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)

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

- 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

Dijkstra(G(V, E, ω), s ∈ V)

1: for v in V do
2:  dist[v] ← ∞
3:  prev[v] ← NIL
4:  dist[s] = 0
5:  H ← {(s, 0)}
6: while H ≠ ∅ do
7:  v ← DELETEMIN(H)
8:  for (v, w) ∈ E do
9:    if dist[w] > dist[v] + ω(v, w) then
10:       dist[w] ← dist[v] + ω(v, w)
11:       prev[w] ← v
12:       INSERT((w, dist[w]), H)

- 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: $O(|V| \cdot \text{deleteMin} + |E| \cdot \text{insert})$

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

General-SP(G(V, E, ω), s ∈ V)

1: for v in V - {s} do
2:  dist[v] ← ∞
3:  prev[v] ← NIL
4:  dist[s] = 0
5: for i = 1, 2, . . . |V| - 1 do
6:  for (v, w) ∈ E do
7:    if dist[w] > dist[v] + ω(v, w) then
8:       dist[w] ← dist[v] + ω(v, w)
9:       prev[w] ← v

- 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 $s \rightarrow u$ using at most $i$ edges is the shortest path from $s \rightarrow v$ using at most $i - 1$ edges plus $ω(v, u)$ 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”.

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.
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:

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. Explain why the word some above cannot be changed to every.
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.

  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.

**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.
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.