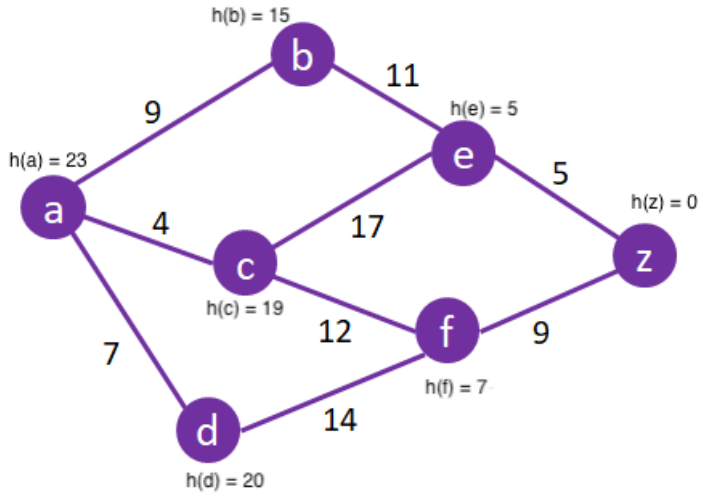
CS 61B Spring 2020

Small Group Tutoring
Section 11: Graphs, Shortest Path, MST, UF

Worksheet 14

1 Dijkstra's and A*



(a) Given the following graph, run Dijkstras algorithm starting at node a. At each step, write down the entire state of the algorithm. This includes the value dist(v) for all vertices v for that iteration as well as what node was popped off of the fringe for that iteration. List the final shortest distances to every vertex and state the runtime.

Solution: Start with a fringe, which is a min priority queue that orders the vertices by dist(a) values. At each iteration, pop a node from the fringe (the vertex with the lowest dist(a)). For each outgoing edge from this vertex, check to see whether the sum of dist(popped) and the edge's value is less than the current dist value of the vertex the edge connects to. If so, set the dist value of that vertex to this lower value. Note that popped vertices will never have their dist values changed because when we pop off a vertex, the distance to that vertex can only increase by considering other vertices and edges (since the popped vertex currently has the min dist value). Continue until all nodes have been popped from the fringe.

v	init	Pop a	Pop c	Pop d	Pop b	Pop f	Pop e	Pop z
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	∞	9	9	9	**9**	**9**	**9**	**9**
c	∞	4	**4**	**4**	**4**	**4**	**4**	**4**
d	∞	7	7	**7**	**7**	**7**	**7**	**7**
e	∞	∞	21	21	20	20	**20**	**20**
f	∞	∞	16	16	16	**16**	**16**	**16**
z	∞	∞	∞	∞	∞	25	25	**25**

Final distances are bolded
Runtime: We remove V nodes and update E edges (each node's neighbors) which takes O(VlogV + ElogV) = O((V + E)logV) time.

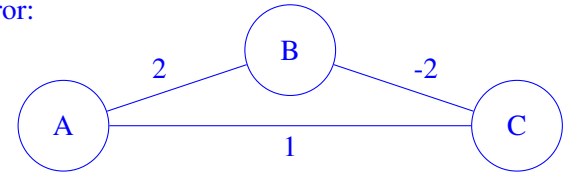
(b) Now run A* search on the graph above to find the shortest distance from node a to node z. If there is a tie, choose the node that was inserted first. Write down the order in which vertices are dequeued.
Solution: Use the dist(start, v) + h(v) rows to keep track of which nodes are currently in the priority queue. Start with a fringe, which is a min priority queue that orders the vertices by dist(start, v) + h(v) values. At each iteration, pop a node v from the fringe (the vertex with the lowest dist(start, v) + h(v)). For each of it's neighbors w that have not yet been visited, if w is not yet in the priority queue, add it. Otherwise, update it's priority and dist values if it's new priority (dist(start, w) + h(w)) is less than the current priority. Continue until we dequeue the goal vertex. (Note: R means the vertex has been removed/visited)

v	init	Pop a	Pop c	Pop f	Pop b	Pop z
(dist(a), h(a))	(0, 23)	**R**				
(dist(b), h(b))	(∞, -)	(9, 15)	(9, 15)	(9, 15)	**R**	
(dist(c), h(c))	(∞, -)	(4, 19)	**R**			
(dist(d), h(d))	(∞, -)	(7, 20)	(7, 20)	(7, 20)	(7, 20)	(7, 20)
(dist(e), h(e))	(∞, -)	(∞, -)	(21, 5)	(21, 5)	(20, 5)	(20, 5)
(dist(f), h(f))	(∞, -)	(∞, -)	(16, 7)	**R**		
(dist(z), h(z))	(∞, -)	(∞, -)	(∞, -)	(25, 0)	(25, 0)	**R**

We dequeue the goal node z and it has a distance of 25.

(c) What must be true about our graph in order to guarantee Dijkstras will return the shortest paths tree to every vertex?

Solution: In order to guarantee Dijkstras will return the shortest path to every vertex, we must have a graph that has no negative edge weights. Take the following graph as an example of why negative edge weights might cause an error:



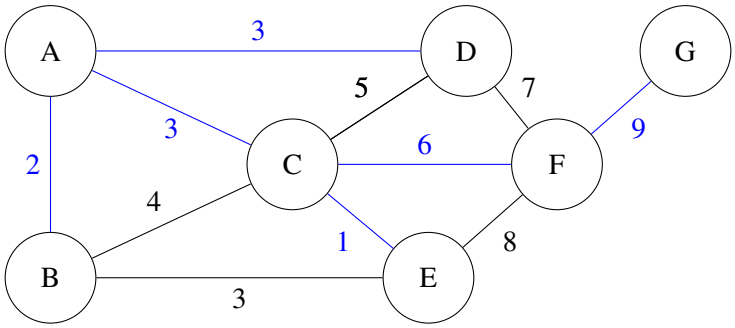
If we ran Dijkstras starting from A, then we would get the incorrect shortest path to C since we would choose the bottom path instead of the top path through B. We choose the bottom path because we reach and pop off C before popping off B and considering its edge to C.

In Dijkstra's, when we pop off a vertex, we do so with the assumption that we have found the shortest distance to that vertex. This assumption only holds with non-negative weights because adding non-negative edges to a path can never decrease the total weight of that path. In other words, the distance to that vertex can only increase by considering other vertices and their edges (since the popped vertex currently has the min dist value). While having negative edge weights does not guarantee Dijkstras will fail, we are guaranteed to get the shortest path when we have all non-negative edge weights.

(d) Recall that the Dijkstra's uses a min priority queue ordered by distance from the start node. How would we modify the priority queue for A* search?

Solution: Recall that A* search finds the shortest path from a start node to a specific goal node while Dijkstra's finds the shortest path to all vertices from a start. A* uses a heuristic h(v) which is a good estimate of the distance from v to the goal node in order to search in the "right" direction. While this is not the only change we have to make to Dijkstra's to run A*, we change the priority to dist(start, v) + h(v) as opposed to just dist(v). If h(v) = 0 for all vertices, then A* is the same as Dijkstra's.

2 Minimum Spanning Tree



(a) Given the graph above, run Kruskal's and Prim's algorithm to determine the minimum spanning tree of this graph. For Prim's algorithm, assume we start at node A and fill in the following chart including the value cost(v) for all vertices v for that iteration as well as which node was popped off of the fringe for that iteration. (Note: Ties are broken in alphabetical order)

Solution: To find the minimum spanning tree, both Kruskal's and Prim's algorithm apply the cut property, which states that **the shortest edge between two disjoint sets of nodes must be in the minimum spanning tree**.

To employ Kruskal's algorithm on this graph, we'll start with all the nodes disconnected. From there, we locate that smallest edge, which is the one between C and E. Since those two nodes are not already connected, we know that this edge must be in the minimum spanning tree. Using the same logic, we add the edge between A and B. In the next iteration, we have three edges with weight 3. We can see that adding the edge between A and D connects disjoint sets of nodes. Now, we add the edge between A and C. Afterwards, the next smallest edge is between B and E, but there is already a path from B to E, so that edge is not in the minimum spanning tree. Anyways, the final edges to the graph are illustrated using the same logic as before.

Meanwhile, Prim's algorithm starts at a node, and we'll use A for this example. Now, we identify the smallest edge out from A, which is the one to B, and add it to the minimum spanning tree because the two nodes are not already connected. We perform the same operation on our new set of nodes, and the smallest edge out is from A to C. After that, our set of connected nodes includes A, B, and C, and the smallest edge out from that is the one between C and E, and so on and so forth until we end up with the resulting image. We can accordingly fill in the chart below:

v	init	Pop a	Pop b	Pop c	Pop e	Pop d	Pop f	Pop g
cost(a)	**0**	0	0	0	0	0	0	0
cost(b)	∞	**2**	2	2	2	2	2	2
cost(c)	∞	3	**3**	3	3	3	3	3
cost(d)	∞	3	3	3	**3**	3	3	3
cost(e)	∞	∞	3	**1**	1	1	1	1
cost(f)	∞	∞	∞	6	6	**6**	6	6
cost(g)	∞	∞	∞	∞	∞	∞	**9**	9

(b) Does Kruskal's algorithm for finding the minimum spanning tree work on graphs with negative edge weights? Does Prim's?

Solution: Yes, both algorithms work with negative edge weights because the cut property still applies.

(c) True or False: A graph with unique edge weights has a unique minimum spanning tree.

Solution: True. This can be proved using the cut property. Unique edge weights implies that for every cut, there exactly one minimum-weighted edge.

3 Union Find Quick Checks

1. Consider the naive implementation of union find, where for each element, we store the set number that an element belongs in. All elements belonging to the same set has the same set number. For connect(), we change the set number of all elements belonging to the smaller set to be the set number of the element that belongs to the larger set. What is the worst case runtime for find and union individually?

   Solution:

   $$\theta(\text{find}) = \theta(1)$$
   $$\theta(\text{union}) = \theta(N)$$

   For find, because it is the case that each member in a given set has the same set number and we store the set number, a find operation is just a lookup of that element's set number. This is just a constant time operation.
   To see why union is a $\theta(N)$ operation, consider the case where you have two sets, with N and $N+1$ elements respectively. This would involve at least N set number change operations, 1 for every element of the smaller set. Ignoring constant factors gives us our $\theta(N)$ runtime.

2. Now consider an implementation of union find where each set is represented by a tree with a single root. When we call connect() on two sets, the root of the smaller set becomes a child of the root of the larger set. What is the worst case runtime for find and union individually?

   Solution:

   $$\theta(\text{find}) = \theta(\log(N))$$
   $$\theta(\text{union}) = \theta(\log(N))$$

   Using this implementation, notice that even in the worst case, where each union links trees of equal size, the height of any tree will still only grow to a height of $\log(N)$. Here the find operation takes $\theta(\log(N))$ time because it just involves swimming up a tree of worst case height $\log(N)$ to find the set number of a given element. Similarly, the union operation will involve **two** find operations, one for each set, and a reassignment of one of the parents to the other parent. This also yields a runtime of $\log(N)$

3. Now, let's consider the same union find implementation as above, but with path compression. When we call connect(), we reassign the parent of every element we pass to be the root of the set. What is the (approximate) amortized runtime for find and union individually?

   Solution: Very near constant amortized time for either operation.
   Understanding the runtime of this is out of scope, but here is a link to the proof for brownie points.
   http://www.uni-trier.de/fileadmin/fb4/prof/INF/DEA/Uebungen$_LVA-Ankuendigungen/ws07/KAuD/effi.pdf$

4 Optional: Fiat Lux

After graduating from Berkeley with solid understanding of CS61B topics, Josh became a billionaire and wants to build power stations across Berkeley campus to help students survive from PG&E power outages. Josh want to minimize his cost, but due to the numerous power outages when he took CS61B, he did not learn anything about Prim's or Kruskal's algorithm and he is asking for your help! We must meet the following constrains to power the whole campus:

- There are V locations where Josh can build power stations, and it costs v_i dollars to build a power station at the i^{th} position.
- There are E positions we can build wires and it cost e_{ij} to build a wire between location i and j.
- All locations must have a power station itself or be connected to another position with power station.
- $e_{ij} << v_i, \forall i, j$

Use the Prim's or Kruskal's algorithm taught in class to find a strategy that will minimize the cost while still fulfilling the constrains above.

Solution:

As the question suggests, this problem can be reduced to a graph problem where we want to have nodes either be marked themselves or connected to a marked one. To do so, we first create a graph with one vertex per location with edges between them corresponding to the cost of building a wire between them.

To also account for the cost to build the power stations (i.e. the value of each node), we will add on dummy node and connect it to every node in our graph. The weight of the edge from the dummy node to node n_i is v_i, which is the cost to build the power station at location i. Now we can simply run Kruskal's or Prim's algorithm to find the MST in this graph. For all edges in the MST we find, if the edge connects n_i to the dummy node, we will build the station at position i; if the edge connects two nodes n_i, n_j in the original graph, we will build a wire between those location i and j.

For example, if originally we have five locations, and we are given the value v_i and e_{ij}, we can first build the graph on the left hand side. Thereafter, we can add a dummy node as mentioned above and reconstruct the graph to obtain the graph on the right hand side. Then, we can run Kruskal's or Prim's algorithm on the new graph and obtain a MST (drawn in blue). In this case, the best strategy is to build power station at n_1, n_2, n_5 and build wire to connect n_1 with n_3, n_2 with n_4;