Trajectory Planning using Bi-RRT Algorithm

Final Report

B.Sc Ahmed Mzid, B.Sc Helmi Abidi, B.Sc Oussema Dhaouadi
Supervisors: M.Sc. Quentin Leboutet, Prof. Dr Gordon Cheng

***Abstract*—One of the key skills to win football matches is to follow a path while dribbling toward the ball as fast as possible under the constraint of avoiding obstacles (opponents, teammates and objects).**
This task is difficult because the robot's environment (football field) is very dynamic. Since the lifetime of the current plan is very short, a continuous replanning is necessary. RoboCup robots are equipped with a limited number of computer resources, but they have to perform many parallel tasks after the acquisition of real-time data. The trajectory planning algorithm must, therefore, perform within a relatively short time interval.
In this report, we describe the theory behind trajectory planning using Bi-RRT* and its implementation on Nao robots intending to improve the trajectory planning implemented on the current Framework. A detailed description of the environment is assumed to be known. Experiments showed that this algorithm outperforms the current implementation.

***Index Terms*—Collision avoidance, path planning, probabilistic trajectory planning, rapidly exploring random tree, Nao, RoboCup, HULKs, robotis.**

I. Introduction

A. Motivation

A robot is a machine equipped with the ability to perceive, decide and act in the environment according to the situation. Most often, these tasks are repetitive and follow a well-defined strategy. The possible types of applications are countless. We are interested in robotics of the interior, especially in soccer robots. Nao is a suited framework used in a soccer game since it can perceive and inform about its environment independently. Such a distributed system generally has to perform three basic tasks, which are location, planning and navigation. Moving without getting lost and while avoiding obstacles seems to be hard in a dynamic environment. A robust trajectory planning is hence necessary. The planning problem to be solved is always the search for a trajectory between the initial state and a target. The current implementation based on obstacle avoiding vectors shows deficiencies. In fact, the generated path is simplistic and results in collisions and a poor performance. This motivates the need for an alternative trajectory planning algorithm which takes into account the state of the environment in the future, by random exploration.

Institute for Cognitive Systems (ICS), Technical University of Munich, Faculty of Electrical Engineering and Computer Science, 45/II Karlstraße 80333 Munich, Germany.

B. Related Works

The environment could be modeled as a set of all possible configurations, called the configuration space C, where a subset $C_{collision}$ contains all points leading to a collision and a subset C_{free} describing the collision-free space. A simple approach to path planning is to discretize the workspace to construct a map grid. In this case, path planning is performed by linking the nodes, elements of C_{free}.
Since our working space is large (football field), the number of samples required can grow to be frighteningly large. An alternative idea for dealing with these situations is to choose the points in the configuration space randomly instead of uniformly. These methods are referred to as probabilistic approaches. We consider two methods:
Probabilistic Road Map (PRM) [1]. This stochastic process generates a random point in the configuration space and checks if it is in free space and selects the k-nearest neighbors by using a predefined distance function (generally Euclidian distance). Afterwards, a local planner decides whether there is a path between the new point and the selected neighbors. Subsequently, a graph-based planning algorithm (e.g. Dijkstra, A-star) finds the optimal path. Since this algorithm randomly samples points in the workspace without following a smart strategy, it fails to generate samples in narrow areas and the planner is not able to find a path within a short time.
Rapidly-exploring Random Trees (RRT) [2]. This incremental method of tree construction aims to quickly explore the state space. It biases the sampling algorithm for the sake of increasing the chances of finding routes in such critical situations. It also considers the start and the goal locations in the sampling procedure, without

the need of spanning a graph over the entire space. The constructed graph is called a tree, where every node is connected to a single parent, and it is rooted at a given starting location.

Since robots should work as fast and safe as possible and since in most cases the environment is complicated, non-structural, dynamic and not known a priori, an adaptive planner must also be developed. RRT is known as a robust path planner performing in a cluttered environment, where the probability of agglomeration is high. It could be used in all areas of robotics activity, such as autonomous vehicle operating in an urban environment [3], humanoid arm motion planning [4] and robot navigation [5].

RRT is a very common algorithm for motion planning. It has, hence, numerous extensions. In this work, an extension of the RRT is discussed, namely the bidirectional RRT*. A detailed description of theses algorithms will be presented in the next section.

II. Theoretical Overview

Trajectory Planning is a well-known problem in robotics. It has been tackled by many approaches which differ on how they explore the collision-free optimal path to the target.
The main idea of Rapidly Exploring Random Trees (RRT) consists in splitting the environment into two spaces: space without obstacles and space with obstacles as a first step. Afterward, the algorithm explores the environment based on randomly sampled points from the free space.

The Bi-RRT is an expansion of the RRT* algorithm. Therefore the RRT* will be presented in a first place.
Before digging deep into the algorithm's explanation, the problem of trajectory planning will be formally introduced.

Problem Statement

As mentioned above the state space: $X \subseteq R^2$ will be divided into two subsets, namely the collision free space X_{free} and the obstacle region $X_{obstacles} = X \setminus X_{free}$. Among X_{free}, we find X_{goal} which defines the goal region. Furthermore we define two growing trees [6] $T_a = (V_a, E_a)$ and $T_b = (V_b, E_b)$, where V denotes the nodes and E denotes the edges connecting these nodes. $\{x^a_{init}, x^b_{init}\} \subset X_{free}$ represent the starting states for T_a and T_b.
The terms introduced above will be used in the description of the RRT* approach.

*RRT**

In this section, the workflow of the RRT* algorithm will be explained based on the following pseudo-code [7].

Algorithm 1 RRT*(x_{init})

$V \leftarrow x_{init}$; $E \leftarrow \emptyset$; $T \leftarrow (V, E)$;
for $i \leftarrow 0$ to N do **do**
  $x_{rand} \leftarrow Sample(i)$
  $x_{nearest} \leftarrow NearestVertex(x_{rand}, T_a)$
  $x_{new} \leftarrow Extend(x_{nearest}, x_{rand})$
  $X_{near} \leftarrow NearestVertices(x_{new}, T)$
  if $X_{near} = \emptyset$ **then**
    $X_{near} \leftarrow NearestVertex(x_{new}, T)$
  end if
  $L_s \leftarrow GetSortedList(x_{new}, X_{near})$
  $x_{min} \leftarrow ChooseBestParent(L_s)$
  if $x_{min} \neq \emptyset$ **then**
    $T \leftarrow InsertVertex(x_{new}, X_{min}, T)$
    $T \leftarrow RewireVertices(x_{new}, L_s, E)$
  end if
end for
return $T = (V, E)$

The key-functions of RRT* will be explained consecutively.
Sample(i). This function returns independent and uniformly distributed random samples from the obstacle free space.
Extend. Exploration of new points is constrained with the parameter $StepSize$. This parameter forces that new points are in circle region of radius step size from old explored points. The extend function will fulfill this constraint by defining the point x_{new} in the direction of $[x_{nearest}, x_{rand}]$ such that $distance(x_{nearest}, x_{new}) = StepSize$.
Near Vertices. The output of this function is the neighborhood set of a point x. Before the start of the algorithm, a neighborhood radius will be set. This being defined, this function will return the set of Tree-vertices that are included in the circle of center x and with neighborhood radius.
NearestVertex. This function will return the nearest vertex in the available Tree to the selected point.
GetSortedList. Based on a predefined cost function (usually the sum of Euclidean distances to target) all elements of X_{near} will be sorted in ascending order.
ChooseBestParent. This function iterates over the list L_s and returns the best parent of the new sampled point. The choice of the best parent aims to minimize the cost from x_{init} to x_{rand}.

RewireVertices. After the addition of the sampled point to the T, this function tries to optimize the wiring of all nodes in X_{near}. It tests whether wiring the old nodes through the new point x_{rand} reduces the cost of each added node.

Discussed steps above will be exhibited by the figure below:

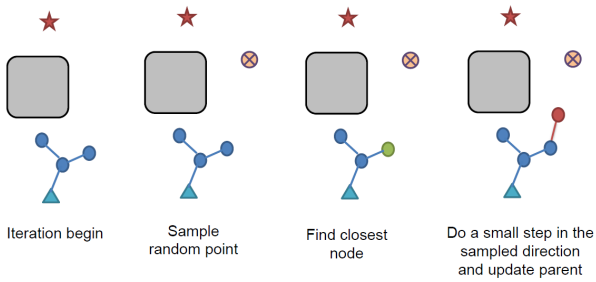

Fig. 1: One Iteration of RRT* algorithm

After some iterations, the Tree and the final path will look as follows:

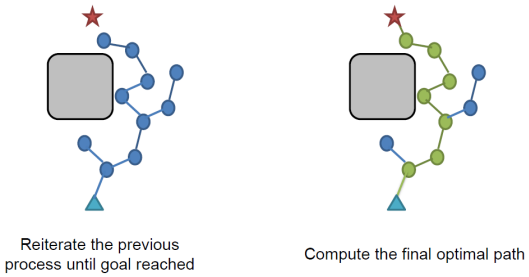

Fig. 2: Reiterating RRT* and Computation of final path

Bi-RRT

This section will outline the main idea and the workflow of the Bidirectional RRT algorithm.
The main idea is that we will have two growing trees. The first one T_a will grow from the starting point x_{init} in the direction of the target point x_{goal}. The second tree T_b grows from the target point in the direction of the initial point. Once the distance between an explored node n_a from tree T_a and node n_b from tree T_b is smaller than a given threshold, the two trees will be merged and a final path will be computed through the nodes n_a and n_b [8].

The workflow of Bi-RRT will be present using the following pseudo-code [7]:

Algorithm 2 Bi-RRT*(x_{init}, x_{goal})

$V \leftarrow \{x_{init}, x_{goal}\}$; $E \leftarrow \emptyset$; $T_a \leftarrow (x_{init}, E)$; $T_b \leftarrow (x_{goal}, E)$; $terminated \leftarrow False$
for $i \leftarrow 0$ to N **do**
  $x_{rand} \leftarrow Sample(i)$
  $x_{nearest} \leftarrow NearestVertex(x_{rand}, T_a)$
  $x_{new} \leftarrow Extend(x_{nearest}, x_{rand})$
  $X_{near} \leftarrow NearestVertices(x_{new}, T_a)$
  if $X_{near} = \emptyset$ **then**
    $X_{near} \leftarrow NearestVertex(x_{rand}, T)$
  end if
  $L_s \leftarrow GetSortedList(x_{rand}, X_{near})$
  $x_{min} \leftarrow ChooseBestParent(L_s)$
  if $x_{min} \neq \emptyset$ **then**
    $T \leftarrow InsertVertex(x_{rand}, X_{min}, T)$
    $T \leftarrow RewireVertices(x_{rand}, L_s, E)$
  end if
  $x_{conn} \leftarrow NearestVertex(x_{new}, T_b)$
  if $distance(x_{new}, x_{conn}) \leq ConnectThreshold$ **then**
    $T \leftarrow merge(T_a, T_b)$
    $terminated \leftarrow True$
  end if
  if $terminated = True$ **then**
    break
  end if
  SwapTrees(T_a, T_b)
end for
return $T = (V, E)$

III. Technical Details

This section provides a deeper overview of the technical aspects which are relevant for the motion planning module. The algorithm implementation was built based on the existing motion planning module. Therefore, the overall structure and dependencies of this module are introduced in the first place. Subsequently, the implementation of the new motion planning approach, as well as the way it is related to the remaining code parts, are explained.

A. Structure of the motion planning module

The motion planning class is derived from the *base module* class in the context of the *brain thread*, which runs at a frequency of 60 Hz allowing camera image processing, environmental perception and modeling, and trajectory planning.

At each *cycle* begin, the module dependencies are updated. These include a few strategical and environmental parameters, e.g. the playing role (e.g. striker,

goalkeeper, etc) and the walking mode (part of *Motion Request* attribute).
The most relevant dependencies for the planning algorithm can be resumed into three components:

- The starting point of trajectory planning is given by the attribute *Robot Position* (absolute coordinates and orientation of the robot).
- The position of the target is given by the attribute *Motion Request*. This depends mainly on the role played. It is determined externally and is given explicitly as an input for the motion planning module.
- The obstacle data to avoid when heading towards the target.

The obstacle acquisition is a relevant part, and any deficiency in this process might lead to huge problems in the planning output. Each robot relies on its sensors to detect the different obstacles in its respective field of view. Environmental perception is based on camera data processing. The ball position can be detected with sufficiently good accuracy, however, the detection of other robots still needs to be implemented and improved. Therefore, sonar sensors are currently used to detect obstacles in the close vicinity of the torso and estimate their position. In case of failure of sonar sensors to detect some obstacles (e.g. fallen robot), foot collision is used as a backup way. However, obstacle data is not only provided by the robot sensors, but also through the communication with teammates. The different detected obstacles are merged into a commonly shared knowledge and lead to a more stable detection behavior. Furthermore, the static obstacles in the field have known global positions and can be therefore updated using the robot odometry (whose performance still needs to be improved). Currently, only the goalposts are supported.

The purpose of the motion planning module is to find a trajectory from the robot position to the target which avoids any collision. The module outputs the next action which should be performed by the robot. This action consists of a combination of translation and rotation. Both components are currently computed separately.
The rotation computation can be performed in two ways depending on the requested walking mode (either maintain a globally fixed orientation or adjust gradually the robot orientation until the target is reached, see the part 5.8.2 of the hulks report 2018 for more details).
The translation computation is the more challenging part and is the focus of this project. A brief overview of the old implementation is given in the following lines. The old implementation is based on a straightforward vector-based approach. The robot translation output vector is a weighted sum of a target vector and a set of displacement vectors. The target translation vector is a normalized vector that points towards the requested destination position, whereas each displacement vector is a normalized vector that points away from the respective obstacle in order to avoid a potential collision.

B. Bi-RRT algorithm: implementation details and integration challenges

After analyzing the different parts of the motion planning module and the performance of the obstacle avoidance vector-based approach, the latter is substituted by the new planning approach while keeping the overall code structure maintained. As discussed above, the module output is split in a rotation and translation computation part. Both parts are implemented as separate methods of the motion planning class. Since the rotation computation part works quite well and is not directly addressed by the Bi-RRT algorithm, no changes were made in the respective method. The method for translation computation takes into consideration the walking mode and calls accordingly the *ObstacleAvoidanceVector* method. Therefore, a new method is added to the class and performs the planning according to the Bi-RRT algorithm. In this way, the developer may choose which method should be called in the translation computation parts. It might be useful to consider different motion planning approaches depending on the scenario (target and obstacle positions).

The main contribution is the method *rrtOutputVector* which was structured in the same way as the method *obstacleAvoidanceVector* in terms of dependencies and output variable. The method parses the different inputs (starting point, target and obstacle vector) and initializes the trees as discussed in section II. Subsequently, the iterative process starts adding nodes to the tress until a threshold distance between at least two elements of both trees is reached. In this case, the final path is computed and the final output is approximately the normalized first derivative of the trajectory evaluated at the starting point (i.e. at the robot position). The method was developed in a modularized way (using help functions) and uses a set of internal state variables (declared as class attributes) and a set of parameters (e.g. initial step size).

IV. Experiments

This section describes conducted experiments during the lab. As a first step, the motivation behind each experiment will be stated and then the respective scenario.

Since testing trajectory planning directly on the real robot can be very costly in terms of possible accidents and damages. Our first experiments were driven in a

simulated environment. After testing the algorithm in different scenarios in the simulation, the approach was tested on the real robot. For this reason, this part will be divided into two subsections: the first one dedicated to the simulation environment and the second one dedicated to the real robot.

Simulation Environment

As a simulation tool, we used SimRobot. For the sake of testing the robustness of our implementation and validating the added value of Bi-RRT in comparison to the traditional method, we set up three different testing levels, describing the ability of the robot to:

- reach the target without any obstacle
- avoid static obstacles
- avoid dynamic obstacle in the role of a striker and a defender.

Path Planning to Target

By absent obstacles, it is expected from the motion planning algorithm to reach the target in a smooth and optimal trajectory. To test the RRT on this level, the ball position was fixed in the field and the robot has to move to its position.

Static Obstacle avoidance

After we made sure that RRT is reaching the target correctly, a more challenging scenario will be introduced. The focus is now on how can the robot avoid two static robots between its initial position and the target position. Two aspects are interesting in this scenario: Would the robot hit the obstacles? How will the final path look like and how optimal it is?

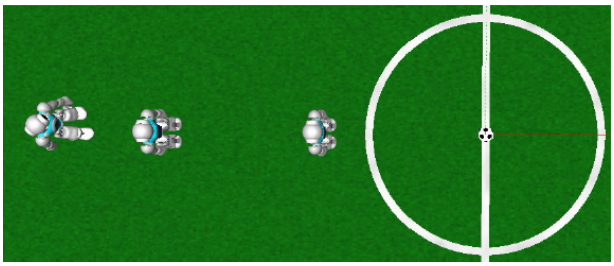

Fig. 3: Static obstacle avoidance

Dynamic Obstacle avoidance

As results were promising in the previous test, we moved to a more realistic scenario in which obstacles are changing their relative position to the robot. This is relevant for the robot in two different roles: the striker trying to avoid the defenders and reaching the goal and the defender trying to stop the striker. The different roles were tested independently in three different scenarios:

- A striker robot working with Bi-RRT trying to avoid two defenders working with the old approach.
- Two defenders working with the Bi-RRT trying to stop a striker working with the old approach.
- A striker against two defenders working all with the Bi-RRT algorithm.

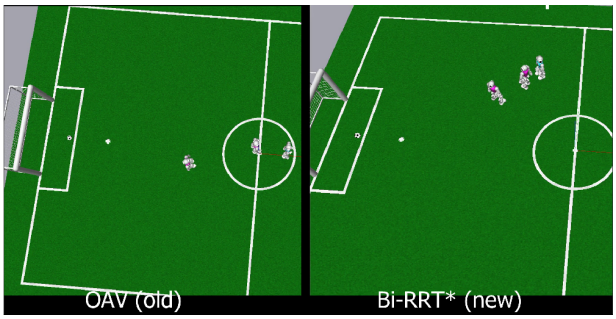

Fig. 4: Dynamic obstacle avoidance

Integration in real robot

After testing the approach in a simulated environment, it is interesting to see how it performs on the real robot. The detection of the robot is accurate only for the ball. Therefore we set the ball as an obstacle that we will actively change its position and put it on the robot's way as long as the robot is avoiding it until it reaches the target. The test is visualized by the figure 5.

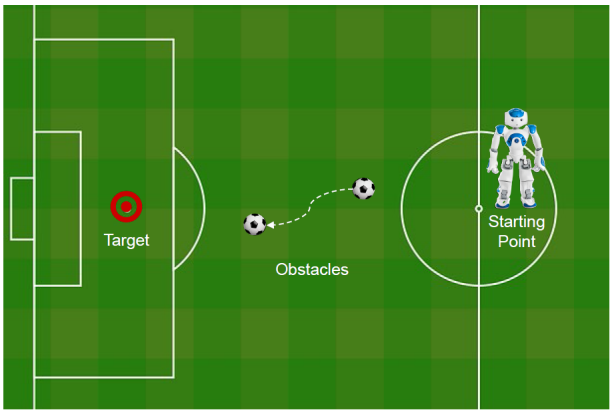

Fig. 5: Integration in real robot

V. Results and Discussion

In this section, the results of static and dynamic obstacle avoidance will be presented.

Static obstacle avoidance

As a first step, the quality of the generated trees by the Bi-RRT will be assessed. For this purpose, the sequence of random points and the planned path to target are exhibited by figure 7.

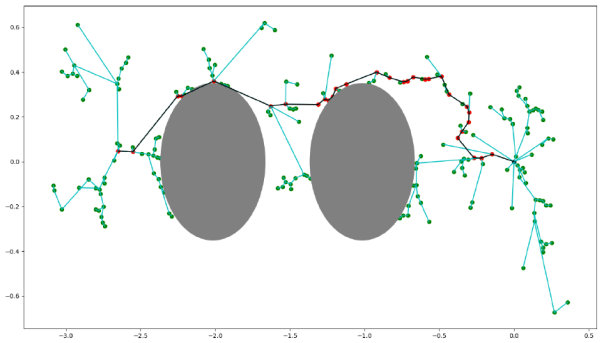

Fig. 6: generated random tree and path to target

The grey circles in the figure show the static obstacles to escape, green points represent explored random points and the black line is the path planned to target at the sampling time.

We can see from the plot that Bi-RRT is exploring the field in many different directions, and the generated path is minimizing the distance to the target and avoiding the obstacle smoothly. As Bi-RRT changes the taken path over time due to new explorations, the final path will be analyzed and compared to the one taken by the original motion planning approach. Both final paths are shown by the figure.
The first shown path is generated by the old implementation and the second plot is for the Bi-RRT path. The first approach is able to avoid the two obstacles but it is taking a longer path than the one taken by Bi-RRT.
To see this aspect more concretely, the number of taken steps from an initial position to target and the elapsed time will be studied.

	Old Approach	Bi-RRT*
Time from starting position to target [s]	142.335	113.716
Number of steps from starting position to target [steps]	2604	2448

TABLE I: Comparison between old and new algorithm

The Bi-RRT* is overtaking the vector-based implementation on the two comparison levels. So we can clearly say that the new implemented method is better than the old one in the context of static obstacles. It will be interesting to see if it will show the same performance in the dynamic obstacle case.

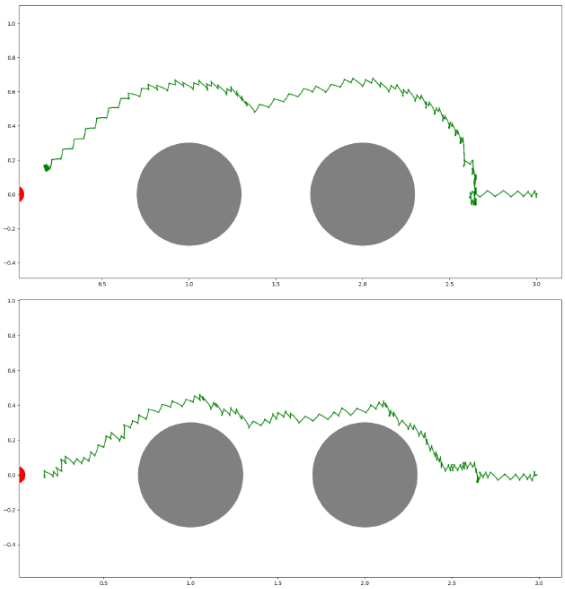

Fig. 7: generated final paths to target

Dynamic obstacle avoidance

The Bi-RRT* is overtaking the vector-based implementation on the two comparison levels. So we can clearly say that the newly implemented method is better than the old one in the context of static obstacles. It will be interesting to see if it will show the same performance in the dynamic obstacle case.
The conclusion derived from this test is that Bi-RRT* make the robot more robust in both roles: Striker and defender.

Integration in real robot

The first thing to mention regarding this test is that the new trajectory planning approach worked successfully on the real robot and succeeded to avoid the moving ball.
Second, the computation cost of Bi-RRT will be assessed. Motion planning is part of the brain module which runs at 60 Hz. Time spent in each cycle was tracked and it is presented by the figure below:

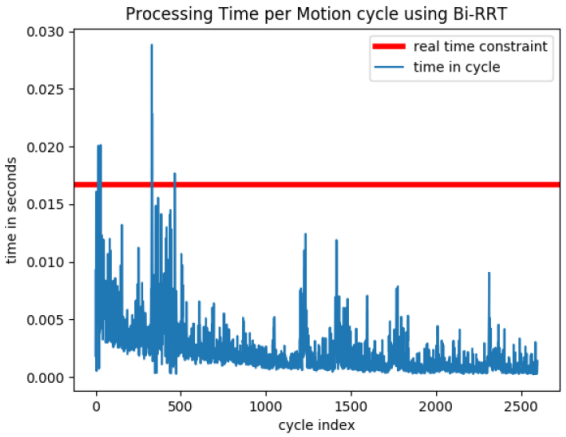

Fig. 8: Processing Time per Motion cycle using Bi-RRT

We can see that in most of the cases, Bi-RRT fulfills the time constraint and the average processing time is equal to **1.924 ms**. Furthermore, it is interesting to see how often a complete path is generated from the initial point to the target. This frequency indicates how often the robot moves forward.

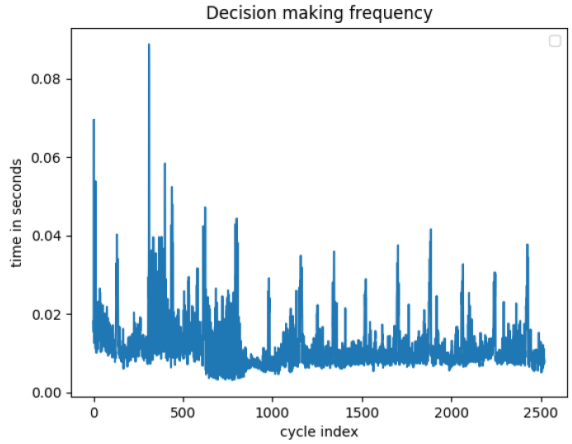

Fig. 9: Decision making frequency

In average a decision is taken each **13.153 ms**. The average is also below the computation time of the brain module.

VI. Conclusion

This work presents a better method for trajectory planning in a robot soccer game. The Bi-RRT* algorithm was used to drastically shorten the travel distance and computing time. It showed a robust avoidance of both static and dynamic obstacles. However, this algorithm cannot consider statistics of observations. An improved version of RRT which takes into account the prediction uncertainties is the so-called Kalman-RRT [9], which could lead to better path generation. Since the generated path is a linear piece-wise function, an improvement could be done by considering smoothing curves. B-Splines functions and Kalman-RRT could improve the accuracy of the planning resulting in more optimal and realistic behavior.

References

[1] L. E. KavrakiyAbstra, "A lazy probabilistic roadmap planner for single query path planning," 2007.

[2] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," tech. rep., 1998.

[3] Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J. P. How, "Real-time motion planning with applications to autonomous urban driving," IEEE Transactions on Control Systems Technology, vol. 17, pp. 1105–1118, Sep. 2009.

[4] S.-J. Lee, S.-H. Baek, and J.-H. Kim, "Arm trajectory generation based on rrt* for humanoid robot," in Robot Intelligence Technology and Applications 3 (J.-H. Kim, W. Yang, J. Jo, P. Sincak, and H. Myung, eds.), (Cham), pp. 373–383, Springer International Publishing, 2015.

[5] J. Bruce and M. Veloso, "Real-time randomized path planning for robot navigation," in IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 3, pp. 2383–2388 vol.3, Sep. 2002.

[6] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[7] A. H. Qureshi and Y. Ayaz, "Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments," Robotics and Autonomous Systems, vol. 68, pp. 1–11, 2015.

[8] X. Zang, W. Yu, L. Zhang, and S. Iqbal, "Path planning based on bi-rrt algorithm for redundant manipulator," in 2015 International Conference on Electrical, Automation and Mechanical Engineering, Atlantis Press, 2015.

[9] Y. Chen and M. Liu, "Rrt* combined with GVO for real-time nonholonomic robot navigation in dynamic environment," CoRR, vol. abs/1710.07102, 2017.