Quad-RRT: a real-time GPU-based global path planner in large-scale real environments

Alejandro Hidalgo-Paniagua[a], Juan Pedro Bandera[a,*], Manuel Ruiz-de-Quintanilla[b], Antonio Bandera[a]

[a]*Department of Electronic Technology, University of Málaga, Higher Technical School of Telecommunication Engineering, Málaga, Spain*
[b]*Aeorum España S.L, Andalusia Technology Park, Málaga, Spain*

Abstract

During the last decade, sampling based methods for motion and path planning have gained more interest. Specifically, in the field of robotics, approaches based on the Rapidly-exploring Random Tree (RRT) algorithm have become the customary technique for solving the single-query motion planning problem. However, dynamic large maps still represent a challenging scenario for these methods to produce fast enough results. Taking advantage of an NVidia CUDA-enabled Graphic Processing Unit (GPU), we present quad-RRT, an extension of the bi-directional strategy to speed up the RRT when dealing with large-scale, bidimensional (2D) maps. Designed for modern GPUs, quad-RRT computes four trees instead of the two ones built by the bidirectional approaches. This modification aims balancing the direct searching ability of these methods with the parallel exploration of those parts of the map at both sides of the path joining the initial and goal poses. Experimental results demonstrate that the proposed algorithm provides a significant speedup dealing with large-scale maps densely populated by obstacles, when compared to other implementations of the RRT. Hence, the algorithm can have a high impact in the field of inspection path planning for distributed infrastructure. It is also a promising approach to allow new generation robots, designed to work in unconstrained environments, dynamically plan large-scale paths.

Keywords: global path planning, large-scale environment, Rapidly-exploring Random Trees, Graphics Processing Unit

*Corresponding author

Email addresses: ahidalgo@uma.es (Alejandro Hidalgo-Paniagua), jpbandera@uma.es (Juan Pedro Bandera), manuel@aeorum.com (Manuel Ruiz-de-Quintanilla), ajbandera@uma.es (Antonio Bandera)

1. Introduction

Path planning is one of the fundamental pillars for the development of autonomous mobile robots (AMRs). Basically, the aim is to calculate the sequence of actions that allows the AMR navigating from an initial location to a goal, while avoiding static and/or dynamic obstacles. The searching of these actions must also ensure that the transition is performed with optimal (or near optimal) cost. The definition of this cost function is one of the main challenges in path planning as there are many different applications and constraints for robots. Therefore, optimal criteria could be based on conditions such as shortest distance, smoothness or low energy consumption. Optimality is also influenced by the holonomic and non-holonomic constraints (LaValle, 2006). Holonomic constraints can be expressed in terms of the generalized position coordinates (e.g. a manipulator turning a crank). Non-holonomic constraints involve the generalized velocity, or higher derivatives, and are not integrable (e.g. no-slip constraints on mobile vehicle wheels) (Divelbiss et al., 1998). However, path planning is also influenced by other *external* aspects or could need to satisfy other requirements. With respect to the environment, they can be static or dynamic and, in both situations, the robot can (i) have a complete knowledge about its layout; or (ii) need to explore and create a map while it navigates. Specifically, the first problem is known as local path planning meanwhile the second one is referred as global path planning (LaValle, 2006). With respect to the user's requirements, our aim is that the planner will be able to deliver a fast response when dealing with a large-scale map. That is, paths can be non-optimal but they must be obtained as fast as possible.

Although we restrict the problem to finding a path that allows an AMR to navigate through a static environment using an a priori known planar map, dealing with large-scale environments is still considered a challenge (Ichnowski & Alterovitz, 2014). In this area, incremental sampling based methods for path planning have gained interest during the last decade. More precisely, the Rapidly-exploring Random Tree (RRT) algorithm has become the customary technique for solving mathematically complex single-query path planning problems (Kuffner & LaValle, 2000). The growth of the computational power available in the current Central Processing Units (CPUs) and its embedded counterparts, such as All Programmable Systems-on-Chips (AP SoCs), has made RRT to be continuously revisited (Ichnowski & Alterovitz, 2014; Devaurs et al., 2013). One of the most popular options for speeding up the execution of the RRT approach is to take advantage of Graphics Processing Units (GPUs), which can be mounted on consumer computers. GPUs were initially designed for graphics processing, but their computational power has made them a valuable tool for scientific applications related with the robotics framework (Benini et al., 2016; Guan & Bai, 2012). GPUs make use of massively parallel architectures where a growing number of data-parallel

logical processing units are employed. Novel path planners can benefit from the use of these platforms if they can arrange data to be processed in large parallel blocks (Park et al., 2013). Thus, they could be able to provide better performance as the number of processing units increase.

The proposal in this paper focuses on challenging path planning problems, that require a large number of nodes to find a path. The aim is achieving the maximum possible speed up when the number of processors increases. The ideal case is a linear speedup (i.e. to speed up the computation time by a factor equal to p when p processor units are used instead of one unit). Previous approaches based on GPUs propose to develop a massively parallelizable algorithm (Bialkowski et al., 2011) by using different threads to concurrently build a single planning tree (Ichnowski & Alterovitz, 2014), or to grow independent trees (Devaurs et al., 2013). In the work by Kuffner and LaValle (Kuffner & LaValle, 2000; LaValle & Kuffner Jr, 2001), the authors proposed the Bidirectional RRT (B-RRT), to build two RRTs rooted at the start and goal configurations. This paper extends B-RRT to exploit the possibilities of dynamically dividing up the whole map into four submaps, and growing the RRT trees in every submap using multiple GPU threads. Figure 1 shows an overview of how the proposal works. It illustrates how the two typical RRT trees built by bidirectional planners, growing from the initial and goal poses for defining a major axis between them, are now extended to four trees. The additional trees expand the exploration capabilities of our approach and are available for defining new routes. Thus, the main novelties of this contribution can be summarized in

- The extension of the Bidirectional RRT proposal by Kuffner and LaValle (Kuffner & LaValle, 2000; LaValle & Kuffner Jr, 2001) to an approach (named quad-RRT) that divides up the map into four submaps and builds four RRTs. The quad-RRT is faster that the Bidirectional RRT when the environment is densely populated by obstacles, and the route from the initial pose to the goal one requires to expand the exploration to those parts of the map that do not lie on the ellipsoid containing these two poses as its foci.

- A massively parallelizable implementation of the proposal within GPUs. Extensive experimental evaluation has been conducted to validate the performance of the approach and its ability to deal with large-scale maps.

The rest of the paper is organized as follows: Section 2 summarizes related work. The basis of the RRT and Bidirectional RRT are briefly presented at Section 3. The proposed quad-RRT approach is described in Section 4. Section 5 presents analysis of the three algorithms in terms of probabilistic completeness and computational complexity.

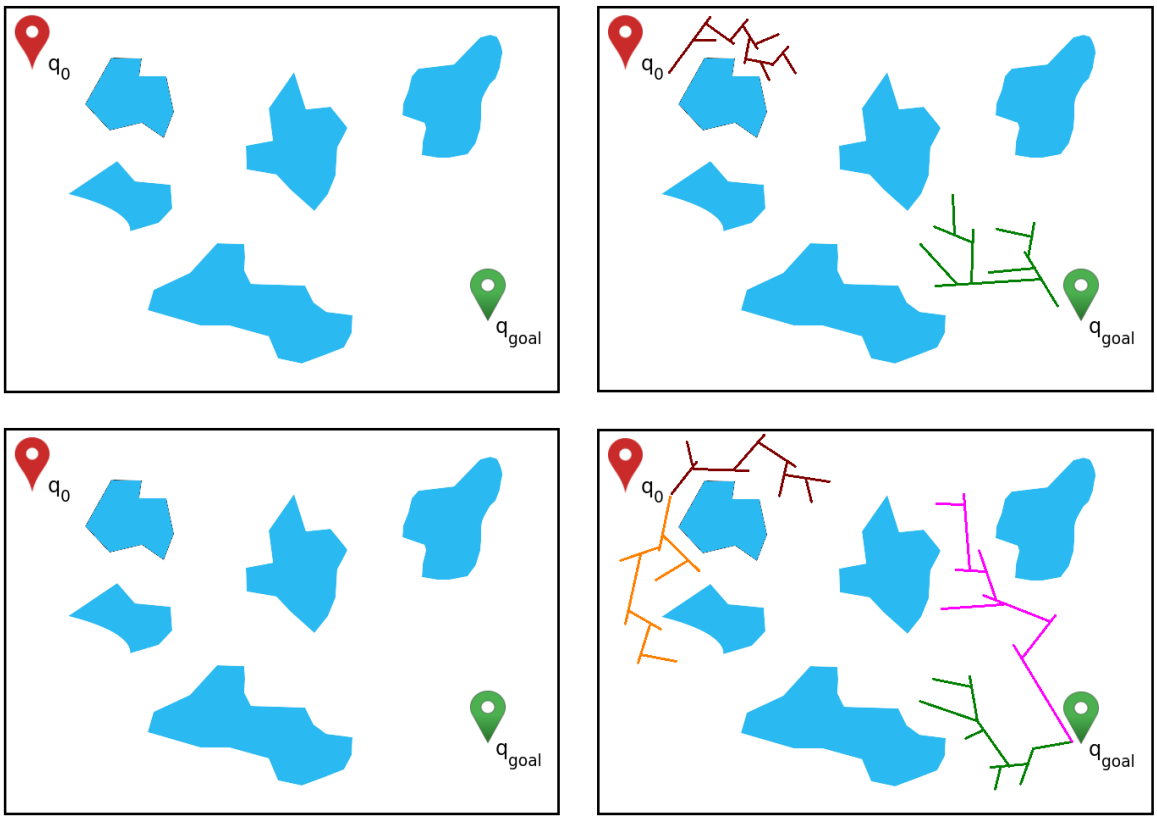


Figure 1: (Top) Execution of a bidirectional RRT (B-RRT); and (down) execution of the proposed quad-RRT. The two additional trees increase the exploratory abilities of the algorithm, which will be able to trace a path faster than the B-RRT when the trajectory from the initial pose to the goal one is densely populated by obstacles

Experimental results are presented in Section 6. Section 7 comments, from a technical point of view, the main advantages and disadvantages of the proposed method. Finally, conclusions and future work are drawn in Section 8.

2. Related work

The problem of navigating through a complex environment has been widely studied in the robotics literature. Restricted to the scenario drawn at Section 1, let $C - space$ denote the state, topological space, consisting of all possible poses that the robot can adopt within the environment. Some poses would result in a collision with an obstacle, and these obstacles are known before planning starts; the remainder of $C - space$ is

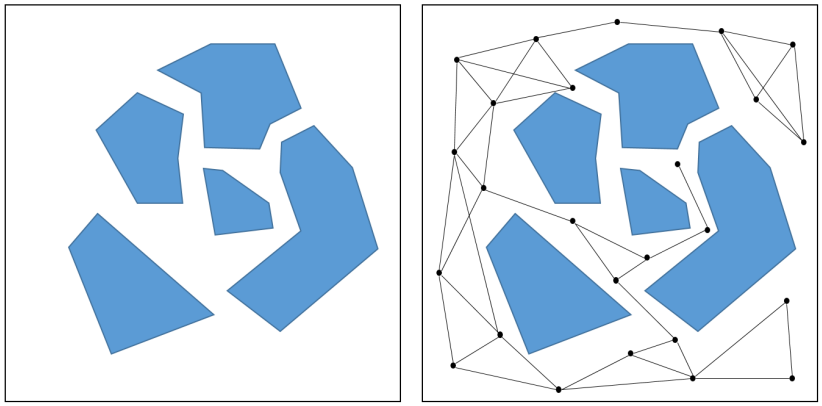


Figure 2: (Left) original map; and (right) a PRM, i.e. a graph consisting of poses (vertexes) in free space, with arcs connecting vertexes when a direct movement between them is collision-free

called free space. The path planning algorithm is the responsible of finding a path through free space from some starting pose $\mathbf{q_0}$ to any pose $\mathbf{q_{goal}}$ within a target set G.

Grid-based approaches provide a straightforward scheme for path-planning. They divide up the whole environment into an array of cells and then apply a search technique (classical AI techniques or more recent methods based on incremental search). Only resolution completeness can be obtained and, when the space size or the dimensionality of the problem are high, these solutions must be discarded due to their large computational cost (Noreen et al., 2016). Sampling planning approaches (the so-called probabilistic roadmaps) were introduced to overcome this problem (Kavraki et al., 1996). A probabilistic roadmap (PRM) consists of a graph $G = (V, E)$, where each node $v_i \in V$ is a pose $\mathbf{q_i}$ in free-space. Each arc $e \in E$ between two nodes, v_i and v_j, denotes that it is possible to move from the pose $\mathbf{q_i}$ to $\mathbf{q_j}$ in a straight-line movement without collision (Murray et al., 2016). Figure 2 shows an example of PRM.

PRMs require the whole graph G to be built in advance. This is a computationally expensive stage that needs to unfold a large number of nodes to assure that all possible routes are considered on the graph. As a great deal of computational effort must be expended to build the graph, a PRM algorithm assumes the same topological space is used each time it is called (i.e. the graph can be reused to find new plans). This is only possible if we assume that obstacles remain unchanged. An on-line counterpart of PRMs is the Rapidly-exploring Random Tree (RRT) (LaValle, 1998). RRT approaches are incremental sampling-based algorithms that, as PRM, are probabilistically complete, i.e. the probability that they provide a solution, if one exists, converges to one as the number of samples converges to infinity (Bialkowski et al., 2011). Contrary to the multi-query PRM, RRT is single-query. A single-query planner assumes

that a new state space is encountered each time it is called. Briefly, RRT samples space points and add them to a tree, $T = (V, E)$, which grows through the whole state space. This growing process can be bounded, as the aim is to provide a single path connecting two poses. The main advantage of the RRT is its ability to look for a solution in a quick and efficient way (Naderi et al., 2015). However, RRT is not asymptotically optimal as, once an arc is set, it cannot be changed. The RRT* algorithm changed this behavior, allowing to redraw the arcs on the tree T (Karaman & Frazzoli, 2011), in order to optimize a given cost function estimated over the path from the initial pose to the goal one. This redefinition of the arcs makes RRT* typically slow, especially when it deals with large environments (Gammell et al., 2014). Real-time variants of the RRT* have been proposed (Gammell et al., 2014; Naderi et al., 2015) to provide faster responses. However, these approaches must bound the sampling space (Gammell et al., 2014) or maintain a tree that iteratively grows through the whole map (Naderi et al., 2015). In this last case the generated tree finally resembles the PRMs, and the strategy is close to the multi-query one. Other approaches make use of heuristics that speed up the original algorithm, but at the cost of computational overload caused by biasing sampling (Qureshi & Ayaz, 2015).

There have been more proposals to speed up the RRT: The ERRT (Bruce & Veloso, 2002) and CL-RRT (Luders et al., 2010) perform the algorithm on a small portion of the environment. However, this cause the trees to be tied to a specific part of the state space, which is not a desirable property. Designed for office-like environments, the topological RRT (Zhao & Li, 2013) proposed to decompose the search of the route into two separate stages. The first one is launched over a topological map. Then, the RRT is employed for continuous state-space exploration within the regions associated to the topological path. The approach provides a significant advantage on time over other RRT approaches, but the procedure to obtain the topological map is complex and off-line. Besides, it cannot be applied with single-query purposes. Taking advantage of the bidirectional search, J. J. Kuffner and S. M. LaValle proposed to speed up the RRT by building two trees instead of a single one (LaValle & Kuffner Jr, 2001). The bidirectional RRT creates two trees: one of them starts from the initial pose and the other starts from the goal pose. These trees grow until a single path is found, i.e. until a state that is 'common' to both trees is found (LaValle & Kuffner Jr, 2001). Figure 1(top) shows an example of this process. The incremental sampling of the bidirectional RRT can be replaced by a more aggressive heuristic to speed up the connection of both trees. The RRT-Connect (Kuffner & LaValle, 2000) employs one of these heuristics (the so-called Connect one) at the end of every iteration to attempt finding a branch joining both trees. On the other hand, RRT and RRT* approaches can naturally be parallelized. Significant work has exploited this parallelism using multicore and

multi-processors CPUs, field programmable gate arrays (FPGAs) and GPUs. Some proposals rely on multiple trees: for instance, C-FOREST (Coupled Forest of Random Engrafting Search Trees) is a parallelization algorithm for single-query path planning (Otte & Correll, 2013). Multiple search trees grow in parallel (e.g., 1 per CPU). Each time a better path is found, it is shared between trees so all of them can benefit from its data. This approach is not a path-planning algorithm by itself, as it must be used on top of a predefined sampling based motion planning algorithm (Otte & Correll, 2013). Other approaches, such as the PRRT algorithm, use multiple threads to build a single motion planning tree. Specifically, the PRRT (Ichnowski & Alterovitz, 2014) is designed to run on a multi-core CPU architecture. It introduces key components relevant to multicore concurrency with the aim of creating opportunities for superlinear speedup (i.e. computation time is sped up by a factor greater than p when p cores are used instead of one). This work shows the importance of embodying the planning approach within the specific hardware to be employed, for achieving the best results. Our focus is on designing a new, parallelized strategy for exploring the state space.

The quad-RRT resembles the data flow of the Sampling-based Roadmap of Trees (SRT) (Plaku et al., 2005). Thus, we also build several trees, distributing the planning problem into subproblems, which are solved by the RRT, and then connected together. As SRT, the quad-RRT proposes to distribute the workload trying to build each tree within a dedicated processing unit. However, the SRT generates a roadmap of trees, integrating the global sampling properties of PRM with the local sampling properties of tree-based planners such as the RRT. Thus, the result is a multi-query approach. On the contrary, the quad-RRT is a tree-based, single-query planner. Moreover, the SRT expands all trees through the whole environment. The quad-RRT creates each tree within a specific region of the map. On the other hand, the quad-RRT is implemented to be endowed within a specific hardware (NVIDIA GPU). Thus, we use the application programming interface (API) CUDA. CUDA programming model uses the GPU as a coprocessor, able of executing a large number of threads in parallel. These threads are grouped into thread blocks. Each block is a collection of threads that share access to a block of device memory. In our implementation, threads within one block cooperate on building one tree. Finally, within the random sampling scheme of the RRT, we minimize the large computational cost of the collision-checking management (Bialkowski et al., 2011) by discarding those nodes which cannot be directly linked to the tree structure.

3. The RRT and B-RRT algorithms

3.1. Problem definition

Let $X \in \Re^2$ be the bidimensional, Euclidean given state space. This space is classified into free and non-free (obstacle) regions denoted by $X_{free} \in X$ and $X_{obs} \in X$,

respectively. Each pose within the state space is a node $q_i \in X_{free}$. The set of all possible nodes, V, coincides with X_{free}. When it is possible for the AMR to traverse from node $q_a \in X_{free}$ to node $q_b \in X_{free}$ in a straight line, an arc $e_{a,b}$ joins both nodes. The set of all arcs will be denoted by E.

Two arcs $e_{i,j}$ and $e_{m,n}$ are adjacent if they share a common node. Two poses q_1 and q_n on the free space are connected if there exists an ordered sequence of $n \geq 2$ nodes $\sigma(q_1, q_n) = \{q_i\}_1^n$ such as

$$\forall q_i, q_{i+1} \in \sigma(q_1, q_n) \quad \exists e_{i,i+1} \in E \tag{1}$$

The set of all arcs joining the nodes at $\sigma(q_1, q_n)$ is called a path. Then, we can also state that a path in X_{free} is a sequence of adjacent arcs. A simple path will be a path with no repeated nodes.

Let $q_0 \in X_{free}$ and $q_{goal} \in X_{free}$ represent the starting and goal poses respectively. The inherent procedure to all RRT algorithms is to explore the state space using random trees until it is possible to find a path joining q_0 and q_{goal}. A random tree $T_x = (V_x, E_x)$ is an acyclic connected graph, where V_x denotes the nodes and E_x the arcs connecting these nodes, which is formed by a stochastic process. The random tree T_x is connected because there is a path from every node $q_i \in V_x$ to every other node $q_j \in V_x$. And it is acyclic because it does not have cycles, i.e. paths whose first and last nodes are the same. Within the tree, all paths are simple paths. This simplifies the final definition of the path: when a RRT-based approach joins q_0 with q_{goal}, there will be a simple path joining both nodes.

The rest of the section presents different strategies for finding this path $\sigma(q_0, q_{goal})$. Our proposal will be described at Section 4.

3.2. RRT algorithm

Algorithm 1 provides the basic implementation of the RRT algorithm (LaValle, 2006). Briefly, the procedure chooses a random sample q_{rand} from the search space ($Sample(\cdot)$ procedure). Then, it estimates q_{near}, the nearest neighbor to q_{rand} from the tree T (see Fig. 3).

$$NN(q_{rand}, T) := argmin_{n \in V} d(q_{rand}, n) \tag{2}$$

when the nearest point in T lies in an arc, then this arc is split into two arcs, and a new node is inserted into T. Within Algorithm 1, this insertion is completed within the $NN(\cdot)$ procedure (LaValle, 2006).

The procedure $New\_config(\cdot)$ allows the RRT algorithm to deal with obstacles, avoiding that the new vertex does not belong to the free space. This procedure provides the nearest free pose, q_{new}, to q_{rand} without obstacles from q_{near}. If this new point is not the goal, the whole loop is repeated.

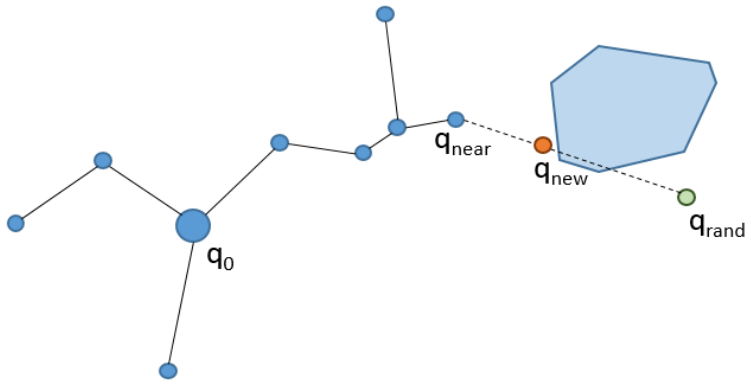


Figure 3: The operation for adding a new node to the tree

Algorithm 1 RRT algorithm

procedure RRT
  $V \leftarrow \{q_0\}; E \leftarrow \varnothing; T \leftarrow (V, E)$
  for 0:N **do**
    $q_{rand} \leftarrow Sample(i)$
    $q_{near} \leftarrow NN(q_{rand}, T)$
    $q_{new} \leftarrow New\_config(q_{near}, q_{rand})$
    if $q_{new} \neq q_{near}$ **then**
      $T.add\_node(q_{new})$
      $T.add\_arc(q_{near}, q_{new})$
    if $q_{new} = q_{goal}$ **then**
      return Reached

3.3. B-RRT algorithm

Algorithm 2 provides an overview of a B-RRT algorithm. The scheme is very similar to the one of the RRT algorithm presented at Algorithm 1. In fact, both algorithms work in exactly the same manner in the initial steps (random sampling, nearest neighbor choosing and obstacle avoiding). The proposal in the Algorithm 2 also tries to ensure that the bidirectional search is balanced (Kuffner & LaValle, 2005), i.e. that the sizes of both trees remain similar.

Briefly, the B-RRT algorithm decomposes the graph into two trees, T_a and T_b. These trees start from q_0 and $_{goal}$, respectively. Both trees can swap after some iterations to balance the search. As aforementioned, in each iteration, T_a grows as the RRT algorithm. The algorithm then uses q_{new} for growing T_b (lines 10-14), trying that the tree T_b grows toward T_a. If both trees connect ($q'_{new} = q_{new}$), a solution has been found. The last step balances the search: it computes the total number of vertexes on the trees and forces new exploration to be performed for the smaller tree.

Algorithm 2 B-RRT algorithm

```
1:  procedure B-RRT
2:      V ← {q_0, q_goal}; E ← ∅; T_a ← (q_0, E); T_b ← (q_goal, E)
3:      for 0:N do
4:          q_rand ← Sample(i)
5:          q_near ← NN(q_rand, T_a)
6:          q_new ← New_config(q_near, q_rand)
7:          if q_new ≠ q_near then
8:              T_a.add_node(q_new)
9:              T_a.add_arc(q_near, q_new)
10:             q'_near ← NN(q_new, T_b)
11:             q'_new ← New_config(q'_near, q_new)
12:             if q'_new ≠ q'_near then
13:                 T_b.add_node(q'_new)
14:                 T_b.add_arc(q'_near, q'_new)
15:             if q'_new = q_new then
16:                 return Reached
17:         if |T_b| > |T_a| then
18:             Swap(T_a, T_b)
```

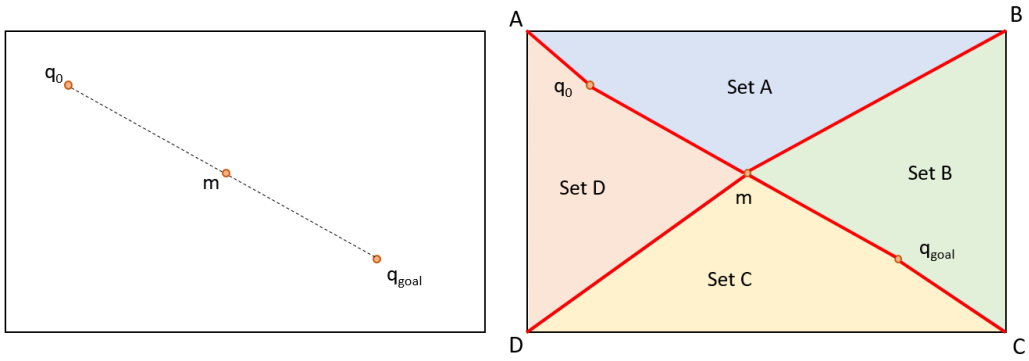


Figure 4: Dividing up the map for defining the four submaps (see text for details)

4. Proposed approach

4.1. Algorithm

As described in Section 1, the possibilities of parallelizing the search using dedicated hardware should be also taken into account on the algorithms. Intimately tied to the use on 2D maps, we propose to built four trees for searching the path from q_0 to q_{goal}. Two of them start from q_0 and the other two start from q_{goal}. Figure 4 outlines how the map is divided. The partition process considers three points: the source and goal 2D points, and the middle point on the straight segment that connects them (m).

Once these three 2D points have been calculated, the next step is to get the partitions of the map. In order to obtain each partition, the algorithm traverses the maps corners in the clockwise direction. Set A in Figure 4 is defined by the top left corner (A), the closest reference point to the current corner (q_0 in the figure), the middle point (m), the closest reference point to the next map corner in the clockwise direction (m again), and the next map corner (B). Boundary points of a map region cannot be repeated, so m will only appear one time into the internal structures storing data. At the end of the partition process, the map will be divided into four different parts (see Figure 4). The quad-RRT grows four trees simultaneously. T_A and T_D start from q_0, choosing random poses from the sets A and D, respectively. T_C and T_B start from q_{goal} and they grow through sets C and B respectively. An important difference with respect to other approaches is that trees are bounded to a specific part of the map. But this constraint does not limit the searching process to a specific part of the map. The quad-tree explores all the map, as Figure 5 shows.

Algorithm 3 describes how the quad-RRT works, while algorithm 4 details the method employed in quad-RRT to grow each tree. In each step of the quad-RRT algorithm, the four trees grow in parallel. At the beginning, each tree only contains

one source point (two of the RRT trees will contain the q_0 reference point, and the other two ones will contain the q_{goal} point). In each iteration, the algorithm selects a random point of each search space ($Sample(i, A)$) and tries to increase the corresponding tree. The $NN(\cdot)$ function employed in the previous algorithms is also used here, but bounded to a specific submap. The algorithm is designed to be implemented on a CUDA architecture (see Section 6). Thus, as Algorithm 4 shows, the q_{rand} is only added to the tree if it can be linked to a node q_{near} on the tree without crossing map's obstacles ($FreeLine(q_{rand}, q_{near})$). This solution provides faster response than the $New\_Config(\cdot)$ procedure employed in Algorithms 1 and 2. The last step of algorithm 3 consists on a connection process between the four RRT trees, whose aim is to find the obstacle-free path between a node of the trees starting from q_0 (T_A or T_D), and a node of the trees starting from q_{goal} (T_B or T_C). This step of the algorithm is in charge, then, of finding the route from q_0 to q_{goal}. In order to avoid blockages when this connection process fails, the algorithm grows the RRT trees $max_{iterations}$ times. If the matching step fails, the algorithm will continue growing the RRT trees and executing the matching step $max_{attempts}$ times.

The connection process takes each RRT tree starting from q_0 and checks if there exists a node in any of the RRT trees starting from q_{goal} which can be safely linked with that tree ($match(T_x, T_y)$ procedure). It must be noted that this connection process is the most expensive step of the proposed algorithm. In fact, the computational cost is proportional to the number of times that the $match(T_x, T_y)$ procedure is called, and this value depends on the number of map partitions:

$$n_{matches} = (\frac{n_{map\_partitions}}{2})^2 \tag{3}$$

It is necessary to take this into account to reduce the computation time of the whole process as much as possible. We have heuristically set that the best performance is obtained when $n_{map\_partitions}$=4.

5. Analysis

5.1. Probabilistic completeness

As commented above, an algorithm is probabilistically complete if the probability of finding a route solution, if one exists, approaches one when the number of samples taken from the search space reaches infinity. RRT is a probabilistically complete algorithm (Qureshi & Ayaz, 2015). As we show on the algorithms at Sections 3 and 4, the random sampling within the quad-RRT is addressed exactly like in the original RRT algorithm. In fact, the parallel growing of the four trees or the connection of these trees for defining

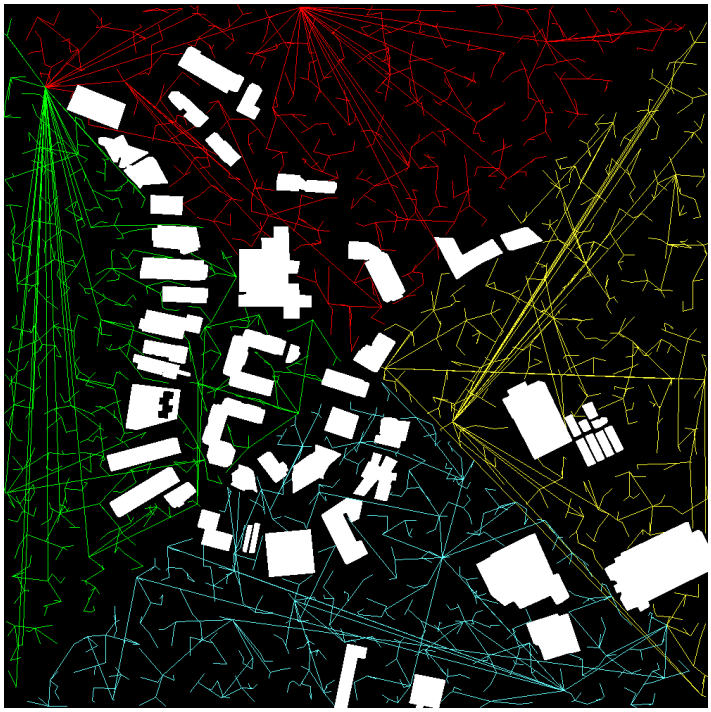


Figure 5: The four trees build by the quad-RRT algorithm. Each tree grows through a bounded part of the map.

Algorithm 3 quad-RRT algorithm

```
1: procedure QUAD-RRT
2:     V ← {q_0, q_goal}; E ← ∅;
3:     T_A ← (q_0, E); T_B ← (q_0, E); T_C ← (q_goal, E); T_D ← (q_goal, E)
4:     Initialize max_iterations
5:     Initialize max_attempts
6:     attempt ← 0
7:     reached ← false
8:     while attempt < max_attempts and reached = false do
9:         iteration ← 0
10:        while iteration < max_iterations and reached = false do
11:            grow_RRT_trees(A, B, C, D, T_A, T_B, T_C, T_D) (Algorithm 4)
12:            reached ← match(T_A, T_B)
13:            if reached = false then
14:                reached ← match(T_A, T_C)
15:                if reached = false then
16:                    reached ← match(T_D, T_B)
17:                    if reached = false then
18:                        reached ← match(T_D, T_C)
19:            iteration ← iteration + 1
20:        attempt ← attempt + 1
21:    return reached
```

Algorithm 4 RRT trees growing algorithm

1: **procedure** $grow\_RRT\_trees(A, B, C, D, T_A, T_B, T_C, T_D)$
2: $q_{rand} \leftarrow Sample(i, A)$
3: $q_{near} \leftarrow NN(q_{rand}, T_A)$
4: **if** $FreeLine(q_{rand}, q_{near})$ **then**
5: $T_A.add\_node(q_{rand})$
6: $T_A.add\_arc(q_{near}, q_{rand})$
7: $q_{rand} \leftarrow Sample(i, B)$
8: $q_{near} \leftarrow NN(q_{rand}, T_B)$
9: **if** $FreeLine(q_{rand}, q_{near})$ **then**
10: $T_B.add\_node(q_{rand})$
11: $T_B.add\_arc(q_{near}, q_{rand})$
12: $q_{rand} \leftarrow Sample(i, C)$
13: $q_{near} \leftarrow NN(q_{rand}, T_C)$
14: **if** $FreeLine(q_{rand}, q_{near})$ **then**
15: $T_C.add\_node(q_{rand})$
16: $T_C.add\_arc(q_{near}, q_{rand})$
17: $q_{rand} \leftarrow Sample(i, D)$
18: $q_{near} \leftarrow NN(q_{rand}, T_D)$
19: **if** $FreeLine(q_{rand}, q_{near})$ **then**
20: $T_D.add\_node(q_{rand})$
21: $T_D.add\_arc(q_{near}, q_{rand})$

a route use versions of the functions employed on the RRT algorithm, which have been slightly modified to bound the growing of the trees to a specific region on the map. It can be proffered that, as RRT, the quad-RRT is a probabilistically complete algorithm.

5.2. Fast convergence to a solution

The main feature of the quad-RRT is its fast convergence to a solution. It is based on the simultaneous building of four trees. This Section provides a proof that our proposal provides faster convergence rates than RRT or B-RRT.

We start making the following assumptions (Qureshi & Ayaz, 2015):

Assumption 1: Uniform sampling

The sampling operator provides samples from a state $C - space$ such that these samples are continuously distributed.

Assumption 2: Cluttered configuration space

The state $C - space$ is cluttered, i.e. the trees will initially grow near its starting state $\mathbf{q_0}$, and then they will incrementally grow towards the non-explored state space.

The first assumption ensures that the sampling operation will not be biased towards a goal, a procedure that has been demonstrated computationally inefficient in high populated environments (Qureshi & Ayaz, 2015). The second assumption is related to this first one: it states that there will be obstacles that will hinder the expansion of the trees in the state space.

Additionally, an important Lemma for sampling-based algorithms is the one stating that they can provide almost-sure convergence to a solution (LaValle & Kuffner Jr, 2001; Karaman & Frazzoli, 2011).

Lemma 1: The probability that the RRT starting at $\mathbf{q_0}$ *will contain* $\mathbf{q_{goal}}$ *as a node approaches one when the number of nodes on the tree approaches infinity*

Within the framework defined by these guidelines for the B-RRT algorithm, let $T_a = (V_a, E_a)$ be a tree starting at q_0 and $T_b = (V_b, E_b)$ a tree starting at q_{goal}. Let p_n be the probability that the n-th sampled pose v provides a final route after n-1 poses have been sampled (see Figure 6)

$$p_n = P(v, \exists x \in V_a / e_{xv} \in E_a \wedge \exists y \in V_b / e_{yv} \in E_b) \tag{4}$$

Let $\Psi(N)$ be the time required to insert a node to a tree containing N nodes (Otte, 2011).

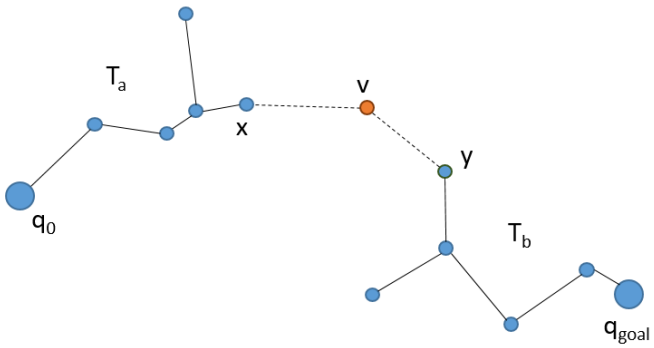


Figure 6: Closing a route within a bi-directional RRT.

Lemma 2: The expected time to find a solution when inserting the n-th node for the B-RRT is

$$E_n = \sum_{n=1}^{\infty} \left[(1-p_n)^{n-1} p_n \left(\sum_{i=0}^{n} \Psi(i) \right) \right] \tag{5}$$

Proof. The time is given by the cumulative insertion of n nodes ($\sum_{i=0}^{n} \Psi(i)$), being the probability the n-th node provides a route equal to $(1-p_n)^{n-1}p_n$.

Now assuming that the quad-RRT contains four random trees, and that a route can be defined using two of these trees (one starting at q_0 and one at q_{goal}), each node insertion within a tree can lead to a solution. Let $p_{\mathbf{T},\mathbf{m}}$ be the probability that the m-th sampled pose v, inserted into tree $\mathbf{T}$, provides a final route. Let assume that these probabilities are the same for all trees. As all trees within the quad-RRT add a new node in parallel, the probability that at least one insertion provides a route is

$$p'_m = 1 - \prod_{\mathbf{T}=1}^{4} (1 - p_{\mathbf{T},m}) \tag{6}$$

Lemma 2 can then be extended to the case of the quad-RRT approach.

Lemma 3: The expected time to find a solution for the quad-RRT is

$$E'_m = \sum_{m=1}^{\infty} \left[(1-p'_m)^{m-1} p'_m \left(\sum_{i=0}^{m} \Psi(i) \right) \right] \tag{7}$$

It can be noted that we assume all proposed nodes have been inserted on the trees (i.e., there were m insertions on each tree) and that the cost of these insertions is

Center (lat,lng)	S size	M size	L size	XL size
(36.738254,-4.552890)	1000×1000	2000×2000	4000×4000	8000×8000

Table 1: Main features of the maps for testing the proposed quad-RRT. Size is given in height per width in meters

the same for all trees. Insertions are made in parallel so we only compute the ones associated to one tree.

It can be assumed that the insertion cost is a non-decreasing function (i.e. $\Psi(i{+}1) \geq \Psi(i) \ \ \forall i > 0$) and that the probability a new node leads to a route is the same for all trees on the quad-RRT or on the B-RRT. Then, it can be stated that the quad-RRT will converge faster than B-RRT to a solution.

Corollary 1: Assuming $m \leq n$ and $p'_m = p_n$, the expected time required to find a route is less for quad-RRT than for B-RRT

Proof. This follows directly from Lemmas 2 and 3. The term associated to the insertion cost in both expressions will be equal or lesser on the quad-RRT.

This conclusion is intuitive. As M. Otte pointed out, *drawing more samples from a random distribution increases the probability of finding what we are looking for* (Otte, 2011).

6. Experiments

6.1. Dataset and hardware specifications

This section details the results obtained by the quad-RRT algorithm when executed over four different 2D maps. Each map corresponds with a real scenario, which is defined by its size (as rows by columns in meters) and the coordinates of its center (expressed in latitude and longitude). Features of the maps are summarized in Table 1. They have been generated by Aeorum[1] for testing a surveillance application based on unmanned aerial vehicles (UAVs). The maps capture real data from a large area at Campanillas (Málaga, Spain). Figure 7 shows a satellite view of the environment from Google Maps. The red flag marks the map center in latitude and longitude.

Within each scenario, we searched for four routes. The starting and target points of these routes were established in the main directions of each map. We call these directions as the main diagonal (MD), the secondary diagonal (SD), the horizontal direction (HD), and the vertical direction (VD). Table 2 provides the real latitude and

[1] https://aeorum.com/

Figure 7: Satellite view of the real environment mapped for testing from Google Maps.

Direction	Source point (lat,lng)	Target point (lat,lng)
MD	(36.739458,-4.559469)	(36.732700,-4.549438)
SD	(36.737850,-4.551176)	(36.733242,-4.559512)
HD	(36.735022,-4.559169)	(36.738045,-4.550800)
VD	(36.733524,-4.556104)	(36.740212,-4.555042)

Table 2: Main map directions (see text for details)

longitude of all source and target points. Figure 8 draws them over the real map from Google Maps.

In order to test the proposed algorithm, several instances with different levels of parallelism have been executed. Table 3 summarizes the different levels of parallelism considered for this study. It can be noted that the number of thread blocks is the same that map partitions (i.e. four blocks). Within our block-based implementation, the growing of each RRT tree is efficiently addressed inside the same thread block. Following the typical CUDA GPU architecture, each thread block can execute several threads at the same time.

Finally, Table 4 provides details about the hardware specifications of the GPU in which the proposed algorithm has been tested. The quad-RRT algorithm has been implemented using C/C++ language and NVidia CUDA v8.0.

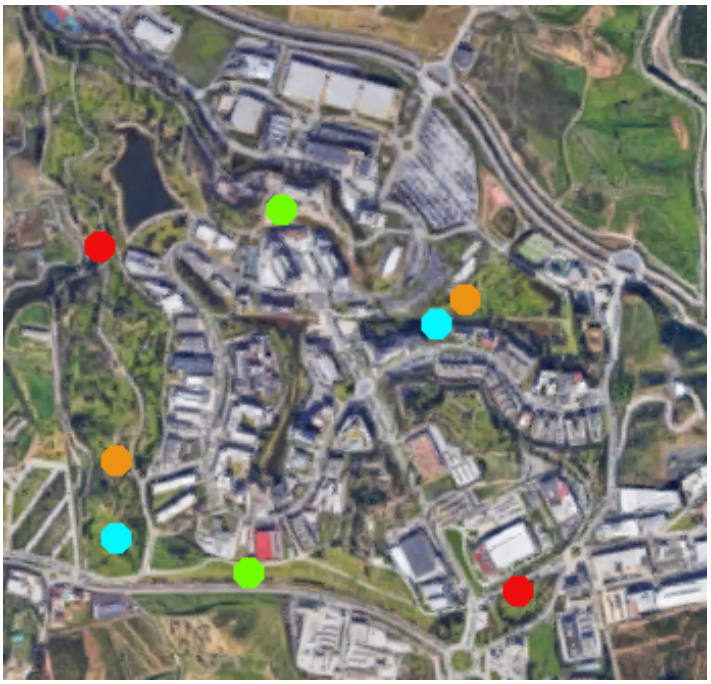


Figure 8: Source and target points of the routes to calculate. Points for MD, SD, HD and VD routes are drawn on red, blue, orange and green color, respectively. The base map has been provided by Google Maps.

Level of Parallelism	Thread blocks	Threads per block
L1	4	1
L2	4	5
L3	4	10
L4	4	25
L5	4	50
L6	4	75

Table 3: Levels of parallelism

Parameter	Value
GPU model	NVidia GeForce GTX 1070
Memory	8 GB
Frequency	1683 MHz
CUDA Cores	1920
Max. Blocks	30
Max. Threads/Block	1024

Table 4: NVidia GeForce GTX 1070 hardware

Map Size	Time (MD) (msec)	Time (SD) (msec)	Time (HD) (msec)	Time (VD) (msec)
S size	339	315	337	328
M size	947	1002	1037	904
L size	4171	3952	4090	3643
XL size	15531	13957	15811	14903

Table 5: Level of parallelism L1. quad-RRT execution times for the MD, SD, HD and VD routes

6.2. Runtime analysis

In this subsection we present and analyze the results obtained by the quad-RRT over the scenarios described at Section 6.1. The proposal has been developed to be a fast planner, so the analysis focuses on the execution times of the algorithm. Tables 5, 6, 7, 8, 9, 10, and 11 show the obtained execution times. Each table is associated to a specific level of parallelism. In order to help understanding the data appearing in these tables, several graphs are also shown. The graphs at Figures 9a-d group the obtained data taking into account the path direction (Table 2) and the level of parallelism (Table 3). These results illustrate that the best performance is generally obtained when the level of parallelism $L3$ and $L4$ are used for computations. It is also interesting to point out that, when the number of threads grows, the execution time increases due to collisions between threads in the blocks.

6.3. Graphical results

With the aim of providing a graphical description of how the quad-RRT works, several images corresponding to one of the executed tests are presented in this Section. Specifically, the images correspond with the execution of the quad-RRT with a level of parallelism L4 and path direction MD. The map used in this example is the S size one

Map Size	Time (MD) (msec)	Time (SD) (msec)	Time (HD) (msec)	Time (VD) (msec)
S size	331	335	338	349
M size	898	912	1046	1063
L size	4044	5592	3889	3772
XL size	14163	14511	15086	14421

Table 6: Level of parallelism L2. quad-RRT execution times for the MD, SD, HD and VD routes

Map Size	Time (MD) (msec)	Time (SD) (msec)	Time (HD) (msec)	Time (VD) (msec)
S size	282	267	274	350
M size	1243	941	1099	1052
L size	4215	4033	4337	3940
XL size	14686	14434	13820	13937

Table 7: Level of parallelism L3. quad-RRT execution times for the MD, SD, HD and VD routes

Map Size	Time (MD) (msec)	Time (SD) (msec)	Time (HD) (msec)	Time (VD) (msec)
S size	405	353	368	417
M size	999	1001	1051	1081
L size	3750	4076	3862	3936
XL size	14417	16340	15691	14552

Table 8: Level of parallelism L4. quad-RRT execution times for the MD, SD, HD and VD routes

Map Size	Time (MD) (msec)	Time (SD) (msec)	Time (HD) (msec)	Time (VD) (msec)
S size	603	633	609	612
M size	1317	1336	1382	1040
L size	4094	3899	4385	4113
XL size	14972	14865	15073	15535

Table 9: Level of parallelism L5. quad-RRT execution times for the MD, SD, HD and VD routes

Map Size	Time (MD) (msec)	Time (SD) (msec)	Time (HD) (msec)	Time (VD) (msec)
S size	839	795	812	865
M size	1527	1521	1552	1641
L size	4449	4889	4737	4589
XL size	15435	14929	16484	15003

Table 10: Level of parallelism L6. quad-RRT execution times for the MD, SD, HD and VD routes

Level of Parallelism	S size	M size	L size	XL size	Total
L1	329.75	972.5	3964	15050.5	5079.19
L2	338.25	979.75	4324.25	14545.25	5046.88
L3	293.25	1083.75	4131.25	14219.25	4931.87
L4	385.75	1033	3906	15250	5143.68
L5	614.25	1268.75	4122.75	15111.25	5279.25
L6	827.75	1560.25	4666	15462.75	5629.18

Table 11: Average execution times (ms), depending on the level of parallelism and the size of the maps

at Table 1. The map consists of a real cartography of the area illustrated at Figure 7. Obstacles in the map have been marked and stored in a geographical data base. The total size of the map is 1000 meters by 1000 meters. Obstacles in the map were composed by smaller pieces with a size of 1 meter $\times$ 1 meter. Figure 10 illustrates the map: obstacles are drawn in the map using the white color, meanwhile free space is drawn using the black one.

Figure 11 shows that the execution of the quad-RRT algorithm builds four RRT trees, which grew through the map. Each RRT tree has been drawn over the map using a different color. As depicted, the calculated RRT trees expand uniformly inside the map. This behavior indicates that the quad-RRT is correctly exploring the complete map. Figure 11 also shows that the final sizes (i.e. number of nodes) for the four trees are balanced. During this growing process, after adding a node to each tree, the matching process tries to find a path to travel from the source pose to the target one in a secure way. Figure 12 shows the finally calculated path.

Take into account that, although the $L1$, $L2$, $L3$, and $L4$ parallelism levels have similar execution times, the $L4$ option, which uses 25 threads per GPU block, explores a higher map space in the same time that those executions using $L1$, $L2$ or $L3$. Again, the example in figure 13 shows that the best results are obtained using the L4 level of

Algorithm	Execution time (msec)	Fail times	Average nodes
Basic-RRT	81918	15	5875
Biased-RRT	59702	6	2606
Greedy-RRT	39499	3	7336
BiasedGreedy-RRT	28412	0	3123
Topological-RRT	3956	0	347
quad-RRT	**396**	**0**	**2652**

Table 12: Performance comparison taking into account the results presented in (Zhao & Li, 2013)

parallelism due to, in these conditions, the quad-RRT algorithm accomplish a deeper exploration of the map spending a similar amount of computation time.

6.4. Comparison to other approaches

Comparison with other RRT approaches has been conducted using the framework provided by K. Zhao and Y. Li's work (Zhao & Li, 2013). In this work, the authors tested several RRT implementations over a specific map. Here, we have tested the quad-RRT approach using the same map and boundary points. Specifically, the map corresponds to an indoor environment. It has a size of 310×169 (Figure 14).

The resulting RRT trees obtained after the execution of the quad-RRT algorithm illustrate that the algorithm correctly explores the available free space in the map (Figure 15). None of the RRT trees explores inaccessible map regions. Due to the location of the boundary points inside the map, and the policy followed by the quad-RRT algorithm to get map partitions (see section 4.1), one of the trees (the green colored tree) grew up inside a very small map region (Figure 16).

Table 12 resumes the execution times obtained by the RRT variants tested by K. Zhao and Y. Li (Zhao & Li, 2013) and the average execution time obtained by the proposed quad-RRT algorithm. These results show that the proposed algorithm is 9.98 times faster than the best alternative tested by K. Zhao and Y. Li and 206.86 times faster than the original RRT (Basic-RRT). As the BiasedGreedy-RRT and the Topological-RRT, the proposed algorithm does not produce any failure on finding a path (a failure is denoted when the number of samples exceeds 50,000). Moreover, being faster, the quad-RRT provides an average number of nodes that is greater than the one provided by the Topological-RRT. Hence, it performs a deeper exploration of the search space. Figure 17 shows one obtained path. It can be noted how the calculated path using the quad-RRT algorithm is always very close to the optimal solution provided by a RRT* approach.

It should be noted that highly-parallelized algorithms have a better performance when they work with large datasets. In this case, the map used by K. Zhao and Y. Li is not big enough. Even so, the proposed quad-RRT algorithm is several times faster than the approaches tested by these authors (Zhao & Li, 2013). K. Zhao and Y. Li do not provide the source code of their studied algorithms, so we cannot test these algorithms using our proposed datasets.

7. Discussion

The original bidirectional RRT creates two trees, which grow until a state that is shared by both trees is found. This scheme was significantly sped up by replacing the incremental sampling by other heuristics, but also by those approaches that distribute the planning problem into subproblems. The quad-RRT is an approach designed to be embodied within a NVIDIA GPU, and aggressively employs these two premises (problem subdivision and GPU parallel processing) to provide a fast solution to searching for a path within a large and populated environment. In fact, it also includes a third premise to work even faster: each chosen node that cannot be directly linked to the tree structure is discarded. From a theoretical point-of-view, the main difference with other approaches is that the quad-RRT divides up the search space in several regions and grows a tree within each region on the map. This scheme could be extended to other n-dimensional state spaces by designing the adequate matching function. The connection process among trees is probably the bottleneck of the approach. Further work should address the design of a parallel mechanism to achieve it, specially when working on n-dimensional state spaces.

Table 13 summarizes the main technical advantages and disadvantages of the proposed approach. As the table shows, the proposed algorithm has only a few disadvantages. Taken into account the new hardware generation and the growing integration level in electronics, we can assert that these minor disadvantages will disappear in a very short period of time, so it is unlikely they have a significant impact in a near future.

8. Conclusion and future work

This paper describes the quad-RRT algorithm, an extension of the popular bidirectional RRT-based (B-RRT) approaches for path planning. The B-RRT improved the performance of the classic RRT algorithm by generating two RRT-trees over the target map at the same time. Our quad-RRT algorithm takes advantage of the novel GP-GPU techniques, and provides faster responses than the B-RRT by building four RRT-trees simultaneously over the target map. The conclusion could not be however

Advantages	Disadvantages
• Between 10 and 200 times (approximately) faster than other RRT-based approaches • Deep and balanced map exploration • Takes advantage of the multicore parallelism • Takes advantage of the general–purpose GPU programming • Feasible to be used within the most demanding applications	• More hardware requirements than other RRT-based approaches • Slightly higher energy consumption (due to the use of the GPU computing techniques)

Table 13: Main advantages and disadvantages of the proposed quad-RRT algorithm

that augmenting the number of trees results in a fast approach. There is a major problem when a map is populated by several independent trees that start from the source or the target poses: the algorithm needs to connect these trees to find the final route. In our case, the matching step at Algorithm 3 performs these connections. It is then necessary to find a balance between the gain on speed achieved by using several trees to explore the map, and the cost of searching for fast routes joining the trees starting from the source pose with the ones starting from the target pose. The situation resembles the one stated on the Amdahl's law (Amdahl, 1967): *the degree of speedup achieved by parallelization is limited by the sequential fraction of the algorithm.* In our case, the situation is even worse. When the number of trees augments (they can be created in parallel), our 'sequential' fraction (i.e. the matching step) also increases. When the number of trees is greater than four in our approach, this increase supposes an important percentage of the computation time, and the resulting algorithm runs slower. Obviously, this situation is not applicable to other strategies where each tree links source and target pose (Otte & Correll, 2013).

On the other hand, the passive solution adopted for some approaches to the bidirectional RRT, consisting on sampling the space until $q'_{new} = q_{new}$ (see Algorithm 2), has been also tested in the quad-RRT. The large scale of maps makes this solution

slower. A second solution for speeding up route searching is to optimize the generation of the trees. Within the specific framework of the CUDA programming model, we have implemented the building of each tree into a thread block. This strategy reveals as an efficient proposal, as all threads on the block are assigned to the same multi-processor, and share the resources included in this multi-processor, such as register file and shared memory. Discarding sampled nodes which are not directly linked to the tree also helps speeding up the algorithm. However, it is also important to note that augmenting the number of threads does not always imply a reduction of the computational time. Our results show that, when the number of threads within the block exceeds a threshold value, they start to collide among them (due to the critical sections[2] in the code). These collisions reduce the overall performance of the algorithm. This is known as the speed-up limit of a parallel algorithm (Popov et al., 2010).

It is however important to highlight that, when execution times are similar, it is preferable to choose those solutions that provide a deeper exploration of the map. The amount of explored map is directly proportional to the RRT-trees density. That is, when the number of nodes in the RRT-trees increases, the amount of explored map also increases. The level of exploration is specially relevant when maps have a huge amount of obstacles. For this reason, in our case, we prefer to choose the level of parallelism $L4$ instead of levels $L1$, $L2$ or $L3$.

Finally, the experiments demonstrate that the quad-RRT proposal converges quickly to a solution. The comparison with other RRT or B-RRT approaches have been conducted using the framework provided by K. Zhao and Y. Li (Zhao & Li, 2013). Despite this scenario is not the best for the quad-RRT (the map is not too large), obtained results show that the quad RRT-algorithm is 206.86 times faster than the original RRT algorithm (Basic-RRT) and 9.98 times faster than the best RRT variant tested by K. Zhao and Y. Li. This significant improvement in the response times of the algorithm makes it suitable to plan paths in dynamic, large scenarios. The following descriptive example may help illustrating the implications of these results: in large maps composed by 1 meter $\times$ 1 meter cells, such as the ones employed in this paper, a robot moving at a comfortable human walking speed of 13 cm/s (Bohannon, 1997), using the proposed quad-RRT algorithm, can update its plan several times per cell (Table 11). However, the best tested alternative may find issues updating the plan even once per cell, meaning the robot should slow down, or even stop, to update the path according to changes in the cell occupancy values.

Future work focuses on comparing the quad-RRT to other approaches within the

[2]A critical section is a sequence of operations that must be executed sequentially by the CUDA threads.

framework provided by the Open Motion Planning Library (OMPL) and migrate this first version of the algorithm to manage 3D scenarios. We will also work on extending the GP-GPU (General Purpose-Graphical Processing Unit) computing model to other motion planning problems. The change on the dimensionality of the state space will demand new studies on the optimal number of trees and a new design for the CUDA hierarchy (threads, blocks...).

Acknowledgements

This work has been partially developed within the projects, Dronecaptor (ITC-20151141), Surveiron (DRS-17-2014-1 and DRS-17-2014-2), and TIN2015-65686-C5. All of these projects have been funded by the Spanish Ministerio de Economía y Competitividad and/or FEDER funds.

References

Amdahl, G. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of American Federation of Information Processing Societies (AFIPS 67)* (pp. 483–485).

Benini, A., Rutherford, M., & Valavanis, K. (2016). Real-time GPU-based pose estimation of a UAV for autonomous takeoff and landing. In *Proceedings of IEEE International Conference on Robotics and Automation* (pp. 3463–3470).

Bialkowski, J., Karaman, S., & Frazzoli, E. (2011). Massively parallelizing the RRT and the RRT. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 3513–3518). doi:10.1109/IROS.2011.6095053.

Bohannon, R. W. (1997). Comfortable and maximum walking speed of adults aged 20-79 years: reference values and determinants. *Age and Ageing*, *26*, 15–19. doi:10.1093/ageing/26.1.15.

Bruce, J., & Veloso, M. (2002). Real-time randomized path planning for robot navigation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 2383–2388). volume 3. doi:10.1109/IRDS.2002.1041624.

Devaurs, D., Simon, T., & Corts, J. (2013). Parallelizing RRT on Large-Scale Distributed-Memory Architectures. *IEEE Transactions on Robotics*, *29*, 571–579. doi:10.1109/TRO.2013.2239571.

Divelbiss, A., Seereeram, S., & Wen, J.-T. (1998). Kinematic Path Planning for Robots with Holonomic and Nonholonomic Constraints. In *Essays on Mathematical Robotics* chapter 104. (pp. 127–150). doi:10.1007/978-1-4612-1710-7_5.

Gammell, J. D., Srinivasa, S. S., & Barfoot, T. D. (2014). Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 2997–3004). doi:10.1109/IROS.2014.6942976.

Guan, X., & Bai, H. (2012). A GPU accelerated real-time self-contained visual navigation system for UAVs. In *Proceedings of IEEE International Conference on Information and Automation* (pp. 578–581). doi:10.1109/ICInfA.2012.6246879.

Ichnowski, J., & Alterovitz, R. (2014). Scalable Multicore Motion Planning Using Lock-Free Concurrency. *IEEE Transactions on Robotics*, *30*, 1123–1136. doi:10.1109/TRO.2014.2331091.

Karaman, S., & Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *The International Journal of Robotics Research*, *30*, 846–894. doi:10.1177/0278364911406761.

Kavraki, L. E., Svestka, P., Latombe, J. C., & Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, *12*, 566–580. doi:10.1109/70.508439.

Kuffner, J. J., & LaValle, S. M. (2000). RRT-connect: An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation* (pp. 995–1001). volume 2. doi:10.1109/ROBOT.2000.844730.

Kuffner, J. J., & LaValle, S. M. (2005). *An efficient approach to path planning using balanced bidirectional RRT search*. Technical Report Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

LaValle, S. M. (1998). *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. Technical Report Computer Science Department, Iowa State University.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press.

LaValle, S. M., & Kuffner Jr, J. J. (2001). Randomized kinodynamic planning. *The International Journal of Robotics Research*, *20*, 378–400. doi:10.1177/02783640122067453.

Luders, B. D., Karaman, S., Frazzoli, E., & How, J. P. (2010). Bounds on tracking error using closed-loop rapidly-exploring random trees. In *Proceedings of the 2010 American Control Conference* (pp. 5406–5412). doi:10.1109/ACC.2010.5530777.

Murray, S., Floyd-Jones, W., Qi, Y., Sorin, D. J., & Konidaris, G. (2016). Robot Motion Planning on a Chip. In *Proceedings of Robotics: Science and Systems*.

Naderi, K., Rajamäki, J., & Hämäläinen, P. (2015). RT-RRT*: A Real-time Path Planning Algorithm Based on RRT*. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games* (pp. 113–118). doi:10.1145/2822013.2822036.

Noreen, I., Khan, A., & Habib, Z. (2016). Optimal Path Planning using RRT* based Approaches: A Survey and Future Directions. *International Journal of Advanced Computer Science and Applications*, *7*, 97–107. doi:10.14569/IJACSA.2016.071114.

Otte, M., & Correll, N. (2013). C-FOREST: Parallel Shortest Path Planning With Superlinear Speedup. *IEEE Transactions on Robotics*, *29*, 798–806. doi:10.1109/TRO.2013.2240176.

Otte, M. W. (2011). *Any-com multi-robot path planning*. Ph.D. thesis University of Colorado at Boulder.

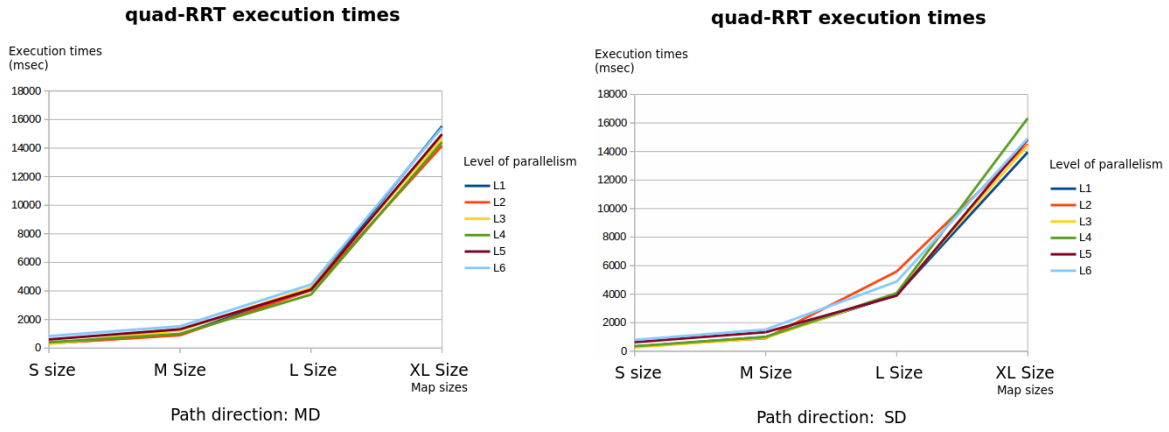
Park, C., Pan, J., & Manocha, D. (2013). Real-time optimization-based planning in dynamic environments using GPUs. In *IEEE International Conference on Robotics and Automation* (pp. 4090–4097). doi:10.1109/ICRA.2013.6631154.

Plaku, E., Bekris, K. E., Chen, B. Y., Ladd, A. M., & Kavraki, L. E. (2005). Sampling-based roadmap of trees for parallel motion planning. *IEEE Transactions on Robotics*, *21*, 597–608. doi:10.1109/TRO.2005.847599.

Popov, G., Mastorakis, N., & Mladenov, V. (2010). Calculation of the Acceleration of Parallel Programs As a Function of the Number of Threads. In *Proceedings of the 14th WSEAS International Conference on Computers* (pp. 411–414).

Qureshi, A. H., & Ayaz, Y. (2015). Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments. *Robotics and Autonomous Systems*, *68*, 1–11. doi:10.1016/j.robot.2015.02.007.

Zhao, K., & Li, Y. (2013). Path planning in large-scale indoor environment using RRT. In *Proceedings of the 32nd Chinese Control Conference* (pp. 5993–5998).



(a) MD direction

(b) SD direction

(c) HD direction

(d) VD direction

Figure 9: Execution times for the calculation of the path in our four directions, using different levels of parallelism and map sizes

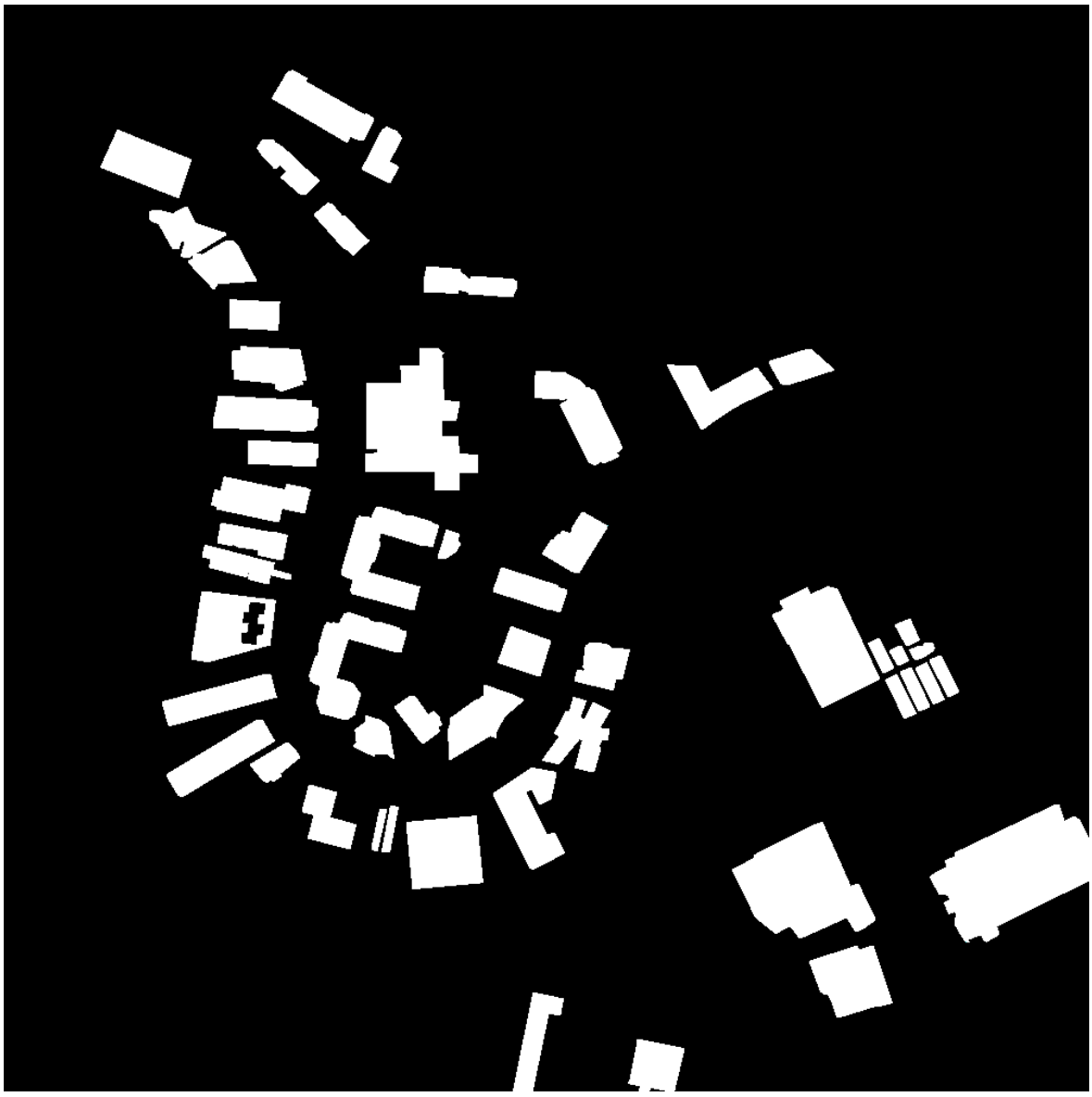


Figure 10: Binary representation of the real map of 1000 meters by 1000 meters. The area is located at Campanillas (Málaga, Spain)

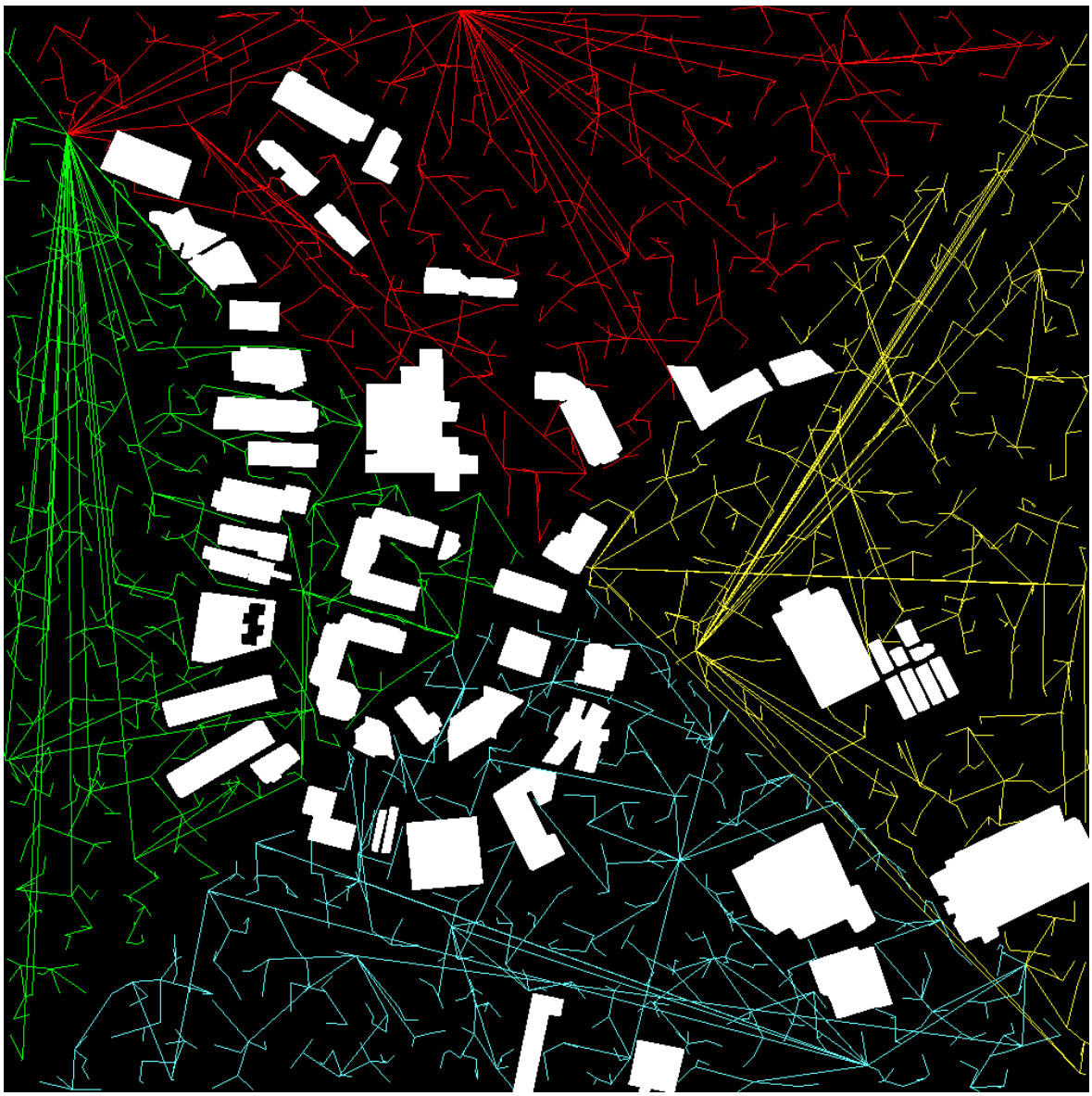
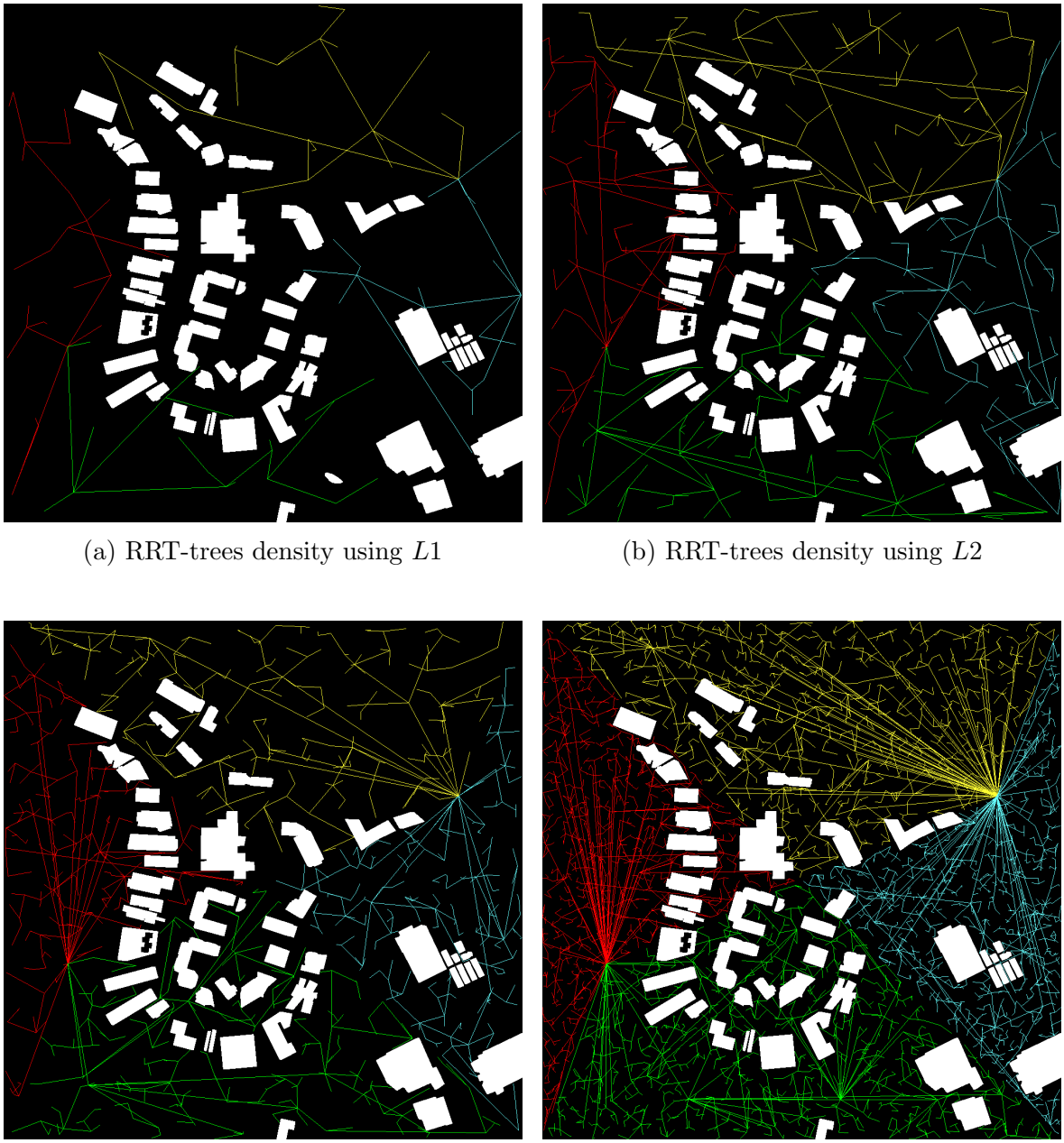


Figure 11: RRT trees generated by the quad-RRT algorithm

Figure 12: The calculated path using the quad-RRT algorithm



(a) RRT-trees density using $L1$

(b) RRT-trees density using $L2$

(c) RRT-trees density using $L3$

(d) RRT-trees density using $L4$

Figure 13: Example of the RRT-trees density depending on the selected level of parallelism

Figure 14: (Left) Indoor map used by K. Zhao and Y. Li (Zhao & Li, 2013), and (right) boundary points to plan a new path inside the map

Figure 15: Resulting RRT trees of the quad-RRT algorithm execution on the K. Zhao and Y. Li's map

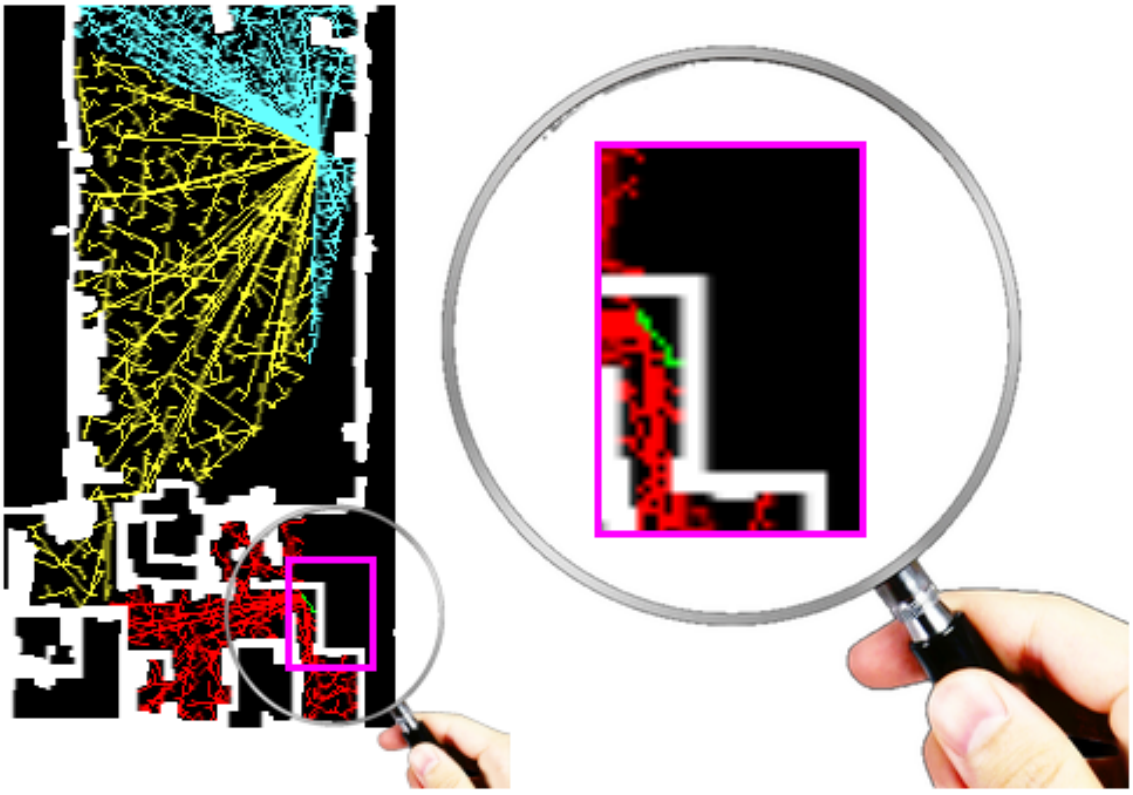


Figure 16: A RRT tree inside a very small region of the map used by K. Zhao and Y. Li

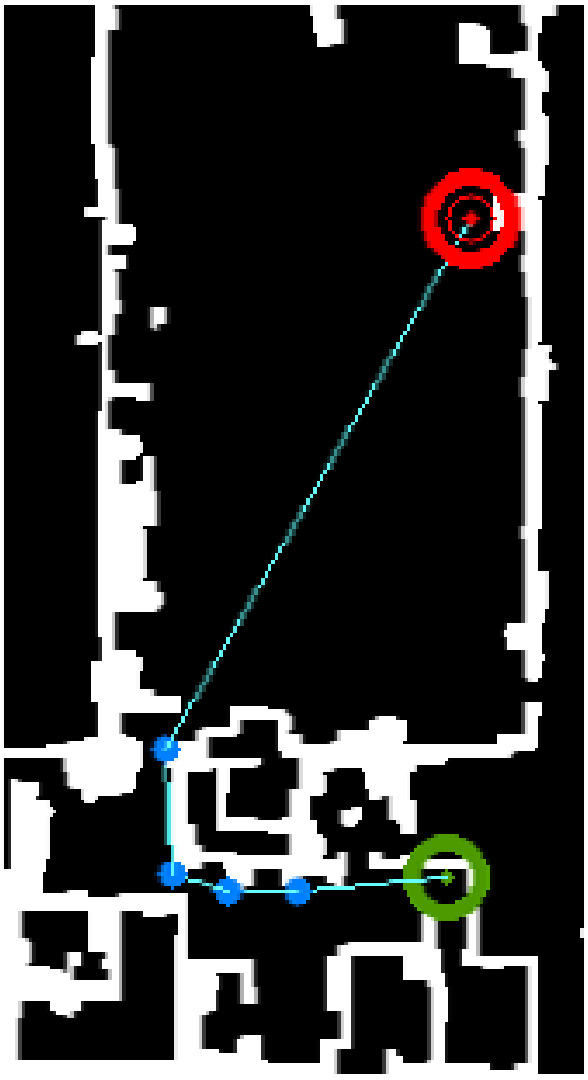


Figure 17: The calculated path over the map used by K. Zhao and Y. Li in (Zhao & Li, 2013)