Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Dijkstra's Algorithm Fails on Negative Edges

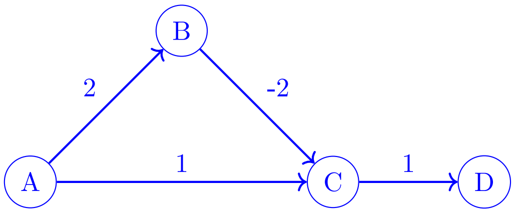
Draw a graph with five vertices or fewer, and indicate the source from which you would start Dijkstra's algorithm.

(a) Draw a graph with no negative cycles for which Dijkstra's algorithm produces the wrong answer.

(b) Draw a graph with at least two negative weight edges for which Dijkstra's algorithm produces the correct answer.

Solution:

1. Here's one example:



Dijkstra's algorithm from source A will give the distance to D as 2 rather than 1, because it visits C before B.

2. Dijkstra's algorithm always works on directed paths. For example:

$$\begin{tikzcd} A \arrow[r, "-1"] & B \arrow[r, "-2"] & C \end{tikzcd}$$

2 Dijkstra's Tiebreaking

We are given a directed graph G with positive weights on its edges. We wish to find a shortest path from s to t, and, among all shortest paths, we want the one in which the longest edge is as short as possible. How would you modify Dijkstra's algorithm to this end? Just a description of your modification is needed.

Note: if there are multiple shortest paths where the longest edge is as short as possible, outputting any of them is fine.

Solution: Modify Dijkstra's algorithm to keep a map $\ell(v)$ which holds the longest edge weight on the current shortest path to v. Initially $\ell(s) := 0$ for source s and $\ell(v) := \infty$ for all $v \in V\ \{s\}$. When we consider a vertex u and its neighbour v, if $dist(u) + w(u, v) < dist(v)$, then we set $\ell(v) := \max(\ell(u), w(u, v))$ in addition to the standard Dijkstra's steps. If there is a neighbour v of u such that $dist(v) = dist(u) + w(u, v)$, *and* $\ell(v) > \max(\ell(u), w(u, v))$, we set v's predecessor to u and update $\ell(v) := \max(\ell(u), w(u, v))$.

Note: An alternate solution that doesn't get full credit is, letting $W = \max_e w(e)$ and assuming all edge weights are integers, to add e.g. $n^{-1-(W-w(e)}$ to e's edge weight. Using these edge weights, of all the shortest paths, the one with the shortest possible longest edge will have the least weight, and no path can become shorter than a path it was previously longer than. However, writing down the edge weights with $n^{-1-(W-w(e)}$ added to them takes an additional $O(W \log n)$ bits per edge weight. As we saw in the last discussion, algorithms with runtime polynomial in the weights of edges are **not** efficient in general.

3 Waypoint

You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights, and there is a special node $v_0 \in V$. Give an efficient algorithm that computes, for all node pairs s, t, the length of the shortest path from s to t that passes through v_0. Your algorithm should take $O(|V|^2 + |E| \log |V|)$ time.

Hint: you should only need 2 calls to Dijkstra's.

Solution: The length of the shortest path from s to t that passes through v_0 is the same as the length of the shortest path from s to v_0 plus the length of the shortest path from v_0 to t.

We compute the shortest path length from v_0 to all vertices t using Dijkstra's. Next, we reverse all edges in G, to get G^R, and then compute the shortest path length from v_0 to all vertices in G^R. The shortest path length from v_0 to s in G^R is the same as the shortest path length from s to v_0 in G.

These two calls to Dijkstra's take $O((|V| + |E|) \log |V|)$ time. To find the lengths of the shortest paths going through v_0 between all pairs, we iterate over the results of the two calls to Dijkstra's, and this takes $O(|V|^2)$ time.

4 Shortest Path Between Sets

Given a undirected weighted graph G with non-negative edge weights $w(\cdot)$, and let $d(s, t)$ be the shortest path length from s to t. Give an efficient algorithm that takes as input two subsets of vertices S and T and outputs $\min_{s \in S, t \in T} d(s, t)$, i.e. the shortest path from any vertex in S to any vertex in T. Your algorithm should take $O(|E| \log |V|)$ time.

Hint: create an extra dummy node so that you can find all relevant distances by running Dijkstra's from there.

Solution:

Solution 1:

Main idea: Take G and add a new vertex s^*, and add a weight 0 edge from s^* to every vertex in S. Run Dijkstra's starting from s^*, and then given the output $dist$, return $\min_{t \in T} dist(t)$.

Correctness: For any path from a vertex S to a vertex T, there is a path of the same length from s^* to the same vertex in T, in vice-versa. $\min_{t \in T} dist(t) = \min_{s \in S, t \in T} d(s, t)$.

Runtime: Running Dijkstra's takes $O((|V| + |E|) \log |V|)$.

Solution 2:

Main idea: We modify Dijkstra's, so that rather than initializing $dist(s) = 0$ for a single vertex, it initializes $dist(s) = 0$ for all $s \in S$, and also updates $dist(v)$ for all v that are neighbors of vertices in

S. Our priority queue is initialized with all vertices in $V \setminus S$ instead of $V \setminus s$. Then, we run Dijkstra's as normal.

Correctness: This can be seen as equivalent to what Solution 1 does, since Solution 1 will start by setting $dist(s) = 0$ for all $s \in S$ anyway.

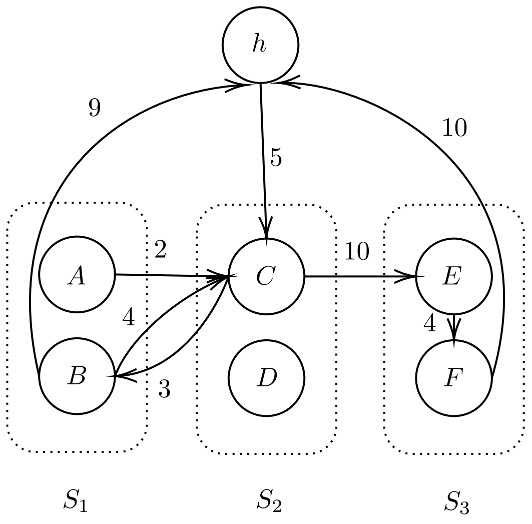
Runtime: Running Dijkstra's takes $O((|V| + |E|) \log |V|)$.

5 Running Errands

You need to run a set of k errands in Berkeley. Berkeley is represented as a directed weighted graph G, where each vertex v is a location in Berkeley, and there is an edge (u, v) with weight w_{uv} if it takes w_{uv} minutes to go from u to v. The errands must be completed in order, and we'll assume the ith errand can be completed immediately upon visiting any vertex in the set S_i (for example, if you need to buy snacks, you could do it at any grocery store). Your home in Berkeley is the vertex h.

Given G, h, and all $(S_i)_{i=1}^k$ as input, give an efficient algorithm that computes the least amount of time (in minutes) required to complete all the errands starting at h. That is, find the shortest path in G that starts at h and passes through a vertex in S_1, then a vertex in S_2, then in S_3, etc.

For instance, in the graph below, the shortest such path is $h \to C \to B \to C \to E$ and the time needed is $5 + 3 + 4 + 10 = 22$.



Hint: try creating copies of the graph G to help "keep track of" the errands you've completed so far.

Solution:

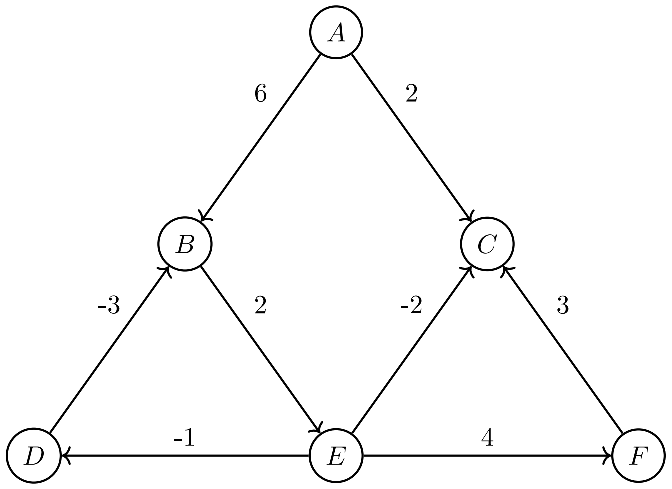
Main idea: Create $k+1$ copies of G, called $G_0, G_1, ...G_k$, to form G'. Let the copy of v in G_i be v_i. For every v in S_i, we add an edge from v_{i-1} to v_i with weight 0. We run Dijkstra's starting from h_0 in G', and output the shortest path length to any vertex in G_k.

Correctness: Any path in G' from h_0 to a vertex G_k can be mapped to a path in G of the same length passing through vertices, by taking each edge (u_i, v_i) and replacing it with the edge (u, v) in G, ignoring edges of the form (v_{i-1}, v_i). These paths must also complete the errands in order, since they must contain edges of the forms $(v_0, v_1), (v_1, v_2), \ldots$ in that order.

Runtime analysis: This takes time $O(k(|V| + |E|) \log k|V|)$ since the new graph is k times the size of the original graph.

6 Bellman-Ford

Consider the graph below.



(a) When running the Bellman-Ford algorithm, at most how many times do we go through all the edges to determine the single source shortest paths? How can we tell if there is a negative cycle?

Solution: We iterate $|V| - 1 = 5$ times through all the vertices. We can iterate a 6th time; if distances update, then there is a negative cycle.

(b) Suppose we ran Bellman-Ford on the graph above and we process edges in the following order:

$$(A, B), (A, C), (B, E), (D, B), (E, C), (E, D), (E, F), (F, C).$$

Fill out the table below with the distances of each node from A after each iteration of Bellman-Ford. Is there a negative cycle in the graph above?

Iteration	**A**	**B**	**C**	**D**	**E**	**F**
Start	0	∞	∞	∞	∞	∞
1						
2						
3						
4						
5						
6						

Solution:

We start off with an array where the distance to A is 0 from A, and the distance to every other node from A is ∞.

Then, we iterate through the edges in the order provided in the problem, and relax distances throughout. We arrive at the following distances table:

Iteration	**A**	**B**	**C**	**D**	**E**	**F**
Start	0	∞	∞	∞	∞	∞
1	0	6	2	7	8	12
2	0	4	2	7	8	12
3	0	4	2	5	6	10
4	0	2	2	5	6	10
5	0	2	2	3	4	8
6	0	0	2	3	4	8

Since the edges updated at the 6th iteration, there is a negative cycle present in the graph.

7 Restaurant Orders

Andrew is the sole chef at CS 170 Diner, and today he is handling a flurry of orders from n customers. Thankfully, each customer only ordered 1 dish, and he knows that it takes c_i minutes to cook the meal for customer i ($1 \leq i \leq n$). However, Andrew is very bad at multitasking and can only cook one meal at a time. To best satisfy the hungry customers, Andrew is trying to figure out the best way to process all the orders to minimize the total wait time.

More formally, let v_i be the time at which customer i gets their food. Please help Andrew determine an efficient algorithm that finds the minimum $\sum_{i=1}^{n} v_i$ over all ways to fulfill the n orders. Please provide an efficient algorithm and runtime analysis; a proof of correctness is not required.

Solution:

Algorithm: We continuously choose the shortest remaining meal to cook, until we have fulfilled all of the meals. Let t_i be the length of the ith shortest meal. Then, our answer is $\sum_{i=1}^{n}(n-i+1)t_i$.

Proof of Correctness: Note that if we cook meal a_1 first, then a_2, etc., then all n people have to wait for meal a_1, and then $n-1$ people have to wait for a_2, and so on. Therefore, for this ordering, the total wait time over everyone is $\sum_{i=1}^{n}(n-i+1)c_{a_i}$. Therefore, we want to have the shortest meal times first: if for $i < j$ we have $c_{a_i} > c_{a_j}$, then swapping meals a_i and a_j in the ordering will give us a total saving of wait time of $(j-i)(c_{a_i} - c_{a_j})$. So, we sort the customers by their meals' cooking time, and then serve in that order.

Runtime: It takes $O(n \log n)$ to sort the meals by cooking length, and then $O(n)$ time to take this summation. So, in total, it takes $O(n \log n)$ time.

8 Longest Huffman Tree

Under a Huffman encoding of n symbols with frequencies $f_1, f_2, \ldots, f_n$, what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case, and argue that it is the longest possible.

Solution: The longest codeword can be of length $n-1$.

An encoding of n symbols with $n-2$ of them having probabilities $1/2, 1/4, \ldots, 1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value.

No codeword can ever by longer than length $n-1$. To see why, we consider a prefix tree of the code. If a codeword has length n or greater, then the prefix tree would have height n or greater, so it would have at least $n+1$ leaves. Our alphabet is of size n, so the prefix tree has exactly n leaves.