From Floyd to SPFA

Sunday Lee

December 20, 2013

Abstract

This paper consists of the following three parts: 1) Introduction of the shortest route problem. 2) Principles and implementations of several shortest route algorithms. 3) A comparison on the time and memory space complexities and prerequisites of these algorithms.

1 Problem Description

The problem of finding the shortest route between two vertices in a graph is a well-researched problem in graph theory, and some definitions and formal descriptions of this problem are as follow:

Route: A route from vertex v_1 to v_n is a sequence of vertices $\{v_1, v_2, \cdots, v_{n-1}, v_n\}$ such that for any pair of adjacent vertices v_i and v_{i+1} there exists edge (v_i, v_{i+1}) (for undirected graph) or $< v_i, v_{i+1} >$ (for directed graph), and its weight is the sum (or maximal or minimal value) of all the weights of concerned edges. A route is also known as a path.

Distance between a pair of vertices: The distance between vertex v_i and vertex v_j is the amount of edges on the shortest route between them.

Shortest route problem: The problem in which a directed or undirected graph is given while the aim is to find out a route with maximal or minimal weight.

Single source shortest route problem: The problem of working out the shortest routes from a given vertex to all other vertices, sometimes also known as Single Source Shortest Path Problem (SSSP). The algorithms designed to solve such problem that will be discussed in this paper include the Dijkstra algorithm, the Bellman-Ford algorithm and the Shortest Path Faster Algorithm (SPFA).

Multi-source shortest route problem: The problem of working out the shortest routes between every pair of vertices $\{v_i, v_j\}$ in a given graph, sometimes also known as Multi-source Shortest Path Problem (MSSP). The algorithm designed to solve such problem that will be discussed in this paper is the Floyd algorithm.

2 Notes

1. For convenience, it will be assumed that the graph we talk about will be directed, in which the weight of a route is defined as the algebric sum of all edge weights, and our goal is to figure out a route with minimal weight. Solution that works under such constraints can be easily generalized to be compatible with other kinds of graphs.

2. Unless separately noted otherwise, the symbol n will stand for $|V|$ (the number of vertices) and the symbol m will stand for $|E|$ (the number of edges). The beginning and ending vertex of the i-th edge will be referred as $u[i]$, $v[i]$, and the weight of the i-th edge will be referred as $w[i]$.

3. The implementations of algorithms described in this paper will be given as pseudocodes.

4. For the Dijkstra algorithm, the given graph cannot contain edge with negative weight. For other algorithms, the given graph cannot contain loop with negative

weight (otherwise there will be no *shortest* route since by following this loop we can *always* decrease the weight of the current route.

3 The Floyd Algorithm

In this section we will first explain the principle of the Floyd algorithm, which is intended to solve MSSP, and then we will give a clear implementation of this algorithm.

The Floyd algorithm uses a pattern similar to Dynamic Programming. At first, we can label the vertices with indices $1-n$. And then we can define symbol $f_{i,j,k}$ to be the weight of the *current* shortest route between vertex i (by vertex i it means the vertex with index i) and vertex j which contains only vertices with indices less than or equal to k (except for vertex i and vertex j). And then we can recursively work out the value of $f_{i,j,k}$ for all (i, j, k).

Now suppose we have all values of $f_{i',j',k'}$ for $k' < k$, and we want to calculate all values of $f_{i,j,k}$. For each combination of i, j, k, we have two choices:

1. To preserve the shortest route given by $f_{i,j,k-1}$, which is still valid for an increased k. So that $f_{i,j,k} = f_{i,j,k-1}$.

2. To introduce a new route that has vertex k on it. Since our goal is to figure our the *shortest* routes from vertex i to vertex j, we should combine the *current* shortest route from vertex i to vertex k and the *current* shortest route from vertex k to vertex j to get a new route going from vertex i to vertex j. Thus, we will have $f_{i,j,k} = f_{i,k,k-1} + f_{k,j,k-1}$.

And since we are attempting to minimize the route weight, we can have $f_{i,j,k} = minimal\{f_{i,j,k-1}, f_{i,k,k-1} + f_{k,j,k-1}\}$.

So far we are already able to construct a raw implementation of the Floyd algorithm which uses memory space that is proportional to n^3. However, with a little further observation it is clear that $f_{i,j,k}$ depends solely on $f_{i,j,k'}$ where $k' = k-1$. So we can safely eliminate the last dimension and obtain an implementation with $\Theta(n^2)$ memory space complexity.

The final transition equation is as below:

$$f_{i,j} = minimal\{f_{i,j}, f_{i,k} + f_{k,j}\}$$

And a pseudocode implementation of the algorithm above is as below (the final result is stored in array f, and INFINITY stands for ∞, which can be replaced with a very big number when it comes to actual implementation:

```
SHORTEST-PATH-FLOYD-ALGORITHM()
for i in [1, n]
 for j in [1, n]
  f[i][j] = INFINITY
for i in [1, m]
 f[u[i]][v[i]] = w[i]
for k in [1, n]
 for i in [1, n]
  for j in [1, n]
   if f[i][k] + f[k][j] < f[i][j] then
    f[i][j] = f[i][k] + f[k][j]
```

It is important to note that as the calculation of $f_{i,j,k}$ solely depends on the value of $f_{i',j',k-1}$, the iteration of k must be the outermost loop of the three loops.

And to analyze the memory space complexity of the above implementation, we notice that the implementation requires a 2-dimensional array with size of each dimension proportional to n, so the memory space complexity is $\Theta(n^2)$. And this implementation requires no addional memory space other than the array used to store the final result.

And as for the time complexity of this implementation, it is clear that there are three loops nested together with size of each one proportional to n, so the time complexity of this implementation is $\Theta(n^3)$. And as this algorithm computes the shortest routes for all $\frac{n(n-1)}{2}$ pairs of vertices, the average time complexity of computing one shortest route is $\Theta(n)$.

4 The Dijkstra Algorithm

The Dijkstra algorithm is an algorithm designed to solve SSSP. It is a greedy-based algorithm. Before introducing this algorithm, we will first introduce a basic operation used in many shortest route algorithms: tense operation.

To apply tense operation to an edge, we first see if for the starting vertex s, ending vertex t and its weight w the following inequation holds (where f_i stands for the *current* shortest path weigth from the given source vertex to vertex i):

$$f_s + w < f_t$$

If such inequation holds, it means if we take the *current* shortest path from vertex s, and combine it with the edge we are applying tense operation to, we will obtain a better path from the given source vertex to vertex t. So we should update f_t as following:

$$f_t = f_s + w$$

And now we can introduce the basic pattern of the Dijkstra algorithm (s stands for the given source vertex):

1. *Initialization*: Put all vertices of the graph in a set S, and set the shortest path weight from the given source vertex to itself, that is f_s, to zero, and set the shortest path weight from the given source vertex to other vertices, that is f_i where $1 \leq i \leq n, i \neq s$, to infinity.

2. *Selection*: Among all vertices in set S we choose one vertex v with minimal f_v. If set S is an empty set, the algorithm terminates.

3. *Tense*: Apply tense operation to all edges whose beginning vertex is vertex v.

4. *Removal*: Remove vertex v from set S, and continue to step 2.

The basic idea of the Dijkstra algorithm is for each time to find a vertex with minimal shortest path weight from the source vertex, and apply tense operation to all edges starting from it, and eventually we can get all shortest routes from the given source vertex settled.

Now we will offer a proof on the accuracy of the algorithm above.

First, everytime we choose vertex v from set S, the shortest path from the given source vertex to vertex v is already *optimal*, that is, the value of the shortest route weight from the given source vertex s to vertex v is no longer *current*, but is rather already *permanent*.

First we can use mathematical induction to prove this point. At the beginning, the only vertex with non-infinity shortest route weight is the given source vertex s, so it is obvious that the statement holds.

And now we will prove that after each operation, the previously valid statement will still hold. We use contradiction to prove this point. If there is such route from source vertex s to vertex v that leads to a smaller route weight than our current one, then we can set t to be vertex that precedes vertex v in this path, and we can classify the situaton into two categories:

1. Vertex t is in set S.

2. Vertex t is not in set S.

For the first category, the pattern of the algorithm ensures that only vertices that have been used to *update* other vertices will be removed from set S. And by the induction prerequisite, we know when vertex t is selected, its shortest route weight f_t is permanent. And during the updating process, t will eventually update f_v with $f_t + w_{t,v}$ (where $w_{t,v}$ stands for the weight of the edge starting at vertex t and ending at vertex v). So after this update f_v can not possibly be larger than $f_t + w_{t,v}$. And this leads to a contradiction with our assumption.

And for the second category, we have:

$$f_t + w_{t,v} < f_v \tag{1}$$

but we always choose vertex v with minimal f_v and $w_{t,v} \geq 0$ (this is an important limitation of the Dijkstra algorithm), we also have:

$$f_v \leq f_t \leq f_t + w_{t,v}$$

and this is inconsistent with equation 1. So this also leads to a contradiction with our assumption.

Combined all the statement above, we can now conclude that everytime we choose vertex v from set S, the shortest path from the given source vertex to vertex v will be already *optimal*.

So when the algorithm terminates, which implies that every vertex has been chosen exactly once, all the shortest route weight f_i will be *optimal*.

And here follows a pseudocode for the Dijkstra algorithm (where s is the source vertex, $w[i][j]$ stands for the weight of minimal-weighted edge starting at vertex i and ending at vertex j, and will be infinity if no such edges exist, and array x stores whether a vertex is in set S, and f stores the *current* shortest route weight):

```
SHORTEST-PATH-DIJKSTRA-ALGORITHM(s)
for i in [1, n]
 f[i] = infinity
 x[i] = true
f[s] = 0
for i in [1, n]
 minn = infinity
 for j in [1, n]
  if f[j] < minn then
   minn = d[j]
   mini = j
 x[mini] = false
 for j in [1, n]
  if f[mini] + w[mini][j] < f[j] then
   f[j] = f[mini] + w[mini][j]
```

Similar to the analysis of the Floyd algorithm's complexities, we can know that the time complexity of the Dijkstra algorithm is $\Theta(n^2)$, and the memory space complexity of this algorithm is $\Theta(n)$.

5 The Bellman-Ford Algorithm

The Bellman-Ford algorithm is another algorithm designed to solve SSSP.

The Bellman-Ford algorithm is divided into stages. In each stage, we apply tense operation to all edges in the graph.

The fundamental basis of the Bellman-Ford algorithm is an important fact: after the first i stages, the shortest route weights of vertices whose distance from the given source vertex is no more than i are already *optimal*.

A brief proof to the above statement, which applies the idea of mathematical induction is as below:

We can let the vertex that precedes vertex i on vertex i's shortest route from source vertex be vertex j, then it is obvious that the *distance* between the source vertex and vertex i is equal to the *distance* between the source vertex and vertex j plus one. Thus, the weight of the shortest route between the source vertex and vertex j has been already settled in the earlier stages. So after another tense operation to the edge between vertex i and j, the weight of shortest route between the source vertex and vertex i can as well be settled.

Since for any vertex i, its distance from the source vertex can be at most $n-1$ (for if not, there must be a loop in the shortest route, which can be removed to obtain a shorter route), after at most $n-1$ stages, all shortest route weights will be settled and then the algorithm can safely terminate.

And a pseudocode implementation of the Bellman-Ford algorithm follows (where s stands for the source vertex and array d stores the *current* shortest route weight):

```
SHORTEST-PATH-BELLMAN-FORD-ALGORITHM(s)
for i in [1, n]
 f[i] = infinity
f[s] = 0
for i in [1, n)
 for j in [1, m]
  if f[u[j]] + w[j] < f[v[j]] then
   f[v[j]] = f[u[j]] + w[j]
```

Using similar techque, we can know that the memory space complexity of this implementation is $\Theta(n)$, and the time complexity of this implementation is $\Theta(nm)$. While it may be intuitive to think that according to complexity analysis Dijkstra algorithm outranks the Bellman-Ford algorithm, the latter actually requires a looser prerequisite, and is therefore compatible to some situations in which the Dijkstra algorithm cannot proceed.

6 The Shortest Path Faster Algorithm

The key factor that makes Bellman-Ford algorithm so time-consuming is that once the shortest route weight of from the source vertex to vertex i gets settled, it may be used to update other vertices in several later stages. But the only useful update is the first time. So if we only use those vertices whose shortest path's weight has been *changed* to do the update, we can still get the same result.

So now we need a efficient way to determine which vertices we need to use to update. In order to do that, we can use a first-in-first-out queue to store the vertices that needs to be used to update. Each time we extract the front item from the queue and use it to update. And if some vertex's shortest route weight has been *changed*, we can put that vertex into the queue. And it's noteworthy that in order to reduce the time used, we need to ensure that at any time for any vertex it would be in the queue only once. So we also need a array to store if a vertex is already in the queue, so that we can ensure we will not add the same vertex into the queue if it is already in.

And a pseudocode implementing the SPFA algorithm is as below (s, f described before, q is the queue, and x is a boolean array to store whether a vertex is already in the queue):

```
SHORTEST-PATH-SPFA-ALGORITHM(s)
for i in [1, n]
 f[i] = infinity
 x[i] = false
x[s] = true
f[s] = 0
q.push(s)
while q not empty do
 p = q.pop()
 x[p] = false
 for i in [1, m]
  if u[i] = p then
   if f[p] + w[i] < f[v[i]] then
    f[v[i]] = f[p] + w[i]
    if x[v[i]] = false then
     x[v[i]] = true
     q.push(v[i])
```

It is clear that the memory space complexity of this implementation is $\Theta(n)$, and its time complexity is $\Theta(km)$. And for most common graphs, by experiment, $k \approx 2$.

7 Comparison

We will do the comparison among the algorithms discussed from the following perspectives: 1) Time complexity. 2) Memory space complexity. 3) Programming complexity. 4) Prerequisites.

For the Dijkstra algorithm, it is necessary that the given graph must contain no edge with negative weight. And this prerequisite greatly limits its use. And for other algorithms, the only prerequisite is the absence of negetive weight loop, which is the gurantee of the existence of shortest routes. So it is fair to say that the compatible situations of all other three algorithms concerned are equivalent.

And for MSSP algorithm, the Floyd algorithm, the memory space complexity it requires is $\Theta(n^2)$, while for all other algorithms intended to solve SSSP problem, the memory space complexity remains $\Theta(n)$. But from the perspective of the average memory space complexity required to store each shortest route computed, all four algorithms are equally memory-consuming.

And for the analysis on time complexity, the Floyd algorithm computes all $\Theta(n^2)$ shortest routes in a time of proportional to n^3. And a raw implementation of the Dijkstra algorithm computes all $\Theta(n)$ shortest routes

in $\Theta(n^2)$ time. Intuitively, these two algorithms seem to have the same average time complexity, but using a variant Dijkstra algorithm with heap-based optimization could decrease the worst time complexity of the Dijkstra algorithm to $\Theta(n \log n)$, which is very efficient at most times. And Bellman-Ford has a seemingly bad time complexity of $\Theta(nm)$, but its variant SPFA has a optimized performance which runs very fast at most times. But it is noteworthy that SPFA does *not* improve the worst time complexity of Bellman-ford, and it is still possible to build a graph in which SPFA runs as slowly as Bellman-Ford.

And in all, each algorithm has its own extension and features. Based on different graphs the choice of algorithm can also differ.