Bellman-Ford Algorithm

Aaron Frye, Ben Lenox

November 2024

1 Overview

The Bellman-Ford algorithm is a method of computing the shortest path from a given source node to each other node in a graph. It is similar to Dijkstra's algorithm in it's function, but has the advantage of being able to handle graphs with negative edge weights. The algorithm can fail to give a correct answer when the graph has negative cycles, as when these are present there will be nodes with no finite shortest path from s.

2 Pseudocode

Algorithm 1 Bellman-Ford algorithm

Inputs: A weighted digraph with negative weights $G(V, E)$, and a start node $s \in V$.

Outputs: Whether or not the graph contains a negative cycle. Modifies the graph so that each node contains it's shortest distance from the start node.

```
for n ∈ G do
    if n == s then
        n.cost ← 0
    else
        n.cost ← ∞
    end if
end for

for i ← 0 to |V| − 1 do
    for e(u, v) ∈ E do
        if v.cost < u.cost + e.weight then
            v.cost ← u.cost + e.weight
        end if
    end for
end for

for e(u, v) ∈ E do
    if v.cost < u.cost + e.weight then
        return False
    end if
end for
s
return True
```

3 Data Structures

Our implementation uses a networkx graph as it's only data structure, but many of it's functionality could be easily represented with other more fundamental structures.

Computing the cost of reaching each node could be implemented as a hash map, which would allow constant time lookup and updating. Similarly, we could store the edges as (*source*, *destination*) pairs in a simple array, and use a hash map to store edge weights.

4 Intuition of Correctness

The Bellman-Ford Algorithm relies on the fact that for a graph with no negative weight cycles the cheapest path between two nodes will itself contain no cycles. This means that the longest path between two nodes will have at most $|V| - 1$ edges, so if for a given node we find the cheapest path of length 0 to $|V| - 1$ we will find the overall cheapest path to that node.

Round k of relaxing edges can be thought of as computing the cheapest path of length $k + 1$ from the source to each other node. For a simplified proof, imagine when $k = 0$ we compute the cheapest path from the start node to each adjacent node. Then assuming that when $k = n$ each node stores the cost of reaching it by a path with up to n edges, we consider all of the ways that we can add one more edge to these costs. Once we update each of the nodes affected we will have stored the optimal cost of reaching each node with up to $n + 1$ edges.

5 Time Complexity

Our algorithm has three main phases. In the first we loop through each node once and initialize it's distance from the start node. The initialization for each node is $\Theta(1)$, so this phase takes $\Theta(|V|)$ time.

In the second phase we iteratively refine our distance estimates for each node. We do this by performing the edge relaxation for each edge a total of $|V| - 1$ times. Since relaxing an edge takes $\Theta(1)$ time this phase will run in $\Theta(|V| \cdot |E|)$ time.

Finally we check if our graph has any negative edges by running our relaxation procedure one more time and seeing if it improves any of the distances. This phase runs in $\Theta(E)$ time.

Overall this means our algorithm will run in $\Theta(|V| \cdot |E|)$ time.

6 Space Complexity

Taking the input graph as given our algorithm only needs to store the cost of reaching each node, so our space complexity will be $\Theta(V)$.

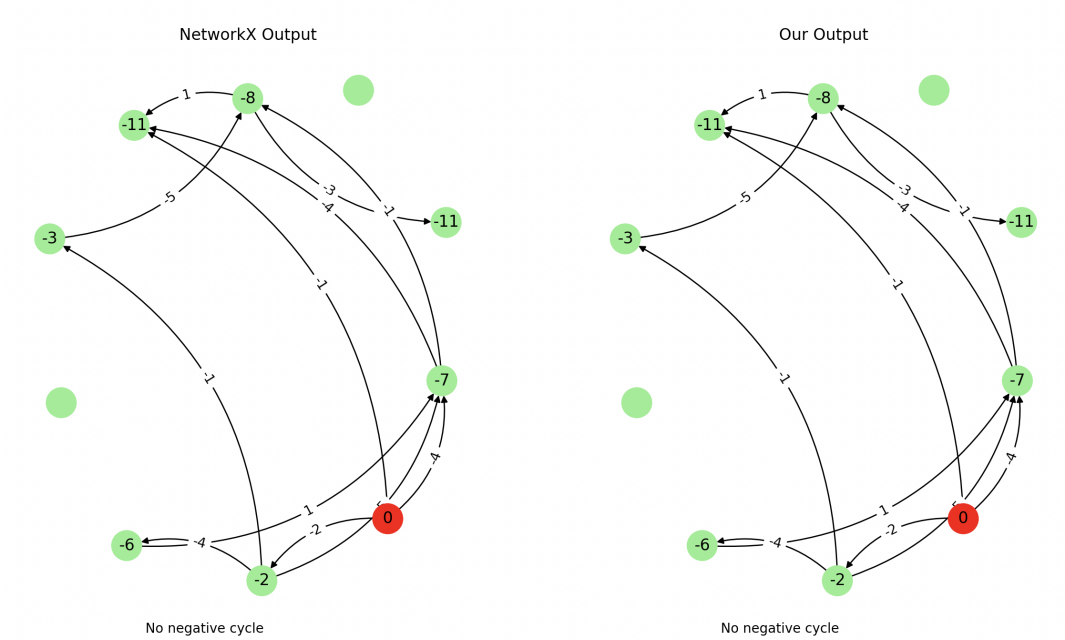
7 Code

```python
# Inputs: A graph to run Bellman-Ford on and a start node
# Outputs: The graph with the lowest cost to reach each node
#          stored at it's nodes
def bellman_ford(G, start):
    # Set cost associated with start node to 0, others to infinity
    for node in G.nodes:
        if node == start:
            G.nodes[node]['cost'] = 0
        else:
            G.nodes[node]['cost'] = math.inf

    # Iterate |V| - 1 times
    for i in range(len(G) - 1):
        # Each time, relax each edge
        for (u, v, data) in G.edges.data():
            edge_weight = data["weight"]
            cost_u = G.nodes[u]["cost"]
            cost_v = G.nodes[v]["cost"]
            # If relaxing an edge yields a better cost update that cost
            if cost_u + edge_weight < cost_v:
                G.nodes[v]['cost'] = cost_u + edge_weight

    # If another relaxation yields better results there is a negative cycle
    for (u, v, data) in G.edges.data():
            if G.nodes[u]["cost"] + data["weight"] < G.nodes[v]["cost"]:
                return 0

    return G
```

8 Results

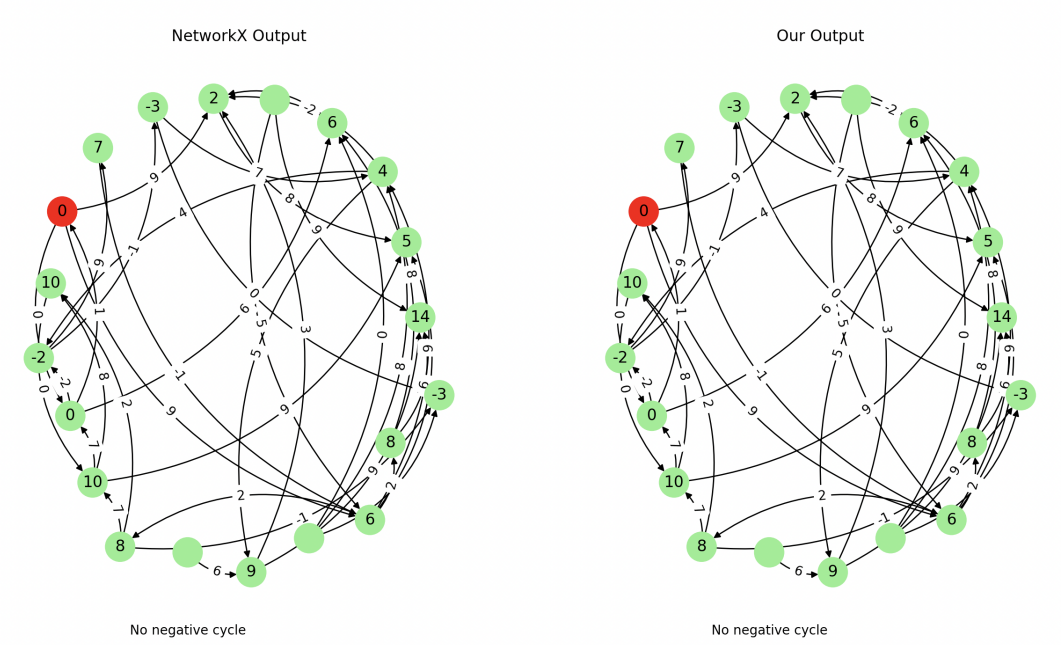
We visually compare the output of our Bellman-Ford implementation with the one provided by NetworkX. In the visualization, each node is labeled with the optimal cost computed by the respective algorithm. Nodes with no labels are nodes that the given algorithm has determined to have infinite cost (unreachable from the start node).

8.1 Small Sparse Graph



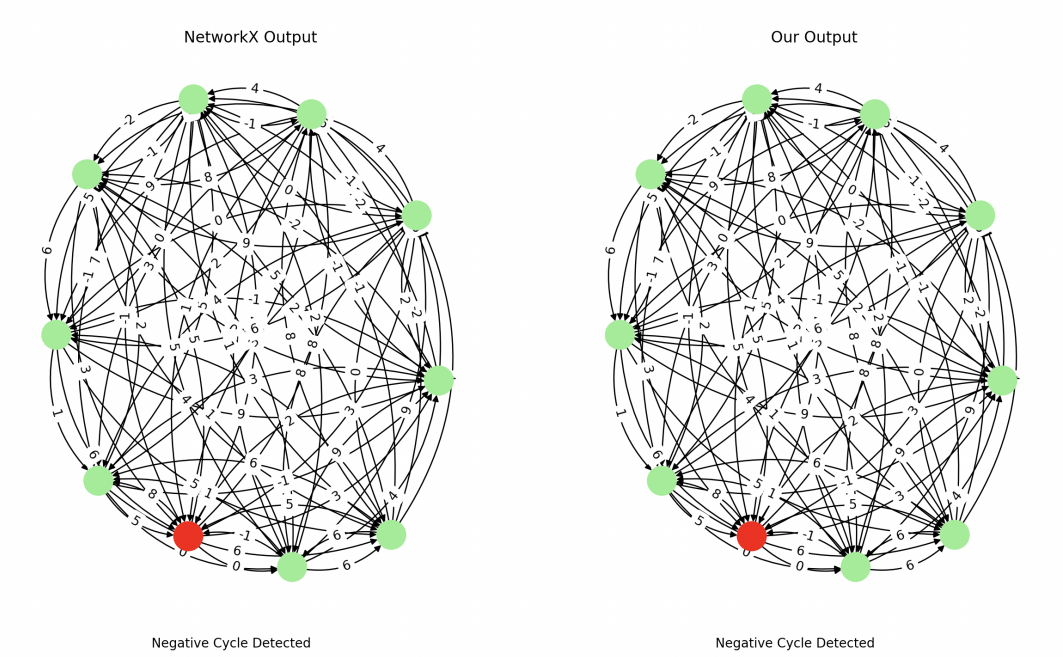
In sparser graphs we expect to see more cases like the graph above, where no negative cycles appear and the algorithm can find optimal paths to each node. In this case we see that our output matches what NetworkX reports. We can also see that our algorithm correctly assigns nodes an infinite weight if they can't be reached from the start.

8.2 Bigger Sparse Graph



This example shows our algorithm working on a larger graph. In this case we can also see that our output matches what we expect based on NetworkX.

8.3 Dense Graphs



As graphs become denser and larger they become increasingly likely to contain negative cycles. This is consistent with the above output, in which both our Bellman-Ford implementation and the one from NetworkX report that the graph contains a negative cycle.

8.4 Tests

In order to test our implementation we generated random inputs and then compared our outputs with NetworkX outputs. Our algorithm consistently provided the same output as NetworkX when tested this way.

The graphs we tested on ranged in size from 10 to 10,000 nodes. Because our algorithm runs in $O(|V| \cdot |E|)$ time, we expect to see runtime on the order of $P(|V|^3)$ for dense graphs. In practice this means that testing large graphs (particularly dense ones) can take a long time, so we ran more tests on smaller graphs than larger ones.

Passing tests on sparse graphs suggest that our algorithm correctly computes the optimal costs when our graph does not contain a negative cycle, while testing on denser graphs suggests that it correctly detects negative cycles.

The following code snippet and output enumerate the specific cases that we tested and how our algorithm performed.

The inputs to the test_helper function are (in order) the number of node for the input graph, a list of "sparsities", and the number of tests to run for each sparsity. Each sparsity is a float representing the probability that any pair of nodes will be connected by an edge.

```
def tests():
    test_helper(10, [0.05, 0.3, 0.8], 100)
    test_helper(100, [0.05, 0.3, 0.8], 50)
    test_helper(1000, [0.02, 0.1], 5)
    test_helper(10000, [0.0001], 1)
```

```
300 / 300 tests pass for graphs with 10 nodes.
150 / 150 tests pass for graphs with 100 nodes.
10 / 10 tests pass for graphs with 1000 nodes.
1 / 1 tests pass for graphs with 10000 nodes.
```