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 $P = (s, v_1), (v_1, v_2), \ldots, (v_{k-1}, t) \in E$ such that the length of that path $c(P) = \sum_{e \in P} c(e)$ is minimized. We'll again call the cost of the shortest path $\delta(s, v)$.
  - 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!
- Once again, I'll present this as a Single-Source Shortest Path (SSSP) algorithm: We compute the cost of the shortest path from s to every other vertex $v \in V$.

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.

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

```
function DIJKSTRA(G = (V, E), c : E → R_{≥0}, s, t)
    Let Q be a priority queue
    Let d be an array with d[v] ← ∞ for all v ∈ V.
    Q.enqueue(s, 0)
    discover(s)
    while Q is non-empty do
        v, cost ← Q.dequeue()
        if v is undiscovered then
            discover(v)
            d[v] ← cost
            for (v, v') ∈ E do
                Q.enqueue(v', cost + c((v, v')))
            end for
        end if
    end while
    return d
end function
```

- This gets us Alg. **??**
- 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

Now to the big claim, that Dijkstra's algorithm is correct.

We'll do this in a similar way to how we proved BFS correct:

Lemma 1. *Let P_t be a shortest path from s to t that passes through a vertex v_i. That is, $P = s, v_1, v_2, \ldots, v_i, \ldots, v_k, t$ Let $P_1 = s, v_1, \ldots, v_i$ be the sub-path from s to v_i and $P_2 = v_i, \ldots, v_k, t$ be the sub-path from v_i 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 $P_t' = P_1^* \cup P_2$, 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 a few that that should be self-evident, given our understanding of BFS, as the proofs transfer over to weighted graphs!

Lemma 2. *For all $s \in V$ and $(v, v') \in E$,*

$$\delta(s, v') \leq \delta(s, v) + c(v, v')$$

and via this first lemma, we see that, like in BFS

Lemma 3. *Throughout our algorithm, $d[v] \geq \delta(s, v)$*

Now, to the critical one:

Lemma 4. *If the pair (v, c) is dequeued prior to pair (v', c') in Dijkstra's, then $c < c'$.*

Proof. First note that since our priority queue dequeues the lowest cost pair in the queue at the time of dequeuing, what's left to show is that what we enqueue will never be smaller than something that's previously been dequeued. This is our strategy.

We will prove this via induction over enqueues and dequeues. Formally, we'll claim that every element dequeued prior (v'', c'') has $c'' \leq cost$ for all $(v, cost) \in Q$

The base case is simple, s is the first vertex removed, so the statement is trivially true.

Our inductive step follows our strategy closely: At the next dequeue, we remove the minimum cost element in our priority queue, $(v, cost)$. Two things are critical here: First, by our IH, all prior elements (v'', c'') have lesser cost: $c' \leq cost$. Second, since it's a priority queue, are newly dequeued vertex has a lower cost than everything else in our queue, so the statement holds.

Then we have a series of enqueues of the form $(v', cost + c((v, v')))$. Observe that, by our assumption the $c(v, v') \geq 0$, for all (v'', c'') dequeued prior,

$$cost + c((v, v')) \geq cost \geq c'$$

Thus all of already dequeued elements must have cost less than the new elements we added! □

If you're comparing to our BFS proof, this statement is the equivalent as our queue ordering property!

Now, correctness in full!

Statement. Dijkstra's is correct (we return d such that $d[v] = \delta(s, v)$) for any graph such that $c(e) \geq 0$ for all $e \in E$.

Proof. Assume for contradiction that we end Dijkstra's with some number of vertices $v \in V$ with $d[v] \neq \delta(s, v)$.

Let v' be the vertex such that $d[v] \neq \delta(s, v')$ and $\delta(s, v')$ is minimized. Observe that $v \neq s$, since we guarantee that $d[s]$ is assigned correctly.

Now consider the true shortest path to v', $s, v_1, \ldots, v_k, v, v'$, where v is the penultimate vertex along that path. Note that since $c(v, v') \geq 0$, the sub-path from s to v is shorter than that to v, and by Lemma **??** we know this is actually the shortest path from s to v'. Thus, because of how we chose v', we know that v has been assigned the correct value, $d[v] = \delta(s, v)$. This means v has had it's adjacent vertices added to the queue, and our while-loop condition tells us it has been dequeued at some point. Thus we must have had $(v', \delta(s, v) + (v, v')) = (v', \delta(s, v'))$ dequeued before termination. We have 2 cases:

Case 1: v' was already discovered. Via Lemma **??**, we know that since it was discovered prior, it has lower cost. Since we got $d[v']$ wrong by assumption, this implies that $d[v'] < \delta(s, v')$. However, we know via Lemma **??** that we can only overestimate cost, so we reach a contradiction.

Case 2: v' was undiscovered. Here, we assign $d[v'] = \delta(s, v')$ and mark v' as discovered. Since discovery ensures we only assign once, we will end the algorithm with $d[v'] = \delta(s, v')$, which contradiction our assumption that we assigned v' the wrong cost. $\square$