Potential Functions based Sampling Heuristic for Optimal Path Planning

Sameer Satish Pusegaonkar
Maryland Applied Graduate Engineering
University of Maryland
College Park, USA
sameer@umd.edu

Pavan Mantripragada
Maryland Applied Graduate Engineering
University of Maryland
College Park, USA
mppavan@umd.edu

***Abstract*—Probabilistic path planning in the field of robotics has a superior advantage over action based techniques. Their fast approach to explore dense and large environments make them feasible for real life implementation. Despite these advantages, they can be further improved by a heuristic function to improve their randomness. The paper by Querishi[3] fills this gap by using a potential field based technique to improve the random sample generated. The results tower over the popular RRT* technique with respect to time for the graph generation and number of nodes required for reach the goal node resulting in a total lower cost for the path generated.**

***Index Terms*—Path Planning, Sampling Based Algorithms, Artificial potential fields**

I. Introduction

For robot navigation, to reach a goal node from a start node in a graph, a plan or strategy is required. Probabilistic techniques provide a new perspective on graph generation. These techniques don't provide a path but help generate a graph which then can be searched using path planning techniques like A* or Dijsktra's Algorithm.

RRT and RRT* are the 2 most poular techniuqes in the domain of probabilistic graph generation [1]. They explore the obstacle-free space uniformly [2]. They use a random function to generate a random sample. RRT* takes it a step further and provides rewiring capability.

Artificial Potential fields are used for graph generation [3]. The idea behind Artificial Potential Field (APF) is that the configuration space or the environment is a potential field [3]. Because of this potential field, the robot will move towards the goal node and away from the obstacles. Here, we assume that we are moving from a higher potential level to a lower potential level [3]. Therefore, it will be obvious that the goal is present at the lower level. Another way to ideate this is to think as if the robot and the obstacles have a positive charge and the goal node has a negative charge. Since the obstacles and the robot have a positive charge, they will repel from one another.

Hence, to traverse from the start to the goal we would move towards a negative gradient in the potential field. Therefore, at each iteration, the robot will look at the negative gradient and move in that direction [3].

II. Literature Review

RRTs explore slowly when the sampling domain is not well adapted to the problem. This issue is addressed by various methods in [1] [2] [3].

RRT* extents the RRT algorithm with the help of rewiring and connecting the random with a neighbour with minimum cost. This neighbour is obtained by getting all the neighbour nodes within a constant radius. This ensures an optimal solution generated for each graph.

Algorithm 1: RRT*(x_{init})

```
1  $V \leftarrow \{x_{\text{init}}\}; E \leftarrow \emptyset; T \leftarrow (V, E);$
2  for $n \leftarrow 0$ to $N$ do
3      $x_{\text{rand}} \leftarrow \text{RandomSample}(n);$
4      $X_{\text{near}} \leftarrow \text{NearbyNodes}(T, x_{\text{rand}}, n);$
5      if $X_{\text{near}} = \emptyset$ then
6          $X_{\text{near}} \leftarrow \text{NearestNode}(x_{\text{rand}}, T = (V, E));$
7      $L \leftarrow \text{GetTuple}(x_{\text{rand}}, X_{\text{near}});$
8      $x_{\text{parent}} \leftarrow \text{SelectBestParent}(L);$
9      if $x_{\text{parent}} \neq \emptyset$ then
10         $T = (V, E) \leftarrow \text{InsertNode}(x_{\text{rand}}, x_{\text{parent}}, T = (V, E));$
11         $E \leftarrow \text{RewireNodes}(x_{\text{rand}}, L, E);$
12 return $T = (V, E);$
```

The second part of RRT* involves rewiring where the newly generated sample is connected to a neighbour node if the euclidian distance between the neighbour node and its parent is greater than the distance between the neighbour node and the new sample node. [3]

Algorithm 2: GetTuple($x_{\text{rand}}, X_{\text{near}}$)

```
1  $L \leftarrow \emptyset;$
2  for $x' \in X_{\text{near}}$ do
3      $\tau \leftarrow \text{ExtendTo}(x', x_{\text{rand}});$
4      $c \leftarrow c(x') + c(\tau);$
5      $L \leftarrow (x', c, \tau);$
6  $L$.sort();
7  return $L$;
```

For all the helper functions, we always check if the path between the new sample node and the neighbour with minimum cost sample node is obstacle free.

Algorithm 3: SelectBestParent(L)

```
1 for $(x', c, \tau) \in L$ do
2     if CollisionFree($\tau$) then
3         return $x'$;
4 return $\emptyset$;
```

For each helper function, a constraint for exceeding maximum distance for each node is also implemented.

Algorithm 4: RewireNodes(x_{rand}, L, E)

```
1 for $(x', c, \tau) \in L$ do
2     if $(c(x_{\text{rand}}) + c(\tau)) < c(x')$ then
3         if CollisionFree($\tau$) then
4             $x'_{\text{parent}} \leftarrow$ GetParent($E, x'$);
5             $E$.remove($x'_{\text{parent}}, x'$);
6             $E$.add($x_{\text{rand}}, x'$);
7 return $E$;
```

In Dynamic-Domain RRT [1] they have reduced the number of iterations required by controlling the Voronoi bias of the nodes. In [2] they have used the expansive property of space to generate random nodes but it faces an issue where most of the generated nodes are discarded due to collision with obstacles increasing computational cost. All the techniques above have the lack of directivity in graph generation which leads to a slow convergence to an optimal solution.

This report is based on [3] which proposes that having a potential field to guide the search of new random nodes will improve the convergence rate. This methodology narrows node generation in the direction of the negative gradient of the potential field. Since the potential function is only used as a sampling heuristic and not directly in the path planning it avoids getting trapped in local minima

Qureshi et al [3] implemented a potential function to sample random points, the results presented show similar optimal paths for a lesser number of iterations without getting trapped in local minima. This technique is called PRRT* where the P emphases on the use of the potential gradients.

Instead of utilizing both the attractive and repulsive attributes of APF, only the attractive component of APF was utilized. This means that the attractive gradient would be the minimum for the goal node and maximum for obstacles.

Hence, a higher gradient value will be present for obstacles and a lower gradient value will be present for a goal position.

Once a cost map is generated for the entire arena, a gradient descent algorithm is used to go towards the goal node.

III. Methodology

A. Environment

A 2D 10x10 unit map was created on Pygame with obstacles, start node and goal node. Each obstacle was defined using a half-plane equation. The robot in the map was also assumed

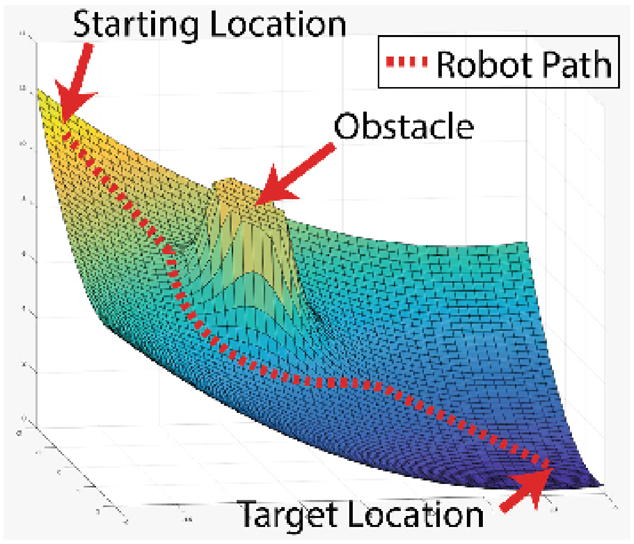

Fig. 1. Total gradient for a obstacle map

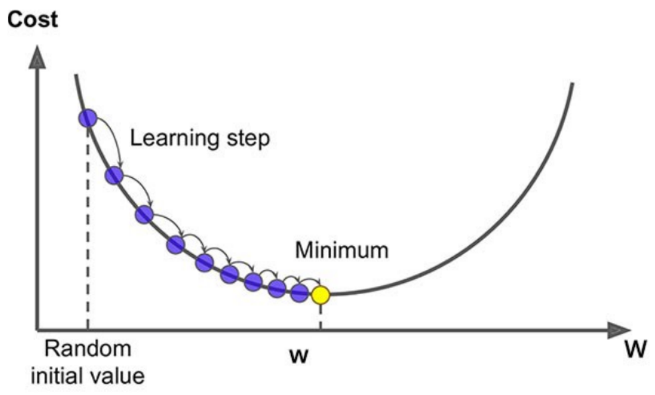

Fig. 2. Gradient Descent

to have a radius and a clearance. For this map a radius of 0.1 units and a clearance of 0.2 units was used.

B. Node

A node is defined as a point in the environment which consists pertinent information regarding its position, cost, child nodes and neighbour nodes.

C. Graph Generation using PRRT*

A graph for an environment is generated for a fixed number of iterations. For each iteration, a sample node was generated using the random function in Python. This sample node was then fixed using a Random Gradient Descent algorithm (RGD).

The role of this algorithm is to make the randomly generated sample node better. For our usecase, this means that the ouput of the RGD is a point closer to the goal node.

This output is computed using the attrative potential using the APG function. The APG fucntion is where the attarctive potential was computed using the node's position property.

After this, we simply follow the remaining algorithm as RRT* where we get the neighbour with the minimum cost

Algorithm 6: P-RRT*(x_{init})

```
1  $V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$; $T \leftarrow (V, E)$;
2  for $n \leftarrow 0$ to $N$ do
3      $x_{\text{rand}} \leftarrow$ RandomSample($n$);
4      $x_{\text{prand}} \leftarrow$ RGD($x_{\text{rand}}$);
5      $X_{\text{near}} \leftarrow$ NearbyNodes($T, x_{\text{prand}}, n$);
6      if $X_{\text{near}} = \emptyset$ then
7          $X_{\text{near}} \leftarrow$ NearestNode($x_{\text{prand}}, T = (V, E)$);
8      $L \leftarrow$ GetTuple($x_{\text{prand}}, X_{\text{near}}$);
9      $x_{\text{parent}} \leftarrow$ SelectBestParent($L$);
10     if $x_{\text{parent}} \neq \emptyset$ then
11         $T = (V, E) \leftarrow$ InsertNode($x_{\text{prand}}, x_{\text{parent}}, T = (V, E)$);
12         $E \leftarrow$ RewireNodes($x_{\text{prand}}, L, E$);
13 return $T = (V, E)$;
```

Algorithm 7: RGD(x_{rand})

```
1  $x_{\text{prand}} \leftarrow x_{\text{rand}}$;
2  for $n \leftarrow 0$ to $k$ do
3      $\vec{F}_{\text{att}} \leftarrow$ APG($X_{\text{goal}}, x_{\text{prand}}$);
4      $d_{\min} \leftarrow$ NearestObstacle($X_{\text{obs}}, x_{\text{prand}}$);
5      if $d_{\min} \leq d^*_{\text{obs}}$ then
6          return $x_{\text{prand}}$;
7      else
8          $x_{\text{prand}} \leftarrow x_{\text{prand}} + \lambda\left(\frac{\vec{F}_{\text{att}}}{|\vec{F}_{\text{att}}|}\right)$;
9  return $x_{\text{prand}}$;
```

within a constant radius. Our better sample is then connected to this neighbour node with minimum cost only if the line between these 2 nodes is not inside an obstacle. This was computed by getting all the points between the 2 node's position. For getting these points discretization had to be performed on a step size of 0.1. Starting from the nearest point towards the neighbour node it was observed if a point is inside the obstacle space the point before that current point would be returned as a better sample. The above process was only done if the line between the better sample and the neighbour node with minimum cost was inside any obstacle.

We then perform rewiring to minimize the cost of all the neighbours within the constant radius. This is done by comparing the costToCome for a given neighbour with their parent with the costToCome of the neighbour node with the better sample we got from RGD. This process is repeated for all the neighbours of the better sample. The entire pipeline is repeated for a cosntant number of iterations.

Pygame was used to perform all the visualization. All the nodes were generated using the pygame.circle() function.

$$U_{\text{att}} = d^2(x_{\text{rand}}, x_{\text{goal}}) : x_{\text{goal}} \in X_{\text{goal}}$$

$$\vec{F}_{\text{att}} = -2d(x_{\text{rand}}, x_{\text{goal}}) : x_{\text{goal}} \in X_{\text{goal}}$$

A connection between the 2 nodes was generated using pygame.line() function.

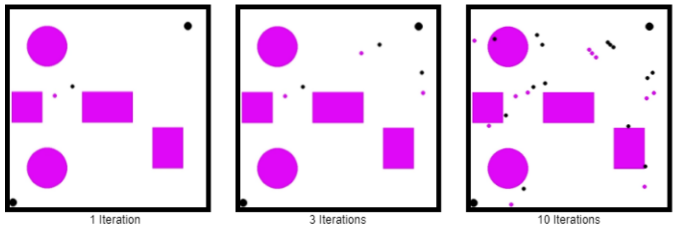

Fig. 3. Points generated for a single iteration vs 3 iterations vs 10 iterations

In the above image, we can see that the bottom left node is the start node and the top right black node is the goal node. A randomly generated sample is shown by a pink circle and the better sample node generated using the random node is shown by the black node. Therefore, for each pink node a black node will be generated. Later on the randomly generated pink node is destroyed. It can be observed how the better black node is always closer to the goal node present at the top right corner of the map.

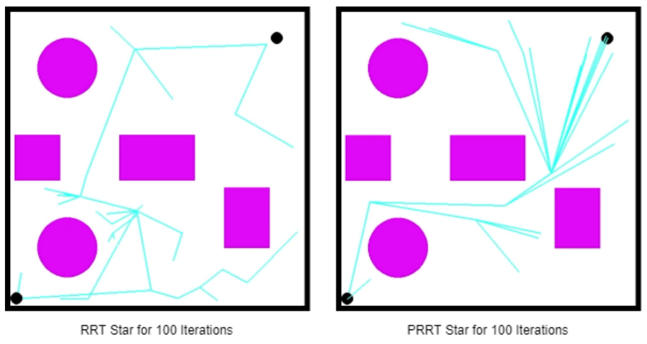

Fig. 4. Nodes generated in RRT* vs PRRT* for the 100 iterations

It can be observed how PRRT* is able to reach to the goal node whereas RRT* is not able to reach the goal node even when the number of the iterations is the same. This is mainly because of the greedy approach of the random gradient descent algorithm. The generated graph also helps us in the next steps as we can then find an optimal path using the A* algorithm

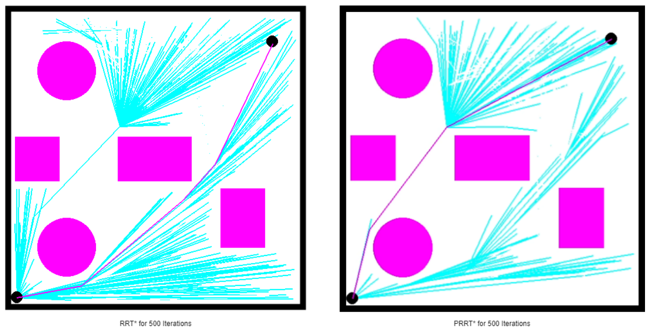

Fig. 5. RRT* vs PRRT* for 500 iterations

It was observed that for the same number of iterations fewer nodes were generated by PRRT* as compared to RRT*.

D. Path Finding using A* star

After a graph is generated we can use it to search for an optimal path. Because the path consists of child-parent relation we find an optimal path using A*Star algorithm. In A* Star algorithm, we utilized a priority queue data structure which will always get the minimum cost node in a O(1) time complexity. The time complexity for generating the priority queue itself is O(log N).

A costToCome and a costTOGo is computed for each and every node. The costToCome is the euclidean distance between the previous node to the current node while the costToGo is the euclidean distance between the current node and the goal node. The costTOCo factor is what influences the A Star algorithm to reach the optimal path.

$$d(sampleNode, neighbourNode) = \sqrt{(sampleNode.x - neighbourNode.x)^2 - (sampleNode.y - neighbourNode.y)^2)} \tag{1}$$

The node is then inserted into the priority queue depending upon the totalCost property.

$$costToCome + costToGo = totalCost \tag{2}$$

Algorithm 1: A* Algorithm

```
Input: Start node and Goal node
Output: List of nodes that connect start to goal
1  priorityQueue = [];
2  push(priorityQueue, (startNode.cost, startNode)) while
     priorityQueue is not empty do
3      currentNode = pop(priorityQueue);
4      if currentNode is in goal node then
5          return True;
6      end
7      for neighbor in currentNode.neighbors do
8          push(priorityQueue, (neighbour.cost,
             neighbour));
9      end
10 end
11 return False;
```

E. Implementation in ROS with Turtlebot3 Burger

We have created a ROS package named Astar for the implementation of path planner on a TurtleBot3 Burger. It consists of two main executable files which plan and control the Turtlebot3. The Turtlebot3 is simulated in a gazebo environment. The planner is Incorporated in the goal_broadcaster.py file. It takes start position, goal position, robot clearances and maximum iterations of PRRT* as inputs from the user. The Path is computed using A* algorithm and each node of the planned path is passed as a way-point to the controller. This is done using a ROS transform broadcaster. We broadcast ”goal” tf w.r.t. ”odom” tf and wait for 1 second to controller reach the set target. This process is iterated until the bot reaches the goal location. The controller is implemented in bot_controller.cpp file. It has tf listener which listens both to the ”goal” and ”base_footprint” tfs w.r.t ”odom” tf. Using this information it estimates the positional error at a given instance. PD controller loop is designed to reduce the error by sending Twist msgs on /cmd_vel topic to Turtlebot3

The code for both the PyGame visualization and ROS visualization is available here.

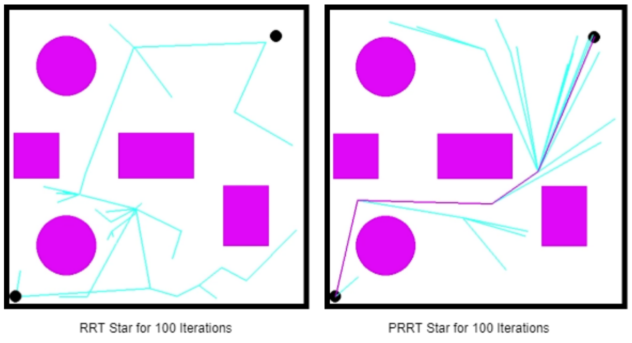

Fig. 6. RRT* vs PRRT*: No path found in RRT* vs Path found in PRRT*

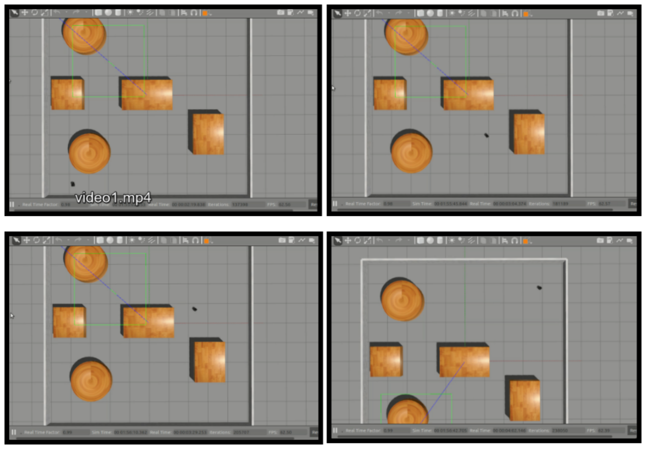

Fig. 7. Simulation of turtlebot3 going from the start node: (0.3,0.3) to goal node: (9, 9)

IV. Results

Both RRT* and PRRT* were compared with each other in the same map. The output metrics were tracked for each iteration.

We compared the cost of each path by keeping the number of iterations same.

It can be observed how where RRT* is not able to provide us a path, PRRT* gives us a path with a miminal cost. Moreover, where RRT* star can find a path, PRRT* can find a path with a lower cost.

We also compared both the algorithms with respect to the cost. Keeping the cost same, we can compare the total number of nodes generated for both algorithms.

TABLE I
COMPARING RRT* AND PRRT* WITH RESPECT TO THE NUMBER OF ITERATIONS

Iterations	***Cost of path RRT Star***	***Cost of path PRRT Star***
100	No path found	13.8336
200	No path found	14.6192
500	No path found	13.3935
2000	14.61	13.4835

TABLE II
COMPARING RRT* AND PRRT* WITH RESPECT TO THE COST

Cost	***Nodes generated in RRT****	***Nodes generated in PRRT****
13.39	553	160
12.9	673	153
13.34	723	139

It can be observed that the same cost in RRT* and PRRT*, PRRT* generates fewer nodes than RRT*. This is mainly because PRRT* reaches the goal node faster than RRT*.

V. CONCLUSION

From the above results, it is evident that PRRT* can obtain a result with fewer nodes when compared with RRT*. This is because of the correction of a sample point using RGD towards the goal. Because of RDG the amount of time required to reach the goal also decreases. This greedy behavior of PRRT* is beneficial with respect to time and cost. A* combined with this always helps us to get the optimal path in the above generated graph.

VI. FUTURE WORK

Because we utlized a gradient descent algorithm which uses a learning rate or lambda which is a constant value, it is possible that this value will have to be tuned for different maps and diffrent obstacle spaces. To avoid the tedious task of tuning the learning rate for every obstacle map, a dynamic learning rate which changes depends on how many nodes are generated and how close we have reached to the goal point would be optimal. This means that the learning rate could be higher when fewer nodes are generated and then gradually increase when the number of nodes are higher. This later will also avoid to overshooting from the sample to reach the goal node.

ACKNOWLEDGEMENT

We would like to thank Professor Reza Monfaredi for his tremendous guidance and mentorship throughout the project.

REFERENCES

[1] A. Yershova, L. Jaillet, T. Simeon and S. M. LaValle, "Dynamic-Domain RRTs: Efficient Exploration by Controlling the Sampling Domain," Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain, 2005, pp. 3856-3861, doi: 10.1109/ROBOT.2005.1570709.
[2] D. Hsu, J. -. Latombe and R. Motwani, "Path planning in expansive configuration spaces," Proceedings of International Conference on Robotics and Automation, Albuquerque, NM, USA, 1997, pp. 2719-2726 vol.3, doi: 10.1109/ROBOT.1997.619371.
[3] Qureshi, A.H., Ayaz, Y. Potential functions based sampling heuristic for optimal path planning. Auton Robot 40, 1079–1093 (2016). (https://doi.org/10.1007/s10514-015-9518-0)
[4] J. Amiryan and M. Jamzad, "Adaptive motion planning with artificial potential fields using a prior path," 2015 3rd RSI International Conference on Robotics and Mechatronics (ICROM), Tehran, Iran, 2015, pp. 731-736, doi: 10.1109/ICRoM.2015.7367873. (https://ieeexplore.ieee.org/document/7367873)
[5] A. H. Qureshi et al., "Adaptive Potential guided directional-RRT," 2013 IEEE International Conference on Robotics and Biomimetics (ROBIO), Shenzhen, China, 2013, pp. 1887-1892, doi: 10.1109/ROBIO.2013.6739744. (https://ieeexplore.ieee.org/document/6739744)
[6] Karaman S, Frazzoli E. Sampling-based algorithms for optimal motion planning. The International Journal of Robotics Research. 2011;30(7):846-894. doi:10.1177/0278364911406761
[7] https://sci.esa.int/web/lisa-pathfinder/-/56434-spacetime-curvature
[8] https://www.researchgate.net/figure/An-example-of-a-traditional-potential-field-which-can-be-used-for-navigating-toward-a_fig2_294645105
[9] https://www.researchgate.net/figure/Example-of-a-Potential-Field_fig2_49402674
[10] https://morioh.com/p/15c995420be6
[11] Steven M. LaValle, Planning Algorithms, 2006, Cambridge University Press