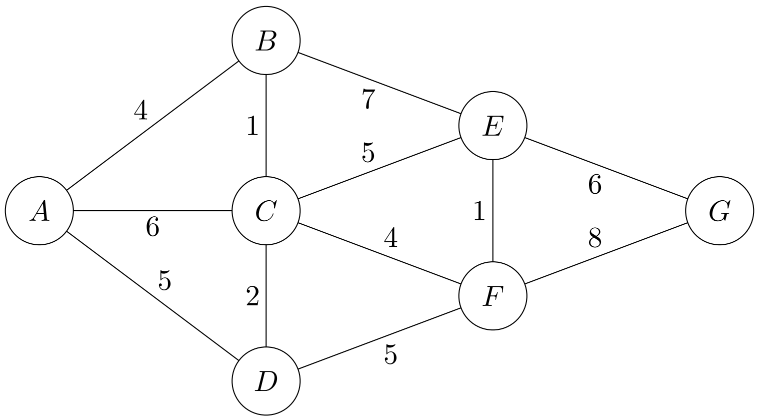
# Graph Algorithms



## 1 Networking

1.1 Suppose we want to design a telephone network connecting all the cities, labeled $A$ to $G$, in a neighborhood. We'd like to do so at the least 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.

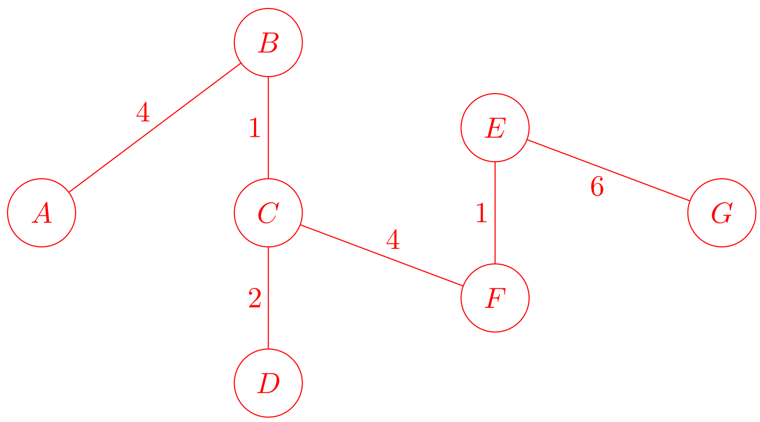
(b) Will the new graph contain any cycles? Describe its structure.

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) One algorithm to find a minimum spanning tree is Kruskal's algorithm.

1. Sort all the edges by increasing order of their weight.
2. Pick the smallest edge and check if it forms a cycle with the spanning tree so far. If it doesn't form a cycle, add this edge to the spanning tree.
3. Repeat the previous step until there are $|V| - 1$ edges in the spanning tree, where $|V|$ is the number of vertices in the graph.

Run Kruskal's Algorithm to find a minimum spanning tree.

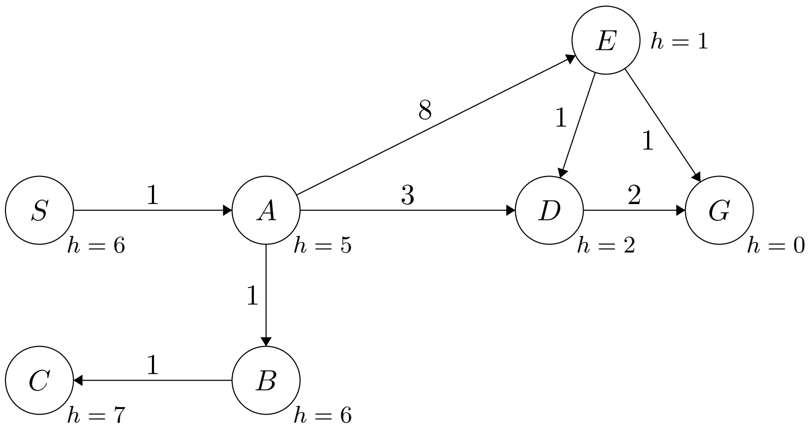



**Meta**: Draw only the nodes and add the edges that are part of the MST as you walk through the problem. Labeling all the edges would take a long time, and erasing is confusing and complicated.

# 2 A* Search

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

The **heuristic**, $h$, estimates the distance from each node to the goal.




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 each node and whether or not we need to keep track of the cumulative distance to a vertex.

(a) Which path does uniform cost search return?

**Uniform cost search** is the same as Dijkstra's, except that the search stops once we visit the goal state.

$S - A - D - G$

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*.

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?

In **greedy search**, we ignore edge weights entirely and only use the heuristic to decide which node to visit next.

$S - A - E - G$

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*.

Note that, unlike UCS or Dijkstra's, we don't add heuristics together because they already estimate the distance to the goal.

(c) Which path does A* search return?

**A* search** is an algorithm that combines the total distance from the start with the heuristic to optimize the search procedure.

$S - A - D - G$

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.

For A* to work, heuristics must be *admissible* and *consistent*.

- Admissible heuristics underestimate the true distance to the goal.
- Consistent heuristics require that the difference in heuristic values between two nodes cannot be greater than the true distance between the two.

# 3 Algorithms *Extra Practice*

**Traversal** Visit all the nodes in the graph.

- Depth-first traversal (preorder and postorder)
- Level-order traversal

**Search** Given $s$, find a goal $v$.

- Depth-first search
- Iterative-deepening depth-first search
- Breadth-first search

**Single Pair Shortest Path** Given $s$, find the shortest path to a goal $v$.

- *Uniform cost search*
- *Greedy search*
- A* search

**Single Source Shortest Path** Given $s$, find the shortest path to all nodes.

- Dijkstra's algorithm

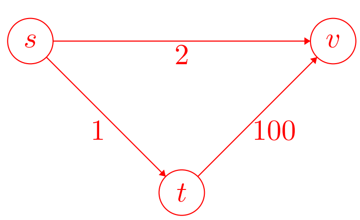
**Minimum Spanning Tree** A *spanning tree*, or acyclic subgraph connecting all the nodes with the least total edge weight.

- Prim's algorithm
- Kruskal's algorithm

3.1 Is this algorithm for computing the *single pair shortest path* correct?

Given a starting vertex, $s$, and an ending vertex, $v$, compute the shortest path between $s$ and $v$ by running DFS, but at each node exploring the shortest outgoing edge first until $v$ is reached. Return the $s$ to $v$ path in the DFS tree.

Incorrect. Consider the graph below whose true shortest path is $s - v$ but whose greedy DFS path is $s - t - v$.



3.2 Briefly describe an efficient algorithm and the runtime for finding a minimum spanning tree in an undirected, connected graph $G = (V, E)$ when the edge weights satisfy:

(a) For all $e \in E$, $w_e = 1$. (All edge weights are 1.)

The key idea here is that any tree which connects all nodes is an MST. We can run DFS and take the DFS tree. You could also take a BFS tree, or run Prim's algorithm with a queue or stack instead of a priority queue (this would be equivalent to BFS/DFS). The runtime of this algorithm is in $\Theta(|V| + |E|) \in \Theta(|E|)$ for simple graphs.

(b) For all $e \in E$, $w_e \in \{1, 2\}$. (All edge weights are either 1 or 2.)

Run Prim's algorithm with a specialized priority queue, comprised of 2 regular queues. When we add items to the priority queue, we add it to the first queue if the weight is 1 and otherwise add it to the second queue (the weight must be 2). When we pop from the priority queue, we take from the first queue, unless it is empty, in which case we take from the second. The runtime of this algorithm is in $\Theta(|E|)$ as priority queue operations are now constant time.

3.3 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 better 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.