Lab 6 – Graphs

Jonatan M. Contreras

University of Texas at El Paso

**Introduction**

In lab six, we implemented graphs as three different kinds of representations: an adjacency list, an edge list, and an adjacency matrix. For part one, we were to implement the functions insert\_edge, delete\_edge, and display for each graph representation. We were also to implement functions that would transform the current graph representation (adjacency list, edge list, or adjacency matrix) into one of the other two graph representations; the functions were named as\_AL(), as\_AM(), and as\_EL().

For part two, we were to solve a classic puzzle. The puzzle is as follows: You have a fox, a chicken and a sack of grain. You must cross a river with only one of them at a time. “If you

leave the fox with the chicken he will eat it; if you leave the chicken with the grain he will eat it. How can you get all three across safely?” We were to implement the puzzle as a graph where the vertices 0-15 represented states of the world. The puzzle was implemented as the three different graph representations (adjacency list, adjacency matrix, and edge list) and unique solutions were found using breadth-first-search and depth-first-search algorithms. These algorithms were also implemented.

**Proposed Solution Design and Implementation**

*Part 1: Completing Graph Representation Implementations:*

1.1: graph\_AM Implementation:

a) insert\_edge:

To implement insert\_edge for graph\_AM, I first checked to see if the graph is directed. If so, we only need to insert the weight at the location of the matrix AM[source][destination]. If the graph is not directed, we need to add two edges to the matrix: one to AM[source][destination] and another to AM[destination][source].

b) delete\_edge:

Delete\_edge was completed similarly to insert\_edge. There is a check to see if the graph is directed. If not, we simply change the weight in AM[source][destination] to -1, which represents an absence of edge between the two vertices. Else, we replace two edges with -1, one in AM[source][destination] and another in AM[destination][source].

c) display:

To display graph\_AM, we can simply print the numpy array.

d) as\_AL:

To implement this function for the AM representation, I first initialized an AL representation of a graph. I then looped through the matrix and checked to see if there were edges between the source and destination. If so, I would use the insert\_edge function to insert that edge into the graph. If the graph is directed, it would only insert that particular edge. If it was not directed, it would insert both the edge from source to destination and from destination to source since the adjacency list would require both edges to be inserted to properly represent the graph.

e) as\_EL:

An EL representation of the graph was first created. The matrix was then looped through to find edges. If an edge was found, it was inserted. It is not necessary to insert two edges for an undirected graph since the edge list only maintains a list of edges that appear once each.

1.2 graph\_EL implementation:

a) insert\_edge:

A for loop is used to loop through the edges in the edge list to check to see if the edge isn’t already in the list. There was also first a check to see if the graph is directed. This is because an edge only appears once in an edge list, so if the graph is not directed, we can check to see if the edge’s source and destination is not equal to the source and destination of the edge we are attempting to insert. For a non-directed graph, we have to make sure the source and destination of the edge we are currently looping through aren’t equal to the source and destination of the edge we are attempting to insert while also making sure that the source and destination of the edge we are currently looping through aren’t equal to the destination and source of the edge we are attempting to insert. This check occurs, and if the edge isn’t’ already in the edge list, we insert it. If it is, the for loop is broken.

b) delete\_edge:

Delete\_edge follows the same algorithm as insert\_edge, but has a difference. It uses a counter to keep count of where the for loop breaks when the edge is found. It then uses this counter to access the edge from the edge list and delete it.

c) Display:

To display the edge list, it was necessary to loop through the edge list and print every edge individually.

d) as\_AL:

To implement the as\_AL function, first an AL representation of the graph was created. Then the edge list was looped through and each edge was inserted in the AL representation of the graph.

e) as\_AM:

An AM representation of the graph was created. The edge list was then looped through and each edge was inserted into the AM representation of the graph.

1.3: as\_AM, as\_EL Implementations for AL:

a) as\_AM:

An AM representation of the graph was created. The adjacency list was then looped through, while looping through each list of edges and inserted each edge into the AM representation of the graph using its insert\_edge function.

b) as\_EL:

An EL representation of the graph was created. The adjacency list was then looped through, while looping through each list of edges and inserted each edge into the EL representation of the graph using its insert\_edge function.

*Part 2: BFS, DFS, and Finding the Solution to the Puzzle:*

The BFS and DFS search algorithms were implemented for each of the graph\_AM, graph\_AL, and graph\_EL classes. The algorithm was the same for all three classes with how the adjacent vertices are accessed being the main difference between the implementations. The queue, discovered, and path lists are created. The zero vertex is pushed into the queue, while acknowledging its been discovered in the discovered list. Then, while the queue is empty, we go through each adjacent vertex in the most recently popped vertex, and if it is not discovered, it is added to the queue, acknowledged as discovered, and its path element is changed to the current vertex to indicate that the path to that vertex is through the current vertex. For DFS, the only thing that changes are that we use a stack instead of a queue.

Finding the solution to the puzzle meant running BFS and DFS on each graph implementation. Each search algorithm found a unique solution to the puzzle by finding the path from state 0 to state 15 through the edges.

2.2: Solution to the Puzzle:

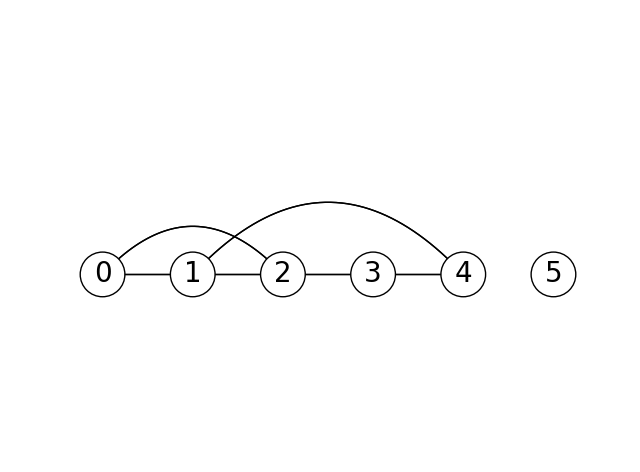
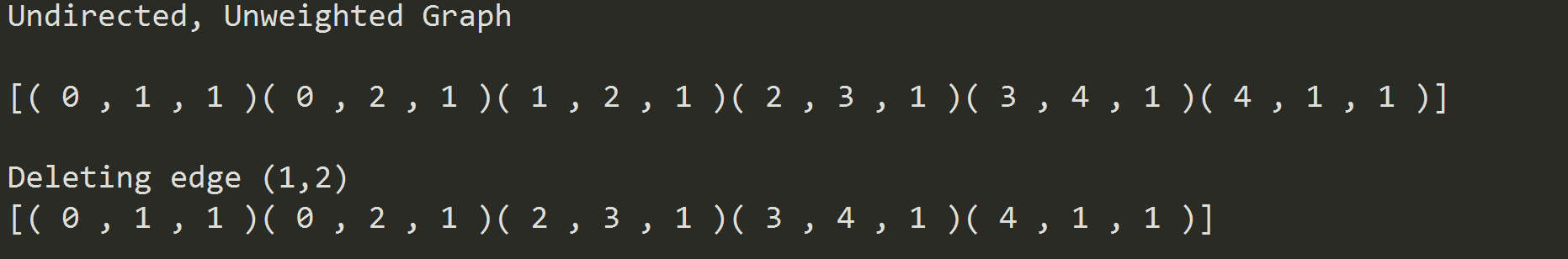
To find the solution to the puzzle, we first represent the puzzle as a graph. The edges are only legal transitions between legal states of the world. These edges were concluded to be the following: (0, 5), (2, 7), (2, 11), (4, 5), (4, 7), (4, 13), (8, 11), (8, 13), (10, 11), (10, 15). The edges were inserted, and all three graph representations are created. The BFS and DFS algorithms then find the two unique solutions for the puzzle.

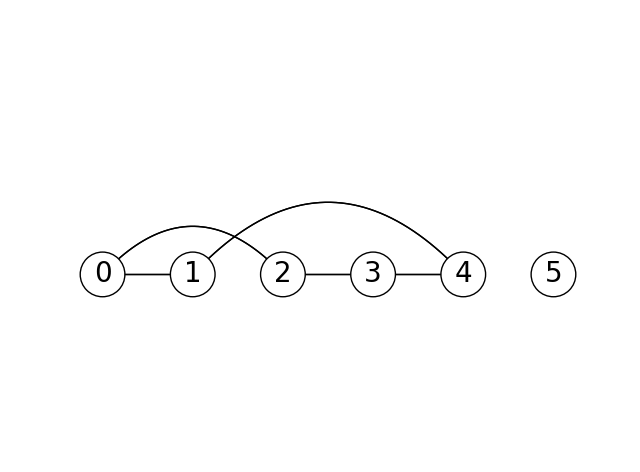
**Experimental Results**

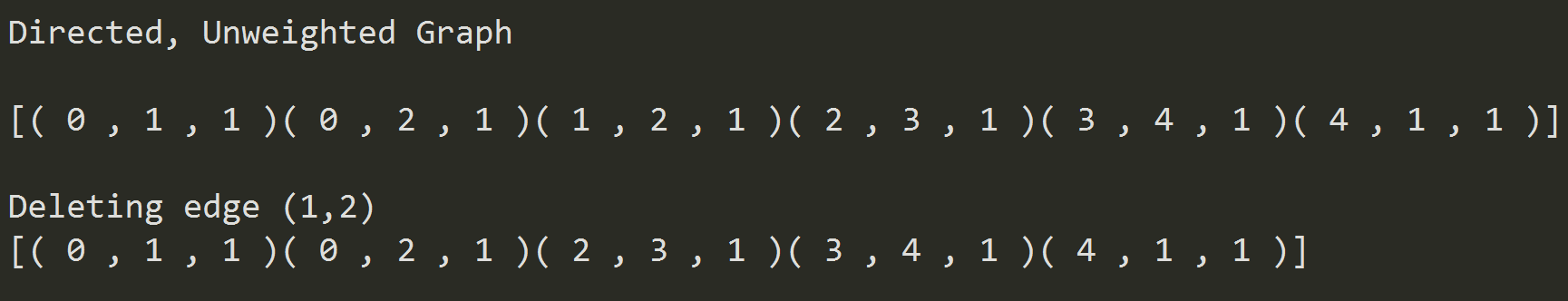
*Part 1:*

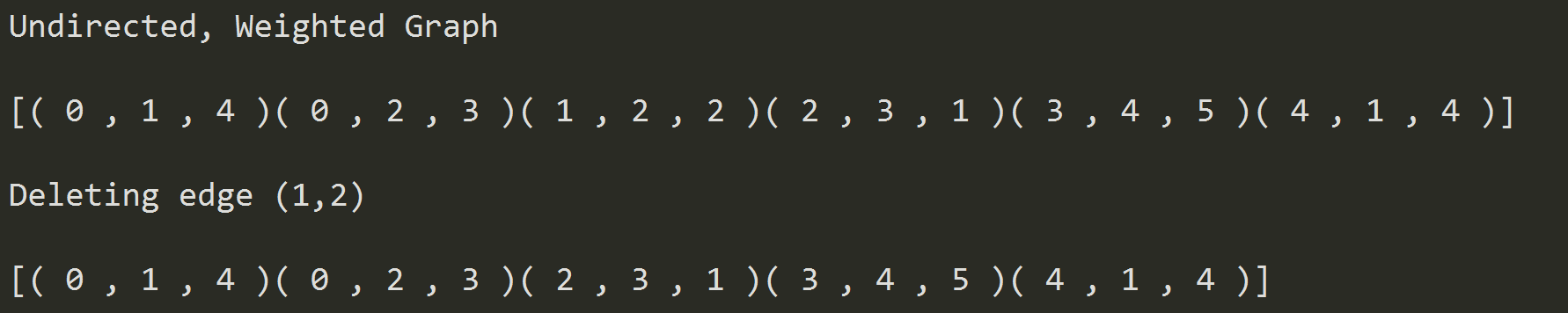
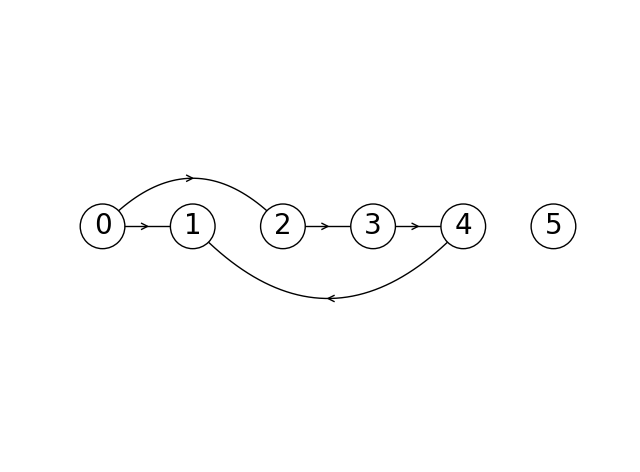
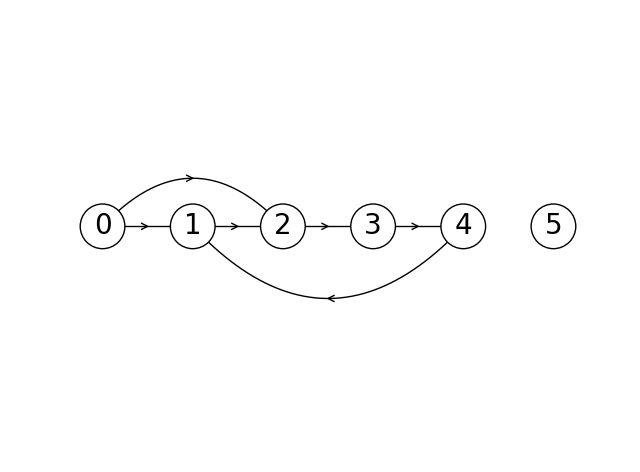
The following are screenshots of the output for test\_graphs.py which tests the functions required by part 1.

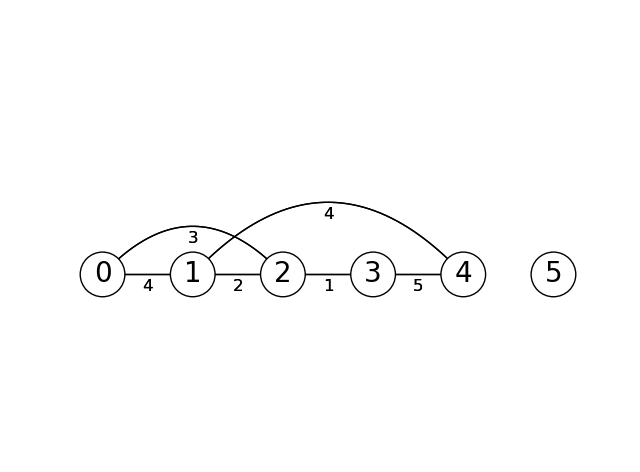
1.1: Edge List Tests:

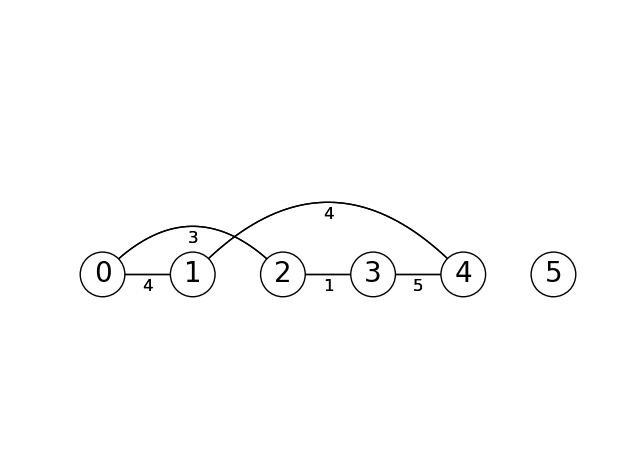


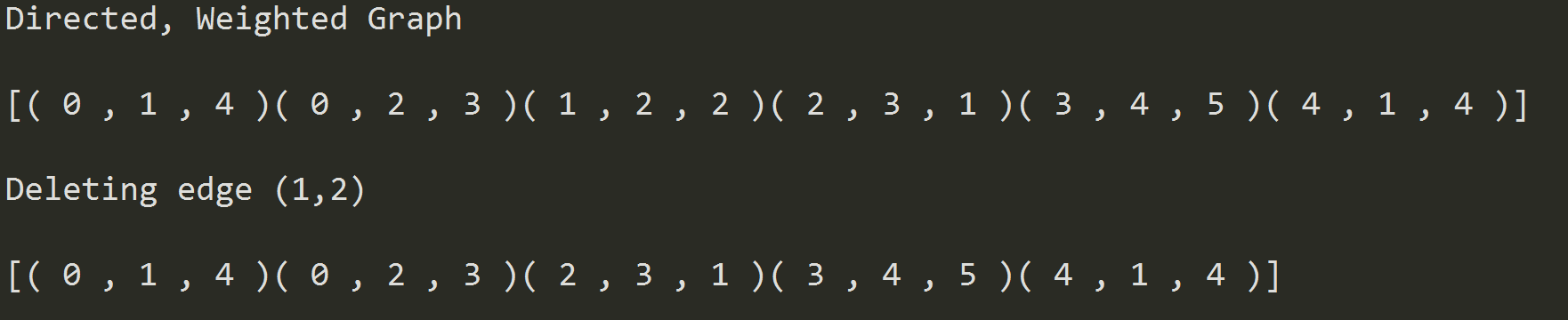


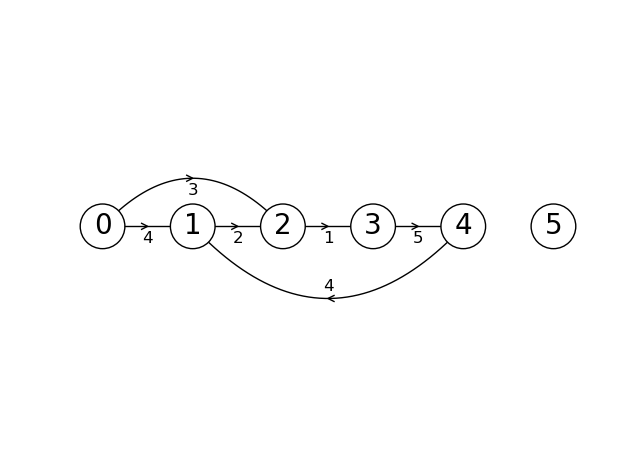
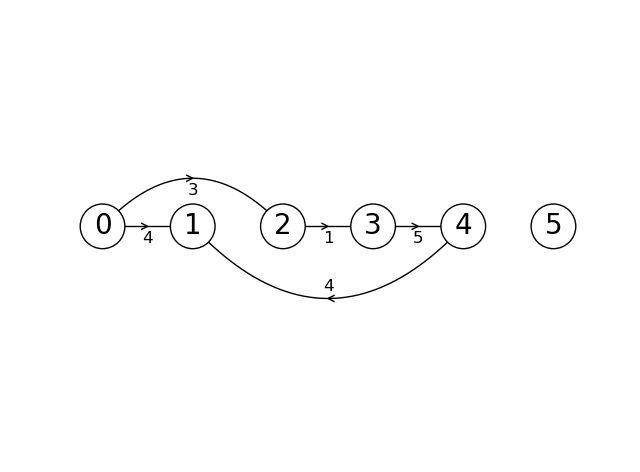




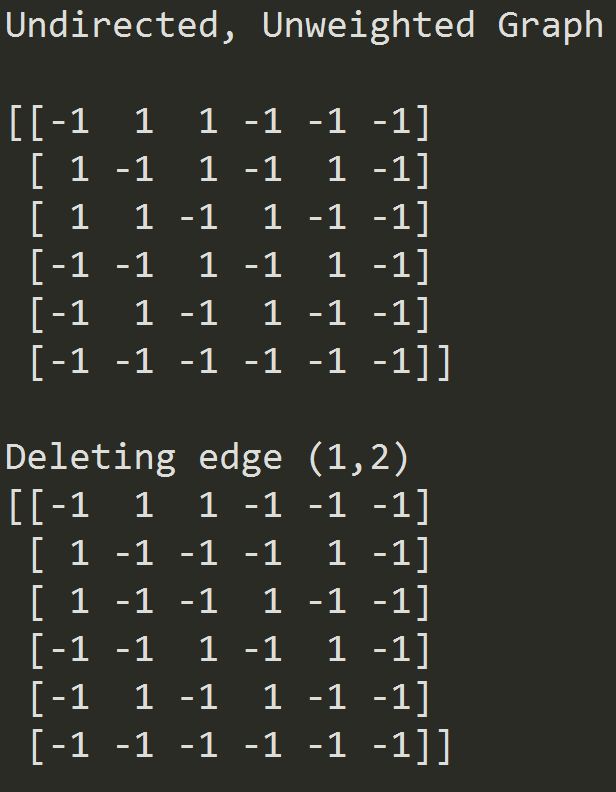


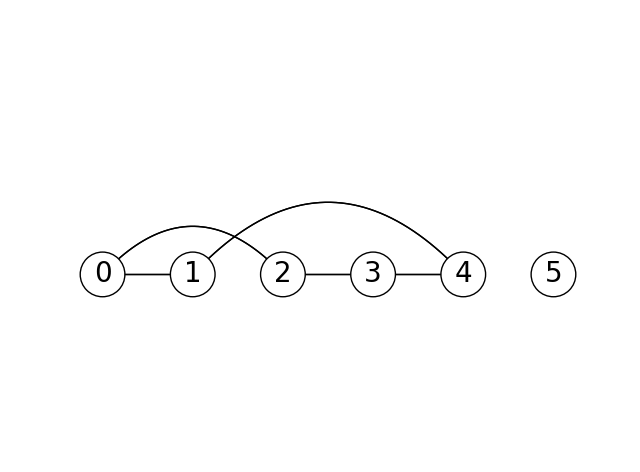
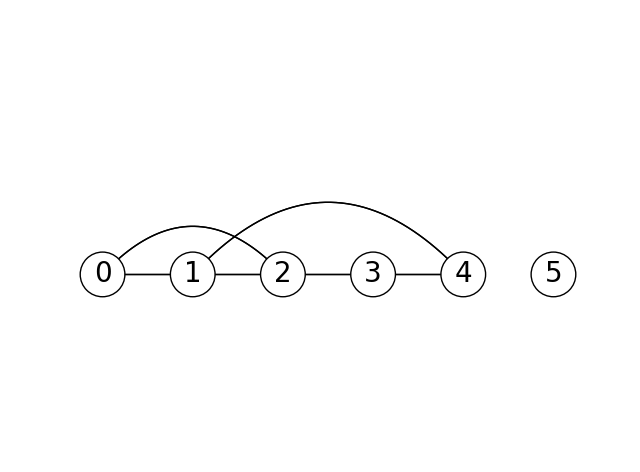


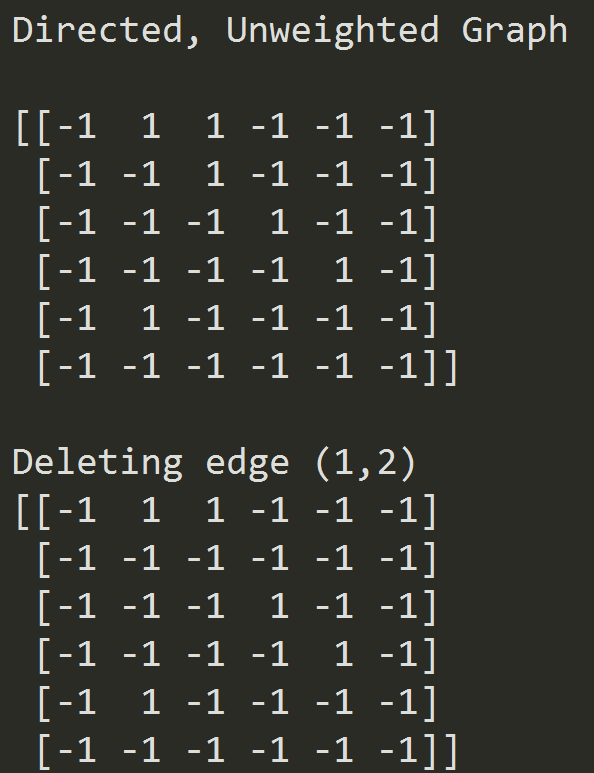


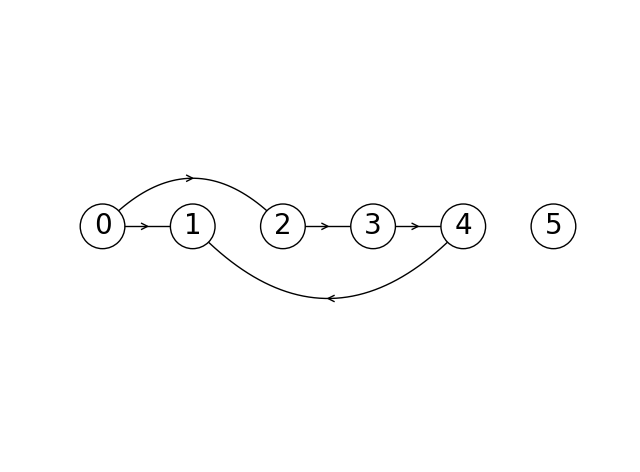
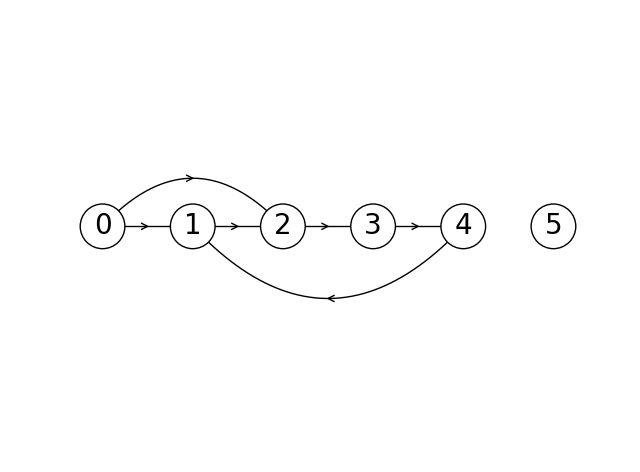


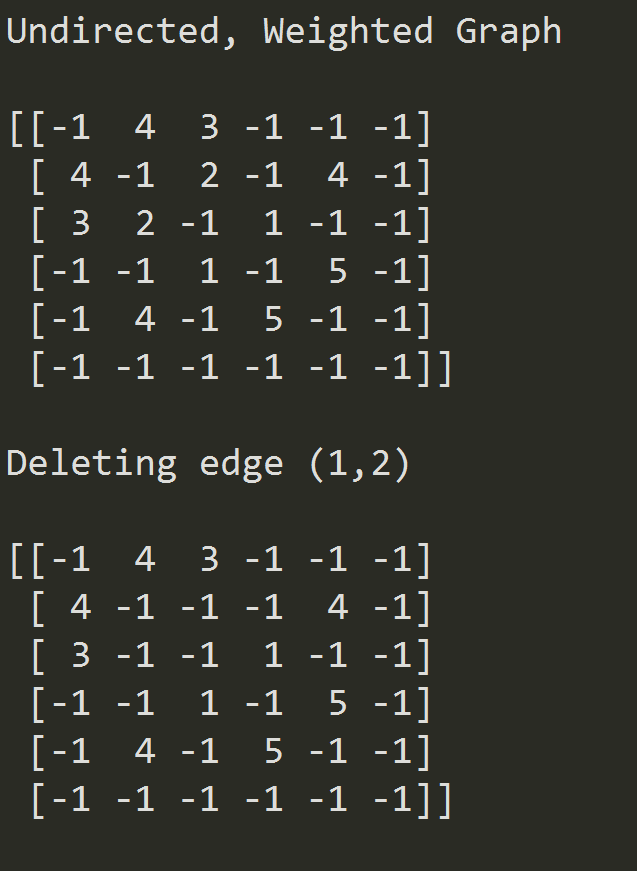
1.2 Adjacency Matrix Tests:

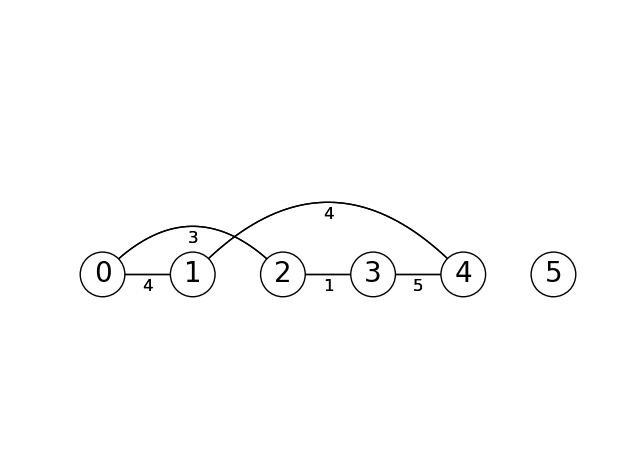
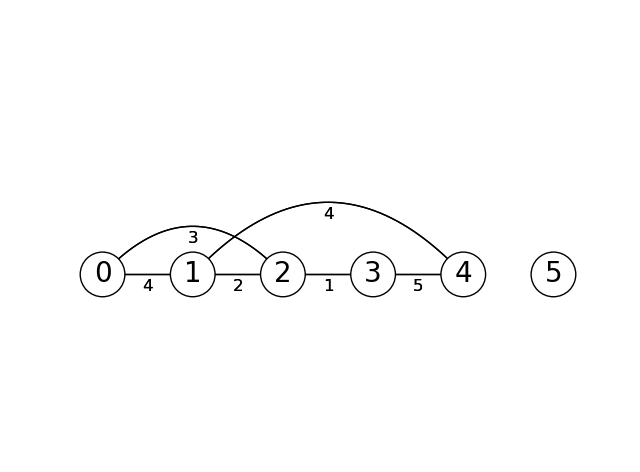


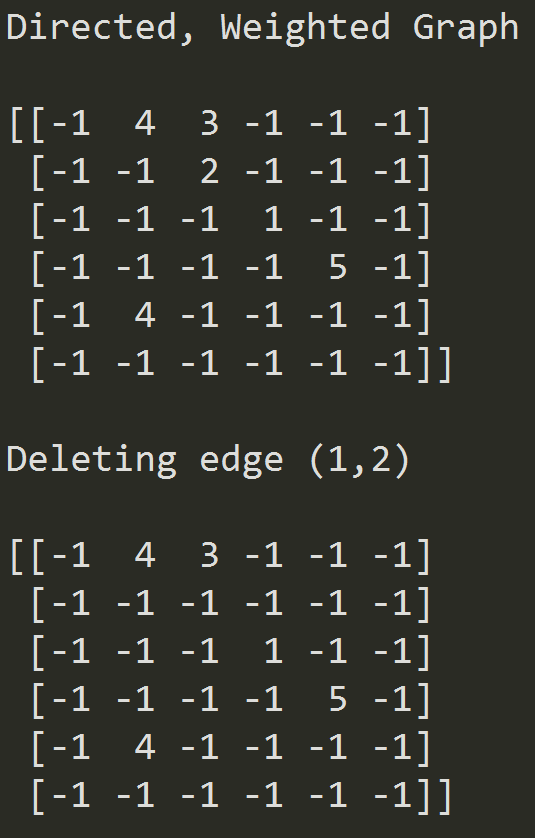


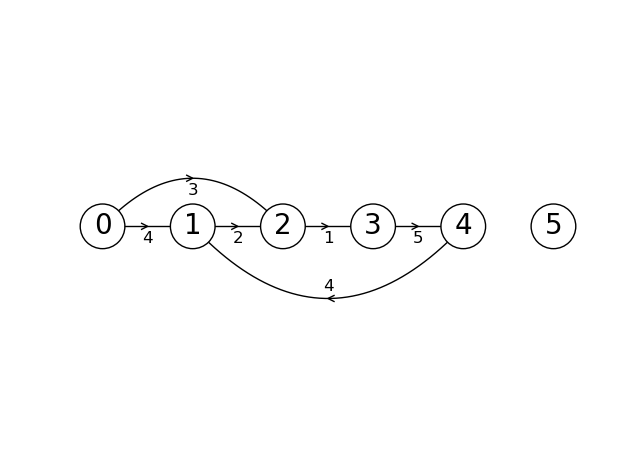
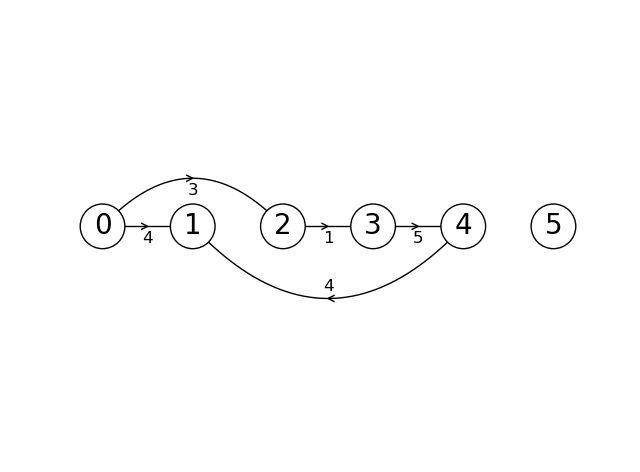


**

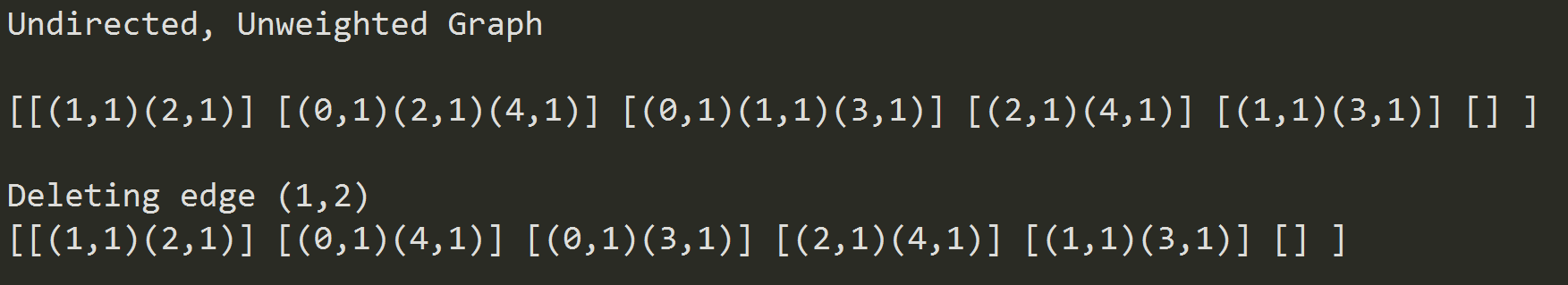
**

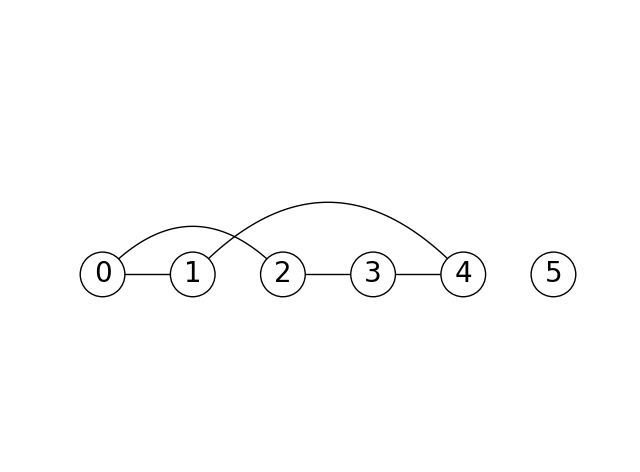
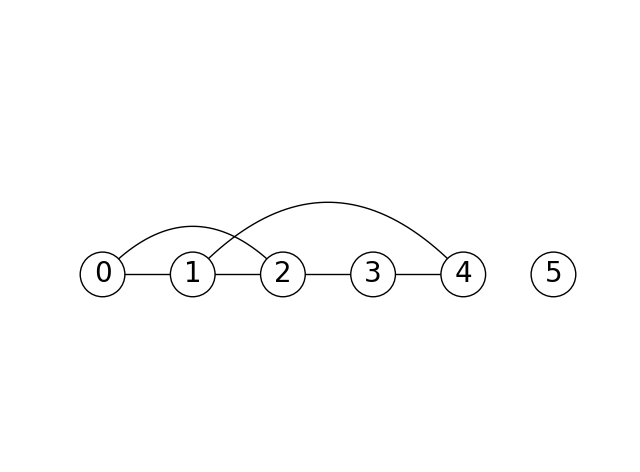
**

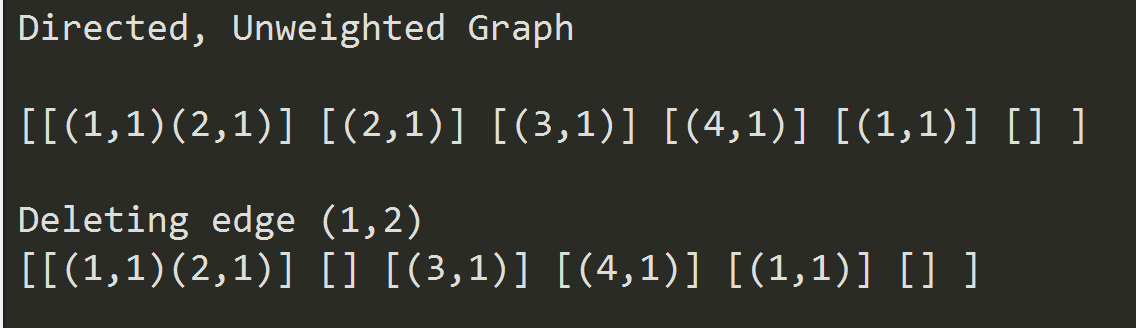
**

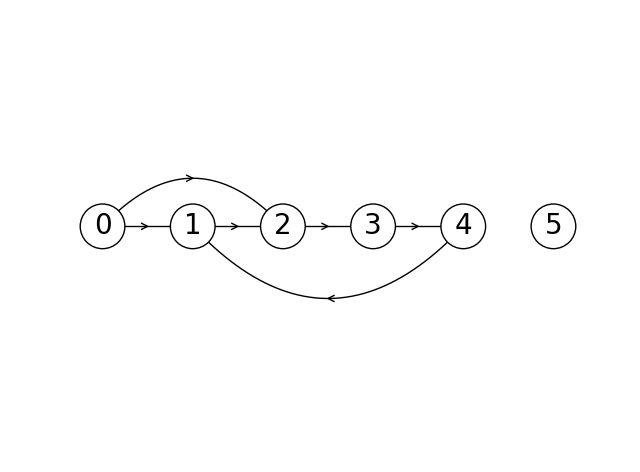
**

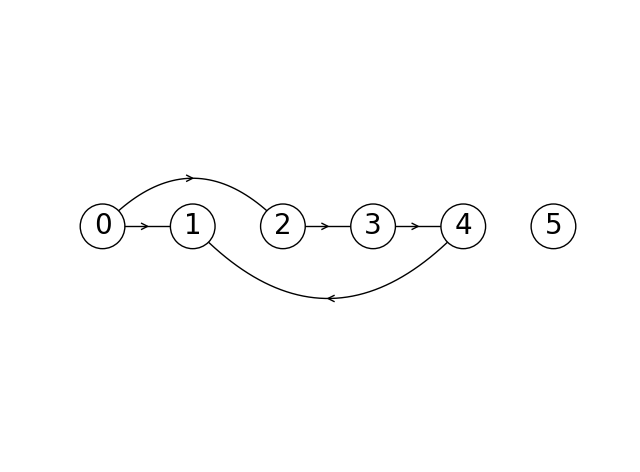
1.3 Adjacency List Tests:

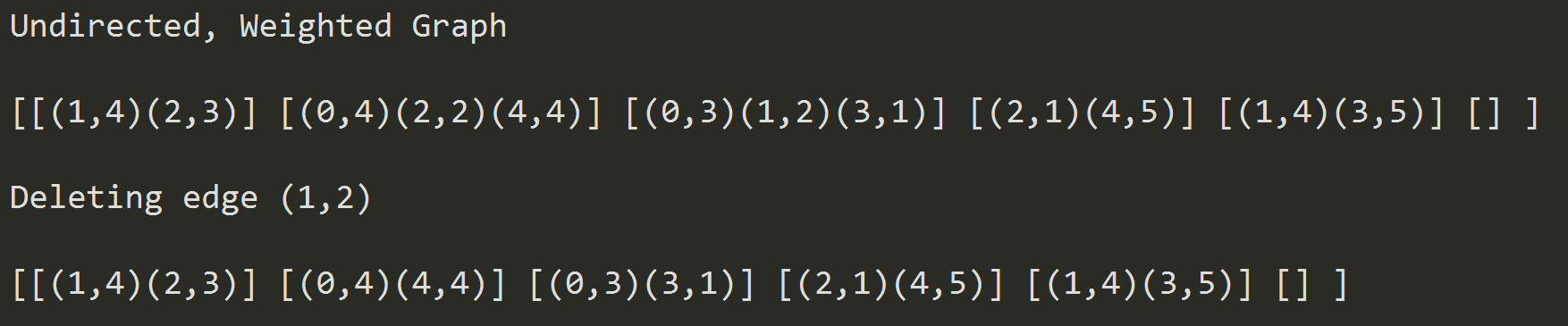
**

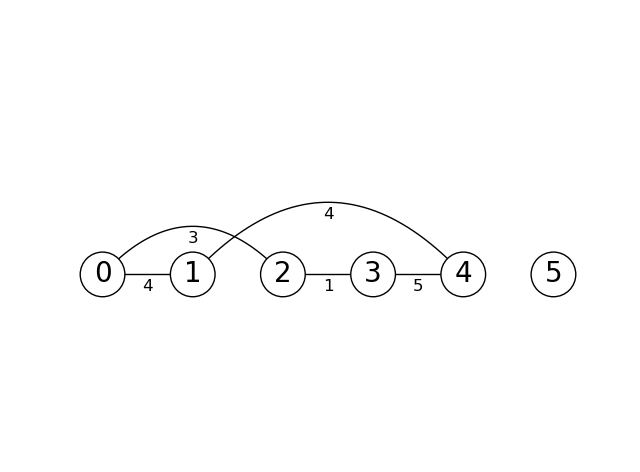
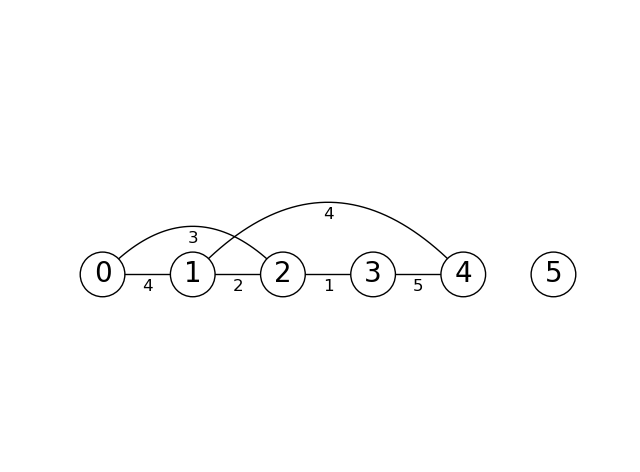
**

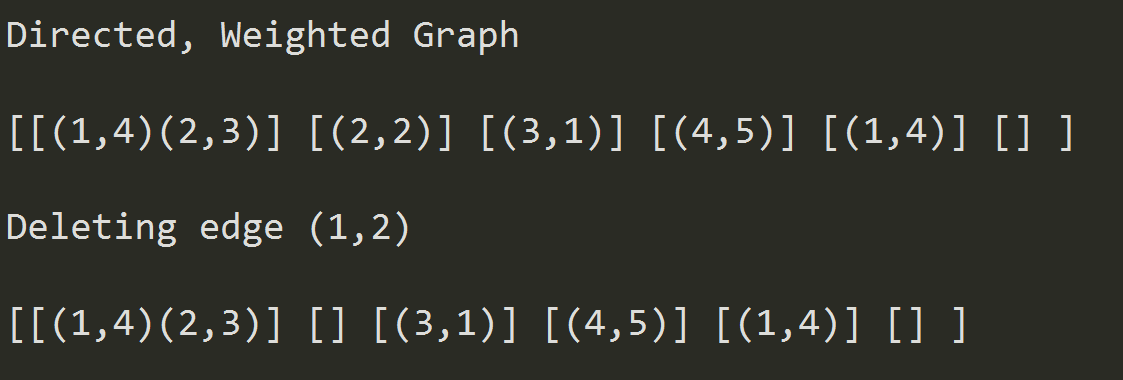
**

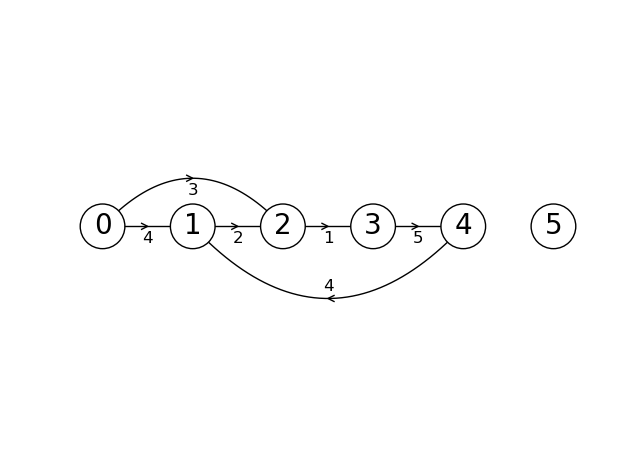
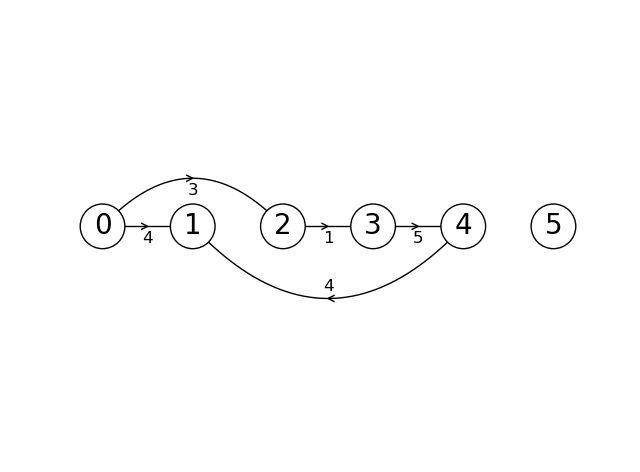
**

**

**

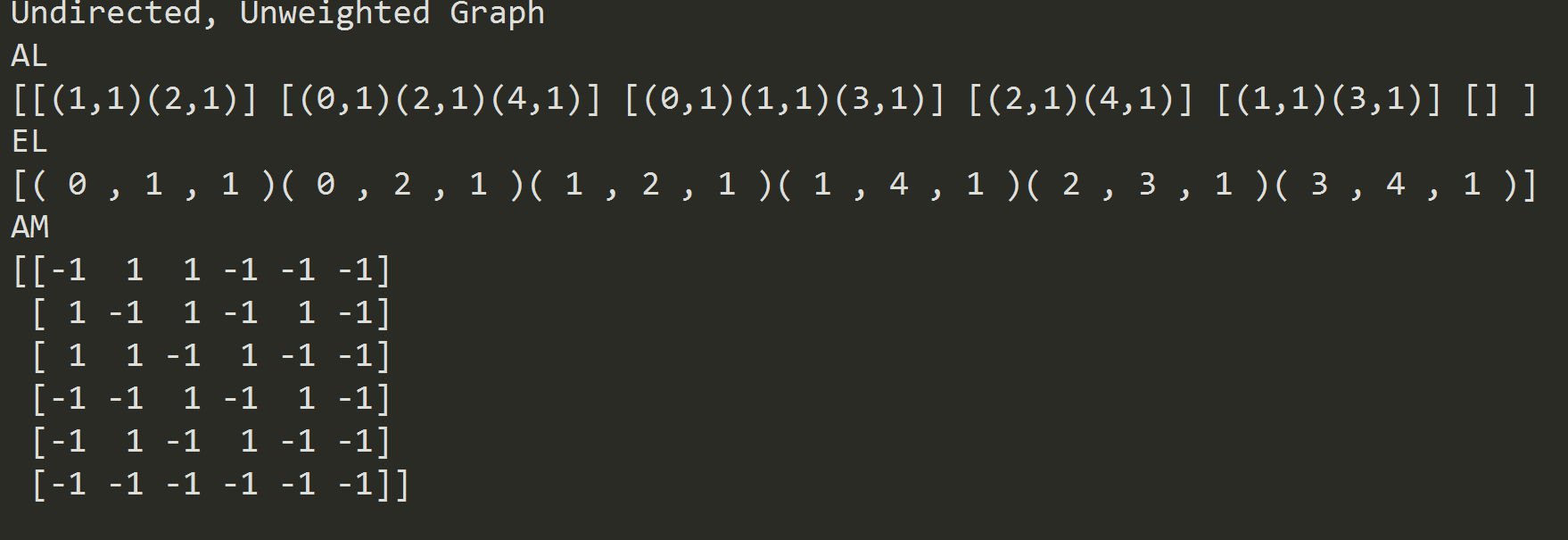
**

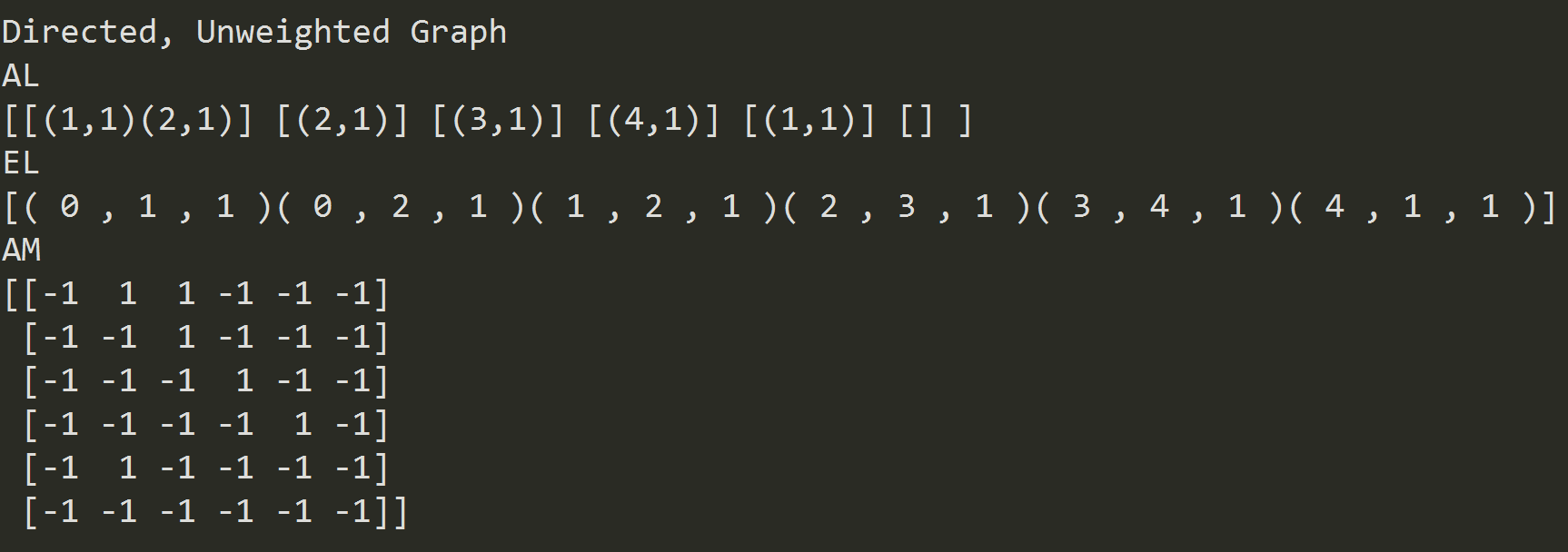
**

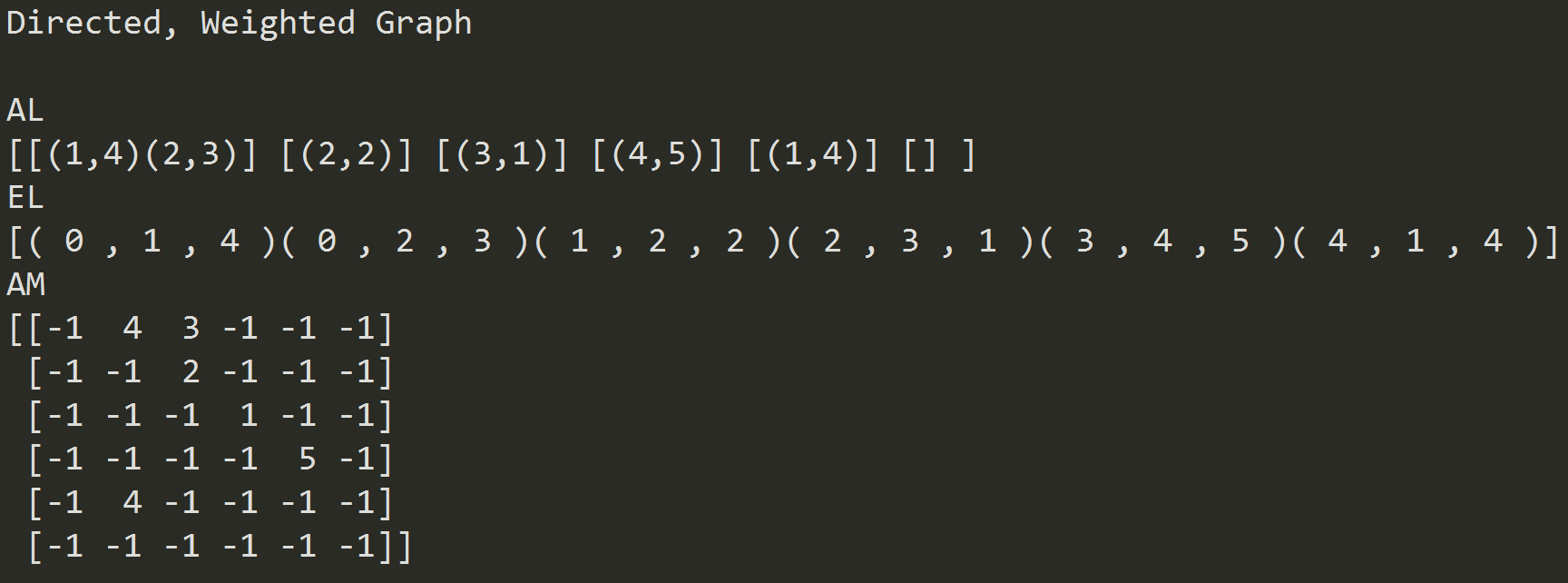
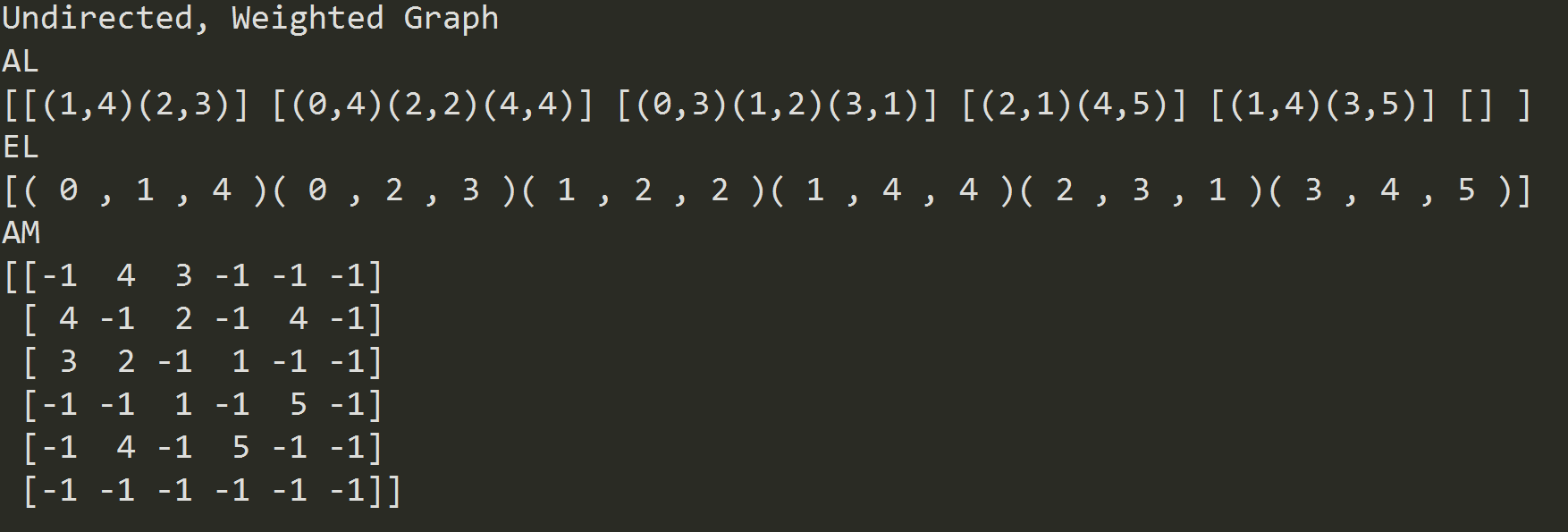
**

1.4 as\_AM, as\_EL representations for the AL graph implementation:

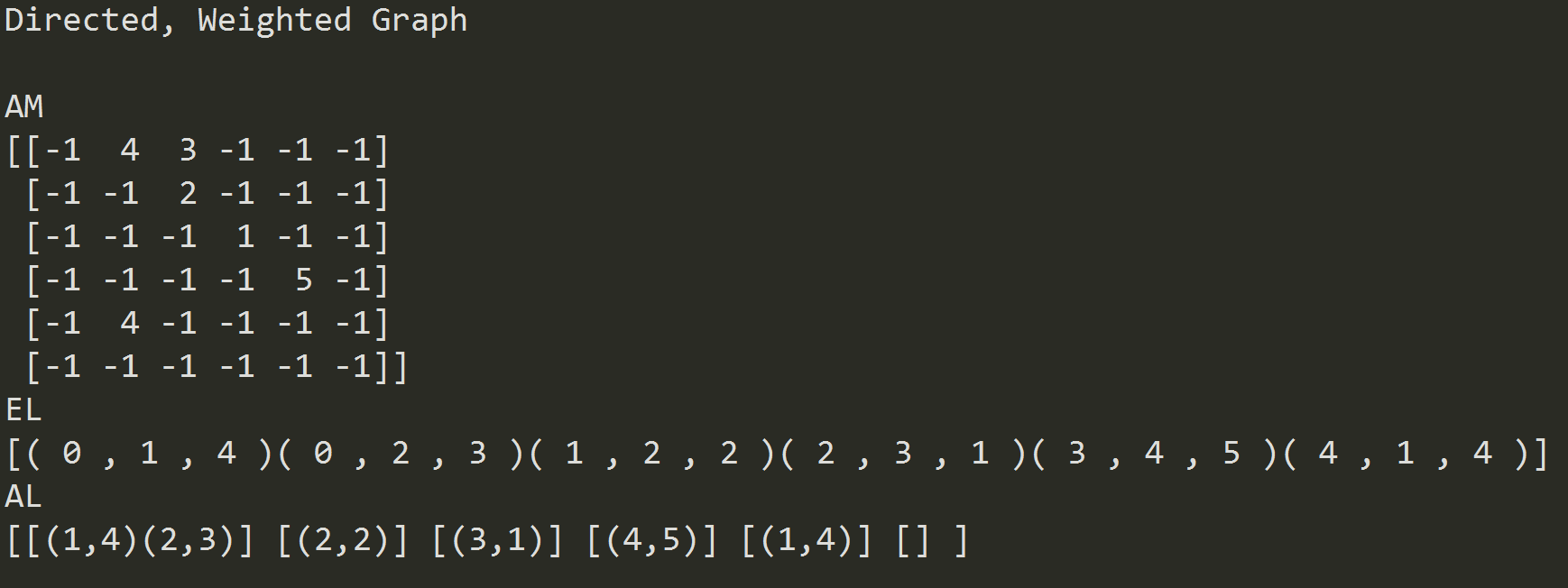
The following are screenshots of AL graphs being converted to edge list and adjacency matrices.

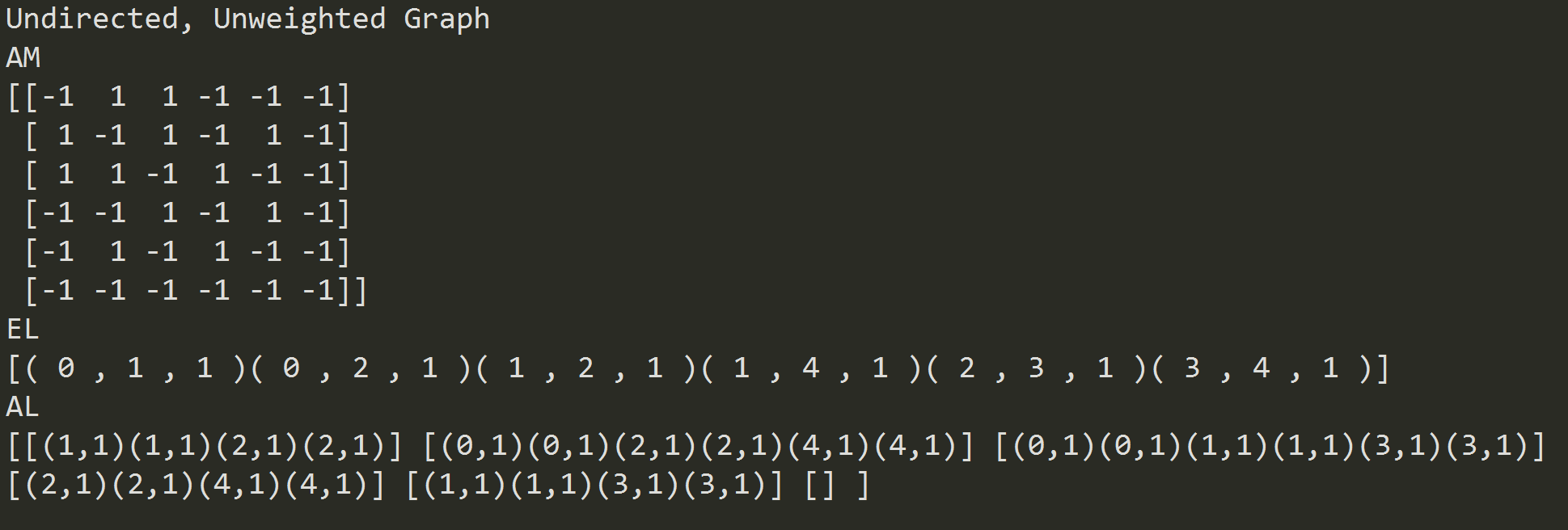
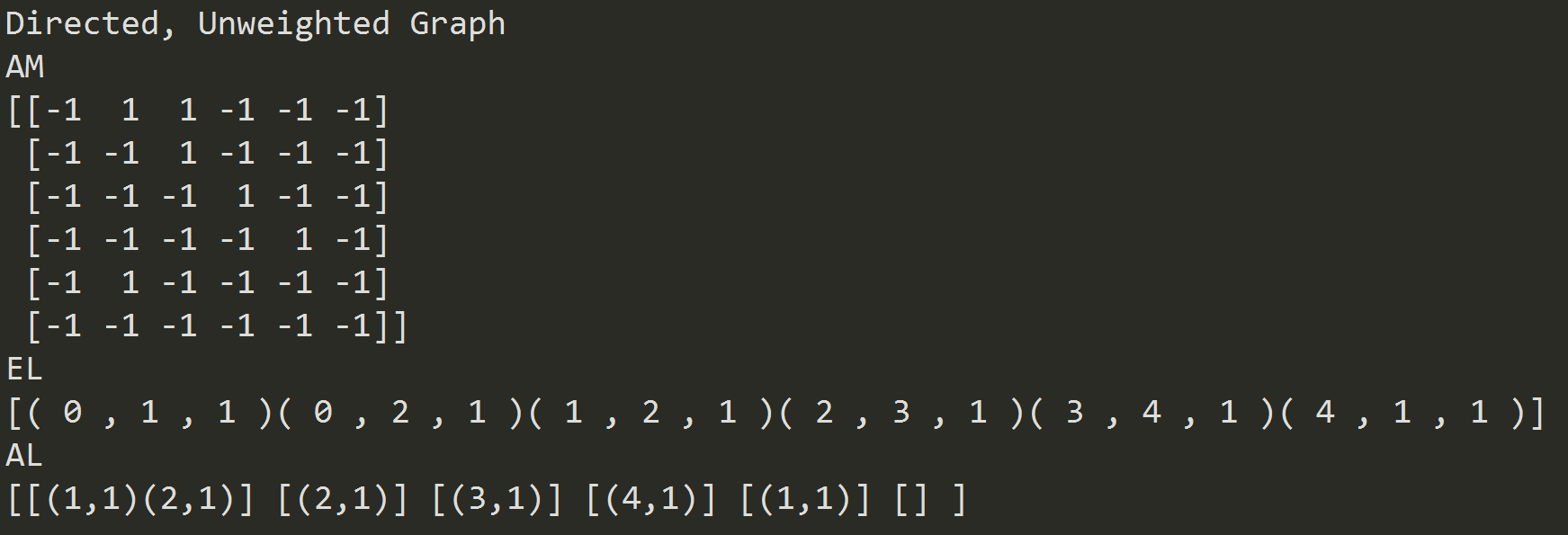
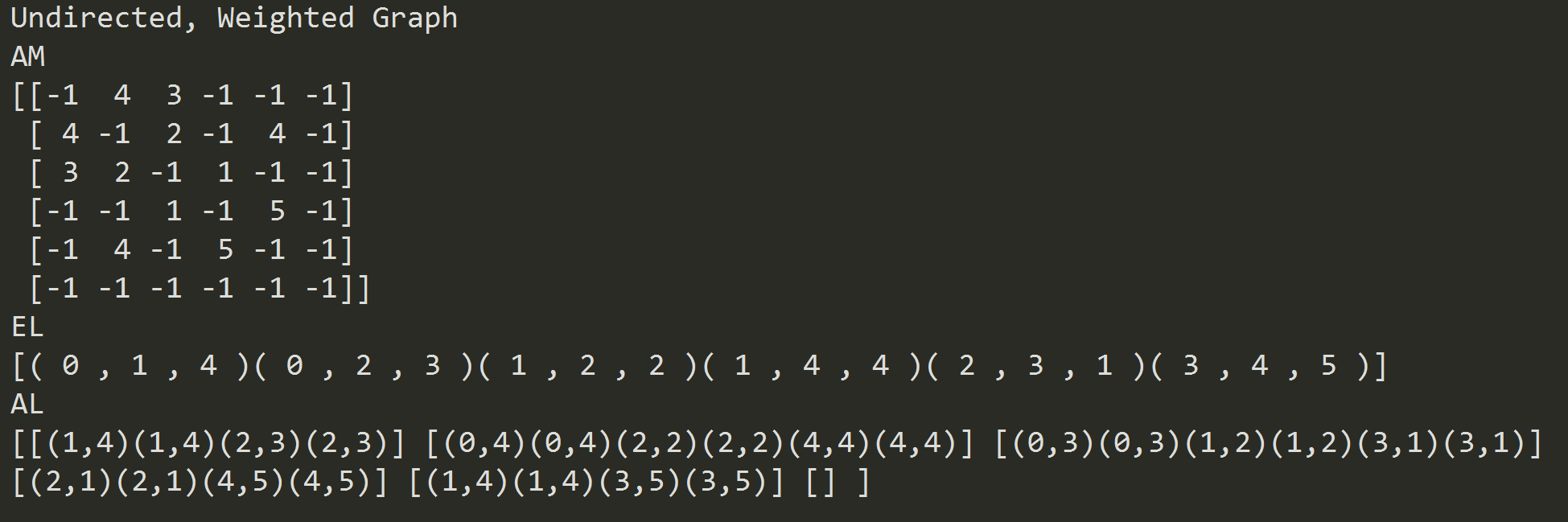


\

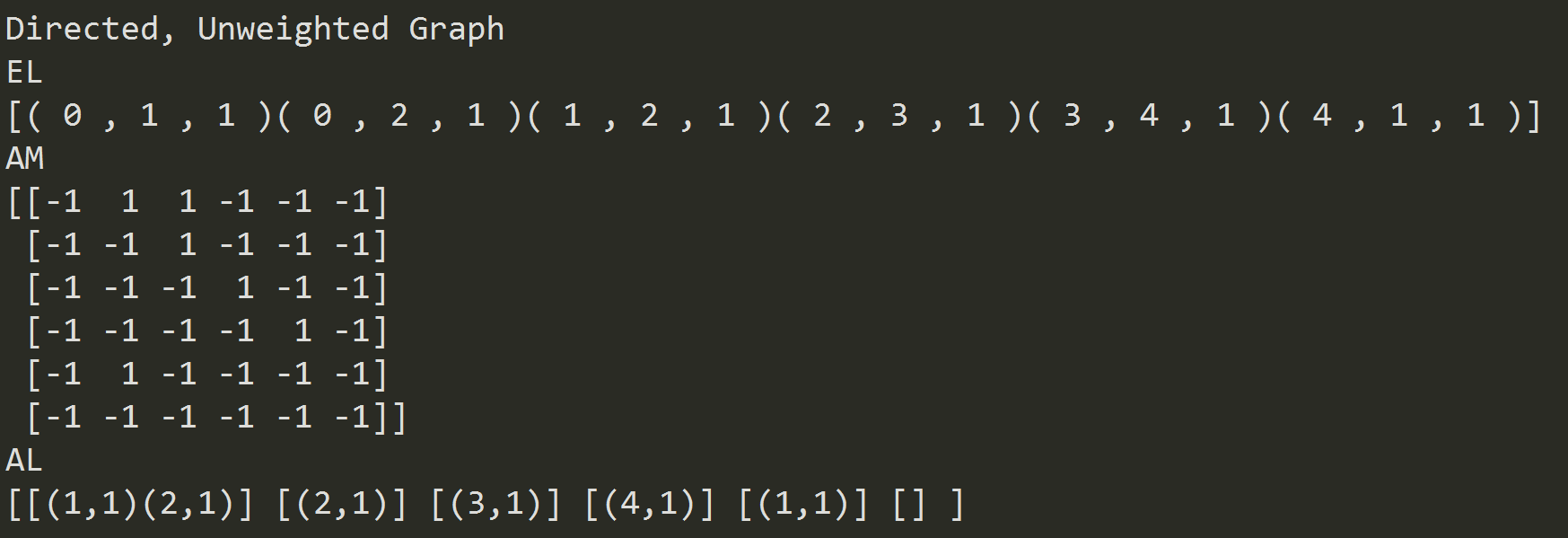


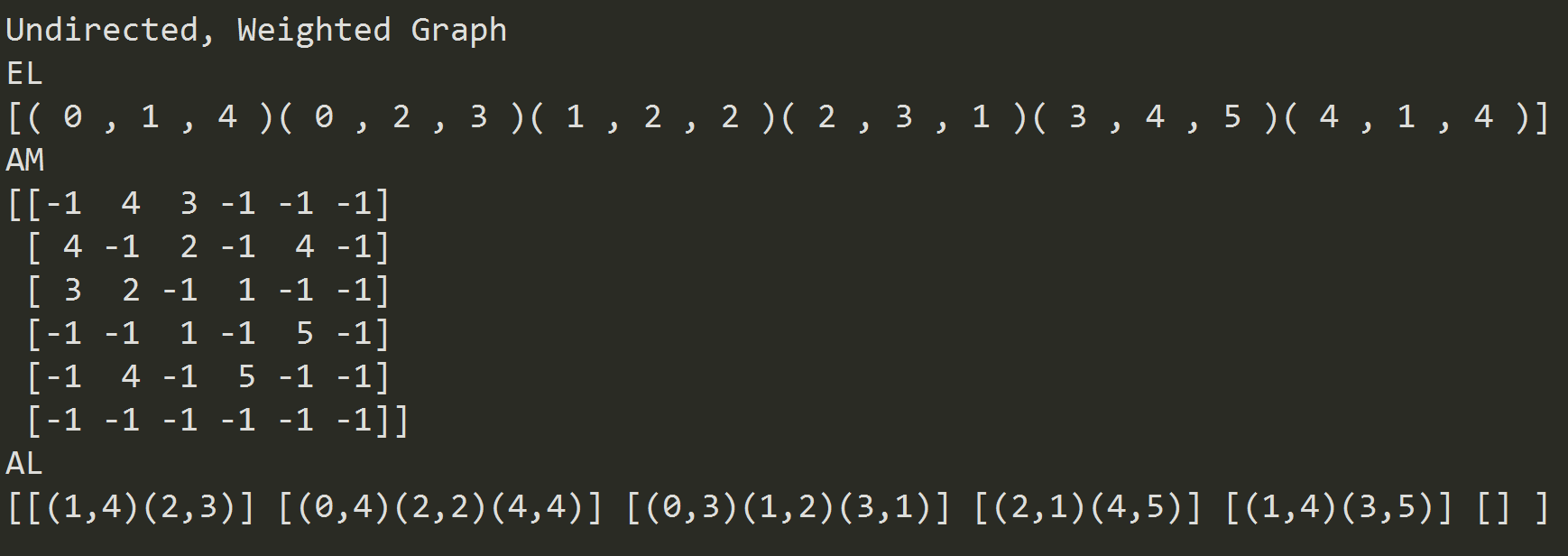
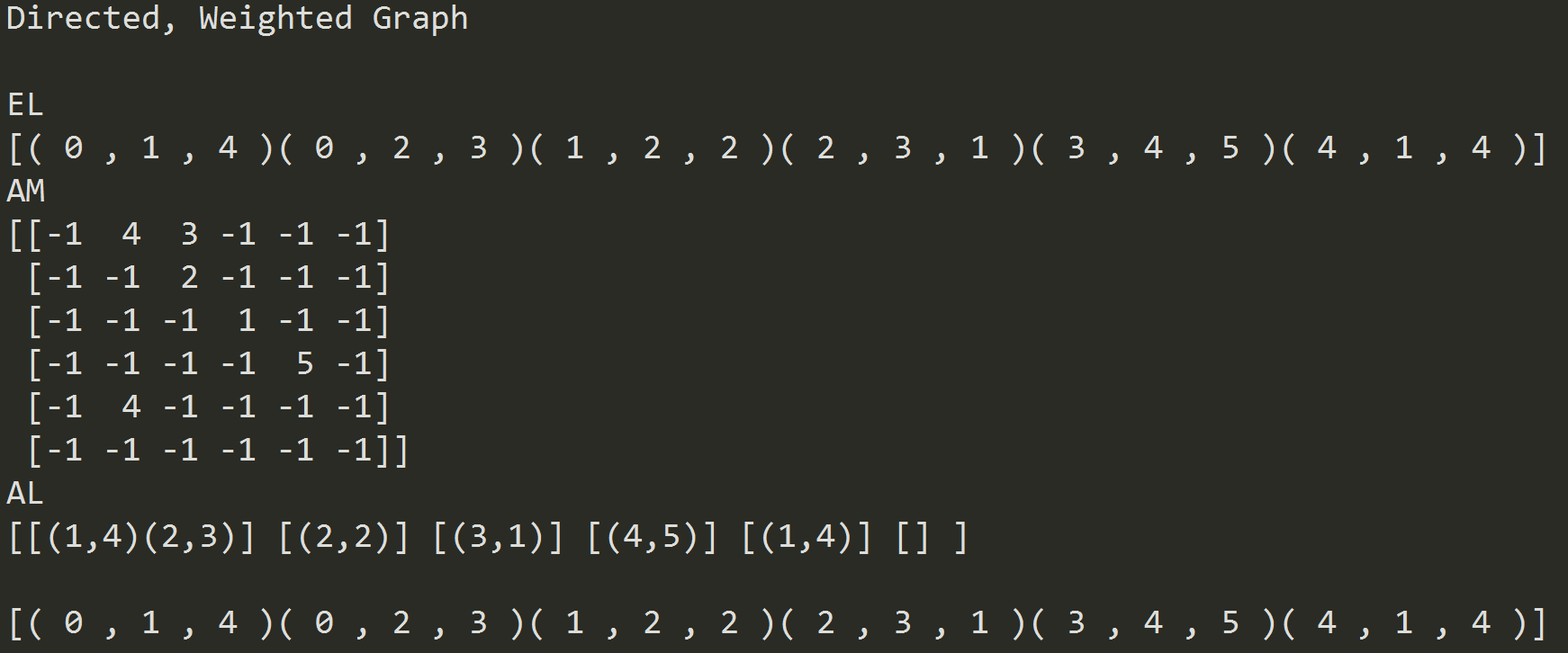
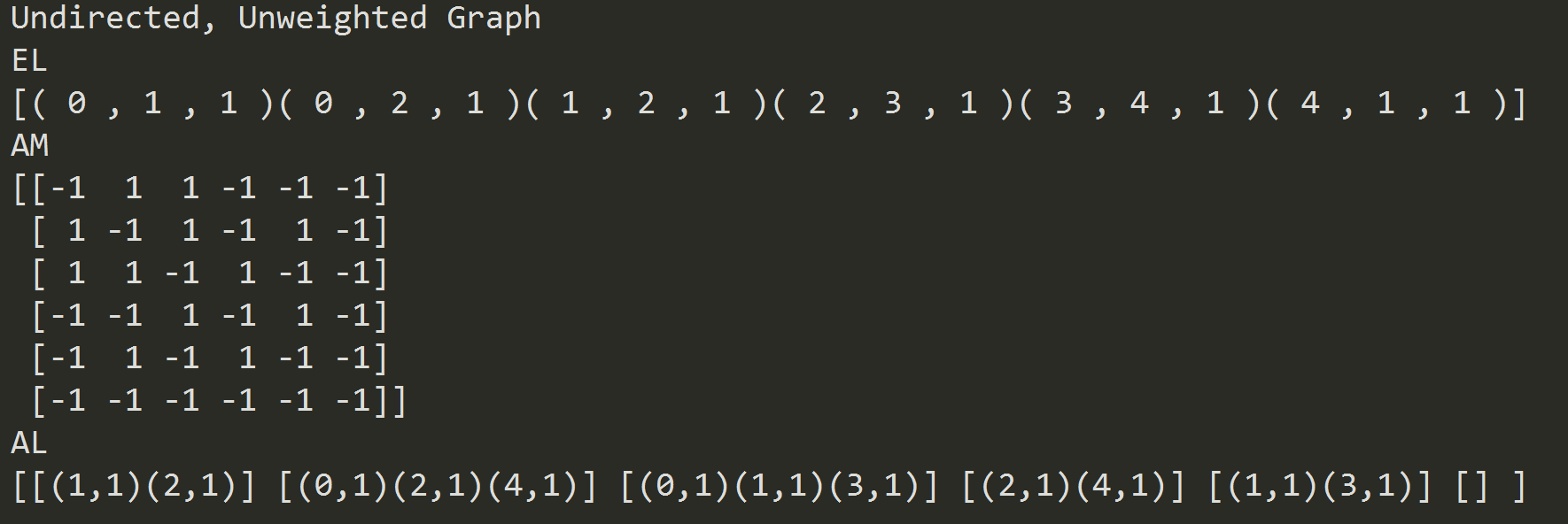
1.4 as\_AL, as\_EL representations for the AM graph implementation:



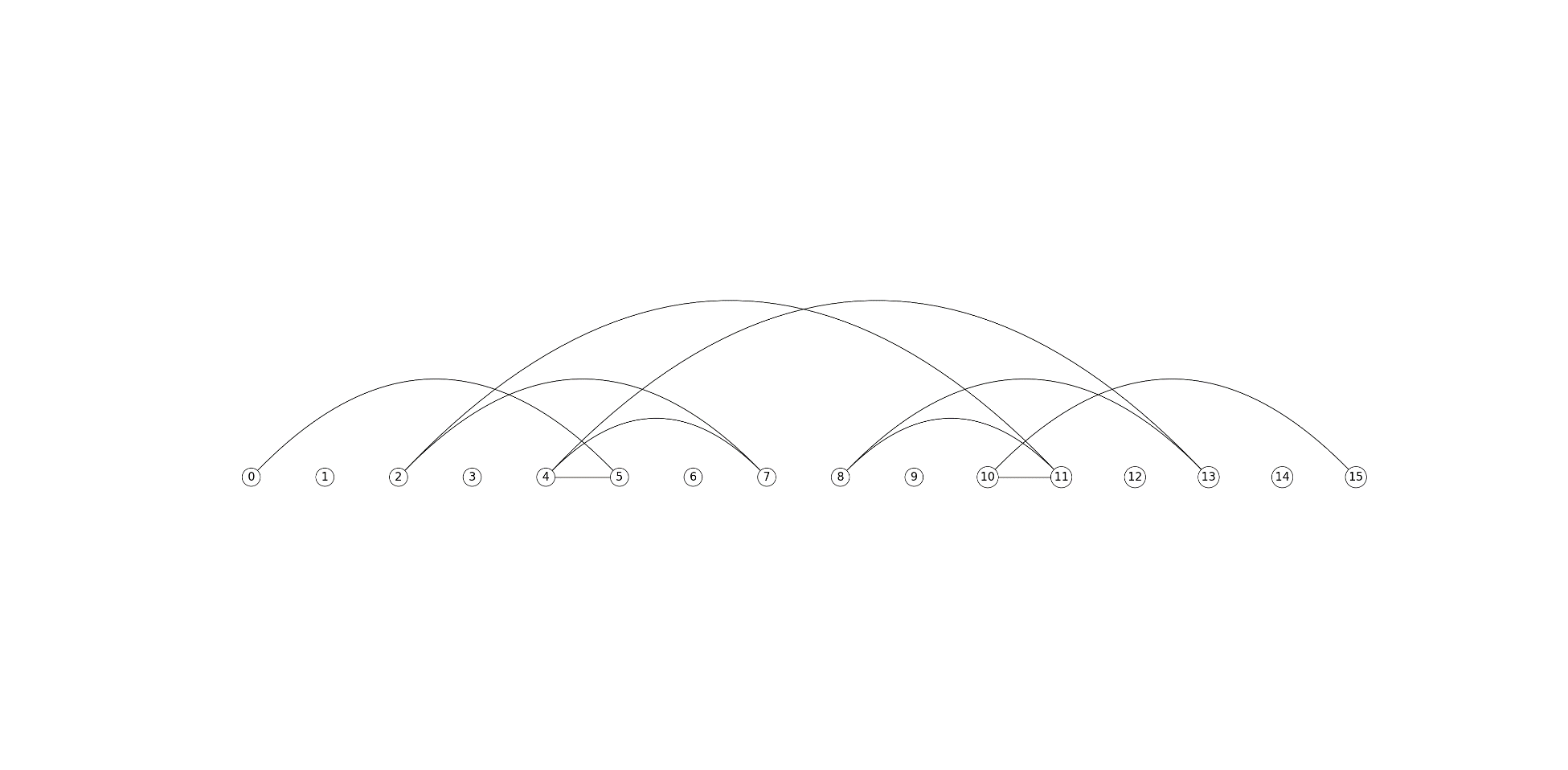


1.5 as\_AM, as\_AL representations for the EL graph implementation:

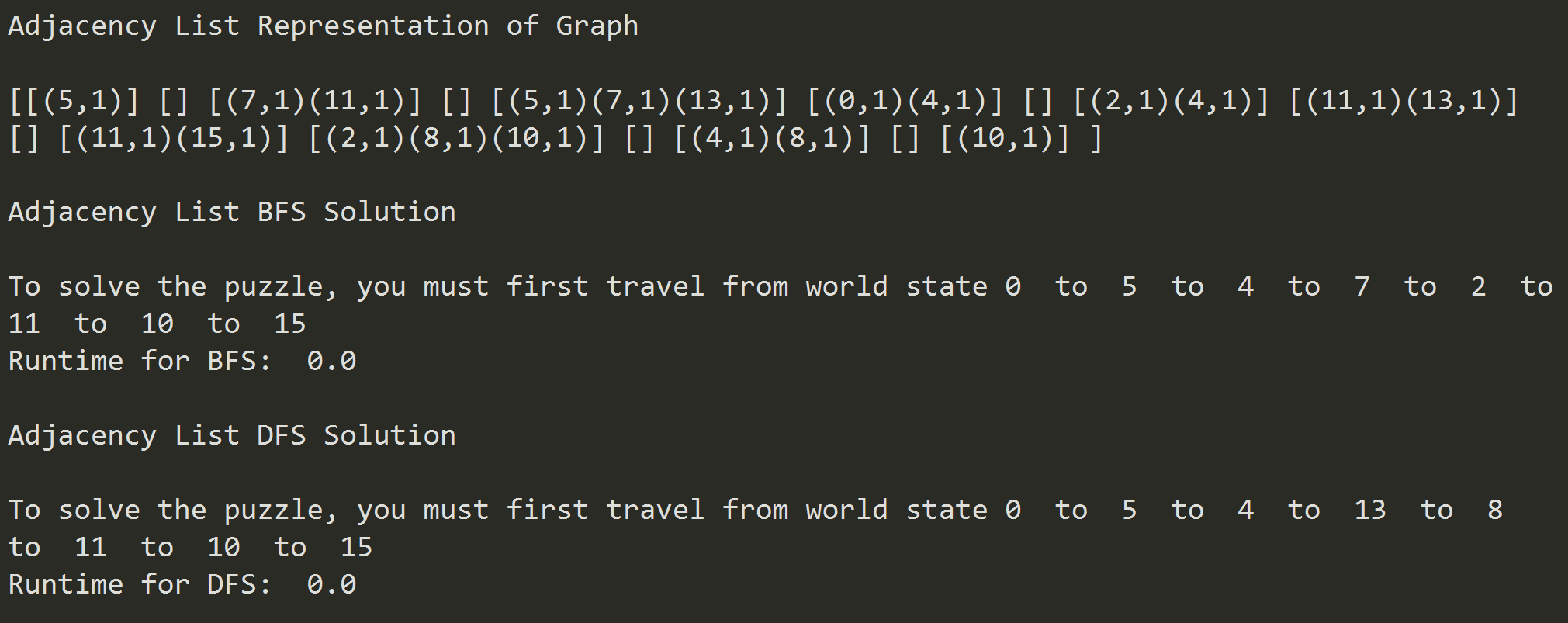
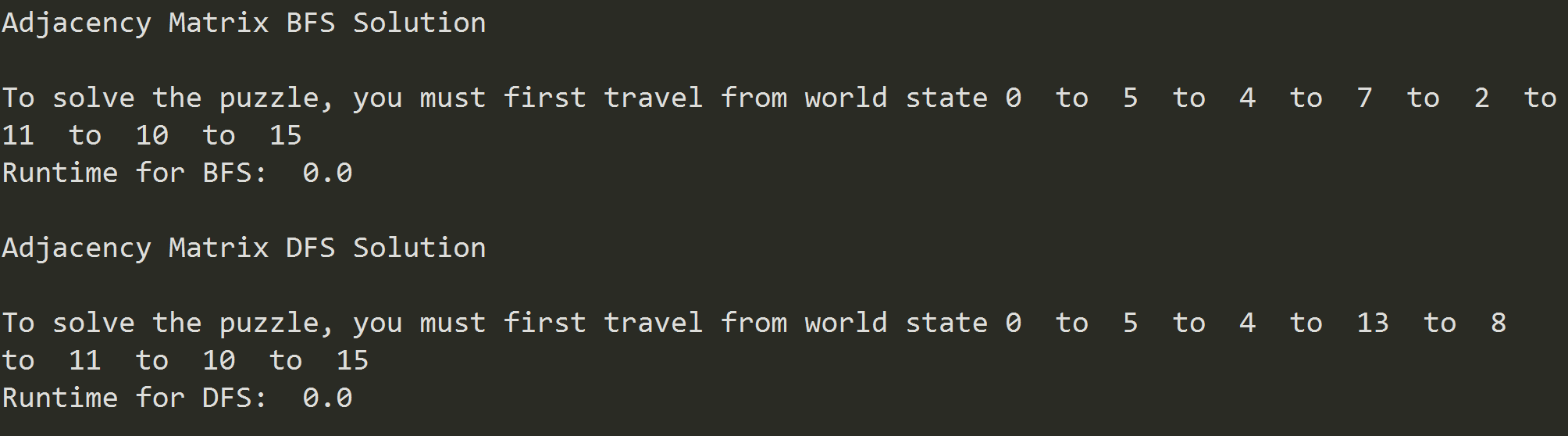
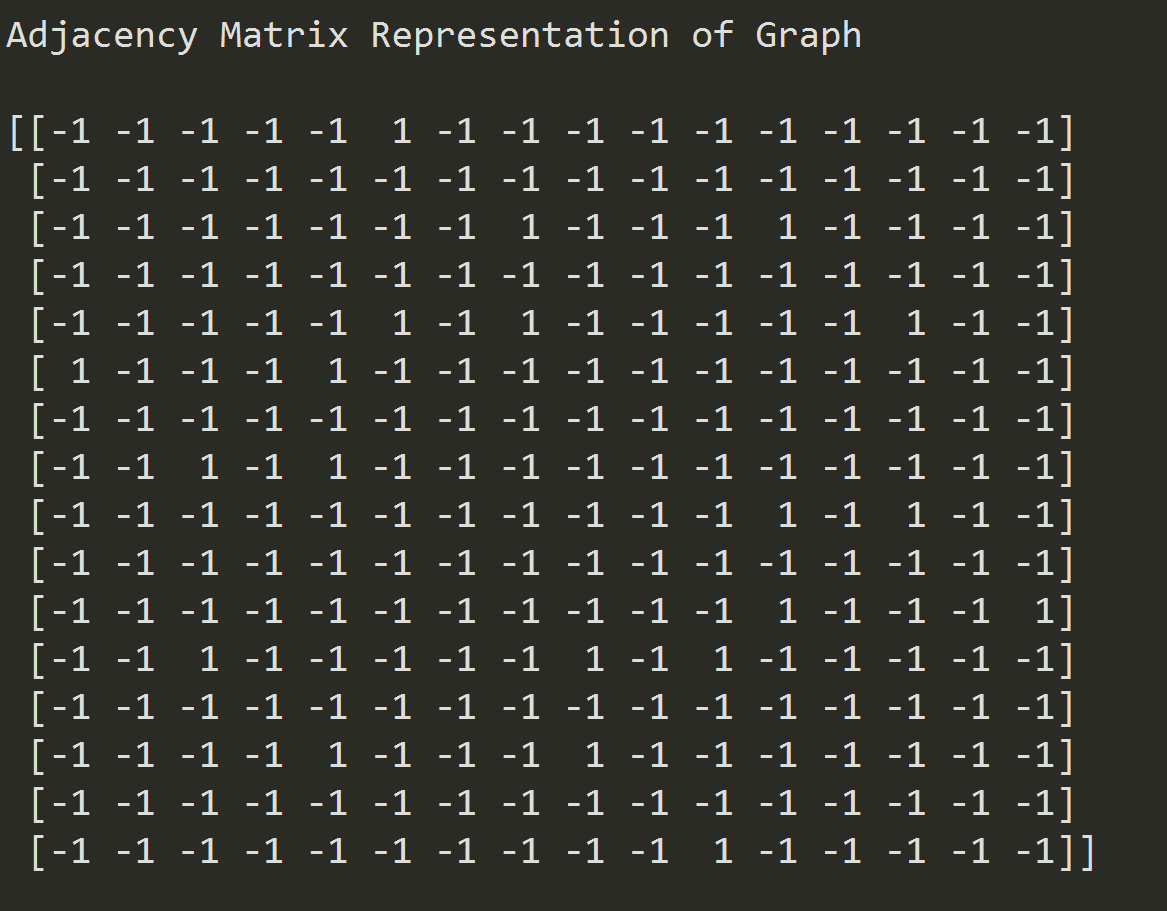
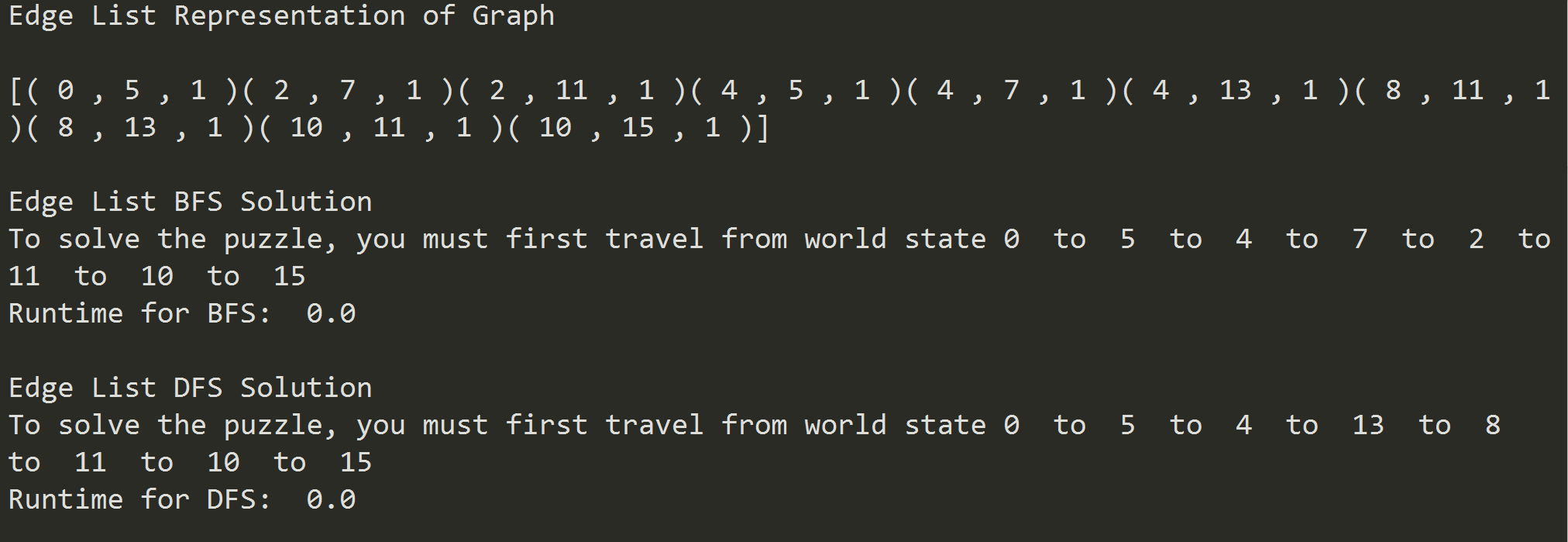




*Part 2: BFS/DFS Results and Puzzle Path:*

The following is a graph of the legal states and legal transitions between the legal states. 

The following is the output for the three graph implementations and the respective results for the BFS and DFS algorithms.

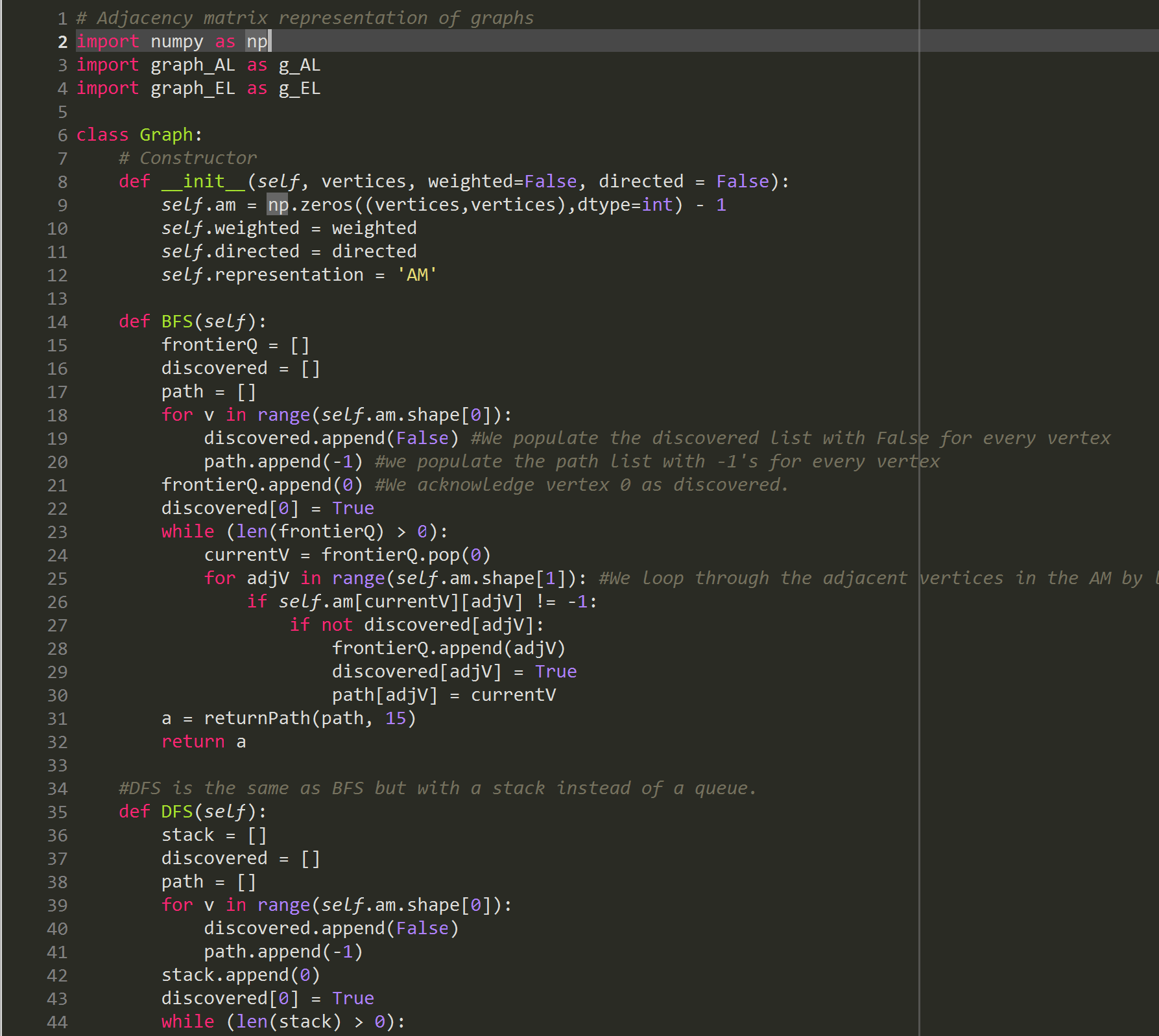
****

**Conclusion**

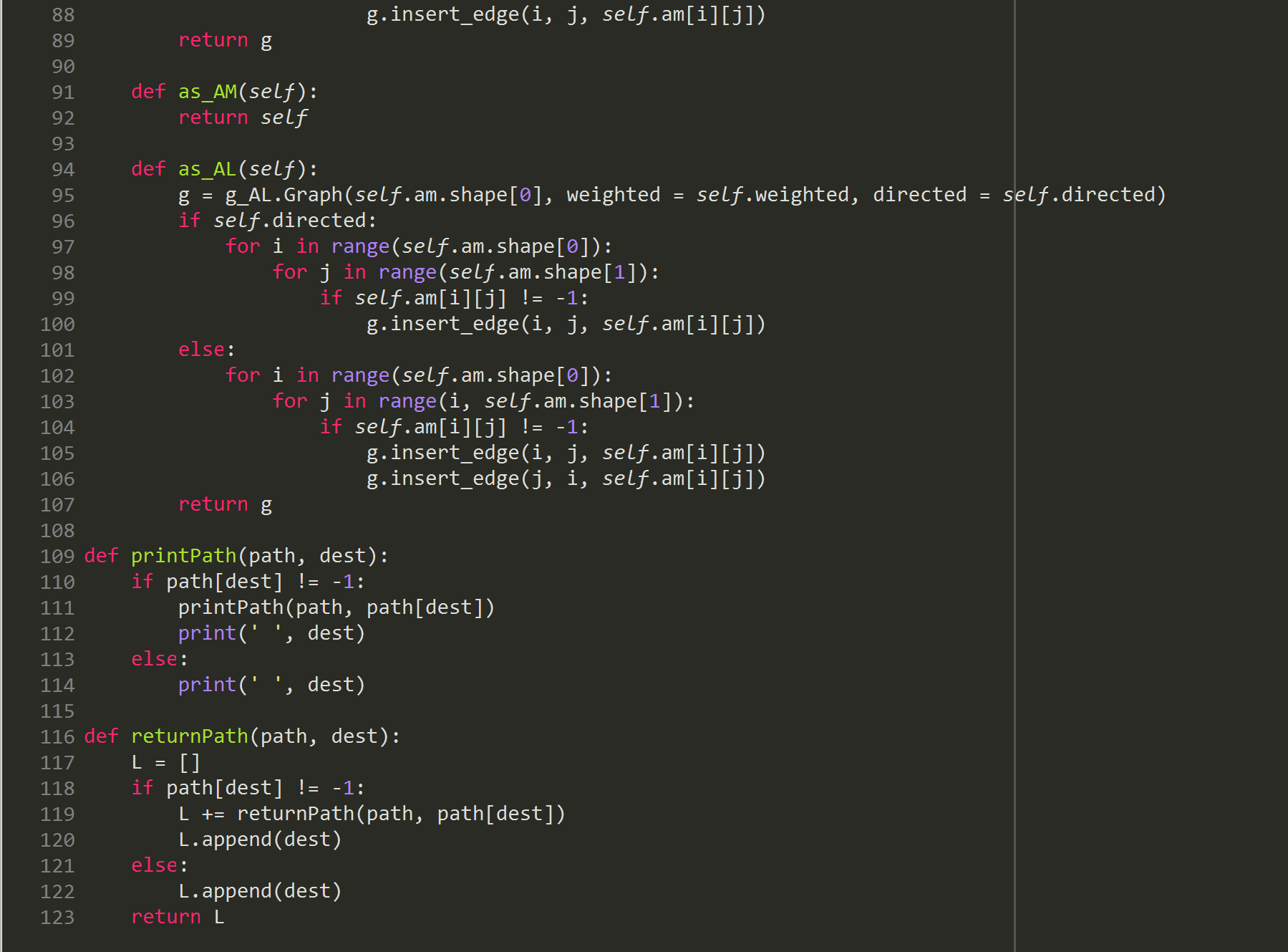
For this lab, I learned how to implement different representations of graphs like the edge list, adjacency list, and adjacency matrix. I practiced and learned how to insert edges into these different representations. I successfully implemented the DFS and BFS search algorithms for graphs. I also learned how to use graphs in a meaningful way – for example, to solve this puzzle.

**Appendix – Source Code**

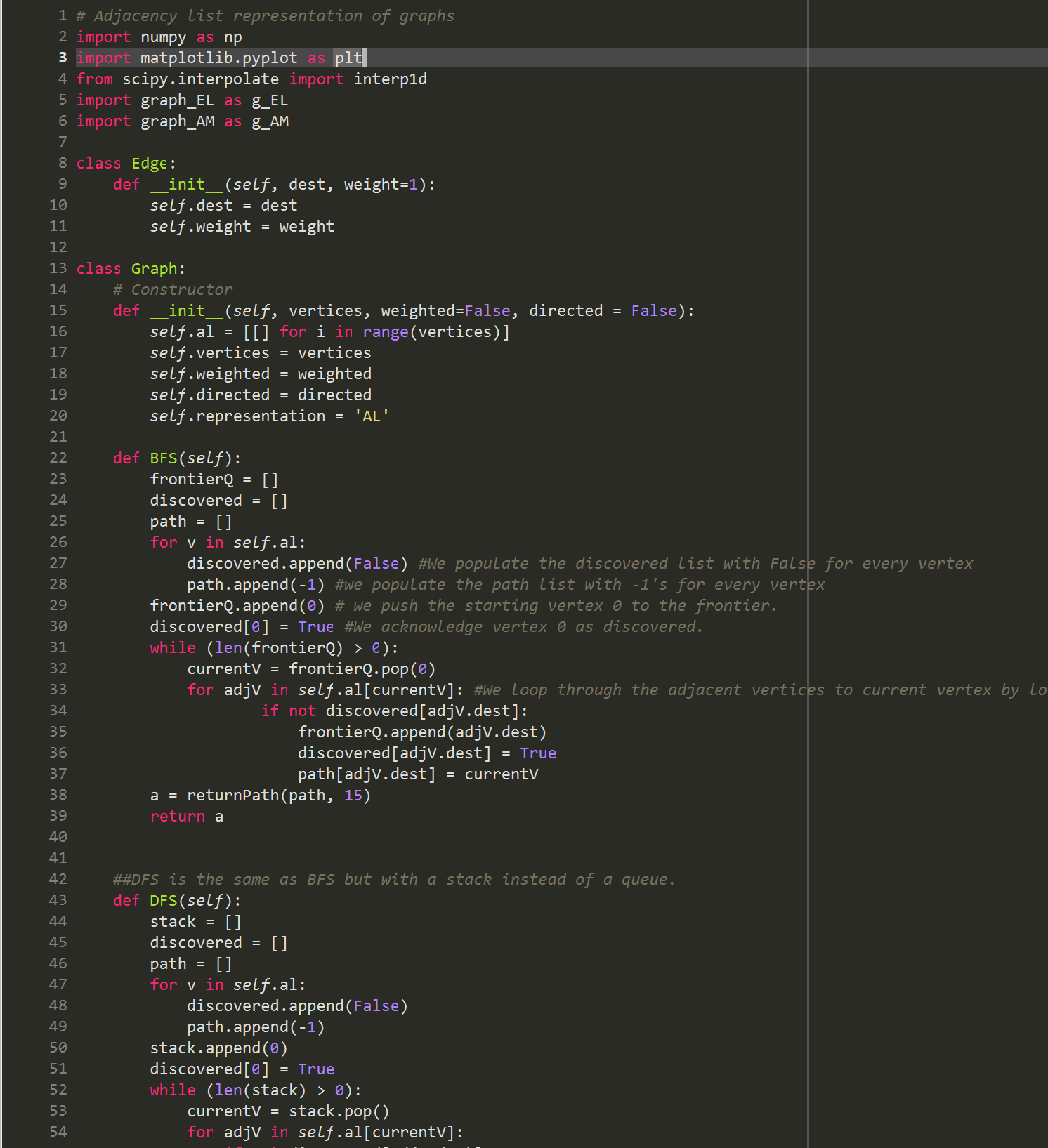
* **Graph\_AM.py:**

****

****

****

* **Graph\_AL.py**

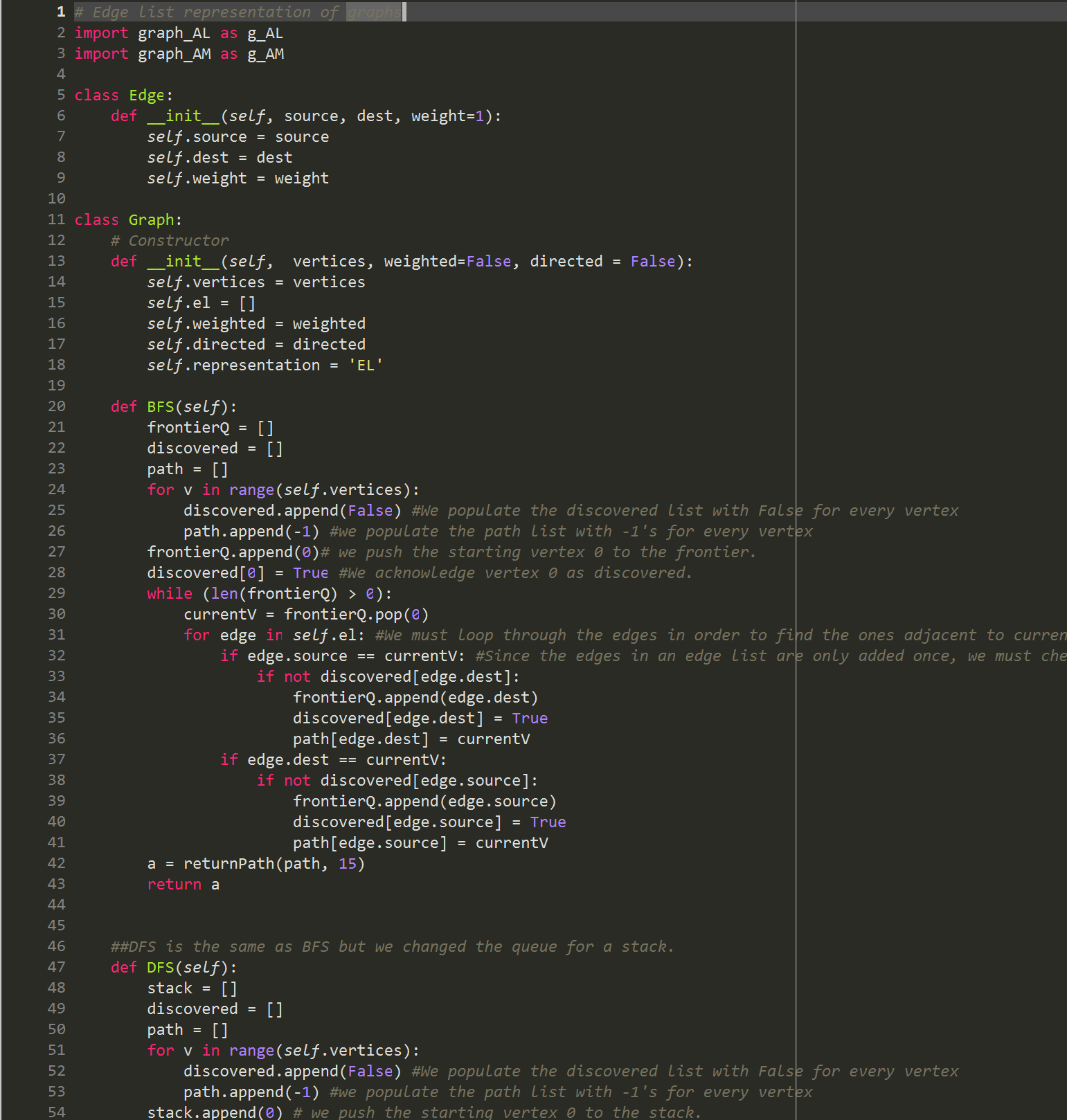
****

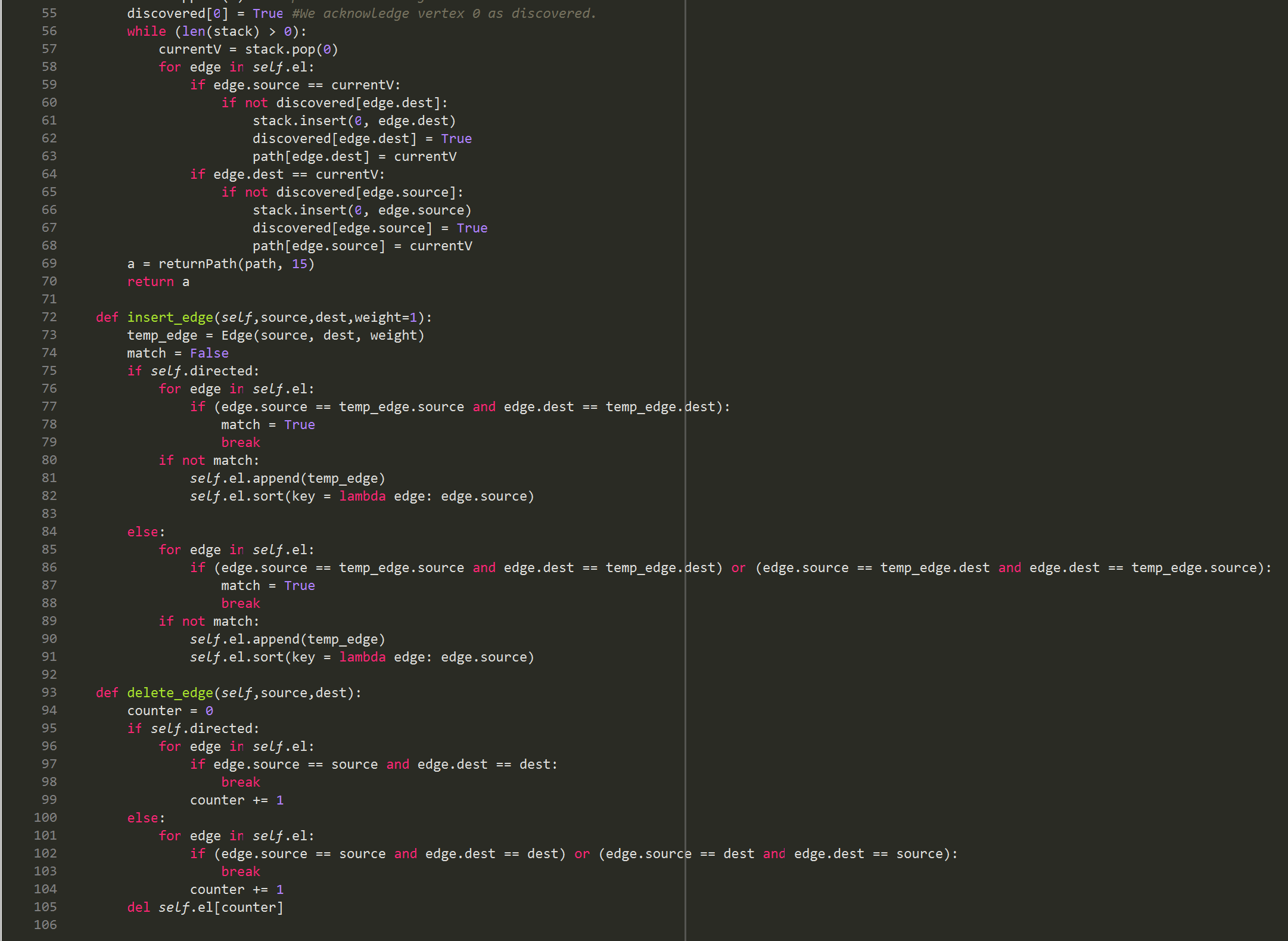
****

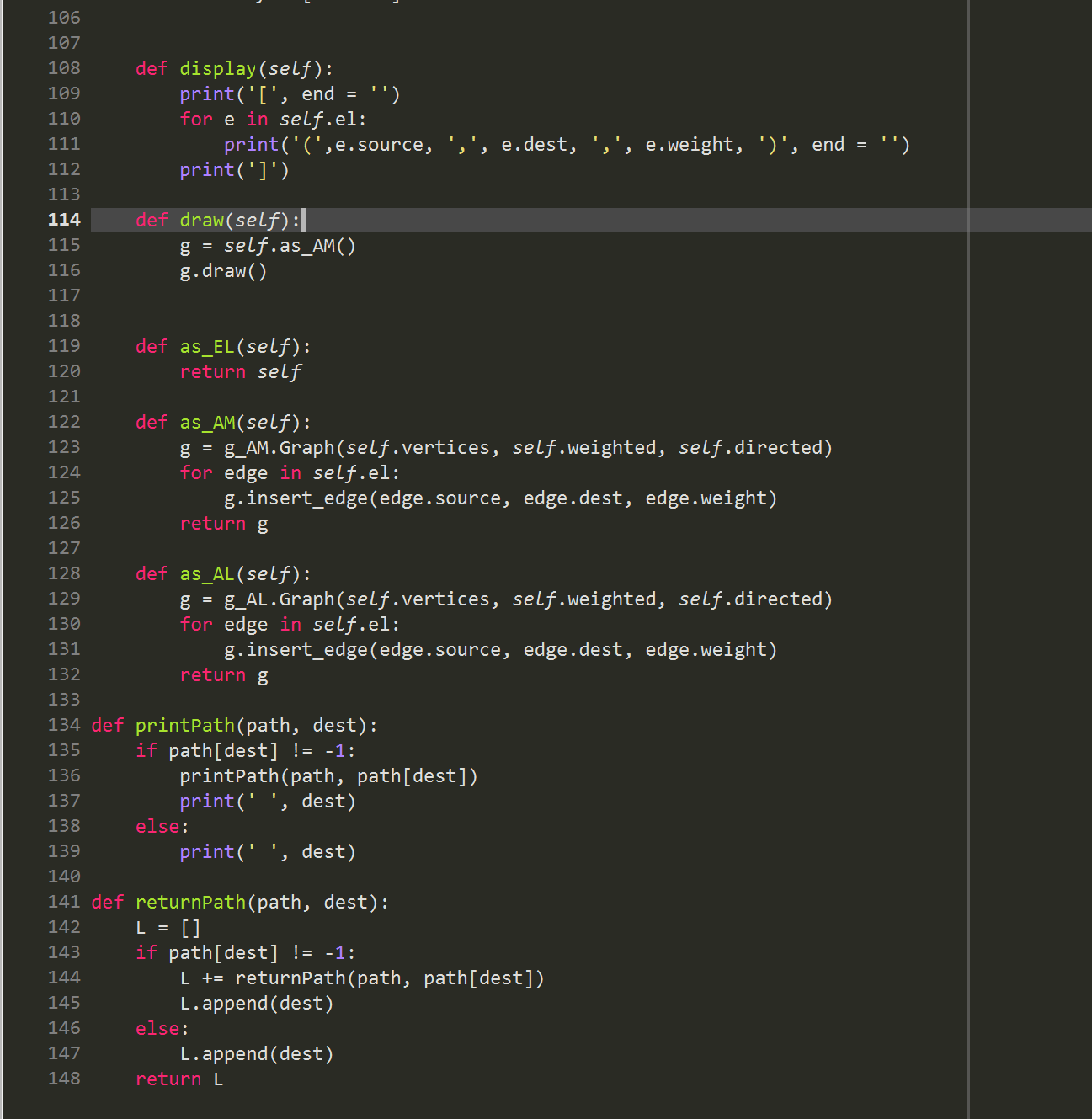
****

****

* **graph\_EL.py**

****

****

****

**Academic Honesty Certification**

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

Jonatan Contreras