Dijkstra's Single Source Shortest Path Algorithm

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

1 The Main Assumption and Its Impact

We look back at the problem of computing the shortest path from a source vertex s to a target vertex t. Recall that computing the shortest path in DAGs has a prerequisite – the topological order. In general graphs, we may not have a topological order, because the graph may have a cycle. Consequently, our previous algorithm fails.

Worse still, even the concept of shortest path is somewhat ambiguous for a graph with arbitrary edge weights. Visualize a graph which has a cycle and all edge weights are negative. Now, if a path from s to t goes via this cycle, then we can simply loop around the cycle infinitely many times to bring the length of the shortest path down to $-\infty$. Practically, this makes little sense.

Summarizing, in a general graph with arbitrary weights, what we are really looking for is a shortest path that visits a vertex at most once. Unfortunately, this problem is very hard. In fact, it is one of the NP-hard problems, which loosely translates to it being extremely unlikely to even have a polynomial time solution (in the number of vertices), let alone an efficient one.[1] So, if we want a decent solution, assumptions are necessary.[2]

Dijkstra's algorithm assumes: **all edge weights are non-negative.**[3]

Let $v_1, v_2, \ldots, v_k$ be the only vertices that have an outgoing edge to t. Then, any path from s to t must travel via one of these k vertices; in particular, the shortest path must also travel through one of these k vertices. Let $D(x, y)$ represent the length of the shortest path from x to y, and for any edge from u to v, let $W(u, v)$ represent its weight. The main intuition is that a shortest path from s to t satisfies the following equation:

$$D(s,t) = \min\{D(s,v_1) + W(v_1,t), D(s,v_2) + W(v_2,t), \ldots, D(s,v_k) + W(v_k,t)\}$$

In other words, the length of the shortest path from s to t can be obtained by adding the weight of the edge (v_i, t), $1 \leq i \leq k$, to the shortest path from s to v_i and then picking the minimum.

Recall that in acyclic graphs, v_1 through v_k are ahead in topological order compared to t. That allowed us to process vertices in topological order. Alas, we do not have that luxury over here.

The non-negative assumption, however, gives us a crucial way of dissecting the problem. We observe that $D(s, v_i) \leq D(s, t)$, where $1 \leq i \leq k$, i.e., the length of the shortest path from s to any v_i is at most the length of the shortest path from s to t.

[1] The problem is closely associated with the Hamiltonian path problem, which is stated as follows: *is there a path that visits each vertex exactly once?* This is an NP-complete problem, which very loosely means that it is unlikely that one can answer this question in polynomial time. If you have heard of the famous $P = NP$ conjecture, know that it can be reformulated as "*is there a polynomial time algorithm for the Hamiltonian path problem?*" If you have not heard of the conjecture, I encourage you to look it up.

[2] Well at least until someone proves that $P = NP$.

[3] Pretty realistic assumption, as graphs with negative edge weights are rare.

2 The Algorithm

The main idea is **process vertices in non-decreasing order of their shortest path distance from** s. We already know that BFS does something of this sort, and it is no surprise that Dijkstra's algorithm can be viewed as BFS, albeit with some modifications.

We keep track of the shortest path distances from the source vertex using a distance array $dist[n]$, where n is the number of vertices in the graph. At any point, $dist(v)$ stores the distance of the shortest path from s to v that has been found so far. Additionally, we also use *closed* to keep track of the vertices to which shortest path has already been found. At any point, if v is in *closed*, then $dist(v)$ records the length of the shortest path from s to v. A vertex v is in *open* if we have found a path to v, but we are not sure if it is the shortest path or not. Specifically, a vertex v is in *open* if v is not in *closed*(v) and $dist(v) \neq \infty$. We use a *parent* array, where $parent(v)$ is the vertex preceding v in the shortest path from s to v that has been found so far. Thus, if v is in *closed* and $parent(v) = u$, then u is the vertex that precedes v on the shortest path from s to v.

Dijkstra's Shortest Path Algorithm

- Initially $dist(s) = 0$ and $dist(v) = \infty$, for all $v \neq s$
- Initially $open = \{s\}$, $closed = \varnothing$ and $parent(v) = \varnothing$, for all v
- While there exists an open vertex, execute:
  - Let u_{min} be an open vertex with the minimum *dist* value.
  - add u_{min} to *closed*
  - For each adjacent vertex v of u_{min}
    - Let $len = dist(u_{min}) + W(u_{min}, v)$
    - If $len < dist(v)$, then
      - $dist(v) = len$
      - $parent(v) = u_{min}$
      - if v is not in *open*, add v to *open*

Constructing a Shortest Path from s to v

- Initialize $path = \langle v \rangle$ and $x = v$
- While $parent(x) \neq \varnothing$, execute:
  - add $parent(x)$ at the front of *path*
  - set $x = parent(x)$

Constructing a Shortest Path Tree

- Initially, simply add all the vertices without adding any edges
- For each added vertex v, add an edge from $parent(v)$ to v

2.1 Simulation

The table below simulates the shortest path algorithm on the graph in the following figure.

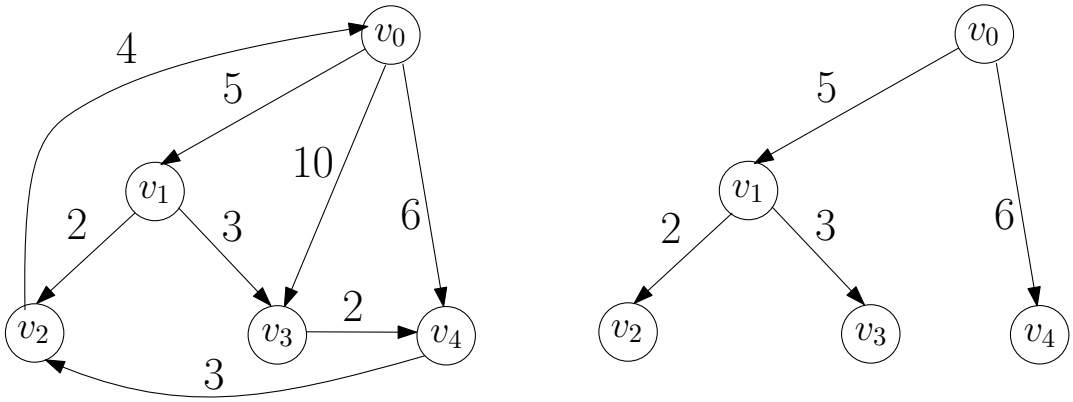

Figure 1: A graph (left) and its shortest path tree (right)

	dist					open	closed	parent				
	v_0	v_1	v_2	v_3	v_4			v_0	v_1	v_2	v_3	v_4
At Start:	0	∞	∞	∞	∞	$\{v_0\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
Min open vertex: v_0 Relaxing v_0	0	5	∞	10	6	$\{v_1, v_3, v_4\}$	$\{v_0\}$	$\varnothing$	v_0	$\varnothing$	v_0	v_0
Min open vertex: v_1 Relaxing v_1	0	5	7	8	6	$\{v_3, v_4, v_2\}$	$\{v_0, v_1\}$	$\varnothing$	v_0	v_1	v_1	v_0
Min open vertex: v_4 Relaxing v_4	0	5	7	8	6	$\{v_3, v_2\}$	$\{v_0, v_1, v_4\}$	$\varnothing$	v_0	v_1	v_1	v_0
Min open vertex: v_2 Relaxing v_2	0	5	7	8	6	$\{v_3\}$	$\{v_0, v_1, v_4, v_2\}$	$\varnothing$	v_0	v_1	v_1	v_0
Min open vertex: v_3 Relaxing v_3	0	5	7	8	6	$\varnothing$	$\{v_0, v_1, v_4, v_2, v_3\}$	$\varnothing$	v_0	v_1	v_1	v_0