Single Source Shortest Path Algorithms Comparison: Dijkstra and Bellman Ford

Shikun Wang (Jason) Xinran Gu

November 29, 2017

Abstract

In this project, we are going to implement a basic directed weighted graph API with Python and use this API to implement Dijkstra and Bellman Ford algorithm for single source shortest path problem. Then, we are going to use networkx package in Python to first verify the correctness of our algorithms and experiment on comparing performances between our implementations and networkx's. We will also test our implementation on graphs with negative edge weights, comparing the results between Dijkstra and Bellman Ford.

1 Introduction

The problem of shortest path is that we need to find a path from a source to a vertex in a graph so that the sum of the edge weights along the path is minimized. There are two basic algorithms to solve this problem. The first one is Dijkstra, whose running time is $O(|E| + |V|l \log |V|)$, but it might be wrong for a graph with edges that have negative weights. The other algorithm, Bellman-Ford, can solve the problem containing negative edges, but has a slower run time of $O(|V||E|)$.

2 Graph API

We implemented a Graph API with Python by using Python dictionary for mapping nodes to their adjacent lists, Python list structure for storing nodes, and Python list for storing edges . We implemented the addNode(node) function which adds a given node to the graph if it does not exist. And We implemented the addWeightedEdge(source, target, weight), which adds a weighted directed edge from source to target by adding a tuple (target, weight) to the source's adjacent list. The API interface is shown below:

$Attributes$:

bag of nodes; adjacent lists; bag of edges

$Methods$:

1. Adding node to the graph:

$addNode(node)$

2. Add edge (source, target) with weight = weight

$addWeightedEdge(source, target, weight)$

3. Get a node given its ID

$getNodebyID(id)$

4. toString method for debugging

$toString()$

3 Dijkstra Algorithm

3.1 Description

The intuition of Dijkstra algorithm is greedy algorithm, picking the least weights amount all the neighbors of current node, then update. It has two sets of nodes, one set of the nodes already included in the shortest path (S), and the other set with the nodes not yet included(X). Starting from a root node added into set S, it picks the least weight edge, adds the node (u) to the set, then updates all adjacent nodes of u with the shortest distances from root s to the neighbors.
Pseudo-code using priority queue:

```
Dijkstra():;
Q ← MakePQ();
insert(Q,(s,0));
for each node u ≠ s: do
  | insert(Q,(u,infinity));
end
X← ∅;
for i=1 to |V| do
  | (v,dist(s,v))=extractMin(Q);
  | X=X∪{v};
  | for each u in Adj(v) do
  |   | decreaseKey(Q,(u,min(dist(u,v),dist(s,u)+l(u,v))))
  | end
end
```

3.2 Implementation

We use priority queue implemented by heap in Python, "heapq" to do the extractMin and decreaseKey operation. When we push item into the priority queue, if the item already exist, it will update the value of that existing item. Therefore, the descreaseKey operation is simply pushing modified item back to the queue. We use a dictionary structure to keep track of the shortest path tree by mapping each node to its parent. We update the path whenever an edge is relaxed. Lastly, We use a visited list to keep track of visited nodes which should not be visited again. Since we did not use Fibonacci Heap, our running time is not optimal which is $O(|E| \log |V| + |V| \log |V|)$.

4 Bellman Ford Algorithm

4.1 Description

The Bellman Ford Algorithm can work on graph with negative edges. It keeps tracks of all distances using an array.It initializes all distances from source to vertex to infinity. It then computes the shortest distances for all vertexes and its edges (u,v) by taking the minimum of the distance to v and the distance to u plus the edge weight (u,v). After |V| -1 iterations, if there is still a edge that has not been relaxed, we have a negative cycle.
pseudo-code:

```
Bellman-Ford(): ;
for each u ∈ V do
 | d(u) ← ∞
end
d(s) ← 0;
for k=1 to n-1 do
 | for each edge (u,v) ∈ Edges do
 |  | d(v)=min(d(v),d(u)+l(u,v))
 | end
end
for each v ∈ V do
 | for each edge (u,v) ∈ Edges do
 |  | if d(v)>d(u)+l(u,v) then
 |  |  | Output "Negative Cycle"
 |  | end
 | end
end
for each v ∈ V do
 | dist(s,v) ← d(v)
end
```

4.2 Implementation

We used dictionary to keep track of the distance to nodes and the shortest path tree. All updating actions are using basic dictionary operations. All nodes in the graph are stored in dictionary called dist with initial value set to infinity. Then iterate $|V| - 1$ times to relax all the edges. When the shortest path is found, it will be stored into the shortest path tree dictionary as the path to that node. Negative cycle detection is implemented by another iteration, so that if the edge is still not relaxed upon $|V| - 1$ times, there is a negative cycle.

5 Correctness Verification

5.1 Dijkstra

We randomly generated 100 graphs with random number of edges and nodes with range (0, 100) and weights with range (0, 100) by networkx, and we created our own graph with my API based own each generated graph. Next, we feed networkx's graphs to its Dijkstra implementation and feed our graphs to our Dijkstra. Lastly, we compared the results of both implementation, which turns out to be the same. Therefore, our implementation of Dijkstra is correct.

5.2 Bellman Ford

And then, we feed networkx's graphs to its Bellman-Ford implementation and feed our graphs to our Bellman-Ford. We compared the results of both implementation, which turns out to be the same. Therefore, our implementation of Bellman-Ford is correct.

6 Graph with Negative Edge Weights

Next, we want to study the effect of negative edge weights, which should cause some problems for our Dijkstra. First, we generated 10 graphs with negative edge weights range (-100, 100). Then we run our own Dijkstra and Bellman-Ford on these graphs. It turns out that Bellman-Ford is able to detects the existence of a negative cycle, but Dijkstra would some times produces wrong answers. Therefore, we can see that Dijkstra might not be able to come up with a right answer when negative.

7 Performance Comparison

We measure the running time of my Dijkstra, my Bellman-Ford, Networkx's Dijsktra and Networkx's Bellman-Ford in executing 100 positive directed weighted edge graphs with number of nodes from 1 to 1000 with step as 20 and the number of edges is the same as number of nodes. Then, we plotted three graphs, respectively between my Dijkstra and Networkx's Dijkstra(fig 1), my Bellman-Ford and Networkx's Bellman Ford (fig 2) and my Dijkstra and my Bellman Ford(fig 3). And in the end, we also combine all the graphs together to have a better visualization (fig 4).

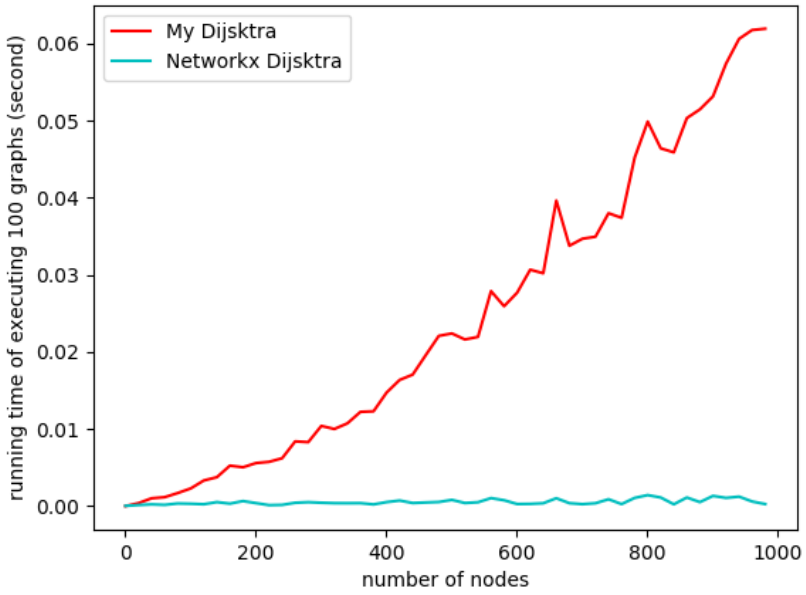

Figure 1: running time comparison between networkx's Dijkstra and my Dijkstra
As you can see, our Dijkstra is roughly three times slower than networkx's in average;

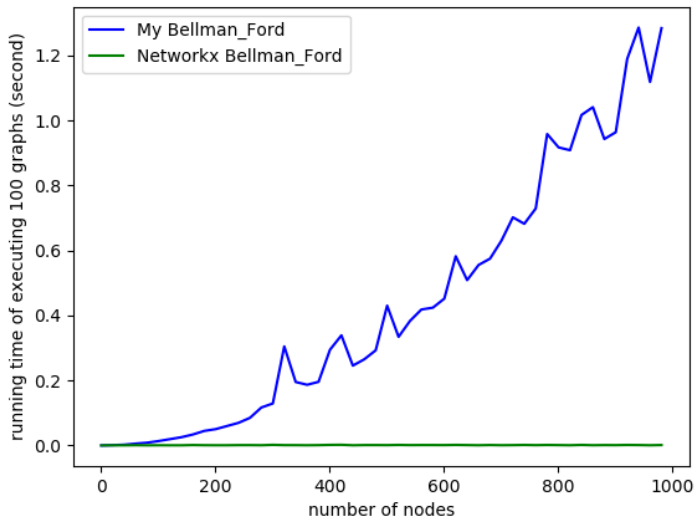

Figure 2: running time comparison between networkx's Bellman-Ford and my Bellman-Ford
Our BellmanFord is roughly six times slower than networkx's in average;

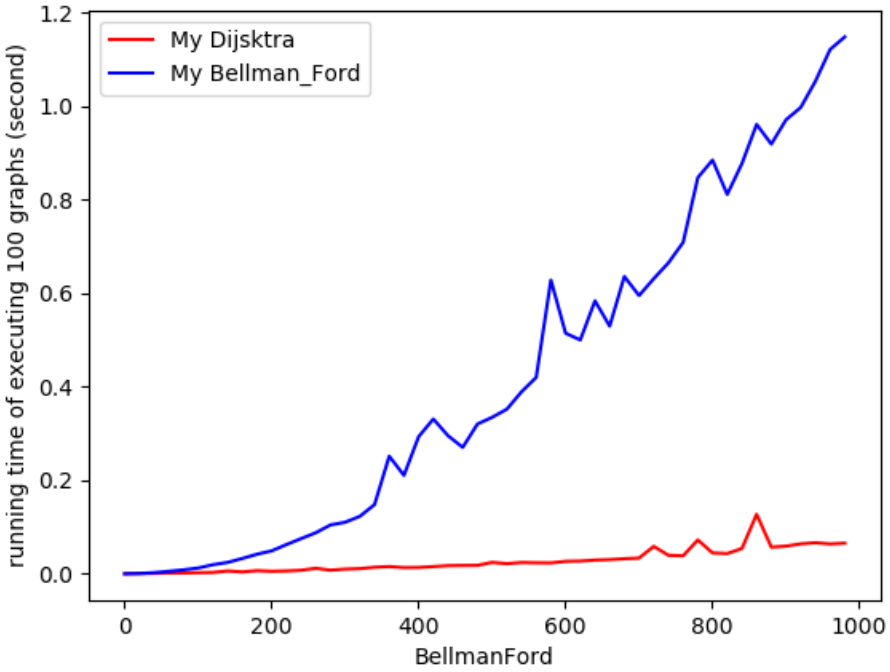

Figure 3: running time comparison between my Dijkstra and my Bellman-Ford Our Bellman-Ford is roughly six times slower than our Dijkstra in average.

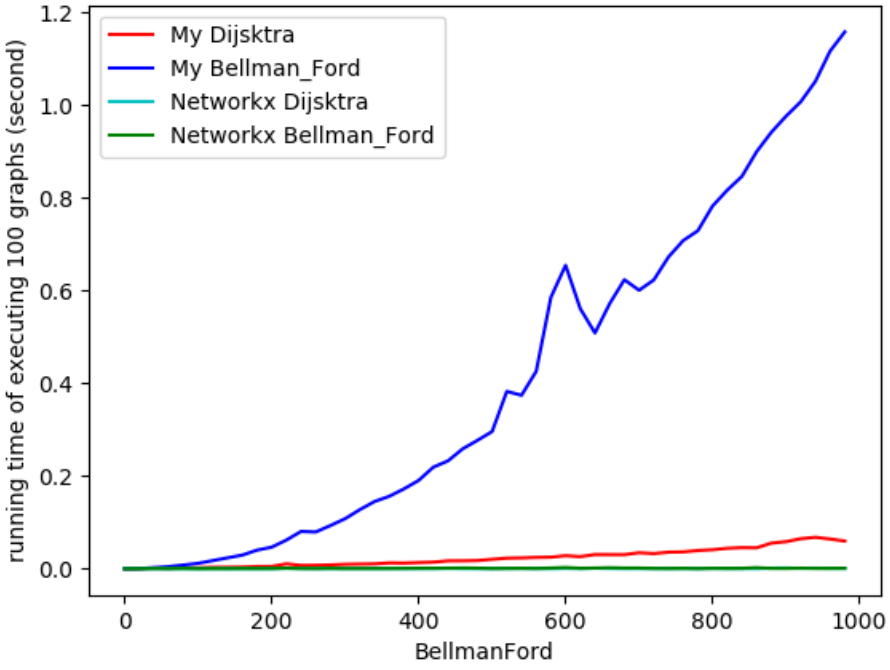

Figure 4: running time comparison between my Dijkstra and my Bellman-Ford My Bellman-Ford does not perform well compared to other algorithms.