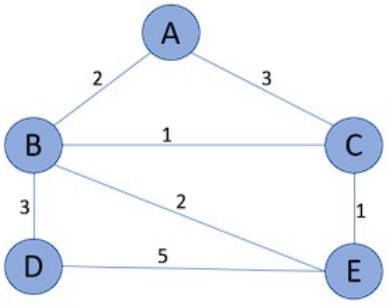
Problem 1: Minimum Spanning Tree

Let G be an undirected graph with vertices V linked by edges E.



$G = (V, E)$, where
$V = \{A, B, C, D, E\}$
$E = \{(AB, AC, BC, BD, BE, CE, DE)\}$

Each edge of E has weight w.

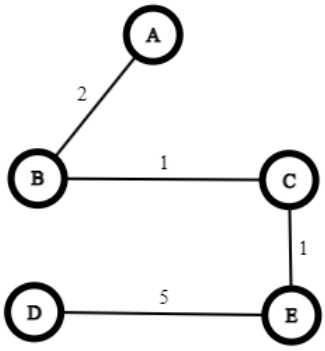
$$w(AB) = 2$$
$$w(AC) = 3$$
$$w(BC) = 1$$
$$w(BD) = 3$$
$$w(BE) = 2$$
$$w(CE) = 1$$
$$w(DE) = 5$$

Continued on next page

Problem 1: MST (Continued)

1a) Drawn MST (Cost 9 — ABCED)

Let T be an MST subset of G.



Vertices V link by lowest cost E'.

$$T = (V, E'), \text{ where}$$
$$E' = \{(AB, BC, CE, ED)\}$$
$$w(E') = \sum \{(AB, BC, CE, ED)\}$$

Each pair of E' has weight w, where $w(E')$ or $\sum \{E'\}$ is the lowest cost.

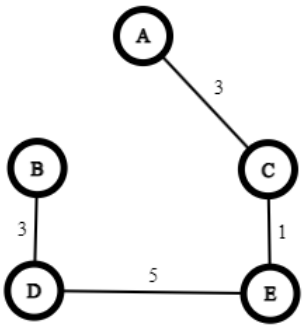
$$w(AB) = 2$$
$$w(BC) = 1$$
$$w(CE) = 1$$
$$w(ED) = 5$$
$$w(E') \text{ or } \sum \{E'\} = 9$$

Continued on next page

Problem 1: MST (Continued)

1b) Drawn Non-MST (Cost 12 — ACEDB)

Let Q be a non-MST subset of G.



Vertices V link by cost E'.

$$\begin{aligned} Q &= (V, E'), \text{ where} \\ E' &= \{(AC, CE, ED, DB)\} \\ w(E') &= \sum \{(AC, CE, ED, DB)\} \end{aligned}$$

Each pair of E' has weight w, where $w(E')$ or $\sum \{E'\}$ isn't the lowest cost.

$$\begin{aligned} w(AC) &= 3 \\ w(CE) &= 1 \\ w(ED) &= 5 \\ w(DB) &= 3 \\ w(E') \text{ or } \sum \{E'\} &= 12 \end{aligned}$$

Problem 2: MST Implementation

2a) Prim's Algorithm

See external file MST.py

2b) Prim's versus Kruskal's

Prim's starts with a root vertex, adding each minimum edge at a time connecting to any known vertex. It uses a priority queue for optimization, moving one edge at a time.

Kruskal's starts with an empty set, sorting edges by ascending order before searching for the most minimum edge in the graph altogether. Gradually, Kruskal's builds the complete minimum spanning tree, while evaluating whether or not it's creating a cycle at each step. It uses a disjoint set to check its vertices and edges.

Problem 3: Graph Traversal Application

A 2D puzzle has M columns and N rows.
Each cell is empty ($-$) or has a barrier ($\#$).
From coordinate (a, b), reach (x, y) minimally by moving orthogonally to empty cells.

3a) Algorithm Pseudocode

The code strongly resembles Dijkstra's algorithm, as it begins by making a deepcopy of the board itself and assigning weights to each cell based on its distance from the destination.

1. Loop the 2D matrix:

   If the cell holds $\#$, assign an infinity value.

   Else, assign a weight that's the total of the two row and column differences.

2. Create:

   An empty set to track visited cells.

   An tuple to track the current coordinate.

   A list to track decided cells.

3. While there's a current cell tuple:

   The row and column are the current cell.

   If the current cell is infinity, exit.

   If the current cell is the destination, exit.

   Find the orthogonal neighbors from the current cell.

   If the neighbor's in bounds and is not a barrier, and it's lower than the current minimum weight, make it the new minimum.

   Add the new cell to the list of all decided cells and restart the loop.

3b) Solution

See external file Puzzle.py

3c) Time Complexity: $\mathcal{O}(n^2)$ or $\mathcal{O}(mn)$

Assuming the matrix has same dimensions, quadratic, but in the event of an irregular matrix it could be mn.

Its time complexity is dictated by the initial deepcopy, which requires iterating over each cell linearly. Otherwise, the while loop at the worst case could iterate the same number of times but is unlikely.

Best case is $\mathcal{O}(1)$; the code checks if the destination cell is a barrier, returning immediately if so.