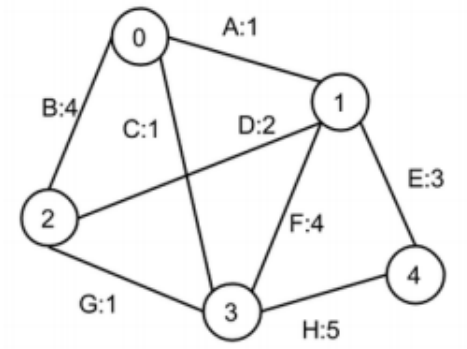
CS 61B Spring 2018

Shortest Paths and MST

Exam Prep 12: April 9, 2018

1 Warmup with MST and SP

(a) For the graph below, use Kruskals algorithm to find the MST. The number on each edge is the weight, and the letter is a unique label you should use in your answer to specify that edge. **Provide the edges in the order theyd be inserted into the MST by Kruskals algorithm**. Break any ties using the alphabetical label. Use the blanks below. You may not need all blanks. **Give your answer in terms of the alphabetical labels**, e.g. if Kruskals starts with the edge between vertices 3 and 4, you would write H in the first blank.

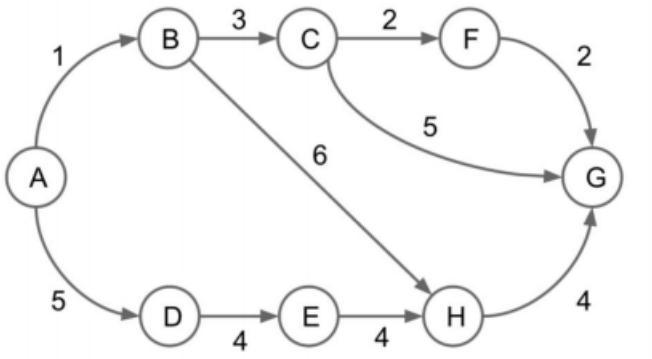


–A– –C– –G– –E–

(b) Repeat part a, but using Prims algorithm, starting from vertex #3. **As before, give your answer in the order added to the MST and in terms of the alphabetical labels**. You may not need all of the blanks.

–C– –A– –G– –E–

(c) For the graph below, give the order in which Dijkstras Algorithm would visit each vertex, starting from vertex A.



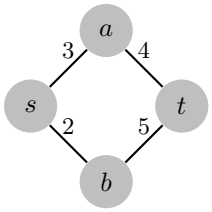
–A– –B– –C– –D– –F– –H– –G– –E–

2 Conceptual MST and SP

Answer the following questions regarding MSTs and shortest path algorithms for a **weighted, undirected graph**. If the question is T/F and the statement is true, provide an explanation. If the statement is false, provide a counterexample.

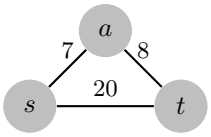
(a) (T/F) If all edge weights are equal and positive, breadth-first search starting from node A will return the shortest path from a node A to a target node B.
True. Breadth-first search finds shortest paths in an unweighted graph. Suppose all the weights were equal to w. If you divide all of the weights by w, then the edge weights are all 1, which can be thought of as an unweighted graph. BFS will return the shortest path from node A that is w dist away, then $2w$ dist, then so on.

(b) (T/F) If all edges have distinct weights, the shortest path between any two vertices is unique.



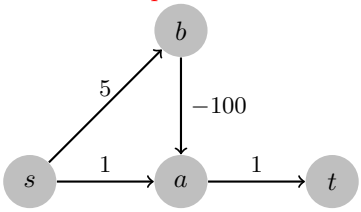
False. Consider the following diamond graph:

(c) (T/F) Adding a constant positive integer k to all edge weights will not affect any shortest path between vertices.
False. Adding a constant to all the edge weights when calculating shortest paths means we start favoring paths that use fewer edges. As a concrete example:



Adding 10 to each edge weight would change the shortest path from initially $s \to a \to t$ to $s \to t$.

(d) Draw a weighted graph (could be directed or undirected) where Dijkstra's would incorrectly give the shortest paths from some vertex.
The correctness of Dijkstra's algorithm relies on the fact that when you visit a vertex, you have found its shortest path to it already. Dijkstra's algorithm is greedy, so once you visit a vertex, you can't visit it again to undo any mistakes. This is where negative edge weights can be a problem. Here is a concrete counterexample.



Starting Dijkstra's as s, we have a and b in our PQ. We pop off a and fill in the shortest path to t as 2. b would be next and would set the shortest path to a

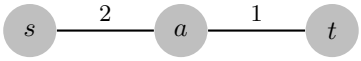
as -95, but by then it's too late to update the better shortest path to t cause a was already removed from the PQ.

(e) (T/F) Adding a constant positive integer k to all edge weights will not affect any MST of the graph.
True. One way to think about is through the cut property. Considering any cut of the graph, if you add k to all the edges crossing the cut, it won't affect which edge you decide to add to the MST: the edge of smallest weight even after adding k is still the edge of smallest weight crossing the cut. Even with multiple edges of the same weight crossing the cut, adding k does not effect the possible options.

(f) (T/F) Prim's algorithm works even when there are negative edge weights in the graph.
True. MST algorithms work fine with negative edge weights. This goes back to the cut property since for any cut, you can choose a negative edge weight as long as it's the smallest. Another way to think about it is to use the fact that adding a constant positive integer k does not affect the MST. If the smallest edge weight in the graph is $-p$, if you added p to all the edge weights, that would make all edge weights non-negative, and the MST still hasn't changed.

(g) Why are disjoint sets used in Kruskal's algorithm?
Disjoint sets are used to keep track of the connected components formed by Kruskal's algorithm. That way, we can use this for cycle detection. When we consider the next edge to add to our MST, if the two vertices for this edge are in the same connected component (analogous to having the same root if you were using union-find), then adding this edge into the MST would create a cycle.

(h) (T/F) The last edge added to the MST by Prim's algorithm is always the highest weight edge of the MST. False. Considering the following graph and starting Prim's algorithm at vertex s:

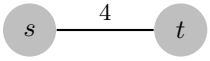


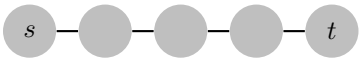
3 Shortest Path Algorithm Design

Design an efficient algorithm for the following problem: Given a weighted, undirected, and connected graph G where the weights of every edge in G are all integers between 1 and 10, and a starting vertex s in G, find the distance from s to every other vertex in the graph (where the distance between two vertices is defined as the weight of the shortest path connecting them).

Your algorithm must run asymptotically faster than Dijkstra's.
Hint: What other shortest path algorithms have we learned? Could we possibly modify the graph to apply other SP algorithms?

One possible solution is to convert every weighted edge of weight w into a chain of $w-1$ unweighted vertices. As an example of what we mean:





Once we do this, we run BFS starting from our source. BFS will give us shortest paths for an unweighted graph, so this is exactly why we make this weighted graph an unweighted graph. In the worse case, every edge has weight 10, so we add 9 extra vertices per edge and 9 extra edges per weighted edge. However, these are both constants so our runtime is still asymptotically $\Theta(|V| + |E|)$ by BFS.

Another possible solution is to modify the priority queue to return the minimum more quickly by creating something like 11 buckets, 1 for each edge weight, with each bucket containing a linked list of vertices currently in the queue that are connected to the edge weight, and then pop the minimum off by looking at the smallest nonzero bucket. This requires some more caution in keeping track of state but would also work.