

Shortest Paths Algorithms

1 The Problem

We are given a *network*: a directed graph $G = \langle V, E \rangle$ on n nodes and m edges, and a cost function $c : E \rightarrow \mathbb{R}_0^+$. The *cost* of a path $P = x_1, x_2, \dots, x_k$ is

$$c(P) = \sum_i c((x_i, x_{i+1})).$$

We will refer to the number of edges on a path as its *length*. The *distance* from x to y is

$$\delta(x) = \begin{cases} \min(c(P) \mid P \text{ path } x \rightarrow y) & \text{if such a path exists,} \\ \infty & \text{otherwise.} \end{cases}$$

These paths should be called minimum cost paths, but are usually referred to as shortest paths.

There are two basic variants:

- Single-Source: The source vertex is fixed, we want to compute $\delta(s, t)$ for some/all targets t .
- All-Pairs: We have to compute $\delta(x, y)$ for all $x, y \in V$.

Note that if $P = s = x_0, x_{1,2}, \dots, x_k = t$ is a minimum cost path from s to t then its initial segments are by necessity minimum cost paths from s to x_i . This affords a simple method to store minimum cost paths: for each vertex x record its immediate predecessor $\pi(x)$ on a such a path.

1.1 Prototype Algorithm: Relaxation

Consider the single-source problem. We maintain an array $\text{dist}(x)$ of (over-)estimates for the true distances $\delta(x)$ from the fixed source. The $\text{dist}(x)$ values always are the costs of a real path s to x , though initially not necessarily the minimum cost one (or they are ∞).

Initialize:

$$\text{dist}(x) = \begin{cases} 0 & \text{if } s = x, \\ \infty & \text{otherwise.} \end{cases}$$

Edge (u, v) *requires attention* iff $\text{dist}(v) > \text{dist}(u) + c(u, v)$. We correct the overestimate by *relaxing* the edge: $\text{dist}(v) = \text{dist}(u) + c(u, v)$.

```
initialize dist[];
```

```
while( some edge (x,y) requires attention )
    relax (x,y);
```

Note that the order in which edges are chosen for relaxation is left entirely open.

Lemma 1.1 *The prototype algorithm for single-source shortest path always terminates. Upon completion, $\text{dist}(x) = \delta(x)$.*

Proof. First note that $\text{dist}(x)$ is always the cost of a real path s to x . Hence the value of $\text{dist}(x)$ can decrease only finitely often: our costs are non-negative. Thus, an edge can require attention only finitely often and every edge ultimately ceases to require attention. In particular, the algorithm terminates.

Now suppose for the sake of a contradiction that upon completion for some path

$$P = s, x_1, \dots, u, v$$

we have $\delta(v) = c(P) < \text{dist}(v)$. By choosing P to be of minimal length (not cost!) we may assume $\text{dist}(u) = \delta(u)$. But after $\text{dist}(u)$ is set to $\delta(u)$, the edge (u, v) requires attention: $\text{dist}(v) > \delta(v) = \delta(u) + c(u, v) = \text{dist}(u) + c(u, v)$. When (u, v) is relaxed, $\text{dist}(v) = \delta(v)$, contradiction. □

2 Easy Special Cases

When the edge costs are uniform, or the graph is acyclic, one can solve the single-source problem in linear time.

2.1 Unit Costs

In the special case where all edges have cost 1 the problem can be solved by Breadth-First-Search (BFS). Here is the algorithm written up in terms of relaxation.

```

initialize dist[];
enqueue( Q, s );

while( ! empty(Q) )
    x = dequeue(Q);
    forall (x,y) in E do
        if( (x,y) requires attention )
            relax edge, enqueue( Q, y );

```

BFS explores the graph level by level in order of increasing distance from s . The crucial observation for correctness is the following. Let $s = v_1, v_2, \dots, v_n$ be the sequence in which nodes enter the queue. Then BFS explores the graph level by level in order of increasing distance from s :

- The sequence $\delta(v_i)$ is non-decreasing.

To see this, note that for any snapshot v_l, v_{l+1}, \dots, v_r during the execution of BFS $\delta(v_r) \leq \delta(v_l) + 1$. As a result, $\delta(v_i) = \text{dist}(v_i)$ when v_i enters the queue.

Clearly the algorithm is $O(n + m)$, linear in the size of G .

2.2 Acyclic Graphs

Another special case arises when the underlying graph G is acyclic. In this case we can perform a topological sort, and then traverse the vertices in topological order.

```

initialize dist[];

```

```

foreach x in V, in topological order do
  forall (x,y) in E do
    if( (x,y) requires attention )
      relax edge;

```

Correctness follows by induction on the length of a minimum cost path.

The algorithm is $O(n + m)$.

3 Dijkstra's Algorithm

With arbitrary non-negative edge costs and in the presence of cycles the simple algorithms fail, but perhaps one can salvage the approach: explore vertices in order of cost, rather than proximity to source. By choosing the edges for relaxation carefully we do not have to relax the same edge twice.

```

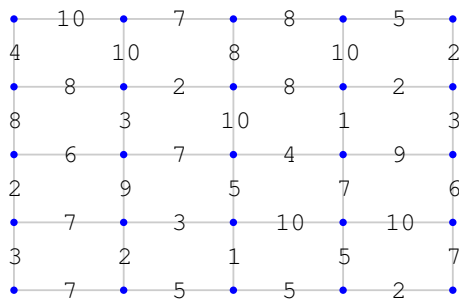
initialize dist[];
insert V into Q;

while( Q not empty )
  x = delete_min( Q );
  forall (x,y) in E do
    if( (x,y) requires attention )
      relax edge

```

Here Q is a priority queue whose priorities are given by dist . Every time an edge is relaxed, the target node in the queue must be promoted accordingly.

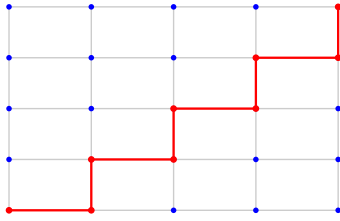
Example 3.1 Here is a 5×5 grid with random integer edge costs in $[10]$. The edges are oriented up/right. The source vertex is the lower left corner.



The distance matrix:

	17	24	24	32	26
	13	14	16	22	24
	5	11	17	21	30
	3	9	12	22	26
	0	7	12	17	19

The shortest path to the top-right corner is as follows.



As before, the $\text{dist}(x)$ values always are the costs of a real path s to x , though initially not necessarily the shortest one.

Lemma 3.1 Upon completion of Dijkstra's algorithm we have $\text{dist}(x) = \delta(x)$ for all x .

Proof.

For correctness one can establish the following loop invariant:

- For $x \in V - Q$: $\text{dist}(x) = \delta(x)$.
- For $x \in Q$, $\text{dist}(x) < \infty$: $\text{dist}(x) =$ minimum cost path in $V - Q$ plus one edge .

□

If we use a standard binary min-heap to hold the vertices relaxing a single edge is $O(\log n)$: the cost of promotion (we traverse the adjacency lists, so the access cost is constant). Also note that one could keep only the elements in the “fringe” in the queue (vertex encountered, but not yet processed).

Lemma 3.2 Dijkstra's shortest path algorithm computes all shortest path from a fixed source s in time $O((n + m) \log n)$.

This is fine for sparse graphs (when $m \ll n^2$), but uncomfortable for dense graphs. There are n `DeleteMin` operations, but possibly many more `Promote` operations during the relaxation steps. To exploit this fact one can use a priority queue where `Promote` is $o(\log n)$ (Fibonacci heaps, amortized running time). Yields $O(n \log n + m)$.

4 Floyd-Warshall Algorithm

To tackle the all-pairs problem we could run Dijkstra n times: $O(n(n + m) \log n) = O(n^3 \log n)$.

Alternatively, we can use dynamic programming by restricting intermediate vertices on a path. Assume $V = [n]$ and set

$$d(i, j, k) = \text{length shortest path } i \rightarrow j \text{ using only } \leq k$$

where $1 \leq i, j \leq n$ and $0 \leq k \leq n$. Note that the endpoints of the path are excluded from the restriction.

We want $k = n$: $d(i, j, n) = \delta(i, j)$. Initializing $k = 0$ is easy:

$$d(i, j, 0) = \begin{cases} 0 & \text{if } i = j, \\ c(i, j) & \text{if } (i, j) \in E, \\ \infty & \text{otherwise.} \end{cases}$$

Step:

$$d(i, j, k) = \min(d(i, j, k - 1), d(i, k, k - 1) + d(k, j, k - 1))$$

```

for( i=1; i<=n; i++ )
  for( j=1; i<=n; j++ )
    initialize dist[i,j,0];

for( k=1; i<=n; k++ )
  for( i=1; i<=n; i++ )
    for( j=1; i<=n; j++ )
      dist[i,j,k] = min( dist[i,j,k-1], dist[i,k,k-1] + dist[k,j,k-1] );

```

Upon completion, $dist(i, j, n) = \delta(i, j)$.

Lemma 4.1 *Floyd-Warshall runs in time $\Theta(n^3)$ and space $\Theta(n^2)$.*

5 General Costs and Bellman-Ford

Note that the shortest path problem may still be meaningful in the presence of negative edges costs: we just cannot have negative cost cycles (more precisely, negative cost cycles that are reachable from the source vertex).

There really are two issues:

- Check if there are negative cost cycles.
- Find shortest paths if there aren't any.

Negative costs can arise very naturally: currency exchange problem.

Here is a relaxation based method that can tackle both problems at once: the algorithm determines whether negative cost cycles exist, and if there are none it will have correctly computed the distances from s .

```

loop n-1 do
  foreach edge (x,y) do
    relax (x,y);

```

The running time is clearly $O(nm)$, but correctness is far from obvious.

Lemma 5.1 *Upon completion, there are no negative cost cycles if no edge requires attention. In this case, the algorithm has correctly computed all distances.*

Proof.

Consider a cycle $x_0, x_1, \dots, x_{k-1}, x_0$. If none of the edges (x_i, x_{i+1}) requires attention a simple induction shows that

$$\text{dist}(x_0) + \sum_{i < j} c((x_i, x_{i+1})) \geq \text{dist}(x_j).$$

Hence the cost of the whole cycle is non-negative. In other words, a negative cost cycles will always lead to edges requiring attention.

So suppose there are no negative cost cycles and consider a shortest path from P from s to t . Since the path is simple, it has length at most $n - 1$. But then by induction all the distances for vertices on the initial segments of P will have proper distance values.

□