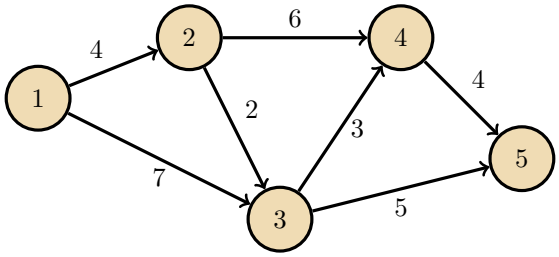
Shortest Paths

Kevin Geng

28 October 2016

1 Introduction

The shortest-paths problem comes up a lot in competitive programming. Given two nodes u and v in a directed graph, we want to find a path between u and v such that the sum of the edge weights of the path is minimized.



2 Floyd-Warshall

The Floyd-Warshall algorithm solves the multi-source shortest paths problem; that is, it solves the shortest path problem for every pair of vertices. We use a matrix of distances $dist$, which stores the shortest distance we have found so far for each pair of vertices.

Then, for every vertex k, and every pair of vertices $i \to j$, we try to see if $dist(i, j)$ can be improved by going through k. In other words, if the current best distance from $i \to k \to j$ is shorter than the current best distance for $i \to j$ (which could be ∞). If it is, we update $dist(i, j)$, which we want to be as short as possible. This is similar to the *relax* operation that we will use for Dijkstra's algorithm, though it may not necessarily involve edges in the original graph.

Algorithm 1 Floyd-Warshall

```
dist(i, j) ← ∞ for vertices i, j
for all vertices i do
    dist(i, i) ← 0
for all edges (u, v) do
    dist(u, v) ← weight(u, v)
for all vertices k do
    for all vertices i do
        for all vertices j do
            if dist(i, j) > dist(i, k) + dist(k, j) then
                dist(i, j) ← dist(i, k) + dist(k, j)
```

One way to see why this works is to consider two vertices u and v for which we already know the shortest path from $u \to v$. Once we consider each of the vertices in that shortest path, $dist(u, v)$ will be the length of the shortest path.

3 Dijkstra's algorithm

Those of you who have already taken AP Computer Science should already be familiar with Dijkstra's algorithm, so we won't spend too much time explaining the algorithm itself.

Dijkstra's algorithm calculates the shortest path from a source vertex u to every other node in the graph. If we're only interested in the shortest path from $u \to v$, we can stop once we remove v from the priority queue.

But why does it work? Every time we remove a vertex u from pq, $dist(u)$ is monotonically increasing. This is because the priority queue always gives the minimum element, and for any u removed from the priority queue with neighbor v added to it, $dist(v) < dist(u)$.

Algorithm 2 Dijkstra's algorithm

```
for all vertices v do
    dist(v) ← ∞
    prev(v) ← −1
    visited(v) ← false
dist(src) ← 0
pq ← priority queue
add src to pq with key 0
while pq is not empty do
    u ← u in pq with minimum dist(u)
    if visited(v) then                          ▷ only remove each node once
        continue
    visited(v) ← true
    for all neighbors v of u do                 ▷ relax edges
        alt ← dist(u) + weight(u, v)
        if not visited(v) and alt < dist(v) then
            dist(v) ← alt
            prev(v) ← u
            add v to pq with key dist(v)        ▷ add instead of update-key
```

An important implementation note: C++ implements a **max heap** rather than a min heap. You can get around this by negating the values of *dist*, or by using `std::greater` as the comparator.

3.1 Complexity

Suppose edge e connects vertices u and v. If $dist(v) > dist(u) + weight(e)$, then we define the process of *relaxing* an edge e as $dist(v) \leftarrow dist(u) + weight(e)$.

In the worst case, the algorithm will check every edge, and every edge will be relaxed. Each relaxation requires a priority queue insertion operation of complexity $O(\log V)$. Therefore, the complexity of Djikstra's Algorithm is $O(E \log V)$.

3.2 Update-key

When relaxing an edge $u \to v$, we update the value of $dist(v)$. However, if v is already in the priority queue, this might cause problems: most priority queues do not implement the *update-key* operation in $O(\log n)$ time.[1] Instead, we can simply insert the same vertex into the priority queue with an updated key. The vertex with the lowest key will be the first to leave the priority queue. Then, if we mark that vertex as visited, we can avoid processing it again later.

This increases the maximum size of the priority queue from V to E, changing the complexity to $O(ElogE)$. In practice, though, this is unimportant as $\log_2 E < 2 \log_2 V$ and the priority queue rarely reaches that size.

[1] You can do this with your own priority queue if you keep a map of key to position in the heap.

3.3 Extensions

The idea behind Dijkstra's algorithm can be used to implement other algorithms as well.

- In a graph where all edge weights are the same, the priority queue in Dijkstra's algorithm can be replaced by a queue, giving a **breadth-first search**.
- If we are only looking for the shortest path to a destination *dest*, then we can use $dist(v) + h(v, dest)$ as the priority queue key for vertex v. This is known as the **A* algorithm**. This requires a heuristic function $h(u, v)$ that provides a lower bound on $dist(u, v)$; Dijkstra's algorithm simply uses $h(u, v) = 0$.
- **Prim's algorithm** for finding the minimum spanning tree of a graph is remarkably similar to Dijkstra's. The only difference is that the graph must be undirected, and we use $weight(u, v)$ instead of $dist(v)$ as the priority queue key. In other words, instead of using the distance to *src*, we use the distance to the tree of vertices in *visited*.

4 Bellman-Ford

If a graph contains negative edge weights, Dijkstra's algorithm cannot be used, because it is a greedy algorithm. Instead, we can use the **Bellman-Ford** algorithm, which simply relaxes all E edges, $V-1$ times, which is the maximum path length.

If the graph contains a negative cycle, there is no "shortest" path. However, the Bellman-Ford algorithm is useful for detecting such negative cycles. To do this, we check if any distance needs to be updated after running the loop $V-1$ times. If so, there is a path of length V, which is impossible without a negative cycle.

Algorithm 3 Bellman-Ford

```
for all vertices v do
    dist(v) ← ∞
    prev(v) ← −1
dist(src) ← 0
for i = 1, V − 1 do
    for all edges (u, v) do                          ▷ relax every edge
        if dist(u) + weight(u, v) < dist(v) then
            dist(v) ← dist(u) + weight(u, v)
            prev(v) ← u
for all edges (u, v) do                              ▷ check for negative cycles
    if dist(u) + weight(u, v) < dist(v) then
        negative cycle detected
```

The complexity of such an algorithm is obviously $O(EV)$. This can be improved by maintaining a queue of vertices whose distances were updated. Although the worst-case complexity remains the same, the average number of iterations decreases to E. However, checking for negative cycles will instead require us to look for cycles in the shortest-path tree[2] defined by *prev*.

[2] Well, if there is a cycle, it's not a tree...