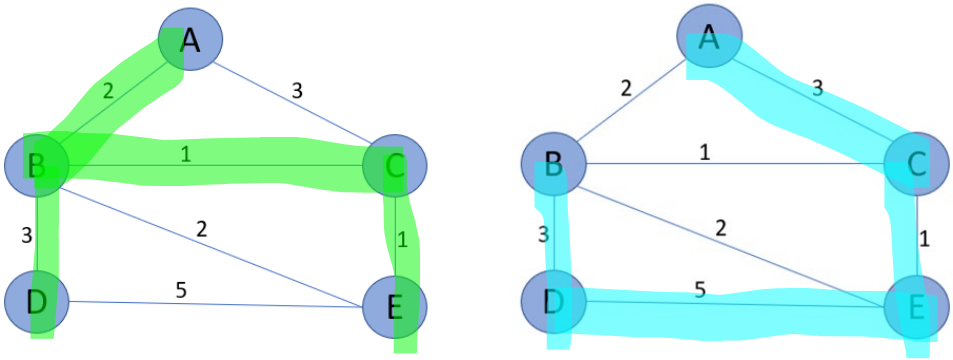
Khrystian Clark

01MAR2022

CS325, Winter 2022

Assignment: Graph Algorithms – II

1. **Draw Minimum Spanning Tree**
   a. Draw minimum spanning tree for the below graph.
      a. A →B→C→E, + B→D
      b. 2+ 1+ 1+ 3 =7
   b. Draw spanning Tree that is not minimum
      a. A→C→E→D→B
      b. 3+ 1+ 5+ 3 =12



2. **MST implementation:**
   a. Implement Prims' algorithm Name your function **Prims(G).** Include function in the file **MST.PY.** Mention in your submission the input format and output format of your program.
      a. Shown is MST.PY. I utilized the implementation from the module, and adjusted it in order for it to handle any sized 2D array. I also added lines to track and return the total weight of the MST.
      b. Input format: G is an array of arrays creating a "table" (on line 1)
      c. Output: returns the list of edges followed and the weights of each edge (lines 13, 27, 31)

```
b. def prims(G): # G is a 2D array.
c.     INF = 9999999
d.     V = len(G) # number of vertices in graph
e.     weight_count = 0 # tracks the total weight of the tree
f.     selected = [] # for the algorithm to handle any amount of
   vertices
g.     for i in G:
h.         selected.append(0)
i.     # set number of edge to 0
```

```
j.      no_edge = 0
k.      # make first vertex true
l.      selected[0] = True
m.      print("Edge : Weight\n") #titles of what is returned in a
    table list like dijkstra
n.      while (no_edge < V - 1):
o.          minimum = INF
p.          x = 0
q.          y = 0
r.          for i in range(V):
s.              if selected[i]:
t.                  for j in range(V):
u.                      if ((not selected[j]) and G[i][j]):
v.                          # not in selected and there is an edge
w.                          if minimum > G[i][j]:
x.                              minimum = G[i][j]
y.                              x = i
z.                              y = j
aa.             print(str(x) + "-" + str(y) + ":" + str(G[x][y])) #
    prints the edge and weight
bb.             weight_count += (G[x][y])
cc.             selected[y] = True
dd.             no_edge += 1
ee.         print("\nTotal weight=", weight_count) # prints the
    total weight
ff.
gg.     G = [[0, 9, 75, 0, 0],
hh.          [9, 0, 95, 19, 42],
ii.          [75, 95, 0, 51, 66],
jj.          [0, 19, 51, 0, 31],
kk.          [0, 42, 66, 31, 0]]
ll.
mm.     prims(G)
```

nn. What is the difference between the Kruskal's and the Prim's algorithm?

a. Prim's picks a random node and builds a spanning tree from there, while Kruskal starts with the minimal edge available that does not create a cycle and adds that to the tree.
b. Prim's algorithm adds to what is already there or connected, whereas this is not necessary in the Kruskal's. Kruskal gauges the tree branches on the cheapest edges available, but not necessarily connected yet.

3. **Apply Graph traversal to solve a problem (Portfolio Project Problem):**
You are given a 2-D puzzle of size MxN, that has N rows and M column (N>=3 ; M >= 3; M and N can be different). Each cell in the puzzle is either empty or has a barrier. An empty cell is marked by '-' (hyphen) and the one with a barrier is marked by '#'. You are given two coordinates from the puzzle (a,b) and (x,y). You are currently located at (a,b) and want to reach (x,y). You can move only in the following directions.

L: move to left cell from the current cell
R: move to right cell from the current cell
U: move to upper cell from the current cell
D: move to the lower cell from the current cell

You can move to only an empty cell and cannot move to a cell with a barrier in it. Your goal is to find the minimum number of cells that you have to cover to reach the destination cell (do not count the starting cell and the destination cell). The coordinates (1,1) represent the first cell; (1,2) represents the second cell in the first row. If there is not possible path from source to destination return None.
Sample Input Puzzle Board: [[-,-,-,-,-],[-,-,#,-,-],[-,-,-,-,-],[#,-,#,#,-],[-#,-,-,-]]

-	-	-	-	-
-	-	#	-	-
-	-	-	-	-
#	-	#	#	-
-	#	-	-	-

Example 1: (a,b) : (1,3) ; (x,y): (3,3)
Output: 3
On possible direction to travel: LDDR
(1,3) →(1,2) → (2,2) → (3,2)→(3,3)

Example 2: (a,b): (1,1) ; (x,y): (5,5)
Output: 7
One possible direction to travel: DDRRRRDD
(1,1) →(2,1)→(3,1)→(3,2)→(3,3)→(3,4)→(3,5)→(4,5)→(5,5)

Example 3: (a,b): (1,1); (x,y) : (5,1)
Output: None

a. Describe an algorithm to solve the above problem.
    a. solve_puzzle(Board, Source, Destination)
        i. Go through scenarios to ensure the board is within parameters
        ii. Go through scenarios to ensure Source and Destination are within the Board
        iii. Create a counter to track weight, edges, selected, and travel directions
        iv. Shape it like a my Prim's Algorithm for MST about but use the variables of the Source and Destination as the ranges.
        v. Return the "weight" of the edges created to get to the destination
        vi. Return the string of travel directions.
b. Implement your solution in a function **solve_puzzle(Board, Source, Destination).** Name your file **Puzzle.py**

```
c. def solve_puzzle(Board, Source, Destination):
d.     weight = 0
e.     no_edge = 0
f.     selected = []
g.     for i in Board:
h.         selected.append("")
i.     selected[0] = True
j.     travel = ""
k.     if len(Board) < 3:
l.         return "Board too small"
m.     for i in Board:
n.         if len(i) < 3:
o.             return "Board too small"
p.         if Source[1] > len(i) or Destination[1] > len(i):
q.             return "Input out of bounds"
r.     if Source[0] > len(Board) or Destination[0] > len(Board):
s.         return "Input out of bounds"
t.     if (Board[Source[0]][Source[1]]) or
   (Board[Destination[0]][Destination[1]]) == "#":
u.         print("None")
v.     while Source != Destination:
w.         for i in range(Source[0], Destination[0]):
x.             if selected[i]:
y.                 for j in range(Source[1], Destination[1]):
z.                     if ((not selected[j]) and Board[i][j]):
aa.                             if i < Source[0]:
bb.                                 travel.append("U")
cc.                             if i > Source[0]:
dd.                                 travel.append("D")
ee.                             if j < Source[1]:
ff.                                 travel.append("L")
gg.                             if j > Source[1]:
hh.                                 travel.append("R")
ii.                             Source[0] = i
jj.                             Source[1] = j
kk.             weight +=1
ll.             selected[Source[1]] = True
mm.             no_edge +=1
nn.             if Source == Destination:
oo.                 print("Weight:" + weight)
pp.                 print(str(travel))
qq.
rr.     Board = [["","","","",""],
ss.              ["","","#","",""],
tt.              ["","","","",""],
uu.              ["#","","#","#",""],
vv.              ["","#","","",""]]
ww.
```

```
xx.     solve_puzzle(Board, (1,3), (3,3))
yy.
zz.     # Should return "Weight: 4, 'RDDL'"
```

aaa. What is the time complexity of your solution?

a. O(ElogV)
b. Because it does what the example algorithm does and uses a nim-heap.

bbb. **(Extra Credit):** For the above puzzle in addition to the output return a set of possible directions as well in the form of a string.

a. Shown attempt in lines of code: 5, 24-31, and 40

For above example 1 Output: 3, LDDR