A Hierarchical Framework for Configuration Space Task Planning

Evan Shapiro

CMU-CS-15-133

July 2015

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Siddhartha Srinivasa
Maxim Likhachev

Submitted in partial fulfillment of the requirements for the degree of Masters of Science.

This work was sponsored in part by a grant from the Toyota Motor Corporation

Keywords: Robotics, Task Planning, Task Specification, Manipulation Planning, Motion Planning, Hierarchical Planning, Configuration Space, Monte Carlo Tree Search, Interleaved Planning and Execution

I dedicate this work to my family and friends. Their support through the many long days and nights of work for this research ensured the most difficult problems and pressing deadlines never seemed insurmountable, and I couldn't imagine accomplishing this thesis without them.

Abstract

Robots are rapidly moving towards proficiency at useful tasks. In order for task planners to adapt to novel scenarios, new architectures and algorithms will be necessary. This work develops a framework and extensions to that framework in order to provide a system that can efficiently and robustly plan for a wide variety of tasks.

The framework dynamically restricts search space by exploiting highly interconnected hierarchically composed actions. The hierarchical action structure can be generated dynamically during planning, allowing geometric state to determine high level search space restrictions. This allows the framework to provide intermediate goals to efficiently plan through high dimensional spaces over long sequences of actions, and integrate with arbitrary existing task and motion planners.

We also develop framework enhancements to improve performance during both planning and during execution. The order in which intermediate steps are explored greatly affects planning time. We apply a variant of Monte Carlo Tree Search to determine the order to compute intermediate steps. Execution performance is improved by interleaving planning and execution. We consider tasks that are composed of reversible trajectories, and construct methods to traverse partially complete plans. We explore the affects of different variants of these enhancements on manipulation tasks.

Acknowledgments

I'd like to thank my adviser Siddhartha Srinivasa, my professors and the entire Personal Robotics Lab. I learned an incredible amount interacting in such a fantastic environment and I'm truly thankful for the opportunity! In particular I'd like to thank professor David Kosbie for taking the time to mentor me throughout my time at Carnegie Mellon, and Prassana Velagapudi who guided me into research at the Personal Robotics Lab.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Challenges in Task Planning

Task solutions exist in configuration space. In this sense, task planning is an extension of motion planning, made more difficult due to two important qualities of tasks.

The first is that tasks solutions can cover the full dimensionality of configuration space. Isolated motion planning problems that are only over a small portion of configuration space (say the robot's arm) are relatively tractable. Planning over the full space, including environment state, vastly increases the number of potential paths that can be searched over, and therefore vastly increases problem difficulty.

The second problem is that solution paths for tasks are usually far longer than solution paths for isolated motion planning problems. As path length increases, the number of potential paths increases exponentially.

These problems can be better understood by looking at an example (Fig. 1.1). The number of valid states in the world is tremendous. The goal in this problem is to place the glass on the kitchen counter (with some flexibility in the z-axis).

There are certain key states in the world that must be passed through to find a solution. The two obvious ones for this problem are that the robot must grasp the glass, and the glass must be placed on the counter. Because we have a characterization of the goal states, it is reasonable for a planner to find glass positions on the counter.

However, an uninformed planner is not aware that it is necessary to first grasp the glass. The space of configurations where the robot is grasping the glass is miniscule compared to the number of valid states. Therefore, it is highly unlikely that a fully uninformed search will find a solution to this pick and place task.

This means the planner must be somehow informed, whether by human intuition or a learning algorithm, about these key intermediate states. Once the planner is given suggestions of potential intermediate states, task length becomes a less lethal problem.

Introducing intermediate state suggestions also implies a solution for the high dimensionality problem. If all pairs of intermediate states can be planned between through smaller subsets of configuration space, then the full space never needs to be searched at once.

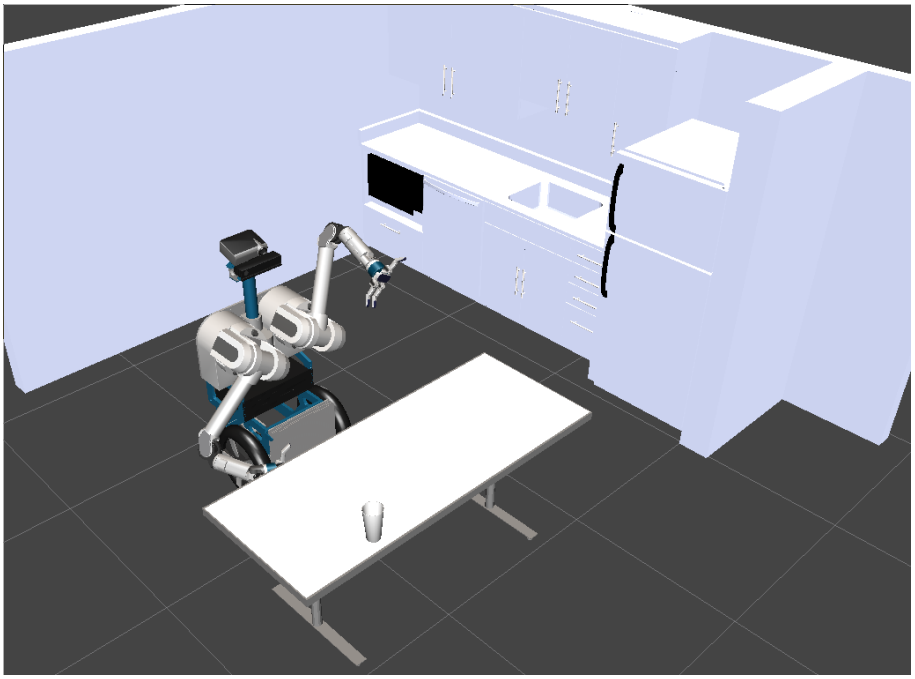

Figure 1.1: An example of a long horizon manipulation task. The robot must move the glass from the table to the kitchen counter.

For the problem of moving items to a bin, this corresponds a sequence of intermediate states (Fig. 1.2) that specify moving the arm near the bowl, then an intermediate state where the arm is grasping the bowl, then an intermediate state where the arm is positioning the bowl in the bin, and an intermediate state where the bowl is in the bin and the arm is removed from the bin. Each of these requires planning over only the base pose or only the robot's arm.

1.1.2 Searching Between Intermediate States

A potentially intuitively appealing implementation of this is to break tasks into sequential motion planning problems that are solved consecutively. In Fig.1.1, first base poses would be searched over, then arm configurations, and so on. However, this naive approach is doomed to failure due to complex dependencies between different parts of a plan.

In the pick and place task, this can be as simple as choosing a base pose where the arm cannot grasp the glass. In more complex tasks, dependencies can be farther separated. In a task with multiple objects, the way an object is manipulated early on in a plan may make later interaction with that object difficult, or potentially impossible. Due to the complexities of robot kinematics and environment configurations, it is far too common for planners operating this way to reach dead ends.

Assuming reversibility during execution or applying backtracking during planning is one way to continue after a dead end is reached. This method of ensuring continued feasibility is far from optimal. Reversibility, when it is even possible, requires the cost of execution time, both to complete the original failed trajectory and reverse it. Backtracking is also often far from the

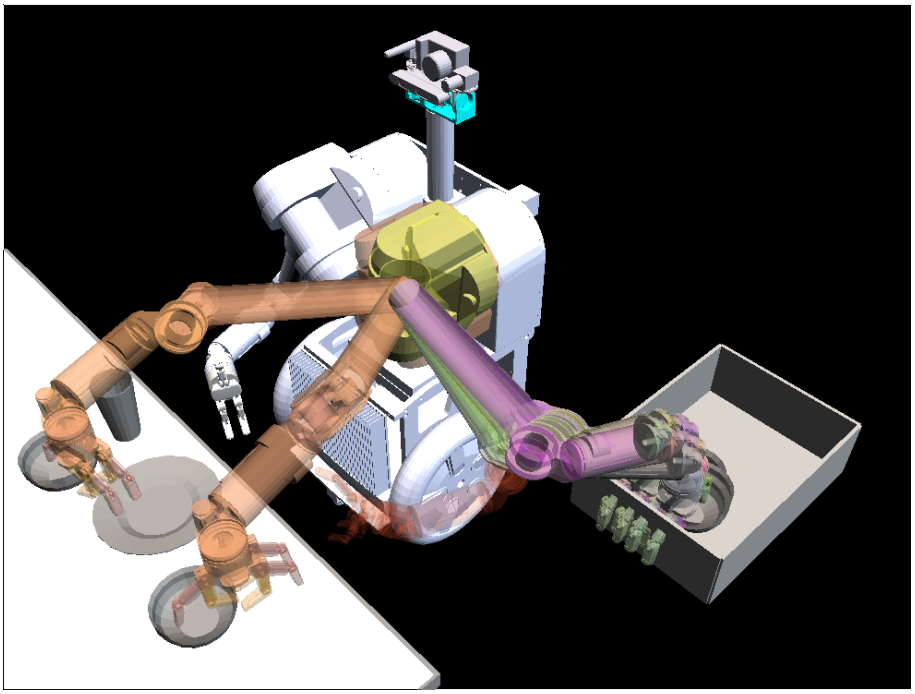

Figure 1.2: A visualization of some intermediate states for a path through configuration space that moves all items from a table to a bin.

optimal strategy, as planning should switch to the step which caused the failure, which is often not the previous step.
For this reason we desire a planner which has the flexibility of considering arbitrary orders of planning between intermediate states.

1.1.3 Dynamically Sampling Intermediate States

There is one more property to consider when planning with intermediate states. In the most simple case, intermediate states can be statically generated. However, in many real world scenarios, static generation is not possible. For example, going back to Fig.1.1, if the planner tries to generate goal states, it must commit to a relative orientation between the robot's hand and the glass. There are many possible ways the robot can grasp the glass. The way the glass is grasped is determined by an earlier intermediate state, the one in which the glass was grasped in the first place. There are many examples of this nature. An item that was placed in an earlier step of the plan must be grasped where it was placed. The robot can only drive to positions in free space, which may change depending on how it manipulates the environment. The state of the environment also includes all objects, even those that are not relevant to the particular subproblem being solved. The state of those too must be included in intermediate states.
This means that intermediate states usually need to be dynamically generated from their preceding intermediate states. This imposes an order on intermediate state generation, that later states can only be generated after earlier ones.

1.1.4 Hierarchical Task Planning

The task planning problem also has structure beyond a sequence of trajectory planning problems. There are many cases that do not fit cleanly into a fixed sequence planning framework. Looking at two example patterns will help gain intuition for non-fixed sequence planning.

Oftentimes there exists multiple alternative methods to solving a problem. The environment may contain duplicate items, or there may be both a short but difficult solution path and an easy but long solution path. In these cases, it is insufficient to assign the same motion planner to all subproblems, as each subproblem can be broken down into a different series of intermediate states. Some problems also need to be solved in an unknown number of steps. A task for example may require rearrangement of an environment, and it is not necessarily known ahead of time which items need to be arranged.

Both of these cases can be solved by introducing hierarchy. Hierarchy allows multiple different subtasks to be attempted to get between pairs of intermediate states. This geometrically corresponds to recursively breaking intermediate state sequences into sets of finer grained state sequences, until the lowest level problems can be solved by a motion planner.

The increased structure provided by hierarchy also improves structure reusability, and the structure itself can be exploited to improve planning efficiency.

In chapter 2 dynamically sampling intermediate states and hierarchical task planning is formalized for the setting of configuration space planning.

1.2 Related Work

1.2.1 Related Robotics Planning Work

Various systems have been created that search over restricted subsets of configuration space. Our work provides a general framework for integrating these different methods and specifying restricted search spaces.

Many symbolic planning systems assume partial geometric independence between actions and focus on symbolic manipulation [3, 5, 12, 15, 16]. While this allows highly efficient abstract planning, it requires geometric actions to be reversible. Our work differs in that it aims to provide a structure for end to end geometric planning, and therefore supports geometric constraints without reversible actions.

Some systems consider producing lengthy plans without breaking them into segments [1, 25]. These planners instead construct complete geometric plans from start configurations directly to goal configurations. However, producing plans in this manner is computationally expensive. Our work allows higher level structure to restrict the search space, allowing for far more efficient planning.

There are also unified systems which integrate a combination of symbolic and geometric techniques [2, 7, 9, 10, 11, 14, 18, 20, 21, 23, 24, 26, 28]. The unified systems [7, 9, 10, 18, 20, 21, 26, 28] in particular share our approach's goal of unifying planners of different levels of abstraction to restict search space.

Our work distinguishes itself by its focus on a general solution to dynamically restricting search space. While these existing planners deal with specific subsets of potential abstract and geometric planning, our work provides a general solution allows for the flexible composition of multiple planners into the same task.
Under our framework, existing methods can be hierarchically composed to restrict planning space to the desired degree. Motion planners such as RRT[19] and DARRT[1] can be used as subactions, as well as symbolic planners such as PDDL [22].

1.2.2 Related Monte Carlo Tree Search Work

We employ a variant of Monte Carlo Tree Search [6] to improve search efficiency. Monte Carlo Tree Search has been employed in many search problems, with notable success in computer Go [13]. Monte Carlo Tree Search performs well for go, because similarly to the robotics planning problem, the game tree is far too large to fully explore. A probabilistic method is needed to simultaneously limit the number of evaluated states while ensuring a high likelihood that a good state is found.
Many of the Computer Go Monte Carlo Tree Search algorithms use functions written by human experts to estimate the utility of a given board state. Handwriting a general purpose configuration space evaluation function is not feasible for the general task planning problem. Our approach instead uses the current tree state to determine distributions over the utility of sampling a particular state, similar to the Bayesian Monte Carlo Tree Search variant proposed by Tesauro et al. [27].

1.3 Contributions

We contribute a hierarchical, general purpose framework for configuration space task planning. Specifically:

1. A hierarchically composable interface for manipulation actions.
2. A library of actions for manipulation task planning.
3. An application of Monte Carlo Tree Search to sampling and computing trajectory trees.
4. A policy for interleaving planning and execution of trajectory trees.

Part I

The Core Framework

Chapter 2

A Hierarchical Formulation of Configuration Space Planning

2.1 A Hierarchical Formulation of Configuration Space Planning

We consider the problem of manipulation planning in a configuration space $\mathcal{C}$ with obstacles $\mathcal{C}_{obs} \subseteq \mathcal{C}$ and free space $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$. The space has a start set $\mathcal{C}_s \subseteq \mathcal{C}_{free}$ and a goal set $\mathcal{C}_g \subseteq \mathcal{C}_{free}$. The goal is to find a path $\tau : [0, 1] \to \mathcal{C}_{free}$ such that $\tau(0) \in \mathcal{C}_s$ and $\tau(1) \in \mathcal{C}_g$. We make no other assumptions about the configuration space. $\mathcal{C}$ may consist of a combination of the joint angles of the robot, the poses of the objects in the environment, and even task-specific discrete state.

The structure our framework exploits in the manipulation planning problem makes two assumptions. These assumptions allow the planner to find τ by breaking the original problem into more tractable subproblems.

The first assumption is that the original problem of finding τ can be broken into the problem of finding the sequence $\tau_1 \dots \tau_n$, where $\tau_i(1) = \tau_{i+1}(0)$, and $\tau_i(1)$ is an intermediate goal.

The second assumption is that each τ_i is a path over a subspace of $\mathcal{C}$ that is significantly lower dimensional.

τ can now be found by traversing a graph structure. Each node in the graph is an intermediate state in configuration space. Each edge corresponds to a τ_i. A path from a node in the start set to a node in the goal set consists of a sequence of τ_i, and concatenated together is τ (Fig.2.1).

The restated problem is far more tractable than the original one. The new difficulty is generating intermediate states that a motion planner can find a path between. For many problems either human intuition, machine learning, or a discrete planner can provide intermediate states that can be planned between, thereby making the problem tractable.

The strength of this perspective on task planning is that not only is it tractable but it is flexible. The set of assumptions that have been made produce a setup that retains flexibility. The process generating intermediate states does not need to always provide feasible graphs, it only needs to have some probability of producing feasible graphs.

As the graph becomes larger, the large number of intermediate states and potential trajectories

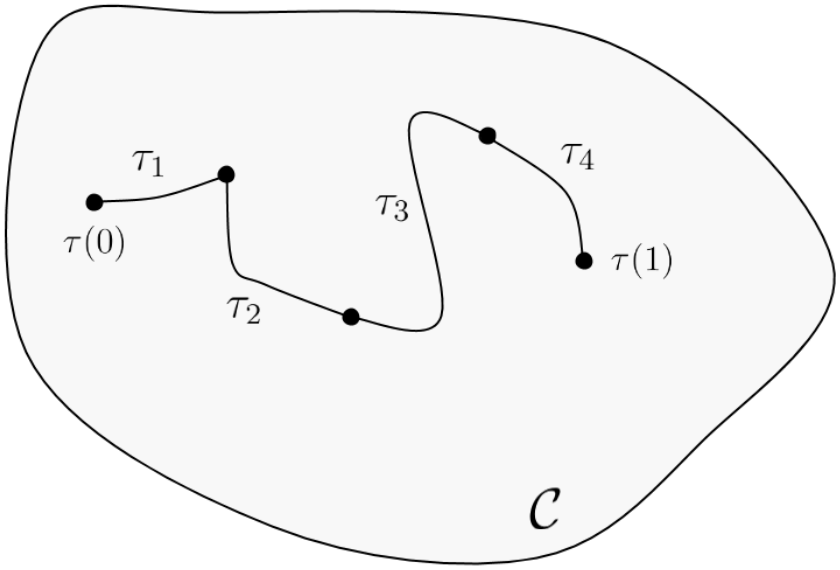

Figure 2.1: Traversing configuration space with a sequence of trajectories.

becomes a bottleneck for finding solutions. To improve scalability, we will treat finding each trajectory τ_i as its own hierarchical task planning problem, with its own graph and its own subtrajectories (Fig.2.2). This restricts the size of the graph at each level.

2.2 Dynamically Constructing and Traversing Hierarchical Graphs in Configuration Space

The extra structure resulting from a hierarchical treatment of the graph search problem can also be usefully exploited by the processes that generate intermediate states.
Constructing and traversing the graph occurs as follows (Alg. 1). At each step, a pair of states is selected. A path between the pair of states is then either solved for directly with a motion planner, or it is recursively evaluated as another hierarchical path finding problem.
If all pairs of states are evaluated directly with a motion planner, the problem is in the non-hierarchical form. Formally this corresponds to a single generator $\mathcal{G}$ that takes as input a set of start configurations and a set of goal configurations and outputs intermediate states that can be solved with a motion planner.
The hierarchical instance of the problem corresponds to a generator $\mathcal{G}$ that takes as input a set of start configurations and a set of goal configurations and outputs intermediate states, each of which can either be solved with a motion planner or some other generator $\mathcal{G}'$.
A key property of the hierarchical problem is that because $\mathcal{G}'$ is generated by $\mathcal{G}$ as a function of states of configuration space, the generator can be chosen dynamically depending on the available states. This makes it possible for geometric configuration space information to influence high level guidance. Communication between geometric and abstracted reasoning is

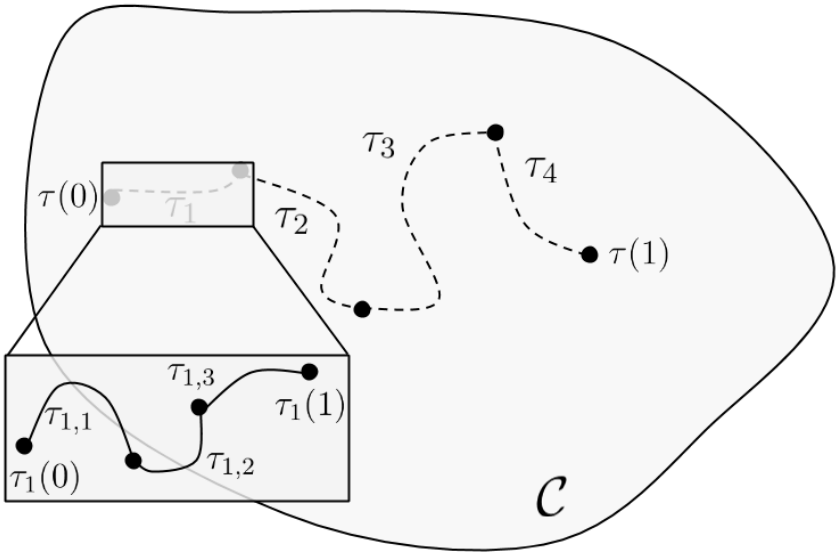

Figure 2.2: Traversing configuration space with a hierarchical sequence of trajectories.

essential for tractable manipulation planning, and it is by this mechanism that that is made possible.

We will refer to a single level that implements SEARCHCONFIGURATIONSPACE as an action. There are four components that are so far unconstrained in this framework.

1. Generating intermediate states and action scope

Action scope refers to what types of intermediate states can be generated for each action. There is a spectrum of possible action scopes, from a single action that only produces primitive subpaths, to a maximally hierarchical design where each action only generates a single set of intermediate goals.

Generating intermediate states and action scope are intertwined because how actions are separated determines where intermediate goals are generated. We will now overview a fully general action that is able to capture all of the more specific action varieties.

An action is parameterized by the following, in addition to the other unconstrained components:

An action generates new intermediate states through the following two methods:

- $Q =$ SELECTGROWGRAPHSOURCES(G)
- GENERATEINTERMEDIATESTATEFROMSOURCES(G, Q)

The action will operate over its subgraph G. Each step it can generate a new state from existing states Q. Edges are created between the new state and the existing states, and are marked as potential trajectories. This process allows new states to be generated as a function of existing states. For example, a state with the robot in a different location can retain the full state of all other aspects of configuration space.

An effective planner will generate a wide variety of intermediate states that each have a high

likelihood of success. Diversity is important as it is often difficult to know what intermediate state is best to solve a task. Diversity increases the likelihood that some intermediate state will be useful in solving the task.

Intermediate states are sampled over a subset of configuration space, for example configurations of the arm such that its hand is near an object that is meant to be grasped. Our task implementations utilize human generated functions that, given a relatively low dimensional set of parameters, produce intermediate states.

Generating good intermediate states is then a problem of finding good parameters. If the function from parameters to states has been well designed, it is often sufficient to randomly sample parameters, under the assumption that the function will take care of turning those parameterizations into realistic states.

A more complex method, left for future work, is to learn functions, which learn mappings from the current task, action, and environment state to parameters that are likely to produce useful states.

2. Choosing which start and goal sets to explore

Choosing where to explore on the graph has huge performance implications. At each action, there exists a large number of edges corresponding to potential trajectories. Many of these edges are a waste of time to explore, as the planner will not be able to find a path between the intermediate states the edge connects. For other edges, a path can be found, but the edge is not useful towards the final solution path.

However, if the intermediate states are of sufficiently high quality, many edges will be useful. Under this assumption, a randomized strategy is good enough.

Adding a slight amount of complexity, we can add the assumption that if the planner has selected an edge multiple times and it keeps failing, it should be selected less. Under this assumption, the planner should select edges with probability inversely proportional to the number of times they have been selected. This promotes selection of underexplored edges.

Another potential strategy is backtracking. Edges are traversed forward until a failure occurs. Upon failure, the planner backtracks to a preceding state and a new intermediate state is generated and attempted. The number of edges that are backtracked increases until the planner makes it past where it is stuck.

If we constrain all graphs to be trees, Monte Carlo Tree Search can be applied to implement a policy for edge selection. This approach will be discussed in Chapter 5. The extra tree constraint is also present by default for many manipulation problems, as a new state generated from an existing one is not always compatible with other existing states. Monte Carlo Tree Search uses a probabilistic analysis of edge success to balance exploration and exploitation.

3. Action implementations

There are many options for actions taking the general form defined above. Actions can assign edges deterministically to some preselected discrete set of subactions. Alternatively, actions can assign edges to a subaction selected from a distribution over actions. This is useful when the correct high level logic to apply is not known. If the high level logic is known, and can be

discrete, edge subactions can be assigned symbolic to a logical planner. Specific actions useful for manipulation planning are defined in Chapter 3.

4. Primitive actions

At the final level of the framework are actions which directly find trajectories. These primitive actions can be anything that can plan between two states of configuration space, such as PRMs[17] or RRTs[19]. These return actual trajectories for sets of joints such as arms, hands, or the robot's base.

Primitive actions also include virtual actions, such as marking an item as grasped, changing the discrete state of a mechanism, or simulating other environment behaviors that are not directly planned over.

Algorithm 1 Finding a path in hierarchical configuration space

procedure FINDCONFIGURATIONSPACEPATH(Q_S : start set, Q_G : goal set, G : trajectory graph)
    while NOPATHBETWEENSETS(Q_S, Q_G, G) **do**
        SEARCHCONFIGURATIONSPACE(Q_S, Q_G, G)
    end while
end procedure

procedure SEARCHCONFIGURATIONSPACE(Q_S, Q_G, G)
    GENERATEINTERMEDIATESTATES(Q_S, Q_G, G)
    $Q'_S, Q'_G \leftarrow$ CHOOSESTARTGOALSETS(Q_S, Q_G, G)
    if PROBLEMISPRIMITIVE(Q'_S, Q'_G) **then**
        $\{\tau\} \leftarrow$ RUNPRIMITIVEPLANNER(Q'_S, Q'_G)
        ADDTRAJECTORIESTOGRAPH($\{\tau\}, G$)
    else
        $G' \leftarrow$ GETSUBGRAPH(Q'_S, Q'_G, G)
        SEARCHCONFIGURATIONSPACE(Q'_S, Q'_G, G')
    end if
end procedure

Chapter 3

Building Blocks for Manipulation Planning

We will now define action types useful for manipulation task planning.

3.1 General Purpose Actions

These actions serve as useful glue for other action types. All actions implement variations of the same interface (Fig.3.1). They are provided a start set $\mathcal{C}_{\text{s}} \subseteq \mathcal{C}_{\text{free}}$ and goal set $\mathcal{C}_{\text{g}} \subseteq \mathcal{C}_{\text{free}}$, and they find trajetories $\tau[0, 1]$ between sampled elements of their start and goal set.

3.1.1 Sequence Action

The *sequence meta action* (Fig.3.2) specifies a collection of subactions that must be performed in a specific order. A sequence meta action consists of an ordered sequence of n actions $A_1, \ldots, A_n$ where each action A_i has a start set $\mathcal{C}_{\text{s}}^i \subseteq \mathcal{C}_{\text{free}}$ and a goal set $\mathcal{C}_{\text{g}}^i \subseteq \mathcal{C}_{\text{free}}$. The first and last actions define the start $\mathcal{C}_{\text{s}} = \mathcal{C}_{\text{s}}^1$ and goal $\mathcal{C}_{\text{g}} = \mathcal{C}_{\text{g}}^n$ of the sequence meta action. Each pair of adjacent actions A_i and A_{i+1} must meet the requirement that $\mathcal{C}_{\text{g}}^i \subseteq \mathcal{C}_{\text{s}}^{i+1}$. These joined configuration sets are denoted $\mathcal{C}_i$, where $\mathcal{C}_i = \mathcal{C}_{\text{s}}^i$, and $\mathcal{C}_{\text{g}}^i \subseteq \mathcal{C}_{i+1}$.
The output $\tau = [\tau_1, \ldots, \tau_n]$ of the meta action is the concatenation of the paths $\tau_1, \ldots, \tau_n$ produced by the subactions. Note that the paths produced by the actions $A_1, \ldots, A_n$ are not independent because the path must satisfy $\tau_i(1) = \tau_{i+1}(0)$.

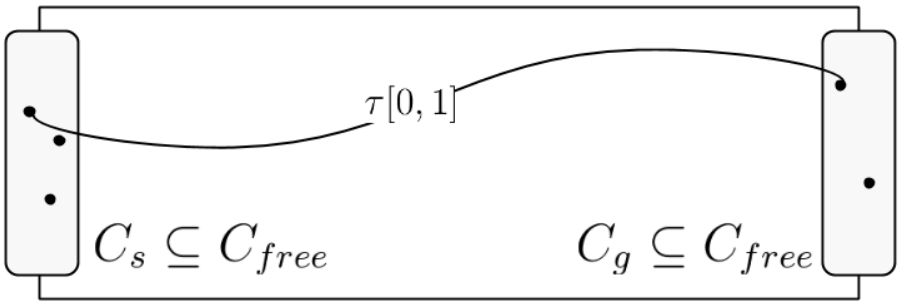

Figure 3.1: Generic Action

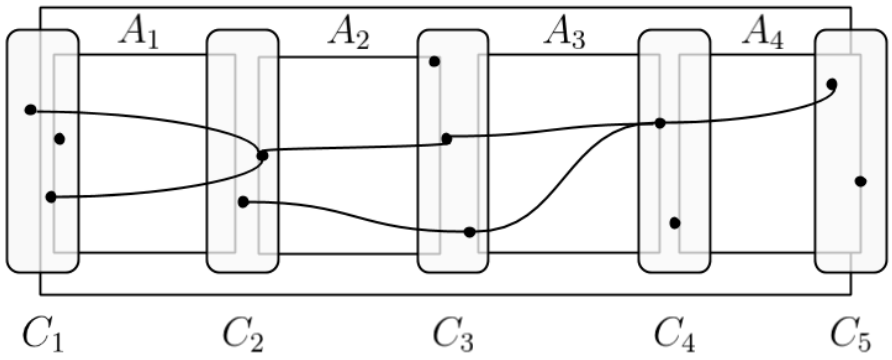

Figure 3.2: Sequence Action

This allows us to concatenate actions together. For example, if we have an action that grasps a glass, an action that moves our robot's base, and an action that places a glass, we can combine all three in sequence to perform pick and place tasks.

3.1.2 Parallel Action

The *parallel meta action* (Fig.3.3) specifies a collection of alternate subactions, only one of which must be performed. A parallel meta action consists of an unordered set of actions $A_1, \ldots, A_n$ where each action A_i covers part of the start $\mathcal{C}_{\text{s}}^i \subseteq \mathcal{C}_{\text{s}}$ and goal $\mathcal{C}_{\text{g}}^i \subseteq \mathcal{C}_{\text{g}}$ sets. The meta action's start and goal sets are defined to span its subactions, $\mathcal{C}_{\text{s}} = \cup_{i=1}^n \mathcal{C}_{\text{s}}^i, \mathcal{C}_{\text{g}} = \cup_{i=1}^n \mathcal{C}_{\text{g}}^i$. This allows us to consider alternative ways of completing tasks. For example, if we need a glass, but there are multiple glasses available, we can place actions for each possible glass in a parallel action. The parallel action produces trajectories that each result in grasping one of the glasses.

3.1.3 Repetition Action

The *repetition meta action* (Fig.3.4) allows a action to be recursively invoked until a goal is satisified. Consider an action A with start set $\mathcal{C}_{\text{s}}^A \subseteq \mathcal{C}_{\text{free}}$ and goal set $\mathcal{C}_{\text{g}}^A \subseteq \mathcal{C}_{\text{free}}$. We want to achieve a goal $\mathcal{C}_{\text{g}} \subseteq \mathcal{C}_{\text{g}}^A$ which is satisfied by some of the configurations produced by A. Any other configurations $q_g \in \mathcal{C}_{\text{g}}^A \setminus \mathcal{C}_{\text{g}} \subseteq \mathcal{C}_{\text{s}}$ are recursively added to $\mathcal{C}_{\text{s}}^A$ for future processing. The output τ is any trajectory produced by A, which may have been produced by invoking A multiple times, such that $\tau(1) \in \mathcal{C}_{\text{g}}$ and some intermediate configurations in τ are $q_g \in \mathcal{C}_{\text{g}}^n \setminus \mathcal{C}_{\text{g}}$. This allows us to repeat an action until another action is possible, or a subtask is complete. For example, this could be used if we need to remove some number of items from a fridge until a target object at the back of the fridge is reachable.

3.1.4 Checkpoint Action

The *checkpoint action* improves planning efficiency. Once its child action finds a path to any configuration $q_g \in \mathcal{C}_{\text{g}}^A$, it prevents future planning. Oftentimes a human expert knows that two

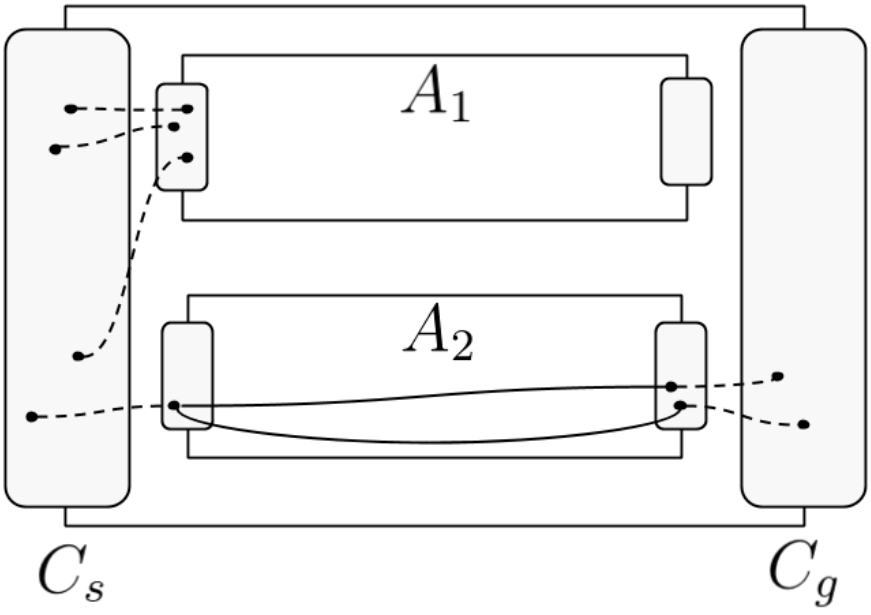

Figure 3.3: Parallel Action

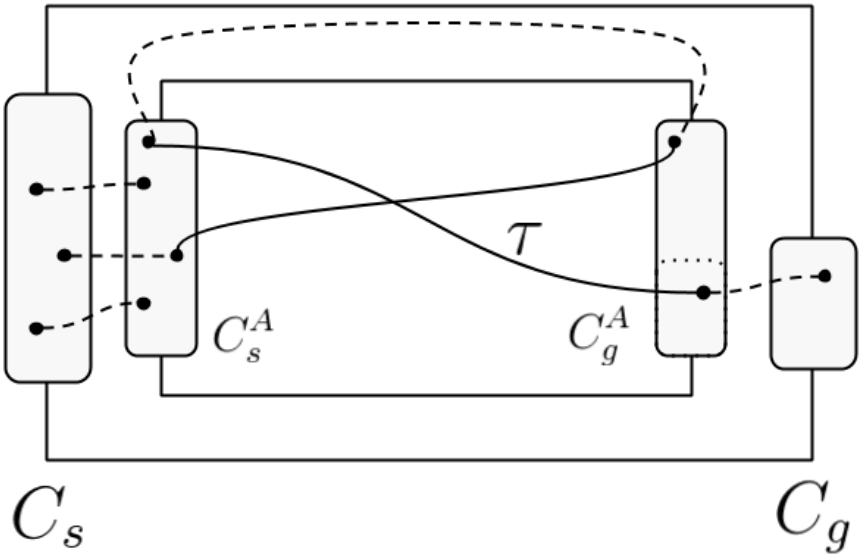

Figure 3.4: Repetition Action

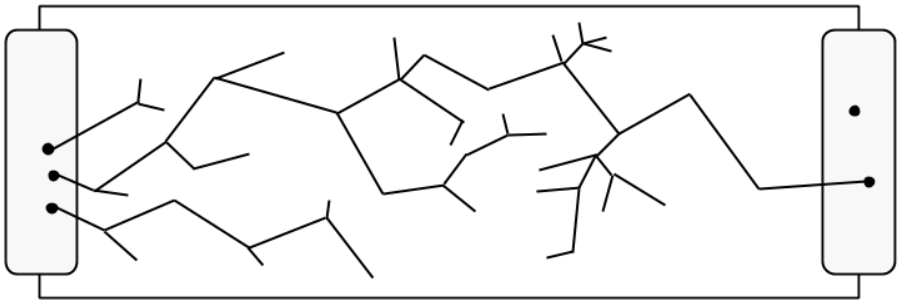

Figure 3.5: Configuration Planning Action Employing a Sampling Tree Planner such as BiRRT

stages of a plan do not have any dependencies. The checkpoint action formally implements this reasoning by locking in a portion of plan once complete.

3.2 Manipulation Planning Actions

3.2.1 Configuration Planning Action

The *configuration planning action* (Fig.3.5) is a primitive action which plans directly from a start set Q_s to a goal set Q_g. The action itself is motion planner independent as any standard motion planner such as CBiRRT, BiRRT, PRM, or SBPL can be used. This is what finds trajectories for parts of the robot such as its arms and base.

3.2.2 Pose Action

The *pose action* (Fig.3.6) is a meta action which produces trajectories from start configurations to configurations where the robot's hand is in a particular pose. The action samples configurations corresponding to intermediate goals, and then delegates planning on edges to the *configuration planning action*.

3.2.3 Sampled Pose Set Action

The *sampled pose set action* (Fig.3.7) is a meta action which produces trajectories from start configurations to configurations where the robot's hand is in one of a set of a set of sampled poses. The sampled poses can be generated from a human specified function, allowing specific instantiations of the action to plan to poses that fulfill certain behaviors, such as grasping a glass or a handle.
The action samples poses, and for each pose generates a *pose action* which then is stored as an option for computing the edge.

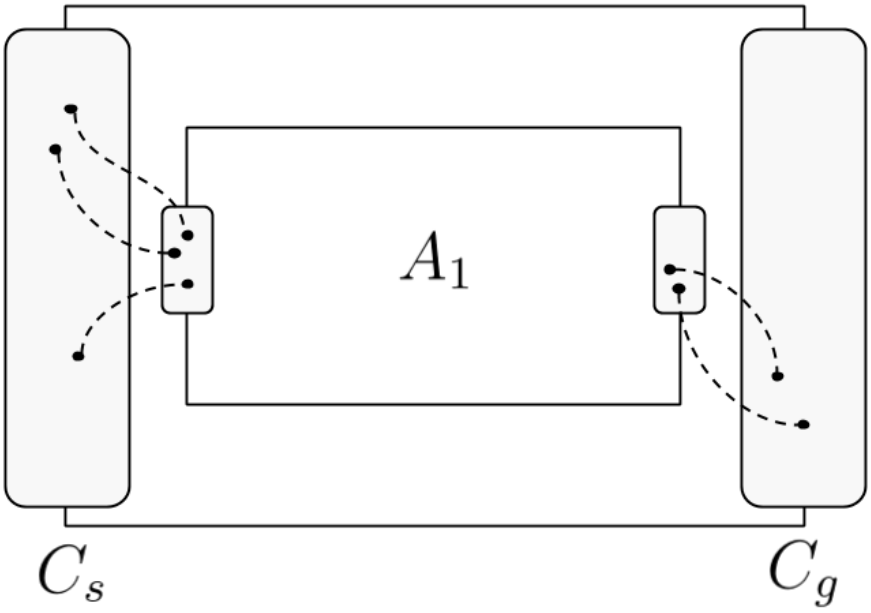

Figure 3.6: Pose Planning Action. A_1 is a configuration planning action

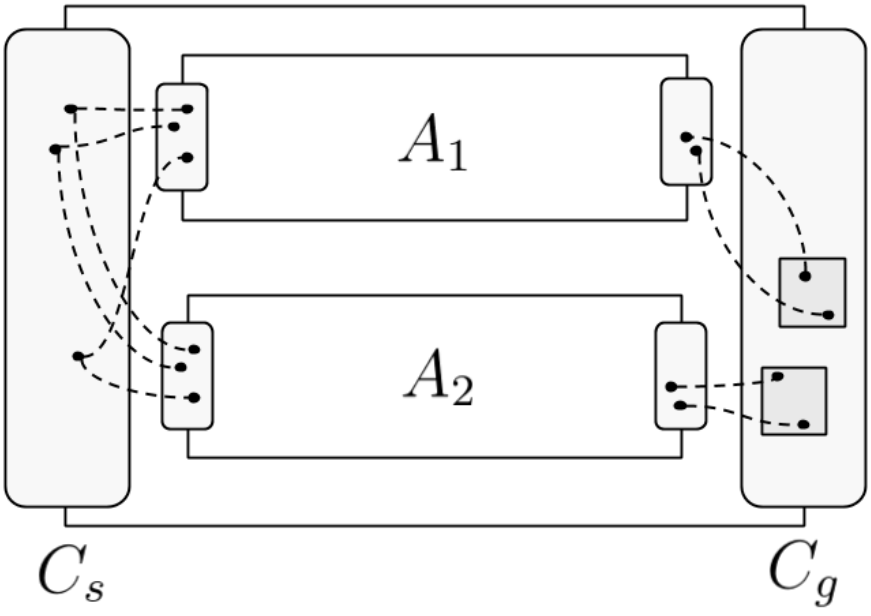

Figure 3.7: Sampled Pose Planning Action. Both A_1 and A_2 correspond to pose actions, and each box contains configurations sampled from the same hand pose.

3.2.4 TSR Action

The *TSR action* is a meta action which produces trajectories from start configurations to configurations sampled from the Task Space Region specification [4]. TSRs useful as they are an expressive and succinct method for representing constraints.
The action samples configurations, and each edge computation is delegated to a *configuration planning action*. Each configuration planning action is also provided the TSR so it can plan with the proper constraints and use a planner that supports TSR constraints.

3.2.5 Hand Movement Action

The *hand movement action* is a primitive action which produces trajectories to move the robot's hand to specific configurations.

3.3 Automated Task Planning Actions

3.3.1 PDDL Action

The PDDL action is a type of parallel meta-action, where its subactions are sequence actions. The sequences are dynamically generated using symbolic planning from the PDDL action's start configurations.
For each configuration $q_s \in C_s$, the state of the action's predicates is evaluated, and a symbolic planner constructs a sequence of subactions. Planning is then delegated to this sequence, which specifies a restricted search space.
Using PDDL as an action means that a single restricted geometric space can be obtained from concatenating the restricted space produced by PDDL and other planners. For example, the first step of a plan could run a motion planner with multiple goals. If the next step is a PDDL planner, the state of the PDDL predicates for different start configurations may cause different action sequences to be produced.
The PDDL subactions can of course be complex, dynamic actions such as the sampled pose set manipulation planning action defined above.

3.3.2 Automata Action

The automata action allows an automata to be used to restrict planning, which often makes it easier to specify complex plans. Each node in the automata corresponds to a subaction. Nodes in the automata take in configurations and pass them to their corresponding subactions. Configurations produced by these subactions are then passed to other nodes in the automata or the automata's C_g, according to the automata's transition rules.
The automata action is implemented through integrating the repetition and parallel actions. The automata action itself is a repetition action containing a parallel action. The subactions of the parallel action each correspond to nodes. The automata action transitions configurations produced by subactions to other subactions or to C_g.

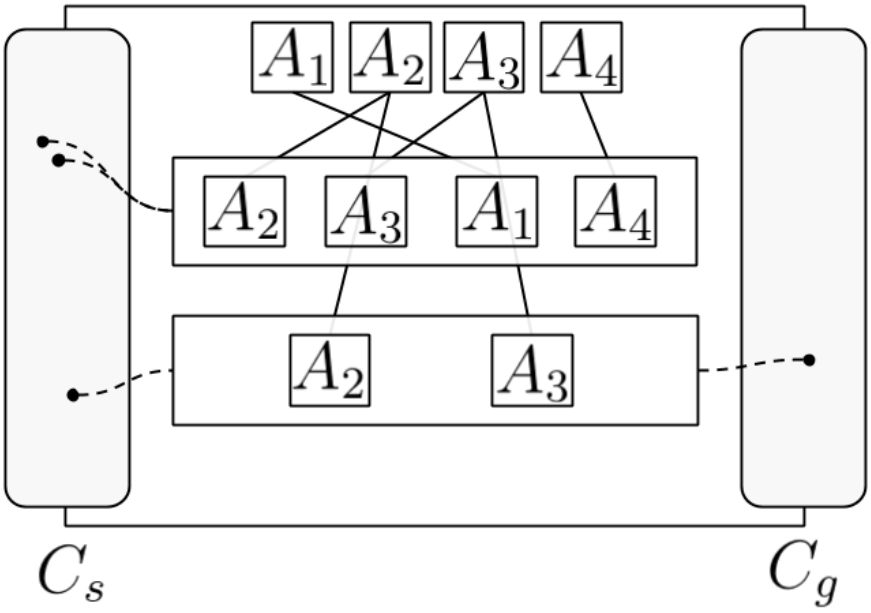

Figure 3.8: PDDL Task Planning Action. Different configurations result in different action sequences.

In doing so this action provides arbitrary user-specified dynamic intermediate guidance, as well as a foundation for integration with symbolic planners that produce automata such as Linear Temporal Logic [8].

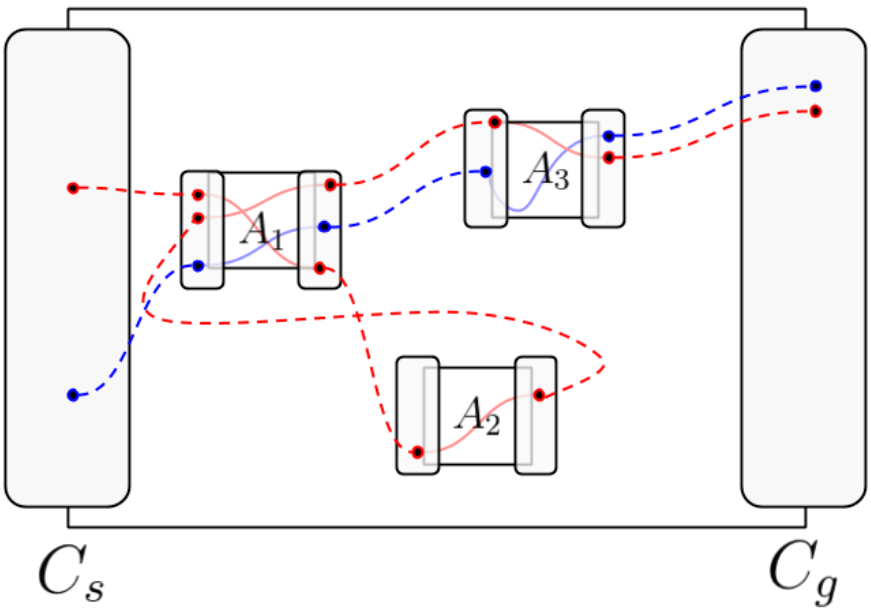

Figure 3.9: Automata Task Planning Action. Paths for the two different starting configurations are shown in different colors.

Chapter 4

Example Manipulation Task Specifications

4.1 Pick and Place

We will now give a fully specified example of performing a pick and place task (Fig.4.1). We wish to find a path through configuration space that results in a glass being moved from one table to another. The configuration space is $\mathcal{C} = \mathcal{C}_{\text{robot}} \times \mathcal{C}_{\text{glass}}$, where $\mathcal{C}_{\text{robot}}$ is the configuration space of the robot and $\mathcal{C}_{glass} = SE(3) \times \{0, 1\}$ is the pose of the glass along with a boolean flag indicating whether it is grasped by the manipulator. We wish to find a path through this space such that the glass is on top of a table.

A path could be found by directly searching through the entirety of configuration space for a plan that achieves this goal. However, doing so would be inefficient, as we intuitively know the general structure of a plan that can solve this task. What follows is an action hierarchy in the framework that can be used to restrict the space to make search more tractable.

At the top level our task can be visualized as a sequence of subactions:

1. Drive the base to a pose near the glass
2. Grasp the glass
3. Move the arm to a reasonable configuration for driving
4. Drive the base to a pose near the other table
5. Place the glass

The drive actions (1, 4) both compute base plans that end up near a space in the environment. These goals can be computed dynamically by sampling base poses of the robot near the target object.

The arm must also be moved for driving (3). It is possible that some configurations of the arm will prevent a base trajectory from being found. We plan the arm to a limited set of possible configurations for the arm, so that the arm is close to the robot and unlikely to prevent a base plan from being found.

The grasp and place actions are similar, and are sequences composed as follows:

Grasp Action:

1. Move the arm to a pose where the end effector is near the glass
2. Close the end effector
3. Mark the item as grasped

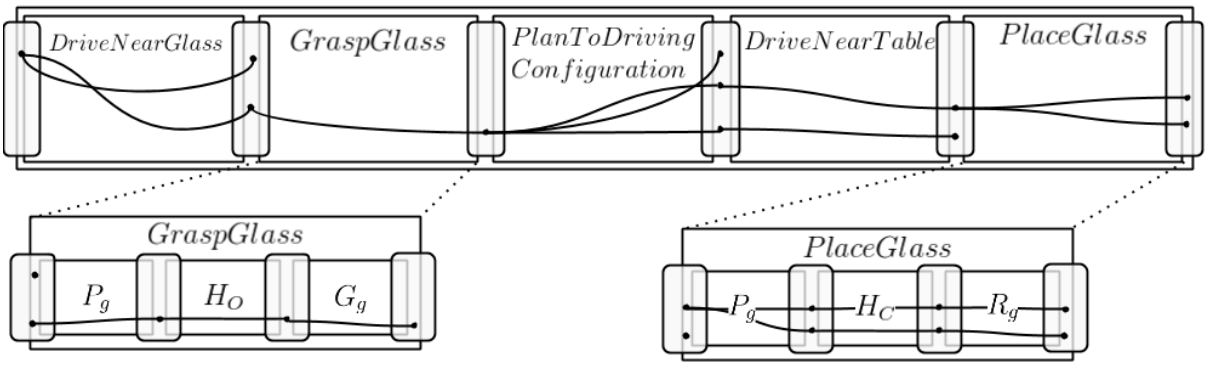

Figure 4.1: The task specification for a pick and place task that includes base movement.

Place Action:

1. Move the arm to a pose where the end effector places the glass on the target surface.
2. Open the end effector
3. Mark the item as not grasped

For moving the arm, we utilize the sampling pose set action defined above. This lets the actions dynamically generate poses, as the set of valid poses for grasping and placing the glass is dependent on the solutions found to the base planning problem.

The key point to realize from this example is how, even in very simple cases, restricting the planning space without discarding potential solutions requires dynamically generating constraints over the planning space.

4.2 Symbolic Table Rearrangement

Integrating symbolic planning allows for high level reasoning to determine the restricted space that is searched. Here we find a plan to rearrange blocks on a table. The symbolic planner returns sequences of actions which accomplish the symbolic goal.

Here are our available predicates:

- On(obj1, obj2)
- Graspable(obj)
- RobotIsGrasping(obj)
- NextTo(obj1, obj2)

Here are our available actions:

- GraspBlock(block)
  - Preconditions: *Graspable(block) AND RobotIsGrasping(block)*
  - Postconditions: *RobotIsGrasping(block) AND* $\forall$ *object* $\in$ *objects,* $\neg$*NextTo(block, object) AND* $\forall$ *object* $\in$ *objects,* $\neg$*On(block, object)*
- PlaceBlockOnBlock(block1, block2)
  - Preconditions: *RobotIsGrasping(block) AND* $\forall$ *object* $\in$ *objects,* $\neg$*On(object, block2*

    - Postconditions: *RobotIsGrasping(block) AND On(block1, block2)*
- PlaceBlockOnTable(block, table)
    - Preconditions: *RobotIsGrasping(block)*
    - Postconditions: *RobotIsGrasping(block) AND On(block1, table)*
- PlaceBlockNextToBlock(block1, block2, table)
    - Preconditions: *RobotIsGrasping(block) AND On(block2, table)*
    - Postconditions: *RobotIsGrasping(block) AND NextTo(block1, block2) AND On(block1, table)*

For our example the initial state will contain 3 blocks (A,B,C), one table T, and the predicate state will be *On(A, T) AND On(B, T) AND NextTo(A, B) AND On(C, A) AND Graspable(A) AND Graspable(B) AND Graspable(C)*.

Say our goal was *On(A,T) AND On(C,T) AND NextTo(C,A) AND On(B,C)*. A sequence of actions returned by the symbolic planner could be:

1. GraspBlock(C)
2. PlaceBlockNextToBlock(C,A,T)
3. GraspBlock(B)
4. PlaceBlockOnBlock(B,C)

Each of these actions will dynamically generate poses, plans for those poses, and grasp and release blocks as seen in the previous examples.

4.3 Table Clearing with Reconfiguration

In this example we will use the framework to specify a task for retrieving a glass from a cluttered table with n glasses and 1 plate. The configuration space is now $\mathcal{C} = \mathcal{C}_{\text{robot}} \times \mathcal{C}_{\text{plate,1}} \times \mathcal{C}_{\text{glass,1}} \times ... \times \mathcal{C}_{\text{glass,n}}$. We hope to find a plan which results in the robot grasping the plate. Our robot is unable to grasp the plate unless it is on the edge of the table. Therefore, it must first be slid to the edge.

To make this possible, some glasses may need to be moved. Other glasses may be blocking glasses that need to be moved. In the worst case scenario, this will become a geometric Tower of Hanoi problem. This means that not only do we not know what glasses to grasp and where to place them, but we also do not know the number of motion plans we will have to compute.

We now present a set of actions which restrict the space to this task. (Fig.4.2)

To focus on the motivation for this task, arbitrary sequences of plans, we make the assumption that the robot's base is in a location such that it can reach everything on the table necessary for completing the task. It is easy to see how base movement could be added, as in the previous example.

We will employ the repetition action to accomplish the task. The repetition action will repeat until a plan has been found that slides the plate to the edge of the table. Each time the repetition action runs, it will either move a glass, or attempt to slide the plate to the edge of the table.

The root action of the task is then the following sequence:

1. The repetition action
2. Grasp the plate

The repetition action repeats the following, a parallel action:

1. Pick and place a glass (as defined above in example 4.1)
2. Slide plate

It is natural to represent this plan in our framework. The search finds fully connected trajectories in configuration space, despite dynamically considering multiple sequences of different lengths over different plans.

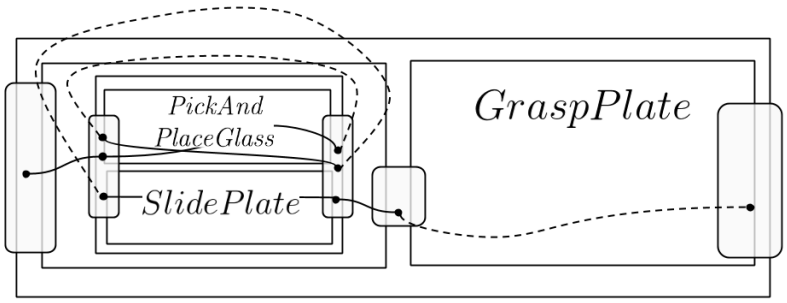

Figure 4.2: A graphical representation of the table reconfiguration task. Dashed lines signify configurations that are shared. In this case, configurations that did not make it into the goal set are returned to the start set of the repetition action. This happens twice, corresponding to two glasses that had to be moved before the plate could be slid.

Part II

Framework Policies

Chapter 5

Focusing Planning Time

In this chapter we develop a heuristic method for selecting which start and goal states to allocate computation time. At its core is Monte Carlo Tree Search and a system for assigning probability distributions over the likelihood the edge will be part of a path that reaches the goal.

5.1 Monte Carlo Tree Search for Configuration Space Planning

We first provide a formulation of MCTS for configuration space planning.

We restrict the scope of the configuration space planning problem as defined in Chapter 2. The algorithm is given a tree in which each node corresponds to a state and each edge corresponds to a trajectory. At each step the algorithm must choose a state to sample from. The edge is then immediately evaluated as either failed or successful.

Monte Carlo Tree Search has historically seen the most success applied to search trees for games such as computer Go. In configuration space planning, unlike computer Go, sampling and validating new states is incredibly expensive, and we do not have a function to evaluate the utility of a state.

The goal itself is also different. For game search trees, the goal is to reach a state that is likely to result in a win. In configuration space planning, the goal is to reach a state in the goal region.

This results in several key differences for when Monte Carlo Tree Search is applied to configuration space planning versus computer Go.

At its core, MCTS for configuration space planning uses a modification of UCT (Upper Confidence bound applied to Trees). Each state provides a distribution over the likelihood that its sampled state and its successor edges reach the goal state. MCTS uses UCT along with the expectation and variance of the distribution to choose which state to sample and evaluate. Intuitively, UCT balances exploration (distribution variance) and exploitation (distribution expectation).

The modification of UCT used, Bayes-UCT2, developed by Tesauro et. al [27], directly uses expectation and variance. The original UCT1 definition chooses the action which maximizes

$$B_i = r_i + \sqrt{2\ln(N)/n_i} \tag{5.1}$$

Where r_i is the average reward obtained in trials from action i, n_i is the number of trials run at action i, and N is the total number of trials.
Under Bayes-UCT2, the action is selected which maximizes

$$B_i = \mu_i + \sqrt{2\ln(N)\sigma_i^2} \tag{5.2}$$

where μ_i is the expectation of action i, σ_i is the standard deviation of action i, and N is the total number of planning attempts.
What is left is to determine the nature of the distributions at each state. The algorithm should take the minimum number of steps to produce a path to the end. Therefore, for each state we compute the distribution over the likelihood the sample and the paths eventually produced from it will reach the end.
As our distribution is over success probability, a Beta distribution is used. The α and β parameters correspond to our confidence in the edge's path's success and failure respectively. The α and β terms are calculated recursively using from the tree of edges. The success rate of a parent is determined by its children, combined with a prior.
We calculate α and β as follows,

$$Beta\left(\alpha_{prior} + N\frac{\sum_c E[P(c \to \text{end})]}{N}, \beta_{prior} + N\left(1 - \frac{\sum_c E[P(c \to \text{end})]}{N}\right)\right) \tag{5.3}$$

in which c is an existing child of the state and N is the number of children of the state. In the second, the beta distribution parameters are calculated to maximize the likelihood that the children were produced from the beta distribution.
The choice of prior has a huge impact on the distributions, as anytime a new state is sampled, its distribution will be based solely on its prior.

5.2 Prior Calculation

For each distribution we calculate a prior with a mean corresponding to our expected mean given the rest of the tree and a constant variance. Constant variance is achieved by setting α, β such that their sum equals 1. This then matches jeffrey's prior ($\alpha = .5, \beta = .5$) for distributions with no prior information.
For the mean prior, we desire to compute the likelihood a sample from a state reaches the end. For a given state, we compute its success likelihood by examining its siblings. The assumption is that it is likely a states' siblings are both the most similar to the state, and given no extra information, they should all be weighted equally in determining the success likelihood of the current state.
The μ used as a prior is calculated from a beta distribution as follows:

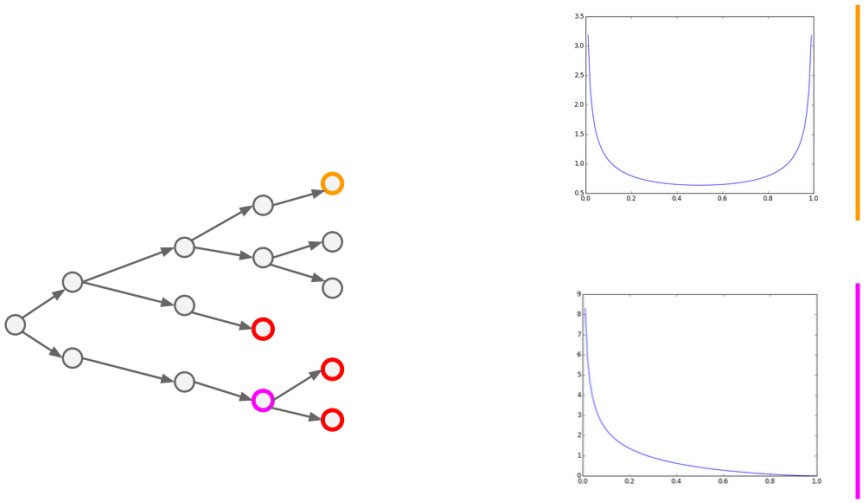

Figure 5.1: MCTS applied to a configuration space search problem. Red states are sampled states that failed as they could not be planned to. The orange state is highly uncertain, while the magenta state is more certain and has a distribution with worse expectation.

$$\mu_{prior} = E\left[Beta\left(\alpha_{prior} + N\frac{\sum_s E[P(s \to \text{end})]}{N}, \beta_{prior} + N\left(1 - \frac{\sum_s E[P(s \to \text{end})]}{N}\right)\right)\right]$$

Where s is the siblings of the target state.

This gives the recursive problem of determining the α and β prior to use for computing the sibling prior. Naturally, to get the expected success of a set of siblings, we look at the success rate of the sibling's cousin's. This proceeds recursively until the states have no additional cousins. At this point, Jeffrey's prior is used. Formally if S is the target state, and $C(k, S)$ returns the kth removed cousins of S (where $k = 1$ are siblings, $k = 2$ are first cousins, and so on),

$$\mu_{k,prior} = E\Bigg[Beta\Bigg(\alpha_{k+1,prior} + N\frac{\sum_{c \in C(k,S)} E[P(c \to \text{end})]}{N}, \beta_{k+1,prior} + N\left(1 - \frac{\sum_{c \in C(k,S)} E[P(c \to \text{end})]}{N}\right)\Bigg)\Bigg]$$

$$\alpha_{k,prior} = \mu_{k,prior}, \beta_{k,prior} = 1 - \mu_{k,prior}$$

For the priors of S, we calculate $\alpha_{0,prior}$ and $\beta_{0,prior}$, using $\alpha_{k,prior} = .5, \beta_{k,prior} = .5$ in the case where $C(k-1, S) = C(k, S)$.

5.3 Experimental Results for the Gate Traversal Domain

To improve intuition and provide an easily understandable domain for experimenting, we built a toy world, the gate traversal domain (Fig.5.2). The world is described as a list of 1 dimensional walls. The walls have openings in them. A point configuration p starts at a position p_1 before the first gate. The point configuration can transition from p_i to p_{i+1}, where $p_{i+1} \in [p_i - c, p_i + c]$, and c is a constant. A valid transition results in the point traveling to an opening.

A solution is a sequence $p_1 ... p_n$, where no p_i collides with a wall.

The problem is made difficult as only the failures and successes are known, the openings themselves are never directly observed.

If we allowed each wall to be of arbitrary dimension and allowed non-linear transition functions, we would have a problem space that includes the sequential robotics task planning problem as a special case. Restricting to a linear transition function and 1-dimensional walls has the advantage that we can easily visualize planner performance.

For creating test environments, we fix the number of walls and generate gates. Our experiments use gates generated by drawing two sets of values from two Dirichlet distributions. Walls are created by creating alternating regions of open gates and gaps between gates, where the size of the gates and the gaps are determined by their corresponding Dirichlet distributions. By modifying the parameters of the dirichlet distribution, we can control the expected spread and size of gates, where on one extreme gates are equivalently sized and evenly spaced, and on the other gates are of varying sizes and unevenly spaced.

We use this method to create two test environments. The first environment has 3 walls, and the second has 8. In each wall, 3 gates and 3 gaps are created. The 3 gates cover 20% of the total wall. The gates and walls are created with with a Dirichlet whose parameters are all 1, leading to somewhat unevenly spaced and unevenly sized gates. See Fig.5.2 for examples of generated environments.

Both the MCTS and normalized strategies were run for the two environments. For the 3 length gate world, 1000 worlds were generated for testing. For the 8 length gate world, 200 worlds were generated.

The success rate for each strategy for a maximum iterations budget is shown in Fig.5.3. Fig.5.4 shows the mean and variance for the number of iterations per trial, where unsuccessful trials were optimistically assumed to be solved if given one more iteration.

For 3 gates (MCTS: $\mu = 59.4, \sigma = 52.6$, Normalized: $\mu = 289.9, \sigma = 479.3, N = 1000$), A z-test determined the distributions to be different with $p < .0001$.

For 8 gates (MCTS: $\mu = 224.17, \sigma = 129.162343971$, Normalized: $\mu = 1913.795, \sigma = 4009.85483315, N = 200$), A z-test determined the distributions to be different with $p < .0001$.

5.4 Experimental Results for a Manipulation Task

We also created an environment to test the MCTS strategy on a pick and place task (Fig.5.5). The task is composed of the steps of driving to a table, picking up a glass, driving to another

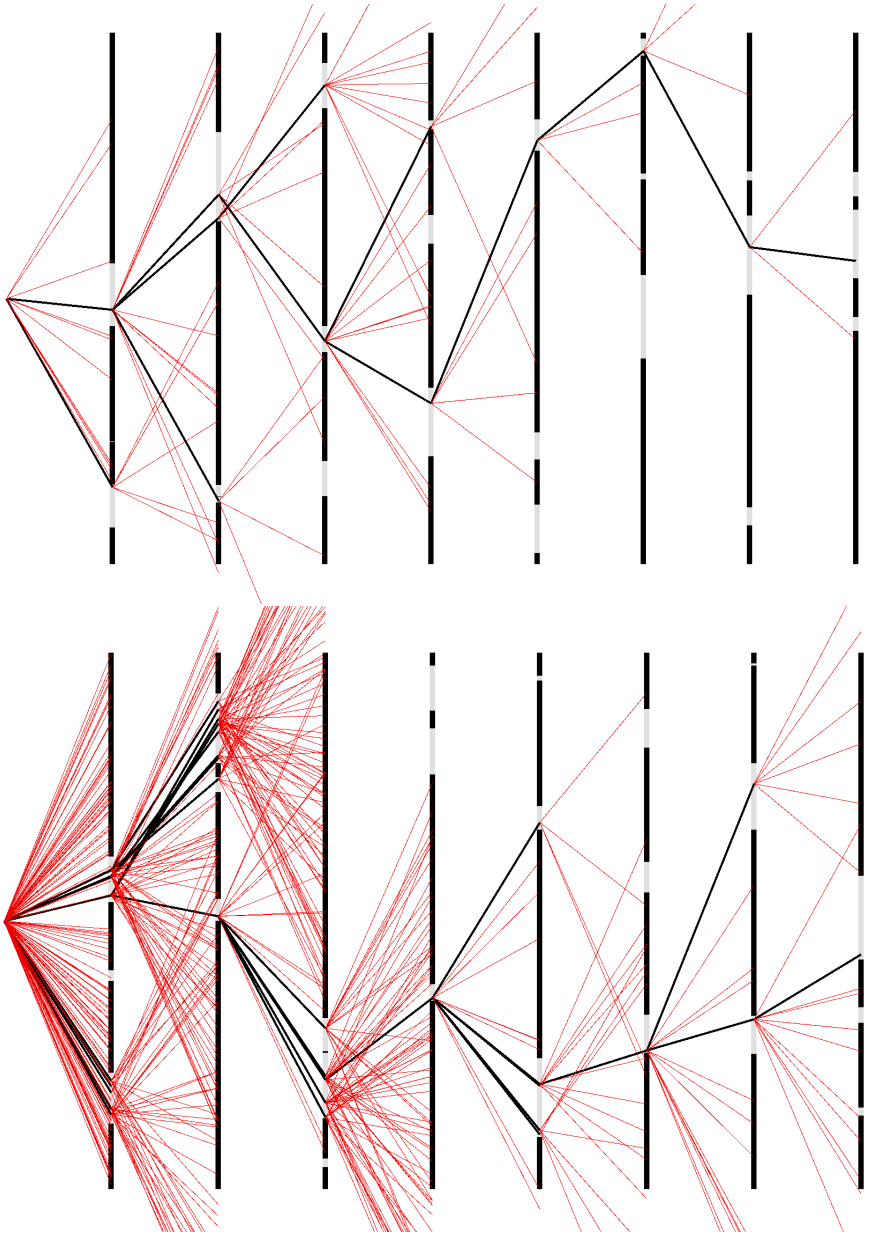

Figure 5.2: An example of a randomly generated world with 8 gates. The black regions are barriers. The planner starts at the left, and it samples configurations until it reaches the right. The top diagram shows paths found using the MCTS strategy, and the bottom shows paths found using the normalized strategy. Red paths are failures while black paths were successful.

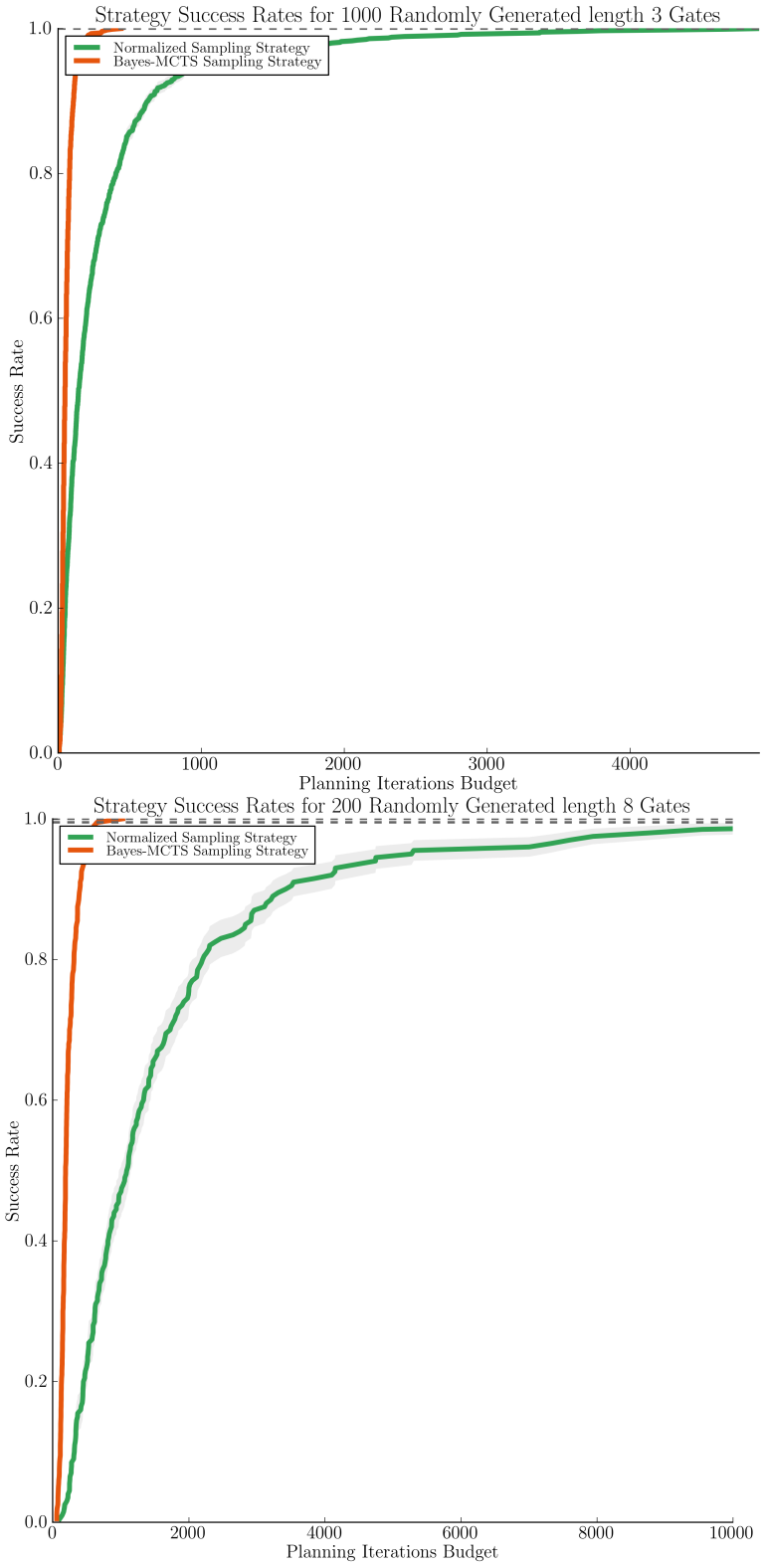

Figure 5.3: The success rates for different strategies for the gate traversal task.

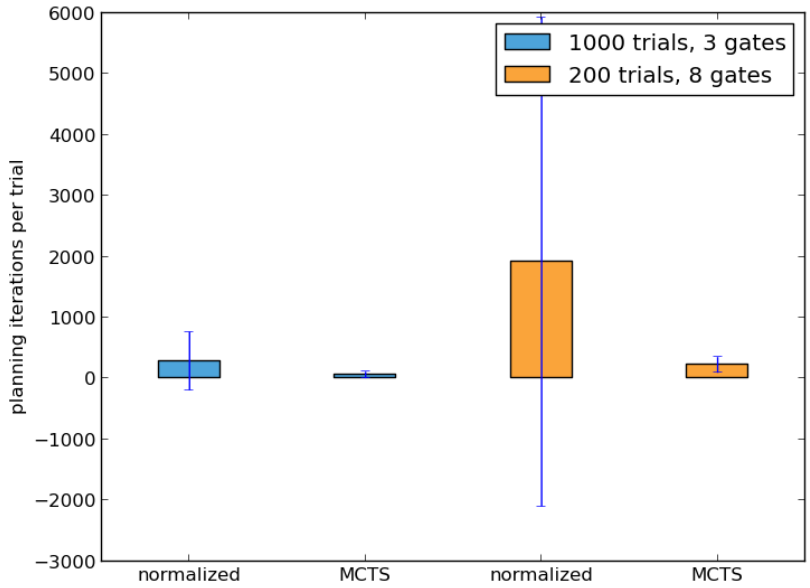

Figure 5.4: A comparison of the number of planning iterations per trial for the gate traversal task.

table, and placing the glass. Geometric dependencies exist between the steps of driving and manipulating the glass.

The success rate of each strategy for a maximum iterations budget is shown in Fig.5.6. Fig.5.7 shows the mean and variance for the number of iterations per trial, where unsuccessful trials were optimistically assumed to be solved if given one more iteration.

For the pick and place domain (MCTS: $\mu = 260.5, \sigma = 178.5$, Normalized: $\mu = 245.6, \sigma = 178.5, N = 30$), A z-test determined the distributions to be different with $p = .0431$.

5.5 Results Discussion

The MCTS strategy outperforms the Normalized strategy in each case.

For the gate traversal domain, the difference between the distributions within strategies is also of much greater magnitude for the normalized strategy versus the MCTS strategy. The normalized strategy gets $7x$ worse, with an $8x$ increase in standard deviation. The MCTS strategy gets $4x$ worse with only an $2.5x$ increase in standard deviation. This relationship in the form of a long tail can also be seen in the iterations budget. The tail is far smaller for MCTS, and grows far more slowly than seen for normalized.

A qualitative analysis of sampling patterns for the gate traversal domain gives way to the conclusion that under the normalized strategy, the data structure it grows biases future samples as to bias the planner away from good solutions. Clusters of poor samples lead the algorithm to increasingly exploring and sampling the poor region. The MCTS strategy sees some relief from

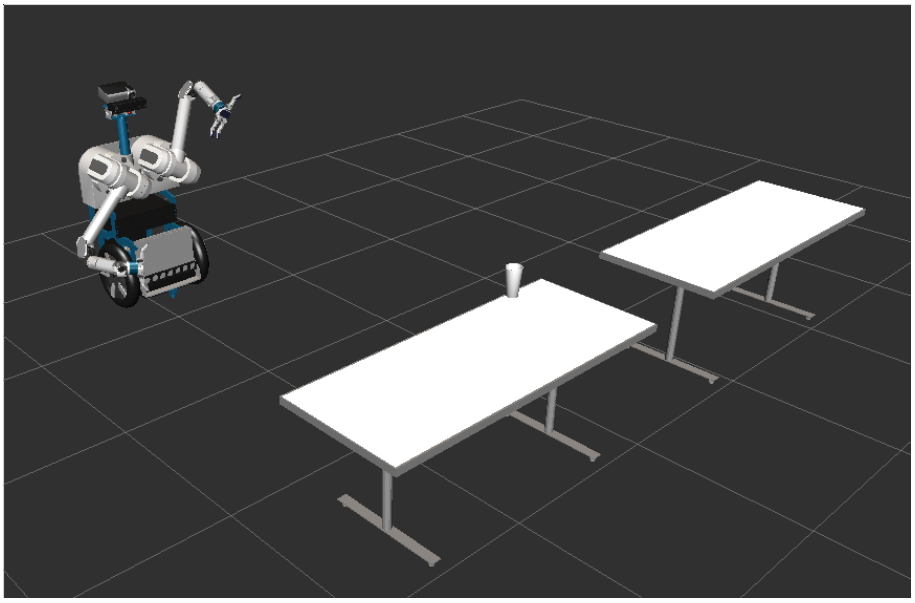

Figure 5.5: The Pick and Place Task. The robot must move the glass from one table to the other.

this problem as both the distributions it calculates and the consideration of variance in sample selection biases the planner away from overly sampled poor regions.

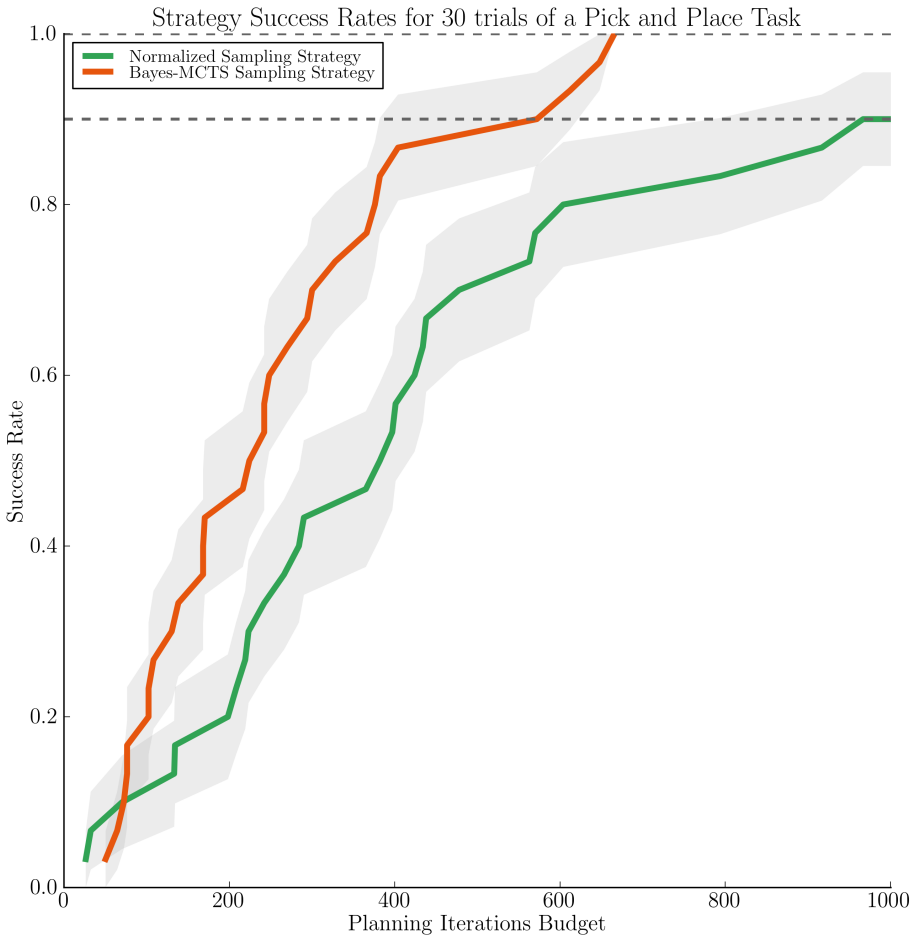

Figure 5.6: The success rate for different strategies for the pick and place task

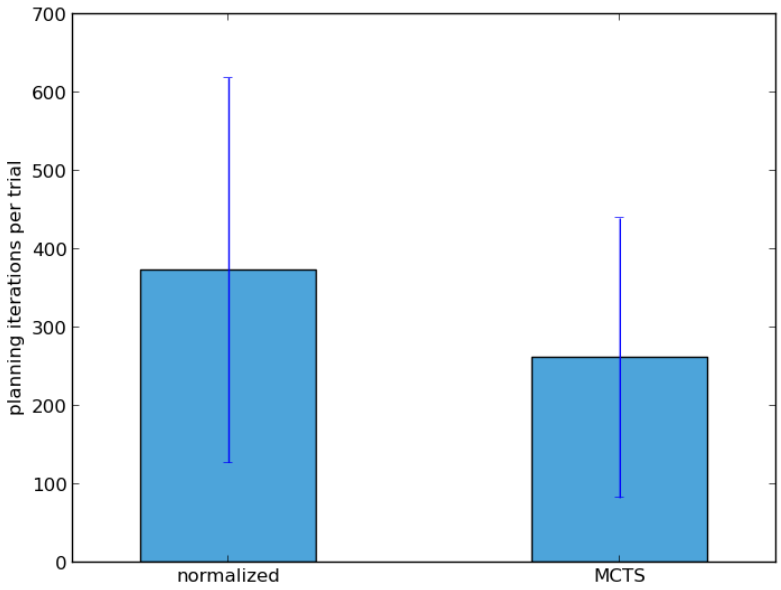

Figure 5.7: A comparison of the number of planning iterations per trial for the pick and place task

Chapter 6

Interleaving Planning and Execution

6.1 A Policy for Trajectory Tree Traversal

So far we have assumed that trajectories are not reversible and all portions of a task are dependent. That is, we never assume that there is a guaranteed intermediate solution from any particular intermediate state to the end. Here we remove the assumption that trajectories are not reversible and use that to interleave planning and execution.

We make one additional requirement, that trajectories compose a tree, instead of the more general possibility, a graph. Under this model, the nodes of the tree are states, connected by edges representing trajectories.

At any moment the real robot exists as a point on the trajectory tree, either on a node or on an edge. We consider the problem of, each time the robot reaches a node, deciding which edge (if any) to traverse. The execution problem is considered independently of the planning problem. The planner is not given knowledge of the robot state, and does not bias planning towards trajectories near the current real robot state.

We compare three strategies for traversing the trajectory tree.

6.1.1 Conservative Traversal Strategy

This is equivalent to the naive execution strategy without interleaving planning and execution, where backtracking is not assumed. The executor waits until a complete path has been found. Until a complete path has been found, the robot stays motionless at the root of the tree. When a complete path has been found, the robot follows the path until it reaches the goal.

6.1.2 Greedy Traversal Strategy

Under this strategy, the robot follows the path towards the node that is farthest from the root. The effectiveness of this strategy is dependent on the types of paths found during planning. It will perform better than traversal without backtracking only if, when the planner is complete, the robot has made forward progress on the correct path. If the robot is not on the correct path, because it is traversing a tree, it will have to pass through the root, and therefore will be slower.

We expect the greedy strategy to perform poorly as, while it optimizes the best case, in expectation, if there are multiple competing paths that are similar length, it will choose one at random and therefore need to not only go back to the root but also take the correct path. If there are k paths the same length n at the time the solution path of length $n+1$ is found, and the robot gets to the end of one of k paths at random, then in expectation the robot will need to traverse $\frac{k}{k+1}(2n+1)+\frac{1}{k+1}$ trajectories to get to the end. If k is 0, then the greedy algortihm will perform excellently, getting to the end in a single step. But if $k > 0$, then performance degrades, quickly approaching expected $2n$, way worse than the strategy that doesn't require backtracking.

6.1.3 Minimum Expected Edge Traversals Strategy

Following the analysis above, we will choose a strategy that minimizes the expected number of edge traversals to get to the goal. We make the assumption that there is a fixed probability r that a trajectory is found from a state. Therefore, two paths of equivalent length have equivalent likelihoods of being prefixes of the solution path.

We also assume that traversing each trajectory has a fixed cost.

For each state, we will calculate the sum of the cost to get to that state and the expected cost from that state to the goal.

For current state c and other state s, we calculate

$$C(s) = |c - s| + E[s \to goal]$$

In which

$E[s \to goal] = \sum_T p(T)|T|$

Where T is a trajectory starting from s that goes to the goal. Each trajectory consists of 2 components, the portion already in the trajectory tree, and the portion that needs to be found. Because there is a fixed number of levels, all trajectories with the same first component have the same likelihood for their second component. Therefore,

$$E[s \to goal] = \sum_{T_1} p(T_2)(|T_1| + |T_2|)$$

where T_1 is the first component of a trajectory, T_2 is the second component of the trajectory, and b is the expected branching factor.

$p(T_2)$ is the likelihood T_2 is found before any other path to the end. We consider all possible paths T from existing nodes in the tree. We can estimate the likelihood T_2 is found first as the likelihood a path is found from it and not found elsewhere. We make the assumption that the root of each T_2 has the same probability of being chosen. This underestimates the likelihood of each root, as each time a node is added to a root's tree, the likelihood of adding to that root's tree increases. If k is the number of unique roots of the to be found trees,

$$p(T_2) \approx \left(\frac{1}{k}\right)^{E[b]^{|T_2|}} \prod_T \left(1 - \left(\frac{1}{k}\right)^{E[b]^{|T|}}\right)$$

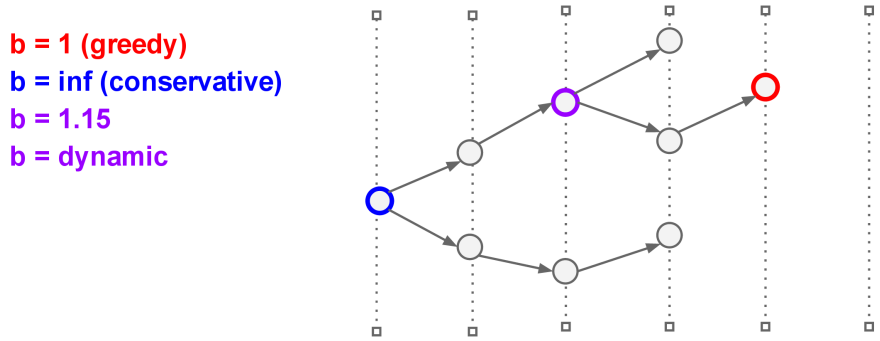

Figure 6.1: An example of a traversable trajectory tree

This gives a final equation of

$$C(s) = |c - s| + \sum_{T}(|T_1| + |T_2|)\left(\frac{1}{k}\right)^{E[b]^{|T_2|}} \prod_{T'}\left(1 - \left(\frac{1}{k}\right)^{E[b]^{|T'|}}\right)$$

If $b = 1$, then the equation simplifies to the greedy one. As b is increased, the algorithm becomes increasingly hesitant to move away from states that have multiple potential paths to the goal - for high b, it stays near the root until a full plan is discovered, matching the conservative strategy.

6.2 Experimental Results for Manipulation Tasks

We compare the policy for different values of b for the task of moving three glasses on a table to a bin (Fig.6.3. Oftentimes either the placement of the objects or failure from our underlying randomized motion planners causes the task planner to create multiple different trajectory sequences for moving items to the bin.
We compare $b = 1$ (greedy), $b = \infty$ (conservative), and dynamically setting b to be the average of the current tree branching factor.
We measure the execution time observed after planning is complete. For the conservative strategy, this equals the time to follow the entire trajectory. For the other strategies, this equals the time to get back to the root (if on the wrong path) plus the time to get from there to the end. This is the right metric as planning times vary wildly between different trials due to different glass configurations and randomized planner successes. Measuring time between finished planning and finished execution directly measures how well the interleaved planner chose the correct path.
Setting b dynamically resulted in values both within trials and between trials ranging between 1 and 1.3, with 1.1 as the most common value used.
In the task domain used, failures more often occurred on the first or second glass. So the greedy strategy was almost always able to be at least somewhat close to the right path and do better

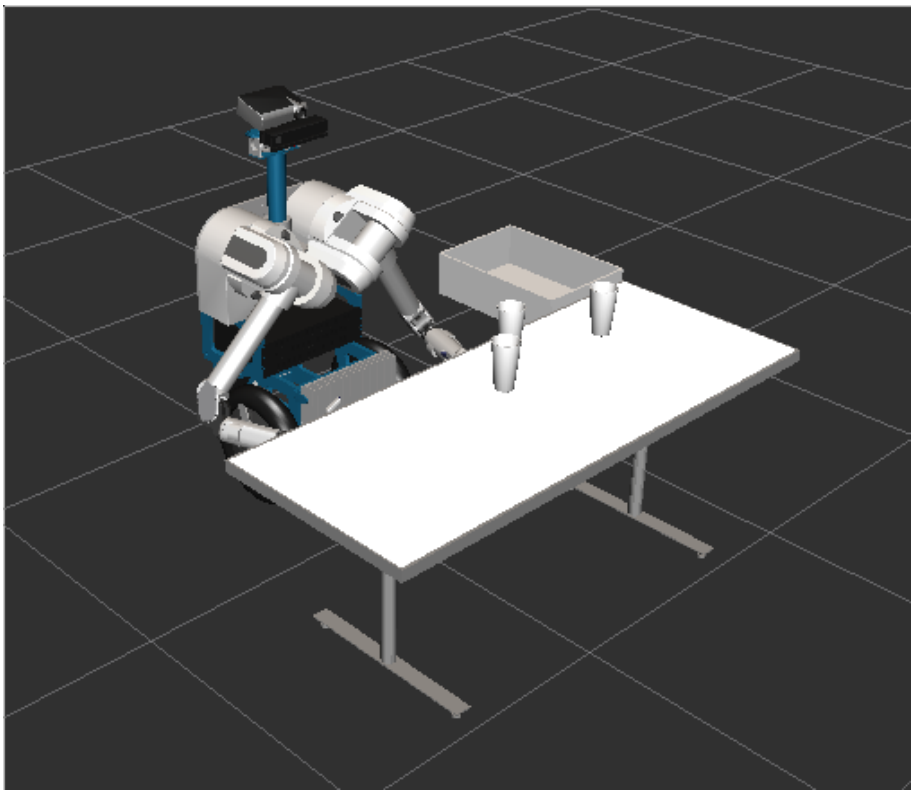

Figure 6.2: The task used for comparing strategies for interleaving planning and execution. The robot must move all three glasses into the bin.

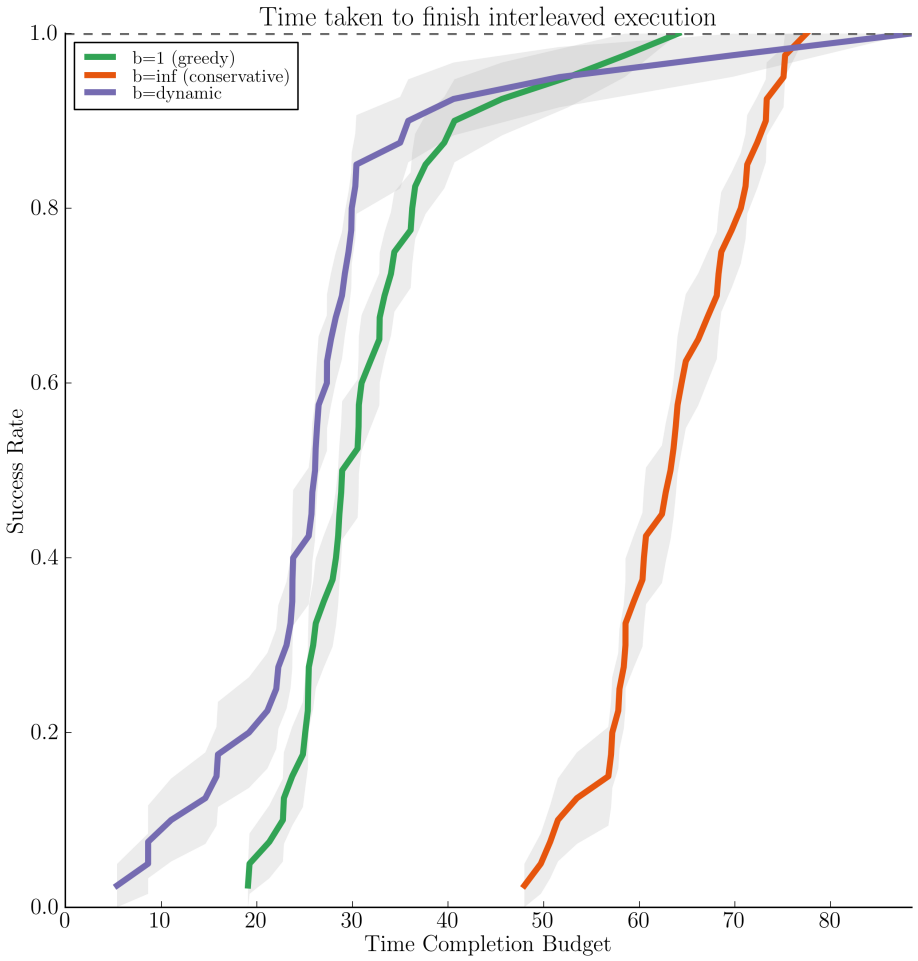

Figure 6.3: The distributions over execution time for different values of b.

than the conservative strategy. The dynamic strategy waited until some progress was made before leaving the root, causing it to be farther along the correct path than the greedy strategy, and therefore generally finish faster.

Part III

Conclusion

Chapter 7

Conclusion

7.1 Future Directions

The general nature of the framework provides many exciting possibilities for extensions. The following three extensions in particular are both generally applicable and highly influential across the operating space of the framework.

7.1.1 Parallel Search

The modular nature of the planning components makes it feasible to fully distribute and parallelize the search. Modules need only to communicate their start and goal state to other modules, otherwise they can operate fully independently. The effect would be a linear performance increase with a linear increase in computing power, as multiple branches can be explored independently.
This assumes the problem is one with a significant number of failures - if the problem can be solved sequentially with a high likelihood of success, parallelization will not help as the dependencies cannot be solved in parallel.

7.1.2 Learning Intermediate Goals

Another possibility for improving planner performance is improving the distributions from which intermediate goals are generated. Currently, intermediate goals are sampled from distributions determined by human experts. Figuring out the best distribution for a situation is challenging, due to the complex interactions and dependencies between the different components in the space.
Fine tuning and determining these distributions can be considered as a machine learning problem. A Gaussian process or other method could be applied here to learn a function that maps a given planning operation and object configuration to an intermediate goal. The assumption is that because the space is continuous, it is reasonable to smoothly interpolate intermediate goals between different object configurations. However, a danger with this approach is overfitting. We believe using a probabilistic approach such as a Gaussian process

should mitigate this problem by accounting for the degree of certainty of different regions of space.

7.1.3 Integrating Computation Focus with Interleaved Planning and Execution

Our current Monte Carlo tree search strategy functions independently of the current robot state. Similarly, the execution interleaving method functions independently of the Monte Carlo strategy. These two approaches could be integrated by considering not only sampling likelihood but also transition cost when selecting what state to sample. And, the interleaved function could consider sampling likelihood when computing the expected time from a state to the end.

7.2 Discussion

We have introduced and developed algorithms for a general purpose manipulation task planning framework. Future tasks for robots will be increasingly unconstrained and long horizon. Solving these tasks will require flexible systems that can reason over huge possibility spaces. To make this possible we have designed a framework that can encode a large variety of tasks that can be composed and efficiently executed.

We built the framework to work with modular, composable components. This makes it possible to reuse existing subtasks in new tasks. Maximizing reusability greatly reduces the work required by a human expert, as most task components have equivalent underlying requirements. Once a task, such as cleaning a table is implemented, it can be reused as part of a larger task, such as cleaning a room, with little to no integration overhead.

Achieving sufficient computational efficiency in such a scenario requires reasoning over the space of possible trajectories. The space is often too large to explore naively. For this purpose we have developed algorithms that use Monte Carlo Tree Search to intelligently choose what trajectories should be computed when. Interleaving planning and execution is also used to improve performance.

We have presented a framework that captures the logic necessary for restricting search spaces, and have demonstrated how it can be used to integrate existing planners to represent complex tasks. Experiments show how the framework can can be used to exploit intermediate goals and high level reasoning to efficiently solve these tasks. Through defining a system for dynamically restricting search and integrating existing planners, our framework provides a first step towards a general, extendable system for enabling complex plans to be easily represented and solved.

Bibliography

[1] Jennifer Barry, Kaijen Hsiao, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Manipulation with multiple action types. In *Experimental Robotics*, 2013. 1.2.1

[2] Jennifer Barry, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. A hierarchical approach to manipulation with diverse actions. In *IEEE International Conference on Robotics and Automation*, 2013. 1.2.1

[3] Michael Beetz, Lorenz Mosenlechner, and Moritz Tenorth. Cram - a cognitive robot abstract machine for everyday manipulation in human environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010. 1.2.1

[4] Dmitry Berenson, Siddhartha Srinivasa, and James Kuffner. Task space regions: A framework for pose-constrained manipulation planning. *International Journal of Robotics Research (IJRR)*, 30(12):1435 – 1460, October 2011. 3.2.4

[5] J Bohren and Steve Cousins. The smach high-level executive. In *IEEE Robotics & Automation Magazine*, 2010. 1.2.1

[6] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, Simon Colton, et al. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012. 1.2.2

[7] Stephane Cambon, Rachid Alami, and Fabien Gravot. A hybrid approach to intricate motion, manipulation and task planning. In *IJRR*, 2009. 1.2.1

[8] Edmund M Clarke, Orna Grumberg, and Doron Peled. Model checking. 1999. 3.3.2

[9] Christian Dornhege, Marc Gissler, Matthias Teschner, and Bernhard Nebel. Integrating symbolic and geometric planning for mobile manipulation. In *IEEE International Symposium on Safety, Security, and Rescue Robotics*, 2009. 1.2.1

[10] Esra Erdem, Kadir Haspalamutgil, Can Palaz, Volkan Patoglu, and Tansel Uras. Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In *IEEE International Conference on Robotics and Automation*, 2011. 1.2.1

[11] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Ffrob: An efficient heuristic for task and motion planning. In *Workshop on the Algorithmic Foundations of Robotics*, 2014. 1.2.1

[12] Andre Gaschler, Ronald PA Petrick, Torsten Kröger, Oussama Khatib, and Alois Knoll.

Robot task and motion planning with sets of convex polyhedra. In *Robotics: Science and Systems*, 2013. 1.2.1

[13] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011. 1.2.2

[14] Fabien Gravot, Stephane Cambon, and Rachid Alami. asymov: a planner that deals with intricate symbolic and geometric problems. In *International Symposium on Robotics Research*. 2005. 1.2.1

[15] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical task and motion planning in the now. In *IEEE International Conference on Robotics and Automation*, 2011. 1.2.1

[16] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Integrated task and motion planning in belief space. In *IJRR*, 2013. 1.2.1

[17] Lydia E. Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE International Conference on Robotics and Automation*, 1996. 2.2

[18] Fabien Lagriffoul, Dimitar Dimitrov, Alessandro Saffiotti, and Lars Karlsson. Constraint propagation on interval bounds for dealing with geometric backtracking. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012. 1.2.1

[19] Steven M LaValle and James J Kuffner. Randomized kinodynamic planning. In *IJRR*, 2001. 1.2.1, 2.2

[20] Zakary Littlefield, Athanasios Krontiris, Andrew Kimmel, Andrew Dobson, Rahul Shome, and Kostas E Bekris. An extensible software architecture for composing motion and task planners. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 327–339. Springer, 2014. 1.2.1

[21] Tomás Lozano-Pérez and Leslie Pack Kaelbling. A constraint-based method for solving sequential manipulation planning problems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014. 1.2.1

[22] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998. 1.2.1

[23] Erion Plaku and Gregory D Hager. Sampling-based motion and symbolic action planning with geometric and differential constraints. In *IEEE International Conference on Robotics and Automation*, 2010. 1.2.1

[24] Erion Plaku, Lydia E. Kavraki, and Moshe Y Vardi. Motion planning with dynamics by a synergistic combination of layers of planning. In *IEEE International Conference on Robotics and Automation*, 2010. 1.2.1

[25] Thierry Siméon, Juan Cortés, Anis Sahbani, and Jean-Paul Laumond. A general manipulation task planner. In *Algorithmic Foundations of Robotics V*. 2004. 1.2.1

[26] Siddharth Srivastavah, Eugene Fang, Lorenzo Riano, Rohan Chitnis, Stuart Russell, and Pieter Abbeel. Combined task and motion planning through an extensible planner-independent interface layer. In *IEEE International Conference on Robotics and Automa-*

tion, 2014. 1.2.1

[27] Gerald Tesauro, VT Rajan, and Richard Segal. Bayesian inference in monte-carlo tree search. *arXiv preprint arXiv:1203.3519*, 2012. 1.2.2, 5.1

[28] Jason Wolfe, Bhaskara Marthi, and Stuart J Russell. Combined task and motion planning for mobile manipulation. In *International Conference on Automated Planning and Scheduling*, 2010. 1.2.1