UNIVERSITY LOGO DEPARTMENT LOGO  
TERM INFORMATION COURSE NAME

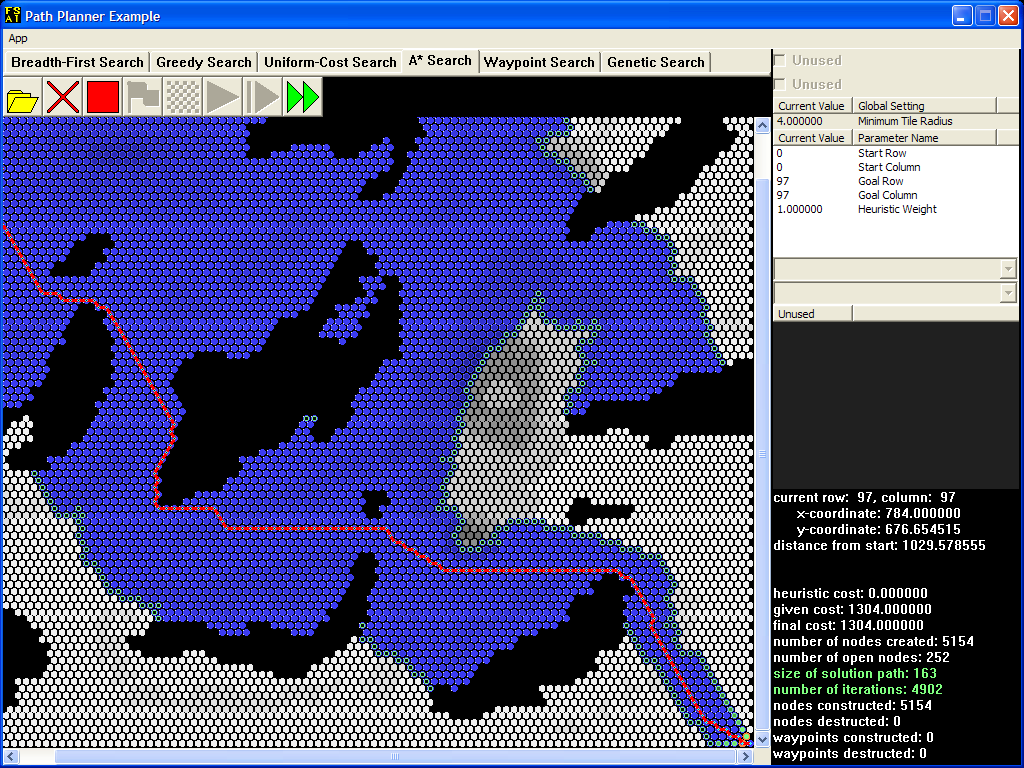
# Path Search Project

# Overview

The path search application is designed to allow for convenient creation, demonstration, and analysis of different search algorithms. As a user of the finished product, you can:

* Choose a search algorithm to study by selecting the appropriate tab at the top row.
* Configure the algorithm by editing or manipulating the various controls near the right side.
* Watch the algorithm in action by using the playback buttons near the top-left corner.

As a developer, you can customize the finished product by writing your own algorithm so that it conforms to a certain programming interface, then adjusting the user interface so that it recognizes your algorithm. By conforming your algorithm to a programming interface that is separate from the user interface, you can port it more easily between code bases (different games, for example.)



You will be implementing a single **search algorithm** (an algorithm that searches for the solution to a problem) from the ground up according to the API specification. Your goal will be to match or exceed the performance and efficiency of the reference search. The user interface is fully functional; however, you will have to create the search class itself. Your algorithm will construct a complete path solution.

# Table of Contents

Contents

[Overview 1](#_Toc453329131)

[Table of Contents 2](#_Toc453329132)

[Objectives & Outcomes 3](#_Toc453329133)

[Instructions 3](#_Toc453329134)

[General User Interface 3](#_Toc453329135)

[Tile Map Display 3](#_Toc453329136)

[Button Controls 4](#_Toc453329137)

[Checkboxes 4](#_Toc453329138)

[Global Settings 4](#_Toc453329139)

[Parameter Settings 5](#_Toc453329140)

[Information Display 5](#_Toc453329141)

[Environment and Tools 5](#_Toc453329142)

[Development Interface 5](#_Toc453329143)

[State Space Structure 6](#_Toc453329144)

[Class API 10](#_Toc453329145)

[PathSearch API 11](#_Toc453329146)

[Test Data 12](#_Toc453329147)

[Frequently Asked Questions 12](#_Toc453329148)

[How Do I... 12](#_Toc453329149)

[Common Errors 13](#_Toc453329150)

# 

# Objectives & Outcomes

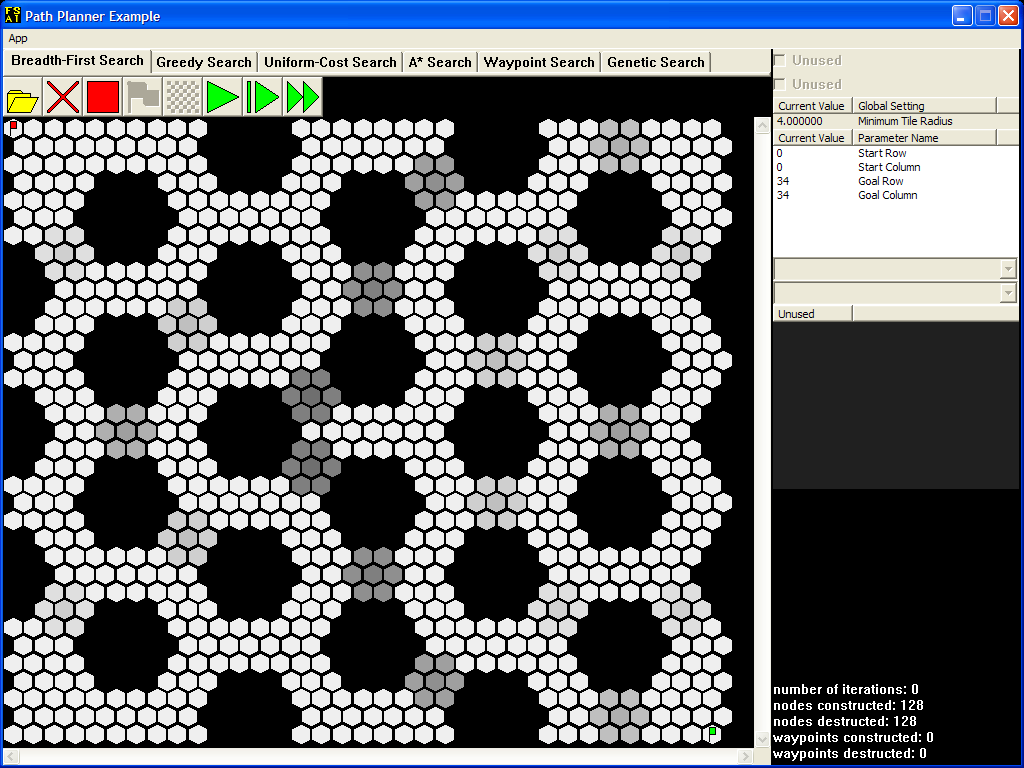
Upon completion of this activity, students should be able to...

* When provided with an API and system, design a module for that system
* Analyze a problem to be solved in order to design custom data structures for the purpose of problem solving AI behaviors (such as path planning).
* Implement a problem solving AI algorithm / behavior (such as path planning).

# Instructions

## General User Interface

The application (the example, not yours) should look like this on start-up.



### Tile Map Display

The search algorithm for this project will run on a hexagon tile map such as the one displayed above. In addition to the reference search implementation, several other common searches are included as examples. For some example search algorithms, lighter tiles incur less cost to traverse than darker tiles. No search algorithm can pass through black tiles, which have a weight of zero. Whichever search algorithm is currently selecte, the red flag sits on top of the starting tile, while the green flag points to the goal tile.

### Checkboxes

Some search algorithms store plenty of information. To keep the clutter out of the tile map display, activate each checkbox near the top-right corner only when you need to see the relevant data. A disabled checkbox means that the current algorithm does not store the data in question.

### Global Settings

These settings affect all search algorithms. For this version of the application, these settings are disabled.

### Button Controls

The following table describes the button controls near the top-left corner.

|  |  |  |  |
| --- | --- | --- | --- |
| **Icon** | **Name** | **Shortcut** | **Description** |
|  | Open Tile Map | Ctrl+O | Brings up a file dialog. Once you select a file, all memory allocated to the old tile map is freed, a new tile map is loaded from that file, all search algorithms are reset, and all node counters are zeroed out. |
|  | Reset All | Delete | Resets all search algorithms and zeroes all node counters. |
|  | Reset Search | Backspace | Resets the current search algorithm and updates the node counters. If all search algorithms have been reset this way, then each pair of constructed/destructed counts should match. |
|  | Run / Pause | Spacebar | When enabled and currently paused, runs the search algorithm until it reaches the goal tile, animating its progress in the meantime. If pressed while the search algorithm is still running, it becomes paused. As soon as the search algorithm reaches the goal tile or can make no further progress, this button will be disabled. |
|  | Step | + | When enabled, runs the search algorithm for exactly one iteration and then pauses it. As soon as the search algorithm reaches the goal tile or can make no further progress, this button will be disabled. |
|  | Time Run | Tab | Resets the search algorithm, then runs the search algorithm until it reaches the goal tile, but does not animate its progress. The number of iterations shown is replaced by the elapsed time from initialization to finish. |

### Parameter Settings

These settings affect the current search algorithm only when it is initialized. Initialization occurs when you start running the search algorithm from its reset state. Most search algorithms take in the start and goal row-column coordinates, so that you can change their positions as necessary. Feel free to play with these numbers so that you can familiarize yourself with the row-column coordinate system that these algorithms use. Note that any changes you make will not affect a search that is currently in progress: it will still look for the goal tile at the old position.

### Information Display

The node counters are shown at the bottom-right corner at all times. As you run a search algorithm, you will also see information regarding the current solution path and the sizes of all containers used by the algorithm so far.

## Environment and Tools

Students will construct a PathSearch class which will perform a search of the terrain and construct a solution vector containing the series of tiles, in order, that make up the path within the following environment.

### Development Interface

The development interface provided includes several elements and classes to help students test and debug their algorithms.

#### Console

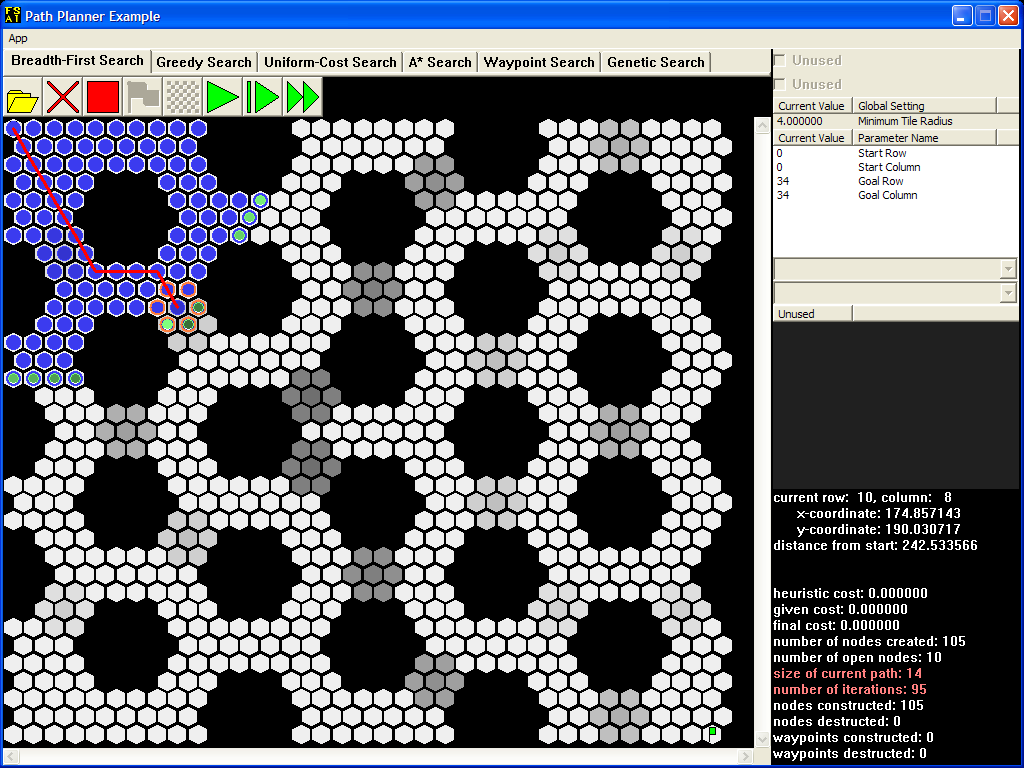
A console window, which can be written to by students using standard streams such as cin and cout, is automatically spawned for use by students. This can be a useful tool for textual output.

#### Drawing System

Several debugging functions and methods allow students to draw to the screen, including filling tiles, drawing markers, setting outlines, and drawing lines. The color space for these drawing functions is 32-bit LRGB space (luminosity, red, green, and blue.) Each color is a 32-bit number with the first (high order) byte being luminosity, the second being red, the third being green, and the fourth being blue. Here are some example colors:

|  |  |
| --- | --- |
|  | L = 255, R = 0, G = 255, B = 0 or 0xFF00FF00 |
|  | L = 127, R = 0, G = 255, B = 0 or 0x7F00FF00 |
|  | L = 0, R = 0, G = 255, B = 0 or 0x0000FF00 |

### State Space Structure



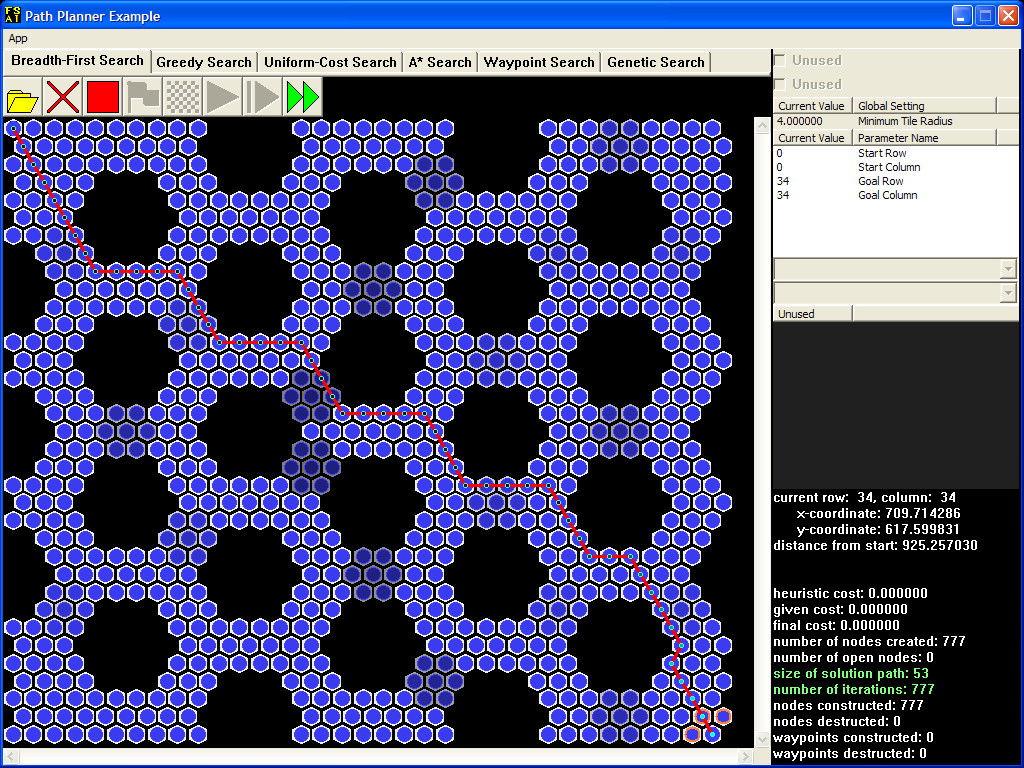
All search algorithms must keep track of different paths that may lead to the goal. By doing this, a search algorithm can extend these paths and so work its way through the tile map.

A path is found using a graph composed of **search nodes**, each of which stores information regarding the corresponding tile and its successors, and **planner nodes**, which store information about paths and– if applicable – cost calculations and estimates. In the examples, the blue circles represent all planner nodes that the search algorithm has **visited** so far. Nodes with green inner circles are **open nodes** that the algorithm can use to extend existing paths; the lighter the green interior, the closer the node is to becoming the **current node**. Note that the current node is ***not*** an open node, so it doesn't have a green inner circle; instead, the application draws a red line from this node through each successive **parent node** until the starting tile is reached.

While the application is paused, depending on the algorithm, either orange circles will appear around the neighbors of the current node as shown above, or orange lines will extend toward those neighbors. The algorithm evaluates only these **successor nodes** along with the current node at each iteration.

After setting the current node at the start of each iteration, the search algorithm checks whether or not this node is at the goal tile. As long as this is not the case, the size of the current path and the number of iterations so far will appear as red text in the [information display](1.htm##mozTocId670993).

When the search algorithm finally reaches the goal, the red line will be decorated with large blue squares that indicate the solution path.



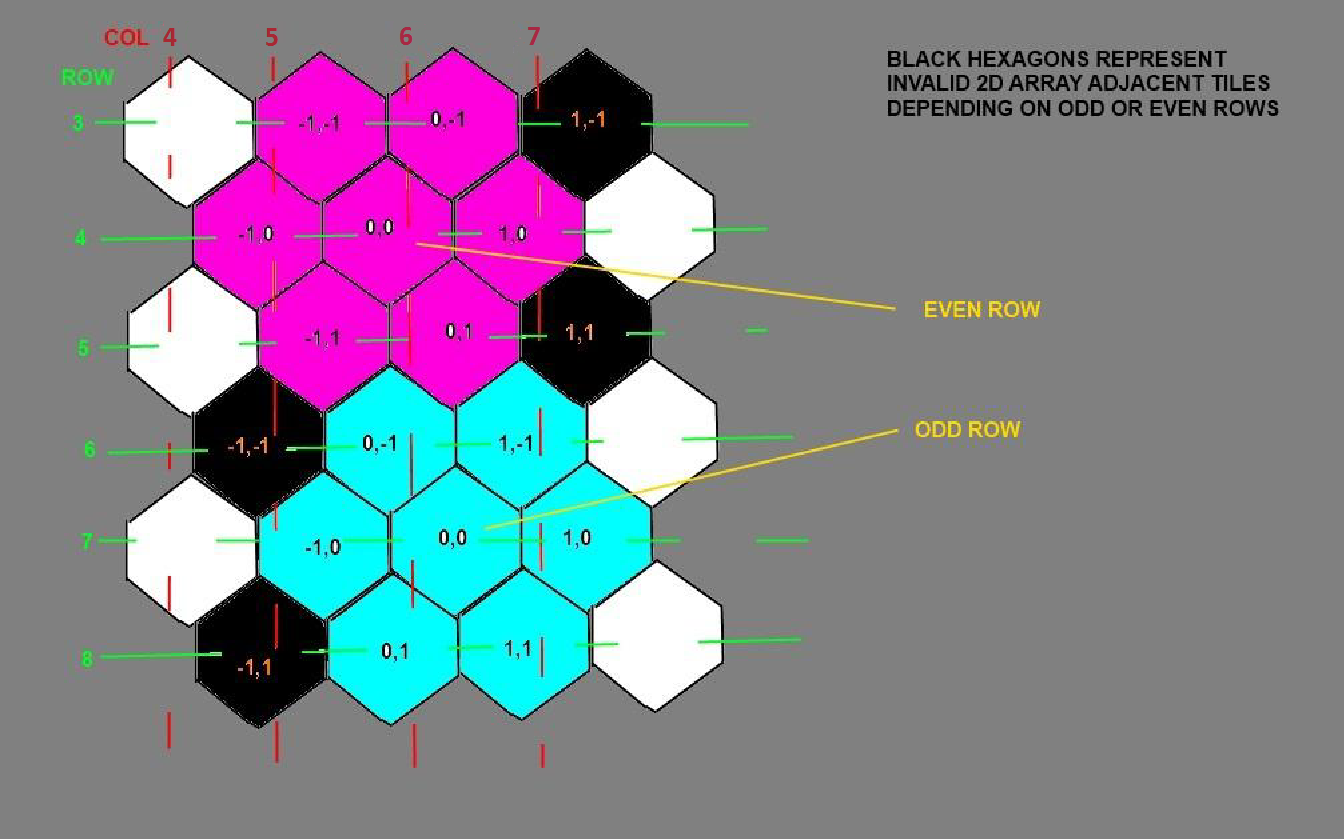
Invalid Endpoints

It's possible for the user to enter row-column coordinates so that either the red flag or the green one will stand on top of an impassable tile. The search algorithm should do nothing in this case.

Hexagon Tiles

The garden-variety square grid assigns either four or eight neighbors to each square tile, depending on whether or not squares that touch at the corners count as neighbors. With a square grid, it's easy to tell where the neighbors are, given the location of a square. When the grid consists of hexagons, finding a tile's neighbors becomes less straightforward.

#### The Row-Column Coordinate System



To help you visualize the tile map diagrams that this manual employs, one such diagram will be superimposed over a blow-up of a small hexagon tile map.

The dashed RED lines separate the tiles into logical columns. The GREEN for rows respectively. Depending on a row being odd or even there is a tile labeled (0,0) surrounded by its possible offsets in MAGENTA or LIGHT BLUE. BLACK tiles indicate offsets that are invalid depending on the odd or even row location of (0,0), do not associate the Black tiles in this diagram with the black impassible tiles that the application draws.

More formally, the following diagram—the same one as before, but no longer superimposed over a screenshot—illustrates how tile maps work in this project.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| column | 0 | 1 | 2 | 3 | 4 | 5 |
| row |
| 0 | X | X | X | X | X | X |
| 1 | X | X | X | X | X | X |
| 2 | X | X | X | X | X | X |
| 3 | X | X | X | X | X | X |
| 4 | X | 0 | X | X | X | X |
| 5 | X | X | X | X | X |  |

The neighbors of the blue tile at location (1, 1) are shown in green, while the neighbors of the blue tile at location (4, 4) are shown in yellow. For any tile whose ***row coordinate*** has an odd value, the tiles that are adjacent to it follow the pattern of the green tiles. For any tile whose row coordinate has an even value, the tiles that are adjacent to it follow the pattern of the yellow tiles.

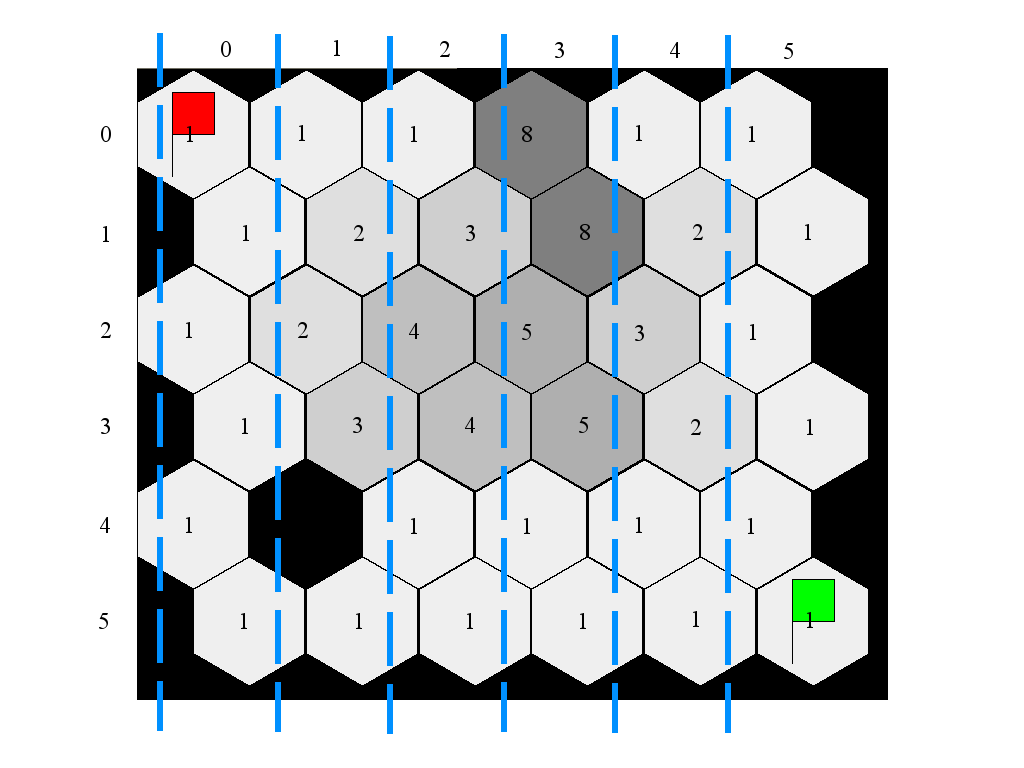
It is recommended that students build a private helper method to determine the adjacency of Tiles to simplify identification of neighboring locations. Here's an example function signature:

private areAdjacent(Tile const\* lhs, Tile const\* rhs);

Calculating Cost

Given a planner node and one of its neighbors, the **succession cost** of the neighbor is the product of the neighbor's tile weight and the distance between the two nodes. The total given cost of a path is simply the sum of the succession costs of all the nodes in that path.

As a reminder of how tile map diagrams work, here is another such diagram superimposed over a blow-up of a hexagon tile map, this time with weight values instead of X marks.



Don't forget: zero-weight tiles are impassable.

The same diagram is repeated here without the blow-up, but with an example path highlighted in red and blue (and labeled by letter):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| column | 0 | 1 | 2 | 3 | 4 | 5 |
| row |
| 0 | 1 (A) | 1 (B) | 1 | 8 | 1 | 1 |
| 1 | 1 | 2 (C) | 3 | 8 | 2 | 1 |
| 2 | 1 | 2 | 4 (D) | 5 | 3 | 1 |
| 3 | 1 | 3 | 4 (E) | 5 | 2 | 1 |

If the set of nodes {A, B, C, D, E} represents the path, then the total cost is:

*1 \* 0 + 1 \* distance(A, B) + 2 \* distance(B, C) + 4 \* distance(C, D) + 4 \* distance(D, E)*

If we remove node E (the blue node) from the path, then the total cost becomes:

*1 \* 0 + 1 \* distance(A, B) + 2 \* distance(B, C) + 4 \* distance(C, D)*

When building a graph to represent the state space, bear in mind these tips:

* Do not create nodes unnecessarily. In particular, do not create a node where one already exists, or where an obstacle is. Also, do not create nodes (or tiles, for that matter) just to store row-column coordinate information.
* When looking at the adjacencies for a specific tile, only look at the immediate neighbors; i.e. do NOT look at the entire tile map each time.

### Class API

Classes are provided to provide terrain information and a debugging interface. The API is as follows:

#### TileMap

void resetTileDrawing()  
Method will reset (wipe) all drawing on the tiles in this map.

Tile\* getTile(int row, int column) const  
Method will return a pointer to the Tile at the designated row and column.

int getRowCount() const  
Method will return the number of rows of tiles in this map.

int getColumnCount() const  
Method will return the number of columns of tiles in this map.

double getTileRadius() const  
Method will return the length of the radius of tiles in this map.

#### Tile

void resetDrawing()  
Method will reset all drawing on this tile.

void setFill(unsigned int color)  
Method will set the tile's fill color to the designated color in the LRGB color space (8-bits each for luminosity, red, green, and blue.)

void setMarker(unsigned int color)  
Method will set the tile's marker color to the designated color in the LRGB color space.

void setOutline(unsigned int color)  
Method will set the tile's outline color to the designated color in the LRGB color space.

void setLineTo(Tile\* destination, unsigned int color)  
Method will set up a line to be drawn from this tile to the destination using the designated color.

int getRow() const  
Method will return the tile's row in the map.

int getColumn() const  
Method will return the tile's column in the map.

int getXCoordinate() const  
Method will return the tile's x-axis coordinate on the map.

int getYCoordinate() const  
Method will return the tile's y-axis coordinate on the map.

### PathSearch API

Project submissions should adhere to the following public interface (no more, no less).

PathSearch()  
The constructor should take no arguments.

~PathSearch()  
The destructor should perform any final cleanup required before deletion of the object.

void initialize(TileMap\* \_tileMap)  
Method will be called after the tile map is loaded. This is usually where the search graph is generated.

void enter(int startRow, int startCol, int goalRow, int goalCol)  
Method will be called before any update of the path search and should prepare for a search to be performed between the tiles at the coordinates indicated.

void update(long timeslice)  
Method will be called to allow the path search to execute for the specified allotted time (in milliseconds). Within this method the search should be performed until the time expires or the solution is found.

If the update's allotted time is zero (0), this method should only do a single iteration of the algorithm. Otherwise the update should only iterate for the time slices number of milliseconds.

void exit()  
Method will be called when the current search data is no longer needed. It should clean up any memory allocated for this search. Note that this is not exactly the same as the destructor, as the object may be reinitialized to perform another search.

void shutdown()  
Method will be called when the tile map is unloaded to clean up any memory allocated for this tile map.

bool isDone()  
Method should return true if the update function has finished because it found a solution, otherwise it should return false.

std::vector<Tile const\*> const getSolution() const  
Method should return a vector containing the solution path as an ordered series of Tile pointers from finish to start.

## Test Data

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tile Map | Start Row | Start Column | Goal Row | Goal Column |
| hex035x035.txt | 31 | 19 | 3 | 15 |
| hex035x035.txt | 0 | 9 | 34 | 26 |
| hex035x035.txt | 2 | 19 | 31 | 19 |
| hex035x035.txt | 17 | 32 | 17 | 2 |
| hex035x035.txt | 32 | 32 | 6 | 0 |
| hex035x035.txt | 17 | 3 | 17 | 31 |
| hex054x045.txt | 44 | 53 | 3 | 51 |
| hex054x045.txt | 30 | 53 | 30 | 0 |
| hex054x045.txt | 22 | 0 | 22 | 53 |
| hex054x045.txt | 3 | 51 | 42 | 53 |
| hex054x045.txt | 8 | 0 | 34 | 53 |
| hex054x045.txt | 22 | 51 | 22 | 53 |
| hex054x045.txt | 2 | 51 | 44 | 53 |
| hex098x098.txt | 97 | 97 | 0 | 0 |
| hex098x098.txt | 38 | 44 | 97 | 0 |
| hex098x098.txt | 3 | 51 | 90 | 40 |
| hex098x098.txt | 52 | 0 | 53 | 97 |
| hex098x098.txt | 50 | 7 | 41 | 97 |
| hex098x098.txt | 93 | 0 | 53 | 80 |
| hex098x098.txt | 5 | 92 | 69 | 52 |
| hex113x083.txt | 82 | 112 | 0 | 0 |
| hex113x083.txt | 0 | 16 | 82 | 97 |
| hex113x083.txt | 14 | 0 | 70 | 112 |
| hex113x083.txt | 81 | 73 | 1 | 15 |
| hex113x083.txt | 41 | 3 | 41 | 109 |

## Frequently Asked Questions

Read this section before asking for help. Even if the question that you plan to ask is not precisely worded here, sometimes a close match may be good enough to help you resolve your issue.

### How Do I...

Q: How would we know if our search algorithms work correctly?  
A: Use the following checklist to determine the functionality of your algorithms. Refer to the test data table for particular test cases.

* For breadth-first search, what matters is that the solution size and the number of visited nodes match the completed example. The nodes should progress outward, as the nature of the algorithm suggests.
* For uniform-cost search, the given cost should also match.
* For A\*, the given and final costs should match.
* For all algorithms, the number of visited nodes should not differ by more than ten percent (10%).
* Don't forget to test your reset() methods by pressing the “Stop” button.
* As usual, check for memory leaks in Debug mode, and test your application's performance in Release mode: elapsed times should not differ by more than thirty percent (30%).

### Common Errors

Q: Why does my search algorithm crash on a Tile method call?  
A: You are most likely trying to call that method on a NULL pointer. The TileMap::getTile() method will return a NULL pointer if the row-column coordinate arguments are out of bounds.

Q: My algorithm is stuck in an infinite loop. Why does it keep revisiting some nodes?  
A: Make sure you use the correct container when performing your dupe check. A dupe check requires efficient searching: use your knowledge of data structures to figure out which container fulfills this requirement. Then, step through your logic with the debugger: do not create a node if you find a duplicate.

Q: These message boxes pop up when my search algorithm finishes! What's going on?  
A: Your search algorithm must update the container that will hold the solution path...correctly. Read the error messages; they'll clue you in on what's wrong.

Q: Why doesn't my search algorithm display the red line?  
A: One or both of the following may apply:

* The application uses an accessor that is part of your algorithm class to draw the current path. (Read the API documentation to find out which method gives this access.) Obviously, you must update the variable that this accessor returns at each iteration.
* The overview discusses how to traverse the current path. You must set the attribute that this traversal routine uses when initializing or updating each PlannerNode object.

Q: When I run my search algorithm, the solution path doesn't look valid. Why?  
A: Set up adjacent nodes to be drawn somehow (using a marker or outline) and use the “Step” button. If tiles aren't marked where they should be at each step, then your tile adjacency check may be incorrect. Otherwise, you're either not using it at all or using it incorrectly.

Q: My algorithm builds the correct solution path, but the red line doesn't follow those small blue circles all the way, and the output in the lower right corner indicates that the algorithm hasn't found the goal. Why?  
A: The code that you wrote to handle the finding of the goal is missing an important step.

Q: Why does my search algorithm find the goal only when it is below and to the right of the start?  
A: The start and goal row-column coordinates are initialization parameters for a reason. Do not mistake them for the tile map dimensions: get ***that*** information from the tile map itself.

Q: My Greedy Search algorithm outputs a valid path, but it goes the wrong way. Why?  
A: Examine the information display. If the “distance from start” does not match the completed example, then your heuristic estimate (distance calculation) may be incorrect.

Q: Why can't we use either the <= or the => comparison operators when implementing the comparison functions?  
A: When passed as an argument to STL heap operations, each comparison function must return false when comparing two “equivalent” elements. The results of not doing so range from invalid operand assertion failures to access violations.

Q: Why does my A\* search find a more costly path and create more nodes than it should when the heuristic weight is 2?  
A: If you find a cheaper path from the start to the successor node, then you must update this node. However, if this node is in the open heap when you update it, you may mess up the heap's internal ordering, so you must restore this ordering.