Motion Planning with Monte Carlo Random Walks

by

Weifeng Chen

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

This thesis applies the Monte Carlo Random Walk method (MRW) to motion planning. We explore different global and local restart strategies to improve the performance. Several new algorithms based on the MRW approach, such as bidirectional Arvand and optimizing planner Arvand*, are introduced and compared with existing motion planning approaches in the Open Motion Planning Library (OMPL). The results of the experiments show that the Arvand planners are competitive against other motion planners on the planning problems provided by OMPL.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Prof. Martin Müller. He is a great researcher and supervisor. His valuable guidance and support helped me in all the time of research and writing of this thesis.

I would also like to thank Prof. Robert Holte, who helped me to present our work at the PlanRob workshop during ICAPS conference when I was not able to go to Israel.

I gratefully acknowledge University of Alberta for financially supporting me.

I thank all my friends in Edmonton who helped me to survive the loneliness and hard time. Special thanks to my friend Ying Xu, who gave me a lot of advices on research and living in Edmonton.

Last, but not least, I thank my parents, Yongzhao Chen and Ting Li, and my dear sister, Jing Chen, for their constant love and support during my whole life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Planning is a branch of artificial intelligence that chooses and organizes strategies or actions by anticipating expected outcomes. The process of planning requires the understanding of applicable actions and their effects on the environment. Planning exists in our everyday activities, such as cooking a meal or playing chess. It is very natural to look for an explicit plan to finish a task efficiently. With the help of powerful computers, we hope they can automatically find a good plan or even optimal plan for us.

1.2 Motion Planning

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 [12].

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) [7] algorithm constructs a roadmap, which connects random milestones, in order to approximate the connectivity of the configuration space. RRT [13] gradually builds a tree that expands effectively in C-space. EST [4] 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 [17] 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 a chain of cells from multiple levels 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 [6].

Motion planning discussed in this thesis only accounts for the geometric and kinematic constraints of the system. It is assumed that any feasible path can be turned into a dynamically feasible trajectory.

1.3 Monte Carlo Random Walk

Monte Carlo Random Walks (MRW) are the basis for a successful family of algorithms for classical deterministic planning with discrete states and actions [15, 14, 16]. The method uses random exploration of the local neighbourhood of a search state. Different MRW variants have been implemented in the Arvand planning systems.

1.4 Contributions of Thesis

This work applies the Monte Carlo Random Walk method to motion planning. Different global and local restart strategies are explored to improve the performance. Several new algorithms based on the MRW approach, such as bidirectional Arvand and optimizing planner Arvand*, are introduced and compared with existing motion planning approaches in the Open Motion Planning Library (OMPL)[18]. Part of this work was presented at the PlanRob workshop during ICAPS 2015 [1].

1.5 Summary of contents

The remainder of this thesis is organized as follow: Chapter 2 introduces the background of classical planning and motion planning and describes the Monte Carlo random walk approach. Chapter 3 describes the application of the MRW method to motion planning and several new algorithms based on MRW. Chapter 4 evaluates the performance of the new planners on planning benchmarks from OMPL [18]. Chapter 5 is dedicated to concluding remarks and some potential directions for future work.

Chapter 2

Background and Related Work

2.1 Classical Planning

The following definition of classical planning is from *Automated planning: theory & practice* [2]:

A restricted state-transition system is a deterministic, static, finite, and fully observable state-transition system with restricted goals and implicit time. Such a system is denoted $\Sigma = (S, A, \gamma)$, where S is a set of states, A is a set of actions, and $\gamma : S \times A \to 2^S$ is a state-transition function. Here, S, A and γ are finite, and $\gamma(s, a)$ indicates a new state when action a is applied to state s.

A planning problem for a restricted state-transition system $\Sigma = (S, A, \gamma)$ is defined as a triple $\mathcal{P} = (\Sigma, s_0, G)$, where s_0 is an initial state and G is a set of goal states. A solution to P is a sequence of actions $(a_1, a_2, ..., a_k)$ corresponding to a sequence of state transitions $(s_0, s_1, ..., s_k)$ such that $s_1 = \gamma(s_0, a_1), ..., s_k = \gamma(s_{k-1}, a_k)$, and s_k is a goal state. The planning problem is to synthesize such a sequence of actions.

Classical planning refers generically to planning for restricted state-transition systems. The most popular way to represent classical planning problems is classical representation. States are represented a sets of logical atoms that are true or false. Actions are represented by planning operators that change the truth values of these atoms.

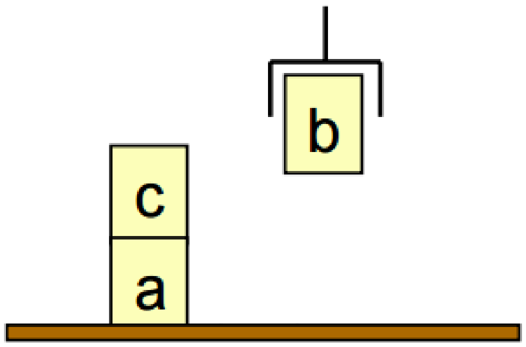


Figure 2.1: Example of classical planning.

Figure 2.1 shows an example about moving blocks on a table. There is one robot hand and three blocks: a, b and c. Using classical representation, here are the predicates:

- ontable(x): block x is on the table
- on(x,y): block x is on block y
- clear(x): block x has nothing on it
- holding(x): the robot hand is holding block x
- handempty: the robot hand is not holding anything

The current state in Figure 2.1 is represented as {ontable(a), on(c,a), clear(c), holding(b)}. An example of an applicable operator is $putdown(b)$:

- Precondition: holding(x)
- Effect: ¬ holding(x), ontable(x), clear(x), handempty

The effect of applying $putdown(b)$ is a transition to the new state: {ontable(a), on(c,a), clear(c), ontable(b), clear(b), handempty}.

2.2 Motion Planning

2.2.1 Introduction

Motion planning is a term used in robotics for the process of breaking down a desired movement task into discrete motions that satisfy movement constraints and possibly optimize some aspect of the movement. A famous motion planning problem is to move a piano through a room with obstacles. Motion planning has many applications, such as robot navigation, manipulation, animating digital characters, video game design and robotic surgery.

Motion planning has the following basic ingredients [12]:

- **State** Planning problems involve a state space that captures all possible situations. The state could, for example, represent the position and orientation of a robot, or the position and velocity of a helicopter. The state space of motion planning is continuous (uncountably infinite) and impossible to represent explicitly.

- **Time** A decision in a motion planning problem must be applied over time. Time might be explicitly modelled, as in a problem such as driving a car as quickly as possible in the presence of obstacles. Time can also be implicit. For example, when moving a piano, the particular speed does not need to be specified in the plan.

- **Actions** A plan generates actions that change the states. In motion planning, actions are applied over time. For most motion planning problems, explicit reference to time is avoided by directly specifying a path through a continuous state space.

- **Initial and goal states** A planning problem usually involves starting in some initial state and trying to arrive at a specified goal state, or any state in a set of goal states.

- **Criterion: Feasibility or Optimality** A feasible plan arrives at a goal state regardless of efficiency, while an optimal plan optimizes some performance objective, such as finding the shortest path.

- **A Plan** In general, a plan imposes a specific strategy or behaviour on a decision maker. It may simply specify a sequence of actions to be taken, or be more complicated, such as a mapping of states to actions.

2.2.2 Representation

Formulating and solving motion planning problems requires defining and manipulating complicated geometric models of a system of bodies in space. There are generally two approaches for geometric modelling: a boundary representation, and a solid representation. Suppose we would like to define a model of a planet. In a boundary representation, an equation of a sphere can represent the planet's surface. Using a solid representation, we describe the set of all points in the sphere.

For the world $\mathcal{W}$, there are two standard choices: 1) a 2D world, in which $\mathcal{W} = \mathbb{R}^2$, and 2) a 3D world, in which $\mathcal{W} = \mathbb{R}^3$. The world generally contains two kinds of entities [12]:

1. **Obstacles:** Portions of the world that are "permanently" occupied, for example, the walls of a building.

2. **Robots:** Movable bodies that are modelled geometrically and are controllable via a motion plan.

Both obstacles and robots can be represented using polygonal and polyhedral models, semi-algebraic models, 3D triangles, or bitmaps. From here on, only the case of a single robot is discussed further.

The state space for motion planning is referred to as the configuration space [10]. The dimension of the configuration space corresponds to the number of degrees of freedom of the robot. Motion planning is viewed as a search in a high-dimensional configuration space that contains implicitly represented obstacles. A motion plan is defined as a continuous path in configuration space.

A configuration describes the pose of the robot, and the configuration space C is the set of all possible configurations. For example, if the robot is a single point moving in a 2-dimensional plane, C is the plane and a configuration can be represented using two parameters (x, y); if the robot is a 2D shape that can translate and rotate, C is the special Euclidean group $SE(2) = R^2 \times SO(2)$, where $SO(n)$ is the group of all rotations about the origin of

n-dimensional Euclidean space under the operation of composition [5]. In this case, a configuration can be represented using 3 parameters (x, y, θ). If the robot is a 3D shape that can translate and rotate, C is the special Euclidean group $SE(3) = R^3 \times SO(3)$, and a configuration requires 6 parameters: (x, y, z) for translation, and Euler angles (α, β, γ) for orientation.

The set of configurations that avoids collision with obstacles is called the free space C_{free}. The complement of C_{free} in C is called the obstacle space C_{obs}. The following gives the definition of a motion planning problem [12]:

Definition:

1. A world $\mathcal{W}$, either $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$.
2. An obstacle region $\mathcal{O} \subset \mathcal{W}$ in the world.
3. A robot $\mathcal{A} \subset \mathcal{W}$.
4. The configuration space $\mathcal{C}$. From this, $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$ are derived.
5. An initial configuration $q_I \in \mathcal{C}_{free}$.
6. A goal configuration $q_G \in \mathcal{C}_{free}$.
7. A complete algorithm must either compute a (continuous) path, $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$, such that $\tau(0) = q_I$ and $\tau(1) = q_G$, or correctly report that such a path does not exist.

2.2.3 Examples of Motion Planning Problems

This section shows several examples of motion planning [12].

A Motion Planning Puzzle

Figure 2.2 shows a problem that requires planning in continuous space. This puzzle was designed to frustrate both humans and motion planning algorithms. It can be solved in a few minutes on a standard personal computer (PC) using the *rapidly exploring tree (RRT)* algorithm in the next subsection.

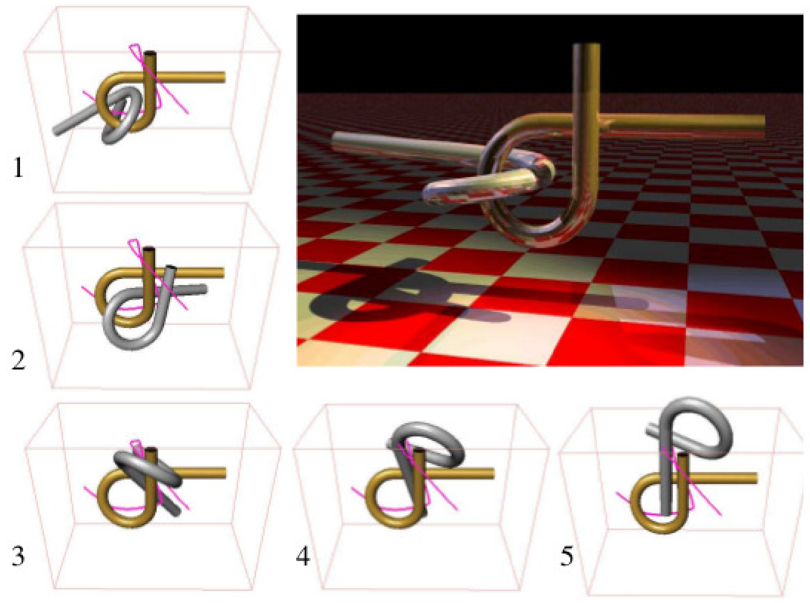


Figure 2.2: The Alpha 1.0 Puzzle, created by Boris Yamrom and posted as a research benchmark by Nancy Amato at Texas A&M University. Solution by James Kuffner [12].

Sealing Cracks in Automotive Assembly

Figure 2.3 shows a simulation of robots performing sealing at the Volvo Cars assembly plant in Torslanda, Sweden. Sealing is the process of using robots to spray a sticky substance along the seams of a car body to prevent dirt and water from entering and causing corrosion. Using motion planning software, engineers need only specify the high-level task of performing the sealing, and the robot motions are computed automatically. This saves enormous time and expense in the manufacturing process.

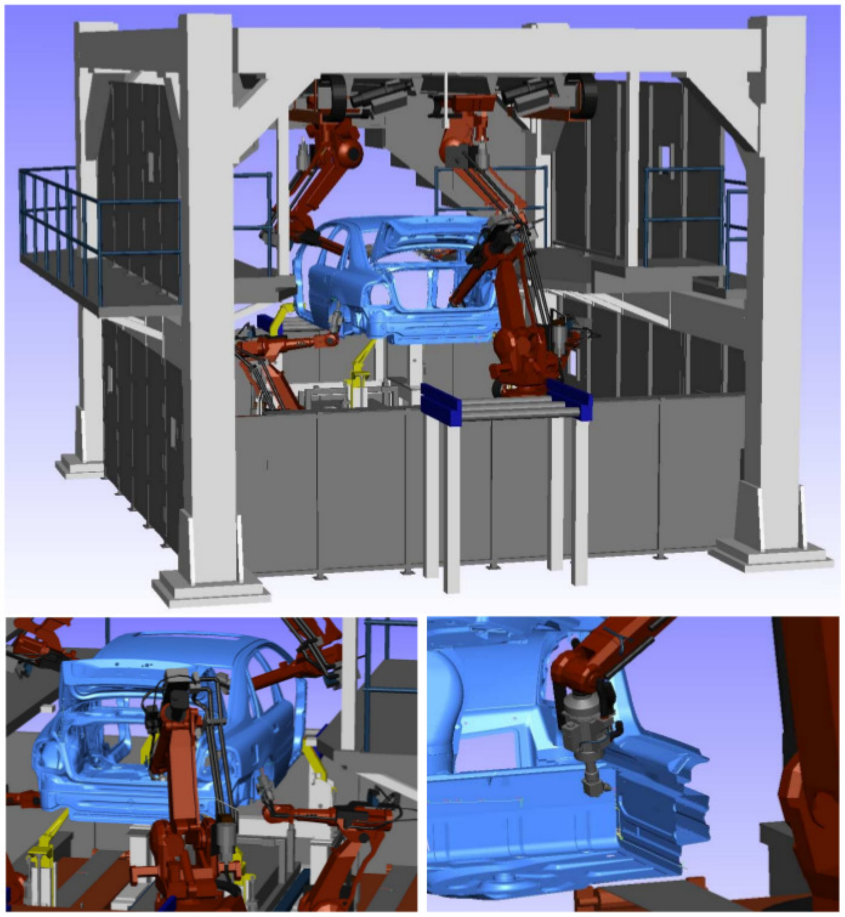
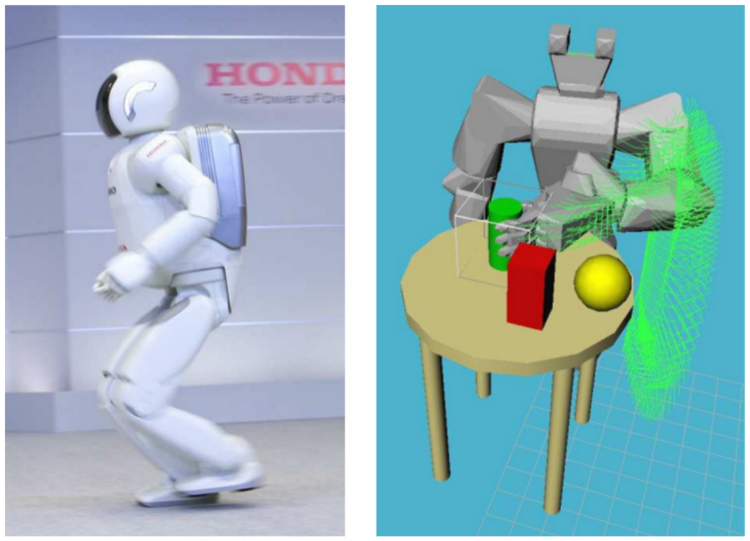


Figure 2.3: An application of motion planning to the sealing process in automotive manufacturing. Planning software developed by the Fraunhofer Chalmers Centre (FCC) is used at the Volvo Cars plant in Sweden [12].

Virtual Humans and Humanoid Robots

Figure 2.4 shows humanoid robots from the Japanese automotive industry.



(a) (b)

Figure 2.4: Humanoid robots from the Japanese automotive industry: (a) The 2011 Asimo robot from Honda can run at 9 km/hr (courtesy of Honda); (b) planning is incorporated with vision in the Toyota humanoid robot so that it plans to grasp objects [12].

Designing Better Drugs

Planning algorithms are even impacting fields as far away from robotics as computational biology. Two major problems in this area are protein folding and drug design. In both cases, scientists attempt to explain behaviours in organisms by the way large organic molecules interact. Such molecules are generally flexible. Drug molecules are small (see Figure 2.5), and proteins usually have thousands of atoms. The docking problem involves determining whether a flexible molecule can insert itself into a protein cavity, as shown in Figure 2.5, while satisfying other constraints, such as maintaining low energy.

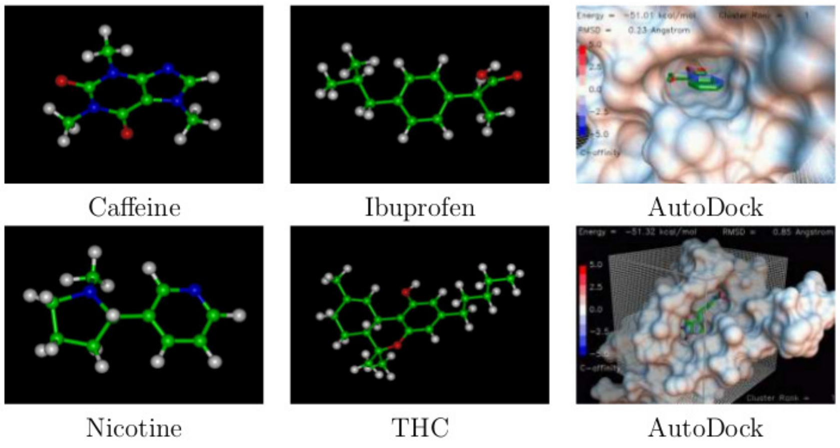


Figure 2.5: On the left, several familiar drugs are pictured as ball-and-stick models (courtesy of the New York University MathMol Library). On the right, 3D models of protein-ligand docking are shown from the AutoDock software package (courtesy of the Scripps Research Institute) [12].

2.2.4 Algorithms for Motion Planning

There are two main methods for solving motion planning problems. One approach is sampling-based, the other one is combinatorial. The following are some of the most popular algorithms in motion planning.

RRT: Rapidly Exploring Random Tree

The rapidly exploring random trees (RRT) algorithm is designed to efficiently search nonconvex, high-dimensional spaces by randomly building a space-filling tree. The tree is constructed incrementally from samples drawn randomly from the search space and is inherently biased to grow towards large unsearched areas of the problem [11].

Algorithm 1 shows the process of growing the tree. Given an initial configuration q_{init} and a maximum incremental distance $\triangle q$, the task is to grow a tree of K vertices. In each iteration, a vertex q_{rand} is sampled randomly from the configuration space, then a new vertex q_{new} is obtained by moving from the nearest existing vertex q_{near} an incremental distance $\triangle q$, in the direction of q_{rand}. If the path from q_{near} to q_{rand} is blocked by an obstacle, q_{new} is the closest vertex towards the obstacle from q_{near}. Finally, a new vertex q_{new} and a new edge from q_{near} to q_{new} is added. Figure 2.6 shows an execution of the algorithm. In the early iterations, RRT quickly reaches the unexplored parts. However, RRT is dense in the limit (with probability one), which means that it gets arbitrarily close to any point in the space [12].

Algorithm 1 RRT

Input Initial configuration q_{init}, number of vertices K, incremental distance $\triangle q$

Output RRT graph G

```
  $G.init(q_{init})$
  for $k = 1$ to $K$ do
    $q_{rand} \leftarrow$ RAND_CONF()
    $q_{near} \leftarrow$ NEAREST_VERTEX($q_{rand}$, $G$)
    $q_{new} \leftarrow$ NEW_CONF($q_{near}$, $q_{rand}$, $\triangle q$)
    $G.add\_vertex(q_{new})$
    $G.add\_edge(q_{near}, q_{new})$
  end for
  return $G$
```

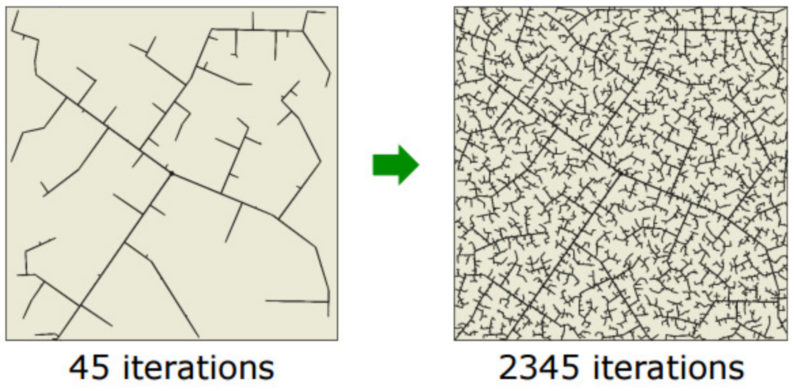


Figure 2.6: Sample RRT execution in the 2D plane without obstacles after 45 and 2345 iterations [12].

PRM: Probabilistic road maps

The basic idea of PRM is to take random samples from the configuration space of the robot, test them for whether they are in free space, and use a local planner to attempt to connect these configurations to other nearby configurations. The initial and goal configurations are added in, and a graph search algorithm is applied to the resulting graph to determine a path between the initial and goal configurations [7].

A probabilistic roadmap planner consists of two phases: a construction and a query phase. In the construction phase, a roadmap is built, approximating the motions that can be made in the environment. First, a random configuration is created. Then, it is connected to some neighbours, typically either the k nearest neighbours or all neighbours less than some predetermined distance, as shown in Figure 2.7. Configurations and connections are added to the graph until the roadmap is dense enough. In the query phase, the initial and goal configuration are connected to the graph, and the path is obtained by a Dijkstra's shortest path query.

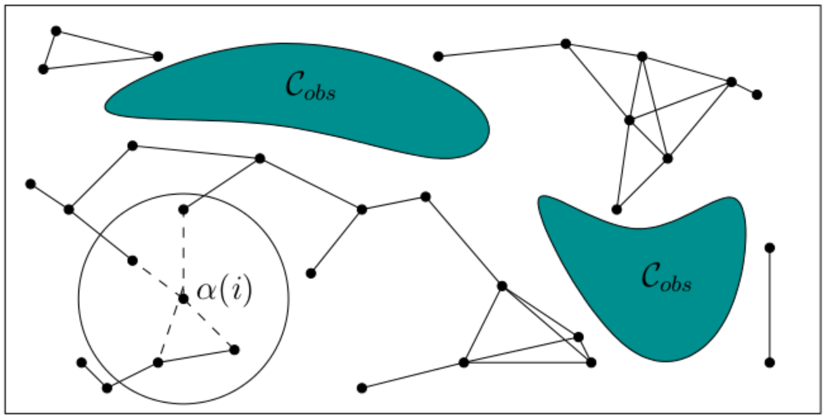


Figure 2.7: The sampling-based roadmap is constructed incrementally by attempting to connect each new sample, $\alpha(i)$, to nearby vertices in the roadmap [12].

KPIECE: Kinodynamic Planning by Interior-Exterior Cell Exploration

KPIECE (Kinodynamic Planning by Interior-Exterior Cell Exploration)[17], a sampling-based motion planning algorithm, is specifically designed for use with physics-based simulation. It can perform motion planning for complex realistic systems, which may need to deal with friction or gravity with other bodies. KPIECE is able to handle high dimensional systems with complex dynamics. It reduces both runtime and memory requirements by making better use of information collected during the planning process. Intuitively, this information is used to decrease the amount of forward propagation the algorithm needs.

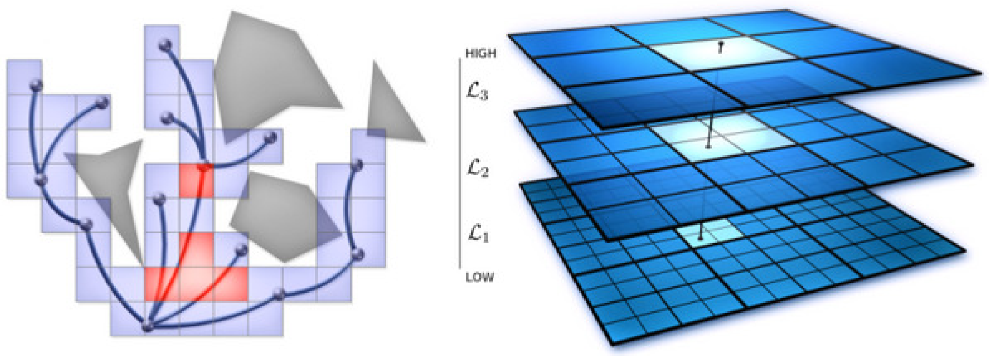


Figure 2.8: An example discretization with three levels. Interior cells in red and exterior cells in blue. [17]

KPIECE iteratively constructs a tree of motions in a projection of the state space. To decide which areas of the state space merit further exploration, multiple levels of discretization are defined for evaluation of the coverage of the state space, as shown in Figure 2.8. Cells on one level all have the same size and they are created when needed, to cover the tree of motions as it grows. For any motion μ, each level contains a cell that μ is part of. Cells are distinguished into interior and exterior:

- Interior cells are ones that have $2n$ neighbours in a n-dimensional space. In the example picture, we have a 2-dimensional space, so 4 neighbours are needed. This is the maximum number of possible neighbours: we do not consider cells on the diagonals to be neighbouring.
- Exterior cells are cells that cover the tree of motions but are not interior.

At each iteration, a chain of cells from different levels is sampled (one cell per level). The sampling of cells starts from the highest level. At each level, KPIECE first decides to expand from an interior or exterior cell, with a bias towards exterior cells. An instantiated cell, either interior or exterior with highest importance, is then deterministically selected. The importance of a cell is affected by the number of times this cell was selected, the number of instantiated neighboring cells and other factors. After the cell in the lowest level is selected, a motion μ in this cell is picked according to a half-normal distribution. A state s along μ is then chosen uniformly at random to continue expanding the tree of motions.

Given the state s from the above sampling process, the next state s' is selected uniformly at random from the state space, or selected by other methods, if available. If the new motion μ_o from s to s' is valid, μ_o is added to the tree motions and the discretization is updated. If μ_o reaches the goal region, the algorithm returns the path to μ_o.

Other Types of Motion Planners

Other types of motion planners include the tree-based planner EST [4] and PDST [9]. EST attempts to detect the less explored area of the space through the use of a grid imposed on a projection of the state space. Using this information, EST continues tree expansion primarily from less explored areas. PDST detects the less explored area through the use of a binary partition of a projection of the state space. Exploration is biased towards larger cells which contain fewer path segments.

Some motion planners, such as RRT and KPIECE, have many variants. One important idea is bidirectional search. RRT-Connect [8] is the bidirectional version of RRT, which builds two rapidly exploring trees rooted at the start and the goal configurations. The trees each explore state space and also advance towards each other through the use of a simple greedy heuristic.

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 [6].

2.2.5 OMPL: Open Motion Planning Library

OMPL (Open Motion Planning Library) [18] contains many state-of-the-art sampling-based motion planning algorithms such as PRM, RRT, EST, PDST, KPIECE, and several variants of these planners. All these planners operate on abstractly defined state spaces. Many commonly used state spaces are already implemented, such as SE(2), SE(3) and $\mathbb{R}^n$.

Components

OMPL provides the components needed for most sampling-based motion planners. The following OMPL classes are analogous to ideas in traditional sampling-based motion planners:

- **StateSampler** The StateSampler class implemented in OMPL provides methods for uniform and Gaussian sampling in the most common state space configurations. Included in the library are methods for sampling in Euclidean spaces, in the space of 2D and 3D rotations, and in any combination thereof with the CompoundStateSampler.

- **NearestNeighbors** This is an abstract class that provides a common interface to the planners for the purpose of performing a nearest neighbor search among samples in the state space.

- **StateValidityChecker** The StateValidityChecker evaluates a state to determine if this configuration collides with an environment obstacle and respects the constraints of the robot.

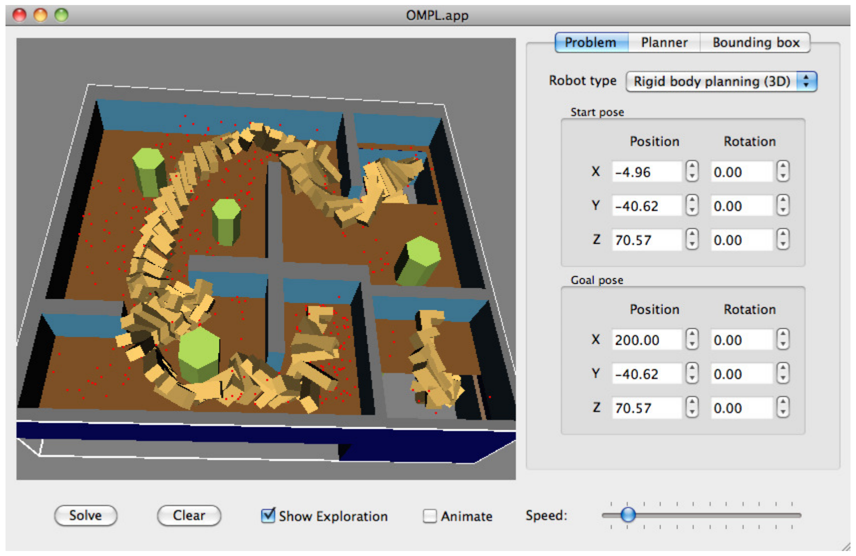
- **MotionValidator** The MotionValidator class (analogous to the local planner) checks whether the motion of the robot between two states is valid. At a high level, the MotionValidator must be able to evaluate whether the motion between two states is collision free and respects all the motion constraints of the robot.

- **OptimizationObjective** Some motion planners optimize objectives that correspond to different cost functions. The OptimizationObjective class provides an abstract interface to the relevant operations with costs that these planners need. The costs could be path length or path clearance.

- **ProblemDefinition** A motion planning query is specified by a ProblemDefinition object. Instances of this class define a start configuration and a goal configuration for the robot, and the optimization objective to meet, if any. The goal can be either a single configuration or a region surrounding a particular state.

Graphical User Interface

OMPL provides a graphical user interface for visualization of the planning environment and solution path, shown in Figure 2.9.



Robot type: Rigid body planning (3D)

Start pose

	Position	Rotation
X	-4.96	0.00
Y	-40.62	0.00
Z	70.57	0.00

Goal pose

	Position	Rotation
X	200.00	0.00
Y	-40.62	0.00
Z	70.57	0.00

[✓ Show Exploration] Animate

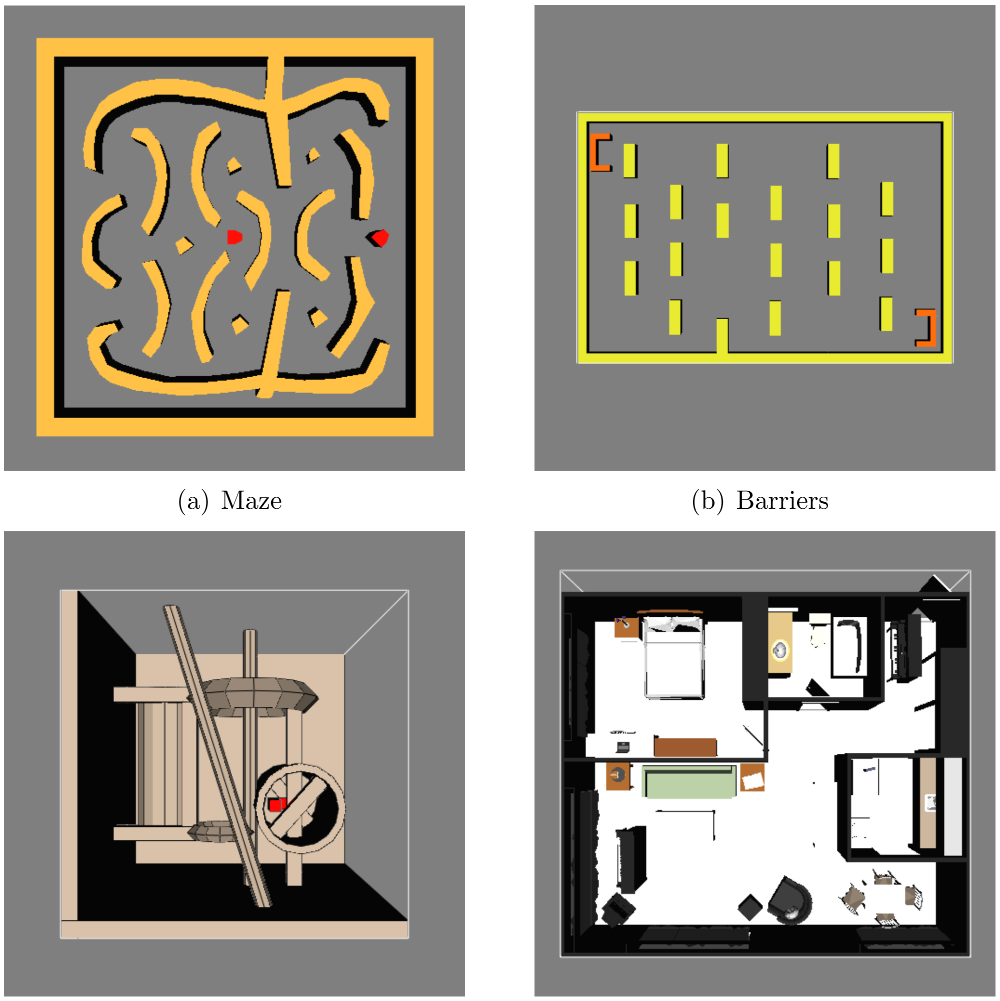
Figure 2.9: Graphical User Interface to OMPL

Benchmark

OMPL provides a Benchmark class that attempts a specific query for a given number of times, and allows the user to try any number of planners. When the benchmarking has finished, the Benchmark instance writes a log file that contains planning information for further analysis. This makes it easy to compare the performance of different motion planners using OMPL.

Test Cases

OMPL provides 20 geometric planning problems. Because 6 of the problems are too hard to solve within 5 minutes, only 14 of the problems are used in the experiments, described in Chapter 4. Four of the planning problems are shown in Figure 2.10.


(a) Maze (b) Barriers

(c) Abstract (d) Apartment

Figure 2.10: Planning scenarios

2.3 Random Walk Planning Framework

Arvand [15, 14, 16] 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.

A MRW algorithm uses the following key ingredients:

- A *heuristic function* h to evaluate the goal distance of 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. Figure 2.11 demonstrates the process of MRW. The length of each random walk is decided by the local restart strategy, and can be fixed or variable. Different choices will be discussed in Section 3.3. 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.

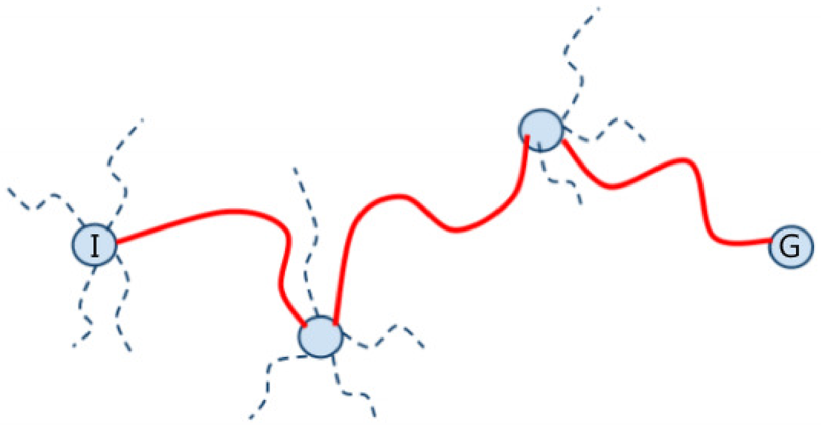


Figure 2.11: Illustration of Monte Carlo Random Walks [21].

2.3.1 Framework of MRW

Algorithm 2, abstracted from [15], shows an outline of MRW planning. This high-level outline is nearly identical for classical and for continuous planning.

The only difference is that the test for achieving the goal G uses a goal condition in classical planning and a goal region in continuous planning.

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 2 Monte Carlo Random Walk Planning

```
Input  Initial State $s_0$, goal $G$
Output  A solution plan

  $s \leftarrow s_0$
  $h_{min} \leftarrow h(s_0)$
  while $s \notin G$ do
    if $h$-value does not improve fast then
      $s \leftarrow s_0$ {restart from initial state}
    end if
    $s \leftarrow randomWalk(s, G)$
    if $h(s) < h_{min}$ then
      $h_{min} \leftarrow h(s)$
    end if
  end while
  return  the plan reaching the state s
```

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 3 shows a pure random walk method similar to the one in [15], 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, Section 3.3 discusses different adaptive global and local restart strategies, which are used by Arvand and can be applied in continuous planning as well.

Algorithm 3 Pure Random Walk

```
Input current state $s$, goal $G$
Output $s_{min}$

1: $h_{min} \leftarrow \infty$
2: $s_{min} \leftarrow NULL$
3: for $i \leftarrow 1$ to $NUM\_WALK$ do
4:     $s' \leftarrow s$
5:     for $j \leftarrow 1$ to $LENGTH\_WALK$ do
6:         $s' \leftarrow sampleNewState(s')$
7:         if $s'$ satisfies $G$ then
8:             return $s'$
9:         end if
10:    end for
11:    if $h(s') < h_{min}$ then
12:        $s_{min} \leftarrow s'$
13:        $h_{min} \leftarrow h(s')$
14:    end if
15: end for
16: if $s_{min} = NULL$ then
17:     return $s$
18: else
19:     return $s_{min}$
20: end if
```

Chapter 3

Application of Random Walk Planning to Motion Planning

3.1 Introduction

The Monte Carlo random walk method (MRW) is successful in classical planning [15, 14, 16]. The framework of MRW is presented in Section 2.3. This chapter applies MRW to motion planning.

The differences of using MRW in classical planning and motion planning are explained in Section 3.2. MRW parameters such as the number and length of random walks, and the maximum number of search episodes, have huge impact on the performance. Section 3.3 shows how to utilize these parameters better and introduces other enhancements for MRW. Section 3.4 shows the algorithms for the continuous versions of MRW, including a bidirectional variant and an optimizing variant. The implementation details of MRW are presented in Section 3.5.

3.2 Approach

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 directly. 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 3.1 summarizes some main differences of applying MRW to classical and motion planning.

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

3.3 Enhancements for MRW

3.3.1 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 [15, 16] 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 $1/r_l$. The fourth strategy called *adaptive local restarting* (ALR) uses a multi-armed bandit method to learn the best r_l from a candidate set based on the heuristic improvement of previous search episodes [16].

Number of Random Walks

The first version of Arvand used a fixed number of random walks in each search episode, then progressed greedily to the best evaluated endpoint. An adaptive method called *acceptable progress* is a better approach, which stops the exploration of an episode if a state with small enough h-value is reached [15]. A simple strategy followed here is to have only one random walk in a local search [14], 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 restarting* (AGR) algorithm described in [16] is a robust method to adaptively adjust t_g during the planning process. Let V_w (V for velocity, w for walks) be the *average heuristic improvement per walk*, s_0 be the initial state, so on average, about $h(s_0)/V_w$ walks should reach $h = 0$. AGR adjusts t_g by continually estimating V_w and setting $t_g = h(s_0)/V_w$.

3.3.2 The Rate of Heuristic Evaluation

While heuristic state evaluations provide key information to guide search, the MRW framework in Section 2.3 evaluates only the endpoint of a random walk as it takes much more time to evaluate every state along the walk. If we evaluate the intermediate states along the walk, what happens to the performance?

Instead of evaluating all states, another strategy introduced in [16] is to evaluate:

- the endpoint of each random walk as in MRW, and

- intermediate states with probability p_{eval}.

This interpolates between evaluating every state with $p_{eval} = 1$ and MRW with $p_{eval} = 0$.

3.3.3 Path Pool

Standard MRW requires very little memory. A *path pool* can store a number of random walks and utilize them for improving later searches [14]. The technique of *Smart Restarts* (SR) is based on a fixed-capacity pool which stores the most promising paths encountered so far. 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.

3.3.4 On-Path Search Continuation

In the MRW framework, after completing a new random walk with endpoint e, the search commits to all the actions on the path to e. The drawback of this is that, if some of the random actions leading to e consume too many resources and the problem becomes unsolvable, then all search effort from this point until the next restart is wasted.

On-Path Search Continuation (OPSC) avoids commitment to all actions leading to e [14]. OPSC randomly picks a state along the existing path to start a new search episode, instead of always starting from an endpoint. If the endpoint of this new search episode has better heuristic value, the new search path is adopted.

3.4 Algorithms

3.4.1 Monte Carlo Random Walk Planning

Algorithm 4 outlines the continuous version of MRW. The main difference from the MRW framework is that it counts the number of consecutive episodes without heuristic improvement. If this number reaches *MAX_EPISODE*, the algorithm restarts from initial state s_0. *MAX_EPISODE* can either be a fixed number or adaptively adjusted as mentioned in Section 3.3.1.

Algorithm 4 Monte Carlo Random Walk Planning

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

  $s \leftarrow s_0$
  $h_{min} \leftarrow h(s_0)$
  $counter \leftarrow 0$
  while $s \notin G$ do
    if $counter > MAX\_EPISODE$ then
      $s \leftarrow s_0$ {restart from initial state}
      $counter \leftarrow 0$
    end if
    $s \leftarrow randomWalk(s, G)$
    if $h(s) < h_{min}$ then
      $h_{min} \leftarrow h(s)$
      $counter \leftarrow 0$
    else
      $counter \leftarrow counter + 1$
    end if
  end while
  return  the plan reaching state $s$ from $s_0$
```

3.4.2 Pure Random Walk

Algorithm 5 shows the continuous version of pure random walks. It contains more details of applying random walks in motion planning than the abstract algorithm in Section 2.3.1. The algorithm first samples a goal state g from the goal region, and tries to reach state g in the search. Function validMotion(a, b) checks whether a motion from state a to state b is valid within the state space. When sampling a new state from the current state s', it always tries to reach g directly from s' first. If such a motion is valid, then a solution is found, otherwise, a state s'' near s' is sampled. This step is repeated if the motion from s' to s'' is invalid. The parameters *NUM_WALK* and *LENGTH_WALK* can be set as fixed numbers, or adaptively adjusted as described in Section 3.3.1.

Algorithm 5 Pure Random Walks.

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

```
1: $h_{min} \leftarrow \infty$
2: $s_{min} \leftarrow NULL$
3: $g \leftarrow$ sampleFromGoalRegion($G$)
4: for $i \leftarrow 1$ to $NUM\_WALK$ do
5:    $s' \leftarrow s$
6:    for $j \leftarrow 1$ to $LENGTH\_WALK$ do
7:       if validMotion($s', g$) then
8:          return $g$
9:       end if
10:      repeat
11:         $s'' \leftarrow$ uniformlySampleFromNear($s', S$)
12:      until validMotion($s', s''$)
13:      $s' \leftarrow s''$
14:   end for
15:   if $h(s') < h_{min}$ then
16:      $s_{min} \leftarrow s'$
17:      $h_{min} \leftarrow h(s')$
18:   end if
19: end for
20: if $s_{min} = NULL$ then
21:    return $s$
22: else
23:    return $s_{min}$
24: end if
```

3.4.3 Path Pool and New Selection Algorithm

The technique of *Smart Restart* mentioned in Section 3.3.3 stores the path of the current search and the heuristic value of its endpoint in a path pool if MRW needs restart. After the pool is full, SR is triggered to restart the search from a random state on a random path in the pool. When adding new path to a full pool, the path with worst h-value is deleted.

Another idea of utilizing a path pool is: to start a new search episode, a path p from the pool is either selected by minimum h-value or randomly picked according to a distribution; then a fixed fraction of the pool contents is replaced by newly generated random walks which extend p. Algorithm 6 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 6 Expand

```
Input current state $s$, goal state $g$, existing path $p$ with endpoint $s$, 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 ← $randomWalk(s, g)$
    new_path ← connect($p$, new_walk)
    store($P$, new_path)
    if solution found then
      return true
    end if
  end for
  return false
```

3.4.4 BArvand: Bidirectional Arvand

Motion planners such as RRT [13] and KPIECE [17] have bidirectional variants with good performance, as mentioned in Section 2.2.4. Bidirectional Arvand (BArvand) uses a similar approach to solve planning problems. It maintains both a forward and a backward path pool. Like RRT and KPIECE, it grows two trees, but the number of branches is limited and a branch can be replaced. 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 7 Bidirectional Arvand

```
Input current state $s_0$, goal state $g_0$, number of new paths $n$
Output A solution path
 1: $h_{min} \leftarrow \infty$
 2: $init \leftarrow true$
 3: repeat
 4:    if $counter > MAX\_EPISODE$ or $init$ then
 5:       $counter \leftarrow 0$
 6:       $fPool, bPool \leftarrow \emptyset$
 7:       $p \leftarrow NULL$
 8:       expand($s_0$, $g_0$, $p$, $n$, $fPool$)
 9:       $s \leftarrow$ closest endpoint towards $g_0$ in $fPool$
10:       expand($s$, $s_0$, $p$, $n$, $bPool$)
11:       $current \leftarrow fPool$
12:       $init \leftarrow false$
13:    end if
14:    reserve($n$, $current$) {reserve room for $n$ new paths}
15:    $s, g \leftarrow \arg\min_{f \in fPool, b \in bPool} h(f, b)$
16:    $p \leftarrow$ complete path towards $s$
17:    expand($s$, $g$, $p$, $n$, $current$) {try to connect two paths}
18:    if $h(fPool, bPool) < h_{min}$ then
19:       $h_{min} \leftarrow h(fPool, bPool)$
20:       $counter \leftarrow 0$
21:    else
22:       $counter \leftarrow counter + 1$
23:    end if
24:    switch forward and backward search direction
25: until a solution is found
26: return solution path
```

Algorithm 7 shows the outline of bidirectional Arvand. When it (re-)starts, *fPool* and *bPool* are reset and an initial forward episode starting from s_0 is conducted using expand() in Algorithm 6, then a initial backward search starts from the closest endpoint towards g_0 in *fPool*. If a solution is found, the algorithm stops and returns, otherwise the initialization is done. In the code, $h(fPool, bPool) = \min_{f \in fPool, b \in bPool} h(f, b)$. Function reserve($n$, *current*) reserves room in the pool for n new paths. It randomly deletes n existing paths if the pool is full. In a normal search episode, search starts from the endpoint of one chosen path s, treats the endpoint of the other chosen path as the search goal g, and tries to connect them. s and g are chosen such that

$h(s, g) = h(fPool, bPool)$. After each normal search episode, the forward and backward search direction is switched.

Enhancements for Bidirectional Arvand

Some enhancements introduced in Section 3.3 also work for Bidirectional Arvand. *Adaptive global restarting* (AGR), *adaptive local restarting* (ALR), stopping randomly using a *local restarting rate* and having more evaluation in BArvand is done in the same way as in Arvand. *On-Path Search Continuation* (OPSC) and *Smart Restart* (SR) are a little different in BArvand.

OPSC in Arvand chooses the best random walk among N walks, and it keeps this new walk only if the endpoint has lower h-value than the endpoint of the previous search path. OPSC in BArvand keeps all N random walks in the pool after expanding the previous path from a random intermediate state. In Arvand, after SR is triggered, only one random path in the pool is used for the new search. Other paths are only kept for the next restart. In BArvand, SR selects one path from *fPool* and one path from *bPool*, and randomly chooses two intermediate states of these two path as s and g for new search exploration. Other paths remain in the pools, and can be chosen in the following normal search episode.

3.4.5 Improving Plan Quality

The algorithms described above stop immediately after a solution is found. An optimizing planner such as RRT* [6], which is an asymptotically-optimal incremental sampling-based motion planning algorithm, keeps optimizing the solution within a time limit. RRT* improves plan quality (path length) by pruning the tree. We use a different method of improving plan quality.

Arvand*, shown in Algorithm 8, is an optimizing version of Continuous Arvand, which keeps restarting even after the first valid plan is found. Function simplify() uses post-processing techniques, such as shortcutting and smoothing, to simplify each newly found solution. Every time MRW returns a new solution *sol*, simplify() is called to check whether the new solution has shorter path length. Only the shortest solution after postprocessing is returned.

Algorithm 8 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 $sol_{min}$
```

Another idea of improving plan quality is to run an optimizing planner such as RRT* for a small amount of time, put the solution path into a path pool, and use Arvand* to improve the initial solution. The search always restarts from a random state on a random path in the pool.

3.5 Implementation Details

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 [18]. OMPL provides implementations of all motion planning primitives as mentioned in Section 2.2.5. 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.

Three motion planners were implemented: Arvand, BArvand and Arvand*. BArvand is the bidirectional Arvand. Besides using fixed parameters, the enhancements for MRW from Section 3.3 and 3.4 are implemented in both Arvand and BArvand. Each enhancement can be switched on and off independently. Some enhancements can not work together as they control one parameter in different ways. For example, regarding the random walk length, only one enhancement can be selected among increasing walk length, using a *local restarting rate* and using *adaptive local restarting*. Considering the number of random walks, either *acceptable progress* or running one random walk per episode can be selected. For *Smart Restart*, *On-Path Search Continuation* and enhancements manipulating different parameters can work together perfectly.

Chapter 4

Experiments

4.1 Experimental Environment

In the experiments, the features of the Arvand planners described in Chapter 3 are tested. Four Arvand planners are compared with a selection of the best-performing planners available in OMPL: RRT [13], KPIECE [17], EST [4], PDST [9], and PRM [7]. Arvand* is tested against RRT* [6], which is an asymptotically-optimal incremental sampling-based motion planning algorithm.

4.1.1 Benchmarking

Experiments used 14 built-in benchmark scenarios from OMPL: Maze, Barriers, Abstract, Apartment, BugTrap, RandomPolygons, UniqueSolutionMaze, Cubicles, Alpha, Easy, Home, Twistycool, Pipedream_ring and Spirelli. These scenarios are chosen as they can be solved by most available planners in reasonable time (less than 10 minutes). We grouped these scenarios into four categories: easy problems (Maze, BugTrap, RandomPolygons, Easy), intermediate problems (Alpha, Barriers, Apartment, Twistycool), 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.

4.1.2 Machines Used

All experiments were run on three machines with the same 8-core CPU Intel Xeon E5420 @ 2.5GHz and 12GB memory (muriel, muriel1 and willingdon with 8GB memory).

4.1.3 Metrics

In the experiment, except for the one run experiment, results for each planner are the median over 10 runs per scenario. The metrics of memory use (MB), path length, simplified path length, planning time and simplification time (in seconds) are considered. The simplified path length is considered as plan quality. From here on, path length refers to simplified path length.

4.2 Stage 1: One Run Experiment

The one run experiment is intended to provide the preview of performance of Arvand planners. The result shows the effect of enabling different features.

4.2.1 Basic Arvand

The basic Arvand uses three parameters N (number of random walks), L (length of random walk) and $maxE$ (maximum number of episodes before restart). The range of N is $[10, 20, 50, 100, 200, 500, 1000]$; the range of L is $[10, 20, 50, 100, 200, 500, 1000]$; the range of $maxE$ is $[5, 10, 20, 50]$. In the one run experiment, we varied all three parameters to get 196 settings for each planning scenario. To see the effect of parameters, we design a "battle" between parameters based on the experimental data. For example, for parameter $N = 10$ and $N = 20$, we compare the path length and planning time of these two "players" with the same setting of other parameters. If $N = 10$ uses less planning time than $N = 20$ on setting $L = 100$, $maxE = 5$ in planning scenario Maze, it means $N = 10$ wins in this round. If one player wins its "battles" against all other players, it is considered as the best setting for this parameter.

N	10	20	50	100	200	500	1000
10	0	**176**	**186**	**190**	**178**	**206**	**207**
20	169	0	**172**	160	**187**	**175**	**182**
50	160	159	0	**167**	**181**	**180**	**182**
100	154	**170**	157	0	**166**	**167**	**168**
200	168	146	142	152	0	**159**	**170**
500	138	159	147	150	146	0	**153**
1000	138	151	146	151	139	145	0

Table 4.1: The battle of different N on path length. There are 392 test cases in a "battle". Test cases in which both "players" time out are not counted.

L	10	20	50	100	200	500	1000
10	0	143	**154**	**175**	168	171	166
20	**157**	0	**167**	**188**	**185**	**184**	178
50	147	127	0	**170**	153	169	169
100	143	125	135	0	150	157	152
200	**175**	161	**187**	**185**	0	**184**	177
500	**190**	176	**191**	**201**	180	0	**184**
1000	**195**	**185**	**195**	**208**	**184**	180	0

Table 4.2: The battle of different L on path length. There are 392 test cases in a "battle". Test cases in which both "players" time out are not counted.

$MaxE$	5	10	20	50
5	0	276	**288**	**286**
10	**284**	0	257	**285**
20	274	**297**	0	**286**
50	280	272	267	0

Table 4.3: The battle of different $maxE$ on path length. There are 686 test cases in a "battle". Test cases in which both "players" time out are not counted.

N	10	20	50	100	200	500	1000
10	0	**208**	**224**	**226**	**215**	**232**	**215**
20	137	0	**194**	**189**	**191**	**204**	**187**
50	122	137	0	**181**	**191**	**188**	**173**
100	118	141	143	0	**169**	**189**	**173**
200	131	142	132	149	0	**174**	**174**
500	112	130	139	128	131	0	149
1000	130	146	155	146	135	149	0

Table 4.4: The battle of different N on planning time. There are 392 test cases in a “battle”. Test cases in which both “players” time out are not counted.

L	10	20	50	100	200	500	1000
10	0	131	125	124	92	83	83
20	**169**	0	132	130	96	80	82
50	**176**	**162**	0	142	119	96	100
100	**194**	**183**	**163**	0	124	105	96
200	**251**	**250**	**221**	**211**	0	150	142
500	**278**	**280**	**264**	**253**	**214**	0	155
1000	**278**	**281**	**264**	**264**	**219**	**209**	0

Table 4.5: The battle of different L on planning time. There are 392 test cases in a “battle”. Test cases in which both “players” time out are not counted.

$MaxE$	5	10	20	50
5	0	**294**	**284**	**289**
10	266	0	**280**	262
20	278	274	0	**286**
50	277	**295**	267	0

Table 4.6: The battle of different $maxE$ on planning time. There are 686 test cases in a “battle”. Test cases in which both “players” time out are not counted.

Table 4.1 shows the battle outcome for parameter N. The number in each row are the number of games this setting wins, when playing against the parameter setting in the given column. From this table, we can see that smaller numbers of random walks tend to provide better plan quality (shorter path length). From Tables 4.2 and 4.3, it is hard to see the effect of parameter

L and $maxE$ on plan quality. Tables 4.4 and 4.5 show that choosing a smaller number of random walks and longer length of random walks is faster. The parameter $maxE$ shows no large effect on performance according to Table 4.6. It seems that the Arvand planner seldom restarts in these planning scenarios.

4.3 Stage 2: 10 Run Experiment

The second stage of the experiment is based on 10 runs per setting per scenario. The data in this and the following sections is the median over 10 runs (the average of 5th and 6th largest number) by default. This section discusses the effect of different enhancements.

4.3.1 Baseline: Basic Arvand

Based on the result of the one run experiment, we narrowed the range of parameters in the following experiments. The new range of N is $[5, 10, 20, 50, 100]$; the range of L is $[10, 100, 1000]$; and $maxE$ is fixed to 10 because larger $maxE$ cannot improve performance as it rarely restarts. Because there is no best setting for all planning scenarios, we need to choose a setting that works well for all problems as our baseline. First, we sort the settings based on path length for each planning scenario. The top 20% are considered as good settings for this planning scenario. We define *score* for a setting as the number of planning scenarios in which this setting is a good one. After repeating the same process on planning time, each setting has a score on path length and a score on planning time. The setting that has highest scores on both metrics is the best one. Using this approach, we choose a setting $[N = 20, L = 1000, maxE = 10]$ as the baseline of Arvand planners. This approach is also used for choosing the best settings when enabling enhancements.

Table 4.7 shows the median result of the baseline on three metrics for all planning scenarios. The time limit for all planning scenarios is 5 minutes. The results show that basic Arvand cannot solve planning scenarios Home and Spirelli within 5 minutes.

Planning Scenario	Memory	Path Length	Time	Rate of Time Out
Abstract	1.59	791.10	44.85	-
Alpha	0.38	427.13	4.20	-
Apartment	0.32	417.70	17.52	-
Barriers	1.00	1,472.27	3.34	-
BugTrap	0.86	174.02	0.45	-
Cubicles	1.06	2,419.55	1.33	-
Easy	1.16	204.40	0.37	-
Home	-	-	-	1
Maze	0.77	113.13	0.68	-
Pipedream_ring	0.50	103.85	1.15	-
RandomPolygons	0.66	129.46	0.10	-
Spirelli	-	-	-	1
Twistycool	11.86	228.87	62.79	0.4
UniqueSolutionMaze	1.30	360.24	5.18	-

Table 4.7: The baseline of Arvand planners

4.3.2 Enhancements for Single Directional Arvand

The result discussed in this subsection is shown in Figures 4.1 and 4.2. The approach of choosing the best setting described in Section 4.3.1 is used in this subsection.

One Random Walk per Episode

This enhancement sets the number of random walks as 1. In the experiment, the path length before simplification can be 10 times longer than for other planners. But the path length after simplification is basically the same as the baseline. The best setting for L is 1000. Figure 4.1(a) shows that it is two times faster in planning scenario UniqueSolutionMaze and 60% faster in scenario Apartment, but slower in scenario Easy. The result shows that this feature has little influence on both plan quality and planning time.

Using a Local Restarting Rate

Introduced in Section 3.3.1, this enhancement involves a local restarting rate $r \in \{0.1, 0.01, 0.001\}$, while $N \in \{5, 10, 20, 50, 100\}$. A good setting is $[N = 5, r = 0.001]$. It is faster in Apartment and UniqueSolutionMaze, but much slower in other scenarios, as shown in Figure 4.1(b). The plan quality is 20% worse than the baseline in Abstract and Alpha. Overall, this is not a good enhancement.

Acceptable Progress

Described in Section 3.3.1, Acceptable Progress adaptively calculates the number of random walks (N) during the search. A good choice of L is 1000 among $[10, 100, 1000]$. Figure 4.1(c) shows that this setting is 3 times faster than the baseline in Abstract and Apartment, but about 40% slower in Barriers, BugTrap, Easy, Maze and UniqueSolutionMaze. This setting has no improvement on plan quality.

Adaptive Global Restarting

Adaptive Global Restarting (AGR), described in Section 3.3.1, adaptively calculates the maximum number of episodes before restart. The range of N is $[5, 10, 20, 50, 100]$, and the range of L is $[10, 100, 1000]$. A good setting for AGR in our experiments is $[N = 5, L = 1000]$. Compared to the baseline, this setting uses much more time in Abstract, Twistycool and UniqueSolutionMaze, but runs twice faster in Apartment, as shown in Figure 4.1(d). Also in 2 out of 10 runs it solved the planning problem Home within 4 minutes.

Adaptive Local Restarting

Adaptive Local Restarting (ALR), described in Section 3.3.1, chooses the best value of r among $[0.1, 0.01, 0.001]$ during the search according to statistics. Among $[5, 10, 20, 50]$, $N = 20$ is the best setting due to the result. According to Figure 4.1(e), this setting is about 50% faster than the baseline in Apartment, Barriers and BugTrap, but 40% slower in Cubicles, RandomPolygons, Twistycool and UniqueSolutionMaze. The plan quality in Alpha is 30% worse than the baseline.

The Rate of Heuristic Evaluation

As introduced in Section 3.3.2, instead of only evaluating the endpoint of each walk, this enhancement evaluates the intermediate states with probability p_{eval}, for $p_{eval} \in \{0, 0.25, 0.5, 0.75, 1\}$. The range of N is $[5, 10, 20, 50, 100]$, and the range of L is $[10, 100, 1000]$. Using the same method as in Section 4.2.1, we compared the effect of different p_{eval}. The result is shown in Tables 4.8 and 4.9.

	0	0.25	0.5	0.75	1
0	0	**99**	78	**96**	**92**
0.25	77	0	81	83	81
0.5	**101**	**97**	0	**95**	**100**
0.75	81	**93**	84	0	86
1	85	**95**	79	**91**	0

Table 4.8: Result of different p_{eval} on plan quality. There are 210 test cases in a "battle". Test cases in which both "players" time out are not counted.

	0	0.25	0.5	0.75	1
0	0	55	57	50	55
0.25	**121**	0	79	74	78
0.5	**122**	**99**	0	78	86
0.75	**127**	**102**	**101**	0	**91**
1	**122**	**98**	**93**	86	0

Table 4.9: Result of different p_{eval} on planning time. There are 210 test cases in a "battle". Test cases in which both "players" time out are not counted.

From the tables, $p_{eval} = 0.5$ is the best setting for plan quality, while $p_{eval} = 0.75$ is good for planning time. The setting we used to compare with the baseline is $[N = 10, L = 1000, p_{eval} = 0.5]$. This setting runs 6 times faster in Twistycool, but 50% slower in Alpha, BugTrap, Easy and Maze, as shown in Figure 4.1(f).

Smart Restarting

Smart Restarting (SR), described in Section 3.3.4, uses a path pool to keep promising paths every time it restarts. It does not utilize a previous path until the pool is full (pool size = 100), so it is only triggered in harder problems. The range of N is $[5, 10, 20, 50, 100]$, and the range of L is $[10, 100, 1000]$. A good setting is $[N = 10, L = 1000]$. Compared to the baseline, this setting is 50% faster in Apartment, RandomPolygons and UniqueSolutionMaze, but 25% slower in other problems, as shown in Figure 4.1(g). The plan quality in Alpha is 40% worse than the baseline.

On Path Search Continuation

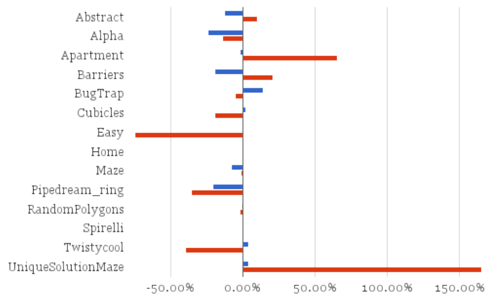
The technique of On-Path Search Continuation (OPSC) is described in Section 3.3.4. In the experiment, the range of N is $[5, 10, 20, 50, 100]$, and the range of L is $[10, 100, 1000]$. The best setting is $[N = 10, L = 1000]$ is $[N = 5, L = 1000]$. This setting runs 50% faster in Apartment and Maze, but about 40% slower in other problems, as shown in Figure 4.1(h).

AGR + ALR

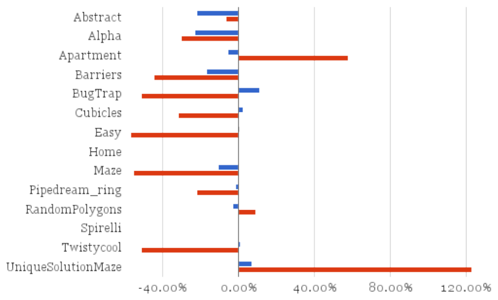
Using AGR and ALR, the only parameter is N, $N \in \{5, 10, 20, 50, 100\}$. $N =$ 10 is the best setting. Compared with the baseline, this setting runs more than 50% faster in Apartment, BugTrap, Maze and Pipedream_dream. It is 30% slower in other problems, as shown in Figure 4.2(a).

Arvand+: AGR + ALR + Acceptable Progress

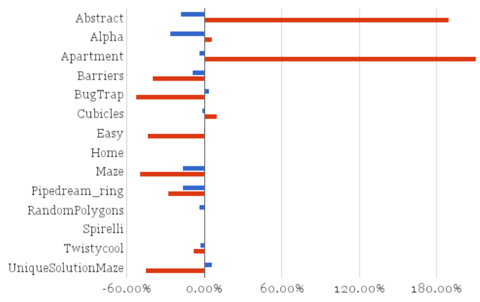
This combination of enhancements requires no setting of parameters. This variant of Arvand is named as Arvand+. Figure 4.2(b) shows that it is two times faster in UniqueSolutionMaze, compared with the baseline. Although the planning time for Cubicles, Easy, Maze, RandomPolygons and Twistycool is 40% slower, it is still reasonable. The plan quality of Arvand+ is basically the same as the baseline. Arvand+ is the ideal variant for a normal user to use Arvand for planning.



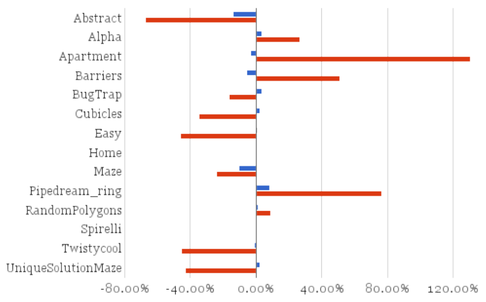
(a)



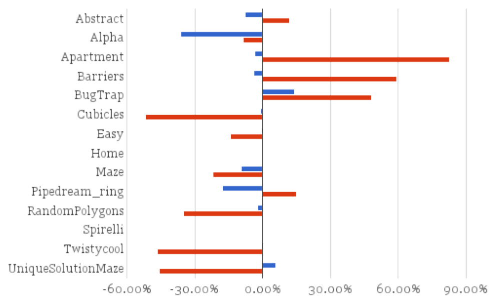
(b)



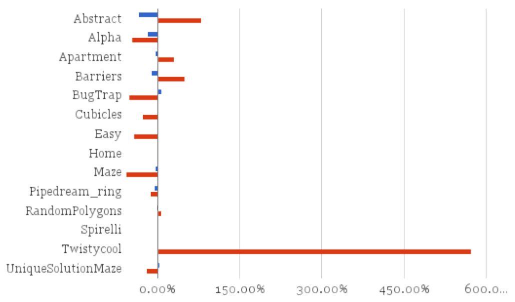
(c)



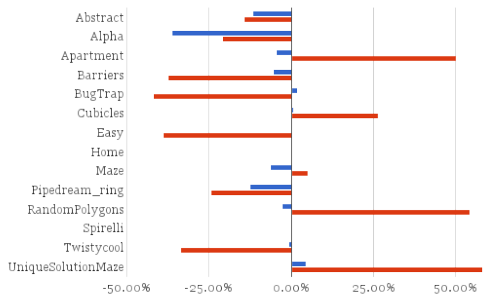
(d)



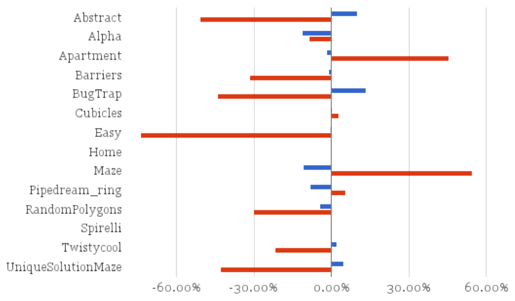
(e)



(f)

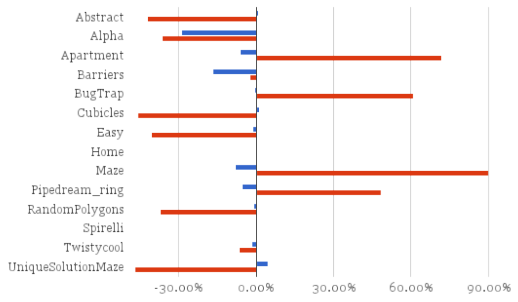


(g)

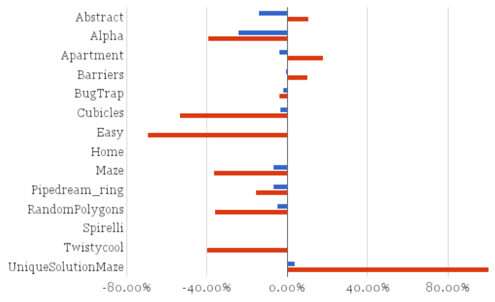


(h)

Figure 4.1: Results of the enhancements for single directional Arvand, compared with the baseline. Plan quality in blue, planning time in red.



(a)



(b)

Figure 4.2: Results of the enhancements for single directional Arvand, compared with the baseline. Plan quality in blue, planning time in red.

4.3.3 Enhancements for Bidirectional Arvand

The result discussed in this subsection is shown in Figures 4.3. The approach of choosing the best setting described in Section 4.3.1 is also used in this subsection.

BArvand: Basic Bidirectional Arvand

The basic bidirectional Arvand (BArvand), as discussed in Section 3.3.4, has one parameter L, the length of a random walk. Among $[10, 100, 1000]$, $L = 1000$ works well for most problems, and is used in the following discussion.

Compared with basic Arvand, BArvand runs 4 times faster in planning scenarios Apartment, but about 50% slower in other problems, as shown in Figure 4.3(a). The plan quality in Abstract and Barriers is about 25% worse. In 2 of 10 runs, it solved the hard problem Home.

BArvand + AGR

As shown in Figure 4.3(b), this combination runs faster than BArvand except for Apartment and Twistycool, and solved problem Home in 3 of 10 runs. Plan quality is good except for Home. The path length in Abstract is also shorter than the baseline.

BArvand + ALR

Figure 4.3(c) shows that this variant has similar result on planning time as using AGR. Home is only solved in one run and the path length is much shorter, compared with BArvand.

BArvand + a Local Restarting Rate

Among $[0.1, 0.01, 0.001]$, $r = 0.001$ is the best setting for this variant. Figure 4.3(d) shows that it is two time faster than BArvand in Barriers and Cubicles. Plan quality is much better than BArvand in Abstract.

BArvand + Full Evaluation

The setting for this variant is $L = 1000, p_{eval} = 1$. Figure 4.3(e) shows that it uses much more time on the easy problems. In 4 of 10 runs, it solved the hard problem Home and the median time for these 4 runs is 79s, less than in the previous experiments.

BArvand + Smart Restarting

From Figure 4.3(f), the result is very similar to the experiment of using a local restarting rate. Smart Restarting is rarely triggered.

BArvand + OPSC

Figure 4.3(g) shows that this variant is 5 times faster than BArvand in Twistycool. It has no improvement on plan quality.

BArvand+: BArvand + AGR + ALR

This adaptive version of bidirectional Arvand is 60% slower than BArvand in Apartment and Twistycool, shown in Figure 4.3(h). The plan quality is similar.

Among the enhancements of BArvand, AGR and ALR are better. Compared with basic Arvand, BArvand can solve the hard problem Home in some runs within 5 minutes, instead of all time out. We can use Arvand to handle easy problems, and use BArvand to try solving harder problems.

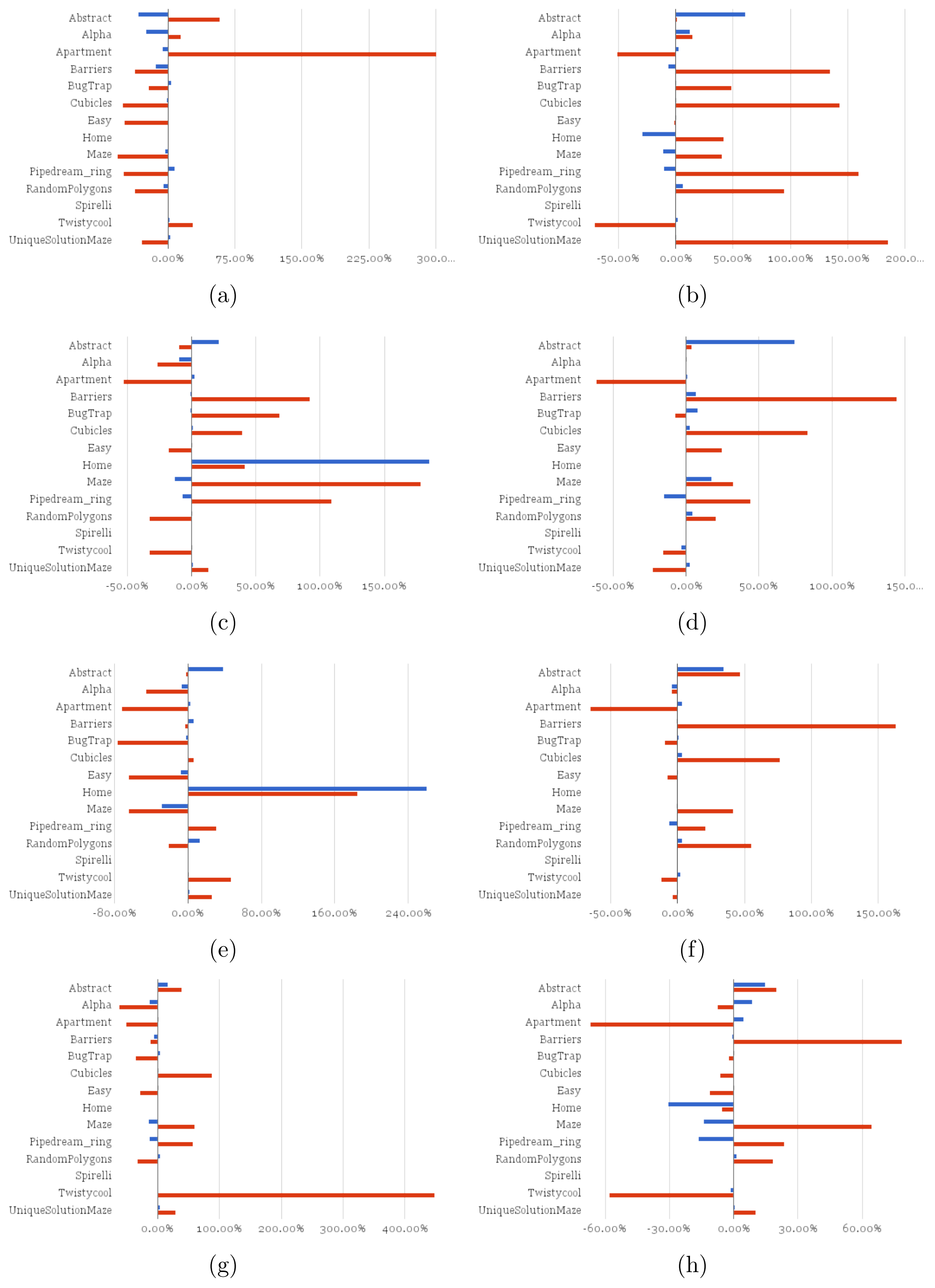


Figure 4.3: (a) Result of BArvand compared with the baseline; (b)-(h) Results of enhancements for BArvand compared with BArvand. Plan quality in blue, planning time in red.

4.4 Comparison with Other Planners

We select the basic Arvand, BArvand and their adaptive variants Arvand+, BArvand+ to compare with other planners: RRT, KPIECE, EST, PDST and PRM.

Tables 4.10-4.23 show the benchmark result. The result is the median over 10 runs. For the metric of memory use, almost all Arvand versions use less memory than all the other planners in most planning scenarios. The two exceptions are BArvand and BArvand+ in scenario Twistycool, which used more memory to maintain two path pools. Another exception is scenario Home for Arvand+ and BArvand+ since this scenario has long detours, and Arvand planners need much longer paths to reach the goal.

Considering the simplified path length, in scenarios Abstract and Apartment, basic Arvand outperforms all other planners. In scenarios Alpha, Barriers, BugTrap, Cubicles, Twistycool and UniqueSolutionMaze, all 4 Arvand versions provide competitive simplified path length.

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.

All Arvand planners can solve easy planning problems as fast as other planners, such as the scenarios Alpha, BugTrap, Easy, Maze and RandomPolygons. For intermediate problems with long detours, such as Cubicles and BugTrap, using wrong parameter settings can significantly drag down the performance. For example, if the length of random walk is set as 10, basic Arvand can never solve Cubicles within 5 minutes. 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. We need longer walks to grab the chance to go against the heuristic. However, if we use the adaptive versions of Arvand, a good setting of parameters is automatically calculated by the algorithm.

For problems with long narrow passages, such as Spirelli and Home, Arvand takes a long time or fails to solve them within the time limit. The main reason is that a narrow passage makes it harder to sample a valid new state for a valid motion from the current position. When the search is trapped in a narrow passage, most of the time is spent on sampling and validating since most sampled states are impossible to reach. RRT works better on these problems. It samples states from the whole state space and builds a tree. When the tree is dense enough around the passage, it can find a way out.

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	13.41	1,146.80	7.31	-
EST	7.02	808.56	8.56	-
PDST	211.44	1,104.24	**6.54**	-
RRT	116.93	998.84	26.60	0.5
PRM	153.09	**510.58**	96.97	0.9
Arvand	**1.59**	791.10	44.85	-
Arvand+	1.68	925.01	40.47	-
BArvand	2.96	1,197.49	28.26	-
BArvand+	3.57	1,043.58	23.50	-

Table 4.10: Abstract

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	1.08	667.09	3.88	-
EST	0.61	607.46	4.56	-
PDST	30.50	714.01	**1.29**	-
RRT	**0.32**	500.28	5.98	-
PRM	430.01	**288.14**	133.41	0.30
Arvand	0.38	427.13	4.20	-
Arvand+	0.39	568.43	6.94	-
BArvand	0.35	569.19	3.67	-
BArvand+	1.85	522.81	3.98	-

Table 4.11: Alpha

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	0.65	451.10	9.48	-
EST	0.52	431.52	10.41	-
PDST	178.74	430.89	19.45	-
RRT	0.76	437.92	9.61	0.2
PRM	60.04	418.55	106.33	0.1
Arvand	0.32	**417.70**	17.52	-
Arvand+	0.31	436.97	14.87	-
BArvand	**0.28**	445.91	**4.34**	-
BArvand+	1.93	425.55	13.15	-

Table 4.12: Apartment

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	1.09	1,690.77	**0.61**	-
EST	2.37	1,407.06	2.51	-
PDST	29.21	1,903.69	1.00	0.2
RRT	268.24	1,562.04	0.71	0.4
PRM	4.21	**1,244.04**	1.37	-
Arvand	1.00	1,472.27	3.34	-
Arvand+	1.68	1,484.91	3.02	-
BArvand	2.12	1,717.66	5.37	-
BArvand+	**0.88**	1,735.41	3.01	-

Table 4.13: Barriers

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	0.87	169.11	0.40	-
EST	0.83	163.78	**0.25**	-
PDST	13.82	166.71	0.34	-
RRT	0.85	175.54	0.27	-
PRM	3.33	**137.44**	1.04	-
Arvand	0.86	174.02	0.45	-
Arvand+	**0.63**	178.67	0.47	-
BArvand	0.98	167.14	0.57	-
BArvand+	1.32	166.53	0.59	-

Table 4.14: BugTrap

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	1.30	2,568.59	1.06	-
EST	2.02	2,437.99	6.50	-
PDST	32.90	2,617.90	1.81	-
RRT	**0.37**	2,528.90	**0.47**	-
PRM	8.71	**2,345.25**	2.33	-
Arvand	1.06	2,419.55	1.33	-
Arvand+	1.56	2,519.44	2.88	-
BArvand	0.99	2,463.48	2.73	-
BArvand+	1.73	2,468.85	2.92	-

Table 4.15: Cubicles

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	2.63	289.69	0.54	-
EST	0.79	280.71	0.12	-
PDST	**0.73**	212.02	**0.05**	-
RRT	0.99	219.57	0.14	-
PRM	1.95	227.65	0.58	-
Arvand	1.16	204.40	0.37	-
Arvand+	1.60	204.75	1.22	-
BArvand	1.57	204.68	0.73	-
BArvand+	2.46	**204.36**	0.83	-

Table 4.16: Easy

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	-	-	-	1
EST	17.69	2,269.98	27.85	-
PDST	3,015.54	2,343.02	128.48	0.1
RRT	**6.54**	2,329.22	**9.38**	-
PRM	75.90	1,508.49	67.31	-
Arvand	-	-	-	1
Arvand+	-	-	-	1
BArvand	116.57	**1,454.94**	228.09	0.8
BArvand+	162.03	2,098.00	241.48	0.8

Table 4.17: Home

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	0.80	149.84	0.85	-
EST	0.71	116.01	1.17	-
PDST	31.86	123.65	0.74	0.2
RRT	**0.39**	**84.35**	**0.32**	-
PRM	0.93	84.96	0.83	-
Arvand	0.77	113.13	0.68	-
Arvand+	0.55	121.73	1.08	-
BArvand	0.69	116.97	1.58	-
BArvand+	0.73	136.15	0.96	-

Table 4.18: Maze

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	25.07	95.21	36.80	0.1
EST	4.41	95.49	1.74	-
PDST	8.22	90.88	**0.53**	-
RRT	**0.27**	92.17	1.07	-
PRM	106.01	**83.71**	26.73	-
Arvand	0.50	103.85	1.15	-
Arvand+	0.35	112.01	1.37	-
BArvand	0.37	96.53	2.31	-
BArvand+	0.33	115.58	1.87	-

Table 4.19: Pipedream_dream

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	0.44	131.37	0.07	-
EST	0.40	130.05	0.24	-
PDST	2.80	**127.88**	0.05	0.10
RRT	**0.33**	131.33	**0.03**	-
PRM	0.73	130.85	**0.03**	-
Arvand	0.66	129.46	0.10	-
Arvand+	0.71	136.60	0.15	-
BArvand	0.49	137.36	0.16	-
BArvand+	0.38	135.38	0.13	-

Table 4.20: RandomPolygons

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	-	-	-	1
EST	-	-	-	1
PDST	1,931.83	**159.21**	237.95	0.9
RRT	**0.58**	167.42	**135.24**	-
PRM	-	-	-	1
Arvand	-	-	-	1
Arvand+	-	-	-	1
BArvand	-	-	-	1
BArvand+	-	-	-	1

Table 4.21: Spirelli

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	256.31	236.38	28.27	0.2
EST	92.75	224.15	8.60	0.1
PDST	1,462.24	233.61	24.34	-
RRT	**9.17**	**221.31**	**4.31**	-
PRM	18.33	243.00	7.86	-
Arvand	11.86	228.87	62.79	0.4
Arvand+	63.79	228.27	104.74	0.2
BArvand	129.30	224.03	48.88	0.3
BArvand+	204.80	227.65	116.90	0.1

Table 4.22: Twistycool

Planner	Memory	Path Length	Total Time	Rate of Time Out
KPIECE	**1.17**	393.80	**1.27**	-
EST	1.18	364.59	3.53	-
PDST	148.63	341.07	7.14	0.2
RRT	2.41	345.21	2.31	-
PRM	1.87	**323.22**	2.04	-
Arvand	1.30	360.24	5.18	-
Arvand+	1.47	346.49	2.59	-
BArvand	2.09	348.71	7.36	-
BArvand+	1.55	345.90	6.65	-

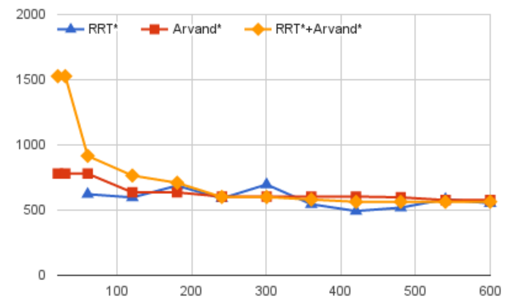
Table 4.23: UniqueSolutionMaze

4.5 Improving Planning Quality

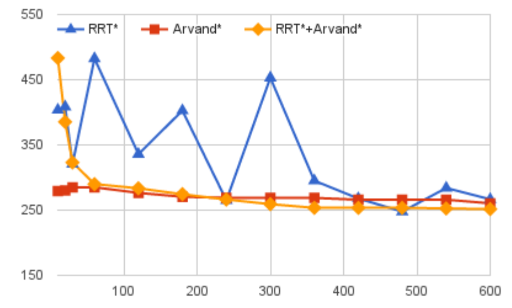
The optimizing planner Arvand* is implemented based on Arvand+. The performance of the optimizing planners RRT* and Arvand* with a 10 minutes limit is shown in Table 4.24. On the main metric of simplified path length, Arvand* is better than RRT* in scenarios Alpha, Apartment, Easy, Pipedream_ring and RandomPolygons. Arvand* can not solve Home and Spirelli within the time limits.

A combination of RRT* and Arvand* runs RRT* for 10s to get a solution, then uses Arvand* to improve this solution for the remaining time. Figures 4.4 and 4.5 show the plan improvement over time for all 14 scenarios. For most planning scenarios, the initial solution found by Arvand* is of rather poor quality, its path length decreases rapidly in the first two minutes and then becomes more stable. For scenarios Barriers, BugTrap, Cubicles, Maze and UniqueSolutionMaze, there is a performance gap between Arvand* and RRT*. RRT* performs better than Arvand* in these problems. Seeding Arvand* with RRT* paths narrows the gap.

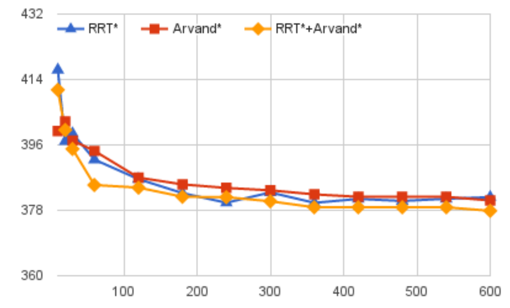
For generating Figures 4.4 and 4.5, RRT* is run separately for different time limits, since the intermediate paths when RRT* is optimizing its plan are not accessible. Each data point is the median of path length over 10 runs, or less number of runs if some runs have no solution at that point. In Figures 4.4(b), 4.4(c) and 4.5(e), the lines of Arvand* go up at some data points because some runs output their first solutions, which are poor, and drag down the medians.



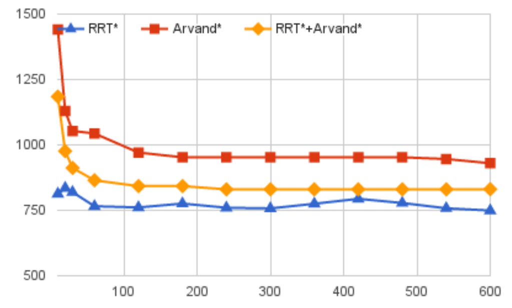
(a) Abstract



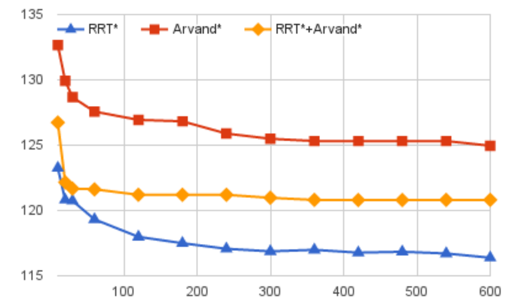
(b) Alpha



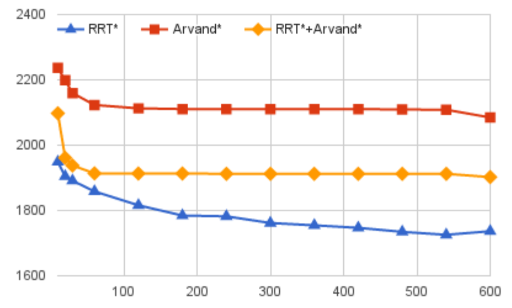
(c) Apartment



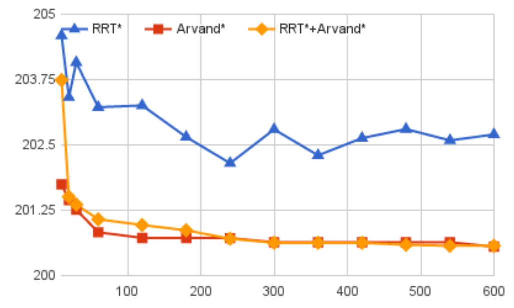
(d) Barriers



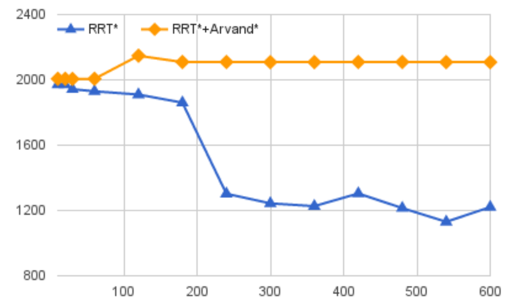
(e) BugTrap



(f) Cubicles

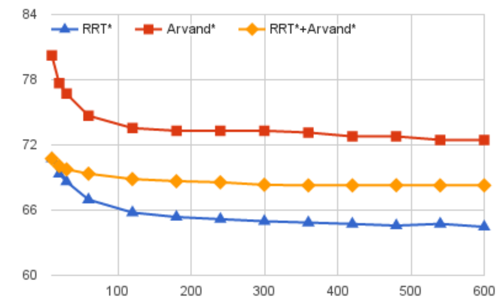


(g) Easy

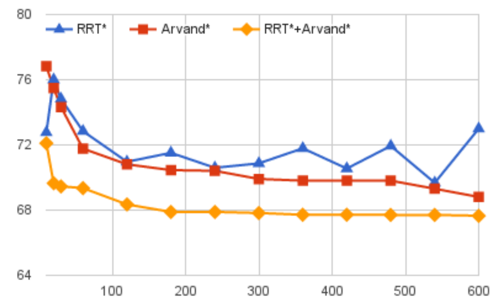


(h) Home

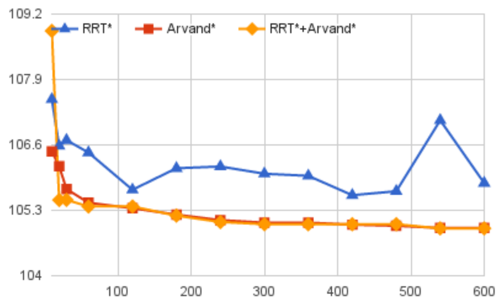
Figure 4.4: Plan improvement over time for RRT*, Arvand* and their combination. Median of path length over 10 runs. RRT* in blue, Arvand* in red, combination of RRT* and Arvand* in orange.



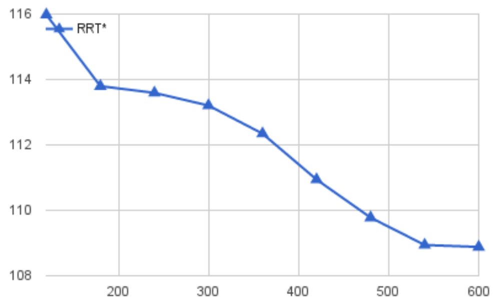
(a) Maze



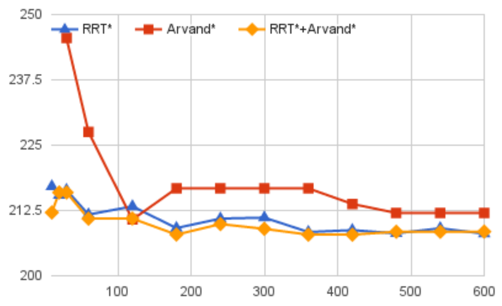
(b) Pipedream_ring



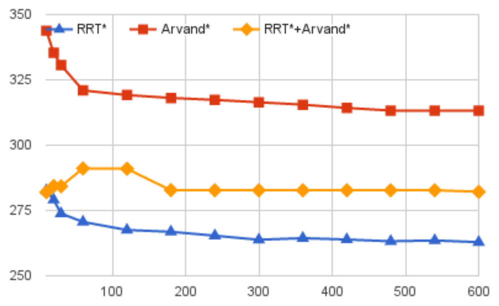
(c) RandomPolygons



(d) Spirelli



(e) Twistycool



(f) UniqueSolutionMaze

Figure 4.5: Plan improvement over time for RRT*, Arvand* and their combination. Median of path length over 10 runs. RRT* in blue, Arvand* in red, combination of RRT* and Arvand* in orange.

Problem	Planner	Memory	Path Length	Rate of Time Out
Abstract	RRT*	173.02	552.50	0.3
	Arvand*	14.24	576.99	-
Alpha	RRT*	371.74	266.60	-
	Arvand*	12.05	260.49	-
Apartment	RRT*	86.41	381.63	-
	Arvand*	2.30	380.71	-
Barriers	RRT*	343.99	749.24	0.2
	Arvand*	189.85	929.91	-
BugTrap	RRT*	618.72	116.38	-
	Arvand*	564.37	124.96	-
Cubicles	RRT*	131.27	1,736.56	-
	Arvand*	208.22	2,084.80	-
Easy	RRT*	310.23	202.70	-
	Arvand*	591.30	200.55	-
Home	RRT*	156.00	1,220.66	-
	Arvand*	-	-	1
Maze	RRT*	321.81	64.46	-
	Arvand*	233.67	72.46	-
Pipedream_ring	RRT*	292.81	73.00	-
	Arvand*	79.82	68.81	-
RandomPolygons	RRT*	407.08	105.84	-
	Arvand*	574.21	104.95	-
Spirelli	RRT*	32.91	108.88	-
	Arvand*	-	-	1
Twistycool	RRT*	257.18	208.05	-
	Arvand*	142.63	212.02	-
UniqueSolutionMaze	RRT*	361.36	262.86	-
	Arvand*	130.28	313.25	-

Table 4.24: Comparing RRT* and Arvand* with a 10 minutes limit.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

This thesis developed algorithms that apply the Monte Carlo Random Walk method to motion planning. The differences of using MRW in classical planning and motion planning are discussed and the approach of applying MRW in motion planning is introduced in Section 3.2. There are several global and local restart strategies in MRW that adapt the parameters, as explained in Section 3.3. These strategies have huge impact on performance, as shown in Sections 4.3.2 and 4.3.3. Besides the existing techniques, a bidirectional search algorithm is proposed in Section 3.4.4, and an optimizing algorithm is introduced in Section 3.4.5. Five planners, Arvand, Arvand+, BArvand, BArvand+ and Arvand* are implemented and compared with other popular motion planners RRT, KPIECE, EST, PDST, PRM and RRT*. The result shows that the Arvand planners are competitive against other planners on the planning problems provided by OMPL. The Arvand planners use much less memory than other planners, which makes them attractive for embedded applications with limited resources.

5.2 Future Work

Portfolio planning [3] 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 [19, 20]. Our results indicate that adding Continuous Arvand to a motion planning portfolio will very likely strengthen its performance.

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.

Bibliography

[1] Weifeng Chen and Martin Müller. Continuous Arvand: Motion planning with Monte Carlo random walks. In *ICAPS 2015 Workshop on Planning and Robotics (PlanRob)*, pages 23–29, 2015.

[2] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice.* Elsevier, 2004.

[3] Carla P Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1):43–62, 2001.

[4] David Hsu, J-C Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 3, pages 2719–2726. IEEE, 1997.

[5] Nathan Jacobson. *Basic algebra I.* Courier Corporation, 2012.

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

[7] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on*, 12(4):566–580, 1996.

[8] James J Kuffner and Steven M LaValle. RRT-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[9] Andrew M Ladd and Lydia E Kavraki. Motion planning in the presence of drift, underactuation and discrete system changes. In *Robotics: Science and Systems*, pages 233–240, 2005.

[10] Jean-Claude Latombe. *Robot motion planning*, volume 124. Springer Science & Business Media, 2012.

[11] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report 98-11, Dept. of Computer Science, Iowa State University, 1998.

[12] Steven M LaValle. *Planning algorithms.* Cambridge University Press, 2006.

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

[14] Hootan Nakhost, Jörg Hoffmann, and Martin Müller. Resource-constrained planning: A Monte Carlo random walk approach. In *ICAPS*, pages 181–189, 2012.

[15] Hootan Nakhost and Martin Müller. Monte-Carlo exploration for deterministic planning. In *IJCAI*, volume 9, pages 1766–1771, 2009.

[16] Hootan Nakhost and Martin Müller. Towards a second generation random walk planner: an experimental exploration. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 2336–2342. AAAI Press, 2013.

[17] Ioan A Şucan and Lydia E Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII*, pages 449–464. Springer, 2010.

[18] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. `http://ompl.kavrakilab.org`.

[19] Richard Valenzano, Hootan Nakhost, Martin Müller, Jonathan Schaeffer, and Nathan Sturtevant. ArvandHerd: Parallel planning with a portfolio. In L. De Raedt, editor, *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 786–791. IOS Press, 2012.

[20] Richard Valenzano, Hootan Nakhost, Martin Müller, Jonathan Schaeffer, and Nathan Sturtevant. Arvandherd 2014. In M. Vallati, L. Chrpa, and T. McCluskey, editors, *The Eighth International Planning Competition*, pages 1–5. University of Huddersfield, 2014.

[21] Fan Xie, Hootan Nakhost, and Martin Müller. Planning via random walk-driven local search. In *ICAPS*, pages 315–322, 2012.