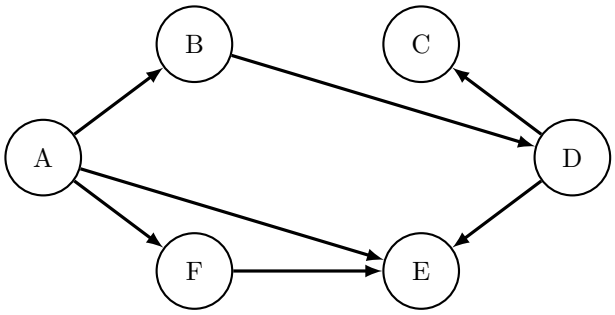
CS 61BL Summer 2021

Shortest Paths and MSTs

Recurring Section 11: Thursday August 5, 2021

1 Topological Sorting

Give a valid topological ordering of the graph below.



Is the topological ordering of the graph unique?

Solution:

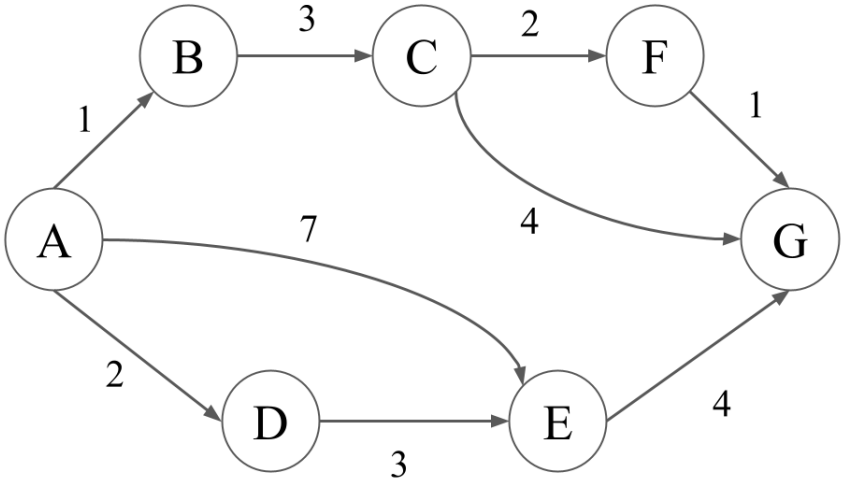
A topological ordering is a linear ordering of nodes such that for every directed edge S $\rightarrow$ T, S is listed before T. For this problem, the topological ordering of the graph is **not** unique. Below, we list two valid topological orderings for the graph.

- One valid ordering: A, B, D, C, F, E
  - Explanation: One way to approach this problem is to take any node with no edges leading to it and return it as the next node. After returning a node, we delete it and any edges leaving from it and look for a node with no incoming edges in the updated graph. We can repeat this until we have no nodes left. If at any point in this process we have a multiple choices for which node to return then the topological ordering is not unique.
- Another possible valid ordering: A, F, B, D, E, C
  - Explanation: Note that this ordering is just the reverse of DFS postorder traversal. Reverse DFS postorder will always be a valid topological ordering. This is because a DFS postorder traversal visits nodes only after all successors have been visited, so the reverse traversal visits nodes only after all predecessors have been visited.

Note: Only Directed Acyclic Graphs (DAGs), which are directed graphs that do not contain any cycles, have topological orderings. This is because within any given cycle, no one node comes before another. There are no valid topological orderings for undirected graphs because there is no direction associated with any edge. No one node comes before another, so it does not make sense to have a topological ordering for undirected graphs.

2 The Shortest Path To Your Heart

For the graph below, let `g(u, v)` be the weight of the edge between any nodes `u` and `v`. Let `h(u, v)` be the value returned by the heuristic for any nodes `u` and `v`.



Below, the pseudocode for Dijkstra's and A* are both shown for your reference throughout the problem.

Dijkstra's Pseudocode

```
PQ = new PriorityQueue()
PQ.add(A, 0)
PQ.add(v, infinity) # (all nodes except A).

distTo = {} # map
distTo[A] = 0
distTo[v] = infinity # (all nodes except A).

while (not PQ.isEmpty()):
  popNode, popPriority = PQ.pop()

  for child in popNode.children:
    if PQ.contains(child):
      potentialDist = distTo[popNode] +
        edgeWeight(popNode, child)
      if potentialDist < distTo[child]:
        distTo.put(child, potentialDist)
        PQ.changePriority(child, potentialDist)
```

A* Pseudocode

```
PQ = new PriorityQueue()
PQ.add(A, h(A))
PQ.add(v, infinity) # (all nodes except A).

distTo = {} # map
distTo[A] = 0
distTo[v] = infinity # (all nodes except A).

while (not PQ.isEmpty()):
  poppedNode, poppedPriority = PQ.pop()
  if (poppedNode == goal): terminate

    for child in poppedNode.children:
      if PQ.contains(child):
        potentialDist = distTo[poppedNode] +
          edgeWeight(poppedNode, child)

        if potentialDist < distTo[child]:
          distTo.put(child, potentialDist)
          PQ.changePriority(child, potentialDist + h(child))
```

(a) Run Dijkstra's algorithm to find the shortest paths from A to every other vertex. You may find it helpful to keep track of the priority queue and make a table of current distances.

	A	B	C	D	E	F	G
DistTo							
EdgeTo							

`B = 1 ; C = 4 ; D = 2 ; E = 5 ; F = 6 ; G = 7`

Explanation:

For the best explanation, it is recommended to check the slideshow linked on the website or watch the walkthrough video, as the text explanation is verbose.

We will maintain a priority queue and a table of distances found so far, as suggested in the problem and pseudocode. We will use {} to represent the PQ, and (()) to represent the distTo array.

`{A:0, B:inf, C:inf, D:inf, E:inf, F:inf, G:inf}. (()).`

`Pop A.`
`{B:inf, C:inf, D:inf, E:inf, F:inf, G:inf}. ((A: 0)).`

`changePriority(B, 1). changePriority(D, 2). changePriority(E, 7).`
`{B:1, D:2, C:inf, E:7, F:inf, G:inf}. ((A: 0)).`

`Pop B.`
`{D:2, C:inf, E:7, F:inf, G:inf}. ((A: 0, B: 1)).`

changePriority(C, 4).
{D:2, C:4, E:7, F:inf, G:inf}. ((A: 0, B: 1)).

Pop D.
{C:4, E:7, F:inf, G:inf}. ((A: 0, B: 1, D: 2)).

changePriority(E, 5).
{C:4, E:5, F:inf, G:inf}. ((A: 0, B: 1, D: 2)).

Pop C.
{E:5, F:inf, G:inf}. ((A: 0, B: 1, D: 2, C: 4)).

changePriority(F, 6). changePriority(G, 8).
{E:5, F:6, G:8}. ((A: 0, B: 1, D: 2, C: 4)).

Pop E.
{F:6, G:8}. ((A: 0, B: 1, D: 2, C: 4, E: 5)).

potentialDistToG = 9, which is worse than our current best known distance to G. No updates made.

Pop F.
{G:8}. ((A: 0, B: 1, D: 2, C: 4, E: 5, F: 6)).

potentialDistToG = 7. changePriority(G, 7).
{G:7}. ((A: 0, B: 1, D: 2, C: 4, E: 5, F: 6)).

Pop G.
{}. ((A: 0, B: 1, D: 2, C: 4, E: 5, F: 6, G: 7)).

(b) Given the weights and heuristic values for the graph above, what path would A* search return, starting from A and with G as a goal?

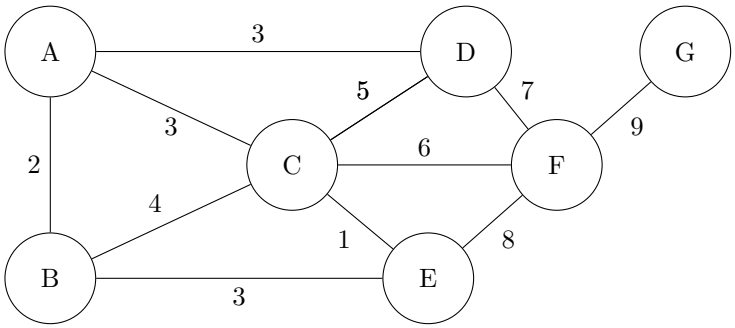
Edge weights	Heuristics
$g(A,B) = 1$	$h(A,G) = 7$
$g(B,C) = 3$	$h(B,G) = 6$
$g(C,F) = 2$	$h(C,G) = 3$
$g(C,G) = 4$	$h(F,G) = 1$
$g(F,G) = 1$	$h(D,G) = 6$
$g(A,D) = 2$	$h(E,G) = 3$
$g(D,E) = 3$	
$g(E,G) = 4$	
$g(A,E) = 7$	

	A	B	C	D	E	F	G
DistTo							
EdgeTo							

A* would return $A - B - C - F - G$. The cost here is 7.

Explanation: A* runs in a very similar fashion to Dijkstra's. We got the same answer for the shortest path to G, though we actually explored less unnecessary nodes in the process (we never popped D and E off the queue). The main difference is the priority in the priority queue. For A*, whenever computing the priority (for the purposes of the priority queue) of a particular node n, always add $h(n)$ to whatever you would use with Dijkstra's. Additionally, note that A* will be run to find the shortest path to a particular goal node (as our heuristic is calculated as our estimate to our specific goal node), whereas Dijkstra's may be run with a specific goal, or it may be run to find the shortest paths to ALL nodes. In the solutions above, we found the shortest paths to all nodes, but if we only needed to know the shortest path to E, for example, we could have stopped after visiting E.

3 Minimum Spanning Trees



(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)

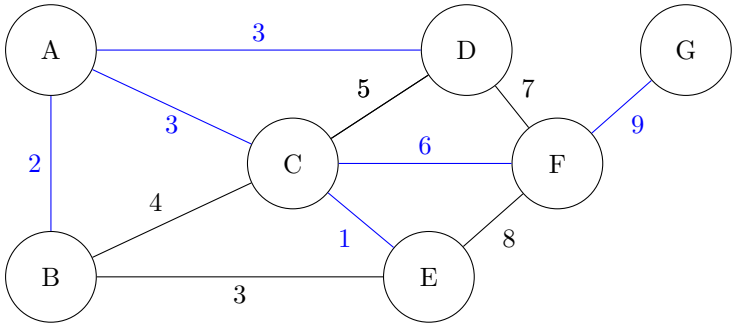
v	init	Pop __	Pop __	Pop __	Pop __	Pop __	Pop __	Pop __
cost(a)	**0**							
cost(b)	∞							
cost(c)	∞							
cost(d)	∞							
cost(e)	∞							
cost(f)	∞							
cost(g)	∞							

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**.

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) Run Kruskal's algorithm on the same graph.

Solution:
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.

(c) 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.

(d) 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.

4 Extra: 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.
- $e_{ij} << v_i, \forall i, j$

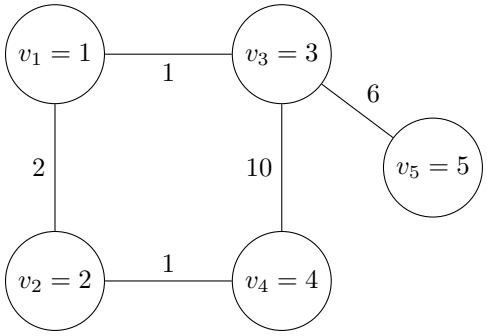
Use the Prim's or Kruskal's algorithm taught in class to find a strategy that will minimize the cost while still fulfilling the constrains above.

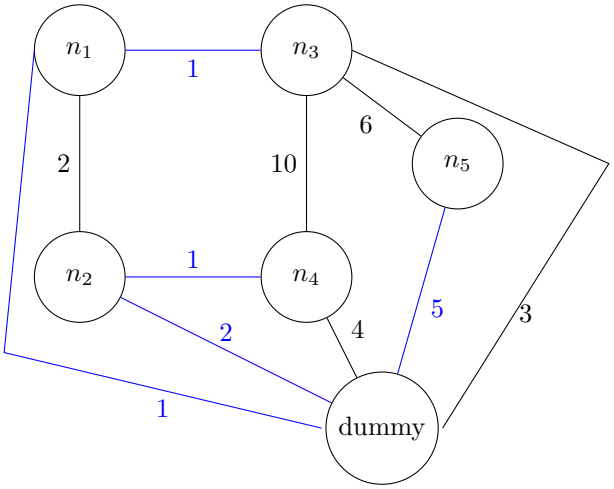
Solution:

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;


$v_1 = 1$
$v_3 = 3$
$v_5 = 5$
$v_2 = 2$
$v_4 = 4$
1
6
2
10
1


n_1
n_3
n_5
n_2
n_4
dummy
1
6
2
10
1
2
4
5
3
1