Sample Proof Activity Report

Tess Gompper 260947251

Comp251, Winter 2022

Claim

Negative Cycles in a Graph [KT 301]: There is no negative cycle with a path to t if and only if $OPT(n, v) = OPT(n-1, v)$ for all nodes v.

Proof

This proof is adapted from Kleinberg and Tardos, *Algorithm Design* [4].

Proof. There are 3 components that lead us to our final proof of .

1. *If a graph, G, has no negative cycles, then there is a shortest path from node s to node t that is simple, and hence has at most n-1 edges.*

   Proof. Since every cycle has non-negative cost, the shortest path P from s to t with the fewest number of edges does not repeat any vertex v.If P were to repeat a vertex v, we could simply remove the portion of P between consecutive visits to v, resulting in a path of no greater cost and fewer edges. □

2. If a graph has no negative cycles, part 1 implies that *if there are no negative cycles in G, then OPT(i, v) = OPT(n-1,v) for all nodes v and all $i \geq n$.* This proves the forward direction.

3. For any node v on a negative cycle that has a path to t, we have:

$$\lim_{i \to \infty} OPT(i, v) = -\infty$$

4. To prove the other direction, suppose OPT(n, v) = OPT(n-1, v) for all nodes v. The values of OPT(n+1, v) can be computed from OPT(n, v); but all these values are the equal to the corresponding OPT(n-1,v). Thus we will have OPT(n+1, v) = OPT(n-1, v). Extending this reasoning to future iterations, we see that none of the values will ever change again, that is, OPT(i, v) = OPT(n-1, v) for all nodes v and all $i \geq n$. Therefore there cannot be a negative cycle C, that has a path to t; for any node w on this cycle C, part 3 implies that the values of OPT(i, w) would have to become arbitrarily negative as i increased.

So, there is no negative cycle with a path to t if and only if OPT(n, v) = OPT(n-1, v) for all nodes v.

□

Proof Summary

The cost of a path is the sum of the weight of its edges, and a negative cycle is a path whose cost is negative, thus when finding the optimal path to a node, a negative cycle would cause the cost to become arbitrarily negative. The argument of the proof is that, in the forward direction, there being no negative cycles in the graph implies there is a minimum cost path that uses fewer than n edges because there exists a shortest path. In the reverse direction, supposing there is a minimum cost path that uses fewer than n edges means there cannot be a negative cycle because the existence of a negative cycle would imply a path with arbitrarily negative cost.

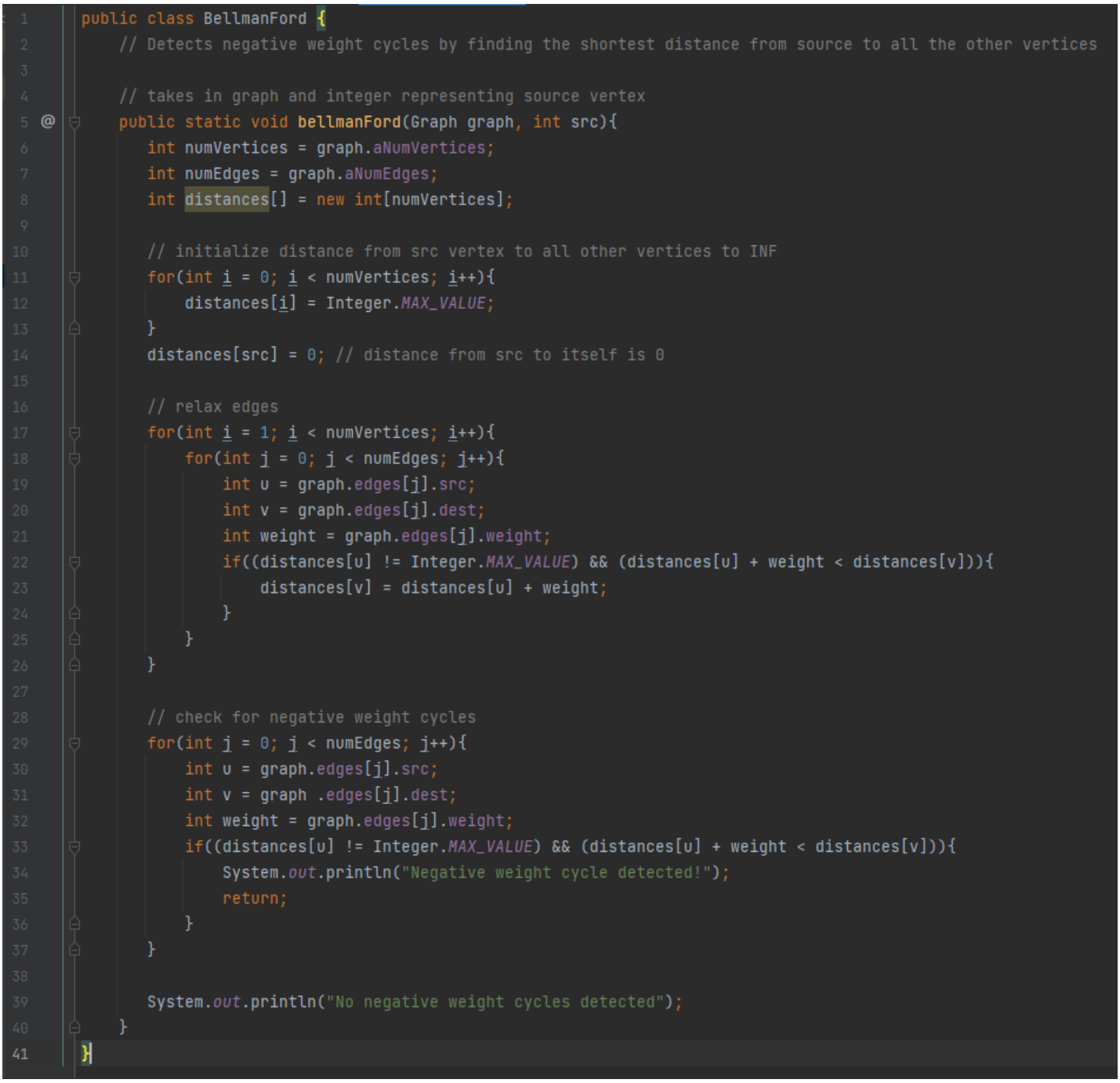
Algorithm

I will use the Bellman-Ford algorithm negative weight cycles in a graph to demonstrate the proof. Below is the Java code for the Bellman-Ford shortest path method inspired by *JavaTPoint*[3] and *GeeksforGeeks*[2].

I created a BellmanFord class (Figure 1) to contain the Bellman-Ford algorithm, a Graph Class (Figure 2), an Edge class (Figure 3) and a Main driver. The Bellman-Ford method prints to the console whether or not at least one negative cycle was found in the given graph.

I ran the code on 2 edge cases and 1 general case. For the general choice I chose a graph with two negative cycles (Figure 4, Figure 5) as when trying to detect negative cycles, the most general case is a graph with negative cycles to be detected. For the first edge case, I chose a graph with 1 negative self loop (Figure 6, Figure 7) as these types of graphs are not as common but still contain negative cycles that must be identified. And, for the second edge case, I chose a graph with no negative cycles (Figure 8, Figure 9) to demonstrate that the algorithm can detect when there are no negative cycles present.

This algorithm demonstrates the proof because when a graph for which $OPT(n, v) = OPT(n-1, v)$ for all nodes v is given, the algorithm accurately reports that there are no negative cycles. And when the given statement does not hold for a graph, the algorithm reports that a negative cycle has been detected.



```java
public class BellmanFord {
    // Detects negative weight cycles by finding the shortest distance from source to all the other vertices

    // takes in graph and integer representing source vertex
    public static void bellmanFord(Graph graph, int src){
        int numVertices = graph.aNumVertices;
        int numEdges = graph.aNumEdges;
        int distances[] = new int[numVertices];

        // initialize distance from src vertex to all other vertices to INF
        for(int i = 0; i < numVertices; i++){
            distances[i] = Integer.MAX_VALUE;
        }
        distances[src] = 0; // distance from src to itself is 0

        // relax edges
        for(int i = 1; i < numVertices; i++){
            for(int j = 0; j < numEdges; j++){
                int u = graph.edges[j].src;
                int v = graph.edges[j].dest;
                int weight = graph.edges[j].weight;
                if((distances[u] != Integer.MAX_VALUE) && (distances[u] + weight < distances[v])){
                    distances[v] = distances[u] + weight;
                }
            }
        }

        // check for negative weight cycles
        for(int j = 0; j < numEdges; j++){
            int u = graph.edges[j].src;
            int v = graph .edges[j].dest;
            int weight = graph.edges[j].weight;
            if((distances[u] != Integer.MAX_VALUE) && (distances[u] + weight < distances[v])){
                System.out.println("Negative weight cycle detected!");
                return;
            }
        }

        System.out.println("No negative weight cycles detected");
    }
}
```

Figure 1: The Bellman-Ford Algorithm from `BellmanFord.java.java`.

```java
public class Edge {
    // represents a weighted edge in a graph
    int src;
    int dest;
    int weight;

    public Edge(int pSrc, int pDest, int pWeight){
        src = pSrc;
        dest = pDest;
        weight = pWeight;
    }
}
```

Figure 2: Edge class from `Edge.java`.

```java
public class Graph {
    // represents a connected, directed, weighted graph

    int aNumVertices;
    int aNumEdges;

    Edge edges[];

    // creates a graph with pNumVertices vertices and pNumEdges edges
    public Graph(int pNumVertices, int pNumEdges, Edge[] pEdges){
        aNumVertices = pNumVertices;
        aNumEdges = pNumEdges;
        edges = pEdges;
    }

}
```

Figure 3: Graph class from `Graph.java`.

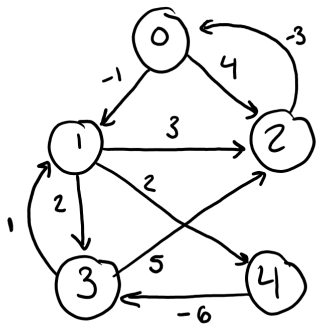


Figure 4: Graph for general test case.

```java
public class Main {
    public static void main(String[] args)
    {
        // general case: a graph with multiple negative cycle (2)
        Edge e1 = new Edge( pSrc: 0, pDest: 1, pWeight: -1);
        Edge e2 = new Edge( pSrc: 0, pDest: 2, pWeight: 4);
        Edge e3 = new Edge( pSrc: 1, pDest: 2, pWeight: 3);
        Edge e4 = new Edge( pSrc: 1, pDest: 3, pWeight: 2);
        Edge e5 = new Edge( pSrc: 1, pDest: 4, pWeight: 2);
        Edge e6 = new Edge( pSrc: 3, pDest: 2, pWeight: 5);
        Edge e7 = new Edge( pSrc: 3, pDest: 1, pWeight: 1);
        Edge e8 = new Edge( pSrc: 4, pDest: 3, pWeight: -6);
        Edge e9 = new Edge( pSrc: 2,  pDest: 0,  pWeight: -3);
        Edge[] es = new Edge[9];
        es[0] = e1; es[1] = e2; es[2]=e3; es[3] = e4; es[4] = e5;
        es[5] = e6; es[6] = e7; es[7]= e8; es[8] = e9;

        Graph graph1 = new Graph( pNumVertices: 5, pNumEdges: 9, es );
        BellmanFord.bellmanFord(graph1, src: 0);
```

```
Run:  Main
C:\Users\tgomp\.jdks\openjdk-17.0.1\bin\java.exe "-javaagent:C:\Progra
Negative weight cycle detected!

Process finished with exit code 0
```

Figure 5: General test case and result from `Main.java`.

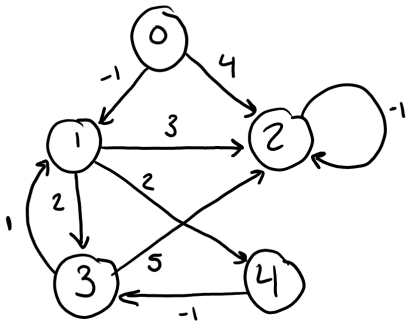


Figure 6: Graph for edge case 1 (negative self loop).

```
21      // edges case 1: a graph with a negative self loop
22      e1 = new Edge( pSrc: 0, pDest: 1, pWeight: -1);
23      e2 = new Edge( pSrc: 0, pDest: 2, pWeight: 4);
24      e3 = new Edge( pSrc: 1, pDest: 2, pWeight: 3);
25      e4 = new Edge( pSrc: 1, pDest: 3, pWeight: 2);
26      e5 = new Edge( pSrc: 1, pDest: 4, pWeight: 2);
27      e6 = new Edge( pSrc: 3, pDest: 2, pWeight: 5);
28      e7 = new Edge( pSrc: 3, pDest: 1, pWeight: 1);
29      e8 = new Edge( pSrc: 4, pDest: 3, pWeight: -1);
30      e9 = new Edge( pSrc: 2, pDest: 2, pWeight: -1);
31      es[0] = e1; es[1] = e2; es[2]=e3; es[3] = e4; es[4] = e5;
32      es[5] = e6; es[6] = e7; es[7]= e8; es[8] = e9;
33      Graph graph2 = new Graph( pNumVertices: 5,  pNumEdges: 9, es);
34      BellmanFord.bellmanFord(graph2,  src: 0);
```

```
Run:  Main
C:\Users\tgomp\.jdks\openjdk-17.0.1\bin\java.exe "-javaagent:C:\Program
Negative weight cycle detected!

Process finished with exit code 0
```

Figure 7: Edge case test 1- negative self loop in graph and result from `Main.java`.

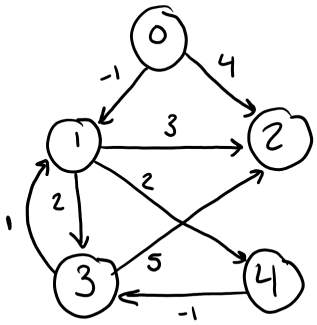


Figure 8: Graph for edge case 2 (no negative cycles).

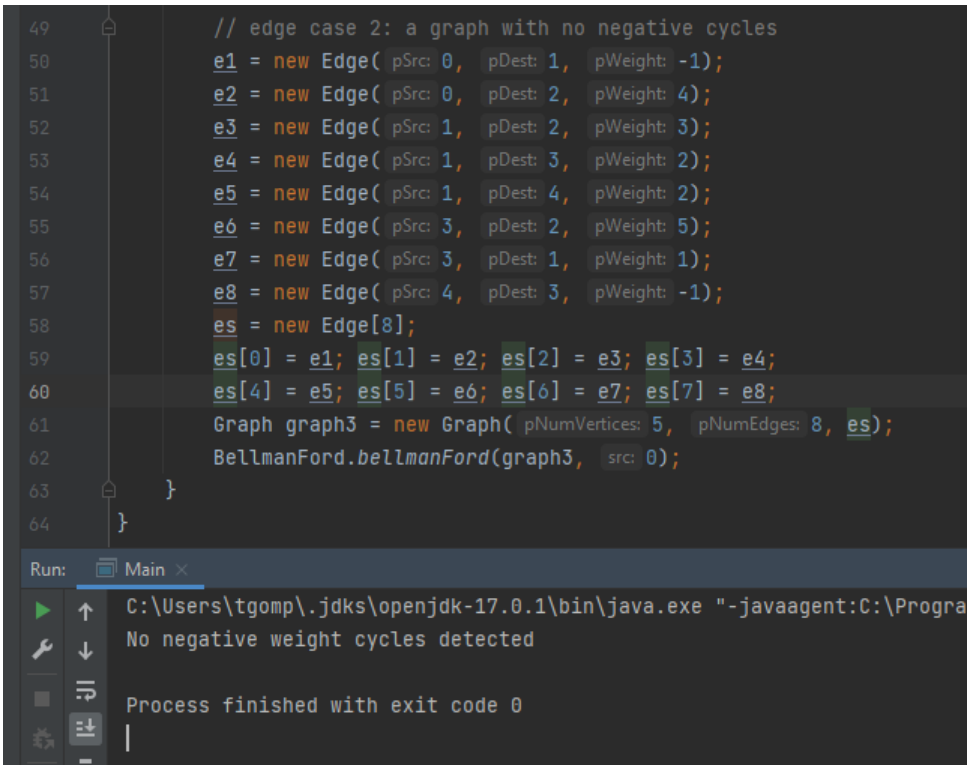


Figure 9: Edge case test 2- no negative cycles and result from `Main.java`.

Real World Application

One application of negative cycles in a graph and the Bellman-Ford algorithm is the Arbitrage problem. An arbitrage is "a way to start with a single unit of some currency and convert it back to more than one unit of that currency through a sequence of exchanges". In a graph, each vertex represents a currency and the weight of the edges represent the exchange rate. Any single positive weight cycle in a graph represents an arbitrage. The Bellman-Ford algorithm finds if there exists a negative weight cycle in the graph. So, by negating the weights of all the edges in the graph, we can then use the Bellman-Ford algorithm to find if there is a single negative weight cycle which would imply an arbitrage. This example is from *The Algorists*[1].

References

[1] Arbitrage. https://www.thealgorists.com/Algo/ShortestPaths/Arbitrage. The Algorists.

[2] Bellman-ford algorithm — dp-23. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/. GeeksforGeeks.

[3] Bellman-ford algorithm java. https://www.javatpoint.com/bellman-ford-algorithm-java. JavaTPoint.

[4] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, 2006.