RMIT UNIVERSITY

Advanced Programming Techniques
COSC1076 | Semester 1 2022
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 10 April 2022 (Before Week 7)
Silence Policy	From 5.00pm, Friday 08 April 2022 (Week 6)
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

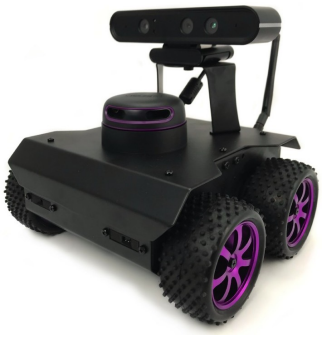

Figure 1: An example robot

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.

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[1]:

```
==========
=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)
  .         .
  .         .
(0,9) . . (9,9)
```

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.

[1]for illustration purposes we are using an environment of 10x10, but the grid size for the implementation is different. Please read the document carefully to find the appropriate grid size for each milestone.

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 (or reach) 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. *Identify Reachable Positions*: Identify all possible locations that the robot can reach given the starting position, and calculate the number of steps need to reach each position from start. The pseudo-code for the algorithm you need to implement is provided to you in Section 3.2.1.
2. *Path Planning*: Using the identified reachable positions, finds a path from the robot's starting position to the specified goal co-ordinate. 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 (denoted by "S").
3. Executes the *Identify Reachable Positions* algorithm (Section 3.2.1).
4. Executes the *Path Planning* algorithm (Section 3.3.1) to find a 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.* For example, if you attempt Milestone 3, but your Milestone 2 is buggy, you won't get any marks for Milestone 3.

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 an environment to solve (as Input), and then test that our program's Output is what we expect it to be.

To run I/O-blackbox testing, we need to design appropriate environments and workout what should be the output if our program is 100% correct.

A test consists of two text-files:

1. `<testname>.env` - The *input* environment for the program to solve
2. `<testname>.expout` - 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. We will also do a demo in the lectorial (Workshop).

You should aim to write a minimum of **four tests**[2]. 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. Identify scenarios where your program might break and construct tests accordingly.

The starter code contains a folder with one sample test case. This should give you an idea of how your tests should be formatted (the sample will not be counted when marking).

[2]This is upto M3, for M4 you should write additional tests.

3.2 Milestone 2: Identify Reachable Positions

In this milestone, you will implement a *Reachable Positions algorithm* that starts from the "start" position and explore the environment in a systematic way. The pseudo-code for the algorithm you need to implement is provided below:

3.2.1 Reachable Positions 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 `distanceToS` (the distance that the algorithm took to reach that position from the robot's starting position, S).

Pseudocode for the Reachable Positions algorithm

```
1  Input: E - the environment
2  Input: S - start location of the robot in the environment
3  Let O be a list of nodes (positions with distances) the robot can reach. Initially contains S. This is
     also called the open-list.
4  Let C be a temporary list of nodes. Initially empty. This is also called the closed-list.
5  Define a variable p and point it to the start node in O
6  repeat
7  |  for Each node q that the robot can reach from p[a] do
8  |  |  Set the distanceToS of q to be one more that that of p
9  |  |  Add q to O if and only if there is no item in O with the same co-ordinate as q.
10 |  end
11 |  Add p to closed-list C.
12 |  Select a node p from the open-list, O, that is NOT in the closed-list C.
13 until no such position p can be found
```

[a] when picking q, you should do that in the following order relative to p: `up`, `right`, `down`, `left`

A video of a worked example of the above algorithm is provided on canvas (assignment 1 page).

3.2.2 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[3]. You will implement 3 classes:

- `Node` class - to represent a position (*col*, *row*, *distanceToS*) of the robot.
- `NodeList` class - provides a method for storing a list of node objects as used in pseudo-code above.
- `PathPlanner` class - that executes the *reachable positions algorithm* and *path planning 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.

3.2.3 `Node` Class

The `Node` class represents a position of the robot. It is a tuple `(col,row,distanceToS)`, 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 `distanceToS`.

[3] 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 distanceToS);
~Node();

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

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

//getter and setter for distance to source
int getDistanceToS();
void setDistanceToS(int distanceToS);
```

3.2.4 `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 `NodeList` 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();

// Get a pointer to the ith node in the node list
NodePtr get(int i);

// Add a COPY node element to the BACK of the nodelist
//    This class now has control over the pointer
//    And should delete the pointer if the position-distance
//    is removed from the list
void addBack(NodePtr newNode);

// Checks if the list contains a node with the same co-ordinate
//    as the given node.
bool containsNode(NodePtr node);

// Remove everything from the list
// Don't forget to clean-up the memory!
void clear();
```

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.

3.2.5 `PathPlanner` Class

The `PathPlanner` class executes the two parts (*reachable positions algorithm*, *path planning algorithm*) of the path planning algorithm by using the NodeList and Node classes. It has two main components:

1. `getReachableNodes`: Execute the reachable positions algorithm. Returns a DEEP COPY of the reachable NodeList in reachable positions algorithm.
2. `getPath`: [To be implemented for milestone 3]. Execute path Planning algorithm and Get a DEEP COPY of the path the robot should travel.

```
// Constructor/Destructor
// Initialise a with a given maze of size (rows,cols)
PathPlanner(Env env, int rows, int cols);
~PathPlanner();

// Set The initial position
void initialPosition(int row, int col);

// Method for Milestone 2
// Return a DEEP COPY of the NodeList of all node's
//    that the robot can reach
NodeList* getReachableNodes();

// Method for Milestone 3
// Get the path from the starting position to the given co-ordinate
//    The path should be a DEEP COPY
NodeList* getPath(int toRow, int toCol);
```

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 `getReachableNodes` method executes the reachable positions algorithm in Section 3.2.1 and 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.

Importantly, the `getReachableNodes` method **must not** modify the environment it is given. The `getReachableNodes` method will generate a list of nodes the robot explored ("closedList" C in pseudo-code) and will store this list of nodes in a private field.

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

3.2.6 `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.

```
// 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 printPath(Env env, NodeList* solution);
```

Some hints for this section are:

- You can read one character from standard input by:

  ```
  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 `PathPlanner` 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 generated in your Milestone 2. 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 Path Planning 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 NodeList you computer for M2. Then search for the the four neighbours of the goal node in NodeList (you should do that in the following order relative to current node:* ***`up`****,* ***`right`****,* ***`down`****,* ***`left`****). If there is a neighbour that has* ***`distanceToS`*** *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 `printPath` 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 PathPlanner.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 `PathPlanner` class
5. Update the `main` file to use the `PathPlanner`

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.

5.1 Silence Period

A silence policy will take effect from **5.00pm, Friday 08 April 2022 (Week 6)**. This means no questions about this assignment will be answered, whether they are asked on the discussion board, by email, or in person. Make sure you ask your questions with plenty of time for them to be answered.

5.2 Late Submissions & Extensions

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.