Published in the IEEE International Conference on Robotics and Automation. :2005-2011 (2010)

On the Implementation of Single-Query Sampling-Based Motion Planners

Ioan A. Şucan and Lydia E. Kavraki

***Abstract*— Single-query sampling-based motion planners are an efficient class of algorithms widely used today to solve challenging motion planning problems. This paper exposes the common core of these planners and presents a tutorial for their implementation. A set of ideas extracted from algorithms existing in the literature is presented. In addition, lower level implementation details that are often skipped in papers due to space limitations are discussed. The purpose of the paper is to improve our understanding of single-query sampling-based motion planners and motivate our community to explore avenues of research that lead to significant improvements of such algorithms.**

I. Introduction

This work aims to contribute to our understanding of single-query sampling-based planners [1], [2] and to promote the advancement of research towards truly substantial improvements of these planners as a whole. This paper systematizes and clarifies a large body of work by articulating *(a)* the common core of single-query sampling-based planners, *(b)* some of the existing heuristics that have been shown to work well in practice, and *(c)* some of the implementation details that are often left out in the corresponding papers but can strongly influence the performance of an algorithm. These implementation details are derived from the authors' implementation and/or use of existing sampling-based motion planning software libraries. This paper also reveals the breadth and wealth of research on this topic and can serve as a reference for future work.

Single-query sampling-based planners have become very popular due to their ability to quickly solve the motion planning problem: finding a continuous valid path from a given start state to a goal state, for a robotic system under a set of constraints [3]. Examples include Rapidly-exploring Random Trees (`RRT`) [4], [5], Expansive Space Trees (`EST`) [6], [7], Single-query Bi-directional probabilistic roadmap planner with Lazy collision checking (`SBL`) [8], and many more (e.g., [9]–[24]). We will generically refer to this class of algorithms as tree planners, due to the main data structure they employ. These planners iteratively grow a tree of motions in the state space of the robotic system (see Figure 1), using different heuristics. The tree is rooted at the starting state of the robotic system. At every iteration, an attempt is made to extend this tree with a new path segment (motion), towards a new state. An advantage of using trees is that they naturally encode the notion of order of states along a path and can be used to provide a time parametrization of states on paths. This becomes important when we are interested in specifying solution paths in terms of control inputs to the robotic system rather than a sequence of states between which a controller could interpolate. The tree data structure is a particular case of the roadmap used in the Probabilistic RoadMap (`PRM`) algorithm [25], hence some of our discussions apply to `PRM` as well. Vice-versa, some of the observations made for `PRM` in earlier work [1], [26] are relevant here.

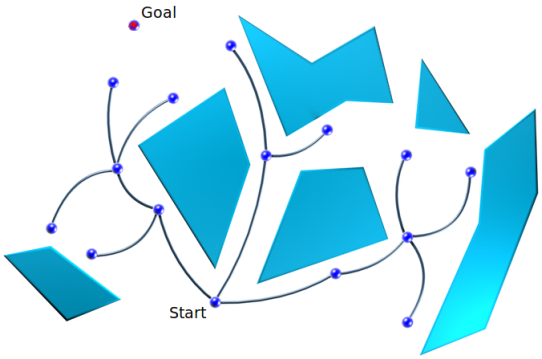

Fig. 1. Example tree of motions grown in the state space $\mathcal{X}$. Nodes represent states in $\mathcal{X}$ and edges represent valid motions between states.

From a theoretical standpoint, the objective of a tree planner is to grow the tree of motions in such a manner that the entire state space can be eventually covered. However, coverage does not always need to be achieved before finding a solution. In general, it is considered a good property if such a planner is probabilistically complete [1] – if a solution exists, it will eventually be found. From a practical point of view, the performance of tree planners – the amount of time spent to find a solution – is very important. This is in fact a key motivation for the development of tree planners. Many algorithms for guiding tree growth have been introduced over the years (e.g., [4]–[24]) with the purpose of improving performance. A number of software libraries for motion planning containing such algorithms have also been developed: `MSL` (Motion Strategy Library) [27], `MPK` (Motion Planning Kit) [28], `OpenRAVE` [29], `OOPSMP` (Object-Oriented Programming System for Motion Planning) [30], `ompl` (Open Motion Planning Library) [31].

Even though tree planners are conceptually simple, correct and efficient implementations are not trivial. In this work, we isolate some of the more prominent ideas used by tree planners and discuss a series of details that arise during their implementation, details that often get left out of papers due

This work supported in part by NSF IIS 0713623, NSF DUE 0920721 and Rice University Funds.

I. A. Şucan and L. E. Kavraki are with Department of Computer Science Rice University, 6100 Main St., Houston, TX {isucan, kavraki}@rice.edu

to space constraints. While this text is intended primarily for readers interested in implementing their own single-query sampling-based motion planner, we believe readers interested in simply using existing implementations will find this text helpful for better understanding and tuning the implementations they use.

This paper is structured as follows. We first present the interface and the typical execution of a tree planner in Section II. The state space and related primitives are described in Section III. In Section IV we give an overview of some of the ideas introduced by previous work. We then continue with lower level details that often get left out in Section V and some tips on debugging tree planners in Section VI. Finally, we conclude in Section VII.

II. Input, Output and Execution of a Tree Planner

Let $\mathcal{X}$ be the state space in which the tree planner operates. For every motion planning query, the following input should be specified:

- Specification of a starting state $s \in \mathcal{X}$. This is where the robotic system is considered to start at.
- Specification of a goal region $\mathcal{G} \subset \mathcal{X}$, $\mathcal{G} \neq \emptyset$. In the simplest case, this can be a state in $\mathcal{X}$ ($\mathcal{G} = \{g\}, g \in \mathcal{X}$). Some algorithms are only applicable if the explicit representation of the goal state is available. More generally, $\mathcal{G}$ is implicitly specified through the use of an indicator function that decides whether a given state is in the goal region or not ($\mathcal{G} = \{x \in \mathcal{X} | g(x) = true$ for some $g : \mathcal{X} \rightarrow \{true, false\}\}$).
- Allowed time $t \in \mathbb{R}^+$. This is the amount of time the planner is allowed to search the state space before reporting failure.

The output of a tree planner is a valid solution path. In case of failure, the solution path is empty. The path can be represented as:

- A sequence of states, i.e., a kinematic path
- A sequence of inputs, i.e., a control path. In case we are planning with controls, the path is discretized with respect to time. Every element of the path will consist of the state at that time, the control applied when in that state and the amount of time the control is applied for.

The execution of a typical tree planner proceeds as follows:

Algorithm BuildTree($\mathcal{X}$ s, $\mathcal{G}$, t)

```
Init(T, X, s)      // unless T already initialized
while ElapsedTime() < t and NoGoalFound(G) do
    x_tree ← StateToExpandFrom(T)
    p_add ← PathToConsider(x_tree)
    if ChooseToAdd(p_add) then
        Insert(T, p_add)
    end if
end while
return T
```

III. State Space and Related Primitives

The state space $\mathcal{X}$ is a manifold consisting of all the states a robotic system could potentially attain. The following represents a minimal list of state space related primitives that tree planners depend on:

- A bounding box for an ambient space $\subset \mathbb{R}^d$ surrounding $\mathcal{X}$. Note that the dimension of this ambient space can sometimes be larger than that of $\mathcal{X}$. We use the bounding box of an ambient space instead of that of the state space to avoid the complexities that arise from the topology of $\mathcal{X}$.
- A bounding box for the control space $\mathcal{U} \subset \mathbb{R}^k$. This is only needed if we are interested in obtaining control paths. Each component of an element in $\mathcal{U}$ represents an input for the robotic system.
- State validator $valid : \mathcal{X} \rightarrow \{true, false\}$. $valid(x) = true$ for $x \in \mathcal{X}$ implies x is a valid state. This usually means x is at least collision free. Often, additional constraints need to be satisfied by x. A more advanced definition of $valid$ is $valid : \mathcal{X} \rightarrow \mathbb{R}$, where $valid(x)$ represents the distance to the nearest invalid state. This latter definition can be used for exact collision checking [32] and the so-called "continuous collision detection" [33].
- Metric $dist : \mathcal{X} \times \mathcal{X} \rightarrow [0, \infty)$. This is optional, but many algorithms need to evaluate distance between states. Depending on the state space, defining this metric may be difficult.
- A low-level function for sampling states. This is usually uniform sampling based on pseudo-random number generators (see [2] for other generators, such as quasi-random). It is very important that the topology of the state space is accounted for in this routine, as this is a common source of error. For certain spaces, uniform sampling of states can be implemented by uniform sampling in each dimension of the state space. However, this is not generally the case. For instance, spaces such as SE(3) need special attention to make sure the sampling is uniform [34]. Based on this low level functionality, the planning algorithm can implement different sampling distributions (e.g., [9]–[11] and many more).
- A function for state expansion. The purpose of this function is to move away from a given state, so that the tree expansion can be continued. In practice, this function is often in the form of a local planner or a model of motion:

  – A local planner. A function of the form $local : \mathcal{X} \times \mathcal{X} \times [0, 1] \rightarrow \mathcal{X}$ generates the states that lie on a path segment between two given end-points. The topology needs to be considered here as well [34]. Note that when planning with controls such a function can be defined only for specific robotic systems [1].

  – Propagation of a control from a given state (forward propagation). A function $propagate : \mathcal{X} \times \mathcal{U} \times [0, \infty) \rightarrow \mathcal{X}$ generates the states the robot passes through when

applying a given control for a given amount of time starting at a given state. This represents the model of motion for the robot.

IV. Tree Planner Heuristics

Many of the algorithms introduced over the years present ideas that can be combined and reused. Earlier on, the heuristics a tree planner employed to expand its tree data structure were what defined the planner (see for instance, EST and RRT). As the research progressed, many other ideas were introduced, combinations of existing ideas were proposed, blurring the distinction among different tree planners. In this section we aim to provide a series of ideas extracted from algorithms that have been shown to work well in practice. Many of these ideas (but not all) are compatible, meaning that they can be combined to produce different algorithms. Depending on the task, one could create an algorithm with increased performance. To evaluate the performance of individual ideas we show relevant experimental results. When such results exist in the literature we provide a summary of those results. For the cases where no experimental data was found, we present our own experiments.

A. Selecting States for Further Expansion

Deciding which parts of the tree of motions merit further exploration is a fundamental step in the execution of a tree planner and it weighs heavily on the planner's overall performance. This decision is problem-dependent and at this time it is unclear whether an optimal approach to making this decision exists. In this section we present some of the better-known techniques for selecting nodes to be expanded, but this list is by no means comprehensive. The research on this topic is so extensive that presenting it entirely is simply not feasible. A small sample of this research is referenced in this work [4]–[24], [35]–[38].

1) Using Voronoi bias: One of the most successful ideas is to extend the tree of motions towards a random state, starting from the state in the tree that is nearest to that random one [4]. Choosing states to expand from in this fashion guides the tree of motions towards the largest Voronoi regions. This approach does not guarantee the tree never grows onto itself but seems to work well as long as a good distance metric is available [35], [39].

2) Using the out-degree of the nodes in the tree: Focusing on continuing the tree growth from nodes that have a lower out-degree is likely to take the search into unexplored space. This approach defines a probability distribution over the nodes in the tree of motions and selects nodes for expansion according to this distribution [6].

3) Using a decomposition of the state space: A similar technique is to split the state space into cells. These cells can be defined by grids imposed on the space [8]. Every new node added to the tree is also placed in one of the defined cells. When continuing the tree expansion, nodes from emptier cells are preferred. This implies a probability distribution is defined over the cells in the grid. Since there are typically fewer cells than nodes, selecting a node to expand from is a more efficient process. This approach has been shown to work well for difficult problems [8].

Using such decompositions can become problematic if the number of cells is large. The number of cells can become very large for high-dimensional spaces. As the tree increases in size, it is likely more will be gained from continuing the tree expansion from the cells corresponding to the boundary of the explored space. This can be achieved by keeping track of the number of neighbors for each cell [21]. Another possible improvement is to use multiple levels of decomposition: we can define larger cells that are themselves split into smaller cells [21]. This combination of ideas can lead to one and even two orders of magnitude speedup, depending on the model of the robot and the environment, as experiments in [21] show.

Another method of decomposing the state space is hierarchical decomposition [19]. The state space is assumed to be bounded and considered to be one large cell at the beginning of the exploration. As the tree grows, the cells being expanded from get split in half. This approach proceeds deterministically by maintaining a queue of cells that need to be explored, prioritized by their volume and the iteration number of their last exploration step [19].

4) Projecting the state space to lower dimensional spaces: More recently, a number of algorithms employ projections from the state space to a lower dimensional space to be used in conjunction with decompositions. The intention is to approximate the coverage of the state space by evaluating the coverage in the projected space. This approach was introduced since evaluating coverage in lower dimensional spaces is easier, as such spaces can often be decomposed into a manageable number of pieces. Although not explicitly mentioned, orthogonal projections are suggested in [8]. Using projections to the workspace has been shown to be useful for mobile robots [20]. To the authors' knowledge, the first explicit use of generic projections is in [40]. It is often the case that even simple, intuitive projections perform well [21], but it is unclear whether this can be done in general [41].

B. Using a Notion of Direction

Accounting for the direction of expansion is another important idea that helps in guiding the tree expansion. Keeping track of previously used directions increases the chance of using a better direction of expansion [18], [35]. In the case of narrow passages, Principal Component Analysis (PCA) can be used locally to find a good direction of growth. This use of PCA can lead to a speedup of up to one order of magnitude, depending on the environment, as reported in [37].

A related idea is that of computing discrete paths that lead to the goal [20]. These discrete paths are a sequence of cells in a decomposition of the workspace, one that connects the starting state to the goal region. Even though these discrete plans are in the workspace, and as such, cannot be directly converted into state space plans, they can serve as a guide, a means to lead the state space exploration. It is typically the case that computing these discrete motion plans is much easier and much faster. As the state space exploration

proceeds, gained information can be used to recompute the discrete plan being used as a guide. This interplay of discrete and continuous search speeds up exploration. It has been shown that in certain mobile robotics applications, a speedup of up to two orders of magnitude can be obtained [20].

C. Bi-directional Search

A very successful technique for improving the performance of tree planners is bi-directional search. This is a search technique borrowed from artificial intelligence that has also been used successfully in the context of tree-based planning [6], [8], [12] – speedup factors of 3 to 4 are reported in [12]. Bi-directional search means that instead of growing a single tree from the start state towards the goal region, two trees are grown: one from the start state towards the goal region and one from the goal region towards the start state. Note this method requires that we have a means of sampling the goal region, or we know the actual goal state [42]. The two trees can take turns at being grown or can be grown in parallel. After each iteration that adds a motion to a tree, an attempt is made to connect to the other tree [8], [12]. If this attempt is successful, a solution path has been found: the path from the start state to the connection state concatenated with the reversed path from the goal to the connection state. Note this is a second requirement for this technique to be applied: the paths that are added to the trees need to be reversible. This is usually the case when computing kinematic paths or when systems of differential equations are used to model the motion along a path segment. However, when using physics-based simulation, the paths can no longer be reversed. An additional problem with bi-directional search is that when planning with controls, even if paths can be reversed, gaps between the two trees need to be closed, and this may be non-trivial [13], [43].

D. Lazy Collision Evaluation

Another very successful idea is that of lazy collision checking [8], [44]. Since collision checking usually takes more than 90% of a sampling-based planner's execution time, it is desired to minimize the number of collision evaluations. A method to do this is through the use of lazy collision evaluation. This means that all states on all paths are assumed valid until a solution is found. The path segments that make up the found solution are then checked for collision. If a segment is found to be valid, it is marked as such. If it is not valid, it and its descendants are removed from the tree. If the entire path was found to be valid, the algorithm completes successfully. If the solution was found to be invalid, the tree continues to be grown, remembering the parts that were marked as valid. This technique allows the planner to check collisions only for the path segments it tries to use as part of the solution, leaving other ones unchecked, thus reducing the number of total collision evaluations.

As an example of the speedup that can be obtained with lazy collision evaluation we present a comparison of two algorithms from the `ompl` [31] library: `EST` and its bi-directional implementation with lazy collision checking, `SBL`. Our own experiments[1] show that performing this comparison for the problem of moving a 7 degree-of-freedom manipulator in the presence of obstacles, from above to underneath a dining table, a speedup factor of 6.3 in favor of `SBL` can be observed.

E. Goal Biasing

If states in the goal region can be sampled (or are known a priori), the tree growth can be biased to grow towards these states. Biasing can be done for example by attempting to connect to goal states periodically (e.g., [14]) or by growing the tree from states that are closer to the goal (e.g., [45]). For the latter approach we need either a distance metric or a heuristic to evaluate the distance to the goal. While this method can lead to significant speedup (we report speedup by a factor of 3 to 90 in [45]), it can also degrade the performance of an algorithm when the solution path first needs to go farther from the goal and then back towards it (`RRT` slowed down by a factor of up to 3 in [21]). Using this approach in conjunction with a learning technique that limits growing towards the goal from specific regions, in case of repeated unsuccessful attempts, may alleviate the problem [38].

F. Projection onto the Constraint Space

If the space of valid samples has a small volume with respect to the state space, most of the sampled states will be invalid. This can lead to significant performance degradation. For instance, if a robot arm is asked to manipulate an open container, we most likely want the arm not to spill the content. This means we will constrain at least one degree of freedom for the arm to a very small range, which effectively makes the volume of the manifold of valid states in the state space be 0. The chance of sampling valid states is then practically null. In such cases, techniques that project samples onto the lower dimensional constraint manifold can be employed [46]. For a review of some methods of sampling in such lower dimensional manifolds, the reader is directed to [47], where three methods are experimentally evaluated: Randomized Gradient Descent, Tangent Space Sampling and First-Order Retraction, with the conclusion that First-Order Retraction is the preferred option.

G. Using Motion Primitives

In certain cases we may be interested in limiting the set of motions a robot is allowed to make. This can be done for instance by discretizing the control space $\mathcal{U}$ [16]. Such an approach is reported to achieve speedup factors of 3 to 20, for some problems [16]. The notion of a maneuver automaton [48] can be used to define a formal language on motion segments that can be used to form valid paths. Such techniques help with a more systematic exploration in the control space: one can guarantee that two controls that are very similar to one another are not both evaluated. The disadvantage is however that selecting a finite set of controls

[1]The data for all experiments conducted for this paper is available at `http://kavrakilab.org/data/ICRA2010TP/index.html`

from an infinite control space may prevent finding solutions when they exist.

H. Parallel Execution

It has been shown that sampling-based planners perform very well when using parallelization, be that an embarrassingly parallel setup [49] (running multiple instances of the planner until one of them finds a solution) or using shared memory parallelism [21]. Due to the randomized nature of the algorithms, super-linear speedup can be observed with respect to computation time, as shown in [21].

V. The Little Details

This section is a list of details the authors feel are important to have in mind when implementing a sampling-based motion planner. The order in which these details are presented roughly follows the implementation of a typical tree planner. For the experiments we conducted, the OOPSMP [30] library was used and all reported values are averaged over 100 runs, on a 2.83 Ghz CPU with 8 GB RAM running Ubuntu Linux.

A. How Far to Grow a New Motion

Based on the set of heuristics used (from Section IV), the algorithm has chosen a state in the tree it is about to expand from and a direction of expansion. The question that remains is how far to expand this motion. In general, a good approach is to grow the tree until an obstacle is hit or some maximum length is reached. Defining minimum and maximum lengths of motions to be added in the tree prevents us from having too many short motions around a single state and from bouncing from one side to the other in the state space. Depending on the space we are planning in, the parameters for minimum and maximum motion lengths likely need to be adjusted in order to get reasonable progress (more on how to evaluate the progress in Section VI). Keeping somewhere around 90% of the valid part of the motion is potentially better than keeping the entire valid part, since the last valid state may be too close to a collision and further expansions from there would be unsuccessful.

To demonstrate the influence of the length of added motions on the runtime of a sampling-based planning algorithm we show running times of EST with different lengths for added motions for a free-flying robot in 2D.

New motions in our EST implementation are started from some state s, already existing in the tree, and extend to a state d, sampled around s using a Gaussian probability distribution. We thus control the length of the new motions by changing the standard deviation of the sampling distribution for d. We show some experiments[2] in Table 1. For low standard deviation, we have slower progress, so higher runtimes, and for standard deviation that is too high, we are bouncing from one side of the state space to the other, again increasing runtime.

[2]The data for all experiments conducted for this paper is available at http://kavrakilab.org/data/ICRA2010TP/index.html

Std. Dev.	0.1	0.2	0.5	1.0	2.0	5.0
Runtime(s)	1.00	0.56	0.40	0.41	0.48	0.63

Table 1. Runtime of EST with varying parameters for extending motions.

When expansion is attempted in narrow passages, it is quite possible even short motions would cause collisions. A means to address this issue is to allow a small penetration of obstacles. This leads to higher chances of finding samples in narrow passages, but introduces erroneous samples. These erroneous samples can however be replaced by valid ones through a re-sampling process in a small vicinity of the penetrating sample [50].

A similar idea is to "retract" the robot and compute valid states along the surface of the colliding obstacle, thus generating a set of candidate motions that take the robot through the narrow passage [51]. This increases the chances of traversing the narrow passage, as more paths inside it are being evaluated. Speedup of more than two orders of magnitude is reported in [51] for particularly difficult narrow passage problems.

B. Intermediate States on Motions

Adding intermediate states along motions can lead to computational speedup, if the number of added states is not too large. In Table 2 we show the benefits of running RRT with adding of intermediate states at varying resolutions, along the generated motions, for a free-flying robot in 3D[2]. The resolution specifies the distance between added intermediate states. We report the speedup achieved with respect to the algorithm running without adding intermediate states (∞ resolution). We observe that adding too many states can slow us down, but we can also obtain speedup of up to 20% for appropriate values of the resolution.

Resolution	Runtime(s)	Speedup
∞	0.367	1.00
0.005	0.492	0.75
0.010	0.310	1.19
0.050	0.334	1.10
0.100	0.339	1.08
0.500	0.317	1.16
1.000	0.307	1.20
5.000	0.369	1.00

Table 2. Runtime of RRT with intermediate states at varying resolutions.

C. Continuation of Exploration

Since tree planners cannot decide that a solution does not exist, they will simply fail after the amount of time allowed for computation elapses. In some cases, a little additional computation time can lead to finding a solution. For this reason it is best to organize the tree planner in such a way that a subsequent call without clearing the data structures in the meantime continues the exploration using the previous tree of motions. Of course, this assumes the environment remains unchanged in between calls.

When the environment does change in between calls to the motion planner, parts of the tree of motions become invalidated. Instead of starting with a new tree, parts of the tree that remain valid can be kept [36], [52].

D. When Using Physics Simulation

When using physics-based simulators (such as ODE [53]), the simulation can become unstable during planning, especially with random selection of controls. This needs to be detected, to avoid obtaining erroneous solutions. In general, sanity checks such as verifying that joints have not been broken and the positions of bodies are valid floating point numbers are sufficient.

Another potential problem with physics simulation is that the results of forward propagation may not seem deterministic if different time steps are used during the planning process (e.g., ODE). For this reason, it is recommended that a constant time step be used throughout the planning process.

E. Path Shortening and Smoothing

Due to the randomized nature of the planners discussed in this work, the obtained solution paths usually contain unnecessary and awkward maneuvers. However, post-processing solution paths using shortening and smoothing algorithms [54], [55] is possible and is encouraged.

VI. Debugging a Tree Planner

This section consists of advice on how to debug and test a tree planner, and a list of suggestions that may help with the development. Once it is implemented, a tree planner is difficult to debug, due to its randomized nature. In addition to typical software engineering approaches, it is recommended that the amount of randomness in the execution of the planner is minimized (fixing the random seed, running in a single thread). Typical things to test are:

1) Running the algorithm on toy problems, where solutions are known to exist. Visualizing the solutions and on-screen projections of the tree of motions is actually one of the best ways to check whether they are correct. Checking whether the states along obtained solution paths are inside the bounding box of the state space is also a good idea. This test should be repeated with different random seeds and different number of threads, if applicable.

2) Assuming the solutions of toy problems seem correct, the next step is to run on more complex problems and compute statistics such as number of iterations, number of created states, average path segment length, average runtime until a solution is found. To make sure the algorithm is doing what it is supposed to, counting events is very useful. What this means is that various counters should be added to the code so that we can check how often certain pieces of code are executed, on average (e.g., to check whether goal biasing is used as often as we would like).

If we are interested in comparing a newly implemented motion planner to other existing ones, a set of benchmark problems will be needed. Unfortunately, there is no known set of good benchmark problems, to the authors' knowledge. In fact, the notion of "good benchmark" is as of yet unclear[3]. On the selected problems, looking at the runtime is important. Due to the random nature of the algorithm, averages need to be taken over multiple runs. In addition to the runtime, the variance of the runtime is important as well: a planner that has lower variance in runtime tends to be more reliable. The average number of states in the tree, used memory, number of calls to collision checker (or physics simulator) are other values to look at.

VII. Conclusions

In this paper we assumed motion planning was performed for robotic systems. However, it should be noted that with small adjustments, motion planning algorithms can be applied to protein folding, digital actors and other problems [1]. As a general rule of thumb, bi-directional search is preferred if a means to sample the goal region is available and the paths of the robotic system are reversible. Lazy collision checking is preferred, if it can be applied. In terms of guiding the tree exploration, the idea of leading the exploration using discrete paths in a projection of the state space provides significant computational advantages. If narrow passages are prevalent, techniques such as the ones based on PCA or retraction should be used. Depending on the task, more of the ideas presented in previous sections can prove beneficial. Furthermore, shared-memory parallelization is advised, if multiple compute cores are available.

The material in this paper is by no means exhaustive and the reader is encouraged to see the referenced literature for more details. However, we made an effort to present high-level decisions and low-level aspects that can lead to the implementation of a good single-query sampling based-motion planner. The high-level decisions are in fact a set of ideas extracted from algorithms that have been shown to perform well in practice. The low-level aspects are details that usually get left out from papers in the interest of space. The purpose of collecting this information in a single paper is to improve our understanding of sampling-based tree planners and to motivate our community to seek truly substantial improvements to these planners as a whole.

Acknowledgements

The authors would like to thank the reviewers, J.-C. Latombe, D. Hsu, K. Bekris, R. Rusu, M. Ciocârlie and the members of the Kavraki Lab for providing valuable comments. Furthermore, the authors thank Marius Şucan for drawing the image in this document.

References

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

[2] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, available at http://planning.cs.uiuc.edu/.

[3] J.-C. Latombe, *Robot Motion Planning*. Boston, MA: Kluwer Academic Publishers, 1991.

[3]Some problems that are known to be more difficult are available at http://parasol-www.cs.tamu.edu/dsmft/benchmarks/

[4] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Computer Science Dept., Iowa State University, Tech. Rep. 11, 1998.

[5] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *Intl. Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

[6] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in *Intl. Conf. on Robotics and Automation*, vol. 3, April 1997, pp. 2719–2726.

[7] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *Intl. Journal of Robotics Research*, vol. 21, no. 3, pp. 233–255, March 2002.

[8] G. Sánchez and J.-C. Latombe, "A single-query bi-directional probabilistic roadmap planner with lazy collision checking," *Intl. Journal of Robotics Research*, vol. 6, pp. 403–417, 2003.

[9] H. Kurniawati and D. Hsu, "Workspace importance sampling for probabilistic roadmap planning," in *Intl. Conf. on Intelligent Robots and Systems*, September 2004, pp. 1618–1623.

[10] S. Thomas, M. Morales, X. Tang, and N. M. Amato, "Biasing samplers to improve motion planning performance," in *Intl. Conf. on Robotics and Automation*, April 2007, pp. 1625–1630.

[11] D. Hsu, G. Sanchez-Ante, and Z. Sun, "Hybrid PRM sampling with a cost-sensitive adaptive strategy," in *Intl. Conf. on Robotics and Automation*, April 2005, pp. 3874–3880.

[12] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Intl. Conf. on Robotics and Automation*, April 2000, pp. 995–1001.

[13] P. Cheng, E. Frazzoli, and S. M. LaValle, "Improving the performance of sampling-based planners by using a symmetry-exploiting gap reduction algorithm," in *Intl. Conf. on Robotics and Automation*, vol. 5, April 2004, pp. 4362–4368.

[14] S. M. LaValle and J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," in *New Directions in Algorithmic and Computational Robotics*, B. Donald, K. Lynch, and D. Rus, Eds. AK Peters, 2001, pp. 293–308.

[15] S. Rodriguez, S. Thomas, R. Pearce, and N. M. Amato, "RESAMPL: A region-sensitive adaptive motion planner," in *Intl. Workshop on the Algorithmic Foundations of Robotics*, New York City, July 2006.

[16] M. Kalisiak and M. v. d. Panne, "RRT-blossom: RRT with a local flood-fill behavior," in *Intl. Conf. on Robotics and Automation*, May 2006, pp. 1237–1242.

[17] I. Belousov, C. Estevès, J.-P. Laumond, and E. Ferré, "Motion planning for the large space manipulators with complicated dynamics," in *Intl. Conf. on Intelligent Robots and Systems*, August 2005, pp. 2160–2166.

[18] B. Burns and O. Brock, "Single-query motion planning with utility-guided random trees," in *Intl. Conf. on Robotics and Automation*, April 2007, pp. 3307–3312.

[19] A. M. Ladd and L. E. Kavraki, "Motion planning in the presence of drift, underactuation and discrete system changes," in *Robotics: Science and Systems*, Boston, Massachusetts, June 2005, pp. 233–241.

[20] E. Plaku, M. Y. Vardi, and L. E. Kavraki, "Discrete search leading continuous exploration for kinodynamic motion planning," in *Robotics: Science and Systems*, W. Burgard, O. Brock, and C. Stachniss, Eds. Atlanta, Georgia: MIT Press, June 2007, pp. 326–333.

[21] I. A. Şucan and L. E. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Intl. Workshop on the Algorithmic Foundations of Robotics*, Guanajuato, Mexico, December 2008.

[22] R. Diankov, N. Ratliff, D. Ferguson, S. Srinivasa, and J. J. Kuffner., "Bispace planning: Concurrent multi-space exploration," in *Robotics: Science and Systems*, Zurich, Switzerland, June 2008.

[23] Y. Li and J. Xiao, "On-line planning of nonholonomic trajectories in crowded and geometrically unknown environments," in *Intl. Conf. on Robotics and Automation*, May 2009, pp. 3230–3236.

[24] A. Shkolnik, M. Walter, and R. Tedrake, "Reachability-guided sampling for planning under differential constraints," in *Intl. Conf. on Robotics and Automation*, May 2009, pp. 2859–2865.

[25] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, August 1996.

[26] D. Hsu, J.-C. Latombe, and H. Kurniawati, "On the probabilistic foundations of probabilistic roadmap planning," *Intl. Journal of Robotics Research*, vol. 25, no. 7, pp. 627–643, 2006.

[27] "http://msl.cs.uiuc.edu/msl/."

[28] "http://robotics.stanford.edu/~mitul/mpk/."

[29] "http://openrave.programmingvision.com/."

[30] "http://kavrakilab.org/oopsmp/."

[31] "http://www.ros.org/wiki/ompl/."

[32] F. Schwarzer, M. Saha, and J.-C. Latombe, "Exact collision checking of robot paths," in *Intl. Workshop on the Algorithmic Foundations of Robotics*, Nice, France, December 2002.

[33] M. Tang, Y. J. Kim, and D. Manocha, "C2A: Controlled conservative advancement for continuous collision detection of polygonal models," in *Intl. Conf. on Robotics and Automation*, May 2009, pp. 849–854.

[34] J. J. Kuffner, "Effective sampling and distance metrics for 3D rigid body path planning," in *Intl. Conf. on Robotics and Automation*, vol. 4, April 2004, pp. 3993–3998.

[35] P. Cheng and S. M. LaValle, "Reducing metric sensitivity in randomized trajectory design," in *Intl. Conf. on Robotics and Automation*, May 2001, pp. 43–48.

[36] K. E. Bekris and L. E. Kavraki, "Greedy but safe replanning under kinodynamic constraints," in *Intl. Conf. on Robotics and Automation*, April 2007, pp. 704–710.

[37] S. Dalibard and J.-P. Laumond, "Control of probabilistic diffusion in motion planning," in *Intl. Workshop on the Algorithmic Foundations of Robotics*, Guanajuato, Mexico, December 2008.

[38] L. Jaillet, A. Yershova, S. M. LaValle, and T. Siméon, "Adaptive tuning of the sampling domain for dynamic-domain RRTs," in *Intl. Conf. on Intelligent Robots and Systems*, August 2005, pp. 2851–2856.

[39] A. Pamecha, I. Ebert-Uphoff, and G. S. Chirikjian, "Useful metrics for modular robot motion planning," *IEEE Transactions on Robotics and Automation*, vol. 13, no. 4, pp. 531–545, 1997.

[40] A. M. Ladd, "Direct motion planning over simulation of rigid body dynamics with contact," Ph.D. dissertation, Rice University, Houston, Texas, December 2006.

[41] I. A. Şucan and L. E. Kavraki, "On the performance of random linear projections for sampling-based motion planning," in *Intl. Conf. on Intelligent Robots and Systems*, October 2009, pp. 2434–2439.

[42] D. Berenson, S. Srinivasa, D. Ferguson, A. Collet, and J. J. Kuffner, "Manipulation planning with workspace goal regions," in *Intl. Conf. on Robotics and Automation*, May 2009, pp. 618–624.

[43] F. Lamiraux, E. Ferr, and E. Vallee, "Kinodynamic motion planning: Connecting exploration trees using trajectory optimization methods," in *Intl. Conf. on Robotics and Automation*, vol. 4, April 2004, pp. 3987–3992.

[44] R. Bohlin and L. Kavraki, "Path planning using Lazy PRM," in *Intl. Conf. on Robotics and Automation*, April 2000, pp. 521–528.

[45] I. A. Şucan, J. F. Kruse, M. Yim, and L. E. Kavraki, "Kinodynamic motion planning with hardware demonstrations," in *Intl. Conf. on Intelligent Robots and Systems*, September 2008, pp. 1661–1666.

[46] D. Berenson, S. Srinivasa, D. Ferguson, and J. J. Kuffner, "Manipulation planning on constraint manifolds," in *Intl. Conf. on Robotics and Automation*, May 2009, pp. 625–632.

[47] M. Stilman, "Task constrained motion planning in robot joint space," in *Intl. Conf. on Intelligent Robots and Systems*, October 2007, pp. 3074–3081.

[48] E. Frazzoli, M. A. Dahleh, and E. Feron, "Maneuver-based motion planning for nonlinear systems with symmetries," *IEEE Transactions on Robotics and Automation*, vol. 21, no. 6, pp. 1077–1091, December 2005.

[49] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Intl. Conf. on Robotics and Automation*, May 1999, pp. 688–694.

[50] D. Hsu, L. E. Kavraki, J.-C. Latombe, R. Motwani, and S. Sorkin, "On finding narrow passages with probabilistic roadmap planners," in *Intl. Workshop on the Algorithmic Foundations of Robotics*, Houston, Texas, March 1998.

[51] L. Zhang and D. Manocha, "An efficient retraction-based RRT planner," in *Intl. Conf. on Robotics and Automation*, May 2008, pp. 3743–3750.

[52] D. Ferguson, N. Kalra, and A. Stentz, "Replanning with RRTs," in *Intl. Conf. on Robotics and Automation*, May 2006, pp. 1243–1248.

[53] "http://www.opende.sourceforge.net."

[54] R. Geraerts and M. Overmars, "Clearance based path optimization for motion planning," in *Intl. Conf. on Robotics and Automation*, vol. 3, April 2004, pp. 2386–2392.

[55] B. Raveh, A. Enosh, and D. Halperin, "A Little More, a Lot Better: Improving Path Quality by a Simple Path Merging Algorithm," *ArXiv e-prints*, Jan. 2010.