FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Multi-Modal Path Planning for Solving Abstract Robot Tasks

Sören Jentzsch

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Multi-Modal Path Planning for Solving Abstract Robot Tasks

Multimodale Pfadplanung zur Lösung abstrakter Roboteraufgaben

Author:	Sören Jentzsch
Supervisor:	Prof. Dr.-Ing. habil. Alois Knoll
Advisor:	Andre Gaschler, M.Sc.
Submission Date:	January 15, 2014

Ich versichere, dass ich diese Masterarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this master's thesis only supported by declared resources.

Garching, den 14. Januar 2014

Sören Jentzsch

Abstract

For intelligent robots to solve real-world tasks, the problem is not only to plan motion paths, but rather to plan for picking, pushing, sliding, and many other diverse manipulation actions in a complex world of movable objects. In this thesis, we present algorithms, which are able to plan for manipulation and follow the multi-modal nature induced by these actions. We extend basic sampling-based motion planning to integrate Diverse Action Manipulation (DAMA) [6], and show that based on the Rapidly-exploring Random Tree (RRT), we can then solve DAMA scenarios of various kinds. We present three DAMA solving algorithms, which are build upon one another.

To show the generic approach of our software for solving abstract tasks with various robot platforms, we evaluate one challenging scenario for a two-dimensional mobile robot, and one even more difficult scenario for a joint robot with ten degrees of freedom in three-dimensional space. The latter scenario was also executed in a real environment to illustrate the feasibility of the whole process from modeling to planning, up to execution. Results reveal that for the second scenario, in 78% of the cases, even our non-hierarchical algorithm finds a solution in under 15 minutes, despite about 93% of the time being wasted on computations related to inverse kinematics and nearest neighbor search, which is another area of interest and can be separated from our work.

The contribution of this thesis is to give an overview of sampling-based manipulation planning, and to provide software and helpful implementation details, but also directions for future investigation to efficiently solve DAMA problems with various robot platforms in an environment featuring multiple objects.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Related Work . . . 1
1.3 Outline . . . 3

2 Background **5**
2.1 Robotics Foundations for Motion Planning . . . 7
2.1.1 Geometric Representations and Transformations . . . 7
2.1.2 Forward Kinematics and Denavit-Hartenberg Convention . 9
2.1.3 Workspace and Inverse Kinematics . . . 13
2.1.4 Configuration Space . . . 14
2.2 Sampling-Based Motion Planning Foundations . . . 18
2.2.1 Distance Metrics in C-Space . . . 19
2.2.2 Sampling in C-Space . . . 21
2.2.3 Collision Detection . . . 22
2.2.4 Paths in C-Space . . . 22
2.3 Sampling-Based Motion Planning Algorithms . . . 23
2.3.1 Rapidly-exploring Random Trees (RRTs) . . . 24
2.3.2 Probabilistic Roadmaps (PRMs) . . . 30
2.4 Multi-Modal Planning . . . 31

3 Diverse Action Manipulation **35**
3.1 Problem Description . . . 35
3.2 Diverse Action RRT Algorithm (DARRT) . . . 38
3.2.1 Empty Space Planner . . . 38
3.2.2 Sampling in the DAMA C-space . . . 42
3.2.3 Distance Metrics for the DAMA C-space . . . 43
3.2.4 DARRT Algorithm . . . 44
3.3 Advanced Diverse Action RRT Algorithms . . . 45
3.3.1 Bidirectional DARRT . . . 45
3.3.2 Hierarchical Approach . . . 47

4 Diverse Action Manipulation Experiments **51**
4.1 World 1: Simple Mobile Robot . . . 51
4.1.1 Scenario 1: Manipulation with Three Movable Objects . . 52
4.1.2 Discussion and Evaluation . . . 52

4.2 World 2: Joint Meka Robot JAMES 54
4.2.1 The JAMES Project 54
4.2.2 Robot Hardware 55
4.2.3 Implementation Details 56
4.2.4 Scenario 2: Rearranging Three Bottles 61
4.2.5 Discussion and Evaluation 61
4.2.6 Execution on real Meka Robot 65

5 Conclusion 69
5.1 Future Work 69

Bibliography 73

List of Figures

2.1 Various applications of motion planning 6
2.2 Yaw, pitch, and roll rotations 7
2.3 Kinematic 2-DOF chain of three rigid bodies (links) connected via two revolute joints 9
2.4 Definition of the four DH parameters 11
2.5 Basic motion planning problem 17
2.6 RRT algorithm exploring the C-space 25
2.7 Solving a motion planning query with RRT-based planning in a maze-like C-space 26
2.8 The PRM algorithm incrementally constructs the sampling-based roadmap 30
2.9 Illustration of multi-modal planning for hybrid systems 32
2.10 Example world for multi-modal planning illustrating modes and mode families. 33

3.1 Empty space planner for a simple 2D mobile robot 41

4.1 Scenario 1: Problem description and solution paths 53
4.2 Barkeeper JAMES acts as a bartender in a bar scenario 55
4.4 Meka robot model used for simulation 56
4.5 Reachable workspace of the Meka robot 57
4.6 Manipulation poses and their derived primitives for our Meka robot 59
4.7 Scenario 2: Problem description and a solution path (1/2) 62
4.8 Scenario 2: Problem description and a solution path (2/2) 63
4.9 Hardware environment for applying a DAMA problem on the real Meka robot 65
4.10 Snapshot of the Meka robot solving a DAMA problem in practice 66
4.11 A solution path for Scenario 2 executed in the real environment . 68

List of Tables

4.1 Averaged performance of DARRT and DARRTConnect on Scenario 1 52
4.2 Averaged performance of DARRTConnect and DARRTHConnect on Scenario 2 64

1 Introduction

1.1 Motivation

Imagine a robot assisting in a kitchen or within another similar environment featuring many objects and obstacles of different kind. Not only does it has to avoid undesired contact with objects and obstacles, it rather has to manipulate objects in different ways to accomplish a certain task. Concretely, in order for a kitchen robot to wash the dishes, it has to apply a broad range of diverse manipulation types: grasp the dishes at the right place, possibly push other objects away, slide along the dishes, and regulating the tap. Manipulation action can also depend on one another. For instance, an object could be too far away to be grasped immediately, but when pushed towards the robot with its finger tips, it can be grasped. Or for cutting vegetables, a knife has to be grasped beforehand, which serves as a tool for the robot.

These are examples of manipulation with diverse actions, in which the robot needs to plan for a sequence of actions to accomplish its given task. Compared to robots, humans are quite inventive, but also very experienced when it comes to diverse action manipulation. For instance, humans intuitively slide a piece of paper lying on the midst of a table towards the edge of the table, in order to subsequently grasp it. Today's robot system still mostly lack the ability to reason and manipulate with diverse actions.

This work contributes to solving Diverse Action Manipulation (DAMA) problems [6]. It builds on the success of sampling-based motion planning algorithms, like Rapidly-exploring Random Trees (RRTs) or Probabilistic Roadmaps (PRMs), and extends them to be suitable for solving DAMA problems in various robot environments. Another source of motivation is therefore to show the realization and integration of the entire process, from planning to execution, for a real robot.

1.2 Related Work

Barry [6] and the previously published major papers concerning manipulation with diverse actions [4, 5] lay the foundation for this work. They define the DAMA problem, in which the goal is to find a sequence of actions to move each object to a goal configuration, given a mobile robot, a set of movable objects, and a set of diverse manipulation actions. As a characteristic of the DAMA problem, there are so-called non-prehensile actions available, in which the robot is not rigidly attached to the object, as is the case when pushing an object for example. Moreover, the robot is equipped with and has to apply diverse manipulation actions to the same object, in order to reach its goal. The algorithms proposed by Barry [6] extend the sampling-based approach of a Rapidly-exploring Random Tree (RRT), which is one of the most famous algorithm for solving motion planning [36, 37, 27], to also solve for DAMA problems. The algorithms presented in this work will be closely related to the algorithms proposed by Barry. However, we implemented the algorithms completely on our own and made major variations concerning some implementation details, which are further described and discussed in Chapter 3.

Miyazawa et al. [39] solves the problem of determining a sequence of fingertip positions to move an object from start to goal configuration by graspless manipulation, like sliding of a cuboid with three degrees of freedom. He shows how RRTs can even plan for manipulation with complicated mechanics involved.

For manipulation tasks, contact points between the robot and objects define an important subspace of the overall configuration space. In order to plan for these accordingly, we need to model this *multi-modal* structure, in which usually each *mode* defines a fixed contact state. Hauser [16, 17] originally proposed the multi-model framework, which is nowadays accounted for in many manipulation scenarios, like grasp and regrasp operations, navigation among movable objects [8], or assembly planning [18]. Hauser applied it first to legged robots, in which the modes represent a fixed set of footfalls, which constrain the feet to be on the ground. His locomotion planner is based on probabilistic roadmaps (PRMs), a sampling-based motion planning algorithm similar to the RRT, and successfully plans for mode transitions, ultimately equipping the robot with the ability to walk. Hauser and Ng-Thow-Hing [18] then extend the mode space to continuous modes for a manipulation scenario, which allows a Honda humanoid robot to push an object to a goal location on a cluttered table, equipped with infinite pushing configurations. The robot can perform the actions walking, pushing, and reaching, which partitions the infinite modes into three mode families. Planning will be done in an 18-dimensional space, which is similar to our Meka robot scenario with 19 dimensions presented in this work. Hauser and Ng-Thow-Hing extend the PRM algorithm to an explicit multi-modal algorithm called Random-MMP algorithm, which is capable of generic multi-modal planning and will be similar to the algorithm presented in this work. However, as pointed out by Barry [6], explicit multi-modal algorithms cannot capture dependencies between mode families. For instance, if grasping an object is only applicable after pushing it to a certain position, modeling this dependency is not immediately provided within the multi-modal framework. Moreover, Hauser and Ng-Thow-Hing utilize some problem-specific domain knowledge and heuristics, which make the algorithm even harder to generalize. Therefore, in this work, we follow the more generic approach presented by Barry [6]. However, the problem of diverse action manipulation still is of multi-modal structure. From the perspective of multi-modality, each primitive or manipulation action of our DAMA problem defines a mode family and each mode within the family is defined by the specific robot-object contact and the location of all objects not being manipulated. We will further investigate multi-modal motion planning in Section 2.4 and discuss its implications in Chapter 3.

Throughout this work, we will further direct to some related resources and work done in the specific areas.

1.3 Outline

After presenting the motivation for this work and related material, Chapter 2 outlines crucial background knowledge for the following chapters. It starts with an introduction to planning algorithms in general, and continues with robotics foundations for motion planning, foundations of sampling-based motion planning, and further examines two popular sampling-based motion planning algorithms and the multi-modal planning framework.

Build upon this background knowledge, Chapter 3 defines the Diverse Action Manipulation (DAMA) problem and presents multiple planning algorithms, which extend the previously discussed sampling-based motion planning algorithms from Chapter 2 to solve for DAMA problems.

In Chapter 4, we define different worlds and scenarios, in which we can perform experiments to test the DAMA algorithms proposed in Chapter 3. We will evaluate these on challenging scenarios and show how to step from simulation and planning to real world execution of solutions to DAMA problems with a high dimensional joint robot.

Finally, in Chapter 5, we summarize and discuss our work and present some future work.

2 Background

Let us first take a brief glance at the history of planning in general, as it comes a long way.

In his famous book *Planning Algorithms*, LaValle [31] outlines how the research area of planning (especially motion planning) emerged from and was mostly influenced by the research in the fields of robotics, artificial intelligence and control theory. Although all three areas now share common ground concerning motion planning, they approached it in a very different manner. Robotics focused primarily on collision-free movements of objects while considering uncertainties and dynamics of the environment, resulting in a list of transformations required to achieve a certain goal. However, artificial intelligence focused mainly on solving discrete planning problems, having a discrete set of states and actions and searching for a valid sequence of actions to transform the start state into a goal state. Moreover, control theory focused on finding open-loop trajectories for a nonlinear dynamical system in order to transition from the start state to a goal state, while dealing with issues such as optimality, feedback and stability.

The range of applications for motion planning is broadly diversified [31], among them are interesting areas like making smart video game characters, solving sliding puzzles, developing parking cars, enabling robots to seal cracks in automotive assembly, navigate autonomous helicopters through obstacles, and countless more fascinating applications (Figure 2.1).

An important component of a planning problem is the *state space*. The state space contains all possible states of the respective planning problem including a certain representation of these. In this thesis, we will focus on motion planning problems in continuous state spaces as they arise in robotics, where states are often defined by the position and orientation of a robot. We will later come back to a precise definition of our state space representation. For now, let us also note that planning requires a certain initial state and the definition of a goal state, or even a set of goal states. The problem of motion planning is to then find a trajectory from the initial to a goal state without collisions.

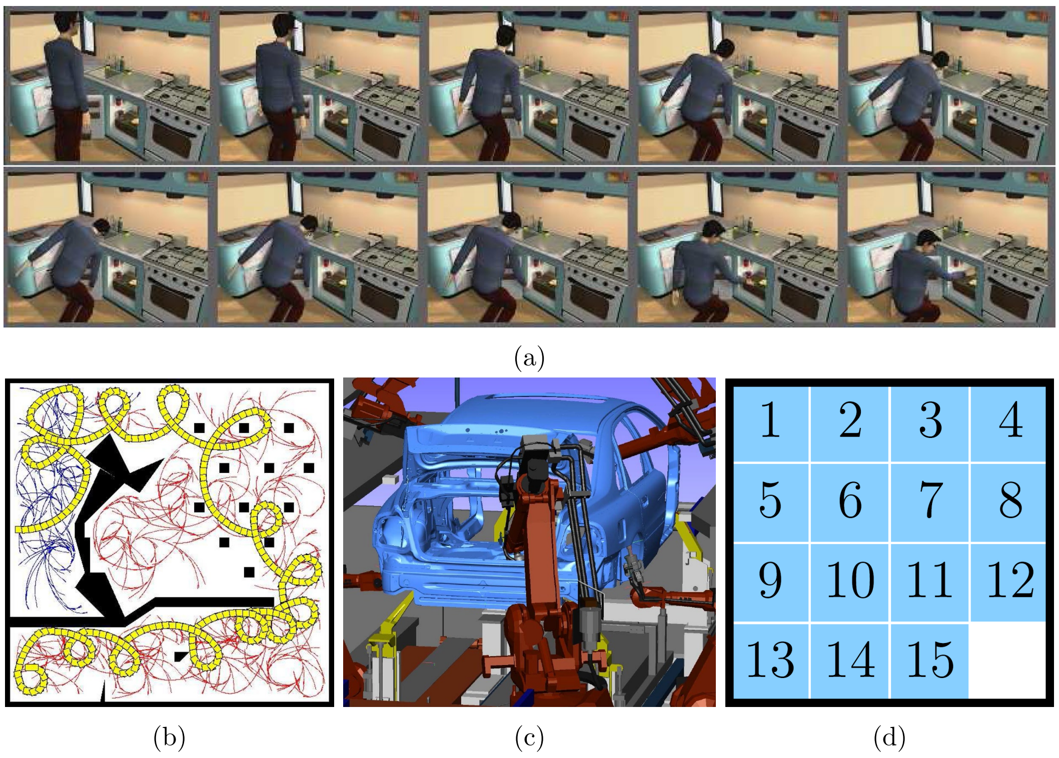


Figure 2.1: **Various applications of motion planning [31, pp. 5-16]**. (a) Animation sequence for a digital actor within a kitchen scenario. (b) Non-holonomic planning for a car avoiding obstacles with differential constraints only being able to move forward and to turn left. (c) Simulation of robots sealing cracks in automotive manufacturing at the Volvo Cars assembly plant in Torslanda, Sweden. (d) Sliding puzzle as an example of discrete planning, in contrast to continuous planning.

2.1 Robotics Foundations for Motion Planning

In this section, we will address foundations of robotics, which are required or quite useful to know when solving motion planning problems. We will start with basic geometric representations and transformations in a 3D world, then derive the forward kinematics of a robot using the Denavit-Hartenberg notation, briefly introduce the notion of a workspace and the inverse kinematics problem, and conclude with explaining the concept of configuration spaces.

2.1.1 Geometric Representations and Transformations

For sampling-based planning methods, which we will discuss later in this work, a *collision detection algorithm* has to decide whether a certain configuration is collision-free or not. For this, we do require a systematic way of representing components geometrically and computationally efficient, in order to deal with lots of collision detection queries. Fortunately for us, in this work, the *Robotics Library* [2] provides us with an implementation of a collision detection algorithm using semi-algebraic models to geometrically represent objects and check for collisions. Thus, we will treat the actual collision detection algorithm as a black box, as its approach will not be the focus of this theses. Nevertheless, in order to appropriately utilize its interfaces defined by various file formats used in the Robotics Library, we need to have basic understanding of geometric representations and transformations.

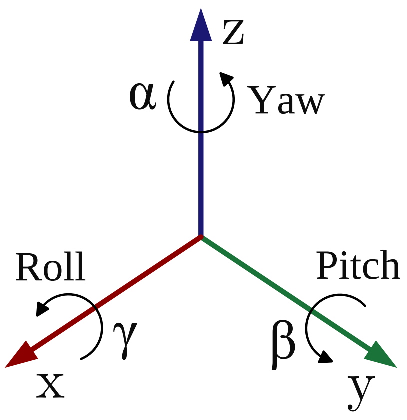


Figure 2.2: **Yaw, pitch, and roll rotation.** Any rotation can be described as a series of axis rotations.

A collision detection algorithm requires both the obstacle region and the robot to be expressed in terms of the *world frame* of $\mathcal{W}$, i.e., sharing the same origin and coordinate basis vectors of $\mathcal{W}$. The obstacle region is fixed and defined within $\mathcal{W}$, and covers all the points, which lie in at least one obstacle. In contrast, the robot usually consists of multiple (rigid) bodies, in which each body is expressed in its own *body frame*. Now that the robot can move within the world, a *rigid-body transformation* h places the robot in $\mathcal{W}$ at a certain position and orientation. At first we will take a look at the basic 3D transformation for a single rigid body, before we will cover transforming a chain of attached rigid bodies.

A rigid-body transformation consists of a translational and a rotational part. The translation by $\mathbf{t} = \begin{pmatrix} t_x & t_y & t_z \end{pmatrix}^\top \in \mathbb{R}^3$ is rather simple:

$$(x, y, z) \mapsto (x + t_x, y + t_y, z + t_z). \tag{2.1}$$

Describing the rotational part in the 3D transformation requires more effort. Any rigid body rotation in 3D space can be described as a series of rotations around the three orthogonal axes, as shown in Figure 2.2. These three angles are called yaw, pitch and roll, respectively, and form the so-called *fixed angle representation*. Now in order to rotate arbitrarily in space, we can compose a single rotation matrix by chaining the roll, then the pitch, and then the yaw rotation as follows:

$$\begin{aligned} &R(\alpha,\beta,\gamma) = R_z(\alpha)\cdot R_y(\beta)\cdot R_x(\gamma) \\ &= \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{pmatrix} \\ &= \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{pmatrix}. \end{aligned} \tag{2.2}$$

We can combine the rotational part with the translational one by increasing the dimension by one using homogeneous coordinates, which yields the *homogeneous transformation matrix* for 3D bodies:

$$\begin{aligned} T &= \begin{pmatrix} R(\alpha,\beta,\gamma) & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \\ &= \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & t_x \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & t_y \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \end{aligned} \tag{2.3}$$

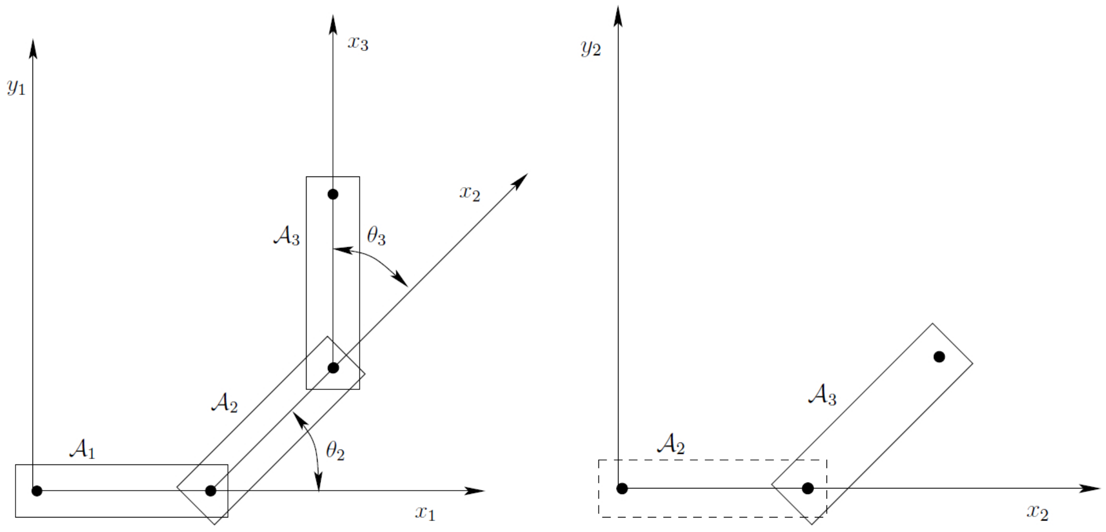


Figure 2.3: **Kinematic 2-DOF chain of three rigid bodies (links) connected via two revolute joints**. $\mathcal{A}_3$ is described in $\mathcal{A}_1$'s body frame via the transformation ${}^{1}T_{2}{}^{2}T_{3}$ (left) and $\mathcal{A}_3$ is described in $\mathcal{A}_2$'s body frame via the transformation ${}^{2}T_{3}$ (right) [31, p. 104].

Note that while T has six degrees of freedom (DOF), the order of operations is crucial: At first we apply the roll rotation, then the pitch, then the yaw, and finally the translation by $\mathbf{t}$. Thus, T represents a rotation followed by a translation.

2.1.2 Forward Kinematics and Denavit-Hartenberg Convention

Having introduced basic knowledge about transformations, let us now discuss modeling traditional robot system, which consist of multiple rigid bodies [31, 41, 21].

Rigid bodies are connected by joints to form a kinematic chain, whereby each active joint is controlled by an actuator. Note that in this section we will only discuss robot systems with a single kinematic chain of rigid bodies. Of course, the architecture of a robot system can be more sophisticated, resulting for example in kinematic trees with multiple end-effectors, closed kinematic chains (featuring loops), links attached to more than two joints, or nonrigid bodies. Most of these architectures are discussed in [31, 21].

For a robot with m rigid bodies (also referred to as links), let $\mathcal{A}_i$ with $1 \leq i < m$ denote the i-th link attached to the following link $\mathcal{A}_{i+1}$. $\mathcal{A}_1$ is also referred to as the *base* and $\mathcal{A}_m$ as the robot's *end-effector*, which usually has some kind of manipulator attached, for example a gripper, and thus describes the robot's *tool frame.*

Each joint in between two links provides some constrained motion for $\mathcal{A}_i$ with respect to $\mathcal{A}_{i+1}$. There are many possible types of joints used to attach bodies, among them are two most common joints, which both provide the robot system with one additional degree of freedom (DOF): *revolute* joints and *prismatic* joints. Whereas a revolute joint introduces rotational motion of one link with respect to the other, a prismatic joint allows one link to slide along the other providing translational motion.

Figure 2.3 illustrates an example of a kinematic chain with three rigid bodies $\mathcal{A}_1, \mathcal{A}_2$ and $\mathcal{A}_3$ connected via two revolute joints in between. Altogether this system needs eight independent parameters to be described entirely within the world frame $\mathcal{W}$: six parameters for the position and orientation of the base frame $\mathcal{A}_1$ and two more joint parameters θ_2 and θ_3 induced by the revolute joints. Usually, when specifying the degrees of freedom (DOF), we only consider the independent parameters of the system itself induced by the joints, resulting in a 2-DOF system in our example.

Recall that each body of our robot is expressed in its own body frame $\mathcal{A}_i$, which also determines the axis of rotation, as visualized in Figure 2.3. Now in order to determine the location of each link we first construct a homogeneous transformation matrix ${}^{i-1}T_i$ for each body $\mathcal{A}_i$ with $1 \le i < m$, which expresses the transformation from body $\mathcal{A}_i$ to body $\mathcal{A}_{i-1}$, respectively to the world frame $\mathcal{W} = \mathcal{A}_0$ for $i = 1$. Thus, applying ${}^1T_2$ to $\mathcal{A}_2$ describes its points within the body frame of $\mathcal{A}_1$. For example in Figure 2.3, ${}^2T_3$ causes a rotation of θ_3 and a certain translation by some constant. Chaining the application of the homogeneous transformation matrices we can then easily compute the location of any point $(x, y, z) \in \mathcal{A}_m$ within the world frame $\mathcal{W}$ as follows:

$$ {}^0T_1 {}^1T_2 \cdots {}^{m-1}T_m \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} . \tag{2.4}$$

Such an analysis is called *forward kinematics*, where ultimately we are interested in expressing the end-effector configuration, respectively the tool frame, within the world frame or at least within the base frame of the robot system.

Now in order to construct each ${}^{i-1}T_i$, we make use of the *Denavit-Hartenberg (DH)* method [31, 41, 21]. At first, we carefully define the link coordinate frame for each $\mathcal{A}_i$:

- z_i-axis is placed along the axis of joint i connecting body $\mathcal{A}_{i-1}$ and $\mathcal{A}_i$.
- x_i-axis is placed along the common normal of the z_i and z_{i+1} axes, directed

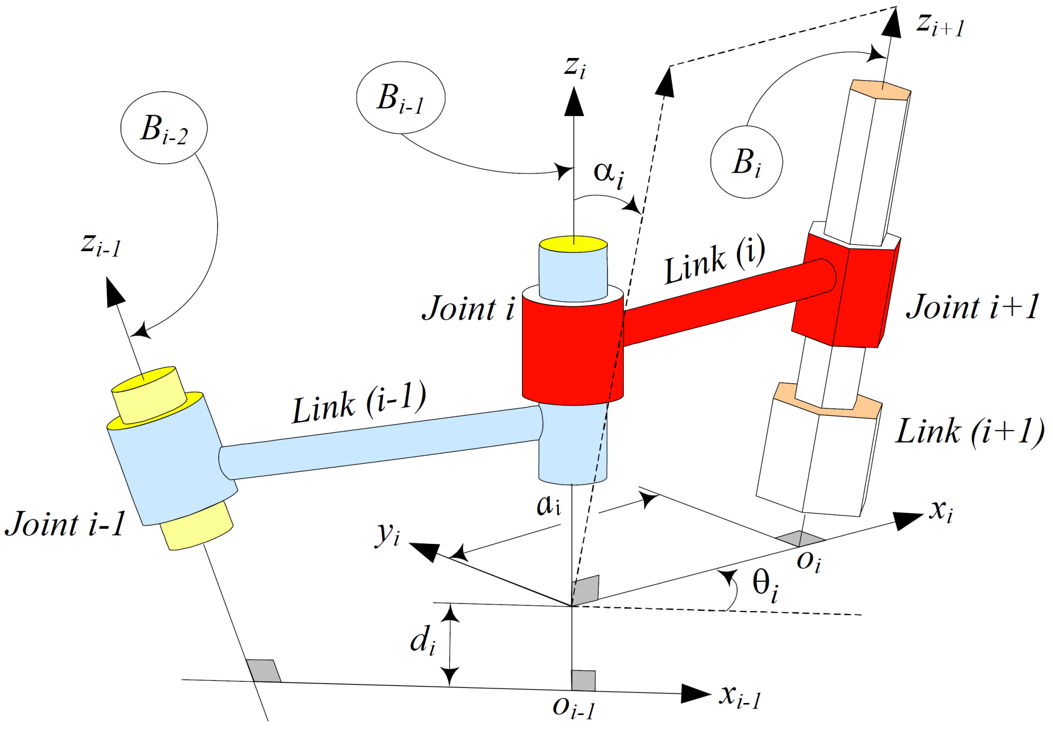


Figure 2.4: **Definition of the four DH parameters**. a_i, α_i, d_i, θ_i are defined for the respective joint i, link (i), and their involved body frames B_i and B_{i-1} [21, p. 258].

towards the z_{i+1}-axis.

- y_i-axis is placed optionally in order to obtain a right-handed coordinate frame.

This method yields four DH parameters $a_{i-1}, \alpha_{i-1}, d_i, \theta_i$ to fully express each transformation ${}^{i-1}T_i$, with definition as follows:

- a_i: distance between the z_i and z_{i+1} axes along the x_i axis
- α_i: angle from z_i to z_{i+1} axes about the x_i axis
- d_i: distance between the x_{i-1} and x_i axes along the z_i axis
- θ_i: angle from the x_{i-1} and x_i axes about the z_i axis

Figure 2.4 further illustrates the DH convention. For a revolute joint i, θ_i represents the controllable *joint variable*, for a prismatic joint it would be d_i. All the other three parameters are usually constant and depend on the structure of the robot. Note that according to Jazar[21], our convention follows the non-standard Denavit-Hartenberg notation, where each body frame $\mathcal{A}_i$ was set at joint i instead of the distal joint $i+1$. However, this has the advantage of θ_i really rotating around the z_i-axis, resulting in an equal number of joints and coordinate frames. However, the homogeneous transformation matrix ${}^{i-1}T_i$ becomes a bit more complicated concerning the indexes.

Finally, for each i such that $1 < i \leq m$, the DH parameters define our homogeneous transformation matrix ${}^{i-1}T_i$:

$$
{}^{i-1}T_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i\cos\alpha_{i-1} & \cos\theta_i\cos\alpha_{i-1} & -\sin\alpha_{i-1} & -d_i\sin\alpha_{i-1} \\ \sin\theta_i\sin\alpha_{i-1} & \cos\theta_i\sin\alpha_{i-1} & \cos\alpha_{i-1} & d_i\cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.5}
$$

Thus, in order to transform from $\mathcal{A}_i$ to $\mathcal{A}_{i-1}$, ${}^{i-1}T_i$ applies the following four operations in succession:

1. Translate by d_i along the z_i-axis
2. Rotate counterclockwise by θ_i about the z_i-axis
3. Translate by a_{i-1} along the x_{i-1}-axis
4. Rotate counterclockwise by α_{i-1} about the x_{i-1}-axis

Note that ${}^{0}T_{1}$ maps the base frame to the world frame $\mathcal{W}$ and is usually not modeled using the DH notation. Instead, we just apply the general rigid-body homogeneous transformation as defined in Equation 2.3.

2.1.3 Workspace and Inverse Kinematics

For a robot system, the *workspace* describes the total volume of space which is accessible by the end-effector. Jazar [21] differentiates between *reachable* and *dexterous* workspace. Whereas the reachable workspace is the volume of space in which every point is reachable by the end-effector in at least one orientation, the dexterous workspace further constrains the space to only contain points which are reachable in all possible orientations. Note that revolute joints are often preferred over prismatic joints as they provide a larger dexterous workspace.

With forward kinematics using the DH notation we can compute the end-effector configuration of a robot controlled in joint space. In this work specifically, and for robots performing manipulation in general, we also need to control the end-effector within its workspace, which is usually a Cartesian space. In other words, given a certain position and orientation of the end-effector configuration in Cartesian space, we need to compute the set of joint variables that positions the robot accordingly. This problem is called the *inverse kinematics* and unfortunately, due to its highly nonlinear nature, is much more difficult to solve than the forward kinematics problem. Jazar [21] gives a structured overview of the inverse kinematics problem and introduces some of the different approaches to solve it. A detailed introduction to inverse kinematics is not the focus of this section. However, note that in this work, we adapted the iterative technique based on the Newton-Raphson method, which is offered by the *Robotics Library* [2] to solve inverse kinematics. The adaptations we made are further explained later on, when we present our planning algorithm applied to the Meka robot. For now, it is also worth mentioning that the iterative technique does not always converge, and can give a poor performance near singular and degenerate configurations, thus, when there are less DOF available for the end-effector than is the dimensional number of the space in which it operates. Moreover, there can be multiple joint configurations representing the same point in Cartesian space, but also no solution, if the end-effector configuration is outside the workspace or some joint limits were reached.

2.1.4 Configuration Space

So far we discussed modeling and transforming a collection of bodies. Now we will come back to motion planning problems which usually search within a well-defined continuous state space. For this, we need to define the state space, the set of all possible rigid-body transformations that could be applied to the robot [31]. This state space is referred to as the *configuration space* or *C-space*, as it describes all configurations of the system. As LaValle [31] states, the C-space is a powerful abstraction from complicated models and transformations in order to solve the general problem of computing a path that traverses a manifold[1].

Before we discuss how to describe obstacles within our C-space, let us first analyze the general C-space for 2D and 3D rigid bodies. We will start with describing a single rigid body, before we continue with multiple or chains of rigid bodies.

2.1.4.1 Configuration Space for a single rigid body

In 2D, the C-space is composed of a translational (x, y) and a rotational part (θ), resulting in the following C-space with 3 dimensions: $\mathcal{C} = \mathbb{R}^2 \times \mathbb{S}^1$. Whereas the translation can be described within the space $\mathbb{R}^2$, the rotational part is homeomorphic to the 1D manifold $\mathbb{S}^1 = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$, which describes a circle. The latter case induces wraparounds in our C-space, because traveling in the same direction on a circle results in visiting the same configuration multiple times. Knowing of wraparounds is crucial for a motion planning algorithm as they may lead to new solutions.

Before we discuss the 3D C-space, let us first introduce two important groups for motion planning: the *special orthogonal group* $SO(n)$ and the *special Euclidean group* $SE(n)$. Whereas the special orthogonal group $SO(n)$ describes all $n \times n$ rotation matrices[2], combining rotation with translation leads to the special Euclidean group $SE(n)$ of all $(n + 1) \times (n + 1)$ transformation matrices:

$$SE(n) = \left\{ \begin{pmatrix} R & v \\ 0 & 1 \end{pmatrix} \middle| R \in SO(n), v \in \mathbb{R}^n \right\}. \tag{2.6}$$

Note that while $SO(2)$ is homeomorphic to $\mathbb{S}^1$, $SE(2)$ is homeomorphic to the 2D C-space $\mathbb{R}^2 \times \mathbb{S}^1$. In general, $SE(n)$ is always homeomorphic to the n-dimensional

[1]Please refer to LaValle (Section 4.1) [31] for an introduction to basic topological concepts, including the definition of a topological manifold and the relation to the C-space, which will be quite helpful for understanding motion planning in general.

[2]More technically, $SO(n)$ is the set of all (nonsingular) $n \times n$ real-valued orthogonal matrices with determinant 1.

C-space $\mathbb{R}^n \times SO(n)$.

Let us now discuss the architecture of the 3D C-space $SE(3)$. Whereas the translation easily yields $\mathbb{R}^3$, determining the topology of $SO(3)$ is significantly more complicated than its 2D counterpart. Recall from Equation 2.2 that we can describe a rotation in 3D space using axis angle representation with only three parameters. Although being good practice for describing a single rotation, this representation can cause serious problems when chaining multiple rotations, as different yaw, pitch, and roll values can yield the same rotation matrix. Visually speaking this refers to a *gimbal lock*, which occurs when two of the three axes are driven into a parallel configuration, resulting in a degenerated 2D space loosing one DOF.

Fortunately there are various ways to parametrize 3D rotations, one of the most convenient ones in robotics are *unit quaternions* [21, 41, 31]. Unit quaternions avoid the gimbal lock problem utilizing four parameters and the condition of unit length to represent a rotation, thus still resulting in three DOF. They are simple to compose, much more numerically stable and need fewer operations than using rotation matrices with their nine parameters. Let $\mathbb{H}$ be the set of all quaternions, each quaternion $h \in \mathbb{H}$ is defined as

$$h = h_0 + h_1 i + h_2 j + h_3 k \tag{2.7}$$

with the constraint $i^2 = j^2 = k^2 = ijk = -1$. Thus, the four-dimensional quaternion consists of a real part or scalar $h_0 \in \mathbb{R}$, and a vector $(h_1, h_2, h_3) \in \mathbb{R}^3$ describing the imaginary components of the quaternion. For unit quaternions needed to describe $SO(3)$, the norm $\|h\|$ has to be 1, yielding the constraint $h_0^2 + h_1^2 + h_2^2 + h_3^2 = 1$. The set of all unit quaternions can be visualized as a unit sphere in $\mathbb{R}^4$, also denoted as the $\mathbb{S}^3$ space.

A unit quaternion h can be mapped to a rotation matrix (and vice versa, but not further outlined here) and thus to $SO(3)$ in the following way [31]:

$$R(h) = \begin{pmatrix} 2(h_0^2 + h_1^2) - 1 & 2(h_1 h_2 - h_0 h_3) & 2(h_1 h_3 + h_0 h_2) \\ 2(h_1 h_2 + h_0 h_3) & 2(h_0^2 + h_2^2) - 1 & 2(h_2 h_3 - h_0 h_1) \\ 2(h_1 h_3 - h_0 h_2) & 2(h_2 h_3 + h_0 h_1) & 2(h_0^2 + h_3^2) - 1 \end{pmatrix} \tag{2.8}$$

However, we can directly transform a point $(x, y, z) \in \mathbb{R}^3$ using a unit quaternion h by first building its *conjugate* $h^* = h_0 - h_1 i - h_2 j - h_3 k$ and the quaternion $p \in \mathbb{H}$ containing the point as $p = 0 + xi + yj + zk$. The rotated quaternion $\overline{p}$ is then given by:

$$\overline{p} = h \cdot p \cdot h^* \tag{2.9}$$

We can then extract the rotated point $(\overline{x}, \overline{y}, \overline{z}) \in \mathbb{R}^3$ from the imaginary components of $\overline{p}$ the same way we constructed p.

Note that when using unit quaternions, we have to consider that h and $-h$ represent the same rotation, as $R(h) = R(-h)$. These points are referred to as *antipodal* or polar opposite points. According to LaValle, one way to force uniqueness of the rotation representation is to require that for all parameters $h_i,\ 0 \leq i \leq 3$, the constraint $h_i \geq 0$ holds, further providing an appropriate mapping to the upper ("northern") hemisphere for paths traveling across the equator of $\mathbb{S}^3$ [31, 32]. The resulting space is referred to as a *3-dimensional real projective space*, denoted $\mathbb{RP}^3$, yielding the six dimensional manifold or C-space for a 3D robot: $\mathcal{C} = \mathbb{R}^3 \times \mathbb{RP}^3$.

2.1.4.2 Configuration Space for a chain of rigid bodies

In this section we will briefly discuss describing the C-space for multiple rigid bodies connected by revolute or prismatic joints forming a kinematic chain. Assuming a robot with a chain of n bodies, let $\mathcal{C}_i$ denote the C-space of body $\mathcal{A}_i$ for $1 \leq i \leq n$.

Each revolute or prismatic joint contributes one degree of freedom to the robot system. Whereas a prismatic joint contributes $\mathbb{R}^1$, we have to distinguish between two cases for a revolute joint. In practice, a revolute joint might not be able to take on any $\theta \in [0, 2\pi)$, thus not providing the ability to wrap-around. In this case, the revolute joint also contributes $\mathbb{R}^1$, however, with the full range of motion a revolute joint contributes $\mathbb{S}^1$. For example, a robot with five rigid bodies, connected by one prismatic joint, two restricted and two revolute joints would have the following combined C-space:

$$\mathcal{C} = \mathbb{R}^1 \times \mathbb{R}^1 \times \mathbb{R}^1 \times \mathbb{S}^1 \times \mathbb{S}^1 = \mathbb{R}^3 \times \mathbb{S}^1 \times \mathbb{S}^1 \tag{2.10}$$

Note that depending on the level of description, we also need to add the possible transformations for the first body $\mathcal{A}_1$.

2.1.4.3 Configuration Space with obstacles

Motion planning would be trivial without having obstacles or the possibility for a robot system to collide. As planning takes place in the C-space, we have to define the obstacle region $\mathcal{C}_{obs}$ as part of the C-space.

For a robot with multiple bodies, a configuration of the robot $q \in \mathcal{C}$ can be contained in $\mathcal{C}_{obs}$ if either at least one link collides with an (external) obstacle, or some specified links of the robot collide with each other [31].

Let $\mathcal{O} \subset \mathcal{W}$ denote the (external) obstacle region as a subset of the world $\mathcal{W} = \mathbb{R}^3$, recall the definition of the m bodies $\mathcal{A}_i \subset \mathcal{W}$ with $1 \leq i \leq m$ of our robot, and let $\mathcal{A}_i(q)$ denote the transformed link i with respect to q. Furthermore let P denote the set of pairs (i, j) with $1 \leq i, j \leq m$, where each pair will cause a collision if $\mathcal{A}_i(q)$ and $\mathcal{A}_j(q)$ overlap. P has to be defined individually for each robot depending on its architecture and does not simply contain all pairs of links, as adjacent links are in contact all the time. For a robot with a kinematic chain, P usually contains all pairs of links which are not connected by a joint.

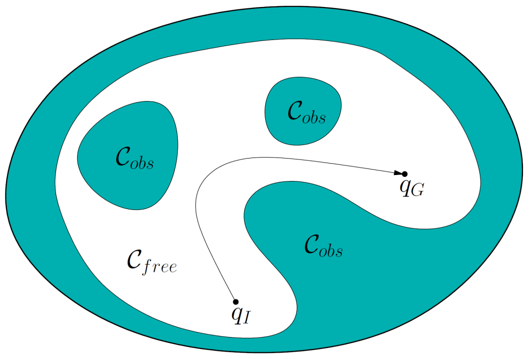


Figure 2.5: **Basic motion planning problem**. The C-space is composed as $\mathcal{C} = \mathcal{C}_{free} \cup \mathcal{C}_{obs}$ and we need to find a path from q_{init} to q_{goal} traversing only within $\mathcal{C}_{free}$ [32, p. 3].

With this notation we can now define the obstacle region $\mathcal{C}_{obs} \subseteq \mathcal{C}$ for both cases described previously [31]:

$$\mathcal{C}_{obs} = \left(\bigcup_{i=1}^{m} \{q \in \mathcal{C} \mid \mathcal{A}_i(q) \cap \mathcal{O} \neq \emptyset\} \right) \bigcup \left(\bigcup_{[i,j] \in P} \{q \in \mathcal{C} \mid \mathcal{A}_i(q) \cap \mathcal{A}_j(q) \neq \emptyset\} \right). \tag{2.11}$$

Finally we can formulate our *basic motion planning problem*, originally known as the *Piano Mover's Problem* [31, 32]. Assume a world $\mathcal{W} = \mathbb{R}^3$, an obstacle region $\mathcal{O} \subset \mathcal{W}$, a rigid robot $\mathcal{A}$ with m links each defined in $\mathcal{W}$, the C-space $\mathcal{C}$ capturing all possible robot transformations, and a definition of $\mathcal{C}_{obs}$. The goal is to find a path starting from a certain *initial configuration* $q_{init} \in \mathcal{C}_{free}$ and finishing in a *goal configuration* $q_{goal} \in \mathcal{C}_{free}$ while completely traversing within the *free space* $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$. More precisely, we want to compute the continuous path $\tau : [0, 1] \to \mathcal{C}_{free}$ with $\tau(0) = q_{init}$ and $\tau(1) = q_{goal}$. Figure 2.5 illustrates this problem conceptually.

However, this problem is PSPACE-hard (and thus NP-hard), and in practice neither $\mathcal{C}_{free}$ nor $\mathcal{C}_{obs}$ can be either constructed, modeled or represented both explicitly and efficiently [31, 32]. In the next two sections we will discuss how to solve this problem.

2.2 Sampling-Based Motion Planning Foundations

In order to solve the basic motion planning problem and to make the continuous C-space tangible, we have to somehow discretize it [32]. There are two main methods for developing an algorithmic solution: *combinatorial planning* and *sampling-based planning.*

Combinatorial planning tries to explicitly and exactly represent and capture the structure of the C-space. While focusing on the representation of the C-space, they construct a roadmap along the way, utilizing properties of the current problem instance. For certain classes of planning problems, for example convex models, low dimensionality, restricted motion, combinatorial planning can yield an elegant and efficient solution. Furthermore, a combinatorial algorithm provides *completeness*, defined as follows: For any input in finite time, the algorithm will either return a solution, or it will correctly report that there is none. Although combinatorial planning might be ideal in some settings, in most practical cases a combinatorial explosion results in numerical issues and inefficiency making the given problem impossible to solve [32].

Sampling-based planning abandons the idea of explicitly characterizing the C-space. Instead, it utilizes a collision detection algorithm to incrementally probe the C-space by determining if a certain configuration sample lies in either $\mathcal{C}_{free}$ or $\mathcal{C}_{obs}$. From these samples a roadmap is constructed exploring the C-space. In contrast to combinatorial planning, the collision detection algorithm is decoupled from the planner itself, separating it from the particular geometric and kinematic models. Sampling-based planning may not be as elegant as the combinatorial approach for some problem classes, but for most problems in practice they are most widely used and offer efficient solutions, even for high-dimensional C-spaces and systems with differential constraints. However, sampling-based planners can usually only guarantee *probabilistic completeness*, which is defined as follows: Given enough time, the probability that the algorithm will find a solution converges to one if a solution exists. The algorithm cannot determine if no solution exists. Moreover, the rate of convergence is usually very difficult to measure, according to LaValle [32], who also gives a well-structured introduction to motion planning.

2.2.1 Distance Metrics in C-Space

As sampling-based planning algorithms approach the exploration of the C-space by samples, they need a function to calculate the distance between two configuration points. Kuffner [28] points out that the ideal metric for robot path planning would correspond to the minimum swept-volume containing all workspace movements between both configurations, as minimizing the swept-volume maximizes the chance of traveling collision-free through $\mathcal{C}_{free}$. However, he also notes that this exact measure is not computable in practice, and that we have to apply heuristic approaches to approximate this metric. One heuristic approach is to use some distance measure in joint space, for example summing up the joint differences between both configurations. Another heuristic approach, which we will examine shortly, is to compare the pose of the end-effector in the C-space $SE(3)$. Therefore, we need to construct an appropriate *metric space* for $\mathcal{C}$, with focus on the C-space $SE(3)$.

Let us briefly introduce the concept of a metric space. A metric space (X, ρ) consists of a topological space X (which will be our C-space) and a distance function $\rho : X \times X \to \mathbb{R}$ following the four conditions of non-negativity, reflexivity, symmetry, and triangle inequality [31]. The most important metric over the vector space $\mathbb{R}^n$ might be the *Euclidean metric* with the *Euclidean distance* and the *Euclidean norm* defined as follows, with $x, x' \in X$:

$$\rho(x, x') = \sqrt{\left(\sum_{i=1}^{n} |x_i - x'_i|^2\right)} = \|x - x'\|. \tag{2.12}$$

Is a topological space Z composed of two metric spaces (X, ρ_x) and (Y, ρ_y), a metric space (Z, ρ_z) can be constructed by defining ρ_z as

$$\rho_z(z, z') = c_1 \rho_x(x, x') + c_2 \rho_y(y, y'), \tag{2.13}$$

with $x, x' \in X$, $y, y' \in Y$, $z = (x, y) \in Z$, $z' = (x', y') \in Z$, and two positive real constants $c_1 > 0$, $c_2 > 0$, which weight their respective subspace and have to be chosen individually. This concept is crucial for us, as we saw previously that in case of multiple bodies our C-spaces are composed of multiple spaces.

Now let us construct a metric for $SE(3)$. Recall from Equation 2.6 that $SE(3)$ is composed of $\mathbb{R}^3$ and $SO(3)$, thus we first need a metric for both of these spaces before we can combine them using Equation 2.13. For the vector space $\mathbb{R}^3$ we use the already introduced Euclidean metric defined by Equation 2.12. Again, finding an appropriate metric for the rotational part $SO(3)$ is more complicated. Huynh [20] presents a detailed comparison and analysis of metrics for 3D rotations.

Recall that unit quaternions are used to describe the 3D rotational space SO(3). We referred to this space as a real projective space, a subset of $\mathbb{R}^4$ taking antipodal points into account, visualized as a unit sphere $\mathbb{S}^3$, where the points lie on the surface of the sphere. It turns out that the scalar inner product of two unit quaternions is a simple but convenient way to measuring a proper geodesic path or distance along $\mathbb{S}^3$, as it is equal to $\cos\theta$ with θ describing the angle between both vectors. Thus, the distance between two unit quaternions $h, h' \in \mathbb{R}^4$ (defined in Equation 2.7) for $\mathbb{S}^3$ can be computed as follows:

$$\rho_{\mathbb{S}^3}(h, h') = \cos^{-1}(h \cdot h') = \cos^{-1}(h_0 h'_0 + h_1 h'_1 + h_2 h'_2 + h_3 h'_3). \tag{2.14}$$

Transitioning from $\mathbb{S}^3$ to $SO(3)$ we just have to take antipodal points into consideration, in which the distance is calculated as the minimum of the distance h to h' and the distance from h to h''s antipodal version $-h'$:

$$\rho_{SO(3)}(h, h') = \min\{\rho_{\mathbb{S}^3}(h, h'), \rho_{\mathbb{S}^3}(h, -h')\}. \tag{2.15}$$

Following Kuffner [28] we can simplify it further and compute a scalar approximate distance measure more efficient to compute as

$$\rho_{SO(3)'}(h, h') = 1 - \|h \cdot h'\|, \tag{2.16}$$

in which we replace $\cos^{-1}$ with the function $1-x$, both sharing the same monotony and the same zero point at $x = 1.0$. Furthermore, the calculation of the minimum distance is encapsulated within $\|h \cdot h'\|$, transforming the range of values from the dot product from $[-1, 1]$ to [0,1]. Thus, the range of values returned by $\rho_{SO(3)'}$ lies nicely between 1 and 0.

However, combining the positional with the rotational space unfortunately leaves constants to be defined, as mentioned earlier and introduced with Equation 2.13. This is unavoidable, as we somehow have to relate the distance in the plane to angular distance, which also might be measured in either rad or degree. Both quantities are completely incompatible, and weighting them accordingly also depends on the planning problem to be solved.

2.2.2 Sampling in C-Space

Sampling-based planners sample the C-space, thus they require a function to incrementally generate such samples.

A sampling function should produce a dense sequence of samples to get arbitrarily close to each configuration in the C-space. Considering a uniform distribution, it prevents us from oversampling or undersampling large C-space regions, thus significantly boosting the performance of our sampling-based planner. Furthermore, as our planner can only operate on a finite number of samples, the order of the samples within our sample sequence also has a major impact on the planner's performance.

The easiest and most straight-forward approach for defining a sampling function is to follow the *uniform random* method. Given a pseudo-random number generator function $rand()$, which returns uniformly distributed real numbers on the scale of $[0, 1)$, we can easily create a random sample for $\mathbb{R}^n$. For example, as outlined in Kuffner [28], the translation component of $SE(3)$ can be sampled by sampling and scaling along each axis:

$$(x, y, z) = (X_{dim}rand(), Y_{dim}rand(), Z_{dim}rand()). \tag{2.17}$$

This illustrates how nicely random samples extend for composed C-spaces. Assuming the abstract C-space $Z = X \times Y$, we can combine the uniform independent random samples x and y taken from X and Y, respectively, to create a uniform random sample (x, y) for the C-space Z.

In order to obtain uniform sampling over the rotation component of $SE(3)$, or $SO(3)$, we have to sample with a uniform probability density over the unit sphere $\mathbb{S}^3$. Kuffner [28] and LaValle [31] refer to a clever and efficient way to generate a uniformly-distributed random unit quaternion h as

$$h = (\sqrt{1-u_1}\sin 2\pi u_2,\ \sqrt{1-u_1}\cos 2\pi u_2,\ \sqrt{u_1}\sin 2\pi u_3,\ \sqrt{u_1}\cos 2\pi u_3), \tag{2.18}$$

in which $u_1, u_2, u_3 \in [0, 1)$ are random numbers, generated by our function $rand()$.

Note that uniform random sampling has also disadvantages, which emerge as the underlying pseudo-random number generator usually cannot guarantee true randomness. Instead, it only simulates its behavior, resulting in some undesired irregularities. An alternative approach is *deterministic sampling*, such as low-dispersion sampling, in which we minimize the largest uncovered area, or lowdiscrepancy sampling. Both methods are further described in LaValle [31], however, in this work, we will not pursue deterministic sampling.

2.2.3 Collision Detection

For each sample, the sampling-based planner has to decide whether that certain configuration causes a collision or not. Formally, as outlined in LaValle [31], a collision detection module implements a logical predicate $\phi : \mathcal{C} \to \{\text{TRUE}, \text{FALSE}\}$, separating $\mathcal{C}_{free}$ from $\mathcal{C}_{obs}$.

Usually, the models involved in collision detection (robot, objects, obstacles) are expressed as semi-algebraic models, for example as polyhedra models. Despite certain simplifications and tricks, such as using bounding boxes, convex hulls, body decompositions using hierarchical methods, or incremental methods, [31, 38], collision checking can be the most time-consuming task for a motion planning algorithm.

As such a collision detection module is usually separated from the planner itself, we will also treat its function as a black box and concentrate on the planner itself. As mentioned previously, we make use of the collision detection module contained within the Robotics Library [2].

2.2.4 Paths in C-Space

Beyond probing samples whether they lie in $\mathcal{C}_{free}$ or $\mathcal{C}_{obs}$, our basic motion planning problem wants us to find an entire collision-free continuous solution path through $\mathcal{C}_{free}$. Thus, we need a method for verifying that a path segment, defined by two certain samples, is collision-free. Formally, for a path segment defined by $\tau : [0, 1] \to \mathcal{C}$ we want to check whether $\tau([0, 1]) \subset \mathcal{C}_{free}$ holds. The problem is that usually collision detection algorithms can only check collisions for configurations, not for entire paths in the C-space.

A common approach to solve this issue, which we will also follow in this work, is to interpolate the path segment with a fixed C-space step size $\epsilon > 0$ and then check collisions only on the resulting points. The step size ϵ is often determined empirically, but it is crucial to set this variable to an appropriate value for the present motion planning problem. If the value is too small, this results in too many unnecessary time-consuming collision checks, and if the value is too large, we might jump over a thin obstacle.

In this work, we will interpolate a path from q_{start} to q_{end} in the C-space $\mathcal{C} = \mathbb{R}^n$ by incrementally applying the following linear interpolation, starting with $q_{prev} =$

q_{start} and setting $q_{prev} = q_{next}$ after each step:

$$q_{next} = \left(1 - \frac{\epsilon}{\rho(q_{prev}, q_{end})}\right) \cdot q_{prev} + \left(\frac{\epsilon}{\rho(q_{prev}, q_{end})}\right) \cdot q_{end}. \tag{2.19}$$

We obtain a set of configuration points for a certain path segment, on which we can then run collision checking. In this approach we simply traverse the discretized path until we end up in a collision or in the end configuration. The advantage is that we can determine the last valid collision-free configuration, traversing right before the collision configuration. However, if we simply want to check whether the path is collision-free or not, a recursive binary strategy checking the middle point of the yet unexplored path segment yields more efficient results[31].

Note that interpolation becomes more complicated for other C-space compositions, e.g. representing revolute joints with wrap-around or directly planning in $SE(3)$. Kuffner [28] outlines how to interpolate two configurations in $SE(3)$, particularly interpolating quaternions using spherical linear interpolation, also denoted as *slerp*.

2.3 Sampling-Based Motion Planning Algorithms

In this section, we will discuss sampling-based motion planning algorithms. These can be divided into two classes: *single-query models* and *multiple-query models*. Whereas for a single-query model, we are only interested in solving a single start-goal query for a certain static workspace, multiple-query models utilize precomputation to answer multiple queries more efficiently.

We will first present the *rapidly-exploring random tree* (*RRT*), which is a single-query model and will be the foundation of our upcoming manipulation planning algorithms in this work. We then present *probabilistic roadmaps (PRMs)* as one of the most famous multiple-query models, compare them with RRTs and discuss some tradeoffs.

Note that there are many more basic sampling-based motion planning algorithms, as further examined in [31]. For instance, we can adapt discrete search algorithms like A* to define a grid over our C-space and incrementally refine the resolution until we find a solution. Moreover, randomized potential fields utilize random walks and clever heuristic functions to solve more challenging tasks than discrete search algorithms can usually do. However, RRTs and PRMs are directly developed for the continuous C-space, do not require much parameter tuning or many heuristic parameters, and are widely and successfully used today [36, 37, 27].

2.3.1 Rapidly-exploring Random Trees (RRTs)

In this section we will cover the basics of one of the most famous sampling-based planning algorithm, the *rapidly-exploring random tree* (*RRT*). As opposed to grid-based searching approaches, the RRT abandons the concept of having a certain fixed grid resolution. Instead, it iteratively constructs its tree to gradually cover the C-space more and more, requiring hardly any parameter tuning. The RRT belongs to the family of the *rapidly-exploring dense trees* (*RDTs*), which in the limit densely cover the C-space, however, the RRT uses a sequence of samples chosen at *random* for the tree construction.

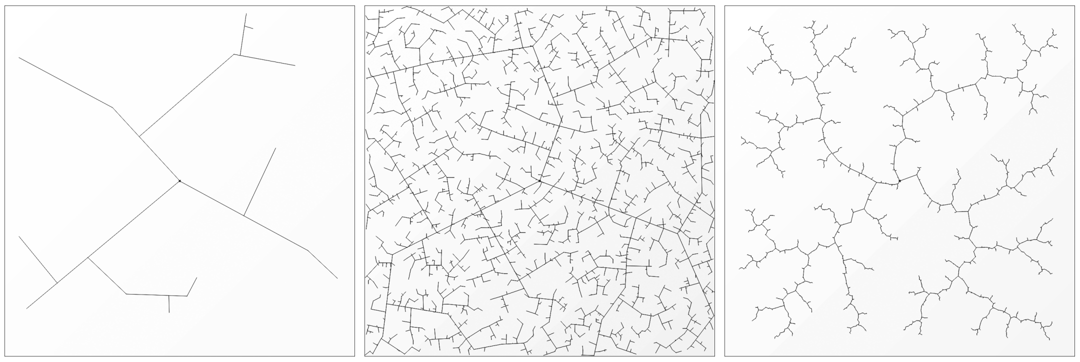
The RRT does not immediately solve a planning query. Instead, starting from an initial configuration, it incrementally constructs a tree to quickly reduce the expected distance of any randomly-chosen configuration to the tree. In the original form, the RRT does not try to reach any goal configuration, it rather explores the space.

Algorithm 1
RRT($\mathcal{C}$, q_{init}, Q_{goal}, ρ), as outlined by Barry [6, p. 20]
Input: C-space $\mathcal{C}$, Initial configuration q_{init}, Goal set Q_{goal}, Distance metric ρ
Output: A path from q_{init} into Q_{goal}

```
1: $V_0 \leftarrow \{q_{init}\}$
2: $k \leftarrow 1$
3: while $V_{k-1} \cap Q_{goal} = \emptyset$ do
4:     $q_{sample} \leftarrow \textsc{Sample}(\mathcal{C})$
5:     $q_{near} \leftarrow \arg\min_{v \in V_{k-1}} \rho(v, q_{sample})$
6:     $V_k \leftarrow V_{k-1} \cup \textsc{Extend}(q_{near}, q_{sample})$
7:     $k \leftarrow k + 1$
8: end while
9: return $\textsc{ExtractPath}(V_{k-1})$
```

Algorithm 1 shows a possible implementation of the basic RRT. Given a C-space with a distance metric, an initial configuration and a goal set, we start with creating the tree at the initial configuration, which will be our root vertex. Then, until our tree does not contain any goal configurations, we repetitively execute the following steps:

1. Sample a new configuration q_{sample} in the C-space
2. Search in the vertices of the tree for the nearest neighbor of q_{sample}, which we will denote as q_{near}, using the distance metric of the C-space.
3. Extend the tree by moving along the shortest possible path from q_{near} in the direction of q_{sample}. For example, consider to simply add the new vertex q_{sample} and the new edge connecting q_{near} with q_{sample} to the tree.



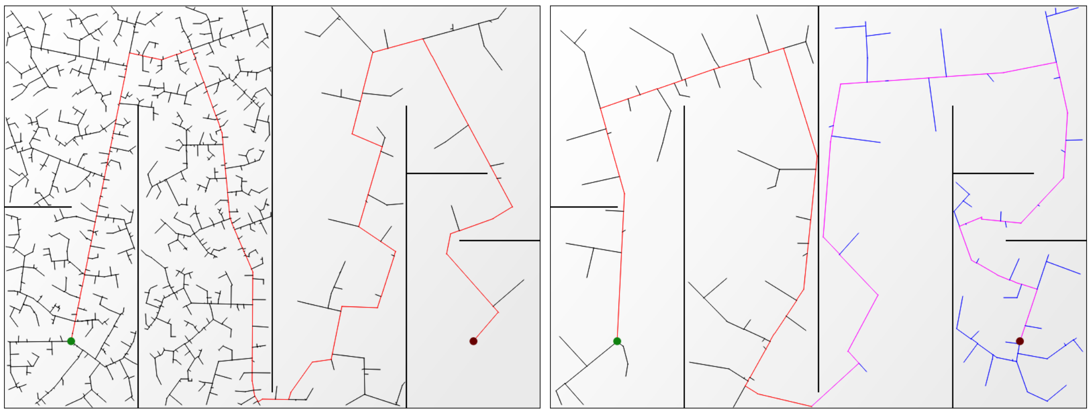
(a) 15 iterations, $\Delta q = 100$ (b) 2000 iterations, $\Delta q = 100$ (c) 2000 iterations, $\Delta q = 1$

Figure 2.6: **RRT algorithm exploring the C-space**. $\mathcal{C} = [0, 100]^2$, $q_{init} = (50, 50)$, ρ is the Euclidean metric. (a) An RRT after 15 iterations with $\Delta q = 100$ quickly exploring the largest yet unexplored regions. (b) An RRT after 2000 iterations with again $\Delta q = 100$, densely covering the C-space in the limit. (c) An RRT with a much lower incremental distance $\Delta q = 1$ after 2000 iterations, which again nicely illustrates the strong bias towards regions not yet visited. All three RRTs were visualized using RRTExplorer, a tool developed by Knispel [26].

In the literature about RRTs you will find different implementations of each of these basic steps, therefore it is important to accurately define the RRT approach we will use in this work.

Firstly, for the nearest neighbor search, we will only consider the vertices already contained in the tree, instead of also considering the infinite number of points lying on the edges of the tree. In order to compensate for ignoring edges in the nearest neighbor search at all, we will replace the edge splitting technique with an easier method to insert intermediate vertices along path segments. Each time covering a distance of $\delta > 0$ when extending the tree while traversing a path, an intermediate vertex will be added to the tree. Throughout the planning algorithm implementations in our work, we will use the *k-d tree* data structure to efficiently find nearest points, which is already implemented within the Robotics Library [2]. The k d-tree operates as a multi-dimensional binary search tree, in which it efficiently organizes points in a k-dimensional space, where k will be the dimension of our C-space.

Secondly, let us briefly outline the basic implementation of our function Extend. When extending the tree, we have to take collision checking into consideration. Recall from the previous section about paths in the C-space to interpolate the path segment from q_{near} to q_{sample} with a fixed C-space step size $\epsilon > 0$. We then traverse along the discretized path checking collisions until we end up in a collision configuration or reach q_{sample}. The last valid collision-free configuration will be added to the vertex list of the tree, along with the edge connecting it with q_{near}.



(a) Single RRT: 1604 branches, $\omega = 0.1$ (b) RRTConnect: 201 branches (99 and 102)

Figure 2.7: **Solving a motion planning query with RRT-based planning in a maze-like C-space**. The green dot illustrates the starting and the red dot the goal configuration. Let again be Δq the maximum step size and ρ the Euclidean metric. (a) The single RRT with 10% goal sampling solves this maze after evolving 1604 branches (red path). (b) In contrast, the bidirectional RRT requires much less branches to pass through the narrow passages. Here, a solution is found after 99 branches of the left tree (black, red) and 102 branches of the right tree (blue, purple). Again we used the tool RRTExplorer, developed by Knispel [26], in order to create and visualize these scenarios.

Besides the very small step size ϵ to interpolate path segments, and the step variable δ to define the maximum distance of two consecutive vertices in the tree, let us introduce one more fixed step variables, which should not be confused with the other two. Often in RRT implementations, we do not really extend all the way from q_{near} to q_{sample}, which might be a long path and unlikely to be collision-free. Rather, we are satisfied with only traversing an incremental distance, which we will denote as Δq, in the direction of q_{sample}.

Figure 2.6 shows three RRTs after a certain iteration count and with a certain incremental distance Δq, respectively, and how they explore the obstacle-free C-space defined by $\mathcal{C} = [0, 100]^2$. Whereas Figure 2.6a and 2.6c illustrate the aggressive exploration of the largest not yet visited places in the beginning, 2.6b already kind of gradually refines the C-space demonstrating how RRTs ensure a dense cover in the limit. With a much smaller incremental distance Δq as shown in Figure 2.6c, the exploration phase of the C-space will be much slower, but with higher resolution. Figure 2.6c nicely illustrates the main branches of the tree trying to reach the far corners of the C-space.

Now that we discussed how the RRT explores the C-space, strictly speaking only the obstacle-free C-space $\mathcal{C}_{free}$, let us now discuss how planning algorithms can use this data structure to solve a planning query.

2.3.1.1 Single RRT-based Planning

The simplest approach for motion planning with RRTs is to just integrate a single RRT into planning. Algorithm 1 already outlines the implementation, but we still have to discuss how to incorporate the goal set. If the goal set only consists of a finite number of points in the continuous C-space, there is zero chance that our exploring RRT will contain any of these points in its vertex list in finite time[3]. Therefore, a rather simple method to consider the goal set in the tree construction is to periodically sample a goal instead of a random configuration. Let $\omega \in [0, 1]$ denote the proportion or percentage of a goal configuration being sampled instead of a random one. For example, if $\omega = 0.1$, every 10th sample will be a goal configuration, thus forcing the RRT to attempt to connect to the goal set. Choosing the right value for ω means balancing greediness and (uniform) exploration.

Finally, when the RRT reaches a certain goal configuration $q_{goal} \in Q_{goal}$, $\textsc{ExtractPath}(V_{k-1})$ returns the path from q_{init} to q_{goal}. This will be done most efficiently for the RRT, which is a directed graph, by starting with q_{goal}, traversing the single incoming edge and its respective vertex, and keep iteratively traversing until there is no more incoming edge, that is when we reached the starting configuration. Now we just have to reverse the path. Figure 2.7a illustrates an example of motion planning with a single RRT and its resulting solution path.

2.3.1.2 Bidirectional RRT-based Planning

A more advanced version of RRT-based planning is the bidirectional search, referred to as the RRTConnect algorithm, in which we grow two trees. The first tree grows forward from the starting configuration and the second tree backwards from the goal set, in which connecting both means that we found a solution path.

Algorithm 2 outlines the implementation of RRTConnect. The basic idea is to pick one of both trees, denoted as V_a, and extend it as usual by random sampling (here sampling goal configurations is not needed). If the extension step induced at least one additional vertex in V_a, RRTConnect then extends the other tree, referred to as V_b, towards the last point added to V_a, denoted as q_l. In other words, while V_a explores in random fashion, V_b will be extended towards the result of V_a's extension, thus trying to connect both trees. If extending V_b succeeds, thus the last added configuration r_k being equal to q_l, we indeed found a solution

[3]As outlined by Barry [6] in Chapter 2.2.1, this is an example of the C-space consisting of an "'important"' measure-zero set, in which case the RRT algorithm is not probabilistically complete. In general, the C-space and the choice of the extend and distance functions determine whether an RRT algorithm will be probabilistically complete.

Algorithm 2

RRTCONNECT($\mathcal{C}$, q_{init}, Q_{goal}, ρ), as outlined by Barry [6, p. 38]

Input: C-space $\mathcal{C}$, Initial configuration q_{init}, Goal set Q_{goal}, Distance metric ρ

Output: A path from q_{init} into Q_{goal}

```
 1: $V_a \leftarrow \{q_{init}\}$
 2: $V_b \leftarrow \{\textsc{Sample}(Q_{goal})\}$
 3: $F \leftarrow$ **true** // *True when extending forwards*
 4: **while true do**
 5:   **if** $F$ **then**
 6:     // *Add a goal configuration to the backwards tree*
 7:     $V_b \leftarrow V_b \cup \{\textsc{Sample}(Q_{goal})\}$
 8:   **end if**
 9:   $q_{sample} \leftarrow \textsc{Sample}(\mathcal{C})$
10:   $q_{near} \leftarrow \arg\min_{v \in V_a} \rho(v, q_{sample})$
11:   $\{q_0, ..., q_l\} \leftarrow \textsc{Extend}(q_{near}, q_{sample}, F)$
12:   $V_a \leftarrow V_a \cup \{q_0, ..., q_l\}$
13:   **if** $l > 0$ **then**
14:     // *Extend $V_b$ towards $V_a$*
15:     $q_{near} \leftarrow \arg\min_{v \in V_b} \rho(v, q_l)$
16:     $\{r_0, ..., r_k\} \leftarrow \textsc{Extend}(q_{near}, q_l, \neg F)$
17:     $V_b \leftarrow V_b \cup \{r_0, ..., r_k\}$
18:     **if** $r_k = q_l$ **then**
19:       **return** $\textsc{ExtractPath}(V_a, V_b)$
20:     **end if**
21:   **end if**
22:   $\textsc{swap}(V_a, V_b)$
23:   $F \leftarrow \neg F$
24: **end while**
```

path. In this work, we will use the unbalanced version of the bidirectional search, which means that we will swap both trees in every iteration, not taking care of equal size, however to define the size of a tree. A balanced version can also be a challenging task when taking into account that the backward tree has to ideally cover the whole goal set, which, unlike for the starting point of the forward tree, might consist of multiple or even infinite configurations. In order to cope with this issue, we simply add a sample from the goal set to our backward tree in every iteration. Note that the variable F keeps track of whether we extend with forward or backward motion, which will be crucial for the EXTEND function in non-holonomic cases, which we introduce shortly and have to deal with later on in this work.

Figure 2.7b illustrates the RRTCONNECT algorithm growing two trees to solve motion planning in a maze with narrow passages much more efficient than single RRT-based planning. In practice, bidirectional search yields better performance, especially when dealing with narrow passages or escaping bug traps in the C-space. Growing even more trees from different parts of the C-space might be even better, but developing and efficiently controlling such an algorithm is in fact not intended within the scope of this work.

2.3.1.3 RRT-based Planning in Non-Holonomic Spaces

So far, the EXTEND function was meant to handle holonomic systems, where the whole C-space is directly controllable, as is the case for a point robot in a 2D maze shown in Figure 2.7. However, we can easily modify the EXTEND function to also handle non-holonomic systems and differential constraints, where we cannot directly move along straight lines in the C-space anymore. For instance, when steering a car, we cannot move sidewards.

Algorithm 3
EXTEND(q_I, q_G, F), as outlined by Barry [6, p. 36]
Input: Initial configuration q_I to extend from, Goal configuration q_G to extend to, Boolean F to keep track of whether extending with forward or backward motion
Output: A collision free path from q_I towards q_G

```
1: $\{q_0, ..., q_m\} \leftarrow$ EMPTYSPACEPLANNER($q_I, q_G, F$)
2: for $q_i \in \{q_0, ..., q_m\}$ do
3:    if COLLIDING($q_i$) then
4:       return $\{q_0, ..., q_{i-1}\}$
5:    end if
6: end for
7: return $\{q_0, ..., q_m\}$
```

Algorithm 3 shows an implementation of the EXTEND function for non-holonomic

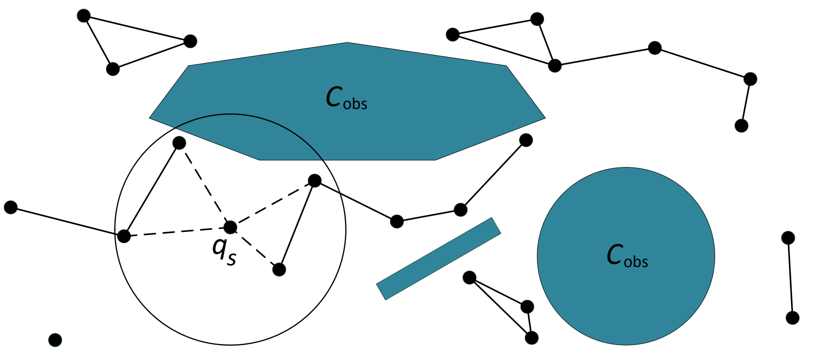


Figure 2.8: **The PRM algorithm incrementally constructs the sampling-based roadmap**. For a new collision-free sample q_s, the PRM tries to connect it to nearby vertices.

spaces. It makes use of an *empty space planner*, which return an executable path for the robot from start to end configuration in the absence of obstacles. Then, this path is checked for and possibly truncated to the first collision, taking previous discussions about the EXTEND function into account. The empty space planner will be crucial for multi-modal planning or solving manipulation problems, due to its ability to "‘manually"’ travel through low-dimensional subspaces of the C-space. It is problem dependent and will be a crucial part of the input to our multi-modal planning algorithm, as clarified by Barry [6].

2.3.2 Probabilistic Roadmaps (PRMs)

Now let us briefly outline the sampling-based roadmap idea of multiple-query models, based on the most famous algorithm of *probabilistic roadmap (PRM)*.

In order to efficiently solve multiple queries for the same static C-space, the PRM first constructs a topological graph, referred to as a *roadmap*. Figure 2.8 illustrates the construction phase of the roadmap, which works as follows. The PRM iteratively samples in the free C-space $\mathcal{C}_{free}$, adds the new configuration to the roadmap and attempts to connect it to each of its neighbors, using a local planner like the EXTEND function we introduced previously. Thereby, the neighborhood of a vertex can be defined very differently, for example based on the k-nearest neighbor method or defining a certain radius. Finally, the roadmap should be large enough to cover most of the free C-space $\mathcal{C}_{free}$, in a way that it will be easily reachable from the initial and goal configurations of our queries.

Given the roadmap, we can then proceed to the query phase, in which we attempt to find a path from the initial to the goal configuration of each query. This can be done by connecting these two configuration to the roadmap and then search for a solution path using graph search methods, for example Dijkstra's algorithm.

In comparison to RRTs, PRMs are much more efficient for answering multiple queries in a static environment due to their precomputed roadmap. However, in this work, we are more interested in solving a single query problem, which RRTs can solve more efficiently. Moreover, RRTs can more effectively handle differential constraints and non-holonomic systems, as is usually the case for multi-modal planning. Thus, RRTs were found to be more suitable to use in this work. However, there are many powerful variations and extensions improving both of these basic algorithms, and even combinations of RRTs and PRMs.

2.4 Multi-Modal Planning

In this section, we will briefly outline the concept of *multi-modal planning* [16, 17, 18], which lays another foundation for our later work on manipulation planning with diverse actions.

In many important planning problems, we have to plan for a so-called *hybrid system*, in which the state space additionally consists of a discrete finite (or countably infinite) set called the *mode space*. In other words, the system moves not only between configurations, but also among a set of modes. Depending on the current mode, the robot is further constrained to traverse only a certain lower dimensional subspace of the actual C-space.

As pointed out by Hauser [16, p. 9], robot systems with contact move between modes, in which each mode defines a fixed contact state. For instance, Hauser [16] uses this concept for legged locomotion, in which the modes represent a fixed set of footfalls, which constrain the feet to be on the ground. Transitioning to one another, these modes ultimately ensure walking. Also manipulation planning exhibits a multi-modal structure. For instance, when a robot pushes an object in a fixed contact point, it traverses one and the same mode. However, when it then plans to push the object in another contact point, it first has to pass through the mode specifying no contact at all, in order to finally reach the mode specifying the new contact point. Compared to conventional robot motion planners, a multi-modal planner has to find a discrete sequence of mode transitions for reaching the goal and continuous paths within each mode to achieve them.

Assume a system's *hybrid state* (q, σ) with $q \in \mathcal{C}$ as a conventional continuous configuration and σ as a mode in a mode space Σ. For each mode $\sigma \in \Sigma$, the corresponding *feasible space* $\mathcal{F}_\sigma \subseteq \mathcal{C}$ consists of all configurations that satisfy certain mode-specific constraints, and is usually a lower dimensional submanifold

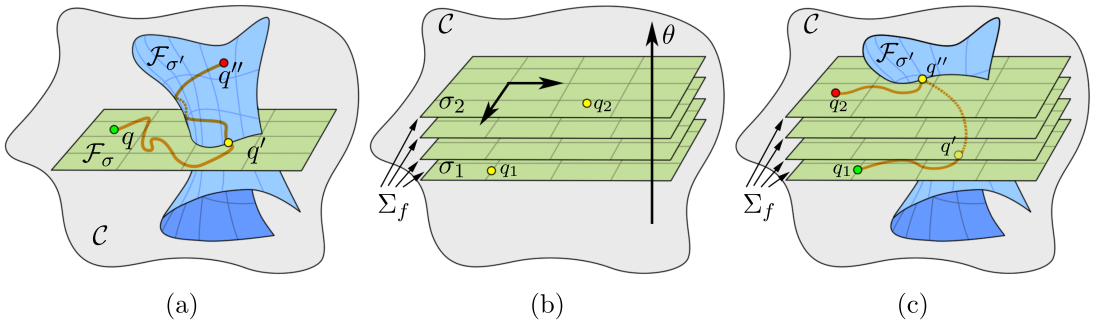


Figure 2.9: **Illustration of multi-modal planning for hybrid systems**. (a) For a valid path from $q \in \mathcal{F}_\sigma$ to $q'' \in \mathcal{F}_{\sigma'}$ of an adjacent mode σ', we have to travel through the transition configuration $q' \in \mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}$. (b) For a single mode family $\Sigma_f \in \{\Sigma_1, ..., \Sigma_F\}$, each instantiation of the continuous coparameter θ defines a different mode $\sigma \in \Sigma_f$ (four are illustrated), which is also uniquely represented by its configuration q_i. (c) In order to transition between modes of the mode family Σ_f from q_1 to q_2, we first have to transition to a configuration q', from which we can transition to a mode of another mode family. From this we can then ultimately transition back to the goal mode of our original family Σ_f and plan towards the goal configuration q_2. All figures from [16, pp. 24, 26].

of $\mathcal{C}$ [18]. The multi-modal planner now has to find a collision-free path from a starting hybrid state (q_s, σ_s) to a goal state (q_g, σ_g), by planning within a single mode, but also through transition configurations $q \in \mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}$, in order to achieve a transition between the modes σ and σ'. A so-called *mode adjacency graph* describes all allowed transitions between modes. Thus, only if σ and σ' are adjacent, a transition configuration $q \in \mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}$ can lead to a mode switch, if such a configuration exists and is reachable for the robot. Figure 2.9a illustrates multi-modal planning we discussed so far.

In case of Σ being infinite and uncountable, we can partition Σ into the set $\{\Sigma_1, ..., \Sigma_F\}$ of F disjoint *mode families*, where each mode of a mode family is defined uniquely by a coparameter θ. Moreover, all modes of a mode family are disjoint, too. Thus, in order for our system to transition between two modes of one and the same family, it must first transition to at least one mode of another family. As a final condition, the coparameter θ must be uniquely derived given the mode family and a configuration $q \in \mathcal{C}$. Figure 2.9b and 2.9c illustrate multi-modal planning with mode families. Figure 2.10 illustrates continuous modes and discrete mode families for a simple 1D world.

The sampling-based planners we discussed so far can easily be used to plan within a single mode. However, the do not respect the multi-modal nature of manipulation planning and therefore have to be adapted accordingly. For instance, in order to cope with mode transitions, we should explicitly sample from mode tran-

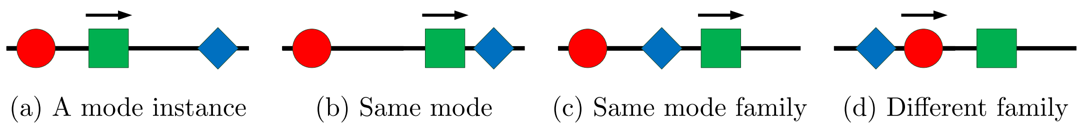


Figure 2.10: **Example world for multi-modal planning illustrating modes and mode families.** Only one object can be moved at a time. A mode family is defined by specifying which object to move, and each mode within a mode family is defined by its coparameter, which specifies the exact locations of all the other objects. Thus, there are three mode families for manipulating each object and infinite and uncountable modes, as objects can be located anywhere along the 1D line. (a) A mode instance of the green mode family. (b) Within the same mode of the green mode family, we can move the green object arbitrary. (c) As the locations of the other objects have changed, so has the mode of the mode family. Note that, in order to transition from mode (a) to this mode, we first have to transition to the mode family of moving the blue object. (d) By moving the red object we change the mode family. Illustration idea inspired from [6, p. 47].

sitions, which usually exist only in low-dimensional subspaces, and therefore will not be sampled when sampling uniformly at random. As summarized by Barry [6], explicit multi-modal algorithms essentially work as follows. Given a current hybrid state (q, σ), we first sample an adjacent mode σ' from the mode adjacency graph, then sample a transition configuration q' from $\mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}$, and finally plan a collision-free path within σ from q to q'.

The algorithms for diverse action manipulation we present in the following chapter have to and will follow this structure of multi-modal planning.

3 Diverse Action Manipulation

In this chapter, we discuss diverse action manipulation and its foundation as multi-modal path planning. After defining the Diverse Action Manipulation (DAMA) problem, we present three sampling-based algorithms for solving the DAMA problem. The first algorithm will be based on the RRT, outlined in Algorithm 1, and named as Diverse Action Rapidly-exploring Random Tree (DARRT) algorithm. The second algorithm will extend the first one to DARRT-Connect, following the idea of extending RRT to RRTConnect (Algorithm 2). Finally, the third algorithm DARRTH will solve the DAMA problem in multiple steps by applying a hierarchical approach.

This chapter and its notation is closely based on the PhD thesis of Jennifer L. Barry *Manipulation with Diverse Actions* [6], who first made the DAMA problem public, and her previously published major papers concerning multi-modal manipulation [4, 5]. However, we implemented the algorithms completely on our own and made major variations concerning some implementation details. When suitable, we will further discuss differences between our work and the work of Jennifer L. Barry.

3.1 Problem Description

Assume a robot with configuration space (C-space) R and n movable objects and their corresponding C-space $O_1, ..., O_n$. For a planning algorithm to be able to plan for the robot and the objects, we combine all C-spaces to the final cross-product composition $X = R \times O_1 \times ... \times O_n$, as pointed out in Chapter 1. Thus, a configuration $x \in X$ includes the configuration for the robot, as well as for the n objects. We can formally define the concept of a *trajectory* as follows:

Definition 3.1 (**Trajectory**):
For configuration $x_S, x_G \in X$, a function $\tau : [0, 1] \rightarrow X$ is a *trajectory from x_S to x_G* if and only if $\tau(0) = x_S$ and $\tau(1) = x_G$.

Now we need a set of so-called *manipulation primitives*, which describe the actions and manipulations the robot is allowed to perform in the composed C-space X.

Manipulation primitives return a trajectory, which usually only traverses a subspace of X. Furthermore, they can only operate within their domain, this means a manipulation primitive can only begin at or reach a certain set of configurations. Formally:

Definition 3.2 (**Manipulation Primitive**):
Given an initial configuration x_S and a goal configuration x_G, a *manipulation primitives* p returns a trajectory from x_S to x_G. It is *applicable* only to pairs of configurations within its domain, denoted as $X(p)$:

$$X(p) = \{(x_S, x_G) \in X \times X \mid (x_S, x_G) \text{ is in domain of } p\}. \tag{3.1}$$

Thus, we can define the set of valid initial configurations $X_S(p)$ and the set of reachable configurations $X_G(p)$ as follows:

$$X_S(p) = \{x_S \in X \mid \exists x_G \in X : (x_S, x_G) \in X(p)\} \tag{3.2}$$
$$X_G(p) = \{x_G \in X \mid \exists x_S \in X : (x_S, x_G) \in X(p)\}. \tag{3.3}$$

Finally, given a certain initial configuration $x_S \in X$, $X_G(p|x_S)$ gives us the set of reachable configurations from x_S using the primitive p:

$$X_G(p|x_S) = \{x_G \in X \mid (x_S, x_G) \in X(p)\}. \tag{3.4}$$

Note that usually, $X_S(p)$ and $X_G(p)$ are lower-dimensional spaces than X, thus preventing our traditional RRT-based algorithm from being able to solve manipulation problems, as it lacks sampling in these spaces, for instance.

Manipulation primitives can be separated into two classes:

Definition 3.3 (**Transit/Transfer Primitive**):
For a primitive p, let $(x_S, x_G) \in X(p)$ and $\tau = p(x_S, x_G)$. The primitive p is a *transit* primitive if and only if for all $\alpha \in [0, 1]$, the configuration of every object in $\tau(\alpha)$ remains unchanged with respect to $\tau(0)$. Otherwise, p is called a *transfer* primitive.

In order to make the concept of primitives more tangible, let us introduce two basic manipulation primitives we will use throughout this work, one transit and one transfer primitive. We will introduce more primitives later on, and the implementation details and possible variations of both of these will be further discussed when presenting the DAMA scenarios.

`Transit` describes moving the robot alone while not changing any object configuration. The domain of `Transit` are only configuration pairs $(x_S, x_G) \in X$, where all objects are placed on a support surface in x_S and x_G and do not change their configuration between x_S and x_G. `Transit` then returns a trajectory from initial to goal configuration following a straight line in the joint space of the robot.

`Push` describes the robot pushing an object in a certain direction. The domain of `Push` consists of configuration pairs $(x_S, x_G) \in X$, in which all objects rest on a support surface in x_S, only one object moves on its support surface between x_S and x_G, and the robot is in pushing contact with this object in x_G along the pushing way. `Push` calls the primitive `Transit` to move the robot to the initial pushing position, in which usually the robot's gripper is in contact with the object and their respective center points are aligned along the pushing path. `Push` then returns a trajectory sequence, in which the robot first moves to the pushing pose via `Transit`, and then along the pushing path.

Note that primitives can also be nested, as `Push` itself calls `Transit` to return a trajectory. In contrast to Barry [6], we found it useful to provide `Push` with the ability to `Transit` to the pushing pose, as it makes sampling more easy later on. However, this leads us to the concept of a trajectory sequence $\{\tau_0, ..., \tau_l\}$, which can be *generated* by sequencing the output of the set of primitives $P = \{p_0, ..., p_l\}$. With having a set of trajectories instead of one enlarged trajectory, we can more easily identify the primitive which generated a certain trajectory. In the same way we defined applicability for a single manipulation primitive, we can extend it to the set of primitives P. For instance, $X(P)$ denotes the set of configurations for which P can generate a trajectory sequence:

$$X(P) = \left\{ (x_S, x_G) \in X \times X \;\middle|\; \begin{array}{c} \text{some sequence of the primitives in } P \text{ can} \\ \text{generate a trajectory sequence from } x_S \text{ to } x_G \end{array} \right\}. \tag{3.5}$$

With this brief introduction, we can formally define the diverse action manipulation problem, as stated by Barry [6]:

Definition 3.4 (DAMA Problem):
The *Diverse Action MAnipulation* (DAMA) problem $\mathcal{P}$ is a tuple $\langle R, \{O_1, ..., O_n\}, \{B_0, ..., B_q\}, \{p_0, ..., p_m\}, x_0, X_G \rangle$, in which R is the configuration space for a robot, $\{O_1, ..., O_n\}$ are the configuration spaces for the movable objects, $\{B_0, ..., B_q\}$ is a set of fixed obstacles, $\{p_0, ..., p_m\}$ is a set of manipulation primitives, x_0 is an initial configuration, and X_G is a set of goal configurations.

Recall that the C-space of the DAMA Problem is the cross product of both the robot and the object C-spaces, thus $X = R \times O_1 \times ... \times O_n$. Furthermore, in practice, the goal set for a DAMA problem often contains infinite configurations, as we are usually only interested in the goal configurations of the objects. Thus for the robot itself, any configuration may be a goal configuration in the end.

Barry [6] introduces the concept of primitives being able to permit certain collisions. For instance, `Push` can disable collision checking between the robot's gripper and the object. However, in this work for planning, we avoid permissible real contact between bodies completely. Thus, when moving an object, the robot never touches it in simulation, and objects float slightly above their support surfaces instead of real contact. Nevertheless, in our implementation, the collision checking routine knows about the currently processed primitive and thus only checks collisions appropriately. For example, when processing `Transit`, collision checking between objects and non-robot bodies (other objects and fixed obstacles) is disabled, because objects do not move during the robot transition.

Let the *free space* $X_{free}(\mathcal{P})$ for a DAMA problem $\mathcal{P}$ be the set of all configurations in which there is no contact between the robot bodies, objects, or fixed obstacles. We then define a collision free trajectory sequence as follows, omitting the concept of primitives allowing certain contacts:

Definition 3.5 (**Collision Free**)**:**
A trajectory τ generated by primitive p is *collision free* in DAMA problem $\mathcal{P}$ if, for all $\alpha \in [0, 1]$, $\tau(\alpha) \in X_{free}(\mathcal{P})$. A trajectory sequence is *collision free* if all of its trajectories are collision free.

To conclude, a *solution* to a DAMA problem is a collision free trajectory sequence generated by the given primitives from x_0 to any configuration in X_G. Let us now discuss RRT-based algorithms to solve the DAMA problem.

3.2 Diverse Action RRT Algorithm (DARRT)

In this section, we discuss the Diverse Action Rapidly-exploring Random Tree (DARRT) algorithm. The DARRT algorithm modifies and extends the basic RRT algorithm (Algorithm 1) to solve a DAMA problem. Before presenting the pseudocode for DARRT, we have to revise and discuss three crucial issues for diverse manipulation to be solvable: The empty space planner, sampling, and distance metrics.

3.2.1 Empty Space Planner

Algorithm 4
EXTEND($x_S, x_G, X, \{B_0, ..., B_q\}, \{p_0, ..., p_m\}$), as outlined by Barry [6, p. 60]
Input: Start configuration x_S, Goal configuration x_G, Composed C-space X, Fixed obstacles $\{B_0, ..., B_q\}$, Manipulation primitives $\{p_0, ..., p_m\}$
Output: A collision free trajectory sequence from x_S towards x_G

```
 1: $\{\tau_0, ..., \tau_l\} \leftarrow$ EMPTYSPACEPLANNER($x_S, x_G, \{p_0, ..., p_m\}$)
 2: for all $i \in \{0, ..., l\}$ do
 3:     for all $\alpha \in [0, 1]$ do
 4:         //$\alpha$ is usually discretized in practice
 5:         if COLLISION($\tau_i(\alpha), \{B_0, ..., B_q\}, X$) then
 6:             return $\{\tau_0, ..., \tau_{i-1}\} \cup \{\tau_i$ from 0 to $\alpha\}$
 7:         end if
 8:     end for
 9: end for
10: return $\{\tau_0, ..., \tau_l\}$
```

In diverse action manipulation, the C-space $X = R \times O_1 \times ... \times O_n$ is not fully controllable, because the n objects cannot move by themselves. Thus, we have to replace the straight-line extension of a holonomic system with the non-holonomic version of the EXTEND function (Algorithm 3) we discussed in Section 2.3.1.3. Recall the idea of using an *empty space planner* (ESP), which returns an executable path for the robot from the initial to the end configuration in the absence of obstacles. Thus, in case of our DAMA problem, the ESP uses the predefined primitives in order to return a trajectory sequence from a given start configuration x_S to a goal configuration x_G with $x_S, x_G \in X(P)$. Algorithm 4 illustrates the implementation of the non-holonomic EXTEND function in case of solving a DAMA problem. After calling the ESP to return a trajectory sequence for the empty space, it checks this trajectory for collisions, truncates it if needed, and finally returns a collision free trajectory sequence.

We can think of the ESP as being a small planner itself, trying to solve the task without any collision checking. With the ESP and its use of the primitives, we will plan through low-dimensional subspaces of the composed DAMA C-space X. Inspired by Barry's implementation [6, p. 62] of the empty space planner, each primitive has to implement two functions: PROPAGATE and ISUSEFUL. The function PROPAGATE of a primitive p applies the primitive's manipulation to a given starting configuration x_S in order to move towards a goal configuration x_G, ultimately returning a trajectory sequence from x_S to x_G. However, if $(x_S, x_G) \notin X(p)$, PROPAGATE returns an empty sequence. Therefore, we will check with ISUSEFUL function of p whether it is applicable to (x_S, x_G). The ESP then works as follows: As long as there are useful primitives to move from the last reached configuration of the ESP towards the goal x_G, we pick one at random and apply its PROPAGATE function. In other words, the ESP chains propagate steps of useful primitives, until there are no more useful primitives. Algorithm 5 illustrates the basic implementation.

Algorithm 5
EMPTYSPACEPLANNER($x_S, x_G, \{p_0, ..., p_m\}$), as outlined by Barry [6, p. 62]
Input: Start configuration x_S, Goal configuration x_G, Primitives $\{p_0, ..., p_m\}$
Output: Trajectory sequence from x_S towards x_G

```
1: if $x_S = x_G$ then
2:     return $\emptyset$
3: end if
4: $p \leftarrow$ pickRandomUsefulPrimitive($x_S, x_G, \{p_0, ..., p_m\}$)
5: $\{\tau_0, ..., \tau_l\} \leftarrow p$.PROPAGATE($x_S, x_G$)
6: return $\{\tau_0, ..., \tau_l\} \cup$ EmptySpacePlanner($\tau_l(1), x_G, \{p_0, ..., p_m\}$)
```

In practice, when solving more sophisticated DAMA problems with our Meka robot, we soften the constraint of $(x_S, x_G) \in X(p)$ for propagating primitive p. Instead, we usually just require $x_S \in X_S(p)$ and let primitive p propagate towards x_G, potentially projecting x_G to some nearby $x'_G \in X_G(p|x_S)$. Moreover, in our implementation, ISUSEFUL does not fully check if PROPAGATE returns an empty sequence. Rather, we implement such a function manually checking for $x_S \in X_S(p)$ and possibly adding some heuristic function. Thus, PROPAGATE can ensure that $x_S \in X_S(p)$ has been checked and ISUSEFUL avoids the computational overhead of PROPAGATE, especially the potential inverse kinematic computations. Note that the PROPAGATE function can still fail and return an empty sequence, even if the primitive was found useful, for instance when the computation of the inverse kinematics does not converge. Thus, in our implementation, the PICKRANDOMUSEFULPRIMITIVE function keeps track of and ignores primitives, which failed since the ESP's last successful propagate step.

For our previously introduced primitives `Transit` and `Push`, both functions are basically implemented as follows:

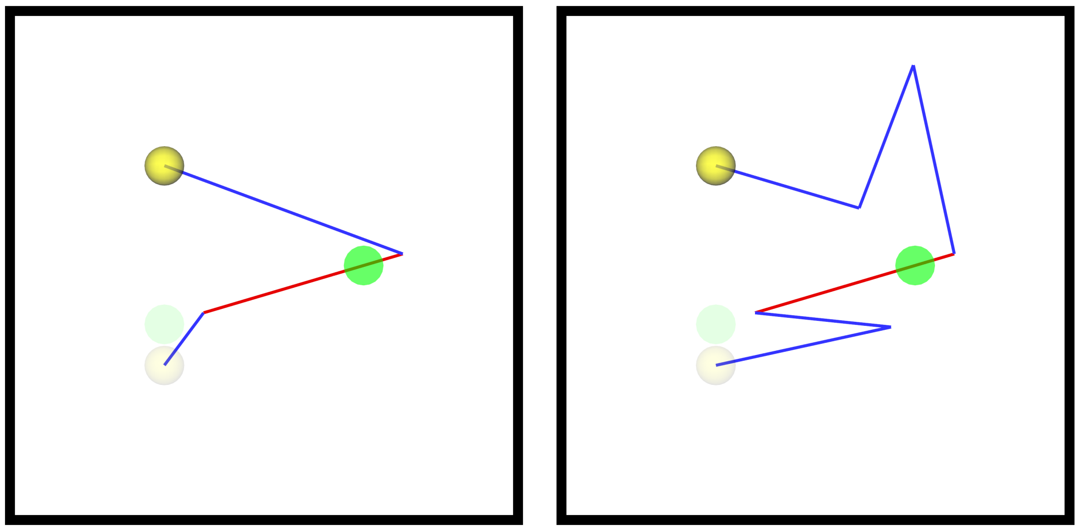
- `Transit`:

  isUseful returns *true* only if all objects are placed on a support surface in x_S and object configurations do not change between x_S and x_G, but the robot position changes.

  propagate returns a trajectory sequence from x_S to x_G following a straight line in the joint space of the robot.

- `Push`:

  isUseful returns *true* only if all objects are placed on a support surface in



(a) Empty space plan (b) Possible DARRT solution

Figure 3.1: **Empty space planner for a simple 2D mobile robot**. Within a 2D world without obstacles, assume x_S in which a robot (yellow) has to move an object (green) to its goal location and finally reach the location right underneath the object (x_G is illustrated by the less visible drawings). Trajectories generated by `Transit` are drawn in blue, trajectories generated by `Push` are drawn in red. (a) Empty space plan generated by the ESP from x_S to reach x_G: First `Transit` to the initial pushing location, then `Push` the object to the goal, finally `Transit` to the robot goal position. (b) Despite the world being obstacle-free, the empty space plan in 3.1a causes collisions between the robot and the object, when approaching or retreating from pushing positions. Here a possible collision-free solution path after several iterations of the DARRT algorithm is shown.

x_S and at least one object moves on its support surface between x_S and x_G.

propagate returns a trajectory sequence from x_S towards x_G by pushing the object, which is nearest to the robot, as follows: We first compute the initial robot pushing position (via inverse kinematics for a joint robot) and propagate via `Transit` to this location. Next, we calculate the final robot pushing position (again usually via inverse kinematics), in which the respective object has reached its goal, and append the trajectory for this pushing path.

Figure 3.1a shows an empty space plan generated by the ESP for a simple 2D mobile robot in a world without obstacles. However, when extending from that trajectory sequence, the path will be truncated before reaching the initial pushing position, due to robot-object collision. A solution to this problem would require the robot to bypass the object, or simply move around in the world alone, as illustrated in Figure 3.1a. Our non-holonomic RRT we discussed so far still samples at random uniformly in the C-space, so there is zero probability in our continuous space of sampling a configuration with the object being placed the same as in x_S, for instance. Moreover, if the robot is initially blocked by a huge obstacle preventing it from reaching any pushing configuration, the non-holonomic RRT fails to solve any manipulation scenario. Thus, we have to revise sampling in our C-space.

3.2.2 Sampling in the DAMA C-space

We just reviewed that sampling in our composed C-space cannot be done uniformly at random anymore. Sampling configurations has to respect important low-dimensional subspaces of the DAMA C-space. For instance, our RRT algorithm has to be able to only move the robot for a certain configuration within one iteration.

Barry solves this problem by defining and using primitive projection functions. For each primitive p_i, she defines a projection function $f_i(x_I, x_S)$, which projects x_S to some constrained subspace defined by the primitive p_i and the configuration x_I. The resulting configuration should be reachable for the respective primitive within the ESP. For instance, the primitive projection function for `Transit` would return the robot position defined by x_S and the object positions defined by x_I. Altogether, we sample x_S uniform at random, pick the nearest neighbor x_T, apply a randomly chosen primitive projection function $f_i(x_I, x_S)$ and connect from x_T towards the projected configuration. Formally, as stated by Barry, the set of primitive functions has to ensure that there is a non-zero probability of the ESP returning every possible set of primitives.

Algorithm 6
SampleDama(X)
Input: C-space $X = R \times O_1 \times ... \times O_n$
Output: Sample x_S respecting important subspaces of our DAMA problem

```
1: $r \leftarrow$ PickRand($\{0, ..., n\}$)
2: for all $i \in \{0, ..., n\}$ do
3:     if $i = r$ then
4:         FillSubspace($X, x_S, i,$ PickRand($\{\texttt{rand}, \texttt{goal}\}$))
5:     else
6:         FillSubspace($X, x_S, i,$ PickRand($\{\texttt{start}, \texttt{goal}, \texttt{free}\}$))
7:     end if
8: end for
9: return $x_S$
```

In this work, however, we developed another approach, which should yield similar results, but is more direct and powerful: Right from the start, we construct each subspace of our sample individually. Instead of generating a fully uniformly random sample, from the $(n + 1)$ subspaces of the composed C-space X we first pick one. Only this subspace can be filled with random values, or the values from the respective subspace of the overall DAMA query's goal configuration. All the other subspaces will be filled with either the DAMA initial configuration, the goal configuration, or the indication of being `free`. Algorithm 6 illustrates the basic sampling procedure. The indication of `free` for a particular subspace means that it will be overwritten by the respective subspace of any configuration when searching for the nearest neighbor in the next step.

Compared to Barry's approach of using a primitive projection function after sampling and determining the nearest neighbor, our nearest neighbor search can really rely on the sample not changing anymore, thus the neighborhood will be well-defined. Moreover, we intrinsically have the opportunity of sampling a whole goal configuration, which is needed for convergence of the single RRT-based planner. Also, when dealing with more complex puzzles and manipulation of multiple objects, sampling combinations such as $(\texttt{free}, \texttt{goal}, \texttt{rand}, \texttt{free})$ can guide the planner to connect from a certain configuration, in which object 1 is in its goal configuration, towards a configuration in which only object 2 has to move somewhere, completely ignoring the position of object 3 when searching for the nearest neighbor. However, the downside of this approach compared to Barry is the fine tuning of many parameters, as we do not want to pick uniformly at random from the sets shown in Algorithm 6. For instance, when picking a subspace to generate random values for, we should guarantee that subspace 0 for the robot will be chosen with appropriate frequency, as only this yields a single `Transit` of the robot.

3.2.3 Distance Metrics for the DAMA C-space

As the DAMA C-space is composed of robot and object configuration spaces, we have to revise our distance metric we discussed in Section 2.2.1, which dealt with only a single movable component in $SE(3)$. Thus, for each subspace S_i of our DAMA C-space $X = R \times O_1 \times ... \times O_n = S_0 \times ... \times S_n$, we can define an individual distance metric ρ_i as discussed beforehand. By following Barry's approach of using the maximum distance of all subspaces, we can combine these subspace metrics to form our overall distance metric ρ. Formally, the distance from configuration $x_I \in X$ to $x_F \in X$ can be defined as

$$\rho(x_I, x_F) = \max_{i \in \{0,...,n\}} \rho_i(x_{I,i}, x_{F,i}), \tag{3.6}$$

with $x_{j,i}$ denoting the projection of configuration x_j onto subspace M_i. Alternatively, instead of taking the maximum of all subspaces, we can also sum them up, yielding the following metric:

$$\rho(x_I, x_F) = \sum_{i \in \{0,\ldots,n\}} \rho_i(x_{I,i}, x_{F,i}). \tag{3.7}$$

However, there is a conflict when defining an appropriate distance metric for our DAMA C-space. On the one hand, as stated out by Barry, the distance between two configurations should reflect how likely both configurations are together members of either $\mathcal{C}_{free}$ or $\mathcal{C}_{obs}$. Thus, the distance metric should consider each subspace individually. On the other hand, our distance metric is responsible for selecting the nearest neighbor from which we propagate towards our sample, and we want that movement to be short in order to minimize the path length and thereby also the probability of colliding. However, following Equation 3.6 or 3.7 does not reflect the real motion in our non-holonomic space at all. For instance, if the distance between each subspace ρ_i is larger than zero but still small, the overall distance will be small, too. Nevertheless, the empty space planner returns a long path, in which the robot has to slightly move all objects.

In practice, executing the empty space planner to return an accurate measure for the robot distance yields immense computational costs for the nearest neighbor search. Therefore, we utilize either Equation 3.6 or 3.7 to give us a heuristic function for $\rho(x_I, x_F)$. Although Barry could only prove that her algorithm was exponentially convergent for taking the minimum distance (Equation 3.6), in this work, we use the sum metric (Equation 3.7). The latter yields better results in our scenarios, and somewhat more reflects the non-holonomic system we are dealing with, as every subspace is really contained in the resulting distance approximation $\rho(x_I, x_F)$.

3.2.4 DARRT Algorithm

With the previously discussed issues we can extend the non-holonomic RRT to yield the DARRT algorithm, which is shown in pseudo-code in Algorithm 7. Similar to the sampling-based RRT, it starts with an initial configuration and then iteratively samples from the C-space, computes the nearest neighbor, and extends the tree by utilizing the empty space planner from the nearest neighbor towards the sample. The resulting collision-free trajectory then yields new vertices and edges, and we continue until we reach a goal configuration. Finally, if the algorithm finds a solution in finite time, we can obtain a trajectory sequence from the initial to a goal configuration.

Algorithm 7
DARRT($X, \{B_0, ..., B_q\}, \{p_0, ..., p_m\}, x_0, X_G, \{\rho_0, ..., \rho_n\}$), adapted from Barry [6, p. 60]
Input: C-space $X = R \times O_1 \times ... \times O_n$, Fixed obstacles $\{B_0, ..., B_q\}$, Primitives $\{p_0, ..., p_m\}$, Initial configuration x_0, Goal set X_G, Distance metrics $\{\rho_0, ..., \rho_n\}$
Output: Trajectory sequence from x_0 into X_G

```
1: $V \leftarrow \{x_0\}$
2: while $V \cap X_G = \emptyset$ do
3:     $x_S \leftarrow$ SampleDama($X$)
4:     $x_T \leftarrow \arg\min_{v \in V} \rho(v, x_S)$
5:     $\{\tau_0, ..., \tau_l\} \leftarrow$ Extend($x_T, x_S, X, \{B_0, ..., B_q\}, \{p_0, ..., p_m\}$)
6:     $V \leftarrow V \cup \bigcup_{\tau \in \{\tau_0, ..., \tau_l\}} \bigcup_{\alpha \in [0,1]} \tau(\alpha)$
7: end while
8: return ExtractTrajectorySequence($V$)
```

3.3 Advanced Diverse Action RRT Algorithms

In this section, we extend the basic DARRT algorithm for solving DAMA problems. We presend two advanced algorithms based on DARRT, which are capable of solving more complex DAMA problems. These are the bidirectional DARRT algorithm and the hierarchical DARRT algorithm.

3.3.1 Bidirectional DARRT

Similar to extending RRT to RRTConnect (Algorithm 2), we can extend DARRT to DARRTConnect. Recall the idea of growing two trees, referred to as bidirectional RRT-based planning discussed in Section 2.3.1.2. Combining DARRT (Algorithm 7) and RRTConnect (Algorithm 2) leads to a straightforward version of the DARRTConnect algorithm, presented with pseudo-code in Algorithm 8.

When planning bidirectional in a non-holonomic system like in our DAMA problem, we have to make a major refinement to the Extend function. The empty space planner has to ensure to always execute primitives with forward control, otherwise pushing when extending in the backward tree becomes pulling. Instead of redefining each primitive with inverse control, we can apply the following strategy: When extending backwards, the empty space planner swaps starting and end configuration, propagates as usual, and finally reverses the trajectory sequence and each trajectory inside. Thus, we obtain a path for the backward tree, which still follows forward control.

Algorithm 8
$\text{DARRTC{\scriptsize ONNECT}}(X, \{B_0, ..., B_q\}, \{p_0, ..., p_m\}, x_0, X_G, \{\rho_0, ..., \rho_n\})$, adapted from Barry [6, p. 68]
Input: C-space $X = R \times O_1 \times ... \times O_n$, Fixed obstacles $\{B_0, ..., B_q\}$, Primitives $\{p_0, ..., p_m\}$, Initial configuration x_0, Goal set X_G, Distance metrics $\{\rho_0, ..., \rho_n\}$
Output: Trajectory sequence from x_0 into X_G

```
1: $V_a \leftarrow \{x_0\}$
2: $V_b \leftarrow \{\text{Sample}(X_G)\}$
3: $F \leftarrow$ **true** // *True when extending forwards, false backwards*
4: **while true do**
5:   **if** $F$ **then**
6:     // *Add a goal configuration to the backwards tree*
7:     $V_b \leftarrow V_b \cup \{\text{Sample}(X_G)\}$
8:   **end if**
9:   $x_S \leftarrow \text{SampleDama}(X)$
10:  $x_T \leftarrow \arg\min_{v \in V_a} \rho(v, x_S)$
11:  $\{\tau_0, ..., \tau_l\} \leftarrow \text{Extend}(x_T, x_S, X, \{B_0, ..., B_q\}, \{p_0, ..., p_m\}, F)$
12:  $V_a \leftarrow V_a \cup \bigcup_{\tau \in \{\tau_0, ..., \tau_l\}} \bigcup_{\alpha \in [0,1]} \tau(\alpha)$
13:  **if** $l > 0$ or there are new configurations in $\tau_0$ **then**
14:    // *Extend $V_b$ towards $V_a$*
15:    $x_T \leftarrow \arg\min_{v \in V_b} \rho(v, \tau_l(1))$
16:    $\{\sigma_0, ..., \sigma_k\} \leftarrow \text{Extend}(x_T, \tau_l(1), X, \{B_0, ..., B_q\}, \{p_0, ..., p_m\}, \neg F)$
17:    $V_b \leftarrow V_b \cup \bigcup_{\sigma \in \{\sigma_0, ..., \sigma_k\}} \bigcup_{\alpha \in [0,1]} \sigma(\alpha)$
18:    **if** $\sigma_k(1) = \tau_l(1)$ **then**
19:      **return** $\text{ExtractTrajectory}(V_a, V_b)$
20:    **end if**
21:  **end if**
22:  $\text{swap}(V_a, V_b)$
23:  $F \leftarrow \neg F$
24: **end while**
25: **return** $\text{ExtractTrajectorySequence}(V)$
```

3.3.2 Hierarchical Approach

Both algorithms DARRT and DARRTConnect discussed so far can be described as *flat* algorithms, because they try to solve the DAMA problem in one step. However, as described in detail by Barry [6], we can break down the DAMA problem to planning for each manipulation primitive separately. For instance, for pushing an object, our robot first has to `transit` to the pushing contact point, and then `push` the object. However, if we know the contact point in advance, we can plan for `transit` and `push` separately. Algorithms, which solve the DAMA problem in multiple steps by breaking it down into smaller parts, can be referred to as *hierarchical* algorithms.

Solving the DAMA problem hierarchically is closely related to its multi-modal nature, which we discussed in Section 2.4. For the DAMA problem, we can define a mode family for each primitive. Each mode of a mode family then defines the contact between the robot and the respective object, and the locations of the objects not being manipulated. For instance, a mode of the mode family `push` specifies the pushing configuration (i.e., the direction), and the configuration of all the other objects.

Closely related to the research of re-grasping an object, Barry [6]'s hierarchical approach is to first plan a sequence of mode families and then plan within each mode. Usually, transfer primitives can only transition to a transit primitive, so Barry further reduces her hierarchical approach to first plan a sequence of transfer mode families, and then plan each set of transfer and transit trajectories. Concretely, she first plans an object path by executing a flat planner as usual, but checking collisions only for the objects. In the next step, Barry converts the object path to a sequence of manipulation transfer primitives, which defines the sequence of primitive subgoals for the hierarchical algorithm. For each primitive subgoal, a flat planner tries to find a path to a configuration in which that primitive is applicable. Finally, Barry's DARRTH algorithm solves for the solution path from the last subgoal to the final goal set.

In this section, we will present a slightly different, simple version of the DARRTH algorithm proposed by Barry [6]. Note again that despite the same name, the DARRTH algorithm presented in this work is not similar to the implementation proposed by Barry [6]. In contrast to Barry's algorithm, our DARRTH algorithm will be able to handle manipulating multiple objects, without the requirement of defining new primitives for each object we want to manipulate. Furthermore, instead of only planning for mode families, our object path will provide us with the actual mode sequence, i.e. with the actual contact points and the location of the objects not being manipulated. Here are the steps taken by our DARRTH algorithm:

1. Disable collision checking between robot bodies and all movable objects
2. Execute the flat planner to find a solution from x_0 to some $x_G \in X_G$
3. Extract the object path from the solution path, yielding the list of configurations $\{o_1, o_2, ..., o_k\}$
4. Enable collision checking between robot bodies and all movable objects
5. Set SolutionPath = $\{\}$
6. For each object configuration $o_i \in \{o_1, o_2, ..., o_k\}$:
   a) If $o_i = o_1$, then $x'_0 = x_0$, otherwise set $x'_0 =$ SolutionPath.back() to the latest configuration added to our solution path
   b) If $o_i = o_k$, then $X'_G = X_G$, otherwise set X'_G to o_i with arbitrary robot pose
   c) Execute the flat planner to find a solution from x'_0 to some $x'_G \in X'_G$
   d) Append the solution path to SolutionPath

The object path is extracted from a certain solution path by appending the respective configuration, each time we first apply a certain manipulation action to an object. Also, the goal configuration is appended. Thus, the object path is the minimal set of the solution path, which still can reconstruct all object movements correctly. The object path is also smoothed with respect to applying the same primitive to a single object multiple times in a row. For instance, pushing a certain object from A to B and then to C, before manipulating any other object, will be shortened to pushing it directly from A to C.

The downside of this algorithm planning the actual modes clearly is that the objects are not guaranteed to be manipulable at all points along the object path. When planning among movable objects, as pointed out by Berg et al. [8], an object path should guarantee a path that is collision free for the objects, but also that the moved object stays manipulable at all points along the path. As we plan the object path with disabled robot-object collisions, the robot can place objects in an intermediate step, where they are not reachable when enabling the collision checking again.

Finally, let us conclude with two more notes concerning the hierarchical algorithm. Firstly, depending on the flat planner we use, we refer to either DARRTH or DARRTHConnect. In most cases, we will only use the latter, bidirectional planner. Moreover, we can specify two different restart conditions for the hierarchical algorithm. On the one hand, we can restart the inner execution of the flat planner solving subgoals. On the other hand, we can also restart the entire algorithm to plan for a new object path.

4 Diverse Action Manipulation Experiments

In this chapter, we present results for the DAMA algorithms DARRT, DARRT-Connect, and DARRTHConnect, which were introduced in Chapter 3. To show that our algorithms are capable of solving for various robots, environments, and manipulation primitives, this chapter is divided into two worlds.

In the first world, a simple holonomic mobile robot manipulates three objects in a simulated two-dimensional world. We will compare the DARRT with the DARRTConnect algorithm in a challenging scenario.

The second world features a ten degrees of freedom joint robot manipulating three bottles in a bartender-like scenario, resulting in a configuration space of nineteen dimensions. Besides `transit`, the robot will be able to `pickup` objects, transfer the rigidly grasped object (`transfer-rigid`), and push objects with the interior (`push-interior`) and the exterior surface (`push-exterior`) of the hand. We will compare the flat DARRTConnect algorithm with the hierarchical counterpart DARRTHConnect in a challenging scenario. Furthermore, we will solve exactly the same scenario with the real robot in a real environment, and present the steps taken from planning to successful execution.

The following experiments were performed on a Dell Latitude E6520 notebook with an Intel Core i5-2520M processor (2.5 GHz), 8 GB working memory, Nvidia NVS 4200M graphics card and an OCZ Vertex 2 SSD running Ubuntu 12.04 and the Robotics Library [2] with our DAMA algorithm software added (roughly 5.000 lines of code).

4.1 World 1: Simple Mobile Robot

In this world, a simple holonomic mobile robot already introduced in Section 3.2.1 (Figure 3.1) has to solve DAMA problems in a two-dimensional five-by-five meter square world with multiple obstacles and objects. The robot can `transit` and `push` objects within the world.

	solved	#iter	#vert	tGlobal	tSam	tNN	tProp	tConn
DARRT	81%	10,911	13,387	28.5s	0.3%	84.1%	0.6%	8.5%
DARRTCon.	97%	7,255	14,588	21.4s	0.3%	72.9%	0.8%	10.1%

(a) **Algorithm properties**. Columns (left to right): proportion of terminated runs, number of iterations (averaged on the terminated runs, as are the following values), number of vertices, time needed for solving, proportion of time spent on sampling, nearest neighbor search, propagating using the empty space planner, and on connecting configurations or path truncation using the collision checking module.

	#vert	lengthGlob	lengthManip	#Transit	#Push-Mobile
DARRT	101	22.9m	3.3m	4.8	3.8
DARRTCon.	119	27.8m	3.6m	5.5	4.5

(b) **Solution path properties**. Columns (left to right, averaged on the terminated runs): number of vertices, length of the entire solution path, length of the path segments in which objects are manipulated, number of times each primitive is executed (only counting primitive transitions, i.e. removing successive duplicates).

Table 4.1: **Averaged performance of DARRT and DARRTConnect on Scenario 1**. We ran each algorithm 100 times starting with a different random seed. For both algorithms we imposed a time limit of 3 minutes to solve this scenario.

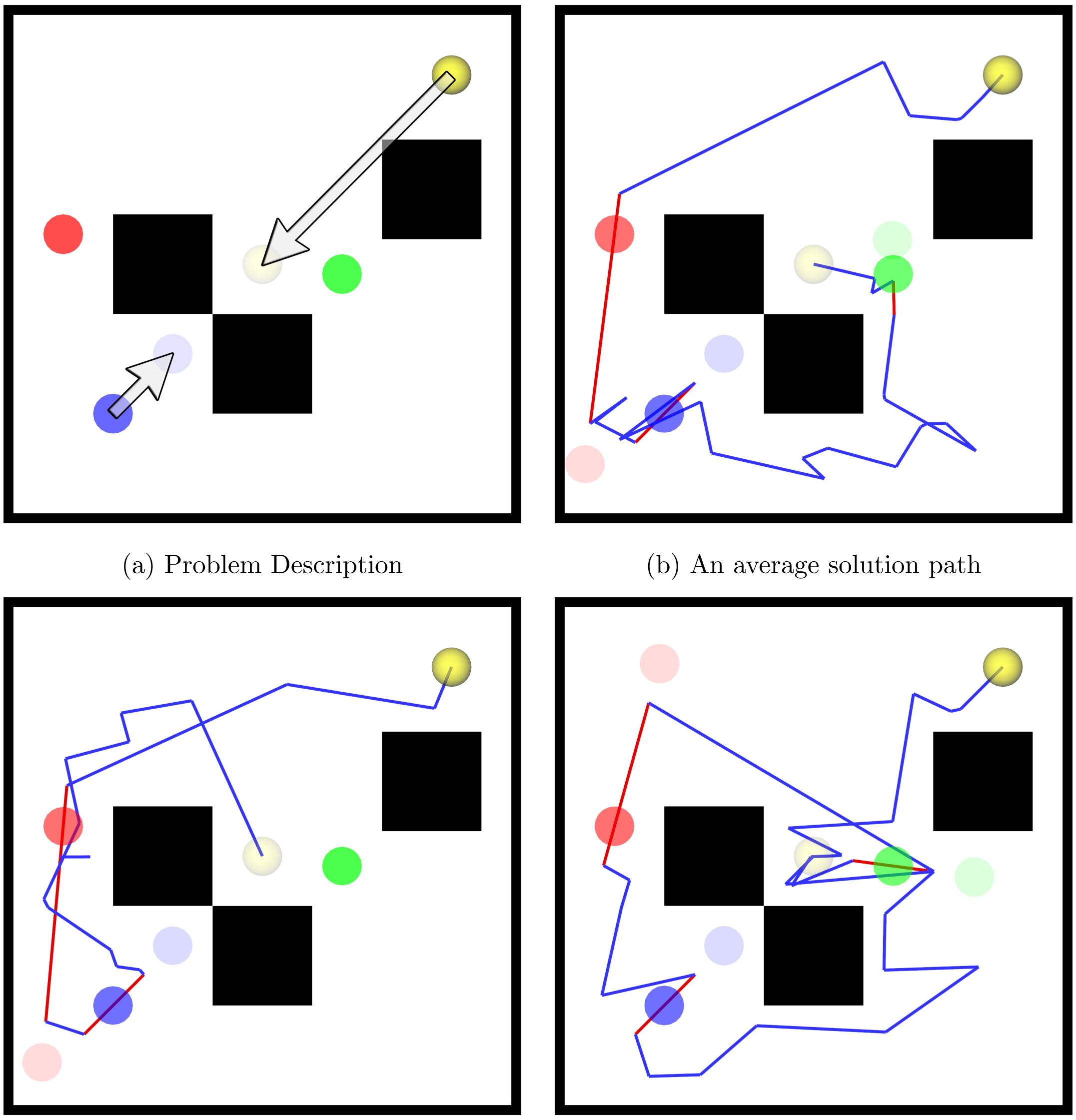
4.1.1 Scenario 1: Manipulation with Three Movable Objects

Figure 4.1a illustrates the problem description of this scenario. Within a square world with three obstacles, the robot and the blue object have to move to their corresponding goal. The goal set is infinite and uncountable, as are goal positions for the red and the green object. The challenging task of this scenario is to move the red or the green object (or both) before the robot can manipulate the blue object. Our DAMA algorithm of course does not have any knowledge about the objects blocking the path and therefore need to be moved. After the robot has moved the blue object to its goal, however, he also has to transition back to its goal, avoiding obstacles and possibly moving objects once again.

4.1.2 Discussion and Evaluation

We run this scenario 100 times with each algorithm, DARRT and DARRTConnect. Thereby, we enforce a time limitation of three minutes to solve this scenario. After three minutes, nearest neighbor search consumes more than 80% of the time, therefore making the algorithm much less efficient.

Table 4.1 presents notable averaged performance values and their detailed description when running both DAMA algorithms 100 times. Let us now discuss two major differences in performance between DARRT and DARRTConnect.



(a) Problem Description (b) An average solution path

(c) Shortest path found (d) Path found the fastest

Figure 4.1: **Scenario 1: Problem description and solution paths**. We ran DARRT and DARRTConnect separately 100 times. All illustrated solution path were computed by DARRTConnect. The individual goal configurations are shown more transparently than the initial configurations. Paths drawn in red correspond to `push`, the blue ones to `transit`. (a) In this scenario, the robot (yellow) and the objects (red, blue, green) start at their indicated positions. The goal is specified only for the robot and the blue object, which have to move to their illustrated goal positions. In order to move the blue object, the robot has to move either the red or the green object first. (b) An average solution path found after 26.2s with a solution path length of 18.4m. (c) The shortest solution path found after 28.4s with a solution path length of 14.8m. (d) The solution path found the fastest after 191ms with a solution path length of 22.7m.

Whereas DARRT succeeded in 81 out of the 100 runs, DARRTConnect found a solution for 97 of the 100 runs within three minutes. Moreover, DARRTConnect needs less time on average to solve this scenario, and also less iterations. This is remarkable, as intuitively, spanning a tree starting from the infinite and uncountable goal set does not seem to be promising at all. However, the path found the fastest, but also the path with the shortest solution path length, was computed by DARRTConnect.

When investigating not the performance on finding a solution path, but rather the properties of the solution path itself, DARRT seems to yield better results than DARRTConnect in this scenario. Compared to DARRTConnect, DARRT finds paths, which are about 18% shorter than the ones computed by DARRTConnect on average. Thus, DARRT paths consist of less `transit` and `push` transitions.

To conclude, it is worth mentioning that with our DAMA algorithm implementation, we can solve this scenario in 191ms (Figure 4.1d), and only fail to find a solution in about 3% of the cases for a time limitation of three minutes using DARRTConnect. Whereas DARRTConnect seems to be more efficient in finding a solution at all, DARRT is more effective in terms of finding the shorter solution path.

Note that in this scenario, a hierarchical approach like DARRTHConnect would not be much useful, as the object path would not reflect that the robot needs to move objects he would otherwise collide with, which is the key to finding a solution.

4.2 World 2: Joint Meka Robot JAMES

In this world, we simulate and really control a ten degrees of freedom joint robot solving DAMA problems. Before introducing the scenario for evaluating DAMA algorithms, we will start with an introduction to the JAMES project (Joint Action for Multimodal Embodied Social Systems). Next, we present the robot hardware and important details concerning some modifications and extensions of our DAMA software, in order to solve DAMA problems with this robot.

4.2.1 The JAMES Project

The research of the JAMES project (Joint Action for Multimodal Embodied Social Systems) [44, 11] is focused upon robots showing socially appropriate behavior while dynamically interacting with multiple humans [12]. In the main demonstration scenario, as shown in figure 4.2, the robot works as a bartender and interacts with human guests by taking orders and serving drinks. Here, detection of social aspects, signals and activities of a single persons but also of human groups are crucial in order for a robot to fulfill the role of a human bartender in a socially appropriate way. However, also traditional robot tasks have to be solved, for instance generating a non-colliding safe robot arm movement.

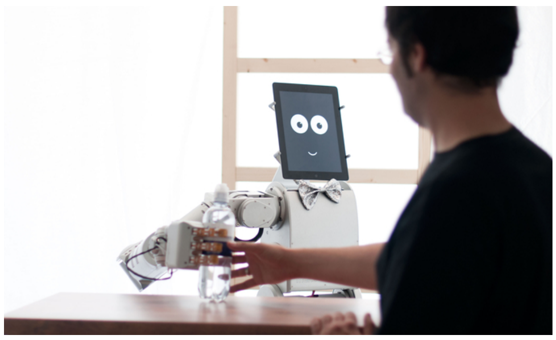
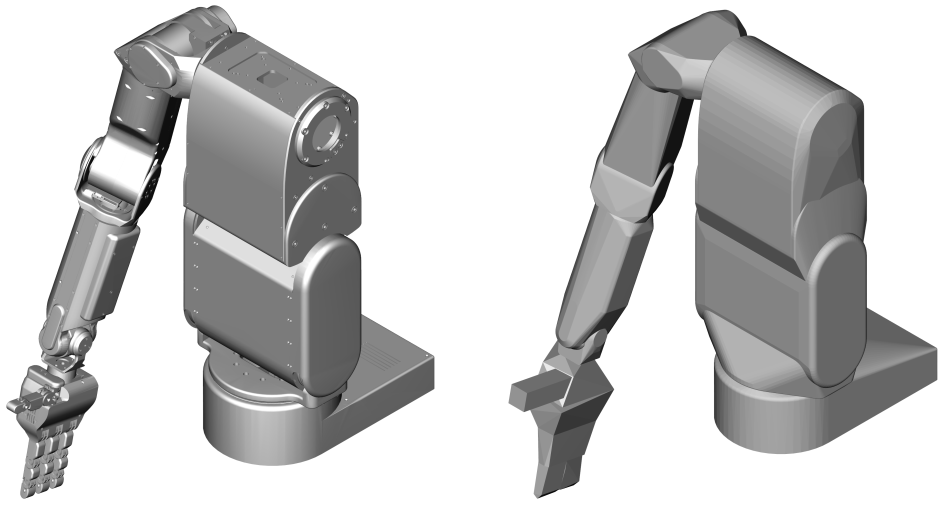


Figure 4.2: **Barkeeper JAMES acts as a bartender in a bar scenario**.

With regard to our work, this bartending scenario intrinsically demands diverse action manipulation. The robot not only has to manage differently shaped drinks, he also has to cope with their various arrangements combined with limited end-effector accessibility. For example, if the robot decides to clean up the table after a group of guests left the bar, the robot might find itself in a situation where empty bottles are clumped up in a way that disallows straight-forward grabbing, thus requires some other kind of manipulation motion like pushing with the finger tips or the palm. Furthermore, introducing new manipulation capabilities like pushing with various hand parts really expands the reachable workspace of the robot. For instance, the robot could push a bottle towards him using the finger tips before grasping is applicable.

4.2.2 Robot Hardware

The robot was designed by Meka Robotics[1] and features force-torque control at each joint, absolute joint angle encoders and compliant, human-safe actuators by placing a spring between the motor and the load. Our robot for the JAMES project is composed of the T2 Humanoid Torso, the A2 Compliant Arm and the H2 Compliant Hand (see [1] for detailed information). Whereas the arm has seven degree of freedom (7-DOF), the torso has two degrees of freedom (2-DOF) with two of the three hip joints being mechanically coupled. The Meka hand, as shown in figure 4.10, features a dual durometer urethane construction in approximately human size and a well suited deformable link structure for manipulation tasks in unstructured environments. Its five Series Elastic Actuators drive twelve joints, one for each finger and one for thumb abduction, resulting in five actuated degree of freedom (5-DOF). All in all, our barkeeper Meka robot JAMES is a 14-DOF



(a) Original Meka robot model (b) Convex decomposition (except thumb)

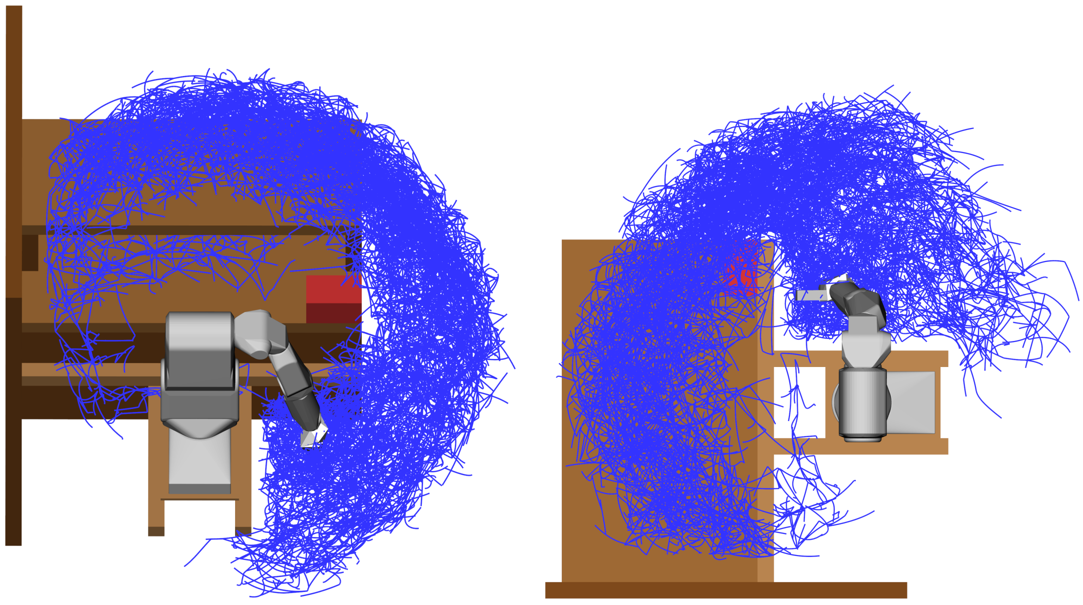
Figure 4.4: **Meka robot model used for simulation**. (a) The original CAD model of the Meka robot. (b) Meka model obtained after applying a 3D mesh approximate convex decomposition of the original model [38, 14, 15, 13]. This decomposition allows collision detection to run much faster and more robust due to a drastically decrease in mesh triangles while increasing the volume of the model. However, the thumb was manually modified and is not part of the convex decomposition. Both models show the starting pose of our robot for this world.

robot system.

4.2.3 Implementation Details

Before we can apply our DAMA problem to the Meka robot, we have to discuss the robot model and several modifications and extensions to our DAMA algorithm.

Let us first take a look at how the robot subspace of the C-space looks like. For the Robotics Library [2], the Meka robot is modeled as a 10-DOF robot, waiving the hand and including all three hip joints of the torso. For planning our path of actions, we indeed do not have to consider the joints of the hand and will leave them static pointing straight ahead all the time. Thus, planning will perform more efficient, as the dimension of the robot C-space significantly decreases. Really controlling the hand will take place after planning during execution time. However, note that collision checking will not be as accurate. Furthermore, as the torso consists of three free hip joints in our Robotics Library model, we have to consider the mechanical coupling of two of them within our algorithm manually. Concretely, we must guarantee that at any point during computation, both joints have the same value. This affect the sampling step, as well as the inverse kinematics.



(a) Rear upper view (b) Top view

Figure 4.5: **Reachable workspace of the Meka robot**. We can visualize approximately the workspace of the Meka robot by sampling only configurations for `transit`. Here the result is shown after roughly 7.000 iterations and a tree consisting of 10.000 edges.

Figure 4.4 illustrates the robot model and its starting pose we will use throughout this world for simulation. The detailed CAD model shown in Figure 4.4a will be used for visualization only. A 3D mesh approximate convex decomposition of this original model shown in Figure 4.4b is the actual model when checking collisions, following the work of [38, 15, 14, 13]. For collision checking to be computationally-efficient in sampling-based motion planning, it is crucial to represent objects, obstacles, and the robot model as sets of convex polyhedra. Therefore, for collision checking, we not only decomposed the robot model, we also replaced the movable bottles presented shortly with a cylinder of equal radius and height. Moreover, all static obstacles are modeled as simple boxes.

For our joint robot we will sample, compare and interpolate configurations in the ten-dimensional joint space R^{10}, instead of planning for a rigid body defined in $SE(3)$, as discussed in Section 2.2 concerning sampling and distance metrics. Concretely, sampling for the robot subspace in the composed DAMA C-space will be done uniformly at random per joint within the joint limits. For computing the distance in the robot subspace, we apply forward kinematics to our joint configuration and then use the formulas discussed in Section 2.2.

Figure 4.5 illustrates an approximation of the reachable Workspace of the Meka robot for its bartending environment. For collision checking, , we use the environment shown there, including the wall on the left, as it is part of the real environment. Also, the workspace is limited by the minimum and maximum values for each joint, which we defined appropriately. From this illustration, we can conclude that the robot can hardly reach the corners of the upper surface and nearby center areas of the upper and the lower support surface. Furthermore, when constructing the tree, the robot needs a lot of time to first reach the lower support surface at all. With objects placed there, it will be even harder to find any valid paths. Thus, we waive the lower support surface of the bar environment for our future scenarios and focus our attention on the upper support surface with possible extensions on top of it.

Now we can introduce the manipulation poses and their derived primitives we defined for our Meka robot. Figure 4.6 illustrates the three hand poses we will use throughout this world for manipulating objects. These are grasping, pushing with the palm or the interior surface of the hand, and pushing with the exterior surface of the hand. The respective primitives are `pickup` and `transfer-rigid` (both utilize the grasping pose), `push-interior`, and `push-exterior`. Whereas pushing an object is fully constrained concerning the contact configuration or direction of the hand, when picking up or transferring a rigidly attached object, the grasp can be from any direction. Thus, together with `transit`, the robot has five primitives available.

Let us now describe the implementation of the five primitives, namely their respective functions isUseful and propagate. Whereas `transit` is implemented as usual, both primitives `push-interior` and `push-exterior` follow the basic `push` primitive, except for the different pushing pose. Moreover, we restrict a single push propagation to a distance of 10 centimeter, because path interpolation is done in joint space, and therefore, the object being pushed will not follow a straight line in Cartesian space. In practice, however, for a distance below 10 centimeter, the deviations are negligible and the object successfully reaches its goal. Of course, multiple pushes can be concatenated to push along a longer distance. The only downside to this approach is the accumulated computation of the inverse kinematics for each push propagation, which slows down the algorithm a bit. Finally, the new primitives `pickup` and `transfer-rigid` are implemented as follows:

- `Pickup`:

  isUseful returns *true* only if all objects are placed on a support surface in x_S and at least one object either moves at least 20 centimeter on its support surface or has a different height value between x_S and x_G.

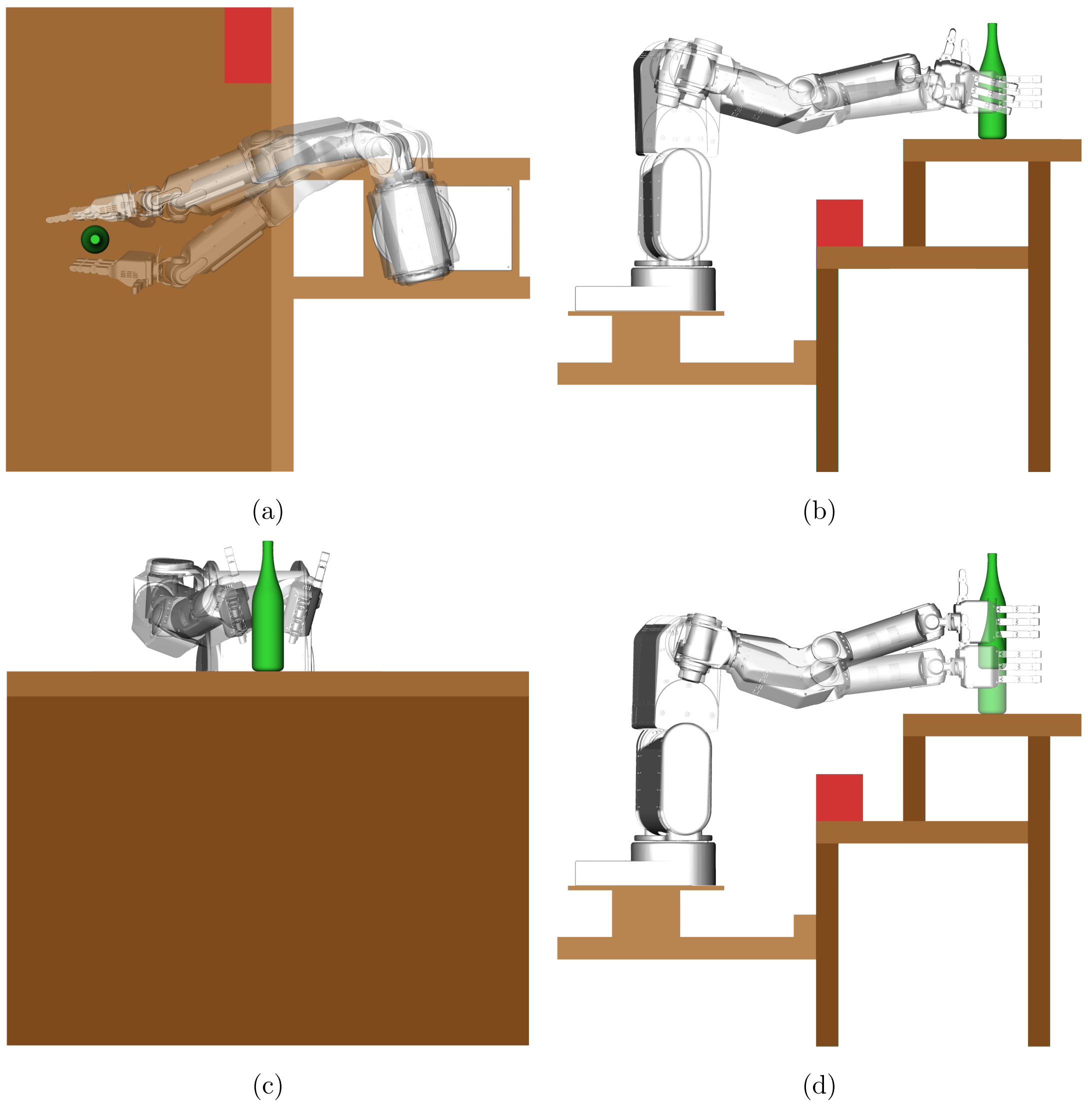


Figure 4.6: **Manipulation poses and their derived primitives for our Meka robot**. (a) Comparison of all three hand poses for manipulation viewed from the ceiling: grasping (top left), push-interior (top), push-exterior (down). (b) For grasping an object, the hand has to be more stretched out, compared to pushing an object. (c) Pushing with the interior (left) and exterior (right) surface of the hand. This front view illustrates the slightly angled hand poses used for smoothly pushing an object with the Meka robot in practice, shifting the contact point near the center of mass of the object. (d) Both primitives `pickup` and `transfer-rigid` use the same grasp hand pose shown here. However, before `transfer-rigid` is applicable, the object has to be picked up a certain height, as illustrated.

propagate returns the following trajectory sequence starting from x_S and manipulating the nearest object to the robot, which should be picked up: We first compute a grasping configuration via inverse kinematics (note that there are infinite and uncountable grasping configurations, as we can grasp from any direction). We then `Transit` to this location, if not already there, and append the trajectory from this configuration to the configuration, in which the robot arm and the object is 10 centimeter higher. The latter configuration with its grasp will also be computed via inverse kinematics.

- `Transfer-rigid`:

isUseful returns *true* only if an object is not placed on a support surface in x_S and either this object or the robot moves between x_S and x_G.

propagate returns a trajectory sequence from x_S towards x_G by computing the final grasp configuration via inverse kinematics when transferring the rigidly attached object to its goal and then propagate to this location. Note that for x_S, we can ensure that we are already grasping the respective object.

In order for our joint robot to apply these primitives, we need to map configurations for the end-effector in Cartesian space to robot configurations in joint space. Recall the brief outline we gave in Section 2.1.3 concerning the concept of inverse kinematics. There are two modifications of the standard iterative approach for solving inverse kinematics worth mentioning in the context of the Meka robot. As there are usually infinite solutions to an inverse kinematics problem for the Meka robot, we further constrain the iterative process to stay away from joint limits, both the minimum and maximum values. This posture optimization not only tries to find a more natural robot pose, it also ensures to stay away from configurations near singularities of the robot, if possible. Secondly, when computing the inverse kinematics for the Meka robot, we have to permanently couple the second and the third joint during the iterative approach, because the hip joints are mechanically coupled. The details of this procedure would require a more detailed introduction to inverse kinematics, which is out of the scope of this paper.

This world will feature multiple support surfaces, which requires the following adaptation to our sampling routine. When sampling a random position (`rand`) for an object, we will either sample a position in free space or on one of the support surfaces. Both, the volume defining the possible object locations in free space, as well as the dimensions of the support surface will be defined by the scenario or domain we are planning for. Moreover, when sampling in free space, in order for it to become a valid configuration, we also have to deliver a grasp configuration of the robot for the respective object. Thus, sampling now also involves computing the inverse kinematics.

Finally, in this world, we will use the distance metric defined by Equation 3.7 and add an additional 2 meters for each object which moved between both configuration, in order to somewhat incorporate that the robot path length increases significantly for each object to be moved (and so does the chance to collide). However, this is a manually tweaked heuristic and determined empirically.

4.2.4 Scenario 2: Rearranging Three Bottles

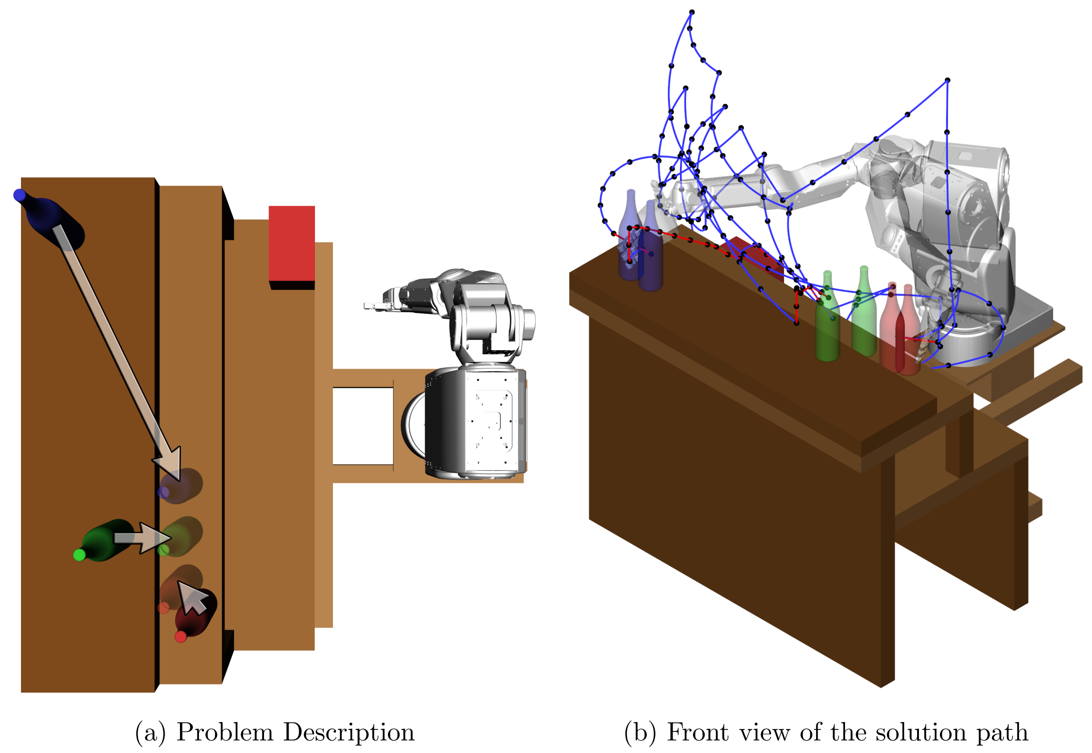
Figure 4.7a illustrates the problem description of this scenario. Starting at the plotted position, the robot has to rearrange three bottles as indicated in the Figure. Finally, he has to move back to his starting position.

There are many challenges involved in solving this task. First of all, the solution path will be quite long, as the robot not only has to find a collision-free path for raising and lowering his arm when starting and finishing the work, respectively, he also has to plan for many transfer primitives. All primitives are highly recommended for solving this task, if not required. For instance, the blue bottle cannot be grasped immediately in its initial configuration, first it has to be pushed with the palm towards the robot, in order to then be reachable for a grasp. However, it has to be grasped, in order to move it to the lower support surface. Moreover, if the green bottle has already been moved to its goal configuration, the red bottle can only be moved via `push-exterior`. Finally, the order in which objects are moved is crucial for finding a solution. For instance, if the blue bottle first reaches its goal configuration, manipulating the green bottle will become nearly impossible without colliding with the blue bottle. Compared to Scenario 1, however, the goal set only consists of a single goal configuration.

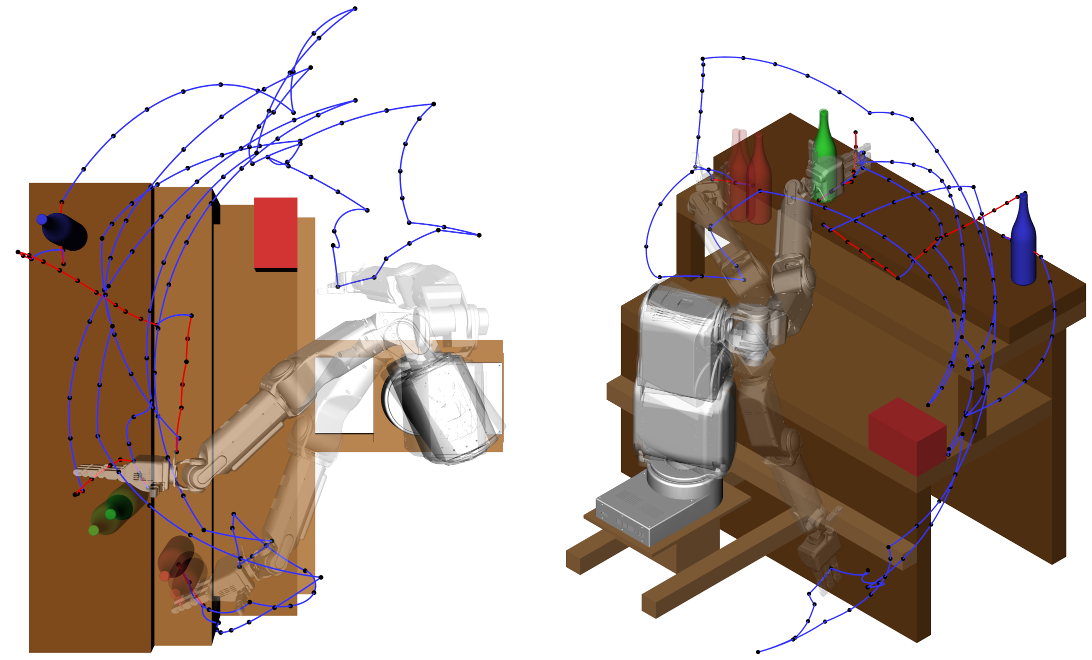
The C-space is highly dimensional with nineteen dimensions, ten for the robot subspace and three dimensions each for the translational description of the bottles. The volume of the free space, in which objects can be sampled, is roughly the intersection of the robot's reachable workspace and the area above the lower support surface.

4.2.5 Discussion and Evaluation

We run this scenario 50 times with each algorithm, the flat DARRTConnect and the hierarchical counterpart DARRTHConnect. Thereby, for DARRTConnect, we enforce a time limitation of fifteen minutes to solve this scenario.

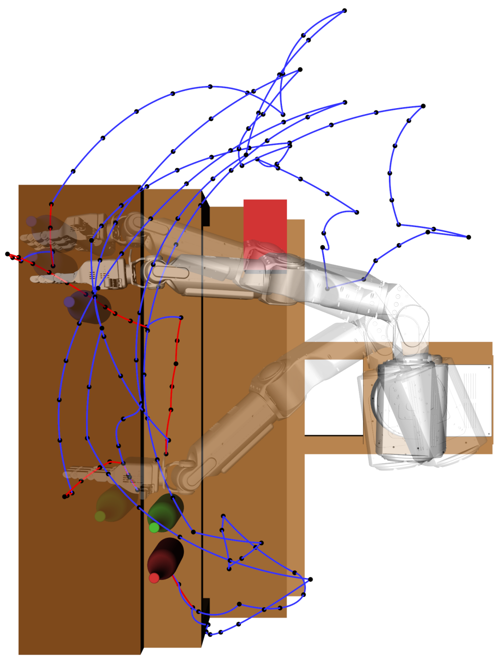


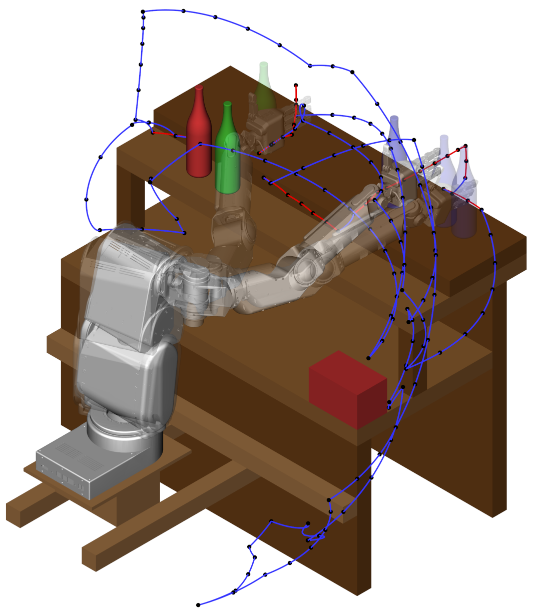
(a) Problem Description (b) Front view of the solution path

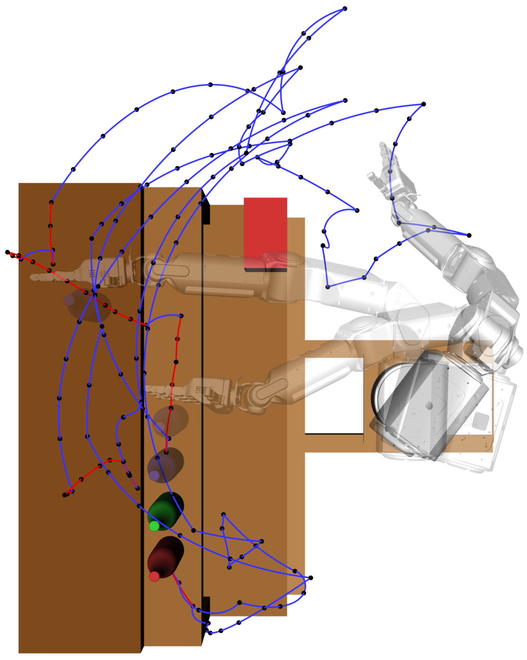


(c) Top view of the solution path (1/3) (d) Rear view of the solution path (1/3)

Figure 4.7: **Scenario 2: Problem description and a solution path (1/2)**. The illustrated, randomly picked solution path was computed by DARRTHConnect. Edges drawn in red correspond to transfer primitives being applied, the blue ones to the only transit primitive `transit`. Vertices of the solution path are drawn as small black dots. (a) In this scenario, the robot has to start and finish in the same illustrated position, after rearranging the three bottles all on the lower support surface as indicated by the arrows. (b)-(d) Different views of the solution path and robot poses.


(a) Top view of the solution path (2/3)


(b) Rear view of the solution path (2/3)


(c) Top view of the solution path (3/3)

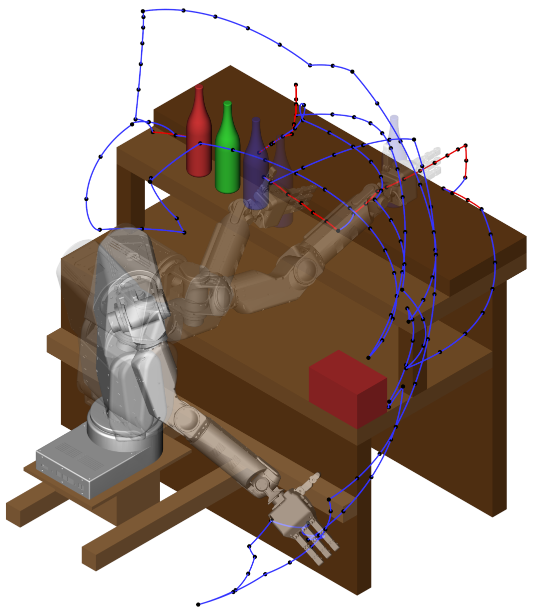

(d) Rear view of the solution path (3/3)

Figure 4.8: **Scenario 2: Problem description and a solution path (2/2)**. The illustrated, randomly picked solution path was computed by DARRTHConnect. Edges drawn in red correspond to transfer primitives being applied, the blue ones to the only transit primitive `transit`. Vertices of the solution path are drawn as small black dots. (a)-(d) Different views of the solution path and robot poses.

	solved	#iter	#vert	tGlobal	tIK	tSam	tNN	tProp	tConn
Flat	78%	7,546	16,509	514.7s	32.9%	3.7%	59.9%	29.4%	6.6%
Hier.	46%	4,046	10,688	131.8s	53.3%	8.4%	31.0%	45.4%	14.6%

(a) **Algorithm properties**. Description of columns see Table 4.1a. New: proportion of global time spent on solving inverse kinematics (tIK), which is needed for sampling and propagating.

	#vert	lengthGlob	lengthManip	#Tr.	#P-I	#P-E	#Pick	#T-R
Flat	181	13.8m	2.0m	8.3	2.6	0.7	4.5	4.5
Hier.	214	16.1m	2.5m	8.2	2.2	0.8	4.8	4.8

(b) **Solution path properties**. Description of columns see Table 4.1b. Primitives: `transit` (Tr.), `push-interior` (P-I), `push-exterior` (P-E), `pickup` (Pick), `transfer-rigid` (T-R).

Table 4.2: **Averaged performance of DARRTConnect (Flat) and DARRTH-Connect (Hier.) on Scenario 2**. We ran each algorithm 50 times starting with a different random seed. For DARRTConnect we imposed a time limit of 15 minutes, for DARRTHConnect five minutes for solving the object path and additional five minutes for each partial solution.

For DARRTHConnect, both solving the object path and solving for intermediate goals have a time limitation of five minutes each. We do not allow any restarts.

With these restrictions, DARRTConnect was able to succeed in 39 out of the 50 runs, and DARRTHConnect in 23 runs. However, the restrictions for a DARRTHConnect run were harder than for a DARRTConnect run. On average, DARRTHConnect failed after 362.8 seconds (roughly six minutes), none of them when searching for an object path. Note that even given 20 minutes and ten runs, the most simple DAMA algorithm DARRT was not able to terminate once.

Table 4.2 presents notable averaged performance values and their detailed description. Let us now discuss major differences in performance between DARRTConnect (Flat) and DARRTHConnect (Hier.).

Examining the algorithm properties presented in Table 4.2a, Hier. needs much less time to find a solution on average. Due to its hierarchical nature, it spends less time for nearest neighbor search, but still the inverse kinematic calculations consume more than half of the overall computation time. Despite the low percentage of solved runs compared to Flat, Hier. can be run twice as often. Overall, we can conclude that the hierarchical version solves this scenario more efficiently than its flat counterpart.

Together, inverse kinematics and nearest neighbor search consume 92.8% (Flat) and 84.3% (Hier.) of the overall computation time. Sampling and propagating

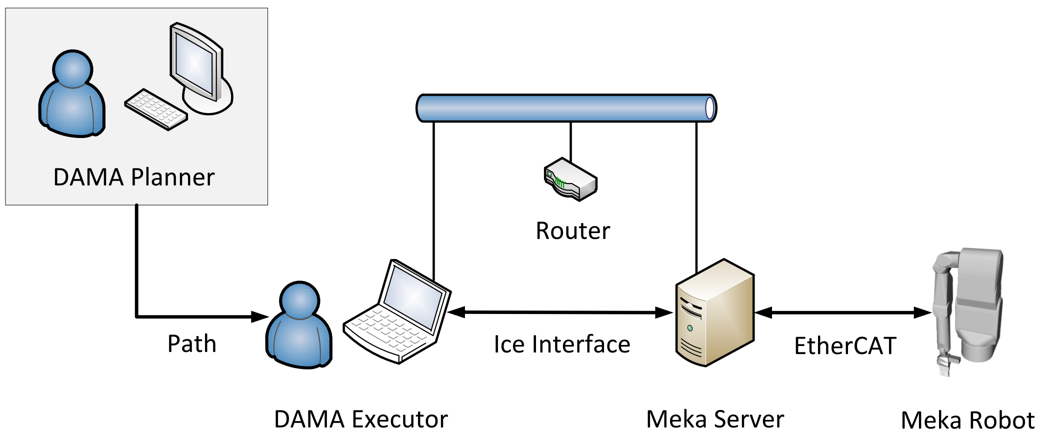


Figure 4.9: **Hardware environment for applying a DAMA problem on the real Meka robot**. The DAMA planner software transmits the planned path to the DAMA executor software, which communicates the interpolated path via an Ice interface to the Meka server, which finally controls the real Meka robot via EtherCAT.

nearly only consists of calculating the inverse kinematics in terms of time costs.

The solution path properties presented in Table 4.2b show that FLAT finds better solutions than HIER.. Note that only in at most 70% (80%) of the runs, we apply `push-exterior` to the red bottle. Furthermore, the number of `pickup` and `transfer-rigid` primitives applied are completely equal, as each `pickup` has to be followed by a `transfer-rigid`.

To conclude, DARRTHCONNECT is more efficient in finding a solution, but must be restarted more often. However, DARRTCONNECT is more effective in finding a good solution, but needs much more time and reaches its limits with respect to tree size and nearest neighbor search.

4.2.6 Execution on real Meka Robot

In this section, we present the steps needed to execute a planned path consisting of vertices and edges on the real Meka robot JAMES in practice.

Figure 4.9 illustrates the underlying architecture and hardware environment of this Section. So far, we discussed the DAMA planner software, which creates solution paths for our DAMA problem. For execution, we let the planner store the path in an external file and programmed a separate DAMA executor software component for path execution, which can ultimately run on a separate computer connected to the Meka server via an Ethernet connection. The Meka server controls the Meka robot via EtherCAT, an open real-time Ethernet-based fieldbus system, and offers a ZeroC Ice interface[1], in order to control the robot within the local area network the server is connected to. But for now, let us focus on the DAMA executor software.

Executing the planned path in a simulated robot environment or in the real world makes a huge difference. In practice, for example, joints might drive with different velocities, having different static frictions (stiction), and different maximum velocities. Therefore, with increasing distance between vertices of our solution path, it becomes more likely that we cannot ensure following the direct connection or shortest path between both vertices anymore.

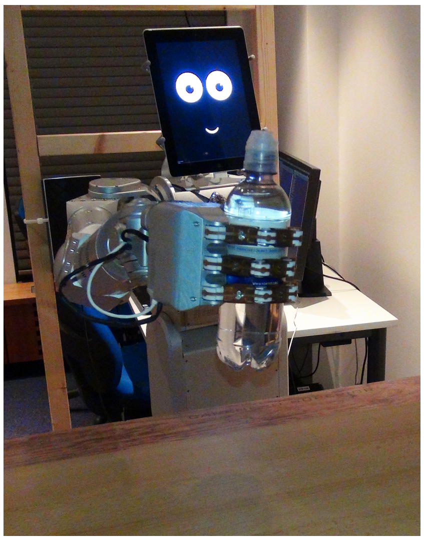


Figure 4.10: **Snapshot of the Meka robot solving a DAMA problem in practice**.

Thus, we need to manually process our solution path generated by the planner in order to obtain the same path but with higher resolution, which we can then utilize to steer our robot. In our rather simple approach, we will avoid controlling or regulating the robot using a PID controller, for example. The Meka robot already internally applies these mechanisms, and an additional controller might only cause problems, such as undesirable oscillations. Furthermore, we have to restrict ourselves to linear interpolation between points, which defines the only path our planner guarantees to be collision-free, avoiding any shortcuts or smooth transition trajectories between points which might contain obstacles.

Now let us briefly discuss our approach. For each joint we define an appropriate velocity limitation, measured in degree per second. Then for each line segment in our planner path, we calculate the time each individual joint needs to reach its goal traveling with its maximum joint velocity. We then take the time of the slowest joint (i.e., the most time consuming joint) as the reference time for all joints. Thus, at any point in time, all joints are guaranteed to travel at most with their respective maximum joint velocity, and the overall execution time is limited by the maximum velocity of the slowest joint in each line segment. After calculating the execution time of each line segment we sample the path with a sample time of 20ms and obtain our final high resolution path ready for execution. A positive side-effect of this approach is that we do not pause or stop motion at intermediate points.

The resulting interpolated path is then send to the Meka server point by point via

[1]`http://www.zeroc.com/ice.html`

asynchronous Ice function calls followed by a sleep of 20ms, which corresponds to the path sample time. In other words, we steer our Meka robot by sending a new joint position every 20ms, ultimately executing our planned path.

For planning we used a static hand model, which meets the requirements for pushing only. However, for execution, we really have to transition between grasps and stretched pushing poses of the robot's manipulator. Concretely, when a new `pickup` primitive is being executed, the hand grasps and incrementally modifies its dynamic model by increasing the end-effector's payload, in order to account for the additional weight. Accordingly, as soon as the primitive `transfer-rigid` ends, we incrementally decrease the end-effector's payload again, open the hand and transition back to the stretched pushing pose.

To conclude, Figure 4.11 presents snapshots of a video recording the execution of a solution path for Scenario 2, following the methods we discussed in this section.

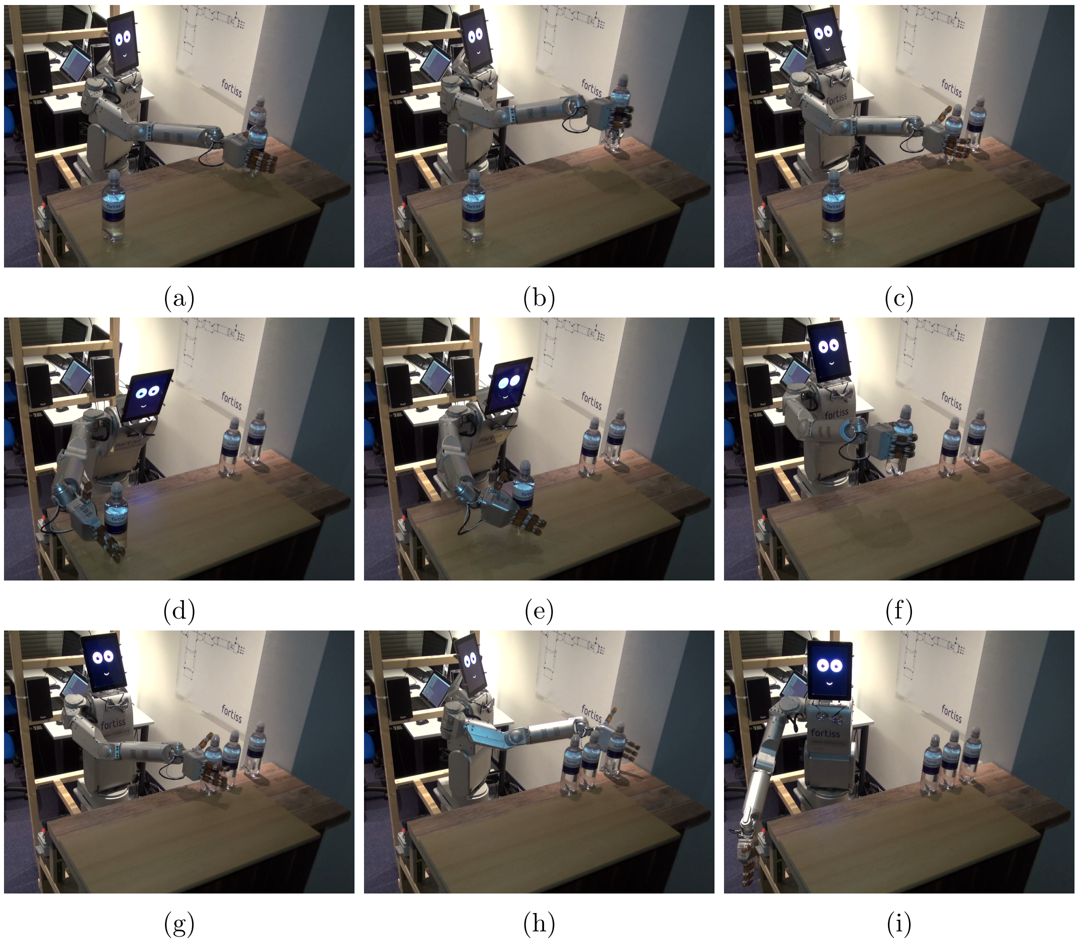


Figure 4.11: **A solution path for Scenario 2 executed in the real environment**. After picking up and transferring the middle bottle to its goal position, the robot reaches out to push the leftmost bottle towards him. He then picks it up and transfers it to its goal. Finally, the rightmost bottle moves to its goal position by being pushed with the exterior surface of the hand. In the end, the robot transits back to its starting configuration. The overall execution time is about 112 seconds. The full video is available on the website `youtu.be/uUKXgkq5Qsk`.

5 Conclusion

In this work, we first presented background knowledge for manipulation planning in the areas of robotics, sampling-based motion planning, and multi-modal planning. We then examined the Diverse Action Manipulation (DAMA) problem, and developed two flat (DARRT and DARRTConnect) and one hierarchical algorithm (DARRTH(Connect)) to solve for it. All three are based upon the sampling-based Rapidly-exploring Random Tree (RRT) algorithm, including some constraints and approaches, which were derived from the multi-modal nature of the DAMA problem. These three algorithms can plan for multiple, possibly non-prehensile actions manipulating multiple objects. Finally, we evaluated them on two challenging scenarios, each in a different world with either a mobile or a joint robot, showing the generic approach of our DAMA algorithms. Also, we presented the entire process from planning to execution on a real robot solving a DAMA problem in a real environment.

Compared to the work of Barry [6], we successfully solved DAMA problems with up to three objects, and showed that our implementation can handle various robots and environments without much adaptation, even for challenging scenarios.

5.1 Future Work

In this thesis, there is much room for future work.

First of all, Barry [6, pp. 162-166] presents many future avenues of research for DAMA problems. Among them are planning for uncertainty in manipulation primitives, solving for cluttered domains with goals for multiple objects, and implementing more dynamic manipulation actions like shoving or throwing. In addition to her list of issues, let us now examine some more important topics of interest.

The RRT algorithm, which lays the foundation of the DAMA algorithms in this work, can be extended in many ways, in order to solve problems more efficiently and effectively. For instance, RRT* provides us with an asymptotically optimal algorithm, which converges to optimal path lengths, along with probabilistic completeness and no substantial computational overhead [23, 24]. This algorithm is especially intended for planning in real-time. RRT* can then be further refined, for example to a memory efficient version, limiting the memory required for storing the tree [3]. We can also think of replanning with RRTs, when the configuration space changes [10]. Balancing exploration and exploitation is also a crucial point, as further examined by [42]. Another extension of RRTs are Resolution-Complete-RRTs (RC-RRTs), in which basically there is a limit in expanding from each vertex in the tree, when connecting from it keeps on failing. This can soften the problem of being stuck at local minima, in which two configurations are near, but still cannot connect. Also, RC-RRTs can also be motivated by the fact that for DAMA problems, the distance function is only a manually tuned heuristic function and therefore quite inaccurate. Exactly determining the nearest neighbor with respect to minimizing the swept volume of movements, which occurs when propagating from one to the other configuration, is far from being computationally efficient. Therefore, we can relax the nearest neighbor determination a bit.

Instead of sampling uniformly at random in a certain subspace, we can apply more intelligent sampling strategies. For example, we can apply sampling on the $\mathcal{C}_{free}$ boundary, Gaussian sampling, or bridge-test sampling. Furthermore, Khanmohammadi and Mahdizadeh [25] proposes density avoided sampling for RRTs. All of these are approaches to sample more useful points in our configuration space, in order to find a solution faster.

After finding a solution with a sampling-based planning algorithm like the RRT, the path is usually jagged and we should apply some path smoothing to clean it up, before evaluating or executing it on a robot. This can be done by iteratively picking a pair of path configurations at random and attempt to directly connect both of them. If the resulting connection is collision-free and feasible, we use this segment and discard the configurations, which originally were in between. Otherwise, we start over again, until the path does not improve anymore. Considering path smoothing, for our DAMA algorithms to actually find a path at all will become more important than finding a short path, because we can smooth jagged paths.

The algorithm implementations in this work for solving DAMA problems with multiple objects are not guaranteed to follow exponential convergence. Barry [6] proved exponential convergence for her algorithms, however, the strong assumptions concerning sampling or the usage of projection functions are certainly not met in this work. We can further investigate on this topic, in order to extend our work to a exponentially convergent version.

As evaluated in Chapter 4, our algorithms spend most of the time on computing the inverse kinematics or searching for the nearest neighbor of a certain configuration in our tree. Both algorithms can and should be improved completely separated from the DAMA algorithms presented in this work, in order for our algorithm to be more efficient in even more complex scenarios.

Also, there are some technical and implementation-dependent details to be solved. For example, when planning to grasp an object, the inverse kinematics always returns the single best grasp configuration, in which ”‘best”’ refers to posture optimization, i.e. staying away from joint limits. However, the inverse kinematics does not check, if a grasp configuration leads to robot bodies colliding. Therefore, even if collision-free grasps exist for the respective object, the inverse kinematics can always return a configuration which leads to a collision.

Instead of terminating as soon as a collision between the robot and an object occurs, we can also think of modeling deformable object, or objects which are able to move also unintended. Simulating physics, we can for instance model the effect of the robot colliding with an object, and consider the result in our sampling-based roadmap somehow. Recall the problem description of Scenario 1. With this approach, the robot could transit right through the left or the right object blocking the path, without him actively and intentionally applying a pushing manipulation.

Furthermore, we can add initial sensing of object placements, but also visual feedback while the planned path is executed, in order to compensate for errors in robot’s odometry or object locations.

Bibliography

[1] Meka robotics - dexterous mobile manipulators and humanoids. `http://mekabot.com/`. Accessed October 20, 2013.

[2] Robotics library (rl). `http://roblib.sourceforge.net/`. Accessed October 28, 2013.

[3] Olzhas Adiyatov and Huseyin Atakan Varol. Rapidly-exploring random tree based memory efficient motion planning. In *IEEE International Conference on Mechatronics and Automation (ICMA)*, pages 354–359, August 2013.

[4] Jennifer Barry, Kaijen Hsiao, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Manipulation with multiple action types. In *Experimental Robotics*, volume 88 of *Springer Tracts in Advanced Robotics*, pages 531–545. Springer International Publishing, June 2012.

[5] Jennifer Barry, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. A hierarchical approach to manipulation with diverse actions. In *IEEE Conference on Robotics and Automation (ICRA)*, 2013.

[6] Jennifer L. Barry. *Manipulation with Diverse Actions.* PhD thesis, Massachusetts Institute of Technology, June 2013.

[7] Dmitry Berenson, Siddhartha S. Srinivasa, Dave Ferguson, Alvaro Collet, and James J. Kuffner. Manipulation planning with workspace goal regions. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 618–624, 2009.

[8] Jur Berg, Mike Stilman, James Kuffner, Ming Lin, and Dinesh Manocha. Path planning among movable obstacles: A probabilistically complete approach. In GregoryS. Chirikjian, Howie Choset, Marco Morales, and Todd Murphey, editors, *Algorithmic Foundation of Robotics VIII*, volume 57 of *Springer Tracts in Advanced Robotics*, pages 599–614. Springer Berlin Heidelberg, 2009.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer Science + Business Media, LLC, 8 edition, 2006. `http://research.microsoft.com/~cmbishop/PRML`.

[10] Dave Ferguson, Nidhi Kalra, and Anthony Stentz. Replanning with rrts. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1243–1248, 2006.

[11] fortiss GmbH. James - joint action for multimodal embodied social systems. `http://www.fortiss.org/en/research/projects/james/`. Accessed October 20, 2013.

[12] Mary Ellen Foster, Andre Gaschler, Manuel Giuliani, Amy Isard, Maria Pateraki, and Ronald P. A. Petrick. Two people walk into a bar: Dynamic multi-party social interaction with a robot agent. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction (ICMI 2012)*, 2012.

[13] A. Gaschler, R. P. A. Petrick, O. Khatib, and A. Knoll. A knowledge of volumes approach to robot task planning. *Artificial Intelligence*, 2014. in revision.

[14] Andre Gaschler, Ronald P. A. Petrick, Manuel Giuliani, Markus Rickert, and Alois Knoll. KVP: A Knowledge of Volumes Approach to Robot Task Planning. In *IEEE/RSJ Intl Conf on Intelligent Robots and Systems (IROS)*, pages 202–208, November 2013.

[15] Andre Gaschler, Ronald P. A. Petrick, Torsten Kröger, Oussama Khatib, and Alois Knoll. Robot task and motion planning with sets of convex polyhedra. In *Robotics: Science and Systems (RSS) Workshop on Combined Robot Motion Planning and AI Planning for Practical Applications*, June 2013.

[16] Kris Hauser. *Motion Planning for Legged and Humanoid Robots*. PhD thesis, Stanford University, December 2008.

[17] Kris Hauser and Jean-Claude Latombe. Multi-modal motion planning in non-expansive spaces. *The International Journal of Robotics Research*, 29(7): 897–915, June 2010.

[18] Kris Hauser and Victor Ng-Thow-Hing. Randomized multi-modal motion planning for a humanoid robot manipulation task. *International Journal of Robotics Research*, 30(6):678–698, February 2011.

[19] David Hsu, Jean-Claude Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 2719–2726, April 1997.

[20] Du Q. Huynh. Metrics for 3d rotations: Comparison and analysis. *Journal of Mathematical Imaging and Vision*, 35(2):155–164, 2009.

[21] Reza N. Jazar. *Theory of Applied Robotics: Kinematics, Dynamics, and Control*. Springer Publishing Company, 2nd edition, 2010.

[22] Sertac Karaman and Emilio Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In *Robotics: Science and Systems (RSS)*, Zaragoza, Spain, June 2010.

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

[24] Sertac Karaman, Matthew R. Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. Anytime motion planning using the rrt*. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1478–1483, May 2011.

[25] Sohrab Khanmohammadi and Amin Mahdizadeh. Density avoided sampling: An intelligent sampling technique for rapidly-exploring random trees. In *Eighth International Conference on Hybrid Intelligent Systems (HIS)*, pages 672–677, 2008.

[26] Lukáš Knispel. Advanced robot path planning (rrt). Master's thesis, Brno University of Technology, 2012. URL `https://sites.google.com/site/rrtexplorer/home`.

[27] Lukas Krammer, Wolfgang Granzer, and Wolfgang Kastner. A new approach for robot motion planning using rrt. In *9th IEEE International Conference on Industrial Informatics (INDIN)*, pages 263–268, 2011.

[28] James J. Kuffner. Effective sampling and distance metrics for 3d rigid body path planning. In *Proceedings of the 2004 IEEE International Conference on Robotics & Automation*, volume 4, pages 3993–3998, 2004.

[29] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[30] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Tr 98-11, Department of Computer Science, Iowa State University, October 1998.

[31] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[32] Steven M. LaValle. Motion planning: The essentials. *IEEE Robotics and Automation Society Magazine*, 18(1):79–89, 2011.

[33] Steven M. LaValle. Motion planning: Wild frontiers. *IEEE Robotics and Automation Society Magazine*, 18(2):108–118, 2011.

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

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

[36] Steven M. LaValle and James J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In B. R. Donald, K. M. Lynch, and D. Rus, editors, *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2001.

[37] Stephen R. Lindemann and Steven M. LaValle. Current issues in sampling-based motion planning. In Paolo Dario and Raja Chatila, editors, *Robotics Research: The Eleventh International Symposium*, volume 15 of *Springer Tracts in Advanced Robotics*, pages 36–54. Springer-Verlag, Berlin, 2005.

[38] Khaled Mamou and Faouzi Ghorbel. A simple and efficient approach for 3d mesh approximate convex decomposition. In *16th IEEE International Conference on Image Processing (ICIP)*, pages 3501–3504, 2009.

[39] K. Miyazawa, Y. Maeda, and T. Arai. Planning of graspless manipulation based on rapidly-exploring random trees. In *The 6th IEEE International Symposium on Assembly and Task Planning (ISATP): From Nano to Macro Assembly and Manufacturing*, pages 7–12, 2005.

[40] Takahiro Otani and Makoto Koshino. Applying a path planner based on rrt to cooperative multirobot box-pushing. *Artificial Life and Robotics*, 13(2): 418–422, 2009.

[41] Markus Rickert. *Efficient Motion Planning for Intuitive Task Execution in Modular Manipulation Systems.* Dissertation, Technische Universität München, Munich, Germany, May 2011. URL `http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20110719-981979-1-6`.

[42] Markus Rickert, Oliver Brock, and Alois Knoll. Balancing exploration and exploitation in motion planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2812–2817, 2008.

[43] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 3 edition, 2010.

[44] JAMES project. Joint action for multimodal embodied social systems. `http://www.james-project.eu`. Accessed October 20, 2013.

[45] Anna Yershova and Steven M. LaValle. Improving motion planning algorithms by efficient nearest-neighbor searching. *IEEE Transactions on Robotics*, 23(1):151–157, February 2007.

[46] Anna Yershova and Steven M. LaValle. Motion planning for highly constrained spaces. Technical Report UIUCDCS-R-2008-2975, Department of Computer Science, University of Illinois, 2008.

[47] Anna Yershova, Léonard Jaillet, Thierry Siméon, and Steven M. LaValle. Dynamic-domain rrts: Efficient exploration by controlling the sampling domain. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 3867–3872, 2005.