Advanced Robot Programming Labs
C++ Programming

Maze generation and solving

1 Content of this lab

In this lab you will use and modify existing code in order to generate and solve mazes. As shown in Fig. 1, the goal is to generate a maze of given dimensions (left picture) and to use a path planning algorithm to find the shortest path from the upper left to the lower right corners (right picture).

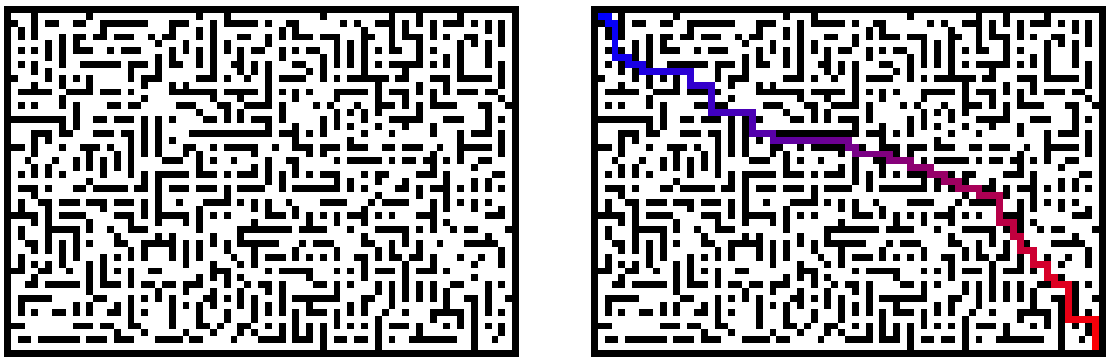

Figure 1: 51×75 maze before (left) and after (right) path planning.

The lab was inspired by this Computerphile video.

We use the classical mix of Git repository and CMake to download and compile the project:

`mkdir build` → `cd build` → `cmake ..` → `make`

The program can the be launched.

2 Required work

Four programs have to be created:

1. Maze generation
2. Maze solving through A* with motions limited to 1 cell (already quite written)
3. Maze solving through A* with motions using straight lines
4. Maze solving through A* with motions using corridors

As in many practical applications, you will start from some given tools (classes and algorithm) and use them inside your own code.
As told during the lectures, understanding and re-using existing code is as important as being able to write something from scratch.

3 Maze generation

The Wikipedia page on maze generation is quite complete and also proposes C-code that generates a perfect maze of a given (odd) dimension. A perfect maze is a maze where there is one and only one path between any two cells.
Create a `generator.cpp` file by copy/pasting the Wikipedia code and modify it so that:

- It compiles as C++.
- The final maze is not displayed on the console but instead it is saved to an image file
- The executable takes a third argument, that is the percentage of walls that are randomly erased in order to build a non-perfect maze.

A good size is typically a few hundred pixels height / width. To debug the code, 51 x 101 gives a very readable maze.
The `ecn::Maze` class (Section A.1) should be used to save the generated maze through its `Maze::dig` method that removes a wall at a given (x,y) position. It can also save a maze into an image file.

4 Maze solving

The given algorithm is described on Wikipedia. It is basically a graph-search algorithm that finds the shortest path and uses a heuristic function in order to get some clues about the direction to be favored.
In terms of implementation, the algorithm can deal with any `Node` class that has the following methods:

- `vector<unique_ptr<Node>> Node::children()`: returns a `vector` of smart pointers[1] to the children (or neighboors) of the considered element
- `int distToParent()`: returns the distance to the node that generated this one
- `bool is(const Node &other)`: returns true if the passed argument is actually the same point
- `double h(const Node &goal)`: returns the heuristic distance to the passed argument
- `void show(bool closed, const Node & parent)`: used for online display of the behavior
- `void print(const Node & parent)`: used for final display

While these functions highly depend on the application, in our case we consider a 2D maze so some of these functions are already implemented in, as seen in Section A.2:

- For the first exercice, only the `children` method is to write.
- The second one adds the `distToParent` method.
- The last one adds the `show` and `print` methods.

[1] The use of smart pointers (here `unique_ptr`) is detailed in Section B

4.1 A* with cell-based motions

The first A* will use cell-based motions, where the algorithm can only jump 1 cell from the current one.
The file to modify is `solve_cell.cpp`. At the top of the file is the definition of a `Position` class that inherits from `ecn::Point` in order not to reinvent the wheel (a point has two coordinates, it can compute the distance to another point, etc.).
The only method to modify is `Position::children` that should generates the neighboors of the current point. The parent node is likely to be in those neighboors, but it will be removed by the algorithm.

4.2 A* with line-based motions

Copy/paste the `solve_cell.cpp` file to `solve_line.cpp`.

Here the children should be generated so that a straight corridor is directly followed (ie children can only be corners, intersections or dead-ends). A utility function `bool is_corridor(int, int)` may be of good use.

The distance to the parent may not be always 1 anymore. As we know the distance when we look for the neighboor nodes, a good thing would be to store it at the generation by using a new Constructor with signature `Position(int _x, int _y, int distance)`.

The existing `ecn::Point` class is already able to display lines between two non-adjacent points (as long as they are on the same horizontal or vertical line). The display should thus work directly.

4.3 A* with corridor-based motions

Copy/paste the `solve_line.cpp` file to `solve_corridor.cpp`.

Here the children should be generated so that any corridor is directly followed (ie children can only be intersections or dead-ends, but not simple corners). A utility function `bool is_corridor(int, int)` may be of good use.

The distance to the parent may not be always 1 anymore. As we know the distance when we look for the neighboor nodes, a good thing would be to store it at the generation by using a new Constructor with signature `Position(int _x, int _y, int distance)`.

The existing `Point::show` and `Point::print` methods are not suited anymore for this problem. Indeed, the path from a point to its parent may not be a straight line. Actually it will be necessary to re-search for the parent using the same approach as to generate the children.

For this problem, remember that by construction the nodes can only be intersections or dead-ends. Still, the starting and goal positions may be in the middle of a corridor. It is thus necessary to check if a candidate position is the goal even if it is not the end of a corridor.

5 Comparison

Compare the approaches using various maze sizes and wall percentage. If it is not the case, compile in `Release` instead of `Debug` and enjoy the speed improvement.

The expected behavior is that mazes with lots of walls (almost perfect mazes) should be solved must faster with the corridor, then line, then cell-based approaches.
With less and less walls, the line- and cell-based approaches should become faster as there are less and less corridors to follow.

A Provided tools

A.1 The `ecn::Maze` class

This class interfaces with an image and allows easy reading / writing to the maze.

Methods for maze creation

- `Maze(std::string filename)`: loads the maze from the image file
- `Maze(int height, int width)`: build a new maze of given dimensions, with only walls
- `dig(int x, int y)`: write a free cell at the given position
- `save()`: saves the maze to `maze.png` and displays it

Methods for maze access

- `int height()`, `int width()`: maze dimensions
- `bool cell(int x, int y)`: returns true for a free position, or false for a wall

Methods for maze display

- `write(int x, int y, int r, int g, int b, bool show = true)`
  will display the (x,y) with the (r,g,b) color and actually shows if asked
- `passThrough(int x, int y)`: write the final path, color will go automatically from blue to red. Ordering is thus important when calling this function
- `saveSolution(std::string suffix)`: saves the final image

A.2 The `ecn::Point` class

This class implements basic properties of a 2D point:

- `bool is(Point)`: returns true if both points have the same x and y coordinates
- `double h(const Point &goal)`: heuristic distance, may use Manhattan ($|x - \texttt{goal}.x| + |y - \texttt{goal}.y|$) or classical Euclidean distance.

- `void show(bool closed, const Point & parent)`: draws a straight line between the point and its parent, that is blue if the point is in the closed set or red if it is in the open set.
- `void print(const Point& parent)`: writes the final path into the maze for display, also considers a straight line

All built classes should inherit from this class. The considered maze is available through the static member variable `ecn::maze` and can thus be accessed from the built member functions.

B Pointers and smart pointers

During an A* algorithm, a lot of nodes are created, deleted and moved between the open and the closed set. This makes it necessary to create the nodes on the heap and manipulate only pointers. In our case, new nodes are created in when calling the `children()` method, while they are deleted by the algorithm itself if needed. In order to avoid memory leaks, we will use smart pointers.

Assuming that a new position can be created from its coordinates, a new position may be created with:

- `Position p(x, y);` for an actual position
- `Position* p = new Position(x, y);` for a raw pointer
- `std::unique_ptr<Position> p(new Position(x, y));` for a unique ptr

As the `children()` method is supposed to return a `std::vector` of pointers, the corresponding syntax may be as follows:

```
typedef std::unique_ptr<Position> PositionPtr;
std::vector<PositionPtr> children()
{
    std::vector<PositionPtr> generated;
    while(..)   // whatever method we use to look for children
    {

        // assuming that we found a child at (x,y)
        generated.push_back(std::make_unique<Position>(x,y)));
    }
    return generated;
}
```

Here the generated children are directly put inside the vector, which is returned at the end. The smart pointer keeps track of the allocated memory and the A* algorithm can decide to keep it or not, knowing that the memory will be managed accordlingly.