Code for Robot Path Planning using A* Algorithm

Rahul Kala
Robotics and Artificial Intelligence Laboratory,
Indian Institute of Information Technology, Allahabad
Devghat, Jhalwa, Allahabad, INDIA

Email: rkala001@gmail.com
Ph: +91-8174967783
Web: http://rkala.in/

Version 1, Released: 7th June, 2014

1. Background

The code provided with this document uses A* algorithm for robot motion planning. Assume that you have a robot arena with an overhead camera as shown in Figure 1. The camera can be easily calibrated and the image coming from the camera can be used to create a robot map, as shown in the same figure. This is a simplistic implementation of the real life scenarios where multiple cameras are used to capture different parts of the entire workspace, and their outputs are fused to create an overall map used by the motion planning algorithms. This tutorial would assume that such a map already exists and is given as an input to the map.
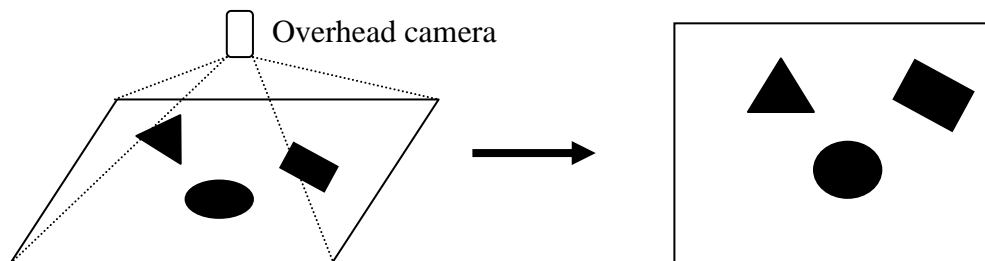

Figure 1: Overhead camera system and creation of robot map

The same camera can also be used to capture the location of the robot at the start of the planning and also as the robot moves. This solves the problem of localization. An interesting looking region of interest becomes the goal of the robot to be used in the motion planning algorithms. The robot is not shown in the map in Figure 1. This tutorial assumes that the source and goal of the robot is explicitly supplied. The aim of the current tutorial is only to plan a path for the robot; the code does not go forth with making the robot move on the desired path, for which a control algorithm is needed.

For simplicity, the robot is taken as a point-sized object. This enables a quick implementation and understanding, without delving into the libraries for collision detection and concepts of multi-dimensional configuration spaces.

2. Problem Solving using A* algorithm

In order to solve the problem using A* algorithm, it is necessary to first model the problem as a standard graph search algorithm. The readers should please familiarize themselves with the A* algorithm before reading this tutorial. The A* algorithm takes a graph as an input and explores, one by one, all the regions, finding the shortest

path from the source to all states in the explored regions. The A* algorithm does that such that all the near regions are explored before the further ones, while the exploration is also biased towards the regions closer to the goal (indicated by the heuristic function).

Since A* algorithm only works in discrete spaces, it is necessary to create a discrete space of the map. Luckily, the map taken as an array of discrete pixels is already in a discrete representation. However the A* algorithm would take a very large computation time on a high resolution map. The map resolution is hence reduced to get the map used by the A* algorithm, while the resolution is an algorithm parameter. In general, higher the resolution of the map, better are the results with excessive computation time.

The graph consists of vertices and edges. Each pixel of the reduced resolution map is taken as a vertex. Each vertex has a number of connections which act as edges (provided the connection is collision free). For any general position of the robot, the possible connections are given by Figure 2, which is coded as a matrix as shown in the same figure and given as an input to the graph. In the coded matrix, the assumed position of the robot is marked as '2'. There needs to be only one assumed position of the robot and hence only one entry of '2' in the matrix. All possible moves are denoted by 1 and all impossible moves are denoted by 0. The connection matrix is an input parameter and you can create your own matrices and see the results. Three typical matrices are shown in Figure 3. Figure 3(a) only allows the robot to take rectilinear moves (up, down, left and right). Figure 3(b) allows the robot to take four diagonal moves along with the 4 rectilinear moves. Figure 3(c) further allows the robot to make more flexible moves adding connections between the diagonal moves. More the number of ones in the matrix, more is the flexibility of turns for the robot, better is the path. However adding ones may result in greater computational costs.
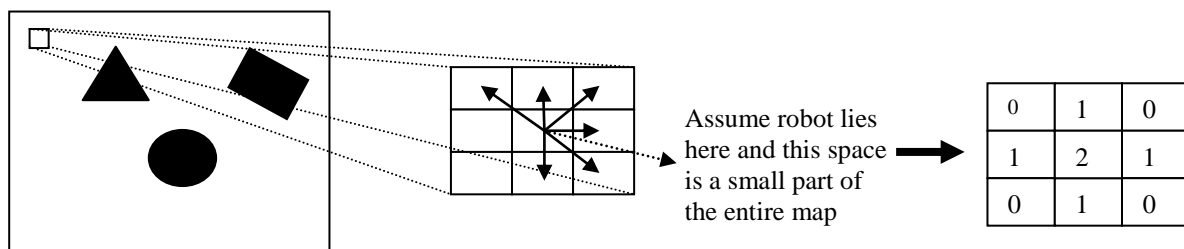

Figure 2: Connection Matrix

| (a) Rectilinear | (b) Rectilinear and Diagonal | (c) Many Moves |

Figure 3: Typical Connection Matrices

The next design specifications are related to costs. These include the weights of the edges, which are taken as the Euclidian distance between the connecting points, and the heuristic function denoting the nearness of the point to the goal which is taken as the Euclidian distance to the goal. Both the functions are given as separate files from where they can be changed.

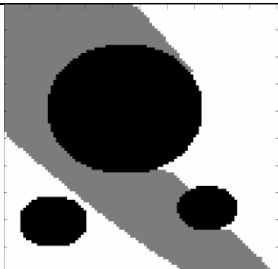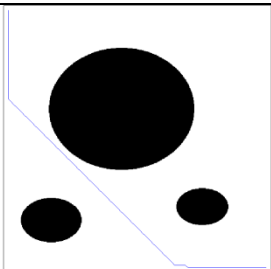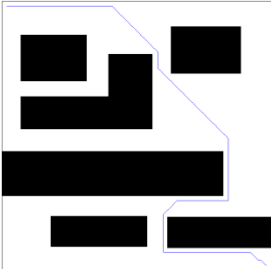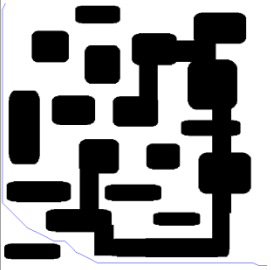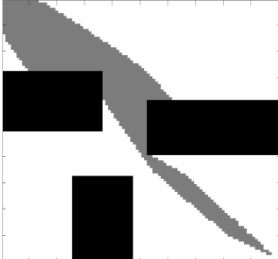## 3. How to Execute and Parameters
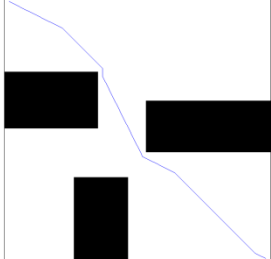
In order to execute the code you need to execute the file 'astart.m'. First make a new bitmap file and draw any map on it, over which you need to execute the algorithm. Paint or any other simple drawing tool can be used. Make sure you save the file as a BMP. Place the file in the project folder. You may prefer to open any of the supplied maps and re-draw them. Change the name of the map file in the code to point out to the map that you created. Supply the
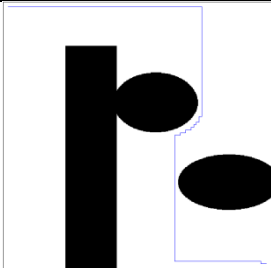
source and the goal positions. You can use paint or any other drawing utility which displays the pixel positions of the points, to locate the source and goal on your drawn map.

Change the X and Y resolutions as per your choice. Note that if your algorithm is taking too long to get a result, you will have to reduce the resolution. Similarly if you get results instantly, you may wish to increase the resolution to hope to get a better path. You may choose between any of the supplied connection matrices by de-commenting the one you select and commenting all the rest. Alternatively you may design your own connection matrix.

Execute the algorithm. You will need to take screenshots of the display, the tool does not do that for you. The first display shows the states, as they are expanded. Once the simulation stops, click on the image to get the path. The path length and execution time are given at the console. Make sure you turn the display off by changing the display variable in the parameters before quoting the execution performance. This will not display any additional graphics and hence benefit from the additional time spent in the display.

3. Sample Results

S. No.	Parameters	Expansion	Path	Path Length	Execution Time (sec)
1.	Connection Matrix: [1 1 1; 1 2 1; 1 1 1]			775	1.58
2.	Connection Matrix: [1 1 1; 1 2 1; 1 1 1]			1045	2.17
3.	Connection Matrix: [1 1 1 1 1; 1 1 1 1 1; 1 1 2 1 1; 1 1 1 1 1; 1 1 1 1 1]			890	5.01
4.	Connection Matrix: [1 1 1 1 1; 1 1 1 1 1; 1 1 2 1 1; 1 1 1 1 1; 1 1 1 1 1]			702	2.83

5.	Connection Matrix: [0 1 0; 1 2 1; 0 1 0]			1060	1.78

All results on Intel i7, 3.4 GHz 3.4 GHz with 12 GB RAM.
For all results:
source=[10 10]
goal=[490 490]
resolution of original map: 500×500
downsized resolution:100 ×100

For all results see that an increase in connection matrix results in better paths. However, worst cases would also involve an increase in computation times.

4. Disclaimer

Please feel free to ask questions and extended explanations by contacting the author at the address above. Please also report any errors, corrections or improvements.

These codes do not necessarily map to any paper by the author or its part. The codes are usually only for reading and preliminary understanding of the topics. Neither do these represent any state-of-the-art research nor any sophisticated research. Neither the author, nor the publisher or any other affiliated bodies guarantee the correctness or validity of the codes.