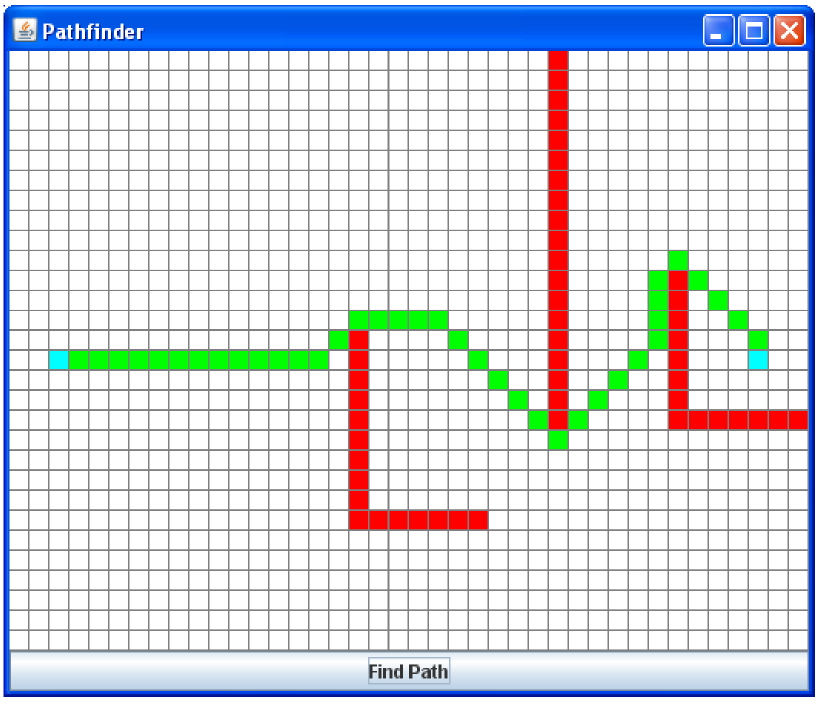
CS11 Java Lab 2: I'm A Star!

If you have ever played any kind of map-based game on the computer, you have probably encountered computer-controlled units that know how to get from point A to point B all by themselves. This is actually a common problem in both games and other kinds of software, how to generate a path from a starting location to a desired destination that successfully navigates obstacles.

One very widely used algorithm for this kind of problem is called A* (pronounced "A-star"), and it is a very effective algorithm for pathfinding in a computer program. The algorithm is conceptually pretty simple. Starting with the initial location, it incrementally builds a path from the source to the destination, always using the "best path so far" to take the next step. This ensures that the final path will also be the optimal one. (If you want to learn much more about the A* pathfinding algorithm, you can start with the Wikipedia article on A*, and follow the references the article gives.)

Fortunately, you don't have to implement the A* algorithm; this has already been done for you. In fact, there is even a nice little user interface for you to experiment with the A* algorithm:



You can click on the various squares to turn them into barriers (red) or passable cells (white). The blue cells mark the start and end of the path. Clicking on the "Find Path" button will compute a path using A*, and then display it in green. Or, if there is no path from start to end, the program will simply not show any path.

The A* algorithm has a lot of information to keep track of, and the Java collection classes are perfect for this kind of task. There are two main kinds of information that this A* implementation must manage:

- **Locations** are simply the sets of coordinates of specific cells on the two-dimensional map. The A* algorithm must be able to refer to specific locations in the map.
- **Waypoints** are individual steps in the path that the A* algorithm generates. For example, the above path in green is simply a sequence of waypoints through the map. Each waypoint has several pieces of information associated with it:
    - The location of the cell that the waypoint is for.
    - A reference to the previous waypoint in the path. The final path is simply a sequence of waypoints from the destination back to the starting point.
    - The actual cost of moving from the starting location to this waypoint's location, following the specific path ending with the waypoint.
    - A heuristic estimate (a *guess*, in other words) of the remaining cost of traveling from this location to the final destination.

As the A* algorithm builds its path, it must maintain two primary collections of waypoints:

- The first collection stores "open waypoints," or waypoints that still need to be considered by the A* algorithm.
- The second collection stores "closed waypoints," or waypoints that have already been considered by the A* algorithm and don't need to be examined again.

Each iteration of the A* algorithm is pretty simple: find the least costly waypoint from the set of open waypoints, take a step in every direction from that waypoint to generate new open waypoints, and then move the waypoint from the open set to the closed set. This is repeated until the current waypoint happens to be at the destination! If, during this process, the algorithm runs out of open waypoints, there is no path from the starting point to the destination.

This processing depends primarily on the locations of waypoints, so it is very useful to store waypoints as mappings from locations to their corresponding waypoints. Thus, you will use the `java.util.HashMap` container for each of these collections, with `Location` objects as the keys, and `Waypoint` objects as the values.

Before You Begin

Before you begin, you should download the source files for this lab assignment:

- `Map2D.java` - represents the map that the A* algorithm navigates, including whether cells are passable or not
- `Location.java` - this class represents the coordinates of a specific cell on the map
- `Waypoint.java` - represents individual waypoints in the generated path
- `AStarPathfinder.java` - this class implements the A* pathfinding algorithm as a static method
- `AStarState.java` - this class stores the collections of open waypoints and closed waypoints, and provides basic operations necessary for the functioning of the A* algorithm
- `AStarApp.java` - a simple Swing application that provides an editable view of a 2D map, and initiates pathfinding when requested
- `JMapCell.java` - this is a custom Swing component that is used to display the state of cells in the map

Note that the application will compile successfully as-is, but the path-finding functionality won't work until you complete the assignment.

The only classes you should have to edit will be **`Location`** and **`AStarState`**. Everything else is framework code that allows you to edit the map and display the path that the algorithm generates. (If you find yourself editing any of the other source files to get the lab to work then stop and ask for help!)

Locations

The first thing that needs to be done is that the `Location` class must be prepared for use with the Java collection classes. Since you will be using hashing containers for this assignment, this involves:

- Providing an implementation of the `equals()` method.
- Providing an implementation of the `hashCode()` method.

Add an implementation of each of these methods to the `Location` class, following the patterns outlined in class. Once this is completed, you can use the `Location` class as a key type in hashing containers like `HashSet` and `HashMap`.

A* State

Once the `Location` class is ready to use as a key, you can finish the implementation of the `AStarState` class. This is the class that maintains the sets of open and closed waypoints, so it really provides the core functionality for the A* implementation.

As mentioned previously, the A* state consists of two collections of waypoints, one of open waypoints, and the other of closed waypoints. To facilitate the algorithm, waypoints will be stored in a hash-map, with the waypoint locations being the keys and the waypoints themselves being the values. Thus, you will have a type like this:

```
    HashMap<Location, Waypoint>
```

(An obvious implication of this is that each map location can have only one waypoint associated with it. This is exactly what we want.)

Add two (non-static) fields to the `AStarState` class with this type, one for "open waypoints" and the other for "closed waypoints." Also, make sure to initialize each of these fields to refer to a new, empty collection.

Once you have the fields set up and initialized properly, you will need to implement the following methods on the `AStarState` class:

1. **`public int numOpenWaypoints()`**

   This method should simply return the number of waypoints in the collection of open waypoints. (Yep it'll be a one-liner...)

2. **`public Waypoint getMinOpenWaypoint()`**

   This function should scan through all waypoints in the collection of open waypoints, and return a reference to the waypoint with the *smallest* total cost. If there are no waypoints in the "open" collection, return `null`.

   Don't remove the waypoint from the collection when you return it; just return a reference to the waypoint with the smallest total cost.

3. **`public boolean addOpenWaypoint(Waypoint newWP)`**

   This is the most complicated method in the A* state class, but honestly it is still pretty simple. What makes it more complicated than the rest is that it should only add the specified waypoint if an existing waypoint at the location is *worse* than the new one. Here is what the method must do:

   - If there is currently no waypoint for this location in the "open waypoints" collection then just add the new waypoint.

- If there is *already* a waypoint for this location in the "open waypoints" collection, then only add the new waypoint if the "previous cost" for the new waypoint is less than the "previous cost" for the current waypoint. (Make sure to use the *previous* cost and not the *total* cost.) In other words, if the new waypoint represents a shorter path to that location than the current waypoint does, replace the current waypoint with the new one.

You can see that you will have to retrieve the existing waypoint from the "open" collection, if there is one, and act accordingly. Fortunately, it is very simple to replace a previous waypoint with a new one; just use the `HashMap.put()` method like usual, and it will replace the old key-value mapping with the new one.

Finally, have this method return `true` if the new waypoint was added to the collection of open waypoints, or `false` if the new waypoint wasn't actually added.

4. **`public boolean isLocationClosed(Location loc)`**

   This function should return `true` if the specified location appears in the collection of closed waypoints, or false otherwise. Since the closed waypoints are stored in a hash-map with locations as the key values, this should be quite simple to implement.

5. **`public void closeWaypoint(Location loc)`**

   This function takes a waypoint and moves it from the collection of open waypoints to the collection of closed waypoints. Since waypoints are keyed by their location, the method takes the location of the waypoint.

   The process should be simple:

   - Remove the waypoint corresponding to the specified location from the collection of open waypoints.
   - Add the waypoint you just removed, to the collection of closed waypoints. Of course, the key should be the location of the waypoint.

Compiling and Testing

Once you get the above functionality implemented, give the pathfinding program a run to see if it works properly. If you have implemented everything correctly, you shouldn't have any problem creating obstacles and then finding paths around them.

You can compile and run the program the same way as always:

```
javac *.java
java AStarApp
```

After you have convinced yourself that everything works correctly, go ahead and submit your work via csman!
