Section 4: Dijkstra's Algorithm

Amy Zhao

February 16, 2024

1 Dijkstra's Algorithm

At a high level, Dijkstra's algorithm is an algorithm which finds the shortest path between a starting vertex and another vertex in the graph. During each iteration, we expand the search by identifying the closest not-yet-processed vertex from the starting vertex.

```
Input: Graph G = (V, E) given as an adjacency list, starting vertex s
Post : Shortest distances from s to v ∈ V, stored in dist[v]

Function Dijkstra(G = (V, E), s):
    Q ← new priority queue

    for v ∈ V do
        distance[v] ← ∞
        previous[v] ← null
        if v ≠ s then
            add v with its distance to Q
    end

    distance[s] ← 0

    while Q is not empty do
        u ← vertex in Q with minimum distance

        for each unvisited neighbor v of u do
            temp ← distance[u] + weight[u, v]
            if temp < distance[v] then
                distance[v] ← temp
                previous[v] ← u
        end
    end
```

2 What Can Go Wrong?

2.1 Negative Edges

In this section, we illustrate a few examples when performing Dijkstra's will yield the wrong answer. First, suppose we have a graph $G = (V, E)$ with edge weights $L = \{\ell_e : e \in E\}$, and we allow for the possibility of negative edges. Assume there are no negative cycles. Evidently, we know that Dijkstra's algorithm doesn't work on graphs with negative edges. Consider a modification to a graph with at least one negative edge:

- Find the minimum edge weight in L, denoted by $\ell^* < 0$.
- For all edges, change the edge weight to $\ell'_e = \ell_e + |\ell^*| + 1$, so that all of the new edge weights are positive. Denote the new set of edges as $L' = \{\ell'_e : e \in E\}$.
- Run Dijkstra's algorithm as usual with the new set of edge weights.

Will this change yield the shortest paths from a source vertex to all other vertices? Can we find a simple example or counterexample first?

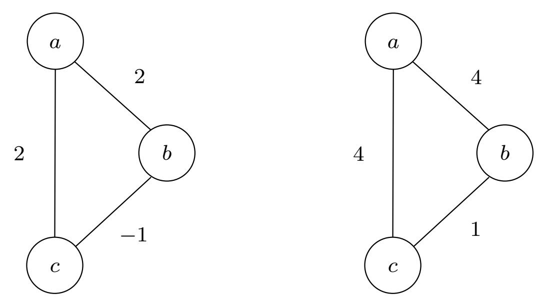

Figure 2.1: Example of a single graph where Dijkstra's algorithm will not yield the correct answer on the modified graph. The shortest path in the original graph from a to c is $a \to b \to c$, while the shortest path in the modified graph is $a \to c$.

We claim that the proposed modification does not find the shortest path and provide a proof. For notation purposes, let u be the source vertex, and let $v \in V$ be any other vertex in the graph. Let $W(P)$ be the sum of edge weights on a path P using the edge weights from the original graph, and let $W'(P)$ be the sum of the edge weights on a path P using the modified weights:

$$W(P) = \sum_{e \in P} \ell_e \qquad\qquad W'(P) = \sum_{e \in P} \ell'_e.$$

We set P to be the shortest path using the original edge weights. First, consider a path P containing k edges and the resulting sum on the modified edges:

$$\begin{aligned} W'(P) = \sum_{e \in P} \ell'_e &= \sum_{e \in P} (\ell_e + |\ell^*| + 1) \\ &= \left(\sum_{e \in P} \ell_e \right) + k(|\ell^*| + 1) \\ &= W(P) + k(|\ell^*| + 1). \end{aligned}$$

Consider a second path Q from u to v with $j < k$ edges but greater weight $W(Q) > W(P)$. We write the sum $W'(Q)$ as:

$$W'(Q) = \sum_{e \in Q} \ell'_e = W(Q) + j(|\ell^*| + 1).$$

These relationships yield the following implication:

$$\begin{aligned} &W(Q) - W(P) < (k - j)(|\ell^*| + 1) \\ &\qquad \implies W'(Q) = W(Q) + j(|\ell^*| + 1) < W(P) + k(|\ell^*| + 1) = W'(P). \end{aligned}$$

In general, $W(P) < W(Q)$ does not imply $W'(P) \leq W'(Q)$, and thus the modification does not always return the shortest path with respect to the original edge weights.

2.2 Adding a Constant

Now, consider a positive weighted graph, and assume we use Dijkstra's algorithm to compute a shortest path P between two vertices u and v. Will Dijkstra's algorithm yield the same path if we add a positive constant to every edge?

Suppose the original graph has two possible paths from s to t: one which includes 50 edges with $\ell_e = 1$ and one which includes a single edge with $\ell_e = 51$. When we run Dijkstra's algorithm on this graph, the shortest path will be the first path because it has a total weight of 50, as compared to the second which has total weight of 51.

If we add a constant to each edge of the graph, say $c = 50$, the total weight of the first path will be $50(1 + 50) = 2550$, while the weight of the second path will be

$51 + 50 = 101$. Thus, this clearly illustrates why Dijkstra's will not work on this modified graph to return the shortest path length of the original graph. Paths with a greater number of edges will have a greater amount of weight added:

$$W'(P) = \sum_{e \in P}(c + \ell_e) = |P|(c) + \sum_{e \in P} \ell_e = |P|(c) + W(P),$$
$$W'(Q) = \sum_{e \in Q}(c + \ell_e) = |Q|(c) + \sum_{e \in Q} \ell_e = |Q|(c) + W(Q).$$

$|P|(c) \gg |Q|(c)$ if $|P| \gg |Q|$, and this is solely dependent on the number of edges in the path.

2.3 Scaling by a Constant

Taking the same setup as in the previous section, we decide to scale each edge by a constant positive factor, instead of adding a constant. Will Dijkstra's algorithm work on this modified graph to produce the shortest path from the original?

Let P be the shortest path from s to t in the original graph, and let Q be an alternate path from s to t in the original graph. Assume also that the number of edges in the two paths may be different. We denote the weight of the original path with W and the weight of the modified path with W'.

As P is the shortest path, we have $W(P) < W(Q)$. We'll look at the scaling modification:

$$W'(P) = \sum_{e \in P}(c \cdot \ell_e) = c \cdot \sum_{e \in P} \ell_e = c \cdot W(P).$$
$$W'(Q) = \sum_{e \in Q}(c \cdot \ell_e) = c \cdot \sum_{e \in Q} \ell_e = c \cdot W(Q).$$

Thus, we have:

$$W'(P) = c \cdot W(P) < c \cdot W(Q) = W'(Q) \Longrightarrow W'(P) < W'(Q),$$

which implies that we will obtain the same shortest path in the modified graph when we run Dijkstra's.

3 Example: Fuel Capacity

Suppose we have a set of cities connected by highways, given in the form of an undirected graph $G = (V, E)$. Each highway $e \in E$ has a specified length $\ell_e > 0$. You are planning a road trip and want to determine if you can travel from city s to city t with a car that has a fuel capacity of L miles. You can refuel at each city, but not between, which means you can take a route if every one of its highways is smaller in length than L.

3.1 Reachability

If we know the car's fuel capacity, how do we determine if we can reach t from s? Revisiting the previous discussion section's examples, we can consider modifying the graph by taking out any edges that are longer than L. Running DFS on the resulting graph will check reachability between s and t, yielding the correct answer. Modification and DFS are $O(|V| + |E|)$ runtime.

3.2 Determining Needed Capacity

However, we face a trickier problem if we don't know the car's capacity. Suppose we are purchasing a new vehicle and want to ensure we buy a car with enough fuel capacity to travel from s to t. Notably, we can refuel at each city as needed, so the car's capacity need only to be at least as large as the longest edge of the smallest path from s to t. In other words, the fuel capacity needed is constrained by the s to t path whose longest edge is the smallest; we'll call this the smallest max length path.

To determine this length, we adapt Dijkstra's algorithm. At each vertex u, we track the constraining edge, instead of the distance, in $m[u]$. Every time we dequeue a vertex u, we compare the length of the edge $e = \{u, v\}$ with the current smallest max length:

- $\max\{\ell_2, m[u]\} < m[v]$: set $m[v] = \max\{\ell_e, m[u]\}$ and rebalance the priority queue; do nothing otherwise.

The algorithm returns the value at $m[t]$, which will accurately report the constraining edge in the smallest path.

Input : Graph $G = (V, E)$ given as an adjacency list with weights $\ell_e > 0$ for $e\ inE$, starting vertex s, target vertex t

Output: Smallest max length path

```
Function FuelConstraint(G = (V, E), s, t):
    for v ∈ V do
        if v = s then
            m[v] ← 0
        else
            m[v] ← ∞
    end

    Q ← new priority queue with objects v ∈ V, associated with priority m[v]

    while Q is not empty do
        v ← ExtractMin(P)
        for w ∈ Adj[v] do
            if m[w] > max{ℓ_vw, m[v]} then
                m[w] ← max{ℓ_vw, m[v]}
                Adjust priority key for w
        end
    end

    return m[t]
```

Tracking the smallest max length path does not alter the runtime of Dijkstra's algorithm, so the overall time complexity remains $O((|E| + |V|)\log(|V|))$.

4 Example: Number of Shortest Paths

We know that Dijkstra's algorithm is used to find the shortest paths in a weighted undirected graph. Is it possible to augment the algorithm to count the number of shortest paths from s to t? Recall that in running Dijkstra's algorithm, all other vertices with a shorter path from s will have already been explored before we expand vertex a in Dijkstra's algorithm. This means that we know the shortest path from s to these vertices is known at this point, and we can simply record additional information about the shortest paths based off a couple comparisons.

At some iteration of Dijkstra's algorithm, let's consider the expansion of vertex u on the edge $\{u, v\}$ with length ℓ_e:

- $\text{dist}[v] > \text{dist}[u] + \ell_e$: We find a shorter path to v, so we set $\text{dist}[v] = \text{dist}[u] + \ell_e$, and we update the number of paths to v to be equal to the number of paths from s to u, numpaths$[v]$ = numpaths$[u]$.

- dist$[v]$ = dist$[u] + \ell_e$: We find a path with the same distance as previously recorded, so there are no updates to make to the distance array. However, we increase the number of paths to v, numpaths$[v]+=$ numpaths$[u]$, as we care about the path distances, rather than the actual edges of the paths.

The algorithm returns the value at numpaths$[t]$, which will accurately report the number of shortest paths because the value is updated in parallel with any updates to the distance array. This means that whenever v has the smallest distance value, there are no other unexplored paths from $s \to v$ that could decrease the value we've stored at dist$[v]$.

```
Input  : Graph G = (V, E) given as an adjacency list, starting vertex s, target vertex t
Output: Number of shortest paths from s to t
Function NumShortestPaths(G = (V, E), s, t):
    for v ∈ V do
        dist[v] ← ∞
        numpaths[v] ← 0
    end

    dist[s] ← 0
    numpaths[s] ← 1
    Q ←new priority queue

    while Q is not empty do
        v ← vertex in Q with minimum distance

        for w ∈ Adj[v] do
            if dist[w] > dist[v] + ℓ_vw then
                dist[w] ← dist[v] + ℓ_vw
                ChangeKey(P, w)
                numpaths[w] ← numpaths[v]
            else
                numpaths[w] ← numpaths[w] + numpaths[v]
        end
    end
    return numpaths[t]
```

The runtime of the algorithm is the same as Dijkstra's, $O((|E| + |V|)\log|V|)$, as the comparisons and updates to the numpaths array don't impose any additional dominating runtime.