Single Source Shortest Paths

Alison Liu

1 Breadth-First Search and shortest paths

Given a directed weighted graph $G = (V, E, w)$, where every edge $(u, v) \in E$ has a weight $w(u, v)$. A *path* P is a sequence of vertices $v_0, v_1, v_2, \cdots, v_k$ such that for any $i \in [1, k]$, $(v_{i-1}, v_i) \in E$. The weight of a path P, denoted by $w(P)$, is defined by $\sum_{i=1}^{k} w(v_{i-1}, v_i)$. We denote the *shortest distance* between $u, v \in V$, $\delta(u, v)$ by $\min\{w(P) \mid P \text{ is a path from } u \text{ to } v\}$. The *single-source shortest paths* problem is to find the shortest distance from the source s to any vertex $v \in V$.

According to the different complexity of the input graphs, we introduce different algorithms for solving the single-source shortest paths problem.

The *breadth-first search (BFS)* algorithm is a graph exploration algorithm. In short, `BFS` keeps a queue (which initially contains the vertex where `BFS` starts at) of visited vertices. In each round, `BFS` dequeues a vertex from the queue and enqueues all its neighbors that are not discovered yet. See Algorithm 2

Algorithm 1 `Breadth-First Search`(s)

```
Mark s as discovered
Enqueue(Q, s)
while Q is not empty do
    u ← Dequeue(Q)
    for each neighbor v of u do
        if v is not-discovered then
            Mark v as discovered
            d[v] ← d[u] + 1
            π[v] ← u
            Enqueue(Q, v)
    Mark u as finished
```

By the aggregate method, the time complexity of `BFS` is $O(|V| + |E|)$, since every vertex is enqueued and dequeued once, and each edge is checked once.

Note that the variable $d[v]$ is the number of edges on the shortest path from s to v. That is, `BFS` finds the shortest distance from s to any vertex $v \in V$ if the graph is unweighted.

Tense edges and relaxation. We use $d[v]$ as the temporal estimation (which is initially set to be ∞) of the shortest distance from s to v. Throughout the single-source shortest paths algorithms we will introduce in this lecture, we keep updating the value of $d[v]$ when we find a shorter distance from s to v. An edge (u, v) is *tense* if $d[v] < d[u] + w(u, v)$. A tense edge is *relaxed* if we update the value of $d[v]$ by $d[v] + w(u, v)$. In other words, throughout the single-source shortest paths algorithms, we keep relaxing tense edges if there are any. The difference between the algorithms is that, according to different input graphs, we have different ordering of relaxation of the tense edges (mostly for the sake of correctness or time complexity).

Using the concept of tense edges and relaxation, we can rewrite the pseudo-code of `BFS` as follows.

Algorithm 2 `Breadth-First Search`(s)

```
Mark s as discovered
Enqueue(Q, s)
while Q is not empty do
    u ← Dequeue(Q)
    for all edges (u, v) do
        if (u, v) is tense then
            Relax(u, v)
            Enqueue(Q, v)
```

2 Shortest paths on a DAG

When there are different weights on the edges, the BFS algorithm may not return the correct shortest paths, since the visiting ordering of neighbors may not reflect the shortest distance properly.

We first try to find single-source shortest paths on a weighted DAG. The algorithm visits the vertices v in the topological order and relaxes every incoming edge (u, v) if it is tense.

Algorithm 3 `DAGSSSP`(s)

```
TOPOLOGICAL SORT the vertices
for each vertex v ≠ s taken in topologically sorted order do
    for all incoming edges (u, v) do
        if (u, v) is tense then
            RELAX (u, v)
```

The time complexity of `DAGSSSP` is $O(|V| + |E|)$.

Correctness. The `DAGSSSP` algorithm is actually a dynamic algorithm. Its correctness relies on the property of DAGs. Since we visit the vertices in the

topological order, when a vertex is visited, all vertices u that can reach it have to be visited already, and the shortest distance from s to u is fixed.

3 Shortest paths on a weighted graph with non-negative weights

The dynamic programming approach for DAGs fails when there are directed cycles in the graph. More specifically, we cannot bound the time complexity as linear as we did in the DAG case.

The `Dijkstra's` algorithm is for finding the shortest paths on non-negative weighted graphs with directed cycles. In a nutshell, the algorithm replaces the ordinary queue in the `BFS` algorithm with a priority queue. The vertices v are inserted into the priority queue with their $d[v]$ value. Every time when a vertex is dequeued, the algorithm always dequeues the one in the priority queue with the smallest $d[v]$ value.

Algorithm 4 `DIJKSTRA`(s)

For every vertex v, initialize $d[v] \leftarrow \infty$, $\pi[v] \leftarrow \phi$, and $f[v] \leftarrow false$
  ▷ $d[v]$ is the estimated shortest distance of $s - v$ path, and $f[v]$ indicates if the shortest distance of $s - v$ path is fixed
$d[s] \leftarrow 0$, $f[s] \leftarrow true$
while there is at least one vertex still unfinished **do**
  $u \leftarrow$ the vertex with $f[u] = false$ and the smallest $d[u]$
  $f[u] \leftarrow true$
  for all edges (u, v) **do**
    if (u, v) is tense **then**
      `RELAX`(u, v)

The time complexity is $O(|V|^2)$.[1]

Correctness. At any time, `Dijkstra's` algorithm maintains a subset V' of vertices, where every vertex has its shortest distance fixed. The algorithm picks vertices (which have their shortest distances not yet fixed) and adds the vertices into the subset. When picking the next vertex, the algorithm always finds the vertex u with the smallest $d[u]$ value. It is sufficient to prove that the shortest distance $\delta(s, u) = d[u]$. Consider any other path P from s to u. This path must contain an edge (x, y) where $x \in V'$ and $y \notin V'$ (note that y cannot be u). Denote the sub-path from y to u by P'. The path weight $w(P') \geq 0$ since there is no negative weight. Therefore, the weight of P is $w(x, y) + w(P') \geq d[y] \geq d[u]$. That is, the weight of any other path P from s to u is at least $d[u]$. It implies that $d[u]$ is the shortest distance from s to u.

[1] The time complexity can be improved into $O(|E| + |V| \log |V|)$ using Fibonacci heaps.

4 Shortest paths on a weighted graph without negative cycle

The last algorithm, `Bellman-Ford`, is to solve the single-source shortest paths problem on graphs that have negative weights (but without negative cycles). In short, the `Bellman-ford` algorithm keeps checking if there is any tense edge in the graph. If so, it checks *every* edge in the graph and relaxes it if needed.

Algorithm 5 `Bellman-Ford`(s)

```
while there is a tense edge do
    for every edge (u, v) do
        if (u, v) is tense then
            RELAX(u, v)
```

There is no shortest path with more than $|V| - 1$ edges. Therefore, the `Bellman-Ford` algorithm takes at most $|V| - 1$ rounds, and each round takes at most $|E|$ checking and relaxation. In total, the time complexity is $O(|V||E|)$.

Correctness. The correctness of `Bellman-Ford` algorithm relies on the following observation:

Observation 1. *For any vertex v, letP be the shortest path s to v, where u is the predecessor of v on the path. If at the moment when the edge (u, v) is checked, $d[u] = \delta(s, u)$, then after checking (u, v), $d[v] = \delta(s, v)$.*

By this observation, as long as `Bellman-Ford` relaxes the edges in order of each shortest path from s to any v, it correctly returns the shortest paths. Indeed, for any shortest path from s to v, $P = [s, v_1, v_2, \cdots, v_k = v]$, `Bellman-Ford` has at most k rounds. Moreover, the edge (s, v_1) is checked (and relaxed if needed) in the first round, the edge (v_1, v_2) is checked in the second round, etc. Therefore, `Bellman-Ford` algorithm correctly returns the shortest paths from the source vertex s.