ECE 241: Data Structures and Algorithms- Fall 2022

Project 4: FunWithGraph (DFS,BFS,MST using Queue/Stack/PQ-Heap)

Due Date: **Deadline: see website**

Description

The goal of this project is to operate a Graph data structure and implement various graph algorithms such as depth-first search (DFS), breadth-first search (BFS), minimum spanning tree Prim's algorithm for weighted (non-directed) graphs (MSTW). The graphs are defined using a rectangular grid and contain vertices that are defined by (integer) Cartesian coordinates (x,y), and edges that connect those vertices with a given (integer) weight.

How to start

The project includes multiple 'source' files:

1. `App1.py` to `App3.py`: three application files (provided).
2. `Queue.py`, `Stack.py`, `Heap.py` files containing the object class Queue, Stack and Heap (provided).
3. `Graph.py` file containing the class objects: Graph, Edges, and Vertex (to complete).
4. `grid4x3.txt`, `grid9x9.txt`, `grid7x5.txt`: coordinate graph files for testing.

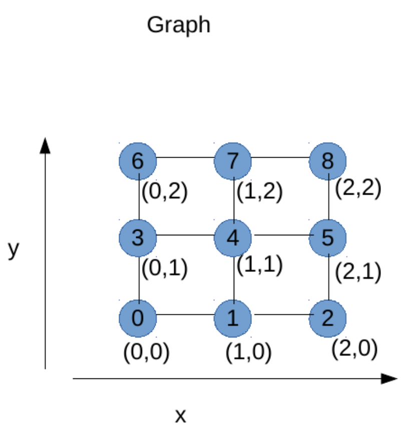
All the functionality of the application files (presented in details below) should be successfully implemented to obtain full credit. You need to proceed step-by-step, application by application. A list of methods to implement is described at the end of each section.

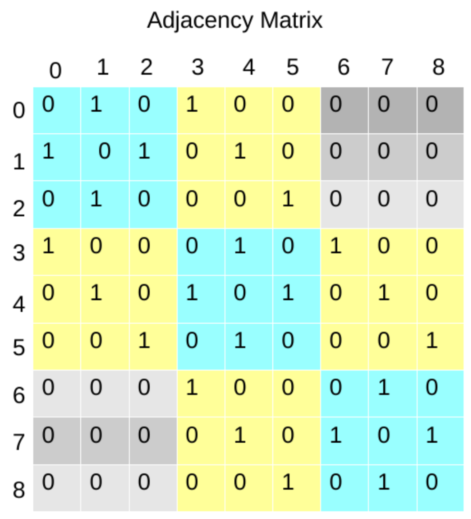
`App1.py`

The code creates a regular grid where all first neighbors are connected. The size of the grid is set by the user. The weights are uniformly equal to 1 for all edges. The code returns some information about the graph (such as connecting edges, total weight and adjacency matrix for small systems).

Figure 1 provides a complete example of a 3x3 grid graph with matrix representation.

Example of code execution:




Adjacency Matrix

	0	1	2	3	4	5	6	7	8
0	0	1	0	1	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0
2	0	1	0	0	0	1	0	0	0
3	1	0	0	0	1	0	1	0	0
4	0	1	0	1	0	1	0	1	0
5	0	0	1	0	1	0	0	0	1
6	0	0	0	1	0	0	0	1	0
7	0	0	0	0	1	0	1	0	1
8	0	0	0	0	0	1	0	1	0

Figure 1: Grid-Graph, local and global coordinates, matrix representation.

```
Welcome to Graph App 1
========================

Enter Total Grid Size Nx and Ny: 3 3

List of edges + weights:
(0,0) <==> (1,0) 1
(0,0) <==> (0,1) 1
(1,0) <==> (2,0) 1
(1,0) <==> (1,1) 1
(2,0) <==> (2,1) 1
(0,1) <==> (1,1) 1
(0,1) <==> (0,2) 1
(1,1) <==> (2,1) 1
(1,1) <==> (1,2) 1
(2,1) <==> (2,2) 1
(0,2) <==> (1,2) 1
(1,2) <==> (2,2) 1
Total weight: 12

Matrix:
0 1 0 1 0 0 0 0 0
1 0 1 0 1 0 0 0 0
0 1 0 0 0 1 0 0 0
1 0 0 0 1 0 1 0 0
0 1 0 1 0 1 0 1 0
0 0 1 0 1 0 0 0 1
0 0 0 1 0 0 0 1 0
```

```
0 0 0 0 1 0 1 0 1
0 0 0 0 0 1 0 1 0
```

At this stage a grid-graph (similar to the one in Figure 2) is plotted in color GRAY.

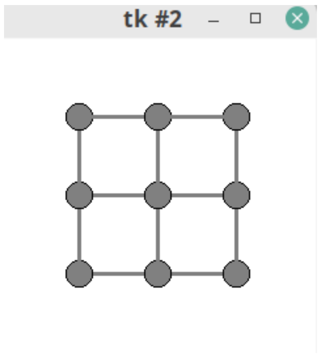


Figure 2: 3x3 grid.

What is needed in `Graph.py`:

1. Create the class Vertex using two attributes `i,j` as inputs that represents the coordinate system. Do not forget to include the `visited` attribute (set to `False`) and a `__str__` method.

2. A constructor for the class Graph that instantiate a rectangular grid graph. Create the list of vertices and initialize the adjacency matrix of size nx∗ny to 0.

3. Implement `form2DGrid` method that sets the adjacency matrix by adding edges to the Graph to form the 2D grid (all with a weight of 1). Hint: only the first neighbors of each vertex are connected, you may need to know how to map local and global coordinates, you also need to define the method `addEdge`.

4. The `displayInfoGraph` method, provides information about the graph. Your output should be similar to the example above. The method returns also the total weight of the graph.

5. The `displayAdjMatrix` method, provides the matrix. Your output should be similar to the example above. You also need to implement the simple `getnVertex` method used to return the value of nVertex in the main code.

6. Implement the `plot` method using Tkinter. Multiple Tips:

   - Start your plot method by `root=Tk()`.
   - You will consider a white canvas of size `w=80*nx` and `h=80*ny`.

- To migrate from your real math coordinate system to the Tkinter coordinate system, you can use the provided static method `toTkinter`. For example, if you want to know the i,j pixel Tkinter coordinates of the grid point (2,3) you could do the following :

  `i,j=Graph.toTkinter(2,3,-1,nx,-1,ny,w,h)`
- Vertices are represented by circle (oval) of radius 10 pixels (and user input color)
- Edges are represented by line of width 3 times the weight of the connection (so 3*1=3 here).
- End your plot method by `root.mainloop()`

App2.py

The first part is similar to App1, but you can continue the execution of the code by selecting a search algorithms (DFS or BFS) and a starting global node for performing the search.

As a result, a new MST graph will be created.

Remark: Depending on the starting point, the MST is not unique since all the weights are the same. You will display the information about the new graph as well (edges connections, total weights, and matrix for small systems).

Here is what happens if we continue the execution using DFS with Vertex 0:

```
Welcome to Graph App 2
========================

Enter Total Grid Size Nx and Ny: 3 3

List of edges + weights:
(0,0) <==> (1,0) 1
(0,0) <==> (0,1) 1
(1,0) <==> (2,0) 1
(1,0) <==> (1,1) 1
(2,0) <==> (2,1) 1
(0,1) <==> (1,1) 1
(0,1) <==> (0,2) 1
(1,1) <==> (2,1) 1
(1,1) <==> (1,2) 1
(2,1) <==> (2,2) 1
(0,2) <==> (1,2) 1
(1,2) <==> (2,2) 1
Total weight: 12

Matrix:
0 1 0 1 0 0 0 0 0
1 0 1 0 1 0 0 0 0
```

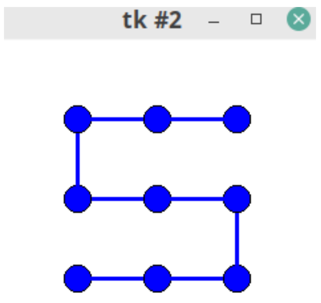
```
0 1 0 0 0 1 0 0 0
1 0 0 0 1 0 1 0 0
0 1 0 1 0 1 0 1 0
0 0 1 0 1 0 0 0 1
0 0 0 1 0 0 0 1 0
0 0 0 0 1 0 1 0 1
0 0 0 0 0 1 0 1 0

Perform: 1-DFS or 2-BFS? 1
Choose the starting node number: 0

List of edges + weights:
(0,0) <==> (1,0) 1
(1,0) <==> (2,0) 1
(2,0) <==> (2,1) 1
(0,1) <==> (1,1) 1
(0,1) <==> (0,2) 1
(1,1) <==> (2,1) 1
(0,2) <==> (1,2) 1
(1,2) <==> (2,2) 1
Total weight: 8
```

Now using the BFS search algorithm starting from Vertex 0, we get:

```
Welcome to Graph App 2
========================

Enter Total Grid Size Nx and Ny: 3 3

List of edges + weights:
(0,0) <==> (1,0) 1
(0,0) <==> (0,1) 1
(1,0) <==> (2,0) 1
(1,0) <==> (1,1) 1
(2,0) <==> (2,1) 1
(0,1) <==> (1,1) 1
(0,1) <==> (0,2) 1
(1,1) <==> (2,1) 1
(1,1) <==> (1,2) 1
(2,1) <==> (2,2) 1
(0,2) <==> (1,2) 1
(1,2) <==> (2,2) 1
Total weight: 12
```

```
Matrix:
0 1 0 1 0 0 0 0 0
1 0 1 0 1 0 0 0 0
0 1 0 0 0 1 0 0 0
1 0 0 0 1 0 1 0 0
0 1 0 1 0 1 0 1 0
0 0 1 0 1 0 0 0 1
0 0 0 1 0 0 0 1 0
0 0 0 0 1 0 1 0 1
0 0 0 0 0 1 0 1 0

Perform: 1-DFS or 2-BFS? 2
Choose the starting node number: 0

List of edges + weights:
(0,0) <==> (1,0) 1
(0,0) <==> (0,1) 1
(1,0) <==> (2,0) 1
(1,0) <==> (1,1) 1
(2,0) <==> (2,1) 1
(0,1) <==> (0,2) 1
(1,1) <==> (1,2) 1
(2,1) <==> (2,2) 1
Total weight: 8
```

The resulting MST are shown in Figure 3:



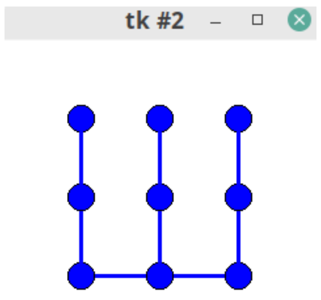


Figure 3: MST result to the 3x3 grid using DFS (left) and BFS (right).

What is needed in `Graph.py`:

1. The `dfs` method to perform the depth first search and return a new MST graph. Your output should be similar to the example above. You need the class `Stack`, provided here. You need to implement the `getAdjUnvisitedNode` method.

2. You need to implement the `bfs` method to perform the breadth first search. Your output should be similar to the example above. You need the class `Queue`, provided here.

`App3.py`

We are now considering a weighted graph that you are going to read from a file. For example let us look at the file `grid3x4.txt`

```
1 0 1
2 1 3
3 0 1
4 1 2
4 3 4
5 2 1
5 4 1
6 3 3
7 4 1
7 6 1
8 5 4
8 7 3
9 6 1
10 7 2
10 9 1
11 8 3
11 10 1
```

The file contains the list of edges making the connections between two vertices (in global coordinates) associated with a given weight.

Here is the execution output:

```
Welcome to Graph App 3
========================

Enter Total Grid Size Nx and Ny: 3 4

List of edges + weights:
(0,0) <==> (1,0) 1
(0,0) <==> (0,1) 1
(1,0) <==> (2,0) 3
(1,0) <==> (1,1) 2
(2,0) <==> (2,1) 1
```

```
(0,1) <==> (1,1) 4
(0,1) <==> (0,2) 3
(1,1) <==> (2,1) 1
(1,1) <==> (1,2) 1
(2,1) <==> (2,2) 4
(0,2) <==> (1,2) 1
(0,2) <==> (0,3) 1
(1,2) <==> (2,2) 3
(1,2) <==> (1,3) 2
(2,2) <==> (2,3) 3
(0,3) <==> (1,3) 1
(1,3) <==> (2,3) 1
Total weight: 33
```

The resulting grid-graph is shown in Figure 4. We note that the method `plot` (provided) uses different thickness to draw the connections between all the nodes (corresponding to the weights). Reminder (x3): a weight of 1 will have a thickness of 3, a weight of 2 will have a thickness of 6, etc.

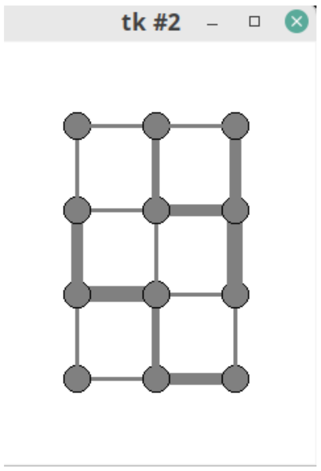


Figure 4: Grid-graph grid1 3x4 with given weights.

From here, after closing the graph, we can continue the execution with different options that are possible. The user can select the options to perform a DFS or BFS like in App2 (the unweighted graph will be considered for the search) and return a corresponding weighted graph from a given starting Vertex.

Here is the example of execution if you choose DFS and Vertex 0:

```
Perform: 1-DFS,  2-BFS or  3-MSTW? 1
Choose the starting node number: 0
```

```
List of edges + weights:
(0,0) <==> (1,0) 1
(1,0) <==> (2,0) 3
(2,0) <==> (2,1) 1
(0,1) <==> (1,1) 4
(0,1) <==> (0,2) 3
(1,1) <==> (2,1) 1
(0,2) <==> (1,2) 1
(1,2) <==> (2,2) 3
(2,2) <==> (2,3) 3
(0,3) <==> (1,3) 1
(1,3) <==> (2,3) 1
Total weight: 22
```

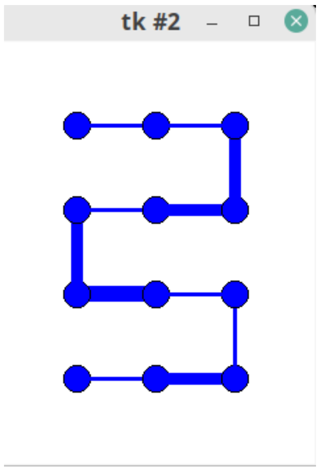
Here is the example of execution if you choose BFS and Vertex 0:

```
Perform: 1-DFS,  2-BFS or  3-MSTW? 2
Choose the starting node number: 0

List of edges + weights:
(0,0) <==> (1,0) 1
(0,0) <==> (0,1) 1
(1,0) <==> (2,0) 3
(1,0) <==> (1,1) 2
(2,0) <==> (2,1) 1
(0,1) <==> (0,2) 3
(1,1) <==> (1,2) 1
(2,1) <==> (2,2) 4
(0,2) <==> (0,3) 1
(1,2) <==> (1,3) 2
(2,2) <==> (2,3) 3
Total weight: 22
```

The resulting graph obtained with DFS and BFS is shown in Figure 5:

The user can also select to compute the MSTW and return the new optimal weighted graph. The code computes then the MSTW of the original graph (this is also a graph data structure). The number of vertices is the same than the original graph but the number of edges is nVertex-1 (11 in the example). Once the new graph is obtained, the code returns the new info containing the optimized connections with weights as well as the total minimum weight, and the new plot.

Here is the example of execution using MSTW



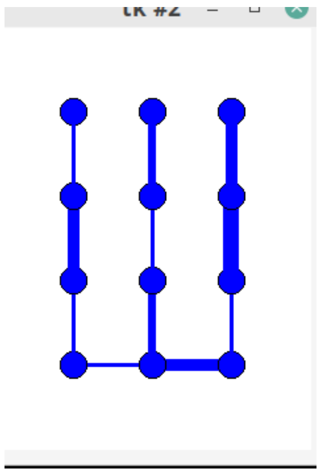


Figure 5: MST result fo the 3x3 weighted grid using DFS (left) and BFS (right).

```
Perform: 1-DFS,  2-BFS or  3-MSTW? 3
Choose the starting node number: 0

List of edges + weights:
(0,0) <==> (1,0) 1
(0,0) <==> (0,1) 1
(1,0) <==> (1,1) 2
(2,0) <==> (2,1) 1
(1,1) <==> (2,1) 1
(1,1) <==> (1,2) 1
(0,2) <==> (1,2) 1
(0,2) <==> (0,3) 1
(2,2) <==> (2,3) 3
(0,3) <==> (1,3) 1
(1,3) <==> (2,3) 1
Total weight: 14
```

We note that the resulting weight is now the smallest. The resulting MSTW is shown in Figure 6:

Remark: If the weight in the input file are randomly generated, using MSTW you are creating a random Maze Generator! (a maze along the blue lines). It is easier to see this using a larger database, Figure 7 shows the MSTW for the `grid9x9.txt` file (total weight of MSTW is 127).

What is needed:

1. The method `load2Dgrid` that reads the edge info from the file and initialize the matrix with weights. If a file does not exist, the code should stop executing without bug (while displaying some info saying the file does not exist).

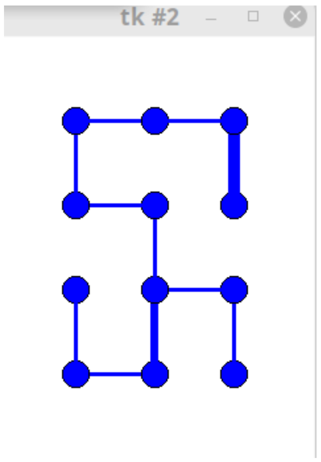


Figure 6: result MSTW on grid1, 3x4 Maze.

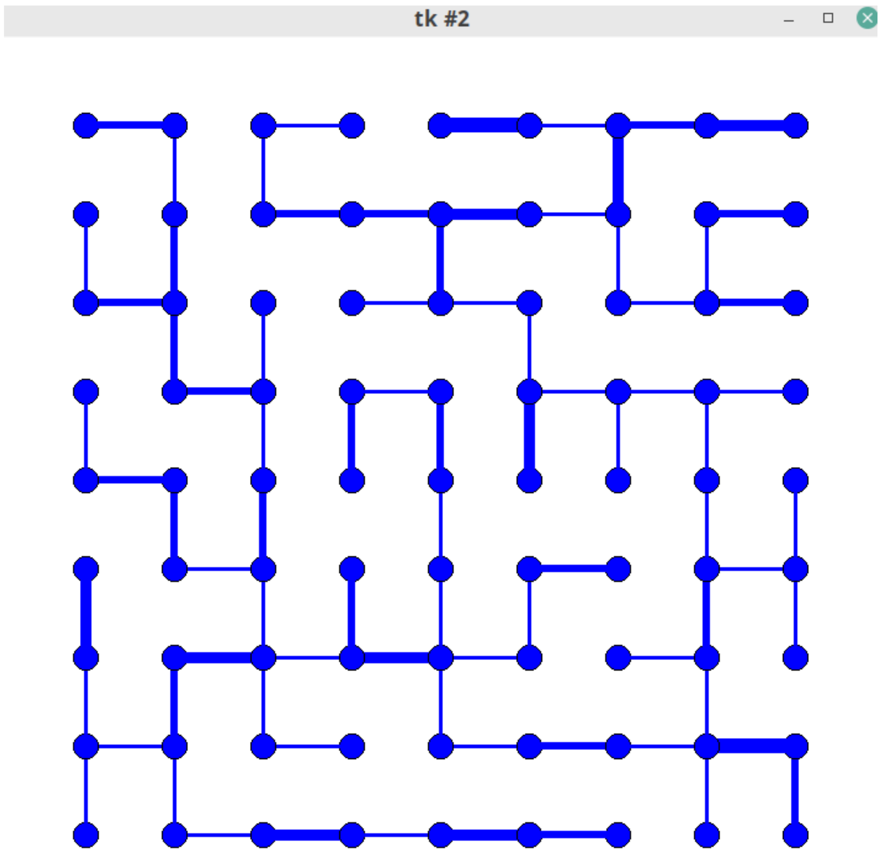


Figure 7: result MSTW on grid3, 9x9 Maze.

2. The method `mstw` that implement the Prim's algorithm for computing the MSTW (that is returned as a new grid-graph). You will use a Heap to implement the priority queue. You need first to implement a new `Edge` class with a constructor that accepts three input arguments: the source node number, the destination node number, and the weight. This way, you will be able to insert or remove object of type `Edge` in the Heap.

3. The provided class `Heap` already considers the smallest "item" as priority item. The Heap is checking the items with symbol `<` and `>` when you trickledown or up. Here you want to overload these operator in the class `Edge` where you could just compare the relative weight of the edges.

`Bonus (5pts)`

Finally, if you apply App3 to the Christmas tree graph `grid7x5.txt` (that does not include all vertices from the grid.. see Figure 8), the code may consider that the graph is not connected and will not work. For full credit, you would need to modify `bst, dfs, mstw` to make the code works fine in this situation as well. For starting node, you will use "3". Hint: you may need to modify your `plot` method as well, so you do not plot the vertices that are not connected. The MSTW graph is shown in Figure 9.

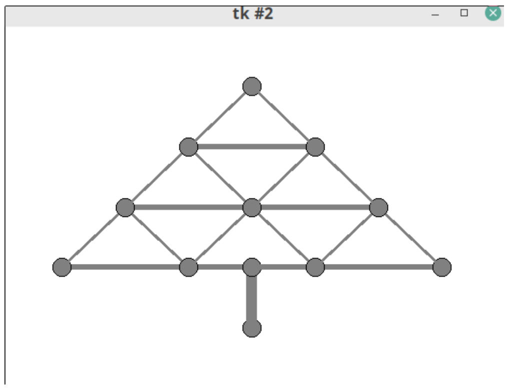


Figure 8: Christmas-Tree graph grid7x5.txt.

Submissions

Submit a single file `Graph.py` on moodle by the due date. Only one submission by group of two. Write your names on top of the file.

Grading Proposal

This project will be graded out of 100 points:

1. Your program should implement all basic functionality and run correctly. (90 points).

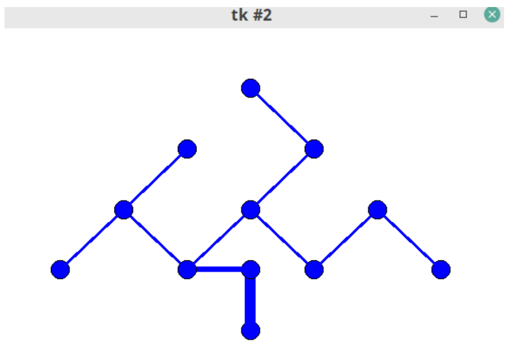


Figure 9: Christmas-Tree MSTW graph grid7x5.txt.

2. Overall programming style: source code should have proper identification, and comments. (5 points)

Another Extra Credit - 10pt (no help from TA or instructor)

You can implement App4 applied to the MSTW of `grid9x9.txt` that computes the shortest path (Dijkstra's algorithm) from the bottom left point (0,0) to the upper right corner (nVertex-1,nVertex-1) point. You will consider that all the weights (between 1 and 4) are now equal to 1 (to obtain a real maze situation). You will display the path, compute/return the cost (sum of weights) and plot (in GREEN) the path (on top of the BLUE MST graph). Send you `App4` to the TAs by email at the time of submission.