Shortest Path in Graphs

Xiangchen Tian
IIIS, Tsinghua University
txc23@mails.tsinghua.edu.cn

Abstract

This paper explores algorithms for finding the shortest paths in graphs, focusing on both the Single-Source Shortest Path (SSSP) problem and the All-Pairs Shortest Path (APSP) problem. For the SSSP, we review Dijkstra's Algorithm for graphs with non-negative edge weights and the Bellman-Ford Algorithm for graphs that may contain negative edge weights. For the APSP, we discuss Johnson's Algorithm, which efficiently handles graphs with negative weights by reweighting edges using feasible potentials and applying Dijkstra's Algorithm, and the Floyd-Warshall Algorithm, a dynamic programming approach for computing shortest paths between all pairs of vertices. Additionally, we explore the connection between min-sum products and the APSP problem. We implemented these algorithms in C++ and tested them on random graphs to compare their performance. The experimental results validate the theoretical analyses and demonstrate the practical efficiency of the algorithms.

The source code is available on Github.

Keywords: Graph Theory, Shortest Path

Contents

1 Introduction

1.1 Background and Outline

In graph theory, a very simple and basic idea is to ask: given a graph with a weight for each edge, find the path with the smallest weight between any two points in the graph. Here, the weight of a path is the sum of the weights of all the edges in the path. This paper first reviews the most basic solutions to this type of problem, then conducts a comprehensive analysis and comparison of the feasibility and running time of its variant algorithms, and finally tests the theory in practice by measuring the running time of the algorithm implemented in C++.

2 Single-Source Shortest Path Problem(SSSP)

2.1 Problem description

Given a graph, we want to find a shortest path from a single source vertex s to every other vertex in the graph. The graph can be directed or undirected, weighted or unweighted (just take all the weights equal to 1, as a special case of the weighted graph), and can contain negative weights.

We first briefly review the case without negative weights. In this case, the algorithm is Dijkstra's Algorithm, which is similar to what is taught in class, except that the algorithm is more closely integrated with the actual implementation. We then briefly review the case with negative weights. In this case, the algorithm is the Bellman-Ford Algorithm, which can report the finding of negative cycles when there are (reachable) negative weight cycles, and give the optimal path for each point given the source point s when there are no (reachable) negative cycles.

2.2 Dijkstra's Algorithm for Non-negative Weights Graphs

Pesudo-code see Algorithm 1.

Algorithm 1 Dijkstra's Algorithm

```
Require: Digraph $G = (V, E)$ with edge-weights $w_e \geq 0$ and source vertex $s \in G$
Ensure: The shortest-path distances from each vertex to $s$
1: add $s$ to heap with key 0
2: for $v \in V \setminus \{s\}$ do
3:     add $v$ to heap with key $\infty$
4: end for
5: while heap not empty do
6:     $u \leftarrow$ deletemin
7:     for each neighbor $v$ of $u$ do
8:         key($v$) $\leftarrow$ min{key($v$), key($u$) + $w_{uv}$} {relax $uv$}
9:     end for
10: end while
```

Below we give a brief process for proving the correctness of the algorithm, which is similar to the proof process in the textbook (Section 4.4, Greedy Algorithm) and the proof process described in class.

Lemma 2.1. *Consider the set $V \backslash \{$current elements in the heap$\}$ at any point in the algorithm's execution. For each vertices u in $V \backslash \{$current elements in the heap$\}$, the key(u) of the priority queue is the shortest-path distances from each vertex to s.*

Proof. We prove the lemma by induction on the size of the set

$$S = V \backslash \{\text{current elements in the heap}\}$$

The case $|S| = 1$ is easy, since then we have $S = \{s\}$ and $d(s) = 0$.

Suppose the claim holds when $|S| = k$ for some value $k \geq 1$. We now grow S to size $k+1$ by adding the node v. Let $e_{uv} = (u, v)$ be the final edge on our $s-v$ path P_v. Now consider any other $s-v$ path P, this P must leave the set S somewhere to reach the vertex v, when it go out of S, it's accumulate weight already larger than the weight of path P_v (which is equal to key(v)), so P_v is the shortest path from s to v. □

Running time analysis: If we use a binary heap, which incurs $O(\log n)$ for decrease-key as well as extract-min operations, we incur a running time of $O(m \log n)$. If we use a Fibonacci heap, which incurs $O(1)$ for decrease-key and $O(\log n)$ for extract-min operations, we incur a running time of $O(m + n \log n)$. These two methods are implemented in C++, see [TBD].

2.3 Bellman-Ford Algorithm for Graphs with Negative Weights

Pesudo-code see Algorithm 2.

Algorithm 2 Bellman-Ford Algorithm

```
Require: A digraph $G = (V, E)$ with edge weights $w_e \in \mathbb{R}$, and source vertex $s \in V$
Ensure: The shortest-path distances from each vertex to $s$, or report that a negative-weight
    cycle exists
 1: dist($s$) $\leftarrow$ 0 {the source has distance 0}
 2: for each $v \in V$ do
 3:     dist($v$) $\leftarrow \infty$
 4: end for
 5: for $|V|$ iterations do
 6:     for each edge $e = (u, v) \in E$ do
 7:         dist($v$) $\leftarrow$ min{dist($v$), dist($u$) + weight($e$)}
 8:     end for
 9: end for
10: if any distances changed in the last (nth) iteration then
11:     output "$G$ has a negative weight cycle"
12: end if
```

Below we also give a brief process for proving the correctness of the algorithm, which is similar to the proof process in the textbook (Section 6.8, Dynamic Programming) and the proof process described in class. In the discussion below, let s be the source vertex, t be the target vertex which is fixed. This case is the opposite of the algorithm above, in which the source point is fixed and the sink point will change; but despite this, the proof process is exactly the same. The simplest idea is to reverse each directed edge and perform the same derivation, in which case the source point and sink point will be swapped.

Lemma 2.2. *If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most $n-1$ edges, where $n = |V|$.*

Proof. Since every cycle has nonnegative cost, the shortest path P from s to t with the fewest number of edges does not repeat any vertex v. For if P did repeat a vertex v, we could remove the portion of P between consecutive visits to v, resulting in a path of no greater cost and fewer edges. There are at most n vertices, and P does not repeat any vertex, so P has at most $n-1$ edges. □

Lemma 2.3. *Denote $OPT(i, v)$ to be the minimum cost of a $v - t$ path using at most i edges. Then if $i > 0$, we have*

$$OPT(i, v) = \min \left\{ OPT(i-1, v), \min_{w \in v\text{'s neighbour}} (OPT(i-1, w) + c_{vw}) \right\}$$

Proof. $\text{OPT}(i, v)$ can only have two possibilities:

(i) The path uses at most $i - 1$ edges, in which case the cost is $\text{OPT}(i - 1, v)$.

(ii) The path uses exactly i edges, in which case the cost is $\min_{w \in v\text{'s neighbour}}(\text{OPT}(i-1, w) + c_{vw})$.

□

Note: The algorithm provided above is the 'Improving the Memory Requirements' version of the Bellman-Ford algorithm, which only use $O(n)$ space. This version's correctness is based on the observation "Throughout the algorithm $\text{dist}(v)$ is the length of some path from v to t, and after i rounds of updates the value $\text{dist}(v)$ is no larger than the length of the shortest path from v to t using at most i edges". For details, see the KT textbook (pages 295 to 297).

Running time analysis: there are $n = |V|$ iterations, and in each iteration, we visit all the edges, so the running time is $O(mn)$.

3 All-Pairs Shortest Path Problem(APSP)

The obvious way to do this is to run an algorithm for SSSP n times, each time with a different vertex being the source. This method will give $O(mn + n^2 \log n)$ runtime for non-negative weights graphs by using n rounds of Dijkstra Algorithm, and $O(mn^2)$ runtime for graphs with negative weights by using n rounds of Bellman-Ford Algorithm. But obviously, we can take advantage of the correlation between n problems from different source points to reduce the time complexity to get a runtime of $O(mn + n^2 \log n)$ with general edge weights, which is known as Johnson's Algorithm.

3.1 Feasible Potentials

General idea: The negative weight is the main obstacle to the efficiency of the algorithm. Is there a way to transform the graph so that the weights are non-negative? The answer is yes, and this is why we introduce the concept of feasible potentials.

Definition 3.1. *For a weighted digraph (a.k.a. directed graph) $G = (V, E)$, a function $\phi : V \to \mathbb{R}$ is a feasible potential if for all edges $e = (u, v) \in E$,*

$$\phi(u) + w_{uv} - \phi(v) \geq 0$$

This formula is a bit like the triangle inequality. (This reminds me of Admissible Heuristics in search algorithms in reinforcement learning.)

Given a feasible potential, we can transform the edge-weights of the graph from w_{uv} to

$$\hat{w}_{uv} = w_{uv} + \phi(u) - \phi(v)$$

Lemma 3.1. *The new weights $\hat{w}$ are all positive.*

Proof. (Straight forward) Directly use the definition of the feasible potential. □

Lemma 3.2. *Let P_{ab} be a path from vertex a to vertex b. Let $\ell(P_{ab})$ be the length of the path P_{ab} when the edge weights are w. Let $\hat{\ell}(P_{ab})$ be the length of the path P_{ab} when the edge weights are $\hat{w}$. Then*

$$\hat{\ell}(P_{ab}) = \ell(P_{ab}) + \phi(a) - \phi(b)$$

Proof. (Straight forward) The feasible potentials of all intermediate nodes cancel each other out. □

Lemma 3.2 tells us that if we can find a feasible potential, we can transform the graph so that the shortest path in the transformed graph is the same as the shortest path in the original graph and the new graph has non-negative weights. This is the key to Johnson's Algorithm. But how can we find feasible potentials? A very clever idea: Use Bellman-Ford Algorithm. suppose there some source vertex $s \in V$ such that every vertex in V is reachable from s, if we set $\phi(v) = \text{dist}(s, v)$, then ϕ is a feasible potential because for any edge $e = (u, v)$, we have

$$\phi(u) + w_{uv} - \phi(v) = \text{dist}(s, u) + w_{uv} - \text{dist}(s, v) \geq 0$$

Intuition: The shortest path distance naturally satisfies the triangle inequality! The original graph can have negative weights, and we want to find a distance matric, obviously we should first run the Bellman-Fold algorithm once. We summarize the above discussion in the following lemma.

Lemma 3.3. *Given a digraph $G = (V, A)$ with vertex s such that all vertices are reachable from s, $\phi(v) = dist(s, v)$ is a feasible potential for G.*

Some other properties of feasible potentials:

Lemma 3.4. *If all edge-weights are non-negative, then $\phi(v) = 0$ is a feasible potential.*

Proof. (Straight forward) For all edges $e = (u, v)$, we have

$$\phi(u) + w_{uv} - \phi(v) = w_{uv} \geq 0$$

□

Lemma 3.5. *Adding a constant to a feasible potential gives another feasible potential.*

Proof. (Straight forward) For all edges $e = (u, v)$, we have

$$\phi(u) + w_{uv} - \phi(v) = \phi(u) + c + w_{uv} - \phi(v) - c = \phi(u) + w_{uv} - \phi(v)$$

□

Lemma 3.6. *If there is a negative cycle in the graph, there can be no feasible potential.*

Proof. The sum of the new weights along the cycle is the same as the sum of the original weights, due to the telescop- ing sum. But since the new weights are non-negative, so the old weight of the cycle must be, too, which is a contradiction. □

Lemma 3.7. *If we set the feasible potential of some vertex s to 0, i.e. $\phi(s) = 0$, then $\phi(v)$ for any other vertex v is an underestimate of the s-to-v distance.*

Proof. For all the path from s to v, we have

$$0 \leq \hat{\ell}(P_{sv}) = \ell(P_{sv}) + \phi(s) - \phi(v) = \ell(P_{sv}) - \phi(v)$$

$$\implies \ell(P_{sv}) \geq \phi(v)$$

□

A result of Lemma 3.7 is that if we set the feasible potential of some vertex s to 0, and try to maximize summation of $\phi(v)$ for other vertices subject to the feasible potential constraints we will get an LP that is the dual of the shortest path LP:

$$\begin{array}{ll} \text{Maximize} & \sum_{x\in V}\phi(x) \\ \text{Subject to} & \phi(s)=0 \\ & \phi(u)+w_{uv}-\phi(v)\geq 0 \quad \forall(u,v)\in E \end{array}$$

3.2 Johnson's Algorithm

Pesudo-code see Algorithm 3.

Algorithm 3 Johnson's Algorithm

```
Require: A weighted digraph $G = (V, A)$
Ensure: A list of the all-pairs shortest paths for $G$
 1: $V' \leftarrow V \cup \{s\}$ {add a new source vertex}
 2: $A' \leftarrow A \cup \{(s, v, 0) \mid v \in V\}$
 3: dist $\leftarrow$ BellmanFord($(V', A')$) {set feasible potentials}
 4: for each edge $e = (u, v) \in A$ do
 5:     weight($e$) $\leftarrow$ weight($e$) + dist($u$) − dist($v$)
 6: end for
 7: $L \leftarrow []$ {the result}
 8: for each $v \in V$ do
 9:     $L \leftarrow L$ + Dijkstra($G, v$)
10: end for
11: return $L$
```

The correctness of Johnson's Algorithm is discussed in Section 3.1.

Running time analysis: The Bellman-Ford algorithm takes $O(mn)$ time, the transformation of the weights takes $O(m)$ time, and the n Dijkstra's algorithm takes $O(n(m+n\log n))$ time (if we use Fibonacci heaps), so the total running time is $O(mn+n^2\log n)$.

3.3 Floyd-Warshall Algorithm

We introduce the Floyd-Warshall algorithm, which is a dynamic programming algorithm that solves the all-pairs shortest path problem in a totally different perspective.

Pesudo-code see Algorithm 4.

We prove the correctness of the Floyd-Warshall algorithm by induction on the number of vertices in the graph.

Lemma 3.8. *After we have considered vertices $V_k = \{z_1, z_2, \cdots, z_k\}$ in the outer loop, dist(u, v) equals the weight of the shortest $x-y$ path that uses only the vertices from V_k as internal vertices. (This is ∞ if there are no such paths.)*

Proof. We prove the lemma by induction on the number of vertices in V_k. (Dynamic programming)

Base case: $k = 0$, the lemma is trivially true because the dist is initialized in that way.

Inductive step: Suppose the lemma holds for $k = i$, we prove it for $k = i + 1$.

Add z_{i+1} to V_i, then the shortest path from x to y that uses only the vertices from V_{i+1} as internal vertices is either the shortest path from x to y that uses only the vertices from V_i as internal vertices, or the shortest path from x to z_{i+1} that uses only the vertices from V_i as

Algorithm 4 Floyd-Warshall Algorithm

Require: A weighted digraph $D = (V, E)$
Ensure: A list of the all-pairs shortest paths for D

```
1: for each $(x, y) \in V \times V$ do
2:   if $(x, y) \in E$ then
3:     $d(x, y) \leftarrow w_{xy}$
4:   else
5:     $d(x, y) \leftarrow \infty$
6:   end if
7: end for
8: for each $z \in V$ do
9:   for each $x \in V$ do
10:    for each $y \in V$ do
11:      $d(x, y) \leftarrow \min\{d(x, y), d(x, z) + d(z, y)\}$
12:    end for
13:  end for
14: end for
15: return $d$
```

internal vertices and the shortest path from z_{i+1} to y that also uses only the vertices from V_i as internal vertices.

□

Finally, when the outer loop is finished, the lemma is true for $k = n$. So the Floyd-Warshall algorithm is correct.

Running time analysis: The Floyd-Warshall algorithm takes $O(n^3)$ time clearly. Although is not better than Johnson's algorithm, but is quite simple and easy to implement with minimal errors; also, the Floyd-Warshall algorithm can parellelizable, which is also quick in practice.

3.4 Min-Sum Products and APSPs

A conceptually different way to get shortest-path algorithms is via matrix products. They seem to have nothing to do with each other, but if we modify the definition of matrix multiplication, we will find that there is a deep connection between them.

The standard definition of matrix multiplication is that

$$[A \cdot B]_{ij} = \sum_k A_{ik} \cdot B_{kj}$$

where $[M]_{ij}$ is the element in the i-th row and j-th column of matrix M. This definition can seen as matrix product on the field $(\mathbb{R}, +, \cdot)$. We can surely change the underlying field to $(\mathbb{R}, \min, +)$, then we get the Min-Sum Product (MSP):

$$[A \odot B]_{ij} = \min_k \{A_{ik} + B_{kj}\}$$

i.e. above equation is the usual matrix multiplication, but over the semiring $(\mathbb{R}, \min, +)$.

It turns out that computing Min-Sum Products is precisely the operation needed for the APSP problem. If we initialize a matrix D as in the same way in Floyd-Warshall algorithm,

$$D_{ij} = \begin{cases} w_{ij} & \text{if } (i, j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

Then consider the meaning of $[D \odot D]_{ij}$,

$$[D \odot D]_{ij} = \min_k \{D_{ik} + D_{kj}\}$$

This is exactly the meaning of the shortest path from i to j that uses only one intermediate vertex! Similarly, $[D \odot D \odot D]_{ij}$ is the shortest path from i to j that uses only two intermediate vertices, and so on. Since the shortest paths would have at most $n-2$ intermediate vertices, we only need to compute $D^{\odot n-1}$.

If we calculate the Min-Sum Products of the matrix one by one, each multiplication will take $O(n^3)$ time, a total of $O(n)$ multiplications will be required, and a total of $O(n^4)$ time will be required, which is slower than the previous algorithms. A simple improve is noticed that

$$D^{\odot 2k} = D^{\odot k} \odot D^{\odot k}$$

then the time complexity is reduced to $O(n^3 \log n)$.

4 Experiment Results

I implemented all the algorithm above in C++, and tested them on a random graph with n vertices and m edges, where

$$m \approx \frac{n \log_2 n}{2}$$

for simplicity.

The code is available on Github.

Recall that the running time of the algorithms are as follows:

- Dijkstra's Algorithm: $O(m \log n)$ (for binary heap, my code belongs to this case) or $O(m + n \log n)$ (for Fibonacci heap)
- Bellman-Ford Algorithm: $O(mn)$
- Johnson's Algorithm: $O(mn \log n)$ (for binary heap, my code belongs to this case) or $O(mn + n^2 \log n)$ (for Fibonacci heap)
- Floyd-Warshall Algorithm: $O(n^3)$
- Min-Sum Products (naive version): $O(n^4)$