Minimising Computational Complexity of the RRT Algorithm

A Practical Approach

Svenstrup, Mikael; Bak, Thomas; Andersen, Hans Jørgen

Published in:
I E E E International Conference on Robotics and Automation. Proceedings

DOI (link to publication from Publisher):
10.1109/ICRA.2011.5979540

Publication date:
2011

Document Version
Publisher's PDF, also known as Version of record

2011 IEEE International Conference on Robotics and Automation
Shanghai International Conference Center
May 9-13, 2011, Shanghai, China

Minimising Computational Complexity of the RRT Algorithm A Practical Approach

Mikael Svenstrup, Thomas Bak and Hans Jørgen Andersen

Abstract— **Sampling based techniques for robot motion planning have become more widespread during the last decade. The algorithms however, still struggle with for example narrow passages in the configuration space and suffer from high number of necessary samples, especially in higher dimensions. A widely used method is Rapidly-exploring Random Trees (RRT's). One problem with this method is the nearest neighbour search time, which grows significantly when adding a large number of vertices. We propose an algorithm which decreases the computation time, such that more vertices can be added in the same amount of time to generate better trajectories. The algorithm is based on subdividing the configuration space into boxes, where only specific boxes needs to be searched to find the nearest neighbour. It is shown that the computational complexity is lowered from a theoretical point of view. The result is an algorithm that can provide better trajectories within a given time period, or alternatively compute trajectories faster. In simulation the algorithm is verified for a simple RRT implementation and in a more specific case where a robot has to plan a path through a human inhabited environment.**

I. Introduction

Sampling based methods for motion and path planning have gained more interest during the last decade, as computer power has increased. Among the most widely used techniques are Rapidly-exploring Random Trees (RRT's)[1], which work by randomly expanding a tree with vertices over the configuration space to find a path from a start location to a goal location. RRT's have been used in various applications such as kinodynamic path planning [2], navigation for urban driving [3], coordination of robot motion [4], and humanoid robot motion planning [5]. The main strengths of an RRT are the ease of implementation, the ability to avoid obstacles and the applicability for systems with differential constraints. RRT algorithms do, however, suffer from the large number of vertices necessary in trees for large dimension problems and for problems with large configuration spaces including many obstacles. When adding a large number of vertices to a tree, especially the time it takes to find the nearest other vertex in each iteration is time consuming.

There are two ways to address this problem. One approach is to make a smarter algorithm, which needs fewer samples to obtain the goal. Another approach is to speed up the search for the nearest neighbour, such that more vertices can be added within the same time period. Most research has focused on the first problem, where most of the ideas come from trying to intelligently bias the exploration towards unexplored portions of the configuration space, such that the algorithm needs fewer samples to converge towards a goal [6], [7]. This can be done by either biasing the sampling distribution or improving the vertex extension operation. The nearest neighbour search typically rely on a naive brute force method, where the distance to all vertices are calculated, to find the nearest. This is mainly because it is by far the simplest way to do it. Not much research within the motion planning community has been done trying to optimise the speed of the nearest neighbour search. Similar problems have, however, been studied generally in the field of computational geometry under the name *range search problems* [8], and have applications in e.g. computer graphics and when querying large databases [9], [10], [11].

Good candidate algorithms for finding the nearest neighbour are found using a tree based structure to subdivide the configuration space. Algorithms include quadtrees, R-trees and kd-trees. Quadtrees and R-trees have a bad worst case performance, which may be why the kd-tree is the most widely used. A kd-tree works by recursively subdividing the space into two half spaces one dimension at the time (see [1], [12]). The query time complexity of a kd-tree is $\mathcal{O}(n^{1-\frac{1}{d}})$ (compared to $\mathcal{O}(n)$ for a brute force approach), where n is number of vertices and d is the number of dimensions. However, query time can be improved substantially for higher dimensional search spaces, if it is enough to find the approximate nearest neighbour (ANN) [13], [12], where the complexity is reduced to $\mathcal{O}(\log n)$. These tree based search algorithms are not designed for use in motion planning, but for database query algorithms, for which you may not know the structure of the data or the size of n. They are made for fast queries and do not consider the time it takes to pre-process the database to obtain the fast query. This can be a problem for motion planning algorithms, which need to do the pre-processing on-line as well.

A desirable characteristic for on-line motion planning problems is to add as many vertices in as short a time period as possible. If the speed of the range query decreases as the tree grows larger it effectively limits how large the tree can grow within some time period, which is critical in an on-line system.

We present a simple practical algorithm for minimising the nearest neighbour search time. The algorithm is demonstrated for typical RRT motion planning problems for a mobile robot. The algorithm is based on a grid in d dimensions, which becomes d-dimensional boxes, where each vertex belongs to a specific box. Only the relevant

M. Svenstrup and T. Bak are with the Department of Electronic Systems, Automation & Control, Aalborg University, 9220 Aalborg, Denmark {ms,tba}@es.aau.dk

H.J. Andersen is with the Department for Media Technology, Aalborg University, 9220 Aalborg, Denmark hja@imi.aau.dk

boxes need to be searched to find the nearest vertex. The background for this algorithm has origin in computational geometry, but has to the best of our knowledge not been exploited for motion planning. The algorithm is applicable to on-line as well as off-line applications and the benefit of this box method increases as number of vertices, that need to be added to the tree, grows.

First the algorithm is presented, and then an analysis of the time complexity versus other methods is given. Then the algorithm is demonstrated in simulations.

II. Methods

The structure of a very basic RRT algorithm can be seen in Algorithm 1.

Algorithm 1 Standard RRT (see [14])

RRTmain()

```
1: Tree = q.start
2: q.new = q.start
3: while Distance(q.new , q.goal) < ErrTolerance do
4:    q.target = SampleTarget()
5:    q.nearest = NearestVertex(Tree , q.target)
6:    q.new = ExtendTowards(q.nearest,q.target)
7:    Tree.add(q.new)
8: end while
9: return Trajectory(Tree,q.new)
```

SampleTarget()

```
1: if Rand() < GoalSamplingProb then
2:    return q.goal
3: else
4:    return RandomConfiguration()
5: end if
```

The objective of the algorithm is to start from an initial configuration and find a path to a goal configuration. This is done by continuously adding vertices to a tree, which is grown from the starting configuration. To extend a tree a random point is sampled from the configuration space. Then the distances to all existing vertices are calculated, and the nearest vertex is chosen. Finally the tree is extended from the chosen vertex towards the sampled configuration. When a leaf vertex reaches within some distance of the goal location, the algorithm is stopped. When the tree becomes large, a significant part of the computation time is spend on the nearest neighbour search (the red line in Algorithm 1).

A. Minimising computation time for finding nearest vertex

The basic idea of the proposed approach, for minimising the computational complexity of finding the nearest vertex, is simple. It consist of partitioning the d-dimensional configuration space in a number of d-dimensional boxes, and only calculate the distance to other vertices in relevant boxes instead of all vertices in the whole configuration space. For simplicity, we use k equally sized boxes in each dimension. That means we will get k^d d-dimensional boxes. The algorithm is started by searching all vertices in the same box as the newly sampled vertex. Then boxes adjacent to the first box are searched and step by step boxes further and further away are searched. The algorithm terminates, when the nearest found vertex is closer than the boundary between the searched and unsearched boxes. In that case it is guaranteed that no vertex in any of the outer boxes, which have not been searched yet, can be closer than the current nearest vertex.

The algorithm is illustrated for two dimensions in Fig. 1. Here an RRT with 100 vertices have already been grown. The first step is to sample a new vertex, which is the red dot. The corresponding box, to which it belongs, is the red hatched area. A zoomed version of this area is shown in Fig. 2.

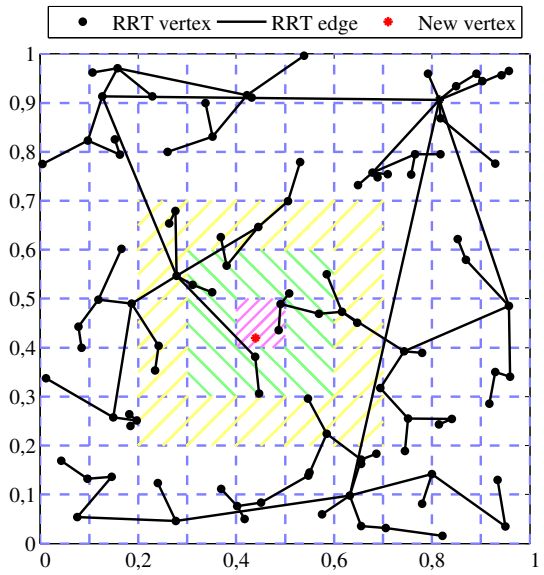

Fig. 1. The configuration space partitioned in a number of boxes in each dimension. A grown tree has to be extended with a newly sampled vertex (the red dot). The nearest boxes to the new vertex is hatched in different colours.

After finding the corresponding box, we start by calculating the distance to all vertices in this box (V_1 and V_2) and the distance to the nearest border of the box, d_{border}, which is shown in Fig. 2. If any of the vertices are closer than d_{border}:

$$\min_{i \in I} \{|V_{new} - V_i|\} \leq d_{border} \quad , \tag{1}$$

where I is the set of vertices in the box, then no other vertex in the configuration space can be closer. In this case the algorithm is terminated after only calculating the distance to two vertices plus the distance to the border of the box instead of calculating the distance to all 100 vertices. However, in the case of Fig. 2 d_{border} is the smallest, and thus, there may be other vertices which are closer. Therefore the distances to all vertices in the adjacent boxes are calculated, which is the green hatched area. In this case there are eight additional vertices (see Fig. 1), for which V_3 in Fig. 2 is the nearest. The distance to V_3 is less than the distance to the border of

the yellow hatched area in Fig. 1, which means that no other vertex can be closer, and thus V_3 is the nearest vertex.

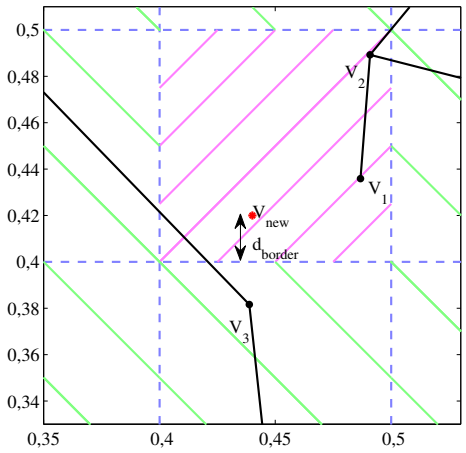

Fig. 2. A zoomed version of Fig. 1, where the three nearest vertices and the newly sampled vertex can be seen.

In this case the algorithm terminated after calculating the distance to 10 vertices, which is $1/10$ of the total number of vertices. When initially building the tree, there are only few vertices and hence a large overhead, as a large number of boxes need to be searched. But once there is a significant density of vertices, the box method is substantially faster than the brute force approach. The method is illustrated here in two dimensions, but is easily generalised to more dimensions.

The algorithm for the described box method is given in Algorithm 2.

Algorithm 2 The algorithm for the box method, which is used to find the nearest vertex (the red line in Algorithm 1).

NearestVertex(Tree, q.target)

```
1: boxNumber = FindBoxNumber(q.target)
2: searchArea = boxNumber
3: while Dist(q.target,searchBoundary)
   < Dist(q.target,q.nearest) do
4:    ExpandSearchArea()
5:    q.nearest = FindNearest(searchArea,q.target)
6: end while
7: InsertIntoBox(q.target,boxNumber)
8: return q.nearest
```

B. Time Complexity Analysis

We start by analysing the time complexity of the standard RRT algorithm as shown in Algorithm 1. The time it takes to add N vertices to the tree (disregarding the initialisation), can be calculated as the sum of the time for N iterations of each of the lines 4-7 in Algorithm 1:

$$T(N) = T_{sample}(N)+T_{nearest}(N)+T_{extend}(N)+T_{add}(N), \quad (2)$$

where each $T_{sample}, T_{nearest}, T_{extend}, T_{add}$ correspond to the lines 4-7 respectively. T_{sample} and T_{add} are simple operations, and can be done in linear time, so the complexity is $\mathcal{O}(N)$. The extension time, T_{extend}, of the tree can take considerable longer if e.g. collision checking or other intelligent extension strategies are used. These calculations do, however, not depend on the number of vertices already in the tree, and can thus still be done in linear time. Each time the nearest vertex has to be found, the distance to all previously added vertices must be calculated:

$$T_{nearest}(N) = \sum_{i=1}^{N}(i-1)T_{dist} = \frac{N^2-N}{2}T_{dist} \quad (3)$$

where T_{dist} is the time it takes to calculate the distance to any other vertex. So even though T_{dist} is small compared to T_{extend}, the complexity is $\mathcal{O}(N^2-N)$. By adding the derived complexity for the *sample, nearest, extend* and *add* operations respectively, the combined complexity for the brute force nearest neighbour search is:

$$\mathcal{O}(N) + \mathcal{O}(N^2-N) + \mathcal{O}(N) + \mathcal{O}(N) \approx \mathcal{O}(N^2) \quad (4)$$

It can be seen that the complexity is bounded by $\mathcal{O}(N^2-N)$, which is the complexity for the *nearest* operation. So when the tree becomes large, a significant part of the computation time is spend on calculating distances to other vertices, and much computing time can be saved if the nearest neighbour search is optimised.

In the following the complexity for finding the nearest vertex, when adding vertex number n is analysed. When using brute force, all $n-1$ previous vertices need to be searched, which therefore has complexity $\mathcal{O}(n)$.

Finding the nearest vertex using the box method depends of the number of boxes M. If M is large compared to n, the probability of having to search many of boxes is large, so this is not a good solution. Opposite, if $M \leq n$ the probability of having to search many boxes is small. In the case of evenly distributed vertices the average number of vertices to check will be n/M, and thus the complexity will be $\mathcal{O}(n/M)$. So optimally, M needs to be chosen approximately equal to n [11], which gives a complexity of $\mathcal{O}(1)$. However, as n increases in each iteration, this is not possible all the time. But experimentally we have found that a good value for M is around $N/2$, where N is the maximum number of vertices in the RRT.

While n is small, there is no benefit of using the algorithm, since too many boxes have to be searched. So if only inserting into boxes until n is larger than some value, increases the speed of the algorithm. In contrast to the brute force approach, there is an overhead when each vertex has to be inserted into the data structure, which contains the boxes. Calculating the corresponding box for a vertex in the box method takes constant time, so the insertion has complexity $\mathcal{O}(1)$, which does not change the overall complexity ($\mathcal{O}(1)$) of the method.

This complexity is compared to the most used tree based approach for minimising the time it takes to find the nearest

neighbour in motion planning problems, namely the kd-tree. According to [12], a kd-tree has a complexity for inserting a vertex of $\mathcal{O}(\log n)$. Furthermore, the complexity for finding the nearest vertex is $\mathcal{O}(n^{1-\frac{1}{d}})$, where d is the number of dimensions [12]. The combined complexity for searching and inserting is therefore $\mathcal{O}(n^{1-\frac{1}{d}} + \log n) \approx \mathcal{O}(n^{1-\frac{1}{d}})$.

This complexity analysis is based on a balanced tree. A disadvantage of the kd-tree is that if initial vertices are not well distributed, the tree will become unbalanced, and thus the performance degrades significantly. This can be somewhat compensated for if the tree is rebuilt after some iterations. The rebuild process has a complexity of $\mathcal{O}(n \log n)$, which does not change the amortised time complexity, since it is only done once or a couple of times. Using the box method, this not a problem, since each vertex belongs to one specific box, and the data structure is therefore always the same, and nothing is gained by rebuilding.

So for finding the nearest other vertex the complexity for the brute force method is $\mathcal{O}(n)$, the kd-tree has a complexity of $\mathcal{O}(n^{1-\frac{1}{d}})$, and if a proper number of boxes are chosen the box method has a complexity of $\mathcal{O}(1)$.

C. General Considerations

One disadvantage with both the box based algorithm and the kd-tree, is the overhead when only adding a few vertices. However, this happens only in limited cases, where the configuration space is relatively small and with few obstacles, in which case an RRT may not be the right algorithm to use anyway. So to obtain good exploration in general, it is useful to use as many vertices as possible, for which the box method performs well. This is especially the case in on-line applications, where there is a time limit for how long the planning can take. It is though, possible to overcome some of the initial overhead using the box based algorithm. Initially when the number of vertices is much smaller than the number of boxes ($n \ll M$), it is faster to just add vertices to the boxes, and do a brute force nearest neighbour search. When n grows, the algorithm can switch to searching in the boxes. Since the time it takes to add a vertex is small and constant (the complexity is $\mathcal{O}(1)$), it does not cost much overhead. It is also a possibility to dynamically increase the number of boxes in each dimension, as the tree grows large. This will enable the ratio n/M to stay close to an optimal value.

Another advantage of the box based algorithm is that it is easy to insert or remove vertices from the data structure compared to inserting and removing from the kd-tree. Inserting or removing vertices only requires the operation to add or remove the vertex from the corresponding box, which is $\mathcal{O}(1)$. Inserting a vertex in the kd-tree requires an $\mathcal{O}(log(n))$ search to find out where to insert. Removal of vertices is generally not easy for the kd-tree. It requires rebuilding the whole subtree beneath the place, where the vertex needs to be inserted. Conceptually it is also easier to understand and implement the nearest neighbour search using the box method, because a kd-tree search includes recursive visiting branches of the tree.

III. Experiments

Two different experiments have been set up to demonstrate the effectiveness of the box based nearest neighbour search algorithm in comparison to the brute force approach. First the box based algorithm is implemented together with a basic RRT algorithm, as shown in Algorithm 1, and the performance is compared to the brute force approach and the kd-tree. Secondly, the algorithm is used in a more realistic environment, where a robot plans a path in an environment with obstacles.

In the first experiment an RRT with 10000 vertices is build in a d-dimensional space of $1m$ in each dimension. This is done for $d = 2, 4, 6$ dimensions using a grid with 10 boxes in each dimension. The time for adding each vertex is continuously logged, and for comparison the same experiment is done using a brute force approach. For comparison to the kd-tree, an existing MATLAB implementation of the algorithm has been used [15]. However, this implementation of the algorithm does not support adding vertices to an existing tree, and the nearest neighbour search relies on balanced tree. To be able to compare to the other algorithms, a new tree is built each time a vertex is added, but only the time it takes to search for the nearest vertex, is then logged for the purpose of the experiment. This is only done for the two dimensional case.

In the second experiment a robot trajectory is planned through a two dimensional configuration space. The configuration space contains obstacles represented by a potential field, and vertices are pruned when the cost gets too high. The potential field represents persons moving in the area. For further information on the setup, see [16]. Fig. 3 illustrates an example of the potential field with a few vertices already added to the tree. The green dot to the left at $(2, 0)$ is the starting position of the robot, the red lines are possible trajectories, and the red dots are vertices. Furthermore, the red areas of the potential field are where there are persons, and hence where the robot should not go. First the tree with 10000 vertices is built using the box based distance search method. Then a tree is built using a brute force nearest neighbour search approach, where the algorithm is allowed to use the same amount of time it took to build the first tree. This experiment makes it possible to see the benefits of the box based method in an on-line application.

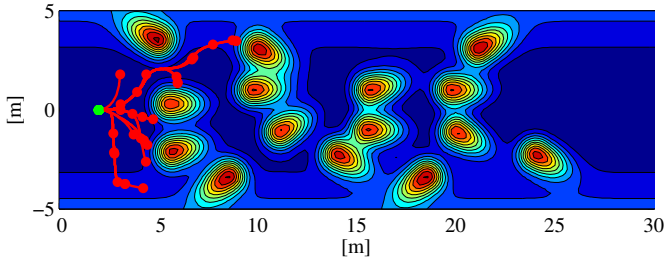

Fig. 3. An example of an RRT with 300 vertices in a potential field landscape, which is used for experimenting with the algorithm.

IV. Results

The time for building an RRT with 10000 vertices in two, four and six dimensions is shown in Fig. 4 using both the box algorithm and the brute force approach.

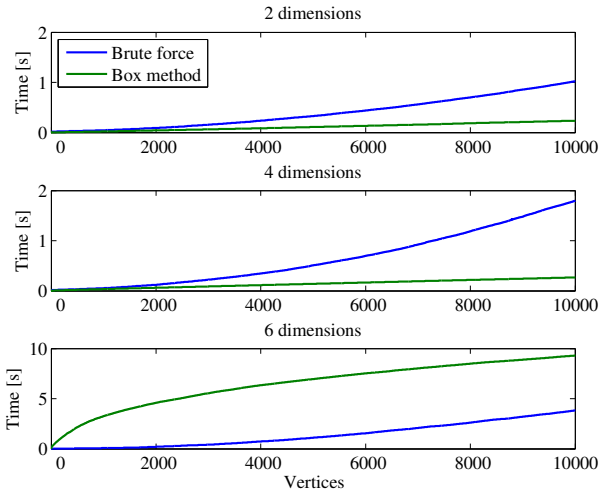

Fig. 4. From top to bottom the figure shows the time it takes for adding 10000 vertices in two, four and six dimensions respectively. The blue line shows the time it takes for the brute force approach and the green line shows the time for the box based method.

A comparison of the nearest vertex search times only, is done for two dimensions in Fig. 5 for the three methods; brute force, box based method, and kd-tree.

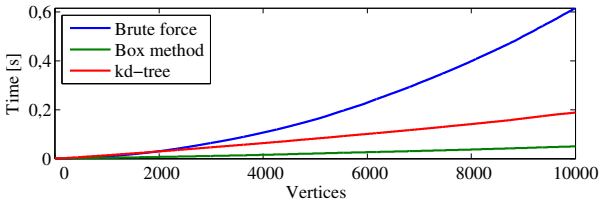

Fig. 5. Comparison of the time it takes to find the nearest neighbour for two dimensions using the kd-tree as well as the brute force method and the box method. Note that comparing to Fig. 4, the brute force and the box method are slightly faster, because only the time for finding the nearest vertex and not the other parts of the RRT algorithm is considered in this experiment.

In Fig. 6 an RRT with 10000 vertices is built in the potential field shown in Fig. 3. And in Fig. 7 the RRT is built using the brute force method for the same amount of time as used in Fig. 6 for the box method. In both figures only $1/10$ of the vertices are plotted to avoid cluttering the figure too much.

V. Discussion

The results demonstrate that the box algorithm clearly outperforms the brute force approach for two and four dimensions (the top and middle plot in Fig. 4). One thing to notice is that the box method has a larger advantage in four dimensions, than in two dimensions. Since there are 10 boxes in each dimension, there are 100 boxes in two

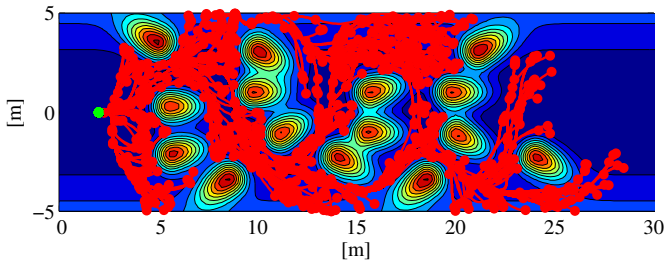

Fig. 6. A tree with 10000 vertices. Here the configuration space is explored very well.

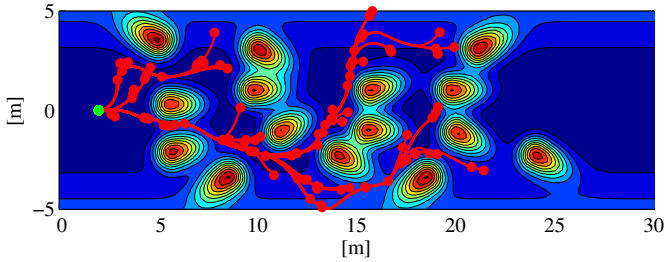

Fig. 7. A tree with approximately 2000 vertices, which does not explore the configuration space very well. This tree is build using brute force search, and has used the same runtime as the box based method in Fig. 6

dimensions and 10^4 boxes in four dimensions. As the optimal number of boxes is around $N/2$, the number of boxes in four dimensions is closer to the optimal number of boxes, which is approximately 5000. In the bottom plot of Fig. 4 it is seen that in 6 dimensions, it starts to take time for the box algorithm. This is a consequence of there being too many boxes compared to the number of vertices. The total number of boxes is 10^6. So if the number of vertices is in the range of a million, the algorithm would perform much better, which can also be seen on the trend of the figure. After for example 30000 vertices, which is not shown in Fig. 4, it is about twice a fast as the brute force method for six dimensions and at 300000 vertices it is 22 times faster. A way to reduce the computation time for this large number of boxes is to only subdivide some dimensions into boxes, like for example only the first three dimensions. This can also be utilised in planning problems, where the variance of the vertex locations varies much in different dimensions. It can be an advantage to have a larger number of boxes in the dimensions, were there is a large vertex variance. Possible examples are for a multilinked arm, where the first joint probably moves close around a nominal operating point, or for a mobile robot, where it may be desirable to explore the position part of the configuration space well, but e.g. the speed and orientation are around nominal values, or for configuration spaces with narrow passages in some dimensions.

According to the complexity calculations, the box based algorithm should have an approximate linear performance ($\mathcal{O}(N)$) when adding N vertices, if the correct number of boxes is chosen. In the middle figure of Fig. 4 it can be seen

that the box method performs almost linearly, which is in accordance with the theory. Similarly in the bottom figure for six dimensions, it is seen that the complexity is starting to get closer to linear as the number of vertices grows larger, even though there is a large overhead in the beginning.

The performance of the kd-tree implementation in Fig. 5 is seen to be worse than using the box based method. The used implementation of the kd-tree ensures that the search is always done in a balanced tree, which would not typically be the case in a real application, and thus in a real application, the kd-tree would perform worse than in this case. Furthermore, because of the complexity of the kd-tree search algorithm, $\mathcal{O}(n^{1-\frac{1}{d}})$, it performs worse in higher dimensions, where it can be seen in Fig. 4 that the box based method performs better for e.g. four dimensions. The speed of the kd-tree algorithm can, however, be improved for high dimensional search spaces by using approximate nearest neighbour (ANN) algorithms. This comes at a tradeoff for not being entirely sure that it is the correct nearest neighbour, which has been found, which is not desirable in some applications. Additionally the bound on the complexity for ANN is $\mathcal{O}(\log n)$ [8], [17], which is sill not better than the proposed method.

In Fig. 6 a tree with 10000 vertices is built. It is seen that the tree covers all of the obstacle free configuration space well. Contrary to this, Fig. 7 shows that the configuration space is not covered well, when planning using the brute force method for the same amount of time. Since using the box method cause much denser population of vertices, it is possible to choose trajectories, which are better. For example, it is seen in Fig. 7 that the tree tends to move along narrow branches in the middle between obstacles, and might thus not find narrow passages or a feasible trajectory closer to obstacles. It is hence clearly advantageous to use the box method to be able to explore the configuration space well.

In general we argue that the presented box based nearest neighbour search algorithm is better than both a brute force approach and the kd-tree for robot trajectory planning. The algorithm can be adapted to perform optimal for a given problem, e.g. the number of boxes in each dimension can be adjusted to improve performance, which is not possible for the kd-tree. It is also easier to implement than the kd-tree, and it is faster to insert and remove vertices from the data structure.

VI. Conclusion

In this paper we presented a practical algorithm for minimising the computational complexity of an RRT algorithm for robot path planning problems. It is shown that the time it takes to find the nearest neighbour in a standard RRT, using a brute force approach is substantial when the number of vertices in the tree grows. Thus, a lot of computation time can potentially be saved, if the nearest neighbour search is minimised. The proposed algorithm partitions the configuration space into a number of boxes, where only relevant boxes need to be searched to find the nearest vertex. The algorithm can be tuned to work better if an approximate bound on the maximum number of vertices is known. But generally the algorithm works better the larger the number of vertices there are in the tree. This also means that there is a relatively large overhead when only a small number of vertices needs to be added to the tree, which is not often the case. However, complexity calculations and simulations show that the proposed box based method performs better than both a brute force approach and using a kd-tree to find the nearest neighbour.

The algorithm can be used to increase the number of vertices, which it is possible to ad within a given time period, and can thus provide better trajectories, or compute a trajectory faster.

References

[1] S. LaValle, *Planning algorithms*. Cambridge Univ Pr, 2006.

[2] S. LaValle and J. Kuffner Jr, "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, no. 5, p. 378, 2001.

[3] Y. Kuwata, G. Fiore, J. Teo, E. Frazzoli, and J. How, "Motion planning for urban driving using rrt," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008. IROS 2008*, 2008, pp. 1681–1686.

[4] D. Ferguson and A. Stentz, "Anytime, dynamic planning in high-dimensional search spaces," in *Proc. IEEE International Conference on Robotics and Automation*, 2007, pp. 1310–1315.

[5] J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue, "Motion planning for humanoid robots," *Robotics Research*, vol. 15, pp. 365–374, 2005.

[6] A. Shkolnik, M. Walter, and R. Tedrake, "Reachability-guided sampling for planning under differential constraints," in *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, May 2009, pp. 2859–2865.

[7] R. Tedrake, "Lqr-trees: Feedback motion planning on sparse randomized trees," in *Proceedings of Robotics: Science and Systems*, Seattle, USA, June 2009.

[8] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," in *Proceedings of Robotics: Science and Systems*, Zaragoza, Spain, June 2010.

[9] S. Arya and M. David, "Mount, Approximate range searching," *Computational Geometry: Theory and Applications*, vol. 17, no. 3-4, pp. 135–152, 2000.

[10] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *J. ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[11] R. Sedgewick, *Algorithms in C*, M. A. Harrison, Ed. Addison-Wesley, Reading, MA, 1990.

[12] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations (Intelligent Robotics and Autonomous Agents)*. The MIT Press, June 2005. [Online]. Available: http://www.worldcat.org/isbn/0262033275

[13] S. Arya and D. Mount, "Approximate nearest neighbor queries in fixed dimensions," in *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1993, p. 280.

[14] D. Ferguson and A. Stentz, "Anytime rrts," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006, pp. 5369–5375.

[15] A. Tagliasacchi, "kd-tree for matlab," MATLAB Central File Exchange, Retrieved July 2010. [Online]. Available: http://www.mathworks.se/matlabcentral/fileexchange/21512

[16] M. Svenstrup, T. Bak, and H. J. Andersen, "Trajectory planning for robots in dynamic human environments," in *IROS 2010: The 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Taipei, Taiwan, October 2010.

[17] A. Yershova and S. M. LaValle, "Improving motion-planning algorithms by efficient nearest-neighbor searching," *Robotics, IEEE Transactions on*, vol. 23, no. 1, pp. 151 –157, 2007.