

Shortest Paths

Adnan Aziz

UT Austin

Based on CLR, Ch 24 and 25.

Motivating example — want to drive from Austin to Berkeley. Given a road map, with distances between each pair of adjacent intersections, how to find the shortest route?

- Enumerate all paths from Austin to Berkeley
 - cycles cause problems; huge number of paths; many should not be considered (e.g., going via Cambridge)

In this unit, we'll address such problems using a graph representation, specifically all complexity results are relative to an **adjacency list representation**.

Given a weighted directed graph $G = (V, E)$, with weight function $w : E \mapsto (-\infty, \infty)$.

- Define weight of a path $p = \langle v_0, v_1, \dots, v_k \rangle$, its weight is defined to be sum of the weights of the edges on the path, i.e.,

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

- Define shortest path weight from u to v by

$$\begin{aligned} \delta(u, v) &= \min\{w(p) : u \stackrel{p}{\rightarrow} v\} \text{ if there is a path from } u \text{ to } v \\ &= \infty \text{ otherwise} \end{aligned}$$

A shortest path from u to v is any path p from u to v such that $w(p) = \delta(u, v)$.

Modeling issues: for travel example, vertices are intersections, edges are road segments, weights are road distances.

- far more general applications — time/cost/penalty lossage, etc. —anything that accumulates linearly along a path

Variants:

1. Single source shortest path (focus of Ch 24)
2. Single destination shortest path
3. Single pair shortest path
4. All pairs shortest path (focus of Ch 25)

Negative-weight edges

In some situations, have a graph where edge weights are negative. If there are no “negative weight” cycles in the graph, then shortest path weights are well defined (even if negative).

- if there is a negative weight cycle reachable from s , can't have shortest paths from s to any vertex on cycle
 - define shortest path weight to be $-\infty$

Conventions about ∞ —for real a , define $a + \infty$ to be ∞ , $a + (-\infty)$ to be $-\infty$.

Representing shortest paths

Usually want more than the shortest paths weights

- want the paths themselves

Will use a scheme similar to that used in BFS — given source vertex s in a graph $G = (V, E)$, will construct a “shortest path tree”

- for each v , maintain a field $\pi[v]$ (“predecessor”)
 - will set in such a way that chain of predecessors starting at v runs backwards along a shortest path to s
 - conceptually π gives us a tree rooted at s . This tree is a subgraph of G , has property that all vertices reachable from s are in the tree, and for all vertices v in the tree the unique simple path from s to v is a shortest path from s to v in G

Key ideas

Lemma: Subpaths of shortest paths are shortest paths

Computing shortest paths from s : maintain $d[v]$ estimates (upper bounds)

INIT(G, s)

```

for each v in V[G]
  do d[v] <- infinity
     p[v] <- NIL
d[s] <- 0

```

Relaxation step: given an edge (u, v) , if $d[v] > d[u] + w(u, v)$ then $d[v] = d[u] + w(u, v)$ (also update $\pi[v]$ to u).

RELAX(u, v, w)

```

if d[v] > d[u] + w(u, v)
  then d[v] <- d[u] + w(u, v)
     p[v] <- u

```

Properties of shortest paths and relaxation

Triangle-inequality for all edges (u, v) , we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$

Upper-bound Always: $d[v] \geq \delta(s, v)$, and once $d[v] = \delta(s, v)$ it never changes

No-path If there is no path from s to v , then $d[v] = \infty$ always

Convergence If $s \rightsquigarrow u \rightarrow v$ is a shortest path and $d[u] = \delta(s, u)$ prior to relaxing (u, v) , then $d[v] = \delta(s, v)$ afterwards

Path relaxation if $p = \langle v_0, \dots, v_k \rangle$ is a shortest path from $v_0 = s$ to v_k and the edges in p are relaxed in order $(v_0, v_1), \dots, (v_{k-1}, v_k)$ then $d[v_k] = \delta(s, v_k)$, regardless of what other relaxations are performed, even if they are interleaved with those for the edges in p .

Bellman-Ford

Works with negative weight edges.

`BF(G, s, w)`

`INIT(G, s)`

 for `i` <- to `|V[G]| - 1`

 do for each edge `e` in `E[G]`

 do `RELAX(u, v, w)`

 for each edge `e` in `E[G]`

 do if `d[v] > d[u] + w(u, v)`

 then return `FALSE`

 return `TRUE`

Time complexity— $O(V \cdot E)$.

Correctness

First assume no negative weight cycles.

Consider a shortest path $p = \langle v_0, \dots, v_k \rangle$ from $s = v_0$ to $v = v_k$. Path has at most $|V| - 1$ edges so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations in BF relaxes all edges; in particular edge (v_{i-1}, v_i) is relaxed in i -th iteration.

Correctness follows from path-relaxation property and no-path property.

Correctness in presence of negative cycles

Suppose there was a negative weight cycle reachable from source; let it be $\langle v_0, \dots, v_k \rangle$, with $v_0 = v_k$. Let BF return true. Then for $i = 1, 2, \dots, k$ we have

$$d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$$

Adding all the inequalities, we get

$$\begin{aligned} \sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i) \end{aligned}$$

However, since $v_0 = v_k$, each vertex in c appears exactly once, in each sum $\sum_{i=1}^k d[v_i]$ and $\sum_{i=1}^k d[v_{i-1}]$, and so $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$. Furthermore, each $d[v_i]$ is finite, so $0 \leq \sum_{i=1}^k w(v_{i-1}, v_i)$, contradicting the assumption that the cycle had negative weight.

Shortest paths in acyclic graphs

Can compute in $\Theta(V + E)$. Key idea—process in topological order.

```

DAG_SHORTEST_PATH( $G, w, s$ )
    topologically sort vertices in  $G$ 
    INIT( $G, s$ )
    for each vertex  $u$  in top sorted order
        do for each vertex  $v$  in Adj[ $u$ ]
            do RELAX( $u, v, w$ )

```

Time complexity— $O(V + E)$ for the topological sort, $O(V)$ for initialization, $\Theta(V)$ iterations of outer loop, $\Theta(E)$ iterations of inner loop; $\Theta(V + E)$ in all.

Why does it work? Because if $\langle v_0, \dots, v_k \rangle$ is a shortest path, we'll relax the edges in that order (because of the initial top sort).

Can use to efficiently compute longest paths in a DAG too (for general graphs the longest simple path problem is very hard).

Dijkstra's Algorithm

Solves single-source shortest paths on weighted directed graph, when edge weights are nonnegative.

Advantage of DA over BF—faster runtime.

Idea—maintain set of vertices S for which final shortest path weights from s are known. Select vertex $u \in V - S$ with minimum shortest path estimate; add u to S , and relax its outgoing edges.

```

DIJKSTRA( $G, w, s$ )
    INIT( $G, s$ )
     $S \leftarrow \{\}$ 
     $Q \leftarrow V[G]$  // min-heap of vertices, keys are d values
    while  $Q \neq \{\}$ 
        do  $u \leftarrow \text{EXTRACT\_MIN}( Q )$ 

```

```

S ← S + {u}
for each vertex v in Adj[u]
  do RELAX(u, v, w)

```

Always choose the “closest” vertex in $V - S$: example of a greedy strategy.

Correctness

Suffices to show that $d[u] = \delta(s, u)$ when u is added to S . (Note that every vertex eventually gets added to S .) Once $d[u] = \delta(s, u)$, it never changes (Upper-bound property).

Suppose for contradiction that we add vertices to S that do not satisfy $d[u] = \delta(s, u)$ at the time of addition. Let u be first such vertex added. Look at the shortest path from s to u , and consider the predecessor x of the first vertex y outside of S on this path.

Observe: $d[y] = \delta(s, y)$ when u is added to S (because u was first vertex failing $d[u] = \delta(s, u)$, we must have $d[x] = \delta(s, x)$; relaxing (x, y) guarantees observation).

From the observation it follows that $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$. But since u and y were in $V - S$ when u was chosen, it must be that $d[u] \leq d[y]$.

So $d[y] = \delta(s, y) = \delta(s, u) = d[u]$. Consequently, $d[u] = \delta(s, u)$, contradicting choice of u .

Performance

Depends on implementation of heap.

Straightforward array, with scans to select minimum element: $O(V^2 + E) = O(V^2)$.

For sparse graphs, using binary heap leads to $\log V$ update and lookup times. There are V initial inserts, then V deletes and E updates. Yields $O(V + E) \log V$ runtime, which is just $E \log V$ if $E \geq V$ (which is usually the case).

For a graph with $\Theta(V^2)$ edges, this is not competitive with the array implementation of the heap, but for “sparse” graphs it’s much faster.

Factoid: can use Fibonacci heaps to achieve a runtime of $O(V \log V + E)$.

Proofs of properties

All Pairs Shortest Paths

Based on CLR 25.

For the single source shortest path problem, we have Dijkstra's algorithm, which runs in time $O(E \log V)$.

⇒ for the all pairs shortest path problem, we could simply run Dijkstra's algorithm for each vertex — overall time complexity of $O(VE \log V)$.

On *dense* graphs (graphs for which the number of edges is close to V^2), we can do somewhat better — $O(V^3)$.

- time — not a big difference
- implementation — much simpler

Will use adjacency matrix representations

- $w_{ij} = 0$ if $i = j$
- $w_{ij} = w(i, j)$ if $i \neq j$ and $(i, j) \in E$
- $w_{ij} = \infty$ if $i \neq j$ and $(i, j) \notin E$

Assume no negative edge weights.

Want to output a matrix D which is $V \times V$; entry d_{ij} is weight of shortest path from i to j .

- how to represent shortest paths?
 - “predecessor matrix” Π — the entry π_{ij} is NIL if $i = j$ or \nexists path from i to j , and π_{ij} is a predecessor of j on a shortest path from i

Observe — the row i of Π yields a shortest path tree rooted at i

```

print_all_pairs_shortest_path(P,i,j)
if i=j
  then print i
else if P(i,j) = NIL
  then print "no path from" i "to" j "exists"
  else print_all_pairs_shortest_path(P,i,P(i,j))
      print j

```

Algorithms for the APSPP

The algorithm in §25.1 is not very good

- idea: find shortest path containing k or fewer edges (recursive; when $k = V$, done

More intuitive, but slower and more complicated

Floyd-Warshall algorithm §25.2 — $O(V^3)$, very simple

Assume vertices are $\{1, 2, \dots, n\}$.

- Idea — consider “intermediate” vertices on path
 - if $p = \langle v_1, v_2, \dots, v_l \rangle$ is a simple path, then the intermediate vertices those in set $\{v_2, v_3, \dots, v_{l-1}\}$

For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are in the set $\{1, 2, \dots, k\}$.

- let p be a minimum path amongst these

KEY: Floyd-Warshall algorithm is based on exploiting the relationship between p and the shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \dots, k - 1\}$.

Recursive formulation of shortest path estimates:

- $d_{ij}^{(k)}$ is weight of a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \dots, k\}$
 - when $k = 0$, there are no intermediate vertices, so path consists of at most one edge, i.e., $d_{ij}^{(0)}$ is w_{ij}

$$d_{ij}^{(k)} = \min \left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) \text{ if } k \geq 1$$

The matrix $D^{(n)}$ gives the final answer, i.e., $d_{ij}^{(n)} = \delta(i, j)$.

```
floyd_warshall(W)
```

```
  n <- rows[W]
```

```
  D(0) <- W
```

```
  for k <- 1 to n
```

```
    do for i <- 1 to n
```

```
      do for j <- 1 to n
```

```
        d^k(i,j) <- min( d^(k-1)(i,j), d^(k-1)(i,k) + d^(k-1)(k,j) )
```

```
  return D(n)
```

Time complexity — $O(n^3)$

- very tight – no elaborate data structures

How about computing Π ?

- Compute $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$, where $\Pi^{(n)} = \Pi$ and $\pi_{ij}^{(k)}$ is predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in $\{1, 2, \dots, k\}$.

$$\begin{aligned}
\pi_{ij}^{(0)} &= \text{NIL if } i = j \text{ or } w_{ij} = \infty \\
&= i \text{ if } i \neq j \text{ and } w_{ij} < \infty \\
\pi_{ij}^{(k)} &= \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\
&= \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}
\end{aligned}$$

Routing in the Internet

Couple of possibilities—distance vector and link-state.

Each router knows the address of each neighbor and the cost of reaching each neighbor.

Routing algorithms allow routers to find global routing information by exchanging information to neighbors only (such algorithms are called “distributed”).

In DVR, every router knows identity of every other router (but not necc the shortest path to it).

Each router maintains a distance vector (a list of $(dest, cost)$ tuples, one per dest).

Initializes cost of nonneighbors to ∞ . Then it periodically exchanges its DV with neighbors, updates estimates.

Although DVR looks different, it’s really just BF—edges are relaxed enough times to satisfy the path-relaxation property.

One problem with DVR is that is fails when a link goes down (count to infinity problem). Can solve by keeping the path along with the cost of the path (this is called the path vector approach, and is used in the BGP protocol). Requires lots of extra storage, hence some intermediate schemes are also used—split horizon.