Alexander Hay
alexanderhay2020@u.northwestern.edu

ME 469 - Machine Learning and AI in Robotics

Assignment #1 - A* Search and Navigation

10/28/2019

Note: please refer to the readme.txt for specific attribute and function definitions.

Part A

1. The grid was constructed by using the Grid class. The Grid class maintains attributes of the grid_map that the algorithm works in. First, the nodes are created using the arange function, then built as an array using the numpy.ones function. This is convenient because it also assigns the node cost. To assign landmarks their node values, a landmark method was created. This method determines if the landmark falls within the ranges defined by the assignment, then finds which node the landmark falls in and reassigns the cost as 1000.
This section proved to be a consistent source of bugs throughout the assignment, and should be re-examined before implementing in another assignment. It'd be worth considering rewriting the class as a function instead, that outputs a grid. An earlier draft of this function used the matplotlib's histogram which seems more analogous to how the A* algorithm operates, but ultimately was discarded due to time constraints.
The grid and its landmarks are plotted via the plot function. This section was also a source of bugs, usually in tandem with grid. Translating from coordinate points to node spaces (bins) proved to be challenging. When the decision was made to discard the histogram plot a similar decision was made to move forward using just coordinate points.

2. The A* algorithm for this assignment was built using the following pseudo-code as a guideline:

```
Initialize grid

Initialize node costs (1)
for i in range(len(xedges)):
        for j in range(len(yedges)):
                grid_map[i,j].g = 1
        /loop
/loop

Initialize barrier costs (1000)
for i in range(len(landmarks)):
        grid_map[[landmark[i],1],[landmark[i],2]].g = 1000

Identify start/goal nodes

node = grid_map(start_coords)
goal = grid_map(goal_coords)

Initialize costs
node.g = distance from node to start
        node.g = 1
node.h = distance from node to goal; heuristic
node.f = g + h cost function; estimate of best route to node (we want the lowest cost)



Initiate open list []
```

```
Initiate closed list []
- Add start node to open list

*loop while open list is not empty
while len(open_list) > 0:
a) find node with least f on open list,
        call it node q

b) pop q off open list
        put q on closed list

c) generate q's 8 successors
        child_list = [
        [qx-1,qy+1],[qx,qy+1],[qx+1,qy+1]
        [qx-1,0],[qx, qy] ,[qx+1,0]
        [qx-1,qy-1],[qx,qy-1],[qx+1,qy-1]
        ]

        *calculate child's parameters
        for i in range(len(child_list)):
                child_list[i].parent = q
                child_list[i].g = child_list[i].parent.g + node.g
        /loop

        *check if any children is the goal
        *putting this here means that the algorithm doesn't care if the goal is a landmark
        for i in range(len(child_list)):
                if child_list[i] == goal:
                        return child_list[i].parent

        *checks if child is parent
        for i in range(len(child_list)):
                if child_list[i] == child_list[i].parent:
                        remove child_list[i] from child_list

        *check if child is out of bounds
        for i in range(len(child_list)):
                if child_list[i].x < -2 or child_list[i].x > 4 or
                child_list[i].y < -6 or child_list[i].y > 5:
                        remove child_list[i] from child_list
                endif
        /loop



*check if child is a landmark
        for i in range(len(child_list)):
```

```
            for j in range(len(landmark)):
                if child_list[i] == landmark[j]:
                    remove child_list[i] from child_list
                endif
            /loop
        /loop

        * check if child is on closed/open list, and if so, checks for lower f cost
        for i in range(len(child_list)):
            for j in range(len(closed_list)):
                if child_list[i] == closed_list[j]:
                    if child_list[i].f > closed_list[j].f:
                        remove child_list[i] from child_list
                    endif
                endif
            /loop
            for k in range(len(open_list)):
                if child_list[i] == open_list[j]:
                    if child_list[i].f > open_list[j].f:
                        remove child_list[i] from child_list
                    endif
                endif
            /loop

        * add child with lowest f to open list
        * if tie, add child with lowest h to open list

plot point for each node in closed list

plot path from goal (from point to point)
```

The A* algorithm is implemented via a class, Astar. The heuristic and cost function are calculated using Eqs 1-3, defined below:

$$node\ cost\ g(n) = 1\ or\ 1000 \qquad Eq.\ 1$$

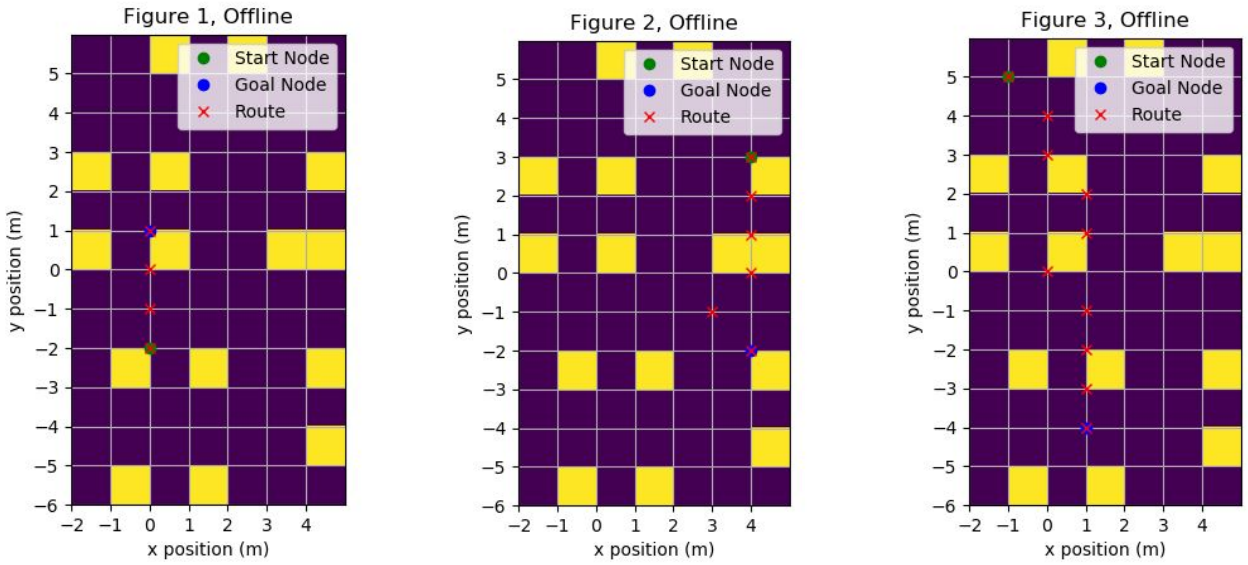
$$heuristic\ h(n) = \frac{\sqrt{(x_i - x_g)^2 + (y_i - y_g)^2}}{2} \qquad Eq.\ 2$$

$$cost\ function\ f(n) = g(n) + h(n) \qquad Eq.\ 3$$

(x_i, y_i) is the position of a given node, (x_g, y_g) is the goal position

The heuristic is admissible because it considers the path regardless of obstacles and because it is the shortest possible path to the goal (the shortest path between two points is a straight line). The distance is then halved, to preserve h(n) be less than f(n), while still preserving relative information to make informed cost decisions. The Astar class was originally a function, something that is more appropriate for what it is accomplishing, but was changed as a work around for a problem due to confusion between methods and functions within Python.

3. Offline pathfinding:



Yellow blocks are obstacles

Set 1, Figure 1
Start Node: (0.5, -1.5)
Goal Node: (0.5, 1.5)
Path: [(0, 1), (0, 0), (0, -1), (0, -2)]

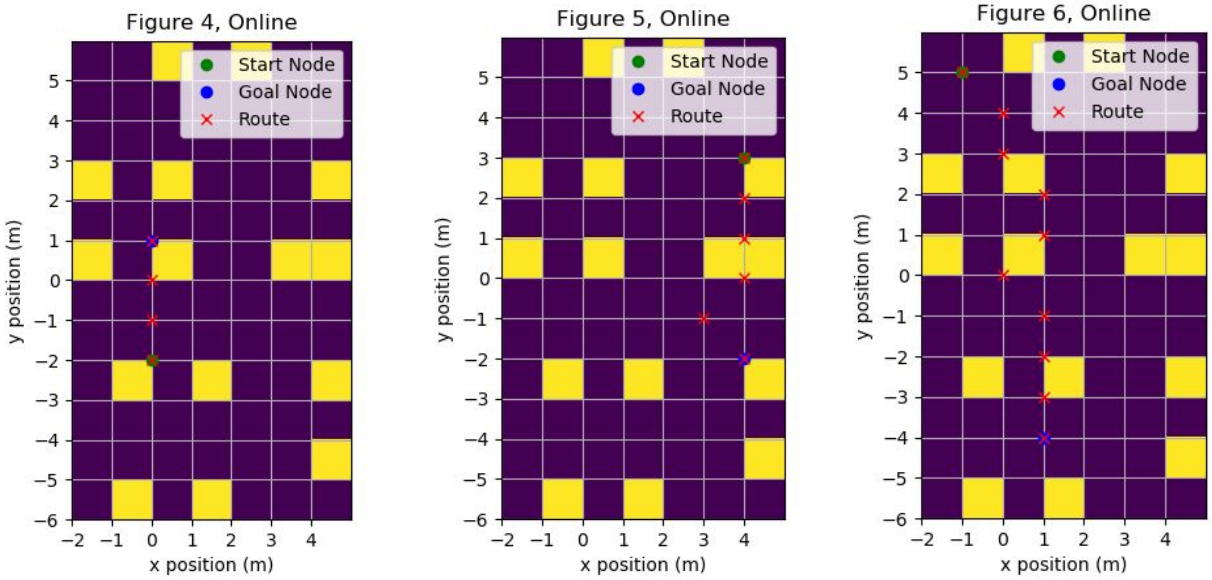
Set 2, Figure 2
Start Node: (4.5, 3.5)
Goal Node: (4.5, -1.5)
Path: [(4, -2), (3, -1), (4, 0), (4, 1), (4, 2), (4, 3)]

Set 3, Figure 3
Start Node: (-0.5, 5.5)
Goal Node: (1.5, -3.5)
Path: [(1, -4), (1, -3), (1, -2), (1, -1), (0, 0), (1, 1), (1, 2), (0, 3), (0, 4), (-1, 5)]

Figures 1-3 highlights that assigning the node values worked, and that paths are being created. The figures seem to show that the A* implementation ignores the barriers, but if that were the case then one would assume the lines would be straighter. Figures 2 and 3 demonstrate that the path is clearly going around what it perceives to be an expensive node. This suggests that the pathfinding algorithm is working but it's the grid that is off, likely as a result of the decision to stick strictly with coordinate points when assigning node costs. Plotting the grid with an offset might be a potential work around for this bug, though would not address all plotting bugs.

4. When the algorithm is online, that means the population of the open list is just the child with the lowest cost function. Changes to this code involved removing the validation step that vetted out the child nodes against nodes already in the open and closed sets. The children are already sorted by their cost function as a product of the method that created them. The open set then, rather than being appended, is redefined as the first index of the list of children created.

5. Online pathfinding



Yellow blocks are obstacles

Set 1, Figure 4
Start Node: (0.5, -1.5)
Goal Node: (0.5, 1.5)
Path: [(0, 1), (0, 0), (0, -1), (0, -2)]

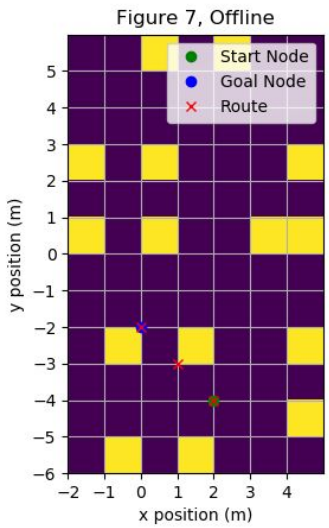
Set 2, Figure 5
Start Node: (4.5, 3.5)
Goal Node: (4.5, -1.5)
Path: [(4, -2), (3, -1), (4, 0), (4, 1), (4, 2), (4, 3)]

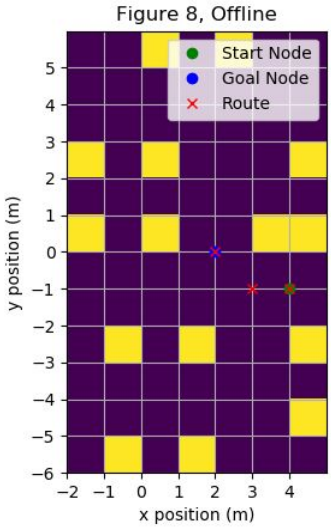
Set 3, Figure 6
Start Node: (-0.5, 5.5)
Goal Node: (1.5, -3.5)
Path: [(1, -4), (1, -3), (1, -2), (1, -1), (0, 0), (1, 1), (1, 2), (0, 3), (0, 4), (-1, 5)]

There were no changes in the path returned. Changes, however, are to be expected because of the nature of how the child is chosen to be moved to the open list, and the role of the closed list. The similarities in paths also point to how the algorithms decide what the best node is. Even though the children compare their cost functions, and if equal compare their heuristic functions, ultimately the list is sorted by the cost function and the lowest (first) child is chosen. If the closed list in the offline algorithm isn't being utilized then we would expect the offline and online paths to be the same.

6. Changing the grid size was not as simple as changing the parameter for the Grid class and caused an indexing error. It fails when trying to assign the node cost of a landmark, within the landmark function of the Grid class (lines 116-132). The function doesn't evaluate the 'if' statements to be true, thus never defining what *x* is. Resolving how the grid is created would likely resolve this issue.

7. As mentioned before, setting the grid cell size to 10 caused the program to crash. Using the same coordinates, but with a cell size of 1, gave these results:





Yellow blocks are obstacles

Set 1:
Start Node: (2.45, -3.55)
Goal Node: (0.95, -1.55)
Path: [(0, -2), (1, -3), (2, -4)]

Set 2:
Start Node: (4.95, -0.05)
Goal Node: (2.45, 0.25)
Path: [(2, 0), (3, -1), (4, -1)]

Set 3:
Start Node: (4.95, -0.05)
Goal Node: (2.45, 0.25)
Path: N/A

Set 3 caused the program to loop infinitely. To recreate it, change the range in line 497 from 2 to 3. Interrupting the loop with the keyboard command often places the interrupt in the validation or children function within the Astar class, suggesting that the program spends most of its time here. A hypothesis could be that the algorithm is circling a point or a set of points. To find out if that is the case, uncommenting the print statement (lines 304-307) reveals that the algorithm is bouncing between nodes (0,3) and (1,2).
When Astar is called the first thing it does is round the start and goal coordinates to their grid locations using the world_to_grid function, a product of the decision made in step 1. In this instance, the goal node became rounded to a coordinate that one of the landmarks is on. This makes the algorithm backtrack, recalculate its children, then redetermine the same child node. This further suggests that the way in which the grid was constructed was flawed. Aside from redesigning how the grid is constructed, a workaround could be implemented in the future to provide a threshold value for the heuristic, so that the algorithm will return a path once the threshold is reached.

8. Inverse Kinematic Controller
The following equations were pulled from a previous project:

Kinematic equations for inputs v and ω:

$$\dot{x} = v * cos() \quad \textit{Eq. 4}$$

$$\dot{y} = v * sin() \quad \textit{Eq. 5}$$

$$\_dot = \omega \quad \textit{Eq. 6}$$

θ in terms of $\dot{x}$, $\dot{y}$:

$$= arccos\left(\frac{\dot{x}}{v}\right) \quad \textit{Eq. 7}$$

$$= arcsin\left(\frac{\dot{x}}{v}\right) \quad \textit{Eq. 8}$$

Velocity in terms of $\dot{x}$, $\dot{y}$:

$$v = \sqrt{\dot{x}^2 + \dot{y}^2} \quad \textit{Eq. 9}$$

$$\omega = \frac{d}{dt} atan\left(\frac{\dot{y}}{\dot{x}}\right) \quad \textit{Eq. 10}$$

Acceleration in terms of v, ω:

$$a = \frac{v - v_{t-1}}{dt} \quad \textit{Eq. 11}$$

$$\alpha = \frac{\omega - \omega_{t-1}}{dt} \quad \textit{Eq. 12}$$

a is linear acceleration, α is angular acceleration, dt is the time step

Let K_v and K_w be gains for linear and angular velocity respectively, then

$$v = K_v * d * dt \qquad \textit{Eq. 13}$$

$$\omega = K_\omega * d * dt \qquad \textit{Eq. 14}$$

Because there are linear and angular limits for acceleration the robot has to make sure that those limits are not exceeded. That is accomplished using the following sketch:

if a > accel_limit:
    $v = v_{t-1}$ - *accel_max*dt*
else if a < accel_limit:
    $v = v_{t-1}$ + *accel_max*dt*

Similarly for angular acceleration:

if α > accel_limit:
    $\omega = \omega_{t-1}$ - *accel_max*dt*
else if α < accel_limit:
    $\omega = \omega_{t-1}$ + *accel_max*dt*

12. Simulations would face quite a bit of difficulty when applied to the real world. The ground isn't always flat or clean, obstacles aren't so clearly defined, and the environment itself is usually very dynamic. It's important to frame these factors within the context of the robot's purpose. For example, this simulation assumes the odometer readings to be 100% correct, and the ground to be completely free of debris. It also assumes the perfect geometry of the landmarks, but also itself. In the real world the robot cannot teleport, so its kinematics would need to be considered. The controller also makes assumptions. It assumes the data being sent to it is valid, and that the encoders for the actuators can precisely execute the commands.

References:

"Informed (Heuristic) Search Strategies." *Artificial Intelligence: A Modern Approach*, by Stuart J. Russell et al., Pearson, 2010, pp. 92–102.

Kaiser, Nate. "Astar." *GitHub*, 27 Dec. 2106, github.com/njkaiser/portfolio/tree/gh-pages/code/astar.

Mohta, Vibhakar. "A* Search Algorithm." *GeeksforGeeks*, 7 Sept. 2018, www.geeksforgeeks.org/a-search-algorithm/.

Swift, Nicholas. "Easy A* (Star) Pathfinding." *Medium*, Medium, 27 Feb. 2017, medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2.