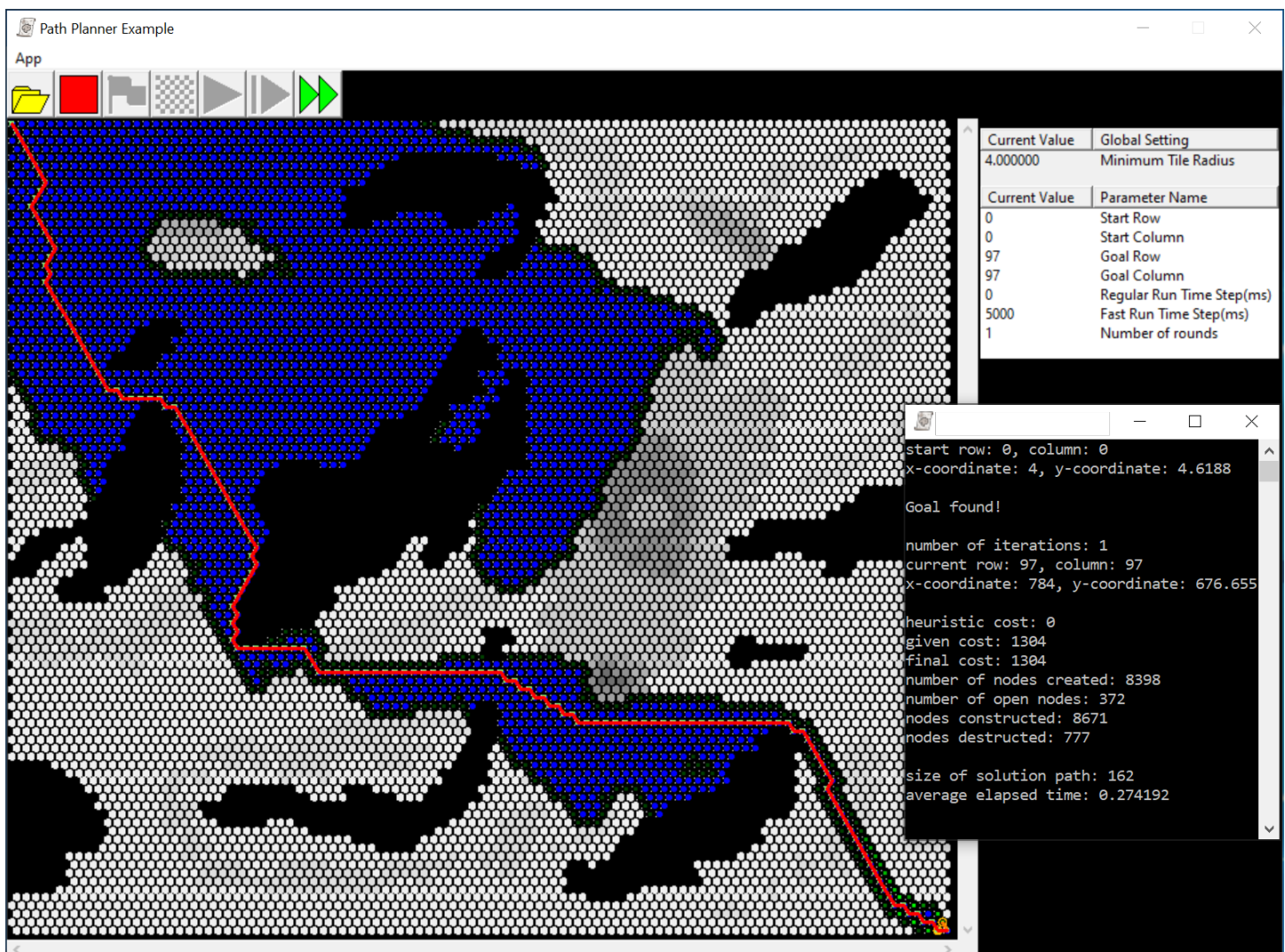# Manual: Path Planner

Overview

In this project, students will implement the search algorithm A* in the context of a tile-based path planner similar to those used in most tile-based games. When the search algorithm is completed, the user should be able to configure the algorithm by changing the parameters in the user interface (UI) and run the algorithm stepwise and / or to completion. The UI code and drawing interface are provided to students; students are expected to implement the search itself by conforming to this specification.

We strongly recommend that students briefly read the manual before beginning the project! It is important to have a holistic understanding of the application and the sections to be written before the student begins work.

Structure

The project should be broken into several smaller steps:

1) Creating a <u>search graph</u> to represent the **TileMap**, which in turn represents the terrain.
2) Writing of a declaration for the **PathSearch** class and initial (likely empty) methods for compilation.
3) Development of a visual debugging strategy using the drawing functions of the **TileMap**.
4) Implementation of a <u>Breadth-First Search</u>, then modifying it into a <u>Greedy Best-First Search</u>.
5) Modification of the Greedy Best-First Search into a <u>Uniform-Cost Search</u>, and then into an <u>A* Search</u>.
6) Optimization of the A* algorithm to meet speed / benchmark requirements.

User Interface (UI)

The UI for the path planner is provided for students so that they may focus on implementing the search algorithm itself within the existing code base (as is common in the game and other industry sectors). Upon launch, the planner will load the default map and run the search algorithm's initialization routine. For example, the search implementation might build and display a search graph upon initialization (**Figure 1**).
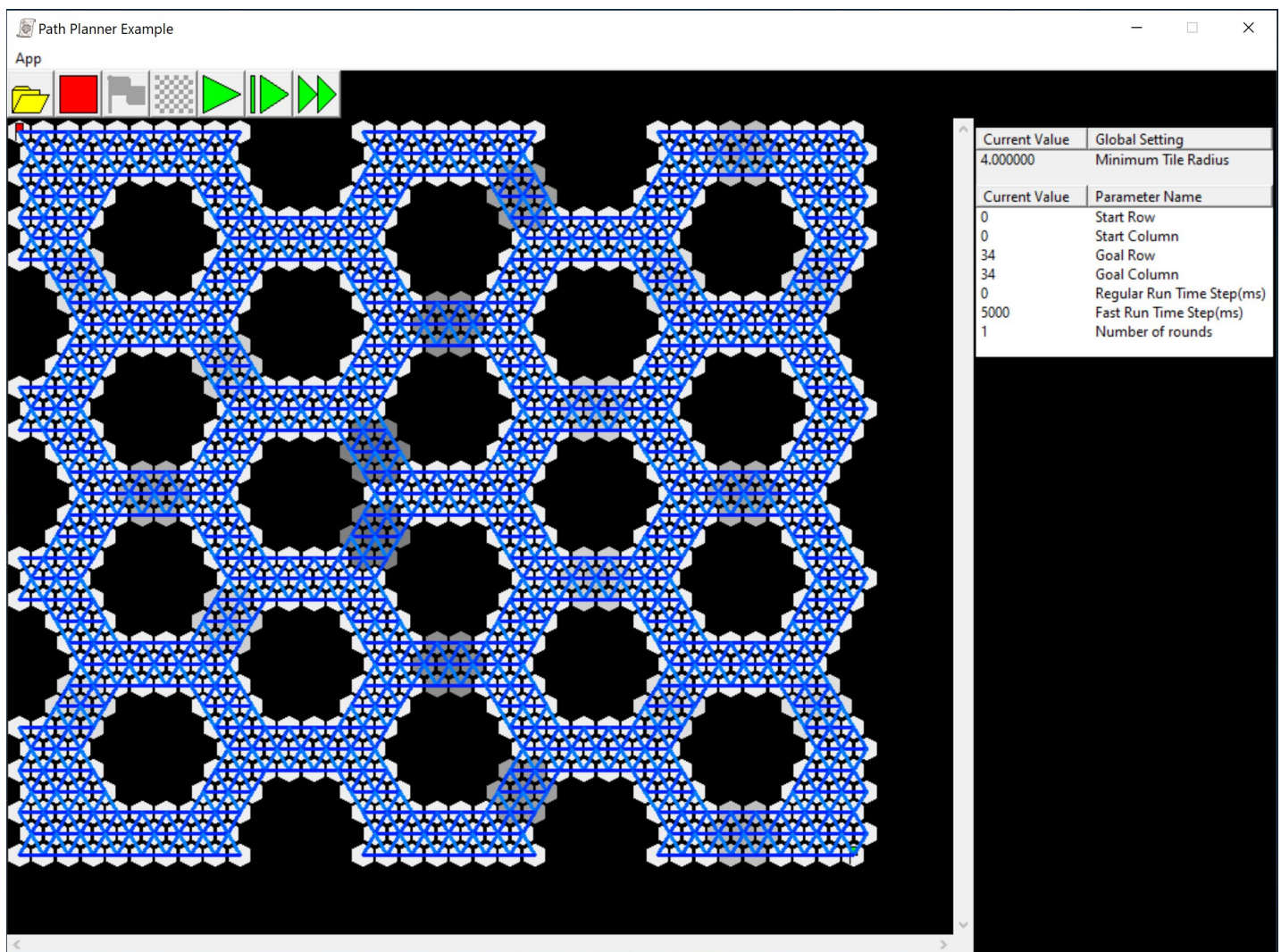
Figure 1. Path planner application upon initial load, displaying a search graph built by the search algorithm.

Time Map Display

The tiles that represent the terrain are displayed, along with any drawing engaged by the search algorithm, in the Tile Map Display (bottom-left). Heavier terrain weights are represented by darker tiles, while black tiles represent untraversable terrain. The red flag represents the starting point and the green flag the goal location.

Button Controls

The Button Controls across (top) the user to control the search as identified (**Table 1**).

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.

Table 1. Search Application Controls

Settings Pane

There are also a several that can be changed in the Settings pane (upper-right). These include the starting and goal locations, as well as the time limit for a single step or full algorithm run. Finally, for full algorithm runs, the number of times to run the search can be adjusted up from one to determine timing performance. *Note that the minimum tile radius is for display only and cannot be changed.*

State Space

Some game engines – including the planner in this project – use hexagonal tiles, as they are equidistant from their neighbors (unlike square tiles, which have a different distance to neighbors in cardinal directions compared to diagonal neighbors). This simplifies and speeds up path calculations. However, this makes storing and retrieving tiles in a typical 2D array less straightforward. An example is outlined below (**Figure 2**). Each tile also holds a terrain weight which identifies the cost of trafersing through a tile.
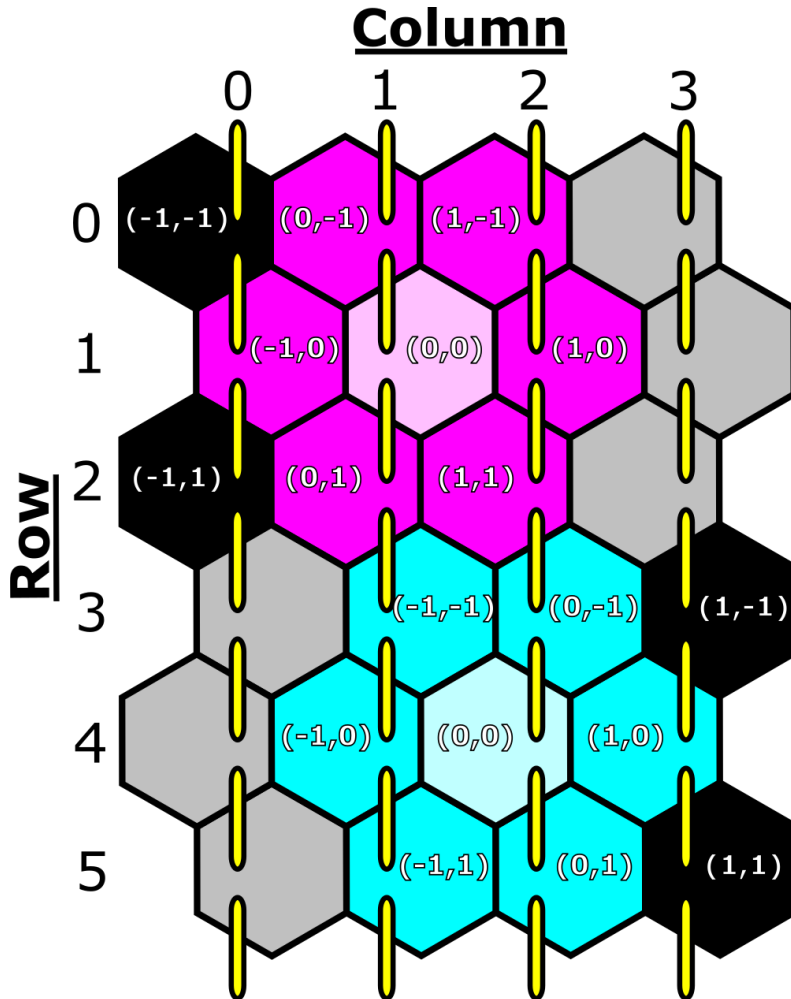
Figure 2. Layout of hexagonal grids (rows and columns).

Figure 3. How tile information is stored in an array.

In this diagram, the dashed yellow lines separate the tiles into logical columns. The offset from the current tile (0, 0) to its neighbors is marked; odd rows have offsets as outlined **MAGENTA** and even rows have offsets as outlined in **CYAN**. **BLACK** tiles indicate offsets that are invalid depending on the odd or even row location of (0,0). **Note that these are not the same as the obstacle tiles in the tile map**. We can see how these data would be stored in a 2D array (**Figure 3**). For any tile whose row coordinate has an odd value, the tiles that are adjacent to it follow the pattern of the magenta tiles. For any tile whose row coordinate has an even value, the tiles that are adjacent to it follow the pattern of the cyan tiles.

It is recommended that students build a private helper method in their search class to determine the adjacency of *Tile* objects to simplify identification of neighboring locations, e.g.:

```
private bool areAdjacent(const Tile *lhs, const Tile *rhs)
```

Development Interface

The development interface includes several elements 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 are provided 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 holding luminosity (**Table 2**, **Figure 4**).

	Lum.	Red	Grn	Blue	Hex
	255	0	255	0	0xFF00FF00
	127	0	255	0	0x7F00FF00
	255	127	0	0	0xFF7F0000
	0	0	0	255	0x000000FF

Table 2. Example colors in the LRGB color model.
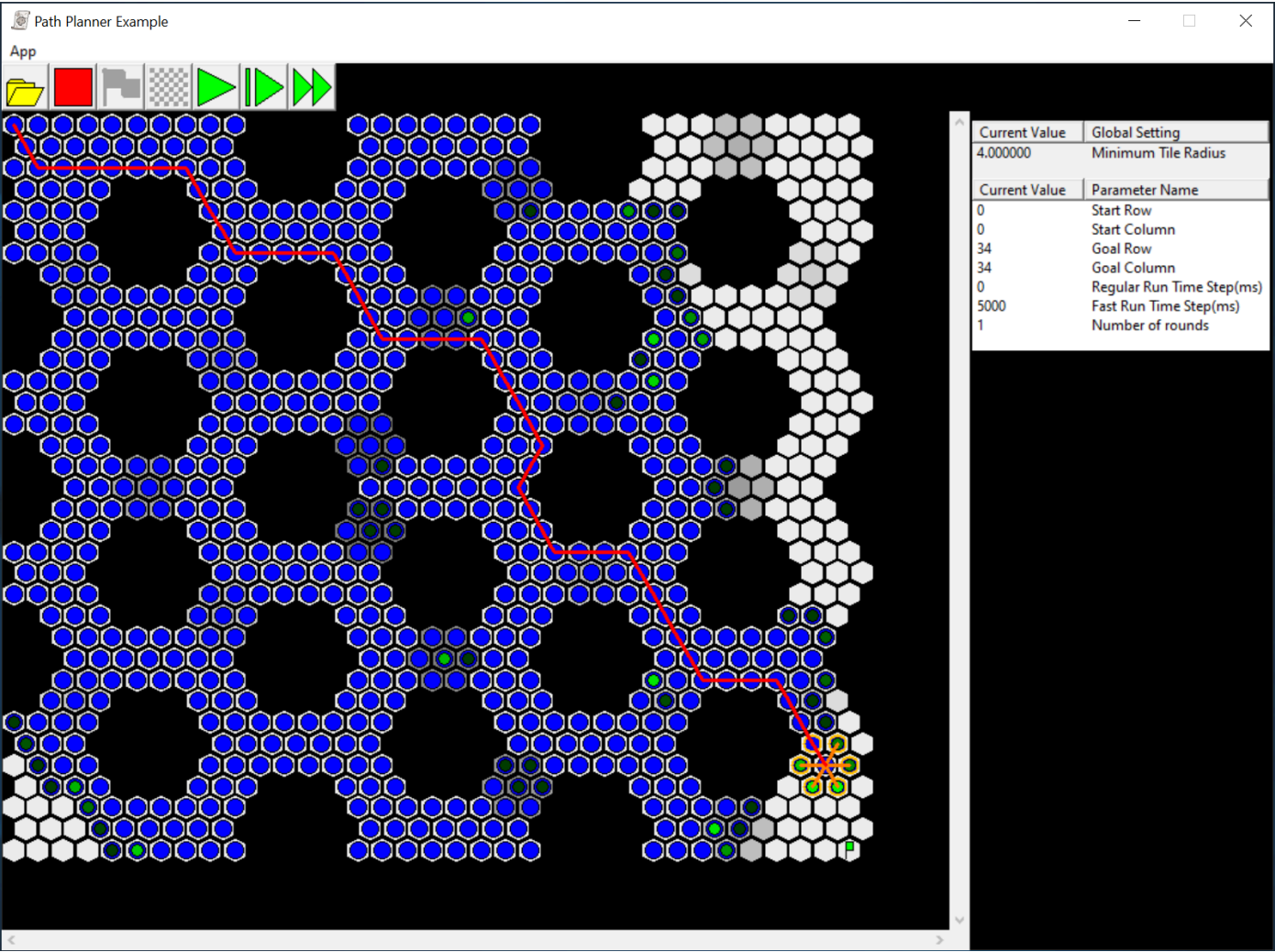
Figure 4. Example of colors in use in the Path Planner application.

Developer API (Provided)

The following programming interface is provided so students can implement the search algorithm. Note: students **may not change** the API provided; student searches must function properly <u>without any API changes</u>.

TileSystem

These classes are in the `ufl_cap4053::searches` namespace and the `Framework/TileSystem` directory.

TileMap

public void **resetDrawing**()
Resets (wipe) all drawing on the tiles in this map.

*public Tile** **getTile**(int row, int column) *const*
Returns a pointer to the *Tile* at the designated **row** and **column**.

public int **getRowCount**() *const*
Returns the number of rows of tiles in this map.

public int **getColumnCount**() *const*
Returns the number of columns of tiles in this map.

public double **getTileRadius**() *const*
Returns the length of the radius of tiles in this map.

Tile

public void **resetDrawing**()
Resets all drawing on this tile.

public void **addLineTo**(*Tile** destination, *unsigned int* color)
Sets up a line to be drawn from this *Tile* to **destination** usintg the designated **color**.

public void **setFill**(*unsigned int* color)
Sets this *Tile*'s fill color to **color** in the 32-bit LRGB space (luminosity, red, green, and blue).

public void **setMarker**(*unsigned int* color)
Sets this *Tile*'s marker color to **color** in the 32-bit LRGB space (luminosity, red, green, and blue).

public void **setOutline**(*unsigned int* color)
Sets this *Tile*'s outline color to **color** in the 32-bit LRGB space (luminosity, red, green, and blue).

public int **getRow**() *const*
Returns this *Tile*'s row in the map.

public int **getColumn**() *const*
Returns this *Tile*'s column in the map.

public int **getXCoordinate**() *const*
Returns this *Tile*'s x-axis coordinate on the map.

public int **getYCoordinate**() *const*
Returns this *Tile*'s y-axis coordinate on the map.

<u>NOTE</u>: users **should not** create any tiles or tile maps; instead, they should use references to existing objects.

This class resides within the **ufl_cap4053** namespace and in the root of the source directory.

PriorityQueue<T>

public **PriorityQueue**(bool (*greaterThan)(*const* T &lhs, *const* T &rhs)
The constructor; creates an object that holds type *T*. Parameter is a function pointer **greaterThan** to a function that compares two values of type *T*, where **greaterThan(lhs, rhs)** returns **true** if **lhs** should be sorted after **rhs** in the queue and **false** otherwise.

public bool **empty**()
Returns **true** if the container has no elements in it, and **false** otherwise.

public void **clear**()
Removes all elements from the container.

public size_t **size**()
Returns the number of elements currently in the container.

public void **push**(*const* T &element)
Pushes **element** into the container; the container maintains sorting / ordering during this operation.

public T **front**() *const*
Returns the lowest value (front) element from the container; <u>does not remove the element.</u>

public void **pop**()
Removes the lowest value (front) element from the container; <u>does not return the element.</u>

public void **remove**(*const* T &element)
Removes all instances of **element** from the container.

public void **enumerate**(std::vector<T> &sorted) *const*
Enumerates all of the elements in the **sorted** container <u>from greatest to least.</u>

Figure 5 is example of how the *PriorityQueue* type might be used and function in code:

```
#include <iostream>
#include <vector>
#include "PriorityQueue.h"

bool isGreaterThan(const int &lhs, const int &rhs)
{
    return lhs > rhs;
}

int main()
{
    ufl_cap4053::PriorityQueue<int> pQ(isGreaterThan);
    pQ.push(4);
    pQ.push(2);
    std::cout << pQ.front() << std::endl;

    std::vector<int> output;
    pQ.enumerate(output);

    for (int value : output)
        std::cout << value << std::endl;
}
```

Figure 5a. Example of PriorityQueue class use in code.

```
2
4
2
_
```

Figure 5b. Behavior of PriorityQueue.

NOTE: Due to the constructor for *PriorityQueue* requiring a parameter, to create a *PriorityQueue* object as a member variable within another class, it will need to be initialized in an **initializer list** in the constructor of the parent class!

Assignment API (TODO)

Students will complete the *PathSearch* class according to the specification outlined here <u>exactly</u>. This class **must reside** within the `ufl_cap4053::searches` namespace.

PathSearch

public **PathSearch**()
The constructor; takes no arguments.

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

public void **load**(TileMap* _tileMap)
Called after the tile map is loaded. This is usually where the search graph is generated. *NOTE*: This method must take no longer than twice the benchmark example for any given map.

public void **initialize**(int startRow, int startCol, int goalRow, int goalCol)
Called before any update of the path planner; should prepare for search to be performed between the tiles at the coordinates indicated. This method is always preceded by at least one call to **load()**.

public void **update**(long timeslice)
Called to allow the path planner to execute for the specified **timeslice** (in milliseconds). Within this method the search should be performed until the time expires or the solution is found. If **timeslice** is zero (0), this method should only do a single iteration of the algorithm. Otherwise the update should only iterate for the indicated number of milliseconds. This method is always preceded by at least one call to **initialize()**.

public void **shutdown**()
Called when the current search data is no longer needed. It should clean up any memory allocated for <u>this search</u>. Note that this is **not** the same as the destructor, as the search object may be reset to perform another search on the same map. <u>May be called before **initialize()** and/or **update()** in some cases.</u>

public void **unload**()
Called when tile map is unloaded and should clean up any memory allocated for the tile map. This is not the same as the destructor, as the search object may be reinitialized with a new map. <u>May be called before **load()**.</u>

public bool **isDone**() *const*
Returns **true** if the update function has finished because it found a solution, and **false** otherwise. Once a search has completed, this method should continue to return true until **initialize()** is called.

public std::vector<Tile const> const* **getSolution**() *const*
Return a *vector* containing the solution path as an ordered series of *Tile* pointers from finish to start. Once a search has been completed, this method continue to return the path until **initialize()** is called.

Execution Frames

As the search algorithm must be split across multiple execution frames, students will need to identify in which method tasks will be accomplished. Below is one example of how a search might be constructed in this way.

load() – Create search graph from the tile map by identifying adjacent tiles.
initialize() – Set starting and ending points for the search, adding the starting point to the search queue.
update() – Run body of algorithm, looping over queue, until time is expired, saving state for next run.
shutdown() – Reset search's member variables so that another search <u>on the same terrain</u> can be run.
unload() – Free all memory associated with the tile map (i.e., the search graph).

Submissions

Students will submit a **zip file** containing the following files at the end of this exercise on Canvas:

- PathSearch.cpp
- PathSearch.h

Place them in the **root directory** of your zip file, not in a subdirectory. Do not submit any other files.

Each test case will be checked for memory leaks! If a test case leaks memory, it will penalized by up to 30%, so if you leak a lot of memory on every test case, your maximum grade penalty will be 30% overall.

Sample Tests

It is recommended that students test using these samples as well as developing a list of **their own unit tests and test cases** in order to thoroughly test their algorithm. Please note that projects will be tested using separate, unique data that follows the specification, so <u>self-testing is crucial to proper function</u>.

Tile Map	Start Row	Start Col.	Goal Row	Goal Col.
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