COMPARISON OF PATH PLANNING ALGORITHMS

by

Fuat GELERİ

BS. In CSE, Marmara University, 2004

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in System and Control Engineering

Boğaziçi University

2007

COMPARISON OF PATH PLANNING ALGORITHMS

APPROVED BY:

Prof. H. Levent Akın (Thesis Supervisor)	.....
Prof. Işıl Bozma	.....
Assoc. Prof. Haluk Topçuoğlu	.....

DATE OF APPROVAL: 20.09.2007

ACKNOWLEDGEMENTS

First, I want to thank my advisor Prof. Levent AKIN for his, and Prof. Işıl BOZMA for her guidance and support in development, completion of this thesis.

Secondly, I thank all my friends for their understanding. Special thanks go to my friends Reyhan AYDOĞAN, Murat BALABAN, Gülefşan BOZKURT, Yağmur GÖK, Barış GÖKÇE, Kemal KAPLAN, Çetin MERİÇLİ, Sarp Baran ÖZKAN, Nuri TAŞDEMİR and Ülker ÖZGEN for their great support and understanding.

Finally, it is my pleasure to thank my family, Ayşe GELERİ and Aydın GELERİ for their endless support and belief in me and my work.

This thesis was supported by Boğaziçi University Research Fund (BAP) and Turkish Republic Prime Ministry State Planning Organization (DPT).

COMPARISON OF PATH PLANNING ALGORITHMS

Path planning problems arise in many different fields such as; robotics, assembly analysis, virtual prototyping, pharmaceutical drug design, manufacturing, and computer animation. Path planning algorithms aim to solve problems that involve computing a continuous sequence, a path, of configurations between an initial and goal configuration. Planning of a path involves some constraints, such as computing a collision-free path.

We compared various path planning and navigation algorithms. As reactive algorithm, an improved version of Artificial Potential Field (APF) algorithm is used. In robot coordination this algorithm is the superior algorithm. It coordinates 250 robots easily. Whereas deliberative algorithms, such as Rapidly-exploring Random Tree Connect (RRT Connect) algorithm, can only coordinate 40 robots with high costs. The other deliberative algorithms, Rapidly-exploring Random Tree (RRT), Probabilistic Roadmap (PRM) and Lazy Probabilistic Roadmap (Lazy PRM), could not coordinate more than 20 robots within feasible resource and time limits in our tests. In robot coordination reactive algorithms are more successful, but, when the environment contains local minima, using a deliberative algorithm is inevitable.

In path planning for multiple robots, decentralized approaches, or partially grouping of the robots show better performances. As the number of the controlled robots in the environment increases, using decentralized approaches becomes a requirement, because the amount of the required time and the resources increases exponentially in centralized approaches, but linearly in decentralized approaches. Partially grouping of the robots gives the best performance results, because the resource requirements increase nearly linear, and nearby robots are controlled in centralized manner.

GÜZERGAH PLANLAMA ALGORİTMALARININ KARŞILAŞTIRILMASI

Güzergah planlama problemleri bir çok alanda karşımıza çıkmaktadır. Örneğin, robotik, montaj analizi, sanal prototip üretimi, ilaç tasarımı, üretim, ve bilgisayar animasyonları bu alanlardan bazılarıdır. Güzergah planlama algoritmaları, başlangıc konfigürasyondan amaç konfigürasyona sürekliliği olan bir sıra hesaplasını sağlamaktadır. Bir güzergahın planlaması çeşitli sınırlamaları içermektedir, örneğin bulunan yol sayesinde robot hiç bir engele çarpmamalıdır.

Tepkisel algoritma olarak kullanılan APF algorithmasının geliştirilmiş modeli robot koordinasyonunda en başarılı algoritmadır. Bu algoritma 250 robotun koordinasyonunu kolaylıkla sağlarken, RRT Connect algoritması, sadece 40 robota kadar büyük masraflarla eşgüdüm yapabilmektedir. Diğer düşünen algorithmalar RRT, PRM ve Lazy PRM algoritması ise sadece 20 robota kadar koordinasyon yapabilmektedir. Robot koordinasyonunda tepkisel algorithmalar daha basarılı olurken, eğer ortam bölgesel minimumlar içeriyorsa düsünen algoritmaların kullanılması kaçınılmazdır.

Özellikle dinamik ortamlarda miskin algoritmaların kullanılması kullanılan kaynak ve geçen zamanı azaltmaktadır. Çoklu robotlar için güzergah planlarken merkezi olmayan yaklaşımlar veya kısmi gruplamalar yapmak daha büyük başarımlar göstermektedir. Merkezi yaklaşımlarda ihtiyaç duyulan zaman ve kaynak üssel artarken, merkezi olmayan yaklaşımlarda doğrusal arttığı için, ortamdaki idare edilen robot sayısı arttığı zaman merkezi olmayan yaklaşımları kullanmak bir gereksinim haline gelmektedir. Robotları kısmi kümelemek, ihtiyaç duyulan kaynaklar yaklaşık doğrusal arttığı ve yakın robotlar merkezi anlamda idare edildiği için en iyi sonuçları vermektedirler.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

3D	Three dimensional
C	Configuration
c_{init}	Initial configuration of a robot
c_{goal}	Goal configuration of a robot
C_{free}	Collision free configuration space
$C_{obstacle}$	Configuration space in collision
C_{space}	Configuration space
d(.)	Distance function
d*	Distance threshold
E	Edges
T	Path tree
U	Potential function
U_{att}	Attractive potential function
U_{rep}	Repulsive potential function
ΔU	Change in the potential function
V	Vertexes
$\alpha(i)$	Step size at i'th iteration
ϵ	Epsilon
APF	Artificial Potential Function
DOF	Degree of Freedom
EST	Expansive-Spaces Trees
GVG	Generalized Voronoi Graphs
NRL	Normalized robot path length
PRM	Probabilistic Roadmap
RBOT	Used interchangeably with APF
RRT	Rapidly-Exploring Random Tree

1. INTRODUCTION

In the development of autonomous robots, devising a way to give robots the capability of making their own plans in various situations is a complex problem. Motion planning is a sub-problem and it refers to the computation of moving from one place to another in the presence of obstacles, either static or dynamic.

1.1. Problem Statement

Path planning plays an essential role in most of the robotic applications. An advanced path planning module can provide robots more mobility and autonomy. However, if the environment, to run the robots on, contains multiple robots, moving obstacles, static obstacles, and also constraints on the motion of the robots, the difficulty increases toward NP completeness.

The knowledge about the behaviour of the algorithms in various environments may play an essential role in the design of successful mobile robot applications. There are variety of path planning and multiple robot coordination algorithms. However, which algorithm is the best for a problem depends on the characteristics of the problem. An algorithm may be the most appropriate for an environment containing only static obstacles, yet another algorithm may be better if the environment contains dynamic obstacles. Change of the efficiency of the algorithm to the increase of the number of the coordinated robots shows the scaleability of the algorithm. The knowledge of scaleability, time complexity, space complexity, and effectiveness of the algorithms may lead us to develop a robust path planning module, that can tackle with various problems.

1.2. Contribution of the Thesis

According to the task to accomplish, properties of the environment, and the robots, various path planning algorithms are designed. In this thesis, we accomplish

the following issues:

- Comparison of path planning algorithms, like PRM [1], Lazy PRM [2], RRT [3], RRT Connect [4], and APF [5] for single, multiple robot cases, in environments containing no obstacle, only static obstacles, and static and dynamic obstacles cases,
- Improvement of RRT Connect and Lazy PRM algorithm by mixing those algorithms,
- Improvement of APF algorithm, so it handles coordination of more than 250 robots, and takes robot properties into coordination.

1.3. Thesis Outline

In the second chapter, a detailed description of the path planning algorithms is provided. First, most basic algorithms are given to describe the problem, and basic concepts. Then computational approaches, and reactive algorithms are described. At the end of the introduction chapter, centralized and decoupled planning for multi-robot coordination is described.

Then, sampling based algorithms, sampling and connection strategies are described. Multiple query algorithms, like PRM and Lazy PRM, single query algorithms like RRT and RRT Connect. Later, we discussed post processing in path planning, and how multi-robot coordination is achieved.

At the third chapter we inspected, and described our improvements on the algorithms. Lazy PRM RRT Connect algorithm is described in this chapter. Improvements on the APF algorithm, and more detail about how multi-robot coordination is achieved is given in this chapter.

At the last chapter, we placed the applied tests and the performance results of the algorithms in these tests. We compared algorithms in no obstacle environments with different number of robots with far, and near target configuration to the other

robots' target configurations. Later we placed the tests in environments with static and dynamic obstacles. We also tested centralized and decentralized approaches in environments with different number of dynamic obstacles.

2. GENERAL BACKGROUND

The degree of the difficulty of motion planning changes depending on the environmental factors. If the environment contains dynamic obstacles, obstacles with information less than required, namely partially known environment, then its degree of difficulty increases.

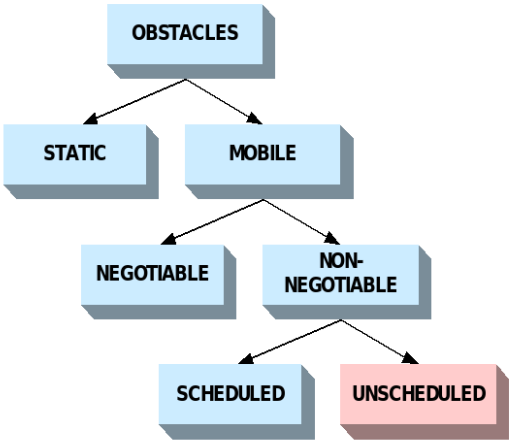

Figure 2.1. Taxonomy of obstacle types

The types of the obstacles are shown in the Figure 2.1. The different possible scenarios are shown in the following Table 2.1.

Table 2.1. Environment Classification

	Static Obstacles	Dynamic Obstacles
Completely Known	Case I	Case II
Partially Known	Case III	Case IV

Case I is the simplest scenario where all obstacles are fixed and well known before a path planning algorithm is used. In this case, the problem is the basic motion planning problem, and it is usually solved in the following two steps:

- Define a graph representing the geometric structure of the environment.
- Perform a graph search to find a connected component between the node containing the start point and the node containing the destination point.

The geometric structure of the graph differs depending on which approach is used to solve the problem. The three most common approaches are

- the roadmap approach [1],
- the cell decomposition approach [6], and
- the potential field approach [7].

These approaches are powerful but they can not work in high dimensional spaces well. To solve this problem sampling based methods are proposed. These algorithms generate a graph representing the free space of the environment by generating random samples, instead of generating a graph representing the geometric structure of the environment, which is difficult, and time consuming. The part of querying for a path is the same for both the sampling based algorithms and roadmap algorithms.

If the environment contains multiple robots, moving obstacles, static obstacles, and also constraints on the motion of the robots, the difficulty increases toward NP completeness.

We may categorize the complexity of path planning as:

1. In $3D$ work space finding exact solution is NP-HARD. [8]
2. Path planning is PSPACE-HARD. [9]
3. The complexity increases exponentially with:
    - Number of DOF [10]
    - Number of agents.

2.1. Open-Loop – Geometric Representation Algorithms

A path is a sequence of robot configurations from a starting configuration to an end configuration. It must be continuous, and in a specific order. Usually a collision-free path with minimum cost is preferred, and as the cost we may use distance, time, battery consumption etc. Path planning is therefore an optimization and search problem.

For path planning, algorithms usually do not use the work space, but instead uses the configuration space. Work space is the n-dimensional space in which the robot moves. The robot, obstacles, and other objects are the closed subsets of the work space. The configuration represents the state of the robot with respect to its environment, and usually it is represented by a data structure that is given as a vector, or a matrix of position and orientation parameters.

Configuration space, also called $CSpace$, is the set of all possible configurations of a robot, [1]. The path planner searches the appropriate solutions in this space. In the configuration space there may be infinite number of configurations. The dimension of the configuration space may be different than the work space it represents. The dimension of a configuration space is the minimum number of parameters needed to completely specify the configuration of the object. For the dimension in the work space we use degree of freedom (DOF), which means set of independent position variables, necessary to specify an object's position in the work space, with respect to a frame of reference.

As seen in Figure 2.2, in the configuration space each point corresponds to a configuration rather than a real point in space. In this figure we see a circular robot, so its configuration space is $2D$. If the heading of the robot mattered then a configuration would consist of a position and an orientation, so the configuration space would be $3D$.

The configuration space contains configurations that lead the robot to be in collision, and configurations in which the robot is not in collision. The free space, $Cfree$

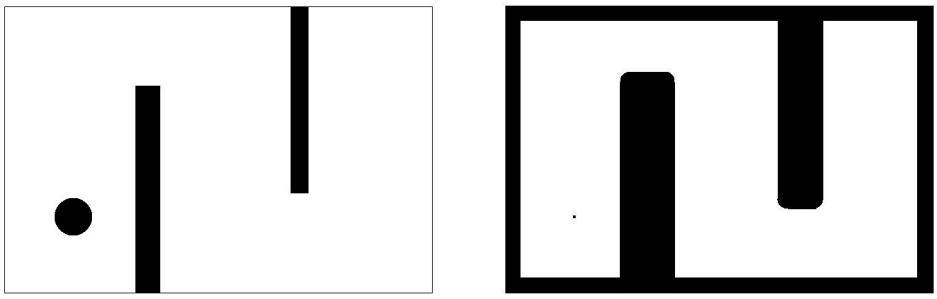

Figure 2.2. Work space and configuration space for a circular robot

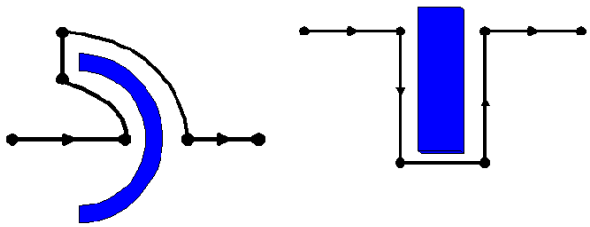

Figure 2.3. Bug algorithms use two basic behaviors: move on straight line and follow a boundary

is the set of configurations, at which the robot is not in collision, and obstacle space, $Cobstacle$ is the set of configurations, at which robot is in collision. The sum of $Cfree$ and $Cobstacle$ gives the total $Cspace$, Equation 2.1. For a path to be collision free, all of the paths should be in Cfree.

$$C_{space} = C_{free} + C_{obstacle} \tag{2.1}$$

2.1.1. Simple Algorithms

Inspecting the simple algorithms may help us to understand interesting and difficult issues of path planning. These simple algorithms are straightforward to implement and analysis shows that when possible their success is guaranteed [1].

The $Bug1$ [1], $Bug2$ [1], $Tangent\ Bug$ [1], and $Wavefront\ Method$ [1] are some of the simple algorithms. The Bug algorithms assume the robot as a point operating

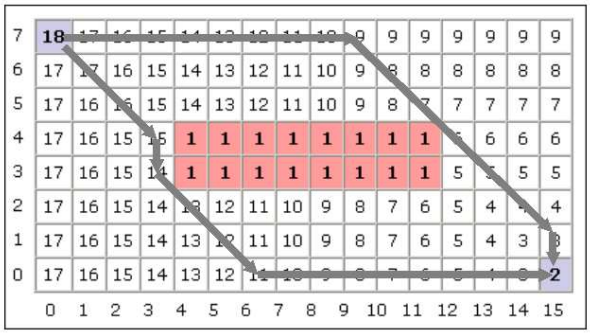

Figure 2.4. Wavefront algorithm is another simple path planning algorithm

in the plane with a contact sensor. The robots detect obstacles with this zero range sensor. When the robot has a finite range sensor, like an infrared sensor, the algorithm used is called Tangent Bug algorithm. These algorithms use two basic behaviors: move on a straight line and follow a boundary, as seen in Figure 2.3. The bug algorithms are not complete, they are local, suboptimal, and dynamic.

In the *Wavefront Algorithm*, we divide the environment into a set of cells. Then, starting with the initial cell, where the robot is located, we assign numbers. Obstacles are numbered as *one*, as a special number to get the robot away from them. Written numbers are started from *two* at the initial cell, and incremented toward the goal at the adjacent cells. We may use $eight-point$ or $four-point$ connectivity, as left, right, up, and down. When all the squares have a number assigned, we go from the cell that contains the goal configuration toward the initial cell as moving toward the adjacent cell with a lower number.

These algorithms are ideal for basic robots with limited capabilities. They require little processing power, and memory. However, for more difficult problems, and for multi-robot path planning problems, improved algorithms, described on the next sections of this document, are required.

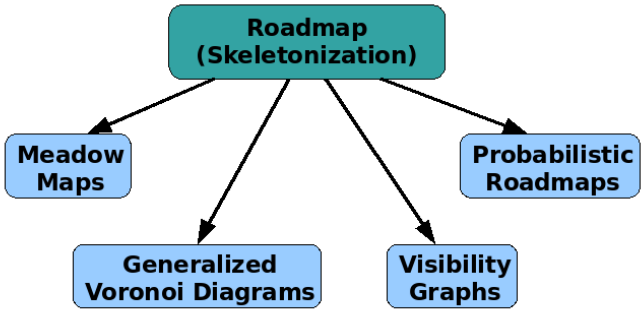

Figure 2.5. Taxonomy of roadmap algorithms

2.1.2. Roadmaps (Skeletonization)

Roadmap algorithms and cell decomposition algorithms are Cspace representation algorithms. They transforms the work space to *Cspace* representations.Meadow maps, visibility graphs, generalized voronoi graphs (GVG) and probabilistic roadmaps are the well known roadmap algorithms [1]. Taxonomy of roadmap algorithms can be seen in the Figure 2.5.

2.1.2.1. Meadow Maps. In meadow maps, first optionally we grow the obstacles as big as the robot. We find the corners of the objects in the work space. Then we connect these corners with edges, as shown in Fig 2.6. For a path planning query, the center point of these edges will be used as milestone points. The start point of the query will be connected to the nearest center point, and the end point will also be connected to its nearest center point. Then a graph algorithm will be used to find the sequence of center points between these two center points.

Meadow maps are not able to generate unique polygons. It is not quite possible to create this type of map with sensor data, and it is difficult for the robot to differentiate, and recognize the right corners, edges, and go to the middle.

2.1.2.2. Visibility Graphs. Another simple procedure of transforming the world space to *Cspace* is Visibility Graphs [11]. In this method, every pair of important points,

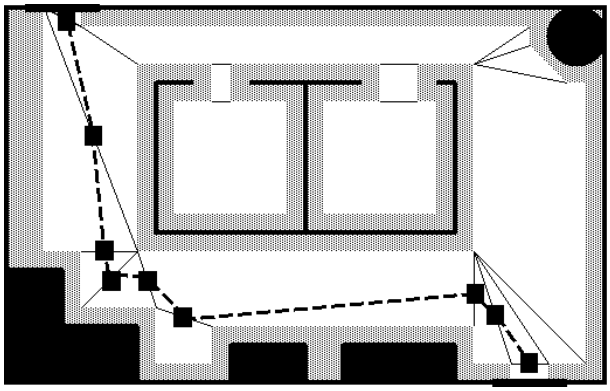

Figure 2.6. Centers of edges connecting edges composes the Meadow Maps

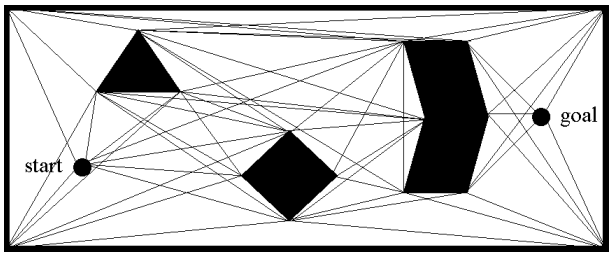

Figure 2.7. Visibility graph is composed of connection of each important point

vertexes of obstacles, initial and final points etc. are connected, as shown in Figure 2.7. These connected edges should not be in collision with any object. The graph composed of this edges and nodes is the *Cspace* representation of the work space. Path searches will be done on this graph.

2.1.2.3. Generalized Voronoi Graphs. The points in the work space, having the same distance to the surrounding obstacles make up the lines of the voronoi graph [12]. The intersections of these lines are the nodes of the relational graph. So the work space is transformed into *Cspace*, as shown in Figure 2.8.

Finding the points equidistance to the nearby obstacles is a quite difficult task, and has a quite high computational cost. Moreover, Voronoi Graph is sensitive to sensor noise, and for path planning, the robot should be able to sense the boundaries of the obstacles, and workspace.

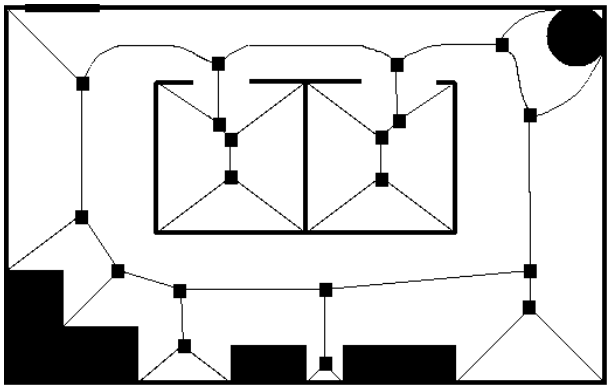

Figure 2.8. Voronoi diagram, equidistance to each obstacle

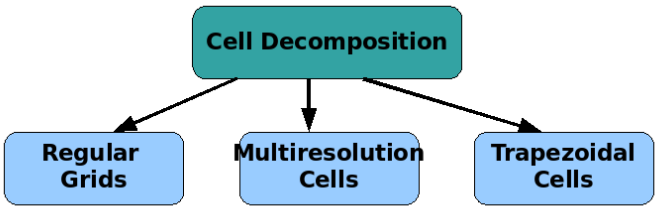

Figure 2.9. Taxonomy of cell decomposition algorithms

2.1.3. Cell Decompositions

Trapezoidal cell decomposition, regular grids and quadtrees are the major types of cell decomposition algorithms. Taxonomy of cell decomposition algorithms can be seen in the Figure 2.9.

2.1.3.1. Trapezoidal Cell Decomposition. The world is converted to a set of union of trapezoid shaped cells. A line is started from the left toward the right, when it touches a new object, or touching an object ends, from these points line is divided, and this divisions are marked as the edges of the trapezoid shapes, as shown in Figure 2.10. $Cfree$ is the union of these trapezoids.

2.1.3.2. Regular Grids. We can see a regular grid as the same environment with enlarged pixels [1]. Each element is an enlarged pixel, as shown in Figure 2.11. A relational graph is generated from these nodes, by connecting each node with its neigh-

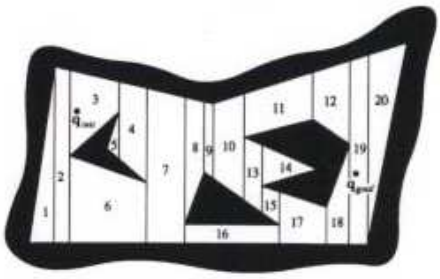

Figure 2.10. Union of trapezoidal cells constructs the $Cfree$

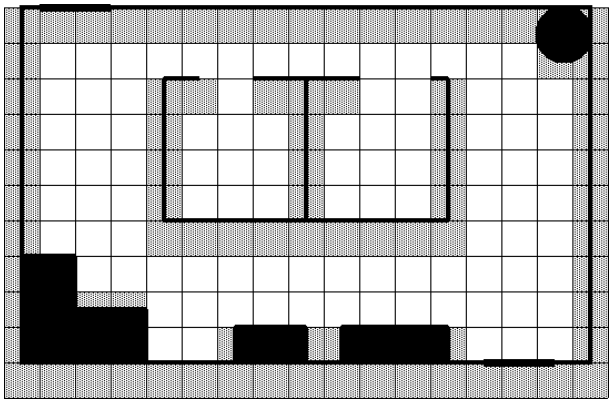

Figure 2.11. Regular grids are like representing n^2 pixel with one pixel

boring nodes.

Because the world does not always line up on grids, applying grid algorithms does not always work well. Moreover, if a cell contains both the free space and the collision space, this cell is seen as in collision space, and this is the digitalization bias the grid algorithms bring. This algorithm is complete, if a path exists it will find it.

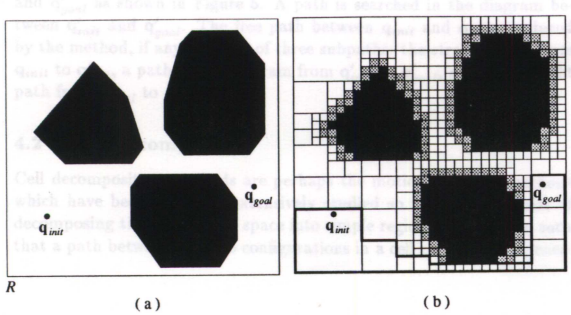

Figure 2.12. Subdivide cells as much as needed

2.1.3.3. Approximate Cell Decomposition (Quadtrees). In exact cell decomposition, some cells may contain both collision space and collision free space, and these cells are counted as in collision space. This method solves the problem by dividing the grid recursively until the cell lies entirely in free space or in *Cobstacle* region, or an arbitrary limit resolution is reached. Because a cell is divided into four smaller cells of the same shape each time it gets decomposed, the method is also called a “quadtree" decomposition, shown in Figure 2.12. Like in the regular grids method, the free path can be easily found by following the adjacent, decomposed cells through free space [1].

2.2. Closed-Loop – Reactive Approaches

The difficulty of explicitly representing the configuration space forced scientists to search new ways for path planning. Incrementally searching the free space while searching a path is emerged as an alternative to the algorithms that use configuration space representation. Bug algorithm is a simple example for incremental search algorithms. However, the bug algorithm works only on $2D$. So, other navigation planners are developed to work for a richer class of robots and produce a greater variety of paths than Bug algorithm.

2.2.1. Potential Field

Potential functions are used in the incremental search path planners. A potential function is a differentiable real-valued function. It can be seen as an energy formula, and its gradient is the force to apply to the robot to navigate. The gradient is used to define a vector field which is directed toward the goal, and enables robot to escape from the obstacles as shown in Figure 2.13.

We may see the whole process as the actions of positively and negatively charged particles' movements. Say, the robots are the particles with positive charge, and the goal is charged negatively. So, the controlled robot will be attracted by the goal, and will keep itself away from other robots, too. When such a gradient vector space is generated, the robots will move from a "high-value" state to a "low-value" state, which

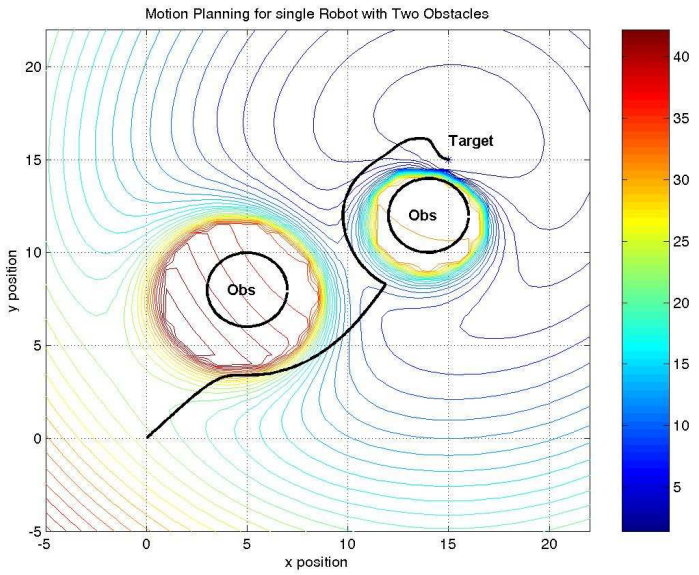

Figure 2.13. Robot finds its path with potential function [13]

will enable them to follow a path "downhill". Such a path is called *gradient descent* path.

Unfortunately, all of the potential functional approaches suffer from the existence of local minima which does not correspond to the goal. This means, the potential function may lead the robot to a point, which is not the goal. Therefore, many potential functions do not lead to complete path planners.

$$\begin{aligned} U(q) &= U_{att}(q) + U_{rep}(q) \\ \nabla U(q) &= \nabla U_{att}(q) + \nabla U_{rep}(q) \end{aligned} \tag{2.2}$$

The simplest approach for potential function is using the attractive and repulsive potentials. Sum of the attractive potential and repulsive potential gives the power of the potential on the robot, shown in Equation (2.2). U_{att} is the attraction function, and U_{rep} is the repulsition function. The sum of derivative of these functions constitute the potential force.

$$\nabla U_{att}(q) = \zeta(q - c_{goal}) \tag{2.3}$$

When the robot gets nearer to the goal, the attraction force of the goal gets smaller and smaller, shown in Equation 2.3. So the robot will be under the control of only repulsive forces, which places a gap between the robot and its goal when the equilibrium is reached. To do not let such a situation, the attractive force formula is changed after the distance between robot and the target reaches a threshold d^*_{goal}. So the $U_{att}(q)$ is updated as

$$\nabla U_{att}(q) = \begin{cases} \zeta(q - c_{goal}), & d(c, c_{goal}) \leq d^*_{goal}, \\ \frac{d^*_{goal} - \zeta(q - c_{goal})}{d(c, c_{goal})}, & d(c, c_{goal}) > d^*_{goal}, \end{cases} \tag{2.4}$$

A repulsive potential keeps the robot away from an obstacle. As the robots get nearer to the obstacles, the power of the repulsive energy should be higher and higher 2.4. Each obstacle will apply a repulsive force on the robot, if the robot is near enough to the obstacles. The threshold of nearness is represented by C^*_i, and the gradient formula for repulsive force of each obstacle over the robot is

$$\nabla U_{rep_i}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{d_i(q)} - \frac{1}{C^*_i})^2, & if\ d_i(q) \leq C^*_i \\ 0 & if\ d_i(q) > C^*_i \end{cases} \tag{2.5}$$

When all of the repulsive forces are summed up, a total repulsive force will be achieved. The navigation function will be the sum of this repulsive forces with the attractive force. The resultant gradient descent algorithms' effect can be seen in Figure 2.14. .

In Figure 2.15, the scalar $\alpha(i)$ determines the step size at the i'th iteration. The value of $\alpha(i)$ should be small enough to avoid collisions with any obstacle, and it should be big enough to not require excessive computation time. When the gradient function gets less than an ϵ value the algorithm will terminate.

2.2.1.1. The Local Minima Problem. Using potential functions in robot navigation gives a powerful mechanism, that successfully works on partially known environments, with moving obstacles. However, as the general problem of gradient descent algorithms, potential functions suffer from local minima problem.

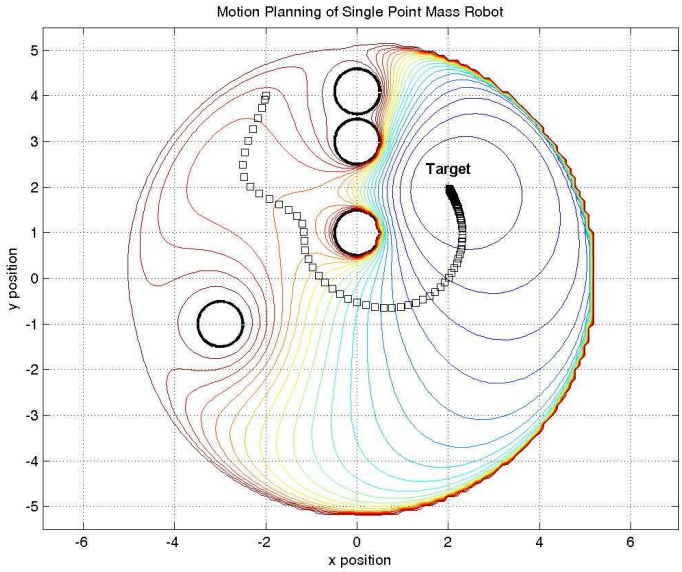

Figure 2.14. Navigation of the robot with potential function [13]

As seen in Figure 2.16, the robot is attracted by its goal, and repelled by the obstacle, and reaches a local minima at the center of the obstacle surrounding it.

2.3. Multiple Robot Coordination

Multiple robot coordination problem deals with path planning for more than one robot. A collision free path for multiple robots means at every step there is no collision between a robot and an obstacle or between a robot and another robot. The solution of this problem should not only find paths for the individual robots, but must also coordinate robots when following these paths so that no robot will be in collision. The coordination of the robots makes the problem significantly harder than the case of a single robot path planning. Multiple robot path planning algorithms can be divided into two groups as centralized and decoupled planning [14, 1, 15].

2.3.1. Centralized Planning

In centralized multiple robot path planning, the robots are thought as a single body, and the robot configurations are added up to generate a single high dimensional

Algorithm 2.1 Gradient Descent

Input:

A means to compute the gradient $\nabla U(q)$ at a point q

Output:

A sequence of points $q(0), q(1), \ldots, q(i)$

1: $c(0) \leftarrow c_{start}$
2: $i \leftarrow 0$
3: **while** $\nabla U(q(i)) \neq 0$ **do**
4: $\quad c(i+1) \leftarrow q(i) + \alpha(i)\nabla U(q(i))$
5: $\quad i \leftarrow i + 1$
6: **end while**

Figure 2.15. Gradient descent algorithm

configuration. The dimensionality of this new configuration is equal to the total number of degrees of freedom of all the robots. Coordination in centralized planning is easy. Because, the generated high dimensional configuration keeps the configuration of each robot, and knowing configuration of each robot at any time leads that ensuring no robot is in collision with some obstacle or some other robots. The dimensionality of the configuration space increases as more robots are added to the control, and it increases the difficulty of centralized planning. Planners, working efficiently in high dimensions are more suitable for centralized planning. Centralized planning ensures a complete algorithm.

2.3.2. Decoupled Planning

In decoupled planning, a path for each robot is calculated as if the robot is the only robot in the environment. After the collision-free paths are computed for each robot individually, by taking only static obstacles into account, these paths are coordinated. Coordination of the paths are done by tuning the velocities of the robots along their path so there will be no collision among them. Finding the paths initially, and tuning the velocities may be complete but decoupled planning is incomplete. It

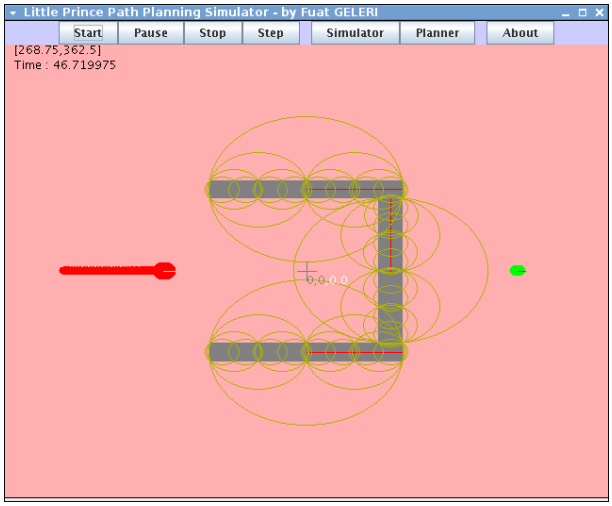

Figure 2.16. Single robot stuck on local minima

may be impossible to coordinate the paths generated during the first stage, so that no collision occurs. Alternative approaches may be prioritized planning, planning the paths of robots in an order, and behaving the robot as moving obstacle when its path has been calculated [16, 17, 14, 1].

Velocity tuning is one of the decoupled planning techniques. It coordinates independently generated paths, by searching a reduced configuration space. After a path for each robot is found, each point in these paths can be indexed, and each value can represent different points in the path. Namely one path can be represented with a single dimension. So each robot will bring one more dimension to search to the non-colliding full path planning. The reduced configuration space can be represented as $P = [0, L1]x[0, L2]x[0, L3]..$, called *coordination space*. A path joining the point $(0, 0, ..)$ to the point $(L1, L2, ..)$ in the collision-free subset of P defines a valid coordination of robots along their respective paths, from start configuration to the final configuration. By setting the relative velocities of the robots along their respective paths, the coordination of the robots is achieved. The robots may go forward and backward in their path, which gives other robots the space for maneuvering. However, if the initial paths are chosen unluckly, there may not be any possibility for coordinating the paths, such as they may lie in two distinct connected component of the free space.

If there are $p > 2$ robots in the environment, the *coordination space* will be $p - dimensional$. The i^{th} axis of this space encodes the curvilinear length along the path of the i^{th} robot. This method is named *velocity tuning global coordination*, because initially found respective paths are wanted to be coordinated together. An alternative to this method is coordination of the paths pairwise, as its name indicates; *pairwise coordination*. In this method, two robots are chosen initially and they are coordinated. This coordinated path is indexed again, so we again have one dimension for two robots. This new dimension and the reduced configuration of another robot is then coordinated. In this step we coordinated three robots and each point in P_3 determines a placement of these three robots. Each robot can be coordinated with the initially coordinated paths.

The method, *velocity tuning global coordination* is inherently incomplete, but *pairwise coordination* is more incomplete. Coordination of first i robots, P_i, may lead P_{i+1} to have no collision-free path.

3. ALGORITHMS

3.1. Sampling Based Algorithms

The algorithms that rely on explicit representation of the Cfree becomes impractical as the dimension of the configuration space grows. The sampling-based algorithms are capable of solving problems that cannot be solved with geometric, or roadmap based methods in reasonable time [1]. Sampling-based methods use various strategies for generating samples, and connecting them, to find solutions to path-planning problems [14, 18].

The efficiency of sampling-based methods comes from the fact that instead of modeling the free space, checking if a single robot configuration is in Cfree or not is much cheaper. The power of sampling based methods is shown using Probabilistic RoadMap (RPM) [1] planner as an example. It first makes a coarse sampling to obtain the nodes of the roadmap and fine sampling to obtain the edges. After the roadmap is generated, path queries are answered by finding the path between initial and goal configurations, as seen in Fig.3.1. Initially random sampling is used with PRM and that shown the probabilistic completeness of the sampling-based algorithms. However, other sampling and node-connection strategies have been shown to bring more advantages in some problems.

Sampling-based algorithms are mainly divided into two groups as single-query and multiple-query sampling based algorithms. PRM is a multiple-query algorithm. Multiple-query algorithms first generate a roadmap of the environment, and answers each query by using this roadmap, like a graph. However, single-query sampling-based algorithms do not generate this roadmap. Single-query algorithms are optimized to answer the query as fast as possible, for this, the samples generated depend on the currently constructed tree and the goal configuration. However, multiple-query algorithms make sampling to fill the space adequately, and make every part of the space accessible. So when the same environment will be used for multiple-queries,

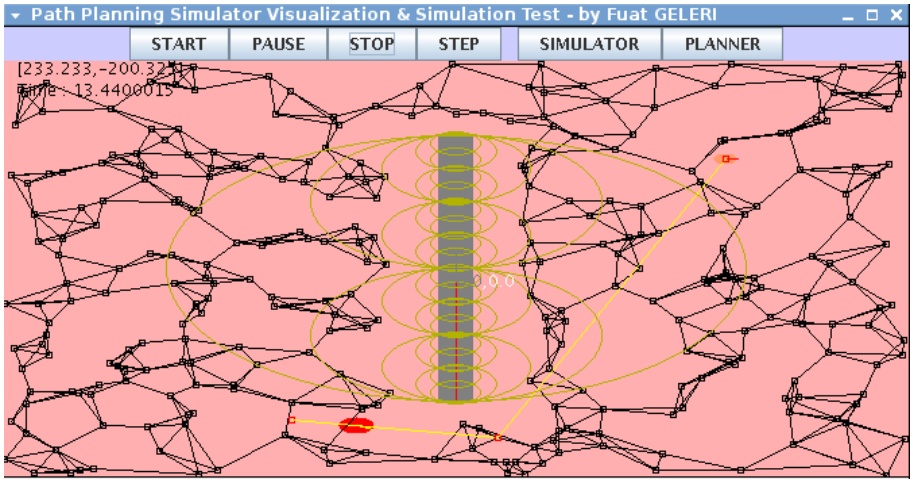

Figure 3.1. Sample run of PRM algorithm for single robot case

exploring the space initially, then making a graph search is better. However, for just a few queries, making searches optimum for these queries is better. Expansive-Space Tree planner (EST) [19], and Rapidly-exploring Random Tree Planner (RRT) [20] are single-query sampling-based path planning algorithms. In some cases, if the problem at the hand is very difficult, single-query planners need to construct very large trees to find a path. So instead of using the idea of only multiple-query or single-query, some algorithms use combination of these ideas. The Sampling-Based Roadmap of Trees (SRT) [21] planner constructs PRM-style roadmap of single-query-planner trees.

An important characteristics of sampling-based algorithms is that, they show some form of completeness. If the sampling is random, they show probabilistic completeness, if the sampling is quasirandom, or sampling on a grid then they show resolution completeness [1].

3.1.1. Probabilistic Roadmaps

Probabilistic roadmaps (PRM) and related methods are effective tools to solve path-planning problems with many degrees of freedom [2, 22]

The PRM planner has two phases, the learning phase, and the query phase. In the learning phase, a roadmap in Cfree is built to capture the connectivity of Cfree

that is to answer the path-planning queries efficiently. In the query phase, user-defined query configurations are added to the roadmap, and a graph search is made. Nodes of the roadmap corresponds to configurations in Cfree, and the edge to the free paths computed by a local planner.

In PRM the roadmap is represented by a undirected graph G = (V, E). Nodes in V are configurations that are elements of Cfree, and edges in E are the edges (c_1, c_2) that are collision-free path between c_1 and c_2.

As shown in Figure 3.2, the algorithm starts with an empty graph G = (V, E). The graph is filled with random configurations from the configuration space, if it is collision-free. After n collision-free sample configuration are added, for each configuration $c \epsilon V$, a set N_q of k closest neighbors to the configuration c is selected according to some metric *dist* from V. To connect c with $c' \epsilon N_q$ a local planner is used, and it checks to achieve a path between c and c'. If a path is found between them it is added to the roadmap.

In roadmap construction we need some components for generating random configurations, finding closest neighbors, calculation of distance function, and to generate local paths.

In the query phase shown in Figure 3.3, first, user-configurations c_{init} and c_{goal} are connected to the generated roadmap. To connect these configurations again k nearest configurations are found for each and a local planner is used to make a path between the found configurations and the user-configurations. If the roadmap is a single connected component a graph search algorithm, like Dijkstra's algorithm [23] or A* algorithm [24], will be employed for the map, and if a path is found between c_{init} and c_{goal}, then it is the result. However, if no path is found then the algorithm is failed. If the roadmap consist of more than one connected component, this means either the Cfree is not connected, or the roadmap has not managed to capture the connectivity of it, then for each component of the roadmap user-configurations will be tried to be connected, and path search will be performed.

Algorithm 3.1 Roadmap Construction Algorithm

Input:

n : number of nodes to put in the roadmap

k : number of closest neighbors to examine for each configuration

Output:

A roadmap G = (V, E)

1: $V \leftarrow \emptyset$
2: $E \leftarrow \emptyset$
3: **while** length$(V) < n$ **do**
4: **repeat**
5: $c \leftarrow$ a random configuration $\in C$
6: **until** c is collision free
7: $V \leftarrow V \cup q$
8: **end while**
9: **for all** $c \in V$ **do**
10: $N_c \leftarrow$ the k closest neighbors of $c \in V$ according to *dist*
11: **for all** $c' \in N_c$ **do**
12: **if** $(c, c') \ni E$ and $\delta(c, c') \neq$ NIL **then**
13: $E \leftarrow E \cup (c, c')$
14: **end if**
15: **end for**
16: **end for**

Figure 3.2. Algorithm for the construction of the roadmaps

Algorithm 3.2 Solve Query Algorithm

Input : q_{init} : the initial configuration; q_{goal} : the goal configuration;

k : the number of closest neighbors to examine for each configuration

G = (V, E) : the roadmap constructed in the first phase

Output : A path from q_{init} to q_{goal} or failure

1: $N_{c_{init}} \leftarrow$ the k closest neighbors of c_{init} from V according to dist
2: $N_{c_{goal}} \leftarrow$ the k closest neighbors of c_{goal} from V according to dist
3: $V \leftarrow c_{init} \cup c_{goal} \cup V$
4: $c' \leftarrow$ the closest neighbor of $c_{init} \in N_{c_{init}}$
5: **repeat**
6: **if** $\delta(c_{init},\ c') \neq NIL$ **then**
7: $E \leftarrow (c_{init},\ c') \cup E$
8: **else**
9: $c' \leftarrow$ the next closest neighbor of $c_{init} \in N_{c_{init}}$
10: **end if**
11: **until** a connection was succesful or $N_{c_{init}} \equiv \emptyset$
12: $c' \leftarrow$ the closest neighbor of $c_{goal} \in N_{c_{goal}}$
13: **repeat**
14: **if** $\delta(c_{goal},\ c') \neq NIL$ **then**
15: $E \leftarrow (c_{goal},\ c') \cup E$
16: **else**
17: $c' \leftarrow$ the next closest neighbor of $c_{goal} \in N_{c_{goal}}$
18: **end if**
19: **until** a connection was succesful or $N_{c_{goal}} \equiv \emptyset$
20: $P \leftarrow$ the shortest path $(c_{init}, c_{goal},\ G)$
21: **if** $P \neq \emptyset$ **then**
22: return P
23: **else**
24: return failure
25: **end if**

Figure 3.3. Algorithm for solving a query

3.1.2. Sampling Strategies

Various node-sampling strategies have been developed over the years for PRM. Sampling from a uniform distribution is the simplest of them, and works well for many problems. However, the choice of the node-sampling strategy can play a significant role in the performance of PRM. The node-sampling strategies should not favor specific orientations because of the representation of the environment used, and sampling distribution should be symmetry invariant.

The main idea for node-sampling is that; after a sample configuration is drawn, it is checked for collision. If it is collision-free, then it is added to the roadmap, otherwise discarded. The quality of collision checking, and speed of collision checking algorithm highly effects the success and speed of the planning algorithm. We will mention some collision checking algorithms in the next sub-sections.

3.1.2.1. Uniform Random Sampling. Uniform random sampling of *Cfree* is the simplest method. It uses uniform probability distribution over each translational degree of freedom for the allowed values of the degree. Uniform random sampling has the advantage that, carefully crafted malicious environment models cannot make the planner fail. However, in difficult planning problems running time of PRM might vary accross different runs when uniform random sampling is used.

Uniform random sampling strategy shows bad performance for some problems, especially for narrow passage problems. For queries that require going through a passage to be solved, the sampling strategy should generate samples for quite a small set. To address such problems different sampling strategies have been designed with the narrow passage problem in mind [25].

3.1.2.2. Sampling Near the Obstacles. Obstacle-based sampling methods generate samples near the obstacles, because they assume narrow passages to be between some obstacles.

Obstacle based PRM, OBPRM, is the first and very successful representative of obstacle-based sampling methods, [22]. The algorithm can be summarized as:

- Generate many configurations at random from a uniform distribution.
- For each configuration in collision generate a random direction and find a free configuration in this direction.
- Make a simple binary search between these configurations, and find the closest collision free configuration to the surface of the obstacle.
- Add this configuration to the roadmap and neglect previous two.

Yet another method is Gaussian sampler [26], that tries to solve the problem by sampling from a Gaussian distribution that is biased near the obstacles. The Gaussian distribution method can be summarized as:

- Generate a configuration using a uniform distribution.
- According to a distance *step* using normal distribution generate another configuration.
- Neglect both of the configurations, if both of them are in collision, or collision free.
- The collision-free configuration is added to the roadmap, if the other configuration is in collision space.

In another algorithm [27], first, sampling is done by letting samples to penetrate to the obstacles to some amount, which we say dilated *Cfree*. Then these samples are pushed toward the free space by performing local resampling operations.

3.1.2.3. Sampling Inside Narrow Passages. Bridge test sampling method is one of the methods using the logic of sampling inside the narrow passages [25]. This method can be summarized as:

- Sample two configurations randomly from a uniform distribution in $Cspace$

- If both of the configurations are collision free, then add both
- If only one of them is collision free, discard both
- If both in collision, then the configuration half way between these configurations will be checked for being collision free, if it is collision free it will be added to the roadmap and others will be neglected otherwise all will be neglected

Inside narrow passages the bridge will be shorter, however in open space the construction of short bridges is difficult, so via favoring the construction of short bridges the bridge planner samples points inside narrow passages.

Another method uses the idea of using Generalized Voronoi Diagrams (GVDs) [1]. Using the Generalized Voronoi Diagrams is costly, but it is possible to find samples on the GVD without computing them explicitly. The algorithm moves each sample configuration until it is equidistant from two points on the boundary of *Cfree*. GVD algorithm captures well narrow passages, and some graphics hardware supports approximate calculation of GVD, which makes the method popular.

3.1.2.4. Visibility-Based Sampling. Add the randomly generated configuration to the roadmap only if it cannot be connected to any previously added configurations, or it can be connected to more than one already generated configurations [11].

3.1.2.5. Quasirandom Sampling. Quasirandom sampling methods are deterministic alternatives to random sampling. Running time of the algorithms are the same for all the runs due to the deterministic nature of quasirandom sequences. The resulting planner is resolution complete [1].

3.1.2.6. Grid-Based Sampling. Initially rather coarse resolution of the grid is used, and in query phase c_{init} and c_{goal} tried to be connected to nearby grid points. The resolution of the grid that is used to build the roadmap can be progressively increased, by either adding points one at a time or by adding an entire hyperplane of samples. This sampling method is also resolution complete [1].

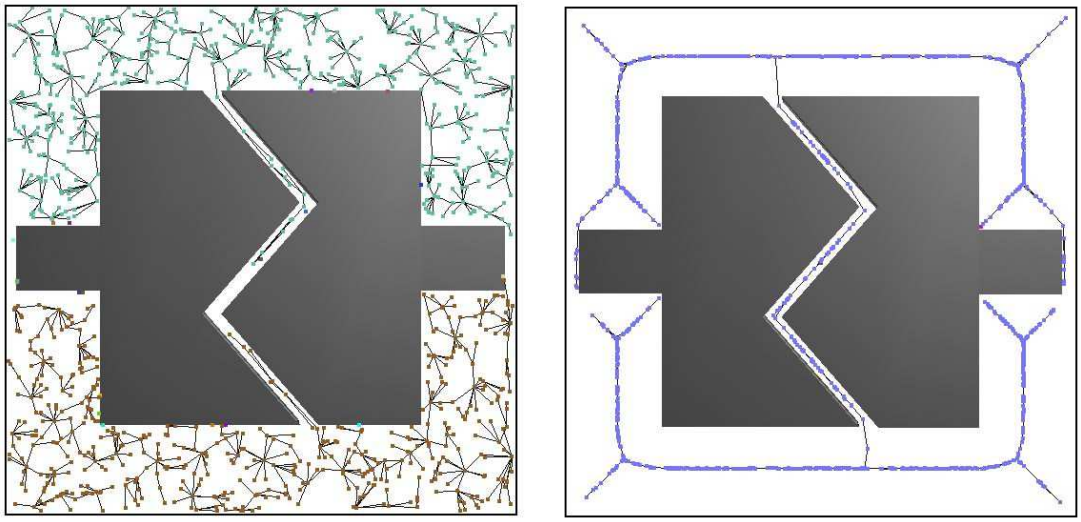

Figure 3.4. Two sampling strategies for long narrow passage problem a. Uniform sampling method, b. Sampling on Medial Axis of the Free Space method [28]

3.1.2.7. Benefits of Different Sampling Methods. Uniform random sampling works well for many problems, however when the issue is consistency in the running time, using quasirandom sampling brings some advantages. When the problem involves narrow passages, algorithms designed for this kind of problems will give better results.

If we compare Figure 3.4.a and Figure 3.4.b, we can see that Figure 3.4.b has less configuration in the roadmap and has connected the environment better.

The same is true for Figure 3.5.a, and Figure 3.5.b. So as shown by the MAPRM algorithm [28], sampling strategy is an important factor for the success of planning algorithms. Selected sampling method may increase the performance of the algorithm, and make the algorithm more robust and faster.

3.1.3. Connection Strategies

Selection of pairs of configurations that will be tried for connections by a local planner is the next step for constructing the roadmap. The objective of the connection strategy is selecting the configurations that are to succeed in making a connection. Short connections have good chance of being collision free, so the algorithms should

Figure 3.5. Two sampling strategies in maze a. Uniform sampling method, b. Sampling on Medial Axis of the Free Space method [28]

try to make such connections. Because k-nearest neighbor algorithm leads nearby samples to be checked for connection, it leads to shorter connections. The selected algorithm for selecting the neighbors and the implemented local planner can affect the performance of the system drastically.

3.1.3.1. Selecting Closest Neighbors. In this method, when a new sample is generated, k nearest already generated sample configurations are searched in the roadmap. Then the new sample is tried to be connected to each one of them. Because the sample is tried to be connected with the samples those near to it, the length of the connection is rather short. Because collision checking is the most time consuming part of the planning, by generating short connections this big consumption is reduced.

3.1.3.2. Creating Coarse Roadmaps. The computation of edges that are part of the same connected component will not improve the connectivity of the roadmap. So preventing from making connections in the same component will result speed up in the roadmap construction. The simplest implementation of this idea is connecting a configuration with the nearest node in each component that lies close enough.

This method achieves good connectivity with less number of samples, but in the

query phase, the found paths may be rather long. This may be fixed by applying postprocessing techniques like smoothing. However, allowing some redundant edges in the roadmap can significantly improve the quality of the initially found path without significant overhead, so we can achieve shorter paths.

3.1.3.3. Connecting Connected Components. In some cases, because of the difficulty of the problem, or inadequate sampling, the resultant roadmap may be composed of several connected components. The quality of the roadmap can be improved by the methods that try to connect these components. So by placing more nodes in difficult regions of *Cfree* more effective algorithms can be achieved for connecting different components of the roadmap.

3.1.4. Collision Checking Methods

Collision checking is the basic operation of all sampling based algorithms. Instead of modeling the free space, sampling based algorithms uses a collision checking algorithm to see if a random configuration is collision free, or not. Collision free samples and edges between them composes the roadmap to be used for path planning queries. Not only the random configurations should be collision free, but also the edges connecting them should be collision free.

In simulating a robotic environment, to make the environment more realistic, the collision checking algorithm should provide high accuracy. In each simulation step the dynamic objects will change their positions, and checking only the start and end position of the dynamic objects for collision may result unidentified collisions to occur in the mean time. So collision checking should be extended to include the time passed between start and arrival. One of the methods is creating a convex hull around a robot's location at these two configurations, as shown in Figure 3.6. This method guaranties to catch the collision if it occurs, but it will definetely make the collision checking slow. Instead of creating such an object, via subdividing the given time interval in half and testing for collision at the midpoint, and repeating this calculation recursively for each

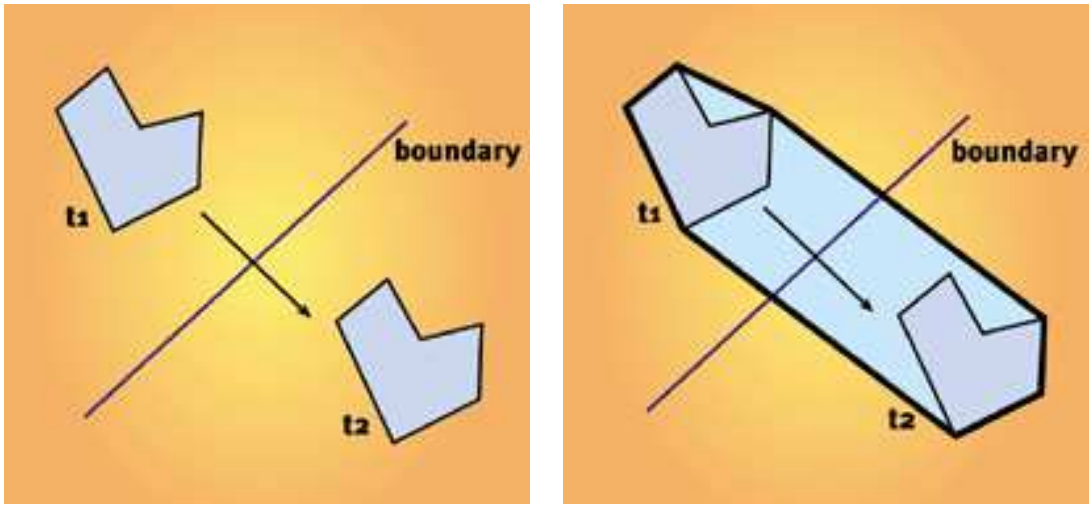

Figure 3.6. Convex hull created around an objects location at two different times. Pictures is from [29]

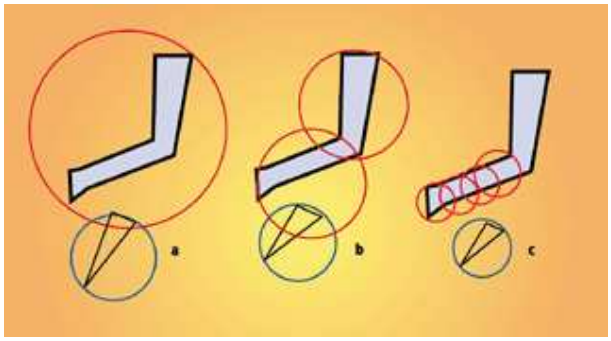

Figure 3.7. Sphere subdivision algorithm for collision checking. Picture is from [29]

resulting half will provide a faster and good enough collision checking. This algorithm is easier to implement, and runs faster but is less accurate.

The other problem in collision checking is checking whether an object intersects with any other object in the environment. If the environment contains many objects checking each object with each other for collision will be at order of $O(N^2)$. Instead of checking each object with each other we can divide the objects into two sets, stationary objects, and moving objects. So we will not check objects for collision those can never collide. Other advanced methods may also be used like building an octree of the scene [30].

One of the simple but efficient methods is approximating each object or part

of objects with a sphere. So when making collision checks only these spheres will be checked for collision, which will make the process computationally cheaper. We should only check whether the square of the distance between two spheres is more than the sum of the squares of the radii of the spheres. If we represent a big object with only a bounding sphere, the approximation will be quite rough. So, via subdividing this big sphere into smaller spheres, we can get a more detailed approximation, Figure 3.7. When making collision checks, first the big encapsulating spheres will be checked for collision, if they are not colliding, then we will not need to check the subdivisions. If the bigger encapsulating sphere collides, then we will check each smaller sphere with each other for collision. We may subdivide the spheres until we are satisfied with the approximation.

There are quite more advanced algorithms for collision checking but for our purpose using sphere subdivision algorithm seem enough.

3.1.5. Expansive-Spaces Trees

EST is an efficient single-query planner that finds a path between c_{init} and c_{goal} rapidly. Kinodynamic problems involve both finding a path and the control inputs for the robots to take the path. Namely, we can see kinodynamic planning as motion planning. For kinodynamic planning a single tree is typically build and EST is better suited for such problems. EST algorithm is a probabilistically complete algorithm.

3.1.5.1. Construction of Trees. Let T be one of the trees T_{init} or T_{goal}. The algorithm first selects a configuration c from T, and then generates a random configuration c_{rand} near c with a uniform distribution. The configuration c is selected randomly with probability $\pi_T(q)$. Then the local planner Δ attempts to make a connection between c and c_{rand}. If the attempt succeeds, the configuration is added to the set of configurations, the vertices of T, and (c, c_{rand}) are added to the edges of T. This process continues until a specified number of configurations are added to the T. The process is described in Figure 3.8 and Figure 3.9.

Algorithm 3.3 Build EST Algorithm

Input:

c_0 : the root configuration of the tree

n : number of attempt to expand the tree

Output:

A tree T = (V, E) that is rooted at c_0 and has less than n configurations

1: $V \leftarrow c_0$
2: $E \leftarrow \emptyset$
3: **for** $i = 0$ to n **do**
4: $\quad c \leftarrow$ a randomly choosen configuration from T with probability $\pi_T(q)$
5: $\quad extend\ EST(T, q)$
6: **end for**
7: return T

Figure 3.8. The algorithm for building an EST tree

Algorithm 3.4 Extend EST Algorithm

Input:

$T = (V, E)$: an EST

c : a configuration from which to grow the tree

Output:

A new configuration c_{new} in the neighborhood of c, or NIL in case of failure

1: $c_{new} \leftarrow$ a random configuration $\in C_{free}$ near the c
2: **if** $\Delta(c, c_{new})$ **then**
3: $\quad V \leftarrow V \cup \{c_{new}\}$
4: $\quad E \leftarrow E \cup \{(c, c_{new})\}$
5: $\quad$ return c_{new}
6: **end if**
7: $return\ NIL$

Figure 3.9. Extend EST Algorithm

In the PRM algorithm, the generated random configuration is added to the roadmap directly if it is in C_{free}. However, the EST algorithm adds the configuration to the tree if it can be connected to the existing configurations in the tree. So there happens a path from the root of the tree to each configuration in T.

The effectiveness of the EST algorithm highly depends on the $\pi_T(q)$. The algorithm to be used for $\pi_T(q)$ should not oversample any region of C_{free}. To scatter the sampled configurations various solution methods are proposed. One of the methods is attaching a weight value $w_T(q)$ to each configuration c that constitutes the count of the number of configurations within some predefined neighborhood of c. Then, choose c as inversely proportional to $w_T(q)$. So, the configurations with sparse neighborhoods are more likely to be picked. Another method is dividing the C into grids and biasing the random configuration selection toward to cells with fewer configurations in it. So first a cell is chosen then a configuration within this cell is choosen randomly.

3.1.5.2. Merging of Trees. While expanding the trees rooted at the c_{init} and c_{goal}, the algorithm also tries to connect these trees, so that a path is constructed. First a new configuration in T_{init} or T_{goal} is generated, then this configuration is tried to be connected with k closest configuration in the other tree. If a connection is generated then the merging is successful. If no connection is successful then the trees are swapped and the process continues.

When a successful connection is found for $c_1 \in T_{init}$ and $c_2 \in T_{goal}$ by using the local planner Δ, the path between c_{init} and c_{goal} can be obtained by concatenating the path from c_{init} to c_1 in T_{init} to the path from c_2 to c_{goal} in T_{goal}.

3.1.6. Rapidly Exploring Random Tree

As like EST, the RRT algorithm is initially developed for kinodynamic motion planning problems, and a single tree is built by this algorithm. The algorithm efficiently covers the space between c_{init} and c_{goal}, and it is shown to be probabilistically complete,

shown in Figure 3.10.

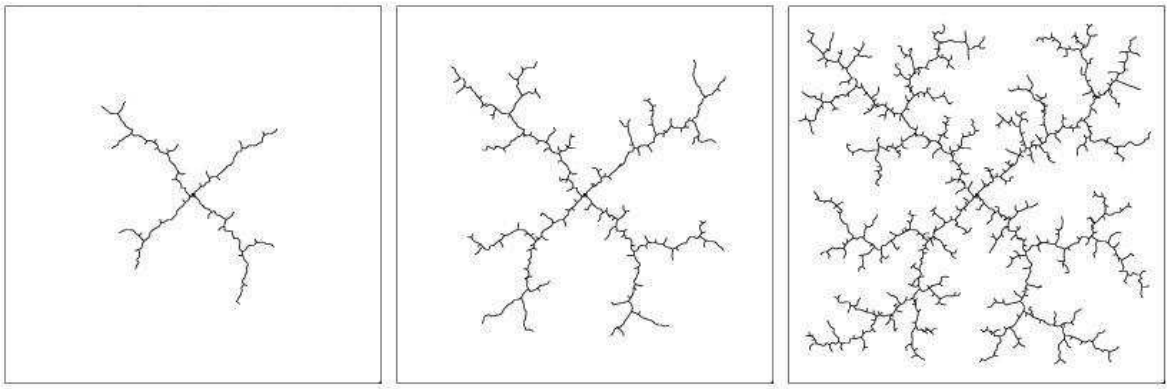

Figure 3.10. Growing of an RRT tree

3.1.6.1. Construction of Trees. Let T be one of the trees T_{init} or T_{goal}. At each iteration a random configuration, c_{rand}, is selected uniformly in C_{free}. Then T is searched to find the nearest configuration, c_{near}, to c_{rand}. A new configuration is generated on the line from c_{near} to c_{rand} with moving a distance $step\_size$ from c_{near}. If this newly generated configuration, c_{new} is collision free, and it can be connected to c_{near}. Next, the configuration is added to the vertices of T and the edge (c_{near}, c_{new}) is added to the edges of T. Pseudecode of algorithm is given in Figure 3.11 and Figure 3.12.

The $step\_size$ parameter can also be chosen dynamically, based on the distance between c_{near} and c_{rand}. If the two configurations are far from colliding, choosing a large $step\_size$ value, and small otherwise, is sensible. Another alternative is a greedier one that tries to move c_{new} toward c_{rand} as much as possible, as shown in Figure 3.13.

If we keep the $step\_size$ small, then the tree will be fed with many configurations. It will make the search for the nearest configuration in the tree to be more expensive, and the memory consumption to be increased. In such a case instead of adding all intermediate samples, adding only the last sample of the Extend RRT iteration may be better. Another optimization may be generating the c_{rand} samples near to the c_{goal} configuration with some small probability. This will bias the sampling toward the goal and increase the efficiency of the algorithm.

Algorithm 3.5 Build RRT Algorithm

Input:

c_0 : the root configuration of the tree

n : number of attempt to expand the tree

Output:

A tree T = (V, E) that is rooted at c_0 and has less than n configurations

1: $V \leftarrow c_0$
2: $E \leftarrow \emptyset$
3: **for** $i = 0$ to n **do**
4: $\quad c_{rand} \leftarrow$ a randomly choosen configuration from C
5: $\quad extend\ RRT(T, c_{rand})$
6: **end for**
7: return T

Figure 3.11. The algorithm for building an RRT tree

Algorithm 3.6 Extend RRT Algorithm

Input:

$T = (V, E)$: an RRT

c : a configuration toward which to grow the tree

Output:

A new configuration c_{new} toward c, or NIL in case of failure

1: $c_{near} \leftarrow$ closest neighbor of $c \in T$
2: $c_{new} \leftarrow$ progress c_{near} by $step\_size$ along the straight line between c_{near} and c_{rand}
3: **if** $c_{new} \in C_{free}$ **then**
4: $\quad V \leftarrow V \cup \{c_{new}\}$
5: $\quad E \leftarrow E \cup \{(c_{near}, c_{new})\}$
6: $\quad$ return c_{new}
7: **end if**
8: $return\ NIL$

Figure 3.12. The algorithm extends an RRT tree

Algorithm 3.7 Connect RRT Algorithm

Input:

$T = (V, E)$: an RRT

c : a configuration toward which to grow the tree

Output:

connected is returned if c is connected to T; $failure$ otherwise

```
1: repeat
2:     c_new ← extend RRT(T, q)
3: until c_new = q or c_new = NIL
4: if c_new = q then
5:     return connected
6: else
7:     return failure
8: end if
```

Figure 3.13. Trying to connect two RRT trees in RRT Connect algorithm

3.1.6.2. Merging of Trees. In the merging step two trees rooted at c_{init} and c_{goal} are tried to be connected. In the literature this algorithm is named as **RRT Connect** algorithm [20] . The main idea of the algorithm is growing T_{init} and T_{goal} toward each other. At each iteration, initially a random configuration is generated. One of the trees tries to extend its closest node toward this random configuration, c_{rand}. So new configurations are added to the tree, c_{new}. In the second step, the other tree is extended toward to c_{new}. If it is successful, then the planner terminates, and the trees are connected. Otherwise, the trees are swapped and the process continues for a certain number of times.

For merging of the trees either the one step toward to c_{rand} or greedily adding new configurations until reaching c_{rand} versions of the algorithm can be used. They may be used in combination too by changing *extendRRT* method in Figure 3.14 with *connectRRT*. Once two RRT trees are connected the path can be found by connecting the paths from the root of the trees to the connected nodes.

Algorithm 3.8 Merge RRT Algorithm

Input :

T_1 : first RRT

T_2 : second RRT

n : number of trials

Output :

merged if the two RRTs are connected to each other, otherwise $failure$

```
1: for i = 1 to n do
2:    c_rand ← a random configuration ∈ C_free
3:    c_new,1 ← extend RRT(T_1, c_rand)
4:    if c_new,1 ≠ NIL then
5:       c_new,2 ← extend RRT(T_2, c_new,1)
6:       if c_new,1 = c_new,2 then
7:          return merged
8:       end if
9:       SWAP(T1, T2)
10:   end if
11: end for
12: return failure
```

Figure 3.14. RRT Connect algorithm's merging part

3.1.7. Lazy Algorithms

It is observed that generally most of the edges in the roadmaps, or trees are not in the final found path. So making collision checks for these edges were not really necessary. A method is proposed that delays the collision checking until a path between c_{init} and c_{goal} is found, and it checks only the edges in the path, exactly once.

Therefore the number of collision-checks performed during planning is minimized. Because already checked edges are not again checked for collision, and only the edges in the found path is checked, both the number of collision-checks are minimized, and

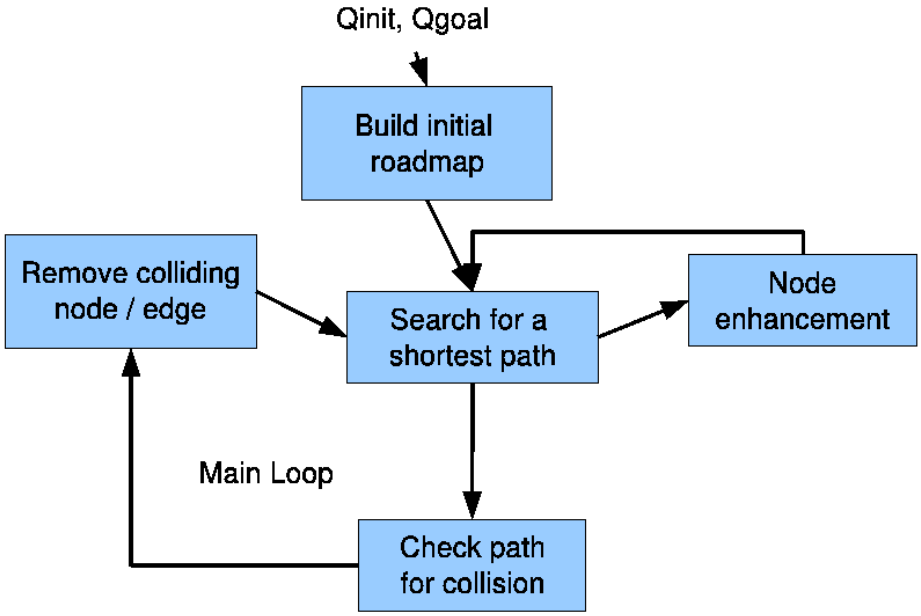

Figure 3.15. High-level description of Lazy PRM

also the knowledge of previous queries is carried to next queries, which speeds up subsequent queries.

3.2. Post Processing

Sampling based path planning algorithms focus on searching a path, but generally they do not propose the found path to be the optimum. Their main aim is finding a path with minimum time and memory space usage.

After a path is found, there are some algorithms to improve the quality of the path according to some criteria. We name this step postprocessing step, that improve the shortness and smoothness of the path connecting c_{init} to c_{goal}.

To improve the shortness of the path, we check whether nonadjacent configurations c_1 and c_2 in the path can be connected by the local planner. If they can be connected, then all the nodes between c_1 and c_2 will be skipped, and a shorter path will remain. The points c_1 and c_2 may be chosen greedy, or randomly. Algorithm of the greedy approach can be seen in Figure 3.17. Figure 3.16 shows the paths before and

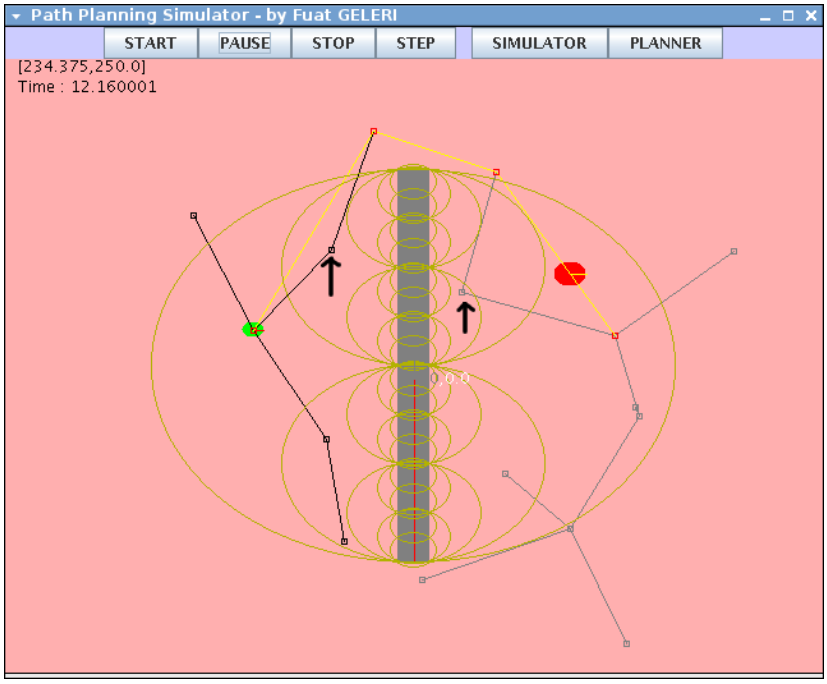

Figure 3.16. Postprocessing is applied to shorten the found path

after the path shortening algorithm is applied. The arrows indicates the configurations those are skipped, and a shorter path is achieved.

The configurations c_1 and c_2 might be in a relatively uncluttered part of C_{free}, but not connected. Reason for this might be the *dist* function applied, which may lead the k *closest* neighbor query not to return them as neighbors. Generally, when sparse roadmap connection strategies are applied such cases appears. So by applying the path shortening algorithm we fix the problem and connect the configurations in the uncluttered part of C_{free}.

Some robots may require the found path to have smooth curvature. So, instead of shortening the path the aim may be generating such paths. To generate more smooth paths we can interpolate the curves, such as; splines can be used. Until curves that satisfy both the smoothness properties and the collision avoidance criteria collision checking should be performed.

Postprocessing step may generate paths those are shorter and smoother. However,

Algorithm 3.9 Shorten Found Path

Input:

path : the found path

n : number of trials

Output:

Apply the operations onto the given input, *path*

```
1: for i = 0 to n do
2:    c_0 ← random configuration on the path
3:    c_1 ← another random configuration on the path
4:    if local planner can connect c_0 and c_1 then
5:       path ← path− configurations between c_0 and c_1
6:    end if
7: end for
```

Figure 3.17. Simple path shortening algorithm

this may bring a significant overhead on the time to respond to the query. Instead of postprocessing, wanted optimality criteria can be tried to be achieved during the roadmap construction phase.

3.3. Multiple Robot Coordination

Path planning is searching for a path that will lead a robot from its initial configuration to a desired configuration. However, if there are more than one robots to search for a path, then the problem involves coordination of the robots. We should both find path for each robot, and coordinate these paths such that no robot will collide.

For multiple robots the path search may be done all together, centralized, or the planning may be done separately, decentralized. Details of these approaches already given in the previous chapters of this thesis. Next, we will discuss how we implemented them.

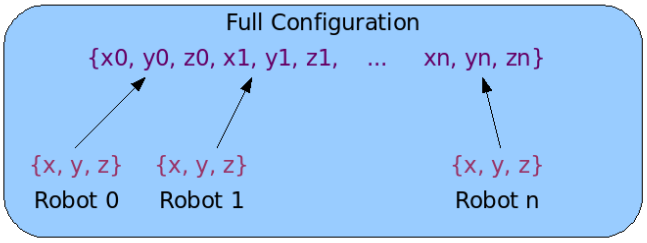

Figure 3.18. A big configuration is achieved by appending robot configurations

3.3.1. Centralized Approach

In the centralized approach, we append the configuration of each robot, and achieved a big configuration representing the configurations of all of the robots. So via using this big configuration, we do not need any other representative to show the configuration of each robot. When we make the search at the universe of this big configuration, the found path will be the path for each robot. So while searching a path, we both find a path for each robot, and we also coordinate the robots so that no collision occurs.

While checking the edges in the path for collision, we divided the edges into parts by dividing them and the parts from their center recursively. Namely, we assumed that each robot will arrive at the center of the edge at the same time, and will reach the end of the edge together. So this approach provides us a methodology for finding the velocities of the robots. An edge in the big configuration space is a set of edges, an edge for each robot. This edge in the big configuration space may result short or long edges for each robot. However, the amount of time each robot will use to take these edges should be the same to make the robots move in coordination. So at each time we can calculate the position of the robot easily.

Figure 3.19 shows two robots, and the next edges of their paths. The robots will reach the end of these edges at the same time. So the robot with the longer edge will take the path with its maximum velocity. However, the robot having shorter edge should lower its velocity so that they will be in coordination. They will be at the center

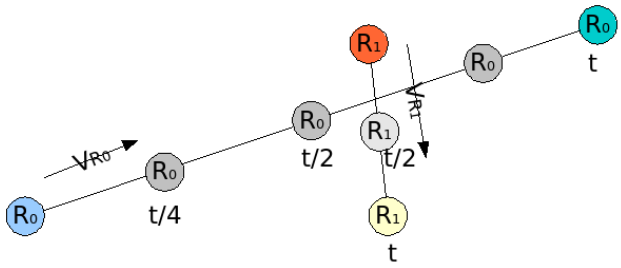

Figure 3.19. Tuning the velocities in centralized approach

of their edges at the same time, and they will also reach to the center of the first half, and the last half at the same time too.

Coordination of the robots should be thought while finding the paths in the centralized approach. The other options may also be tuning the velocities after finding the paths. In our case, the collision checker algorithm we used checks the robots as they will be coordinated in the Figure 3.19. This approach simplified the problem and path planning algorithms designed for single robots could also be used by extending collision checker only a little.

3.3.2. Decentralized Approach

In this approach the controllers find the path of each robot initially, and let the robots to take the found paths. When they coincide with another robot in their ways they try to escape from these robots, first reactively, then intelligently.

In centralized approaches because the robots are coordinated from the start, if one robot violates this coordination while escaping from the robot cluttering its path, then whole the path planning should be done again. However, in the decentralized approach, each robot has the responsibility of itself. So if a path is disturbed, the other robots may continue to follow their own paths.

3.3.2.1. Simple Reactive Escape. This is a truly basic escaping type. If the robot sees a robot cluttering its path, it tries to get away from the robot, that will lead possible collisions. The direction of the escape will be toward the reverse of the vector connecting those two robots.

If the reactive escape is successful, the robot continues to its path. However, if the number of trials reaches a limit, while escaping from the moving obstacle, then the robots should search for an intelligent way to escape from the robot.

3.3.2.2. Intelligent Escape. If the robot is not able to escape from the moving obstacle, or from other robots we may understand that there is a local minima problem. Therefore, the intelligent escape algorithm generates a random configuration and leads the robot to first go to this random configuration, than replan path to the real goal.

Simple reactive escape makes the robots to escape from each other with simple, small steps, and intelligent escape makes this escape to be a big step.

3.4. Artificial Potential Function (RBOT)

In [5] and [7], an algorithm which uses potential function to navigate circular robots is given. This algorithm requires circular representation of the robots, and the environment. The algorithm can work in a centralized, decentralized, or partially centralized fashion. In [5], a navigation function is proven to be analytic, admissible, polar, and morse. If a navigation function is proven to be admissible, that means this function attains its maximum on the boundary. If it is polar, its unique minimum must be at the goal configuration. Being morse means all critical points to be non-degenerate.

APF algorithm is proven to be an exact navigation function. It is capable of coordinating multiple robots. Next we will describe the algorithm simply. The algorithm is quite powerful, however it contains some multiplication factors those lead the algo-

rithm to produce underflows or overflows for multiple robot cases with more than 15 robots. We solved this problem by not altering the navigation function, but converting some portion to log sums.

According to the destination configuration, current configurations of other robots, and the radius of the environment, two forces for each robot is calculated. Each robot calculates these two forces and takes their next step according to those values. One of them is the attractive force, which leads the robot to its goal, and the other is the repel force, which saves the robot from colliding with other objects, and the robots in the environment.

These two forces will carry the robots to their destinations. They will change the position of the robot. So an integration formula is devised according to properties of the robots, that will sum the force, and calculate next positions. More information about integration can be found in Appendix A. The step size of the integration formula decides about the affect of the force at each step. Adaptive integration formulas calculates the forces more than once before taking any step, and calculates the error rates. These error rates are used to accord the step size values, Figure 3.20.

In Figure 3.20, calculation of next control inputs for a robot is shown. First, a runge kutta algorithm is applied to calculate control inputs. This call is used to calculate the scalings. Then, again runge kutta algorithm is used to get the next configuration of the robot. This configuration is the calculated configuration, the robot supposed to be. So the velocity is calculated as taken path divided by time. Scaling factors are decreased when robots get near to each other or to their targets.

We made some improvements on this algorithm. First of all, the algorithm has a part that multiplies the distances of robots to each other. This part results overflows or underflows. Instead of using multiplication, we changed it to sum of logarithms. Furthermore, the algorithm was lacking velocity parameter. We added a velocity parameter to both calculation of integration potential and to the runge kutta algorithm. Again, centralized, and partially centralized versions of the algorithm is implemented.

Algorithm 3.10 Integrating Potential

Global:

$xScale$: used in adapting integration step size

$yScale$: used in adapting integration step size

Input:

$robot$: The robot to calculate next control inputs

$neighbors$: The neighbors of the robot

Output:

Next control inputs for the $robot$

1: $savedConfig \leftarrow current\ configuration\ of\ the\ robot$
2: $ApplyRungeKutta(robot, neighbors)$
3: $xScale \leftarrow \|robot\_config - target\_config\|^2 + |k1x * h| * maxVel + tiny$
4: $yScale \leftarrow \|robot\_config - target\_config\|^2 + |k1y * h| * maxVel + tiny$
5: $newConfig \leftarrow adaptRungeKutta(robot, neighbors)$
6: $diff \leftarrow newConfig - savedConfig$
7: $velocity \leftarrow diff/simulation\_step\_time$
8: **if** $\|velocity\| < maxVel$ **then**
9: $\quad velocity \leftarrow maxVel$
10: **end if**
11: $result \leftarrow velocity$

Figure 3.20. Algorithm for finding the next control inputs

The results of these changes can be found in the results section of the thesis. The algorithm was working properly for approximately $15, 20$ robots previously. Now, the algorithm handles 250 robots, without causing any overflow or underflow, easily.

3.5. Lazy PRM RRT Connect

We classified sampling based path planning algorithms as multiple query, and single query algorithms. Multiple query sampling based path planning algorihtms generally generates a model of the environment in the preprocessing step. The generated

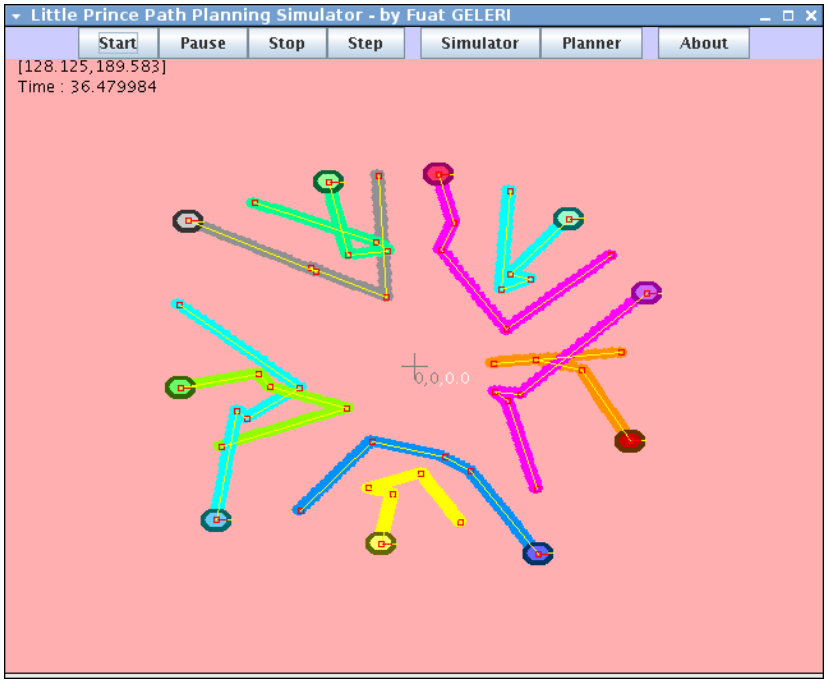

Figure 3.21. RRT Connect algorithm leads the robots toward the center of the free space

model in probabilistic roadmap algorithms is a map of the collision free space. This map is generated by generating random samples and connecting them. This preprocessing step handles most of the process, and at the query phase only a graph search is performed. These algorithms are better suited for multiple query scenarios. The aim of these algorithms is modelling the collision free configuration space as well as possible with minimum number of representatives as possible.

Multiple query path planning algorithms are modelled to answer various path search queries. However, if the query is known beforehand, the search algorithm may make the search according to the request. Therefore, single query sampling based path planning algorithms make biased searches, and they try to search the free configuration space as little as possible. As can be seen, two of the different approaches, one is trying to cover the free space as much as possible, and the other wants to limit this search. In multiple robot path planning problems, biased searches make the robots to get too near to each other. As seen in Figure 3.21, the RRTConnect algorithm first gets the robots toward the center and then the robots are travelled toward their goals. Whereas, the

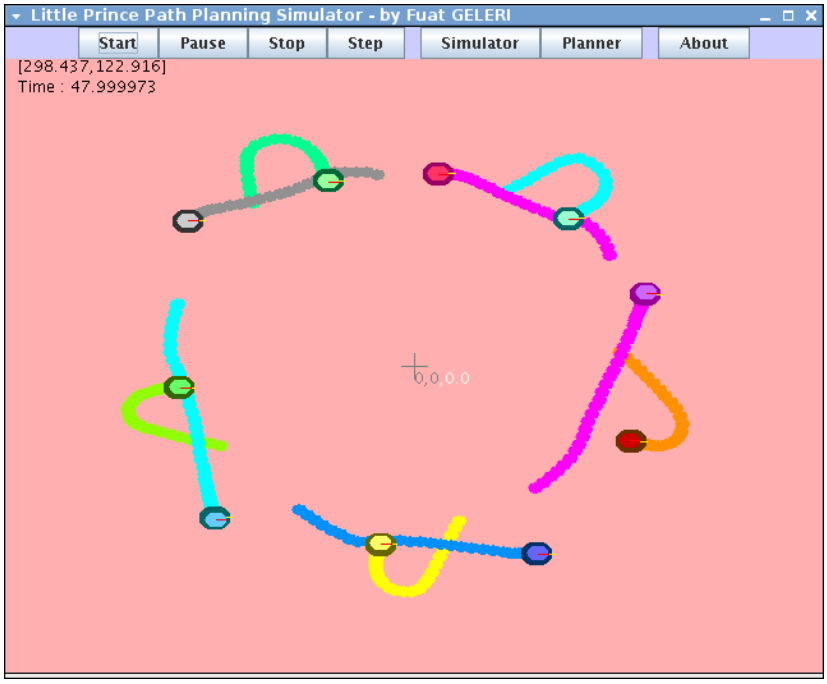

Figure 3.22. RboT algorithm uses the free space effectively

multiple query path planning algorithms would make the robots to use the big free space, and will lead them to easier paths. As seen in Figure 3.22, the RboT algorithm leads the robots to use the free space as much as possible. So, if we combine the RRTConnect algorithm with LazyPRM algorithm we may achieve an algorithm that will both make a biased search, and will also try to use the free space as much as possible.

Lazy PRM RRT Connect algorithm first generates a simple roadmap of the free configuration space. This graph will be the main roadmap in the path planning searches. After a path is found on this main roadmap, the mission of the RRT Connect algorithm will be connecting the robots to the start and end of this path.

4. SIMULATIONS

4.1. Simulator Design

There are many robotic simulators available. However, non of them is specialized for path planning problems, and have their limits. These limits prevent detailed inspection and successful comparison of path planning and navigation algorithms.

Engineers want to have a highly configurable platform, that will enable them to generate test scenarios they think. Adding new robots, changing the types of the obstacles, adding robots to different controllers, taking statistics of different controllers for the same environment is some of their wishes.

There are many path planning and coordination algorithms, those have been designed for different problems. For example some algorithms are designed to generate the map of the environment and find a path on this map. Their aim is representing the free configuration space better, and their comparison must be on this basis. They do not propose anything about multiple robot coordination. However, some algorithms directly focus on this problem, and also work reactively.

Because of these reasons, a simulator, specialized for path planning, is designed, that gives some great flexibilities to the users, and enables different algorithms to be compared in fair.

In Appendix B, the simulator will be described in detail. Topics will be about what kind of work spaces can be generated by the simulator, how algorithms communicate with the simulation part, the boiler-plate supplied for the planning algorithms, and supplied path planning algorithms. Moreover, because we implemented this simulator we can extend it according to our needs by adding new robot, obstacle, and controller types.

4.2. Measures

The algorithms will be compared by various measures. One algorithm may give the best result for a measure, however its result for another measure may be unacceptable. So the performance of the algorithm is a mixture of its performace on every measure. We have various measures like success rate, average elapsed time, normalized robot path length, and number of nodes if applicable.

4.2.1. Success Rate

We run the simulations for each algorithm in each scenario 100 times. Success rate shows the percentage of successful endings to all runs. Each algorithm may propose other rules for successful completion. In general, if the robots reach to their goal without colliding to each other, or any other obstacle in the environment in an acceptable time amount, the run is said to be successful.

4.2.2. Average Elapsed Time

This measure is used to compare the time complexity of each algorithm. Average elapsed time value may change according to the number of robots in the environment in each algorithm. This change may be exponential or linear. Linear increases are better than exponential increases.

4.2.3. Normalized Robot Path Length

Finding the shortest path is another criteria for the performance of the algorithms. Some algorithms result to shortest path, but they require too much time to compute. Moreover they lead robots to get too near to each other. Big NRL values may be acceptable, if it is required for the algorithm to lead robots to use the free space better.

4.2.4. Space Complexity

Sampling based algorithms' success rate increases with the number of used nodes. However addition of new nodes means increase in the used memory, and the search time for a valid path in the graph. So the algorithm which uses less number of nodes and covers the space better is the better algorithm.

4.3. Scenarios

After implementing various types of reactive and deliberative path planning algorithms, we designed some test scenarios that will assess all properties of these algorithms. For example, APF method works on the environments with no obstacles, so to compare APF with other path planning algorithms we prepared simulation environments those involve no obstacles. In these tests capability of the algorithms in path finding, their scability for multiple robots, and their success in the environments those contain moving obstacles are inspected.

To test different kind of path planning, and robot navigation algorithms, the tests are designed to let each algorithm to show their power. Algorithms like PRM, Lazy PRM, Lazy PRM RRT Connect are designed for multiple query problems. These algorithms initially generates a roadmap, then for each query searches this roadmap to find a path. Generating the initial roadmap is an expensive process, but searching a path in the roadmap is cheap in these algorithms. Especially PRM algorithm makes most of its processing in the initial roadmap generation, so to solve a path planning query only a graph search is performed. However, Lazy PRM algorithm both searches the graph, and continues checking the found path for collision. So Lazy PRM distributes the job between the initialization and the query steps. Lazy PRM RRT Connect algorithm is an extension to Lazy PRM. In this algorithm, the process is also distributed between the initialization and the query steps.

The other algorithms, like RRT, and RRT Connect, are designed to be used for single queries. Their aim is to respond a single query as fast as possible. Instead of

inspecting the configuration space, and trying to fill the space adequately, their aim is finding a path by using as minimum nodes as possible. So single query algorithms have nearly no preprocessing, but they make the search job fully in the query step.

Reactive algorithms resemble to single query algorithms. They do not generate any initial roadmap but make their searches in a reactive manner in the query step.

As seen, we have different approaches and to make a fair comparison we have to take all these into consideration. A testing platform is designed to enable users for testing the algorithms with various options. An algorithm may be initialized from the stratch for each query, or it may make the preprocessing once and use this already generated information for the next queries. To make a fair comparison between the multiple query approaches and the single query approaches we selected the second option. So we gave chance to these approaches to compete in the equal conditions.

For each test the algorithms build their initial roadmaps, if they need, at the first step, and they use the same roadmap for the subsequent calls. Already, some algorithms like Lazy PRM and Lazy PRM RRT Connect works cummulatively. At each query they try to optimize the roadmaps they have already generated.

Next, we will test the algorihtms for different cases. Initially algorithms will be tested in an environment with no dynamic or static obstables. Then we will compare sampling based algorithms in a maze. As the last test scenario algorithms will be tested in an environment that contains moving obstacles. The algorithms used in the last test scenario are improved to compete in such dynamic environments. They make simple escapes and replannings to escape from dynamic objects and robots in the environment.

4.4. No Obstacles

APF algorithm is a powerful, reactive algorithm which uses potential function to navigate the robots. Because it is implemented for the environments with no obstacles, test scenarios involve no obstacles. In these tests the robots will try to reach some near

target configurations which are distributed circularly, without colliding each other. Because the environment does not contain any static obstacle, the problem can be seen as a robot coordination problem. One algorithm coordinates the robots in reactive basis, and the others coordinate in deliberative manner.

In these tests we have an environment with no static or dynamic obstacles. The aim is coordination of the robots without touching to each other toward their goal configurations. The volume of the environment is kept unchanged and number of robots is increased, and the distance between the targets of the robots is decreased to achieve various test configurations for this scenario. Each test is run for 100 times, and their results are supplied in the next subsections.

The path planning algorithms used in these tests are centralized sampling based path planning algorithms. So the dimension of the algorithm is increased by the dimension of the robot's configuration space for each newly added robot.

4.4.1. Five Robots

In this test scenario, we have five robots, and each will have three dimensions as x, y coordinates and the orientation of the robot. Therefore, the path planning algorithms will search for the path in the configuration space with fifteen dimensions.

As the targets, and the initial configurations of the robots get nearer, the problem gets more difficult. Near targets need the algorithm to make deeper searches. Let us think it in two dimensions, if the targets are far, then we can divide the environment into big rectangles and can still have the targets in two different rectangles. However, if the targets are near, the size of the rectangles should be small enough to have the targets in different rectangles. The problem, that we are trying to solve here, is quite the same. If the targets are near, the path planning algorithms should make higher amount of sampling to differentiate the targets, and to do not cause any collision.

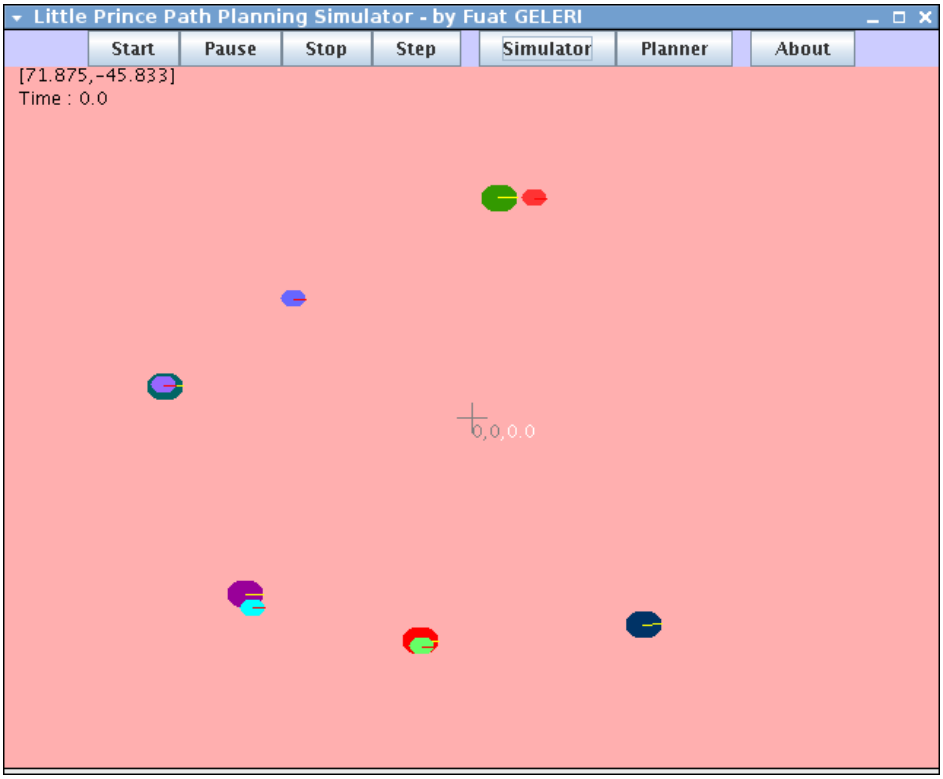

Figure 4.1. Five robots distributed widely

4.4.1.1. Targets Distributed Widely. As seen in Figure 4.1, the robot targets are distributed quite sparsely. According to the statistics, RRT Connect algorithm is found to be the best algorithm for this case.

$$\begin{aligned} Displacement &= Total\ Initial\ Distance - Total\ Final\ Distance \\ \boldsymbol{NRL} &= Total\ Travelled\ Path / Displacement \end{aligned} \tag{4.1}$$

Table 4.1 shows it as the fastest algorithm, that uses memory less than the others. Moveover, if we look at Table 4.1, for the normalized robot path length (NRL) values, whose definition is shown in Equation (4.1), RRT Connect algorithm gives results nearly equal to the length of a direct connection to the targets.

Lazy PRM RRT Connect algorithm also shows good results. Its elapsed time variance is quite low, and it seems as the second algorithm around all. However, if we compare Lazy PRM RRT Connect with Lazy PRM and PRM, which are the multiple query algorithms, Lazy PRM RRT Connect algorithm shows improvements.

Table 4.1. Statistics for Five Robots Coarse Case

Algorithm	Suc. Rate	Elapsed Time	NRL	Node Count
Lazy PRM	100	375.23/∓307.25	1.65/∓0.57	300
LP.RRTCon.	100	320.31/∓144.34	1.28/∓0.45	132.95/∓0.22
PRM	100	1116.74/∓7465.12	1.48/∓0.43	1000
Rbot	100	323.43/∓70.11	1.59/∓0.18	0
RRT	100	7180.37/∓276.31	1.18/∓0.22	1001
RRT Connect	100	216.72/∓81.14	1.06/∓0.11	21.06/∓9.54

Figure 4.2 shows the taken path when the Rbot algorithm, and Figure 4.3 shows the taken path when the RRT Connect algorithm is used.

Rbot algorithm uses no sample nodes or edges, so we assume it as using nearly no memory. For the robots with limited memory, it should be the main option. In fact RRT Connect also used little memory for this case. Rbot shows 100 percent success, but its average NRL is a little higher than RRT Connect's results. This is because of the repulsive potential function which it uses to escape from colliding. To escape from colliding it takes a little longer paths. Moreover, when we look at the behaviour of the robots we see that the robots could not reached to the maximum speed at any time. When the robots get nearer to their targets, their speed even gets too vanishing. However, in RRT Connect like algorithms, because the path is found initially, each robot tries to follow the path as fast as possible, so uses maximum velocity. This is one other reason for the elapsed time of the Rbot algorithm is higher than some others.

4.4.1.2. Targets Distributed Near. As the targets get nearer, and the count of robots increases, we wait Rbot algorithm to start to be the best choice. For five robots and near targets, still sampling based algorithms give better results than the Rbot algorithm as shown in Table 4.2.

Again to save the robots from collision, Rbot algorithm prevented robots to reach

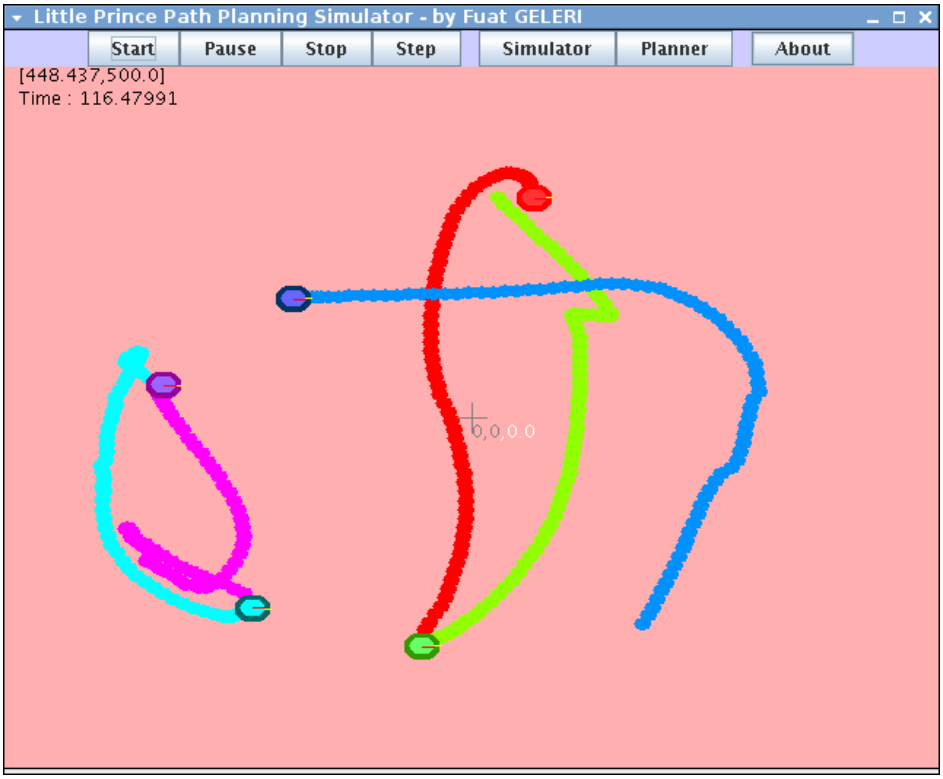

Figure 4.2. Five robots coordinated with Rbot algorithm

their maximum velocity. If we choose a small value for the maximum velocity Rbot algorithm may be the best choice in this test too.

4.4.1.3. Targets Distributed Tightly. If we look at Table 4.1, Table 4.2 and Table 4.3, we see that Rbot algorithm gives quite near results. The elapsed time, success rate, and variance of elapsed time seem quite near to each other. So for the five robots case we see that Rbot algorithm is not affected by the tightness of the targets very much.

Again RRT Connect algorithm gives the best results. Table 4.2 shows that when the targets get nearer RRT algorithm fails in finding a collision free path. Figure 4.4 shows RRT algorithm as the worst one. Of course the success rate of RRT algorithm can increase as the number of samples are increased, because these algorithms are generally probabilistically complete.

RRT Connect algorithm did not highly affected by the adverse affect of getting

Table 4.2. Statistics for Five Robots Normal Case

Algorithm	Suc. Rate	Elapsed Time	NRL	Node Count
Lazy PRM	100	359.82/∓342.88	1.68/∓0.67	300
LP.RRTCon.	100	263.16/∓94.65	1.43/∓0.65	121.01/∓0.1
PRM	100	1147.44/∓6865.06	1.77/∓0.77	1000
Rbot	100	311.21/∓94.83	1.79/∓0.26	0
RRT	100	7089.9/∓172.71	1.16/∓0.19	1000.98/∓0.14
RRT Connect	100	210.65/∓105.9	1.07/∓0.09	20.78/∓11.07

Table 4.3. Statistics for Five Robots Tight Case

Algorithm	Suc. Rate	Elapsed Time	NRL	Node Count
Lazy PRM	100	1240.2/∓2409.78	1.65	675.86/∓182.51
LP.RRTCon.	100	344.95/∓148.66	1.28	126.1/∓0.3
PRM	100	1306.56/∓7026.11	1.48	1000
Rbot	100	354.69/∓71.4	1.77/∓0.18	0
RRT	38	7214.26/∓478.27	1.18	1001
RRT Connect	100	255.77/∓118.12	1.06	27.41/∓14.5

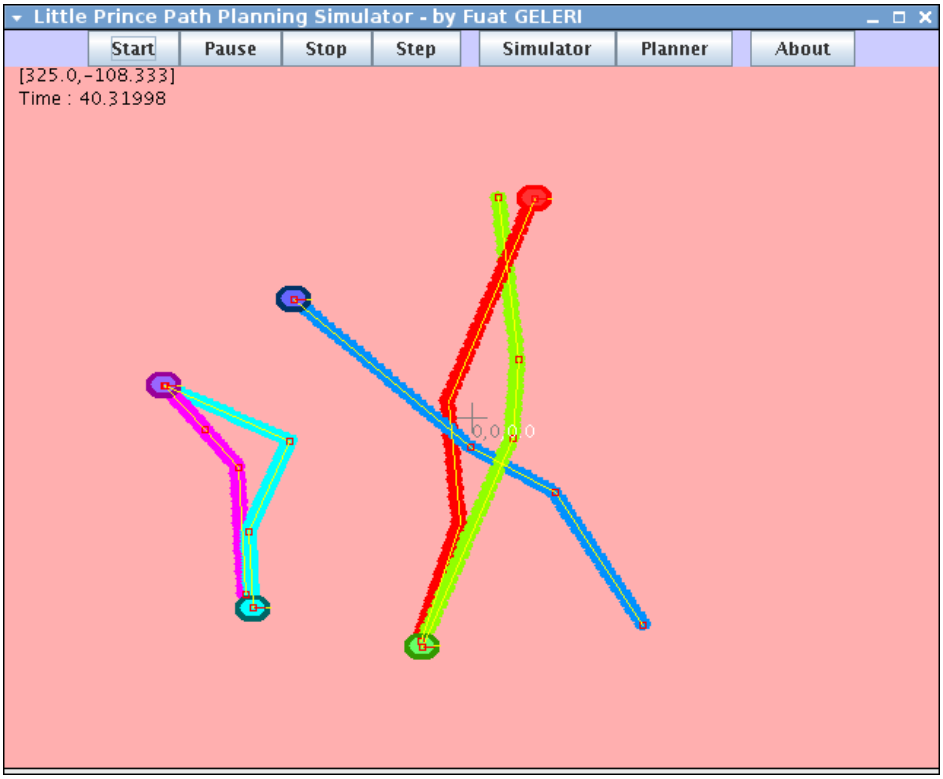

Figure 4.3. Five robots coordinated with RRTConnect algorithm

targets nearer for the five robots case. We will further inspect the behaviour of the algorithm for more robots.

4.4.2. 10 Robots

For some algorithms, these tests will be like stress testing. The algorithms will try to search a universe with three times ten, thirty dimensions. If we calculate the space, $(1000x1000x6.28)^{10}$ is the size of the volume to be searched.

We do not want to reach every part of this volume, but only test the algorithms to see their success rates when three thousand samples are used for the planning. Space filling algorithms like PRM and Lazy PRM is expected to give bad results, because the space is quite big to be filled sufficiently. We expect aim based algorithms, like single query algorithms to show better results. Moreover, because the dimension of the configuration is increased to quite high values, and we are using centralized algorithms, we wait Rbot algorithm to come the best one, which is working like decentralized

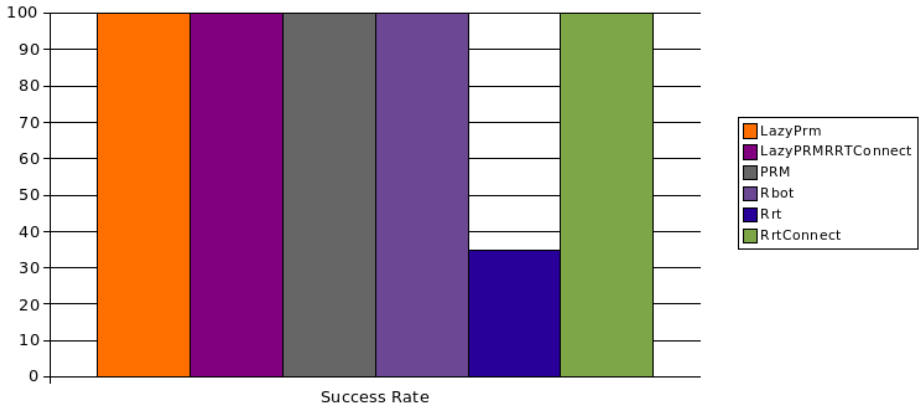

Figure 4.4. Success rates for five robots tight case

algorithms.

4.4.2.1. Targets Distributed Widely. As seen in Figure 4.5, the targets of the robots are quite far from each other. Especially for the cases that the robots or the targets are near to each other, connecting them to the generated roadmap is difficult. However, in this world the requirement is filling the space adequately. The targets may seem far from each other, but they represent really some quite small volume if we think the volume to be searched. The algorithms should reach this quite small volume, and should be able to coordinate the robots so well that in this volume robots will not collide too.

Table 4.4. Statistics for 10 Robots Coarse Case

Algorithm	Suc. Rate	Elapsed Time	NRL	Node Count
Lazy PRM	0	15852.05/∓19574.96	N/A	2983.02/∓49.64
LP.RRTCon.	22	27718.14/∓41800.26	4.02/∓1.83	3097.32/∓466.88
PRM	13	2170.15/∓589.24	3.99/∓1.34	3000
Rbot	100	1487.04/∓256.19	1.8/∓0.16	0
RRT	38	90941.74/∓2946.81	1.69/∓0.29	2987.89/∓11.62
RRT Connect	98	5653.99/∓5196.19	1.3/∓0.21	216.18/∓217.3

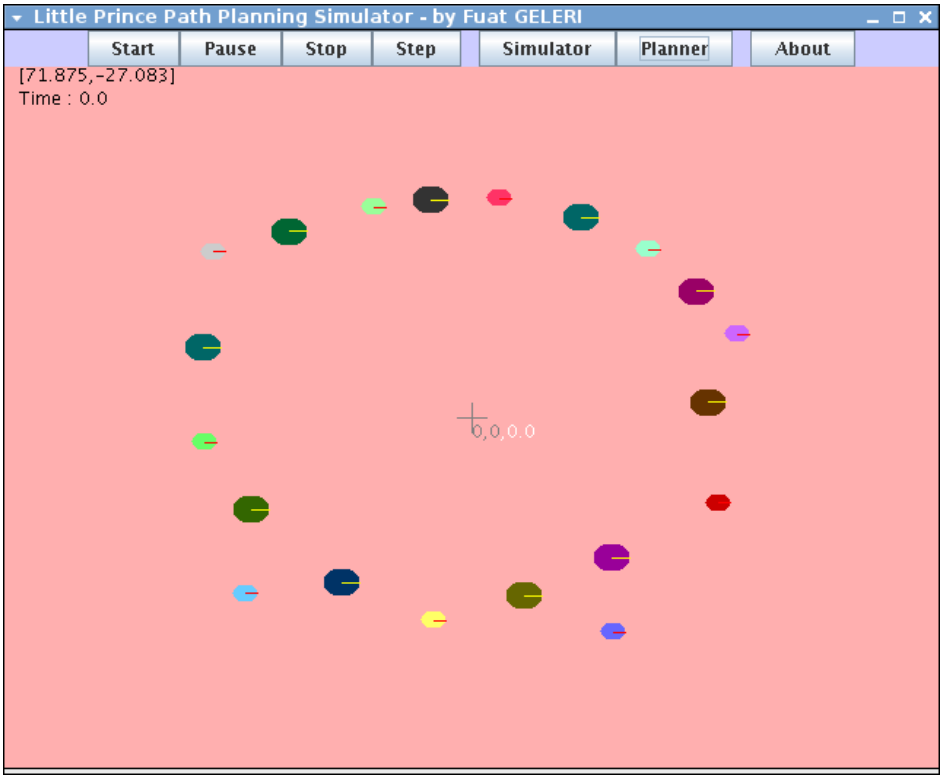

Figure 4.5. Ten robots distributed sparsely

RRT Connect algorithm and Rbot algorithm shows quite similar results for the coarse case, as seen in Table 4.4. When we look at Figure 4.6, we see that the RRT Connect algorithm collects the robots to the center and distributes them from there. However, Rbot algorithm uses the space, and does not make the robots to stuck in the center as seen in Figure 4.7. This is the reason of implementing an algorithm named Lazy PRM RRT Connect. This algorithm first generate an initial roadmap by Lazy PRM algorithm, and uses this roadmap for global path finding. After global path is found, it searches for the local path by RRT Connect algorithm. Namely, the bad effect of RRT Connect, collecting robots to the center, because of making aim based searches, is eliminated with the help of an algorithm which is not aim based, but aims to fill the space adequately. The result of Lazy PRM RRT Connect algorithm can be seen in Figure 4.8 for comparison.

4.4.2.2. Targets Distributed Near. When the targets of the robots get nearer, the space to be reached is became smaller. So, the algorithm should fill the full configura-

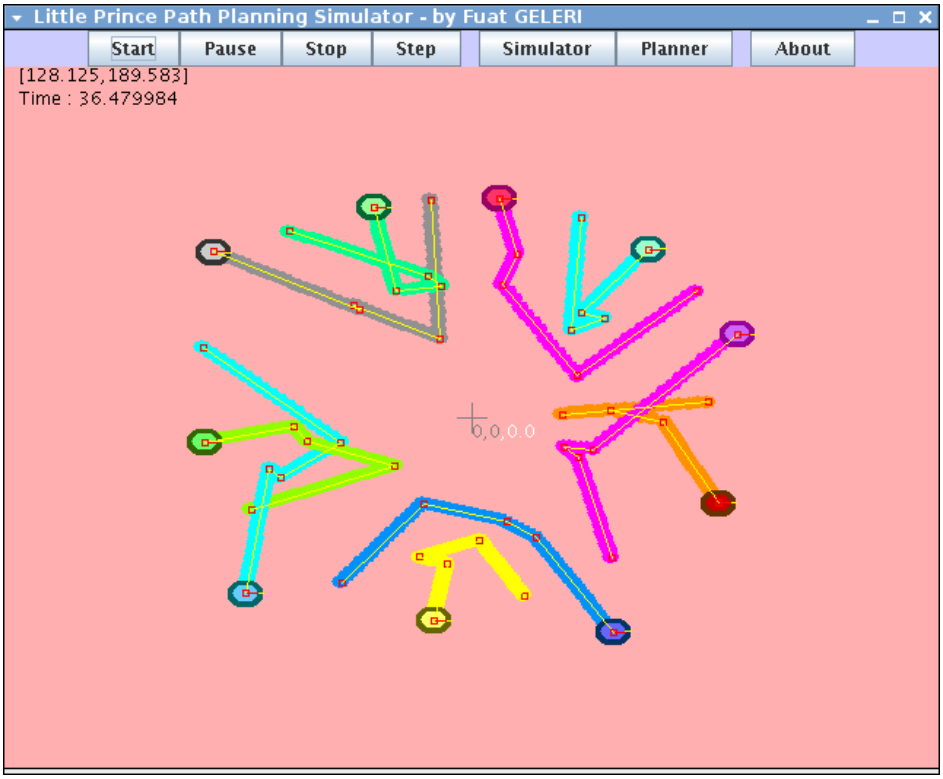

Figure 4.6. RRT Connect algorithm is used to coordinate 10 robots in no obstacle universe

tion space more tightly. However, when this tiny space is reached, the second search will be easier. This is why Lazy PRM RRT Connect algorithm gives better results than the coarse case. The results are given in Table 4.5. We will inspect the capabilities of Lazy PRM RRT Connect next, in another tests, and more information about the increase in the success of this algorithm will be given next.

4.4.2.3. Targets Distributed Tightly. When the targets are placed more tightly, the space to reach becomes too little as compared to the total volume of the configuration space. So, single query algorithms like RRT algorithm fails, like PRM and Lazy PRM algorithm fail.

RRT Connect algorithm still gives good results, and this shows its powerful nature. As a surprize Lazy PRM RRT Connect algorithm gives the best results. Combining a multiple query algorithm, Lazy PRM, with RRT Connect algorithm made

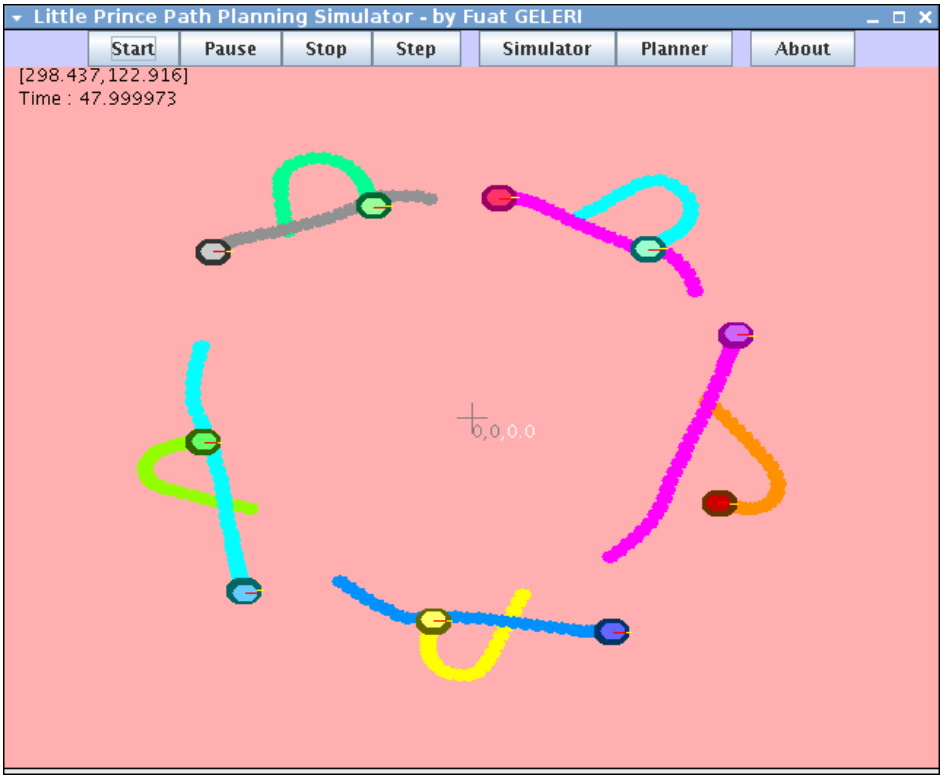

Figure 4.7. Rbot algorithm is used to coordinate 10 robots in no obstacle universe

the algorithms more powerful. Because Lazy PRM algorithm generates the initial roadmap, the RRT Connect algorithm to be used next is forced to make their searches in a more widely manner. This gave the algorithm the chance of beating RRT Connect algorithm in the amount of success, as seen in Figure 4.9.

In all the cases, Rbot algorithm gave similar results for the elapsed time amount.

Lazy PRM RRT Connect algorithm is the best algorithm according to Table 4.6. RRT Connect algorithm is defeated for the first time at the tests until this point. RRT Connect algorithm makes sampling to find the path in the shortest time amount, so the samples are generated as biased toward the goal. When multiple robots are used, to make the coordination of the robots easier, distributing the robots a little more to the environment may give more place to maneuver. So, Lazy PRM RRT Connect algorithm both gives robots the change to use the environment widely, and also go to the deeper levels by using RRT Connect algorithm. Lazy PRM RRT Connect algorithm localizes and solves the problem, with RRT Connect algorithm. When the

Table 4.5. Statistics for 10 Robots Normal Case

Algorithm	Suc. Rate	Elapsed Time	NRL	Node Count
Lazy PRM	10	359.82/∓342.88	1.68/∓0.67	2952.5/∓27.44
LP.RRTCon.	89	263.16/∓94.65	1.43/∓0.65	1484.65/∓231.36
PRM	0	7202.94/∓61712.92	1.77/∓0.77	3000
Rbot	100	1837.3/∓443.33	2.02/∓0.21	0
RRT	15	89271.53/∓2249.18	1.16/∓0.19	2989/∓8.27
RRT Connect	98	6151.89/∓5145.74	1.07/∓0.09	246.16/∓211.67

Table 4.6. Statistics for 10 Robots Tight Case

Algorithm	Suc. Rate	Elapsed Time	NRL	Node Count
Lazy PRM	0	15701.06/∓23815.55	N/A	2978.11/∓45.7
LP.RRTCon.	98	31881.39/∓39886.97	1.28	126.1/∓0.3
PRM	0	7508.64/∓63898.5	N/A	3000
Rbot	100	1967.47/∓357.29	2.13/∓0.23	0
RRT	0	89410.26/∓2331.36	N/A	2988.97/∓9.25
RRT Connect	86	9445.63/∓5742.86	1.06	27.41/∓14.5

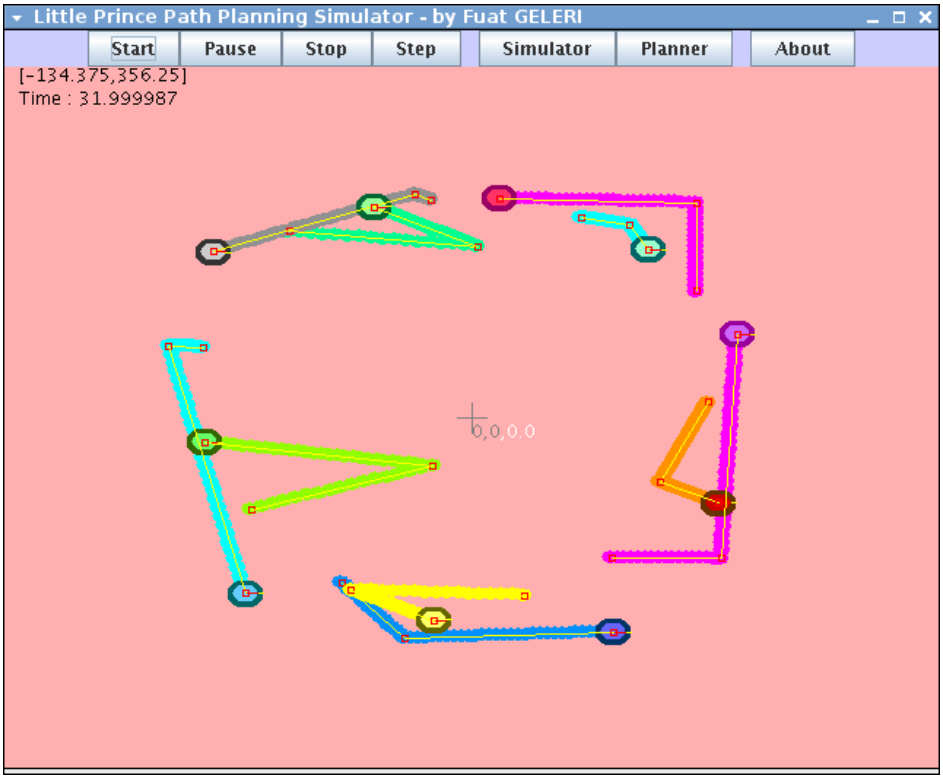

Figure 4.8. Lazy PRM RRT Connect algorithm is used to coordinate 10 robots in no obstacle universe

targets are nearer, because we use RRT Connect with quite low number of nodes, the Lazy PRM RRT Connect gives better results. However when the robots are distributed to the environment the algorithms turns to Lazy PRM and the help of RRT Connect is diminished.

4.4.3. Discussion

APF algorithm uses potential function to navigate the robots through their destinations. At each step it recalculates the control inputs, velocities. Therefore there is no plan beforehand. If the environment contains no local minima, because of this reactive nature, the algorithm's success is shown to be high. In fact, it iteratively searches the world, biased toward the goals. However, sampling based algorithms makes this search beforehand. They not only calculate the next input but the whole path.

When the dimension of the configuration space gets high aim oriented searching

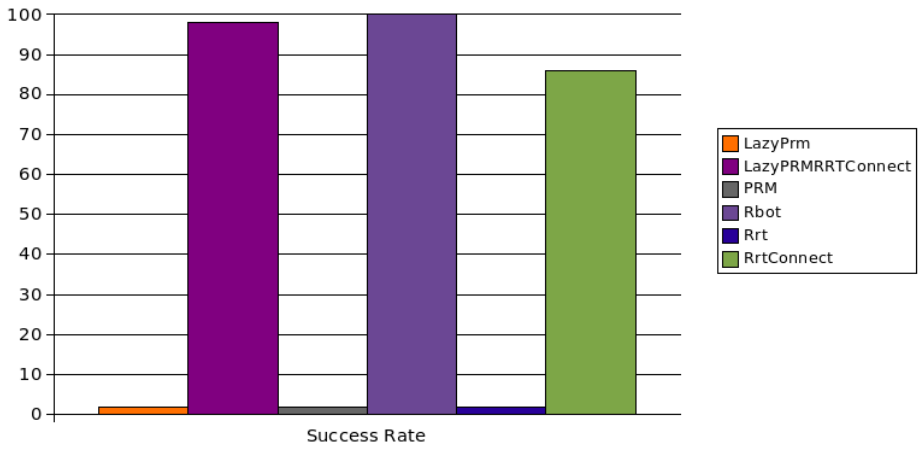

Figure 4.9. Success rates for 10 robots tight case

gives better results, because the space is too big to be filled adequately. However, as Lazy PRM RRT Connect algorithm shows that generating an initial roadmap, and using aim oriented algorithms in the local searches gives better results in especially higher dimensions. The roadmap generated by the Lazy PRM part of the algorithm may be thought as the main road ways, and the ones generated by the RRT Connect part can be seen as the sub-roads. Generally main roads do not need to have big changes. Instead of using a pre-generated main road, making the search for the main roads too in each query is only loss of time. It also decreases the possibility of success. Therefore, Lazy PRM RRT ConnectAlgorithm is shown to be better for the high dimensional cases.

Lazy PRM RRT Connect algorithm is inspected in more detail next. Furthermore, changing the parameters of RRT Connect algorithm in this algorithm is tested.

4.4.4. More than 10 Robots

We increased the number of robots and compared the best two algorithms mentioned above. The RRT Connect algorithm and RboT algorithm is tested for 20, 30, 40 and 50 robots. For the RRT Connect algorithm the count of samples is kept limited to 100000.

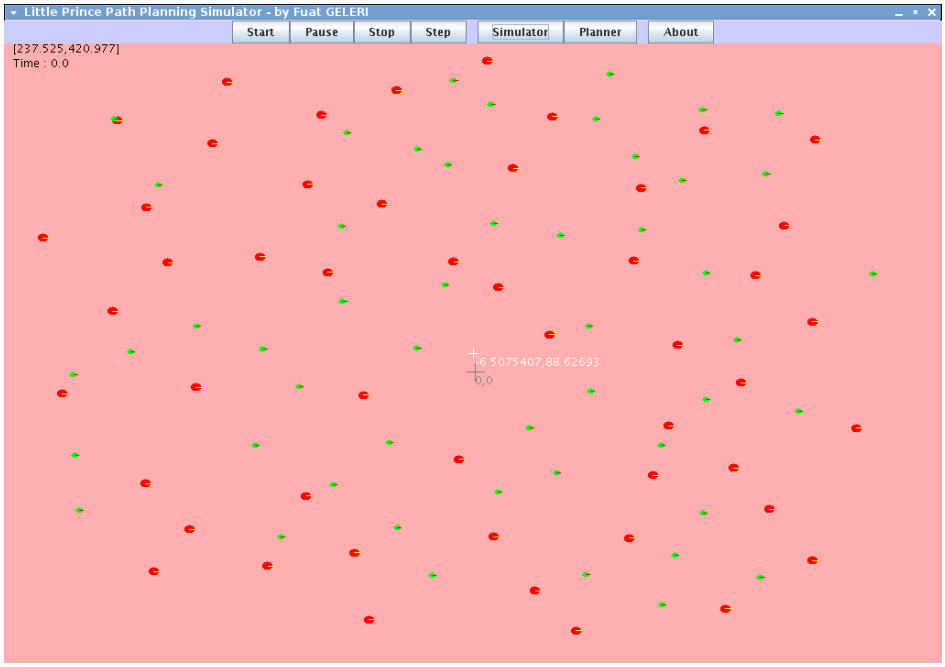

Figure 4.10. 50 robots in a circular world

In these tests the RRT Connect algorithm is run in centralized manner. Namely the configurations of the robots are summed up to achieve one big configuration, that represents the configuration of all of the robots. When N robots are used in the tests, the volume of the search space can be calculated by the Equation (4.2) as;

$$volume = (length(x) * length(y) * length(\theta))^N \tag{4.2}$$

Equation (4.2) shows that the growth of the volume of the search space is expontential with respect to the number of robots. For the case of the size of the x axis is equal to 1000 unit and y axis is equal to 1000 unit and the amount range length of the angle is 6.28 unit, then the volume of the search space for single robot is equal to 6.280.000 $unit^3$. So, for 20 robots the sampling will be made on a space with a volume of 10^{140}. Multiple query sampling based algorithms aim to fill the space adequately, so that for different queries the robots will find their paths, without touching each other. So, in such a big space their job is quite difficult.

As the number of robots is increased the amount of samples required increases too. We wait RRT Connect algorithm to show good results for the simulations having

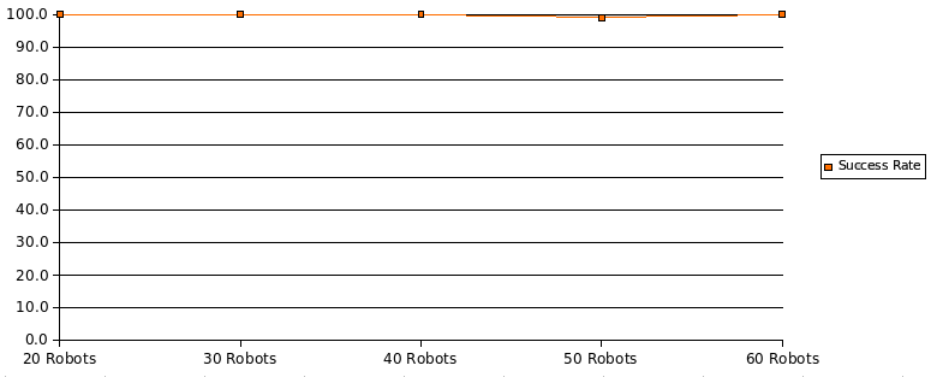

Figure 4.11. Change of the success rates of the RboT algorithm in many robots case

a few robots, and it may start to degrade for 20 and 30 robots cases. For 40 and 50 robots cases the environment will be filled by the robots quite densely as seen in Figure 4.10, and we wait the RRT Connect algorithm not to be very successful for these cases.

4.4.4.1. Success Rates. RboT algorithm is a quite powerful algorithm in robot coordination. It successfully coordinates 250 robots and this is not near to its upper limit.

In our improved version the algorithm generates local groupings and the number of robots in these local groups is already small as compared to the count of the robots in the environment. So the total count of robots will only affect the response time of the algorithm, but will not make the algorithm to fail.

Figure 4.11 shows that, the rate of success is not affected by the count of robots in the environment.

RRT Connect algorithm is a sampling based algorithm. Because these kind of algorithms are probabilistically complete their success is affected by the number of samples used in finding the path. Figure 4.12 shows that the algorithm successfully coordinated 20 and 30 robots, but started to show failures in coordinating 40 robots. As the number of robots increased to 50 and 60 RRT Connect algorithm shows no

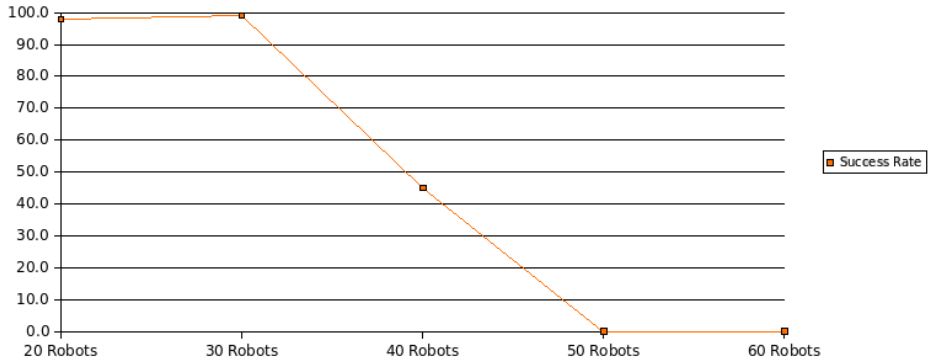

Figure 4.12. Change of the success rates of the RRT Connect algorithm in many robots case

success at all. Increasing the number of samples used and decreasing the step size should make the algorithm be successful for more robots however the amount of time elapsed will be too much for a reasonable application.

When the environment does not contain any local minima, the problem is only the coordination of the robots, and potential function using reactive algorithms are better in these kind of problems.

4.4.4.2. Mean Elapsed Time Amounts. When the number of robots to coordinate is increased the elapsed time is waited to increase exponentially. The power of the exponential function may vary for each algorithm. Some algorithms can be double exponential too.

When we look at Figure 4.13, we see that RboT algorithm gives quite stable results. The mean elapsed time is seen increasing quite linearly till the 50 robots part of the graph. When the robots are coordinated, some robots generate local minimas for some others and this may lead to oscillations. When we inspected the results, the average of the results lead to this graph, however this is because of this oscillations. In fact generally the RboT algorithm coordinated the robots in nearly 9000 milliseconds. Instead of directly taking the average, weighted average would be a better choice in calculating the means.

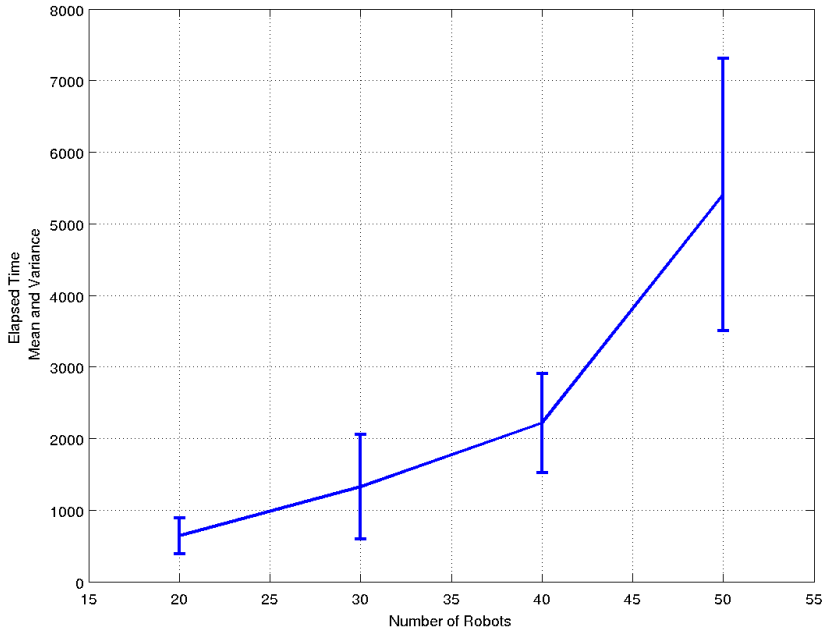

Figure 4.13. Change of the mean elapsed time amount for the RboT algorithm in many robots case

RRT Connect algorithm gave no successful results for 50 robots. We increased the limit of sample count to 200000, but the result observed remains the same. So we put an upper time value to the graph to make the graph more understandable.

In Figure 4.14, the exponential increase of the mean elapsed time can be observed. RRT Connect algorithm can be identified as the best algorithm about the mean elapsed time in finding path between sampling based algorithm, but it also shows this exponential time increase. Therefore, we may conclude that, sampling based algorithms need some adjustments, and additions to successfully cope with coordination of the robots. For RboT algorithm coordination of 40 robots takes only 2.4 seconds however for RRT Connect algorithm it took 1500 seconds.

4.4.4.3. Discussion. RRT Connect algorithm is a single query algorithm that biases its sampling toward the goal configurations, which makes it the quickest path planning algorithm so far. However, as it is the case for all other path planning algorithms for the centralized case the RRT Connect algorithm makes the sampling in a bigger

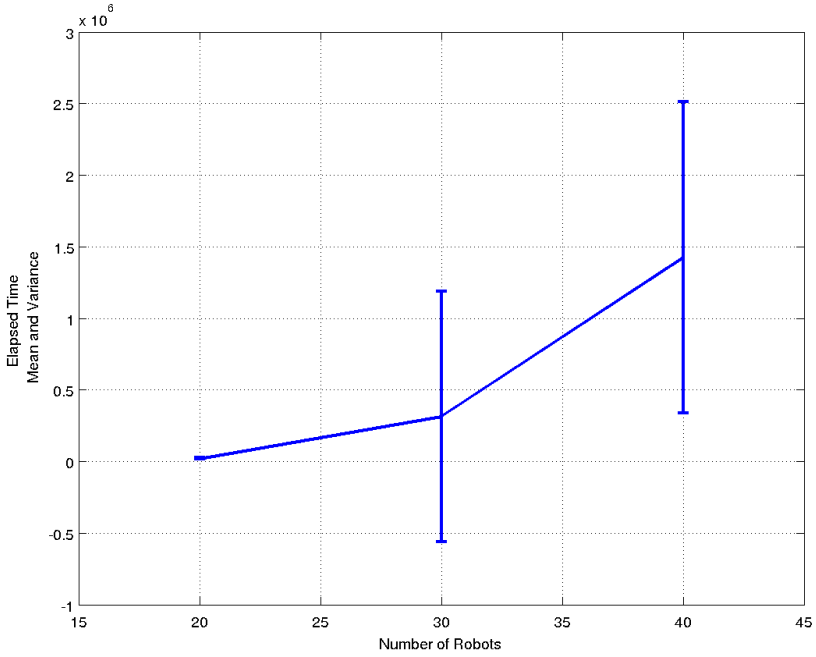

Figure 4.14. Change of the mean elapsed time amount for the RRT connect algorithm in many robots case

configuration space. The sampled configuration is tried to be connected to the tree directly with a straight line. When the count of robots increase too much, connecting a big configuration with yet another newly sampled one gets more and more difficult.

In these tests the algorithms in fact tested for not only finding path, but also coordinating the robots. RboT algorithm uses potential functions to coordinate the robots, and because the environment contains no local minima, it successfully coordinates the robots. Improved version of the RboT algorithm easily manages coordination of 250 robots.

As seen in Figure 4.15, the number of steps taken to arrive to target configurations are bigger for APF algorithm, than RRT-Connect algorithm. This is because of that RboT algorithm leads the robots to take too little steps, however RRT Connect algorithm wants the steps to be some big, which leads collisions too much. The step size of the RRT-Connect algorithm may also be decreased considerably. However, as this time, the number of nodes in the tree will also be very high too. Moreover, the

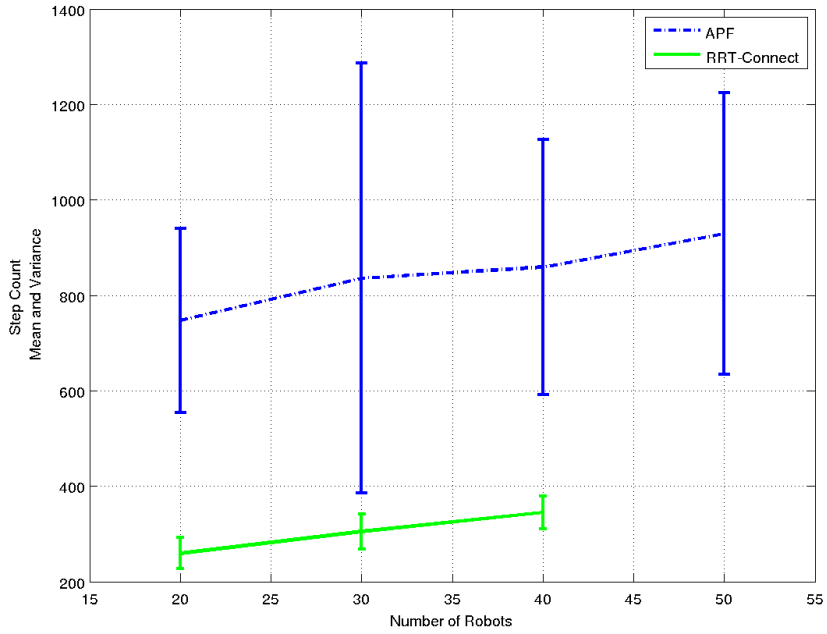

Figure 4.15. Count of steps taken by APF and RRT-Connect algorithm for different robot counts

normalized robot path length (NRL) for APF (RboT) algorithm is bigger than the values for RRT-Connect algorithm as seen in Figure 4.16.

In fact when we look at Figure 4.10, it is already difficult to find a big configuration that is collision free. So connecting a collision free configuration for the case is quite more complex.

4.5. Static Obstacle (Maze Problems)

In this test we will compare the algorithms Lazy PRM RRT Connect, RRT Connect, and Lazy PRM for the robots in a maze. As shown in Figure 4.17, we have a maze with four rooms and a blocked corridor.

The test is done for various robot counts. Environment only contains static obstacles, but no dynamic obstacle in these tests. Each test is taken for 100 times.

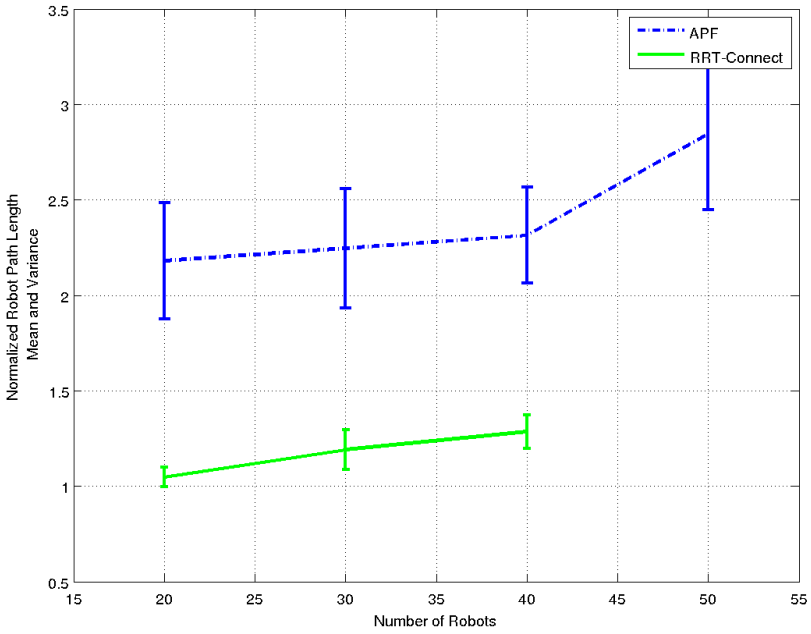

Figure 4.16. NRL results of APF and RRT-Connect algorithm for different robot counts

4.5.1. One Robot

In this test, the algorithms are tested for their speed in responding queries, and the length of the found path. Because we have only one robot to coordinate, the search will be in a configuration space with only three dimensions. Namely, space filling algorithms can easily fill the space adequately for low numbers of random samples as seen in Figure 4.18.

We wait all of the algorithms to find the path, and coordinate the robot successfully for each case. However the amount of used memory, namely the number of samples and edges, and the response time may make the algorithm distinguished. This test is important for especially decentralized path planning. In decentralized planning, all the robots will plan path for itself only. Namely, it is a single robot path planning case. Therefore, the algorithm found as the best in here is highly possible to come to be the best for the decentralized cases for multiple robots too.

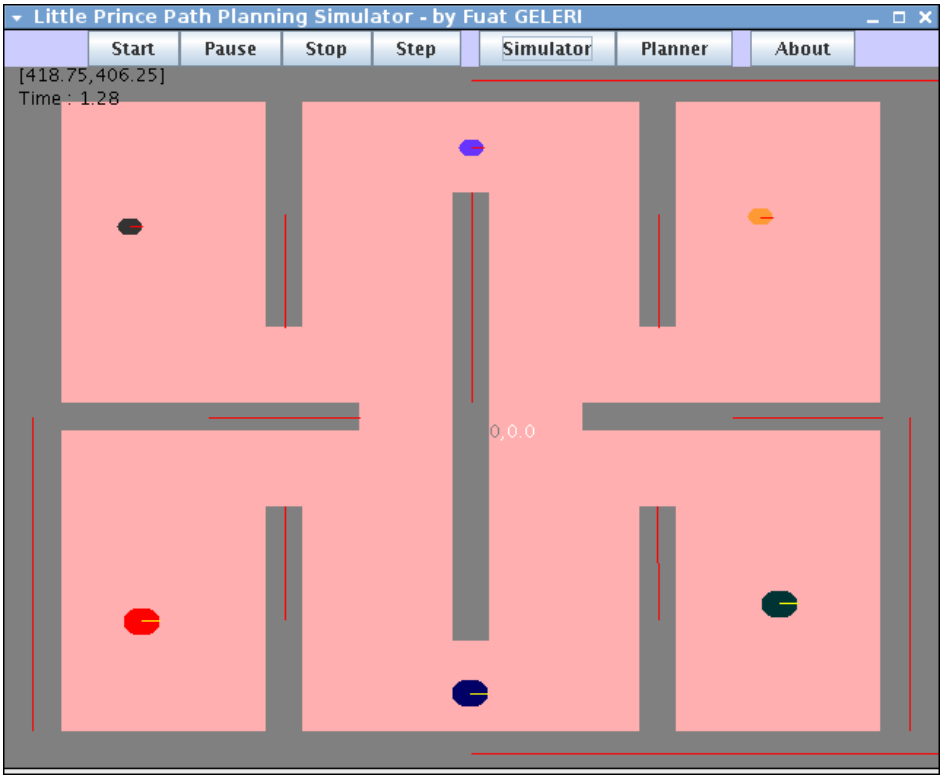

Figure 4.17. Maze with four rooms and a blocked corridor

As seen in Figure 4.19, the elapsed time result of Lazy PRM RRT Connect algorithm is nearly equal to Lazy PRM. Namely, extending the algorithm does not add too much increase to the calculations. Morever, this is valid for variance values too.

Because, for the one robot case, Lazy PRM can fill the space adequately with low numbers of samples, Lazy PRM algorithm is the best algorithm for the single robot case for the average time used for each planning. In fact, Lazy PRM algorithm continues making collision checks while answering each query. However, if we have a chance to make preprocessing, we can make all the preprocessing beforehand too. So, if we make the collision checks at the initialization part, the algorithm may be quicker in the query step. The algorithm, which makes the collision checking of edges at the initialization step is the PRM algorithm.

We propose the PRM algorithm to be used in spaces with low dimensions. If precomputation time is important, single query algorithms may be advised, like RRT Connect. If the targets are near to each other generally, Lazy PRM RRT Connect

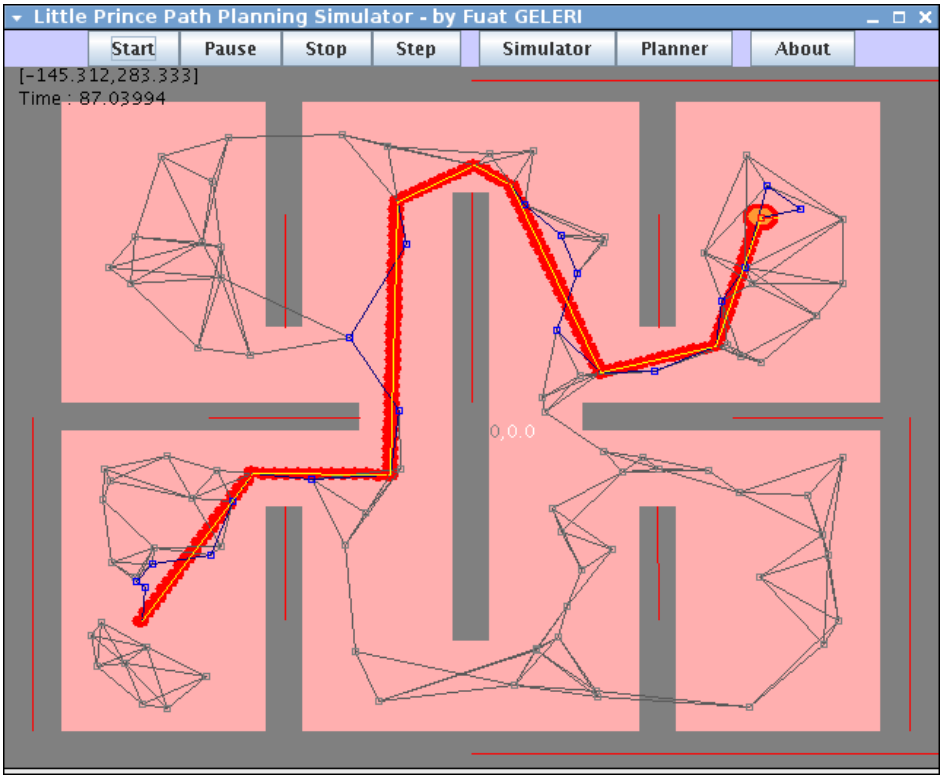

Figure 4.18. Lazy PRM fills the space adequately with only 100 samples

algorithm will add not much extra computation to Lazy PRM algorithm.

Table 4.7. NRL and memory usages for one robot in the maze

Algorithm	NRL	Node Count	Edge Count
RRT Connect	1.57/∓0.98	13.59/∓24.66	11.59/∓24.66
LazyPrm	1.33/∓0.54	688/∓68.23	4182.02/∓478.71
LP.RRTCon.	1.56/∓0.72	302.53/∓19.86	1640.27/∓113.54

Normalized robot path length, NRL, is the ratio of the taken path by the robot to the euclidian distance to the target. If the NRL value is high, it means the robot took a longer path. As the samples in the configuration space high, graph search algorithm can find the shorter paths. In Table 4.7 we see that Lazy PRM algorithm has 1.33 as the NRL value. This value is the minimum of NRL values. This is highly because of the count of nodes the Lazy PRM uses is high. RRT Connect algorithm only uses

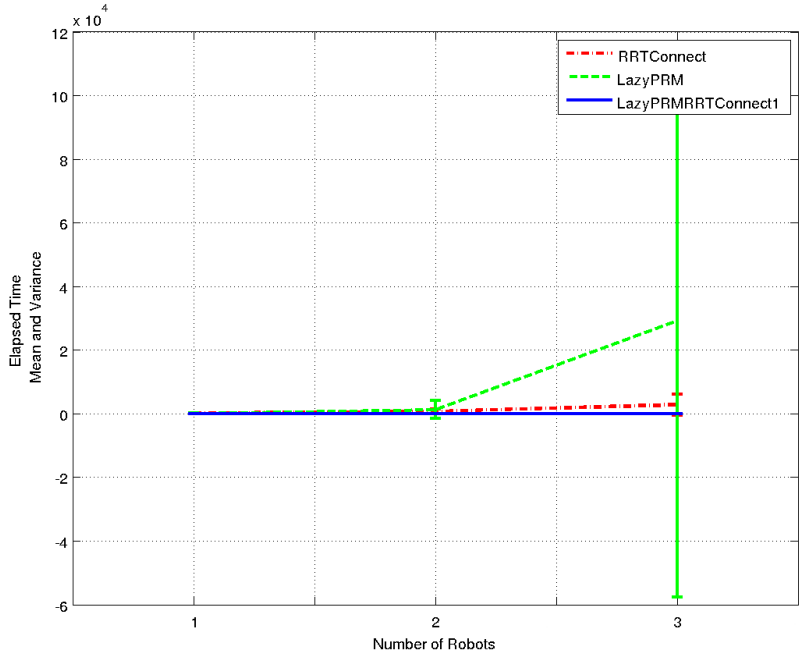

Figure 4.19. Amount of time used for path planning by Lazy PRM, RRT Connect and Lazy PRM RRT Connect algorithms in the maze

13.59 nodes in average. So RRT Connect algorithm uses minimum amount of memory. Lazy PRM RRT Connect algorithm uses less memory than Lazy PRM algorithm and its average NRL value is less than RRT Connect. Namely Lazy PRM RRT Connect gets the good properties of both of the algorithms.

4.5.2. Two Robots

In this test we used two robots, which increases the dimension of the configuration space to 6. This increases the difficulty of the path planning. We wait RRT Connect algorithm to handle this easily. However, for the space filling algorithm, Lazy PRM, this will mean doubling the space to fill. Lazy PRM RRT Connect algorithm will try to ease the adverse effect of dimension increase, and we expect it to decrease the amount of samples to an acceptable amount.

As seen in Figure 4.20, the algorithms gives nearly the same success rates for two robots. However, Lazy PRM RRT Connect algorithm gives the best average time

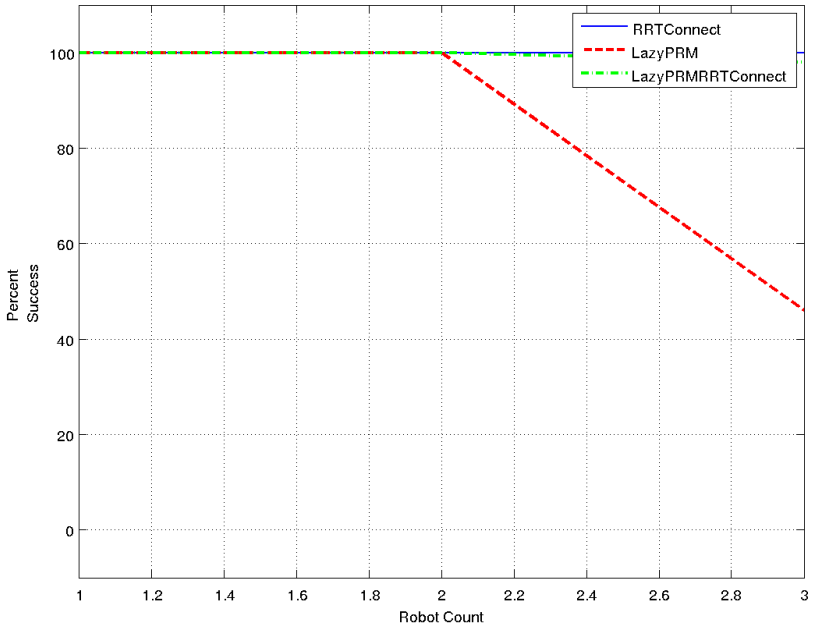

Figure 4.20. Success rates for path planning by Lazy PRM, RRT Connect and Lazy PRM RRT Connect algorithms in the maze

usage. RRT Connect algorithm's average time usage is a little higher but variance of time usage is minimum for this algorithm.

Table 4.8. NRL and memory usages for two robots in the maze

Algorithm	NRL/Variance	Node Count	Edge Count
RRT Connect	2.19/∓1.08	22.91/∓18.14	20.91/∓18.14
LazyPrm	2.95/∓2.07	1836.84/∓1545.73	11485.54/∓11897.76
LP.RRTCon.	3.9/∓3.21	691.06/∓74.71	3092.26/∓319.11

Table 4.8 shows that for the RRT Connect algorithm the average NRL value increases from 1.57 to 2.19. For Lazy PRM algorithm the average NRL value increases from 1.33 to 2.95, and for Lazy PRM RRT Connect it increases from 1.56 to 3.9. The increase in the NRL value is minimum for RRT Connect algorithm. This is due to the nature of the RRT Connect algorithm, it is aim based. The paths that are results of the Lazy PRM algorithm have higher increase in NRL value. This means that Lazy PRM

algorithm makes the robots use the open area of the configuration space more. Aim based searches makes the robots to gather on the center and be distributed from there, but Lazy PRM algorithm uses the configuration space better. So combining these two algorithms may lead to better paths, which neither makes the robots to gather, nor increases the sample count too much.

4.5.3. Three Robots

In this test there are three robots to coordinate, which increases the dimension of the configuration space to 9 dimensions. As seen in Table 4.9 the success rate of the Lazy PRM algorithm really dropped by a high amount to 48 percent. For high dimensional problems space filling algorithms need too many sampling to cope with. This problem is solved by Lazy PRM RRT Connect. It gives 99 percent success and uses less count of samples in average.

Table 4.9. Statistics for three robots in the maze

Algorithm	Suc. Rate	Elapsed Time	Variance
RRT Connect	100	3600.41	3951.25
LazyPrm	48	34219.48	145656.11
LP.RRTCon.	99	26497	147712.77

Table 4.10. More statistics for three robots in the maze

Algorithm	NRL/Variance	Node Count	Edge Count
RRT Connect	2.42/∓1.17	65.93/∓82.51	63.93/∓82.51
LazyPrm	3.47/∓1.85	10061.88/∓4.95	66017.21/∓2975.12
LP.RRTCon.	8.33/∓5.06	4503.48/∓900.55	17076.88/∓4137.88

RRT Connect algorithm gives the best paths with minimum amount of node and edge usage. Lazy PRM RRT Connect algorithm gives the longest paths, however

decreases the amount of used samples, and edges in Lazy PRM, as shown in Table 4.10.

4.5.4. Discussion

Lazy PRM RRT Connect algorithm is a combination of Lazy PRM algorithm with RRT Connect algorithm. The property of this algorithm is that, it combines a multiple query algorithm with a single query algorithm. Multiple query algorithms make their most of the calculations in the preprocessing time. Their preprocessing time is quite high, but when successfully optimized the query times are quite low. Single query algorithms make no preprocessing or quite low preprocessing, but makes most of the job in the query step. So when compared to query step of multiple query algorithms, single query algorithms are waited to give bigger elapsed time amounts per query.

Multiple query algorithms generally generate a roadmap of the environment and use this roadmap in the query step. If the roadmap is not enough for the environment than most of the queries will be unanswered. Combining a single query algorithm with a multiple query algorithm solves this problem. The roadmap may not be dense enough for the case, however the RRT Connect algorithm in here focuses to the start and end points and leads the robots till connecting them to the global roadmap. This is like the approach of quadtrees, divide the area requiring more processing more than the areas those are clear.

Lazy PRM RRT Connect algorithm is an improved version of the Lazy PRM algorithm that shows better results in both the amount of used memory and time. As seen in Figure 4.21 the path taken by robots are bigger than the paths generated by Lazy PRM and RRT-Connect. In fact RRT-Connect algorithm leads robots to get too near to each other, and this algorithm also solves this problem. Like APF algorithm, it uses the free space effectively, and generates safer paths.

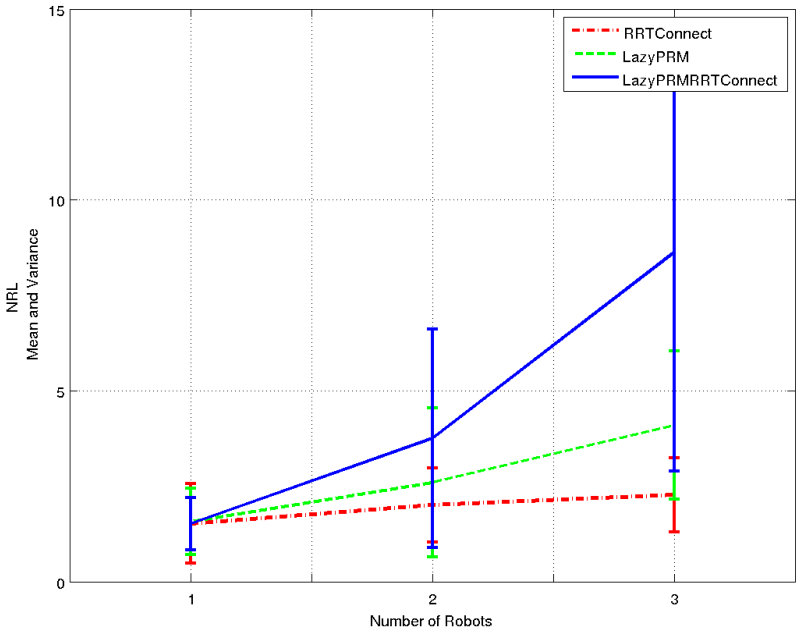

Figure 4.21. Normalized Robot Path Length (NRL) values for path planning by Lazy PRM, RRT Connect and Lazy PRM RRT Connect algorithms in the maze

4.6. Dynamic Environments

Until this point path planning algorithms are tested for stationary environments, those have static obstacles, or no obstacle at all. Moreover, the algorithms were always centralized. Therefore, when a path is calculated, there will be no factor that will make the correctness of the path questionable. However, in real life generally robots will wanted to be used in the environments those may include moving obstacles. If the enviroment includes dynamic obstacles, then the used algorithms must cope with this situation. Generally a simple escape algorithm will be enough for saving the robots. However, for a more advanced solution the controller should also have capability of replanning of the path.

Next, we will test RRT Connect, Lazy PRM and Lazy PRM RRT Connect algorithms in an enviroment that contains both static and dynamic obstacles. We will increase the count of random obstacles and observe the affects. Both centralized and decentralized versions of these algorithms will be tested.

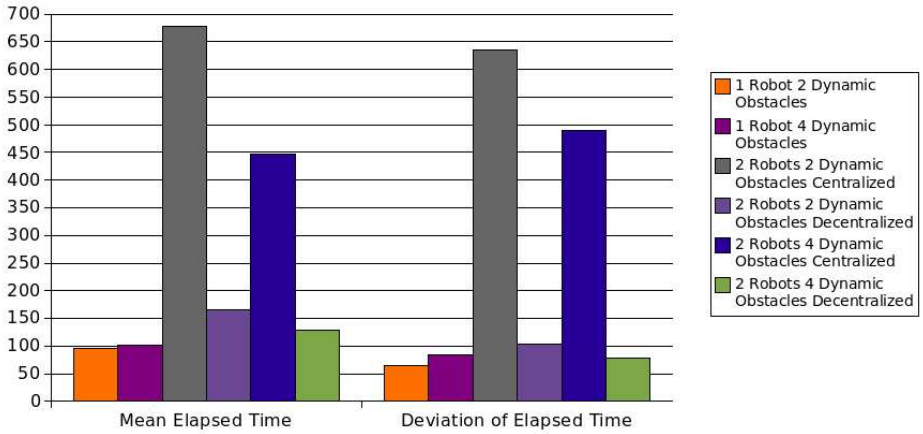

Figure 4.22. Change of elapsed time amounts for RRT Connect in various dynamic environments

We wait when the centralized approaches are applied the amount of time to be multiple of the amount of time used by the decentralized approaches. Moreover, Lazy PRM RRT Connect algorithm is waited to decrease the time usage, and increase the success rates of Lazy PRM algorithm.

As seen in Figure 4.22 when decentralized approaches are used the amount of time increase is neglectable. The time varies between 100 and 150 milliseconds for one robot with no obstacle to two robots with four obstacles case. However, when centralized approach is applied then the elapsed time amount shows four to five times more than the their decentralized ones.

Decentralized approaches may fail to find paths those can be found by centralized approaches, however if the environment contains dynamic obstacles the elapsed time a decentralized approach consumes is much less than a centralized approach.

When we look at Figure 4.23 and Figure 4.24, the case is the same. The amount of elapsed time for centralized approaches is multiple of their amount in the decentralized approaches.

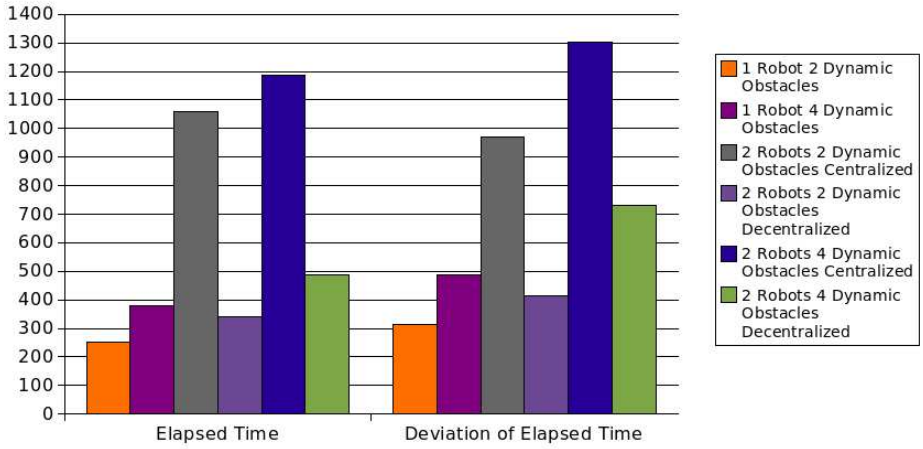

Figure 4.23. Change of elapsed time amounts for Lazy PRM in various dynamic environments

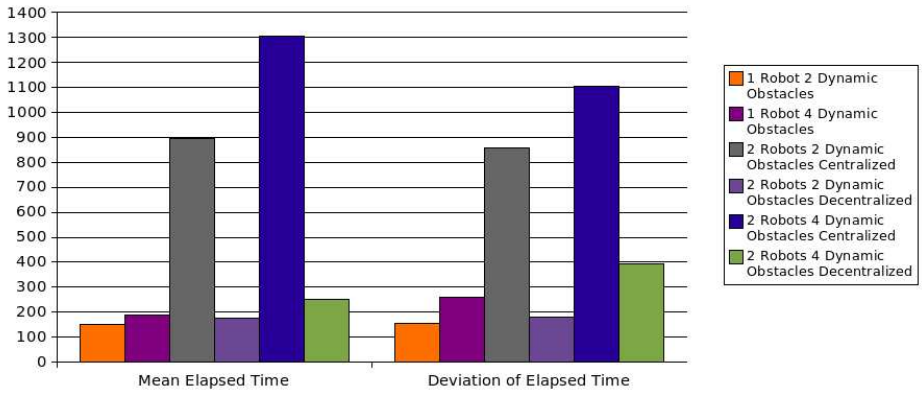

Figure 4.24. Change of elapsed time amounts for Lazy PRM RRT Connect in various dynamic environments

4.6.1. Discussion

Gradient-descent algorithms are very powerful in coordination of the robots. They coordinate the robots such that they incrementally reach to their goals without colliding to each other. In an environment with many robots, coordination of the robots with deliberative path planning algorithms would take high amount of time, but with such algorithms the robots will be coordinated with just a few calculations.

Deliberative path planning algorithms coordinate the robots before they start to move. Because this coordination map is generated beforehand, the robots know

position of each other when they move. However, in reactive algorithms the position of the robots are now known prior. So the robots should have some distance to the others, to save themselves from collision. In some places to save from collision the path taken by the robots might be enlarged and this enlargement may be not needed at all.

At the start of this document we divided the environments to four according to the difficulty of path planning in these environments. In the first case we know the enviromnent fully, and there were no dynamic obstacles. In this case the path planning algorithms can find paths that is shorter than the potential function approaches. However, in the second case there are moving obstacles in the environment. In such a case we do not need to alter the formula of potential function at all. Already each robot sees others as moving obstacles, so no need for such an update in the formula. However in deliberative path planning algorithms, the controllers should check the enviroment at each step to see whether there is any reason thay may violate the path to be followed. If there is an obstacle in front of the robot, than robot should make some extra planning to escape from this moving obstacle. So adding dynamic obstacles to the environment brings more difficulty to deliberative algorithm but not much difficulty to reactive algorithms.

Moreover, if the environment is partially known the problem is getting more difficult. To make a full deliberative plan, we should know the environment exactly. Furthermore, some algorithms need also the position of dynamic obstacles at each time step to make a plan. However this is not a case for potential function using algorithms. Already they do not make any prior plan. They just go toward their goal incrementally and escape from the obstacles as they appear. Also, if the environment contains dynamic obstacles, as it is partially known, which is the case four, the potential function approach will be able to coordinate the robots in this environment without any addition too.

As the problem of each gradient-descent approach in any field of artificial intelligent, local minima is a problem for potential function using path planning algorithms too. Combining local minima escaping algorithms with potential functions may give us

another powerful algorithm that works for the case four, paritally known environments with dynamic obstacles.

5. CONCLUSIONS

In robot coordination, reactive approaches give better results than deliberative approaches. Especially, if the environment is local minima free, algorithms those use potential function handle path planning of many robots successfully. However, if the environment contains local minimum, then it is inevitable to use some escaping methods, or deliberative approaches.

If the environment does not change frequently, and the problem requires solution of many path planning queries, using a deliberative algorithm from a set of multiple query path planning algorithms will reduce the problem to just a graph search. Lazy multiple-query algorithms show better performance especially in frequently changing environments. Combining single query algorithms with lazy multiple-query algorithms gave well distributed paths when used for multiple-robot problems.

The centralized deliberative approaches show exponential increase in complexity with respect to the count of robots. When the count of robots increase, decentralized, or combinatorial approaches should be thought to be used as the path planning algorithm. However, decentralized approaches are not complete. Using local groupings may also alleviate this problem.

According to the environment and the aim of the usage, one path planning algorithm may show better performance than the others. Therefore, a good combination of the path planning algorithms may be the best choice. If an algorithm is wanted that is to be simple, but powerful, and working quite good for most of the problems RRT-Connect algorithm is the algorithm. Using various sampling and connection strategies can further upgrade the success of the algorithms.

The algorithms can be tested for kinodynamic path planning problems, for various robot properties as the future work. Implementation of the algorithms can be adapted for partially known environments. An hybrid approach, that is both reactive

and deliberative, may solve the local minima problem, while coordinating the robots effectively.

APPENDIX A: INTEGRATION

$$y_n = y_n + hf(\mathrm{x_n} + \mathrm{y_n}) \tag{A.1}$$

Because the Euler method, shown in Equation (A.1), is unsymmetrical, that is when advancing the solution through an interval h, it only uses the derivative information at the beginning, its error is only one power of h, Figure A.1. So Euler method is not very accurate compared to other algorithms, and it is not stable either.

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{A.2}$$

Runge-Kutta algorithm, shown in Equation (A.2), is symmetrical, and this symmetrization cancels out the first-order error term, and makes the method second order, Figure A.2. Adding up the right combination of first order f(x,y) terms the errr terms can be eliminated order by order. It will give higher order error terms, which means more correct, and stable results.

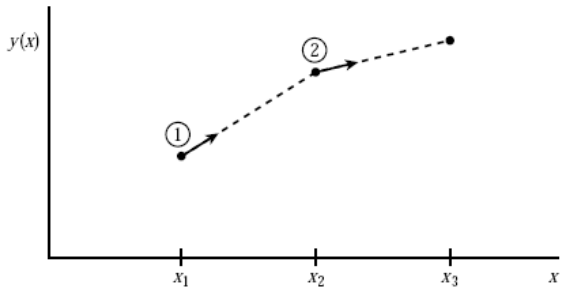

Figure A.1. Euler's integration method has only first order accuracy [31]

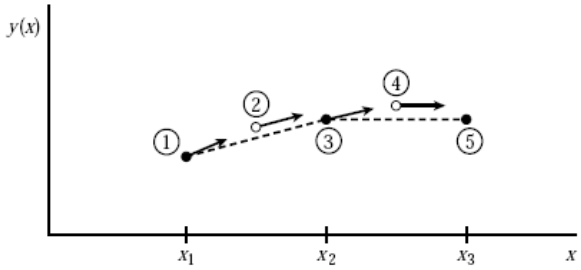

Figure A.2. Midpoint, second order Runge-Kutta integration method gives second order accuracy [31]

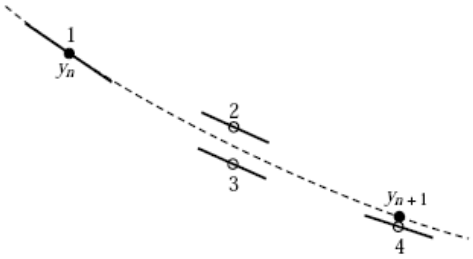

Figure A.3. Fourth order Runge-Kutta method is the most used integration formula with third order accuracy [31]

$$
\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\
k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\
k_4 &= hf(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\end{aligned}
\tag{A.3}
$$

By far the most often used integration formula is the classical fourth-order Runge-Kutta formula, shown in Equation (A.3), and it often gives superior results to midpoint, second-order Runge-Kutta method. So in the simulator, we use fourth-order Runge-Kutta for the calculation of next locations of the robots as the integration formula, Figure A.3.

Runge Kutta 4 algorithm is used as the integrator. It is both a simple to implement algorithm, and has less parameters to think about. It generates quite correct

and stable results. Different integration methods may also be used with the simulator by implementing the *IIntegrator* interface.

APPENDIX B: SIMULATOR

B.1. Simple Robot Simulator

B.1.1. Work Space

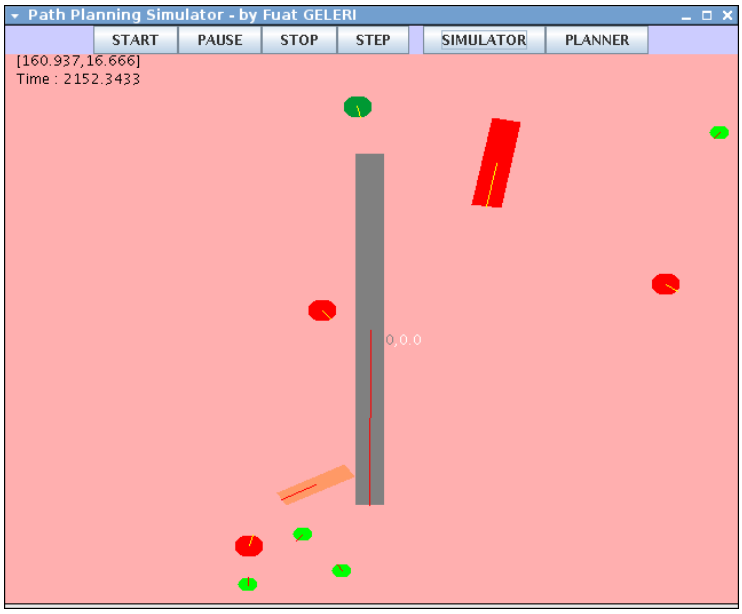

Figure B.1. The simulation environment

The work space in the path planning problems is modelled in the *Simple Robot Simulator* module of the project. It contains robots, stationary and dynamic obstacles as entities. Each robot has an initial configuration, and a target configuration. The obstacles, and the robots may have various shapes, and also the world may have various properties.

The project has the capability of saving and loading all of the various configurations generated.

B.1.1.1. World Content. We keep the content of the work space in *WorldContent* class. This class keeps a list of static objects, and dynamic objects.

B.1.1.2. Static Object. A static object is modelled as an object that may have various shapes, colors, and configurations. As supported shapes, various rectangular shapes, and sphere shape is added to the design. The supported shapes may be extended too, by implementing common shape interface *IObjectShape* and adding it to the list of supported shapes.

B.1.1.3. Dynamic Object. Dynamic objects have all of the properties that static objects have. Moreover, they have a control input property, and a target configuration is assigned to each of them.

B.1.2. Simulator

The simulator adds time factor to the work space information. As time passes dynamic objects change their configurations. The change of the configuration is done by integration of ordinary differential equations, ODE, of the dynamic objects. So, the simulator keeps an object named "WorldContent", which keeps all the static and dynamic objects. It also has maps, to access the current information of the simulated objects, from the name or id to the simulated objects.

Simulator uses an ordinary differential equation implementation with an integrator to calculate the new configuration values for the dynamic objects. In our implementation this ordinary differential equation is designed for velocity values. So the control inputs for the dynamic objects will be velocity values of them. As the simulation steps, the integrator will be issued with the differential equation and the control inputs to generate the next configurations of the dynamic objects.

According to the type of the robot, the ordinary differential equation of the robot should be adapted. For an omni-directional robot, the position of the robot will be direct integration of the velocity it has. However, if the robot has other dynamical properties, they should be also added into consideration when designing its gradient, defining its control inputs, and generating a controller for them.

B.1.2.1. Ordinary Differential Equation.

$$\begin{aligned} y' &= F(y, x) \\ y &= (y, \ldots, y_n) \end{aligned} \tag{B.1}$$

In our simulator implementation, we use a simple differential equation definition; given the configuration and the time, give value of the derivative to be used, shown in Equation (B.1). So, this derivative value shows the amount of the change in the configuration. Because we use velocity values as the control input, the ordinary differential equation will give this value as the derivative value when asked by the integrator, and this will be the amount of change in the position of the robots.

Different differential equations may be added to the system by implementing $IDifferentialEquation$ interface, so the robots with different control inputs, those show different motions, and properties, may be achieved. Presently the robots are modelled as moving omni-directional objects, and having a rotation angle. So the control inputs are the directional velocity values, and rotational velocity value.

B.1.2.2. Integrator. The simulator keeps a *step size* parameter. At each step, as the time value is incremented by *step size* much, the integrators calculate the next configurations by integrating the differential equations by *step size* amount. Fourth order Runge Kutta implementation is used as the integrator, as defined in the Numerical Recipes book [31]. A detailed information about the integrators are given in the Appendix A.

So we achieved a simple self running simulation environment. This environment will be used by other modules of the project. To access the simulated objects' current configurations we added mappings to the simulator. At each step, the timer is increased, the next actions are taken, and by using the mappings other modules can access these new configuration values from the names of the objects.

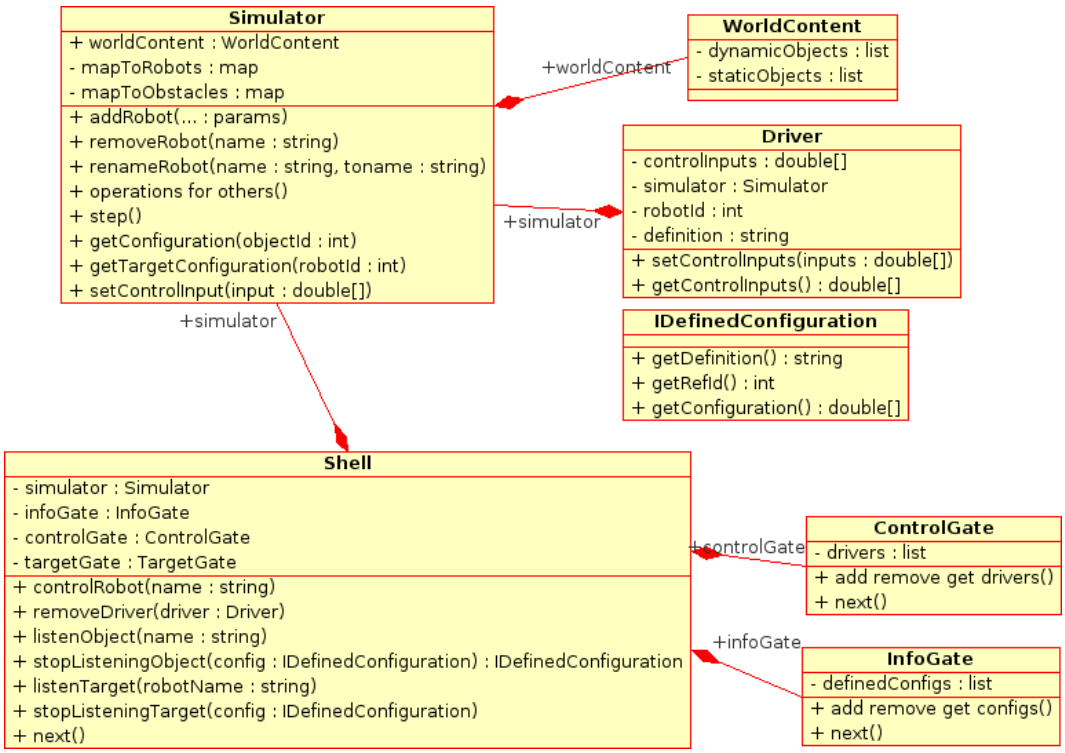

Figure B.2. Class diagram of Shell module

B.1.3. Shell

In order to make the simulator easier to be used by the planning algorithms a shell is designed on top of this simulator design.

The shell contains listen, and control gates. Via listeners the shell listens to the changes at the configurations of wanted objects and via control gates it controls these changes. See Figure B.3 for a simple representation of the logic and see Figure B.2 for the class diagram of the Shell. This shell implementation abstracts the simulator from the planning part. Therefore, different simulators can be used by these planners too. Shell is a bridge design pattern implementation, and different shells can be implemented for different simulators to have a bridge between the plans and the simulators.

B.1.3.1. Info Gates. Info gates update the listened configuration information of the objects at the start of each simulation step. The simulator changes the configuration of the objects gradually. However, because the plan should be done at discrete time

Algorithm B.1 Simplified Simulation Execution

```
1: take initial simulation step
2: while simulation continues do
3:    Shell::InfoGate updates configuration of the objects in the world model
4:    Algorithms execute, drivers updated
5:    Shell::ControlGate updates control inputs
6:    simulation takes next step
7: end while
```

Figure B.3. Steps of a simplified simulation execution

steps, the configuration values must be supplied to the plans in discrete times. So the purpose of the info gate is updating the configuration information, which will be used by the algorithms, at the start of each step.

Presently there are three different info gates implemented, but they may be increased too. Those are InfoGate, TargetGate and PropertyListenGate. The InfoGate listens to the current configuration values of the wanted objects. The TargetGate listens to the changes at the targets of the dynamic objects. The PropertListenGate is empty for now. However, it may listen to the change of the properties like battery power, weight of the robot, etc.

B.1.3.2. Control Gates. For each dynamic object in the simulation we generate a driver, and add this driver to the ControlGate. At each simulation step, the control gate updates the control input field of the dynamic objects with the values in these drivers. The algorithm updates the control input values in the drivers, and the control gate applies these control inputs to the simulation at the end of the simulation step.

B.1.4. Visualization and User Inputs

The simulation is reflected to the computer screen in a multi-layer drawing fashion. First the background, then the track layer, the simulation layer, and the foreground

layer draws to the screen. More layers can be added in between or to the top or bottom of this drawing stack. To show the calculated path tree, and collision tree of the objects a layer is added in the next stages of the project.

The background layer cleans the screen, and draws the world with its shape and bounds. The robots and the obstacles should be bounded by this area. When the robots move they leave footsteps in the track layer. In the simulation layer, we draw the obstacles and the robots. The foreground layer is for information displaying purposes, like showing the current time, and the coordinate of the point mouse cursor is showing.

To make the visualization more powerful a camera object is added to the design. This camera object has a position and a range, and it only shows the objects in its range. The objects shown by the camera are reflected to the screen. So a point in the real work space has a reference in the camera, and it also has a reference at the screen. A circular object may be seen like an object with ellipse shape in the screen, because of these mappings.

We also track the mouse gestures and keyboard strokes. The user is capable of moving, rotating objects in the simulation, and changing the position and range of the camera with some special mouse gestures and key strokes given in Table B.1.

Table B.1. Special mouse gestures and keyboard strokes

Gesture	Action
SHIFT + Left Mouse Button	Select an object to move
SHIFT + Right Mouse Button	Select an object to rotate
Mouse Button Click	Deselect an object, or show it enlighted.
CTRL + Drag with Left Mouse Button	Move the camera
CTRL + Drag with Right Mouse Button	Zoom in and out the camera
CTRL + ALT + Mouse Button Click	Reset the camera to its saved settings

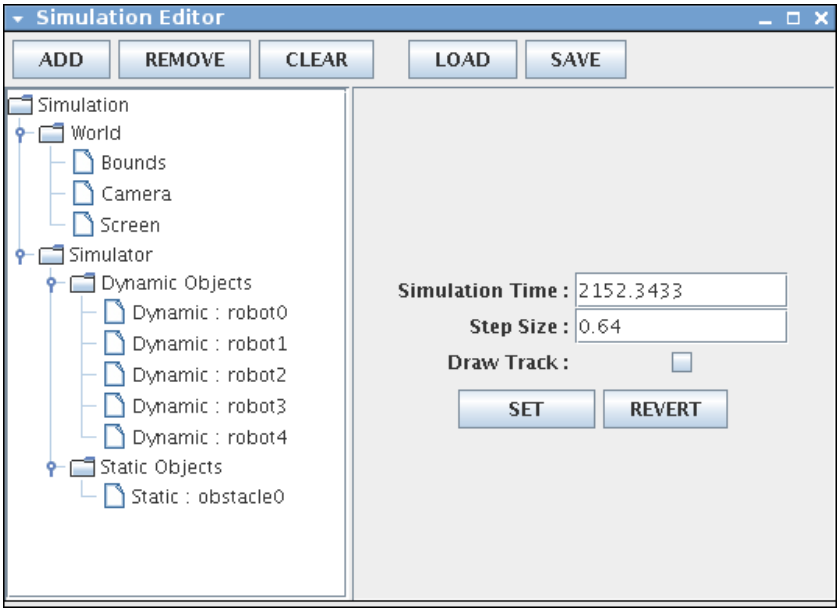

Figure B.4. The Simulation Editor helps generating different simulation scenarios

When the bounds of the world is changed, defaults of the camera is set to this new values too. *Simulator Editor* can also be utilized for all these changes, and to see the exact values.

B.1.5. Saving and Loading

For file operations we used Apache Foundation's Betwixt Java Library [32]. So the properties of the objects, the robots in the work space, the work space's properties, properties of the simulation screen, and the simulator properties all can be saved and loaded by this library. All the information is kept in XML files, and the user can also change the values with other XML editors, and run the simulation with these new values too.

B.2. Robot Simulator Editor

The Simulation Editor, shown in Figure B.4, is designed to edit each property of the work space, the simulation, the simulation screen, and to save and load these settings. User can edit the bounds of the work space, the location and the range of the camera, the size of the screen for showing the simulation within this editor. Some

simulation properties like the time of the simulation, the amount of time elapsed in each step, and whether showing or not showing the ootsteps of the dynamic objects can be edited too. Furthermore, the content of the simulation will be generated with this editor by adding new static and dynamic objects to the work space, and by editing their properties with appropriate property panels.

B.2.1. Dynamic Objects

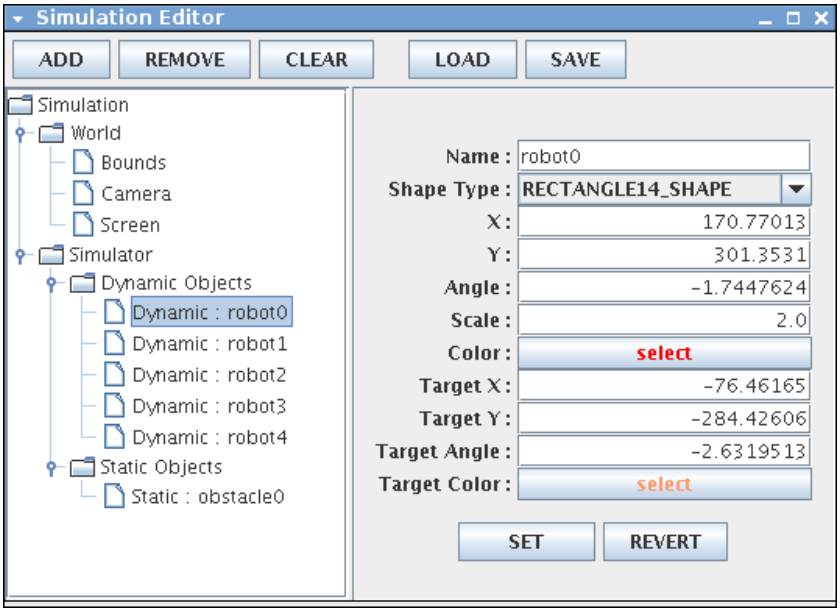

Figure B.5. Via The Simulation Editor we can add, remove, edit dynamic objects

When a dynamic object is selected from the Simulation Editor, the properties of the selected dynamic object is shown at the center editing part. The name of the dynamic object, type of its shape, the scale of this shape can be edited with the help of this propery panel. The user can also change the current configuration and the target configuration of a dynamic object. User can also add different colors to the robots to make them easily distinguishable.

When we selected the *Dynamic Objects* branch of the editor tree, when "ADD" button is pressed, a new dynamic object will be added with a random name, to the center of the work space. After making appropriate settings, pressing "SET" button at the panel in the center of the Simulator Editor, will update the properties of the newly added dynamic object, as shown in Figure B.5. A dynamic object can be either

a controlled robot, or a dynamic obstacle, according to the controller assigned to it. As many dynamic objects as desired can be added to the simulation.

B.2.2. Static Objects

Static objects refer to the stationary obstacles. Similar to the dynamic objects, when a static object selected, the property panel for the selected static object will be shown at the center of *Simulator Editor*, as in Figure B.6. A static object has the properties of the current configuration, as x, y coordinate positions, and the *angle*. It has also a shape type, a scale, a color, and most importantly a name value.

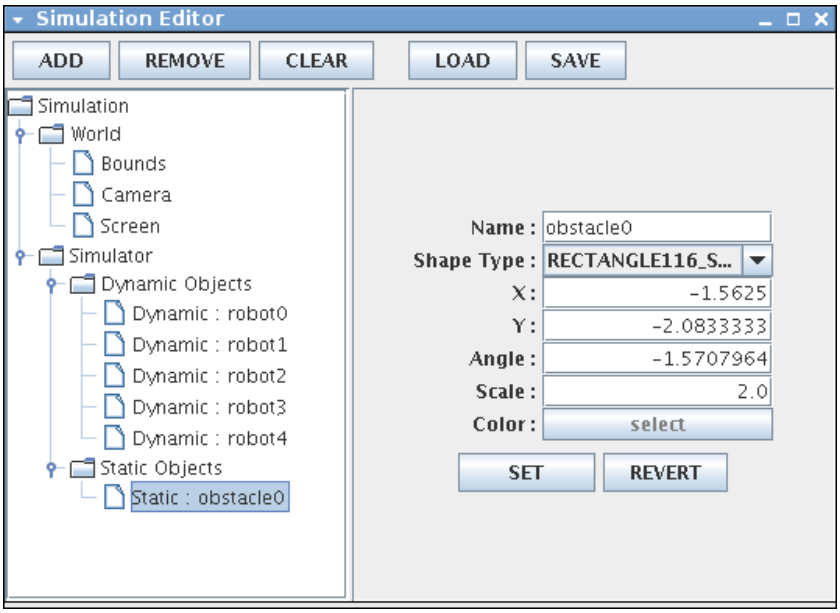

Figure B.6. Via the *Simulation Editor* we can add, remove, and edit static objects

B.3. Planning Base

We will use the simulation environment to generate and simulate the scenarios to test our algorithms. This module is implemented to provide the upper layers a general medium, that will provide them a poiler-plate that supports the "Sense, Plan, Act" scenario. This boiler-plate will have basic implementations to successfully proceed the simulation and run the algorithms.

In this boiler-plate, we will have an abstract world model and sample world

modelings depending on this abstract model. Furthermore, we will have a generic collision checking module. The most important part of this module is the "Executer" part. In this part, we implemented basic execution schema. Coordination of the robot controllers and the simulation will be in the responsibility of the executer.

There is nothing special for path planning in this module. Other modules can be built on top of this module to have the simulator to be used in different planning purposes. Path planning base is also built on top of this module, and it contains path planning specific classes.

This module contains common entities for modeling the work space, a collision checker implementation and a generic controller model for the control algorithms. This module is not designed specifically for path planning algorithms, in fact a highly generic model that could be used for different approaches is tried to be implemented.

B.3.1. Modelling the World

Other than the world at the simulator module, we will have a model of this world in the planning part. In fact, each controller will have a model of the simulation environment. A model may contain controller specific information about the work space. In this module we designed a general world model, that can be extended on the purpose.

At the start of this document, we divided the path planning problems into four categories according to their difficulties. The case one and two are the cases, where the environment is fully observable. The other cases assume the environment to be partially known. To support both of these scenarios we made the generation of the world model adaptable. According to each controller we will have various world models, and the content of the world model will be editable. Therefore, if an algorithm, which should have partial information about the world, and which should model the world as it proceeds, will have a chance, and also algorithms which required the environment to be known fully beforehand will have also a chance to be implemented.

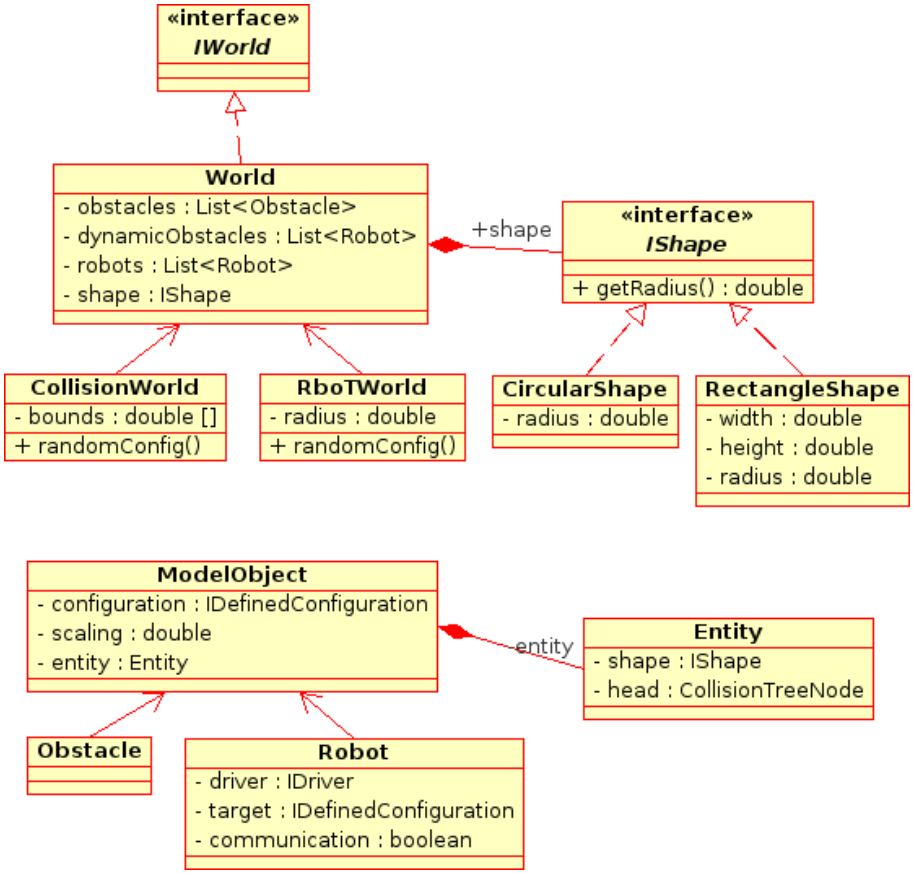

Figure B.7. Class diagram of modelling of the world

In our implementation we assume that the model of the world is provided to the robot control algorithms. So before starting the controller we generate a world model from the simulation and pass this model to the controller. In fact, this is not obligatory, that is, if the algorithm has capability of self inpecting the environment, it can inspect and model its own world too. However, we give the algorithms the opportunity to access the world beforehand.

In this most generic world model implementation we modelled the world as a bounded area, with spheric or rectangular bounds, that contains the controlled robots, the dynamic and the static obstacles. As seen in the class diagram of this most generic world class, in Figure B.7, we have capability of adding, renaming, removing every object, and also have chance to differentiate between the types of the world with the *WorldType* property.

We have two different world model implementations, as those having circular bounds and rectangular bounds. RboTWorld has a spheric bound, and CollisionWorld has a rectangular bound. Both implementations have the same content. The content contains the robots, and the obstacles. As like their definition in the simulation, the robots and the obstacles have configuration, shape properties and a scaling property. The robots have also a target configuration property. This shape and configuration properties will be used by the collision checker algorithm.

Each object in this world model is an entity, a member of Entity class. We have a ModelObject class, which holds this entity, and information about the configuration and scaling of the entity. So, from the entity property we get information about the properties of the object, and configuration related information is separate from them. An obstacle, and a robot is also a model object. A robot has a driver and a target configuration as an addition. The controller of the robot will use this driver to give next control inputs to the robot. Each entity will have a shape, and according to the shape and scaling of the model objects we will perform collision checks.

In fact the world in the planning part is a reflection of the work space of the simulation part. However, according to the used algorithms an abstraction is applied, shown in Figure B.7. The properties of the objects in the world model will be updated by the execution at the start of each simulation step, whereas the object properties in the simulation will show a continuous change.

B.3.2. Collision Checker

There are various collision checking algorithms mentioned earlier in this report. We selected to represent objects as trees of circular collidables. So each object in the world model has also a collision tree assigned to it, as shown in Figure B.8. The root of this collision tree encapsulates the object. When checking for collision, at the first the root of the trees will be checked for collision. If the roots are not colliding then no need to check the branches of the trees. However, if the parents are colliding, then we go into the branches. If the check continues till the leaves, and some of the leaves are

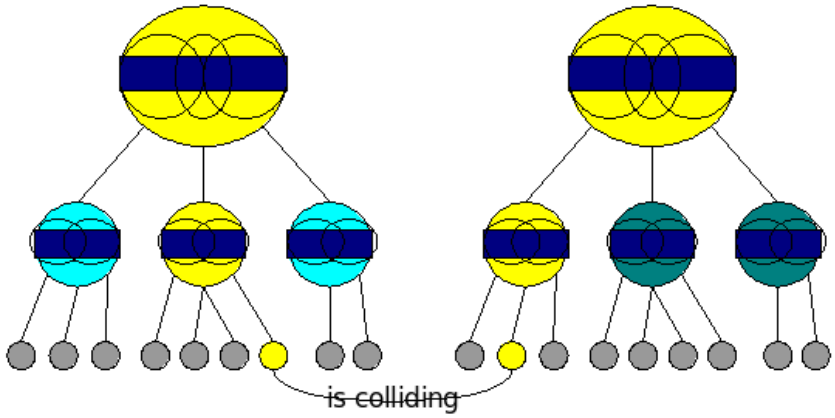

Figure B.8. Diagram of example usage of collision tree in collision check

coliding then we say collision occurs. However, if no leaf is colliding then there is no collision.

A plan may let the robots to collide, and this may not be a problem for it. For example, a plan may see a robot as a carrier for the others, and collision of these robots may mean entrance of the robot to the carrier. There may be different scenarios too. However, the collision checking should be handled according to the purpose of the execution, and this decision should be given by the executer. So we placed collision checking module not in the simulator, but in the planning base module.

As mentioned, the planning base provides a collision checking mechanism that divides the objects to the collidable circulars. Each entity in the world model will have a collision tree assigned to it. The collidable circulars will take the center of the object as the reference point. So, robot may have different configuration and scaling values, but the underlying structure, the collision tree, never changes. It only makes some trigonometric calculations and a tree based collision search.

The collision checker module implemented in this module is as simple as it can be. It checks whether there is a collision between two objects with given configuration and scaling values, or not. It does not need any information about why these two objects are compared etc.

B.3.3. The Executer

The executer module is designed to coordinate the execution of the controllers, and the simulator. It knows nothing about the process of the controller, or the responses of the simulator to these actions. However, it keeps a list of controller and makes them run on the given simulation.

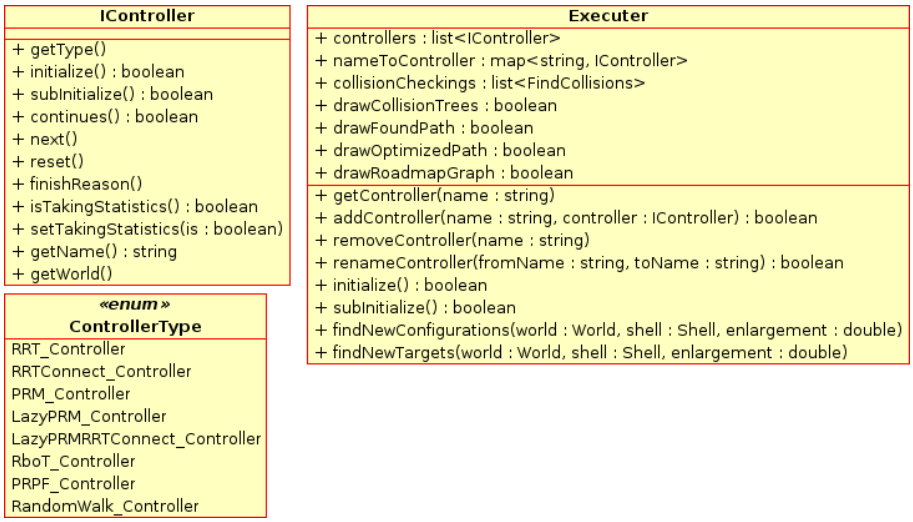

Figure B.9. Class diagram of the Executer

There may be more than one controller executed on the current simulation. The executer may also be improved too. For now it waits each controller to generate their responses for the next step. However, the controllers may also be run in different threads and executer may try to give only some amount of time to each controller to operate.

Controllers may contain a TCP/IP implementation, and distribute their execution to different computers. All the required information will be passed to the other computer, and the other computer will make the calculations. After the calculations are completed, the control inputs for the robots will be provided back. For the sake of simplicity, the executer is designed to wait each controller. So this kind of approaches will be easier to implement. However, already in the statistics we can examine the time elapsed for each controller to operate, and see the latency the controller adds to the simulation.

The executer also decides about the collision checks. If an algorithm is statistically important we add this algorithm's world to the worlds, that will be checked for collision. Only the controlled robots in the world will be checked with themselves for collision and the other objects in the world. However other objects will not be checked between each other. So we know the controllers are only responsible to make their controlled robots do not make any collision. The algorithms, which we do not want to collect any statistical information about, but needed to make some moving obstacles, will have the worlds too but these worlds will not be checked for collision.

We want the simulator to be used not only for statistical purposes, but also to inform the users graphically about the underlying process. Therefore, the executer has some boolean values about showing the found path, the optimized path, the collision trees of the objects, and the generated roadmap graphs. Drawing these artifacts gives a good intuition to the user about the progress of the algorithm.

B.4. Path Planning Base

In this module common structures for the path planning algorithms are implemented. Most of the path planning algorithms, those model the configuration space, uses a graph, and this module contains various graph implementations and graph search algorithms. For tree based path planning algorithms the module contains a path tree implementatation too.

Sampling based path planning algorithms use a local planner in their execution. World specific information, collision checking, generation of random samples are usually done by this local planner. The local planner is also used to connect different configurations with a local path. In thid module, an interface for path planners, and an interface for the local planners, moreover a generic local planner implementation is provided.

Furthermore, the way of taking statistics is implemented in this module too. IStatistics class contains methods to be called at some levels of execution, that will lead successfully collecting statistical information about the algorithms execution.

B.4.1. Local Planner

Local planner will be used to abstract the world from the path planner. A path planner will use a local planner, and search for a path according to the responses of the local planner to its questions.

As seen in the class diagram in the Figure B.10, the local planner is designed as such, the planner will have no information about the configuration space it executes. All of the configuration space related operations will be done by the local planner. Therefore, we achieved highly abstract sampling based path planner implementations. These implementations can work on different environments, with different dimensions, with different count of controlled robots, and dynamic obstacles.

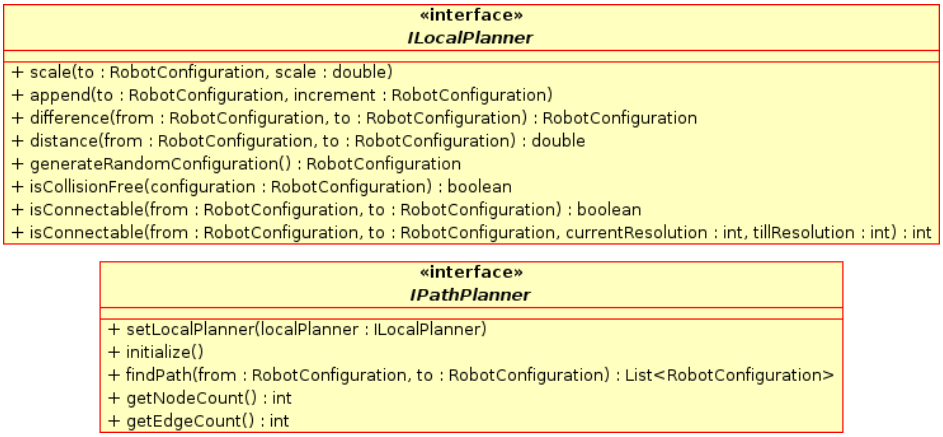

Figure B.10. Class diagram for the interfaces of the path planner and the local planner

The local planner tries to respond to every need of the planning algorithms. Some algorithms may need the collision checker algorithm to take only the static obstacles into consideration, when they are making their initial plans. They may later need the collision checker to inform them about possible next collisions with the dynamic obstacles too. So, the local planner supports different options, like checking the robots with only the static obstacles for collision, checking the collision between themselves, or collision with dynamic obstacles etc.

In fact, we should not think the term collision as two robots are touching to each other, or a robot is touching with an obstacle. By saying collision checking we in fact mean checking the robots for their constraints. In different scenarios, the collision checking implementation may be different. For example in our implementations we used velocity as the control input, and assumed every robot is omnidirectional with infinite acceleration capability. When the control inputs are the velocity values, the configuration values are bounded to only position of the robots, and their orientation. However, if the acceleration values of the robots were limited, then the current velocity of a robot would be a reason for the collision of the robot in the next simulation steps. If robot is near to an obstacle, and has high speed, the robot may not be able to save from colliding with the obstacle. However, in infinite acceleration case, there would be no such problem. So, by doing collision checks we not only mean checking the robots and the obstacles for collision, but also checking the capabilities of the robots.

It might be more clear if we describe the configuration space and collision checking with examples. For example, if we have a robot in $2D$ space which has position, orientation, velocity, battery, and weight information the configuration space will be really interesting. For such a scenario we will have 3 dimensions for the location, 3 dimensions for velocities in the x, y axis, and the rotational velocity, and one dimension for the battery and one dimension for the current weight of the robot. In total we will have 8 dimensions in the configuration space per robot. As the control input we will have force values. It will let the ordinary differential equation of the robot and the integrator of the simulator to generate new velocity and location values. The simulator will also calculate the next battery power amount, according to ordinary differential equation for the battery consumption.

In this scenario, according to the velocity of the robot, robot may take a passage without any collision, or the initial velocity may lead the robot to collide. So the collision checking algorithm should be modified to include the affect of the initial speed when planning the full path.

Moreover, the robot has a battery, and the power in the battery may affect the limit of the torque the robot can have. If we have some hills in the work space to climb, the amount of the power on battery will decide about making a connection between the nodes in the roadmap or give a collision like result for passing the hill. If the battery is low, robot may not be able to climb the hill, however it may successfully pass the hill if the batter power, so the torque is enough.

By implementing different collision checking algorithms, we can use the sampling based path planning algorithms in various path and motion planning problems. When we think about the kinodynamic properties of the robots, path planning will be named as kinodynamic path planning. In this kind of plannings the collision checker should take the kinodynamical constraints and properties of the robots into consideration.

B.4.1.1. Generic Local Planner. Generic local planner implementation is designed to support both decentralized and centralized path planning algorithms. In the centralized algorithms a bigger configuration is generated by summing the configurations of each individual controlled robot.

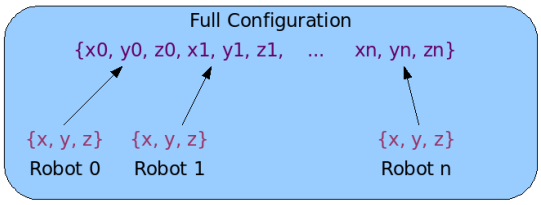

Figure B.11. A big configuration is achieved by appending robot configurations

As seen in Figure B.11, from the configurations of each robot, we achieve a big single configuration space. The path planning algorithms will make their searches within this space, without changing anything on their implementations, because of this generic local planner implementation.

Generic local planner implementation will make the operations on this big configuration, and will accord the collision checking method, so that each robot will have the configurations in this big configuration, and collision checks will be done in the work space for each robot.

By implementing such a local planner, a mean for path planning algorithms to work both for single robot, and multiple robots in centralized manner is provided.

B.5. Planning Base Editor

Planning base editor is designed to generate the planning part for the simulations. A plan may be implemented by only one controller or a set of controllers can cooperate on the same environment for a plan.

First, we add a controller, then adapt it for the environment. We select the algorithm to be used in this controller, and accord the parameters of the algorithm.

Then, the controller is ready to be run. We apply our settings, perhaps save them to load later. When the simulation is started, the controller will be visited for each time step. According to the selected algorithm the controller will provide next coordination inputs for the robots for the next simulation step.

B.6. Robot Controllers

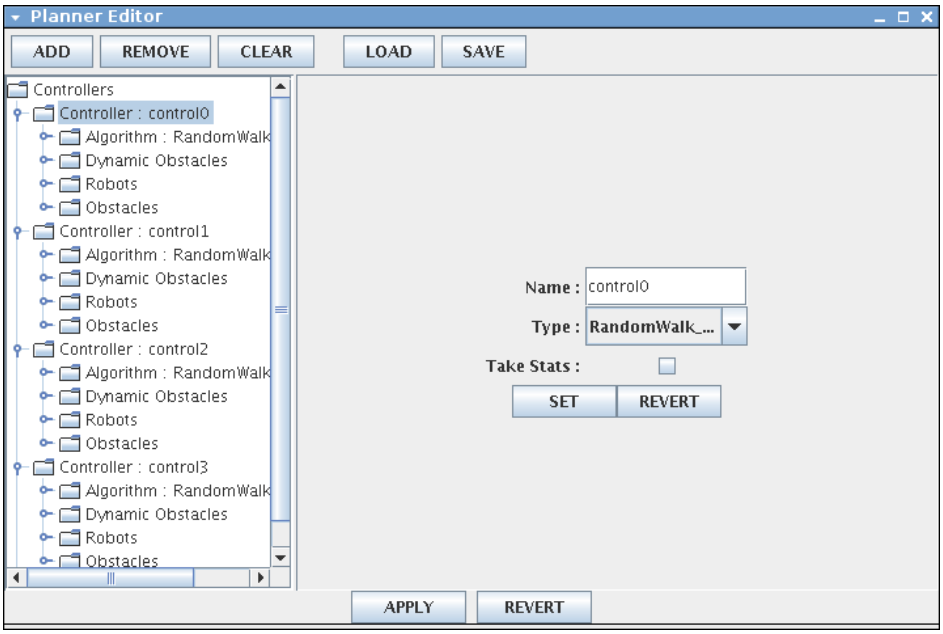

Figure B.12. Four controllers added to the simulation with the planner editor

Robots may be controlled by only one controller, and a centralized approach can be achieved by doing so. Moreover, we can distribute the robots to more than one controller, and can also make the algorithms run in decentralized manner.

In Figure B.12, the planner editor shows four controllers are added to the simulation. The tree view at the left enables user to see the list of controllers, and their properties. We can add robots, dynamic obstacles, and static obstacles to the world model of the controllers with this editor. When a controller is selected, the type of the controller algorithm, name of the controller, and whether the controller is statistically important or not can be adapted within the panel shown on the center of the planner editor.

The user can also add, rename, remove the entities in the world model of a controller. The type of the world will be changed according to the selected algorithm for the controller. User can also change the properties of the world according to the needs. The objects in the world model, can be scaled up to prevent the robots to collide with the obstacles in the case of a noise applied to the field.

In Figure B.13 we see the listed controllers on the left pane. Controller named *controller*1 is opened on the tree. Its algorithm is *RandomWalk*. The center pane shows the properties of the algorithm. Its name and the selected algorithm is shown. We see that the algorithm is changed to *LazyPRM*, but did not applied yet. Because when the algorithm change is applied, the tree at the left pane should be updated with this new algorithm information too. At the tree, in the algorithm branch we see a world property. This is a branch of the algorithm, because world type will change with the algorithm type. Some algorithms use RboTWorld, whereas others use CollisionWorld. User can select the world item, and properties of the world will be shown at the center pane.

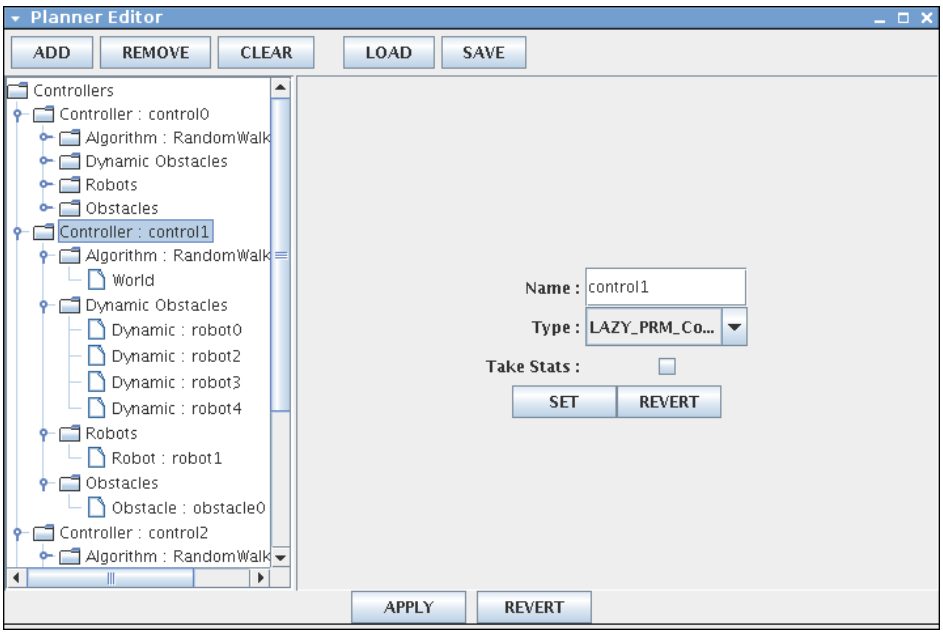

Figure B.13. A controller is shown in detail in the planner editor

We have *Dynamic Obstacles*, *Robots*, and *Obstacles* branches at the tree too for each controller. All the robots, not controller by the controller should be added into

Dynamic Obstacles. Static obstacles and controlled robots will also be added. The properties of the objects will be accorded according to the objects in the workspace. There is a scaling property assigned for each object. Objects may be scaled up or down according to the needs of the algorithm.

Controllers will use *InfoGate* to listen the configurations of the objects, however the type of the objects should be provided by the planner editor to the controllers. For some algorithms, those generate the world model as themselves, the world model is not needed to be provided in the planner editor.

B.7. Little Prince Path Planning Simulator

Because of the limitations of the simulators in the bazaar, we implemented our own simulator implementation that is specialized for path planning purposes. At the center it has a simple simulation logic, a shell, an editor for the simulation environment. On top of them a planning base is implemented, and it extended to a base for path planning algorithms. All the path planning algorithms, and according controllers are implemented and added to the whole system. The algorithms are made available to be used by the robot controllers. We made all of these settings to be easily saved and loaded.

So we achieved a highly configurable, robust, specialized for path planning algorithms, simulator environment, named *Little Prince Path Planning Simulator*.

Test scenarios are generated in this simulator and console executer part of the project is used to run these tests in quick mode. In the console executer part of the project, we load the test scenarios and run them with different options, like regenerating the initial and target configurations of the robots, how many times to run the tests and more. The statistics of these runs are written to files by the statistics module, and also their averages and standard deviation values are calculated for each case of the results. For the successful completions we calculated each different results' mean and deviation values, also for other completion results we calculated them. So user may

see the amount of time past if the algorithm fails, and decides about its failure.

The simulator is designed to be used in quick algorithm development. It helps algorithm designers in testing their algorithms with different scenarios easily.

REFERENCES

1. Choset, H., K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki and S. Thrun, *Principles of Robot Motion, Theory, Algorithms and Implementations*, The MIT Press, June 2005.

2. Bohlin, R. and L. Kavraki, "Path Planning Using Lazy PRM", *Proceedings IEEE International Conference on Robotics and Automation*, 2000.

3. Kuffner, J. J. and S. M. LaValle, *"An Efficient Approach to Path Planning Using Balanced Bidirectional RRT Search"*, Tech. Rep., CMU-RI-TR-05-34, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Aug. 2005.

4. Kuffner, J. J. and S. M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning", *Proceedings IEEE International Conference on Robotics and Automaton*, pp. 995-1001, 2000.

5. Karagöz, C. S., H. I. Bozma and D. E. Koditschek, "A Feedback-Based Event-Driven Parts Moving Robot", *IEEE Transactions on Robotics*, 2004.

6. Lavalle, S. M., *Planning Algorithms*, Cambridge University Press, Cambridge, U.K., 2006.

7. Karagöz, C. S., H. I. Bozma and D. E. Koditschek, "Coordinated Navigation of Multiple Independent Disk-Shaped Robots", *to appear in IEEE Transactions on Robotsics*, 2006.

8. Goodwin, R. and R. Simmons, "Rational Handling of Multiple Goals for Mobile Robots", *Proceedings of the First International Conference (AIPS-92), College Park, MD*, June 1992.

9. Reif, J. H., "Data Flow Analysis of Communication Processes", *The annual Sym-*

posium on Principles of Programming Languages, pp. 257–268, 1979.

10. Canny, J., "Some Algebraic and Geometric Problems in PSpace", *Proceedinds 20. ACM Stoc*, pp. 460–467, 1988.

11. Siméon, T., J.-P. Laumond and C. Nissoux, *Visibility Based Probabilistic Roadmaps for Motion Planning*, Advanced Robotics, Vol. 14, No. 6, 2000.

12. Aurenhammer, F., "Voronoi diagrams – A Survey of a Fundamental Geometric Structure", *ACM Computing Surveys*, Vol. 23, pp. 345–405, 1991.

13. Leng-Feng, *Decentralized Motion Planning Within a Artificial Potential Framework (APF) for Cooperative Payload Transport by Multi-robot Collectives*, M.S. Thesis, The State University of New York at Buffalo, 2005.

14. Sánchez, G. and J.-C. Latombe, "On delaying collision checking in PRM planning: Application to multi-robot coordination". *International Journal of Robotics Research*, Vol. 21, No. 1, pp. 5–26, 2002.

15. Latombe, J.-C., "Robot Motion Planning" *Kluwer, Boston, MA*, 1991.

16. Berg, J. and M. Overmars, "Prioritized Motion Planning for Multiple Robots", *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2217–2222, 2005.

17. Peng, J. and S. Akella, "Coordinating Multiple Robots with Kinodynamic Constraints Along Specified Paths", *Algorithmic Foundations of Robotics V (WAFR 2002), Springer-Verlag, Berlin*, pp. 221–237, 2002.

18. Lindemann, S. R. and S. M. LaValle, "Current Issues in Sampling-Based Motion Planning", *Proceedings International Symposium on Robotics Research, Springer-Verlag, Berlin*, 2004.

19. Hsu, D., J.-C. Latombe and R. Motwani, "Path Planning in Expansive Configu-

ration Spaces", *International Journal Computational Geometry and Applications*, Vol. 4, pp. 495–512, 1999.

20. LaValle, S. M., *Rapidly-Exploring Random Trees: A New Tool for Path Planning*, *Computer Science Dept., Iowa State University*, Oct. 1998.

21. Plaku, E. and L. E. Kavraki, "Distributed sampling-based roadmap of trees for large-scale motion planning", *Proceedings IEEE International Conference on Robotics and Automation*, 2005.

22. Amato, N. M., O. B. Bayazit, L. K. Dale, C. Jones and D. Vallejo, "OBPRM: An Obstacle-Based PRM for 3D Workspaces", *Proceedings Workshop on Algorithmic Foundations of Robotics*, pp. 155-168, 1998.

23. Dijkstra, E. W., *A note on two problems in connexion with graphs*, Numerische Mathematik, Vol. 1, 269–271, 1959.

24. Dechter, R. and J. Pearl, "Generalized best-first search strategies and the optimality af A*", *Journal of the ACM*, Vol. 32, No. 3, pp 505–536, 1985.

25. Hsu, D., T. Jiang, J. Reif and Z. Sun, "The Bridge Test for Sampling Narrow Passages with Probabilistic Roadmap Planners", *Proceedings IEEE International Conference on Robotics and Automation*, 2003.

26. Boor, V., M. H. Overmars and A. F. van der Stappen, "The Gaussian Sampling Strategy for Probabilistic Roadmap Planners", *Proceedings IEEE International Conference on Robotics and Automation*, pp. 1018–1023, 1999.

27. Hsu, D., L. E. Kavraki, J-C. Latombe, R. Motwani and S. Sorkin, "On Finding Narrow Passages with Probabilistic Roadmap Planners", *Proceedings of the Workshop on Algorithmic Foundations of Robotics (WAFR'98)*, pp. 155–168, 1998.

28. Wilmarth, S. A., N. M. Amato and P. F. Stiller, "MAPRM: A Probabilistic Roadmap Planner with Sampling on the Medial Axis of the Free Space", *Proceedings*

IEEE International Conference on Robotics and Automation, pp. 1024–1031, 1999.

29. Bobic, N., Advanced Collision Detection Techniques, *Gamasutra : http://www.gamasutra.com/features/20000330/bobic_ 01.htm*, Mar. 2007.

30. Samet, H. and R. E. Webber, "Hierarchical Data Structures and Algorithms for Computer Graphics. Part I.", *IEEE Comput. Graph. Appl., IEEE Computer Society Press*, Los Alamitos, CA, USA, Vol. 8, No. 3, pp. 48–68, 1988.

31. Flannery, B. P., S. A. Teukolsky and W. T. Vetterling, *Numerical Recipes", 2nd edn.*, Cambridge University Press, Cambridge, 1992.

32. Apache Foundation's Betwixt Developers Team, Betwixt Library, http://jakarta.apache.org/commons/betwixt/, 2007.