)$, i.e., none of $w_{j+1}, w_{j+2}, \ldots, w_q$ are equal to x.

All in all, we have shown that x does not appear in the sequence $w_1, \ldots, w_q$. This means that x does not appear in the sequence $b_1, \ldots, b_n$, either. However, this contradicts our initial assumption that $b_k = x$.

Therefore, the only remaining possibility is that there is no vertex that is visited by both paths. $\square$