Rapidly Re-planning RRT*
A Novel Re-Planning Algorithm

Author

USAMA TARIQ KHAN

Regn # 00000206755

Supervisor

Dr. YASAR AYAZ

DEPARTMENT OF ROBOTICS & ARTIFICIAL INTELLIGENCE

SCHOOL OF MECHANICAL & MANUFACTURING ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

ISLAMABAD

AUGUST, 2021

Rapidly Re-planning RRT*
A Novel Re-Planning Algorithm

Author

USAMA TARIQ KHAN

Regn 00000206755

A thesis submitted in partial fulfillment of the requirements for the degree of

MS Robotics & Intelligent Machine Engineering

Thesis Supervisor:

Dr. Yasar Ayaz

Thesis Supervisor's Signature: _____

DEPARTMENT OF ROBOTICS & ARTIFICIAL INTELLIGENCE

SCHOOL OF MECHANICAL & MANUFACTURING ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

ISLAMABAD

AUGUST, 2021

National University of Sciences & Technology

MASTER THESIS WORK

We hereby recommend that the dissertation prepared under our supervision by: **Mr. Usama Tariq Khan, Reg. # 0000206755** Titled: "**Rapidly Re-planning RRT* - A Novel Re-Planning Algorithm**" be accepted in partial fulfillment of the requirements for the award of **MS Robotics & Intelligent Machine Engineering** degree. (**Grade:** _____)

Examination Committee Members

1. Name: Dr. Sara Ali Signature:_____
2. Name: Dr. Ahmed Hussain Qureshi Signature:_____
3. Name: Mr. Zaid Tahir Signature:_____

Supervisor's name: _____ Signature: _____

Date:_____

_____ _____

Head of Department Date

COUNTERSINGED

Date:_____ _____

Dean/Principal

Thesis Acceptance Certificate

It is certified that the final copy of MS Thesis written by Mr. Usama Tariq Khan (Registration No. 00000206755), of Department of Robotics and Intelligent Machine Engineering (SMME) has been vetted by undersigned, found complete in all respects as per NUST statutes / regulations, is free from plagiarism, errors and mistakes and is accepted as a partial fulfilment for award of MS Degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in this dissertation.

Signature: _____

Name of Supervisor: Dr. Yasar Ayaz

Date: _____

Signature (HOD): _____

Date: _____

Signature (Principal): _____

Date: _____

Plagiarism Certificate (Turnitin Report)

This thesis has been checked for Plagiarism. Turnitin report endorsed by Supervisor is attached.

_____

Usama Tariq Khan

Regn # 00000206755

_____

Dr. Yasar Ayaz

Supervisor

Declaration

I certify that this research work titled "*Rapidly Re-planning RRT* - A Novel Re-Planning Algorithm"* is my own work. The work has not been presented elsewhere for assessment. The material that has been used from other sources it has been properly acknowledged / referred.

_____

Usama Tariq Khan

Regn # 00000206755

Copyright Statement

Acknowledgements

All praise and thanks is for Almighty Allah, the Source of all knowledge and wisdom, without Whose divine help and guidance nothing is possible.

I am thankful to my Supervisor Dr. Yasar Ayaz for his continuous guidance, inspiration and support throughout this work and during the tenure of MS program.

I would also like to pay thanks to Dr. Ahmed H. Qureshi for always being there with his knowledgeable suggestions and support, despite his busy schedule and 12 hrs. time difference. Similarly, I would also like to thank Mr. Zaid Tahir for his support and guidance.

I am also grateful to Mr. Mudassar Ayub for crucial support in the time of need.

Finally, I would like to express my gratitude to all fraternity of SMME for making this place a wonderful hub of learning.

Dedicated to the Squad: Mumtaz, Hussain, Faateh & Rafay

Abstract

Sampling based motion planning algorithms such as RRT* provide an optimal path from a start to goal point. However, any change in either of these points requires re-spawning of the tree from scratch or using a multi-query algorithm, both of which are time consuming options. An alternative is to re-use the existing tree to find path between the new start and goal points. A novel algorithm, Rapidly Re-planning RRT* [R4T*], is being presented here which caters for these requirements. R4T* builds a Smart-Graph using an existing RRT* tree to find optimal paths between any two points in the workspace. The graph can be developed from an existing RRT* tree or alongside one being built. The algorithm caters for a real-time environment, where the robot starts moving as soon as a path to goal is found. If the goal is changed at any stage, the algorithm yields a path from current position of the robot to the new goal.

The path found has comparable cost to a 7000 node RRT* algorithm run for the same start and goal points. The research work thus presents an optimal re-planning algorithm which yields optimal real-time paths between any two points in the workspace with minimum computational overload.

Key Words: Path planning, Re-planning, RRT*, Motion planning, Robotics,

Table of Contents

List of Figures

List of Tables

CHAPTER 1: INTRODUCTION

1.1 Path Planning Significance & Basics

Robotics & Artificial Intelligence are amongst the most promising fields of science and technology in today's world. These technologies are increasingly being utilized in every walk of human life. Their applications are see in a variety of fields such as large-scale manufacturing, warehousing, autonomous vehicles, robotic surgeries along with various commercial, domestic and military avenues. This variety has led to greater interaction of robots with their environment and increased the complexity of robotics design and control. As path planning is one of the crucial components of Robotics, it has been subject of heavy research for several past decades. Besides robotics, path planning has its applications in other fields including video games, animations etc.

Path Planning algorithms attempt to find a series of control inputs which will take the robot from current configuration (start configuration) to desired configuration (goal) while avoiding the obstacles along the way.

Let X represent the configuration space of the robot with X_{obs} defining the space occupied by obstacles. X_{free} is the free space, void of obstacles, defined as $X_{free} = X \backslash X_{obs}$. The initial point is defined as z_{start} while the goal point is defined as z_{goal}. The path planner attempts to find a series of control inputs u = [0, T] which result in a feasible path $x(t) \epsilon X_{free}$, such that $x(0) = z_{start}$ and $x(T) = z_{goal}$.

The obstacle configuration is generally difficult to be represented explicitly using geometric representation. Therefore, it is represented in the form of collision checking algorithm which returns 'None' if the configuration falls in the obstacles space.

The path planning algorithms generally focus on two parameters to judge the efficiency of the algorithm. These include completeness and optimality. Completeness refers to the ability of an algorithm to return a path in finite time (if the path is available) or return failure if the path does not exist. The optimality of the path is also main concern of these algorithms, which focuses to return optimal path between given start and goal points in minimum possible.

Overall, these algorithms are divided into two main types: single query and multi query algorithms. The single query algorithms are optimized for single use, whose focus is to find an optimal path between given start and goal pair in minimum possible time.

The multi-query algorithms focus on developing a graph which can be used to find path between multiple points in the environment. This requires a pre-planning phase which develops the graph in a comprehensive manner followed by execution phase, where the paths between different points are yielded by searching through the paths.

On the basis of completeness, path planning algorithms can be categorized as follows.

- Complete algorithms – These algorithms return a solution in finite time or correctly report failure if there is no solution.
- Resolution Complete Algorithms – These algorithms are guaranteed to return a path, if the resolution of the underlying grid is set to fine enough value as required by the algorithm. In case the path does not exist or the resolution is not fine enough, they report failure.
- Probabilistically complete algorithms – The probability of these algorithms returning a solution approaches one as the number of samples approaches infinity.

CHAPTER 2: SAMPLING BASED PATH PLANNING ALGORITHMS

Sampling based path planning algorithms are quite popular and fairly practical algorithms. These algorithms have been proven to be probabilistically complete and are generally able to return an optimized path between two goal points. The sampling based algorithms include both single query and multi-query algorithms. Popular sampling based single query algorithms include RRT, RRT* and its variations while popular multi-query sampling based algorithms include PRM, PRM* etc. This section will focus on main single-query sampling based algorithms and their popular variations.

2.1 Rapidly Exploring Random Tree [RRT]

Rapidly Exploring Random Tree or RRT was one of the primary algorithms which successfully explored non-conventional, high-dimensional spaces by randomly building a tree. The tree is configured to be rooted at the start point. Random samples are drawn from the configuration space. A node is added to the tree in the direction of the sample drawn if the node does not come in the obstacle space. The node is connected to the existing tree if the path connecting the node to its parent also lies in the obstacle free space.

The algorithm is described in Table 2.1.

1.	T.init (q_{start})
2.	for k = 1 to nodesmax do
3.	q_{rand} ← RAND_CONF()
4.	q_{near} ← NEAREST_VERTEX(q_{rand}, T)
5.	q_{new} ← NEW_CONF(q_{near}, q_{rand}, Δq)+
6.	T.add_vertex(q_{new})
7.	T.add_edge(q_{near}, q_{new})
8.	return T

Table 2.1 – RRT Algorithm

The tree T is initialized with the start node. This is followed by expansion of the tree for a maximum of *nodesmax* nodes. The RAND_CONF function returns a random point in C_{free}. The NEAREST_VERTEX function returns a node q_{near} which is nearest to the random point in the tree

T. A new node q_{new} is added to tree T such that it is at a distance Δq from q_{near} by NEW_CONF. The next line adds a new edge between q_{near} and q_{new}. Both NEW_CONF and add_edge functions ensure that the new node q_{new} and the edge connecting q_{new} to q_{near} lie in free space C_{free}.

The algorithm was, however, shown to always converge to a non-optimal path. This led to development of other algorithms which deal with the non-optimality problem.

2.2 Rapidly Exploring Random Tree* [RRT*]

RRT* is asymptotically optimal version of RRT, i.e., it is proven to converge to optimal solution and possesses probabilistic completeness property. Like RRT, the algorithm samples points in the configuration space. It attempts to find optimized path from the start point to all the points in the configuration space. In doing so, the algorithm finds an optimal path to the goal point z_{goal}.

RRT* algorithm achieves optimality by using the Rewire algorithm. After addition of new node q_{new} to the tree, the Rewire function checks all the nodes in the neighborhood of q_{new} to see if the cost to reach these nodes is less through q_{new} than their existing cost to reach (through their existing parents). If the cost to reach a particular node is less through q_{new} then that node is rewired as a child of q_{new} and its connection with existing parent is severed. In this way, the nodes which are already present in the tree are optimized with the addition of new nodes. This optimization makes RRT* different from RRT and results in asymptotic optimality.

RRT* algorithm is discussed in detail in chapter 5.

Variations of Rapidly Exploring Random Tree* [RRT*]

Many variations of RRT* have been proposed over the years to increase its speed of convergence to an optimal solution. Following is an overview of these algorithms.

2.3 Any Time – RRT*

The algorithm focuses on finding a quick path to the goal point. Once the execution phase of the plan is started, i.e., once the robot starts moving, the plan is improved towards an optimal solution. It takes advantage of the fact that most robotic systems take more time in execution, so that time is used to optimize the trajectory.

RRT* is used to find an initial solution. After that two main programs are employed: i) committed trajectory ii) branch and bound tree adaptation. Once the path has been found, the robot starts to execute the first portion of the trajectory for committed time. This portion is known as committed

trajectory. The end of committed trajectory is taken as the new root of the tree and the path is improved from the new root to the goal. Once the robot reaches the end of committed trajectory, the process is repeated until the robot reaches the goal.

2.4 Potential Functions based Sampling Heuristic for Dynamic Domain RRT (P-RRT*)

The algorithm functions to optimize the memory and time utilization by RRT*. It incorporates artificial potential field algorithm in RRT* to overcome time and memory limitations. The incorporation of APF result in decrease of number of iterations of RRT* leading to efficient memory utilization and accelerated convergence rate.

Artificial Potential Fields [APF] is a resolution complete method, i.e., the method is able to effectively plan optimal paths if the resolution of the grid is properly optimized. Using APF directly results in pure exploitation which makes the planner greedy as it assumes that the provided information is sufficient for path computation. Pure exploitation allows APF to quickly find the solution but it also causes the algorithm to suffer from local minima problem. On the other hand, sampling based algorithms perform pure exploration of the configuration space so as to improve the planner's understanding of the space. So the idea of idea of potentially guided, directionalized sampling by incorporating APF into RRT* results in guided exploration of the environment.

APF utilizes gradient descent planning that tries to minimize artificial potential energy. The main robot and the goal are assigned attractive potentials while the obstacle regions are assigned repulsive potentials. These attractive and repulsive potentials cause the robot to experience a force F, equal to the negated gradient of the potentials. $F = -\nabla U$.

Under the influence of both attractive and repulsive potentials the robot moves down the slope and reaches the goal region without any collisions. Two constants K_a and K_r are used to scale the magnitude of attractive and repulsive potential. A circular region (d_g) is defined around the goal, the robot moves rapidly if it is outside this region. Inside this region the potential starts to vary conically causing the robot to move slowly when it comes close to the goal preventing it from overshooting.

$$U_{\text{att}} = \begin{cases} K_a d^2(x, x_g) & d(x, x_g) > d_g^* \\ K_a(d_g^* d(x, x_g) - (d_g^*)^2) & d(x, x_g) \le d_g^* \end{cases}$$

$$\vec{F}_{\text{att}} = \begin{cases} -2K_a d(x, x_g) & d(x, x_g) > d_g^* \\ -2d_g^* K_a \dfrac{x - x_g}{d(x, x_g)} & d(x, x_g) \le d_g^* \end{cases}$$

The attractive forces are computed as follows. The attractive potential varies quadratically when the distance is greater than d_g. A distance d_{obs} defines the minimum distance from the obstacles while current distance from the closest vertex of the obstacle is denoted by d_{min}. If $d_{min} > d_{obs}$, denoting that the robot is far from obstacle, then the repulsive force becomes zero and the robot is allowed to move freely.

The repulsive forces are calculated as follows. Repulsive potential is 0 if distance is greater than d_{obs}. X' is the nearest obstacle vertex. The overall potential is the sum of both attractive and repulsive potentials.

$$d_{\min} = \min_{x' \in X_{\text{obs}}} d(x, x')$$

$$U_{\text{rep}} = \begin{cases} \dfrac{1}{2} K_r \left(\dfrac{1}{d_{\min}} - \dfrac{1}{d_{\text{obs}}^*} \right)^2 & d_{\min} \le d_{\text{obs}}^* \\ 0 & d_{\min} > d_{\text{obs}}^* \end{cases}$$

$$\frac{\partial d_{\min}}{\partial x} = \frac{(x - x')}{d(x, x')}$$

$$\vec{F}_{\text{rep}} = \begin{cases} K_r \left(\dfrac{1}{d_{\text{obs}}^*} - \dfrac{1}{d_{\min}} \right) \dfrac{1}{d_{\min}^2} \dfrac{\partial d_{\min}}{\partial x} & d_{\min} \le d_{\text{obs}}^* \\ 0 & d_{\min} > d_{\text{obs}}^* \end{cases}$$

Potential Function based RRT* incorporates APF into RRT*. Its working is described as follows. After a random sample x_{rand} is generated by RRT*, it is improved by APG function and labelled as x_{prand}. After this, normal RRT* functioning is resumed with x_{prand} as the random sample. The Attractive Potential Gradient (APG) function utilizes only quadratic variation in the attractive potential fields instead of shifting between quadratic and conical versions. Overshooting is not an issue here since it's not the robot but the random sample which is being guided. The function computes the attractive force. Minimum distance (d_{min}) from the nearest obstacle is computed. If the random sample is close to the obstacle, then the random sample is returned as x_{prand}. Otherwise, x_{prand} is guided towards the goal using the following function.

$$x_{\text{prand}} \leftarrow x_{\text{prand}} + \lambda \left(\frac{\vec{F}_{\text{att}}}{|\vec{F}_{\text{att}}|} \right)$$

2.5 RRT*-Smart

The algorithm aims to accelerate convergence time of RRT* using two techniques namely Path Optimization and Intelligent Sampling. The algorithm starts with RRT* and continues until a path is found. Once an initial path is found, the Path Optimization kicks in. Path Optimization starts an

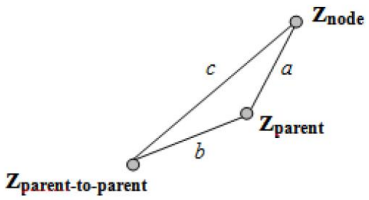


Figure 2.1: Triangular Inequality

iterative process from z_{goal} and moves up to z_{init}, optimizing the path based upon triangular inequality. The triangular inequality is shown in figure 2.1. The path from z_{node} to z_{parent} and from z_{parent} to $z_{parent\text{-}to\text{-}parent}$ is higher in cost than a direct path from z_{node} to $z_{parent\text{-}to\text{-}parent}$. If a direct path is possible from z_{node} to zparent-to-parent, then the algorithm checks with latter's parent and so on until direct connection is not possible due to an obstacle. It checks for direct connections with successive parents of each node until the collision free condition fails or the start node z_{init} is reached. In this way, the complete path is optimized with straight line connections wherever possible. This substantially reduces the number of nodes present in the path as compared to original path. The nodes at which the path breaks are termed as beacon nodes, $Z_{beacons}$, which forms the basis of intelligent sampling.

Intelligent sampling generates nodes as close as possible to the beacons ensuring that path around the corners is optimized as much as possible. It is started once the initial path has been found.

Algorithm 2: $T= (V,E) \leftarrow$ **RRT*Smart(**z_{init})

```
T ← InitializeTree();
T ← InsertNode(Ø, z_init, T);
for i=0 to i=N do
  if i=n+b, n+2b, n+3b.... then
    z_rand ← Sample(i, z_beacons);
  else
    z_rand ← Sample(i);
    z_nearest ← Nearest(T, z_rand);
    (x_new, u_new, T_new) ← Steer (z_nearest, z_rand);
    if Obstaclefree(x_new) then
      Z_near ← Near(T, z_new, |V|);
      z_min ← Chooseparent (Z_near, z_nearest, z_new, x_new);
      T ← InsertNode(z_min, z_new, T);
      T ← Rewire (T, Z_near, z_min, z_new);
      if InitialPathFound then
        n ← i;
      (T, directcost) ← PathOptimization(T, z_init, z_goal);
      if (directcost_new < directcost_old)
        z_beacons ← PathOptimization(T, z_init, z_goal);
return T
```

Table 2.2: RRT*-Smart Algorithm

Intelligent sampling ensures early optimization around obstacles, something that happens when the number of samples in standard RRT* approaches infinity.

2.6 Triangular Geometrized Sampling Heuristic for Fast Optimal Motion Planning

The algorithm uses triangular geometrized sampling heuristic for optimized sampling to ensure quicker optimization of the path. The successful methods include Incentre (intersection of 3 angles bisectors) and Centroid (point of intersection of three medians). The algorithm works in the following manner: when a random sample g_{rand} is generated, the algorithms redirects the sample by computing the geometric center of g_{init}, g_{goal} and g_{rand}. The geometric center is taken as the new random sample, denoted by g_{nrand}.

The random sample is directed for a fixed number of iterations denoted by variable k, after that uniform sampling is carried out. The value of k is selected to maintain a balance between exploitation and exploration. This allows the proposed planner to exploit the configuration space by sampling the region closer to initial state and goal region and then begin exploring by sampling the remaining region.

2.7 Bi-directional RRT* (B-RRT*)

The algorithm grows two trees each from start and goal point to ensure quick path finding and optimization. The algorithm works as follows. A sample is taken from the configuration space and then various operations are performed on the sample similar to RRT*. After the sample (x_{new}) has been inserted into the Tree (e.g., T_a), bidirectional action is carried out to connect with the other Tree (T_b in this case). Nearest vertex operation is carried to find nearest vertex in the Tree T_b to the inserted sample x_{new}. The nearest node in tree T_b is denoted as x_{conn}. Then, connect procedure is carried out on x_{new}, x_{conn} and T_b. Connect procedure is a slight variation of greedy RRT-Connect heuristic. It first employs the extend function to generate a new node in Tree T_b which is closer to the newly inserted node of Tree T_a, than x_{conn}. That is, it generates a new node in T_b which lies between x_{conn} of T_b and x_{new} of T_a. This node is denoted as x_{new}. Next, it uses NearVertices to find near vertices X_{near} to x_{new}. It sorts these near vertices according to cost in a list L_s. Best parent is chosen from amongst these vertices. If such a node is found, then x_{new} of T_a is connected to this parent of T_b. The connect function returns the cost of this new path (from x_{init} to x_{goal}). If this path cost is less than the existing best path cost then the best path cost is updated to this one.

Next, the trees are swapped and the process restarts, i.e., now sampling is done with T_b and connection is done with T_a.

2.8 Intelligent bidirectional RRT* for optimal motion planning in complex cluttered environments

This algorithm is an improved variant of both RRT* and bi-directional RRT*. The algorithm introduces intelligent sample insertion heuristic for fast convergence to optimal path solution using uniform sampling heuristic. The algorithm is designed for complex cluttered environments where exploration of configuration space is difficult. Contrary to standard Bi directional RRT*, both trees are taken in parallel.

The algorithm works as follows: A random sample is generated whose near vertices are found from both trees denoted as X^a_{near} and X^b_{near}. If both X^a_{near} and X^b_{near} are empty then the nearest vertex is found and added to X^a_{near} and X^b_{near} and connection variable is set to False. If X^a_{near} and X^b_{near} are not empty then X^a_{near} and X^b_{near} are sorted according to cost into L^a_s and L^b_s. The best selected triplets from T_a and T_b are assigned to $\{x^a_{min}, c^a_{min}, \sigma^a\} \in L^a_s$ and $\{x^b_{min}, C^b_{min}, \sigma^b\} \in L^b_s$ respectively. The new node is attached to the Tree with which least c_{min} is associated and rewiring of vertices of that tree is carried out. If the cost of the concatenated path is less than the cost of existing end to end path, then the end to end collision free path is updated. Connection between the two trees is carried out if the Connection variable set during NearVertices call is true. This denotes that there are nodes of both trees present in the ball around the random sample. This makes the algorithm less greedy, i.e., it only makes a connection if there are nodes of both trees present in the circle.

2.9 PB-RRT* / PIB-RRT (Potentially guided RRT* for fast Optimal Path Planning in Cluttered Environments)

B-RRT* and Bi-RRT* don't work well in cluttered environments as both perform pure exploration. The algorithm incorporates Artificial Potential Field (APF) to Bi-directional RRT* and to Intelligent Bi-directional RRT* through the new functions: potentially guided bidirectional RRT* (PB-RRT*) and potentially guided intelligent bidirectional RRT* (PIB-RRT*). Both algorithms use a Bi-directional Potential Gradient [BPG] function, to guide the random sample using APF, rest of the algorithm performs normal B-RRT* and Bi-RRT* using the guided sample. The BPG function works as follows:

If the iteration number is even, the random sample is passed to BPGgoal function which computes the attractive force F_{att} on the sample, given z_{goal} as the attractive pole. Then the distance from

nearest obstacle $d^*_{nearest}$ is computed. If it is less than a constant d^*_{obs}, then the random sample is returned directly. Otherwise sample z_{pb} is directed downhill in the direction of decreasing potential towards goal region in ε sized small steps. The process is repeated for n iterations.

If the iteration number is odd, the random sample is passed to BPGinit function which computes the attractive force F_{att} on the sample, given z_{init} as the attractive pole. The rest of the procedure is same as above. The process is repeated for n iterations or until $d^*_{nearest}$ remains greater than d^*_{obs}. For PB-RRT*, this operation is performed once on T_a (growing from root) and then on T_b (growing from goal) resulting in pulling both trees towards each other and a faster convergence than RRT*. For PIB-RRT* the sample is connected to the tree it is closer to and so on.

CHAPTER 3: REPLANNING ALGORITHMS

The challenges faced by a robot working in real-life environment are quite different. It is highly probable that the mobile robot will face changes in some or all of the following.

- Change in the goal point
- Change in the position of the robot as it starts to move
- Change in the obstacles of its environment
- Change of the environment altogether

Therefore, an algorithm that is envisaged to work in real-life shall be able to cope with all of these changes. Traditional single query and multi query algorithms can handle these changes, albeit with certain limitations.

3.1 Using Single Query Algorithms for Re-planning

In this section, we will see consider how single query algorithms can be used to respond to the changes mentioned above and what are the limitations associated with them. As RRT* is one of the most successful single query algorithms, it will be used for the comparison. Any adaptabilities or limitations of RRT* against these requirements can be extrapolated to apply to all sampling based single query algorithms.

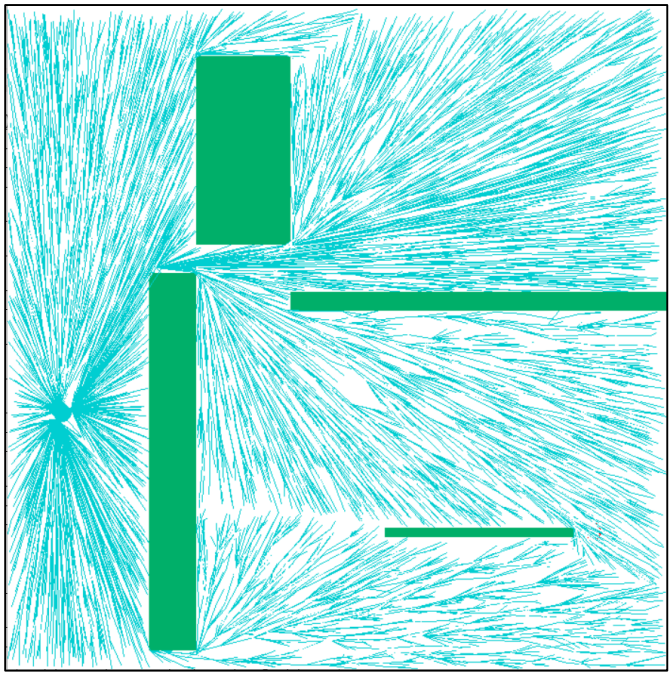


Figure 3.1: RRT* in a Typical Environment

A single query algorithm such as RRT* can be modified to handle goal point changes. As shown in figure 3.1 above, the RRT* tree already contains optimal paths to almost all points in the configuration space. In case of any change in the goal point, a simple reverse traversal from the point closest to the new goal point to the root would provide the optimized path to the new goal point. Therefore, RRT* shall be able to cope up with goal point changes, however, this might not be possible with the movement of the robot.

In a real-life environment, the robot will not be content with just finding the optimized path to the goal. Rather it will have to walk the talk, i.e., it will have to follow the path down to the goal. RRT* is able to provide an optimized path to the goal and the robot can be programmed to move once the path is found. The RRT* tree, however, becomes obsolete as soon as the robot starts movement. It will not be able to handle any goal changes after the movement of the robot, as the tree root is no longer centered at the robot's current position. Such a scenario will require running the algorithm from scratch which will be infeasible. Some alternate algorithms keep the tree root centered with the movement of the robot. This adaptation can enable the algorithm to handle goal changes as the tree will remain intact and a path can be traced from any point back to the root which is the current position of the robot.

Other variations of RRT* enable the algorithm to handle dynamic obstacles during execution. However, changes in any major obstacles during the execution of the program do not make sure that optimality is retained. The algorithm also becomes intractable when major changes are done. If the environment is changed altogether, e.g., the robot moves to a new environment, RRT* needs to be restarted from scratch and it consumes a lot of time to find an optimized path in the new environment.

As explained above, standard RRT* is meant for single time use. It is not able to cope up with the requirements faced by a robot in practical usage. A few variations of RRT* are able to partially respond to some of these changes but none would be able to respond to all of them in real-time. Thus single query algorithms would not be useful for real-time movement of the robot.

3.2 Using Multi Query Algorithms for Re-planning

Multi-query algorithms, as described in previous chapter, are quite robust but require extensive pre-processing time. However, the amount of time taken for construction of the graph during the preprocessing phase would be compensated by its re-usability.

These algorithms, such as PRM*, can easily handle any goal point changes, as the new path from start to goal point is only a search away. The start point of the robot can be kept updated along with the movement of the robot. Additionally, no requiring would be required with the robot's movement as is required by single query algorithms such as RRT*.

Handling of dynamic obstacles along the way seems to be a cumbersome task for multi query algorithms as any movement of the obstacle would affect a complete portion of the graph. New edges will have to be defined between the vertices followed by search for an alternate path.

If the environment is changed altogether, the complete graph will have to be regenerated from scratch, before the robot can take a single step along the path. This would be expensive both in terms of time and computations, which cannot be performed in real-time.

Therefore, both single and multi-query algorithms are in-efficient when it comes to real-life environment where the robot should be able to respond in a real-time manner.

3.3 Re-planning Algorithms

Re-planning algorithms focus on reusing the existing solution to respond to any changes in the goal, the obstacles or the environment. Recently proposed re-planners include Online RRT* & FMT*, RT-RRT * etc. To cater for demands of real-life environment, the algorithms perform various tasks as described below.

The re-planners interleave various tasks which include movement of the robot, shifting of tree root to the robot's new position, rewiring of the tree, further expansion of tree and acceptance of new goal.

The robot starts moving as soon as a path to the goal is found. The algorithm has to ensure that current position of robot is always maintained as the tree root. This is achieved by rewiring of the tree along with movement of the robot. Once the robot has taken a single step along the path, that is, from tree root to the child of the root along the path, the root is also shifted to its first child on the path. The parent child relationship of the first child and original root is inverted. The child is made the root of the tree by setting its parent to None while the original root's parent is changed to new root. Rewiring of the tree is then carried out to update the costs (and parents) according to the new root. The rewiring process gets computationally expensive as the tree size increases. So some algorithms do selective rewiring based upon gird based spatial indexing, selective rewiring or random rewiring.

This process is also accompanied by sampling of new points in the configuration space. To keep the tree size within limits, the algorithms do not add points beyond the predefined tree size. However, new sampling remains in progress and is used to optimize the rewiring process. Online RRT* & FMT* sample a new point but do not add it to the tree. The point is used to select point of rewiring, i.e., least cost node is selected in the neighborhood of the newly sampled point which is used for standard RRT* rewiring. The algorithms also accept any goal change whose path is yielded from the same start point.

The current re-planning algorithms are able to find paths to alternate goal points in real-time and enable robot's movement along with it, but they do not focus on the optimality of the paths generated by the algorithm and focus only on providing a path in real-time.

CHAPTER 4: RAPIDLY RE-PLANNING RRT*

4.1 Background

In the previous chapters, we have seen the utility of current motion planning algorithms for real time path planning. An overview of single query path planning algorithms and re-planning algorithms was presented.

The existing single query algorithms are not meant for real-time use or for re-planning. They have to be run from scratch every time there is a change in the environment, the location of the robot, the location of its goal point or any change in the environment. If there is a change in the location of the robot, the root of RRT* tree would no longer remain at position of the robot rendering all path planning through the tree useless. Since the RRT* tree provides an optimized path to all points of the workspace, handling the change in goal point should be straight forward, however, no such provision exists in the current algorithms. If there is any change in the environment, some algorithms exist for handling dynamic obstacles but no algorithms exist for handling major environmental changes, such as those in which the major obstacles are changed or where the robot's location is changed altogether.

All of these changes would require the algorithms to be restarted from scratch which is expensive in terms of time and computation. Any single query algorithm fashioned for real-time use would also be non-optimal due to the requirement of providing the result in real-time. Additionally, restarting the tree at every step of the robot would be highly inefficient.

The multi-query algorithms are not suitable for real-time use as well. The first problem is the time required for pre-planning phase. If the environment is changed at an instant, the whole pre-planning is also rendered useless and has to be restarted from scratch.

Re-planners provide a solution for handling real-time movement of the robot. The current re-planning algorithms, however, do not consider optimality of the path yielded by the algorithm.

4.2 Introduction to Rapidly Re-planning RRT*

Rapidly Re-planning RRT* or R4T* is a re-planning algorithm which aims to address the requirements necessary for the robot to perform efficiently in a real-time environment. It enables the robot to move in a new environment as soon as a path to the given goal point is found. The robot starts movement along the path which is further optimized on the go. Any new goal point, which is given to the robot during this process can also be handled by the algorithm. The new goal point can be assigned irrespective of whether the robot has reached the goal point or not. The algorithm immediately searches for the path from the current position of the robot to the new goal point. The new path is followed by the robot immediately.

The Rapidly Re-planning RRT* algorithm initially starts with RRT*, which is used to yield initial path to the goal. The algorithm enables continuous optimization of the path along with further spawning of the RRT* tree. Any change in the goal point, at this point, is handled by RRT* which gives an optimal path from the goal to robot's position. During spawning of RRT* tree, a Smart-Graph is built after every n-thousandth nodes of the RRT* tree. Once the predefined number of nodes of RRT* have been reached, further paths are yielded by the Smart-Graph. If the environment is changed altogether, the algorithm restarts with RRT* followed by other elements of the algorithm.

Consider a real world scenario, where for instance a pet robot is required to follow a particular object such as a human being. Using this algo., when the pet reaches a new room, it is able to find the path to the human using R4T*. As soon as the path is found, the robot starts its approach towards the human following this path. The path is continuously improved using RRT*. If the human moves about the robot is able to update the goal and find a new path to the target. After the initial development of the algorithm, the paths are yielded by the Smart-Graph which ensures optimized paths to any new goal points in the given environment. If the human moves to a new environment the process is restarted. However, the pet is able to move immediately in the new environment as soon as the path is found.

4.3 Overview of Rapidly Re-planning RRT* Algorithm

The algorithm builds upon RRT* to generate a real-time algorithm able to respond to real-life requirements faced by the robot. Contrary to the re-planners previously described, the algorithm is able to yield paths to alternate goal points which are equal or exceeding in optimality when compared to the path generated by a 7000 node standard RRT* algorithm.

To perform this task, the R4T* algorithm develops a Smart-Graph which comprises of RRT* nodes present around the corners of the configuration space obstacles. The Smart-Graph is essentially a visibility graph which the vertices are the nodes at the corners of the obstacles while the edges are formed between the vertices which are mutually visible to each other. Developing a visibility graph directly in high dimensional spaces is computationally difficult whereas a Smart-Graph can easily be built in high-dimensional spaces where RRT* already works efficiently. The following figures show a typical visibility graph.

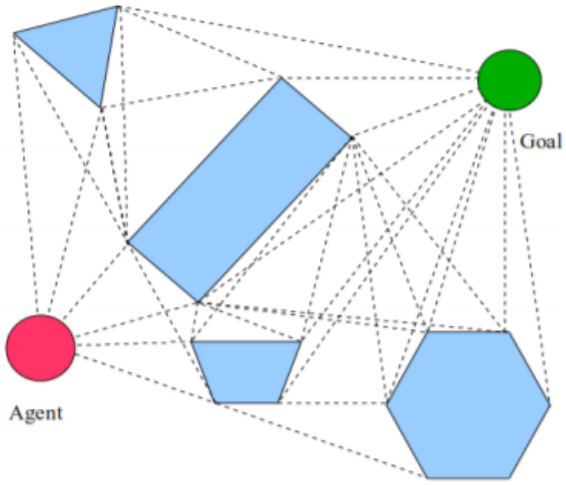


Figure 4.1: Visibility Graph

These nodes are found using a modification of RRT*-Smart algorithm. The RRT*-Smart algorithm, as described earlier, searches for nodes around the corners of configuration space obstacles. These nodes are then used for further sampling of nodes close to these corner nodes in order to optimize RRT* path around the corners.

Randomly Re-planning RRT*, however, uses only the first part of RRT*-Smart. The algorithm is used to find the nodes around the corners of the configuration space obstacles after every n x 1000 nodes of RRT* tree have been spawned. These nodes are added to a graph called Smart-Graph. Unlike other re-planners, RRT* is stopped once a predefined number of nodes has been achieved after which all the paths are yielded by Smart-Graph. The algorithm enables the Smart-Graph to be developed from an existing RRT* tree or it can be developed alongside an RRT* tree being built. At the end of this phase, we have a Smart-Graph with small number of vertices which is able to yield optimized paths between any two points with computation time equivalent only to the time invested in searching through the graph. The paths between the any pair of start and goal points can be found by adding both of these points to the Smart-Graph and finding the optimized path from start to goal point. The optimized path is found using Dijkestra's algorithm. Due to reduced number of nodes in the Smart-Graph, the time taken to find the paths is only equal to the search time of Dijkstra's algorithm.

4.4 Structure of the Algorithm

The algorithm is divided into four basic sections, which handle the following functions. basic functions including:

i) Basic RRT* path planning to the goal,

ii) Movement towards the goal point as soon as a path is found or max number of RRT* nodes has been reached,

iii) Building/update of Smart-Graph at every n^{th} (user-defined) node of RRT*

iv) Handling of goal changes at any point.

The RRT* algorithm is run for user defined number of nodes, after which RRT* and update of Smart-Graph is stopped and other functions carry on until the end of the program.

The algorithm starts with RRT* which is used to find path from a start to goal point (both randomly generated). As soon as a path is found, the agent starts navigating it by moving a single step at a time. In one step, the robot moves from current root of the tree to first child along the path. To keep RRT* algorithm progressing, the tree-root is also moved to the new position of the robot and rewiring is carried out accordingly. This is similar to the concept used in RT-RRT* and Online FMT* & RRT*. Rewiring in R4T* is carried out at a single level to ensure that minimal time is consumed in the process.

The Smart-Graph is generated using a modified version of RRT*-Smart, which is applied on leaf nodes of RRT* tree after enough new nodes have been generated. RRT*-Smart algorithm finds the nodes around the corners of the obstacles, which are subsequently added to the Smart-Graph.
The R4T* algorithm has the capability of handling goal point changes at any time. Whenever a new goal point is selected, its path from the current tree root (agent's position) is found using both RRT* and Smart-Graph. The RRT* path is followed by the agent as long as the RRT* tree (and Smart-Graph) is being built, afterwards Smart-Graph's path is followed.
The algorithm is tested rigorously for multiple start and goal points in three different 2D environments with different obstacle configurations. The optimality of Smart-Graph path is shown empirically by comparing its generated paths with those generated by a standard RRT* tree for the same goal points. The algorithm is shown to generate paths which are similar or better in optimality to the corresponding 7000 node RRT* trees. In terms of size, the Smart Graph in a typical environment comprises of fewer nodes as compared to RRT*, which enables searching and finding new paths with minimal additional overload.
The R4T* thus provides an efficient multi-query function without the limitations imposed by either single-query or multi-query planners.

4.5 Related Work

In this section, the algorithms which are used in Rapidly Re-planning RRT* algorithm are explained in detail.

4.5.1 RRT* Algorithm

A brief description of RRT* is given in this section as it is the basic building block of R4T*. RRT* is a sampling based algorithm which incrementally samples points in the configuration space, trying to find and optimize the paths to any point in the configuration space from the start point. In doing so, it tries to find an optimal path to the goal point z_{goal}.
Let X represent the configuration space of the robot with X_{obs} defining the space occupied by obstacles. X_{free} is the free space, void of obstacles, defined as $X_{free} = X \backslash X_{obs}$. The initial point is defined as z_{start} while the goal point is defined as z_{goal}. The RRT* algorithm attempts to find a control input $u = [0,T]$ that results in a feasible path $x(t) \epsilon X_{free}$, such that $x(0) = z_{start}$ and $x(T) = z_{goal}$. The algorithm constructs and maintains a tree $T= (V, E)$ which comprises of a set of vertices (nodes) V joined by edges E, such that both vertices and their interconnecting edges lie completely in X_{free}.

The algorithm is outlined in Algorithm-4.1. A brief description of the functions of the algorithms is given below.

The algorithm starts with initialization of the tree T with the start node z_{start}. The nodes are sampled to a predefined maximum *nodesmax.*

Algorithm-4.1 **RRT* (T, G, nodesmax)**	
1.	T ← InitializeTree()
2.	T ← InsertNode(z_{start}, T)
3.	**While** (i < nodesmax)
4.	z_{rand} ← RandomSample(i)
5.	$z_{closest}$ ← ClosestNode(z_{rand}, T)
6.	z_{new} ← Extend($z_{closest}$, z_{rand})
7.	If ObstacleFree(z_{new}):
8.	Z_{near} ← CloseNodes(z_{new}, T,|V|)
9.	z_{parent} ← ChooseParent(z_{new}, Z_{near})
10.	T (N, E) ← InsertNode(z_{new}, z_{parent}, T)
11.	E ← Rewire(z_{new}, Z_{near}, E)

RandomSample:

The function generates a random point z_{rand} in the obstacle free space in the environment, i.e., $z_{rand} \in X_{free}$

ClosestNode:

The function returns a node, $z_{closest}$, in the tree T which is closest (based on Euclidean distance) to the random point z_{rand}, generated by the function *RandomSample*.

Extend:

The function takes a single step from $z_{closest}$ in the direction of z_{rand} and returns this point in the form of z_{new}.

ObstacleFree:

The function checks for two things. First that z_{new} lies in X_{free} and second that the path from $z_{closest}$ (current parent of z_{new}) to z_{new} completely lies in X_{free}. If any of these conditions are not met, the function returns 0.

CloseNodes:

The function returns the set of nodes Z_{near} which present around z_{new} in a ball of volume whose radius is given by $r = \gamma (\log n/n)^{1/d}$ where γ is a constant, n is the number of nodes and d is the dimension of the state space.

ChooseParent:

The function chooses a node z_{parent} from Z_{near}, to be the parent of z_{new} through which the cost to reach z_{new} from the root of the tree is lowest.

InsertNode:

The function adds the node z_{new} to the tree T with z_{parent} as its parent.

Rewire:

The function rewires the nodes in Z_{near}, by making them a child of z_{new} if the cost to reach them is lower through z_{new} than through their existing parent.

4.5.2 RRT*-Smart Algorithm

The algorithm provides optimal paths with a faster rate of conversion, as compared to RRT*, by using Path Optimization and Intelligent Sampling.

Path Optimization:

This function optimizes the path found by RRT* nodes based upon visibility between the nodes in the path. Once a path has been found by RRT*, the algorithm starts from z_{goal} and tries to connect z_{goal} directly with the parent of its parent through a straight line path which lies completely in X_{free}. If the connection is possible, the same is checked with the next parent up the tree (i.e., from z_{goal} to its parent's parents' parent). The process is repeated until a node in the path is reached, to which a direct connection is not possible. In this case, z_{goal} is connected directly to the last successful connection and the process is restarted from the node to which the connection failed. The process is repeated iteratively until the whole path has been traversed and z_{start} has been reached. The process results in an optimized path with least number of nodes which are called beacons.

Intelligent Sampling:

Intelligent sampling is carried out using the beacons identified during Path Optimization. As the beacons are nodes around the corner of the obstacles, the algorithm tries to sample more points in a ball around the beacons enabling further optimization around the corners resulting in an optimized path.

4.5.3 Rapidly Re-planning RRT* [R4T*]

This section describes Rapidly Re-planning RRT* algorithm in detail. The pseudo-code is outlined in Algorithm-4.2.

Algorithm-4.2: R4T* ()	
1.	T ← InitializeAll()
2.	T ← InsertNode(z_{start})
3.	**While** run **do**:
4.	**If not**(path_fnd) **do**:
5.	(T,G, path_fnd, nodes_cnt) ← RRS(T)
6.	**if not**(path_fnd):
7.	$path_{new}$ ← WindupGoal(z_{goal}) path_fnd ← 1
8.	**If not** (Tree_comp) **do**:
9.	(T,G) ← RRS50(T)
10.	(z_{goal}, $path_{new}$, goal_set) ← RedefineGoal()
11.	**If** goal_set **do:**
12.	**If** start_vert **in** G.vertices:
13.	G ← RemVert(goal_vert, start_vert)
14.	(G, $path_{smrt}$, $cost_{smrt}$) = SmartPath (G, z_{goal})
15.	**If not**(Tree_comp) **do**:
16.	T ← MoveRw($path_{new}$, T)
17.	**else**:
18.	G ← MoveSmrt($path_{smrt}$, G)

The algorithm starts with creation of the Tree *T* and its initialization with the start node z_{start}. The *run* variable executes the algorithm until it is terminated by the user. The initial start and goal points, z_{start} and z_{goal}, are selected randomly at start of the program. This is followed by execution of *RRS* function [line 5] to find an optimal path to z_{goal} and develop the Smart-Graph along with it. *RRS* is a combination of RRT* and RRT*-Smart algorithms, where RRT*-Smart is executed after every 1000^{th} node of the RRT* to develop the Smart-Graph.

In the first part of the algorithm, *RRS* is run until a path to goal is found or maximum number of RRT* nodes have been reached. The function returns to the main program as soon as any of these two conditions is met. [lines 4 & 5]

If the goal has not been found but maximum number of nodes has been reached, then *R4T** finds the path to the node closest to z_{goal} through *WindupGoal* function. [lines 6 & 7]. On the other hand, if the goal has been found by RRS but max number of RRT* nodes has not been reached, then the further nodes of RRT* are generated through RRS50 algorithm in the second part of the R4T*. By the end of first part [lines 4 to 7], the R4T* has yielded a path to the goal or to the closest node to goal.

The second part of the algorithm performs four tasks which are executed iteratively. These tasks include: RRT* tree expansion (up to max nodes count), acceptance of new goal with derivation of its path from the existing tree, Smart-Graph update and movement of the robot. The variable *Tree_comp* is used to follow the completion of the Tree; it allows further generation of 50 RRT* nodes and Smart-Graph update (through RRS50 function) in every iteration [line 8-9]. The function *RedefineGoal* [line 10] selects the new goal point (based on mouse click) and returns it as z_{goal}. It also returns the path to z_{goal} from current position of the robot as $path_{new}$ and a *goal_set* variable which denotes that goal has been changed.

If the goal has been changed, the function *RemVert* removes the old start and goal vertices from Smart-Graph G, if they exist in the graph. This is controlled by the variable *start_vert*, which is initialized as None in *InitializeAll* function. *Start_vert* is assigned a value during first execution of *SmartPath* function, so it would not exist in the graph if the *SmartPath* function has not been executed before.

SmartPath function [line 14] adds the current position of the robot and new goal point as vertices of the graph G and then computes the path from start to goal point by using Dijkstra's search

algorithm. If *Tree_comp* variable is 0, i.e., the RRT* tree is not complete, the robot follows the RRT* path through *MoveRw* function.

The RRT* tree root is centered at the robot's position. The *MoveRw* function moves the robot from the tree root to its first child along $path_{new}$. The tree root is then shifted to the new position of the robot and limited level of rewiring is carried out to update the parent-child relationships and the associated costs.

By keeping the tree root at the robot's current position, path to any new goal point within the workspace can easily be found from the robot's current position.

If the *Tree_comp* variable is '1', i.e., the RRT* tree is complete, further spawning of RRT* nodes is stopped and robot movement is handled directly by the Smart-Graph through *MoveSmrt* function. The algorithm continues in the same manner until the program is terminated.

The functions used in the Rapidly Re-planning RRT* algorithm are described in detail below.

RRS [Algorithm-III]

RRS is the main function of R4T* which handles three critical functions, which include, modified form of RRT*, RRT*-Smart and graph building algorithm. RRS function is outlined in Algorithm-4.3.

Algorithm-4.3 **RRS (T, G, nodesmax)**	
1.	**While** (i < nodesmax)
2.	z_{rand} ← RandomSample(i)
3.	$z_{closest}$ ← ClosestNode(z_{rand}, T)
4.	z_{new} ← Extend($z_{closest}$, z_{rand})
5.	Z_{near} ← CloseNodes(z_{new}, T)
6.	z_{parent} ← ChooseParent(z_{new}, Z_{near})
7.	**If** z_{parent} **then**
8.	T (N, E) ← InsertNode(z_{new}, z_{parent}, T)
9.	E ← Rewire(z_{new}, Z_{near}, E)
10.	**If** dist (z_{new}, z_{goal}) < step_size **then**
11.	path_fnd = 1
12.	**If** not (i%1000) **or** path_fnd **then**
13.	L ← FindLeaves(T)

14.	Vert ← RRTStarSmart (L)
15.	G ← DefineVertices(Vert)
16.	(G,E) ← CompVisbility(G)
17.	**If** path_fnd **then**
18.	**Break**
19.	**If not** (nodes_cnt < nodesmax) **then** Tree_comp = 1
20.	**return** (T, G, Tree_comp, nodes_cnt)

The function starts with standard RRT* from line 2–9. RRT* and its functions have already been detailed previously.

The modification starts from line 10. The *path_fnd* variable is turned to 1, if the new node z_{new} is within step_size of the goal. This variable controls exit from the RRS function.

The next part of the function develops the Smart-Graph [lines 12 to 15]. These lines are executed every thousandth node and before exiting RRS when *path_fnd* becomes 1.

There are four functions in this portion of the algorithm, which are detailed below.

FindLeaves: This function searches the tree T and returns a list, L, of leaf nodes present in the tree. The function is followed by *RRTStarSmart*.

RRTStarSmart: A slight modification of original RRT*-Smart is used here to find corner nodes in the tree T using the list L of leaf nodes. The algorithm takes a leaf node from the list L and starts traversing back from it towards the root. It tries to make a straight line connection between the leaf node and its preceding nodes, as shown in Figure 4.2.

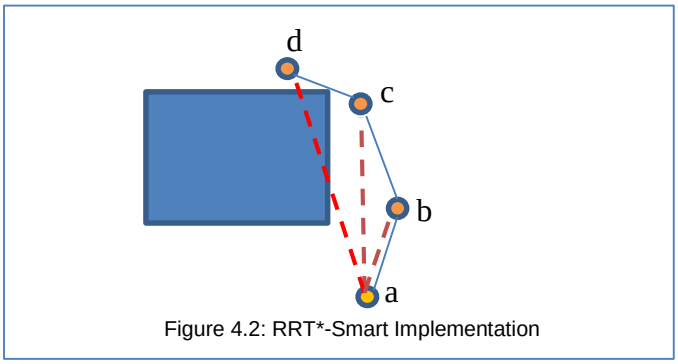


Figure 4.2: RRT*-Smart Implementation

In the figure, the leaf node is shown as ‘a’, its parent nodes are shown as b, c and d while the obstacle is shown as blue block. RRTStarSmart tries to connect ‘a’ to ‘b’ through a straight line, once it is successful it tries to connect ‘a’ to b’s parent, i.e., ‘c’. This continues until it reaches the node ‘d’, with which direct connection is not possible due to presence of the obstacle. The algorithm marks ‘c’ and ‘d’ as corner nodes and restarts the process from the last node with which connection was unsuccessful, i.e., node ‘d’. Now ‘d’ is taken as the main node and visibility from ‘d’ to its ancestors is checked. The process continues until the paths from all the nodes in L have been checked and corner nodes found. Any node that has been checked once is not checked again during single iteration of RRTStarSmart. The function returns the corner nodes as a list, Vert.

DefineVertices: This function updates the vertices of graph G, by adding the nodes from *Vert* which are not already present in the graph.

CompVisibility: The function defines edges between the vertices of the graph G, if a straight line connection is possible between them. As the number of nodes in the tree T increases, the Smart-Graph developed from them represents the workspace in a better manner. Figures 3-5 show the Smart-Graph developed after 1000 and 5000 nodes for randomly generated initial start and goal points for three different environments.

Line 17-18 break the While loop if the path has been found, otherwise the program continues in the loop. Once outside the While loop, the program checks for completion of the tree and if so turns *Tree_comp* to 1. Line 20 returns to the main function.

The RRS function returns updated tree T, graph G, path_fnd variable and the current number of nodes in the tree T.

WindupGoal [line 7, R4T]*: The function returns a path from the tree root to given goal point by finding the least cost node in neighborhood of z_{goal} and traversing back from it to the tree root.

RRS50: The function is same as RRS function, except the following changes. The *nodesmax* value is set at current number of nodes in the tree + 50 which ensures that only 50 new nodes are added to the tree in one call to the function. Secondly, the RRS50 does not have path_fnd variable as the path to goal has already been found. Similar to RRS, the function updates the Smart-Graph in case the total number of nodes reaches a multiple of 1000 during execution of the function. The function is executed in every iteration of R4T* until the predefined number of RRT* Tree nodes are achieved. Once the predefined number is achieved *Tree_comp* is turned 1 which stops the execution of this function.

RedefineGoal: The function enables acceptance of a new goal. For simulation, it takes the goal input directly from the mouse click position. The selected location is returned as z_{goal}. The function also returns the path to z_{goal} using *WindupGoal* and sets the *goal_set* variable 1 when a new goal is selected.

The *goal_set* variable returned by *RedefineGoal* denotes that a new goal has been chosen. If the variable is 1, the algorithm uses *SmartPath* function [line 14] to find $path_{smrt}$ to the new goal. This path, however, has to be found from the current position of the robot to the new goal point z_{goal}. Therefore, the previous start and goal vertices in the graph G, denoted by the variable *start_vert* & *goal_vert*, have to be removed before the execution of *SmartPath*. This is achieved by RemVert function [line 13].

RemVert: The function is executed if *start_vert* and *goal_vert* vertices have already been added to *Smart-Graph*. These variables are initialized as None in *InitializeAll*, and assigned during execution of *SmartPath*. Hence, they would still be None during the first execution of the R4T* which would skip the execution of *RemVert*. *RemVert* removes these vertices from the graph to make way for the new ones during the up-coming iteration of *SmartPath* function.

SmartPath [Algorithm-4.4]:

	Algorithm-4.4 **SmartPath(**$path_{new}$**,** z_{goal} **,**G**)**
1.	start_vert ← Vertex($path_{new}$[0])
2.	goal_vert ← Vertex(z_{goal})
3.	G ← Insert(start_vert, G)
4.	G ← Insert(goal_vert, G)
5.	G ← CompVisbility (G)
6.	G ← Dikestra (G, start_vert)
7.	$path_{smrt}$ ← shortest(goal_vert, G)
8.	$path_{smrt}$ ← Optimize($path_{smrt}$)
9.	**Return** start_vert, goal_vert, $path_{smrt}$

The algorithm starts by adding two new vertices to the *Smart-Graph*. The first vertex is the current position of robot, which is always maintained as the first element of $path_{new}$ (the path retuned by *WindupGoal* based on RRT*). The *Vertex* function defines the current position as an object of vertex class. Similarly, z_{goal} is added as another object of vertex class. Both *start_vert* and *goal_vert*

are added to the *Smart-Graph*. *CompVisibility* [line 5] is used to redefine edges amongst the vertices of the updated graph. This is followed by application of Dijkstra's algorithm to find the minimum cost path between *start_vert* and *goal_vert*. The algorithm is implemented in two functions namely *Dikestra* and *Shortest* and the final path is optimized further through *Optimize* function. *Dikestra* finds minimum cost to reach *start_vert* from any vertex of the graph, while *Shortest* starts from the *goal_vert* and makes its way back to the *start_vert* by following the least cost parent assigned by *Dikestra*. *Optimize* function further optimizes the generated path by dividing straight lines in the $path_{smrt}$ into small equal length segments. A vertex is defined at the end of each segment and a mini-graph of these segments is created, complete with edges and vertices. Dijkstra's algorithm is applied on the mini-graph to yield a further optimized path. Since mini-graph size is very small, the improvement in path is substantial with little computational load. The path generated by the algorithm is returned as $path_{smrt}$.

MoveRw[Algorithm 4.5]:

Algorithm-4.5 **MoveRw($path_{new}$, T)**	
1.	If len($path_{new}$) > 1:
2.	$path_{new}[0] \leftarrow path_{new}[0].RemChild(path_{new}[1])$
3.	$path_{new}[0].parent \leftarrow path_{new}[1]$
4.	$path_{new}[1].parent \leftarrow$ None
5.	$path_{new}[1] \leftarrow path_{new}[1].AddChild(path_{new}[0])$
6.	$path_{new}[1] \leftarrow path_{new}[1].Update\ Cost()$
7.	$path_{new} \leftarrow$ del ($path_{new}[0]$)
8.	E ← GoRewire ($path_{new}[0]$, E)
9.	**return** $Path_{new}$

This function is utilized in case of RRT* tree not being complete, denoted by *Tree_comp* variable. Its objective is to move the robot to the next point along the path while keeping the tree root centered at the new position of the robot. This keeps the RRT* algorithm intact and allows further nodes

spawning along finding of path to any new goal point at any time. The concept of moving the root along with robot's position is similar to that presented in [12] and [13].

MoveRw moves the robot one step i.e., from current tree root to first child along the path, shifts the tree root to the robot's new position and adjusts the parent-child relationships accordingly. Further it rewires around the new root to update the costs and parents according to low cost to reach rule.

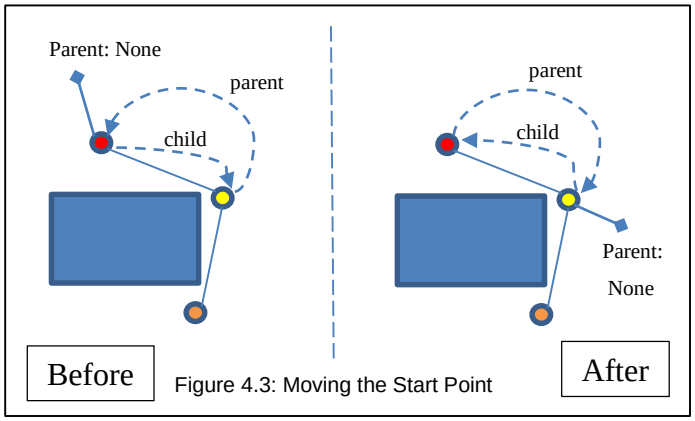


Figure 4.3: Moving the Start Point

The current position of the robot is at $path_{new}[0]$ and the first child along the path is at $path_{new}[1]$. Figure 4.3 depicts the movement process where $path_{new}[0]$ is shown as red node while $path_{new}[1]$ is shown as yellow node. Lines 2-5 of Algorithm-V, shift the tree root from $path_{new}[0]$ to $path_{new}[1]$. To do so, in line 2, $path_{new}[1]$ is removed from children of $path_{new}[0]$. In line 3, $path_{new}[1]$ is made the parent of $path_{new}[0]$. Line 4 assigns the parent pointer of $path_{new}[1]$ as None because it is now the root of the tree and line 5 adds $path_{new}[0]$ to the children of $path_{new}[1]$. The cost of $path_{new}[1]$ is then updated to reflect that it is now the root of the Tree. At this point, it is assumed that the robot has moved to its new position, so $path_{new}[0]$ is removed from the path list.

With the change in root node of the tree, the cost to approach the root from all the nodes has been changed as well. Therefore, rewiring of the tree is required to update the costs and parents (where required). Updating the whole tree at every step of the robot is a time consuming process and will cause the algorithm to be non-real time. Rewiring is thus limited to a single level around the root node. This does not substantially affect the optimality of the paths generated by rewired RRT* and speeds up the whole process of movement.

Rewiring during movement is implemented through *GoRewire* function. The function rewires the nodes present in the neighborhood of the root node and makes the tree root the parent of these nodes. The function works exactly like the *Rewire* function of RRT* except that z_{root} takes the place of z_{new}, and the neighborhood nodes are automatically connected to it because the cost to reach the

z_{root} is only the distance between them. Since a part of the tree is rewired, the path from root to goal is recomputed using *WindupGoal*. The command for actual movement of the robot can easily be embedded in the algorithm to ensure physical movement along with update in the tree root.

MoveSmrt: This function is executed after complete RRT* tree has been spawned and the generation of Smart-Graph has been completed. It simply moves the robot to the next vertex in $path_{smrt}$ and updates $path_{smrt}$ by removing the previous vertex.

4.5.4 Performance of R4T*

The algorithm starts developing the Smart-Graph after 1000^{th} node of RRT* has been spawned. With every additional 1000 nodes, the Smart-Graph is further updated enabling it to cover increasing area of the workspace hence yielding optimal results even at fewer nodes as compared to RRT*.

The algorithm was tested in three environments with different obstacles sizes and configurations. Figures 4.4, 4.5 and 4.6 shows the coverage status of Smart-Graph in these environments after 1000 and 5000 nodes of RRT*.

In the figures above, RRT* tree is shown in Cyan while the Smart-Graph edges are shown in red. The vertices of the Smart-Graph are shown as blue dots. As depicted in the figures, by 5000 nodes, maximum coverage of the workspace has been achieved by the Smart-Graph which enables it to yield optimal paths between any two points in the workspace.

Number of vertices of the Smart-Graph for different environments is also shown in the figures. In Environment-III, which has most obstacles and corners, the maximum number of vertices comes out to be 356 against 5000 RRT* nodes.

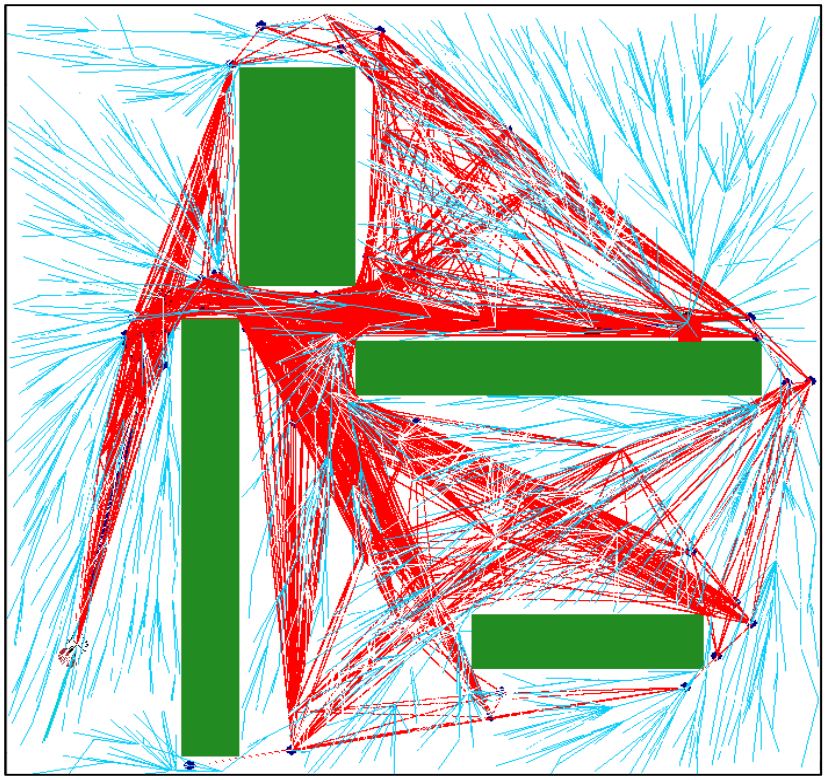
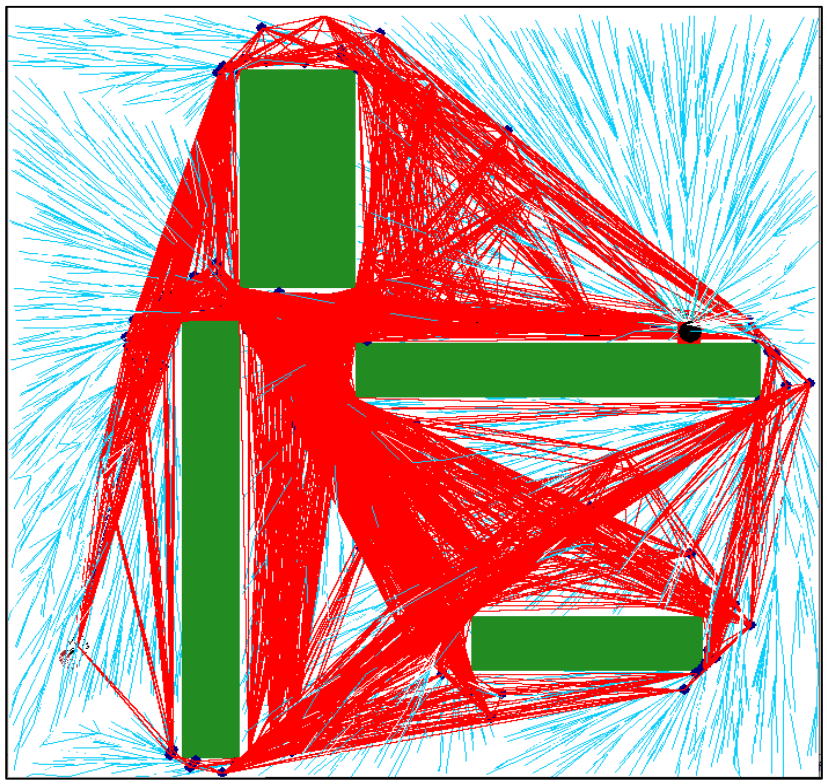


Fig. 4.4: Environment-I; Smart-Graph at 1000(above) and 5000 RRT* nodes (below).
Number of Vertices: 42 & 163 resp.



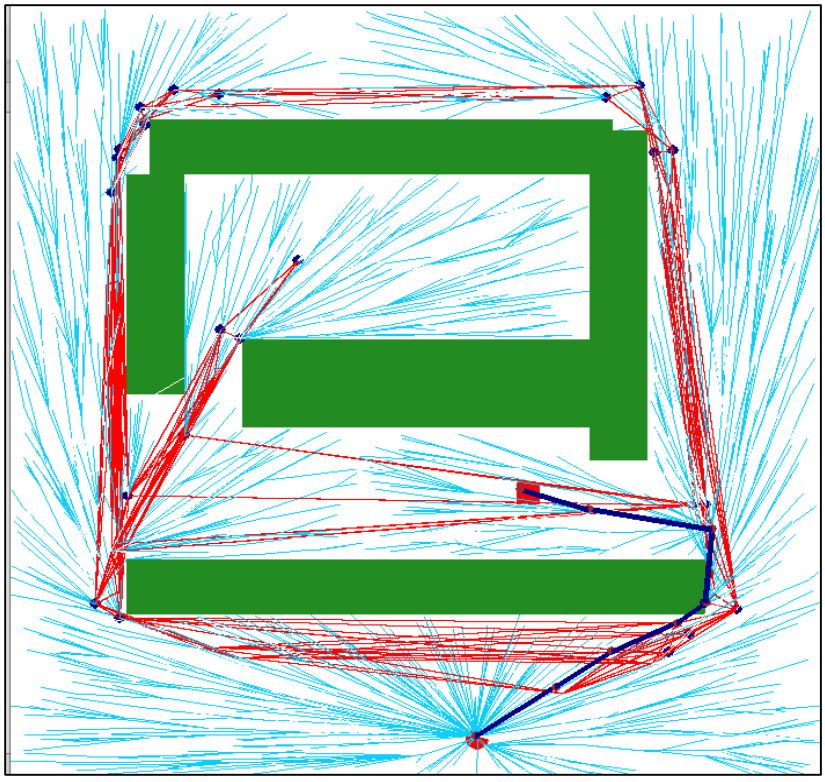
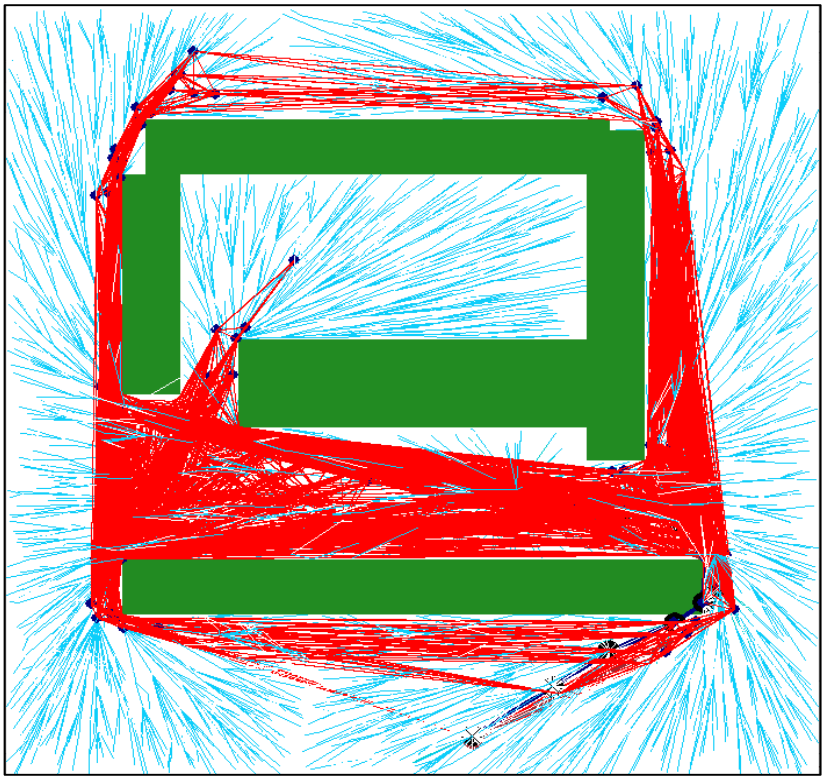


Fig. 4.5: Environment-II; Smart-Graph at 1000 (above) and 5000 RRT* nodes (below).
Number of Vertices: 48 & 135 resp.



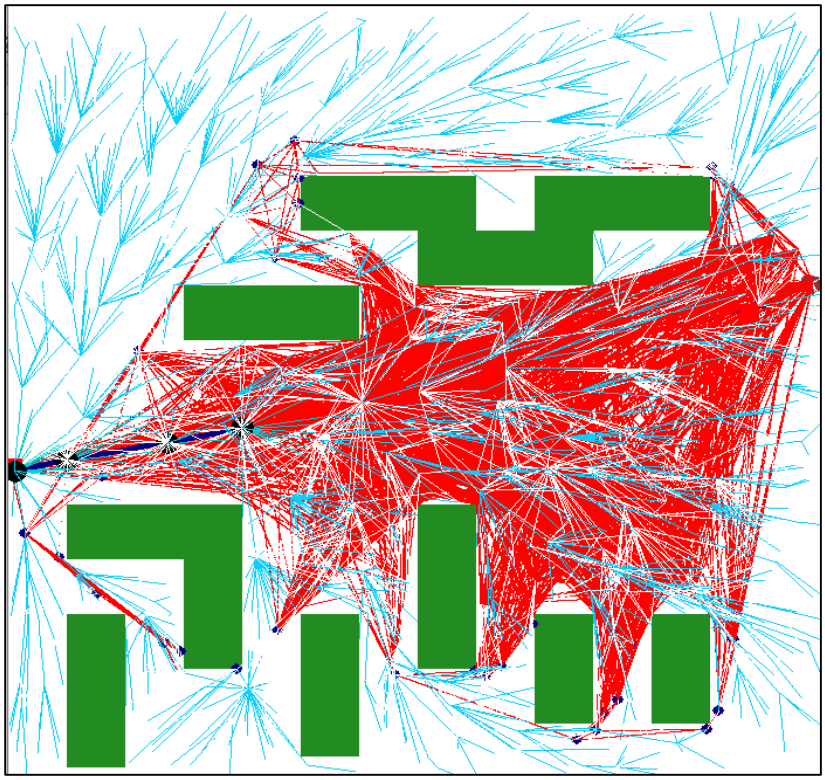
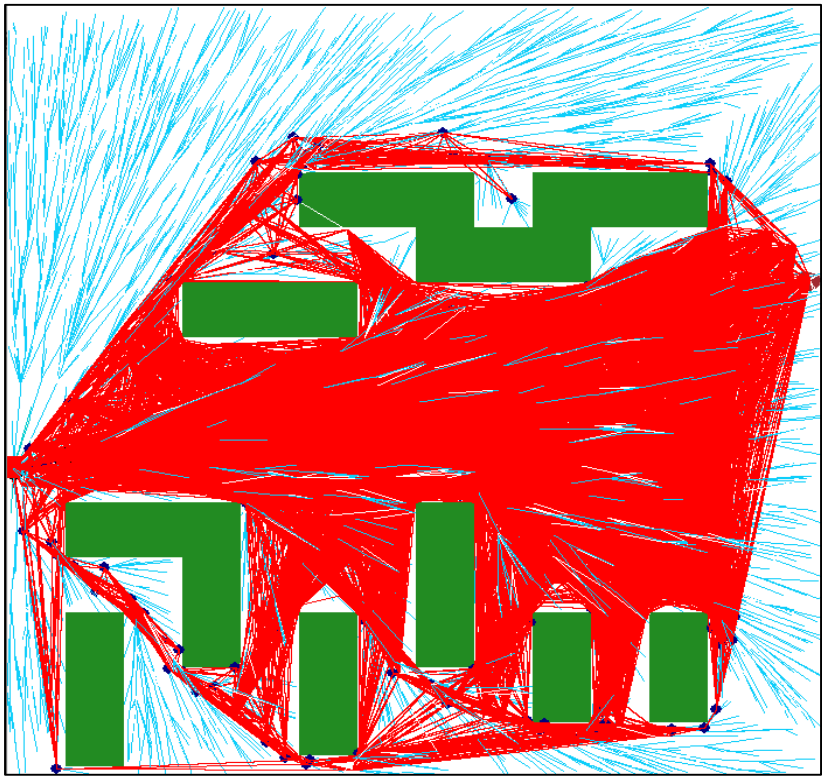


Fig. 4.6: Environment-III; Smart-Graph at 1000 (above) and 5000 RRT* nodes (below)
Number of Vertices: 90 & 356 resp.



As shown in the above figures, the Smart-Graph with a small number of vertices is able to cover the configuration space area which is covered by at least a 5000 node RRT* tree. Coverage here means having enough nodes or vertices in the configuration space to be able to generate an optimized path between any two points in the space. It may be noted that even at this point the RRT* tree is only optimized for providing path from a specific start point to all points in the configuration space and cannot provide paths between all pair of points. The Smart-Graph on the other hand is able to do that, in real time and with computational overload only equivalent to that of a Dijkstra.

In the next phase of the algorithm testing, different goal points were given to the robot during different phases of its movement using mouse in all three environments. As shown in the figures 4.7 – 4.9, the Smart-Graph was able to yield optimized paths between these points.

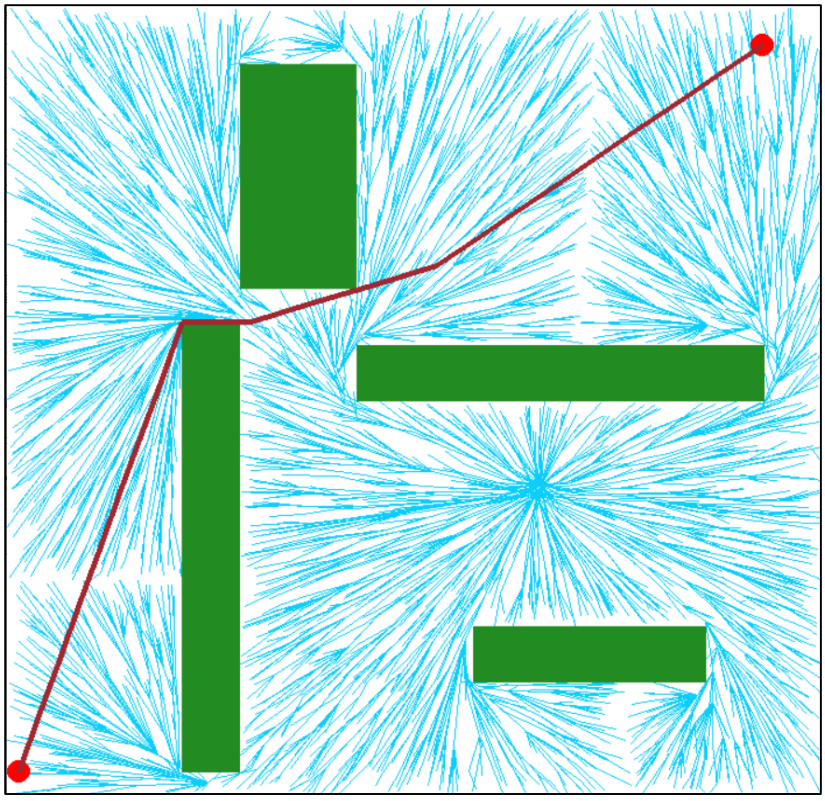
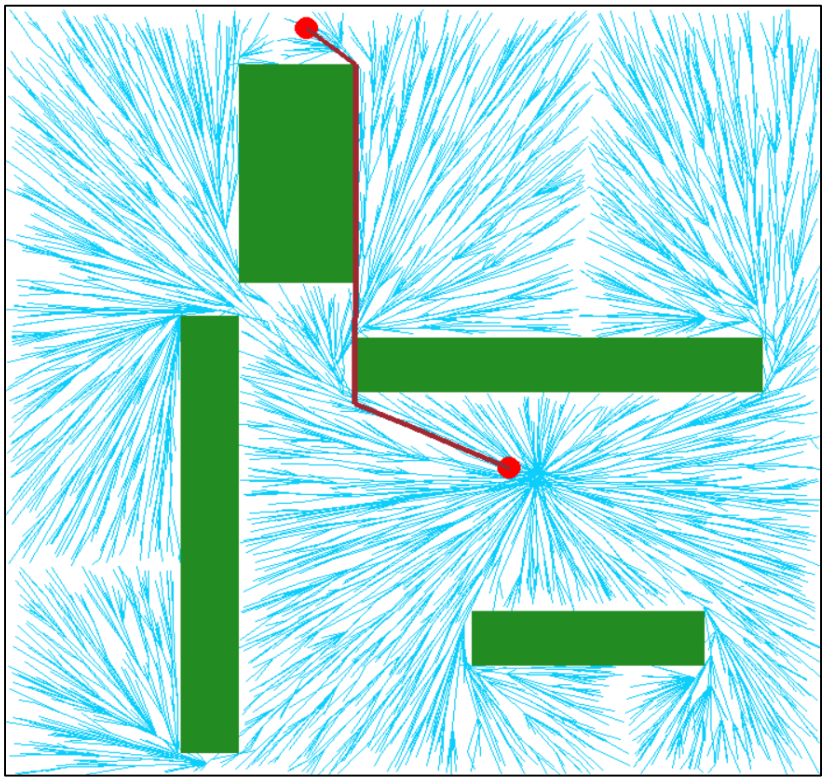


Fig. 4.7: Paths generated by Smart-Graph in Environment-I



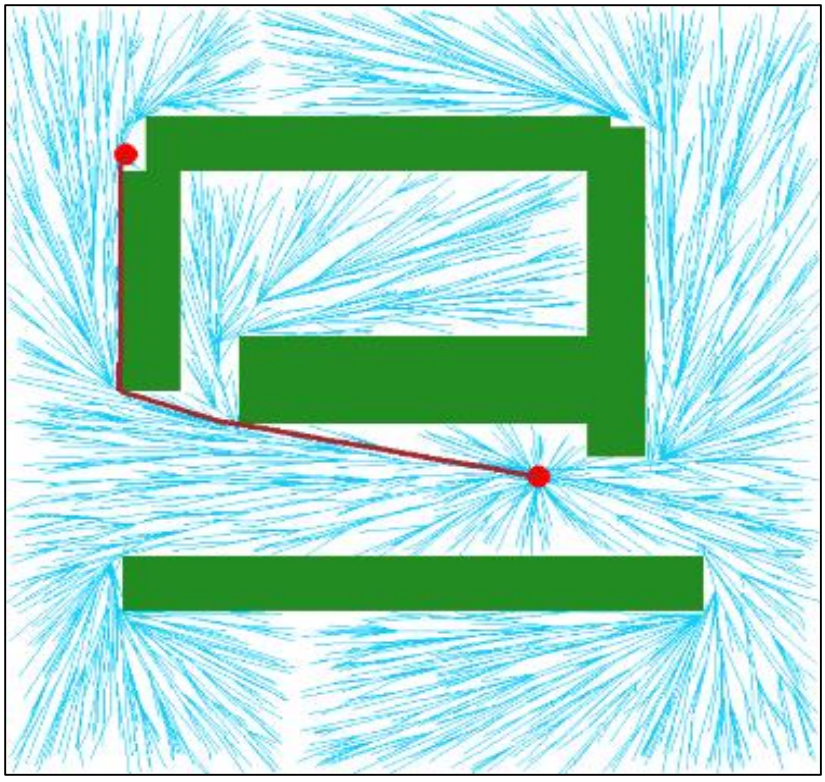
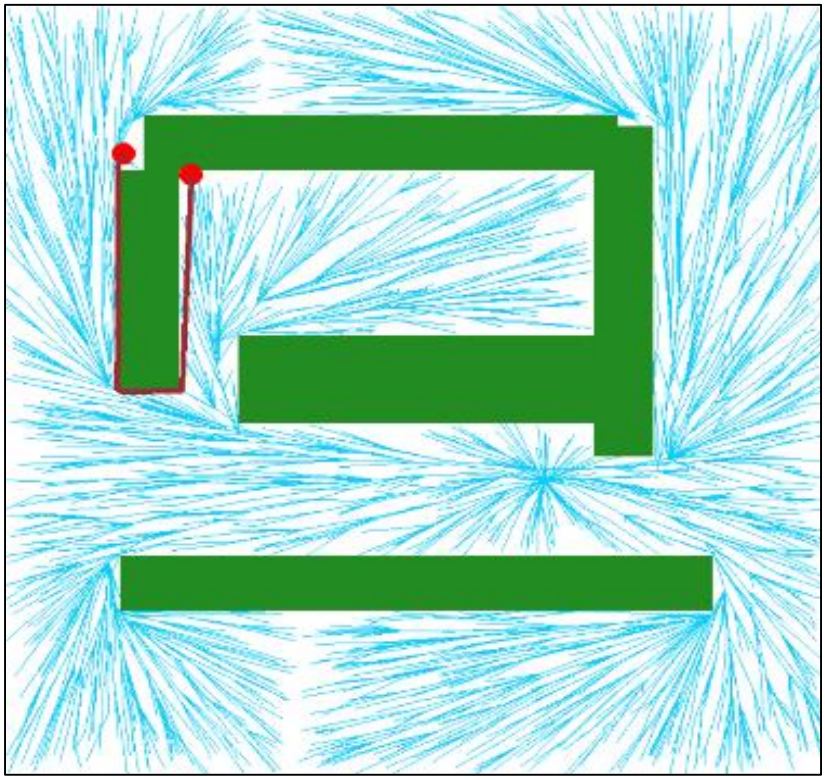


Fig. 4.8: Paths generated by Smart-Graph in Environment-II



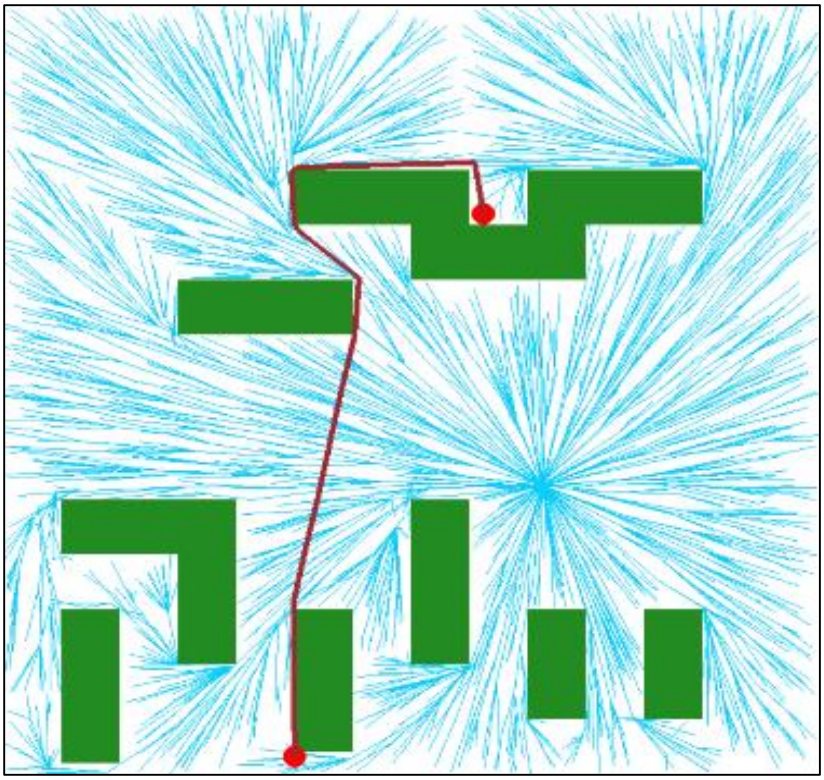
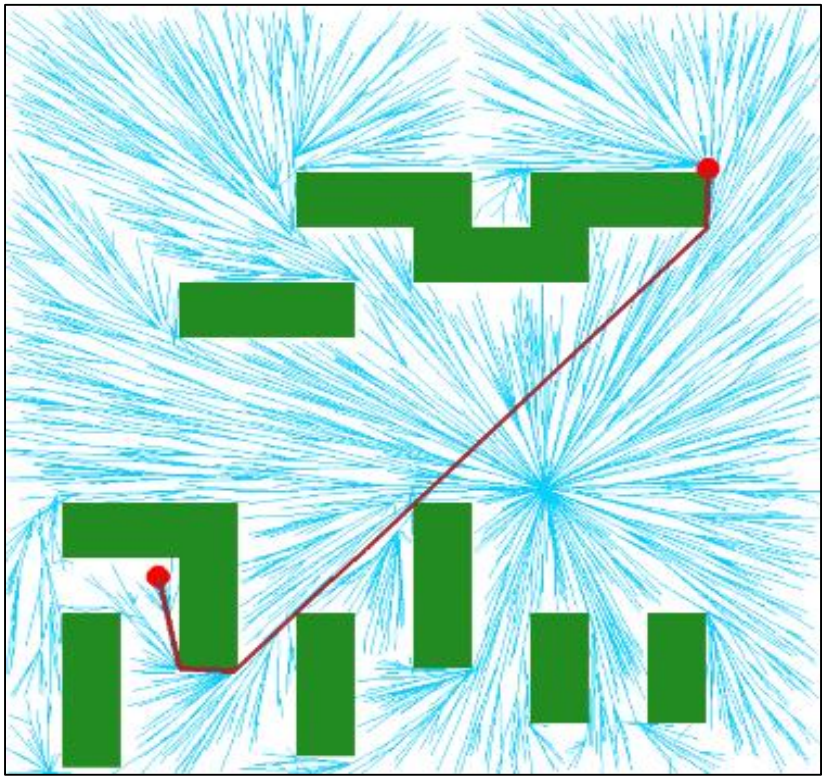


Fig. 4.9: Paths generated by Smart-Graph in Environment-III



*5.5.5 Comparison with RRT**

The algorithm's performance (path optimality) was empirically compared with that of a standard RRT*. To do so, 5 different pairs of start and goal points were selected. Paths for these pairs were computed using a 7000 node RRT* tree, for all three environments. The costs of paths generated by RRT* were logged and are shown in Table-4.6 below.

Pair#	Start	Goal	7000 Node RRT* Path Cost		
			Env – I	Env – II	Env – III
1	10,679	648,33	990	1061	975
2	648,33	514,640	673	646	625
3	514,640	432,419	316	365	243
4	432,419	258,17	512	740	471
5	258,17	29,641	688	714	674

Table 4.6: RRT* Path Costs for Different Environments

In the next step, R4T* algorithm was run for 5000 RRT* nodes initialized with random start and goal points. Its Smart-Graph was used to find paths for the predefined start/goal pairs (shown above) and their path costs were noted. The algorithm was run with 10 random start and goal points for each Environment. The maximum, minimum and average path costs generated during these test runs is shown in the Table 4.7 – 4.9.

Pair#	Start	Goal	Min	Max	Average	RRT*
1	10,679	648,33	988	996	991	990
2	648,33	514,640	668	811	685	673
3	514,640	432,419	307	314	309	316
4	432,419	258,17	503	519	508	512
5	258,17	29,641	683	686	685	688

Table 4.7: Smart-Graph vs RRT* Path Costs in Env-I

Pair#	Start	Goal	Min	Max	Average	RRT*
1	10,679	648,33	1052	1069	1058	1061
2	648,33	514,640	644	647	645	646
3	514,640	432,419	363	380	367	365
4	432,419	258,17	727	752	735	740
5	258,17	29,641	712	714	713	714

Table 4.8: Smart-Graph vs RRT* Path Costs in Env-II

Pair#	Start	Goal	Min	Max	Average	RRT*
1	10,679	648,33	967	1017	979	975
2	648,33	514,640	622	622	622	625
3	514,640	432,419	239	243	240	243
4	432,419	258,17	470	479	473	471
5	258,17	29,641	673	675	674	674

Table 4.9: Smart-Graph vs RRT* Path Costs in Env-III

Tables 4.2 – 4.4 above, show the performance of Smart-Graph when compared to 7000 node RRT*. The results show that the path costs generated by R4T* remained consistently equal to or better than RRT* path costs in most of the cases.

CHAPTER 5: CONCLUSION & FUTURE WORK

The thesis presents a re-planning algorithm which provides the functionality of finding optimized paths between any two points in the workspace using a Smart-Graph built over an RRT* tree. The tree can be built from an already spawned RRT* tree or developed alongside a tree being built. Further it provides the provision of interleaving the movements of the robot with path planning. The robot is able to move as soon as an initial path to the given goal point is found. The algorithm continues path planning along with the movement and keeps on optimizing the path on the go. The process continues until a user-defined value of tree density has been achieved after which the path planning shifts completely to the Smart-Graph. This results in direct provision of optimized paths for the robot along with handling of its movement. The algorithm will be useful in cases where start/goal points change on real-time basis during execution of the algorithm or during movement of the robot.

R4T* was empirically compared with a 7000 nodes RRT* tree. The paths generated by the algorithm in real-time were shown to have similar optimality to the RRT* tree in comparison. The algorithm provides a clear advantage as compared to the existing single and multi-query algorithms. In comparison with existing re-planners it stands apart by providing an optimal solution.

The tree density for the RRT* algorithm is user-defined which can vary along with the environment. Future work can focus on defining a tree density based on some property or characteristic of the environment. Similarly, the algorithms implementation in higher dimensions can also be evaluated.

The algorithm provides optimal solution for start-goal pairs in given environments and needs to be restarted if the environment is changed altogether. Even in that scenario, it can enable movement early on during the algorithm.

Future work can focus on using the algorithm to expedite optimized path finding in new environments as well. This can be done by combining R4T* with algorithms such as Informed RRT*[12], where R4T* can suggest a path using a Smart-Graph built on 1000 or 2000 RRT* nodes. This un-optimized path can be fed to Informed RRT* to quickly find optimal paths in any environment. Hence, enabling the combo to provide real-time optimized paths in scenarios where environments are changing as well.

REFERENCES

[1] LaValle, Steven M. (October 1998). "Rapidly-exploring random trees: A new tool for path planning" Technical Report. Computer Science Department, Iowa State University (TR 98–11).

[2] Khatib, Oussama.; 1986, "Real-time obstacle avoidance for manipulators and mobile robots." Autonomous robot vehicles. Springer, New York, NY, 1986. 396-404.

[3] LaValle, Steven M.; 1998, "Rapidly-exploring random trees: A new tool for path planning." (1998): 98-11.

[4] Karaman, Sertac, and Emilio Frazzoli,; 2011, "Sampling-based algorithms for optimal motion planning." The international journal of robotics research 30.7 (2011): 846-894.

[5] Kavraki, Lydia E., et al.; 1996, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces." IEEE transactions on Robotics and Automation 12.4 (1996): 566-580.

[6] Qureshi, Ahmed Hussain, and Yasar Ayaz.; 2016, "Potential functions based sampling heuristic for optimal path planning." Autonomous Robots 40.6 (2016): 1079-1093.

[7] Islam, F., Nasir, J., Malik, U., Ayaz, Y. and Hasan, O.,; 2012, August. RRT∗-smart: Rapid convergence implementation of rrt∗ towards optimal solution. In 2012 IEEE international conference on mechatronics and automation (pp. 1651-1656). IEEE.

[8] Qureshi, Ahmed Hussain, et al.; 2013, "Potential guided directional-RRT* for accelerated motion planning in cluttered environments." 2013 IEEE International Conference on Mechatronics and Automation. IEEE, 2013.

[9] Qureshi, Ahmed Hussain, and Yasar Ayaz,; 2015, "Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments." Robotics and Autonomous Systems 68 (2015): 1-11.

[10] Tahir, Zaid, et al.; 2018, "Potentially guided bidirectionalized RRT* for fast optimal path planning in cluttered environments." Robotics and Autonomous Systems 108 (2018): 13-27.

[11] Kavraki, Lydia E., et al.; 1996, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces." IEEE transactions on Robotics and Automation 12.4 (1996): 566-580.

[12] Siméon, Thierry, J-P. Laumond, and Carole Nissoux.; 2000, "Visibility-based probabilistic roadmaps for motion planning." Advanced Robotics 14.6 (2000): 477-493.

[13] Naderi, Kourosh, Joose Rajamäki, and Perttu Hämäläinen.; 2015, "RT-RRT* a real-time path planning algorithm based on RRT." Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games. 2015.

[14] Chandler, Bryant, and Michael A. Goodrich.; 2017, "Online RRT∗ and online FMT∗: Rapid replanning with dynamic cost." 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2017.

[15] J.D. Gammell, S.S. Srinivasa, T.D. Barfoot.; 2014, Informed RRT*: Optimal samplingbased path planning focused via direct sampling of an admissible ellipsoidal heuristic, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, 2014, pp. 2997–3004.

Usama Tariq Thesis

ORIGINALITY REPORT

9%	6%	8%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	e-space.mmu.ac.uk Internet Source	1%
2	arxiv.org Internet Source	1%
3	Ahmed Hussain Qureshi, Saba Mumtaz, Yasar Ayaz, Osman Hasan, Mannan Saeed Muhammad, Muhammad Tariq Mahmood. "Triangular Geometrized Sampling Heuristics for Fast Optimal Motion Planning", International Journal of Advanced Robotic Systems, 2015 Publication	1%
4	link.springer.com Internet Source	1%
5	hdl.handle.net Internet Source	1%
6	Submitted to Higher Education Commission Pakistan Student Paper	1%

7	Ahmed Hussain Qureshi, Yasar Ayaz. "Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments", Robotics and Autonomous Systems, 2015 Publication	1 %
8	Cheng Zhang, Amin Hammad. "Improving lifting motion planning and re-planning of cranes with consideration for safety and efficiency", Advanced Engineering Informatics, 2012 Publication	<1 %
9	"Algorithmic Foundations of Robotics X", Springer Science and Business Media LLC, 2013 Publication	<1 %
10	"Algorithmic Foundations of Robotics XI", Springer Science and Business Media LLC, 2015 Publication	<1 %
11	export.arxiv.org Internet Source	<1 %
12	Ahmed Hussain Qureshi, Yasar Ayaz. "Potential functions based sampling heuristic for optimal path planning", Autonomous Robots, 2015 Publication	<1 %

13	Luiz GDO Véras, Felipe LL Medeiros, Lamartine NF Guimarães. "Rapidly exploring Random Tree* with a sampling method based on Sukharev grids and convex vertices of safety hulls of obstacles", International Journal of Advanced Robotic Systems, 2019 Publication	<1 %
14	journals.sagepub.com Internet Source	<1 %
15	tel.archives-ouvertes.fr Internet Source	<1 %
16	"Robot Intelligence Technology and Applications 3", Springer Science and Business Media LLC, 2015 Publication	<1 %
17	inba.info Internet Source	<1 %
18	www.hindawi.com Internet Source	<1 %
19	Luiz Gustavo D. O. Veras, Felipe L. L. Medeiros, Lamartine N. F. Guimaraes. "Systematic Literature Review of Sampling Process in Rapidly-Exploring Random Trees", IEEE Access, 2019 Publication	<1 %

20	Xiaojing Wu, Lei Xu, Ran Zhen, Xueli Wu. " Biased Sampling Potentially Guided Intelligent Bidirectional RRT Algorithm for UAV Path Planning in 3D Environment ", Mathematical Problems in Engineering, 2019 Publication	<1 %
21	Submitted to Universiti Kebangsaan Malaysia Student Paper	<1 %
22	www.spiedigitallibrary.org Internet Source	<1 %
23	Ray Jarvis. "Robot path planning", Proceedings of the 2006 international symposium on Practical cognitive agents and robots - PCAR 06 PCAR 06, 2006 Publication	<1 %
24	Submitted to VIT University Student Paper	<1 %
25	Weria Khaksar, Tang Sai Hong, Mansoor Khaksar, Omid Motlagh. "A Low Dispersion Probabilistic Roadmaps (LD-PRM) Algorithm for Fast and Efficient Sampling-Based Motion Planning", International Journal of Advanced Robotic Systems, 2013 Publication	<1 %
26	www.ideals.illinois.edu Internet Source	<1 %

27	Bin Liao, Fangyi Wan, Yi Hua, Ruirui Ma, Shenrui Zhu, Xinlin Qing. "F-RRT*: An improved path planning algorithm with improved initial solution and convergence rate", Expert Systems with Applications, 2021 Publication	<1 %
28	Ignacio Pérez-Hurtado, Miguel Á. Martínez-del-Amor, Gexiang Zhang, Ferrante Neri, Mario J. Pérez-Jiménez. "A membrane parallel rapidly-exploring random tree algorithm for robotic motion planning", Integrated Computer-Aided Engineering, 2020 Publication	<1 %
29	Islam, Fahad, Jauwairia Nasir, Usman Malik, Yasar Ayaz, and Osman Hasan. "RRT□-Smart: Rapid convergence implementation of RRT□ towards optimal solution", 2012 IEEE International Conference on Mechatronics and Automation, 2012. Publication	<1 %
30	J. Borenstein, Y. Koren. "Teleautonomous guidance for mobile robots", IEEE Transactions on Systems, Man, and Cybernetics, 1990 Publication	<1 %
31	Jiankun Wang, Baopu Li, Max Q.-H. Meng. "Kinematic Constrained Bi-directional RRT	<1 %

with Efficient Branch Pruning for robot path planning", Expert Systems with Applications, 2021
Publication

32	Lecture Notes in Computer Science, 2006. Publication	<1 %
33	Mike Vande Weghe, Dave Ferguson, Siddhartha S. Srinivasa. "Randomized path planning for redundant manipulators without inverse kinematics", 2007 7th IEEE-RAS International Conference on Humanoid Robots, 2007 Publication	<1 %
34	S.M. LaValle. "RRT-connect: An efficient approach to single-query path planning", Proceedings 2000 ICRA Millennium Conference IEEE International Conference on Robotics and Automation Symposia Proceedings (Cat No 00CH37065) ROBOT-00, 2000 Publication	<1 %
35	repositorio-aberto.up.pt Internet Source	<1 %
36	"Experimental Robotics", Springer Science and Business Media LLC, 2021 Publication	<1 %

37	Jauwairia Nasir, Fahad Islam, Usman Malik, Yasar Ayaz, Osman Hasan, Mushtaq Khan, Mannan Saeed Muhammad. "RRT*-SMART: A Rapid Convergence Implementation of RRT*", International Journal of Advanced Robotic Systems, 2013 Publication	<1 %

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		