Readings

- [Lecture Notes Chapter 20: Dijkstra’s Algorithm]

Review: Single Source Shortest Path (SSSP) Problem

In the Single Source Shortest Path (SSSP) Problem, we are given a graph \( G = (V, E) \), and we want to find a shortest path from a given source vertex \( s \in V \) to each vertex \( v \in V \). From Recitation 6, we proved that BFS solves this problem for unweighted graphs. However, for weighted graphs, we need something a little more robust. Thus, Dijkstra’s algorithm is an algorithm that solves this problem for graphs with non-negative edge weights; some relevant definitions are as follows:

**Shortest Path:** A path from vertex \( s \) to vertex \( t \) with the property that no other such path has a lower total edge weight.

**Negative Weight Cycle:** A cycle with weights that sum to a negative number.

Review: Dijkstra’s Algorithm

Dijkstra’s algorithm finds the shortest path between two given vertices in a weighted graph, assuming that the graph’s edge weights are non-negative. The running time of the algorithm is \( O(|E| \log |V| + |V| \log |V|) \) when the graph is implemented using adjacency lists. The pseudocode for the algorithm is given below:

```
Dijkstra(G, s)
    for each v ∈ V do
        dist[v] = ∞
        parent[v] = NIL
    dist[s] = 0
    S = ∅
    Q = min-priority queue on all vertices, keyed by dist value
    while Q is not empty do
        u = Extract-Min(Q)
        S = S ∪ {u}
        for each v ∈ Adj[u] do
            if v ∈ Q and dist[v] > dist[u] + w(u, v) then
                dist[v] = dist[u] + w(u, v)
                parent[v] = u
```
**Runtime Analysis**

The running time of Dijkstra’s algorithm has two terms: $E \log V$ and $V \log |V|$. We first consider the $V \log |V|$ term: each Extract-Min operation takes $O(\log |V|)$ time, and it is called $|V|$ times because there are $|V|$ vertices.

The $E \log |V|$ term has to do with the relaxation step of Dijkstra’s algorithm. Each edge examined may result in a relaxation of the neighboring node in the heap — a Decrease-Key operation that takes $O(\log |V|)$ time. The number of vertices examined in line 8 above is bounded by the total degree of all vertices, as each vertex is added and popped exactly once from the min-heap. This value is $2|E|$ by the Handshaking Lemma, so in the worst-case we have $2|E|$ decrease-key operations, for a total of $O(|E| \log |V|)$.

This analysis works for easily proving our runtime, but we can actually do a better analysis. Each edge $(u, v)$ can only cause one relaxation, not 2 as the Handshaking Lemma suggests. This is because $(u, v)$ is explored only when node $u$ is popped from the min-heap. This means that when $(u, v)$ is explored from node $v$, we know node $u$ has already been removed, so its key cannot be decreased. Hence, the $O(|E| \log |V|)$ term comes from the $O(\log |V|)$ cost of a Decrease-Key operation, which is called at most $|E|$ times overall.

**Greedy Algorithms**

A greedy algorithm is an algorithm that always makes the locally optimal choice — the best available choice at that moment—in order to find the best globally optimal solution. Greedy algorithms do not always yield optimal solutions, but for many problems they do. Note that Dijkstra’s algorithm solves the “Single Source Shortest Path” problem by following this paradigm — it uses a priority queue structure that always yields the node with the shortest distance from the source node when polled. Consider the set $S$ of vertices in Dijkstra’s whose final shortest-path weights from the source have already been determined. At each step in the algorithm, since we ultimately want to find the shortest path from our source, we use our priority queue to make the locally optimal choice by adding the node with the current shortest distance from the source node to the set $S$.

**Problems**

**Problem 1**

Find the shortest paths from A:

![Diagram of a graph with nodes A, B, C, D, and E connected by edges labeled with distances.](image)

**Solution**

Running Dijkstra’s algorithm produces the following state:
### Problem 2

Does Dijkstra’s algorithm work with negative weights? Why or why not?

#### Solution

No, Dijkstra’s Algorithm does not work on negative weighted graphs. First, if there exists a negative cycle, the concept of a shortest path does not exist. Secondly, a negative weight breaks an important assumption in the canonical proof of correctness for Dijkstra’s algorithm. Specifically, Dijkstra’s algorithm relies on non-negative edge weights to yield the correct solution because it implies that we can never decrease a path’s weight by traversing more edges, allowing us to incrementally make locally optimal decisions to reach a globally optimal solution. However, this assumption breaks when we have negative edge weights. For instance, consider the counterexample below. If we run Dijkstra’s algorithm starting at vertex $a$, we will update $b$ and $c$’s distances to be 4 and 2, respectively. Then, we will conclude that the shortest path from $a$ to $c$ is $a \rightarrow c$, with a weight of 2. However, the actual shortest path from $a$ to $c$ is $a \rightarrow b \rightarrow c$, with a weight of 1. Here, the assumption in the proof of correctness breaks because by traversing the $b \rightarrow c$ edge which has a negative weight, we are able to decrease the weight of the path from $a$ to $c$ that goes through $b$. Hence, running Dijkstra’s here yields an incorrect solution.

![Diagram of counterexample](attachment:diagram.png)

### Problem 3: True or False

1. Dijkstra’s algorithm will not terminate when run on a graph with negative edge weights.

2. If we double the weights of all edges, then Dijkstra’s algorithm produces the same shortest path.

3. If we square the weights of all edges, then Dijkstra’s algorithm produces the same shortest path.

#### Solution

1. **False.** The algorithm will terminate, since in each iteration of our while loop, we still remove the vertex at the root of the heap, and the algorithm terminates when the heap is empty. However, when the algorithm terminates, the output may be incorrect.

2. **True.** Any scaling by a positive factor on the weights does not affect the calculation of shortest paths because we maintain relative path weights. Consider a path with total weight $a$ and a path with total weight $b$, where WLOG $a < b$. When we apply this transformation of doubling edge weights, observe that our first path now has total weight $2a$ and our second path now has total weight $2b$. In other words, since $2a < 2b$ still, scaling by a positive factor maintains these relative path weights, so Dijkstra’s still produces the same shortest path. Alternatively, this is analogous to unit-conversion. For example, converting edge weights from miles to kilometers will not affect the shortest path.
3. False. In contrast to above, squaring the weights of all edges will not always produce the same shortest path because it is not a transformation that preserves relative path weights. As a counterexample, consider the graph below and the shortest path from $s$ to $t$. In this original graph, the shortest path is just from $s \rightarrow t$; however, after squaring the edge weights, the shortest path becomes $s \rightarrow a \rightarrow b \rightarrow c \rightarrow t$.

![Graph Image]

Problem 4
(Adapted from CLRS 24.3-6) 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 $0 < r(u, v) \leq 1$ that represents the reliability of a communication channel from $u$ to $v$. We thus 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. Design an efficient algorithm to find the most reliable path between two given vertices.

Solution
Algorithm: We run a modified version of Dijkstra’s: we initialize the distances to $-\infty$ instead of $\infty$; we initialize the distance of the source node to be 1 instead of 0; we use a max-heap and call EXTRACT-MAX instead of using a min-heap and calling EXTRACT-MIN; and in the edge-relaxation step, we switch the inequality to a $<$ instead of a $>$ and check for products instead of sums. After the algorithm terminates, we backtrack from $v$ to the source $u$ via parent pointers to output the path with maximum reliability. The pseudocode is as follows:

```
Maximum-Reliability(G, s)
for each v \in V do
    dist[v] = -\infty
    parent[v] = NIL

dist[s] = 1
S = \emptyset
Q = max-priority queue on all vertices, keyed by dist value

while Q is not empty do
    u = Extract-Max(Q)
    S = S \cup \{u\}
    for each v \in Adj[u] do
        if v \in Q and dist[v] < dist[u] \times w(u, v) then
            dist[v] = dist[u] \times w(u, v)
            parent[v] = u
```

Proof of Correctness: Since edge weights represent reliabilities, observe that we want to find the path from $u$ to $v$ with the maximum weight, where the weight of a path is now the product (instead of sum) of all weights along the path because reliabilities are independent. At a high-level, our algorithm modifies
Dijkstra's by ensuring we keep track of maximum instead of minimum length paths. Formally, we can prove
the correctness of our algorithm by modifying the proof of correctness for Dijkstra's. That is, consider the
set $S$ at any point in the algorithm's execution. We want to show via induction on $|S|$ that for each $u \in S$,
the path $P_v$ is a path $s \rightarrow u$ with maximum reliability.

**Base Case:** $S = \{s\}$ and $\text{dist}[s] = 1$, so $|S| = 1$ holds because the maximum reliability of any path is just 1.

**Induction Hypothesis:** Assume the claim holds when $|S| = k$ for some $k \geq 1$.

**Induction Step:** We want to show the claim holds for some $S$ with size $k + 1$. Consider the $k + 1$-th vertex
$v$. Let $(u,v)$ be the last edge on our $s \rightarrow v$ path $P_v$. By IH, we know $P_v$ is a $s \rightarrow u$ path with maximum
reliability. Now, we want to show that any other $s \rightarrow v$ path $P$ has reliability at most the reliability of $P_v$.
Note that in order to reach $v$, this path $P$ must have left the set $S$. Let $y$ be the first node on $P$ that is not
in $S$ and let $x \in S$ be the node on $P$ just before $y$ such that we have $(x,y)$.

Observe that $P$ cannot have higher reliability than $P_v$ because it already has at most the reliability of $P_v$
by the time it has left the set $S$. This is because in iteration $k + 1$, our algorithm would have considered
adding $y$ to the set $S$ via this $(x,y)$ edge but rejected it in favor of adding $v$ since we use a max-heap. So,
the reliability of the subpath of $P$ up until $y$ has at most the reliability of $P$. Since edge weights are between
0 and 1, we know that the reliability of $P$ can only decrease as we traverse more edges since we are dealing
with products. Therefore, we know that $P$ cannot have higher reliability than $P_v$ – so $P_v$ is a $s \rightarrow v$ path
with maximum reliability, completing our Induction Step and thus our proof.

**Runtime Analysis:** Note that none of the modifications affect the runtime of Dijkstra's algorithm, so
running our modified version also takes $O((m+n) \log n)$ time. We backtrack no more than $O(n)$ times since
the longest path in a graph has $n-1$ edges. Therefore, this algorithm runs in $O((m+n) \log n)$ time.

**Alternate Solution**

**Algorithm:** Modify $G$ so that the weight of an edge $(u,v)$ is equal to $-\log(r(u,v))$, or the negative log
of its reliability. Run Dijkstra's now on $G$. Backtrack from $v$ to the source node $u$ via parent pointers to
output the path with the maximum reliability.

**Proof of Correctness:** Essentially, we want to prove that our algorithm has modified $G$ such that we
transformed this problem into a shortest path problem. Since edge weights represent reliabilities, observe
that we want to find the path from $u$ to $v$ with the maximum weight, where the weight of a path is now
the product (instead of sum) of all weights along the path because reliabilities are independent. Note that
$-\log r(u,v) = \log(1/r(u,v))$. By taking the inverse of every $r(u,v)$, we have converted this maximization
problem into a minimization problem, and by then taking the log of these inverses, we have converted the
problem of maximizing a product into minimizing a sum because of log properties. Therefore, the transfor-
mation we applied has converted the problem into a shortest path problem. The newly transformed edge
weights are all non-negative, so we know that Dijkstra's algorithm will correctly calculate all shortest paths
and thus backtracking will yield the most reliable path.

**Runtime Analysis:** Modifying $G$ requires iterating through its adjacency list and updating the edge
weights, which takes $O(m+n)$ time. Running Dijkstra's takes $O((m+n) \log n)$ time and we backtrack no
more than $O(n)$ times since the longest path in a graph has $n-1$ edges. Therefore, this algorithm runs in
$O((m+n) \log n)$ time.