

## Single-Source Shortest Paths

**Reading:** CLR Sections 23.1 -- 23.2, 25.1 -- 25.2

---

### Single-Source Shortest Paths Problem

*Definitions:*

In a weighted, directed graph  $(G, w)$ , the **weight of a path**  $p = \langle v_0, v_1, \dots, v_k \rangle$  is

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i).$$

The **shortest-path weight** from  $a$  to  $b$  is  $w(a,b) = \min\{w(p) \mid p \text{ paths}(a,b)\}$ .

Note:  $\min\{\} = \infty$ .

A **shortest path** from  $a$  to  $b$  is any path  $p$  such that  $w(p) = w(a,b)$ . (May not be unique.)

The **single-source shortest path problem**: given a weighted directed graph  $((V, E), w)$  and a source vertex  $s$  in  $V$ , find a shortest path from  $s$  to every vertex of  $V$ .

*Notes:*

- If a path has negative weight edges in a cycle, then shortest path is not defined. Some algorithms (like the Dijkstra algorithm we will study) assume non-negative weights. Other algorithms (such as Bellman-Ford, which we will not study) can handle negative weight edges as long as they don't appear in cycles.
- The problem of finding the shortest path between two particular vertices (**the single-pair shortest path problem**) may seem easier than the single-source shortest path problem, but no solution for the single-pair problem is known that is asymptotically faster than a solution for the single-source problem!
- The **all-pairs shortest path problem** finds the shortest path between every pair of vertices. We shall not study this problem this semester; CLR Chapter 26 contains details.

---

## Relaxation

Shortest path algorithms maintain for each vertex in the graph:

- a **shortest-path estimate**  $d_s[v]$  (s,v) and
- a **shortest-path parent**  $\text{parent}[v]$  such that  $d_s[v] = d_s[\text{parent}[v]] + w(\text{parent}[v], v)$ .

```
Initialize-Single-Source(G,s)
  for v vertices(G) do
     $d_s[v] = \infty$ 
     $\text{parent}[v] \leftarrow \text{nil}$ 
   $d_s[s] = 0$ 
```

Relaxation is an operation on edges (a, b) that attempts to reduce the shortest-path estimate for b.

```
Relax((a, b), w)
  if  $d_s[b] > d_s[a] + w(a,b)$  then
     $d_s[b] = d_s[a] + w(a,b)$ 
     $\text{parent}[b] = a$ 
```

Shortest path algorithms work by initializing  $d_s[v]$  as above and then repeatedly relaxing edges until  $d_s[v] = d_s[s] + w(s,v)$ . A **shortest-path tree** rooted at the source s is induced by the parent fields.

---

## Dijkstra's Algorithm

*Idea:* Grow a shortest-path tree from the source. Every node maintains a shortest-path estimate and parent that indicates the final edge of a path with this estimate. At each step, add the node with the smallest estimate to the tree via the edge to its parent. Use a priority queue to manage extracting the node with the smallest shortest-path estimate.

This algorithm is greedy in the sense that, at each step, it adds the "best" node (node with smallest shortest-path estimate) to the growing shortest-path tree.

```

{Note: PQ stands for Priority Queue}
Dijkstra(G, w, s)
  Initialize-Single-Source(G,s)
  shortest  {}                                {vertices at which  $d_s[v] = (s,v)$ }
  Q    Build-PQ (vertices(G))  {Invariant: Q contains
                                (V - shortest) ordered by  $d_s[v]$ }

  while not PQ-Empty?(Q) do
    a    PQ-Extract-Min(Q)
    shortest  shortest  {a}
    for b  Adj[a] do
      Relax((a,b), w)
      PQ-Decrease-Key(Q, b,  $d_s[b]$ )

```

*Analysis:*

- Build-PQ called once on  $|V|$  elements
- PQ-Extract-Min called  $|V|$  times
- Relax and PQ-Decrease-Key called  $|E|$  times.

Priority Queue implementation	Build-PQ	$ V  \cdot$ PQ-Extract-Min	$ E  \cdot$ PQ-Decrease-Key	Total
unsorted array/list	$O(V)$	$O(V^2)$	$O(E)$	$O(V^2)$
binary heap	$O(V)$	$O(V \cdot \lg(V))$	$O(E \cdot \lg(V))$	$O(E \cdot \lg(V))$
Fibonacci heap	$O(V)$	$O(V \cdot \lg(V))$	$O(E)$	$O(V \cdot \lg(V) + E)$

The  $O(E \cdot \lg(V))$  time for binary heaps assumes that  $O(V) \leq O(E)$ , which is true for connected graphs and even for most unconnected graphs. Note that a final distance of  $\infty$  indicates a vertex that is in a different connected component from the source  $s$ .

---

## Breadth First Search

In the simple case where all weights = 1, we can expand a "frontier" outward from the source, level by level. We can color the nodes according to the following scheme:

- white = undiscovered
- gray = frontier node = discovered node whose edges have not been processed
- black = fully processed = discovered node directly connected only to other discovered nodes

A queue (regular queue, not priority queue) can be used to manage order in which nodes are processed.

```
BFS(G,s)
  {Initialization}
  for v in vertices(G) do
    color[v]  white  {All nodes originally undiscovered}
    d[v]
    parent[v]  nil
  color[s]  gray
  d[s]  0
  Q  Enq(s, Empty-Queue)  {Invariant: Q contains only gray nodes.}
  while not Empty-Queue?(Q) do
    f  Deq(Q)  {Next frontier node to process.}
    for g in Adj[f] do
      if color[g] = white then
        color[g] = gray
        d[g] = d[f] + 1
        parent[g] = f
        Enq(g, Q)
    color[f] = black  {Color node black when completely processed.}
```

*Analysis:*

- Each vertex enqueued and dequeued once at  $O(1)$  time per operation:  $O(V)$
- Each edge scanned once:  $O(E)$
- Total =  $O(V + E)$  (=  $O(E)$  for a connected graph)

