Graph Class Implementation

Part 1

Initialization

```
class Graph:
    def __init__(self, nodes):
        self.nodes = nodes
        self.edges = self.generate_edges()
```

Graph is initialized with a passed-in collection of nodes. My graph is represented in the following way:

```
example_graph = {
    "a": ["b", "c"],
    "b": ["a", "c", "d"],
    "c": ["a", "b", "d", "e"],
    "d": ["b", "c", "e"],
    "e": ["c", "d"]
}
```

Generating a list of edges

```
def generate_edges(self, nodes=None):
    graph_nodes = nodes if nodes else self.nodes
    edges = []

    for node in graph_nodes:
        for neighbor in graph_nodes[node]:
            edges.append((node, neighbor))  # nodes are tuples of
```

```
the form (neighbor, path cost)
    return edges
```

edges is initialized to an empty list. The outer loop iterates over each node and the inner loop iterates over each of that nodes neighbors. edges is appended with a tuple contain the current node and its neighbor.

finding isolated nodes

```
def find_isolated_nodes(self):
    """ returns a list of isolated nodes. """
    isolated = []

    for node in self.nodes:
        if not self.nodes[node]:
            isolated += node
    return isolated
```

nodes are not isolated if they may be reached from at least one other node in the graph. This equates to requiring that a node be a neighbor to atleast one node to not be isolated. We iterate over each node and check to see if this node has any neibors, if not then it is isolated.

Path finding between two verticies

```
def find_path(self, start_vertex, end_vertex, path=None):
    """ find a path from start_vertex to end_vertex in graph """

    if path == None:
        path = []

    path = path + [start_vertex]

    if start_vertex == end_vertex:
        return path
```

```
    if start_vertex not in self.nodes:
        return None

    for vertex in self.nodes[start_vertex]:
        if vertex not in path:
            extended_path = self.find_path(vertex, end_vertex,
path)

            if extended_path:
                return extended_path

    return None
```

find path recursively calls itself to find a path from start_vertex to end_vertex. it does this by trying (recursively) every neighboring node until the desired one is reached.

checking if graph is connected

```
def is_connected(self, vertices_encountered=None,
start_vertex=None):
    """ determines if the graph is connected """

    if vertices_encountered is None:
        vertices_encountered = set()
    vertices = list(self.nodes.keys())  # "list" necessary in
Python 3

    if not start_vertex:
        # choose a vertex from graph as a starting point
        start_vertex = vertices[0]
        vertices_encountered.add(start_vertex)
        if len(vertices_encountered) != len(vertices):
            for vertex in self.nodes[start_vertex]:
                if vertex not in vertices_encountered:
```

```
                    if self.is_connected(vertices_encountered,
vertex):
                        return True
                else:
                    return True
    return False
```

if the graph is connected, it means that all verticies have atleast 1 edge connecting them to some other vertex. The algorithm determines this by trying to traverse the graph and log all verties it finds. If the set of verticies that it has visited is equal to that of the set of all verticies, then it was able to take a path to each verticies...thus connected.

BFS

```
def bfs(self, start_node):
    """
        1. visit a node        2. save to visited list        3.
append neighbors to queue    """    queue = []
    visited = []

    # put start node into the queue
    queue.append(start_node)

    while queue:
        cur = queue.pop(0)
        if cur not in visited:

            visited.append(cur)
            queue.extend(self.nodes[cur]) # add child nodes to the
queue... will be visited after nodes on same level

    print("\nBFS Traversal: ")
    self.print_paths(paths=[visited])
```

Breadth-First Search (BFS) traverses each "layer" of nodes before continuing down a level of depth. This is accomplished by the use of a queue, which ensures that all nodes from the the same level are visited before progressing.

DFS

```
def dfs(self, start_node):
    """same idea as bfs but use a stack to sequence node visitation"""
    # using a list to model a stack...child node at top of stack is visited next    stack = [start_node]
    visited = []

    while stack:
        cur = stack.pop()

        if cur not in visited:
            visited.append(cur)
            stack.extend(self.nodes[cur])

    print("\nDFS Traversal: ")
    self.print_paths(paths=[visited])
```

DFS works in a fairly similar way to BFS, except with one crucial difference. DFS uses a stack to order the nodes to be visited next. When a node's neighbors are added to the stack, they are visited immediatly. This ensures that we traverse the depth through a path of nodes before moving on to the neighbor of the root.

Part 2

Graph Representation (Part 2)

The Graph has been modified for my part 2 (to include path cost). It is represented in the following way:

```
example_graph = {'start_node' : [('neighbor_1', 1), ('neighbor_2', 3)]}

# 1 and 3 are the path costs from start_node to neighbors
```

Helper Functions for Prims and Kruskals

```
def remove_loops(self, graph=None):
    cp_graph = graph if graph else copy.deepcopy(self.nodes) # doesnt actually modify self.nodes

    for node in cp_graph:  # Use cp_graph instead of self.nodes
        no_loops = []

        for neighbor in cp_graph[node]:
            if neighbor[0] != node:  # Check if the neighbor is not the same as the node
                no_loops.append(neighbor)

        cp_graph[node] = no_loops

    return cp_graph

def remove_redundant_paths(self, graph=None):
    cp_graph = graph if graph else copy.deepcopy(self.nodes)

    # iterate over nodes
    for node in self.nodes:
        # dict to hold the cheapest of the duplicate paths (for each node)
        cheapest_paths = {}
```

```
        for neighbor, path_cost in self.nodes[node]:
            # determine if this path cost is cheaper than the
currently stored one
            if neighbor in cheapest_paths:
                if path_cost < cheapest_paths[neighbor]:
                    cheapest_paths[neighbor] = path_cost
            else:
                cheapest_paths[neighbor] = path_cost

        cp_graph[node] = [(n, cost) for n, cost in
cheapest_paths.items()]

    return cp_graph

# sorts by ascending order
def getSortedEdges(self, nodes=None):
    if not nodes:
        nodes = self.nodes

    edges = self.generate_edges(nodes)
    return sorted(edges,key=lambda path: path[2])
```

Kruskals Algorithm

```
def mst_kruskal(self):
    # Removing all loops and redundant paths
    processed =
self.remove_redundant_paths(self.remove_loops(self.nodes))

    # Generating edges
    processed_edges = self.generate_edges(processed)

    # sort the edges in ascending order
    processed_edges=self.getSortedEdges(processed)
```

```
    # Sorting the edges by ascending order
    processed_edges.sort(key=lambda path: path[2])  # Edges are
tuples (node_1, node_2, path cost)
    #print(processed_edges)

    # Connect the nodes together, store each node in a visited list
to ensure that we don't add it again    visited = []
    mst = []

    traversal_cost = 0

    for tup in processed_edges:
        ns, ne, ps = tup

        if ns not in visited or ne not in visited:
            mst.append(tup)

            # add the past cost to counter
            traversal_cost+=tup[2]
            visited.extend([ns, ne])

        if len(visited) == len(self.nodes):
            break  # Stop when all nodes are visited

    print(f"\n(Krushkal's Algorithm) Total Path Cost:
{traversal_cost}")
    return mst
```

Kruskals algorithm involves viewing the graph as a whole and breaking it down. We start by removing all paths which are self loops. We then remove redundant paths (paths from node a to node b where a cheaper path already exists). After this, we order the paths from least to greatest and then begin adding nodes from this ordering, only adding nodes if they have not already been added.

Prims Algorithm

Helper Functions

```
def getStandardFrontier(self, nodes):
    out = {}  # will hold all nodes that are in the frontier but
are pulled from self.nodes
    for node in nodes:
        if node not in out:
            out[node[0]] = self.nodes[node[0]]
    return out

def updateFrontier(self, nodes, graph):
    out = {}

    for node in nodes:
        if node not in out.keys():
            out[node] = self.nodes[node]
    return out
```

Prim's:

```
def mst_prims(self):
    """Prim's algorithm:
       1. Remove all loops       2. Find the cheapest node to
travel to from the start node       3. Treat the currently
connected series of nodes as a single 'node' and continue adding
nodes with the cheapest path    """    # remove self loops
no_loops = self.remove_loops(self.nodes)

    # setup visited list and frontier (holds nodes up next to be
visited)
    visited = []
    mst = []
    frontier = {}

    # find the node with the cheapest path from no_loops to start
    start_edge = self.getSortedEdges(nodes=no_loops)[0]
```

```
    # Add the starting node to visited
    visited.append(start_edge[0])
    visited.append(start_edge[1])

    # add start point to output
    mst.append(start_edge)

    #  add starting node and neighbors to the frontier
    frontier = self.updateFrontier([start_edge[0], start_edge[1]],
graph=no_loops)

    while len(visited) < len(self.nodes):

        # Find  neighbor with the cheapest path cost
        min_cost = float('inf') # setting up min cost with max
value

        for node in visited:
            for neighbor, cost in no_loops[node]:

                # if we havent visited the node, add it to the
frontier
                if neighbor not in visited and cost < min_cost:
                    # update the min cost and node to the one with
the lowest cost and then set it to curr
                    min_cost = cost
                    curr = (node, neighbor, cost)

        # Add the current node to visited
        visited.append(curr[1])
        mst.append(curr)

        # Update the frontier with neighbors of the current node
        frontier = self.updateFrontier([curr[1]], graph=no_loops)

    print("\n(Prim's Algorithm) Total Path Cost: ", sum([x[2] for x
```

```
in mst]))
    return mst
```

prims works by selecting a starting node and then visiting that nodes' neighbors. When a neighbor is visited. We add its neighbors to the frontier and then treat the collection of nodes as a single "node". We then add each of this "nodes" neighbors to the frontier (neighbors of the outer most nodes of our graph that have been visited so far).

Dijkstras Algorithm

```
def mst_djikstras(self, start_node=None):
    """Dijkstra's algorithm without priority queue:
        1. Mark all nodes as unvisited and put them into an
unvisited set.        2. Assign every node a tentative distance
value (set to infinity for unvisited nodes and 0 for the initial
node).        3. Set the initial node as the current node.
4. For the current node, consider all unvisited neighbors and
calculate their tentative distances through the current node and
assign.           - If the distance was marked previously with a
longer path, change it to the shortest one that's now found.
5. If the destination node has been found OR the unvisited set is
empty, HALT.    """
    #  first node in our graph    start_node =
list(self.nodes.keys())[0]

    #  unvisited set with tentative distances
    distances = {node: float('infinity') for node in self.nodes}
    distances[start_node] = 0

    #   empty set for visited nodes
    visited = set()

    while True:
        # Find the unvisited node with the smallest tentative
distance:
```

```
        # add in nodes from distances if not visited
unvisited_nodes = [node for node in distances if node not in
visited]

        # all nodes have been vitsed...
        if not unvisited_nodes:
            break

        # find the node with the lowest path cost from our current
node
        current_node = min(unvisited_nodes, key=lambda node:
distances[node])

        # Mark the current node as visited
        visited.add(current_node)

        # Update tentative distances for neighbors (distance up
until now plus known path cost to this node)
        for neighbor, path_cost in self.nodes[current_node]:
            distance = distances[current_node] + path_cost

            # if shorter path is found then update the distance
            if distance < distances[neighbor]: # when starting well
be comparing path cost to infinity
                distances[neighbor] = distance

    # return all distnaces from start node
    return distances
```

Dijkstras Algorithm is designed to find the shortest path between all nodes in a graph. It works by visiting each node and seeing if there exists a shorter path to this node from the node that its visiting, if so it updates the path cost required to reach that node and continues until all nodes have been visited.

Testing

Part 1

```
g1 = graph(example_graph)
edges = g1.generate_edges()
print(edges)

adpaths = g1.find_all_paths('a','d')
g1.print_paths(adpaths)


g1.bfs('a')
g1.dfs('a')
```

Output

```
# Edges
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('b', 'd'), ('c',
'a'), ('c', 'b'), ('c', 'd'), ('c', 'e'), ('d', 'b'), ('d', 'c'),
('d', 'e'), ('e', 'c'), ('e', 'd')]

# paths from a to d
a -> b -> c -> d
a -> b -> c -> e -> d
a -> b -> d
a -> c -> b -> d
a -> c -> d
a -> c -> e -> d

# traversals

BFS Traversal:
a -> b -> c -> d -> e
```

```
DFS Traversal:
a -> c -> e -> d -> b
```

Part 2

Testing

```
g1 = Graph(test1)
g2= Graph(test2)


# Testing Graph 1
print(g1.mst_kruskal())
print(g1.mst_prims())

print("\nDijkstra Graph 1")
print(g1.mst_djikstras())

print("\n---------------------------------")

# Testing Graph 2
print(g2.mst_kruskal())
print(g2.mst_prims())

print("\nDijkstra Graph 2")
print(g2.mst_djikstras())
```

Output:

```
# testing g1
(Krushkal's Algorithm) Total Path Cost: 12
[('a', 'b', 1), ('b', 'd', 2), ('b', 'c', 4), ('d', 'e', 5)]

(Prim's Algorithm) Total Path Cost:  12
```

```
[('a', 'b', 1), ('b', 'd', 2), ('b', 'c', 4), ('d', 'e', 5)]

Dijkstra Graph 1
{'a': 0, 'b': 1, 'c': 5, 'd': 3, 'e': 8}

--------------------------------

# testing g2
(Krushkal's Algorithm) Total Path Cost: 16
[('A', 'C', 2), ('C', 'D', 3), ('A', 'B', 4), ('D', 'E', 7)]

(Prim's Algorithm) Total Path Cost:  16
[('A', 'C', 2), ('C', 'D', 3), ('A', 'B', 4), ('D', 'E', 7)]

Dijkstra Graph 2
{'A': 0, 'B': 4, 'C': 2, 'D': 5, 'E': 12}
```