CO2301 / CO3519 Assignment - Path Finding

Deadline: Saturday 25th January 2020, 11.59pm

Coursework counts **70**% towards the overall mark for **CO2301**, with the exam counting 30%.
Coursework counts **50**% towards the overall mark for **CO3519**, with the exam counting 50%.

Introduction

This assignment requires you to implement one or more path finding algorithms, with the option of providing a visual representation. There are two different routes permitted:

Route A. This route asks you to implement the **breadth-first** and **depth-first** algorithms.

- If you choose this route **you cannot gain more than a bare pass mark of 40%**.
- If you follow this route, then you do not need to implement a visual implementation.
- Essentially I am asking you to complete Exercise 6 and 7 from the pathfinding Worksheet.

Route B. This route asks you to implement the **A*** algorithm and **at least one other** algorithm.

- Your implementation of the algorithms does not have to be perfect in order to *pass* the assignment. It can be possible to pass the assignment with errors.
- You need to implement your algorithm(s) as part of a **framework**.
  - I have given you some code as a starting point. (see explanation later…)
  - Your algorithms must be implemented in classes derived from the **interface** supplied.
- Marks above 60% are available for the following (in increasing difficulty):
  - Implementing a graphical output from your algorithm(s) (using TL-Engine).
  - Displaying the progress of the search in real-time.
  - Applying path smoothing to your generated path.
  - Setting the target location and calling the search interactively.
- Marks will be lost for:
  - Significant errors.
  - Inefficient implementation of the algorithm(s).

This is an **individual** project and group work is not permitted.

Deliverables, Submission and Assessment

- Upload the following files to Blackboard by the deadline:
  - The **C++ source**(s) for your solution
  - Your **text output** files for each algorithm
  - Any **model or texture files** you use which are NOT supplied with the TL-Engine
  - **IF** you have implemented a graphical solution, also submit:
    - A word file containing labelled **screen shots** for each completed map.
  - **IF** you have implemented a **real-time** visual representation
    - A link to a video of your solution in action (e.g. on YouTube)
- Make sure your **name** is included as a comment at the top of EVERY source code file.
- All files should be sensibly named, and all code should compile and run without errors.

- **Assessment will be by demonstration either in the lab, or at a special session to be arranged. Nearer the deadline, there will be the opportunity to sign up for a specific time slot to demonstrate.**

- Unless you have an extension (agreed in advance) or extenuating circumstances, then your work will be treated as **late** if you are not prepared to demonstrate in the lab following the submission deadline.
  - If you have a problem, then you need to formally request an extension or submit extenuating circumstances.

Assignment Details

File Input and Output

- All files should be read *relative* to the current working directory.
- Two associated sets of map files have been provided initially:
    - **mMap.txt** and **dMap.txt**
    - The map files contain multiple lines of text.
        - The first line of text is the dimensions of the map: **x y**, with a space in-between.
        - Subsequent lines give the terrain costs of each map square
    - These maps are both 10 by 10 in size, but for marks above 40% your solution should accept maps of any size
- Your program must read the start and goal coordinates from the files provided:
    - **mCoords.txt** (for use with mMap) and **dCoords.txt** (for use with dMap).
    - The coordinates file contains two lines of text.
        - The first line gives the coordinates of the start location: **x y**, with a space in-between.
        - The second line gives the coordinates of the goal location: **x y**, with a space in-between.

- Your program should work for **ANY** map (following the naming convention *x*Map.txt etc)
    - Your program should **NOT NEED TO BE RECOMPILED** to load different maps!
    - Ideally, you should only need to type in the first letter of the map/coords file – i.e. m or d

- The results of the search should be written to an output file named **output.txt**.
    - This file should **only** contain the route of the selected paths
    - The route should be presented as a list of all of the coordinates of the route **in order**.
        - It is *easier* to print out the route in reverse order from goal to start. However, you will obtain **better marks** if you print out the route from the start position to the goal position including all of the locations in between.
        - The route should include the coordinates of the start and end squares.
- If you implement the A* algorithm, you must **keep a count of how many times the open list gets sorted**, and **output the total** somewhere (to console or file) at the end of the search.

Coordinates and Movement

- You should use a square grid.
    - Movement is only allowed along the horizontal and vertical – not diagonal
        - i.e. North, South, East and West.
    - Each grid reference is identified by an x and y co-ordinate.
    - The x axis is positive to the right and the y axis is positive upwards.
    - The coordinate numbering starts at 0. (see the figure overleaf)
- **You will lose marks** if you do not adhere to the coordinate system provided.
- For the purposes of this assignment, the heuristic is simply the Manhattan distance to the goal.

User Interaction

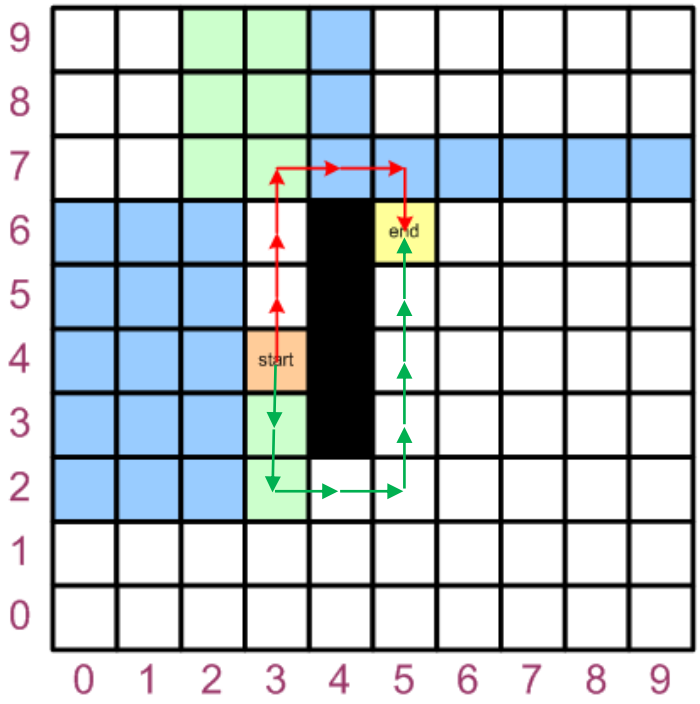
- I do not want to have to keep pressing keys in order to advance the search.
    - **You will lose marks** if you require unnecessary user input.
- After the search begins, the program should produce the final route without requiring any further user input.
    - The only exception to this is that **if** a visual representation of the search is provided, then you *may* want pause execution to show the search expanding.
    - You are not required to, but if you do then do it sensibly – i.e. pause the tree ply-by-ply, not node-by-node.

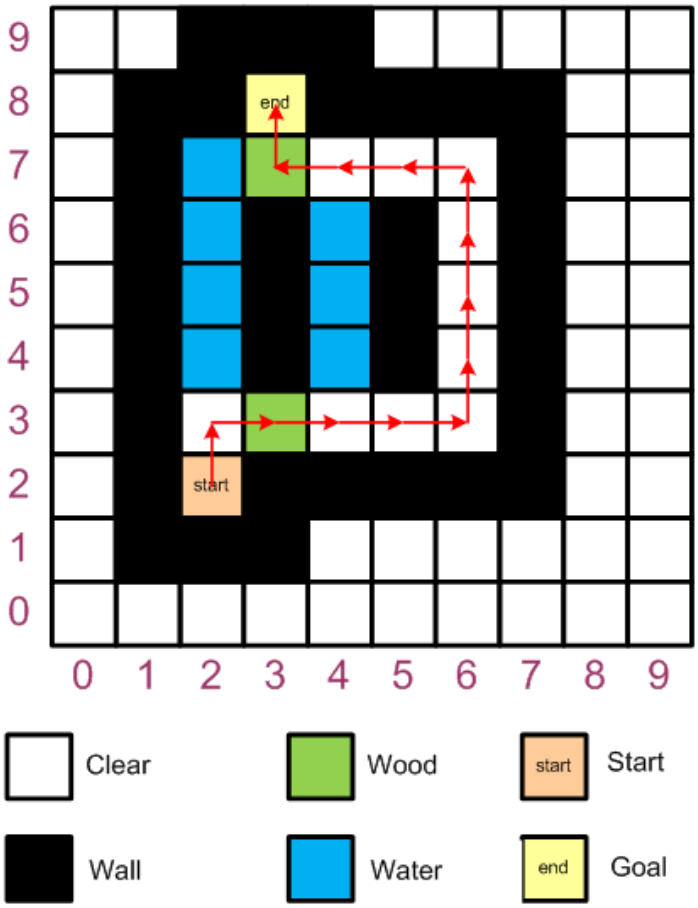
Program Architecture

- For a mark above 40% you should use the type definitions, factory function and interface class supplied.
    - See later section for an explanation of how to use the sample code.

Example Maps and expected routes

mMap (*above*) and dMap (*below*)





Movement costs
All of the algorithms use a map with a wall.

A* requires a map with terrain costs as follows:

- Clear 1
- Wood 2
- Water 3
- Wall 0

Therefore, the A* algorithm will choose the path shown in green arrows that goes south of the wall on the mMap (top map) rather than the shortest route shown in red arrows (which would be found by e.g. breadth first).

Path Finding Algorithm Implementation

- You must implement the **algorithm as provided** on the module page.
- You must use **STL container classes** to implement the algorithm.
- You must implement the assignment in **C++** using **smart pointers**.
- You will need to implement **map reads from file**.

Route A: *Breadth-first* and *depth-first algorithms for a maximum mark of 40%*

- Do not attempt this section if you want to access the full range of marks.
    - The task is basically to complete Exercises 6 and 7 from the lab worksheet.
- Implement a breadth-first search.
- Implement a depth-first search.
- You must include backtracking and avoid repetition. Use the Manhattan distance between the goal and the current location for your heuristic.
- You are being graded on:
    - Correct implementation of the algorithm.
    - Use of object-orientation.
    - Use of STL.
    - Map read.
    - Code quality, e.g. meaningful names, well commented and good style.

Route B: *A* and one other algorithm for a mark potentially above 40%*

- You must implement each algorithm in its own class **derived from the interface provided**.
- Implement an A* search and at least one other search of your choice. You must include backtracking and avoid repetition. Use the Manhattan distance between the goal and the current location for your heuristic.
- Test your algorithm with other maps. I may use a selection of other maps to mark your work.

Code Style and Layout

- Your code needs to adhere to Gareth Bellaby's style guide.
- The style guide can be found inside the CO2402 Advanced C++ module page on Blackboard.

Visual representation using the "output.txt"

- Use the **TL-Engine** to implement a visual representation of the searches. You will not be graded on the underlying graphics code but the clarity of the visual representation and successful execution of the program.
- The search algorithm should run in its entirety. After the algorithm has run, use the output file to produce a visual representation of the progress of the search.

Advanced: real-time visual representation

- Implement a real-time visual representation of the algorithms.
- The visual representation will be updated as the search progresses.
- You will need to add a method to the interface class which performs one "step" of the algorithm to achieve this.

More advanced: spline applied to movement on map

- Move an actual model over the map following your generated route.
    - For better marks, the model should turn and face the direction of travel.
- Smooth the chosen path using a spline. A Bézier spline will probably produce the best effect.

Really advanced: Use character location and/or user input to determine start and goal nodes

- This could involve selecting a grid location as the new goal, or ultimately, using a player controlled character to trigger path finding in a scenario such as a simple thief-style game...

Using the Sample Code

The code I have supplied is designed to help you build your program architecture.

I want you to separate the pathfinding functionality from the main program, so you have a re-usable library. Also, I want you to make your path finding polymorphic, so that the same main program code can use different search algorithms just by changing one word.

Basically, it works like this:

There is a base class `ISearch` (defined in **Search.h**) whose functions are all *pure* virtual (so have no bodies), which means you can't instantiate objects of that class. It does, however, give the search function declaration (signature). This kind of class is called an **interface**.

You need to provide your own class (or, ideally more than one) which inherits the interface class, and implements the search functionality using your chosen algorithm(s). One class per algorithm.

I have provided you with an example of such a derived class `CSearchNoStar` (defined in **SearchNoStar.h** with method implementations in **SearchNoStar.cpp**) – although the "no star" algorithm I have implemented simply returns a fake path.

The framework uses a *factory function* called `NewSearch` (defined in **SearchFactory.h**). Associated with the factory function is an enumerated type `ESearchType` which specifies the different implementations available. By passing the desired search type into the factory function, the appropriate `CSearch` object is created, and a raw pointer to that object is returned. Note that the *type* of the pointer is pointer-to-the-base-class – i.e. `ISearch*`. This is the mechanism by which polymorphism works in C++… allowing a base class pointer to point to derived class objects. When the main program code dereferences the pointer, it calls the derived class code for the actual object.

Along with the factory function, interface and example derived class, I have given you some type definitions in **Definitions.h** as follows:

`SNode`	a structure comprising coordinates, a score and a raw pointer to a parent node
`NodeList`	a deque of unique node pointers
`ETerrainCost`	an enumerated type to define cost of terrain types
`TerrainMap`	a 2-dimensional vector to store the map data

All of these files need to be added into your project. It doesn't matter whether the project has been created by the TL-Engine wizard or is just a standard Windows console application.

In your main program, you will need to include the interface, factory and type definitions headers:

```
#include "Search.h"
#include "SearchFactory.h"
#include "Definitions.h"
```

Instantiate a search object as follows, eventually substituting NoStar for one of your own search types: (this should be familiar from using the TL-Engine, which works the same way)

```
ISearch* PathFinder = NewSearch(NoStar);
```

Create a map object and unique pointers to start and goal nodes

```
TerrainMap myMap;
unique_ptr<SNode> start(new SNode);
unique_ptr<SNode> goal(new SNode);
```

You will need to read from the map file to set the size of the terrain map and fill in its details. Similarly the corresponding coordinates file will allow the details of the start and goal nodes to be filled in.

Finally, you can create a path, and pass it to the search object's FindPath function to be filled in:

```
NodeList path;
bool success = PathFinder->FindPath(myMap, move(start), move(goal), path);
```

If you use the code snippets above with the libraries supplied, everything should compile and run!

You will be able to inspect the returned (incorrect) path with the debugger, but the first thing you will want to do is make a display function. I suggest you make a "utilities" library with a separate header and cpp file for this sort of thing. Later on, it can contain other useful things like the comparison function for the nodes (required for priority queue as part of A*) etc.

We won't be covering the material on Program Architecture in the lectures until Semester 2. Currently it is scheduled on GamesNorthWest.net for Week 18, though I intend to bring it forward by a few weeks. There is a worksheet for that week which you may find useful, and I have deliberately structured the framework code supplied with this assignment in the same way as that worksheet does it.

I have also made a 20 min. video going through what is needed to complete that worksheet.
https://vls.uclan.ac.uk/Player/16384