MEM380: Applied Autonomous Robots

Wk 2 Assignment

Due: Monday, Oct. 15, 11:59pm EST

1 Objective

The objective of this assignment is to implement the A* algorithm. You are given the following MATLAB files:

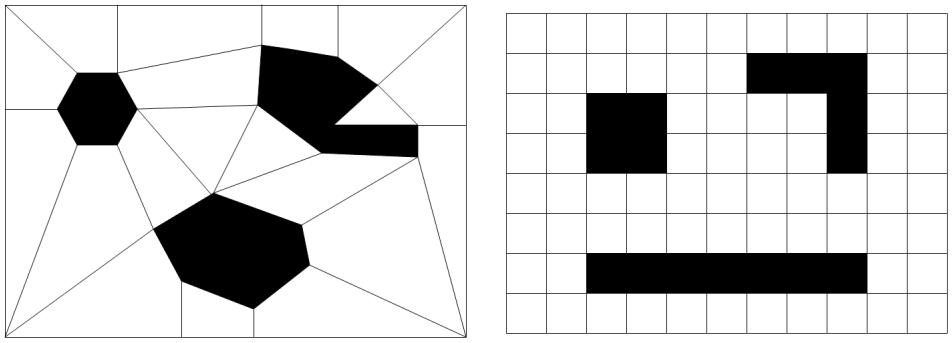
- `addToStack.m`: Usage `outStack = addToStack(newElement, originalStack)` where `newElement` can be an array or structure and `orginialStack` is an $L \times 1$ array of elements of type `newElement`. This function will attach `newElement` to the top of the stack given by `originalStack`.
- `popOffStack.m`: Usage `[outElement, newStack] = popOffStack(originalStack)` where `outElement` is the first element in `originalStack` and `newStack` is the `originalStack` with the first element removed.
- `AstarTestmap.txt`: A sample map. To load this into Matlab use `map = load('testmap.txt')`. This results in an $N \times M$ matrix such that every element represents a cell in the environment - a 0 denotes empty space, a 1 denotes an obstacle. This is the *occupancy grid* for your environment.
- `AStar_user.m`: Usage `path = AStar_user(start, qgoal,map)` where `path` gives you the path from `start` and `qgoal` and `map` is an occupancy grid, *i.e*, a matrix representing a map of the workspace.

Requirements: You MUST use MATLAB to complete this assignment.

2 Background

The problem of planning a path given a start and a goal position is called the *path planning problem*. We often encounter such problems in the area of mobile robotics where the goal is to give the robot the ability to determine how it should navigate from its current location to some desired goal position. As such, we must consider the robot's start and goal positions as well as the geometry of the environment, *i.e.* the location of the obstacles, the size of the obstacles, and the such. (This is also a common problem in video game design where the programmer must be able to compute the trajectories that the numerous characters follow in a game.)

There are numerous existing methods that can be employed to solve the path planning problem. A common approach is to tesselate the workspace into cells. Once the workspace has been properly discretized, one can represent the layout of the workspace using what is called an *occupancy grid*. An occupancy grid is simply a data structure used to encode the layout of the workspace. One can interpret the occupancy grid as simply a map of the environment where free space is represented by clear cells and occupied space, *i.e.* space where obstacles reside, is represented by solid cells. Figure 1(a) shows an occupancy grid for a workspace that has been tesselated into cells of varied size and shapes while Figure 1(b) shows an occupancy grid obtained using a fix grid decomposition of the workspace. From these two examples, we see that it is possible to represent any given occupancy grid as arrays of 0s and 1s where 0s denote free space and 1s denote occupied space. Furthermore,



(a) (b)

Figure 1: (a) General tesselation of an environment. Such methods will result in an occupancy grid with cells of varying sizes and shapes. (b) A grid decomposition of an environment. These results in square cells of the same dimensions.

by representing our workspace using an occupancy grid, we can easily find the shortest path between any two free cells in the environment by determining the shortest sequence of free cells that connects the two cells.

To do this, we pose the path planning problem as a *graph search problem*. Thus, for each empty cell in our occupancy grid we represent it as a node in a graph. An edge exist between two nodes if the cells they represent are adjacent and you can traverse from one cell to another. Once we have performed this transformation, our path planning problem is now simply a graph search problem. There are numerous graph search algorithms, breadth–first search, depth–first search, Dijkstra's, and numerous others. A* (pronounced as A star) can be seen as a variation of the ones I just mentioned. What distinguishes A* from breadth–first and depth–first search algorithms is that it employs a *heuristic function*, $h(x)$. The heuristic function provides an approximation for the cost of the best route that goes through each node. And it employs this heuristic estimate when determining which node to visit next in its search process. As such, A* is an example of a *best–first search* algorithm [5]. Furthermore, if we choose $h(x) = 0$, we can show that Dijkstra's algorithm is simply a special case of A*. Lastly, A* is both *complete* and *optimal*. This means that A* will always find a path if a path exists and report failure if a path does not exist and the path that A* returns will be optimal in terms of the heuristic function.

2.1 Algorithm Structure

In this section we take a closer look at the mechanics and the structure of the A* algorithm. Let's take the example shown in Figure 2. The objective is to find the shortest path between the start and goal. In other words, what is the minimum number of squares we need to tranverse in order to reach goal from start?

Starting the Search

Given our occupancy grid, we begin our search by starting at the start position and check its adjacent cells, searching outward towards the goal. Thus, we do the following:

(a) Begin at start and add it to an *open list* of cells to be considered. The open list contains the cells that *may* fall on the optimal path we want. In other words, the open list contains the cells we need to take a closer look at in our search process. As we expand outward, this open list will grow.

(b) Look at all the reachable cells, *i.e.* cells that do not contain obstacles, adjacent to start and add them to the open list. For each of these cells, save start as its *parent cell*, *i.e.* the cell you were at before reaching the adjacent cell. Saving this information correctly is extremely important since this is how we will trace our path once we have reached the goal, *i.e.* we will get our optimal path by tracing backwards from the goal to the start position.

(c) Drop the start the open list, and add it to a *closed/visited list* of cells.

Next, we choose one of the adjacent cells that we have added to the open list and more or less repeat the above process. To determine which cell to choose, we look at the cost associated with each cell.

2.1.1 Path Scoring

This is where our heuristic function comes in. In order to figure out which of the cells in our open list to "visit" next, consider the following equation:

$$f(x) = g(x) + h(x) \tag{1}$$

where x denotes the current cell, $g(x)$ gives the cost to move from the start position , start, to the current cell x, and $h(x)$ is the heuristic function which gives an estimate of the cost to move from the current cell x to the goal position, goal. In general, $h(x)$ can be seen as an educated guess of the cost to move from the current cell to the goal position. In the path planning community, $h(x)$ is often chosen to be the Euclidean distance between the current cell and the goal cell, *i.e.* the straight line distance between a robot's current position and the goal position disregarding all obstacles in the environment. As such $h(x)$ is generally chosen as a lower bound on the actual cost. Thus, given $g(x)$ and $h(x)$, the function $f(x)$ is a conservative estimate of the cost of the shortest path from start to goal through the current cell x. Our path is then generated by repeatedly going through our open list and choosing the cell with the lowest cost $f(x)$.

2.1.2 Completing the Search

To complete our search, at every iteration, we simply choose the cell from our open list with the lowest cost $f(x)$ and do the following on that cell:

(a) Drop it from the open list and move it to the closed list.

(b) Check all the adjacent, traversible cells and add those cells to the open list if they are not already on the list. Specify the current cell as the parent of the cells you add to the open list.

(c) If an adjacent cell is already on the open list, check to see if your current path to that cell is a better one. In other words, check to see if the value of $g(x)$ for the cell is lower if we use the current cell to get there. If not, dont do anything, otherwise, change the parent of the adjacent cell to the current cell. Remember to recalculate both the $f(x)$ and $g(x)$ scores of that cell.

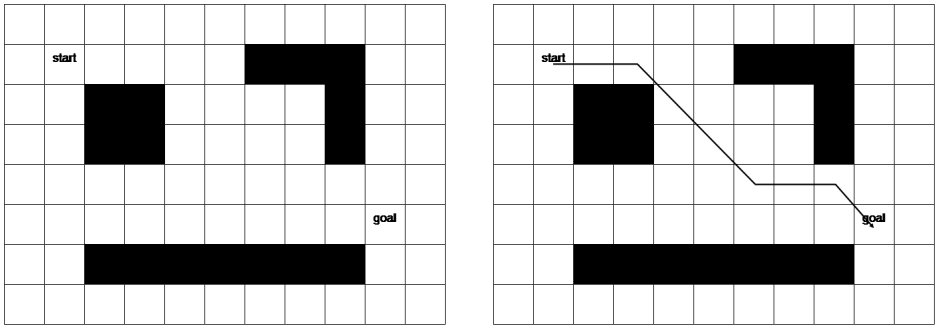
In summary:

(a) Add start to the open list.

(b) Repeat the following:

  (a) Look for the lowest cost, $f(x)$, cell on the open list and let this cell be the current cell x.

  (b) Move x to the closed list.



(a) (b)

Figure 2: An example.

(c) For each of the 8 adjacent cells of x, if the adjacent cell is occupied, ignore it, otherwise do the following:

- Add it to the open list if it isn't on the list. Specify x as the parent cell and compute $f(x)$, $g(x)$, and $h(x)$.
- If it is already on the open list, check to see if the current path to the cell is better by looking at the current value of $g(x)$ and the previously stored value of $g(x)$ associated with the cell. If the current $g(x)$ is lower, change the parent of the cell to x and recalculate $f(x)$, $g(x)$, and $h(x)$. If you are keeping your open list sorted by $f(x)$, you may need to resort the list to account for the change.

(c) Stop when you: 1) Add goal to the closed list, in which case the path has been found, or 2) fail to find goal, and the open list is empty. This occurs when no path exists between start and goal.

To print out the path, work backwards from the goal position. Look at the parent of goal, print this out. Then look at the parent of the parent of goal, print this out. And so on and so forth until you reach start. This is your path. The A* algorithm is also summarized in pseudo-code form in Algorithm 1. In our example the shortest path length is 9 and the path is shown in Figure 2(b).

This description of the A* algorithm and its implementation is a summary of the one provided by Patrick Lester in [3]. Look at this website for more implementation details.

3 Task 1: (20 points)

Assume a point robot. Write a function `path = AStar_user(start, qgoal, map)` where `start` and `qgoal` are 2×1 vectors containing the start and goal coordinates for the robot, `map` is your $N \times M$ matrix which encodes the map of the environment, and `path` is a $P \times 2$ vector of coordinates that encodes the shortest path found by the your A* function. Since your map is stored as a matrix, the *coordinates* can simply be (i, j) where i denotes the i^{th} row of the matrix and j denotes the j^{th} column of the matrix.

Hints: Use $(1, 3)$ and $(10, 7)$ as your sample start and goal positions. To maintain a sorted list in Matlab:

(a) Add the element to your array;

(b) Use Matlab's `sort()` command.

Algorithm 1 A* algorithm [5]

```
function A*(start,goal)
var closed = the empty set
var q = make_queue(path(start))
while q is not empty do
  var p = remove_first(q)
  var x = the last node of p
  if x in closed then
    continue
  end if
  if x = goal then
    return p
  end if
  add x to closed
  for y in successors(x) do
    enqueue(q, p, y)
  end for
  return failure
end while
```

4 Task 2: (5 points)

Modify your A* code so that you it now find Dijkstra shortest paths. Rename this function to be `path = Dijkstra_user(start, qgoal, map)`.

5 Extra Credit 1: (5 points)

Right a function that uses a Bread First Search to find shortest paths. Name this function to be `path = bfs_user(start, qgoal, map)`.

6 Extra Credit 2: (5 points)

Right a function that uses a Depth First Search to find shortest paths. Name this function to be `path = bfs_user(start, qgoal, map)`.

7 What to Submit

Put all your m-files into a single zip compressed file. Make sure your code is well commented! Make sure to include ALL helper functions you were provided with and have written. In your submission email, please provide a list of instructions on how to run your code. To start, provide a list of all the m-files included in your zipped file and explain how to use them. Make sure you say which of your m-files are scripts and which of those are functions. Next, provide a set of instructions on how to call your m-files so we obtain the solution. You may include any map files you used to test your code on in your zipped file.

References

[1] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Boston, 2005.

[2] A* Algorithm Tutorial. `http://www.geocities.com/jheyesjones/astar.html`

[3] Patrick Lester. *A* Pathfinding for Beginners.* http://www.policyalmanac.org/games/aStarTutorial.htm

[4] Maxim Likhachev, Geoff Gordon and Sebastian Thrun, "ARA*: Anytime A* with Provable Bounds on Sub-Optimality," Advances in Neural Information Processing Systems 16 (NIPS), MIT Press, Cambridge, MA, 2004.

[5] A* Search Algorithm. http://en.wikipedia.org/wiki/A*_search_algorithm