FULL SAIL UNIVERSITY

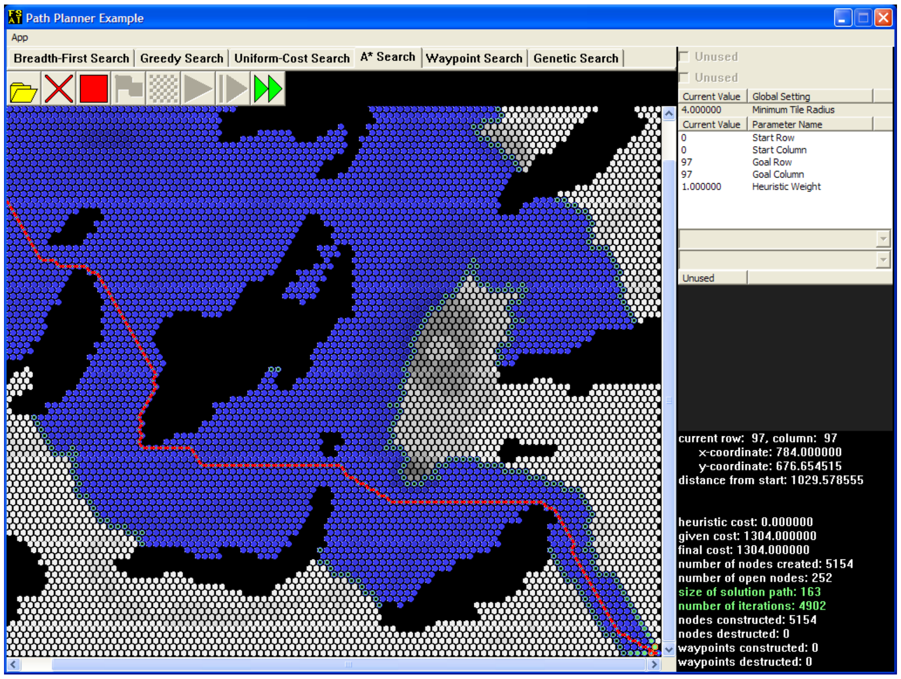
# Path Planner – Lab

**NOTE: Use the PriorityQueue provided! Do not implement your own. Also, you are allowed to make it (and directly related code) public/global (but only these!)**

The path planner 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.)



| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 97 | Goal Row |
| 97 | Goal Column |
| 1.000000 | Heuristic Weight |

```
current row:  97, column:  97
     x-coordinate: 784.000000
     y-coordinate: 676.654515
distance from start: 1029.578555


heuristic cost: 0.000000
given cost: 1304.000000
final cost: 1304.000000
number of nodes created: 5154
number of open nodes: 252
size of solution path: 163
number of iterations: 4902
nodes constructed: 5154
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

For the **path planner** lab, you will implement 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, which is a variant of the A* algorithm. The user interface is fully functional; however, the search algorithm consists of no-op functions and methods at the moment. Your algorithm will construct a complete path solution.

## Table of Contents

- Objectives & Outcomes
- Level of Effort
- Reading & Resources
- Instructions
    - General User Interface
        - Tile Map Display
        - Button Controls
        - Checkboxes
        - Global Settings
        - Parameter Settings
        - Information Display
    - Environment and Tools
        - Development Interface
            - Console
            - Drawing System
        - State Space Structure
            - Invalid Endpoints
            - Hexagon Tiles
            - Calculating Cost
        - STL Containers
            - std::pair<T1,T2>
            - std::vector<T>
            - std::set<T>
            - std::map<K,V>
            - std::deque<T>
        - Class API
            - TileMap
            - Tile
        - PathSearch API (Student TODO)
    - Search Algorithms
        - Breadth-First Search
            - Algorithmic Outline
            - Example Implementation
        - Greedy (Best-First) Search

            - Algorithmic Outline
        - Uniform-Cost Search
            - Same Endpoints, Different Costs
            - Algorithmic Outline
        - A* (A-Star)
            - Algorithmic Outline
        - Test Data
    - Frequently Asked Questions
        - How Do I...
        - Common Errors
    - Code Conventions
- Deliverables
    - Deliverable File(s) and Contents

## Objectives & Outcomes

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

- When provided with an engine API, design an architecture for AI behaviors (such as path planning).
- 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).

## Level of Effort

This activity should take approximately 675m to complete. It will require:

- 60m Research
- 15m Prep & Delivery
- 600m Work

If you find that this activity takes you significantly less or more time than this estimate, please contact me for guidance.

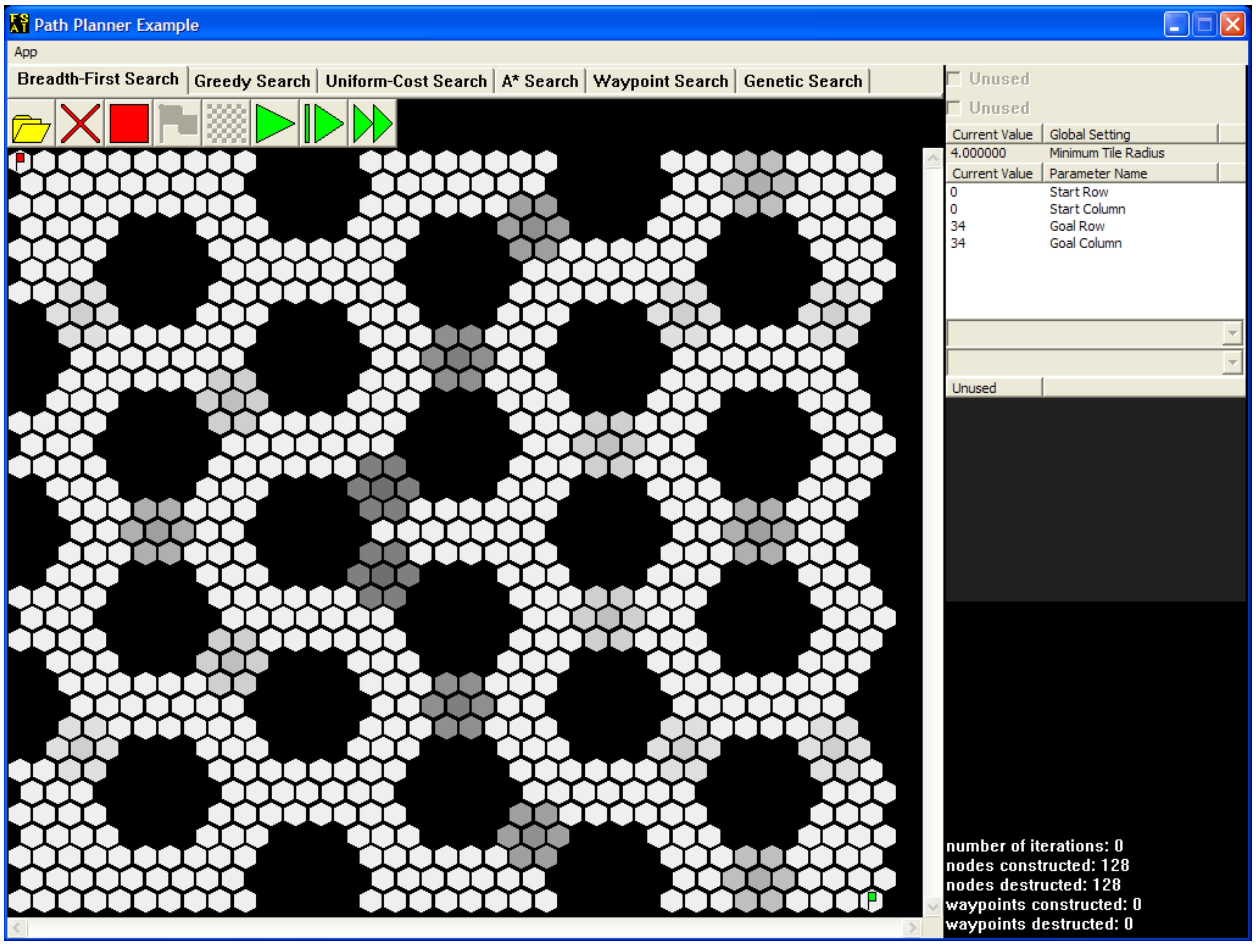
## Reading & Resources

Artificial Intelligence for Games *(helpful)*
*Chapters: 4 (4.1-4.8)*
*Page(s): 197-282*
Course textbook

# Instructions

## General User Interface

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



| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 34 | Goal Row |
| 34 | Goal Column |

```
number of iterations: 0
nodes constructed: 128
nodes destructed: 128
waypoints constructed: 0
waypoints destructed: 0
```

### *Tile Map Display*

The search algorithms for this lab 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 selected to run, the red flag sits on top of the starting tile, while the green flag points to the goal tile.

## 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. |

### *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.

### *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, from the ground up, 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

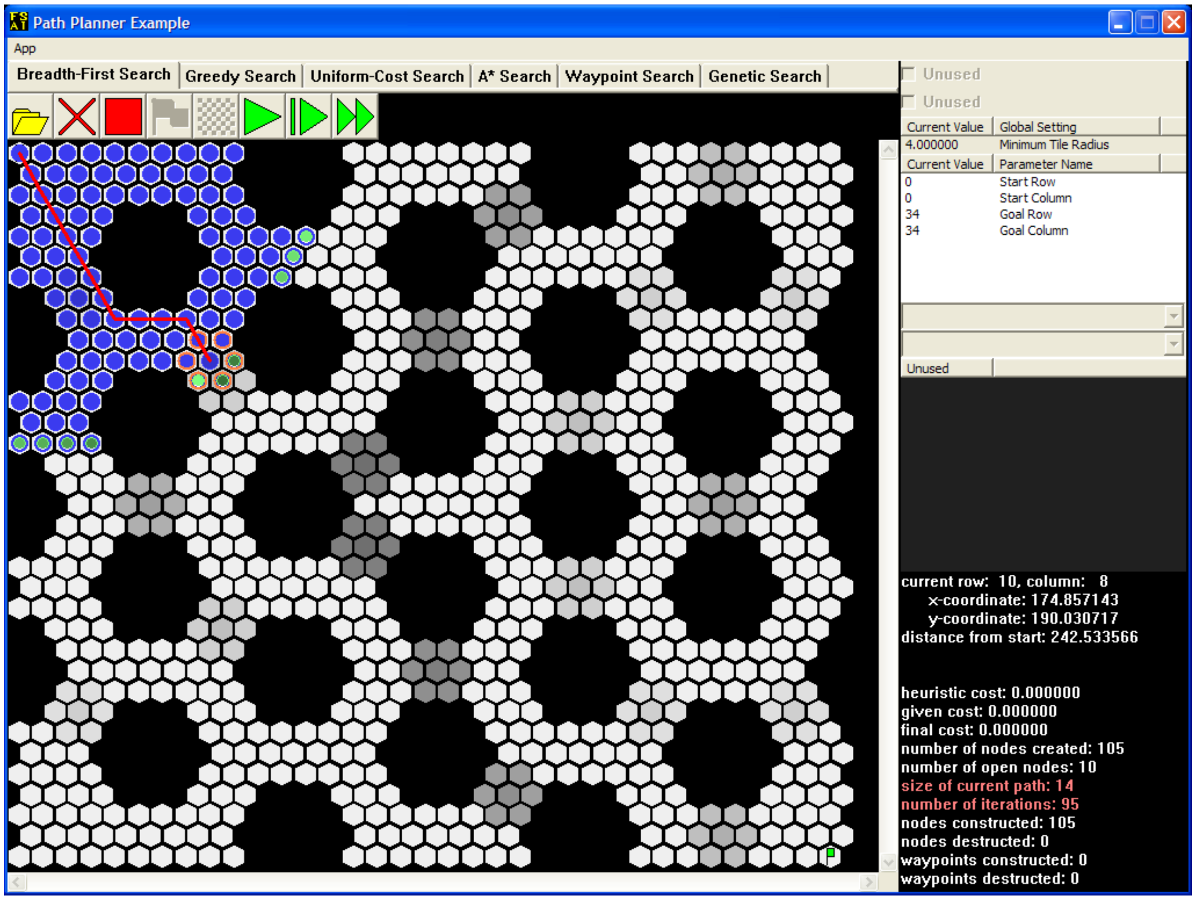
A console window, which can be written to by students using standard streams such as cin and cout, is automatically spawned for use by students. This can be a useful tool for textual output.

## 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*



| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 34 | Goal Row |
| 34 | Goal Column |

Unused

```
current row:  10, column:   8
     x-coordinate: 174.857143
     y-coordinate: 190.030717
distance from start: 242.533566


heuristic cost: 0.000000
given cost: 0.000000
final cost: 0.000000
number of nodes created: 105
number of open nodes: 10
size of current path: 14
number of iterations: 95
nodes constructed: 105
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

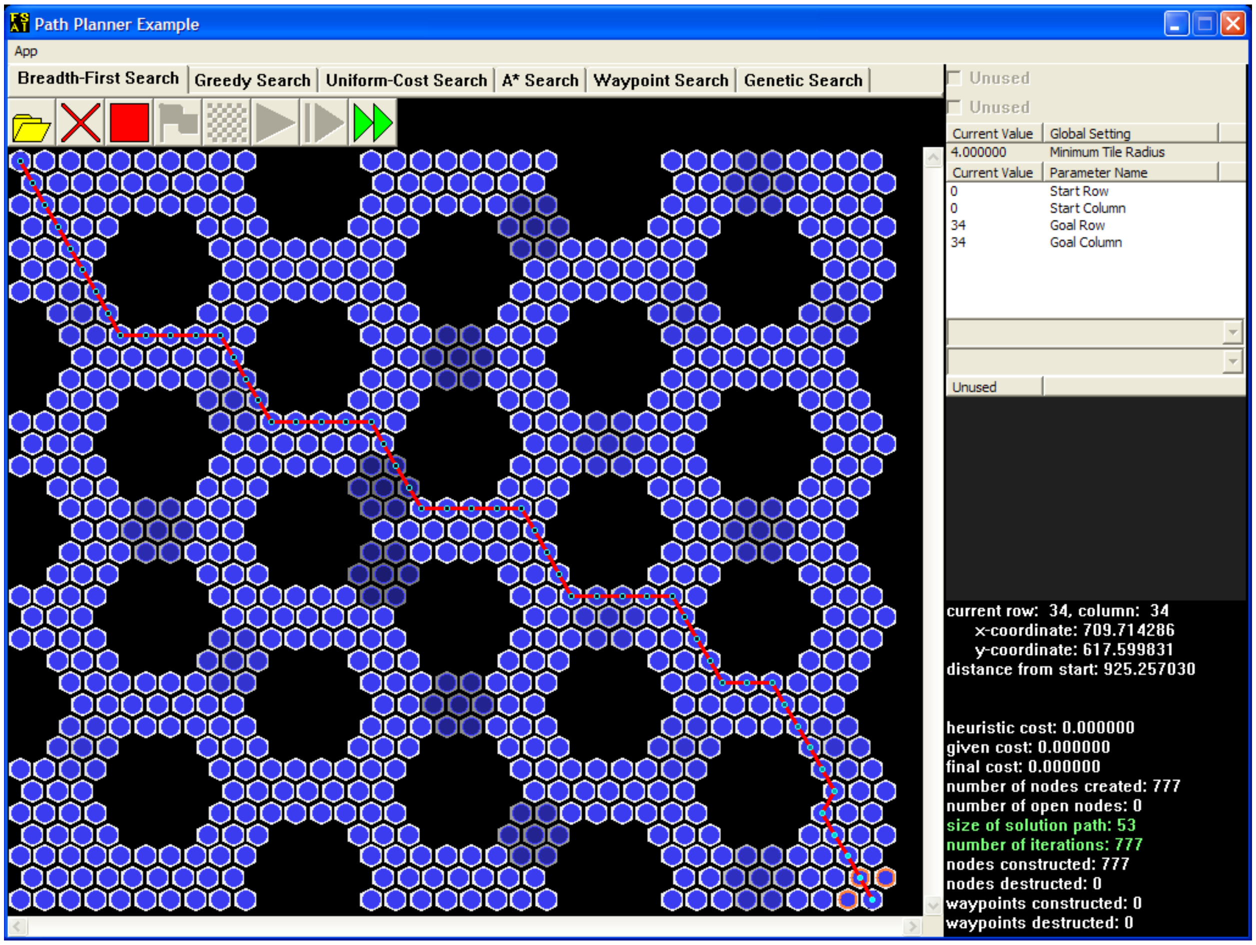
All search algorithms must keep track of different paths that may lead to the goal. By doing this, a search algorithm can extend these paths and so work its way through the tile map.

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.

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



| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 34 | Goal Row |
| 34 | Goal Column |

```
current row:  34, column:  34
    x-coordinate: 709.714286
    y-coordinate: 617.599831
distance from start: 925.257030

heuristic cost: 0.000000
given cost: 0.000000
final cost: 0.000000
number of nodes created: 777
number of open nodes: 0
size of solution path: 53
number of iterations: 777
nodes constructed: 777
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

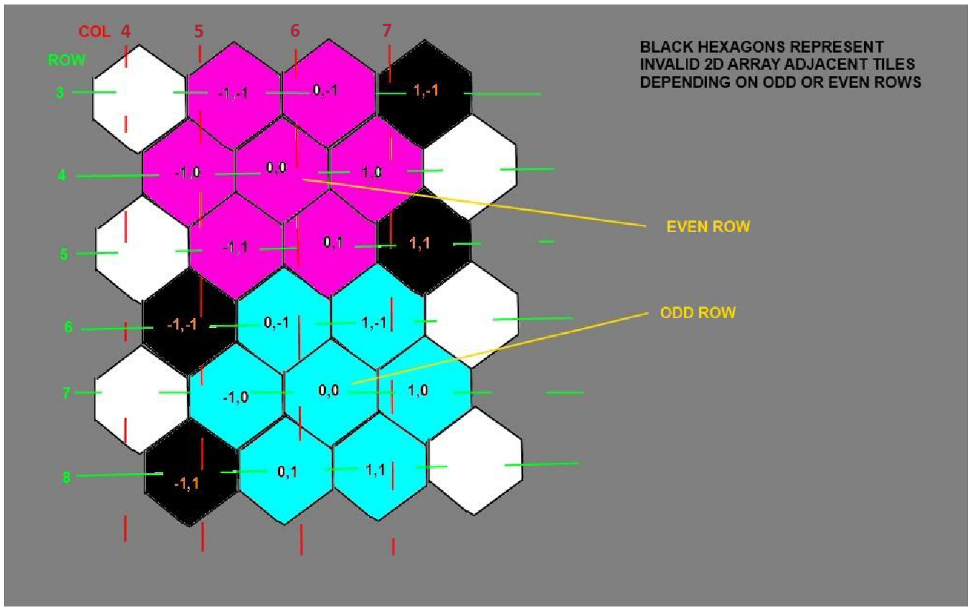
## Invalid Endpoints

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 lab.

| 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.

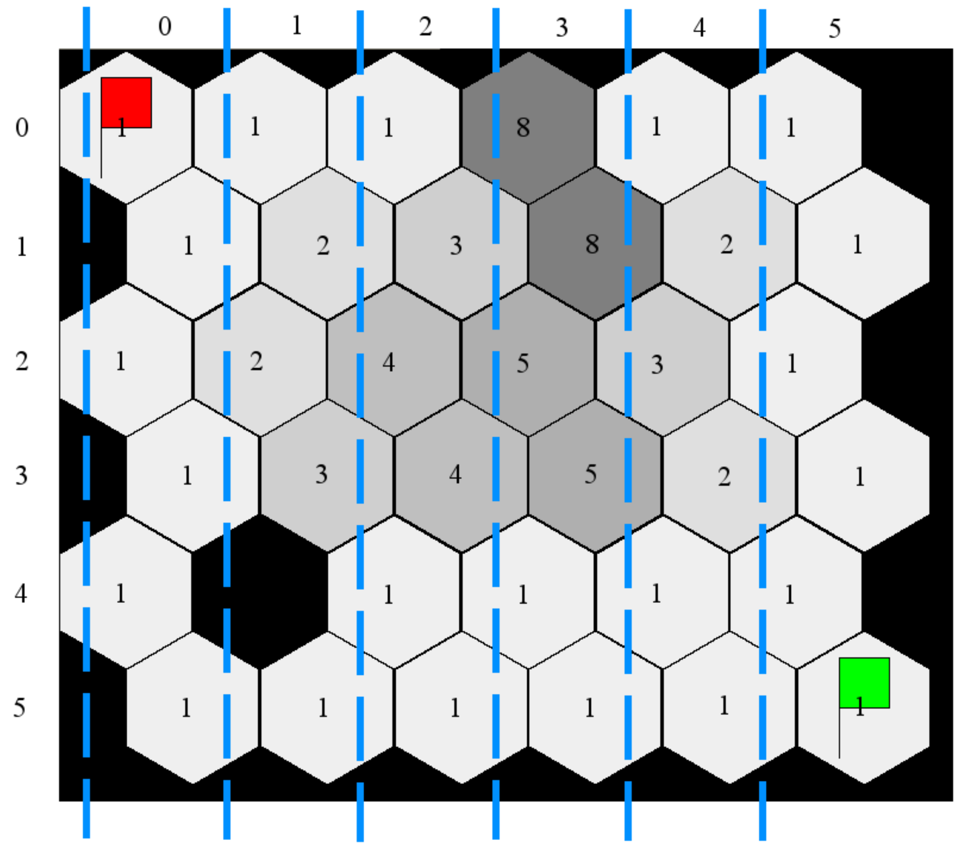
It is recommended that students build a private helper method to determine the adjacency of Tiles to simplify identification of neighboring locations. Here's an example function signature:

```
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** / **row** | **0** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| **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.

## STL Containers

Most of the types of containers that will store your data come straight from the Standard Template Library. You will need to know how to use them in order to successfully implement these search algorithms. For a full description of all classes and their members, refer to the API documentation accompanying this manual.

std::pair<T1,T2>

This type represents a pair of values. It provides the following relevant attributes:

`T1& first`
The first element in the pair.

`T2& second`
The second element in the pair.

std::vector<T>
You can treat a `std::vector` as a runtime-resizable array. It provides the following relevant operations, pulled from the API documentation:

`bool empty() const`
Returns true if there are no elements left in this container, false otherwise.

`void clear()`
Removes all elements from this container.

`std::size_t size() const`
Returns the number of elements in this container.

`T& operator[](std::size_t index)`
Returns the element at the specified index in this container.

`T& back()`
Returns the last element in this container.

`void push_back(T& element)`
Appends the specified element to the back of this container.

`void pop_back()`
Removes the last element from this container.

std::set<T>

A set can contain only a single instance of a particular value of a type. This makes it useful for identifying values that are part of a group (such as a set of elements that have been visited or seen.) It is most efficient when used for random access of elements.

`bool empty() const`
Returns true if there are no elements left in this container, false otherwise.

`void clear()`
Removes all elements from this container.

`iterator begin()`
Returns an iterator pointing to the first value in this container. Returns end() if the container is empty.

`iterator end()`
Returns an iterator pointing past-the-end of this container.

`iterator find(T& element)`
Returns an iterator to the specified element (if contained in the set)

`pair<iterator, bool> insert(T& element)`
Inserts the element into the container.

`void push_back(T& element)`
Appends the specified element to the back of this container.

`void pop_back()`
Removes the last element from this container.

std::map<K,V>

This type defines the container that should hold all the planner nodes you will create in this algorithm. The `std::map` is a map data structure, meaning it holds key-value pairs (much like a hash map does.)

`bool empty() const`
Returns true if there are no key-value pairs left in this container, false otherwise.

`void clear()`
Removes all key-value pairs from this container.

`iterator begin()`
Returns an iterator pointing to the first key-value pair in this container. Returns end() if the container is empty.

`iterator end()`
Returns an iterator pointing past-the-end of this container.

`V& operator[](K const& key)`
Finds the key-value pair whose first element matches the specified key, and returns the second element. If the type V is a pointer type, returns NULL if the key is not found; this results in a new `std::pair(key, NULL)` being inserted into this container.

## std::deque<T>

This container type defines the open list that you will use in this algorithm. Similar in functionality to the std::list type, it takes up much less internal memory and suffers less memory fragmentation in general. It is often used to implement the heap for a priority queue. Here's some relevant operations:

`bool empty() const`
Returns true if there are no elements left in this container, false otherwise.

`void clear()`
Removes all elements from this container.

`T& front()`
Returns the first element in this container.

`void push_front(T& element)`
Appends the specified element to the front of this container.

`void pop_front()`
Removes the first element from this container.

`T& back()`
Returns the last element in this container.

`void push_back(T& element)`
Appends the specified element to the back of this container.

`void pop_back()`
Removes the last element from this container.

This example program starts off with some C-style strings being added to a `std::deque` that is initially empty:

```
std::deque<char const*> names;
names.push_front("charlie");
names.push_front("bravo");
names.push_front("alpha");
names.push_back("charlie");
names.push_back("echo");
names.push_back("foxtrot");
```

The program ends by removing all names, one at a time, alternating from either end.

```
while (!names.empty())
{
 if (names.size() % 2)
 {
  names.pop_front();
 }
 else
 {
  names.pop_back();
 }
}
```

When working with this container, the word element is interchangeable with planner node.

## *Class API*

In addition, classes will be provided to students 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*

All student searches will 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 planner 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 planner 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. It should 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.

## Search Algorithms

This manual covers the several common searches in the breadth-first family of search algorithms. Students are strongly advised to start with the breadth-first search and build up to A* as each algorithm is an built on top of the previous ones: you will not be able to understand A*, the most common optimal search used in games today, without a solid grasp of the other basic search algorithms.

As you implement each algorithm, be sure to compare its performance is comparable with that of the completed application so that you know you're on the right track. Also, test your debug build for memory usage and stability, and test your release build for speed.

## *Breadth-First Search*

Conceptually speaking, this algorithm is a generalization of the tree traversal method with the same name that you've seen before in the Data Structures and other courses. (On the other hand, the programming interface that you will work with here is not as *generic* because you cannot substitute arbitrary data types.)

### Algorithmic Outline

The basic data structures will have roughly the following definitions.

```
Tile                                TileMap
{                                   {
 row, column;                        getTile(row, column);
 x, y, weight;                       rowCount, columnCount;
};                                  };
```

Tile objects store physical data such as ***coordinates*** and ***weights***. As the Tiles are already created for you, you should never create `Tile` objects: instead, they are always accessible from a passed-in `TileMap` instance.

In addition, breadth-first derived searches use a tree structure to search the state space (i.e., the terrain) so a tree node data structure for planning will be necessary. Here is an example of the skeleton of such a node data structure:

```
PlannerNode
{
  Node* parent;
  SearchNode* vertex;
};
```

PlannerNode objects accumulate path information. Each path can be considered a linked list in reverse, from the end node to the start.

Here's a sketch of how a breadth-first search would work over tiles:

```
1. Find the graph vertex that represents the start tile.
2. Create a new PlannerNode at the vertex.

3. Enqueue this node into the open container.
4. Associate the start location with this node in the visited map.
5. While the open container possesses nodes, do the following:
    1. Dequeue the current node from the open container.
    2. If this node is at the goal, build the solution list and exit.
    3. Otherwise, for each successor of current:
        1. If successor is not a duplicate:
            1. Enqueue a PlannerNode at the successor into the open
               container.
            2. Associate the successor's location with this node in
               the visited map.
```

Remember, each algorithm is split into five major parts: an `initialize()` method that will take in data about the map, an `enter()` method that will take the start and goal parameters and contain the code before the `while` loop, an `update()` that will contain the while loop itself, an `exit()` method called upon completion of the planning of a path, and a `shutdown()` that will be called when the map is unloaded. This division enables the path planner application to step through or pause at each iteration.

Below is a simple example of using a `std::map` to find a key-value pair – in this case, a `PlannerNode` at the specified vertex:

```
created[vertex] = myPlannerNode;
```

Finally, you will need to use iterators in order to properly clean up after each algorithm. The path planner application executes a subroutine just like this one to render your created planner nodes.

```
for (auto iterator itr = created.begin(); itr !=
created.end(); ++itr)
{
  drawNode(itr->second);
}
```

Example Implementation

Following are guidelines for implementing a Breadth First Search (BFS) in the

`PathSearch` methods. Students are encouraged to implement these, and then use the test data table to check performance against the Breadth First Search example.

`void initialize(...)`

- Create a `SearchNode` for each valid tile in the tile map.
- Create edges to connect each pair of adjacent tiles.
- Add all created node to an index (such as a map) that be easily accessed to find the nodes by tile and allows for easy clean up later.

`void enter(...)`

- Find the `SearchNode` that represents the starting tile.
- Set the goal tile (or node) so that you know when to stop the algorithm.
- Add the starting node to the open queue and mark it as visited.

`void update(long timeslice)`

- While the open list contains planner nodes...
  1. Dequeue what will become the current node.
  2. Once you find the goal, build the solution list and exit.
  3. Determine the successor nodes. Make sure that you go through only those nodes whose tiles are adjacent to that of the current node.
     - If a node has not been visited—this check is called a dupe check—add it to the open queue and mark it as visited.
     - Don't forget the rule regarding impassable tiles.

`void exit()`

- Reset member variables as needed.

`void shutdown()`

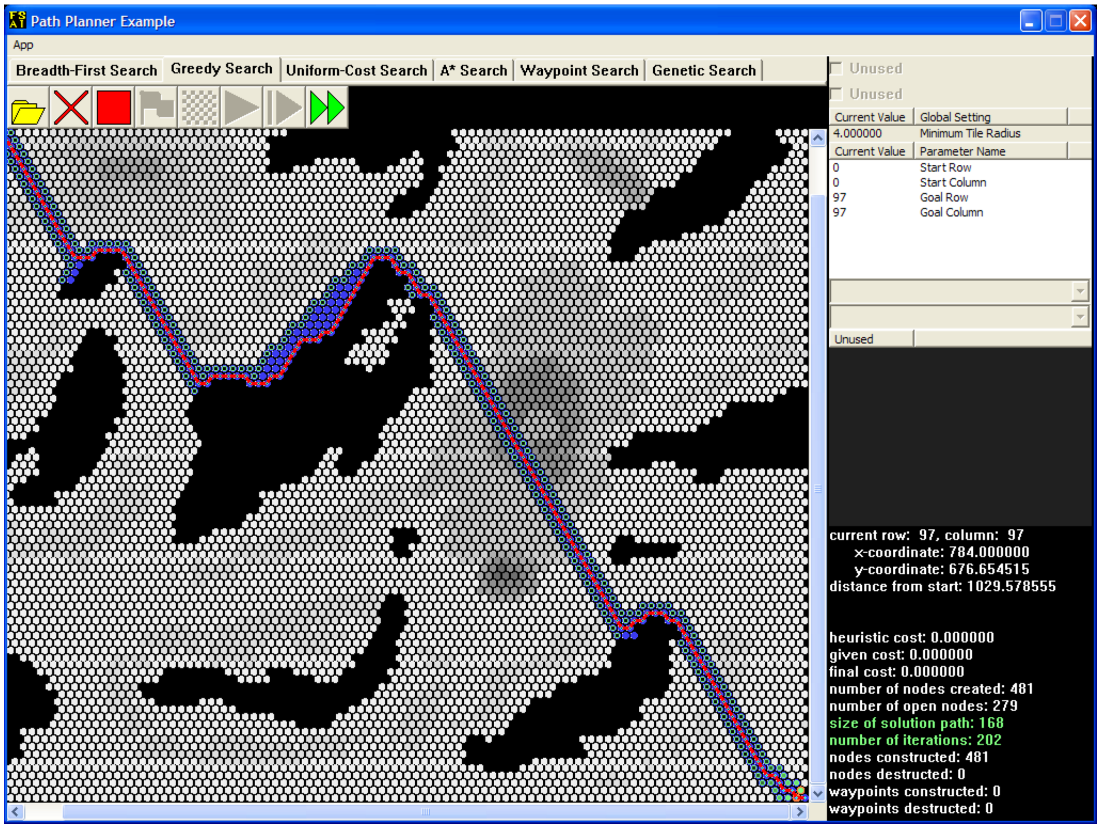
- Free all `SearchNode` memory.

## *Greedy (Best-First) Search*

While the breadth-first search algorithm will find the solution path with the least number of planner nodes, it may create an extremely large number of them before that happens.The main problem is that the open list is treated like a regular queue. By changing the behavior of the open list so that it orders its nodes a certain way, we can attempt a more efficient search.

The **greedy best-first search algorithm**, or just the **greedy search algorithm**, orders its nodes so that the ones closest to the goal are the nearest to the front of the open list using a **priority queue**. This algorithm is **informed** because its ordering is based on goal information.Using the distance to the goal to estimate which open node is "best" is an example of a **heuristic**.

Here is what happens when you open "`hex098x098.txt`" and run the algorithm.



| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

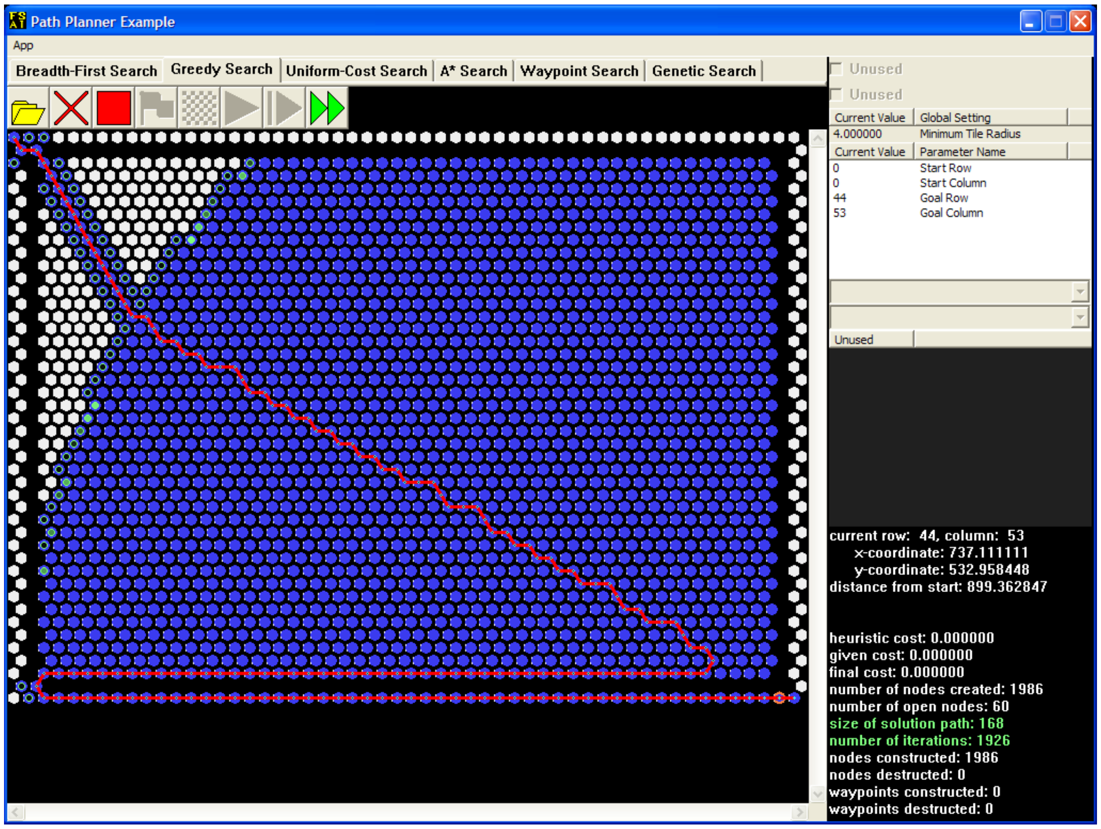
| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 97 | Goal Row |
| 97 | Goal Column |

```
current row:  97, column:  97
     x-coordinate: 784.000000
     y-coordinate: 676.654515
distance from start: 1029.578555

heuristic cost: 0.000000
given cost: 0.000000
final cost: 0.000000
number of nodes created: 481
number of open nodes: 279
size of solution path: 168
number of iterations: 202
nodes constructed: 481
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

The breadth-first search algorithm would have visited all 8001 tiles before reaching the goal. In straightforward cases like this one, the greedy search algorithm presents huge savings in memory and CPU usage.

However, in certain cases, the greedy search algorithm may produce a solution that is far worse than the result of a breadth-first search. Open "`hex054x045.txt`" and run the algorithm again.



| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 44 | Goal Row |
| 53 | Goal Column |

```
current row:  44, column:  53
    x-coordinate: 737.111111
    y-coordinate: 532.958448
distance from start: 899.362847


heuristic cost: 0.000000
given cost: 0.000000
final cost: 0.000000
number of nodes created: 1986
number of open nodes: 60
size of solution path: 168
number of iterations: 1926
nodes constructed: 1986
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

You can easily see that greedy search is not an **optimal** algorithm. While greedy search may be effective in certain environments, we should note that a greedy search's results are highly dependent on the search space.

## Algorithmic Outline

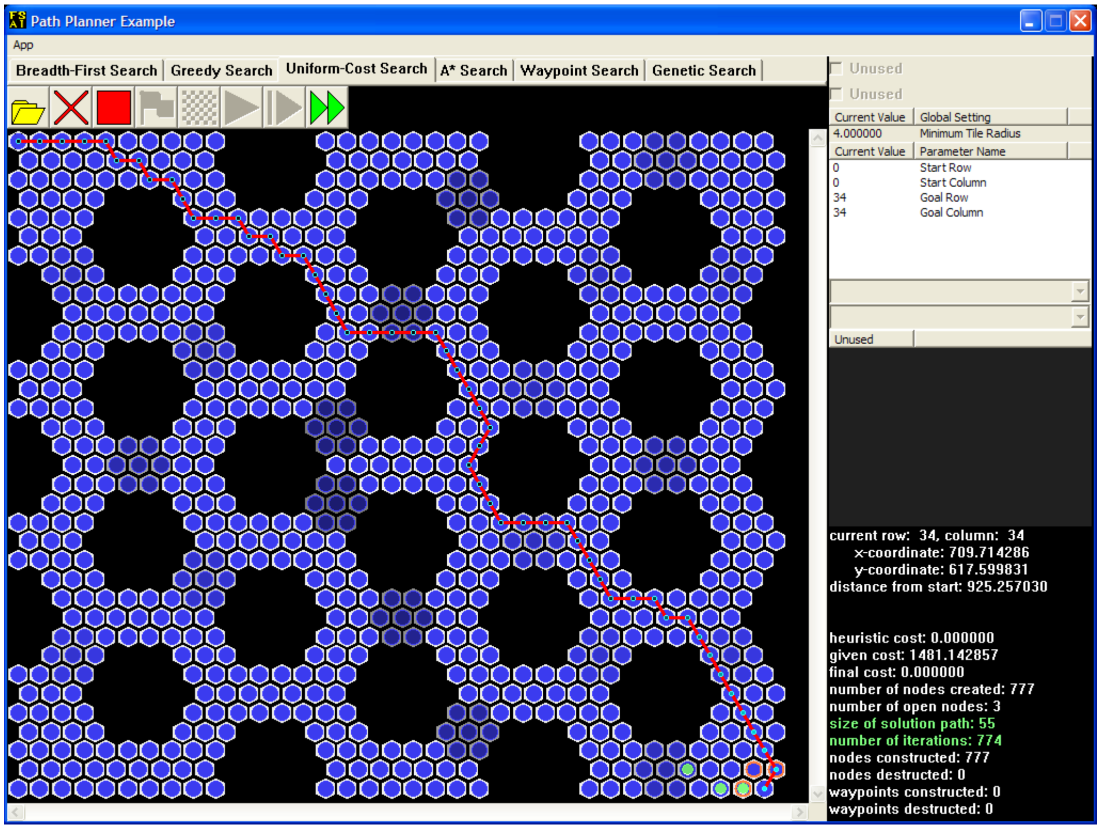
Here's a sketch of how a greedy search would work over tiles. Instructions highlighted in green indicate additions to the breadth-first search outline.

```
1. Determine the SearchNode that represents the start tile.
2. Set its heuristicCost equal to the distance between the start and goal
   tiles.
3. Enqueue this node into the open container (prioritized).
4. Add this node to the visited map.
5. While the open container possesses nodes, do the following:
   1. Dequeue the current node from the open container.
   2. If this node is at the goal tile, build the solution list and exit.
   3. Otherwise, for each successor of current:
      1. If successor is not a duplicate:
         1. Set its heuristicCost equal to the distance between its tile
            and the goal tile.
         2. Enqueue this node into the open container.
         3. Add successor to the visited map.
```

## *Uniform-Cost Search*

Both the breadth-first search algorithm and the greedy search algorithm assume that each passable tile incurs the same cost to traverse. The **uniform-cost search algorithm**—which is actually a variant of **Dijkstra's algorithm** that searches for only one goal instead of multiple goals—handles the different tile weights by incorporating them into each planner node's **given cost**, which the algorithm then uses to order its open nodes.

Since the ordering imposed on the open nodes is *not* based on any goal information, this algorithm, like the breadth-first search algorithm, is **uninformed**. It is also **optimal** in that the solution path it finds will always have the lowest cost, while the breadth-first search algorithm is optimal only with regard to the number of nodes in the path. Otherwise, a uniform-cost search takes up just as much memory in terms of node count as a breadth-first search that was initialized with the same row-column coordinates.



| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 34 | Goal Row |
| 34 | Goal Column |

```
current row:  34, column:  34
    x-coordinate: 709.714286
    y-coordinate: 617.599831
distance from start: 925.257030

heuristic cost: 0.000000
given cost: 1481.142857
final cost: 0.000000
number of nodes created: 777
number of open nodes: 3
size of solution path: 55
number of iterations: 774
nodes constructed: 777
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

## Same Endpoints, Different Costs

Previous search algorithms would not create a successor node where one already exists; in fact, that successor node would no longer be processed during that iteration. This algorithm, however, needs to perform further processing of existing successor nodes. Consider these two examples:

| **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 |

| **column** | **0** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| **row** | | | | | | |
| **0** | 1 | 1 | 1 | 8 | 1 | 1 |
| **1** | 1 | 2 | 3 | 8 | 2 | 1 |
| **2** | 1 | 2 | 4 | 5 | 3 | 1 |
| **3** | 1 | 3 | 4 | 5 | 2 | 1 |

The first path is more expensive than the second one, but the successor node shown in blue still points to the old path. In this case, the algorithm should update the successor node so that it is part of the new path and holds the cheaper given cost.

The given cost of any node can be calculated in terms of its succession cost and its parent's given cost:
*node->givenCost = parent->givenCost + successionCost*

## Algorithmic Outline

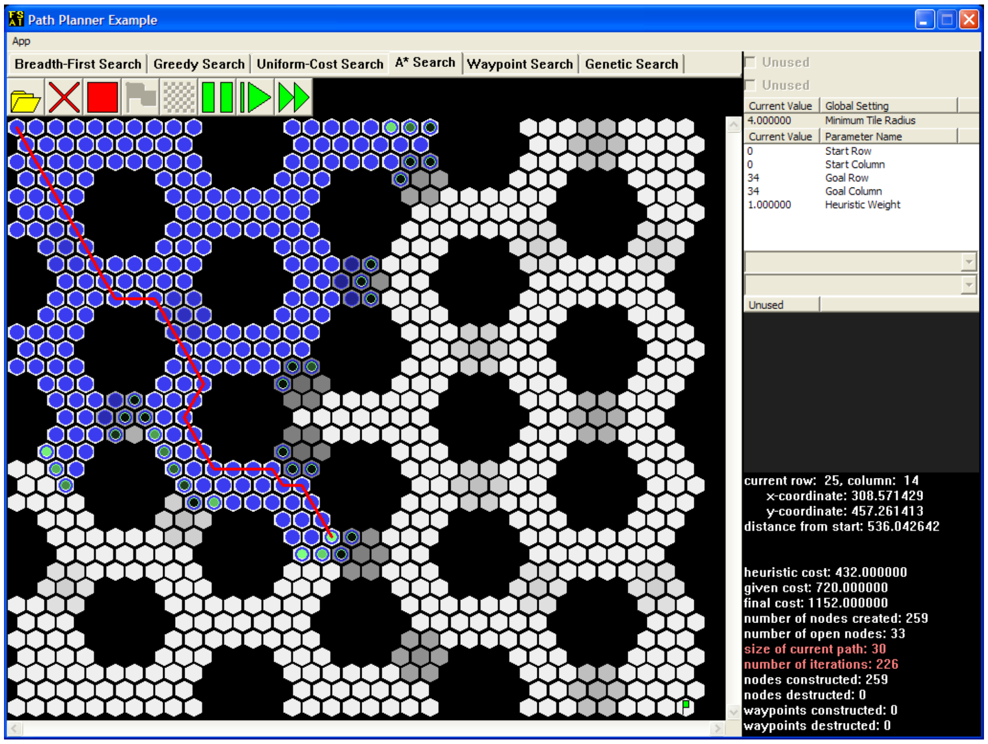
Here's a sketch of how a uniform-cost search would work over tiles. Instructions highlighted in green indicate additions to the breadth-first search outline.

```
1. Determine the SearchNode that represents the start tile.
2. Enqueue this node into the open container.
3. Add this node to the visited map.
4. While the open container possesses nodes, do the following:
    1. Dequeue the current node from the open container.
    2. If this node is at the goal tile, build the solution list and exit.
    3. Otherwise, for each successor of current:
        1. Compute the newGivenCost of the successor.
        2. If successor is not a duplicate:
            1. Set its givenCost equal to the newGivenCost.
            2. Enqueue this node into the open container.
            3. Add successor to the visited map.
        3. If successor is a duplicate, but newGivenCost is cheaper than its
            givenCost:
            1. Set its givenCost equal to the newGivenCost.
            2. Update the path to successor.
            3. Update the open container.
```

## *A* (A-Star)*

A* combines the look-ahead abilities of the greedy search algorithm with the look-behind abilities of the uniform-cost search algorithm by taking into account both the heuristic cost and the given cost of each planner node. While the definition of the given cost hasn't changed, the heuristic cost is now the product of the greedy search heuristic cost and the **heuristic weight**, whose value can be changed to customize the behavior of A*. The sum of the new heuristic cost and the given cost is the **final cost**, which A* then uses to sort the open nodes.

With the default settings, observe A* in action. While it tends to favor paths that head more directly toward the goal, it tries to avoid crossing regions with higher weights until there is no other choice.



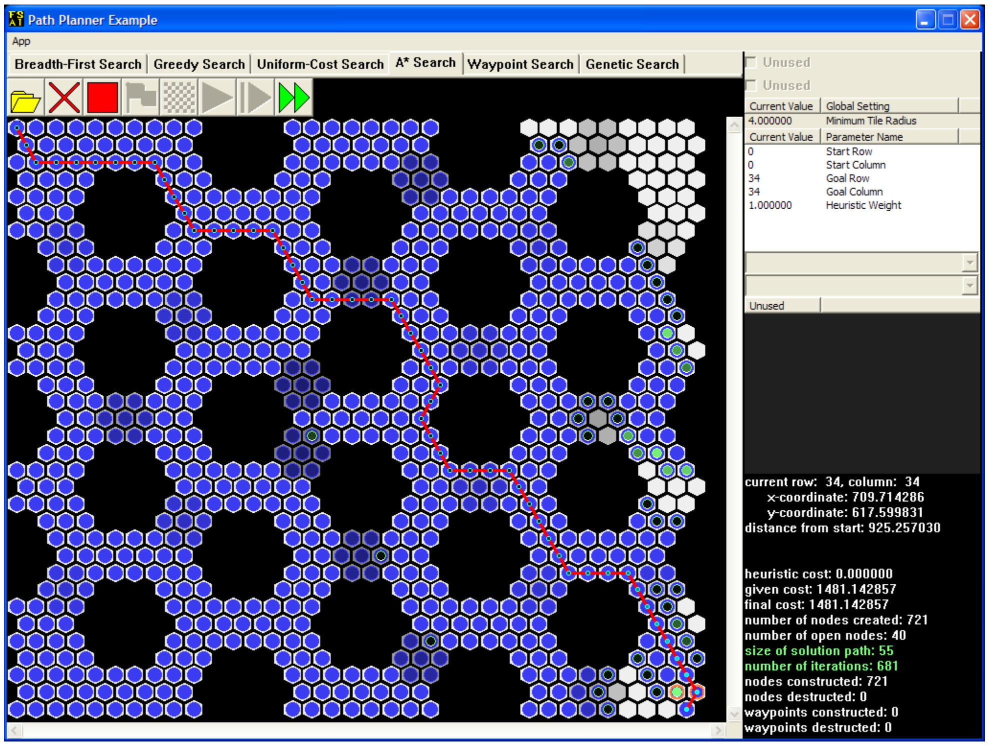
| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 34 | Goal Row |
| 34 | Goal Column |
| 1.000000 | Heuristic Weight |

```
current row:  25, column:  14
     x-coordinate: 308.571429
     y-coordinate: 457.261413
distance from start: 536.042642

heuristic cost: 432.000000
given cost: 720.000000
final cost: 1152.000000
number of nodes created: 259
number of open nodes: 33
size of current path: 30
number of iterations: 226
nodes constructed: 259
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

When it reaches the goal, the solution path will have the same given cost as the path found by a uniform-cost search with the same settings, but A* will have visited fewer nodes.



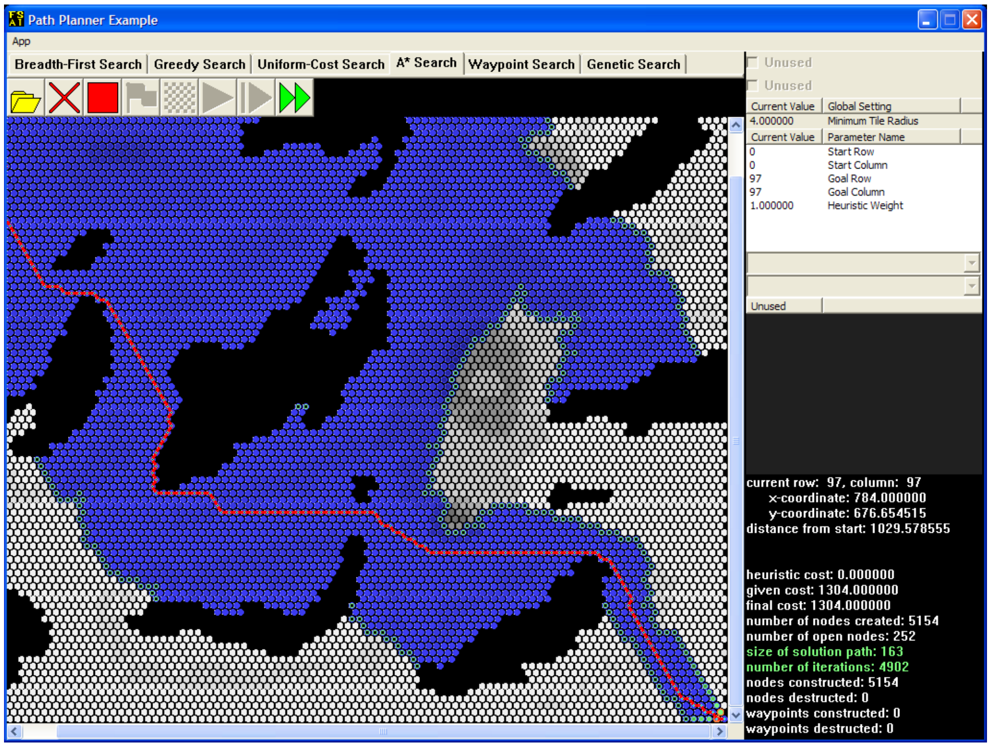
| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 34 | Goal Row |
| 34 | Goal Column |
| 1.000000 | Heuristic Weight |

```
current row:  34, column:  34
     x-coordinate: 709.714286
     y-coordinate: 617.599831
distance from start: 925.257030

heuristic cost: 0.000000
given cost: 1481.142857
final cost: 1481.142857
number of nodes created: 721
number of open nodes: 40
size of solution path: 55
number of iterations: 681
nodes constructed: 721
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

The time savings becomes more apparent in large maps like this one.



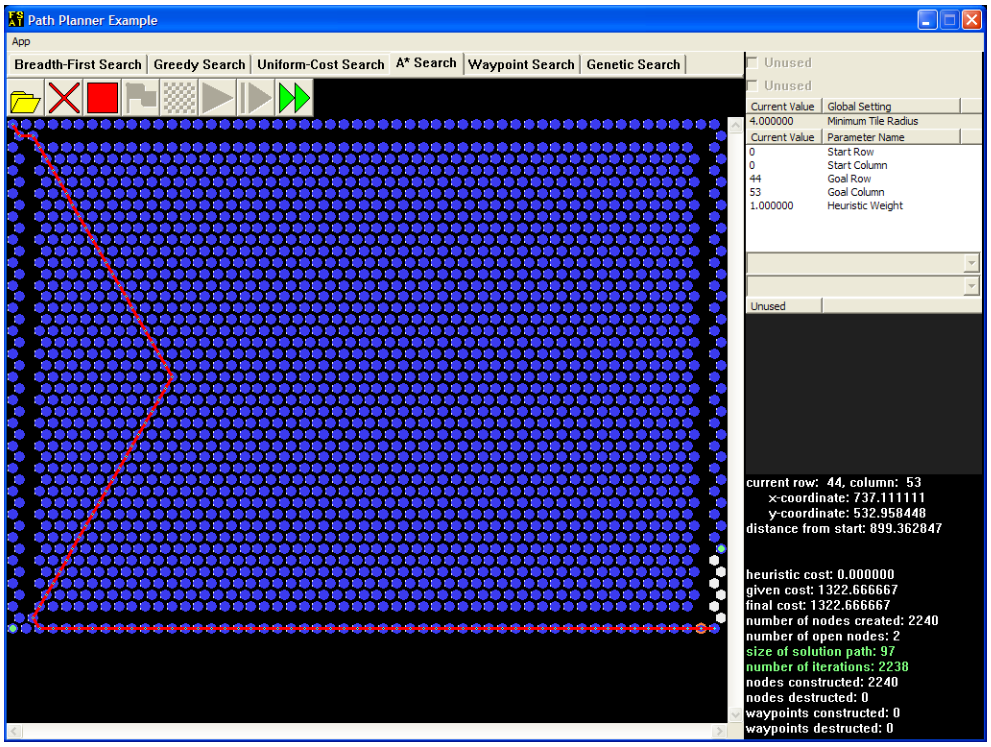
| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 97 | Goal Row |
| 97 | Goal Column |
| 1.000000 | Heuristic Weight |

```
current row:  97, column:  97
     x-coordinate: 784.000000
     y-coordinate: 676.654515
distance from start: 1029.578555

heuristic cost: 0.000000
given cost: 1304.000000
final cost: 1304.000000
number of nodes created: 5154
number of open nodes: 252
size of solution path: 163
number of iterations: 4902
nodes constructed: 5154
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

Most importantly, A* can perform well in those cases where the greedy search algorithm does poorly.



| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 44 | Goal Row |
| 53 | Goal Column |
| 1.000000 | Heuristic Weight |

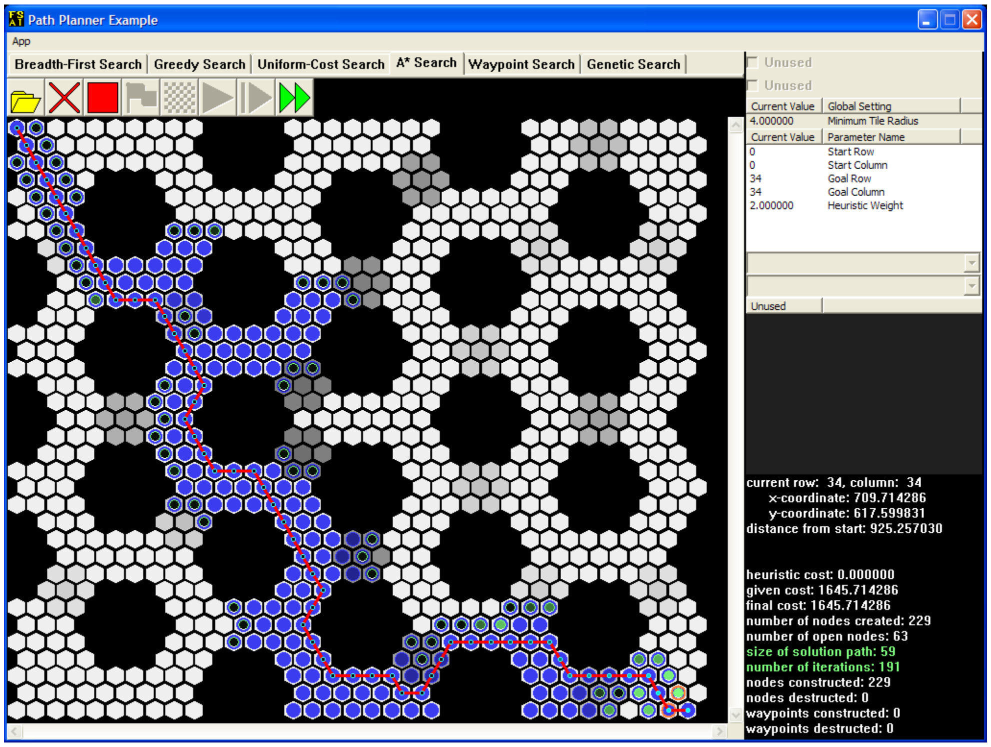
```
current row:  44, column:  53
     x-coordinate: 737.111111
     y-coordinate: 532.958448
distance from start: 899.362847

heuristic cost: 0.000000
given cost: 1322.666667
final cost: 1322.666667
number of nodes created: 2240
number of open nodes: 2
size of solution path: 97
number of iterations: 2238
nodes constructed: 2240
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

It turns out that when a planner node's heuristic cost is calculated using only its distance to the goal, A* never overestimates the given cost, so the heuristic is **admissible**. When the heuristic used is admissible, A* solution paths are always **optimal**. For this lab, any heuristic weight with a value between zero and one inclusive results in an admissible heuristic.

A* with a heuristic weight of zero is a special case: it behaves exactly like the uniform-cost search algorithm.

Now, just because a heuristic is inadmissible doesn't mean it can't be useful. In situations with tighter CPU or memory usage constraints, a near-optimal solution path may be good enough. Open “`hex035x035.txt`” (the default tile map) and run the algorithm with a heuristic weight of 2.



| Current Value | Global Setting |
|---|---|
| 4.000000 | Minimum Tile Radius |

| Current Value | Parameter Name |
|---|---|
| 0 | Start Row |
| 0 | Start Column |
| 34 | Goal Row |
| 34 | Goal Column |
| 2.000000 | Heuristic Weight |

Unused

```
current row:  34, column:  34
    x-coordinate: 709.714286
    y-coordinate: 617.599831
distance from start: 925.257030


heuristic cost: 0.000000
given cost: 1645.714286
final cost: 1645.714286
number of nodes created: 229
number of open nodes: 63
size of solution path: 59
number of iterations: 191
nodes constructed: 229
nodes destructed: 0
waypoints constructed: 0
waypoints destructed: 0
```

The result of this search is not much more costly than an optimal solution, while the number of tiles explored is substantially reduced.

Be aware that as you raise the heuristic weight to a very large value, the behavior of A* starts to approximate that of the greedy-search algorithm.

## Algorithmic Outline

Here's a sketch of how A* would work over tiles. Instructions highlighted in green indicate additions to the uniform-cost search outline.

```
1. Determine the SearchNode that represents the start tile.
2. Set both its heuristicCost and its finalCost equal to the distance
     between the start and goal tiles.
3. Enqueue this node into the open container.
4. Add this node to the visited map.
5. While the open container possesses nodes, do the following:
    1. Dequeue the current node from the open container.
    2. If this node is at the goal tile, build the solution list and exit.
    3. Otherwise, for each successor of current:
        1. Compute the newGivenCost of the successor.
        2. If successor is not a duplicate:
            1. Set its heuristicCost equal to the distance between its tile
                 and the goal tile.
            2. Set its givenCost equal to the newGivenCost.
            3. Set its finalCost equal to the sum of the two costs.
            4. Enqueue this node into the open container.
            5. Add successor to the visited map.
        3. If successor is a duplicate, but newGivenCost is cheaper than its
            givenCost:
            1. Set its givenCost equal to the newGivenCost.
            2. Set its finalCost the same way you would if successor were
                 not a duplicate.
            3. Update the path to successor.
            4. Update the open container.
```

### 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 from a lab specialist. 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. Most of these questions reference other material that you should also keep handy.

## How Do I...

Q: How do I use this STL container / PriorityQueue / other object I've never seen before?
A: For a brief STL overview, visit the section on STL containers. For more details on the other core components of the search algorithms, read the API documentation accompanying this manual. It contains two major sections: one for the STL containers, and one for the AI-specific components. Each class has a separate page associated with it.

Q: How can I use the given-cost formula to compute the given cost of a successor node that may or may not yet exist?
A: First of all, do not create an entire `PlannerNode` object just to store a cost value. Second, even though you cannot directly access node attributes such as `node->parent` or `node->tile`, you should be able to figure out who they ***will*** be:

- You have direct access to row and column information, from which you can access `Tile` objects.
- For any two nodes such that the first node will be a successor of the second node, the second node should therefore become the parent of the first node.

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.

1. 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.
2. For uniform-cost search, the given cost should also match.
3. For A*, the given and final costs should match.
4. For all algorithms, the number of visited nodes should not differ by more than ten percent (10%).
5. Don't forget to test your reset() methods by pressing the “Stop” button.
6. 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 search algorithm is stuck in an “infinite loop”. Why does it keep revisiting certain 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 blasted 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 search 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. What happened?
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 does my search algorithm leave plenty of open nodes all over the place?
A: Either your comparison function(s) may be checking the wrong cost, or your algorithm is computing the right cost incorrectly for each node.

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 mysterious `0xcccccccc` or `0xcdcdcdcd` 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.

## Code Conventions

The lab grader will enforce some/all of these conventions by penalizing any violations.

- Do not change the public, protected, or friend interfaces of an existing class.
- If you absolutely have to add helper methods or variables, you must place them in private scope (not protected scope), and you must preserve const correctness.
- If you plan to declare helper function prototypes outside a class body, choose one of these options instead:
- You may define such functions in the source (.cpp) file where they are needed.
- You may declare them as private static methods.
- Do not const_cast. This implies a deficiency in the project API. File a bug report instead.
- Do not C-style cast in place of a const_cast.

## Grading

Your grade on the lab is calculated as follows:

- 50% correctness (based on percentage of test cases passed)
- 50% speed, according to this scale:
  - Student speed is faster than FullSail’s:  Speed grade is 100%, plus extra credit proportional to speed gain
  - Student speed is 100% – 130% of FullSail’s:  Speed grade is 100%
  - Student speed is 130% – 230% of FullSail’s:  Speed grade is 100% to 0% (percentage - 130%)
  - Student speed is > 230% of FullSail’s:  Speed grade is 0%

For speed purposes, each test case is run 100 times and then averaged.
Memory leaks may be penalized at the discretion of the grader.

## Deliverables

Students will submit the completed PathSearch project.

### Deliverable File(s) and Contents

You will upload a compressed (Zipped) file named
`<lastName>.<firstName>.PathPlannerLab.zip` which should include:

```
PathSearch (folder)
        \
        PathSearch.cpp
        PathSearch.h
        PathSearch.vcxproj
        PathSearch.vcxproj.filters
```

**Turn in Check List:**

- Make sure your project compiles and runs in Release mode using the double arrow button. No pop ups of any kind allowed (if any pop ups occur, that means there's an error in your algorithm, which must be fixed).

- Check your output using ALL of the test cases. Otherwise, you may miss some bugs that do not show up on the base map. If your program crashes in our grading suite because of a hidden bug (with code that receives a non-zero grade), you must fix it, resubmit, and receive late penalty charges. If you have tested all 25 cases (in Debug and Release) and get a grade of 100%, you will be guaranteed this full grade, even if our grading suite finds unforeseen errors later.

- Zip up the PathSearch folder with the files listed above. When we unzip your lab zip file, it should have this structure exactly, and not with another layer of folder in between. If you have created extra Cpp or Header files (which are generally not necessary), they need to be alongside the existing files within this folder.

- If you have included "vld.h" for memory leak check purposes, please remove or comment it out before turning it in. Otherwise, it will be counted as a late turn-in/resubmission, because it will not compile when we attempt to grade it. We will have to manually edit your files and regrade, and give a late penalty charge.

- Re-download your submission, and make sure it runs as you have expected.