1 Shortest Paths

There are 3 types of shortest paths problems:

- Single source single destination
- Single source to all destinations
- All pairs shortest path

In today's section, we will be focusing on the second type of shortest paths problem and review 3 algorithms for solving this problem when edge weights are all 1, all positive, and the general case.

1.1 Breadth First Search (BFS)

BFS($G(V, E), s \in V$)

```
for v in V − {s} do
    DIST[v] ← ∞
    PREV[v] ← NIL
    PLACED[v] ← False
end for
DIST[s] ← 0
PREV[s] ← nil
PLACED[s] ← 1
QUEUE q
ENQUEUE(q, s)
while q ≠ ∅ do
    u = DEQUEUE(q)
    for (u, v) ∈ E do
        if not PLACED[v] then
            DIST[v] ← DIST[u] + 1
            PREV[v] ← u
            PLACED[v] ← True
            ENQUEUE(q, v)
        end if
    end for
end while
```

- We use a queue (first-in first-out) to keep track of which vertices have already been visited. Unlike Depth-first search, we visit all of our closest neighbors before moving on to further ones.
- Can be used to find the shortest path when all the edge weights are equal to 1. This is not the only application of BFS, as we will see in a later exercise.

- **Run-time:** $\boxed{O(|V| + |E|)}$
  The initialization process takes $O(|V|)$ time, line 11 is run $|V|$ times, and the for loop at line 12 is run a total of $|E|$ times overall.

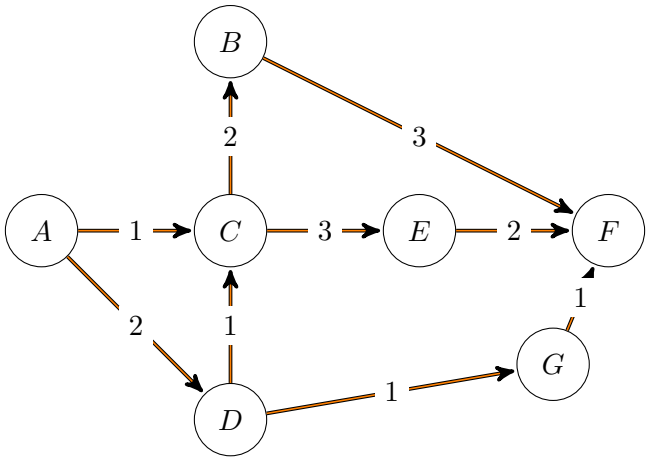
1.2 Dijkstra's Algorithm

Dijkstra($G(V, E, \omega), s \in V$)

```
1:  for v in V do
2:      DIST[v] ← ∞
3:      PREV[v] ← NIL
4:  end for
5:  DIST[s] = 0
6:  H ← {(s, 0)}
7:  while H ≠ ∅ do
8:      v ← DeleteMin(H)
9:      for (v, w) ∈ E do
10:         if DIST[w] > DIST[v] + ω(v, w) then
11:             DIST[w] ← DIST[v] + ω(v, w)
12:             PREV[w] ← v
13:             Insert((w, DIST[w]), H)
14:         end if
15:     end for
16: end while
```

- Dijkstra's algorithm us to find shortest paths in graphs with any positive edge weights. It uses a heap to determine which node to consider next. When a node is popped off the heap, that means we have our *final answer* for that node's distance and path from the source.
- Note that Dijkstra's algorithm does not work for negative edge weights! When a node is popped off the heap, we assume that the distance for that node is completely set. Since all other nodes in the heap have a larger distance than the one just popped off, we know that any path through those nodes cannot be used to reach the node just popped off.
- **Run-time:** $\boxed{O(|V| \cdot \text{deleteMin} + |E| \cdot \text{insert})}$.
  The running time of Dijkstra's algorithm depends on the implementation of the heap H. For each vertex, we perform a delete min, while for each edge we perform an insertion.

Exercise 1. Run dijkstra on this graph from A to F.



Solution

Looking at	Heap
A	C(1) D(2)
C	D(2) B(3) E(4)
D	B(3) G(3) E(4)
B	G(3) E(4) F(6)
G	E(4) F(4)
F	

1.3 General Shortest Path (Bellman Ford)

General-SP($G(V, E, \omega), s \in V$)

```
1: for v in V − {s} do
2:     DIST[v] ← ∞
3:     PREV[v] ← NIL
4: end for
5: DIST[s] = 0
6: for i = 1, 2, ... |V| − 1 do
7:     for (v, w) ∈ E do
8:         if DIST[w] > DIST[v] + ω(v, w) then
9:             DIST[w] ← DIST[v] + ω(v, w)
10:            PREV[w] ← v
11:        end if
12:    end for
13: end for
```

- At each iteration of the **for** loop on line 5, we have found the shortest path from s to each of the vertices using at most i edges.
- This algorithm is correct because the shortest path to $s \to u$ using at most i edges is the shortest path from $s \to v$ using at most $i-1$ edges plus $\omega(v, u)$ for some v that has an edge $(v, u) \in E$. This is true even if negative edge weights are present.
- **Run-time:** $\boxed{O(|V||E|)}$ as you can tell by the double for loop.

1.4 Shortest Path in DAG

- We are guaranteed to not have negative cycles because the graph is acyclic.
- Can be solved by doing a topological sort and only looking at the nodes between s and t in the topological order and doing something similar to Problem Set 2 Problem 4, running in $\boxed{O(|V| + |E|)}$ time.

2 Shortest Path Exercises

Exercise 2. (2014 Problem Set 2) There are n students standing in a playground trying to split into two teams to play kickball. No one cares that the teams have equal, or even nearly equal sizes, but the students *do* care that they are not on the same team as any of their mortal enemies. You are given a set of m enemy links, which are mutual. If person A has an enemy link with B, they absolutely cannot be on the same team. Give an algorithm to partition the students into two teams such that no enemies are on the same team. If this is not possible, return "Impossible".

Solution
To assign each student to a team, we will use a modified BFS. In the graph, each student is represented by a vertex, and each enemy link is represented by an edge between the two vertices representing the students. First, pick a student and WLOG assign him to Team 1. Then, conduct a modified BFS. At each vertex, examine all of its neighbors. If they are assigned to the same team as the vertex, return Impossible. Otherwise, assign all of the neighbors to the opposite team. Once the BFS completes, if there are still unassigned vertices, pick another vertex and randomly assign him a team. Repeat until all vertices are assigned a team, or we return Impossible.

Correctness: If our algorithm returns a team assignment, it must be valid since our algorithm guarantees that no vertex and its neighbor will be assigned to the same team. Otherwise, when assigning teams, we would have encountered a vertex which was the same team as its neighbor and returned impossible. If our algorithm returns Impossible, there is really no solution, since two neighboring vertices must be on the same team to satisfy the other enemy links. Therefore, our algorithm is correct.

Running Time Analysis: The running time of this algorithm is $O(|E| + |V|)$ since it is just modified BFS.

Exercise 3. (2015 Problem Set 2) You are a merchant, and you want to walk from your home to your shop. You figure that you might as well get a good workout while doing so. Your city contains various roads and intersections, with various lengths of road between each intersection. Each intersection is also elevated at some height above sea level. Your primary goal is to get to your shop using the shortest route possible. However, you want a path in which all road segments you take are initially all inclined upward (the exercise) followed by road segments which are all inclined downward (to cool down). The number of road segments you take inclined upward before you switch to going downward is up to you (you could also choose a route that is entirely upward or downward). Give efficient algorithms to find the shortest such paths in the following two cases:

(a) All elevations are distinct.

(b) Some elevations may be the same. You may walk on flat roads during both the uphill and downhill portions of your walk.

Solution

(a) For each intersection x, we find the minimum distance from your home to x via only uphill roads, and the minimum distance from x to your shop via only downhill roads. When we only consider uphill roads, we have a directed acyclic graph, so we can find all shortest paths in $O(n+m)$. Similarly, we find all shortest downhill paths to the shop in $O(n+m)$. Then we iterate over each vertex and add the two values, keeping track of the minimum, which takes $O(n)$ time. Thus, the overall time is $O(n+m)$.

(b) We can use a similar approach, except we now use DijkstraâĂŹs to compute the shortest uphill and downhill distances, since we no longer have a DAG. This now runs in $O(m+nlogn)$ using Fibonacci Heaps (remember that you don't need to know what those are, just that they exist and give you this good running time for Dijkstra's.

Alternatively, one could build a directed graph G_0 on $2n$ vertices in the following way. For each vertex x we make two vertices x^0, x^1 where being at x^0 means we are currently still trying to go uphill, and being at x^1 means we've switched to trying to go downhill. Thus for every edge (x, y) in the original graph, if (x, y) goes uphill then we make the edge (x^0, y^0). If it is going downhill then we make the edge we make the edge (x^1, y^1). If they are the same altitude, then we create both edges. We also make the edge (x^0, x^1) of length 0 for every x (so that we can always transition to going downhill from now on, at zero cost). If h is the home location and s is the shop, then we want the shortest path from h^0 to s^1. . If the elevations are distinct, G_0 is a DAG and we can do this in $O(n+m)$ time using shortest path in a DAG algorithm. If elevations are not distinct, Dijkstra's algorithm can be used to yield time $O(m + n \log n)$ (if using Fibonacci heaps), or $O(m \log n)$ time using binary heaps.

3 Minimum Spanning Trees

3.1 Introduction

A tree is an undirected graph $T = (V, E)$ satisfying all of the following conditions:

(a) T is connected,

(b) T is acyclic,

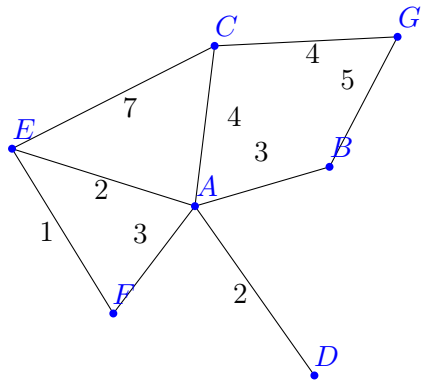
(c) $|E| = |V| - 1$.

Any two conditions above imply the third.

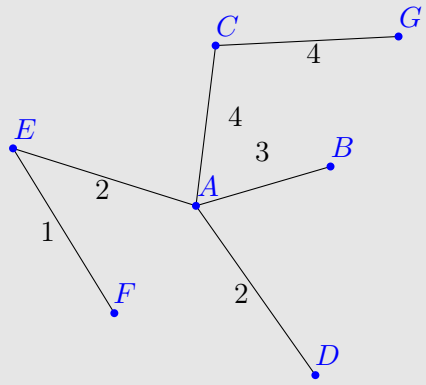
A **spanning tree** of an undirected graph $G = (V, E)$ is a subgraph which is a tree and which connects all the vertices. (If G is not connected, G has no spanning trees.)

A **minimum spanning tree** is a spanning tree whose total sum of edge costs is minimum.

Exercise 4. Compute a minimum spanning tree of the following graph:



Solution



Cut property. Let X be a subgraph of $G = (V, E)$ such that X is contained in some MST of G. Let $S \subset V$ be a cut such that no edge of X crosses the cut. Suppose e has minimum weight among the edges crossing the cut: then $X \cup \{e\}$ is contained in some MST of G. This means that it's *safe* to add e to our partial MST and still be on-track to finding an MST.

Exercise 5. Use the cut property to show that given any vertex, the edge extending out of that vertex with the lightest weight is contained in some MST.

Solution
Let u be the vertex we are considering, and let $v_1, v_2 \dots v_k$ be the neighbors of u. Let $X = \emptyset$ and $S = \{u\}$ in the cut property above. Clearly, X is a subgraph of some MST because the empty graph is a subgraph of all graphs. Our cut is between the sets S and $V - S$, and clearly no edges in our empty X crosses that cut. The only edges that cross this cut are $(u, v_1), (u, v_2), \dots, (u, v_k)$ and thus by the cut property, the lightest one of them must be contained in some MST.

3.2 Algorithms

Because of the cut property, we will use **greedy algorithms** to construct MST's: Start with $X = \emptyset$, and inductively do the following: Choose a cut $S \subset V$ with no edge of X crossing the cut, find the lightest edge e crossing the cut, set $X := X \cup \{e\}$. Repeat until X spans the graph. The exercise above shows one possible first step we can take when choosing our cut S.

Two efficient algorithms.

- **Prim's algorithm.** X is a tree, and we repeatedly set S to be the edges between the vertices of X and the vertices not in X, adding edges until X spans the graph.

  Thus we are always adding the "closest" vertex to the tree. The implementation of this is almost identical to that of Dijkstra's algorithm.

- **Kruskal's algorithm.** Sort the edges. Repeatedly add the lightest edge that doesn't create a cycle, until a spanning tree is found. We use the *Union-Find* data structure to make this very efficient (next section).

Notice that in Prim's we explicitly set S based on X, whereas here we implicitly "choose" the cut S after we find the lightest edge that doesn't create a cycle. In other words, we don't actually find a cut corresponding to the edge e to be added, but we know that one must exist | if not, adding e would create a cycle.

MST Exercises

Exercise 6 (CLRS 21.3-6)**.** Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge (one of minimum weight) crossing the cut. Conclude that this implies that if a graph has unique edge weights, then it has a unique minimum spanning tree.

Solution
Suppose we had two MST's T and T' of a graph $G = (V, E)$ that satisfied the above property. Consider any edge $t = (u, v) \in T$. Removing this edge from T separates the tree into 2 disjoint sets of vertices, S which contains u and $V - S$ which contains v. S and $V = S$ represent a particular cut in the graph, so we know that any MST of G must contain the lightest edge crossing this cut. But from our assumption, we know that t is indeed the lightest edge crossing this cut, so t is in all minimum spanning trees of G. In particular, $t \in T'$. Since t was arbitrarily chosen, we know that each edge in T is also an edge in T'. Since $|T| = |T'|$, we know that $T = T'$ and the MST for G is unique.
If a graph has unique edge weights, then any cut will have a unique light edge, and thus we have shown that the graph has a unique minimum spanning tree.

Exercise 7. (Fun Problem) In a city there are N houses, each of which is in need of a water supply. It costs W_i dollars to build a well at house i, and it costs C_{ij} to build a pipe in between houses i and j. A house can receive water if either there is a well built there or there is some path of pipes to a house with a well. Give an algorithm to find the minimum amount of money needed to supply every house with water.

Solution
Create a graph with N nodes representing the houses and edges between every pair of nodes representing the potential pipes. Then add an additional "source" node which connects to house i with cost W_i, so that the graph now has $N + 1$ nodes. Find the MST of this new graph.

Notes on the Programming Assignment

- Whereas the code is important, the write-up is equally important. You should have clear and concise explanations of what everything you did and why you chose to do it that way. You should think of the programming assignment as more of a *lab* than a *competition*.
- Remember that this assignment is experimental in nature. Your goal is to provide an insightful answer to the question: "How does the total weight of MST's behave in different types of random graphs?" using your program and results.
- Test your code on small examples (small enough that you can work them out by hand) to check for bugs.
- Important decisions to be made before writing the program:

  - **Language**: remember that you will be testing your program on complete graphs with 131072 vertices and over 17 billion edges! Definitely do look at the *hints* section in the assignment specifications.
  - **Graph Representation**: adjacency matrix versus adjacency list.
  - **Algorithm**: Prim's versus Kruskal's. Students in the past have been successful implementing either algorithm.
- Be careful when (and how often) you seed the random number generator.