EMARO+ M1
European Master on Advanced Robotics

Project Report

24.06.2019

Project 17 - Nonholonomic Path Planning with A*

Astha Gupta
Sthithpragya Gupta

Supervisor(s)

Olivier Kermorgant, Associate Professor of Robotics at Centrale Nantes, France
Alexandre Goldsztejn, CNRS Research Associate

CONTENTS

1. INTRODUCTION

This report deals with the work done as a part of the M1 CORO Group Project course, carried out under the guidance of Prof. Olivier Kermorgant and Prof. Alexandre Goldsztejn. The report documents the progress and findings of the work done to explore the topic of using A* algorithm for planning the path of a non-holonomic mobile robot. The code for the approach has been developed over an initial architecture provided by Prof. Kermorgant.

1.1. PROBLEM STATEMENT

A* is a well-known algorithm widely used for solving graph-based search and path-finding problems. While the traditional algorithm is suited for graph search, it does not seem to be well suited for actual planning because it requires a discretization of the environment and cannot really take into account the nonholonomic constraints governing a mobile robot's motion.

Though the traditional path-planning approach of discretizing the environment space and using the A* algorithm may not be suited, different other approaches can be devised which utilise the A* technique and also satisfy the nonholonomic constraints governing the robot's motion. These approaches may promise satisfactory performance. However, this area remains relatively unexplored.

1.2. OBJECTIVE

The objective of the present study is to explore a variation of the traditional path-planning approach which involves discretization of not the environment, but the control inputs to solve path planning problem for a (2,0) robot. An expected outcome is that the discretization happens in lower dimension than when done on the environment, and that the generated path complies with the nonholonomic constraints.

2. BACKGROUND

The project heavily borrows from the elements of A* algorithm and non-holonomic constraints which govern the motion of a mobile robot – (2,0) robot in this case – and works towards an integration of the ideas to solve the path-planning problem.

2.1. A* SEARCH ALGORITHM

The A* algorithm is a best-first search algorithm which assigns weights to nodes of the graph taking into account a measure of distance of that node from the start and an approximation of the distance to the goal computed via a heuristic method.

Elements of A* algorithm:

- $\boldsymbol{n}$: current node
- $\boldsymbol{g}$: measure of actual distance from Start node
- $\boldsymbol{g(n)}$: measure of the actual distance between current and Start nodes
- $\boldsymbol{h}$: heuristic method to approximate distance to Goal node – e.g. Euclidean distance
- $\boldsymbol{h(n)}$: approximate distance between current and Goal nodes
- $\boldsymbol{f(n)}$: weight assigned to the current node where $\boldsymbol{f(n) = h(n) + g(n)}$

Admissibility of the heuristic – The heuristic function never over-estimates the actual distance between the current and the Goal nodes. If the heuristic function used in A* is admissible, the solution will be the optimal solution. Else, the solution may not necessarily be optimal.

Effect of h on performance – Lower the value of $\boldsymbol{h(n)}$ is, slower the algorithm runs but produces a solution which is optimal (if $\boldsymbol{h}$ is admissible) or closer to the optima. Higher $\boldsymbol{h(n)}$ is, faster the algorithm runs, but at the cost of optimality of the solution. It is important to find a heuristic which performs sufficiently well on both fronts – speed and optimality of the solution. Thus, different choices of $\boldsymbol{h}$ result in A* performing differently.

2.2. (2,0) TYPE MOBILE ROBOT

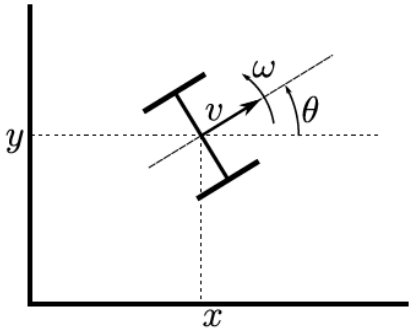

Figure 1 Schematic of a (2,0) robot - borrowed from Prof. Kermorgant's lecture slides

Parameters of a (2,0) mobile robot:

- State of the robot: $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$
- Parameters: (track gauge **'$\boldsymbol{t}$',** wheel radius **'$\boldsymbol{r}$')**
- Control input: (angular velocity of right wheel **'$\boldsymbol{\omega_R}$',** left wheel **'$\boldsymbol{\omega_L}$'**)
- Robot Pose: $(\boldsymbol{\omega_R}, \boldsymbol{\omega_L}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$

Equations of motion:

- $\boldsymbol{v = (r\,\omega_R + r\,\omega_L)/2}$
- $\boldsymbol{\omega = (r\,\omega_R - r\,\omega_L)/t}$
- $\boldsymbol{\dot{x} = v\cos\theta;\ \dot{y} = v\sin\theta;\ \dot{\theta} = \omega}$

Robot architecture – The (2,0) mobile robot is a two degrees-of-freedom system which contains two independently actuated fixed type wheels (left and right). The robot is controlled via setting the angular velocities of the left and right wheels which determine the overall behaviour of the robot.

Non-holonomic constraints – Constraints which can't be expressed solely as functions of the coordinate parameters of the system and time elapsed are called non-holonomic constraints. For mobile robots, these constraints dictate what types of motions are feasible. In case of a (2,0) type mobile robot, at any instant, the non-holonomic constraints render the robot capable of:

- Moving in a straight line along the local axis $\boldsymbol{X_M}$ of the mobile frame at linear velocity $\boldsymbol{v}$
- Rotating about its centre along the $\boldsymbol{Z_M}$ axis at angular velocity $\boldsymbol{\omega}$
- A combination of both

3. METHODOLOGY

The control input of the robot (ω_R, ω_L) has been discretised into multiple intervals. Each of these is used to make a possible child node for construction of the graph to be then explored by the A* algorithm and find the optimal path from Start to Goal. Each child node represents a unique robot position and orientation (hereby collectively referred to as a single metric - Robot Pose) and has been identified using the following 5 parameters – $(\omega_R, \omega_L, x, y, \theta)$

The methodology followed to plan the path between the Start and Goal nodes can be broken down into the following steps:

1. Discretise (ω_R, ω_L) of the parent node into various intervals (within the maximum permissible limits on the velocities)
2. Use all of these intervals to create possible child nodes (the child node may not necessarily be formed as the path between parent and child may be blocked by an obstacle OR the child itself maybe representing a position containing an obstacle)
3. Assign weights to the created child nodes in accordance with A* convention
4. Run an iteration of the A* algorithm
5. This continues until a path to the goal is found

3.1. SKELETON OF THE ARCHITECTURE

In order to implement the aforementioned steps, an architecture was designed to run the logic.

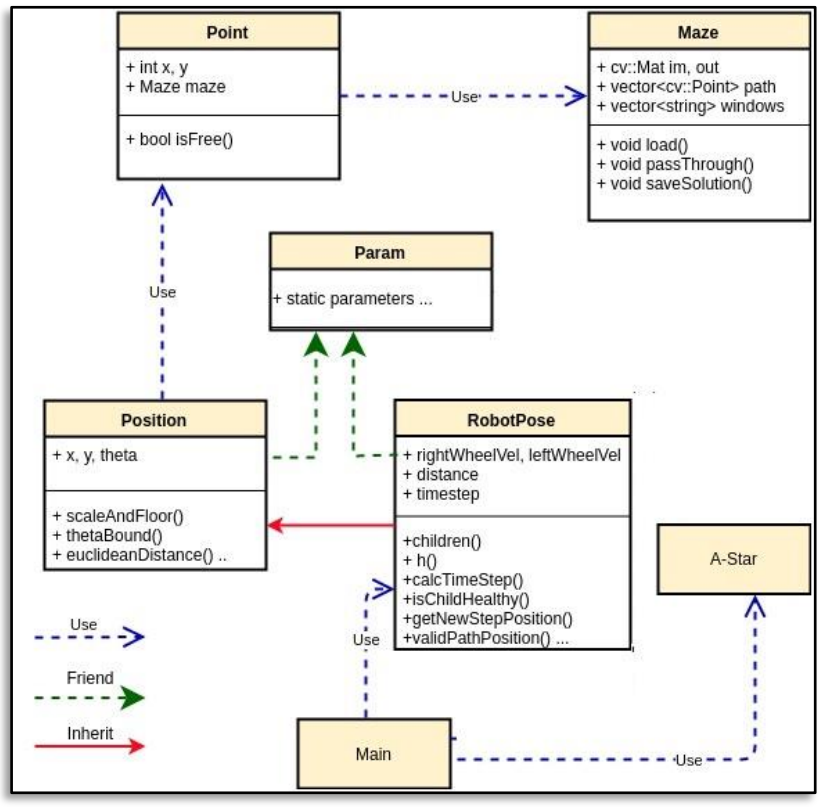

Figure 2 Architecture of the program generated using https://www.draw.io/

3.2. PROGRAM OUTLINE

The logic to implement the path planning problem has been distributed amongst various classes and methods of the program which work congruously to plan a path for the mobile robot while respecting the nonholonomic constraints.

3.2.1. DISCRETIZATION OF THE CONTROL INPUT

The **generateVelChoices** method returns a vector of possible control input wheel velocity pairs **(rightWheelVel, leftWheelVel)** which can be used to explore child nodes from the current parent node **RobotPose.** The method generates left and right wheel velocities within **velocityIncrementLimit** of the parent node's left and right wheel velocities while respecting the **[wheelVelocityMin, wheelVelocityMax]** range of wheel velocities (same for both wheels).

3.2.2. GENERATION OF A POSSIBLE CHILD NODE

Starting from $(\omega_R, \omega_L, x, y, \theta)_{parent}$ node and traversing through the environment at a possible control input pair (from **generateVelChoices**) for **timeStep** duration generates a possible child **Position**. This task is carried out via **getNewStepPosition** method.

3.2.3. VALIDATION AND CREATION OF CHILDREN NODES

The **validPathPosition** method assesses a possible child position to check if it actually represents a position which can be reached by the robot while travelling in the environment. This method is repeated for all possible children nodes generated from a common parent node. The **children** method converts all these valid nodes into instances of **RobotPose** and stores the distance of each valid child node from its parent in the **distance** attribute. A vector of all these **RobotPose** nodes is returned.

3.2.4. CALCULATING THE HEURISTIC DISTANCE

The **h** method computes the approximate distance between the current node and the goal node. This method has been implemented to assign heuristic distance values to all the children nodes that were created using the **children** method.

3.2.5. ASSIGNING WEIGHTS TO THE NODES

All the nodes to be explored (including the newly created child nodes) are assigned weight. The weight assigned to a node is the sum of the heuristic value (computed via the **h** method) and the distance from Start node (cumulative of the **distance** attributes) computed via **distTravelled** method.

3.2.6. VISUALISATION OF THE RESULT

Post the computation of the planned path, it is visualised via the **reconstructPath** method which in-turn uses the **print** method to illuminate the pixels in the maze which correspond to the planned path.

3.3. ASSUMPTIONS

In order to implement the logic, some assumptions have been made viz –

- During the **timeStep** duration of robot's motion, it moves with constant velocities.
- The **obstacleCheckInterval** is sufficiently small enough to detect any blockage in the path between 2 nodes.
- All dimensions assumed in cm and angles in radians.
- Each pixel represents **scaleFactor** * 1cm along the image axis.

4. PROGRESS

The progress made during the project has been divided into 5 phases with each phase improving over the previous one. The development of different aspects of the program over phases, changes made, subsequent observations and inferences have been documented below.

4.1. PHASE I

In the first phase of project, work was done to explore the topic and make an initial architecture to run the logic.

4.1.1. WORK DONE

generateVelChoices():

$(\boldsymbol{\omega_i})_{child} = (\boldsymbol{\omega_i})_{parent} \pm \boldsymbol{k} * \textbf{velocityIncrementStep}$ where $i \in \{right, left\}$ and k is an integer such that:

- $\boldsymbol{max}[(\boldsymbol{\omega_i})_{parent} - \textbf{velocityIncrementLimit, wheelVelocityMin}] \leq (\boldsymbol{\omega_i})_{child}$
- $\boldsymbol{min}[(\boldsymbol{\omega_i})_{parent} + \textbf{velocityIncrementLimit , wheelVelocityMax}] \geq (\boldsymbol{\omega_i})_{child}$

robotVelocity(): Converts $(\boldsymbol{\omega_R}, \boldsymbol{\omega_L})$ to $(\boldsymbol{v}, \boldsymbol{\omega})$ following the equation of motion

getNewStepPosition():

$Child_\theta = Parent_\theta + \boldsymbol{\omega} * \textbf{timeStep}$

if($\boldsymbol{\omega}$!= 0){
$Child_X = Parent_X + (v * sin(\omega * timeStep))/\omega$
$Child_Y = Parent_Y + (v * (1 - cos(\omega * timeStep)))/\omega$
}
else{
$Child_X = Parent_X + (v * cos(Parent_\theta))$
$Child_Y = Parent_Y + (v * sin(Parent_\theta))$
}
return Position($Child_X$, $Child_Y$, $Child_\theta$)

validPathPosition():

- map Child.x and Child.y to maze's pixel location with help of **scaleAndFloor()**
- check if the pixel location is blocked or not:
  - if yes: return false
  - if no: return true

children():

- generate all the wheel velocity choices $(\boldsymbol{\omega_R}, \boldsymbol{\omega_L})$ using **generateVelChoices()**
- for all the wheel velocity choices:
  - calculate the robot velocity in terms of $(\boldsymbol{v}, \boldsymbol{\omega})$ using **robotVelocity()**

- calculate the (x, y, θ) of the possible child node that can be reached from Parent node using **getNewStepPosition()**
- check if this possible child node is a valid position using **validPathPosition()**
  if valid : then add to children set
  otherwise : continue, don't add to children set

- return the set of the children created

h(): returns the Euclidean distance between the current position and goal node

distTravelled(): returns v * **timeStep**

print(): calls **passThrough()** method for each node of the generated path

4.1.2. OBSERVATIONS AND ISSUES FACED

The program takes a very long time to generate a path between the start and the goal node. To plan a path as small as between (0,0) and (5,5), the program takes around 300 seconds. This happens because we use the Euclidean distance as the heuristic which does not take into account the information in θ parameter of the node. Consequently, the program has to explore a lot of nodes with similar heuristic distances leading to a very long runtime.

4.1.3. SCOPE OF IMPROVEMENT

To device a better heuristic which takes into account contributions from (x, y, θ) parameter of the node position. This may help in reducing the number of overall nodes to be explored thereby improving the speed and reducing runtime.

4.2. PHASE II

4.2.1. WORK DONE

h(): Instead of using Position to calculate the heuristic distance, the new heuristic approximates the distance between the current node and the goal node as a measure of average wheel spun by the robot.

Since a (2,0) type mobile robot is capable of moving in a straight line along the local axis X_M of the mobile frame as well as rotating about its centre along the Z_M axis, the heuristic assumes takes the average wheelspin from 3 segments into consideration:

1. Aligning from $\theta_{current}$ to the direction from $(x, y)_{current}$ to $(x, y)_{goal}$
2. Traversing the straight line distance between $(x, y)_{current}$ and $(x, y)_{goal}$
3. Aligning to θ_{goal}

distTravelled(): Average wheel spun during motion from parent to current node.

- return (absolute(ω_L* **timeStep**) + absolute(ω_R* **timeStep**))/2

4.2.2. OBSERVATIONS AND ISSUES FACED

- As compared to before, the program runs much faster as the inclusion of $\boldsymbol{\theta}$ information in the heuristic results in much less nodes being explored overall. The runtime to find a path between (0,0,0) and (5,5,0) - $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$ is around 40 seconds.
- However, the heuristic is not admissible. While the program may generate a path, it may not necessarily be the optimal one. The number of nodes to be explored are still considerably high as the number of nodes explode exponentially.
- Currently, the **print()** method only highlights those pixels which correspond to a child node generated during path finding and not the complete path between the nodes.
- The **validPathPosition()** method only checks if the created child node is in a blocked position. However, it does not check if the path to a valid child node may be blocked i.e. it does not prevent leaping over an obstacle.

4.2.3. SCOPE OF IMPROVEMENT

- The performance of the code can be improved by reducing the overall nodes to be explored thereby reducing the height of tree to be explored. This may be achieved via by using a variable time step. Instead of fixing the value of **timestep**, we can set it to have a heuristic dependant value so that we may use a bigger time step when the goal is far and cover bigger distance initially
- Check if the robot has leaped over an obstacle for it be a valid path. As an added benefit, this too may reduce the overall nodes to be explored by removing a stem of nodes representing an incorrect solution.
- Print the entire path between two subsequent nodes by checking the robot position at a small interval for better visualisation.

4.3. PHASE III

4.3.1. WORK DONE

validPathPosition():

- given: Parent Node; $(\boldsymbol{v}, \boldsymbol{\omega})$ and **timestep**
- consider a small time interval **obstacleCheckInterval**
- while(timePassed < **timeStep**):
    - timePassed = timePassed + **obstacleCheckInterval**;
    - generate this intermediate position reached after timePassed while moving with $(\boldsymbol{v}, \boldsymbol{\omega})$ using **getNewStepPosition()**
    - check if the position maps to a free pixel in maze:
        - if yes: continue, check the rest of the path
        - otherwise: return false

print():

- given: Parent Node; $(\boldsymbol{v}, \boldsymbol{\omega})$ and **timestep**

- while(timePassed < **timeStep**):
    - timePassed = timePassed + **obstacleCheckInterval**;
    - generate this intermediate position reached after timePassed while moving with $(\boldsymbol{v}, \boldsymbol{\omega})$ using **getNewStepPosition()**
    - highlight the pixel corresponding to this intermediate positon via **passThrough()**

4.3.2. OBSERVATIONS AND ISSUES FACED

- The overall number of nodes generated is fewer since the potentially incorrect stems of the tree are removed.
- Computation cost per node creation has increased, since for each node to be validated, the path between the parent and the potential child have to inspected at **obstacleCheckInterval**
- Overall, there is a trade-off between the no of nodes generated and runtime required
- Print function also adds to the computation time, but path can now be better visualised.

4.3.3. SCOPE OF IMPROVEMENT

- A better method to check for obstacles in the path between two nodes may be devised which can reduce the overall computations required. This method maybe based on prior knowledge of the obstacles.
- Augmenting the print functionality to illustrate the robot orientation $\boldsymbol{\theta}$ as well.

4.4. PHASE IV

4.4.1. WORK DONE

fillIntervalVector():

- Create a vector of possible durations suitable for the **timeStep** using two different tuning parameters - **bigTimeStep** and **intervalCount.**
- **bigTimeStep** is the largest possible value of **timeStep** and **intervalCount** is the total number of possible values **timeStep** can assume.
- Make small intervals of duration **(bigTimeStep/intervalCount)**
- Use these to make a vector of possible **timeStep** candidates by equally spacing **intervalCount** steps at previously calculated small interval
- for example if **bigTimeStep** = 2 and **intervalCount** = 4
    - then return vector(2, 1.5, 1, 0.5)

fillRadiusRegionLimitVector():

- Based on the tuning parameters - **highestThreshold** and **intervalCount**
- **highestThreshold** represents maximum heuristic distance which triggers smaller time steps
- Creates a vector storing **intervalCount** number of heuristic distance limits, with each interval corresponding to a different **timeStep**
- smallestStep = **highestThreshold** /(**intervalCount** - 1)

- for(i = **intervalCount** - 1; i >= 0; i--){
    - vector.push(i*smallestStep)}
- return vector

calcTimeStep():

- Create a vector of the possible timestep intervals using **fillIntervalVector()**
- Create a vector of distance regions using **fillRadiusRegionLimitVector()**
- From this, check the region which corresponds to the heuristic distance of the goal node from current node and set select the corresponding **timeStep** value from **fillIntervalVector()**
- For instance, if heuristic is very large, implying that goal is very far away, consider a larger value for **timeStep**

children():

Now uses the **calcTimeStep()** to set the **timeStep**

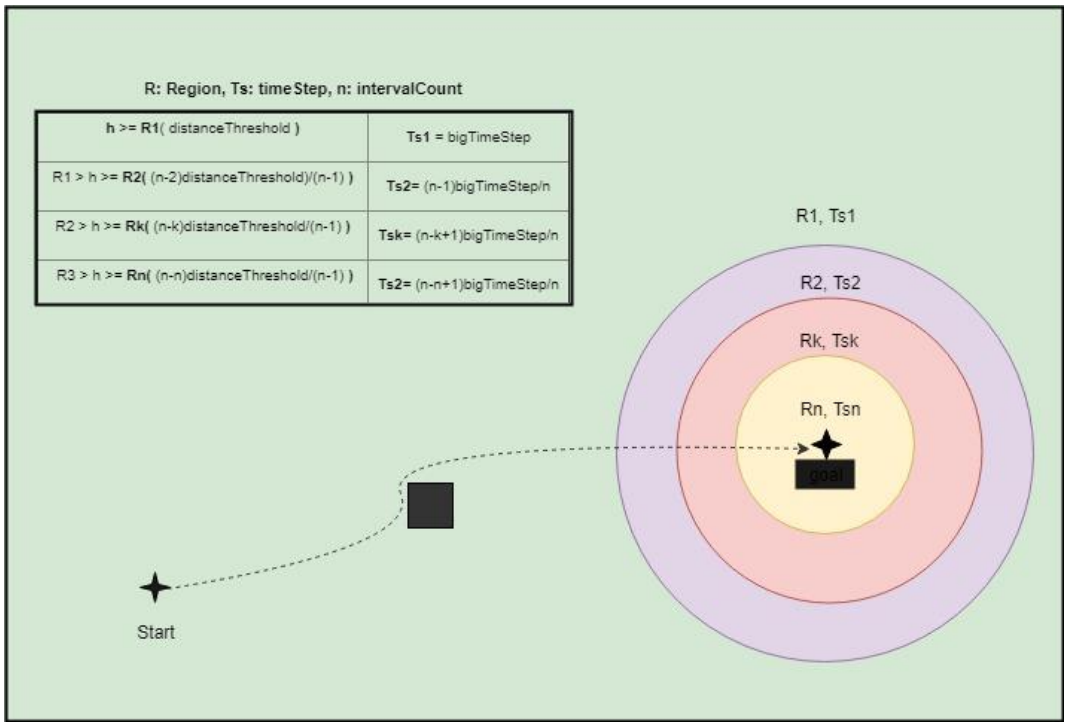

Figure 3 Illustration of the logic for implementing variable timeStep

4.4.2. OBSERVATIONS AND ISSUES FACED

- If the **bigTimeStep**, **intervalCount** and **highestThreshold** are well tuned (via trial and error) the performance shows major improvements with drastic reductions in the run time.
- In case of poorly tuned parameters, the overall performance decreases as the number of nodes to be evaluated increases thereby increasing the runtime.
- In case the path between two nodes is obstructed, the program altogether eliminates that node from consideration. This might eliminate a potentially valid and optimal path.
- In case of poorly tuned **highestThreshold,** we might overshoot our goal multiple times before actually reaching.

4.4.3. SCOPE OF IMPROVEMENT

- Currently the program explores a possible (ω_R, ω_L) pair for node creation via the **timeStep** generated from the **calcTimeStep()** method based on the current heuristic distance to the goal. In case the robot encounters an obstacle, this entire node is eliminated from consideration. Instead of this, upon encountering an obstacle, the robot should use the next smaller value of **timeStep** to create and validate the node and eliminate the possible pair iff all **timeStep** values have been exhausted.
- Reduce the overall number of nodes to be created and explored by assessing the 'health' of a child before deciding to explore a branch created from that node.

4.5. PHASE V

4.5.1. WORK DONE

children(): appended the following logic:

- Using smaller values for **timeStep** to explore and generate children nodes for each combination of (ω_R, ω_L) – if a child generated using an (ω_R, ω_L) pair with the intended **timeStep** is not valid:
    - while(temporary child is not valid and current **timeStep** $\geq$ smallest **timeStep**)
        - **timeStep** = next smaller **timeStep** value
        - compute possible child position for this **timeStep** at (ω_R, ω_L) using **getNewStepPosition()**
        - Find validity of this child via **validPathPosition()**
            - if valid: use this child
            - else: continue to find a valid child with smaller step
- Pruning the graph based on child base metric – Health check for newly created children is performed by comparing their heuristic distances to the goal:
    - From among all the children generated from a parent node, identify the least heuristic distance
    - For each of these children nodes:
        - Check if the heuristic distance to the goal is within a threshold percentage **childHealthLimit** of the previously identified least heuristic distance
            - if yes: continue
            - else: remove this child from further consideration

4.5.2. OBSERVATIONS AND ISSUES FACED

- Though the reduced **timeStep** logic is a little more computationally expensive in the initial stages of program execution, it reduces the overall runtime and nodes explored.
- Earlier if the child was not valid, we would altogether eliminate an (ω_R, ω_L) pair from consideration. However, with the inclusion of running this pair with a smaller **timeStep,** the program has a lower chance of getting stuck and may even return an optimal path more quickly which was previously being ignored.

- By correctly tuning the **childHealthLimit** we can prune the generated tree and decrease the over all node count. This can significantly reduce the runtime of the program while also finding a sufficiently optimal path.
- Also, overall higher **timeStep** values may be used without fear of eliminating a possibly optimal path which in-turn reduces runtime and number of nodes created.

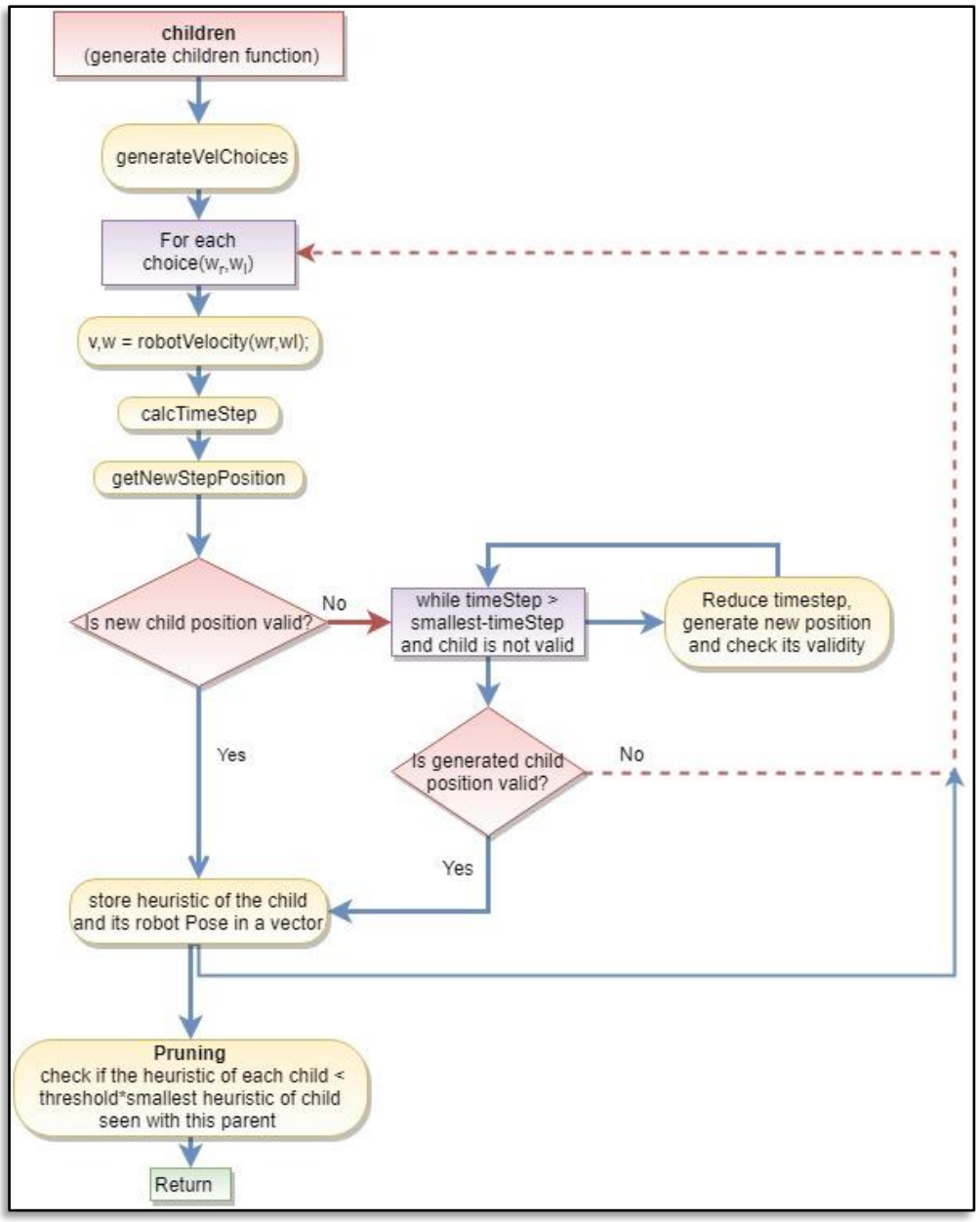

Figure 4 Logic behind children() method. Flowchart generated using https://www.draw.io/

4.5.3. SCOPE OF IMPROVEMENT

- A better logic for the variable **timeStep** can be devised such as one in which the current **timeStep** gets halved.

- Currently pruning is done by comparing the heuristics associated with the children being generated, with the minimum heuristic distance from amongst them being the central metric. However, this might not be the best way to prune and a better, more dynamic logic based on the actual distance from parent, or based on the statistical data of the heuristic of all the leaves can be implemented.

5. RESULTS & DISCUSSION

The following section discusses the effects of tuning of different parameters on the results of the program and how it affects the planned path when compared to the performance with using optimally tuned parameters.

Various different combinations of the parameters are run for a 50x50 corridor (depicted in fig.5) with:

- Start node at Position $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$: (5, 15, 0)
- $(\boldsymbol{\omega_R}, \boldsymbol{\omega_L})$ for Start node: (0,0)
- Goal node at Position $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$: (45, 15, 0)
- $(\boldsymbol{\omega_R}, \boldsymbol{\omega_L})$ for Goal node: (0,0)

Through trial and error, the values of the optimally tuned parameters for this case have been found to be:

- **thetaSpinWeight = 1**
- **straightSpinWeight = 1**
- **childHealthLimit = 1.05**
- **bigTimeStep = 4**
- **intervalCount = 4**
- **highestThreshold = 12**

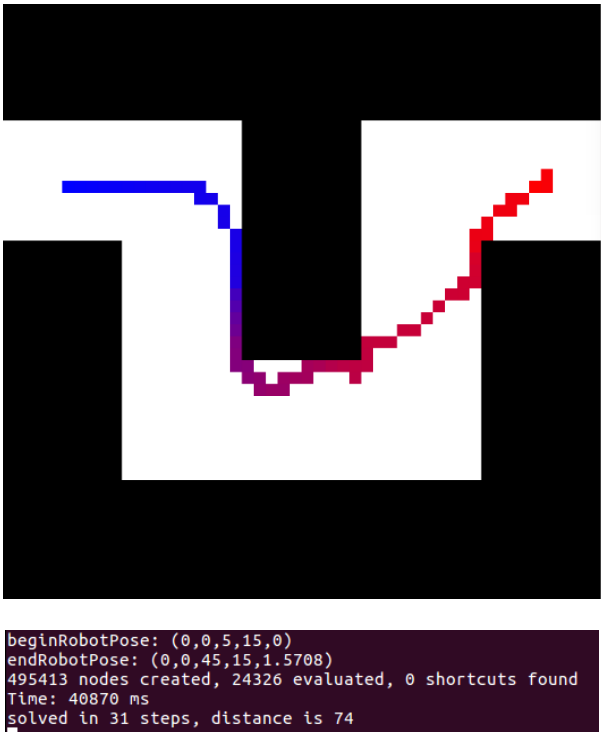

Figure 5 Performance with optimally tuned parameters

Parameter name	Definition	Desired trend
Nodes created	Total number of nodes created during the execution of the program	Lower is better
Nodes evaluated	Total number of nodes evaluated during the execution of the program	Lower is better
Distance	Cumulative parent to child distance for the planned path	Lower is better
Time	Total runtime of the program	Lower is better
Step count	Number of nodes lying in the planned path	-

Table 1 Parameters to evaluate the performance of the program

In an ideal setting, we would want the total distance of the planned path and the nodes in the graph to be the least. However, while tuning it was observed that both the parameters can simultaneously be minimised only to a certain extent after which the improvement in one is at the cost of other. Thus in our program, we have tuned to achieve the configuration in which both are minimal simultaneously.

5.1. bigTimeStep

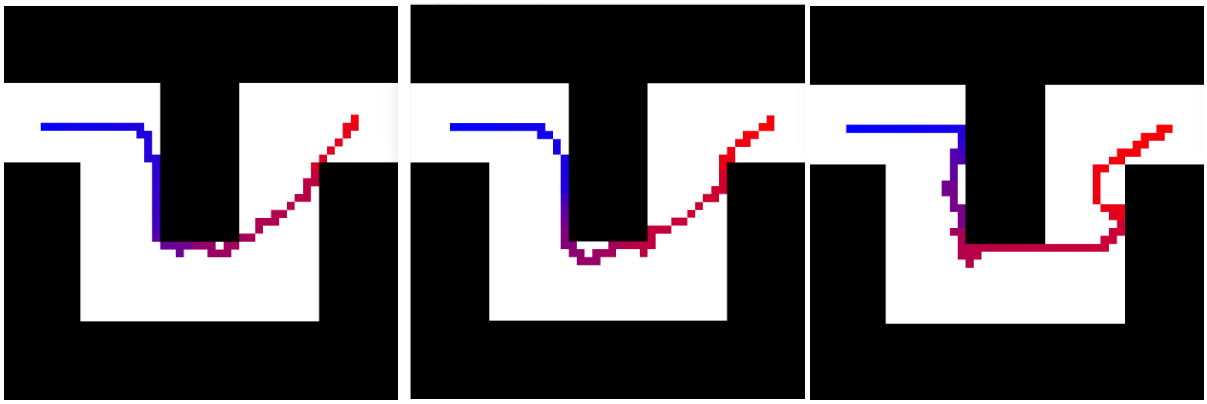

Figure 6 Left to Right: Underestimated, Correctly estimated and Overestimated

Tuning status	bigTimeStep	Nodes Created	Nodes evaluated	Run time	Steps	Distance
Underestimated	2	936473	43223	127083	31	64
Correct	4	495413	24326	40870	31	74
Overestimated	8	1200313	56045	228657	28	115

Table 2 Data for bigTimeStep tuning

- **Underestimation**: Each possible **timeStep** value will be small. Consequently, the individual step size will reduce and the number of nodes to be created and evaluated will increase. While the total distance is lower as the path found is closer to the optimal one, the runtime is higher.
- **Overestimation**: Each possible **timeStep** value will be large. Consequently, the individual step size will increase which will increase the computational cost during **validPathPosition()** execution. Consequently, the overall runtime will increase. Furthermore, due to overshooting when moving with bigger **timeStep**, the total number of nodes to explore and overall distance also increase.

- **Correctly tuned**: Computational cost for **validPathPosition()** execution and number of nodes generated are optimal.

5.2. intervalCount

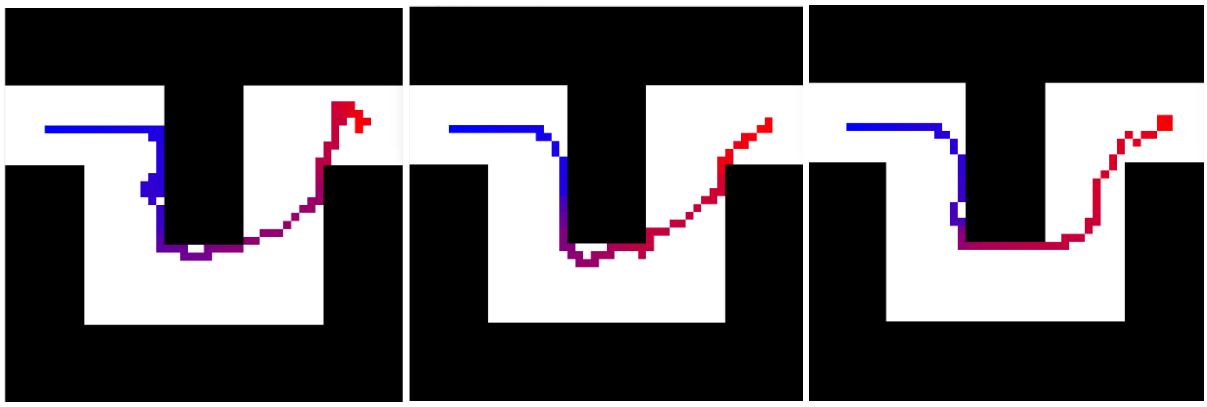

Figure 7 Left to Right: Underestimated, Correctly estimated and Overestimated

Tuning status	interval Count	Nodes Created	Nodes evaluated	Run time	Steps	distance
Underestimated	2	747519	76542	418549	31	99
Correct	4	495413	24326	40870	31	74
Overestimated	8	499153	25469	45449	20	65

Table 3 Data for intervalCount tuning

- **Underestimation:** The effect of underestimating the **intervalCount** is analogous to overestimating the **bigTimeStep** as in both the cases, the effective value of the individual **timeStep** ends up increasing.
- **Overestimation:** The effect of overestimating the **intervalCount** is analogous to underestimating the **bigTimeStep** as in both the cases, the effective value of the individual **timeStep** ends up decreasing.
- **Correctly tuned:** Both the runtime and the nodes in the graph are optimal.

5.3. childHealthLimit

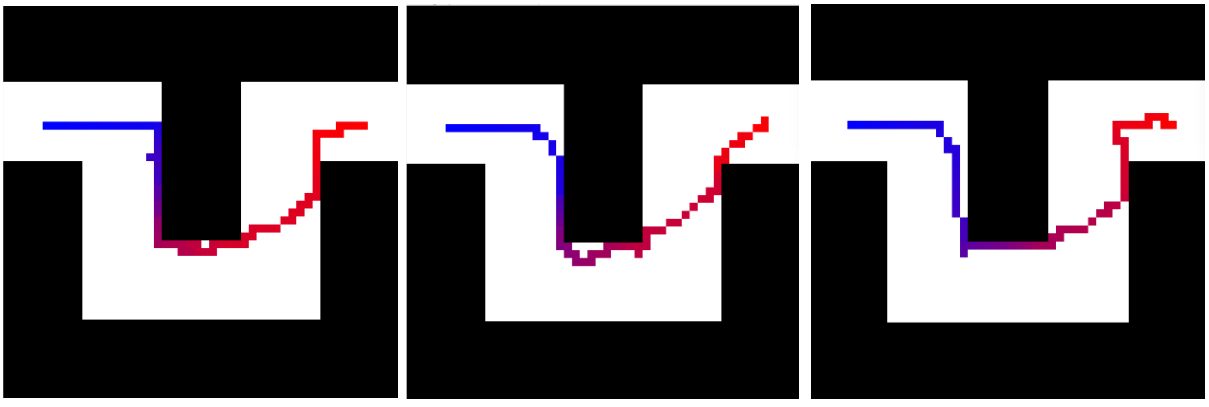

Figure 8 Left to Right: Underestimated, Correctly estimated and Overestimated

Tuning status	childHealthLimit	Nodes Created	Nodes evaluated	Run time	Steps	distance
Underestimated	1.03	26156	16314	12160	46	103
Correct	1.05	495413	24326	40870	31	74
Overestimated	1.1	1545730	62165	325329	25	66

Table 4 Data for childHealthLimit tuning

Tuning the **childHealthLimit** for optimal pruning:

- **Underestimation**: The health check is more strict, a lot of significant nodes may be pruned making path planning more difficult. While the runtime is lower due to very few nodes needing to be evaluated, the path generated is far from optimal and the program generates a path with very high distance.
- **Overestimation**: The health check is more lax and majority of the nodes will be included, not only defeating the purpose of pruning, but also increasing the computational cost due to implementation of health check for all the nodes being created. While the distance is lower as the path is closer to the optimal one, overall runtime is higher.
- **Correctly tuned**: Computational cost addition due to health check is offset by the benefit due to reduction in number of nodes being created.

5.4. highestThreshold

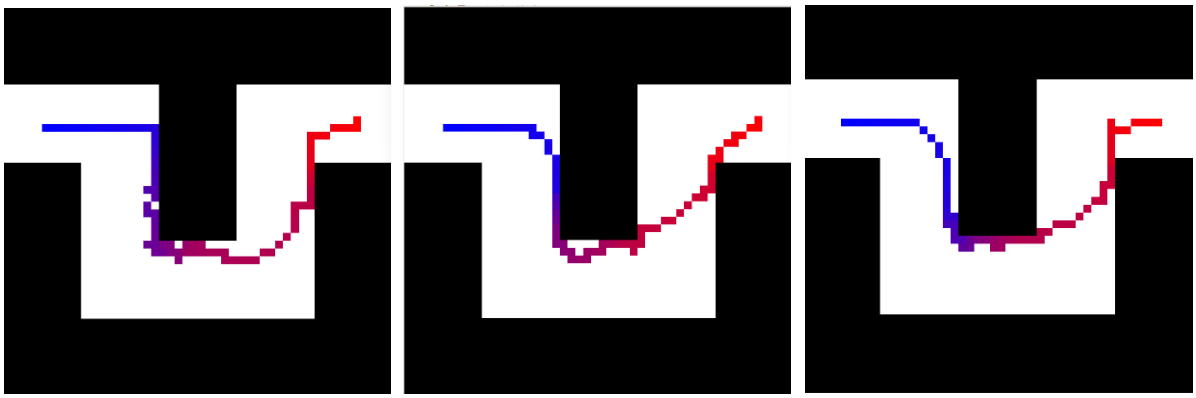

Figure 9 Left to Right: Underestimated, Correctly estimated and Overestimated

Tuning status	HighestThreshold	Nodes Created	Nodes evaluated	Run time	Steps	distance
Underestimated	10	767983	36517	89258	41	88
Correct	12	495413	24326	40870	31	74
Overestimated	14	524784	26702	49225	31	67

Table 5 Data for highestThreshold tuning

Tuning **highestThreshold:**

- **Underestimation**: The robot will move at the larger of **timeStep** values for greater distances. This will increase the computational cost during **validPathPosition()** execution which leads to

an increase in the runtime of the program. Furthermore, due to possible overshoot due to larger steps, the program ends up evaluating more nodes than required and returns a path further from optimal conditions with a high distance value.

- **Overestimation**: The robot will move at the smaller of **timeStep** values for greater distances. Consequently, the individual step size will reduce and the number of nodes to be created and evaluated will increase. While the generated path may be closer to the optimal path (lesser distance value), the runtime ends up increasing.
- **Correctly** tuned: Computational cost for **validPathPosition()** execution, runtime of the program and number of nodes generated are optimal.

5.5 CASES FOR SAMPLE PATH GENERATION - 1

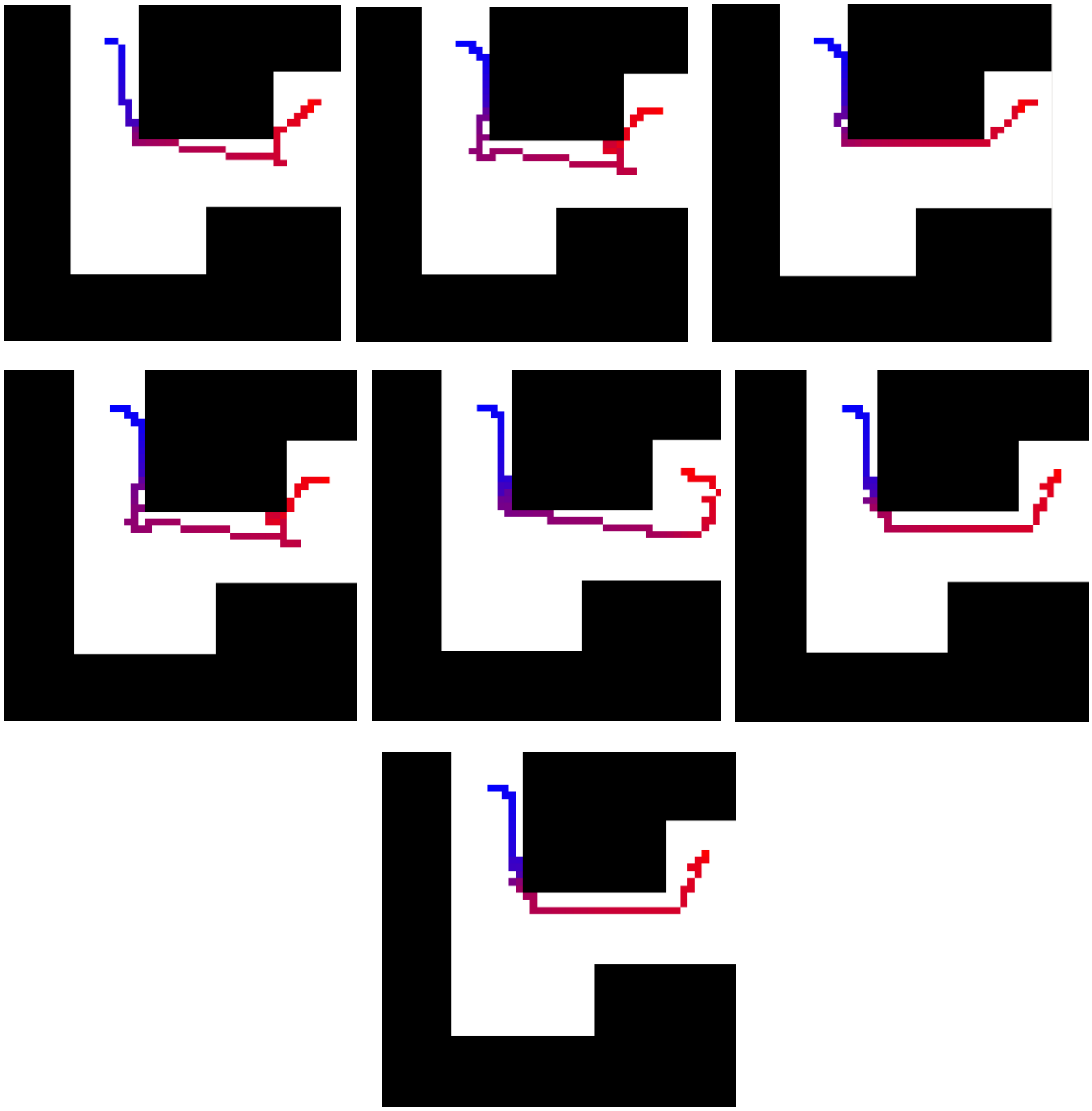

Figure 10 Path generated for cases 1 through 7

ImageName	x_begin	y_begin	theta_begin	ω_R_begin	ω_L_begin	x_end	y_end	theta_end	ω_R_goal	ω_L_goal	runtime	nodes created	nodes explored	steps	distance
1	15	5	0	0	0	45	15	PI/2	0	0	25773	40419	20052	18	55
2	15	5	0	0.5	0	45	15	0	0.5	0	39459	91691	24957	19	68
3	15	5	0	0.5	0.5	45	15	0	0	0	9173	213648	11804	19	54
4	15	5	0	0.5	0.5	45	15	0	1	1	39466	491691	24957	19	68
5	15	5	0	0.5	0.5	45	15	PI	1	1	36805	414181	24824	24	68
6	15	5	0	0.5	0.5	45	15	PI	0	0	5550	138691	10123	18	54
7	15	5	0	0.5	0	45	15	PI	0.5	0	5312	138691	10123	18	54

Table 6 Statistics of the paths generated for images 1 through 7

5.6 SAMPLE CASES FOR PATH GENERATION – 2

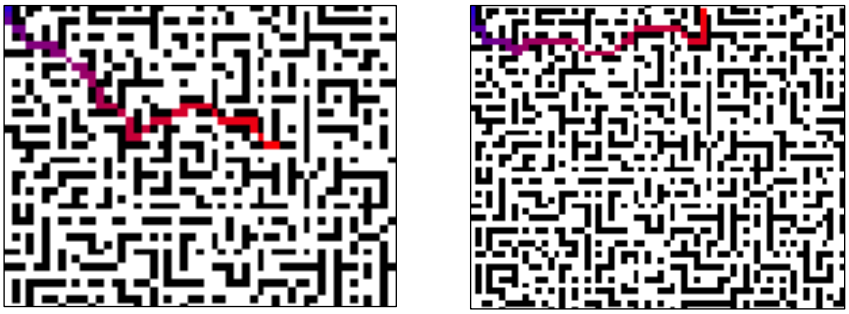

Figure 11 Path generated for cases 1 and 2 (borrowed from Prof. Kermorgant's architecture)

ImageName	x_begin	y_begin	theta_begin	ω_R_begin	ω_L_begin	x_end	y_end	theta_end	ω_R_goal	ω_L_goal	runtime	nodes created	nodes explored	steps	distance
1	1	1	0	0	0	36	21	0	0	0	67088	341574	40417	24	54
2	1	1	0	0	0	40	2	0	0	0	74241	370624	39061	23	55

Table 7 Statistics of the paths generated for images 1 and 2

6. EXPERIMENTAL APPROACHES & FUTURE WORK

This section discusses the current experimental approaches which were being tried by us in conjunction with what may be done to further improve the performance of the program.

6.1. IMPROVING THE HEURISTIC

Currently, the heuristic takes into account only the $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$ parameters of a node. However, if we are able to include information from the full state i.e. $(\boldsymbol{\omega_R}, \boldsymbol{\omega_L}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$ performance of the algorithm can be improved as it will become easier to assign unique heuristic distance values to the nodes making path generation much quicker.

An unsuccessful attempt – In a bid to include the . $(\boldsymbol{\omega_R}, \boldsymbol{\omega_L}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$ information in the heuristic, we tried devising an energy based heuristic method **h():**

- workDoneForce = **force***linearDistance + **moment***thetaDisplacement
- KECurrent = 0.5(**mass** v^2 + **inertia** ω^2)
- delKE = abs(KEFinal - KECurrent)
- return delKE + workDoneForce + workDoneMoment;

Here **force, moment, mass** and **inertia** are tuneable parameters. We were unable to properly tune these parameters for this approach to work and perhaps a better tuning methodology may prove fruitful.

6.2. AUTOTUNING THE PARAMETERS

h():

- Previously the heuristic distance was computed using the average wheel spun by the robot during the reorienting from $\boldsymbol{\theta_{current}}$ to $\boldsymbol{\theta_{goal}}$ plus during traversing the straight line distance between these nodes. This logic has been appended with tuneable parameters to assign separate weights to these components.
- return **straightSpinWeight***(wheel spun to go straight) + **thetaSpinWeight***(wheel spun to reorient)

Param class

- All the parameters used in the various different methods of the program, both tunable and fixed, have been accumulated in the **Param** class. Of these parameters, the tunable ones are:
  - **thetaSpinWeight**
  - **straightSpinWeight**
  - **childHealthLimit**
  - **bigTimeStep**
  - **intervalCount**
  - **highestThreshold**

Tune class

- The **Tune** class has been created specifically to tune the aforementioned parameters. Instead of tuning the parameters by hand following trial and error process, class methods have been created to automatically tune the parameters to be used for a specific path planning problem.
- **tuneHeuristicElements():**
    - Tunes the **straightSpinWeight** and **thetaSpinWeight** parameters used by running multiple iterations using different values of these parameters until 1000 nodes are created.
    - For a combination of possible weights, post the creation of 1000 nodes, from among these nodes the Euclidean distance of the closest node to the goal is recorded.
    - The weight combination which offers the least Euclidean distance is then used for running the complete program

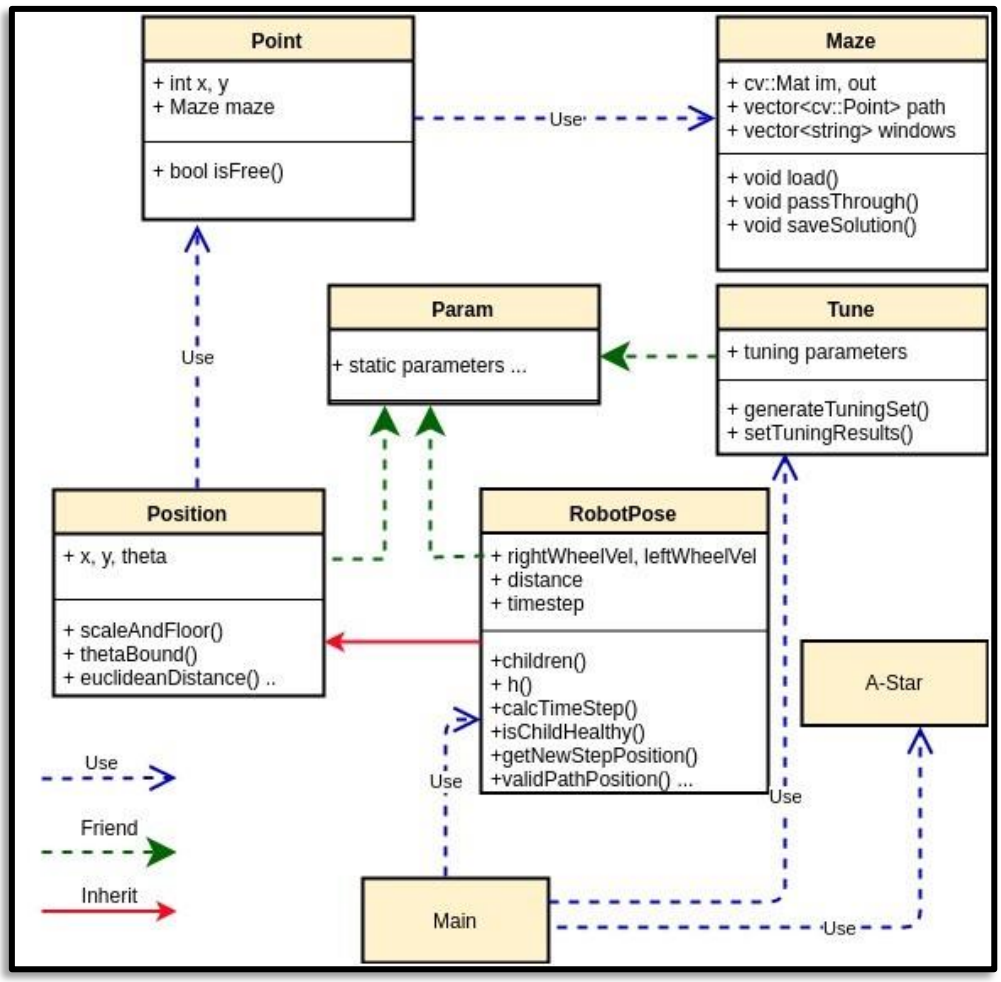

Figure 12 Flowchart for the completed logic generated using https://www.draw.io/

Ramifications of this approach:

- Though the program will may a longer initial runtime to run various test cases and tune the parameters, the overall runtime and performance of the algorithm will improve. Also, there would be no need to explicitly tune the parameters by hand for every test case.

- This approach would be beneficial only when the Start and Goal nodes are sufficiently far. In case they are close, the runtime with poorly tuned parameters may be better than first tuning the parameters and then running the program.
- While tuning the heuristic weights, currently we are using the Euclidean distance to quantify the 'closeness' between the best node (from 1000 others) and the Goal node. A better approach may be to use portion of path travelled formulated as:
    - Comparison metric = $\frac{\text{StartNode.h()} - \text{CurrentNode.h()}}{\text{StartNode.h()}}$
- Methods similar to **tuneHeuristicElements()** can be implemented to tune the remainder of the tuneable parameters for an optimally performing algorithm.

7. CONCLUSION

The approach is able to generate sufficiently optimal paths for various test cases. However, since A* is a graph based search, the number of nodes during exploration of the path explode exponentially increasing the runtime drastically.

In order to control the explosion of nodes, we introduced various tuning parameters to alter the behaviour of the program and modify it. Post tuning, the program was able to generate the path within acceptable runtime, however the program is highly dependant on tuning parameters and slightly mistuned parameters are detrimental to the performance.