

1 Graph Representation

Represent the graph above with an adjacency list and an adjacency matrix representation.

NOTE: Edge lists and adjacency lists are not the same! An edge list is a list stored in each node that contains all successors and possibly predecessors of that node (see lecture). An adjacency list is more of a table that lists the adjacent vertices for each vertex in the graph; it is a representation of the graph as a whole. Graphs are commonly represented using adjacency lists and matrices but be aware of what is meant by an edge list.

Here, we stick with a convention that nodes don't have an edge to themselves; conventions vary.

Adjacency List				TO					
				A	B	C	D	E	F
A	→	[B, E, F]		0	1	0	0	1	1
B	→	[D]		0	0	0	1	0	0
C	→	[]		0	0	0	0	0	0
D	→	[C, E]	FROM	0	0	1	0	1	0
E	→	[]		0	0	0	0	0	0
F	→	[E]		0	0	0	0	1	0

(in the above matrix 0 means false and 1 means true)

2 Searches and Traversals

Run depth first search (DFS) and breadth first search (BFS) on the graph above, starting from node A. List the order in which each node is first visited. Whenever there is a choice of which node to visit next, visit nodes in alphabetical order.

DFS has both a preorder and postorder traversal. Preorder means we visit the node before visiting its children. Postorder order means we visit the node only after visiting its children.

DFS preorder: A, B, D, C, E, F

DFS postorder: C, E, D, B, F, A

BFS: A, B, E, F, D, C

3 Topological Sorting

Give a valid topological ordering of the graph. Is the topological ordering of the graph unique?

A topological ordering is a linear ordering of nodes such that for every directed edge $A \rightarrow B$, A comes before B .

Only DAGs (Directed Acyclic Graphs) have topological orderings. This is because within a cycle, no one node comes before another. With undirected graphs, no one node comes before another.

One valid ordering: A, B, D, C, F, E

The ordering is not unique. 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

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.

4 Dijkstra's Algorithm

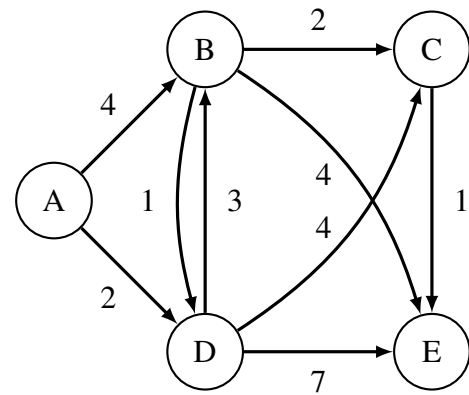
Given the following graph, run Dijkstra's algorithm starting at node A . For each iteration, write down the entire state of the algorithm. This includes the value $\text{dist}(v)$ for all vertices v as well as what node was popped off of the fringe for that iteration.

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

At each iteration, pop off a node from the fringe (this will be the vertex in the fringe with the lowest $\text{dist}(v)$). For each outgoing edge e from this popped vertex, check to see whether the sum of $\text{dist}(\text{popped})$ 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 min dist value).

Continue until all nodes have been popped from the fringe.

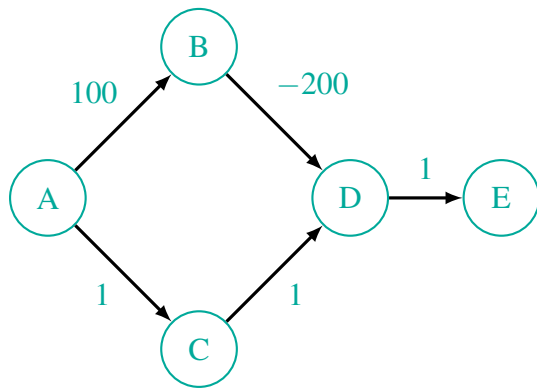
v	Init	Pop A	Pop D	Pop B	Pop C	Pop E
A	0	0	0	0	0	0
B	∞	4	4	4	4	4
C	∞	∞	6	6	6	6
D	∞	2	2	2	2	2
E	∞	∞	9	8	7	7



5 Dijkstra's Correctness

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. This is great if we want to use Dijkstra's on graphs that represent distances in real life since those are all positive!