INDIAN INSTITUTE OF TECHNOLOGY

KHARAGPUR

KSHITIJ 2013

ROBOTIX

TECHNOLOGY ROBOTIX SOCIETY

Brainstorming A.C.R.O.S.S. 1 – Grid Algorithms

Introduction

Welcome to Brainstorming ACROSS. This is a series of articles that will cover various aspects of the problem statement and their possible solutions. This particular article will cover the *Recursive Backtracker* algorithm that can be used by a bot to navigate on an 8x8 grid containing obstacles at unknown locations. Note that the article does not cover on how to implement this algorithm in an autonomous robot. That will be covered in future articles. At the end you can also find the link to a software for testing grid solving algorithms.

Objective

The objective is to go from point on the 8x8 to any other point. As an additional constraint, we consider that the arena in unknown to the bot. In other words, when the bot starts moving, it has no knowledge of the location of obstacles on the grid. It knows only its starting position and the required ending position.

Location

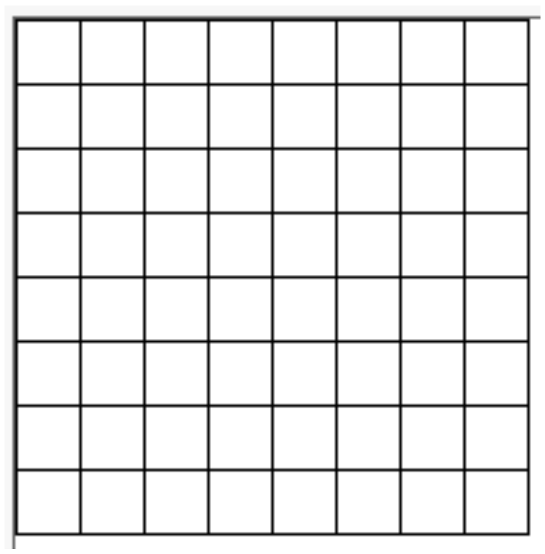

Figure 1 The 8x8 ACROSS Grid

Let the location of bot at any point be denoted by the 2D coordinate system that is by X and Y coordinates. In the arena shown below, each square denotes one such point. The *point (0, 0)* is denoted by the *top leftmost square*.

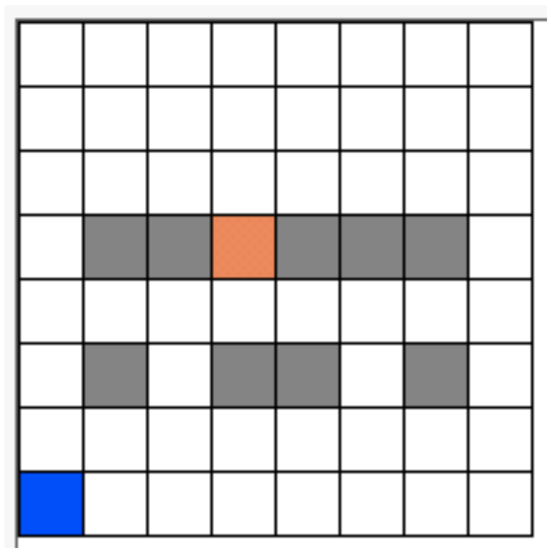

Figure 2 The ACROSS Round 1 Arena

- Grey denotes an obstacle in this case raised platforms
- Orange denotes the final location in this a gap between two platforms (3, 3)
- Dark Blue denotes the initial location (0, 7)
- Light Blue lines will denote the path taken by the robot

Movements

From any point on the grid, the bot can move in four directions:

- North - (x, y) to (x, y-1)
- East - (x, y) to (x+1, y)
- West - (x, y) to (x-1, y)
- South - (x, y) to (x, y+1)

Algorithm

At each coordinate of the grid, the basic roles of the bot remain the same. As a result we need only one basic function to navigate which can repeatedly called. The roles can be divided into two parts:

- *Base Cases:* These indicate whether the current location is valid that is whether it is possible to move to it or not. They also indicate if the bot has reached the destination thereby indicating it is time to stop.

- *Recursive Parts:* These scan the four positions in the four directions around the current position to check if any of them satisfy the base cases. In programming terms this involves placing a call to the same function but with a different set of arguments, which in this case correspond to a position. Hence this part is termed as recursive and it is responsible for the robot advancing through the maze.

Recursive Parts

Let's start by assuming there is already some algorithm that finds a path from some point in a maze to the goal and call it *RecurPath (x, y)*.

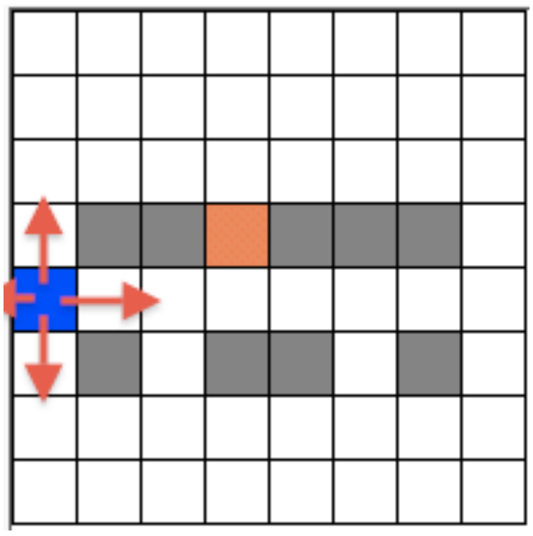
Figure 3 Robot at (0, 4)

Let us assume that the robot managed to reach a point (0, 4) using this function. Now it is required to check all the four directions to find the path ahead. Thus we place four calls:

- *RecurPath (0, 3) - North*
- *RecurPath (1, 4) - East*
- *RecurPath (-1, 3) - West*
- *RecurPath (0, 5) – South*

Note that these checks can be performed in any order. Now since going west will mean going out of the grid, that function call should return something to indicate the same. While rest of the directions are valid and the function should indicate the same.

Generalizing this, we can call *RecurPath ()* recursively to move from any location in the maze to adjacent locations. In this way, we move through the maze.

Base Cases

It's not enough to recursively advance through the maze. We also need to determine when to stop.

One such *base case* is to stop when it *reaches the goal*.

The other base cases have to do with moving to invalid positions. For example, we have mentioned how to search North of the current position, but disregarded whether that North position is legal. In order words, we must ask:

- *Is the position in the maze* (...or did we just go outside its bounds)?
- *Is the position open* (...or is it blocked with an obstacle)?

Pseudo-code

```
RecurPath(x,y) {
        Move robot to (x, y)
        if (x,y outside maze) return false
        if (x,y is goal) return true
        if (x,y not open or is marked visited) return false
        mark x,y as visited
        if (RecurPath(North of x,y) == true) return true
        if (RecurPath(East of x,y) == true) return true
        if (RecurPath(West of x,y) == true) return true
        if (RecurPath(South of x,y) == true) return true
        Move robot to (x, y)
        return false
}
```

Marking x,y as visited is necessary to prevent the bot from going around in circles. For the explanation to the last call to *Move robot to (x,y)* see the Dead Ends section.
To use to find and mark a path from the start to the goal with our given representation of mazes, we just need to:

1. Locate the start position (call it *startx*, *starty*).
2. Call *RecurPath(startx, starty)*
3. Re-mark the start position.

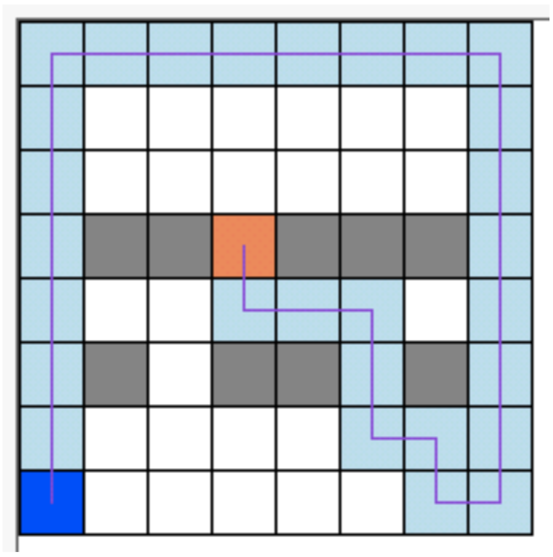
Figure 4 Sample Run

You can verify that each colored square that is part of the final path satisfies one of the 'if statements' of the Pseudo-code. As an example lets see what happens when the bot is at (0, 0):

First lets go one step backward to position (0, 1). At this position:

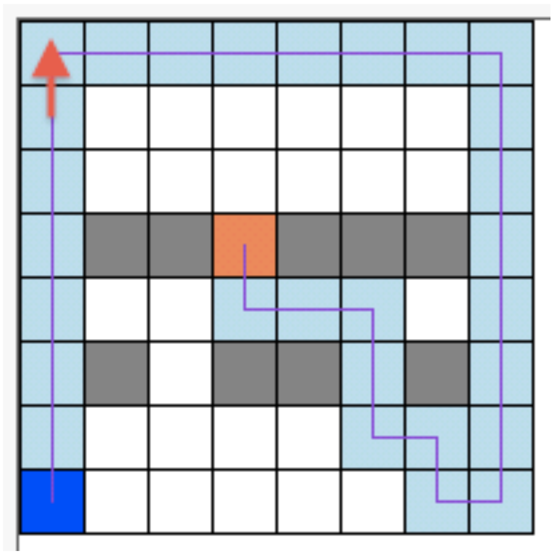

Figure 5 At position (0, 1)

```
RecurPath(0,1) {
        Move robot to (x, y)
        if (x,y outside maze) return false
        if (x,y is goal) return true
        if (x,y not open or is marked visited) return false
        mark x,y as visited
        if (RecurPath(North of x,y) == true) return true
        if (RecurPath(East of x,y) == true) return true
        if (RecurPath(West of x,y) == true) return true
        if (RecurPath(South of x,y) == true) return true
        Move robot to (x, y)
        return false }
```

Since none of the base cases are satisfied, code reaches the highlighted step, placing a call to *RecurPath (0, 0).* Note that the call has not yet finished executing.

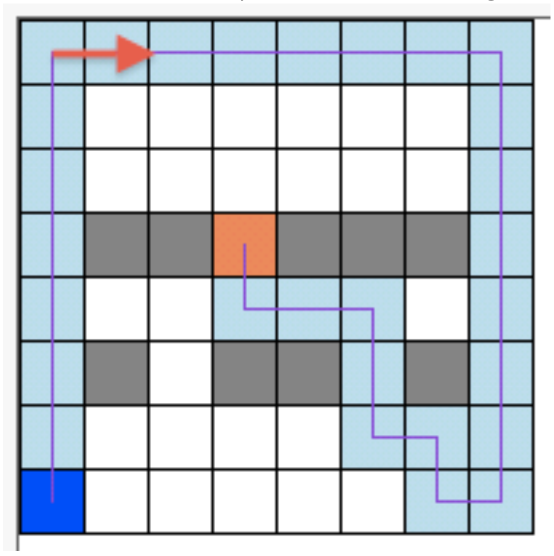

Figure 6 At position (0, 0)

```
RecurPath(0,0) {
        Move robot to (x, y)
        if (x,y outside maze) return false
        if (x,y is goal) return true
        if (x,y not open or is marked visited) return false
        mark x,y as visited
        if (RecurPath(North of x,y) == true)  return true
        if (RecurPath(East of x,y) == true) return true
        if (RecurPath(West of x,y) == true) return true
        if (RecurPath(South of x,y) == true) return true
        Move robot to (x, y)
        return false
}
```

Thus the robot moves to (0, 0). Reaches the code highlighted in Red, placing a call to *RecurPath(0, -1).* This will return false since the point is outside the maze. So the code proceeds to the next step, calling *RecurPath(1, 1).* Thus the robot moves to (1, 1) and in this manner the sequence keeps going on. The sequence will end only when the robot reaches the goal, that function call will then return true and subsequently all the calls before it will return true as well. Thus the output of our initial call to *RecurPath(startx, starty)* will be true.

Dead Ends and Backtracking

So what happens if the bot takes a wrong turn, a path, which leads to a dead end, or a path, which we know wont reach, the end point? Let us consider this run again:

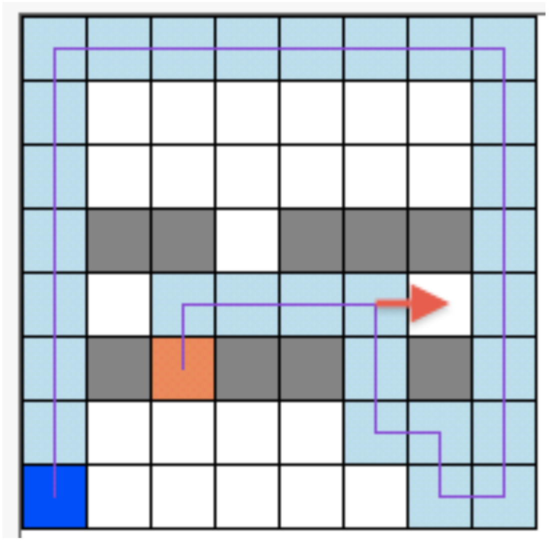

Figure 7 At position (5,4)

Now since the North is blocked so the bot turns to East calling *RecurPath(6,4):*

```
RecurPath(6,4) {
        Move robot to (x, y)
        if (x,y outside maze) return false
        if (x,y is goal) return true
        if (x,y not open or is marked visited) return false
        mark x,y as visited
        if (RecurPath(North of x,y) == true)  return true
        if (RecurPath(East of x,y) == true) return true
        if (RecurPath(West of x,y) == true) return true
        if (RecurPath(South of x,y) == true) return true
        Move robot to (x, y)
        return false }
```

Since all directions are either marked visited or are blocked, *Move robot to (6,4).* Since the bot is already there the call ends, returning false. Now the code goes back to the previous call *RecurPath(5,4)* with the bot still at (6,4)

```
RecurPath(5,4) {
        Move robot to (x, y)
        if (x,y outside maze) return false
        if (x,y is goal) return true
        if (x,y not open or is marked visited) return false
        mark x,y as visited
        if (RecurPath(North of x,y) == true)  return true   //Code reaches here
        if (RecurPath(East of x,y) == true) return true
        if (RecurPath(West of x,y) == true) return true
        if (RecurPath(South of x,y) == true) return true
        Move robot to (x, y)
        return false
}
```

Again the code reaches Move robot (5,4) moving the bot from (6,4). In this way the bot backtracks on reaching a dead end.

Resources

Software that demonstrates the above algorithm can be found here (https://dl.dropbox.com/u/79327907/MinotaurPathfinder.zip). Extract the zip file. You can find more information on how to use in the readme file contained within. For a source code of the software, written in C# (using Visual Studio 2012), click here (https://dl.dropbox.com/u/79327907/MinotaurPathfinder_Source.zip). Note that the software only marks the final valid path. The robot might have to additional squares and backtracked on finding a dead end.

This tutorial has been compiled using the following resources:

http://www.cs.bu.edu/teaching/alg/maze/
http://www.astrolog.org/labyrnth/algrithm.htm
http://www.dotnetperls.com/pathfinding

Looking Ahead

In future articles we will look at more algorithms, ways to a map a grid, how to implement the recursive backtracker in a bot and much more. Stay tuned. Meanwhile you can also try to optimize the backtracker algorithm according to the A.C.R.O.S.S arena. One way can be prioritize which direction is checked first. Remember that a juicy cash prize is waiting for the bot with the Best Algorithm in ACROSS in addition to the top prizes. May the best algorithm win!