Exploring the World, Through Graphs!

An In-Depth Analysis of Bellman-Ford and Dijkstra’s Algorithms

Michael Stewart

Colorado State University Global

CSC506-1: Design and Analysis of Algorithms

Dr. Isaac K. Gang

14 December 2023

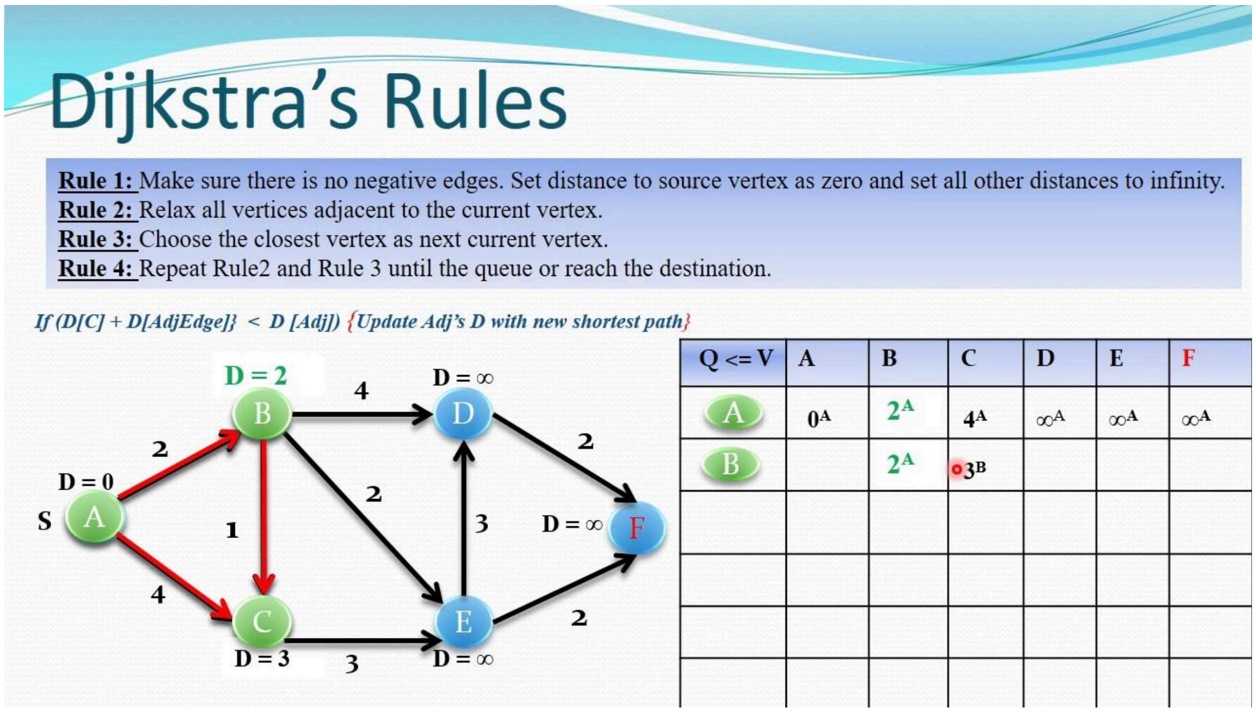
Exploring the World, Through Graphs! An In-Depth Analysis of Bellman-Ford and Dijkstra's Algorithms

Learning about algorithms and their differences is a valuable part of the toolbelt for a software engineer. With more algorithms learned and analyzed, there are more problems that we can solve and analyze. Hopefully, there is also a mindset and ability to learn which comes with learning multiple complex algorithms and using them multiple times. To pursue this sharpening of our intellectual skills and metaphorical toolbelt, this paper will discuss my impressions, implementation, and analysis of Bellman Ford's and Dijkstra's shortest path algorithms. According to Britannica (n.d.), Edsger Dijkstra developed his search algorithm in 20 minutes while sitting in a cafe! What an interesting date that must have been. This small amount of time in a coffee shop has led to these algorithms being used in many applications like communication, flight planning, and network infrastructure.

Now that the algorithms have been introduced, we will get into the nitty-gritty. Dijkstra's algorithm goal is to find the shortest path between nodes within a weighted graph. It is also known as a shortest-path algorithm (Geeks for Geeks, 2022). Similar to this, the Bellman-Ford is also a single path shortest path algorithm. However, Bellman-Ford is guaranteed to find the shortest path. This is because it is more thorough than Dijkstra's algorithm and can use the shortest path even if negative weights exist within the graph. This is because Bellman Ford's algorithm goes through the entire path and checks all paths between each of the two nodes. This avoids the problem where Dijkstra's algorithm is greedy and cannot handle negative edge weights. This means that Bellman-Ford's algorithm is much slower than Dijkstra's to cover all possible paths and handle negative weights.

As mentioned, These algorithms are very useful in industries like flight planning, network infrastructure, and communications. Flight planning uses these algorithms to help figure out what the most efficient and timely route is for planes. Some might say that since there are so many different paths to fly, that straight edges and nodes wouldn't represent this well. However, when you have so many airports and standardized routes of aircraft, you can predict a lot with a weighted graph. Problems like planning flights with layovers, storms, or even normal flights can be solved using solutions that are calculated from versions of these weighted graph algorithms. Another interesting application is network infrastructure.

Dijkstra's Algorithm:



Dijkstra's Rules

Rule 1: Make sure there is no negative edges. Set distance to source vertex as zero and set all other distances to infinity.
Rule 2: Relax all vertices adjacent to the current vertex.
Rule 3: Choose the closest vertex as next current vertex.
Rule 4: Repeat Rule2 and Rule 3 until the queue or reach the destination.

If (D[C] + D[AdjEdge]} < D [Adj]) {Update Adj's D with new shortest path}

Q <= V	A	B	C	D	E	F
A	0^A	2^A	4^A	∞^A	∞^A	∞^A
B		2^A	3^B			

Dijkstra's algorithm from Iyappan C https://www.youtube.com/watch?app=desktop&v=Lfb8qkXzHY0

Dijkstra's algorithm is a great algorithm for finding the shortest path with these specific requirements. This is a good example of how the algorithm works. With this graph and all of its weighted edges, it keeps track of all the nodes visited and the distances to them to understand where the shortest path is. Once this algorithm is done, then we have a list of shortest paths to different nodes. Dijkstra’s algorithm is one that’s considered a “greedy” algorithm. A greedy algorithm tries to find the solution to the problem in the shortest time possible. This differs from normal algorithms because some algorithms (like the Bellman-Ford algorithm) will try to cover all possible cases. Dijkstra differs from this in that once it has found the vertex it was looking for, it will stop. This saves lots of time and processing power but could be bad in specific cases.

General Description:

Dijkstra’s algorithm uses the technique called “Relaxation”. Relaxation is the technique of starting at the source node, assigning all distances to other nodes as infinity, and then slowly ‘relaxing’ the graph by traveling to the lowest edge weight and then decreasing the vertex distance. For example, if we had a very simple graph with 2 nodes and an edge weight between them of 50. The algorithm would start with the first node, and assume the distance is infinity to the second node, or that it is not connected. Then it will check all the possible paths from the starting node and choose the lowest-weight one. Since we only have one path, it will record that there is a weight 50 edge to our goal node. This 50 will replace the infinity and relax the graph's shortest distance array.

Dijkstra has a time complexity of O(ElogV). E represents the number of edges inside the graph and V represents the vertices or nodes within the graph. This makes sense because as there are more edges added to the graph, there are a lot more places for the algorithm to travel. However, with nodes, some nodes may not be connected or have many paths connected to them.

Applications:

We have discussed some applications in real life like flight paths and network infrastructure, but there are so many other cool applications. One of the most interesting ones is social networking. Some social media applications will use Dijkstra’s algorithm to suggest friends to be added to your friends list. Finding the shortest connections between each person will help show which people are closely connected and then the platform can have a better chance of creating a network of friends (Geeks for Geeks, 2022).

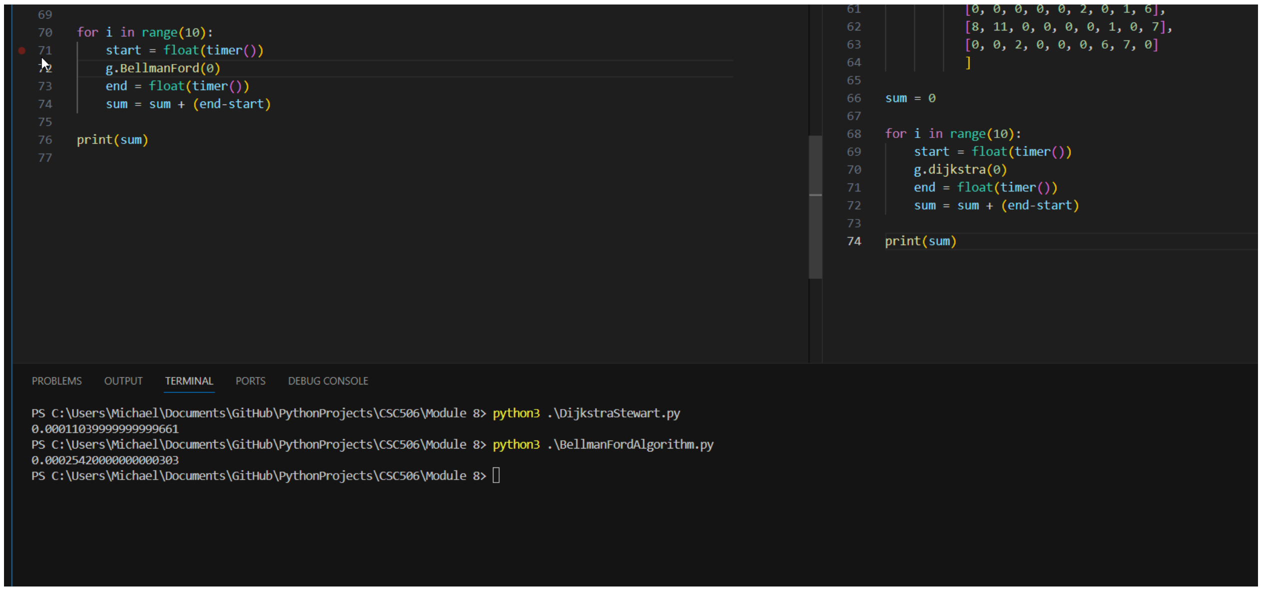
Design and Implementation:

I based the design and implementation of my algorithm on the method from Divyanshu Mehta and Updated by Pranav Singh Sambyal from Geeks for Geeks. I liked their representation of the graph using an adjacency matrix and the ability to print all vertices and paths. I thought this would be helpful for my testing so that I could test the amount of overhead that is saved from using Dijkstra's algorithm rather than the Bellman-Ford.

I decided to have Dijkstra's algorithm run ten times and find the minimum path to each vertex within this example graph. There are 9 nodes and 15 edges between them which gives enough complexity for our testing purposes. I used the timer method from Timeit to test this algorithm as well. I found that this testing method provided reliable and consistent results.

If I were to redo this testing design over again, I would try adding some testing to find one specific vertex. This testing method will test Dijkstra's speed for each vertex in the graph, but if we did a random vertex in the middle of the graph for the target, our time results compared to the Bellman-Ford might be more drastic.

Analysis:



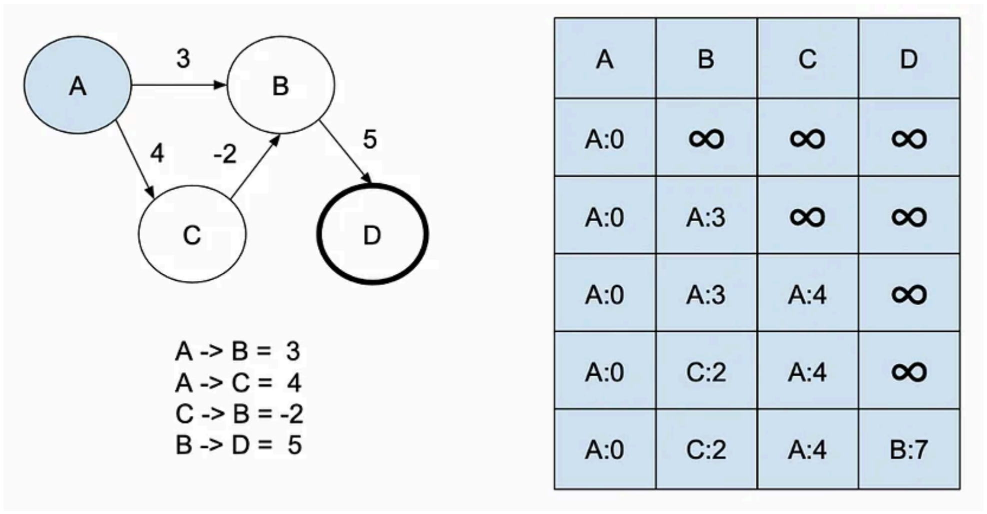
```
for i in range(10):
    start = float(timer())
    g.BellmanFord(0)
    end = float(timer())
    sum = sum + (end-start)

print(sum)
```

```
            [0, 0, 0, 0, 0, 2, 0, 1, 6],
            [8, 11, 0, 0, 0, 0, 1, 0, 7],
            [0, 0, 2, 0, 0, 0, 6, 7, 0]
            ]

sum = 0

for i in range(10):
    start = float(timer())
    g.dijkstra(0)
    end = float(timer())
    sum = sum + (end-start)

print(sum)
```

```
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 8> python3 .\DijkstraStewart.py
0.00011039999999999661
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 8> python3 .\BellmanFordAlgorithm.py
0.00025420000000000303
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 8> 
```

Using this testing method, as you can see for Dijkstra's algorithm the results we got were: 0.110ms for 10 searches through the whole graph. This test and time accumulate data for the algorithm searching for all 9 nodes 10 different times. This has led to very good consistent results for the time searching. Now that we have an understanding of Dijkstra's algorithm, we can compare this with a new understanding of Bellman-Ford's algorithm.

Bellman-Ford's Algorithm:



A -> B = 3
A -> C = 4
C -> B = -2
B -> D = 5

A	B	C	D
A:0	∞	∞	∞
A:0	A:3	∞	∞
A:0	A:3	A:4	∞
A:0	C:2	A:4	∞
A:0	C:2	A:4	B:7

Weighted Graph with Negative Edges https://levelup.gitconnected.com/bellman-ford-6bd907c6c4c0

Bellman-Ford's algorithm is very close to Dijkstra's algorithm with a couple of changes in order to cover negative weighted edges. This helps to cover situations where certain edges counteract each other. This is very helpful in situations where programmers would like to influence traffic to go through one node or another.

General Description:

For a basic description of this algorithm, the process of "Relaxation" is still used in Bellman-Ford but it will explore the whole graph to search for one vertex. This is because if there are negative weights within the graph, it is possible that you will not find the shortest path the first time you find the vertex. Because of this, Bellman-Ford is usually slower than Dijkstra's algorithm.

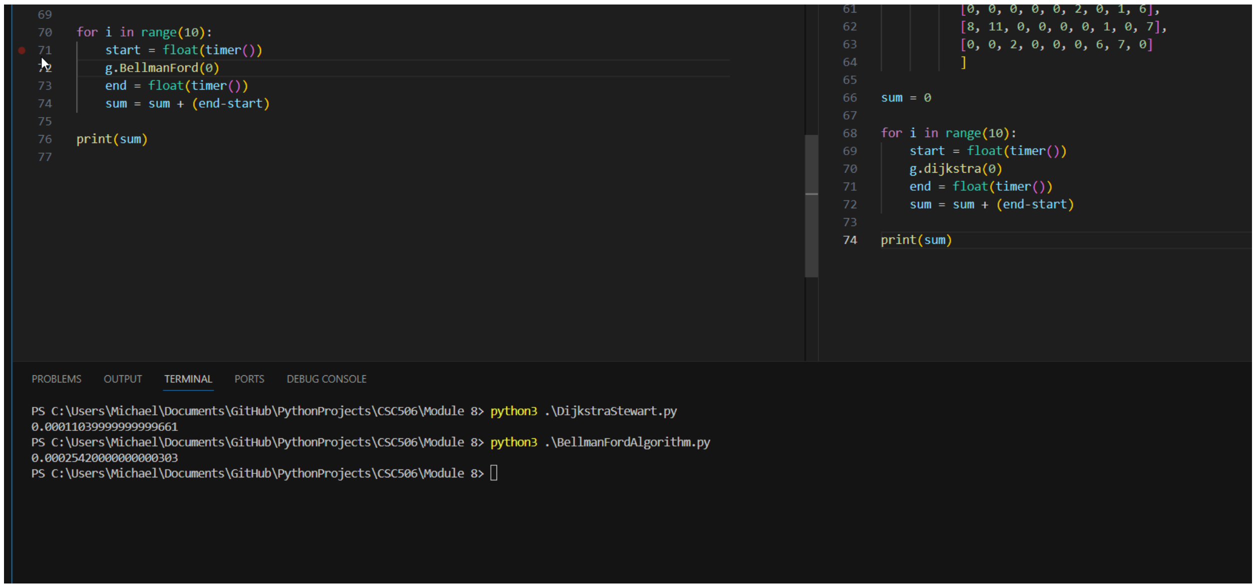
Applications:

Bellman-Ford's algorithm is very useful in real-life scenarios that could use negative weight edges. One common industry is one mentioned before, cities and roads. One example is that if you are driving from Colorado to Los Angeles. If you travel from Colorado to Los Angeles and would like to find the shortest path that you would use the least gas, a negative weight road could be a road that is downhill where you travel a distance while not consuming any gas.

Design and Implementation:

My Bellman-Ford algorithm is also based on a lesson from Geeks for Geeks from Neelam Yadav and Himanshu Garg (Geeks for Geeks, 2023d). I liked this implementation and based my code on this because it used a method that adds edges using the nodes and weight rather than an adjacency matrix previously. It also seemed very simple and searched for each vertex just like our Dijkstra's algorithm. I edited and added a baseline for our test to be fair and easy to use.

Analysis:



```
for i in range(10):
    start = float(timer())
    g.BellmanFord(0)
    end = float(timer())
    sum = sum + (end-start)

print(sum)
```

```
                [0, 0, 0, 0, 0, 2, 0, 1, 6],
                [8, 11, 0, 0, 0, 0, 1, 0, 7],
                [0, 0, 2, 0, 0, 0, 6, 7, 0]
                ]

sum = 0

for i in range(10):
    start = float(timer())
    g.dijkstra(0)
    end = float(timer())
    sum = sum + (end-start)

print(sum)
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 8> python3 .\DijkstraStewart.py
0.00011039999999999661
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 8> python3 .\BellmanFordAlgorithm.py
0.00025420000000000303
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 8> 
```

When testing, I got the results for the Bellman-Ford's algorithm to be 0.254ms for 10 searches through the whole graph. This is more than double the amount of time that Dijkstra's algorithm got with about 0.11ms. This is because of the large overhead that Bellman-Ford has compared to Dijkstra's.

Comparison:

Introduction:

I was able to compare these two algorithms' ability to search graphs and benchmark their results. Our hypothesis that Dijkstra's is faster compared to Bellman-Ford's seems to have been confirmed with our initial testing.

Results Analysis:

As previously stated in the other sections, we got Dijkstra's: 0.110ms for 10 searches through the whole graph, and Bellman-Ford's time was 0.254ms for 10 searches through the whole graph. This is because our test would go through the whole graph for each vertex, and each search, Dijkstra's algorithm would stop and Bellman-Ford would keep going.

Lessons Learned:

Some of the lessons I learned through this process are the basics of the algorithms, the nuances of representing graphs, and the complexities of choosing algorithms for specific applications in the engineering world. With this new information and knowledge, I can make more informed inferences about existing systems and how new systems should be designed.

Future Goals and Research:

A couple of improvements I would make to this testing is that I would expand the regions of testing for each algorithm. I would expand the example graphs to have more complex graphs and also test Bellman-Ford's ability to detect negative weight. This would be an interesting part of the process because it was not explored here and we might have interesting results while analyzing negative edge weights.

Conclusion:

Comparing Dijkstra's and Bellman-Ford's algorithms has been a fulfilling and interesting experience. We were able to define the different algorithms, give industries they are useful in, and apply them. Our testing of these algorithms was also very productive and I improved the way that I tested different algorithms in Python as well. We concluded that Dijkstra's algorithm in our implementation was much faster than Bellman-Ford's when we were using graphs that both algorithms could search. Things definitely would have changed if we used less complex or more complex graphs, but the actual changes are hard to know. A future effort could be made to explore the complexities of different graphs and their effect on each of these algorithms' speeds.

Key Findings:

We have confirmed our hypothesis and benchmarked algorithms that are very similar with slightly different capabilities. Now in future projects for my engineering career, I will have a lot more background knowledge on these different algorithms as well as the intuition to know that if I am choosing between different algorithms, there is always a give and take.

References

Encyclopædia Britannica, inc. (n.d.). *Edsger Dijkstra*. Encyclopædia Britannica. https://www.britannica.com/biography/Edsger-Dijkstra

GeeksforGeeks. (2022, June 23). *What are the differences between bellman Ford's and Dijkstra's algorithms?* GeeksforGeeks. https://www.geeksforgeeks.org/what-are-the-differences-between-bellman-fords-and-dijkstras-algorithms/

GeeksforGeeks. (2023d, December 6). *Bellman–Ford algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

Lysecky, R. (2019a). Chapter 1. In F. Vahid (Ed.), *Design and Analysis of Algorithms*. essay, Zyante Inc.

Navone, E. C. (2022, February 3). *Dijkstra's shortest path algorithm - a detailed and visual introduction*. freeCodeCamp.org. https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/#:~:text=Dijkstra%27s%20Algorithm%20finds%20the%20shortest,node%20and%20all%20other%20nodes

Wikimedia Foundation. (2023, December 6). *Bellman–Ford algorithm*. Wikipedia. https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm