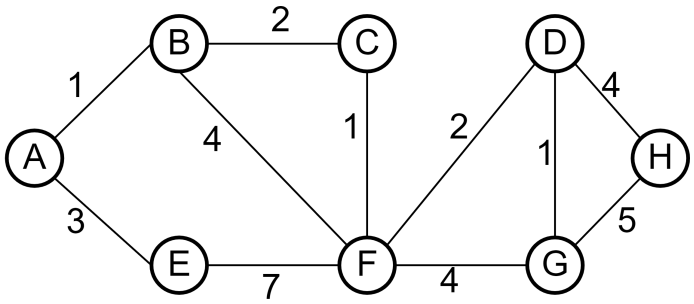
CS 61B Spring 2020 Exam Prep 14: A*, Shortest Path

1 DFS, BFS, Dijkstra's, A*

For the following questions, use the graph below and assume that we break ties by visiting lexicographically earlier nodes first.

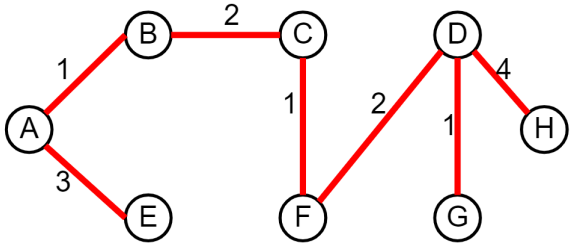


(a) Give the depth first search preorder traversal starting from vertex *A* (ignore edge weights).
A, B, C, F, D, G, H, E

(b) Give the depth first search postorder traversal starting from vertex *A* (ignore edge weights).
H, G, D, E, F, C, B, A

(c) Give the breadth first search traversal starting from vertex *A* (ignore edge weights).
A, B, E, C, F, D, G, H

(d) Give the order in which Dijkstra's Algorithm would visit each vertex, starting from vertex *A*. Sketch the resulting shortest paths tree.
A, B, C, E, F, D, G, H



(e) Give the path A* search would return, starting from A and with G as a goal. Let $h(u,v)$ be the valued returned by the heuristic for nodes u and v.

u	v	$h(u,v)$
A	G	9
B	G	7
C	G	4
D	G	1
E	G	10
F	G	3
H	G	5

$A \to B, B \to C, C \to F, F \to D, D \to G$

2 Cycle Detection

Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm. You may use either an adjacency list or an adjacency matrix to represent your graph.

We do a depth first search traversal through the graph. While we recurse, if we visit a node that we visited already, then we've found a cycle. Assuming integer labels, we can use something like a `visited` boolean array to keep track of the elements that we've seen, and while looking through a node's neighbors, if `visited` gives true, then that indicates a cycle. Note that this algorithm differs slightly if the graph is directed. If the graph is directed, then there is a cycle if a node has already been visited *and it is still in the recursive call stack (i.e. still in the fringe and not popped off yet).*

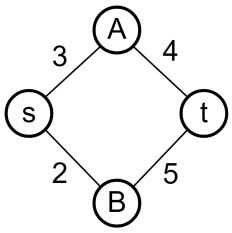
In the worst case, we have to explore V edges to find a cycle (number of edges doesn't matter). So, this runs in $\Theta(V)$.

3 Conceptual Shortest Paths

Answer the following questions regarding 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.

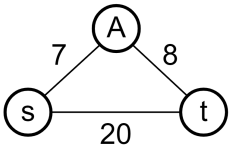
(a) (T/F) If all edge weights are equal and positive, breadth-first search starting from node A will return the shortest path from a node A to a target node B.
True. Breadth-first search finds shortest paths in an unweighted graph. Suppose all the weights are equal to w. If you divide all of the weights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



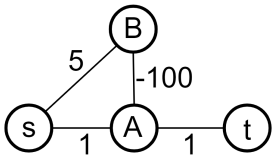
(c) (T/F) Adding a constant positive integer k to all edge weights will not affect any shortest path between vertices.

False. Adding a constant to all the edge weights when calculating shortest paths means we start favoring paths that use fewer edges. A concrete counterexample:



(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 the 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. A concrete counterexample:



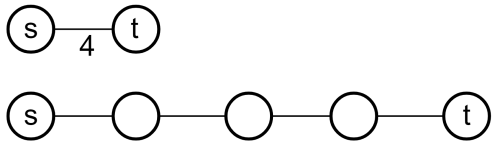
Starting Dijkstra's at s, we have A and B in our priority queue. We pop off A and fill in the shortest path to t as 2. B would be next and would set the shortest path to A as -95, but by then it's too late to update the better shortest path to T, as A was already removed from the priority queue.

4 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*: Think about the different graph traversals that you learned in lecture.

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



Once we do this, we run BFS starting from our source. BFS will give us the shortest path for an unweighted graph, so this procedure is exactly why we converted this weighted graph into an unweighted graph. In the worst case, every edge has weight 10, so we add 9 extra vertices 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 looing at the smallest nonzero bucket. This requires some more caution in keeping track of state but would also work.