Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 6 REPORT

Ahmet Yasir NACAK
161044042

Course Assistant: Fatma Nur Esirci

1 Common Algorithms

This section describes the plot_graph, is_cyclic and is_undirected methods. Since these 3 algorithms are used in all 3 parts, I choose to give information about them beforehand and describe the specifically needed algorithms in their specific parts(shortest_path: part1, is_connected: part2, DFS and BFS: part 3)

Plot Graph:
This method is pretty simple. I iterated through all of the vertices and looked for all of the vertices adjacent to it. After that, I printed the label of the adjacent vertex and the weight of the edge between them.

Is Undirected:
I iterated through all the possible edge pairs (n^2 loop) and tried to find a matching opposite direction edge for each node. This means that their starting points and end points are swapped but they have the same weight. If there's not such a match after all the inner iteration, I returned false. If the loop does not get interrupt by this return call, it reaches to end of the method where I return true which means that the graph is undirected.

Is Cyclic:
To find if a given graph is cyclic or not, we first need to know if the graph is directed or undirected. After learning that, for undirected graphs, we do a DFS of the given graph. For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don't find such an adjacent for any vertex, we say that there is no cycle. If the graph is directed, DFS of this graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by DFS. To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree.

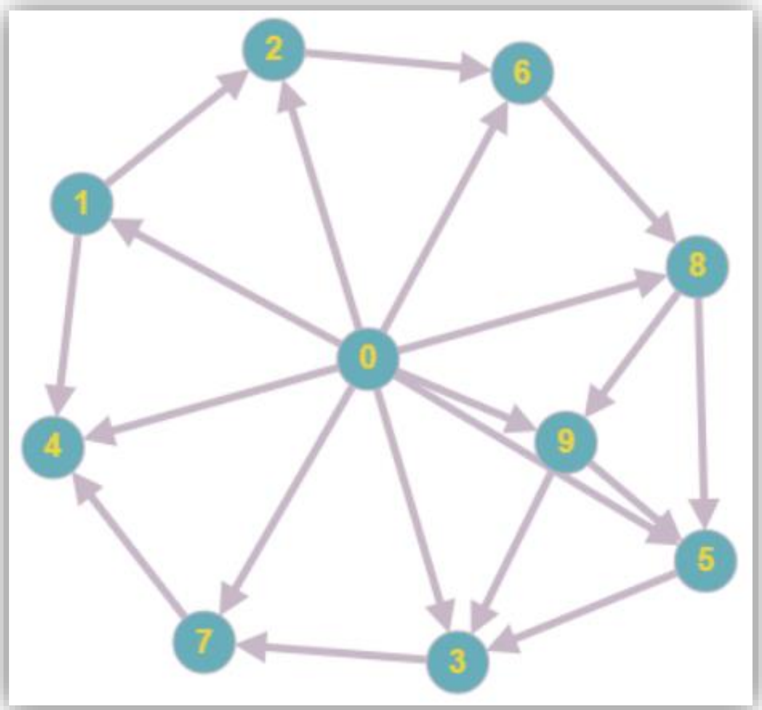
2 Q1

Question 1 wants us to create a directed acyclic graph with 10 vertices and 20 edges and test the methods that find if the graph is undirected and if the graph is acyclic. This question also wants us to test the shortest path method for 3 different vertex pairs.

2.1 Problem Solution Approach

Shortest Path:
Firstly, I created two arrays. One of them is for the distance for each vertex and the other is for determining if vertices are visited. After that, I set all of the distances to maximum integer value except for the starting vertex. The starting vertex counts as visited and has the distance of 0 because the search starts from there. After that, I ran a loop that does not end unless the destination is reached. This loop looks for all the unvisited adjacent vertices and makes their cost the weight of the edge between current vertex and the unvisited adjacent vertex + cost of the current vertex if its smaller than the cost of the adjacent vertex. After that, the algorithm chooses the vertex with minimum cost to continue and loops again and again. This algorithm terminates if such a path exists from start to finish. After the path cost is found, I backtracked from end to start with trying ways to come back with certain cost numbers. After that, I pushed all the values to a stack and popped them to a vector and returned the resulting path.

2.2 Test Cases



This is the graph that is being tested. Since the weight is being randomized, I did't put any value for that in this visual representation.

Show that this func results ->

- plot_graph:

```
Plotting Graph:
adjacent vertices of 0:
       1 (weight:8)
       2 (weight:2)
       3 (weight:5)
       4 (weight:1)
       5 (weight:3)
       6 (weight:8)
       7 (weight:9)
       8 (weight:9)
       9 (weight:1)

adjacent vertices of 1:
       2 (weight:9)
       4 (weight:4)

adjacent vertices of 2:
       6 (weight:7)

adjacent vertices of 3:
       7 (weight:10)

adjacent vertices of 4:

adjacent vertices of 5:
       3 (weight:9)

adjacent vertices of 6:
       8 (weight:3)

adjacent vertices of 7:
       4 (weight:4)

adjacent vertices of 8:
       5 (weight:3)
       9 (weight:1)

adjacent vertices of 9:
       3 (weight:7)
```

- is_undirected

```
Is this graph undirected: false
```

- is_acyclic_graph

```
Is this graph acyclic: true
```

- shortest_path (use least 3 different label pair)

```
Shortest path from node 0 to 7

0 7

Shortest path from node 6 to 4

6 8 9 3 7 4

Shortest path from node 1 to 9

1 2 6 8 9
```

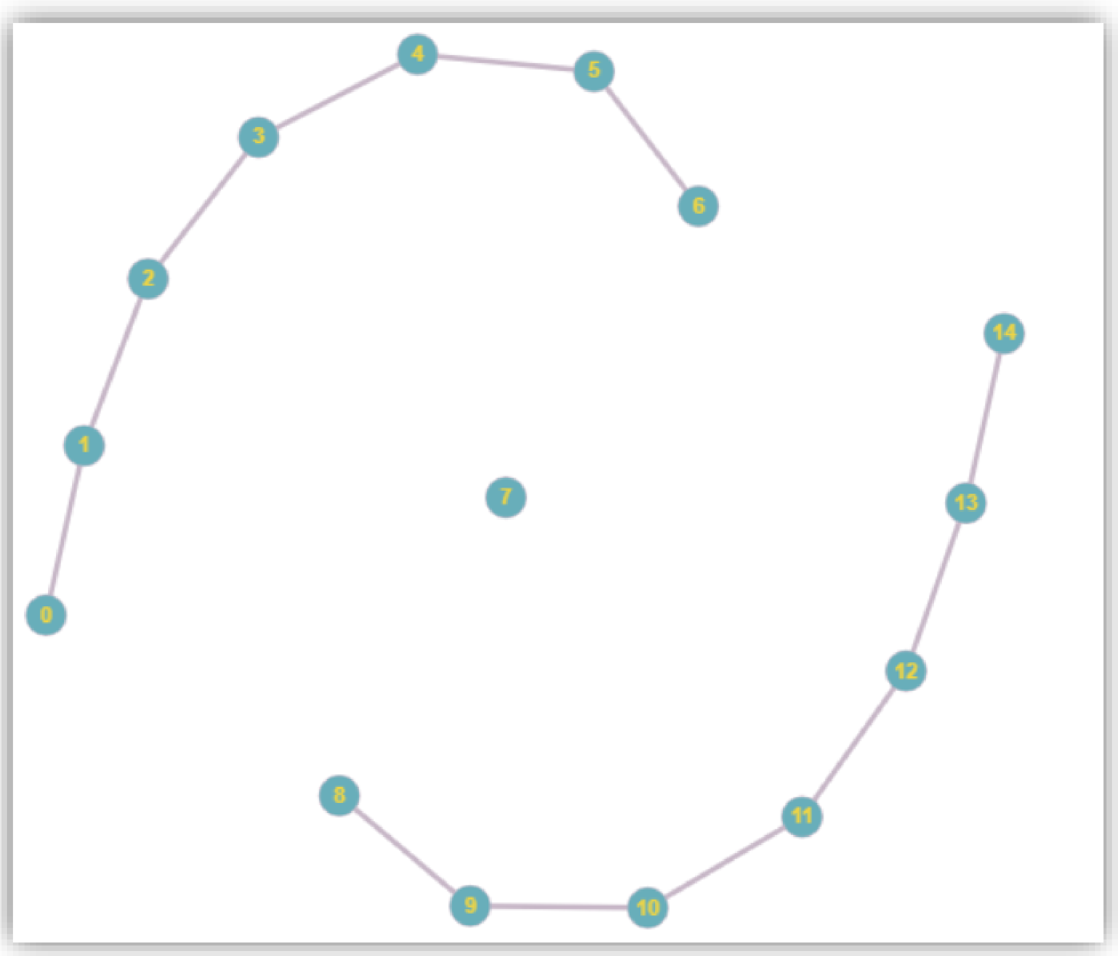
3 Q2

Question 2 wants us to create a undirected acyclic graph with 15 vertices with no weighted edges and test the methods that find if the graph is undirected and if the graph is acyclic. This question also wants us to test the is_connected method for 3 different vertex pairs.

3.1 Problem Solution Approach

Is Connected:
For this method, I firstly look for the starting vertex and the end vertex in the graph. If one of the vertices is not present in the graph, I throw a NoSuchElementException. After that, I check if the start and the end are the same. If this is the case, I return true. If any of the conditions are not true with the method call, I do a simpler version of BFS and keep track of the each vertex I pass through. If the vertex that is being searched is one of the vertices that the BFS goes through, I return true. If the method does not reach a case of that, I return false.

3.2 Test Cases



This is the graph that is being tested. It is undirected and acyclic with no weight on edges and a node count of 15. It also has 3 parts which makes us able to test the is_connected method properly. Show that this func results ->

- plot_graph

```
adjacent vertices of 0:
	1 (weight:1)
adjacent vertices of 1:
	0 (weight:1)
	2 (weight:1)
adjacent vertices of 2:
	1 (weight:1)
	3 (weight:1)
adjacent vertices of 3:
	2 (weight:1)
	4 (weight:1)
adjacent vertices of 4:
	3 (weight:1)
	5 (weight:1)
adjacent vertices of 5:
	4 (weight:1)
	6 (weight:1)
adjacent vertices of 6:
	5 (weight:1)
adjacent vertices of 7:
adjacent vertices of 8:
	9 (weight:1)
adjacent vertices of 9:
	8 (weight:1)
	10 (weight:1)
adjacent vertices of 10:
	9 (weight:1)
	11 (weight:1)
adjacent vertices of 11:
	10 (weight:1)
	12 (weight:1)
adjacent vertices of 12:
	11 (weight:1)
	13 (weight:1)
adjacent vertices of 13:
	12 (weight:1)
	14 (weight:1)
adjacent vertices of 14:
	13 (weight:1)
```

- is_undirected

```
Is this graph undirected: true
```

- is_acyclic_graph

```
Is this graph acyclic: true
```

- is_connected function (use least 3 different label pair)

```
Are nodes 0 and 6 connected : true

Are nodes 1 and 13 connected : false

Are nodes 7 and 8 connected : false
```

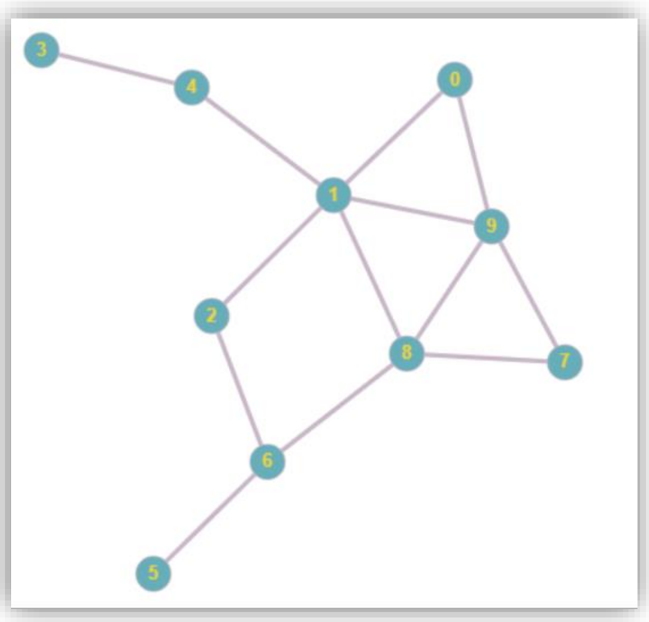
4 Q3

Question 2 wants us to create a undirected cyclic graph with 10 vertices with no weighted edges and test the methods that find if the graph is undirected and if the graph is acyclic. This question also wants us to do Depth First and Breadth First searches in the graph and print out the spanning trees of those searches.

4.1 Problem Solution Approach

DFS and BFS:
I created both DFS and BFS nearly the same way except for the main data structure utilized by it. The general approach is to create an array that tells you if a node vertex is visited or not and iterate through all the adjacent vertices of the starting node, if the vertex is not visited, add it to the data structure and make it visited. Remove the element from the data structure and continue on until the data structure is empty. The data structure in DFS is a stack and the data structure in BFS is a queue.

4.2 Test Cases



Show that this func results ->

- plot_graph

```
Plotting Graph:
adjacent vertices of 0:
	1 (weight:1)
	9 (weight:1)
adjacent vertices of 1:
	0 (weight:1)
	2 (weight:1)
	4 (weight:1)
	8 (weight:1)
	9 (weight:1)
adjacent vertices of 2:
	1 (weight:1)
	6 (weight:1)
adjacent vertices of 3:
	4 (weight:1)
adjacent vertices of 4:
	1 (weight:1)
	3 (weight:1)
adjacent vertices of 5:
	6 (weight:1)
adjacent vertices of 6:
	2 (weight:1)
	5 (weight:1)
adjacent vertices of 7:
	8 (weight:1)
	9 (weight:1)
adjacent vertices of 8:
	1 (weight:1)
	7 (weight:1)
	9 (weight:1)
adjacent vertices of 9:
	0 (weight:1)
	1 (weight:1)
	7 (weight:1)
	8 (weight:1)
```

- is_undirected

```
Is this graph undirected: true
```

- is_acyclic_graph

```
Is this graph acyclic: false
```

- DepthFirstSearch (Show that spanning tree)

```
Spanning Tree of DFS:

0

  1

    2

      6

        5

    4

      3

  9

    1

    7

    8

      1
```

- BreathFirstSearch (Show that spanning tree)

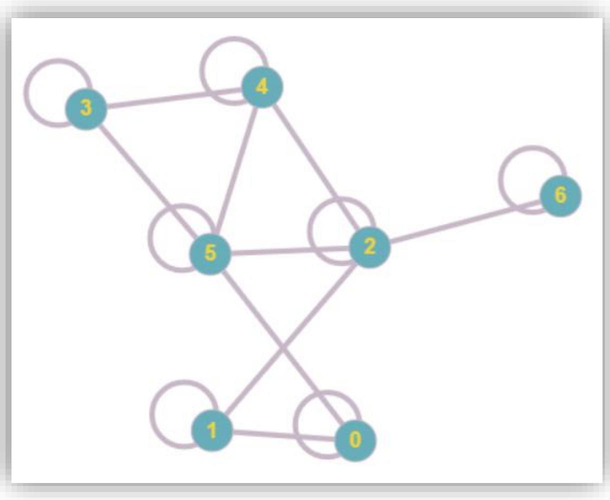
```
Spanning Tree of BFS:

0

  1

    2

      6

        5

    4

      3

    8

  9

    7
```

5 Q4

Differences of DFS and BFS

DFS	BFS
Starts the traversal from the root vertex and explores the search as far as possible from the root vertex i.e. depth wise.	Starts traversal from the root vertex and then explores the search in the level by level manner i.e. as close as possible from the root vertex.
Is being done using stack (LIFO)	Is being done using queue (FIFO)
Faster than BFS and requires less memory	Slower than DFS and requires more memory
Being used in cycle detection	Being used in path detection

The Given Graph:



Spanning Tree of DFS Starting From Node 0:

```
Spanning Tree of DFS:
0
  1
  5
    2
      1
      6
    3
    4
      2
      3
```

Spanning Tree of BFS Starting From Node 0:

```
Spanning Tree of BFS:
0
  1
    2
      6
  5
    3
    4
```