CS260: Project Final Report
Single-source Shortest Path Algorithms

Wenting Wu, Guoqing Ma, Peihao Zhu, Yuanzhao Chen
wenting.wu@kaust.edu.sa, guoqing.ma@kaust.edu.sa,
peihao.zhu@kaust.edu.sa, yuanzhao.chen@kaust.edu.sa

December 3, 2017

1 Introduction

Shortest path problem is the problem of finding a path between two vertices in a graph, which is one with the smallest weight among all paths between these two vertices. Algorithms of shortest path problem has been widely used in real world applications like best path finding in maps, robotics and VLSI design [7]. The weight of a path can represent properties like distance, time, cost, etc. Apart from finding shortest path between a single source vertex and a single destination vertex, called the single-pair shortest path problem, there are several variations. Single-source shortest path problem is a variation in which we have to find shortest paths from a source vertex to all other vertices in the graph. In single-destination shortest path problem, we have to find shortest paths to a destination vertex from all other vertices. Single-destination shortest path problem can be reduced to single-source shortest path problem by reversing the directed edges in the directed graph. In all-pairs shortest path problem, we have to find shortest paths between every pair of vertices [7]. In this project, we are going to compare two algorithms for solving single-source shortest path problem, Dijkstra's Algorithm [4] and Shortest Path Faster Algorithm (SPFA) [6]. More specifically, we are going to use Dijkstra's algorithm with a priority queue optimization, to compare to SPFA, which is a FIFO queue optimization of the classical Bellman-Ford algorithm [3]. We will conduct a series of experiments with different settings to examine their performances in time complexity. We will further discuss their differences based on the experimental results.

2 Preliminaries

Let $G = (V, E)$ be a directed graph with $s \in V$ being the source node. Formally, single-source shortest path problem can be formulated as finding a shortest path from source node s to every other node. Each edge e is associated with its weight $\omega(e)$. A path p is a sequence of edges which connect a sequence of vertices. The source node of the first edge in a path is the source of a path, whilst the sink node of the last edge is called the destination of a path. The weight of a path is defined as the summation of all edges' weight along the path, $\omega(p) = \sum_{e \in p} \omega(e)$. The shortest path from node u to node v is the one with the smallest weight among all paths from u to v.

3 Algorithms

3.1 Dijkstra's Algorithm

3.1.1 Algorithm Description

Dijkstra's algorithm is an algorithm to solve single-source shortest path problem, which can be used in cases that the weight of every edge is non-negative. In Dijkstra's algorithm, we use $d(v)$ to describe the length of the shortest path from source s to node v. We use $prev(v)$ to describe the source node of the last edge of the found shortest path from s to v. In the beginning, we set $d(s) = 0$ and $d(v) = +\infty, \forall v \neq s$, and enqueue all nodes into an empty priority queue Q with d as the key value. In each iteration, we dequeue the node u from the priority queue which has the smallest d value. For each edge (u, v) starting from u, we set $d(v)$ to $\min\{d(v), d(u) + \omega(u, v)\}$. The iteration terminates when Q is empty. Dijkstra's algorithm can be written in the following pseudo-code:

Algorithm 1: Dijkstra's algorithm

```
1  Input:Directed Graph G = (V, E), weights ω, source s
2  Onput:d, prev;
3  d(s) ← 0;
4  for v ∈ V, v ≠ s do
5      d(v) ← ∞;
6  Q.initialize(key = d);
7  for v ∈ V do
8      Q.enque(v);
9  while Q is not empty do
10     u ← Q.deque();
11     for (u, v) ∈ E do
12         if d(u) + ω(u, v) < d(v) then
13             d(v) ← d(u) + ω(u, v);
14             Q.decrease_key(v, d(v));
15             prev(v) = u;
```

3.1.2 Proof of Correctness

Let us denote S to be the set $V \backslash Q$. During each iteration, we remove one node v from Q and add it to S. A path p_v can be formed by backtrack using the $prev(.)$ information, and $d(v)$ is the length of p_v. We can prove by induction on $|S|$ that p_v is the shortest path from s to v.

Proof. [4]

Let $|S| = 1$, then $d(s) = 0$ it is trivial that the shortest path from s to s is an empty edge sequence thus the length is 0.

Let us assume the considered statement holds when $|S| = k$ for some $1 \leq k \leq |V| - 1$.

During the iteration number $k+1$, let v be the node selected by the algorithm, and $e = (u, v)$ where $u = prev(v)$.

Let p be an arbitrary path from s to v, since $s \in S$ and $v \notin S$, there must exist an edge $e' = (x, y)$ where $x \in S, y \notin S$. Let us denote $l(p)$ the length of path p. According to the assumption, $l(p) \geq d(x) + w(e')$. According to the selection criterion of v, $l(p_v) = d(v) \leq d(x) + w(e')$, thus $l(p_v) \leq l(p)$. The assumption holds for $|S| = k + 1$.

□

3.1.3 Time Complexity

In Dijkstra's algorithm, each edge is visited at most once if we use adjacent list to store the edges. The time complexity of visiting edges and updating d is $O(|E|)$. Decrease key is performed also at most $|E|$ times [4]. For any implementation of the priority queue, the time complexity of decreasing key and dequeuing is $O(|E| \cdot T_{\mathrm{dk}} + |V| \cdot T_{\mathrm{em}})$, where T_{dk} and T_{em} are the complexities of the decrease-key and extract-minimum operations in the priority queue implementation Q, respectively. If we use Fibonacci heap [5], $T_{\mathrm{dk}} = O(1)$ and $T_{\mathrm{em}} = O(\log |V|)$. So the time complexity of decreasing key and dequeuing is $O(|E| + |V| \log |V|)$. The total complexity is $O(|E| + |V| \log |V|)$[4].

3.2 Shortest Path Faster Algorithm

3.2.1 Algorithm Description

Shortest Path Faster Algorithm (SPFA) is a solution to single-source shortest path problem which is believed to deal with edges with negative weights and work well on sparse graphs. In SPFA, we use $d(v)$ to describe the length of the shortest path from source s to node v. We use $prev(v)$ to describe the source node of the last edge of the found shortest path from s to v. In the beginning, we set $d(s) = 0$ and $d(v) = +\infty, \forall v \neq s$, and enqueue s into a FIFO queue Q with d as the key value. In each iteration, we dequeue the first node u from the Q. For each edge (u, v) starting from u, we set $d(v)$ to $\min\{d(v), d(u) + \omega(u, v)\}$. If $d(v)$ is updated and not in the FIFO queue, we enqueue v into Q. The iteration terminates when Q is empty. SPFA can be written in the following pseudo-code:

Algorithm 2: SPFA

```
1  Input:Directed Graph G = (V, E), weights ω, source s
2  Onput:d, prev;
3  d(s) ← 0;
4  for v ∈ V, v ≠ s do
5      d(v) ← ∞;
6  Q.initialize();
7  for v ∈ V do
8      Q.enque(v);
9  while Q is not empty do
10     u ← Q.deque();
11     for (u, v) ∈ E do
12         if d(u) + ω(u, v) < d(v) then
13             d(v) ← d(u) + ω(u, v);
14             prev(v) = u;
15             if v ∉ Q then
16                 Q.enque(v);
```

3.2.2 Proof of Correctness

As the algorithm fails to terminate in the case that negative-weight cycles present, we cannot prove it always gives correct results. Instead, we prove it never computes incorrect results [2].

Lemma 1. *Whenever the queue is checked for emptiness, any vertex currently capable of causing relaxation is in the queue.*

Proof. [2]
We want to show that if $d(v) > d(u) + w(u, v)$ for any two vertices u and v at the time the condition is checked, u is in the queue. We do so by induction on the number of iterations of the loop that have already occurred. First we note that this certainly holds before the loop is entered: if $u \neq s$, then relaxation is not possible; relaxation is possible from $u = s$, and this is added to the queue immediately before the while loop is entered. Now, consider what happens inside the loop. A vertex u is popped, and is used to relax all its neighbors, if possible. Therefore, immediately after that iteration of the loop, u is not capable of causing any more relaxations (and does not have to be in the queue anymore). However, the relaxation by u might cause some other vertices to become capable of causing relaxation. If there exists some x such that $d(x) > d(v) + w(v, w)$ before the current loop iteration, then v is already in the queue. If this condition becomes true during the current loop iteration, then either $d(x)$ increased, which is impossible, or $d(v)$ decreased, implying that v was relaxed. But after v is relaxed, it is added to the queue if it is not already present.

□

Corollary 1. *The algorithm terminates when and only when no further relaxations are possible.*

Proof. [2]
If no further relaxations are possible, the algorithm continues to remove vertices from the queue, but does not add any more into the queue, because vertices are added only upon successful relaxations. Therefore, the queue becomes empty and the algorithm terminates. If any further relaxations are possible, the queue is not empty, and the algorithm continues to run.

□

In a graph with no cycles of negative weight, when no more relaxations are possible, the correct shortest paths have been computed[1]. Therefore, whenever the algorithm terminates, it gives correct shortest paths.

3.2.3 Time Complexity

The algorithm of SPFA is an improvement of Bellman-Ford algorithm. But in the worst-case, its time complexity is the same as Bellman-Ford algorithm, which is $O(|E||V|)$, because each edge is visited at most $|V|$ times [4]. In connected non-negative graphs, $|E| \geq |V| - 1$, so the worst case time complexity of SPFA is always worse than Dijkstra's Algorithm. However, in random and sparse graphs, SPFA sometimes outperforms Dijkstra's algorithms. [6]

For the average case, we assume node v_i has $\epsilon = \frac{|E|}{|V|}$ edges starting from other nodes in set $\{V_i\}$ and terminating at v_i. So the total number of node v_i who are pushed into the queue $enq(v_i)$ is no more than the summation of its starting nodes pushed into the queue. We can show the relationship using this formula $enq(v_i) \leq \sum_{v \in V_i} enq(v) = \epsilon * enq(v)$. And for each starting node of v_i, the situations are same. So, using induction rules, we can get $enq(v_i) = \epsilon^{i-1}$, in which i is the layer number of node v_i if we assume ϵ nodes are in the same layer. Then we can get the total numbers of all nodes pushed in to the queue show in below

$\sum_{v \in V} enq(v) \leq \epsilon * (1 + \epsilon + \epsilon^2 + ... + \epsilon^{\lceil \frac{|V|^2}{|E|} \rceil - 1}) = \epsilon * \frac{\epsilon^{\lceil \frac{|V|^2}{|E|} \rceil} - 1}{\epsilon - 1}$

where $\epsilon = \frac{|E|}{|V|}$ is the average links number of each node. Thus, the average number of each node pushed in to the queue is $\frac{\epsilon}{|V|} * \frac{\epsilon^{\lceil \frac{|V|^2}{|E|} \rceil} - 1}{\epsilon - 1}$ and the time complexity is $O(\frac{\epsilon}{|V|} * \frac{\epsilon^{\lceil \frac{|V|^2}{|E|} \rceil} - 1}{\epsilon - 1} * (|E| + |V|))$. For some special cases, if $|E| \approx |V|$, the time complexity becomes to $O(|E| + |V|)$. If $|E| \gg |V|$ and $|E| \approx |V|^2$, the time complexity becomes to $O(|V| * (|V| + |E|) = O(|V| * |E|))$.

4 Implementation

Actually, as the pseudo codes show, the two algorithms are very similar except for the queue data structure. In Dijkstra we need priority queue, and in SPFA we only need a FIFO queue.

4.1 Dijkstra

In the experiment, we implemented two versions of Dijkstra, one is normal binary heap, and the other one is pairing heap. As we all know that fibonacci heap is the best implementation of Dijkstra, but considering of the complexity and understandability, we finally chose pairing heap.

The following table shows the time complexity of basic heap operations for three versions of heap.

Operation	Binary Heap	Pairing Heap	Fibonacci Heap
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(logn)$	$\Theta(logn)$	$\Theta(logn)$
insert	$\Theta(logn)$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(logn)$	$\Theta(logn)$	$\Theta(1)$
merge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Table 1: Time Complexities

From the table we know that pairing heap has the same time complexity for most heap operations as fibonacci heap, and it's easy to understand and implement.

4.2 SPFA

As for SPFA, we just need a simple First-In-First-Out queue.

5 Experiment Results

We ran these two algorithms on three different datasets.

5.1 Dataset1

In Figure 1, x-axis means the number of nodes in graph, the green line is SPFA's result, orange line is pairing heap Dijkstra and blue one is binary heap Dijkstra. As the graph shows, the SPFA's result is not better than pairing heap version but it's quite good when compared with binary heap version.

Therefore, we know that SPFA performs quite well when number of nodes is not too large.

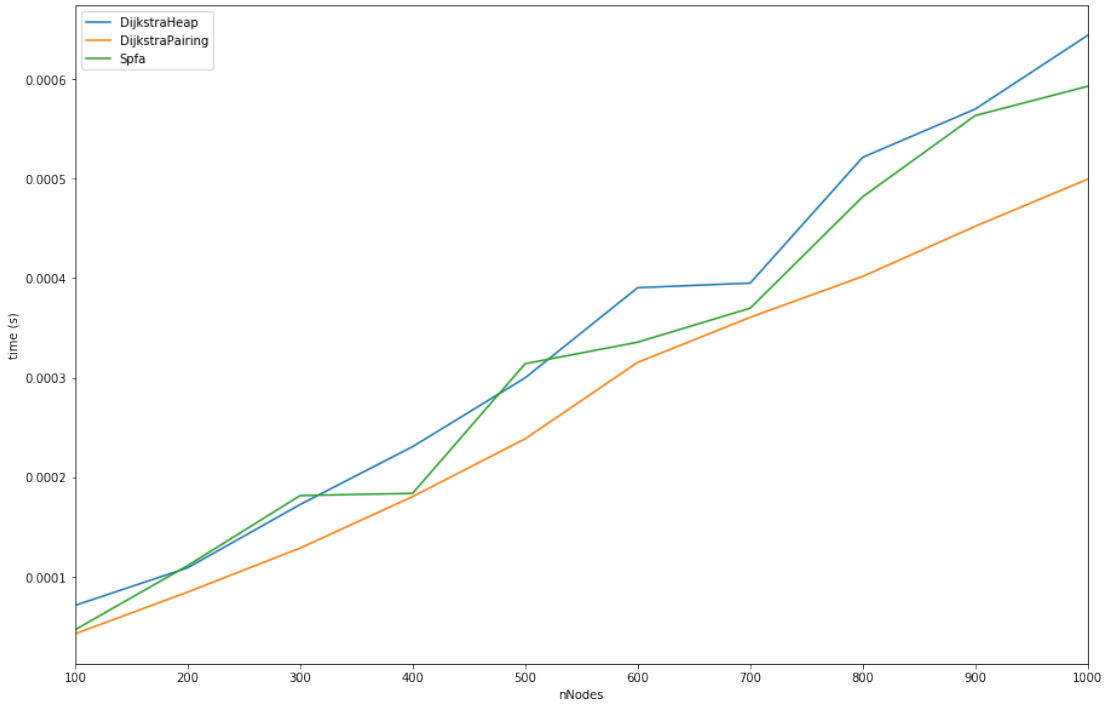

Figure 1: Running Time for Different #Nodes

5.2 Dataset2

Figure 2 is the result for another dataset which each graph has 1000 nodes but average node degree varies from 1 to 30.

When average node degree is not too large, then SPFA performs better than binary heap version Dijkstra, but not better than pairing heap version. However, when node degree is very large, SPFA becomes worst one among all of these algorithms.

In fact, larger node degree means in average each node will have more edge connected to that node, Dijkstra will not care about that because it's a greedy algorithm, it only choose one edge and expand the optimal node in current step, so as the degree grows, running time of Dijkstra will not change much. In contrast, SPFA doesn't have a priority queue to determine which node is optimal in current step, so it will try to expand all possible nodes through all possible edges. That will cause the node dequeued from the queue is not optimal, it needs to be enqueued in the future to relax it's distance, therefore the same node can be enqueued several times, here the cost of repeatly enqueueing is larger than the cost of maintaining a priority queue.

5.3 Real Dataset

We also ran the two algorithms for real dataset, Figure 3 is USA road dataset.

Unfortunately, SPFA needs a lot of time to run, it can not handle the complicated graph. However, Dijkstra's performance is perfect.

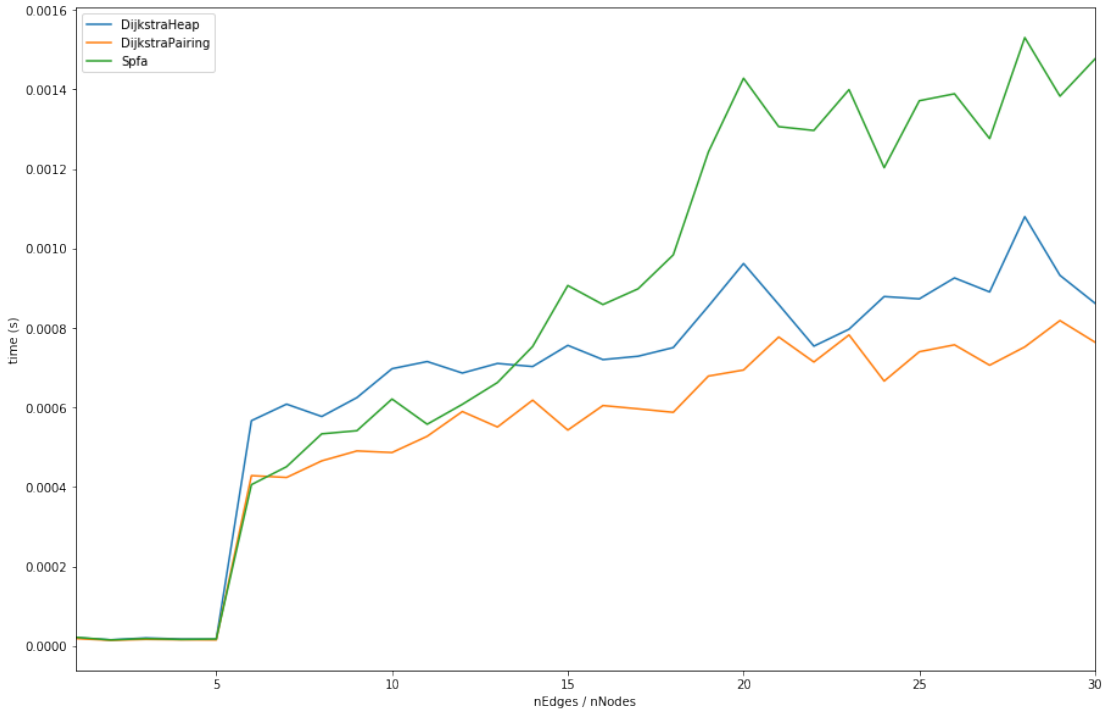

Figure 2: Running Time for Different Degrees

6 Conclusions

From the experiment result, we can obtain some useful experiences.

First, Dijkstra(with fibonacci heap) is exactly the best algorithm for single source shortest path problem, either Bellman-Ford or SPFA can not beat it when graph is very complicated.

However, we have to admit that when input graph is small, and when the node degree is small, then SPFA's performance is quite well, at least not bad than binary heap version Dijkstra. And another advantage of SPFA is it is extremely easy for us to implement, we do not need a priority queue, we do not need to consider how to implement a priority queue. A simple solution can handle many common cases well so we like it.

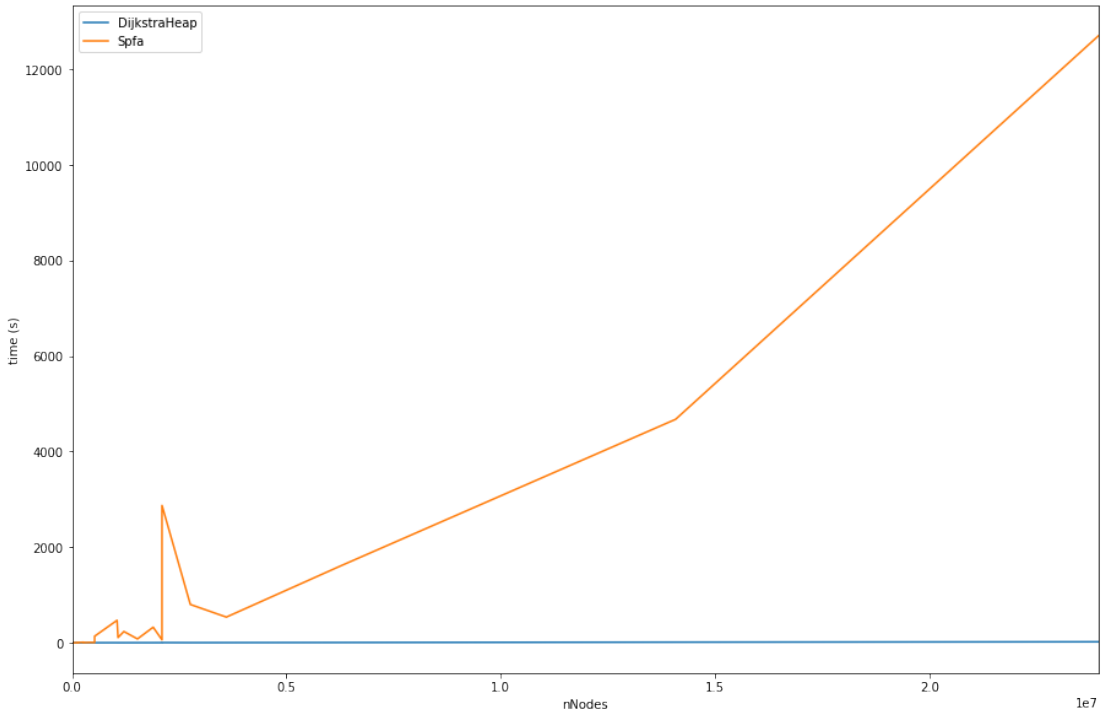

Figure 3: Running Time for Real Datasets

References

[1] Shortest path - pegwiki. `http://wcipeg.com/wiki/Shortest_path#Relaxation`. (Accessed on 10/23/2017).

[2] Shortest path faster algorithm - pegwiki. `http://wcipeg.com/wiki/Shortest_Path_Faster_Algorithm`. (Accessed on 10/23/2017).

[3] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[4] Thomas H Cormen. *Introduction to Algorithms*. MIT Press, 2009.

[5] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[6] Wikipedia. Shortest path faster algorithm — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Shortest_Path_Faster_Algorithm&oldid=801345364`, 2017. [Online; accessed 23-October-2017].

[7] Wikipedia. Shortest path problem — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Shortest_path_problem&oldid=806052520`, 2017. [Online; accessed 23-October-2017].