Paths in Graphs: Most Direct Route

Length

Length of the path, $L(P)$, is the number of edges in the path.

Distance

The distance between two vertices is the length of the shortest path between them.

Distance layer

A directed graph with nodes at different levels. Number of levels between the root and the node is the distance.

Breadth-first search

```
BFS(G, S):
for all u in V:
    dist[u] <- infinity
dist[S] <- 0
Q <- {S} {queue containing just S, the root}
while Q is not empty:
    u <- Dequeue(Q):
    for all (u, v) in E:
        if dist[v] = infinity
            Enqueue(Q, v)
            dist[v] = dist[u] + 1
```

Lemma

The running time of breadth-first search is $O(|V| + |E|)$.

Proof:

- Each vertex is enqueued at most once
- Each edge is examined either once (for directed graphs) or twice (for undirected graphs)

Reachable

Node u is reachable from node S if there is a path from S to u.

Lemma

Reachable nodes are discovered at some point, so they get a finite distance estimate from the source. Unreachable are not discovered at any point, and the distance to them stays infinite.

Proof:

- Let u be a reachable undiscovered closest node to S.
- $S - v_1 - \ldots - v_k - u$ is the shortest path
- Then, u will be discovered while processing v_k (recall BFS algorithm)
- Let u be the first unreachable discovered node
- u must be discovered while processing some v
- It implies u is reachable from v and since v is reachable from S, u must be also reachable from S
- If we force u to be unreachable from S, then it must not be discovered

Order Lemma

By the time node u at distance d from S is dequeued, all the nodes at distance at most d have already been discovered (enqueued)

Proof:

Suppose u has distance d to S, v has distance d' to S. Suppose $d' \leq d$, v has distance to S at most d but hasn't been enqueued. Let u' be the last node of the shortest path to u, and v' be the last node of the shortest path to v.

$$u' \to u, v' \to v$$

According to `BFS`, u' has a distance to S at least $d-1$. Otherwise, there exists another node which is in the shortest path.

Consider the first time the order was broken: $d' \leq d \implies d'-1 \leq d-1$. So, v' was discovered before u' was dequeued because Order Lemma still holds at $d-1$. Since v' is discovered before u is discovered in the process of dequeuing u', v' will be dequeued before u is dequeued. As v is enqueued (discovered) when v' is dequeued, it contradicts the assumption that v is not discovered by the time u is dequeued.

Lemma (Correctness)

When node u is discovered (enqueued), `dist[u]` is assigned exactly $d(S,u)$.

Proof:

- Use mathematical induction
- Base: When S is discovered, `dist[s]` is assigned $0 = d(S,S)$.
- Inductive step: suppose proved for all nodes at distance $\leq k$ from S. We try to prove for nodes at distance $k+1$.
- Take a node v at distance $k+1$ from S
- v was discovered while processing u. The distance between S and v is at most $d(S,u)+1$ (It can be the case that $d(S,v) = d(S,u)$)
- $d(S,v) \leq d(S,u)+1 \implies d(S,u) \geq k$
- v is discovered after u is dequeued, so $d(S,u) < d(S,v) = k+1$
- So, $d(S,u) = k$, and `dist[v] <- dist[u] + 1 = k + 1`

Lemma (Queue property)

At any moment, if the first node in the queue is at distance d from S, then all the nodes in the queue are either at distance d from S or at distance $d+1$ from S. All nodes in the queue at distance d go before (if any) all the nodes at distance $d+1$.

Proof:

- All nodes at distance d were enqueued before first such node is dequeued, so they go before nodes at distance $d+1$ (Order Lemma).
- Nodes at distance $d-1$ were enqueued before nodes at d, so they are not in the queue anymore
- Nodes at distance $> d+1$ will be discovered when all d are gone

Lemma

Shortest-path tree is indeed a tree, i.e. it doesn't contain cycles (it is a connected component by construction).

Proof:

Suppose there is a cycle in the shortest-path tree $A \to B \to C \to D \to E \to A$

The distance to S should be decreasing after going by edge (Suppose $D(S, A) = d$ and there is an exit in the cycle connected to S)

However, we observe that
$D(S, A) = d, D(S, B) = d - 1, D(S, C) = d - 2, D(S, D) = d - 3, D(S, E) = d - 4, D(S, A) = d - 5$
which is a contradiction

```
BFS(G, S):
for all u in V:
    dist[u] <- infinity, prev[u] <- nil
dist[S] <- 0
Q <- {S} {queue containing just S, the root}
while Q is not empty:
    u <- Dequeue(Q):
    for all (u, v) in E:
        if dist[v] = infinity
            Enqueue(Q, v)
            dist[v] = dist[u] + 1, prev[v] <- u
```

```
ReconstructPath(S, u, prev):
result <- empty
while u != S:
    result.append(u)
    u <- prev[u]
reeturn Reverse(result)
```

Paths in Graphs: Fastest Route

Lemma

Any subpath of an optimal path is also optimal.

Proof:

Consider an optimal path from S to t and two vertices u and v on this path. If there were a shorter path from u to v, we would get a shorter path from S to t.

Corollary

If $S \to \ldots \to u \to t$ is a shortest path from S to t, then $d(S, t) = d(S, u) + w(u, t)$

Edge relaxation

- `dist[v]` will be an upper bound on the actual distance from S to v
- The edge relaxation procedure for an edge (u, v) just checks whether going from S to v through u improves the current value of `dist[v]`

```
Relax((u, v) in E):
if dist[v] > dist[u] + w(u, v):
    dist[v] <- dist[u] + w(u, v)
    prev[v] <- u
```

```
Naive(G, S):
for all u in V:
    dist[u] <- infinity
    prev[u] <- nil
dist[S] <- 0
do:
    relax all the edges
while at least one dist changes
```

Lemma

After the call to `Naive` algorithm, all the distances are set correctly.

Proof:

Assume, for the sake of contradiction, that no edge can be relaxed and there is a vertex v such that $\text{dist}[v] > d(S, v)$. (i.e. $\text{dist}[v]$ in the algorithm is greater than the actual shortest distance from S to v)

Consider a shortest path from S to v and let u be the first vertex on this path with the same property. Let p be the vertex right before u.

$$S \to \ldots \to p \to u \to \ldots \to v$$

Then, $d(S, p) = \text{dist}[p]$ and hence $d(S, u) = d(S, p) + w(p, u) = \text{dist}[p] + w(p, u)$. Note that p is located before u in the path. So, the lemma still applies for p. As the lemma does not apply starting from u, $\text{dist}[u] > d(S, u) = \text{dist}[p] + w(p, u)$. It implies that edge (p, u) can be relaxed - which is a contradiction.

```
Dijkstra(G, S):
for all u in V:
    dist[u] <- infinity
    prev[u] <- nil
dist[S] <- 0
H <- MakeQueue(V) {dist-values are keys}
while H is not empty:
    u <- ExtractMin(H)
    for all (u, v) in E:
        if dist[v] > dist[u] + w(u, v):
            dist[v] <- dist[u] + w(u, v)
            prev[v] <- u
            ChangePriority(H, v, dist[v])
```

Lemma

When a node u is selected via `ExtractMin`, $\text{dist}[u] = d(S, u)$.

Proof:

Suppose the known region has nodes $\{S, A, B\}$ with all edges relaxed and C is the node outside the known region from `ExtractMin`. Assume $\text{dist}[C] > d(S, C) = \text{dist}[B] + w(B, C)$. It implies that the edge from B to C hasn't been relaxed when B comes from `ExtractMin`, which is impossible.

Total running time: $T(\text{MakeQueue}) = |V| \cdot T(\text{ExtractMin}) + |E| \cdot T(\text{ChangePriority})$

Implementation in binary heap: $O(|V| + |V| \log |V| + |E| \log |V|) = O((|V| + |E|) \log |V|)$

Paths in Graphs: Currency Exchange

Maximum product over paths

Input: Currency exchange graph with weighted directed edges e_i between some pairs of currencies with weights r_{e_i} corresponding to the exchange rate.

Output: Maximize $\prod_{j=1}^{k} r_{ej} = r_{e_1} r_{e_2} \cdots r_{e_k}$ over paths $(e_1, e_2, \ldots, e_k)$ from USD to RUR in the graph.

Equivalence problem

Minimize $\sum_{j=1}^{k} (-\log(r_{e_j}))$

However, Dijkstra's algorithm cannot be applied to negative edges.

```
BellmanFord(G, S):
{no negative weight cycles in G}
for all u in V:
    dist[u] <- infinity
    prev[u] <- nil
dist[S] <- 0
repeat |V| - 1 times:
    for all (u, v) in E:
        Relax(u, v)
```

Lemma

The running time of Bellman-Ford algorithm is $O(|V||E|)$.

Proof:

Initialize `dist` , $O(|V|)$

$|V| - 1$ iterations, each $O(|E|)$, in total $O(|V||E|)$

Lemma

After k iterations of relaxations, for any node u, $\text{dist}[u]$ is the smallest length of a path from S to u that contains at most k edges.

Proof:

- Use mathematical induction
- Base: After 0 iterations, all dist-values are ∞, but for $\text{dist}[S] = 0$, which is correct
- Induction: proved for k, prove for $k + 1$
- Before $k + 1$-th iteration, $\text{dist}[u]$ is the smallest length of a path from S to u containing at most k edges
- Each path from S to u goes through one of the incoming edges (v, u)
- Relaxing by (v, u) is comparing it with the smallest length of a path from S to u through v containing at most $k + 1$ edge

Corollary

In a graph without negative weight cycles, Bellman-Ford algorithm correctly find all distances from the starting node S.

Corollary

If there is no negative weight cycle reachable from S such that u is reachable from this negative weight cycle, Bellman-Ford algorithm correctly finds $\text{dist}[u] = d(S, u)$.

Lemma

A graph G contains a negative weight cycle if and only if $|V|$-th (additional) iteration of `BellmanFord(G, S)` updates some `dist`-value

Proof:

$\leftarrow$ If there are no negative cycles, then all shortest paths from S contain at most $|V| - 1$ edges, so no `dist`-value can be updated on $|V|$-th iteration.

$\rightarrow$ There's a negative weight cycle, say $a \rightarrow b \rightarrow c \rightarrow a$, but no relaxations on $|V|$-th iteration.

$$\begin{gathered}\text{dist}[b] \leq \text{dist}[a] + w(a,b)\\ \text{dist}[c] \leq \text{dist}[b] + w(b,c)\\ \text{dist}[a] \leq \text{dist}[c] + w(c,a)\end{gathered}$$

Then, $w(a,b) + w(b,c) + w(c,a) \geq 0$ which cannot be a negative weight cycle, and implies a contradiction.

Finding negative cycle

- Run $|V|$ iterations of Bellman-Ford algorithm, save node v relaxed on the last iteration
- v is reachable from a negative cycle
- Start from $x \leftarrow v$, follow the link $x \leftarrow \text{prev}[x]$ for $|V|$ times, will be definitely on the cycle
- Save $y \leftarrow x$ and go $x \leftarrow \text{prev}[x]$ until $x = y$ again

Lemma

It is possible to get any amount of currency u from currency S if and only if u is reachable from some node w for which $\text{dist}[w]$ decreased on iteration V of Bellman-Ford.

$\leftarrow$

- $\text{dist}[w]$ decreased on iteration $V \implies w$ is reachable from a negative weight cycle
- w is reachable $\implies u$ is also reachable $\implies$ infinite arbitrage

$\rightarrow$

- Let L be the length of the shortest path to u with at most $V - 1$ edges.
- After $V - 1$ iterations, $\text{dist}[u]$ is equal to L
- Infinite arbitrage to $u \implies$ there exists a path shorter than L
- Thus, $\text{dist}[u]$ will be decreased on some iteration $k \geq V$
- If edge (x, y) was not relaxed and $\text{dist}[x]$ did not decrease on i-th iteration, then edge (x, y) will not be relaxed on $i + 1$-st iteration
- Only nodes reachable from those relaxed on previous iterations can be relaxed
- $\text{dist}[u]$ is decreased on iteration $k \geq V \implies u$ is reachable from some nodes relaxed on V-th iteration

Detect infinite arbitrage

- Do $|V|$ iterations of Bellman-Ford, save all nodes relaxed on V-th iteration - set A
- Put all nodes from A in queue Q
- Do breadth-first search with queue Q and find all nodes reachable from A
- All those nodes and only those can have infinite arbitrage

Reconstruct infinite arbitrage

- During breadth-first search, remember the parent of each visited node
- Reconstruct the path to u from some node w relaxed on iteration V
- Go back from w to find negative cycle from which w is reachable
- Use this negative cycle to achieve infinite arbitrage from S to u