Programming Assignment 5 - Dijkstra's Algorithm

Due: May 3

This is an individual assignment!

You are responsible for adhering to the Aggie Honor Code and student rules regarding original work. You must write and submit your own code to Mimir. Plagiarism checks will be used to check the code in addition to a manual review from the grader.

There is a provided `report.txt` where you will track all of the references you use. You may use references to help you get a high-level understanding of the problem, but remember:
Do not share code! Do not copy code!

Overview

For our final programming assignment, we will look at the problem of pathing, also referred to as pathfinding: https://en.wikipedia.org/wiki/Pathfinding. In short, the goal of pathfinding is to locate an optimal path between two nodes on a graph.

Pathfinding and Graphs

Pathing and graphs are intrinsically linked because graphs are used to model so many kinds of problems. Lists and trees can both be modeled using graphs, and graphs have many real-world applications like road layout for GPS, network topology for internet traffic, airport flight planning, flow of information between neurons in the brain, mutual connections and relationships between nodes in social networks and search engines, and more. We have talked about some of these applications already, and you are welcome to explore more! Graphs are the capstone of this class and will appear in future CS classes.

Dijkstra's Algorithm

The best-known algorithm for finding the shortest path between two vertices of a graph is *Dijkstra's Algorithm*: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm. This algorithm is distinct from the traversal methods we discussed (BFS and DFS) in that it makes determinations based on cost or weight of edges. A simple example is flight routing between airports; the edges could represent the distance in miles.

Maze Solving Application

Throughout our PAs this semester, my intention has been to provide a sampling of real-world use cases for the concepts we cover in class. We have seen how course topics relate to networking hardware, software distribution and libraries, machine learning, cryptography, block chain, and more. I hope you have seen enough to understand the core underlying principles, while also appreciating that we are only scratching the surface. One could spend years studying any of these topics!

The last domain we'll be exploring in a PA is AI, particularly how it applies to pathing and maze solving. Since graph search is one of the key problems in traditional AI, this is a perfect time to visit it. However, because this is a short PA, being only 2 weeks in length, we will focus on algorithmic construction. Mimir will provide test cases that showcase the maze solving application.

Getting Started

1. Go to the assignment on Mimir and download the starter code.
    a. There is a `report.txt` file where you should track the references you use for this assignment. There are several questions to answer as well, once you complete the assignment.
    b. The following files are fully implemented and will not need to be edited:
        i. `LocatorHeap.h` - a min-heap implementation with locators for direct access
    c. Both Graph.h and main.cpp have skeleton code. Mostly what has been provided ensures the input/output tests on Mimir match the expected result.
2. I recommend implementing the graph structure first. In this assignment, you'll use an adjacency list, but this version is slightly different from the one in lab. Here, you save vertices as objects which contain a list of edge pointers. For simplicity, you can use the graph structure to hold both the vertices and edges to make dynamic memory operations easier.
3. The core task is to implement Dijkstra's algorithm. The algorithm returns the shortest path as a list, starting with the start node and printing all the nodes to take until reaching the end node.
4. You will need to finish the main.cpp file to read the graph files for Mimir's tests.
    a. The files are written with the first line giving the number of vertices followed by the number of edges.
    b. The next lines are the edges themselves, along with their weight.
        i. **Note**: In this assignment, all graphs are undirected (bidirectional), so you should add bidirectional edges for each edge you're given!
    c. The last line will be the start and end node between which to find the shortest path using Dijkstra's algorithm.
5. You won't need to create a makefile for this assignment, but you may. Likewise, you may rely on some STL structures, such as map or unordered_map to help track information for the shortest path.
6. Be sure to update the bibliography in the report to track your references!

Task Breakdown

Unlike previous PAs, this is only a 2-week assignment. For that reason, the scope is smaller, and it is not divided into two parts. It may look like a lot, but you'll find everything is pretty trivial except completing the rule of 3 for the graph and implementing Dijkstra's algorithm.

Graph.h

Vertex and Edge

Graph.h provides simple vertex and edge implementations. The final graph structure is going to be similar to the adjacency list you saw in lab, only instead of the graph storing a vector of vectors, we will store a vector of Vertex pointers. The Vertex objects contain vectors of their edges. This will yield a slightly more efficient adjacency list.

Additionally, we have simplified the path finding problem by saving space in the Vertex object to hold information like “visited” and “distanceTo”. You are free to ignore these if you choose. Having the ability to store them with a vertex can save the effort of putting them elsewhere, but you can certainly use auxiliary data structures to hold such information. In the traversal lab, you probably used an auxiliary structure to hold “visited” booleans, and in the report for this assignment, you’ll be asked about different approaches you could use.

As further practice of overloading operators, you should implement the operator< to compare vertices. This is being included for pathfinding, so you will use the “distanceTo” for comparison.

Vertex and Edge	Description
bool operator< (const Vertex &v)	This comparison will be used to determine the relative distance of two vertices. You can overload the operator by just returning the result of comparing the distance members.

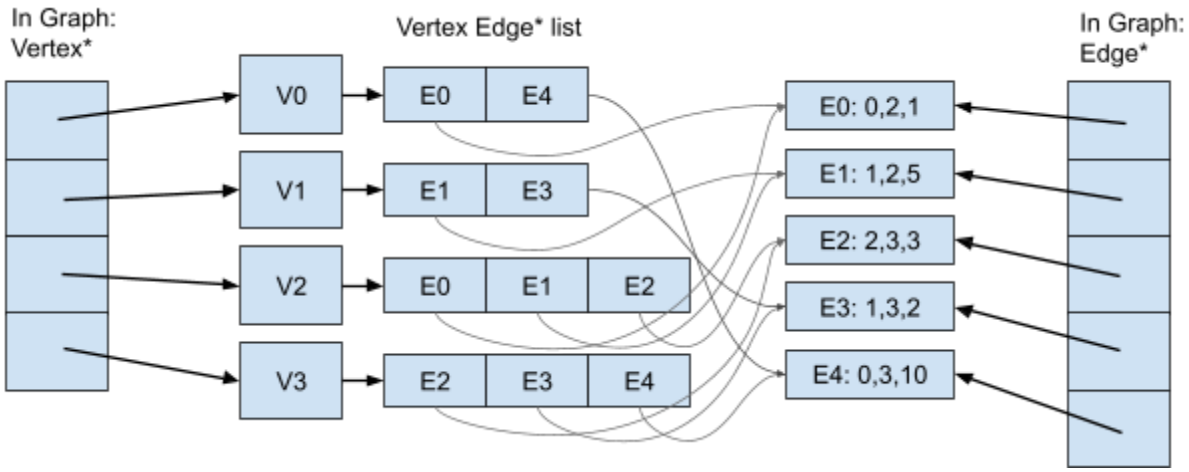
Graph

Data Members

The graph itself will save a list of Vertex and Edge pointers. Again, this is similar to the adjacency list you saw in lab but with actual objects to hold the information.

The advantage of having lists (vectors) to hold the vertex and edge pointers at the graph level is that it moves the dynamic memory operations into the Graph. For example, you can use these members to clear all the dynamic memory in the Graph destructor.

The following image shows an illustration of how the information relates to each other in this Graph structure. The left-hand side represents a list or vector of Vertex*, and the right-hand side has the corresponding Edge* list. These are the only two members in the Graph. Everything else is stored in memory and just associated with each other through pointers.



Rule of Three

The Graph destructor should clear all vertices and edges associated with the graph. Also, to further practice "rule of 3" concepts,you should implement the copy constructor and copy assignment operator. It may be useful to create helper functions for clearing and copying a graph.

Inserting Vertices and Edges

When given a new vertex label, create the new Vertex object and push back the pointer into the graph's vertex list. Note that we are using integer labels for ease; this allows all vertices to be indexed directly using their labels. You can imagine that string or char labels could still be indexed in certain structures, such as a hash table.

When given the information for an edge, create a new edge going from vertex `l1` to `l2` with the given `weight`. This should be pushed back onto the edge list of the graph. Also, you should push the edge into the Edge* list belonging to the vertex.

All edges will be bidirectional / unidirectional in this assignment. You can just add the single edge in this function and call it with the opposite direction later in the runner (Mimir assumes this), or you can create the opposite direction edge inside the insertEdge function, as long as your logic is careful.

Finding Shortest Path

The shortestPath() function takes the labels of a starting node and an ending node. It will use Dijkstra's algorithm to compute the shortest path and return the sequence of nodes (a vector of Vertex*) which should be traversed from start to end.

We will discuss more about Dijkstra's algorithm in lecture, including pseudocode, but the idea of the algorithm is very straightforward.

1. First, you should set all the vertex visited values to false, clear the pathTo, and reset the distanceTo to "infinite" (use the max value that can be stored in a float). Alternatively, you can create auxiliary structures to track some or all of this information. A vector of booleans can work well to track visited nodes in this example since all labels are integer-based, or you can use STL structures like maps, sets, unordered_maps, etc.
2. Next, set the start node as the current node and set its distance to 0.

3. Use a priority queue to store all the nodes. A heap-based priority queue is provided in LocatorHeap.h. This is a min-heap where you can save Vertex pointers which can be compared using the '<' operator you defined. The heap is important in selecting the next node to visit.
    a. **Important note**: This is a locator heap. We touched on the concept in lecture, even though we have not used one in practice. When we add elements, a locator heap returns a reference (pointer) to that element. You can use an auxiliary structure like a vector or map to save the locators, or store them inside the vertex object. These allow direct access to elements inside the heap and will be useful for efficiently updating distances of vertices during the shortest path calculation.
4. Remove the minimum element from the heap and check the distance to all neighbors. The edge list is useful here. If you find the current path yields a distance lower than the one already saved for a vertex, update the value for the distance in the vertex and heap.
    a. Removing the minimum element at each step makes Dijkstra's algorithm a *greedy* algorithm in which it selects the locally optimal choice. There are other ways to do the selection, and a variety of algorithms that build on Dijkstra's add heuristic information to enhance the result. More pathing algorithms will be seen in lecture and lab.
5. Mark the current node visited.
6. Repeat steps 4 & 5 until the end node is marked visited.

Graph	**Description**
Data member to hold the Vertex* and Edge* for the graph	We'll hold pointers so that it is easy to store the same object in multiple places, such as the Vertex edge list, the locator heap, or any other auxiliary data structure you use.
Rule of Three functions: - destructor (declaration in starter) - copy constructor - copy assignment	Vertex and Edge creation is handled by the graph, so you'll want to follow the rule of 3 guidelines for supporting dynamic memory operations. This means there should be a destructor to free memory, as well as copy constructor and copy assignment for deep copies of graphs.
insertVertex(int) insertEdge(int,int,float)	You only need to implement the insert operations for the graph structure. This consists of allocating a new vertex with the given integer label and saving it in the graph list of vertices. For inserting an edge, allocate the new edge and add it to both the graph list of edge pointers and the edge list for the originating vertex. The Mimir tests are written to insert both edge directions. For this assignment, all edges will be undirected. Also, note you are only implementing insertion; we are not testing removal for this graph.
vector<Vertex*> shortestPath (int start, int end);	The shortestPath function should implement Dijkstra's algorithm, which is described above, to return a list of vertices (vector of Vertex* type) to traverse along the optimal path to the end node.

main.cpp

The main file has several snippets that need to be filled in, and the starter code provides comments for these segments.

1. You should read in the user-provided filename and use fstream to open the file.
2. The file is organized into 3 segments:
    - The first line is the number of vertices and then number of edges.
    - The middle portion is the collection of edges, with each line representing the node labels and the weight of an edge.
        - **Note:** you are creating a unidirectional graph, so make sure to insert the bidirectional version of each edge into your graph!
    - The last line is the start and end node to find the shortest path between.

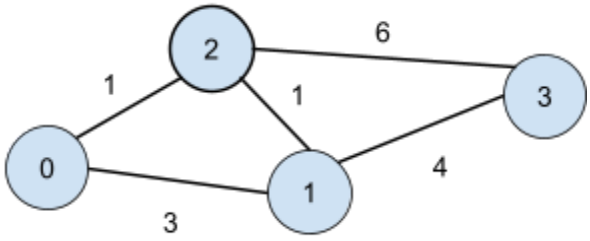
You are given all the output to reduce issues with the autograder. This program shouldn't be difficult since it is only extra practice with I/O and file parsing.

An example file and possible representation of its graph is shown below. The file starts by identifying there are 4 vertices and 5 edges. The next 5 lines define the edges. The last line specifies that the goal is to find the shortest path between nodes 0 and 3. In this case, that would be 0, 2, 1, and 3 for a total cost of 6.

graph file

```
4 5
0 1 3
0 2 1
1 2 1
1 3 4
2 3 6
0 3
```

graph representation



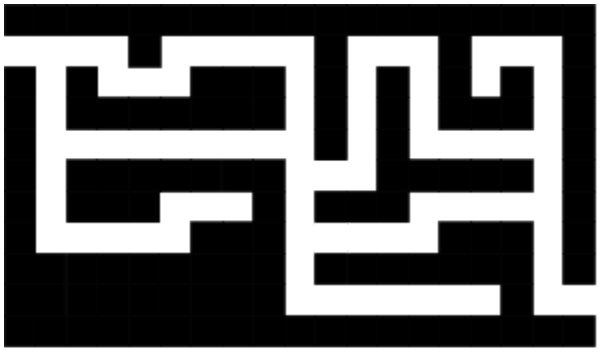
At the bottom of main.cpp you'll see a for loop that looks slightly different than ones we've used before. This is a "for-each" loop. Just as we upgraded from simple counting loops to iterators that gave us pointers, we can upgrade further to "for-each" logic. These coding patterns use iterators to traverse an object. You get the dereferenced item itself, making it even easier to traverse without the hassle of specifying starting and ending conditions.

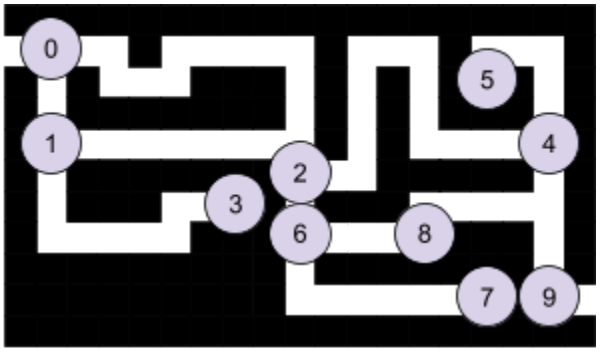
Maze Solving Application

You have completed the PA code by this point! Mimir will provide the results of the maze finding application using your Dijkstra implementation.

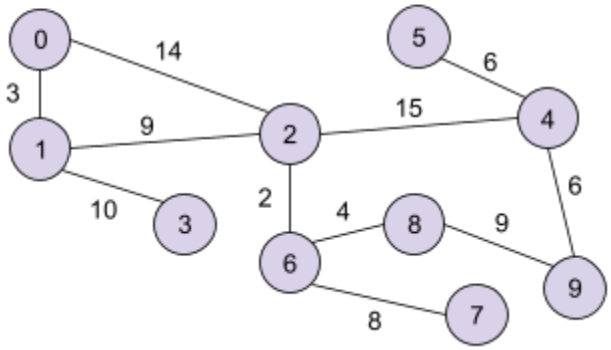
The idea is simple: if we can represent points along a path using graph nodes and assign distances between them using weighted edges, we can model the problem space and use pathfinding algorithms to find routes.

The top set of images below show a pixel block maze. If we overlay nodes at key points on this maze, for instance at each branching point and end of path, we can create a graphical representation of it. We show a file and corresponding structure that models the maze below, using pixels as the "distance" or cost.



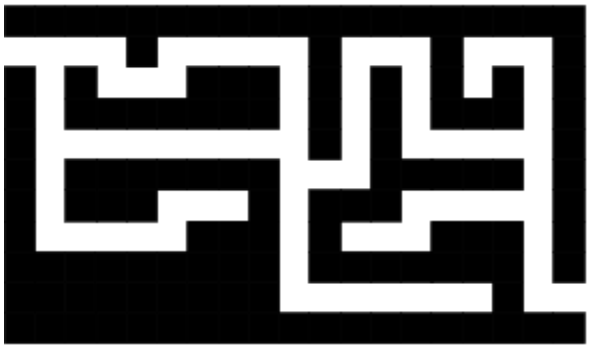


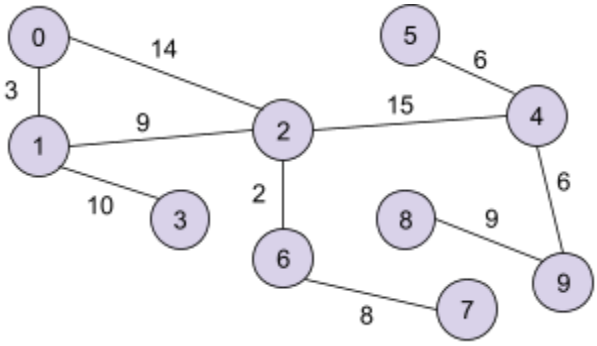

```
10 11
0 2 14
0 1 3
1 2 9
1 3 10
2 4 15
2 6 2
4 5 6
4 9 6
6 8 4
6 7 8
8 9 9
0 9
```



From this visualization, we can quickly see that Dijkstra's is relevant to solving the problem. I can query the best route from 0 to 9, and in this case, it will return the path 0, 1, 2, 6, 8, and 9.

If we modify the maze to close the pathway between nodes 6 and 8, the model and route will change.





Even though the maze is not difficult to solve, it demonstrates the approach.

In traditional AI, graph search is one of the main problem spaces, and we can see how it is used in game play where the AI usually needs some sort of spatial reasoning to interact with a player. Dijkstra's can be used, but most of the time we want a faster algorithm that can make a good choice quickly. We use "heuristic" search to help with this. Heuristics are sets of rules and estimates that can be used to bypass expensive operations in an algorithm.

A very famous algorithm for heuristic search, which builds on Dijkstra's, is A* (pronounced A-star). A* is used in some video games. The original Starcraft used A* for certain things, and even with its fast operation, computers of the time could be overtaxed by graph search algorithms. In an anecdote shared from one of the original Starcraft developers, they ended up turning off collision detection for certain units to avoid spending too much CPU power on pathfinding activities[1].

AI is an interesting field in itself, and it goes well beyond just graph search.

Report

Remember to update your references and to answer the few questions in the report. They ask you to think about the approach you used and give examples of alternatives you could try.

Point Distribution

Unlike many of the previous assignments, this PA is largely evaluated through one algorithm. You'll need to get enough of the graph structure working on your own to begin implementing Dijkstra's algorithm. The shortestPath() function is utilized in all the graph tests, so after it is written, you'll be able to fill in the details for the rule of 3 and other operations.

You are given several sample graphs to help test the main.cpp program.

The points are organized as follows:

- 40% for graph structure and Dijkstra's algorithm
- 10% for copy constructor and copy assignment
- 18% for destructor and memory checks
- 12% for maze solver application (provided by Mimir but leverages your shortestPath function)
- 20% for report and manual review

[1] This hack continues to be a part of the more recent Starcraft games, even though modern computers don't really need to do this.