Minimum Spanning Trees

```
# Minimum Spanning Trees are data structures that have
# the minimum required number of edges to connect all
# vertices. Will investigate how to create one using the
# Depth First Search.

# The major difference between a DFS and a MST is that the
# MST must record the edges traveled. You'll know you have
# created a MST when the number of edges equals the number
# of vertices - 1.

class MyStack:
    def __init__(self, size):
        self.size = size
        self.my_stack = [0] * self.size
        self.top = -1

    def push(self, val):
        self.top += 1
        self.my_stack[self.top] = val

    def pop(self):
        self.top -= 1
        return self.my_stack[self.top + 1]

    def peek(self):
        return self.my_stack[self.top]

    def is_empty(self):
        return self.top == -1


class Vertex:
    def __init__(self, name):
        self.name = name
        # Used for searching
        self.visited = False


class Graph:
    def __init__(self):
        self.max_vertices = 10
        self.vertex_list = [0]*10
        self.adjacency_matrix = [[0] * self.max_vertices for i in range(self.max_vertices)]
        self.vertex_count = 0
        self.the_stack = MyStack(20)

    def add_vertex(self, name):
        self.vertex_list[self.vertex_count] = Vertex(name)
        self.vertex_count += 1

    def add_edge(self, first, last):
```

```
        self.adjacency_matrix[first][last] = 1
        self.adjacency_matrix[last][first] = 1

    def print_vertices(self):
        for i in self.vertex_list:
            if isinstance(i, int):
                print(0)
            else:
                print(i.name)

    def print_edges(self):
        for row in self.adjacency_matrix:
            for elem in row:
                print(elem, end=' ')
            print()

    # NEW print vertex name
    def print_vertex(self, index):
        print(self.vertex_list[index].name, end="")

    def get_next_unvisited_vertex(self, curr_vertex):
        for i in range(0, self.vertex_count):
            if self.adjacency_matrix[curr_vertex][i] == 1 and self.vertex_list[i].visited is False:
                return i
        return -1

    # The Minimum Spanning Tree Function
    def min_span_tree(self):
        # Start searching at vertex in index 0
        self.vertex_list[0].visited = True

        # Push 0 in the stack
        self.the_stack.push(0)

        while not self.the_stack.is_empty():
            # Get the current vertex
            curr_vert = self.the_stack.peek()

            # Get the next neighbor vertex
            next_vert = self.get_next_unvisited_vertex(curr_vert)

            # If no more neighbor vertices
            if next_vert == -1:
                # Get rid of it
                self.the_stack.pop()
            else:
                # Visit it
                self.vertex_list[next_vert].visited = True
                self.the_stack.push(next_vert)

                # NEW Major change versus Depth First Search
                # Edge from this vertex
                self.print_vertex(curr_vert)
                # to this vertex
```

```
            self.print_vertex(next_vert)
            print()

        # Stack is empty so set all vertices back to unvisited
        for i in range(0, self.max_vertices):
            if isinstance(i, int):
                pass
            else:
                self.vertex_list[i].visited = False


# Test the Minimum Spanning Tree
graph = Graph()
graph.add_vertex('A')
graph.add_vertex('B')
graph.add_vertex('C')
graph.add_vertex('D')
graph.add_vertex('E')

# Make edges to each vertex
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(0, 3)
graph.add_edge(0, 4)
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(1, 4)
graph.add_edge(2, 3)
graph.add_edge(2, 4)
graph.add_edge(3, 4)

# Have the algorithm figure out one line that connects
# from one vertex to the others
graph.min_span_tree()
```