Introduction

In this project, you will work in groups of 2-3 to investigate a variety of *pathfinding algorithms*, including [versions of] breadth-first search, depth-first search, and Dijkstra's algorithm.

There are two deadlines for this project:

- November 22: midpoint check-in
    - Graded only for completion (anyone who submits all of the required materials by the deadline will receive full credit)
    - Each group will submit the following:
        - Any code (and any auxiliary files) written so far
        - A written document containing
            - Any of the written work you've done for the project
            - A rough plan/timeline for finishing the project, including a high-level outline of how work will be split among group members
    - Each person will also individually submit a short paragraph describing your contributions to the project so far. (Not required if working solo)
- Final due date: December 4
    - Each group will submit the following:
        - Your code (and any auxiliary files needed to run it)
        - A design document (as a pdf, Word doc, etc)
    - Each person will also individually submit a short paragraph describing their contributions to the project. (Not required if working solo)

Part A – Graphs and Maps

In this part, you will write code for creating and modifying maps. In later parts, you will create more advanced maps and explore them using pathfinding algorithms. There is some (very minimal) Python starter code provided for you to use, but you can complete the project in a language of your choice.

Implementing Graphs

Let's start by implementing undirected weighted graphs. (The instructions below are written under the assumption that vertices are implemented as objects with a unique key `k` and a value `val`, but you can implement them however you want as long as your graph supports the required features. Likewise, the variable names are just for purposes of explanation; you do not need to match them exactly.)

Your graph implementation must support the following operations:

- Create a new (empty) graph
- Add a vertex with key `k`, a value `val`, and (optional) list of neighbors

- Delete a vertex with key `k`
- Add an edge from vertex `k_u` to `k_v` with weight `w` (if the edge already exists, update its weight to `w`)
- Delete an edge from `k_u` to `k_v` (if it exists)
- Get the weight of the edge between vertices `k_u`, `k_v`

If you are using the provided Python starter code, you will find function definitions for each of these functions already there. You are free to modify variable names, argument types (e.g. passing Vertex objects as the input to a function instead of a key), etc as long as the general purpose of the operations remains the same.

You can also define any other functions that you find useful. You do not have to handle error checking for unexpected user inputs (although it may be helpful for debugging if you write informative error messages).

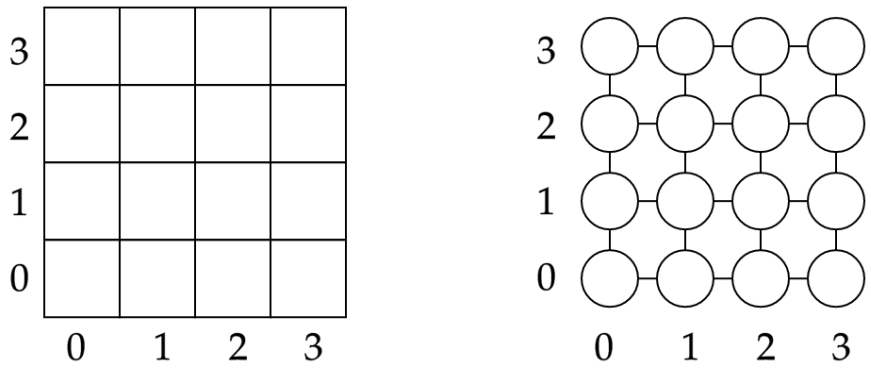
In your design document, write a short paragraph explaining how you chose to implement your graphs (adjacency lists vs. adjacency matrices; specific data structures; etc).

Finally, add a short tutorial/example to your design document demonstrating how to use your functions.

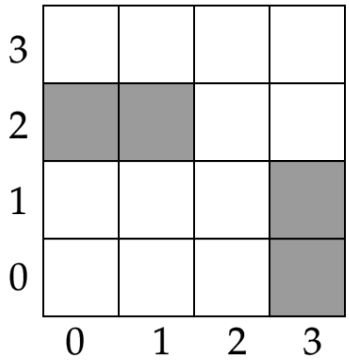
Implementing Maps

Next, we'll implement maps on top of our new graph implementation. In the context of this project, a map is a simple n x n grid, possibly with some obstacles.

Here is a 4-by-4 map, along with a graph representing the same map.



Movement is allowed between adjacent squares (north, south, east, and west). Here is another 4-by-4 map, this time with some obstacles (gray squares):



Movement is not allowed into/through a square that contains an obstacle. It is up to you how to implement obstacles. Here are a few possibilities to consider:

- Remove the vertex (and its edges) entirely
- Remove all of the vertex's incident edges
- Set the weight of each incident edge to +infinity

Using the graphs from the previous section, implement n-by-n maps with obstacles. Your implementation must support the following operations:

- Create a new n-by-n map
- Add an obstacle to the map at space (x,y), blocking movement through that space
- Remove an obstacle at (x,y), restoring normal movement through that space
- Check to see if an obstacle exists at (x,y)
- Display a pretty ASCII picture of your map using different characters to represent open spaces and obstacles. (Can be printed to stdout, a plaintext file, ….)

Add a short tutorial/example to your design document that shows off your map-making functionality. Include a sample ASCII map (with a key that explains what each character represents).

Part A Checklist

When you are finished with Part A, your project files should include the following elements:

Code

- Graphs
    - Create new graph
    - Add/delete vertex
    - Add/delete edge
    - Get edge weight
- Maps
    - Create new map

- Add/remove obstacle at (x,y)
- Check if obstacle at (x,y)
- Display map as ASCII

Design document

- A short paragraph explaining graph implementation, including:
  - Which data structures are used to store vertices, edges, and weights?
  - Why did you choose them?
- Graph tutorial/demo
- A few sentences explaining how obstacles are represented in your maps
- Map tutorial/demo
  - Sample map + key