Dijkstra's Algorithm

Michael Levet

December 19, 2015

1 Introduction

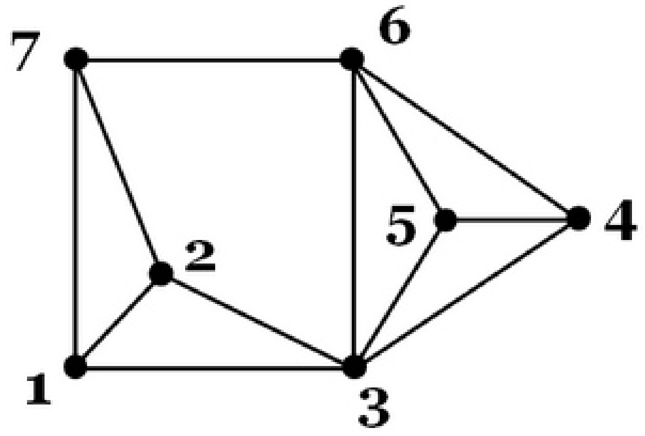
This tutorial will introduce Dijkstra's algorithm, including a proof of correctness and time complexity analysis.

Dijkstra's algorithm is used to find the shortest path between two vertices of a graph. More formally, we fix a starting vertex in the graph, vertex a. Dijkstra's algorithm then returns the shortest path from vertex a to every other vertex in the graph. We assume the graph is weighted and has no negative weights.

2 Breadth-First Traversal

Dijkstra's algorithm is an adaptation of the breadth-first traversal algorithm. So it is important to first understand the breadth-first traversal (BFS) algorithm. The BFS algorithm accepts a graph $G(V, E)$ and initial vertex $a \in V$. The algorithm then manages a queue, which dictates the order to visit the vertices. It then visits each of a's neighbors, pushing them onto the queue as they are visited. Then as long as the queue is non-empty, we poll a vertex, mark it as visited, and add all of its unvisited neighbors to the queue.

Let's consider an example. Given the graph below and the starting vertex 1, we examine the order in which the vertices are visited. While you can choose to visit neighbor vertices in any order, I will visit them from lowest label to highest label.



Start at the initial vertex 1 and mark it as visited. Now we visit each neighbor in the order $2, 3, 7$, mark them as visited, and push each onto the queue respectively. So the queue is now: $Q = [2, 3, 7]$.

Next, we poll 2 from the queue, leaving $Q = [3, 7]$. We have that 2's neighbors are $1, 3, 7$, all of which have been visited. So we do nothing. Next, poll 3 from the queue. We push its unvisited neighbors- $4, 5, 6$ onto the queue, marking them as visited along the way. This leaves $Q = [7, 4, 5, 6]$.

Notice now that the vertices in Q have no unvisited neighbors in the graph. So the order the vertices are visited by the algorithm is: $1, 2, 3, 7, 4, 5, 6$.

3 Dijkstra's Algorithm

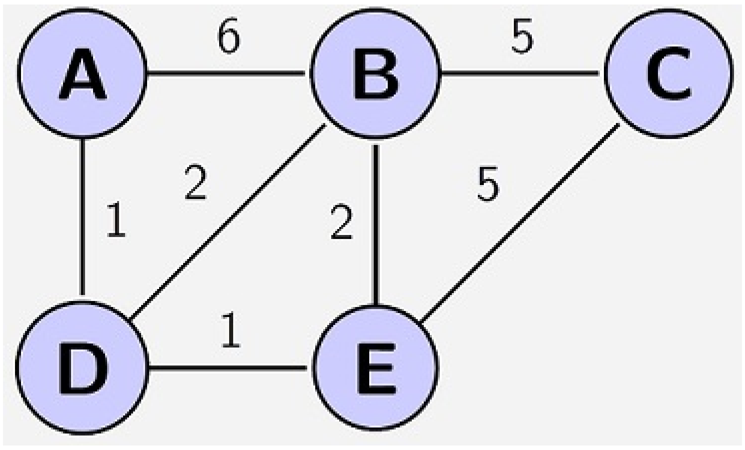
Dijkstra's Algorithm works similarly as the BFS algorithm. We begin with a weighted graph $G(V, E, W)$ where W is the weight function $W : E \to \mathbb{R}^+$, as well as an initial vertex $a \in V$. That is, each edge has a non-negative weight. With each vertex, we associate a distance marker which denotes the distance from a, as well as the predecessor in the shortest path found by the algorithm. This allows us to not only find the distance of the shortest $a - v$ path for any vertex $v \in V$, but to explicitly access the path as well.

We begin by starting at a, and marking every other vertex as having distance infinity from a. We visit each neighbor v of a and: first set v's distance marker to $W(a, v)$ then v's predecessor to a. Then, we push vertex adjacent to a onto a priority queue. The priority queue orders vertices by their distance markers. Finally, we mark the initial vertex as visited. No marked vertex can be used in future iterations.

Now, while the priority queue still has elements, we do the following. First, poll a vertex v from the priority queue. For each unvisited neighbor x of v, we check if $\text{dist}(a, v) + W(v, x) < \text{dist}(a, x)$. If this condition is satisfied, we update x's distance marker to $\text{dist}(a, v) + W(v, x)$. Next, we update x's predecessor to v, then update x's position in the priority queue.

After visiting all of v's neighbors, we mark v as visited.

Let's work through an example. Consider the following graph, and suppose we want vertex A as the root.



We start by setting $\text{dist}(A, B) = 6, \text{dist}(A, D) = 1$. Then we set B's and D's predecessor to A. Next, we push B and D into the priority queue, which is ordered: $[D, B]$. Finally, we mark A as visited.

Now poll D from the priority queue. We note $\text{dist}(A, D) + W(D, B) = 3 < \text{dist}(A, B) = 6$. So update $\text{dist}(A, B) := 3$ and set B's predecessor to D. Now, examine (D, E). We set $\text{dist}(A, E) = \text{dist}(A, D) + W(D, E) = 2$ and E's predecessor to D. We now push E onto the priority queue, which stores the elements in order: $[E, B]$. Now mark D as visited.

Now poll E from the priority queue. We note $\text{dist}(A, E) + W(E, B) = 4 > \text{dist}(A, B)$, so we discard (E, B). Now consider (A, C). Since $\text{dist}(A, E) + W(E, C) = 7 < \infty$, so we set $\text{dist}(A, C) := 7$ and C's predecessor to E. Next, we push C onto the priority queue, which stores the elements in order $[B, C]$. Now mark E as visited.

Now poll B from the priority queue. We note $\text{dist}(A, B) + W(B, C) = 8 > \text{dist}(A, C)$, we discard (B, C). We mark B as visited. The priority queue now contains $[C]$ and no other vertices are unvisited. After polling C, the algorithm terminates.

The lengths of the shortest paths are as follows:

- $\text{dist}(A, B) = 3$
- $\text{dist}(A, C) = 7$

- $\text{dist}(A, D) = 1$
- $\text{dist}(A, E) = 2$

An implementation and explanation is provided with the main tutorial.

4 Analysis of Dijkstra's Algorithm

In this section, we analyze the correctness and complexity of Dijkstra's Algorithm. The design of this algorithm leverages the optimal substructure exhibited by the shortest path problem. Formally, a problem is said to exhibit the **optimal substructure property** if an optimal solution contains within it solutions to the present optimal subproblems. We prove this formally.

Claim 4.1. *Let G be a connected graph and let $x, y \in V$. Let P be a shortest $x - y$ path. Then for any pair of vertices $a, b \in P$, it follows that P contains a shortest $a - b$ path.*

Proof. Suppose to the contrary. Let P be a shortest $x - y$ path such that for vertices $a, b \in P$, that the $a - b$ sub-path in P is not a shortest $a - b$ path. Let Q be a shortest $a - b$ path. We replace the $a - b$ sub-path in P with Q to obtain an $x - y$ path shorter than P, a contradiction. □

Any proof of correctness for Dijkstra's Algorithm leverages the optimal substructure problem. The main idea is this: when the algorithm marks a vertex v as visited, v's distance marker is the length of any shortest path from the initial vertex to v.

Before proving Dijkstra's algorithm, define $d : V \times V \to \mathbb{R}^+$ be the shortest path metric on the graph. That is, for any pair of vertices x and y in the graph, $d(x, y)$ is the length of any shortest $x - y$ path.

Theorem 4.1. *Let G be a connected graph. Dijkstra's algorithm terminates. Let v be the initial vertex used by the algorithm. Upon termination, the distance $\text{dist}(v, y)$ computed by the algorithm is equal to $d(v, y)$ for every vertex $y \in G$.*

We begin by showing the algorithm terminates.

Claim 4.2. *Dijkstra's algorithm terminates.*

Proof. Suppose to the contrary that the algorithm does not terminate. Then the priority queue always has an element. This implies that there is always some unvisited vertex in the graph added to the priority queue upon polling some existing vertex already within the priority queue. This contradicts the assumption that the graph is finite. □

Claim 4.3. *Upon termination, the distance $\text{dist}(v, y)$ computed by the algorithm is equal to $d(v, y)$ for every vertex $y \in G$.*

Proof. The proof is by induction on the number of vertices polled from the priority queue. When no elements have been polled from the priority queue, we have correctly computed $\text{dist}(v, v) = d(v, v) = 0$. Suppose the claim holds true for the first $k - 1$ elements polled from the priority queue. Let x be the kth vertex polled from the priority queue.

Suppose that $\text{dist}(v, x) > d(v, x)$. By the algorithm, $\text{dist}(v, x)$ is computed using only the previous k marked vertices (the previously polled $k - 1$ vertices and the initial vertex v). Then an unmarked vertex w must be present in any shortest path from v to x. Let P be the $v - x$ path computed by the algorithm. Let P' be a shortest $x - v$ path containing w. Without loss of generality, suppose w is the first unmarked vertex in P'. By Claim 1, the $v - w$ sub-path in P' is a shortest $v - w$ path. It is necessary that $d(v, w) \leq d(v, x)$ since w is along a shortest $v - x$ path. It follows that w's predecessor would have visited w and pushed it onto the priority queue. So w would have been polled from the priority queue before x, a contradiction.

It follows that the algorithm correctly computes the lengths of the shortest $v - y$ paths for all vertices $y \in G$. □

Theorem 4.2. *Dijkstra's Algorithm runs in $\mathcal{O}(E \log V)$ time, where E is the number of edges and V is the number of vertices in the graph.*

Proof. Suppose we use a heap where the key can be updated as the priority queue. Polling the vertex with the smallest distance marker from the heap takes $\mathcal{O}(\log V)$ time. For each polled vertex, we must evaluate all of its neighbors. While a vertex can have at most $V - 1$ neighbors, there are E total neighbors to be considered. When evaluating a vertex's neighbor, updating the distance takes $\mathcal{O}(1)$ time. However, updating the vertex's position in the heap takes $\mathcal{O}(\log V)$ time. So the runtime is $\mathcal{O}(V \log V + E \log V)$. Since $V \in \mathcal{O}(E)$ in the case of a connected graph, we have the runtime is $\mathcal{O}(E \log V)$. $\square$