Assignment 6: Trailblazer

*Developed by Keith Schwarz and Dawson Zhou, and based on the **Pathfinder** assignment developed by Leonid Shamis at UC Davis, Stanford's **Pathfinder** assignment by Eric Roberts and Julie Zelenski, and and the **Maze Generator** assignment by Jerry Cain.*

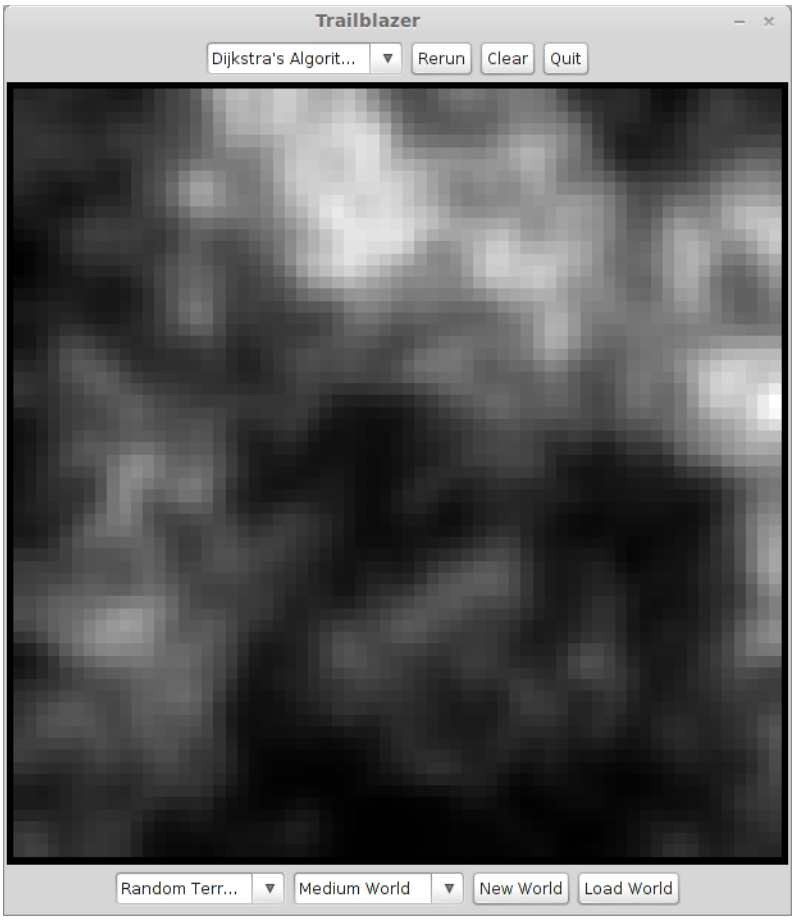
For your final assignment of the quarter, you will use your skills to implement three classic graph algorithms – Dijkstra's algorithm, A* search, and Kruskal's algorithm – to build and navigate different terrains. In the course of doing so, you'll get to see how real pathfinding algorithms work and will gain familiarity representing abstract structures (namely, graphs) directly in software. By the time that you're done with this assignment, you will have a much deeper understanding of graphs and graph algorithms. You'll be more comfortable turning high-level pseudocode directly into executable C++. And, to top it off, you'll have a really nifty piece of software!

Due Friday, August 16 at 11AM.
No late submissions can be accepted, and no late days can be used.

The Program

Your task in this assignment (described in more detail later on) is to implement Dijkstra's algorithm and A* search so that you can find shortest paths across various terrains, and then to implement Kruskal's algorithm in order to generate random mazes.

To let you see your algorithms in action, we have provided a graphical program you can use to test and visualize your implementations of the three algorithms. When you start up the program, you will see a window containing a randomly-generated terrain. Here, bright colors indicate higher elevations, while darker colors represent lower elevations. Therefore, mountain ranges will appear in bright white, while deep canyons will appear black.



If you click on any two points in the world, the program will use the algorithm selected in the top dropdown menu (either Dijkstra's algorithm or A* search) to find a shortest path from the starting position to the ending position. As it does so, it will color the nodes green, yellow, and gray based on the colors assigned to them by the algorithm. Once the final path is found, the program will highlight it and display the total path cost in the console.

The window also contains several controls. You can generate random terrains of different sizes (small, medium, large, and huge) by choosing the "Random Terrain" option from the bottom-left dropdown menu and choosing the size from the bot-

tom-right dropdown menu, then clicking the "New World" button. Alternatively, you can run the pathfinding algorithms in a 2D maze by choosing the "Random Maze" option from the bottom-right dropdown menu and clicking the "New World" button. Alternatively, you can load a preset world from a file on-disk by choosing the "Load World" button and selecting the file with the file chooser.

When you first begin working on this assignment, the pathfinding algorithm will not be implemented and you will get an error message if you try to find paths between pairs of points. Similarly, the maze generation code will not have been written, so the Random Maze option will not work. Therefore, we suggest trying out our sample application to get a sense for how to use the program and to see the expected behavior. That way, when you start making progress on the assignment, you will have a better sense for what to expect.

The Starter Code

We have provided a lot of starter code for this assignment. Here is a quick breakdown of what each file contains. We recommend that you open and read through the files marked with a star:

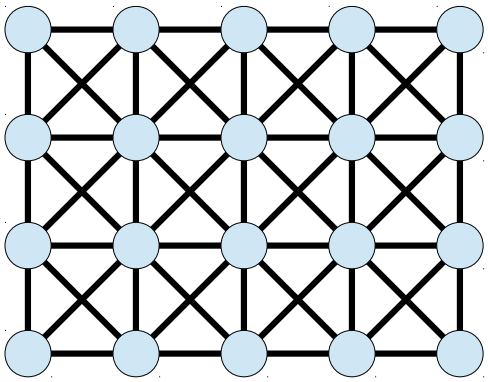
- ★ **`Trailblazer.h`** / **`.cpp`**. These are the files where you are likely to do most, if not all, of the work for the assignment. You will write your implementations of the three graph algorithms in this file.
- ★ **`TrailblazerTypes.h`** / **`.cpp`**. Defines several types fundamental to the assignment, such as types for storing locations, edges, and colors. You should not need to modify these files.
- ★ **`TrailblazerPQueue.h`**. Defines the type **`TrailblazerPQueue`**, a slightly modified version of **`PriorityQueue`** that supports the **`decreaseKey`** operation. You will almost certainly want to use this version of the priority queue instead of the normal **`PriorityQueue`** type. You should not need to modify this file.
- **`TrailblazerConstants.h`**. Defines constants needed across multiple modules. The constants here are mostly used by driver code, so you should not need to directly use or modify this file.
- **`TrailblazerCosts.h`** / **`.cpp`**. Defines functions for computing the cost of moving from square to square in terrains and mazes, as well as heuristic functions. You should not need to directly use or modify these files, though you may want to do so as an extension.
- **`TrailblazerGraphics.h`** / **`.cpp`**. The main program module, which is responsible for setting up the window, interacting with the mouse, loading files, etc. You should not need to modify these files.
- **`WorldGenerator.h`** / **`.cpp`**. Defines functions for generating terrains and mazes. You should not need to use or directly modify these files.

Although there is a lot of code here, you will most likely only need to modify **`Trailblazer.h`** and **`Trailblazer.cpp`**. Most of the other files are either implementations of helper functions or part of our driver code.

Step One: Implement Dijkstra's Algorithm

Your task for this portion of the assignment is to implement Dijkstra's algorithm so that it is possible to find the shortest path from one point in the world to another. If you think of the terrain as a graph (described in more detail in the next section), then finding the shortest path from the start location to the destination location ends up being equivalent to finding the shortest path in an appropriate graph.

The world that we give to you is represented by a **Grid<double>**, where each **double** represents an elevation between **0** (lowland) and **1** (high mountain peak). Although we aren't presenting this **Grid** to you as a graph, you can treat it as if it were a graph by assuming each grid location is connected to the eight cells directly adjacent to it, as shown here:



(This is similar to how the cells in the Boggle grid were connected to one another.)

To represent the fact that it is difficult to move through the terrain, each edge in the graph has a cost associated with it, which represents the amount of work required to follow that edge. We will provide you a function that, when provided two adjacent locations on the grid, will tell you the cost of the edge connecting those locations. The function will return a nonnegative real number representing the cost of following that edge. The cost depends both on the direction of movement (moving diagonally takes more work than moving horizontally or vertically) and the change in altitude (climbing or descending a steep gradient takes more work than walking on level ground).

Your task for this portion of the assignment is to implement a function with the following signature:

```
Vector<Loc>
shortestPath(Loc start, Loc end,
             Grid<double>& world,
             double costFunction(Loc one, Loc two, Grid<double>&
world));
```

This function takes in four parameters, which are described below.

- **Loc start** and **Loc end**. The **Loc** type is a **struct** with two fields – **row** and **col** – that identify locations on a grid. It's defined in the **TrailblazerTypes.h** header file. The **start** and **end** parameters give the source and destination cell on the grid, and your function will compute a shortest path between them.
- **Grid<double>& world**. This **Grid** represents the terrain in the world, as described above.
- **double costFunction(Loc one, Loc two, Grid<double>& world)**. This parameter is a function (see the **Functions as Parameters** handout for more details about this). It takes in three arguments – two **Loc**s representing points on the grid, and the grid itself – then returns the cost of the edge between those two points. You will need to use this function to determine the cost of moving from one grid location to another.

Your function should use Dijkstra's algorithm to compute a shortest path from **start** to **end**, and then return that path as a **Vector<Loc>** containing all of the points in that path in the order in which they appear, starting with **start** and ending with **end**. The path should include both **start** and **end**, as well as the intermediary nodes.

An important note: The version of Dijkstra's algorithm suggested in the course reader is slightly different than the version we discussed in lecture and is less efficient. Your implementation of Dijkstra's algorithm should follow the version we discussed in lecture. Specifically, your implementation should not store complete paths in the priority queue. Instead, the priority queue should just store nodes, and you should use an auxiliary structure to reconstruct the shortest path tree.

For reference, the pseudocode for the version of Dijkstra's algorithm described in lecture is on the next page of this handout.

Dijkstra's Algorithm from node *startNode* to node *endNode*:

- Color all nodes gray.
- Color *startNode* yellow.
- Set *startNode*'s candidate distance to 0.
- Enqueue *startNode* into the priority queue with priority 0.
- While the queue is not empty:
  - Dequeue the lowest-cost node *curr* from the priority queue.
  - Color *curr* green. (The candidate distance *dist* that is currently stored for node *curr* is the length of the shortest path from *startNode* to *curr*.)
  - If *curr* is the destination node *endNode*, you have found the shortest path from *startNode* to *endNode* and are done.
  - For each node *v* connected to *curr* by an edge of length *L*:
    - If *v* is gray:
      - Color *v* yellow.
      - Set *v*'s candidate distance to be *dist* + *L*.
      - Set *v*'s parent to be *curr*.
      - Enqueue *v* into the priority queue with priority *dist* + *L*.
    - Otherwise, if *v* is yellow and the candidate distance to *v* is greater than *dist* + *L*:
      - Set *v*'s candidate distance to be *dist* + *L*.
      - Set *v*'s parent to be *curr*.
      - Update *v*'s priority in the priority queue to *dist* + *L*.

Aside from the restriction on which version of Dijkstra's algorithm you use, you are free to implement this function however you see fit and can use any data structures that you'd like when doing so. Before you start coding anything up, we suggest thinking about the following questions:

- How will you keep track of the parent cell for each cell in the grid?
- How will you track which cells are unvisited (gray), expanded (green), or enqueued (yellow)?
- How will you keep track of the distances to each node?

While you do not need to aggressively optimize your implementation, you should try to choose data structures that are efficient for the task at hand. Part of your style grade for this assignment will be based on the choices you make here, so we suggest providing comments explaining your decisions. For example, it would probably not be a good idea to keep track of which cells are green using a **`Vector<Loc>`**, since doing so would be inefficient and inelegant. As a reminder, you might find the **`TrailblazerPQueue`** type useful here, as it supports **`decreaseKey`**.

When running Dijkstra's algorithm, your program will calculate the shortest paths to multiple different nodes, not just the single destination node that you had in mind. This is normal and is related to how Dijkstra's algorithm works. To help you get a better sense for what Dijkstra's algorithm is doing, we have provided the following function in **`TrailblazerGraphics.h`** to color squares in the world:

```
void colorCell(Grid<double>& world, Loc loc, Color color);
```

This function accepts three parameters – the world grid, a location in that world, and a color – then colors the specified location in the world the color given by **`color`**. (Note that the **`Color`** enumerated type is defined in **`TrailblazerTypes.h`**. If you are not familiar with enumerated types, you can read more about them in Chapter 1.5 and 1.7 of the course reader). Your **`shortestPath`** function should call **`colorCell`** in the following circumstances:

- When enqueuing a node into the priority queue for the first time (i.e. coloring it yellow for the first time), you should call **`colorCell`** to highlight that node yellow.
- When dequeuing a node from the priority queue (i.e. converting it from yellow to green), you should call **`colorCell`** to highlight that node green.

Since the display starts off with all cells colored gray, you don't need to use **`colorCell`** to color all initial cells gray. If you do, you might see a long pause before your algorithm starts running, since your program is busy redrawing the world.

Although we provide you a **`colorCell`** function for this part of the assignment, you still need to keep track of the node colors yourself. The **`colorCell`** function just updates the display.

As your algorithm runs, the above calls will help you visualize exactly what the algorithm is doing, which will help you get a better feel for how the algorithm works. (Plus, it is really, *really* fun to watch!) In the next part of the assignment, you'll be able to use this code to compare how many nodes Dijkstra's algorithm explores to how many nodes the A* search algorithm explores.

Before moving on to the next section, we strongly suggest testing your algorithm on a variety of test worlds. We've provided several different sample worlds, and you can compare the paths that your algorithm finds against the paths that our reference implementation finds. Note that there might be many different but equally valid paths between two points, so your algorithm might not produce the exact same path as ours. However, your algorithm should always return a path with the same total cost as ours; if the costs differ, there is likely an error in your algorithm. To help with testing, our starter code will automatically print out the cost of the path that your function returns.

For reference, our implementation of this function is roughly 75 lines of code, including whitespace, comments, and function prototypes. If you find yourself writing dramatically more code than this, you might want to reevaluate your solution.

Step Two: Implement A* Search

For the next part of this assignment, you will modify your shortest path code so that it supports both A* search and Dijkstra's algorithm. Modify the signature for **`shortestPath`** so that it looks like this:

```
Vector<Loc>
shortestPath(Loc start, Loc end,
             Grid<double>& world,
             double costFunction(Loc one, Loc two, Grid<double>& world),
             double heuristic(Loc start, Loc end, Grid<double>& world));
```

(You will need to change the function signature in two places – once in the **`Trailblazer.cpp`** source file, and once in the **`Trailblazer.h`** header file. If you don't change the signature in both places, you will get a linker error.)

Your function now accepts a heuristic function as an additional parameter. This heuristic function takes in three parameters – a start location, an end location, and the terrain – then provides a guess of the distance from the start location to the end location. You can assume that this is an admissible heur-

istic, meaning that it never overestimates the distance to the destination node. Using this extra parameter, modify your function so that it performs A* search instead of using Dijkstra's algorithm. For reference, the pseudocode for A* search is given on the next page of this handout.

Note that even though your function now takes in a heuristic function, you can still use this new function to run Dijkstra's algorithm. Recall that if the heuristic function is a function that always evaluates to 0, then A* search and Dijkstra's algorithm behave identically. In fact, our starter code is programmed so that when you change your implementation to take in the extra parameter for the heuristic, we will automatically call it and pass a function that always evaluates to zero when you choose "Dijkstra's Algorithm" from the drop-down menu.

If you click on the A* button at the main program, our starter code will call your function and pass in as the final parameter a heuristic function we've specifically crafted to work with our distance function. Because your function already marked all locations that are enqueued when finding a shortest path, once you change your code to incorporate the heuristic you should immediately see a difference in how many locations are explored during A* search. Pretty amazing, isn't it?

You should need to make only minor code changes for this part of the assignment. Our implementation required us to change roughly five lines of code to convert from Dijkstra's algorithm to A* search, though depending on how you've coded up your solution you might need to change more code (or perhaps even less!) It might help to review the pseudocode for A* (given on the next page) and compare it to the pseudocode for Dijkstra's algorithm.

A* search from node *startNode* to node *endNode* using heuristic *h*:

- Color all nodes gray.
- Color *startNode* yellow.
- Set *startNode*'s candidate distance to 0.
- Enqueue *startNode* into the priority queue with priority *h*(*startNode*, *endNode*).
- While the queue is not empty:
  - Dequeue the lowest-cost node *curr* from the priority queue.
  - Color *curr* green. (The candidate distance *dist* that is currently stored for node *curr* is the length of the shortest path from *startNode* to *curr*.)
  - If *curr* is the destination node *endNode*, you have found the shortest path from *startNode* to *endNode* and are done.
  - For each node *v* connected to *curr* by an edge of length *L*:
    - If *v* is gray:
      - Color *v* yellow.
      - Set *v*'s candidate distance to be *dist* + *L*.
      - Set *v*'s parent to be *curr*.
      - Enqueue *v* into the priority queue with priority *dist* + *L* + *h*(*v*, *endNode*).
    - Otherwise, if *v* is yellow and the candidate distance to *v* is greater than *dist* + *L*:
      - Set *v*'s candidate distance to be *dist* + *L*.
      - Set *v*'s parent to be *curr*.
      - Update *v*'s priority in the priority queue to *dist* + *L* + *h*(*v*, *endNode*).

To help you compare the behavior of Dijkstra's algorithm and A* search, we have provided a "rerun" button at the top of the window. If you click on this button after performing a search between two points, it will repeat that search using the algorithm currently selected in the drop-down menu at the top of the program. Try running a search using Dijkstra's algorithm, switch the algorithm choice to "A* Search," and run that search again. You might be pleasantly surprised by just how much more efficient A* search is!

To test whether your program is working correctly, try comparing the output of your program to the output of our reference solution. As before, you might find slightly different shortest paths than our reference solution because there might be multiple possible shortest paths, but your implementation should not return a path whose total cost differs from our path. You should also look at which locations the implementations explored; if there's a huge difference between the two, that might indicate a bug in your code.

Additionally, since our provided heuristic is an admissible heuristic, your A* search algorithm should always return a path with the same cost as the path found by Dijkstra's algorithm. If you find that the algorithms give paths of different costs, it probably indicates a bug in your solution.

Step Three: Implement Kruskal's Algorithm

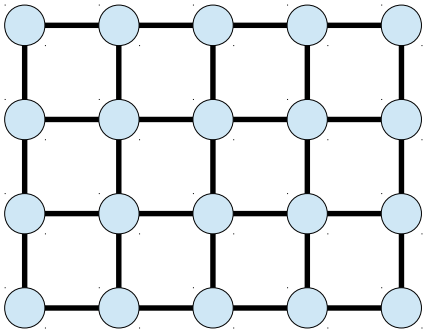
You now have a slick, efficient pathfinding algorithm that can navigate a variety of different terrains. But how well would it do trying to help you get out of a maze?

Your final task in this assignment is to implement a maze generator using Kruskal's algorithm. If you'll recall from lecture, Kruskal's algorithm can be used to find a minimum spanning tree for a given graph. The pseudocode for Kruskal's algorithm is described below:

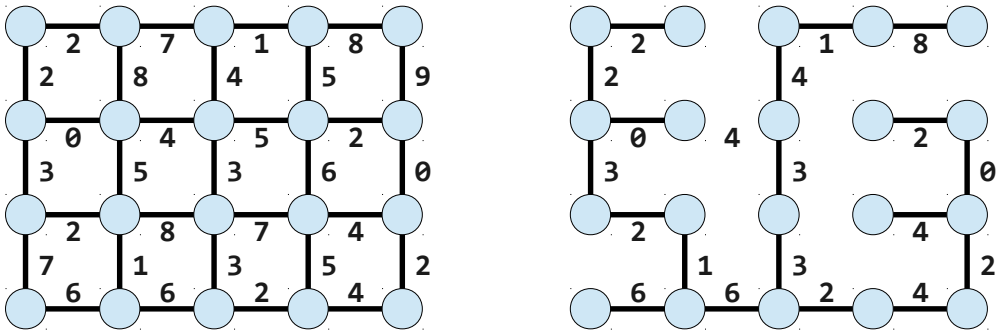
Kruskal's algorithm on a graph:

- Place each node into its own cluster.
- Insert all edges in the graph into a priority queue.
- While there are two or more clusters remaining:
  - Dequeue an edge *e* from the priority queue.
  - If the endpoints of *e* are not in the same cluster:
    - Merge the clusters containing the endpoints of *e*.
    - Add *e* to the resulting spanning tree.
- Return the spanning tree formed this way.

In lecture, we saw how Kruskal's algorithm can be used to find minimum spanning trees and perform data clustering. You can also use Kruskal's algorithm will produce mazes. Specifically, suppose that you have a *grid graph*, where the nodes are connected as follows:

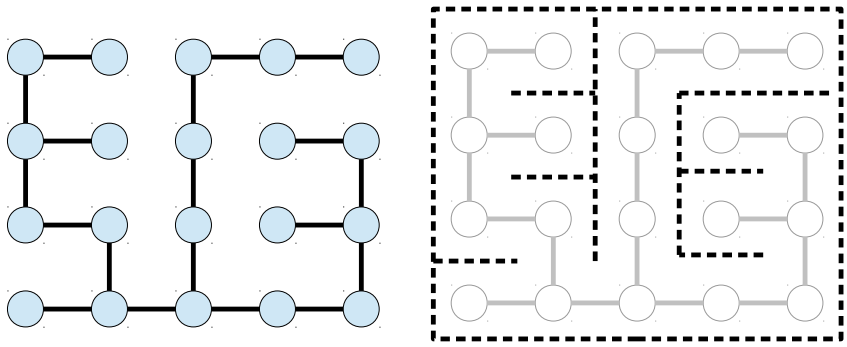


If you assign each edge a random weight and then run Kruskal's algorithm on the resulting graph, you will end up with a spanning tree; there will be exactly one path between each pair of nodes. For example, assigning the edges in the above graph weights as follows and running Kruskal's algorithm would produce the following result:



Take a minute to make sure you understand why Kruskal's algorithm produces this tree.

Interestingly, you can think of this tree as a maze. Typically, when drawing a maze, lines represent walls, which are impassable, and emptiness represents floors, which are passable. The above tree has the opposite property: lines represent edges, which are passable, and empty space represents the absence of edges, which is impassable. You can turn the above tree into a maze by drawing lines in all of the empty space, as shown here:



Your task in this part of the assignment is to use the randomized version of Kruskal's algorithm to generate a random maze. Specifically, your task is to write a function with the following signature:

```
Set<Edge> createMaze(int numRows, int numCols);
```

This function will accept as input two integers representing the number of rows and columns in the maze (the number of nodes in each row and column of the grid graph), then generate a random maze of that size using Kruskal's algorithm. The function should return **`Set<Edge>`** representing the edges present in the spanning tree that you generated with Kruskal's algorithm. Here, the **`Edge`** type (defined in **`TrailblazerTypes.h`**) is a **`struct`** that stores two **`Loc`**s representing the two endpoints of the edge.

Note that unlike the first part of this assignment, we only provide you the dimensions of the graph as input to the function, rather than the graph itself. Your function will need to construct the grid graph with the specified dimensions and assign random weights to each edge (to get a good distribution of edge weights, we recommend using **`randomReal(0, 1)`** to assign the weights).

Before you start coding anything up, we suggest thinking about the following questions:

- How will you keep track of the nodes and edges in the graph?
- How will you keep track of which nodes are in each cluster?
- How will you determine which cluster a node belongs to?
- How will you merge together two clusters?

Note that for this part of the assignment, you will need to explicitly construct all of the edges in the grid graph. You might want to draw some pictures to figure out how you are going to do this.

Our starter code is programmed to call your function to generate random mazes whenever you use the New World button with "Random Maze" selected. Once you've generated a maze, you can run your Dijkstra's or A* implementation to find paths between two points in the maze. If you'll recall, your shortest path function takes as input two functions – one giving the cost of an edge between two points and one providing a heuristic. When you use our provided program to find the shortest path between two points in a maze, we will pass into **`shortestPath`** a cost function that assigns costs as follows:

- The cost of stepping onto a wall of the maze is infinite.
- The cost of taking a step diagonally is infinite.

- The cost of taking a step in a cardinal direction is a small constant.

This cost function ensures that a shortest path between any two points will never cross over a wall and will never take a step diagonally. This means that given this cost function, your shortest path code from before will automatically find shortest paths through the maze subject to the restriction that all steps are in cardinal directions and no steps are diagonal.

Make sure that the mazes that your function produces actually have a unique path between each pair of locations. If there are two locations that aren't reachable from one another, or if there are multiple paths between points, your implementation of Kruskal's algorithm might contain an error.

Advice, Tips and Tricks

This assignment is not as hard as it might initially appear to be. Although you will be coding up three classic graph algorithms, the total amount of code you actually need to write is rather small; our initial solution required only about 200 lines of code, including whitespace, **`#include`**s, and comments.

We strongly suggest taking the time to trace through the execution of Dijkstra's algorithm, A* search, and Kruskal's algorithm on small sample graphs before attempting to code them up. These algorithms are not particularly hard to work through by hand (though computing heuristics for A* by hand can be a bit tedious), and if you get to the point where you can trace through the algorithms without having to look at the pseudocode you will have a much easier time implementing them.

Here are some specific suggestions for the different parts of the assignment:

- Our **`Map`**, **`Set`**, and **`TrailblazerPQueue`** types are backed by balanced binary search trees, which means that the key type (in a map), value type (in a set), or element type (in a **`TrailblazerPQueue`**) must be comparable using the **`<`**, **`==`**, and **`>`** operators. All C++ primitive types can be compared this way, as can the **`string`** type, but our container types like **`Vector`**, **`Stack`**, **`Map`**, etc. cannot. Consequently, you cannot create a **`Set<Set<int>>`**, a **`Map<Vector<int>, string>`**, or a **`TrailblazerPQueue<Vector<int>>`**. This may be particularly relevant as you try to implement Kruskal's algorithm, as you may find yourself trying to nest container types to represent clusters. However, our custom **`struct`** types (**`Loc`** and **`Edge`**) have these relational operators defined, so you can make a **`Set<Edge>`**, a **`Map<Loc, double>`**, or a **`TrailblazerPQueue<Loc>`**, for example.
- Make sure you are comfortable with the idea of passing functions into other functions. The first function you will write will take in two parameters as input, so it's worth reviewing this in-depth before proceeding. Section Handout 8 discusses this, as does Chapter 20.2 of the course reader. (Note that the course reader uses a slightly different syntax than what we are using since it discusses pointers to functions, but the idea is exactly the same.)
- Although you're expected to implement the two functions that we've provided, you are encouraged to decompose the function into helper functions. It will make the code a lot easier to read and maintain. If you do introduce helper functions, you should not put their prototypes in the **`Trailblazer.h`** file. Header files should only export functions that other source files would have a reason to call directly.
- When writing Dijkstra's algorithm, take care to keep track of which nodes are already in the priority queue and which nodes have not yet been enqueued. You cannot call **`decreaseKey`** on a node that is not already in the queue.
- Remember that edge costs are **`double`**s, not **`int`**s.

- In Dijkstra's algorithm and A* search, you will need to track candidate distances separately from the priorities in the priority queue, since you will need to be able to determine the candidate distances to a node's neighbors after you dequeue that node from the priority queue. We suggest having an auxiliary structure that holds this information. Because of this, you will probably have to update costs twice in Dijkstra's algorithm – once to update your own local copy, and once to update the priorities in the priority queue. Make sure to keep these two copies in sync with one another – if you don't, you will probably get errors when calling **`decreaseKey`**.
- Don't forget to adjust the parent pointers in Dijkstra's algorithm or A* search after calling **`decreaseKey`**. Otherwise, even though you'll dequeue the nodes in the proper order, your resulting path will be incorrect.
- Dijkstra's algorithm has found the shortest path from the start node to the end only when the end node has been *dequeued* from the priority queue (that is, when it colors the node green). It is possible to *enqueue* the end node into the priority queue but still not have a shortest path to it, since there might be a shorter path to the end node that has not been considered yet.
- Although A* search enqueues nodes into the priority queue with a priority based on both the node candidate distances and their heuristic values, it tracks their candidate distances independently of their heuristic costs. When storing the candidate distance to a node, do not add the heuristic value in. The heuristic is only used when setting the priorities in the priority queue.
- When merging the endpoints of an edge together in Kruskal's algorithm, remember that *every* node in the same cluster as either endpoint should be merged together into one resulting cluster.
- For Dijkstra's algorithm and A* search, each cell implicitly has an edge to each of the eight cells horizontally, vertically, or diagonally adjacent to it. For Kruskal's algorithm, each cell only has edges to the four cells horizontally or vertically adjacent to it.
- It can take a while to generate a large maze using Kruskal's algorithm if you do not use a specialized data structure to keep track of which nodes are in the same cluster as one another (see the *Possible Extensions* section below for more details). It's normal for it to take a few seconds to generate a random maze using Kruskal's algorithm, so don't worry if this happens. If you find yourself waiting twenty seconds or longer, you may need to reevaluate your strategy for storing the clusters. Our reference solution uses an optimized data structure to generate its mazes, so don't worry if your program is a bit slower than ours.
- Since the cost of taking a diagonal step in a maze or crossing a maze wall is infinite, if your pathfinding code produces a path that moves diagonally or crosses a wall, it likely means that there is no path from your starting position to the end position. Since the randomly-generated mazes should have a path between any pair of points in the world, if you encounter this behavior when testing out your mazes it likely means that you have an error in your maze generator.

Possible Extensions

There are *many* possible extensions to this assignment. Here are a few ideas to get you started:

- **Implement a disjoint-set forest**. When implementing Kruskal's algorithm, you need a way to keep track of which nodes in the graph are connected to one another. While it's possible to do this using the standard collections types, there is an extremely simple and much faster way to do this using a *disjoint-set forest*, a specialized data structure that makes it easy to determine if two nodes are connected and to connect pairs of nodes. It is not particularly hard to code up a disjoint-set forest, and doing so can dramatically reduce time required to create a maze.

- **Write better heuristics**. The heuristics we have provided for estimating terrain costs and maze distances are simple admissible heuristics that work reasonably well. Try seeing if you can modify these functions to produce more accurate heuristics. If you do this correctly, you can dramatically cut down on the amount of unnecessary searching required. However, make sure that your heuristics are admissible – that is, they should never overestimate the distance from any node to the destination node.
- **Implement bidirectional search**. A common alternative to using A* search is to use a *bidirectional search algorithm*, in which you search outward from both the start and end nodes simultaneously. As soon as the two searches find a node in common, you can construct a path from the start node to the end node by joining the two paths to that node together. Try coding this algorithm up as a third algorithm choice.
- **Choose a different maze-generation algorithm**. Kruskal's algorithm is only one of many ways to generate a random maze. As you saw in lecture, depth-first search can also be used to generate mazes. Another minimum spanning tree algorithm called *Prim's algorithm* can also be used here to generate random mazes. Try replacing Kruskal's algorithm with a different maze-generation algorithm. Can you generate more complicated mazes?
- **Write a better terrain generator**. Our starter code generates terrain uses the *diamond-square algorithm*, coupled with a Gaussian blur, to generate terrains. Many other algorithms exist that can generate random terrains, such as the 2D Perlin Noise algorithm. Try implementing a different terrain generator and see if it produces better results.

It has been a pleasure teaching CS106B this quarter.
Best of luck on the assignment, and have a great summer!