East West University

Course Title: Algorithms

Course Code: Cse246

Section: 3

Semester: Spring 2022

Project Report on

Johnson's Algorithm

Course Instructor:

Jesan Ahmed Ovi

Senior Lecturer, Department of CSE,

East West University, Dhaka

Prepared By

Name	ID
Mehrab Islam Arnab	2020-1-60-015
Rokeya Jahan Chowdhury Ettifa	2020-1-60-232

Date of Submission: 16th May, 2022

Problem statement:

Johnson’s algorithm is a way to find the shortest paths between all pairs of vertices in a sparse, edge-weighted, directed graph. It allows some of the edge weights to be negative numbers, but no negative-weight cycles may exist. It works by using the Bellman–Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra’s algorithm to be used on the transformed graph.

In this project, we had to find shortest paths between every pair of vertices in a given weighted directed Graph where weights may be negative as well using Johnson's Algorithm without using Brute-Force.

System Requirements:

Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz   1.80 GHz

RAM: 8.00 GB

Operating System: Windows 11 Home

IDE: PyCharm 2021.2.2 (Community Edition)

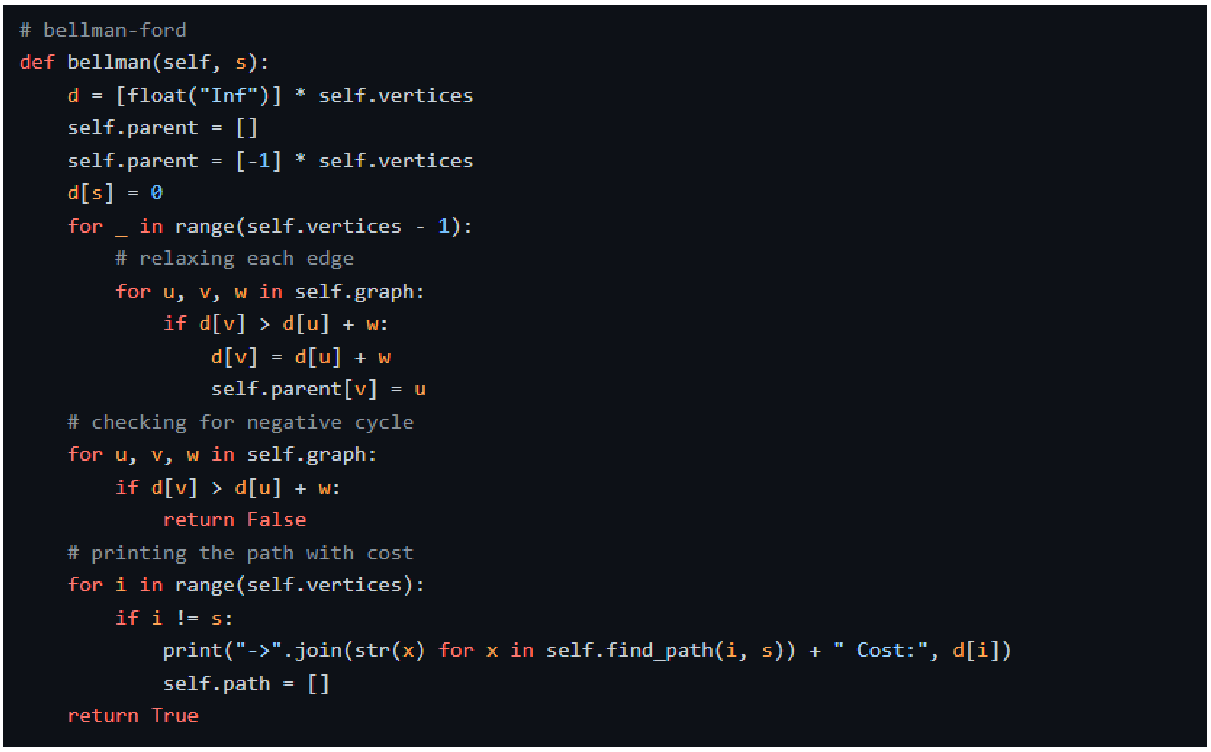
System design:

We are explicitly told to use bellman-ford so that’s exactly what this algorithm does. It uses every vertice as source and runs bellman ford shortest path algorithm while looking for negative cycles. If no negative cycle is found, it returns path from source to all other nodes with cost otherwise halts all operation and exits the algorithm completely.

If no negative cycle exists and all paths are printed, the algorithm enters its next phase which is committing a transformation to the graph so that dijkstra’s shortest path algorithm can be used to traverse it in future. It does so by finding the biggest negative value, if any, and adds its positive counterpart to every edge value. Thus all negative values are removed and the overall difference between edges is kept consistent.

The entire code can be found here

Implementation:

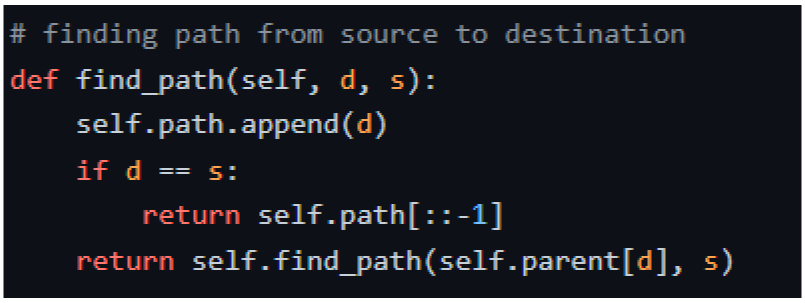


```python
# bellman-ford
def bellman(self, s):
    d = [float("Inf")] * self.vertices
    self.parent = []
    self.parent = [-1] * self.vertices
    d[s] = 0
    for _ in range(self.vertices - 1):
        # relaxing each edge
        for u, v, w in self.graph:
            if d[v] > d[u] + w:
                d[v] = d[u] + w
                self.parent[v] = u
    # checking for negative cycle
    for u, v, w in self.graph:
        if d[v] > d[u] + w:
            return False
    # printing the path with cost
    for i in range(self.vertices):
        if i != s:
            print("->".join(str(x) for x in self.find_path(i, s)) + " Cost:", d[i])
            self.path = []
    return True
```

The main function that finds the shortest distance from source to each other vertices. It initialises distance of source as 0 and distance to all other nodes as infinite. It then checks each edge with the relaxation condition (d[v] > d[u] + w) and updates if they meet it. It does it for v - 1 times because a shortest path in a graph can have at most v - 1 nodes in between source and destination.

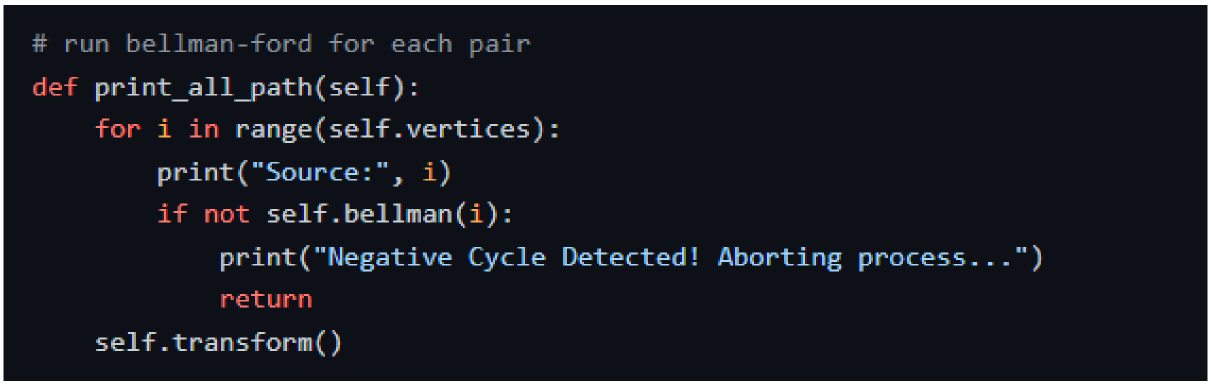
It then checks each edge once again in order to detect negative cycles and returns False if any are found.

If no negative cycles are found then it calls the find_path() function and prints the cost and path from source to all other destinations and returns True.



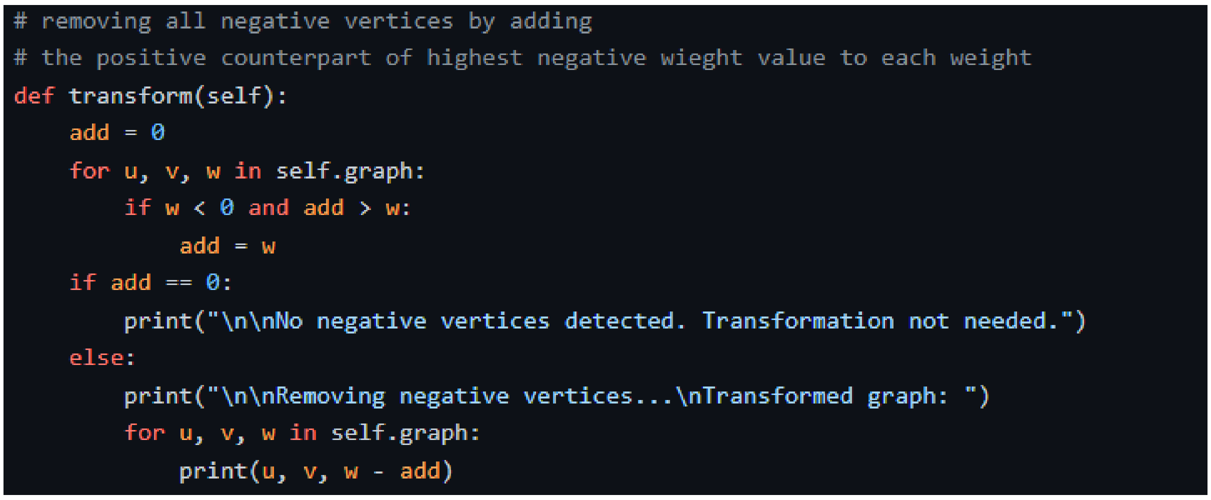
```python
# finding path from source to destination
def find_path(self, d, s):
    self.path.append(d)
    if d == s:
        return self.path[::-1]
    return self.find_path(self.parent[d], s)
```

find_path() function starts from destination and backtracks to source using recursion to find the path and returns it.



```python
# run bellman-ford for each pair
def print_all_path(self):
    for i in range(self.vertices):
        print("Source:", i)
        if not self.bellman(i):
            print("Negative Cycle Detected! Aborting process...")
            return
    self.transform()
```

Cycling through each vertice, running bellman ford to print shortest path between each pair and aborting the entire process if any negative cycle is found. If the entire process is completed without detecting any negative cycles then calls the transform() function to enter the next phase of the algorithm.

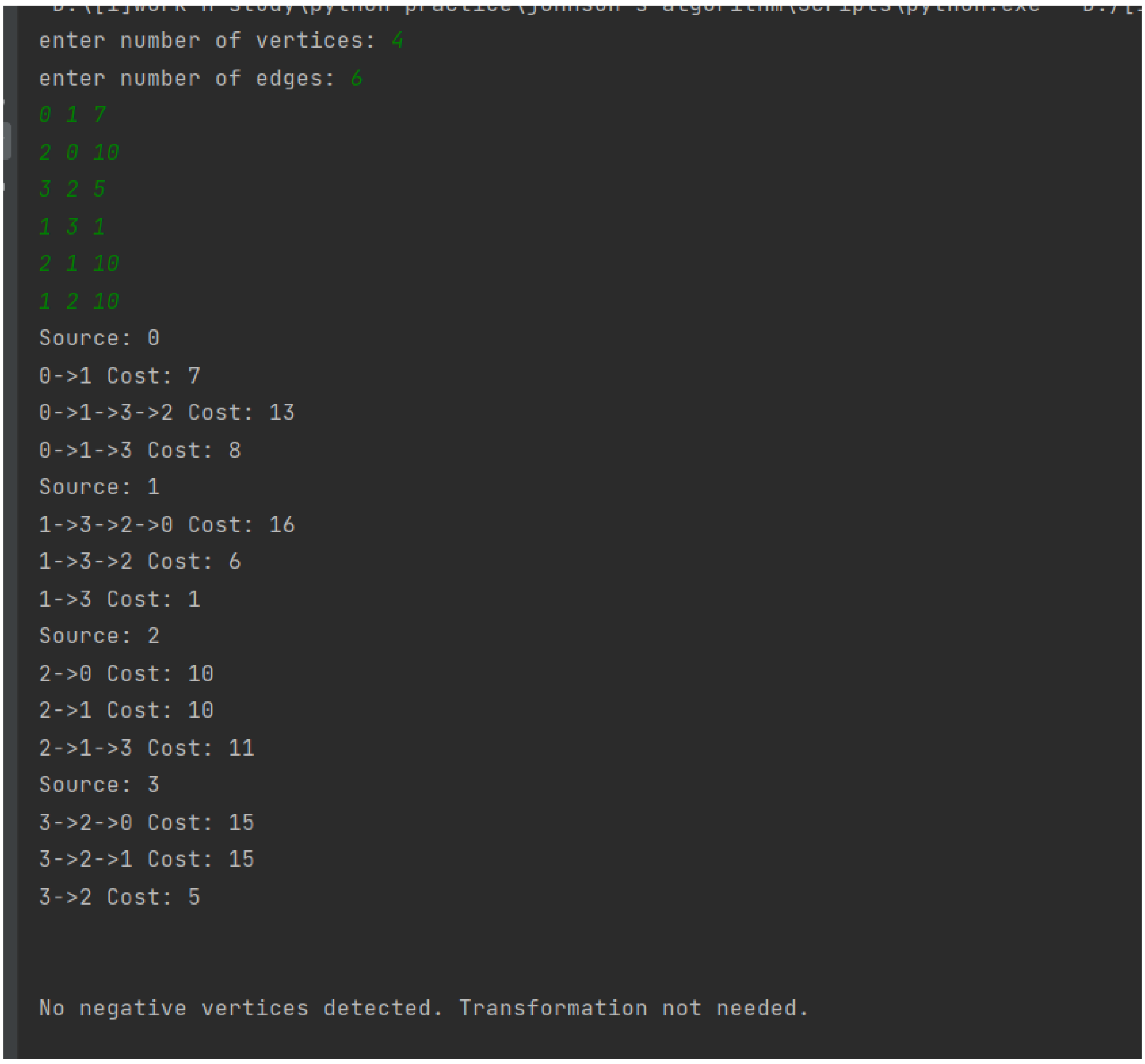


```python
# removing all negative vertices by adding
# the positive counterpart of highest negative wieght value to each weight
def transform(self):
    add = 0
    for u, v, w in self.graph:
        if w < 0 and add > w:
            add = w
    if add == 0:
        print("\n\nNo negative vertices detected. Transformation not needed.")
    else:
        print("\n\nRemoving negative vertices...\nTransformed graph: ")
        for u, v, w in self.graph:
            print(u, v, w - add)
```

Checking for negative edges. If none is found, let the user know that. If found, remove them by adding the positive counterpart of the biggest negative value to every other edge and printing the new edge information.

Testing Results:

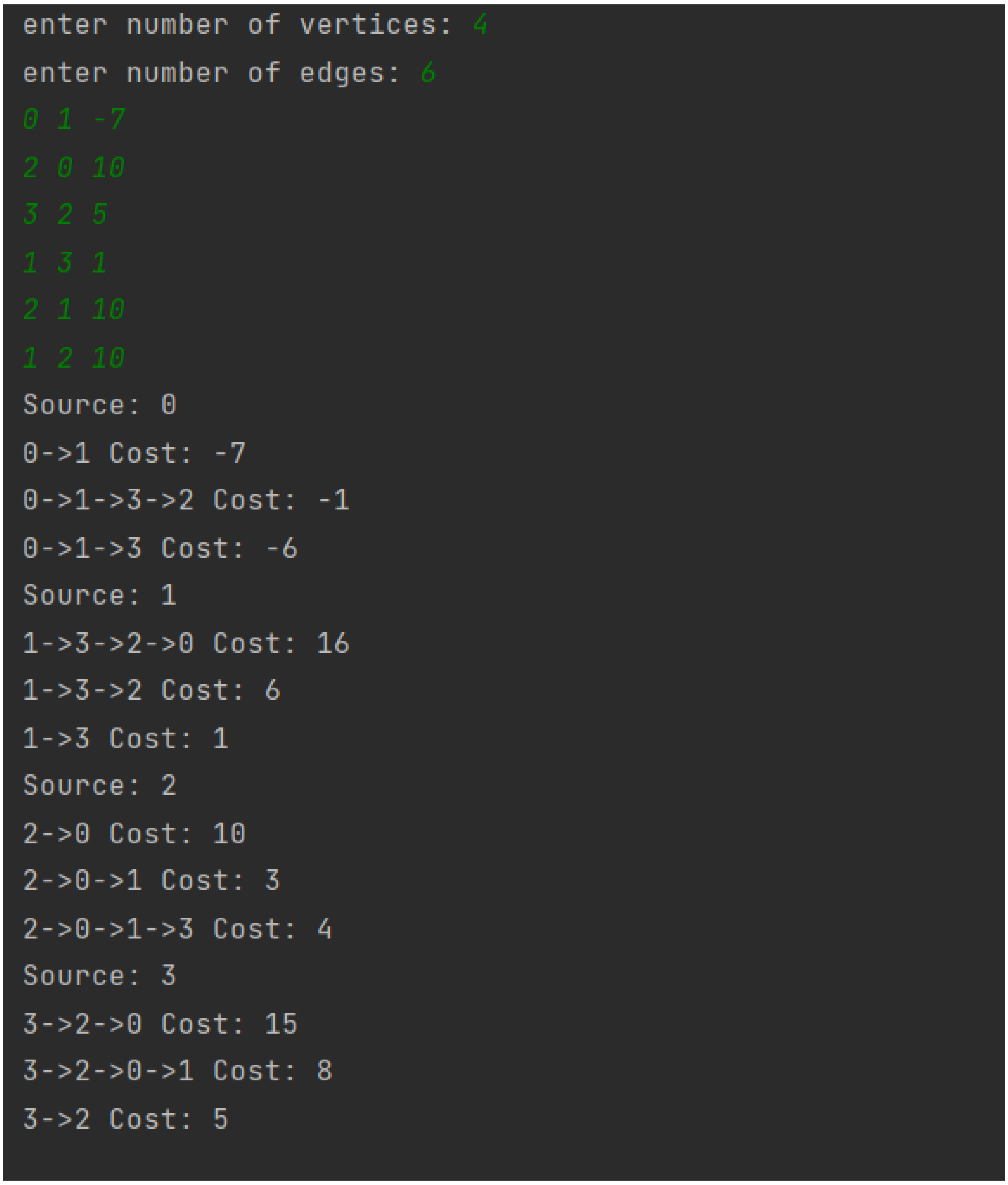
Scenario 1: No negative cycles. No negative edges. No transformation needed.



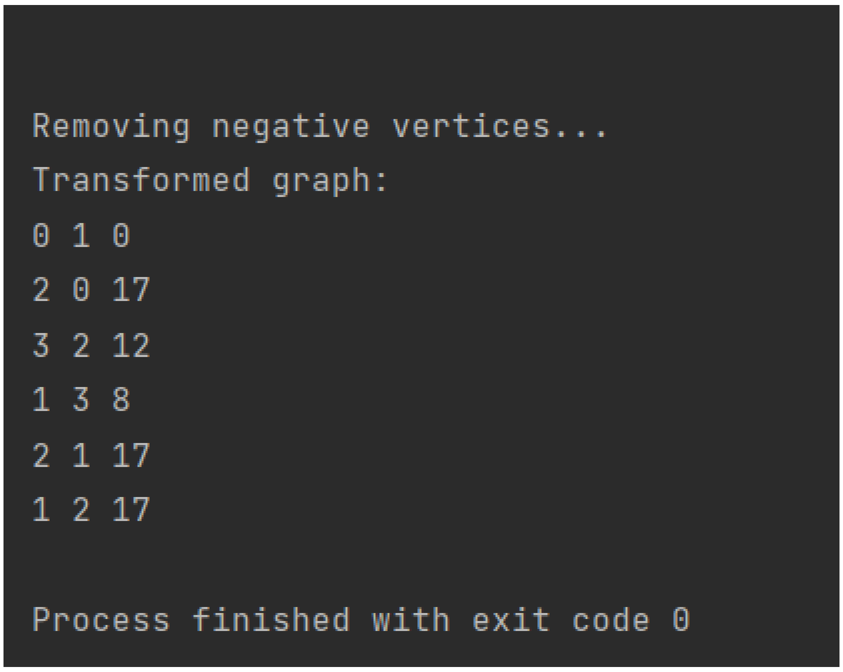
```
D:\[1]work n study\python practice\johnson's algorithm\Scripts\python.exe "D:/[
enter number of vertices: 4
enter number of edges: 6
0 1 7
2 0 10
3 2 5
1 3 1
2 1 10
1 2 10
Source: 0
0->1 Cost: 7
0->1->3->2 Cost: 13
0->1->3 Cost: 8
Source: 1
1->3->2->0 Cost: 16
1->3->2 Cost: 6
1->3 Cost: 1
Source: 2
2->0 Cost: 10
2->1 Cost: 10
2->1->3 Cost: 11
Source: 3
3->2->0 Cost: 15
3->2->1 Cost: 15
3->2 Cost: 5


No negative vertices detected. Transformation not needed.
```

Scenario 2: No negative cycles. Negative edges exist and transformation is needed.



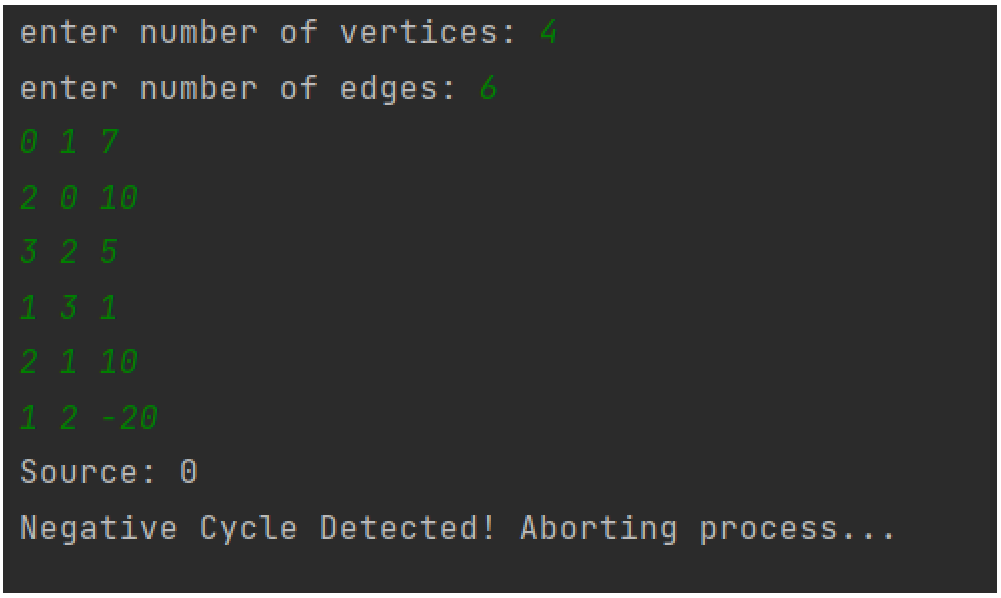
```
enter number of vertices: 4
enter number of edges: 6
0 1 -7
2 0 10
3 2 5
1 3 1
2 1 10
1 2 10
Source: 0
0->1 Cost: -7
0->1->3->2 Cost: -1
0->1->3 Cost: -6
Source: 1
1->3->2->0 Cost: 16
1->3->2 Cost: 6
1->3 Cost: 1
Source: 2
2->0 Cost: 10
2->0->1 Cost: 3
2->0->1->3 Cost: 4
Source: 3
3->2->0 Cost: 15
3->2->0->1 Cost: 8
3->2 Cost: 5
```



```
Removing negative vertices...
Transformed graph:
0 1 0
2 0 17
3 2 12
1 3 8
2 1 17
1 2 17

Process finished with exit code 0
```

Scenario 3: Negative cycles exist. Shortest path doesn't exist. Exiting system.



```
enter number of vertices: 4
enter number of edges: 6
0 1 7
2 0 10
3 2 5
1 3 1
2 1 10
1 2 -20
Source: 0
Negative Cycle Detected! Aborting process...
```

Future Scope:

- The algorithm, even though it can detect negative cycles, it can not tell the user where that negative cycle is. It can be optimised to do so.

- We can only see a bunch of numbers that represent the graph for now. The user experience can be vastly improved by better visualising the graph and the various paths.