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.

Breadth First Search (BFS)

\[
\text{BFS}(G(V, E), s \in V)
\]

1: \text{for } v \text{ in } V - \{s\} \text{ do}
2: \quad \text{DIST}[v] \leftarrow \infty
3: \quad \text{PREV}[v] \leftarrow \text{NIL}
4: \quad \text{PLACED}[v] \leftarrow \text{False}
5: \quad \text{DIST}[s] \leftarrow 0
6: \quad \text{PREV}[s] \leftarrow \text{nil}
7: \quad \text{PLACED}[s] \leftarrow 1
8: \quad \text{QUEUE } q
9: \quad \text{ENQUEUE}(q, s)
10: \quad \text{while } q \neq \emptyset \text{ do}
11: \quad \quad u = \text{DEQUEUE}(q)
12: \quad \quad \text{for } (u, v) \in E \text{ do}
13: \quad \quad \quad \text{if not PLACED[v] then}
14: \quad \quad \quad \quad \text{DIST}[v] \leftarrow \text{DIST}[u] + 1
15: \quad \quad \quad \quad \text{PREV}[v] \leftarrow u
16: \quad \quad \quad \quad \text{PLACED}[v] \leftarrow \text{True}
17: \quad \quad \quad \text{ENQUEUE}(q, v)

- 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:** \[ O(|V| + |E|) \]
  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.
Dijkstra’s Algorithm

\[
\text{Dijkstra}(G(V, E, \omega), s \in V)
\]

1: \textbf{for} \ v \ \textbf{in} \ V \ \textbf{do}
2: \hspace{1em} \text{DIST}[v] \leftarrow \infty
3: \hspace{1em} \text{PREV}[v] \leftarrow \text{NIL}
4: \hspace{1em} \text{DIST}[s] = 0
5: \hspace{1em} H \leftarrow \{(s, 0)\}
6: \textbf{while} \ H \neq \emptyset \ \textbf{do}
7: \hspace{1em} v \leftarrow \text{DeleteMin}(H)
8: \hspace{1em} \textbf{for} \ (v, w) \ \in \ E \ \textbf{do}
9: \hspace{2em} \text{if} \ \text{DIST}[w] > \text{DIST}[v] + \omega(v, w) \ \textbf{then}
10: \hspace{3em} \text{DIST}[w] \leftarrow \text{DIST}[v] + \omega(v, w)
11: \hspace{3em} \text{PREV}[w] \leftarrow v
12: \hspace{2em} \text{INSERT}((w, \text{DIST}[w]), H)

- Dijkstra’s algorithm allows 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: \(O(|V| \cdot \text{deleteMin} + |E| \cdot \text{insert})\)

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.

General Shortest Path

\[
\text{General-SP}(G(V, E, \omega), s \in V)
\]

1: \textbf{for} \ v \ \textbf{in} \ V - \{s\} \ \textbf{do}
2: \hspace{1em} \text{DIST}[v] \leftarrow \infty
3: \hspace{1em} \text{PREV}[v] \leftarrow \text{NIL}
4: \hspace{1em} \text{DIST}[s] = 0
5: \hspace{1em} \textbf{for} \ i = 1, 2, \ldots |V| - 1 \ \textbf{do}
6: \hspace{1em} \textbf{for} \ (v, w) \ \in \ E \ \textbf{do}
7: \hspace{2em} \textbf{if} \ \text{DIST}[w] > \text{DIST}[v] + \omega(v, w) \ \textbf{then}
8: \hspace{3em} \text{DIST}[w] \leftarrow \text{DIST}[v] + \omega(v, w)
9: \hspace{3em} \text{PREV}[w] \leftarrow v

- At each iteration of the \textbf{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 \(s \rightarrow u\) using at most \(i\) edges is the shortest path from \(s \rightarrow v\) 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: \(O(|V||E|)\) as you can tell by the double for loop.
Shortest Path Exercises

Exercise 1. (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 2 (CLRS 24.3-4). We are given a directed graph \( G = (V, E) \) on which each edge \((u, v) \in E\) has an associated value \( r(u, v) \), which is a real number in the range \( 0 \leq r(u, v) \leq 1 \) that represents the reliability of a communication channel from vertex \( u \) to vertex \( v \). We interpret \( r(u, v) \) as the probability that the channel from \( u \) to \( v \) will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

Solution
The reliability of any path is the product of the probabilities of not failure over each segment of the path. Therefore, this problem is very similar to the shortest path problem except we are trying to maximize the product instead of minimizing the sum. There are two ways we can approach this problem:

1. Modify the probabilities
For each edge \((u, v) \in E\), we replace \( r(u, v) \) with \(- \log r(u, v)\). If \( r(u, v) = 0 \), then let \( \log r(u, v) = \infty \). The resulting numbers will be between 0 and \( \infty \). This works because by taking the log of a product, we convert multiplication to addition. Then, taking the negative of that quantity makes minimizing the sum of the logs equivalent to maximizing the product of the \( r \)’s. We have that
\[
\max \left( \prod r(u, v) \right) = \min \left( -\log \prod r(u, v) \right) = \min \left( -\sum \log r(u, v) \right)
\]
where each sum or product is taken over \((u, v) \in \text{path}\). Therefore, by minimizing the right hand
side, we maximize the left side. We can perform a standard Dijkstra’s Algorithm after modifying the weights in this manner because the $- \log r$ (where $0 \leq r \leq 1$) is always non-negative.

2. Modify the algorithm

- We can replace the min-heap in Dijkstra’s Algorithm with a max-heap because now we are trying to maximize some quantity.
- Instead of checking $\text{dist}[w] > \text{dist}[v] + w(v, w)$ for some edge $(v, w)$, we replace the $+$ with a $\times$ and the $<$ with a $>$. The resulting expression is: $\text{dist}[w] < \text{dist}[v] \cdot w(v, w)$. We update the distance of some node $w$ a larger value can be achieved by $\text{dist}[v] \cdot w(v, w)$. In other words, going through $v$ to get to $w$ gets us a larger total distance (i.e. greater reliability).

The correctness of either solutions falls immediately from that of Dijkstra’s. Their run-time is therefore also the same as that of Dijkstra’s.
Minimum Spanning Trees

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 3. Compute a minimum spanning tree of the following graph:

Solution

Exercise 3. Compute a minimum spanning tree of the following graph:
**Cut property.** Let \( X \) be a subgraph of \( G = (V, E) \) such that \( X \) is contained in some MST of \( G \). Let \( S \subseteq V \) 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 \( G \).

**Exercise 4.** 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 \( v_1, v_2, \ldots, v_k \) 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 \((u, v_1), (u, v_2), \ldots, (u, v_k)\) and thus by the cut property, the lightest one of them must be contained in some MST.

Furthermore, if we assume that \((u, v_1)\) is the unique lightest edge crossing this cut, then we can say that \((u, v_1)\) is in all MST’s. This is proved as part of the next exercise.

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 \( S \subseteq V \) 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.

  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 5** (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**

First, we start off with a claim: **If \( e \) is the unique lightest edge crossing some cut \( S, V - S \), then \( e \) must be part of every MST.** This is true because imagine an MST that did not use \( e \). Well, by definition of the MST being a tree, it needs to have at least one edge (that is not \( e \)) which crosses
the cut $S, V - S$. But since $e$ was the lightest edge crossing that cut, suppose has an edge $e'$ that crosses the cut, but $w(e') > w(e)$. Now, we can show that our MST isn’t truly an MST by adding in $e$ to the "MST", creating a cycle, and the other edge in the cycle that crosses the cut $e'$, which has strictly greater weight than $e$. Thus, we didn’t have an MST to begin with and therefore MST’s must all use $e$.

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 (by the claim above). 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 6.** (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.
- 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 such 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 65536 vertices and over 2 billion edges!
  - **Graph Representation**: adjacency matrix versus adjacency list.
  - **Algorithm**: Prim’s versus Kruskal’s. Definitely keep in mind that you’re working with a *complete* graph when trying to figure out which one will be more efficient.
• Be careful when (and how often) you seed the random number generator.