1 Dijkstra's Algorithm

(a) Given the following graph, run Dijkstra's algorithm starting at node A. For each iteration, write down the entire state of the algorithm. This includes the value `dist(v)` for all vertices v as well as what node was popped off of the fringe for that iteration.

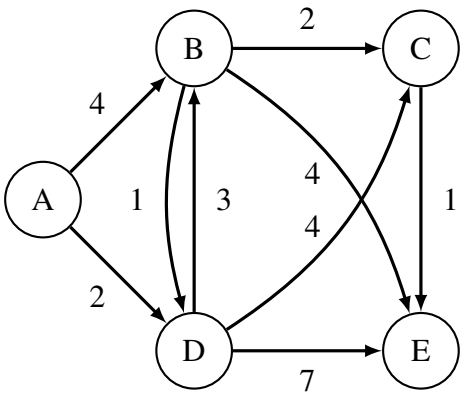
Note: If you want to keep track of the vertices traversed along the shortest paths from A to every other node in the graph, you will need to maintain an edgeTo array.

To run Dijkstra's algorithm, start with `dist(v)` for all vertices v set to ∞ and a fringe that includes all the vertices. The fringe is a minimum priority queue that orders the vertices by `dist(v)` values.

At each iteration, pop off a node from the fringe (this will be the vertex in the fringe with the lowest `dist(v)`). For each outgoing edge e from this popped vertex, check to see whether the sum of `dist(`*popped*`)` and the edge e'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 vertices that have already been popped from the fringe will never have their `dist` values changed. This is because when we pop off a vertex, the distance to that vertex can only increase by considering other vertices and their edges (since the popped vertex currently has the minimum `dist` value).

Continue until all nodes have been popped from the fringe.

	`dist(v)`					
v	Init	Pop A	Pop D	Pop B	Pop C	Pop E
A	**0**	0	0	0	0	0
B	∞	4	4	**4**	4	4
C	∞	∞	6	6	**6**	6
D	∞	2	**2**	2	2	2
E	∞	∞	9	8	7	**7**

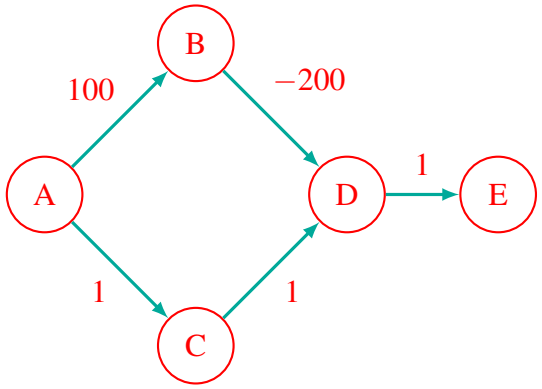


Note: For reference, here is the final edgeTo array that we get after running Dijkstra's algorithm to completion. `edgeTo(A)` is "-" because it was used as the starting vertex. `edgeTo(C)` is D because it is the first `edgeTo` vertex that was encountered along the shortest path to C. We would get D for `edgeTo(C)` if we update the edgeTo and dist arrays when a candidate path is strictly shorter than the existing path. We would get B for `edgeTo(C)` if we update the edgeTo and dist arrays when a candidate path is shorter than or the same distance as the existing path.

v	`edgeTo(`v`)`
A	-
B	A
C	D (or B)
D	A
E	C

(b) What must be true about our graph in order to guarantee Dijkstra's will return the shortest path's tree to every vertex? Draw an example of a graph that demonstrates why Dijkstra's might fail if we do not satisfy this condition.

In order to guarantee Dijkstra's 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:



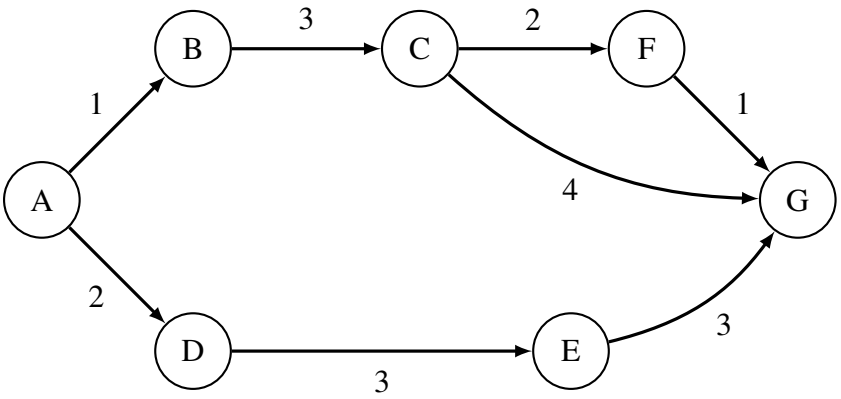
For this graph, if we ran Dijkstra's starting from A, then we would get the incorrect shortest path to E since we would choose the bottom path through C instead of the top path through B.

We choose the bottom path because we reach and pop off vertices C, D, and E before popping off vertex B and considering its edge to D. This is because in Dijkstra's, when we pop off a vertex, we do so with the assumption that 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). With negative edges, this assumption is no longer true.

Note that having negative edge weights does not guarantee Dijkstra's will fail, but if we have all non-negative edge weights then we are guaranteed to get the shortest path. When working with distances from the real world, we don't have to worry about negative edge weights because all distances in reality are strictly non-negative.

2 A* Search

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`. Remember the heuristic serves to estimate the distance between two nodes `u` and `v`.




```
Edge weights:    Heuristics:
g(A, B) = 1      h(A, G) = 8
g(B, C) = 3      h(B, G) = 6
g(C, F) = 2      h(C, G) = 5
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) = 3
```

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

A* would return the path A-D-E-G.

A* is different from Dijkstra's because it finds the shortest distance from a start node to a specific goal node g (rather than to all nodes). Instead of choosing the node with the smallest `dist(v)` value to pop off the fringe, in A* we choose the node with the smallest `dist(v) + h(v, g)` sum. Remember, `dist(v)` represents the distance from the start node to node v. The search is finished when we pop the goal node off of the fringe.

In the chart below, we keep track of dist values at each iteration. Note that we stop as soon as we pop G.

	`dist(v)`					
v	Init	Pop A	Pop B	Pop D	Pop E	Pop G
A	**0**	0	0	0	0	
B	∞	**1**	1	1	1	
C	∞	∞	4	4	4	
D	∞	2	**2**	2	2	
E	∞	∞	∞	**5**	5	
F	∞	∞	∞	∞	∞	
G	∞	∞	∞	∞	**8**	

In order to now derive the path from this table, we start with G and find the node that we traversed right before G. This will be the node that was popped when G obtained its final dist value (the value it has when it's popped). We notice this node is E. We then find the node we traversed right before E and so on. This gets us A-D-E-G.

(b) Is the heuristic admissible? Why or why not? A heuristic is admissible if it never overestimates the distance it is estimating.

The heuristic is not admissible because `h(C, G) = 5`, but the shortest path from C to G has length 3.

This is why A* returns A-D-E-G (total path distance of 8) when A-B-C-F-G (total path distance of 7) is actually the shortest path. Without an admissible heuristic, A* cannot guarantee that it will return the shortest path.