CS106X Autumn 2015

Cynthia Lee

Assignment 7: Trailblazer

Thanks to Keith Schwarz, Dawson Zhou, Eric Roberts, Julie Zelenski, Nick Parlante, Jerry Cain, and Leonid Shamis (UC Davis) for creating and evolving this assignment and its predecessor, "Pathfinder." BasicGraph class and other modifications by Marty Stepp. Final edits by Cynthia Lee.

Due: Wednesday, November 18th at 5:00pm

Pair programming is permitted on this assignment. See course information sheet and honor code.

The assignment has two main purposes:

1. To give you experience using one implementation of a Graph ADT.
2. To give you experience coding several graph algorithms of theoretical and practical interest.

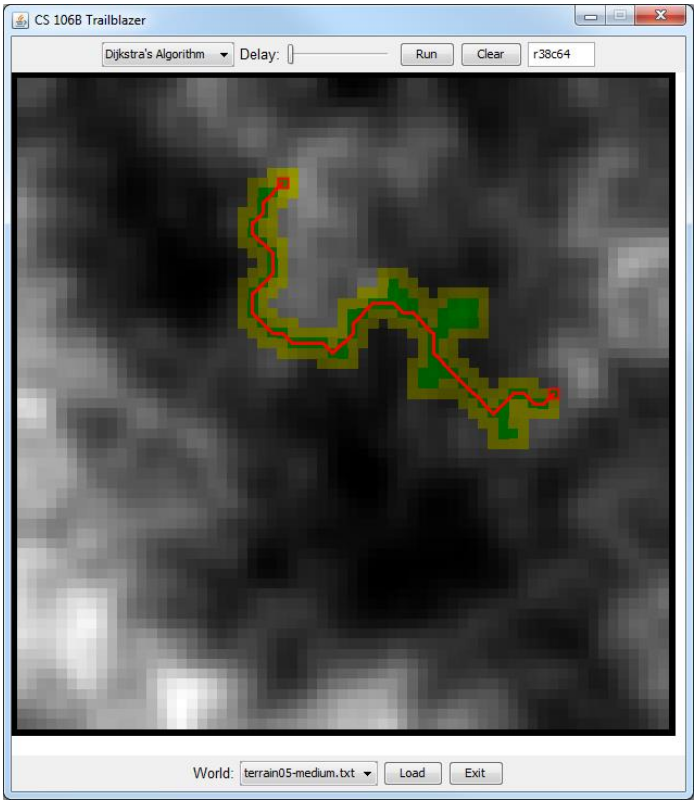
Deliverables (turn in these files):

- `trailblazer.cpp`: code to perform graph path searches
- `map-custom.txt, map-custom.jpg`: a world map of your own creation
- Do not modify the other files in the starter code.

This is a pair assignment. If you work as a pair, comment both members' names on top of every submitted code file. Only one of you should submit the assignment; do not turn in two copies.

PROGRAM OVERVIEW

This program displays various 2-dimensional worlds that represent either maps, mazes, or terrain and allows the user to generate paths in a world from one point to another. When you start up the program, you will see a graphical window containing a 2D maze, where white squares are open and black ones represent walls. The program is also able to display terrain, where bright colors indicate higher elevations and darker colors represent lower elevations. Mountain ranges appear in bright white, while deep canyons are closer to black.



If you click on any two points in the world, the program will find a path from the starting position to the ending position. As it does so, it will color the vertices green, yellow, and gray based on the colors

assigned to them by the algorithm. Once the path is found, the program will highlight it and display information about the path cost in the console. The user can select one of four path-searching algorithms in the top menu:

- **Depth-first search (DFS)**
- **Breadth-first search (BFS)**
- **Dijkstra's algorithm**
- **A* search**
- **(We didn't forget about Kruskal's! That fits in later.)**

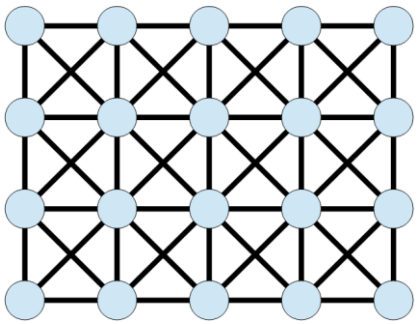
The window also contains several controls. You can load mazes and terrains of different sizes (tiny, small, medium, large, and huge) from the bottom drop-down menu and then clicking the "Load" button. In your trailblazer.cpp file, you must write the following 5 functions for finding paths and creating mazes in a graph:

- `Vector<Vertex*> depthFirstSearch(BasicGraph& graph, Vertex* start, Vertex* end)`
- `Vector<Vertex*> breadthFirstSearch(BasicGraph& graph, Vertex* start, Vertex* end)`
- `Vector<Vertex*> dijkstrasAlgorithm(BasicGraph& graph, Vertex* start, Vertex* end)`
- `Vector<Vertex*> aStar(BasicGraph& graph, Vertex* start, Vertex* end)`
- `Set<Edge*> kruskal(BasicGraph& graph)`

Each of the first four implements a path-searching algorithm taught in class. You should search the given graph for a path from the given start vertex to the given end vertex. If you find such a path, the path you return should be a list of all vertices along that path, with the starting vertex first (index 0 of the vector) and the ending vertex last.

If no path is found, return an empty vector. If the start and end vertices are the same, return a one-element vector containing only that vertex. Though the mazes and terrains in our main app are undirected graphs, your code should not assume this. You may assume that the graph passed has a valid state.

Our provided main client program will allow you to test each algorithm one at a time before moving on to the next. You can add more functions as helpers if you like, particularly to help you implement any recursive algorithms and/or to remove redundancy between some algorithms containing similar code.



The 2D world is represented by a Stanford Library `BasicGraph`, where each vertex represents a specific location on the world. If it is a maze, each location represents one square in the maze's grid-like world. Open squares are connected by edges to any other neighboring open squares that are directly adjacent to them (differ by +/- 1 row or column exactly). Black "wall" squares are not connected as neighbors to any other squares; no

edges touch them. If the world is a terrain rather than a maze, each location represents some elevation between 0 (lowland) and 1 (high mountain peak). Terrain locations are connected to neighbors in all 8 directions including diagonal neighbors, but maze locations are only connected to neighbors directly up, down, left, and right.

Your code can treat maps, mazes, and terrains exactly the same. You should just think of each kind of world as a graph with vertices and edges that connect neighboring vertices. In the case of mazes, vertices happen to represent 2D locations and neighbors happen to be directly up, down, left, right, etc., but your code does not utilize or rely on that information. Your path-searching algorithms will work on any kind of graph that might be passed to them.

PROVIDED CODE

We provide you with a lot of starter code for this assignment. Here is a quick breakdown of what each file contains, though you do not need to examine or know about each file or its contents in order to complete the assignment.

Don't modify any code except trailblazer.cpp.

- **`trailblazer.h/.cpp`**: We provide a skeleton version of these files where you will write your path-searching code for the assignment.
- **`Color.h/.cpp`**: Constants representing colors of vertices.
- **`TrailblazerGUI.h/.cpp`**: The app's graphical user interface.
- **`trailblazermain/.cpp`**: The main function that initializes the GUI and launches the application.
- **`World(Abstract,Grid,Map,Maze,Terrain).h/.cpp`**: Hierarchy of types of world graphs.

Each vertex in the graph is represented by an instance of the Vertex structure, which has the following members:

Vertex member	Description
`string` **`name`**	vertex's name, such as `"r34c25"` or `"vertex17"`
`Set<Edge*>` **`edges`**	edges outbound from this vertex
`double` **`cost`**	cost to reach this vertex (initially `0`)
`bool` **`visited`**	whether this vertex has been visited yet (initially `false`)
`Vertex*` **`previous`**	pointer to a vertex that comes before this one; initially `NULL`
`void` **`setColor`**`(Color c)`	sets this vertex to be drawn in the given color in the GUI, one of `WHITE`, `GRAY`, `YELLOW`, or `GREEN`
`Color` **`getColor`**`()`	returns color you set previously using `setColor`; initially `UNCOLORED`

`void` **`resetData`**`()`	sets `cost`, `visited`, `previous`, and color back to their initial values
`string` **`toString`**`()`	returns a printable string representation of the vertex for debugging

The cost, visited, and previous member variables are for you to use in path-search algorithms. Several algorithms depend on being able to "mark" a vertex as visited, or to associate a cost with a vertex, or to keep pointers from one vertex to another to trace a path. Use these members in each vertex to keep track of such information. Call `resetData` on a vertex to wipe this data (returns color to "UNCOLORED"), or on the `BasicGraph` as a whole to wipe all such data for all vertices.

Each edge in the graph is represented by an instance of the Edge structure, which has the following members:

Edge member	**Description**
`Vertex*` **`start`**	the starting vertex of this edge
`Vertex*` **`finish`**	the ending vertex of this edge (i.e., `finish` is a neighbor of `start`)
`double` **`cost`**	cost to traverse this edge
`string` **`toString`**`()`	returns a printable string representation of the vertex for debugging

The vertices and edges are contained inside a BasicGraph object passed to each of your algorithm functions. **See the Stanford C++ library documentation for descriptions of the members of the `BasicGraph` class.** In addition to those members, `BasicGraph` includes all of the public members from its parent class Graph.

`BasicGraph` has a useful public member named `resetData`. You must call `resetData` on the graph at the *start* of any path-searching algorithm that wants to store data in the vertices, to make sure that no stale data is left in the vertices from some prior call. Call it at the start of your algorithm and not at the end, to ensure that any old state is cleaned out before your algorithm begins. If you don't call it, your algorithms may fail for subsequent calls.

COLORING VERTICES

In addition to searching for a path in each algorithm, we also want you to add some code to give colors to various vertices at various times. This coloring information is used by the GUI to show the progress of your algorithm and to provide the appearance of animation. To give a color to a

vertex, call the color member function on that vertex's Vertex object, passing it a global color constant such as GRAY, YELLOW, or GREEN. For example:

```
Vertex* myVertex = graph.getVertex("foo");
myVertex->setColor(GREEN); // set the vertex's color to green
```

Here is a listing of colors available and when you should use them (see also lecture notes from the Graph lectures):

- **default** = uncolored**:** This is the color after a call to `resetData`. In class, we talked about all the nodes starting with gray color (and then changing to yellow and green). For this assignment, that beginning/unexplored color should be this default value UNCOLORED, not the actual color GRAY. Do not loop over all the nodes and color them GRAY to start; just call `resetData`. We will use the actual color GRAY for a different purpose (see below).

- **enqueued = yellow:** Whenever you enqueue a vertex to be visited for the first time, such as in BFS and Dijkstra's algorithm when you add a vertex to a data structure for later processing, color it yellow (YELLOW).

- **visited = green:** Whenever your algorithm directly visits and examines a particular vertex, such as when it is dequeued from the processing queue in BFS or Dijkstra's algorithm, or when it is the starting vertex of a recursive call in DFS, color it green (GREEN).

- **eliminated = gray:** When the DFS algorithm has finished exploring a vertex and did not find a path from that vertex, and is therefore "giving up" on that vertex as a candidate, color it gray (GRAY). The only algorithm that explicitly "backtracks" like this is depth-first search (DFS). You don't need to set any vertices to gray in any other path-searching algorithms besides DFS.

The provided GUI has an animation **slider** that you can drag to set a delay between coloring calls. If the slider is not all the way to its left edge, each call to setColor on a vertex will pause the GUI briefly, causing the appearance of **animation** so that you can watch your algorithms run.

GRAPH ALGORITHMS

Depth-first search implementation notes: You can implement it recursively as shown in lecture, or non-recursively. The choice is up to you. A recursive solution can sometimes run slowly or crash on extremely large worlds; this is okay. You do not need to modify your DFS implementation to avoid crashes due to excessive call stack size.

Breadth-first search implementation notes: Your code will need to regenerate the path that it finds, so look at the version of the algorithm pseudo-code from lecture that keeps track of paths along the way. One interesting note is that BFS and Dijkstra's algorithm behave exactly the same when run on a maze, but differently on a terrain. (Why?)

Dijkstra's algorithm implementation notes: 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. The priority queue should store vertices to visit, and once you find the destination, you should reconstruct the shortest path back. See the lecture slides for more details.

Our pseudocode for Dijkstra's algorithm occasionally refers to "**infinity**" as an initial value when talking about the cost of reaching a vertex. If you want to refer to infinity in your code, you can use the double constant POSITIVE_INFINITY that is visible to your code.

Both Dijkstra's algorithm and A* involve a **priority queue** of vertices to process, and furthermore, they each depend on the ability to **alter a given vertex's priority** in the queue as the algorithm progresses. To do this, call the changePriority member function on the priority queue and pass it the new priority to use. It is important to use this function here because otherwise there is no way to access an arbitrary element from the priority queue to find the one whose priority you want to change. You would have to remove vertices repeatedly until you found the one you wanted, which would be very expensive and wasteful. The new priority you pass must be at least as urgent as the old priority for that vertex (because the function bubbles a value upward in the priority queue's internal heap structure).

Note that the notion of a given vertex's current priority might be stored in two places in your code: in the cost field of the Vertex structure itself, and in the priority queue's ordering. You'll have to keep these two in sync yourself; if you update just the vertex, the priority queue won't know about it if you don't call `changePriority`, and vice versa. If the two values get out of sync, this can lead to bugs in your program.

A* implementation notes: As discussed in class, the A* search algorithm is essentially a variation of Dijkstra's algorithm that uses heuristics to fine-tune the order of elements in its priority queue to explore more likely desirable elements first. So when you are implementing A*, you need a **heuristic function** to incorporate into the algorithm. We supply you with a global function called heuristicFunction that accepts a pointer to two vertices *v1* and *v2* and returns a heuristic value from *v1* to *v2* as a double. You can assume that this is an *admissible heuristic*, meaning that it never overestimates the distance to the destination (which is important for A*). For example:

```
Vertex* v1 = graph.getVertex("foo");
Vertex* v2 = graph.getVertex("bar");
double h = heuristicFunction(v1, v2); // get an A* heuristic
```

You can compare the behavior of Dijkstra's algorithm and A* (or any pair of algorithms). First try performing a search between two points using Dijkstra's algorithm, then select A* and press the "Run" button at the top of the GUI window. This will repeat the same search using the currently selected algorithm. Run a search using Dijkstra's algorithm, switch the algorithm choice to "A*," then run that search to see how much more efficient A* is.

Your A* search algorithm should always return a path with the **same length and 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. For mazes, all three of BFS, Dijkstra's algorithm, and A* should return paths with the same length and cost.

The A* algorithm performs no better than Dijkstra's algorithm when run on **maps** because they have no heuristic.

Several **expected output files** have been posted to the class web site. If you have implemented each path-searching algorithm correctly, for DFS you should get any valid path from the start to the end; for BFS you should get the **same path lengths** as shown in the expected outputs posted on the class web site. For Dijkstra's and A* you should get the **same path costs** as shown in the expected outputs. But you do *not* need to exactly match our path itself, nor its "locations visited", so long as your path is a correct one. For Kruskal's algorithm (described next), your code must find a valid minimum spanning tree on the given graph. If there are several of equal total weight, any will suffice.

As mentioned previously, your code should **not** assume that the graph is undirected; we will test your code with directed graphs as well as undirected ones.

RANDOM MAZE GENERATION (KRUSKAL'S MST)

Your final task in this assignment is to implement Kruskal's algorithm for finding a minimum spanning tree. Your function should accept a graph as a parameter, and you should return a set of pointers to edges in the graph such that those edges would connect the graph's vertices into a minimum spanning tree. (Don't actually add/remove edges from the graph object passed in by calling `addEdge`, `removeEdge`, etc. on it. Just return the set of edges separately.) Specifically, your task is to write a function with the following signature:

```
Set<Edge*> kruskal(BasicGraph& graph)
```

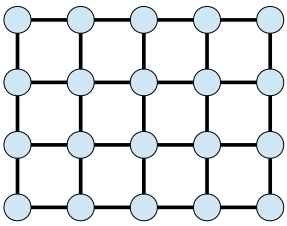
The pseudocode for Kruskal's is as follows:

kruskal(graph):

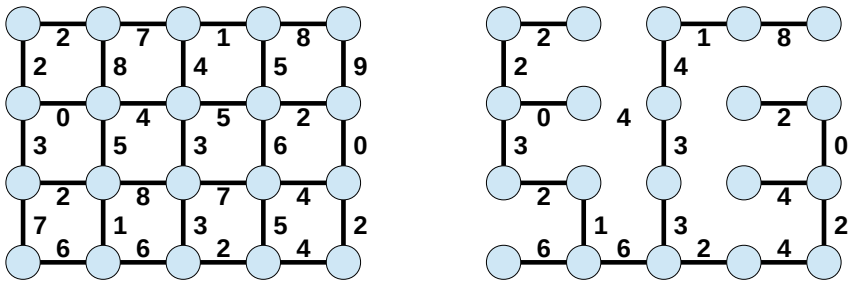
1. Place each vertex into its own "cluster" (group of reachable vertices).
2. Put all edges into a priority queue, using weights as priorities.
3. While there are two or more separate clusters remaining:
    - Dequeue an edge e from the priority queue.
    - If the start and finish vertices of e are not in the same cluster:
        - Merge the clusters containing the start and finish vertices of e.
        - Add e to your spanning tree.
    - Else:
        - Do not add e to your spanning tree.
4. Once the while loop terminates, your spanning tree is complete.

The specific application we'll Kruskal's algorithm to solve is the problem of generating new random mazes. As discussed earlier in this handout, you can think of a maze as a graph, where

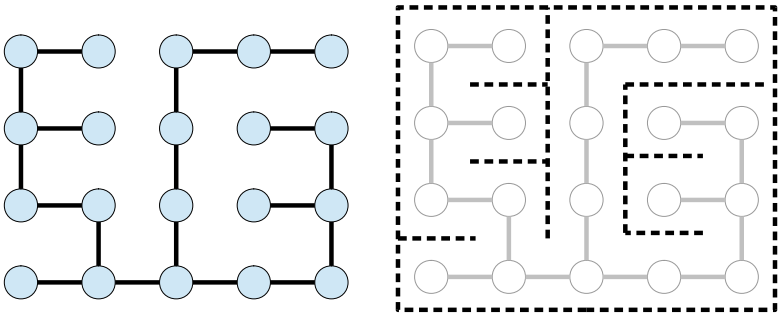
the vertices are connected as follows. The following figure would be a fully connected maze with no walls:



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 vertices. For example, assigning the edges in the above graph weights as follows and running Kruskal's algorithm would produce the following result:



In the above tree, lines represent edges between connected neighbors, which are passable. Neighbors that are not connected by an edge can be thought of as having an impassable "wall" between them. You can turn the above tree into a maze by drawing lines in all of the empty space, as shown here:



Whenever you click the GUI's "Load" button with one of the "Random" options selected, our starter code will generate a maze of the given size with every vertex connected to all of its neighbors, as in the "fully connected" maze figure above. **Our code will randomly assign weights to each edge for you**; you shouldn't change the weights we pass in. Then we will pass the graph to your algorithm for you to find the minimum spanning tree. Once you return your set of edges, our starter code will process your set and fill in "walls" between any neighbors that are not directly connected by an edge. The resulting maze will show in the GUI. Once you've made a maze, you can run your path-finding algorithms to locate paths between points in the maze. Implementing this function raises questions such as:

- 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?

Think about these issues yourself and come up with a clean and efficient way of solving the problem. Our own sample solution is able to generate "Large" sized mazes in a few seconds' time at most, and you should strive for similar efficiency. If your maze generation algorithm takes, say, close to a minute or more to finish, optimize it.

CREATIVE ASPECT: YOUR OWN MAP

Turn in files `map-custom.txt` and `map-custom.jpg` representing a map graph of your own. Put the files into the `res/` folder of your project. The text file contains information about the graph's vertices and edges. The graph can be whatever you want, so long as it is not essentially the same as any of the provided graphs. Your image can be any (non-offensive) JPEG image you like; we encourage you to use a search engine like Google Image Search to find an interesting background. The text file's format should exactly match the following example, from map-small.txt. For full credit, your file should load successfully into the program without causing an error and be searchable by the user.

```
IMAGE
map-usa.jpg                          ← image file name
654                                  ← image width, in pixels
399                                  ← image height, in pixels

VERTEXES
Washington, D.C.;536;176             ← vertex format is: name;x;y
Minneapolis;349;100
San Francisco;26;170

EDGES
Minneapolis;San Francisco;1777       ← edge format is: vertex1;vertex2;weight
Minneapolis;Washington, D.C.;1600        (or, for a directed one-way edge:
San Francisco;Washington, D.C.;2200       vertex1;vertex2;weight;true )
```

DEVELOPMENT STRATEGY AND TIPS

- Trace through the algorithms by hand on small sample graphs before coding them.
- Work step-by-step. Complete each algorithm before starting the next one. You can test each individually even if others are incomplete. We suggest doing DFS/BFS, then Dijkstra's, then A*, and finally Kruskal's.
- Start out with tiny worlds first. It is much easier to trace your algorithm and/or print every step of its execution if the world is small. Once your output matches perfectly on tiny files, go to small, medium, large.
- In Dijkstra's algorithm, you cannot call changePriority on a vertex that is not already in the queue. You also cannot call changePriority with a priority less urgent (greater) than the existing priority in the queue.
- Remember that edge costs are doubles, not ints.
- In A* search, when storing the candidate distance to a vertex, *do not* add the heuristic value in. The heuristic is only used when setting the priorities in the priority queue, not when setting the cost fields of vertices.

- Don't forget to keep previous pointers up-to-date in Dijkstra's algorithm or A* search. Otherwise, though you'll dequeue the vertices in the proper order, your resulting path might end up incorrect.
- In Dijkstra's algorithm, don't stop your algorithm early when you *enqueue* the ending vertex; stop it when you *dequeue* the ending vertex (that is, when you color the vertex green).
- When merging clusters together in Kruskal's algorithm, remember that *every* vertex in the same cluster as either endpoint (not just the endpoints themselves) should be merged together into one resulting cluster.

STYLE AND GRADING

Items from prior assignment(s) specs also apply here; refer to those documents as needed. Follow instructions about procedural decomposition, variables, types, parameters, value vs. reference, and commenting. Place inline comments as needed on complex code inside function bodies. Don't forget to cite sources you used in your comments. Refer to the course Style Guide for more discussion of coding style.

Part of your grade will come from appropriately implementing the graph algorithms as described in class. Redundancy is another major grading focus; avoid repeated logic as much as possible.

Your code should have no memory leaks. You should not need to free anything because your code should not need to use the new operator. If for some reason you do use new, free it as soon as you are done using the object.

Please remember to follow the Honor Code on this assignment. Submit your own (pair's) work; do not look at others' solutions. Cite sources. Do not give out your solution; do not place a solution on a public web site or forum.

SUGGESTED EXTENSIONS

Here are some suggested extensions. Those I especially recommend for X students are marked with **.

- **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 vertices simultaneously. As soon as the two searches find a vertex in common, you can construct a path from the start vertex to the end vertex by joining the two paths to that vertex together. Try coding this algorithm up as a fifth algorithm choice.
- ****Disjoint-set forest:** When implementing Kruskal's algorithm, you need a way to keep track of which vertices 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 vertices are connected and to connect pairs of vertices. It is not particularly hard to code up a disjoint-set forest, and doing so can dramatically reduce time required to create a maze.

- **New world type:** The existing code has several classes that extend a superclass World to represent maps, mazes, and terrains. Add your own new subclass for a type of world that we didn't include.
- **Write better heuristics:** The heuristics we have provided for estimating terrain costs and map / 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 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 starting vertex to any destination vertex.
- ****Choose a different maze-generation algorithm:** Kruskal's algorithm is only one of many ways to generate a random maze. Another minimum spanning tree algorithm called **Prim's algorithm** can also be used here to generate random mazes. Try adding Prim's algorithm in addition to Kruskal's algorithm for maze generation. Can you generate more complicated mazes and maps?
- **Write a better terrain generator:** Our starter code generates terrain uses the diamond-square algorithm, coupled with a Gaussian blur. 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.
- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extras that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your extras cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit your program twice: a first time without any extra features added (or with all necessary extras disabled or commented out), and a second time with the extras enabled. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.