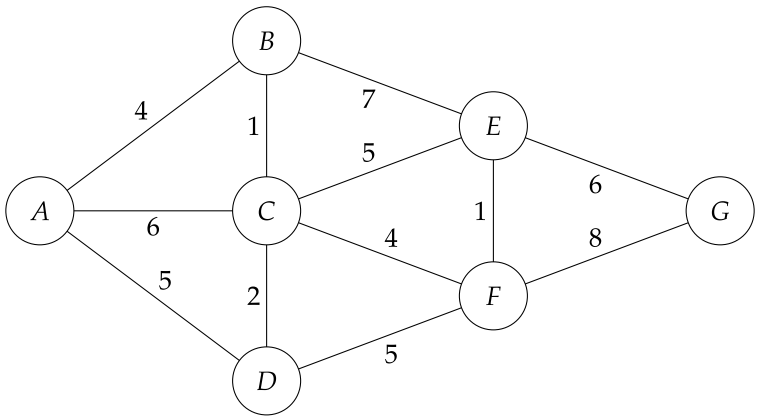
CSM 61B — Spring 2017

MSTs & Graph Algorithms

Week 12: April 11, 2017

1 Networking

1.1 Suppose we need to design a telephone network connecting all the residents, labeled *A* through *G*, in a neighborhood. How can we create a network that guarantees connectivity between all subscribers at the least possible cost?



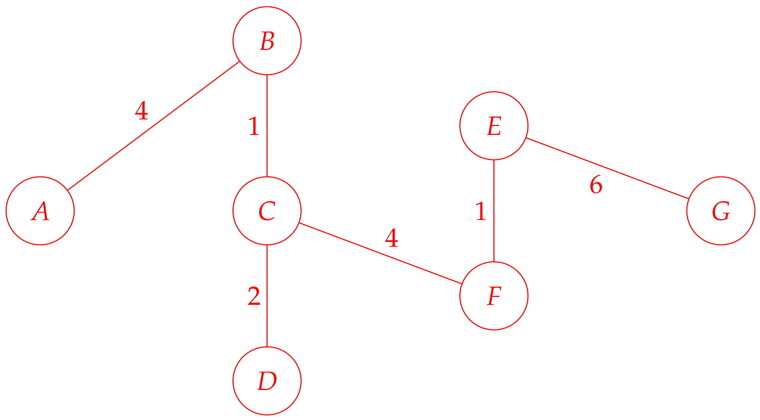
(a) In a graph with *N* vertices and *M* edges, how many edges form a minimum spanning tree?

$N-1$, or 6 edges in the above graph.

(b) Will the new graph contain any cycles? Describe the structure of the graph.

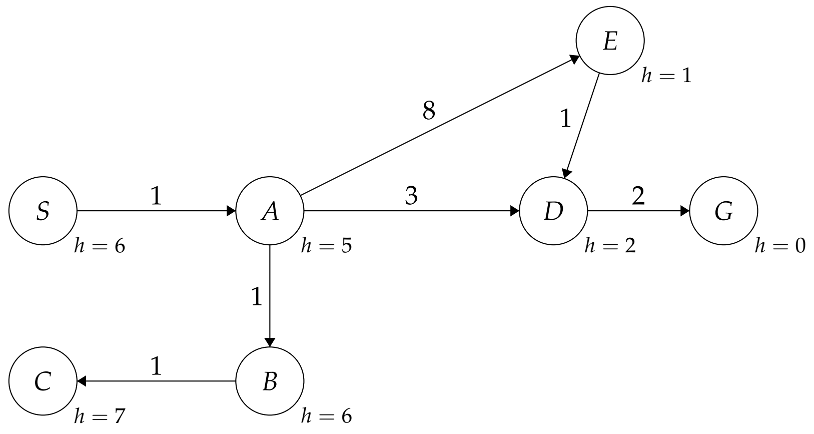
The resulting graph is a tree which implies that it contains no cycles. If the tree reaches every node in the graph, then it is a **spanning tree**. A graph may have many spanning trees, but we are particularly interested in **minimum spanning trees**, or spanning trees that minimize the total weight of the tree.

(c) Run Kruskal's Algorithm to find the minimum spanning tree.



2 A* Search

2.1 Find the path from the start, S, to the goal, G, when running each of the following algorithms.



Note that uniform cost search and greedy search are not a part of the course. The algorithms for all 3 of these searches are extremely similar, the only difference being the priority given to a key and whether or not we need the cumulative distance to a vertex.

(a) Which path does uniform cost search return?

$S - A - D - G$

From the starting node, choose the path that has the least cost, go to that node, and repeat until we reach the goal node. We choose the lowest *backward cost*.

Note that this is very similar to Dijkstra's, just a little more general. We keep a priority-queue fringe that keeps track of paths. At each step, we remove the shortest path from the fringe and add its children to the fringe, trying all paths in increasing cost order until we reach G.

(b) Which path does greedy search return?

$S - A - E - D - G$ or $S - A - D - G$ depending on tie-breaking behavior.

From the starting node, travel to the next node with the lowest value returned by the heuristic function until we reach the goal node. We choose the path with the lowest *forward cost*.

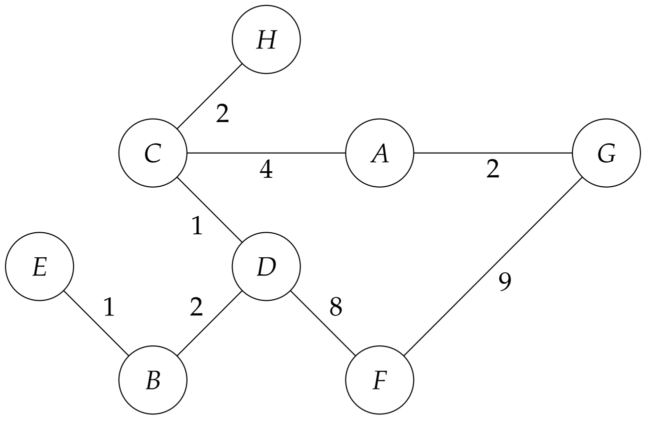
(c) Which path does A* search return?

$S - A - D - G$

At each node, we choose the next node that has the lowest sum of the path cost and $h(\cdot)$ value. This is essentially uniform cost search and greedy search combined.

3 Searches

3.1 For the graph below, write the order in which vertices are visited using the specified algorithm starting from A. Break ties by alphabetical order.



(a) DFS

$A - C - D - B - E - F - G - H$

(b) BFS

$A - C - G - D - H - F - B - E$

(c) Dijkstra's

$A - G - C - D - H - B - E - F$

4 Shortest Paths Algorithms *Extra for Experts*

4.1 Given a weighted, directed 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, or infinity if no such path exists.

(a) Design an algorithm for solving the problem that runs faster than Dijkstra's.

For every edge e in the graph, replace e with a chain of $w - 1$ vertices (where w is the weight of e) where the two ends of the chain are the endpoints of e.

Then run BFS on the modified graph, keeping track of the distance from v to each vertex from the original graph.

Alternatively, we can modify Dijkstra's algorithm. Since the runtime of Dijkstra's is bounded by the priority queue implementation, if we can come up with a faster priority queue, we can improve the runtime. Define our priority queue as an array of 11 linked list buckets. Keep track of a counter that represents our current position in the array. Each bucket corresponds to vertices of some distance from the start, s.

`removeMin()`: If the bucket with index counter is non-empty, remove and return the first vertex in its linked list. Otherwise, increment counter until we find a non-empty bucket. If the counter reaches 11, reset it to 0 and continue (so in effect our array is circular).

`insert()`: Given a vertex v and a distance d, insert the vertex into the beginning of the linked list at index $d \mod 11$.

This strategy works because the vertices we add to the priority queue at any time have a distance that is no more than 10 greater than the current distance.

(b) Give the runtime of your algorithm.

$\Theta(|V| + |E|)$ for running BFS on the modified graph, and $O(|V| + |E|)$ for modifying Dijkstra's priority queue.