Data Structures, Lab
CRN: 19897

Salvador Robles Herrera
ID: 80683116

# Lab #6 Report

## Problem 1

This problem consisted on implementing Kruskal's algorithm and coding our implementation of the Graph and Disjoint Set Forest data structures. Note that the Kruskal's algorithms is a way of getting the minimum weight by connecting a graph with edges such that there is a path between every node to any other node in the graph.

For this problem I used a graph with an Adjacency List representation.

- **class DisjointSetForest:** It is a very important part of Kruskal's algorithm to check if adding an edge would create a cycle on the graph, that's why a Disjoint Set Forest is needed mainly a way of finding the root of any given node and a way of uniting any two given nodes. A DSF consists of an array of n elements with the respective root values of those, *find(self, a)* method: It has a recursive helper method "*helper_find*" that checks if the value is negative number (implying that it is root and return the value of the node) or continue to the node pointed by the value of the array until a negative number is found.
- **class Edge:** Every edge has a source and destination node, along with the weight of the edge.
- **class GraphAL**: This graph contains if the graph is weighted or directed and the list of lists (self.al) that will contain Edge objects, note that the list will order the nodes with an empty list for each node. To insert a vertex with the method "*insert_vertex(self)",* it is appended an empty list to self.al. To insert an edge with the method "*insert_edge(self, source, dest, weight=1)"* it appends an object Edge with the given attributes source, dest and weight. It is appended on the correct list in self.al. An important method that will be used in the second problem in this Lab is "*vertices_reachable_from_u(self,u)"* which given a given node it returns a list with all the possible destinations connected by an edge with that given node. It does this by appending to a list all the destinations from the list with index u of the self.al.
- **kruskal(graph)** method: This is the main method to solve the problem, the first thing to do is to sort all the edges of the graph by first getting all the edge by traversing through graph.al and appending it to a list, then use list.sort() to sort it. Then create a min_spanning_tree which is a graph of type AL with the number of vertices equal to the number of vertices of the given graph. Then create a Disjoint Set Forest with the length of the number of nodes in the graph, this will help check if an edge creates a cycle or not. Then the Kruskal's algorithm traverses through all the edges from the sorted edges and then add the edge to the min_spanning_tree if it does not create cycle. At the end return the min_spanning_tree.

This are the graphs that I used, which gave the expected graph (I used display and draw to visually see that it works):

Salvador Robles Herrera
ID: 80683116

```
print("FIRST GRAPH")
g = GraphAL(6, weighted=True)
g.insert_edge(0, 1,6)
g.insert_edge(0, 2,4)
g.insert_edge(1, 2,20)
g.insert_edge(2, 3,5)
g.insert_edge(3, 4,50)
g.insert_edge(4, 1,2)
g.insert_edge(3, 5,3)
g.insert_edge(4, 5,70)
g.display()
g.draw()

g1 = kruskalAlg(g)
g1.draw()
print("Expected: [[(2,4)(1,6)] [(4,2)(0,6)] [(0,4)(3,5)] [(5,3)(2,5)] [(1,
2)] [(3,3)] ]")
print("What I got: ")
g1.display()

print()
print()

print("SECOND GRAPH")
g2 = GraphAL(6, weighted=True)
g2.insert_edge(0, 1,1)
g2.insert_edge(1, 2,2)
g2.insert_edge(2, 3,3)
g2.insert_edge(3, 4,4)
g2.insert_edge(4, 5,5)
g2.insert_edge(4, 1,20)
g2.insert_edge(3, 5,30)
g2.insert_edge(2, 5,70)
g2.insert_edge(3, 1,20)
g2.insert_edge(1, 5,30)
g2.insert_edge(2, 4,70)
g2.display()
g2.draw()

g3 = kruskalAlg(g2)
g3.draw()
print("Expected: [[(1,1)] [(0,1)(2,2)] [(1,2)(3,3)] [(2,3)(4,4)] [(3,4)(5,
5)] [(4,5)] ]")
print("What I got: ")
g3.display()
```

Salvador Robles Herrera
ID: 80683116

# Problem 2

This weird problem requires us to get a path or way to get a chicken, a fox and a grain across to the right side of the river. Note that if the fox and chicken are alone, then the fox will eat the chicken and if the chicken and grain are alone then the chicken will eat the grain, also note that the person crossing can only cross with one of them at a time. The thing is that to solve this problem it is needed to solve it with the graph representations of Adjacency List, Adjacency Matrix and Edge List, and each of them needs a Breath First Search and Depth Search First traversal algorithm to get the path to solve this problem.

The main idea of this problem is to have four bits representing with bit0 the location of the fox, bit1 the location of the chicken, bit2 the location of the grain and bit3 the location of the person. A bit with value zero will represent that the element is on the left side of the river and a one will represent that the element is on the right side of the river.

For this problem I needed to implement a class of graph type EL and AM (AL is already done in the first problem) like this:

- **class GraphAM**: This looks like a 2D array with the rows representing the nodes being sources in the edges and the columns representing the destinations, the value of that entry will be the weight of the edge, if it's zero then there is no edge. To insert an edge it just adds updates the weight of that entry with the value passed with the method *"insert_edge(self, src, dest, weight=1)".* An important method is *"vertices_reachable from(self, u)"* by traversing the row of the given u and then appending to a list if the entry value is not zero.
- **class GraphEL**: The graph has all the edges of the graph, with the source, destination and weight of every edge. This contains the number of vertices as attribute, as like if the graph is weighted or directed. The *"num_vertices(self)"* is done by traversing through all the edges and then checking source and destinations and putting everything into a set. An important method is "*vertices_reachable from(self, u)"* by traversing through all the edges and then checking if the given u is the destination or source of the node and if that's the case then appending it to a list and the return it.

Too traverse the graph it's needed a Breath First Search algorithm b_f_s and Depth First Search algorithm d_f_s.

- **d_f_s_AL(graph, start)** method: This is done by creating a visited_order, visited and path lists. A stack is used to keep track of the nodes that are adjacent to a certain node and have not been visited. While the stack has elements, traverse through all the adjacent vertices of the popped element of the stack and then add it to the stack and path if the adjacent vertex has not been visited. At the end return the visited_order.
- **b_f_s_AL(graph, start)** method: This is sort of like d_f_s but instead of using a stack it uses a queue. Using the same idea of keep going until the queue is empty and traversing the adjacent vertices of a given node.

Salvador Robles Herrera
ID: 80683116

- **b_f_s_EL(graph, start)**, **d_f_s_EL(graph, start)**,**b_f_s_AM(graph, start)**, **d_f_s_AM(graph, start)** methods: are almost the same as the previous two, just with the difference of getting the number of vertices of the graph.

Also, it is very important for the main algorithm to have these methods:

- **validPosition(n)** method: Given a node which is represented by a number from 0 to 15, it is invalid if the chicken and fox, or if the grain and the chicken are together without being supervised by the person. Otherwise it is valid.
- **validTransition(i,j)** method: Given two valid nodes it checks if the person is in the same position on both nodes, meaning that there wasn't a change and therefore invalid. Then, it counts the number of one to zero and zero to one changes in position of the elements, if there was at most one change of the fox, chicken and grain in position and it was a valid change (if the person ends with a value of one there can't be a change of one to zero, that's the opposite way) and also if the opposite changes in the other directions is not zero than the transition is invalid. Otherwise, the transition is valid.
- **get_bit(n, pos)** method: It returns True or False if the value of the bit at any position is one or zero.
- **get_fox(n), get_chicken(n), get_grain(n) and get_person(n) methods:** Returns True of False if the element is at position 1, or in other words, if it's on the right side of the river.

The main algorithm will be something like this. Note that the methods for EL, AM have the same logic, with the only difference of the type of the graph being returned.

- **fox_chicken_grain_AL** method: At first a graph is created with two lists of valid positions and edges in the graph, as well with a graph of type, in this case, AL. It goes over all the 16 possible states of the nodes and then if its valid (using the method validPosition) it appends it to the first list. Then it checks every possible edge from every valid position to every valid position and then checks if the the edge is valid (using the method validTransition), the edge is inserted into the graph. At the end the graph is returned.

This are my test cases, testing all the three representations with all the traversal algorithms (6 in total):

```
print("Expected BFS: ", [0, 1, 2, 4, 5, 10, 11, 13, 14, 15])
print("Expected DFS: ", [0, 10, 2, 14, 4, 13, 5, 15, 1, 11])
print("ADJACENCY LIST")
graphFCG1 = fox_chicken_grain_AL()
graphFCG1.display()
graphFCG1.draw()
visited_order1 = b_f_s_AL(graphFCG1, 0)
print("Visited order BFS: ", visited_order1)
visited_order2 = d_f_s_AL(graphFCG1, 0)
print("Visited order DFS: ", visited_order2)

print()
print()
```

Salvador Robles Herrera
ID: 80683116

```
print("ADJACENCY MATRIX")
graphFCG2 = fox_chicken_grain_AM()
graphFCG2.display()
graphFCG2.draw()
visited_order3 = b_f_s_AM(graphFCG2, 0)
print("Visited order3: ", visited_order3)
visited_order4 = d_f_s_AM(graphFCG2, 0)
print("Visited order4: ", visited_order4)

print()
print()
print("EDGE LIST")
graphFCG3 = fox_chicken_grain_EL()
print("EDGES OF GRAPHHHH")
print(graphFCG3.el)
visited_order5 = b_f_s_EL(graphFCG3, 0)
print("Visited order5: ", visited_order5)
visited_order6 = d_f_s_EL(graphFCG3, 0)
print("Visited order6: ", visited_order6)
```