Continuous Arvand: Motion Planning with Monte Carlo Random Walks

Weifeng Chen and Martin Müller
Department of Computing Science
University of Alberta
{weifeng3,mmueller}@ualberta.ca

Abstract

Sampling-based approaches such as Probabilistic Roadmaps and Rapidly-exploring Random Trees are very popular in motion planning. Monte Carlo Random Walks (MRW) are a quite different sampling method. They were implemented in the Arvand family of planners, which have been successful in classical planning with its discrete state spaces and actions. The work described here develops an MRW approach for domains with continuous state and action spaces, as encountered in motion planning. Several new algorithms based on MRW are introduced, implemented in the Continuous Arvand system, and compared with existing motion planning approaches in the Open Motion Planning Library (OMPL).

1 Introduction

Motion planning refers to breaking down a movement task into discrete motions that satisfy movement constraints. For example, to pick up an object in an environment, the arm of the robot must move to the target location by using its existing actuators, and without colliding with other objects. Motion planning has many applications including robot navigation, manipulation, animating digital characters, automotive assembly and video game design (LaValle 2006).

Among the many approaches to the motion planning problem, sampling based methods have been very popular. A large number of these methods sample randomly from the state space, which is usually called configuration space, or short C-space, in motion planning. The Probabilistic Roadmaps (PRM) (Kavraki et al. 1996) algorithm constructs a roadmap, which connects random milestones, in order to approximate the connectivity of the configuration space. RRT (LaValle and Kuffner 2001) gradually builds a tree that expands effectively in C-space. EST (Hsu, Latombe, and Motwani 1997) attempts to detect the less explored area of the space through the use of a grid imposed on a projection of C-space.

In contrast to sampling from C-space directly, KPIECE (Şucan and Kavraki 2010) is a tree-based planner that explores a continuous space from the given starting point. KPIECE uses a multi-level grid-based discretization for guidance. Given a projection of state space, KPIECE samples cell chains in each iteration when building the exploring tree. The goal of KPIECE is to estimate the coverage of the state space by looking at the coverage of the different cells, and reduce the time used for forward propagation.

Two main criteria for motion planning are feasibility and optimality of plans. The motion planners mentioned above all return the first feasible plan they find. In contrast, planners such as RRT* keep improving their best plan over time, and some are proven to be asymptotically optimal (Karaman and Frazzoli 2011).

Monte Carlo Random Walks (MRW) are the basis for a successful family of algorithms for classical deterministic planning with discrete states and actions (Nakhost and Müller 2009; Nakhost, Hoffmann, and Müller 2012; Nakhost and Müller 2013). The method uses random exploration of the local neighbourhood of a search state. Different MRW variants have been implemented in the Arvand planning systems. The current work applies MRW to continuous planning, using a local sampling forward search framework which, like KPIECE, does not require sampling globally from C-space.

Component	Classical planning	Motion planning
State space	discrete	continuous
Goal checker	deterministic	approximate
Action execution	instant	gradual
Random walk	sample action $\rightarrow$ new state	sample state $\rightarrow$ new motion
Heuristic	Instance-specific, e.g. Fast Forward	C-space-specific, e.g. geometric distance

Table 1: Main differences between using MRW in classical and motion planning.

The high-level view of MRW for continuous planning is similar to classical planning: Random walks are used to explore the neighbourhood of a state and to escape from local minima. A heuristic function which estimates goal distance is used to evaluate sampled states. The main differences between MRW for classical and continuous planning lie in the mechanisms for action selection and action execution within the random walks. In classical planning, for each state s in a random walk, the successor state s' is found by randomly sampling and executing a legal action in s. In contrast, in continuous planning random actions are not generated di-

rectly. Instead, a nearby successor state s' is sampled locally from the state space, and the motion planner is invoked to try to generate a valid motion from s to s'. In classical planning, actions take effect instantly. The solution to a planning problem is simply an action sequence that achieves a goal condition. In continuous planning, each motion action takes time to complete. A solution is a sequence of valid, collision-free motions that get "close enough" to a goal. Table 1 summarizes some main differences of applying MRW to classical and motion planning.

The remainder of this paper is organized as follows: Section 2 describes the main ideas of MRW planning and their application to continuous planning, including different restart strategies and variants for bidirectional search and continuous plan improvement. Section 3 describes the implementation of MRW planning algorithms in the Continuous Arvand system. Section 4 evaluates the performance of the new planners on planning benchmarks from OMPL (Şucan, Moll, and Kavraki 2012). Section 5 is dedicated to concluding remarks and some potential directions for future work.

2 Applying MRW to Continuous Planning

Arvand (Nakhost and Müller 2009; Nakhost, Hoffmann, and Müller 2012; Nakhost and Müller 2013) is a successful family of stochastic planners in classical planning. These planners use Monte Carlo random walks to explore the neighbourhood of a search state. In this work, a similar approach is developed for continuous planning, and implemented in the Continuous Arvand system.

Monte Carlo Random Walk Planning

A MRW algorithm uses the following key ingredients:

- A *heuristic function* h to evaluate the goal distance for the endpoints of random walks. Strong heuristics lead to better performance.
- A *global restart strategy* is used to escape from local minima and plateaus.
- A *local restart strategy* is used for exploration.

In MRW, given a current state s, a number of random walks sample a relatively large set of states S in the neighbourhood of s: the endpoints of each walk. All states in S are evaluated by the heuristic function h. Finally, a new state $s \in S$ with minimum h-value is selected as the next current state, concluding one *search step*, and the process repeats from there. The length of each random walk is decided by the local restart strategy, and could be fixed or variable. Different choices will be discussed below. If the best observed h-value does not improve after a number of search steps, as controlled by the global restart strategy, the search will restart. A good global restart strategy can quickly escape from local minima, and recover from areas of the state space where the heuristic evaluation is poor. The MRW approach does not rely on any assumptions about local properties of the search space or heuristic function. It locally explores the state space before it commits to an action sequence that leads to the best explored state.

MRW Algorithm

Algorithm 1, slightly adapted from (Nakhost and Müller 2009), shows an outline of MRW planning. This high-level outline is nearly identical for classical and for continuous planning. The only change is that a goal condition G is replaced by a goal region G.

The algorithm uses a forward-chaining search in the state space of the problem to find a solution. The chain of states leads from initial state s_0 to goal state s_n. Each transition $s_j \rightarrow s_{j+1}$ is generated by MRW exploring the neighbourhood of s_j. If the best h-value does not improve after a given number of search episodes, MRW simply restarts from s_0.

Algorithm 1 Monte Carlo Random Walk Planning

Input Initial State s_0, goal region G
Output A solution plan

```
s ← s_0
h_min ← h(s_0)
counter ← 0
while s does not satisfy G do
  if counter > MAX_EPISODES then
    s ← s_0 {restart from initial state}
    counter ← 0
  end if
  s ← randomWalk(s, G)
  if h(s) < h_min then
    h_min ← h(s)
    counter ← 0
  else
    counter ← counter + 1
  end if
end while
return the plan reaching the state s
```

Pure Random Walks

The main motivation for MRW planning is to better explore the local neighbourhood, compared to the greedy search algorithms which have been the standard in classical planning. The simplest MRW approach uses a fixed number of pure random walks to sample the neighborhood of a state s. Algorithm 2 shows a pure random walk method similar to the one in (Nakhost and Müller 2009), but adapted to the case of continuous planning. In classical planning, a random legal action is sampled given a current state s' in a random walk, and then applied to reach the next state s''. For continuous planning, instead of an action, the next state is sampled from a region of the state space near s'. Before s'' can succeed s' as the current state, a check is performed to make sure there is a valid motion from s' to s''. A random walk stops either when a goal state is directly reachable, or when the number of consecutive motions reaches a bound *LENGTH_WALK*. The end state of each random walk is evaluated by the heuristic h. The algorithm terminates when either a goal state is reached, or *NUM_WALK* walks have been completed. The function returns the state s_{min} with minimum h-value among all reached endpoints, and the state

sequence leading to it. If no improvement was found, the algorithm simply returns s.

The chosen limits on the length and number of random walks have a huge impact on the performance of this algorithm. Good choices depend on the planning problem. While they are constant in the basic algorithm shown here, the next subsection discusses different adaptive global and local restart strategies, which are used by Arvand and can be applied in continuous planning as well.

Algorithm 2 Pure Random Walks.

Input current state s, goal region G and state space S
Output s_{min}

$h_{min} \leftarrow \infty$
$s_{min} \leftarrow NULL$
$g \leftarrow$ sampleFromGoalRegion(G)
for $i \leftarrow 1$ to *NUM_WALK* **do**
  $s' \leftarrow s$
  for $j \leftarrow 1$ to *LENGTH_WALK* **do**
    if validMotion(s', g) **then**
      return g
    end if
    repeat
      $s'' \leftarrow$ uniformlySampleFromNear(s', S)
    until validMotion(s', s'')
    $s' \leftarrow s''$
  end for
  if $h(s') < h_{min}$ **then**
    $s_{min} \leftarrow s'$
    $h_{min} \leftarrow h(s')$
  end if
end for
if $s_{min} = NULL$ **then**
  return s
else
  s_{min}
end if

Global and Local Restart Strategy

MRW parameters such as the number and length of random walks, and the maximum number of search episodes, are tedious to set by hand. Nakhost and Müller (2009; 2013) introduce several global and local restart strategies.

Random Walk Length While the simplest approach is to use fixed length random walks, a better strategy in classical planning uses an *initial length bound*, and successively increases it if the best seen h-value does not improve quickly enough. If the algorithm encounters better states frequently enough, the length bound remains the same. A third strategy uses a *local restarting rate* to terminate a random walk with a fixed probability r_l after each motion. In this case, the length of walks is geometrically distributed with mean $1/r_l$.

Number of Random Walks The first version of Arvand used a fixed number of random walks in each search step, then progressed greedily to the best evaluated endpoint. This approach was later replaced by a number of adaptive methods (Nakhost and Müller 2009; 2013). A simple strategy followed here is to have only one random walk in a local search (Nakhost, Hoffmann, and Müller 2012), which is faster than choosing from among several walks, at the cost of solution quality.

Number of Search Episodes and Global Restarting The simplest global restart strategy restarts from initial state s_0 whenever the h-value fails to improve for a fixed number t_g of random walks. An *adaptive global restart* (AGR) algorithm is described in (Nakhost and Müller 2013).

Path Pool

Most versions of Arvand require very little memory. A *path pool* can store a number of random walks and utilize them for improving later searches (Nakhost, Hoffmann, and Müller 2012). The techniques of *On-Path Search Continuation* (OPSC) and *Smart Restarts* (SR) are based on a fixed-capacity pool which stores the most promising episodes encountered so far. OPSC randomly picks a state along the existing path to start a new search episode, instead of always starting from an endpoint. SR is used for global restart: instead of always restarting from s_0, the search restarts from a random state on a random path in the pool.

This current work only uses the path pool idea and pursued a different approach: to start a new search episode, a path p from the pool is either selected with the minimum h-value or randomly picked with a distribution; then a fixed fraction of the pool contents is replaced by newly generated random walks which extend p. Algorithm 3 shows details. The algorithm begins with an empty pool at each global (re-)start. A fixed number n, for example 10% of the pool size, is chosen for addition/replacement. n random walks are performed from start state s_0 and stored in the pool. During the search after (re-)start, one path in the pool is selected and expanded by local exploration to generate n new paths. If the pool is full, n randomly selected existing paths are replaced by new paths. Each path in the pool is a state sequence from s_0 to an endpoint s_j. If a solution is found during expansion, the plan is returned immediately.

Algorithm 3 Expand

Input current state s, goal state g, existing path p, number of new paths n, pool P
Output n new paths added to P, returns whether a solution was found
for n iterations **do**
  new_walk $\leftarrow randomWalk(s, g)$
  new_path $\leftarrow p+$ new_walk
  store(P, new_path)
  if solution found **then**
    return *true*
  end if
end for
return *false*

Bidirectional Arvand

Motion planners such as RRT and KPIECE have bidirectional variants with good performance. Bidirectional Arvand uses similar approach to solve planning problems. It maintains both a forward and a backward path pool. Explorations start from both the start state s_0 and a goal state g_0, and try to connect two search frontiers. For each pair of paths (p_f, p_b) in the two pools, the heuristic distance of their endpoints is stored. If the size of each pool is m, the time complexity of replacing n paths in the pool in each episode and updating the heuristic values is $O(nm)$.

Algorithm 4 shows the outline of bidirectional Arvand. In each search episode, search starts from the endpoint of one chosen path, treats the endpoint of the other chosen path as the search goal, and tries to connect them. In the code, $h(fPool, bPool) = \min_{f \in fPool, b \in bPool} h(f, b)$.

Algorithm 4 Bidirectional Arvand

Input current state s_0, goal state g_0, number of new paths n
Output A solution path

$h_{min} \leftarrow \infty$
$init \leftarrow$ *true*
repeat
  if $counter >$ *MAX_EPISODES* or $init$ **then**
    $counter \leftarrow 0$
    fPool,*bPool* $\leftarrow \emptyset$
    $p \leftarrow NULL$
    expand(s_0, g_0, p, n, *fPool*)
    $s \leftarrow$ closest endpoint towards g_0 in *fPool*
    expand(s, s_0, p, n, *bPool*)
    $current \leftarrow fPool$
    $init \leftarrow$ *false*
  end if
  reserve(n, $current$) {reserve room for n new paths}
  $s, g \leftarrow \text{argmin}_{f \in fPool, b \in bPool} h(f, b)$
  $p \leftarrow$ complete path towards s
  expand(s, g, p, n, $current$) {try to connect two paths}
  if h(*fPool*, *bPool*) $< h_{min}$ **then**
    $h_{min} \leftarrow h(fPool, bPool)$
    $counter \leftarrow 0$
  else
    $counter \leftarrow counter + 1$
  end if
  switch forward and backward search direction
until a solution is found
return solution path

Improving Plan Quality

The algorithms described above stop immediately after a solution is found. Arvand*, shown in Algorithm 5, is an optimizing version of Continuous Arvand, which keeps restarting even after the first valid plan is found. Arvand* uses post-processing techniques, such as shortcutting and smoothing, to simplify each newly found solution. The shortest solution after postprocessing is returned.

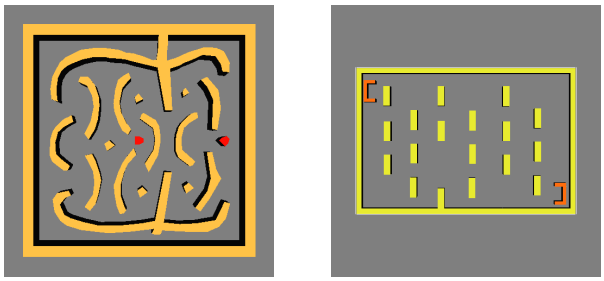
Algorithm 5 Arvand*

Input current state s_0, goal region G
Output A solution path with shortest length

$sol_{min} \leftarrow NULL$
while keep_going() **do**
  $sol \leftarrow$ monteCarloRandomWalk(s_0, G)
  $sol \leftarrow$ simplify(sol)
  if $sol_{min} = NULL$ **or** length(sol) $<$ length(sol_{min}) **then**
    $sol_{min} \leftarrow sol$
  end if
end while
return sol_{min}

3 Implementation - the Continuous Arvand System

Continuous Arvand implements a framework for MRW motion planning, and several different planners. The program is built on top of OMPL, the Open Motion Planning Library (Şucan, Moll, and Kavraki 2012). OMPL provides implementations of all motion planning primitives such as distance heuristics, collision detection, and random state sampling. The heuristic in Continuous Arvand is the distance function provided by OMPL, which differs depending on the type of state space. For instance, for state space $SO(3, \mathbb{R})$ the distance is the angle between quaternions, while $\mathbb{R}^3$ uses euclidean distance. The *simplify(path)* post-processing function provided by OMPL is used in all experiments to shorten the solutions.



(a) Maze (b) Barriers

(c) Abstract (d) Apartment

Figure 1: Planning scenarios

Six motion planners were implemented: Arvand_fixed and Arvand_extend are based on the techniques introduced

in (Nakhost and Müller 2009), while Arvand2 and Arvand2_AGR use ideas from (Nakhost and Müller 2013). BArvand and Arvand* are the bidirectional and optimizing variants of Arvand described in Section 2.

Arvand_fixed is the simplest implementation and uses constant parameters for global restart rate t_g, and number and length of random walks. In the experiments below, $t_g = 20$, *NUM_WALK* $= 20$, and *LENGTH_WALK* is tuned to find the best setting for each planning scenario, in the range from 10 to 800. Tuning these parameters is inconvenient and time consuming. The other versions of Arvand try to automatically adapt the setting for these variables. For Arvand_extend, *NUM_WALK* $= 800$, and *LENGTH_WALK* $= 10$ initially, and is multiplied by a factor of 2 whenever the h-value does not improve over 100 walks. In Arvand2, *NUM_WALK* $= 1$ and a local restarting rate of $r_l = 0.01$ is used to control the random walk length. Arvand2_AGR is similar to Arvand2, but adds adaptive global restarts. BArvand is the bidirectional version of Arvand. In experiments, the size of the forward and backward pools is 100 each, and the setting of other parameters is as in Arvand_fixed. Arvand* uses the same settings as Arvand2_AGR, but keeps running to improve solutions until a given time limit is reached.

4 Experiments

In this section, the five planners Arvand_fixed, Arvand_extend, Arvand2, Arvand2_AGR and BArvand are compared with a selection of the best-performing planners available in OMPL: RRT (LaValle and Kuffner 2001), KPIECE (Şucan and Kavraki 2010), EST (Hsu, Latombe, and Motwani 1997), PDST (Ladd and Kavraki 2005), and PRM (Kavraki et al. 1996). Arvand* is tested against RRT* (Karaman and Frazzoli 2011), which is an asymptotically-optimal incremental sampling-based motion planning algorithm.

Experiments used 13 built-in benchmark scenarios from OMPL: Maze, Barriers, Abstract, Apartment, BugTrap, RandomPolygons, UniqueSolutionMaze, Cubicles, Alpha, Easy, Home, Pipedream_ring and Spirelli. These scenarios are chosen as they can be solved by most available planners in reasonable time (less than 10 minutes). Four of them are shown in Figure 1. We grouped these scenarios into four categories: easy problems (Maze, BugTrap, RandomPolygons, Easy), intermediate problems (Alpha, Barriers, Apartment), intermediate problems with long detour (UniqueSolutionMaze, Cubicles, Pipedream_ring, Abstract) and hard problems (Home, Spirelli). The configuration space used in these problems is either $SE(2)$ or $SE(3)$. We used the recommended time limit provided in OMPL for each scenario in our experiments.

All experiments were run on a machine with 8-core CPU Intel Xeon E5420 @ 2.5GHz and 8GB memory. Results for each planner are averaged over 20 runs per scenario. The metrics of memory use (MB), path length, simplified path length, planning time and simplification time (in seconds) are considered.

Tables 3-16 show the benchmark results. For the metric of memory use, almost all Arvand versions always use less memory than all the other planners. One exception is the BArvand version in scenario Cubicles, which used more memory to maintain two path pools as this scenario has long detours, and BArvand needs much longer paths to reach the goal.

Considering the path length, Arvand2 and Arvand2_AGR always output solutions with huge path lengths. The reason is that these two Arvand versions do not run multiple random walks and choose the best one. Therefore they run faster but produce much longer paths. However, after post-processing, the simplified path length is good enough to compete with other planners. In scenarios Maze, RandomPolygons, Apartment and Easy, Arvand_fixed and Arvand_extend are comparable to other planners on original path length. In scenario Cubicles, these two planners are worse by a factor of 3 to 8. BArvand usually does not provide a competitive initial path length, but it performs very well after simplification. For instance, BArvand outperforms all other planners in scenarios Alpha and Barriers.

The total time in the experiments consists of planning time plus simplification time. The simplification time is insignificant: it is usually below 0.1s for all planners, and never reached 0.5s in any of the experiments. Therefore, only the total time is shown in the tables.

Among all Arvand versions, Arvand_fixed and Arvand_extend are slower because they run many random walks in one episode. This causes them to time out in scenarios UniqueSolutionMaze, Home and Spirelli. Arvand_fixed times out in more scenarios: Cubicles, Alpha, and Pipedream_ring. Arvand2 and Arvand2_AGR are competitive in terms of planning time for the easy planning problems Maze and BugTrap. They also do well in the intermediate problems Cubicles, Pipedream_ring and Apartment. BArvand performs well in most scenarios. It is the best planner in scenario Apartment, always takes a reasonable amount of time when comparing among all Arvand versions, and produces competitive short solutions.

For intermediate problems with long detours, almost all Arvand results are poor. The reason is that Arvand uses a heuristic to guide the exploration, and a detour requires the exploration to go multiple steps against the heuristic. Since Arvand2 and Arvand2_AGR only run one random walk per episode, the planning time is not bad. However, Arvand versions Arvand_fixed and Arvand_extend choose the heuristically best random walk among several walks. They have little chance to go against the heuristic for several successive steps, and need much more time to find a solution, or even time out. The only competitive results on these planning scenarios is for BArvand in scenario Abstract.

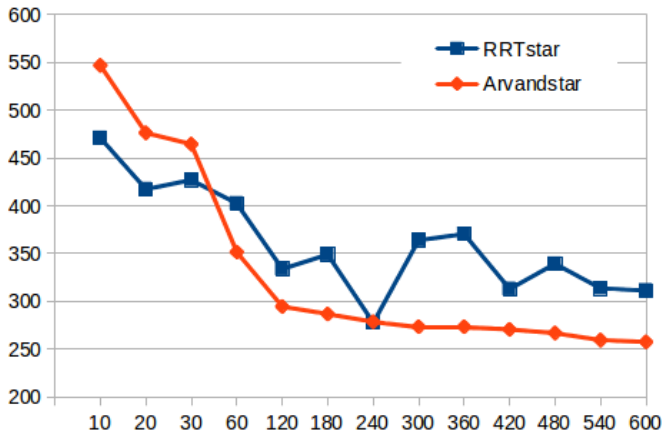


Figure 2: Plan improvement over time for Arvand* and RRT*. Average over 10 runs.

The performance of the optimizing planners RRT* and Arvand* within the recommended time limit is shown in Table 16. On the main metric of simplified path length, RRT* is always better than Arvand*, and Arvand* comes close to the performance of RRT* only in scenarios Alpha and Easy.

The picture can change with longer time limits. Figure 2 compares the plan improvement over time for the two planners in scenario Alpha. While the initial solution found by Arvand* is of rather poor quality, its path length decreases rapidly over time and becomes better than RRT* in this example. At the current time, this is an isolated positive result and it is not clear whether it generalizes to other scenarios.

For generating Figure 2, since the intermediate paths when RRT* is optimizing its plan are not accessible, RRT* is run separately for different time limits. Each data point is averaged over 10 runs.

As a final example, Table 2 shows the importance of choosing the right parameters for MRW with fixed settings. The example is from scenario Barriers, as solved by Arvand_fixed with different parameter settings. For setting 1, the length of random walks is 20, the number of walks per episode is 20, and the maximum number of episodes is 10; for setting 2, the length of random walks is 1000, the number of walk per episode is 50, and the maximum number of episodes is 100. In this planning scenario, setting 1 has better performance.

	Solution Path	Planning Time
Setting 1	1205.7	8.7s
Setting 2	2063.8	24.1s

Table 2: Influence of parameter setting on performance.

5 Conclusions and Future Work

The algorithms developed in this paper apply the Monte Carlo Random Walk method to motion planning. Global and local restart strategies in this method have huge impact on performance. Our work is still preliminary, but the results are already interesting. Continuous Arvand works well for problems that do not require long detours for which the distance heuristic is misleading. The algorithms use much less memory than other planners, which makes them attractive for embedded applications with limited resources.

Portfolio planning (Gomes and Selman 2001) combines several algorithms into a portfolio and runs them in sequence or in parallel. This is a very successful approach in classical planning. The *ArvandHerd* system, winner of the parallel satisficing track of the 2011 and 2014 International Planning Competitions, is such a portfolio which combines (classical) Arvand with another state of the art planner, LAMA (Valenzano et al. 2012; 2014). Our results indicate that adding Continuous Arvand to a motion planning portfolio will very likely strengthen its performance.

The current versions of Continuous Arvand do not work well on planning problems with long forced detours that go against the heuristic. Improving the performance on these kinds of problems is the most important task for future work. Some existing MRW techniques from classical planning, such as On-Path Search Continuation (OPSC) and Smart Restarts (SR) (Nakhost, Hoffmann, and Müller 2012), are not yet used in the Continuous Arvand implementation. *Adaptive local restarting* (Nakhost and Müller 2013) is a technique used to estimate the best parameter for local restarting. In addition of only evaluating the endpoint of a random walk, Arvand could benefit from the heuristic evaluation of the intermediate states along the walk (Nakhost, Hoffmann, and Müller 2012). Finally, there is work to do to research the many different ways of using memory, such as different strategies for using path pools, adding a tree as in RRT, or a UCT-like approach.

References

Gomes, C. P., and Selman, B. 2001. Algorithm portfolios. *Artificial Intelligence* 126(1):43–62.

Hsu, D.; Latombe, J.-C.; and Motwani, R. 1997. Path planning in expansive configuration spaces. In *IEEE Robotics and Automation*, volume 3, 2719–2726. IEEE.

Karaman, S., and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research* 30(7):846–894.

Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; and Overmars, M. H. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Robotics and Automation* 12(4):566–580.

Ladd, A. M., and Kavraki, L. E. 2005. Motion planning in the presence of drift, underactuation and discrete system changes. In *Robotics: Science and Systems*, 233–240.

LaValle, S. M., and Kuffner, J. J. 2001. Randomized kinodynamic planning. *The International Journal of Robotics Research* 20(5):378–400.

LaValle, S. M. 2006. *Planning algorithms*. Cambridge University Press.

Nakhost, H., and Müller, M. 2009. Monte-Carlo exploration for deterministic planning. In *IJCAI*, volume 9, 1766–1771.

Nakhost, H., and Müller, M. 2013. Towards a second generation random walk planner: an experimental exploration. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 2336–2342. AAAI Press.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2012. Resource-constrained planning: A Monte Carlo random walk approach. In McCluskey, L.; Williams, B.; Reinaldo Silva, J.; and Bonet, B., eds., *ICAPS*, 181–189. AAAI Press.

Şucan, I. A., and Kavraki, L. E. 2010. Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII*. Springer. 449–464.

Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine* 19(4):72–82. `http://ompl.kavrakilab.org`.

Valenzano, R.; Nakhost, H.; Müller, M.; Sturtevant, N.; and Schaeffer, J. 2012. ArvandHerd: Parallel planning with a portfolio. In De Raedt, L., ed., *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, 786–791. IOS Press.

Valenzano, R.; Nakhost, H.; Müller, M.; Schaeffer, J.; and Sturtevant, N. 2014. Arvandherd 2014. In Vallati, M.; Chrpa, L.; and McCluskey, T., eds., *The Eighth International Planning Competition*, 1–5. University of Huddersfield.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	1.26	285.35	149.64	**0.49**
EST	2.27	189.72	118.11	1.58
PDST	16.64	195.17	117.50	0.59
RRT	0.89	152.16	125.07	0.59
PRM	1.64	134.95	116.70	1.10
Arvand_fixed	0.36	**120.68**	**88.72**	5.39
Arvand_extend	0.47	187.00	105.30	7.16
Arvand2	0.98	4,630.43	139.96	1.74
Arvand2_AGR	2.04	10,739.10	153.31	1.75
BArvand	**0.52**	364.63	108.33	0.70

Table 3: Scenario Maze, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	2.71	3,410.69	1,738.58	**0.61**
EST	6.11	2,130.88	1,544.93	2.37
PDST	29.83	3,058.34	2,078.17	0.91
RRT	243.23	1,723.06	1,519.71	1.20
Arvand_fixed	**0.40**	**1,468.52**	1,075.18	15.93
Arvand_extend	0.70	4,253.87	1,416.57	28.52
Arvand2	1.65	35,791.87	1,623.66	4.10
Arvand2_AGR	3.14	111,872.24	1,714.82	3.49
BArvand	4.48	7,690.83	**864.12**	5.36

Table 4: Scenario Barriers, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	19.87	3,180.60	1,070.92	14.53
EST	17.18	1,567.90	855.60	16.59
PDST	199.83	2,764.19	1,228.19	14.71
RRT	153.39	1,256.63	949.70	29.57
PRM	160.16	**805.39**	706.16	258.84
Arvand_fixed	0.88	1,388.88	647.05	166.39
Arvand_extend	2.05	10,127.76	887.14	96.83
Arvand2	2.46	23,285.37	786.86	133.29
Arvand2_AGR	21.16	904,221.82	998.34	36.69
BArvand	**0.59**	1,395.73	**589.79**	**11.02**

Table 5: Scenario Abstract, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	10.43	1,133.78	452.93	17.25
EST	3.79	716.99	444.24	12.95
PDST	92.36	920.69	437.46	19.85
RRT	3.35	523.93	425.96	**8.30**
PRM	50.04	**485.74**	**409.94**	102.30
Arvand_fixed	**0.36**	529.79	428.66	38.09
Arvand_extend	0.52	859.32	437.66	96.77
Arvand2	0.63	2,859.56	431.09	9.17
Arvand2_AGR	0.72	4,233.13	458.73	12.35
BArvand	2.41	2,436.83	445.05	10.78

Table 6: Scenario Apartment, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	4.62	446.65	170.58	0.33
EST	2.73	286.95	162.31	0.41
PDST	17.22	303.57	175.14	**0.26**
RRT	2.50	254.89	177.31	0.33
PRM	9.73	**163.42**	**140.59**	3.42
Arvand_fixed				time out
Arvand_extend	**0.79**	867.29	165.84	4.94
Arvand2	1.91	10,514.92	167.31	0.73
Arvand2_AGR	2.78	16,058.34	160.42	0.77
BArvand	3.74	1,210.20	162.08	1.28

Table 7: Scenario BugTrap, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	0.94	310.00	133.65	0.11
EST	0.75	228.59	130.62	0.31
PDST	1.60	196.65	133.00	0.06
RRT	0.57	155.47	130.49	**0.05**
PRM	0.55	**149.82**	133.57	0.10
Arvand_fixed	**0.34**	189.09	**123.66**	4.20
Arvand_extend	0.43	290.43	127.64	2.26
Arvand2	0.79	2,192.15	137.73	0.18
Arvand2_AGR	0.86	2,137.46	132.41	0.14
BArvand	0.63	404.06	124.78	0.21

Table 8: Scenario RandomPolygons, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	3.29	663.00	393.32	**1.33**
EST	18.46	491.09	367.62	3.71
PDST	202.19	483.52	337.94	5.63
RRT	2.99	399.27	344.27	2.77
PRM	3.09	**340.60**	**328.99**	2.31
Arvand_fixed				time out
Arvand_extend				time out
Arvand2	**2.55**	8,134.98	346.50	6.66
Arvand2_AGR	6.13	42,158.52	341.17	7.73
BArvand	7.81	1,092.37	352.22	4.63

Table 9: Scenario UniqueSolutionMaze, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	6.25	6,592.95	2,606.57	1.14
EST	10.96	3,888.98	2,450.79	6.81
PDST	55.02	4,805.96	2,555.92	2.41
RRT	**0.91**	3,242.16	2,587.69	**0.60**
PRM	12.34	**2,512.20**	**2,292.76**	5.33
Arvand_fixed				time out
Arvand_extend	1.14	20,054.38	2,481.16	46.52
Arvand2	2.16	61,197.39	2,442.39	1.95
Arvand2_AGR	3.01	85,836.54	2,423.80	1.81
BArvand	48.50	34,533.70	2,454.40	7.59

Table 10: Scenario Cubicles, time limit $= 60s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	**0.61**	1,614.49	637.16	2.79
EST	0.78	**938.13**	550.47	4.12
PDST	19.65	1,569.88	576.56	**2.04**
RRT	17.15	949.06	583.25	4.12
PRM				time out
Arvand_fixed				time out
Arvand_extend	3.19	2,000.73	496.25	25.07
Arvand2	3.45	17,592.26	563.81	8.75
Arvand2_AGR	3.67	19,263.26	622.31	6.12
BArvand	3.45	6,947.17	**481.80**	18.85

Table 11: Scenario Alpha, time limit $= 60s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	2.41	1,140.28	234.30	0.63
EST	2.94	593.66	236.81	0.64
PDST	6.05	595.02	233.84	0.17
RRT	8.41	**347.84**	209.43	**0.15**
PRM	8.63	508.85	250.27	0.46
Arvand_fixed	**0.39**	369.39	205.96	0.56
Arvand_extend	0.43	594.30	**204.28**	0.73
Arvand2	0.92	33,561.37	216.11	1.19
Arvand2_AGR	2.49	126,037.08	208.22	1.10
BArvand	4.48	8,522.32	236.58	0.90

Table 12: Scenario Easy, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE				time out
EST	**303.22**	4,127.54	2,364.86	31.00
PDST	3,229.55	3,955.31	1,908.11	128.53
RRT	3,581.81	2,585.87	2,141.99	**26.56**
PRM	3,584.59	**1,825.84**	**1,637.01**	59.76
Arvand_fixed				time out
Arvand_extend				time out
Arvand2				time out
Arvand2_AGR				time out
BArvand				time out

Table 13: Scenario Home, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	21.54	242.65	98.14	62.14
EST	19.45	161.73	89.26	2.52
PDST	22.73	241.60	108.95	**1.03**
RRT	23.45	157.41	105.98	2.02
PRM	223.71	**128.58**	**86.23**	90.12
Arvand_fixed				time out
Arvand_extend	**2.91**	385.03	131.80	75.86
Arvand2	3.18	1,564.64	116.08	1.53
Arvand2_AGR	3.30	1,862.63	104.08	1.42
BArvand	3.97	957.35	133.25	14.39

Table 14: Scenario Pipedream_ring, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE				time out
EST				time out
PDST				time out
RRT	2,229.50	**203.22**	**166.05**	102.99
PRM				time out
Arvand_fixed				time out
Arvand_extend				time out
Arvand2				time out
Arvand2_AGR	1.41	3,043.27	222.48	**78.68**
BArvand	**0.51**	480.48	166.80	157.13

Table 15: Scenario Spirelli, time limit $= 180s$.

Problem	Planner	Memory	Path length	Simplified path length
Alpha	RRTstar	324.36	358.81	328.52
	Arvandstar	37.63	5,890.36	358.97
Barriers	RRTstar	510.36	822.51	806.70
	Arvandstar	120.97	51,703.69	1,294.87
Easy	RRTstar	266.48	208.19	203.44
	Arvandstar	55.46	62,327.05	211.09
Pipedream_ring	RRTstar	471.33	84.77	77.14
	Arvandstar	82.89	375.72	99.38
Spirelli	RRTstar	8.52	118.59	114.89
	Arvandstar	0.70	2,877.63	193.00
Abstract	RRTstar	236.63	600.64	569.85
	Arvandstar	298.18	438,366.05	862.49
Apartment	RRTstar	57.75	404.29	382.32
	Arvandstar	4.42	2,062.75	432.15
BugTrap	RRTstar	29.55	124.66	121.78
	Arvandstar	51.72	7,040.70	163.44
Cubicles	RRTstar	18.38	1,916.82	1,833.96
	Arvandstar	60.56	45,079.29	2,378.78
Maze	RRTstar	9.71	71.51	69.69
	Arvandstar	19.45	829.89	102.78
RandomPolygons	RRTstar	16.51	108.78	106.53
	Arvandstar	37.21	646.64	127.31
UniqueSolutionMaze	RRTstar	11.35	280.85	277.49
	Arvandstar	15.93	25,511.39	345.38

Table 16: Comparing RRT* and Arvand*, with the same time limits as in previous tables.