Uniform Cost Search Algorithm

Objects to represent the graph

```
class Node:
    def __init__(self, data, edges: [] = None):
        self.data = data
        if edges:
            self.edges = edges
        else:
            self.edges = []
    def add_child(self, child, cost):
        self.edges.append(Edge(self, child, cost))

    def __repr__(self):
        if len(self.edges) == 0:
            return str(self.data) + " -> " + "X"
        return str(self.data) + " -> " + str(self.edges)

    def __gt__(self, other):
        """required for priority queue to compare and sort"""
        return self.data > other

    def  __lt__(self, other):
        """required for priority queue to compare and sort"""
        return self.data < other


class Edge:
    def __init__(self, parent: Node, child: Node, cost: int = 1):
        self.parent = parent
        self.child = child
        self.cost = cost

    def __repr__(self):
        return str({self.child.data : self.cost})
```

UCS Algorithm

```python
from Graph import Node, Edge

from queue import PriorityQueue

def ucs(initial_state: Node, goal_state: Node):
    """Algorithm
        1. check to make sure we're not at goal, if we are then exit and
return the node
        2. put initial node into the queue
        3. unpack initial node and check children for one with lowest
path cost
        4. perform step (1.) on this child node
        5. check children against nodes already in the queue
            - if same child node with lower path cost is found and it
already exists in queue
                replace node thats in the queues cost with the cost of
the child


        Sources used (Priority  Queue Usage and functions)
https://realpython.com/queue-in-python/
    """

    queue = PriorityQueue()

    queue.put((0, [initial_state])) #inserting the intial state

    while not queue.empty():
        val = queue.get()
        cur_node = val[1][-1] #checking node with lowest cost
        if cur_node.data == goal_state.data: #terminating condition
            return val[1]

        for edge in cur_node.edges:
            path = list(val[1])
            path.append(edge.child) #adding all children of node to the
queue

            updated_cost = val[0] + edge.cost
            queue.put((updated_cost, path)) #update cumulative cost of
node and provide child node
```

```
A = Node('A')
B = Node('B')
C = Node('C')
D = Node('D')
E = Node('E')
F = Node('F')

A.add_child(B, 10)
A.add_child(C, 13)
B.add_child(D, 4)
D.add_child(E, 2)
D.add_child(F, 50)
E.add_child(F, 3)


print("Optimum Path: ")
print(ucs(A, F))
```

Output

```
Optimum Path:
[A -> [{'B': 10}, {'C': 13}], B -> [{'D': 4}], D -> [{'E': 2}, {'F':
50}], E -> [{'F': 3}], F -> X]
```