Shortest Paths

Arnav Bansal

May 4, 2017

1 Introduction

A graph is a set of vertices connected by a set of edges. The shortest path problem is the problem of finding the shortest path between two vertices in a graph. This problem shows up often in competitive programming problems, and it is important to know the various algorithms that solve the problem, as well as when to use these algorithms.

2 Unweighted Graphs

In an unweighted graph, the length of a path between two vertices u and v equals the number of edges in the path. We can use breadth-first search (BFS) to find the shortest path between u and v.

BFS is an algorithm for traversing a graph. In BFS, we start at a source vertex s and traverse the graph "breadth-first". First, we visit the neighbors of the source vertex, then the neighbors of the neighbors, and so on.

BFS starts by inserting s into a queue. Then, while the queue is not empty, BFS pops vertex v from the queue, marks it as visited, and pushes all unvisited neighbors of v to the queue.

To find the shortest path between s and all other vertices, we simply maintain an array of distances, and each time we pop a vertex v from the queue and enqueue an unvisited neighbor u, we set the distance from s to u as 1 more than the distance from s to v, or $dist(u) = dist(v) + 1$. We increase the distance by 1 to each subsequent layer because we visit the vertices layer-by-layer in BFS. Thus, the distance to the neighbors of s is 1, to the neighbors of the neighbors is 2, and so on.

The time complexity of BFS is $O(V + E)$, since every vertex and edge is visited in the worst case.

Algorithm 1 Breath-first Search (BFS)

```
create empty queue q
push s to q
dist(v) ← 0 for all vertices v
visited(v) ← false for all vertices v
visited(s) ← true
while q is not empty do
    v ← vertex popped from q
    for all neighbors u of v do
        if not visited(u) then
            dist(u) ← dist(v) + 1
            push u to q
```

3 Weighted Graphs

In a weighted graph, the length of a path between two vertices u and v equals the sum of the weights of the edges on the path. There are many algorithms to solve the shortest path problem on weighted graphs, such as Dijkstra's, Bellman-Ford, Shortest Path Faster, and Floyd-Warshall (note that Floyd-Warshall solves the all-pairs shortest path problem).

The key idea in the first three algorithms is that the distance from a source vertex s to a vertex u can be minimized by following the shortest path to a neighboring vertex v of u and then the edge from v to u. That is, $dist(u) = min(dist(u), dist(v) + weight(v, u))$. The key idea in Floyd-Warshall is similar, except that v does not have to be a neighboring vertex of u and can be any intermediate vertex. Thus, for Floyd-Warshall, $dist(u) = min(dist(u), dist(v) + dist(v, u))$.

3.1 Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a source vertex s to all other vertices. Using Dijkstra's requires that there are no negative-weight edges in the graph. Dijkstra's is commonly used in solving shortest path problems in weighted graphs because it is one of the most efficient algorithms for doing so.

Like BFS, Dijkstra's starts by inserting s into a queue. Then, while the queue is not empty, Dijkstra's pops vertex v from the queue, marks it as visited, and pushes all unvisited neighbors of v to the queue.

However, what makes Dijkstra's different from BFS and correct on weighted graphs is its use of a min-priority queue instead of a regular queue. Thus, when a vertex v is popped from the queue, the distance to v is the shortest than to any other vertex in the queue. Hence, Dijkstra's tries to reduce the distance to an unvisited neighbor u of v by considering the path to u, through v. More formally, $dist(u) = min(dist(u), dist(v) + weight(v, u))$.

The time complexity of Dijkstra's algorithm is $O((V + E) \log V)$, since every

vertex and edge is visited in the worst case, with a $O(\log V)$ cost for each push and pop operation of the min-priority queue.

Algorithm 2 Dijkstra's Algorithm

```
create empty min-priority queue pq
push s to pq with key 0
dist(v) ← ∞ for all vertices v
dist(s) ← 0
visited(v) ← false for all vertices v
visited(s) ← true
while pq is not empty do
    v ← vertex popped from pq (vertex with minimum dist(v) in pq)
    if visited(v) then
        continue
    visited(v) = true
    for all neighbors u of v do
        if not visited(u) and dist(v) + weight(v, u) < dist(u) then
            dist(u) ← dist(v) + weight(v, u)
            push u to pq with key dist(u)
```

3.2 Bellman-Ford Algorithm

The Bellman-Ford algorithm find the shortest path from a source vertex s to all other vertices, where some edges may have negative-weight. However, Bellman-Ford requires that there are no cycles with negative length. To understand why there cannot be cycles with negative length, note that we can keep travelling the cycle and each time, our path length becomes shorter, so it can become infinitely short. Thus, a graph with a cycle with negative length has no true shortest path. Nonetheless, if the graph does contain a cycle with negative length, Bellman-Ford can detect this.

Like Dijkstra's, Bellman-Ford tries to reduce distances. Bellman-Ford does this by going through all edges of the graph. However, because negative-weight edges must be accounted for, the algorithm makes $v-1$ such rounds of distance reduction. The algorithm consists of $v-1$ rounds because a shortest path can contain at most v vertices and thus $v-1$ edges. Hence, after $v-1$ rounds, all distances will be reduced. Note that Bellman-Ford does not require the use of a priority queue.

Bellman-Ford can detect if a graph contains a cycle with negative length by running the algorithm for V rounds because if the last round further reduces any distances, the graph must contain a cyle with negative length.

The time complexity of the Bellman-Ford algorithm is $O(VE)$ because it consists of $V-1$ rounds and visits all E edges each round.

Algorithm 3 Bellman-Ford Algorithm

```
dist(v) ← ∞ for all vertices v
dist(s) ← 0
for i = 1, V − 1 do
    for all edges (v, u) do
        dist(u) ← min(dist(u), dist(v) + weight(v, u))
```

3.3 Shortest Path Faster Algorithm

Shortest Path Faster algorithm (SPFA) is a variant of the Bellman-Ford algorithm. It is usually more efficient than Bellman-Ford because it does not visit all edges each round. Instead, SPFA maintains a queue of vertices, but only of vertices that can reduce distances. SPFA starts by inserting a source vertex s into the queue. Then, while the queue is not empty, SPFA pops vertex v from the queue and tries to reduce the distance to a neighbor u of v. If and only if edge (v, u) reduces a distance and u hasn't already been considered/visited, u is added to the queue. Thus, SPFA ends up only considering vertices that can reduce distances and only considering these vertices once.

The time complexity of SPFA is still $O(VE)$ because in the worst case, $V-1$ rounds are needed. However, SPFA usually runs considerably faster.

Algorithm 4 Shortest Path Faster Algorithm

```
create empty queue q
add s to q
dist(v) ← ∞ for all vertices v
dist(s) ← 0
visited(v) ← false for all vertices v
while q is not empty do
    v ← vertex popped from q
    visited(v) ← true
    for all neighbors u of v do
        if dist(v) + weight(v, u) < dist(u) then
            dist(u) ← dist(v) + weight(v, u)
            if not visited(u) then
                add u to q
                visited(u) ← true
```

3.4 Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm solves the all-pairs shortest path problem as it finds the shortest path between any two vertices in a graph, not just from one source vertex s to all other vertices. Floyd-Warshall uses dynamic programming, and the graph may contain negative-weight edges. However, the graph cannot contain cycles with negative length. An advantage Floyd-Warshall has over

other solutions to the all-pairs shortest path problem is that the code for Floyd-Warshall is very simple and short!

Floyd-Warshall maintains a two-dimensional array of distances between any two vertices i and j in the graph. First, the matrix is initialized appropriately (see implementation details in pseudo-code). Then, the algorithm iterates over all vertices k such that k is a vertex on the path from i to j. Thus, the algorithm considers the distance between i and j, through k. More formally, $dist(i, j) = min(dist(i, j), dist(i, k) + dist(k, j))$

The time complexity of the Floyd-Warshall algorithm is $O(V^3)$ because all vertices k are considered for each pair of vertices (i, j). While the complexity is rather high, it is important to note Floyd-Warshall's advantage of its code being simple and short!.

Algorithm 5 Floyd-Warshall Algorithm

```
for i = 1, V do
    for j = 1, V do
        dist(i, j) ← ∞
        if weight(i, j) is not equal to 0 then
            dist(i, j) ← weight(i, j)
        if i is equal to j then
            dist(i, j) ← 0
for k = 1, V do
    for i = 1, V do
        for j = 1, V do
            dist(i, j) = min(dist(i, j), dist(i, k) + dist(k, j))
```