MACHINE LEARNING & ARTIFICIAL INTELLIGENCE FOR ROBOTICS - HOMEWORK 1

A* Search and Navigation

October 28th, 2019

Maurice Rahme

1 Part A

1.1 Gridding the Task Space

In this exercise, the task space spans $[-2, 5]$, $[-6, 6]$ in x and y respectively. To grid this space, a 2-dimensional array was used to store each grid cell in array index coordinates. Landmarks from Data Set 1 of Homework 0 were used as obstacles. The obstacle cells were assigned a value of 1000, and were plotted using the imshow method of matplotlib which assigns a color-map to the plot based on fed values. Their cartesian coordinates were converted to grid index coordinates by identifying the number of cell-size increments required to represent them. For the given task space, a node at cartesian coordinates $(1.5, 0.5)$ has grid coordinates of $(1, 0)$ for a cell size of 1. A landmark takes up the whole cell. Each cell within this grid is a node. Nodes can house obstacles (landmarks in this case) and trajectory waypoints, including the start and goal points.

1.2 Offline A* Search

As a form of best-first search, A* evaluates neighbouring nodes using $f(n) = g(n) + h(n)$ where $g(n)$ (usually equal to 1, although $\sqrt{2}$ for diagonal paths showed better performance, for which plots are available in the bottom of the README) is the traversal cost from the current node n to the neighbour node being evaluated, and $h(n)$ is the heuristic cost, which estimates the cheapest path from the current node n to the goal node. Hence, for any given step, the neighbouring node with the lowest $f(n)$ cost will be chosen. If two nodes have the same $f(n)$ cost, they are evaluated by whichever has the lowest $h(n)$ cost [1].

A* search is both complete, meaning that it will always find a solution if one exists, and optimal, meaning that it will always find the cheapest solution in a given set of solutions. For this to be true, the heuristic cost function $h(n)$ must be admissible, meaning that it must never over-estimate the cost to reach the goal. Furthermore, $h(n)$ must be consistent, meaning that the $h(n)$ of parent nodes must always be non-decreasing. Since $g(n)$ represents the true cost to reach a node along a path, an admissible heuristic cost results in $f(n)$ never over-estimating the true cost to the goal [1]. In a grid that allows diagonal movement (i.e each node has a maximum of 8 neighbours), a viable heuristic is the Diagonal Distance Heuristic [2]:

$$h(n) = D1 * (dx + dy) + (D2 - 2 * D1) * min(dx, dy) \tag{1.1}$$

where dx and dy are the coordinate-respective distances of the evaluated node to the goal node in x, y coordinates, and $D1 = 1$. There are two options for setting $D2$. If $D2 = \sqrt{2}$, $h(n)$ is known as the Octile distance. If $D2 = 1$, $h(n)$ is known as the Chebyshev distance [2]. Experimentally, the latter choice for $D2$ exhibited better performance with a $g(n)$ of $\sqrt{2}$ for diagonal paths. However, for all $g(n) = 1$, The Octile distance performed best.

Following from this, the Offline A* Algorithm is presented [1]:

1. Add the start node to the open list.

2. Loop while open list contains at least one node or until the goal node is found:

   (a) Set the current node as the node on the open list with the lowest $f(n)$. If two nodes have the same, lowest $f(n)$, pick the one with the lowest $h(n)$ between the two. Remove the chosen node from the open list, and append it to the closed list.

(b) Acquire up to 8 neighbours of the current node. For each neighbour:

   i. If it is on the closed list, ignore it.

   ii. If it is an obstacle, ignore it (this is done instead of obstacle avoidance using a high $g(n)$ cost to prevent the path from going through obstacles if no alternative is available).

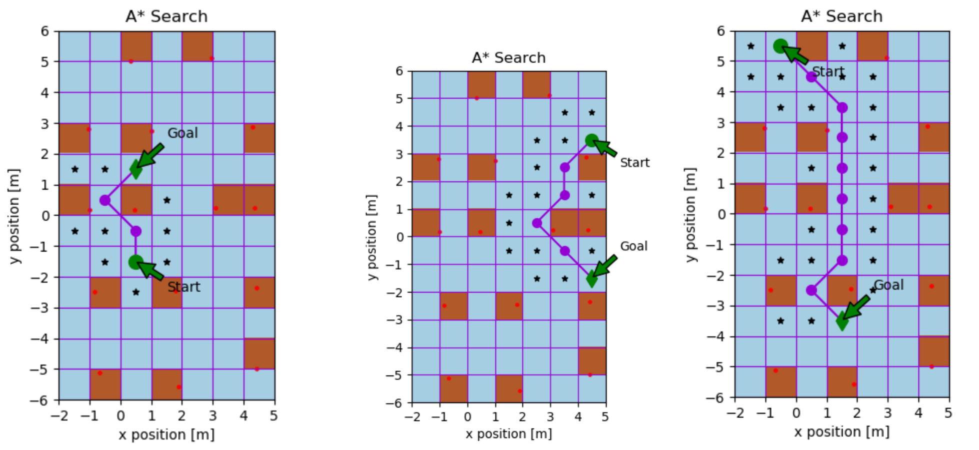
   iii. If it is on the open list, examine its $g(n)$ cost. If its $g(n)$ cost is higher than that derived by adding it as the next point in the path through the current node, then this new path is a better alternative for that neighbour node. In this case, re-assign its parent as the current node, and replace its $g(n)$ cost with the one calculated for the evaluation.

   iv. If the neighbour node doesn't satisfy any of the above conditions, add it to the open list, make the current node its parent, and calculate its $g(n)$, $h(n)$, and hence $f(n)$ costs.

3. If the goal node is found, break the loop and trace backwards from the goal node to the start node using the stored parents of subsequent nodes to build the path. If the loop breaks because the open list is empty, this means that no solution was found, and hence that none exists. After being identified, the current node is always added to the closed list to avoid backtracking in the algorithm, hence finding an optimal solution.

1.3 Planning Paths with Offline A*

Using the above algorithm, the following paths are planned:



(a) Start: (0.5, -1.5), Goal: (0.5, 1.5)

(b) Start: (4.5, 3.5), Goal: (4.5, -1.5)

(c) Start: (-0.5, 5.5), Goal: (1.5, -3.5)

Figure 1: Offline A* Paths for 1x1 Grid.

Here, the brown cells indicate obstacles, the blue cells indicate traversable nodes, and the black stars indicate expanded nodes to show the extent of the search, the purple points and lines indicate the planned path, the green circle indicates the start node, and the green diamond indicates the goal node. Notably, in figure 1 (c), a better path with less turns could go straight down before turning diagonally to avoid the obstacle and reach the goal, but due to the use of $g(n) = 1$ for all neighbour distances, and hence the Octile $h(n)$, this was not possible as the shortest distance to cover was prioritised in this case [2]. A $y = mx + b$ transformation is used to convert the path from grid to world coordinates, where x is the input grid coordinate, and y is the output world coordinate.

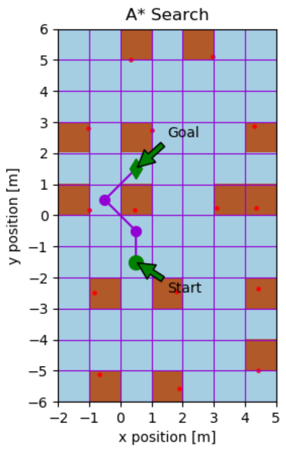
1.4 Online A* Search

The main loop for the online A* is nearly identical to its offline counterpart. Before stating the differences, some intuition is provided. The offline A* algorithm plans until the goal node is found before choosing an optimal path. This means that as the space of explored nodes expands, the algorithm has greater opportunity to find cheaper path solutions, and the next chosen 'current' node can come from anywhere in the open list. Effectively, in a real system, this means that the observer evaluating the path would need to teleport in real-time to achieve this search style.

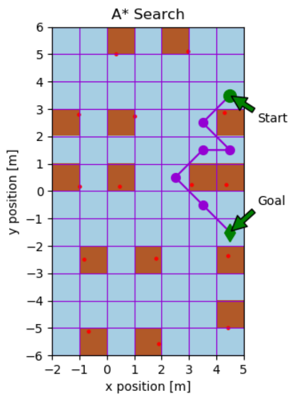
In this exercise, the observer is a wheeled mobile robot. Clearly, it cannot teleport, and so to model realistic searching, the next 'current' node in the path must be a neighbour of the current node at every search update. To achieve this behaviour, one modification is made: of all the neighbours evaluated, only the one with the lowest $f(n)$ cost (or, if two have the same $f(n)$, the one with the lowest $h(n)$) is added to the open list. This means that the open list always has a maximum size of 1 node in this modified algorithm.

1.5 Planning Paths with Online A*

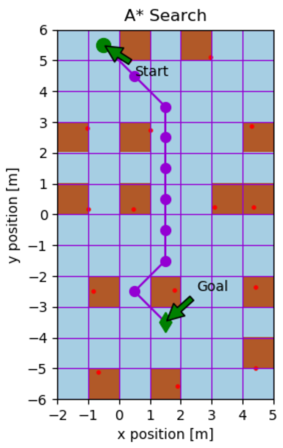
Using the above algorithm, the following paths are planned:



(a) Start: (0.5, -1.5), Goal: (0.5, 1.5)



(b) Start: (4.5, 3.5), Goal: (4.5, -1.5)



(c) Start: (-0.5, 5.5), Goal: (1.5, -3.5)

Figure 2: Online A* Paths for 1x1 Grid using Octile h(n).

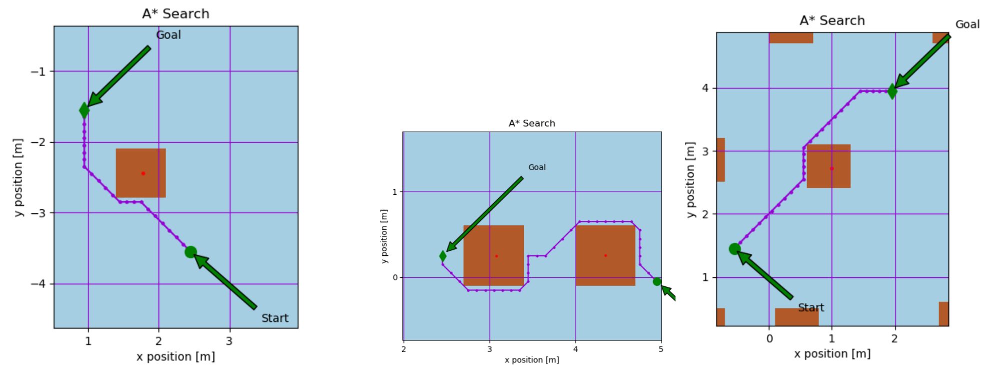
Figure 2 shows the resultant plots using the Octile heuristic with $g(n) = 1$. As mentioned before, this performs worse than the Chebyshev heuristic with $g(n_{diagonal}) = \sqrt{2}$. You may verify this in code by changing the value of $D2$ in line 170, and un-commenting line 182 of run.py. With the Octile heuristic, the path traced is intuitive for an online path, whereby obstacle-avoidance only occurs when the obstacle is one of the neighbour nodes. This is evident when comparing figure 2 (b) to figure 1 (b). However, this is because the Octile heuristic covers the smaller of the x and y distances first, whereas the Chebyshev prioritises covering the bigger of the two. Online paths do not show expanded nodes as only the best neighbour is added to the open list.

1.6 Reducing the Grid Size

A finer 0.1x0.1 grid is created to achieve more realistic path-planning, where the obstacles are inflated beyond their actual size to account for the robot's footprint. They are inflated by .3m in each direction by recording the grid coordinates of each obstacle, and assigning obstacle status to the 3 adjacent nodes laterally and diagonally in each direction. The resultant obstacles are still smaller than their 1x1 counterparts.

1.7 Planning Online Paths with a Fine Grid

Using the finer grid, the following paths are planned:



(a) Start: (2.45, 3.55), Goal: (0.95, 1.55)

(b) Start: (4.95, 0.05), Goal: (2.45, 0.25)

(c) Start: (0.55, 1.45), Goal: (1.95, 3.95)

Figure 3: Online A* Paths for 0.1x0.1 Grid.

As with the coarse grid plots, the online A* tries to find the optimal path without prior knowledge of obstacles. Once an obstacle is reached, the algorithm finds the best path around it. The best example of this is figure 3 (b), where after moving around the first obstacle, the algorithm attempts to plan in a straight line until this path is obstructed by the second obstacle, causing the path to move around it.

2 Part B

2.1 Designing the Motion Model

The subject being tracked by this particle filter is a mobile robot moving on a 2D plane. Its equations of motion assume a unicycle model whereby motion can be achieved by a combination of linear and angular velocity, thus mapping the robot as a 3 degree of freedom system represented by coordinates x, y, and θ. Hence, the nonlinear motion model is the following for each iteration of time elapsed dt [3]:

$$x_t = x_{t-1} + v * (1 + \epsilon_v) * cos(\theta_{t-1}) * dt \tag{2.1}$$

$$y_t + = y_{t-1} + v * (1 + \epsilon_v) * sin(\theta_{t-1}) * dt \tag{2.2}$$

$$\theta_t + = \theta_{t-1} + \omega * (1 + \epsilon_\omega) * dt \tag{2.3}$$

where v is linear velocity, ω is angular velocity, and ϵ is the noise for each velocity, linear and angular. The model is nonlinear due to the sin and cos relationship between the robot coordinates and velocity [3].

A Proportional (P) controller actuates the robot with minimal position overshoot, whereby linear v and angular w velocity are set as follows:

$$v_t = P_v * \sqrt{(dx)^2 + (dy)^2} \tag{2.4}$$

$$w_t = P_w * (atan2(dy, dx) - \theta_t) \tag{2.5}$$

where $dx = x_{goal} - x_{robot}$, $dy = y_{goal} - y_{robot}$, P_w and P_v are the proportional gain constants for w amd v respectively, and θ_t is the robot's current heading relative to the grid's x-axis. The P controller causes increased actuation for a higher error, and vice versa. This results in stable behaviour for a simulated robot, but does not inherently limit maximum linear and angular acceleration.

To combat this, the resultant acceleration for a given velocity command at each time step is computed as:

$$\dot{v} = \frac{(v_t - v_{t-1})}{dt} \tag{2.6}$$

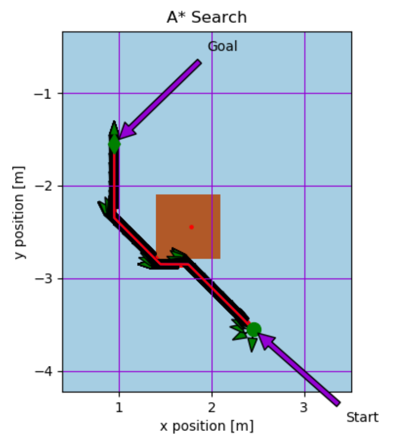
$$\dot{w} = \frac{(w_t - w_{t-1})}{dt} \tag{2.7}$$

Then, taking $\dot{v}$ as an example (the same follows for $\dot{w}$ with respect to maximum angular acceleration, $\dot{w}_{max}$), if it is greater than the maximum linear acceleration, $\dot{v}_{max}$, it is set as $v_t = v_{t-1} + \dot{v}_{max} * dt$. However, if it is less than the negative of $\dot{v}_{max}$, it is instead set as $v_t = v_{t-1} - \dot{v}_{max} * dt$.

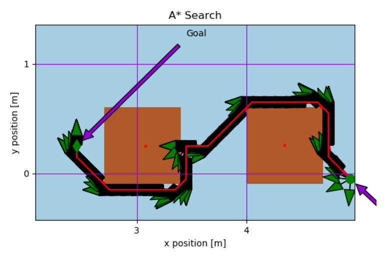
Hence, maximum linear and angular accelerations are limited at every time step. However, this allows the simulated robot's velocities to reach infinity, which is unrealistic. If the maxima were provided, it would be simple to cap the robot's velocities in the simulation, by setting them equal to the maximum when necessary.

2.2 Following Pre-Planned Paths

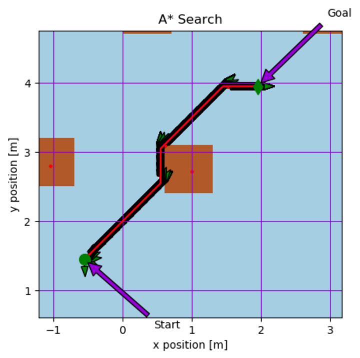
Using the fine grid and the online algorithm for realistic robot motion, the following paths were planned. In addition to the previous plot elements, the robot path is shown in red, and the heading arrows are plotted in green at each time step. Noise sampled from a Gaussian distribution with a standard deviation of 0.1 was added to each control input.



(a) Start: (2.45, 3.55), Goal: (0.95, 1.55)



(b) Start: (4.95, 0.05), Goal: (2.45, 0.25)



(c) Start: (0.55, 1.45), Goal: (1.95, 3.95)

Figure 4: Robot following Online A* Paths for 0.1x0.1 Grid.

The main challenge in executing this control method is to find a balance between P_v and P_w. When P_v is too high, the robot's motion becomes unstable; because the robot's starting condition requires a $-\pi/2$ heading, with simultaneous v and w actuation, the robot will move away from the goal before it can fully correct its heading. Referencing equation 2.4, an increase in distance to goal causes an increase in linear velocity, so the robot will move away from the first waypoint, and will not complete the path well. For reliable robot motion, $P_v = 0.02$, and $P_w = 0.6$. Below is an example where $P_v = P_w = 0.02$; the goal node is still reached, but the path is sub-optimal. Note that the arrows are plotted so densely that they appear black sometimes. When $P_w < 0.7 * P_v$, the path never completes.

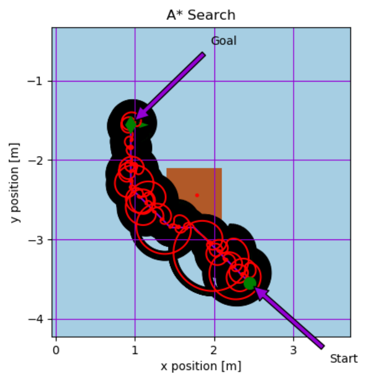
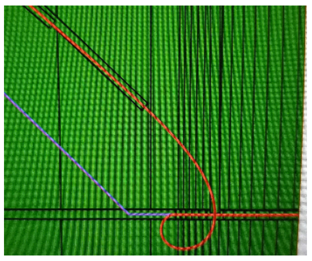
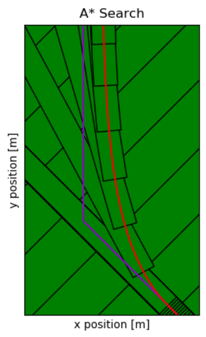


Figure 5: Path (a) above with $P_w = 0.8 * P_v$.

Another issue was encountered when the path commanded sudden changes in direction, whereby the bearing error term in equation 2.5 would go above or below $\pm\pi$, resulting in path loops instead of simple turns as seen in figure 6 (a).


(a) Undesirable Looped Robot Path (red).



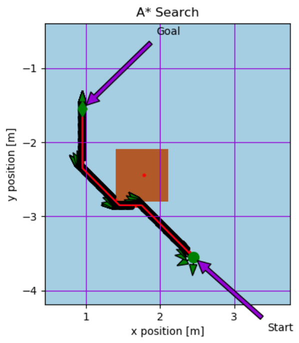
(b) Zoomed plot to distinguish Planned Path (purple) and Robot Path(red).

Figure 6: Motion considerations.

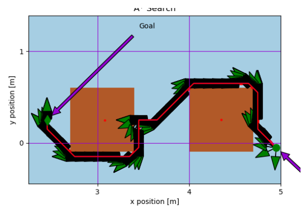
This was corrected by subtracting 2π from the bearing error term if it breached π, and adding 2π to it if it breached $-\pi$. After these corrections, the Proportional control scheme allows for good motion response from the robot, where the robot's path in red remains close to the planned path in purple, even on corners. Figure 6 (b) above shows path with a 0.002m error in the x coordinate at the corner point:

2.3 Simultaneous Planning and Movement

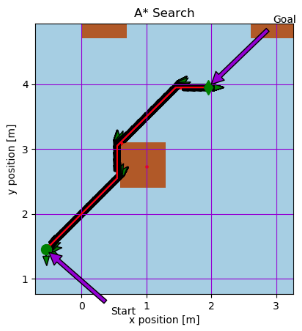
The A* algorithm is modified again in this step. Instead of looping until the goal node is found, or until the open list is empty, the modified algorithm loops once, then commands the robot to move to the waypoint. Then, once the robot has reached the waypoint within a threshold 0.005, the last saved 'current node' position is overwritten with the robot's actual reached position converted to grid coordinates, and the algorithm re-plans from this new node's neighbours. This is done until the robot's current position is within a threshold 0.005 of the goal.



(a) Start: (2.45, 3.55), Goal: (0.95, 1.55)



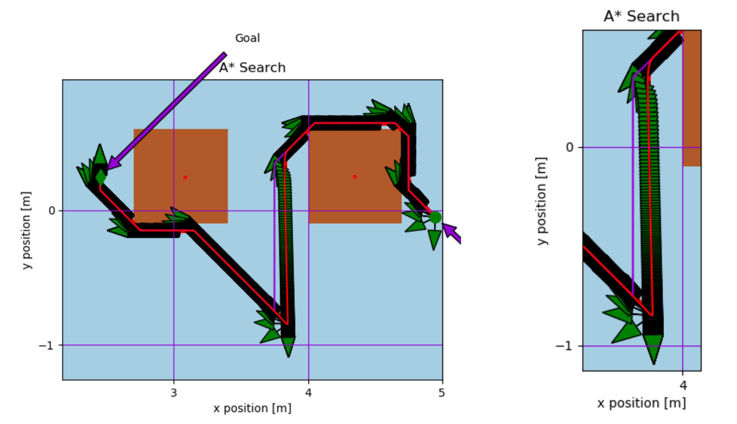
(b) Start: (4.95, 0.05), Goal: (2.45, 0.25)



(c) Start: (0.55, 1.45), Goal: (1.95, 3.95)

Figure 7: Robot simultaneously Planning and Moving on the 0.1x0.1 Grid.

Thanks to the P controller and the small position error threshold of 0.005, the robot never actually landed in an un-intended grid cell, so as a dual to the above (b), the 15^{th} path iteration was perturbed by -1.2m in the y direction to create a re-plan case. This can be achieved by un-commenting lines 1014 and 1015 in run.py.



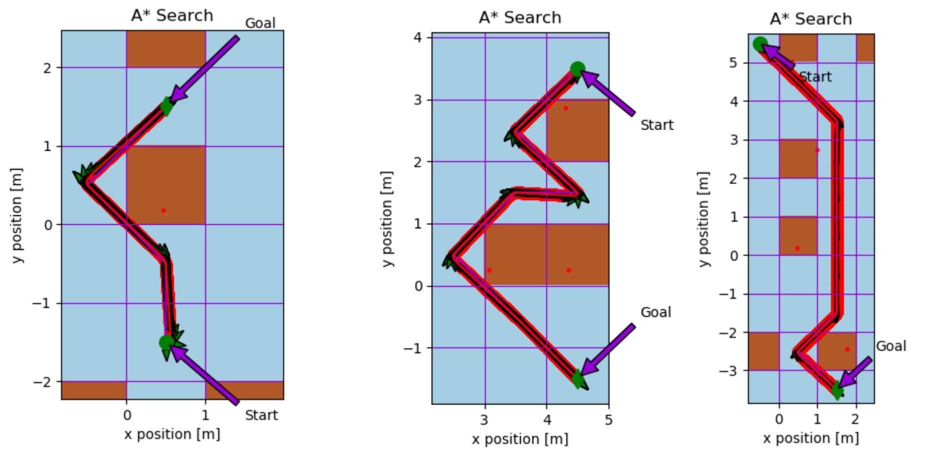
(a) The path is successfully re-planned from the cell in which the robot landed

(b) Zoom on expected waypoint versus perturbation

Figure 8: Re-plan case for path in figure 8 (b).

2.4 Simultaneous Planning and Movement for Low-Resolution Coordinates

Using the low resolution coordinates from step 3 in the exercise for the coarse (1x1) resulted in much quicker paths than for the fine-grid implementation due to the Proportional controller's tendency to actuate more aggressively in proportion to increased error (distance and heading to waypoint). However, as can be seen in figure 10 (b), the robot follows the planned path less accurately, and intersects te obstacle cell. This is not a major issue as the obstacle is actually smaller than the cell, and P_w and P_v can be re-tuned.

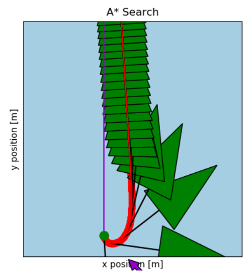


(a) Start: (2.45, 3.55), Goal: (0.95, 1.55)

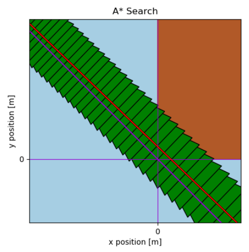
(b) Start: (4.95, 0.05), Goal: (2.45, 0.25)

(c) Start: (0.55, 1.45), Goal: (1.95, 3.95)

Figure 9: Robot simultaneously Planning and Moving on the 1x1 Grid.



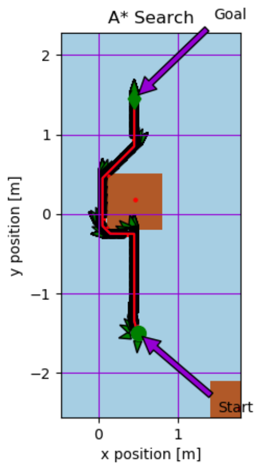
(a) Zoomed on start node



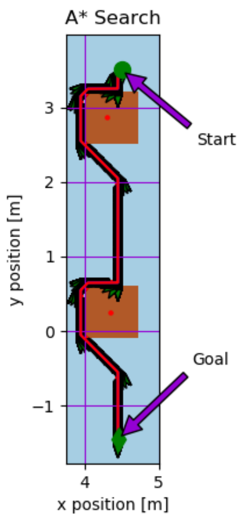
(b) Zoomed on intersect with obstacle

Figure 10: Zoomed on (a) for above plot.

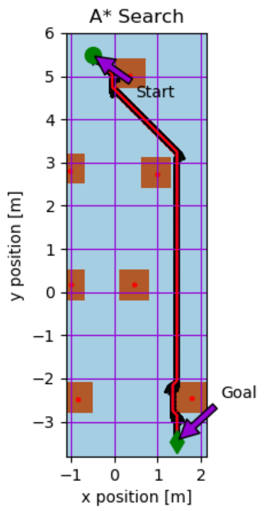
The fine grid provides slower but more accurate paths. However, the goal node is not exactly reached, as the .0 resolution coordinates cannot be resolved in a 0.1x0.1 grid, so there is a 0.05m offset in each coordinate axis. This means that the 'goal reached' threshold using low-resolution coordinates is more relaxed (0.08 instead of 0.005) than for high-resolution coordinates. For example, in the figure below, the goal at $(0.5, 1.5)$ is actually reached at $(0.45, 1.55)$. The fine grid also allows for more accurate tracking of higher-resolution coordinates, both in terms of representing the coordinates on the grid, and actuating the robot to follow them well. The fine grid often results in shorter paths, and has a smaller chance of getting stuck as it has more options for going around obstacles. Furthermore, the fine grid represents the landmarks (whose true position is shown as a red dot in the plots) more accurately. However, the coarse grid path demands less turns than the find grid path in most cases. Finally, the coarse grid is quicker as the number of planning stages is cut down significantly. Ultimately, the choice between a coarse and fine grid comes down to the specific needs of the environment coupled with the robot's sensing resolution and agility in motion.



(a) Start: (2.45, 3.55), Goal: (0.95, 1.55)



(b) Start: (4.95, 0.05), Goal: (2.45, 0.25)



(c) Start: (0.55, 1.45), Goal: (1.95, 3.95)

Figure 11: Robot simultaneously Planning and Moving on the 0.1x0.1 Grid.

2.5 Real-World Considerations

The first consideration in translating this simulation into a real-world system is that the robot's equations of motion will not hold if it not a unicycle robot. A more common differential-drive or kinematic car robot will have more complex equations of motion, which may require actuating each wheel individually.

Another consideration is that although noise is added to perturb the velocity commands, it is assumed that the robot's location is always known. This is likely not the case, as Homework 0 showed that dead-reckoning is an inaccurate way of measuring a wheeled robot's location post-actuation. To remedy this in a real system, a Particle or Unscented Kalman Filter could be implemented using the provided obstacles as landmarks by taking regular measurements as often as possible to better estimate the robot's location, and hence move in a path less prone to collision.

Furthermore, the acceleration limits provided help model a realistic robot, but no velocity ceilings were given, so for a given velocity command, it cannot be expected that the robot will be actuated at the given rate. Maximum linear and angular velocities must be determined and accounted for as was suggested in section 2.1. Additionally, a real robot with more complex equations of motion will likely require a more sophisticated controller to move reliably, such as a PID controller. Finally, a good path planning algorithm should include increased cost for nodes with difficult traversal properties, such as a slippery or cobbled floor for a wheeled robot, where the former would cause erroneous motion and increase the chance of collision, and the latter would cause mechanical vibrations which might damage its internals and cause the sensor readings to become skewed.

3 Citations

[1] Russel, Stuart Norvig Peter. Artificial Intelligence: a Modern Approach. PEARSON, 2018.

[2] Amit. "Pathfinding: Heuristics." Game Theory, Stanford, 2018, theory.stanford.edu/ amitp/ GameProgramming/Heuristics.htmlexact-heuristics.

[3] "Robot Motion." Probabilistic Robotics, by Sebastian Thrun et al., MIT Press, 2010.