AUTONOMOUS AND MOBILE ROBOTICS
Course Project 2017/2018
Anytime asymptotically-optimal motion planning

Alessandro Santopaolo ,Cheikh Melainine Elbou,Emanuele Caruso

December 2018

SAPIENZA
UNIVERSITÀ DI ROMA

This report is submitted in fulfilment of the requirements for the course
'"AUTONOMOUS AND MOBILE ROBOTICS" by Prof. Giuseppe Oriolo,
Sapienza University of Rome

1 Abstract

The goal of our project is to use an optimal motion planning algorithm to move a mobile that is assigned a navigation task without taking into account the dynamic constraints of this robot. The optimal motion planning algorithm chosen is the anytime motion planning based on the RRT*, which is a sampling-based algorithm with an asymptotic optimality property. Contrarily to the basic RRT, it accounts for the quality of the generated solution trajectories. On the other hand, it introduces some computational overhead that is problematic when planning under time limitations. The anytime version of RRT* addresses this issue by quickly computing an initial sub-optimal solution and repeatedly improving it during the simultaneous execution. The proposed motion planning algorithm was thoroughly evaluated in several scenarios of increasing complexity. We programmed this algorithm in $C++$ and we tested it on the mobile robot $Youbot$ which is available in the V-rep simulator.

2 Introduction

Motion planning algorithms are used in many fields including bioinformatics, character animation, videogames AI, computer-aided design and computer-aided manufacturing , architectural design, industrial automation, robotic surgery, and single and multiple robot navigation in both two and three dimensions.
The problem of motion planning can be identified as: Given a start pose of the robot, a desired goal pose, a geometric description of the robot and a geometric description of the world, the objective is to find a path that moves the robot gradually from start to goal while never touching any obstacles.
From a computational complexity point of view, even a simple form of the motion planning problem is PSPACE-hard [6], which means that any complete algorithm is computational difficult to analyze.
In order to face this problem and find a solution that achieve computational efficiency, there are several practical motion planning algorithms that relax the completeness requirements. Most sampling-based algorithms have been proven to be probabilistically complete, i.e., the probability that the algorithm finds a solution, if one exists, converges to one as the number of samples approaches infinity. They are able even in the high dimensional state space to find a feasible motion plan relatively quickly if it is exit. Those practical motion planning algorithms include Probabilistic RoadMap (PRM) [7] ,and it is one of the most significant categories of motion planning approaches studied in the literature ,and RRT [5] which is practical algorithm for motion planning on state of art in robotic platforms, because it can handle system with differential constraints.
The RRT algorithm has the ability to find a feasible solution if it exit ,but like Karaman and Frazzoli proved in [2], if the number of samples increases the probability of the RRT algorithm converging to an optimal solution is actually zero. This characteristic make this algorithm less useful in practical situations.
The problem could be solved by an alternative method, RRT^*, a sampling-based algorithm with the asymptotic optimality property, i.e., almost-sure convergence to an optimal solution, along with probabilistic completeness guarantees. RRT^* provides substantial benefits, especially for real-time applications. Like the RRT, RRT^* quickly finds a feasible motion plan.
Moreover, it improves the plan toward the optimal solution in the time remaining before plan execution is complete. This refinement property is advantageous, as most robotic systems take significantly more time to execute trajectories than to plan them. In such settings, asymptotic optimality is particularly useful, since the available computation time as the robot is mov-

ing along its trajectory can be used to improve the quality of the remaining portion of the planned path. An anytime motion planning algorithm can be achieved in different ways. The first one, proposed by Frazzoli in [4] and adopted in our project, is based on optimality criteria. A system based on this anytime planning overlaps two functions in time: execution of its current plan (some initial portion of), and computation to replace (any pending portion of) the current plan with an improved plan. So from the beginning the planner,after the first time budget, find a complete solution, actually not an optimal path, that could bring the robot to the goal and during the motion improves it. On the contrary the other method, proposed by Ferrari, Oriolo and Cognetti in[1] for any planning interval gives rise only to a partial solution of the motion. The algorithm interleaves planning and execution intervals: a previously planned whole-body motion is executed while simultaneously planning a new solution for the subsequent execution interval. At each planning interval, differently from the previous anytime paradigm, this algorithm focuses on obtaining the best solution, guaranteed to be valid in the next execution interval, among those that the planner has been able to produce within the deliberation time, without relying on continuous refinements of the initial solution. The overall results to be constituted by a sequence of on-line computed short horizon solutions.
In our project , we leverage the anytime asymptotic optimality property of the RRT* algorithm to improve the online convergence of the plan during execution. We implement such approach for the case of a mobile robot, a Youbot, that is assigned a navigation task. We make a performance assessment of the method testing the algorithm in scenarios of increasing complexity on V-rep environment.
In the following sections we will discuss these arguments: some basic background about RRT ,RRT^* and its anytime version, in section three; simulation result for the different scenarios in section four and conclusion in section five.

3 Background

The goal for our project is to use an optimal motion planning to moving a mobile robot. In order to achieve this goal we used the algorithm introduced in the paper [4], which is an anytime motion planning based on $RRT^*$with two extensions, committed trajectory and branch and bound , for more efficiency in the online implementation. In order to understand this algorithm we need to describe the RRT and RRT^* algorithms and the extensions to it.

3.1 RRT Algorithm

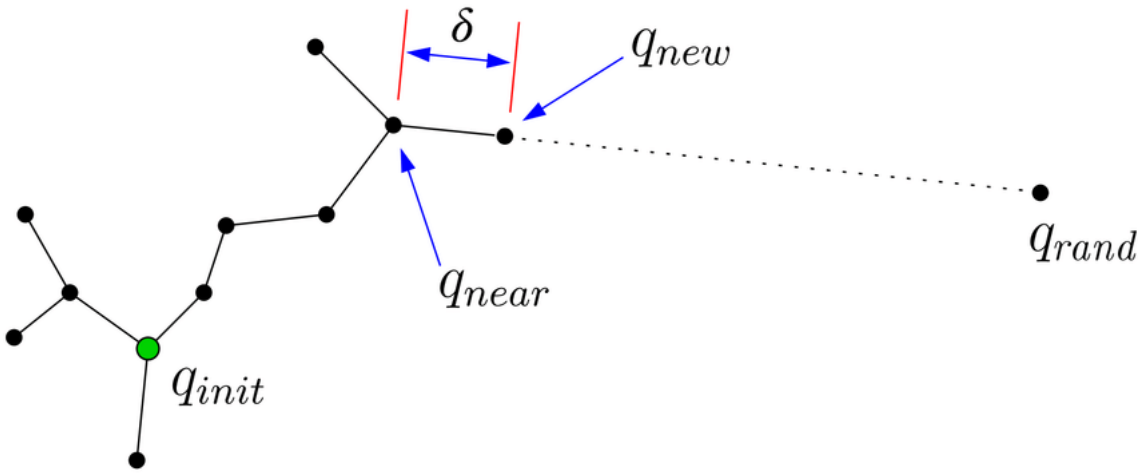

Figure 1: Tree construction process of the RRT algorithm

Let's define
$C = SE(2) = \mathbb{R}^2 \times SO(2)$: the configuration space
notice that if $q \in C$ then $q = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$ where $\begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2$ and $\theta \in SO(2)$
$C_{ob} \subseteq C$: the obstacles region
$C_{free} = C - C_{obs}$: the free region
$C_{goal} \subseteq C_{free}$: the goal region
Let's define the tree $T = (V, E)$ where V is the set of vertex and $E \subseteq V \times V$ the set of edges . We assume that the vertex set V are states from the C_{free} connected by directed $E \subseteq V \times V$.
The RRT (Rapidly-Exploring Random Trees) as described in [5],constructs a tree using random sampling in search space . The tree start from an initial state $q_{init} \rightarrow C_{free}$ and expands to find a path towards a configuration $q_{goal} \in C_{goal}$. The tree gradually expands as the iteration continue. During each iteration, a random state q_{rand} is selected from the configuration space C, then a nearest node say $q_{nearest}$ is searched in the tree T according to a defined metric ρ. Then by using a steer function, a new state q_{new} is generated according to a predefined step size δ, on the segment joining $q_{nearest}$ to q_{rand}. then if q_{new} is accessible to $q_{nearest}$ tree is expanded by connecting q_{new} and $q_{nearest}$. A Boolean collision checking process is performed to ensure collision free connection between tree nodes q_{new} and $q_{nearest}$. The process stops when a path is found between q_{init} and q_{goal}, or if the predefined time period expires . Node expansion process is described in Figure 2.

T=(V,E) ← RRT(q_{init})

```
1 T ← InitializeTree()
2 T ←InsertNode(Ø,q_init,T)
3 for i=0 to i=N do
4 q_rand ← Sample(i)
5 q_Nearest ← Nearest(T,q_rand)
6 (q_new,U_new) ← Steer(q_nearest,q_rand)
7 if ObstacleFree(q_new) then
8 T ← InsertNode(q_nearest,q_new,T)
9 Return T
```

Figure 2: RRT algorithm

Further detail of some major functions is described as the following:

Sample: This function generates a random configuration $q_{rand} \in C$.
Nearest: This function returns the nearest node from T=(V,E) to q_{rand} according to a distance metric ρ.
Steer: This function computes q_{new} at a distance δ from $q_{nearest}$ on the segment joining $q_{nearest}$ to q_{rand}.
Obstaclefree: This function is used to collision checking on the tree edge, and returns true if every point of that edge lie in obstacle free region, i.e., whether a path $q : [0, T]$ lies in the Q_{free} for all t=0 to t=T.
InsertNode: This function adds a node q_{new} to V in the tree $T = (V, E)$ to connect node $q_{nearest}$ as its parent.

3.2 RRT* Algorithm

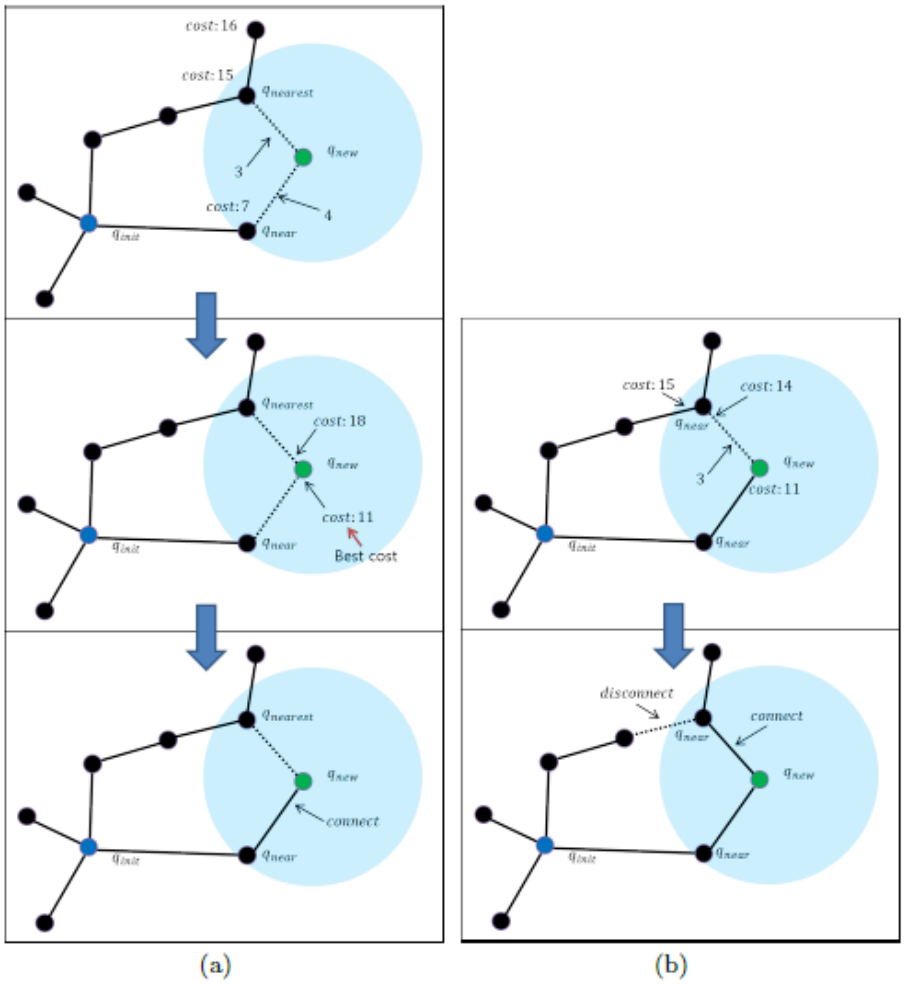

Figure 3: *RRT** methods (a) finding the parent point (b) rewiring.

*RRT** algorithm was introduced in [2] by Sertac Karaman and Emilio Frazzoli. It solve the problem of RRT algorithm for which it not converges to an optimal path as the number of samples increases. The cost of the path returned by *RRT** converges to the optimal value and it maintain a tree structure like *RRT*.
While *RRT** inherits all the properties of *RRT* and works similar to *RRT*. It has two novel features called *choose parent* and *rewire*. *choose parent* finds the best parent node for the new node before its insertion in tree. This process is performed within the area centered at q_{new} of a ball of radius

defined by.

$$k = \gamma \left(\frac{log(n)}{n} \right)^{\frac{1}{d}}$$

where d is the search space dimension and γ is the planning constant based on environment. *rewire* rebuilds the tree connections within this radius of area k to maintain the tree with minimal cost between tree connections.
Space exploration and improvement of path quality is shown in Figure 3. As the number of iterations increase, RRT^* improves its path cost gradually due to its asymptotic quality, whereas RRT does not improves its jagged and suboptimal path. The details of the two new features introduced in RRT^* are as follows:

- Rather than selecting the nearest node as the parent, the RRT^* look upon all nodes q_{near} in a neighborhood of q_{new}, and each of all belongs to the subspace $C_{near} \in C$. In fact, the algorithm calculates the cost of selecting each q_{near} as the parent. This operation evaluates the total cost as the additive combination of the cost associated with reaching the potential parent node and the cost of the trajectory to q_{new} . The node q_{min}, which is the one that lead to the lowest cost, becomes the parent as the new node is inserted in the tree.

- The rewire algorithm checks each node q_{near} in the proximity of q_{new} to verify whether reaching it via q_{new} would achieve lower cost than doing so via its current parent. When this relationship decreases the total cost related to q_{near}, the algorithm rewires the tree to make q_{new} the parent of q_{near}.

Thanks to these features the path quality is improved without incurring substantial computational overhead and the algorithm achieves asymptotic optimality property, almost-sure convergence to an optimal solution [3].
RRT^* algorithm is described in Figure 4. In RRT^* we used the same functions used in RRT algorithm and we add the two functions of *choose parent* and *rewire* that works like explained above.

Algorithm 1: $T=(V,E) \leftarrow RRT^*(q_{init})$

```
T ← InitializeTree()
T ← InsertNode(∅, q_init, T)
for i=0 to i=N do
    q_rand ← Sample(i)
    q_nearest ← Nearest(T, q_rand)
    (q_new, u_new, T_new) ← Steer(q_nearest, q_rand)
    if ObstacleFree(q_new) then
        q_near ← Near(T, q_new, |V|)
        q_min ← ChooseParent(Q_near, q_nearest, q_new, x_new)
        T ← InsertNode(q_min, q_new, T)
        T ← ReWire(T, Q_near, q_min, q_new)
Return T
```

Algorithm 2: $q_{min} \leftarrow ChooseParent(C_{near}, q_{nearest}, x_{new})$

```
q_min ← q_nearest
c_min ← Cost(q_nearest)+c(x_new)
for q_near ∈ Cnear do
    (x',u',T') ← Steer(q_nearest, q_rand)
    if ObstacleFree(x') and x'(T')=q_new then
        c'= Cost(q_near)+c(x')
        if c'< Cost(q_new) and c'< c_min then
            q_min ← q_near;
            c_min ← c';
Return q_min
```

Algorithm 3: $T \leftarrow ReWire(T, C_{near}, q_{min}, q_{new})$

```
for q_near ∈ Cnear/{q_min} do
    (x',u',T') ← Steer(q_nearest, q_rand)
    if ObstacleFree(x') and x'(T')=q_near and
    Cost(q_new)+c(x') < Cost(q_near) then
        T ← ReConnect(q_new, q_near, T)
Return T
```

Figure 4: RRT^* Algorithm

4 Anytime RRT* Algorithm

For practical point of view any robot should operate within limited real-time computational resources. An anytime motion planning algorithm based on RRT^* will gives us an optimal solution and for more efficiency in the online implementation we used two extensions keys *Committed Trajectory* and *Branch and Bound* [4].
The online planning algorithm is done in two phases. In the first phase ,which is the initial planning phase, we run the RRT^* for a small amount of time until the robot must start moving toward its goal.
Once the initial phase is completed the algorithm starts the second phase which is the iterative phase. This phase is explained as follow: The robot start to execute a portion $q : [0, t_{com}]$ of the given motion plan $q : [0, T] \rightarrow C_{free}$ generated by the RRT^* algorithm ,which is called the committed trajectory. At the same time declares $q(t_{com})$ to be the new root of the tree, deleting the branches before $q(t_{com})$. This effectively shields the committed trajectory from any further modification. While the robot is executing the committed trajectory, the RRT^* algorithm in the uncommitted trajectory continue to improve the path toward the goal. At the end of the execution of the committed trajectory the procedure is repeated, so the previous $q(t_{com})$ becomes the root of the new committed trajectory, that ends at a new $q(t_{com})$. The iterative phase repeats until the robot reaches its goal.

4.0.1 Further improvement

Let's first define $T = (V, E)$ to be a tree and $q \in V$ be a vertex in T. In order to give more details about this technique we will define first this two functions as follow:
1)$Cost(q)$: denotes the cost to reach a node q of the tree T from its root q_{init}. The cost of q, is simply the sum of the length of the edges of the shorter path, that connects q_{init} to q.
2)$CostToGo(q)$: this function gives the euclidean distance in *(x,y)*, from the node q, to the goal q_{goal}.
Let's define the function $Branch - and - bound$: let $q \rightarrow V'$ such that V' is the subset of nodes from V satisfying the following properties:

$$V' = \{q \in V | Cost(q) + CosToGo(q) \geq Cost(q_{goal})\}$$

then all $q \in V'$, are removed from the tree T, because they can't allow a better path than the current one. This function can entails an improvement on performances, reducing the dimension of the tree.

5 Implementation

After understanding the behavior of the anytime RRT* technique, we have tested the efficiency of the algorithm implementing it on V-rep.
V-rep makes available a lot of models that can be added in the scenario, and for this project those models are used. In fact, the scenario is made by a flat map of 5x5 meters, a *youBot* (from *KUKA Robotics*), a disc with a diameter of 0.5 meters that represent the goal region, and some collectible obstacles depending on the specific scenario.
In our framework, the configuration of the youBot is described by a vector:
$q \in C$, where: $C = SE(2) = \mathbb{R}^2 \times SO(2)$, then: $q = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$
where: $\begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2$ and $\theta \in SO(2)$.
The distance metric used in the implementation is the euclidean distance in $\begin{pmatrix} x \\ y \end{pmatrix}$, and we have computed the angle θ of q_{new} along the direction from its parent, in order to find a smoother path. Reminding that this is a pure navigation problem, the meaning under the choice of the *youBot* is that this robot has *sweedish wheels*, which are omnidirectional, in order to not be subject to non-holonomic constraints. Indeed, the motion of the youBot inside V-rep, is not controlled by its input, but is just a roto-translation of the model along all the points of the path, obtained through interpolation.
Finally, the anytime RRT* algorithm has been implemented as showed in the previous section, including also the *branch and bound* improvement.

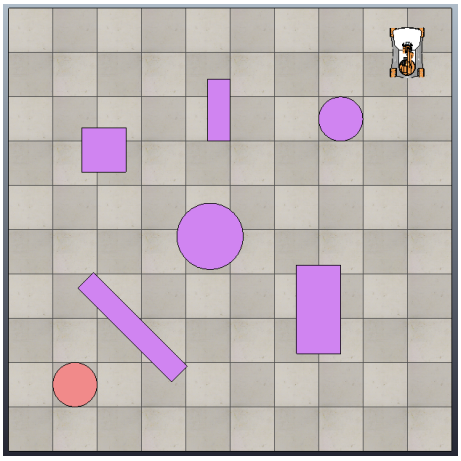

Figure 5: One of the scenarios used for this project. In the upper-right corner the youBot can be seen, in the lower-left corner the red disc represent the goal region, and all the other purple shapes are the obstacles of this scenario.

5.1 Implementation details

The algorithm has been implemented using the $C++plugin$ for $V-rep$ making use of the regular $APIfunctions$. The framework can be conceptually divided in two different sections, planning and execution.
During the planning phase, the RRT* algorithm is runned directly in the SIM thread (the thread that communicate directly with the application of V-rep), and at each iteration the new node (and the connection from his parent) is plotted on the map with the primitive shape of a disc (and a rectangle) with a green color. The state of the tree is represented with 4 different dynamic vectors (*std vector* of C++) inside the script:

- **representation of the nodes**: Each element of this vector contains a three-dimensional vector q=*(x, y, θ)* (using the *Eigen* library), in which the *(x,y)* pair is the flat coordinate of q and θ is the rotation angle of the youBot with respect to the z axis. This representation permit to get the position and the orientation of the youBot in the space with only 3 variables.

  While the *(x,y)* pair of q is generated by the algorithm with respect to q_{rand} , its $q_{nearest}$, and the step size δ; the angle θ is computed with respect to the direction of the vector that starts from the parent node and ends to the current node, using the $ATAN2$ function. This method allows to the youBot to rotate itself to "look toward" the path direction, having a smooth path.

- **index of parent nodes**: Each element of this vector contains an int-type variable, which is the index of the parent node of the current node.
- **indexes of children nodes**: Each element of this vector contains another dynamic vector (still *std vector*), which contains all the children of the current node (of course may be an empty vector if the node hasn't children).
- **distance from parent**: Each element of this vector contains a float variable which is the distance from the parent node (this may be redundant because the distance can be computed each time knowing the parent node index and its representation).

In order to check collisions inside the planning function, has been created an invisible clone of the model of the youBot, and then it's translated and rotated in points in which the collision checking is done, using the API function of V-rep: *simCheckCollision*, that works simply checking if the models are intersecting.
During the planning, if a node of the tree is dropped into the goal region, then that node will be the q_{goal}, that is the node in which the youBot will terminate its motion, and once that q_{goal} is found, it will not change. At the same time, when q_{goal} is found, means that exists a path that links it to the root node q_{init}. In fact, this path is plotted with a different color (blue) from the one of the tree (green). While the planning function runs, with the extension and the rewiring of the tree, the path can be improved, so at each iteration the algorithm checks if the path is improved, and in that case, the path will be saved and replotted in the correct way. The nodes of the path are discovered going backward, parent by parent, from q_{goal} to q_{init}.
Furthermore, regarding the *branch and bound* function, it is called:

- Firstly when q_{goal} is found, on every node $q \in T$.
- On q_{new} when it's generated by the anytime RRT* algorithm.
- When the current path to the goal is updated, on every node $q \in T$.

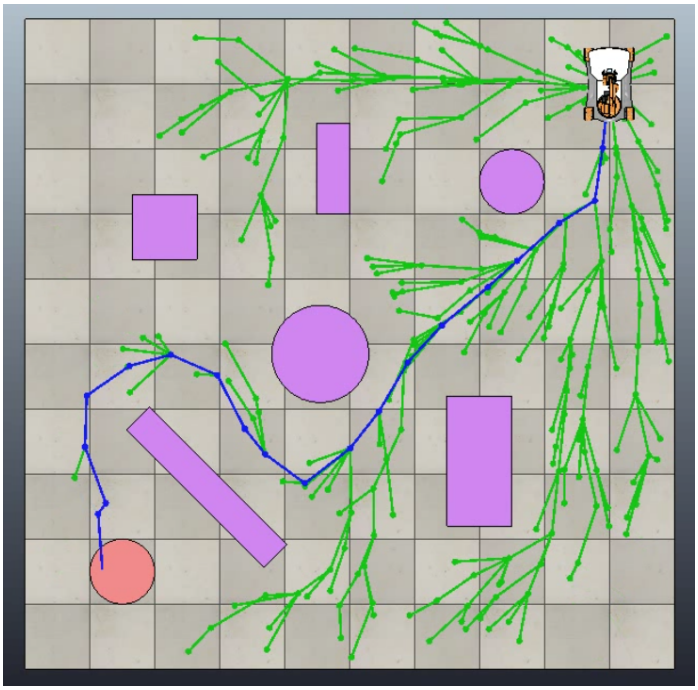

Figure 6: The image shows the status of the tree after having found a solution during the planning phase.

The planning phase finish at the expiration of the time budget, which is set to 30 seconds in this project. Furthermore, the time is referred with the *simulation time* of V-rep, which is accessible through the regular API function *simGetSimulationTime*. The default time step of 50 milliseconds has been used. So sometimes, the simulation time spent may not be equal to the real one.
After this time budget, if a solution is not found then the algorithms stops, else starts the second phase of execution.
The execution phase entails an execution and a planning function. The execution function starts the motion of the youBot toward the current path, interpolating between nodes with a constant of 80 points per meter, in order to maintain the same velocity for the youBot, independently from the length of the interval between two nodes.
However, having an anytime algorithm, during the motion the youBot still plans in order to improve his path. In order to do this it's been used a multi-threading approach, using the *POSIX Thread* library. In fact, the execution function is runned in the SIM thread, while the planning function is runned into a parallel pthread, in order to have the motion of the youBot which is independent and parallel from the planner.

In this phase, the planning function is different from the previous one, because q_{init} changes, being the next first node of the current path, and at the same time, the algorithm have to "prune" the branches that the youBot has surpassed, using a recursive function that removes the children from the tree and recall itself for them, starting from the previous q_{init} and stopping for the new one. Furthermore, the function that plots nodes and links of the tree on the scenario, is not in the planning function, because shapes can be created only in the SIM thread. Therefore, the plotting function is inserted in the execution function, and in order to not slow down the motion, the nodes are plotted only when the youBot reaches a node. So the RRT* algorithm always runs, but the status of the tree and the path, is plotted only at the achievement of a path's node from the youBot. Finally, for the plotting function during the execution phase, the path surpassed is painted in red.

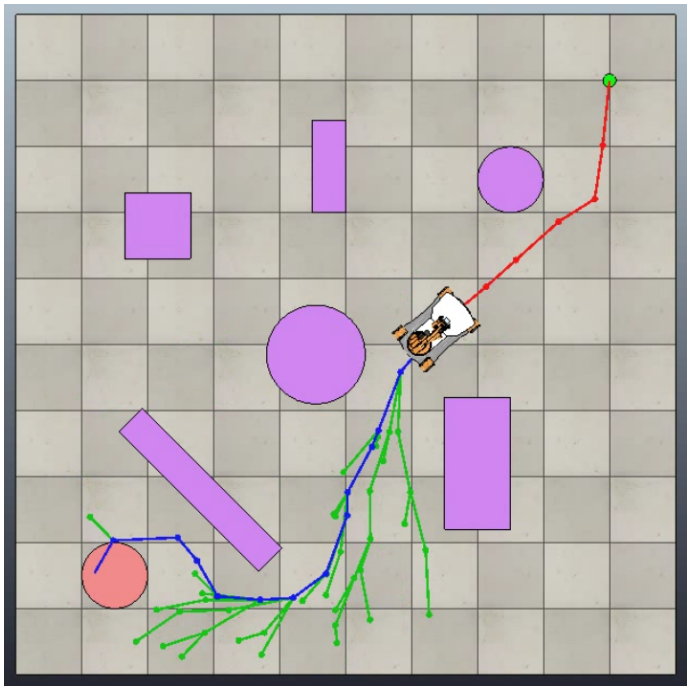

Figure 7: The image shows the status of the tree of the same run of the previous image, but during the execution phase.

Working with parallel threads that use the same global variables can be dangerous, so we have dealt with this problem using a *FIFO queue* (made with an std vector), in order to exchange information between threads.
When the youBot finally reaches the goal, the algorithm stops, and its "performance" related to the scenario and to the respective run, are measured in term of *cost for each path found*, sorted chronologically with the respective

simulation time for each of all. The cost of the path is simply the total meters of the path, and is computed at each time using the "distance from parent" vector.

5.2 Planning Experiments

Once the implementation of the code has been done, we have started some experiments, in order to study the asymptotic optimality convergence of the algorithm, considering different features for each run: scenario, number of nodes for the committed trajectory and time budget.
The scenarios used for the experiments are 3: the *basic scenario* allows different paths with similar cost, the *cluttered scenario* that leads the youBot to follow a restricted path, and the *indoor scenario* that wants to represent a real environment having shapes of real objects.
For each combination of those features 5 experiments has been done, and for those is calculated the average and the standard deviation of:

- time of the first solution found (*seconds*)
- cost of the path for the first solution (*meters*)
- cost of the path after time budget (*meters*)
- cost of the final path (*meters*)

5.2.1 Basic scenario

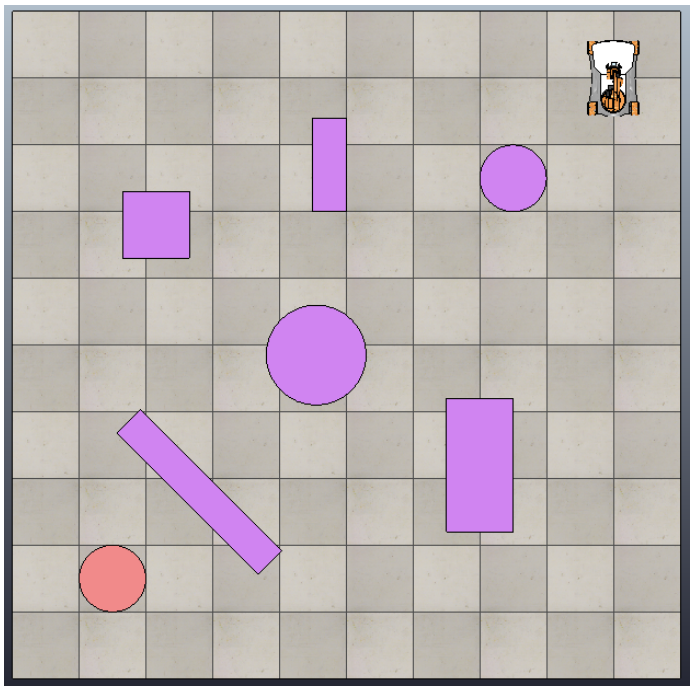

Figure 8: As can be seen by the image, there are 7 different obstacles in this scenario, disposed quite homogeneously in the space. This allows to have more than one path with a similar cost.

time of the first solution found *(s)*	18.86 ± 9.28		
time budget: 30 seconds			
	length of the committed trajectory		
	1 node	3 nodes	5 nodes
cost of the path for the first solution *(m)*	6.31 ± 0.24	6.92 ± 0.53	6.77 ± 0.81
cost of the path after time budget *(m)*	6.18 ± 0.06	6.51 ± 0.56	6.55 ± 0.52
cost of the final path *(m)*	6.16 ± 0.07	6.46 ± 0.54	6.51 ± 0.52
time budget: 60 seconds			
	length of the committed trajectory		
	1 node	3 nodes	5 nodes
cost of the path for the first solution *(m)*	6.65 ± 0.84	6.50 ± 0.23	6.65 ± 0.42
cost of the path after time budget *(m)*	6.07 ± 0.18	6.11 ± 0.10	6.09 ± 0.07
cost of the final path *(m)*	6.06 ± 0.19	6.10 ± 0.08	6.06 ± 0.06

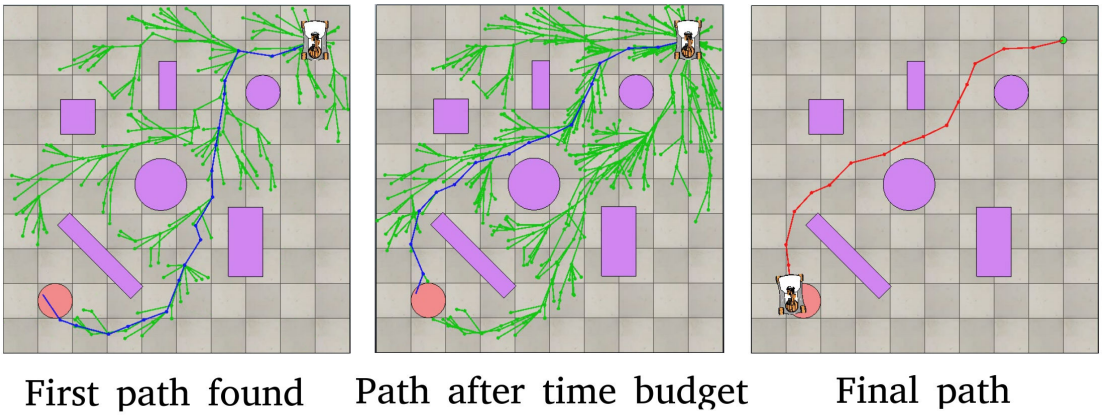

Figure 9: Screenshots for an experiment in the basic scenario.

As we said previously, for each combination of the length of the committed trajectory and the time budget, the average of the costs are computed with 5 experiments, while for the time of the first solution found, the average and the standard deviation are computed considering all the 30 experiments done for the scenario.
As can be seen from the table, in general the best final cost path is reached with a time budget of 60 seconds, but at the same time, with the half of time budget, a solution has been always achieved, and it may have a similar cost. Another remark to do, is that with the time budget of 60 seconds, the path is not soo much improved during the execution, because the most of the optimality is reached during the planning phase due to the high time budget. In the other hand, can be seen that the cost improves during the execution with a time budget of 30 seconds, even if is not crucial as in the planning phase.
Furthermore, considering the fact that the radius of the goal area is 0.25 meters, comparing different path costs of different runs with different q_{goal}, it introduces an inaccuracy on the average of the costs. For this reason we can assert that the results are quite similar changing the length of the committed trajectory, because their differences are comparable with the diameter of the goal area, and so, the length of the committed trajectory isn't a very influent feature.
Finally, the standard deviation is more high for the costs of the first solution found, this is explainable by the fact that the scenario allows multiple paths, and so the first solution can vary at each time. Nevertheless, the standard deviation tends to be reduced trough time due to the convergence to the optimal solution.

5.2.2 Cluttered scenario

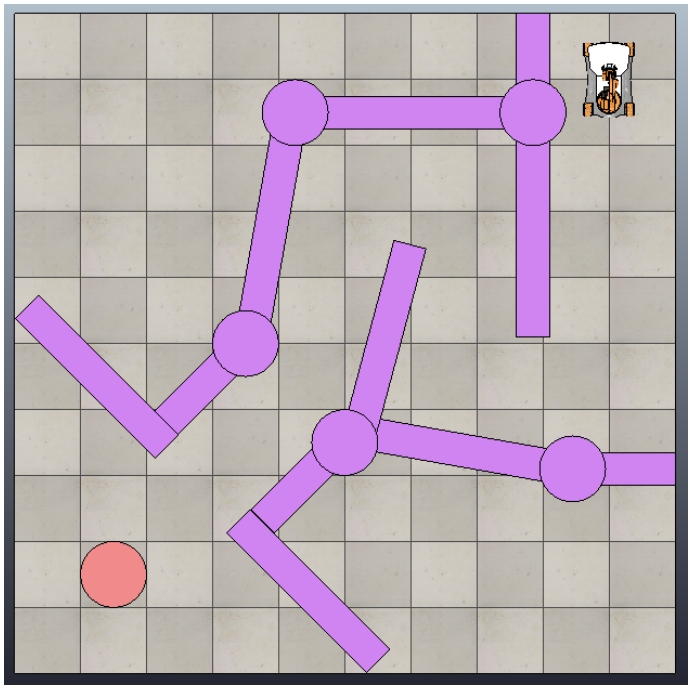

Figure 10

time of the first solution found *(s)*	42.35 ± 10.77		
time budget: 60 seconds			
	length of the committed trajectory		
	1 node	3 nodes	5 nodes
cost of the path for the first solution *(m)*	8.42 ± 0.07	8.29 ± 0.04	8.28 ± 0.51
cost of the path after time budget *(m)*	8.35 ± 0.08	8.23 ± 0.08	8.09 ± 0.22
cost of the final path *(m)*	8.27 ± 0.06	8.20 ± 0.08	8.07 ± 0.23

The table shows that in general the time of the first solution found is higher than 30 seconds, in fact in 15 experiments, only 2 times the solution was less than 30 seconds. Because of that, only the time budget of 60 seconds is considered. The reason under this result, is that the scenario is more complex than the previous, with an high number of obstacles that forbid the tree to expand freely, so the probability to discard a generated node is very high due to the several obstacles, that are close each other.
Looking at the results, can be seen that the costs are very similar changing the length of the committed trajectory, and during time can't be seen significant

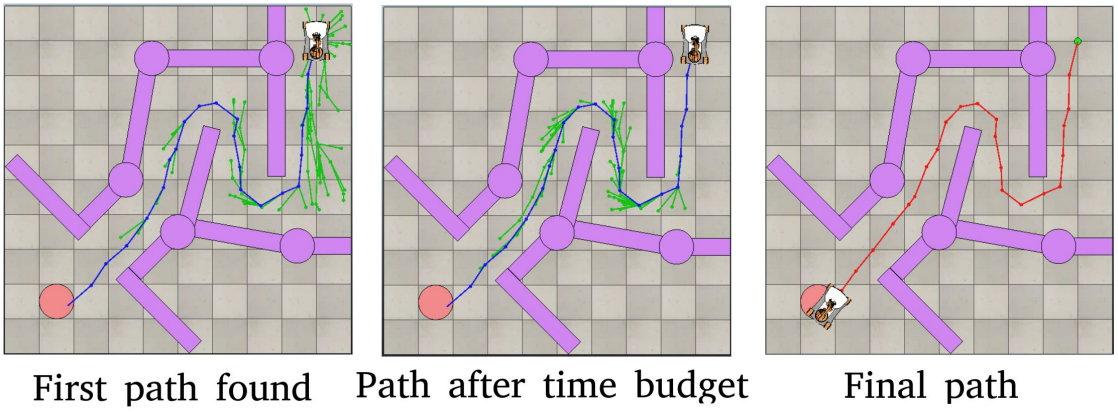

Figure 11: Screenshots for an experiment in the cluttered scenario.

improvement (it can be seen also in the screenshots). This is due to the scenario that admit always the same path (of course saying that the path is always the same is a simplification, because the paths found are different due to the "large road" set up by the obstacles, but they are very similar from each other).

5.2.3 Indoor scenario

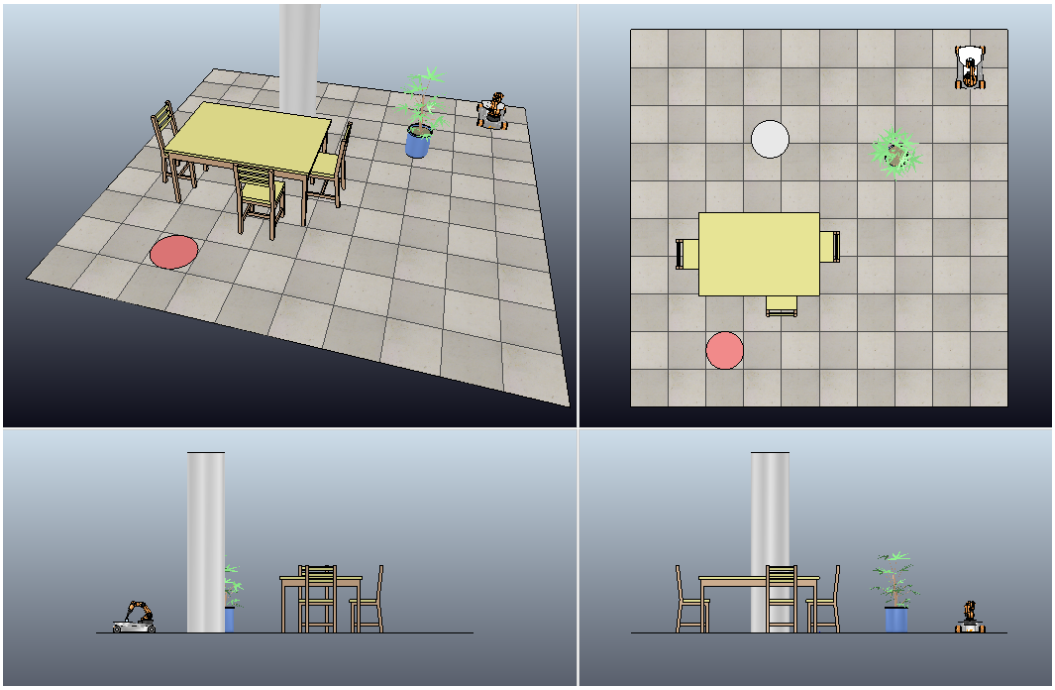

Figure 12

This scenario represent a real indoor environment with realistic shapes for the obstacles, so in order to understand it well, the image shows the map in an isometric point of view from all directions, and in the upper left corner, can be seen the scenario from a perspective point of view.

time of the first solution found *(s)*	9.40 ± 3.41		
time budget: 30 seconds			
	length of the committed trajectory		
	1 node	3 nodes	5 nodes
cost of the path for the first solution *(m)*	5.46 ± 0.09	5.63 ± 0.39	5.48 ± 0.27
cost of the path after time budget *(m)*	5.22 ± 0.18	5.29 ± 0.12	5.34 ± 0.27
cost of the final path *(m)*	5.16 ± 0.16	5.26 ± 0.13	5.19 ± 0.14
time budget: 60 seconds			
	length of the committed trajectory		
	1 node	3 nodes	5 nodes
cost of the path for the first solution *(m)*	5.60 ± 0.24	5.50 ± 0.32	5.53 ± 0.30
cost of the path after time budget *(m)*	5.23 ± 0.18	5.13 ± 0.17	5.13 ± 0.15
cost of the final path *(m)*	5.17 ± 0.16	5.10 ± 0.17	5.09 ± 0.15

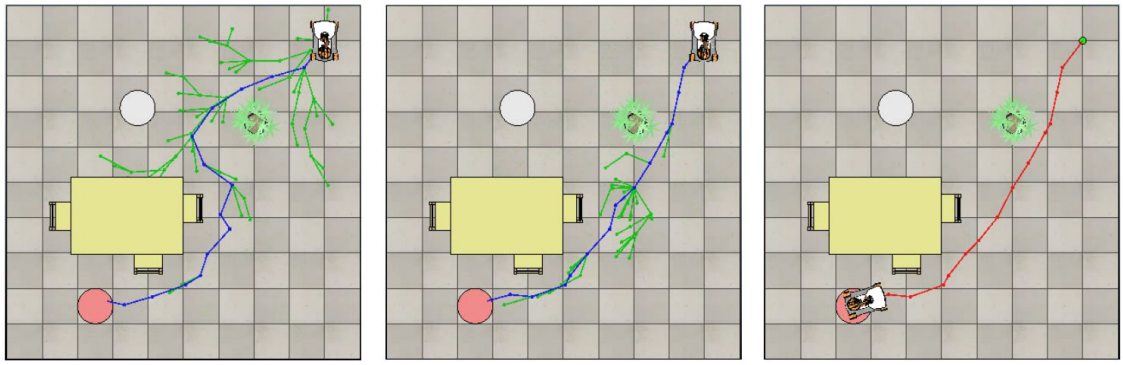

Figure 13: Screenshots for an experiment in the outdoor scenario.

This scenario is very simple, although has particular realistic shapes for the objects. In fact, the average of the time for the first solution is very small, only 9.4 seconds, with a standard deviation of 3.41 seconds. This because the tree expands freely in the map due to the few zone forbidden by the collision with the obstacles.
The remarks are similar to the ones of the basic scenario, indeed, the best cost for the final path is reached with a time budget of 60 seconds, even if with the half of the time budget a similar solution is found, and the planning during trajectory execution can be better exploited.

6 Conclusion

We started our experimental simulation by the offline implementation of the RRT and RRT^* algorithms by fixing the number of iteration and applied the best solution found during the simulation. For the online implementation we used an anytime motion planning algorithm based on the RRT^* algorithm to plan the trajectory for a Youbot from an initial point to final goal point. We tested this proposed algorithm in different complexity scenarios and we obtain good results, on average the solution converges quickly to the optimal one. In our simulations there are static obstacle and we neglect the dynamic constraint of the robot . It would be a challenging problem adding the dynamic constraints of robot and inserting dynamic obstacles.
Finally, this implementation can be more complex by considering other issues like:

- The dynamic constraints of the robot
- Using another robot with non-holonomic constraints
- Using an unknown environment

References

[1] Paolo Ferrari, Marco Cognetti, and Giuseppe Oriolo. Anytime whole-body planning/replanning for humanoid robots. 2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids), 2018.

[2] Sertac Karaman and Emilio Frazzoli. Incremental sampling-based algorithms for optimal motion planning. *Robotics Science and Systems VI*, 104:2, 2010.

[3] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[4] Sertac Karaman, Matthew R Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. Anytime motion planning using the rrt. Institute of Electrical and Electronics Engineers, 2011.

[5] Steven M LaValle and James J Kuffner Jr. Randomized kinodynamic planning. *The international journal of robotics research*, 20(5):378–400, 2001.

[6] John Reif. Complexity of the mover's problem and generalizations. Proc. IEEE Symp. on Foundations of Computer Science, 1979.

[7] P Svestka, JC Latombe, and LE Overmars Kavraki. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.