RMIT UNIVERSITY

Advanced Programming Techniques

Assignment 1 | Implementing a Path Planning Algorithm

Assessment Type	Individual Assessment. Clarifications/updates may be made via announcements/relevant discussion forums.
Due Date	11.59pm, Sunday of Week 5
Weight	30% of the final course mark
Submission	Online via Canvas. Submission instructions are provided on Canvas.

1 Overview

In this assignment you will implement a simplified algorithm for **Path Planning**, and use it with a simple simulated 2D robot moving around a room.

In this assignment you will:

- Practice the programming skills such as:
  - Pointers
  - Dynamic Memory Management
  - Arrays
- Implement a medium size C++ program using predefined classes
- Use a prescribed set of C++11/14 language features

This assignment is marked on three criteria, as given on the Marking Rubric on Canvas:

- **Milestone 1:** Writing Tests.
- **Milestone 2-4:** Implementation of the Path Planner.
- **Style & Code Description:** Producing well formatted, and well documented code.

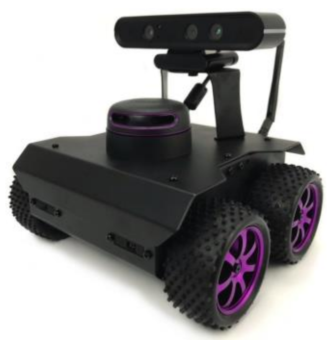

Figure 1: An example robot

1.1 Relevant Lecture/Lab Material

To complete this assignment, you will require skills and knowledge from workshop and lab material for Weeks 2 to 4 (inclusive). You may find that you will be unable to complete some of the activities until you have completed the relevant lab work. However, you will be able to commence work on some sections. Thus, do the work you can initially, and continue to build in new features as you learn the relevant skills.

1.2 Learning Outcomes

This assessment relates to all of the learning outcomes of the course which are:

- Analyse and Solve computing problems; Design and Develop suitable algorithmic solutions using software concepts and skills both (a) introduced in this course, and (b) taught in pre-requisite courses; Implement and Code the algorithmic solutions in the C++ programming language.
- Discuss and Analyse software design and development strategies; Make and Justify choices in software design and development; Explore underpinning concepts as related to both theoretical and practical applications of software design and development using advanced programming techniques.
- Discuss, Analyse, and Use appropriate strategies to develop error-free software including static code analysis, modern debugging skills and practices, and C++ debugging tools.
- Implement small to medium software programs of varying complexity; Demonstrate and Adhere to good programming style, and modern standards and practices; Appropriately Use typical features of the C++ language include basic language constructs, abstract data types, encapsulation and polymorphism, dynamic memory management, dynamic data structures, file management, and managing large projects containing multiple source files; Adhere to the C++14 ISO language features.
- Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.

2 Introduction

One challenge in robotics is called *path planning*. This is the process of the robot figuring out how to navigate between two points within some environment. In this assignment you will implement a simplified path planning algorithm for a robot moving about a simple *2D environment* - a rectangular room with obstacles.

We will represent a simple 2D environment as a grid of ASCII characters. For example:

```
==========
=S.......=
========.=
=......=.=
==.=.=...=
=..=..=.==
=.===.=..=
=..==G==.=
===.======
==========
```

Aspects of the environment are represented by different symbols:

Symbol	Meaning
= (equal)	Wall or Obstacle within the environment. The robot cannot pass obstacles
. (dot)	Empty/Open Space.
S	The start position of the robot
G	The goal point that the robot is trying to reach.

Each location in the environment is indexed by a (col,row) co-ordinate. The top-left corner of the environment is always the co-ordinate (0,0), the col-coordinate increases right-wards, and the row-coordinate increases down-wards. For the above environment, the four corners have the following co-ordinates:

```
(0,0) . . (9,0)
  .         .
  .         .
(0,8) . . (9,8)
```

Two dimensional environments can be designed in different ways. For this assignment we will use environments where:

1. There is one goal (ending) point denoted by character "G".
2. The environment is *always* surrounded by walls.
3. The environment can contains *junctions*, *corridors* "open" space, loops or islands.

For the purposes of this assignment we will make several important assumptions:

- The robot knows the map of the whole environment it is in at the start.
- Robot can only be located at cells marked as empty/open spaces.
- It may not be possible for the robot to reach every empty space.
- The robot can move to any one of 4 cells, that are to the left, right, up, or down from the robots originating cell. Robot **cannot** move diagonally.
- For this assignment the direction the robot is "facing" is ignored.

In this assignment, the path planning problem is divided into two parts:

1. *forward search*: Conduct a forward search algorithm starting from the "start", until the goal is reached. The pseudo-code for the algorithm you need to implement is provided to you in Section 3.2.1.
2. *backtracking* : Move backwards from the "goal" using, the results of *forward search*, to identify the *shortest path* from the start to goal. You will have to develop the pseudo-code for this part. A description of the algorithm you need to implement is provided to you in Section 3.3.1.

While there are many ways to navigate a 2D environment, **you must implement the algorithms provided in this document**. If you don't, you will receive a NN grade.

3 Assessment Details

The task for this assignment is to write a full C++ program that:

1. Reads in a 20x20 environment from standard-input (std::cin).
2. Finds the robot's starting position within the environment.
3. Executes the *forward search* algorithm (Section 3.2.1) until the robot reaches the goal.
4. Executes the *backtracking* algorithm (Section 3.3.1) to find the shortest path.
5. Prints out the environment and the path to be followed by robot to standard output (std::cout).

You may assume that the environment is a fixed size of 20x20, except for Milestone 4.

This assignment has four Milestones. To receive a PA/CR grade, you only need to complete Milestones 1 & 2. To receive higher grades, you will need to complete Milestones 3 & 4. Take careful note of the Marking Rubric on Canvas. Milestones should be completed in sequence.

3.1 Milestone 1: Writing Tests

Before starting out on implementation, it is good practice to write some tests. We are going to use I/O-blackbox testing. That is, we will give our program a environment to solve (as Input), and then test that our program's Output is what we expect it to be.

A test consists of two text-files:

1. <testname>.env - The *input* environment for the program to solve
2. <testname>.out - The *expected output* which is the solution to the environment.

A test *passes* if the output of your program *matches* the expected output. Section 4.4 explains how to run your tests using the diff tool.

You should aim to write a minimum of **four tests**. We will mark your tests based on how suitable they are for testing that your program is 100% correct. Just having four tests is not enough for full marks.

The starter code contains a folder with one sample test case. This should give you an idea of how your tests should be formatted.

3.2 Milestone 2: Forward Search

In this milestone, you will implement a *forward search algorithm* that starts from the "start" position and explore the environment is a systematic way until it reaches the "goal" position. The pseudo-code for the algorithm you need to implement is provided below:

3.2.1 Forward Search Algorithm

In this algorithm, a position in the environment that the robot can reach is denoted by a node and each node will contain the (col,row) co-ordinate and distance_travelled (the distance that the algorithm took to reach that position from the robot's starting position).

Pseudocode for the forward Search algorithm

Input: E - the environment
Input: S - start location of the robot in the environment
Input: G - goal location for the robot to get reach
Let P be a list of positions the robot can reach, with distances (initially contains S). This is also called the open-list.
Let C be a list of positions the that has already being explored, with distances (initially empty). This is also called the closed-list.
repeat
    Select the node p from the open-list P that has the smallest estimated distance (see Section 3.2.2) to goal and, is not in the closed-list C.
    for *Each position q in E that the robot can reach from p* **do**
        Set the distance_travelled of q to be one more that that of p
        Add q to open-list P only if it is not already in it.
    end
    Add p to closed-list C.
until *The robot reaches the goal, that is, $p == G$, or no such position p can be found*

3.2.2 Estimated distance

For the above algorithm we need to estimate the distance from a given node to the goal node. It is not possible to know the exact distance from a given node to the goal before solving the path planning problem. However we can come up with an approximation. In this implementation we use the following approximation as the estimated distance from a given node, p, to the goal node G.

$$\text{Estimated distance} = \text{distance\_travelled of node } p + \text{Manhattan distance from } p \text{ to } G$$

The Manhattan distance from node p with coordinates (col_p, row_p) to node G with coordinates (col_G, row_G) is computed as:

$$\text{Manhattan\_distance} = |col_p - col_G| + |row_p - row_G|$$

here $|x|$ is the absolute value of x. The Manhattan distance represent the shortest distance from a node to goal if there are no obstacles.

3.2.3 Implementation details

It is important to have a good *design* for our programs and use suitable *data structures* and *classes*. In Assignment 1, you will implement our design[1]. You will implement 3 classes:

- `Node` class - to represent a position (col, row, $distance\_travelled$) of the robot.
- `NodeList` class - provides a method for storing a list of node objects as used in pseudo-code above.
- `PathSolver` class - that executes the forward search and backtracking algorithms.
- The main file that uses these classes, and does any reading/writing to standard input/output.

You are given these classes in the starter code. You may add any of your own code, but you **must not modify** the definitions of the provided class methods and fields.

3.2.4 `Node` Class

The `Node` class represents a position of the robot. It is a tuple (col,row,distance_travelled), which is the x-y location of the robot, and the distance that the algorithm took to reach that position from the robot's starting position. It contains getters for this information and setter for distance_travelled.

[1] This won't be the case for Assignment 2, where you will have to make these decisions for yourself.

```
// Constructor/Desctructor
Node(int row, int col, int dist_traveled);
~Node();

// get row-coodinate of the node
int getRow();

// get column-coodinate of the node
int getCol();

//getter and setter for distance traveled
int getDistanceTraveled();
void setDistanceTraveled(int dist_traveled);

//getter for estimated dist to goal - need to return -> Manhatten distance + distance traveled
int getEstimatedDist2Goal(Node* goal);
```

3.2.5 NodeList **Class**

The NodeList class provides a method for storing a list of Node objects. It stores an *array* of Node objects. Since it's an array we also need to track the number of position objects in the NodeList.

You *must* implement the NodeList **class using an *array*.**

```
// NodeList: list of node objects
// You may assume a fixed size for M1, M2, M3
Node* nodes[NODE_LIST_ARRAY_MAX_SIZE];

// Number of nodes currently in the NodeList
int length;
```

The constant NODE_LIST_ARRAY_MAX_SIZE is the maximum number of objects that can be in a NodeList. This constant is given in the Types.h header file.

```
#define ENV_DIM                       20
#define NODE_LIST_ARRAY_MAX_SIZE      4*(ENV_DIM * ENV_DIM)
```

The NodeLsit class has the following methods:

```
// Constructor/Desctructor
NodeList();
~NodeList();

// Copy Constructor
// Produces a DEEP COPY of the NodeList
NodeList(NodeList& other);

// Number of elements in the NodeList
int getLength();

// Add a COPY node element to the BACK of the nodelist.
void addElement(Node* newNode);

// Get a pointer to the ith node in the node list
Node* getNode(int i);
```

These methods let you add positions to the NodeList, and get a pointer to an existing position. Be aware, that the NodeList class has full control over all position objects that are stored in the array. Thus, if position objects are removed from the array you must remember to "delete" the objects.

3.2.6 PathSolver **Class**

The PathSolver class executes the two parts (forward search, backtracking) of the path planning algorithm by using the NodeList and Node classes. It has three main components:

1. forwardSearch: Execute the forward search algorithm.
2. getNodesExplored: returns a DEEP COPY of the explored NodeList in forward search.
3. getPath: Execute backtracking and Get a DEEP COPY of the path the robot should travel. To be implemented for milestone 3.

```
// Constructor/Destructor
PathSolver();
~PathSolver();

// Execute forward search algorithm
// To be implemented for Milestone 2
void forwardSearch(Env env);

// Get a DEEP COPY of the explored NodeList in forward search
// To be implemented for Milestone 2
NodeList* getNodesExplored();

// Execute backtracking and Get a DEEP COPY of the path the
// robot should travel
// To be implemented for Milestone 3
NodeList* getPath(Env env);
```

This uses a custom data type Env, which is given in the Types.h. It is a 2D array of characters that represents a environment using the format in Section 2. It is a fixed size, because we assume the size of the environment is known.

```
// A 2D array to represent the environment or observations
// REMEMBER: in a environment, the location (x,y) is found by env[y][x]!
typedef char Env[ENV_DIM][ENV_DIM];
```

It is very important to understand the Env type. It is defined as a 2D array. If you recall from lectures/labs, a 2D array is indexed by *rows* then *columns*. So if you want to look-up a position (col,row) in the environment, you find this by env[row][col], that is, first you look-up the row value, *then* you look-up the col value.

The forwardSearch method is given an environment, and conducts the forward search in Section 3.2.1. Remember, that the initial position of the robot is recorded in the environment. Importantly, the forwardSearch method **must not** modify the environment it is given. The forwardSearch method will generate a list of nodes the robot explored ("closedList" C in pseudo-code) and will store this list of positions in the private field:

```
// list of positions from the forward search algorithm
NodeList* nodesExplored;
```

The getNodesExplored method returns a **deep copy** of the nodesExplored field. Be aware that this is a **deep copy**, so you need to return a new NodeList object.

The implementation of getPath method is part of milestone 3 and will be discussed in detail in Section 3.3.

3.2.7 main **file**

The main file:

1. Reads in an environment from standard input.
2. Executes the forward search algorithm.
3. Gets the nodes explored in the forward search.
4. Gets the full navigation path (to be implemented in Milestone 3).
5. Outputs the environment (with the path) to standard output (to be implemented in Milestone 3).

The starter code gives you the outline of this program. It has two functions for you to implement that read in the environment and print out the solution.

```cpp
// Read a environment from standard input.
void readEnvStdin(Env env);

// Print out a Environment to standard output with path.
// To be implemented for Milestone 3
void printEnvStdout(Env env, NodeList* path);
```

Some hints for this section are:

- You can read one character from standard input by:

```cpp
char c;
std::cin >> c;
```

- Remember that is *ignores all white space* including newlines!

3.3 Milestone 3: Finding the path

For Milestone 3, you will implement the getPath method of class PathSolver and printEnvStdout method in main.cpp.

3.3.1 Implementing getPath method

This method finds a path from the robot's starting position to the specified goal position using the NodeList stored in field nodesExplored (generated in your Milestone 2) - The backtracking algorithm. Then the path found should be returned as a deep copy. The path should contain an ordered sequence of Node objects including the starting position and the given goal position. You may assume that the goal co-ordinate can be reached from the starting position.

The backtracking algorithm is not given to you as a pseudo code. However, the following hint is given to help you formulate a pseudo-code.

Hint: *"Start from the goal node in the list nodesExplored. This would be your final element of the path. Then search for the the four neighbours of the goal node in nodesExplored. If there is a neighbour that has distance_traveled one less than the goal node. Then that should be the node in the path before the goal node. Repeat this backtracking process for each node you add to the path until you reach the start node."*

Think carefully the path that you return **must be from start to finish, not finish to start**.

Be aware that the returned path is a **deep copy** of the path, so you need to return a new NodeList object.

3.3.2 Printing the path

The next step is showing the path the robot should take in navigating from where it started until it reached the goal. You should implement this in printEnvStdout method in main.cpp. For example, using the environment from the Introduction section, the robot's path is below:

```
==========
=S>>>>>>v=
========v=
=...v<<=v=
==.=v=^<<=
=..=>v=.==
=.===v=..=
=..==G==.=
===.======
==========
```

When showing the output of the path, it must show the direction in needs to be in order to get to the next position. To represent the robot's **direction**, we use the 4 symbols shown below:

Symbol	Meaning
>	Move Right
<	Move Left
^	Move Up
v	Move Down.

When printing the environment, you might find it easier to first update the environment with navigation path, and then print out the whole environment.

3.4 Milestone 4: Dynamic Array Allocation

This is a *challenging* Milestone. Attempt this once you have completed the other milestones.

For Milestones 1 - 3, we assume that the environment is *always* of a fixed size (20x20). This means, that for the Env data type and the NodeList class we could define the size of the arrays before-hand.

For Milestone 4, you must modify your implementation to accommodate two significant changes:

- Use a Environment of *any rectangular* size.
- Dynamically resize the nodes field of the Nodelist as more elements are required in the array, rather than use a fixed size.

To do this, you will need to modify a number of aspects of your implementation to **dynamically** allocate memory for 1D and 2D arrays. You will need to consider the following modifications:

- Change the type of Env to a generic 2D array:
  ```
  typedef char** Env;
  ```
  The milestone4.h file in the starter code has a sample method to help you dynamically allocate memory for a 2D array.
- Change the type of the field nodes in the NodeList class to a generic 1D array of pointers:
  ```
  Node** nodes;
  ```
- Create memory as necessary for the environment and NodeList.
- When reading in the environment, you will need to be able to spot newline characters. You can't do this if you follow the suggestion for Milestone 2. Instead you will need the get method of std::cin:
  ```
  char c;
  std::cin.get(c);
  ```
  See the C++ Reference documentation for more information.

3.5 Documentation, Style and Code Description

Making sure your code is 100% correct is very important. Making sure your code is understandable is equally important. Your code should follow the Course Style Guide, as given on Canvas (including not using any banned elements), and should be well documented with clear comments.

Finally, you need to provide a **short 1-paragraph description** (at the top of your main file) to:

- Describe (briefly) the approach you have taken in your implementation
- Describe (briefly) any issues you encountered

If you completed Milestones 3 or 4, this code description should include what you had to do for these milestones. **You may only use C++ languages features and STL elements that are covered in class.**

4 Getting Started

4.1 Starter Code

We have provided starter code to help you get underway. This includes files for the classes, the Types.h and main files, and the.

To compile your program, you will need to use a command similar to the following:

```
g++ -Wall -Werror -std=c++14 -O -o assign1 Node.cpp NodeList.cpp PathSolver.cpp main.cpp
```

4.2 Suggestions for starting Milestone 1

The starter code also contains a folder with one sample test case for Milestone 3. This should give you an idea of how your tests should be formatted.

4.3 Suggestions for starting Milestone 2

Part of the learning process of the skill of programming is devising how to solve problems. In this assignment, the problem solving is turning an *algorithm* and *pseudocode* into a *complete functioning program*.

This process involves completing small tasks one-at-a-time. We recommend the sequence of tasks:

1. In the main file, read in a environment from standard input and print out this environment (unmodified)
2. Implement the Node class
3. Implement the NodeList class
4. Implement the PathSolver class
5. Update the main file to use the PathSolver

Testing is also an important part of this process. The tests you need to write for Milestone 1 test *your whole program*. This has a problem, because this means you have to write the whole program first. However, you can write small programs to ***test your program as you go***. The main file in the starter code has a couple of examples to help you test that your Node and NodeList class as you develop them. Of course, once you finish the assignment, you can delete this testing code. This **lets you test small parts of your program as you go** rather than waiting until the end and just hoping the whole thing works.

4.4 Running Milestone 1 Tests

As a reminder, you can run a test as below. Recall that is uses the diff program to compare the *actual* and *expected* output of your program.

```
» ./assign1 <testname.env >actual.out
» diff actual.out testname.out
```

5 Submission

Follow the detailed instructions **on Canvas** to complete your submission for Assignment 1.

Assessment declaration: When you submit work electronically, you agree to the assessment declaration.

A penalty of 10% per day is applied to late submissions up to 5 days, after which you will lose ALL the assignment marks. Extensions will be given only in exceptional cases; refer to Special Consideration process.

Special Considerations given after grades and/or solutions have been released *will automatically result in an equivalent assessment in the form of a test*, assessing the same knowledge and skills of the assignment.

6 Marking guidelines

The marks are divided into three categories:

- Tests: 4/30 (15%)
- Software Implementation: 18/30 (60%)
- Code Style, Documentation & Code Description: 8/30 (25%)

The detailed breakdown of this marking guidelines is provided on the rubric linked on Canvas.

Please take note that the rubric is structured with with three “brackets”:

- If you do a good job on Milestone 1 & 2, then your final mark will be a CR. This will mean you have a CR in all three rubric categories
- If you do a good job for Milestone 3, then you mark will be a DI, getting a DI in all rubric categories
- If you do a good job for Milestone 4, your mark will be a HD.

The purpose of this is for you to focus on successfully completing each Milestone *one-at-a-time*. You will also notice there are not many marks for "trying" or just "getting started". This is because this is an *advanced* course. You need to make *significant* progress on solving the task in this assignment before you get any marks.

7 Academic integrity and plagiarism (standard warning)

CLO 6 for this course is: *Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.*

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites. If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students

For further information on our policies and procedures, please refer to the RMIT Academic Integrity Website.

The penalty for plagiarised assignments include zero marks for that assignment, or failure for this course. Please keep in mind that RMIT University uses plagiarism detection software.