From Floyd to SPFA

Sunday Lee

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 & memory space complexities and the 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 this section of the paper introduces definitions and formal descriptions relavant to this problem:

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) in the graph. And the weight of such route is defined as the sum (or maximal minimal value, depending on specific application) of all the weights of concerned edges. A route is also known as a path.

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

Shortest route problem: The problem of identifying the route(s) with maximal (or minimal) weight in a given directed or undirected graph.

Single source shortest route problem: The problem of identifying out the shortest routes from a given vertex to all other vertices. 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 all pairs of vertices in a given graph. 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. Without loss of generality, this paper assumes the graph under discussion is connected and directed, with the weight of a route defined as the algebric sum of all edge weights. Furthermore, the aim is set to identify route with minimal weight. Algorithms that work under such specific case can be easily generalized to be compatible with other kinds of graphs by means including weight manipulation and undirected edge expansion.

2. Unless separately noted otherwise, the symbol n is defined as $|V|$ (the number of vertices in the graph) and the symbol m is defined as $|E|$ (the number of edges in the graph). The starting and terminating vertex of the i-th edge will be referred to as $u[i]$, $v[i]$, and the weight of the i-th edge will be referred to as $w[i]$.

3. The implementations of algorithms discussed 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 *indefinitely* 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. Then we will give a clear implementation of this algorithm.

The Floyd algorithm uses a pattern akin to dynamic programming. Initially, we label the vertices with indices $1-n$. Consequently, we 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, potentially, for vertex i and vertex j). And then we can recursively calculate the values of $f_{i,j,k}$ for all triples (i, j, k) as follows:

Suppose we have all values of $f_{i',j',k'}$ for $k' < k$. Based on those values, we then 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 identify the *shortest* route 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 through vertex k. Thus, we should have $f_{i,j,k} = f_{i,k,k-1} + f_{k,j,k-1}$.

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

A raw implementation of the Floyd algorithm as disccused uses an amount of memory space that is $\Theta(n^3)$. However, with further observation it is clear that the calculation of values of $f_{i,j,k}$ depends solely on the values of $f_{i,j,k-1}$. Therefore, we can safely eliminate the last dimension in the state and obtain an implementation with $\Theta(n^2)$ memory space complexity.

The final transition equation is as below:

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

A pseudocode implementation of the algorithm above is as below (the final result is stored in the array f, and INFINITY stands for ∞, which can be replaced with a sufficiently large number in 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 $f_{i,j,k}$ depends on the values of $f_{i',j',k-1}$, the iteration of k must be the outer-most loop of the three loop layers.

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 being $\Theta(n)$. Thus, the memory space complexity of the implementation is $\Theta(n^2)$. Further more, 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 nested loops each with a size of $\Theta(n)$. Therefore, 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 a greedy-based algorithm designed to solve SSSP. To begin with, we will introduce a basic operation used in various shortest route algorithms: the loosen operation.

To apply the loosen operation to an edge, we first check if the following inequality holds for the starting vertex s, ending vertex t and the edge weight w (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 inequality holds, it means if we take the *current* shortest route from source vertex to vertex s, and combine it with the edge we are applying the loosen operation to, we will obtain a better (in terms of minimizing route weight) route from the given source vertex to vertex t. So we should update f_t as follows:

$$f_t = f_s + w$$

With the loosen operation, 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. Set the shortest route weight from the given source vertex to itself, that is f_s, to zero. 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 (or a sufficiently large number in actual implementation).

2. *Selection*: Among all vertices in set S select the vertex v with minimal f_v (break ties arbitrarily). If set S is an empty set, the algorithm terminates.

3. *Loosen*: Apply the loosen operation to all edges starting at vertex v.

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

Generally, the principle of the Dijkstra algorithm is to continuously identify the vertex with minimal shortest path weight from the source vertex, and then to apply the loosen operation to all edges starting at it.

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

To begin with, everytime we select 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 *temporary*, but already *permanent*.

To prove the statement above, we use mathematical induction. Initially, the only vertex with non-infinity shortest route weight is the given source vertex s, rendering the initial case trivial.

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 condition, we know that 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 terminating at vertex v). So after this update f_v must be less than or equal to $f_t + w_{t,v}$. 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 since we always choose vertex v with minimal f_v and $w_{t,v} \geq 0$ (note that this is an

major limitation of the Dijkstra algorithm), we also have:

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

which is inconsistent with equation 1. Therefore, this also leads to a contradiction with our assumption.

Combined the two cases 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 the *globally optimal*.

Thus, at the end of the algorithmś execution, as the algorithm terminates only when all vertices have been selected exactly once, all shortest route weights f_i will be *optimal*.

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

```
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]
```

Following complexities analysis similar to that of the Floyd algorithm, we can derive that the time complexity of the Dijkstra algorithm is $\Theta(n^2)$, and the memory space complexity of the 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.