# Shortest Paths and Dijkstra's Algorithm

Suhas Arehalli

COMP221 - Spring 2024

## 1 Shortest Paths in Weighted Graphs

- Consider a *weighted* graph $G = (V, E)$ with a weight function $c : E \to \mathbb{R}$.
- The shortest path from a vertex $s \in V$ to a vertex $t \in V$ is a path $P = (s, v_1), (v_1, v_2), \ldots, (v_{k-1}, t) \subseteq E$ such that the length of that path $c(P) = \sum_{e \in P} c(e)$ is minimized.
    - Just like for MSTs, when we want something to be minimized in a definition, we mean that a path from $s$ to $t$ *isn't* the shortest path if there exists a shorter path from $s$ to $t$.
- Note that the SP for weighted graphs is defined not in terms of the number of edges, but in terms of the sum of the weights of the path's edges. Again, mirroring MSTs!

## 2 Dijkstra's Algorithm

### 2.1 The Idea

- A blend between Prim's and BFS!
- Like Prim's, except we build a shortest path tree instead of an MST!
    - This means we consider the cost of the path from $s$ to a vertex rather than just the cost of the new edge!
- Like BFS, we structure our algorithm so that we consider paths in order of increasing length!
- We can apply a slight semi-optimization to the textbook's algorithm by, like we suggested for Prim's, using a priority queue to order the paths we consider.
    - Note that to achieve a properly optimized Dijkstra's, we need to make sure that we can quickly update the priority of an item in the queue (i.e., if we find a faster path to a particular vertex!). This would mean we only ever need to do $|V|$ $\Theta(\log|V|)$ enqueues!
    - This requires a data structure slightly more complex than the binary heaps we've seen (a *Fibonacci* heap), so it's slightly beyond the scope of this class.
- This gets us Alg. 1

**Algorithm 1** A Priority Queue version of Dijkstra's. Be warned: this is fairly unoptimized!

```
function DIJKSTRA($G = (V, E)$, $c : E \to R_{\geq 0}$, $s$, $t$)
    Let $Q$ be a priority queue
    $Q.enqueue(s, 0)$
    $visit(s)$
    while $Q$ is non-empty do
        $v, d \leftarrow Q.dequeue()$
        if $v$ is unvisited then
            $visit(v)$
            if $v == t$ then
                return d
            end if
            for $(v, w) \in E$ do
                $Q.enqueue(w, d + c((v, w))$
            end for
        end if
    end while
end function
```

- Like Prim's selecting edges for the MST, the idea is that once we select a *path* from $s$ to $v$ off of the priority queue, this must be the shortest path from $s$ to $v$! This will structure our proof of correctness.
- Note one last **critical** thing: We're going to assume that edge weights are *non-negative*. That is, we assume that $c$ return non-negative values! We'll see why this matters in the proof of correctness.

## 2.2 Proof of Correctness

Let's start with a small lemma:

**Lemma:** Let $P = (s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v), (v, t)$ be a shortest path from $s$ to $t$ in $G$. Then $P' = (s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v)$ is the shortest path from $s$ to $v$ in $G$.

**Proof:** Suppose for contradiction that $P'$ was not a shortest path from $s$ to $v$. Then there exists some other path $P'^* = (s, w_1), \ldots (w_{k-1}, v)$ that's shorter ($c(P'^*) < c(P')$). Now let's construct $P^* = (s, w_1), \ldots (w_{k-1}, v), (v, t)$. To confirm: we know $(v, t) \in E$, since this edge is in $P$, and since the rest of the path is just $P'^*$, we know that this is a valid path from $s$ to $t$. Now let's consider it's length relative to $P$:

$$\begin{aligned}c(P^*) &= c(P'^*) + c((v, t))\\ c(P^*) &< c(P') + c((v, t))\\ c(P^*) &< c(P)\end{aligned}$$

Which is a contradiction, since $P$ is assumed to be the shortest path from $s$ to $t$!

Thus $P'$ must also be the shortest path from $s$ to $t$.

This tells us that in order to find the shortest path to $t$, we only need to consider other shortest paths from $s$ along the way! In fact we can make this even clearer:

**Lemma:** Let $P$ be a shortest path from $s$ to $t$ that passes through a vertex $v$. That is, $P = (s, w_1), (w_1, w_2), \ldots (w_k, v), (v, w_{k+1}), \ldots (w_{k+l}, t)$ Let $P_1 = (s, w_1), \ldots (w_k v)$ be the sub-path from $s$ to $v$ and $P_2 = (v, w_{k+1}), \ldots, (w_{k+l}, t)$ be the sub-path from $v$ to $t$. Both $P_1$ and $P_2$ are shortest paths.

**Proof:** Since the cases are perfectly symmetric, let's show this to be true for $P_1$ and the same argument holds for $P_2$. Assume for contradiction that there is a shorter path $P_1^*$. Then construct $P^* = P_1^* \cup P_2$, a path from $s$ to $t$, with cost

$$c(P^*) = c(P_1^*) + c(P_2) < c(P_1) + c(P_2) = c(P).$$

Contradiction, since $P$ is the shortest path from $s$ to $t$!

Now to the big claim, that Dijkstra's algorithm is correct. We'll do this using the following pseudo-loop invariant (we've hopefully grown to the point that we can do slightly more weird inductive structures!)

**Statement:** When a vertex $v$ is visited, $d$ is the length of the shortest path from $s$ to $t$.

To make discussing the various values of $d$ over the course of the algorithm's runtime less painful, let $dist(v)$ be the value of $d$ during the iteration of the while loop where $v$ is first visited. Let's also refer to the shortest path from $s$ to some vertex $v$ and $SP(s, v)$.

**Proof:** Proceed by induction *over visitations on vertices.*

**Base Case:** The first vertex visited is $s$, with distance 0.

**Inductive Step:** Our inductive hypothesis is that every vertex that has previously been visited $w$ has been visited with $dist(w)$ as it's shortest path. We now need to show the next vertex we visit, $v$, has $dist(v)$ (the current value of $d$ we pull out of the priority queue $Q$) as the length of the shortest path from $s$ to $v$.

Now let's suppose for contradiction that there is a path $P^* = (s, v_1), (v_1, v_2), \ldots, (v_k, v)$ to $v$ with cost less than $dist(v)$. Since $s$ is the first vertex we visit, $s$ must be visited at this point of time, so we know there is at least one visited vertex along this path. Using that fact, let's break into cases:

Case 1: **There are no unvisited vertices in $P^*$.** This means that $v_k$ is visited. Since $v_k$ is visited, this means that $(v, dist(v_k) + c((v_k, v)))$ has been enqueued. Note that we don't need to consider earlier vertices in the path, since we know that this must be the shortest path to $v_k$ because of the lemma, and by our inductive hypothesis we find that path when we visited $v_k$ previously. However, since we are considering a visit where $v$ is visited for the first time and this is *not* the pair that was dequeued from the priority queue, we know that

$$dist(v) \leq dist(v_k) + c((v_k, v))$$

But since $dist(v)$ represents the cost of the path we've dequeued and $dist(v_k) + c((v_k, v))$ is the length of the supposed shorter path, we reach a contradiction! Thus $dist(v)$ must actually be the shortest path.

Case 2: **Let $v'$ be the first unvisited vertex in $P^*$.** Break $P^*$ into two pieces: $P_1^*$, a path from $s$ to $v'$, and $P_2^*$, a path from $v'$ to $t$. Since the vertex visited before $v'$ must be visited, an entry corresponding to $P_1^*$ (i.e., an entry $(v', c(P_1^*))$ has been added to the priority queue when that prior vertex was visited — again, by IH and the lemma!). Because $v'$ remains unvisited, that entry must remain on $Q$, which means that the cost of the currently under consideration path must be less than or equal to the cost of that path. That is,

$$dist(v) \leq c(P_1^*)$$

Now we use our assumption about edge weights! Since all edges have non-negative weights ($c(e) \geq 0$), the costs of paths must be non-negative ($\sum_{e \in P} c(e) \geq 0$)! Thus $c(P_2^*) \geq 0$, and thus

$$\begin{aligned} dist(v) &\leq c(P_1^*) + c(P_2^*) \\ &\leq c(P^*) \end{aligned}$$

Which contradicts our assumption that $P^*$ has cost less than $dist(v)$! Contradiction! Therefore, $dist(v)$ must be the length of the shortest path from $s$ to $v$.

This completes our proof of the pseudo-loop invariant.

To prove the correctness of Dijkstra's, simply observe that if $v == t$, we return $d = dist(v)$, the length of that shortest path.

## 2.3 Additional notes

- If we want to be super careful, we should realize that our definition of the shortest path problem should probably consider the fact that a shortest path might not exist! Consider a graph where $s$ and $t$ are not connected!
- To do this, we would define the shortest path problem a little more carefully: An algorithm should return the length of the shortest path if it exists, or $NULL$ otherwise.
- It shouldn't take too much argument to see that if $s$ is disconnected from $t$, the inner loop will never have $v == t$, and thus we will exit the while-loop without returning (Good practice to convince yourself!). Adding a return value onto the end of the algorithm to handle non-existence would then handle the $NULL$ case.