Learning to guide task and motion planning

by

Beomjoon Kim

BMath, University of Waterloo (2012)

MSc, McGill University (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
July 23, 2020

Certified by .....
Leslie Pack Kaelbling
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by .....
Tomás Lozano-Pérez
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Learning to guide task and motion planning

by

Beomjoon Kim

Submitted to the Department of Electrical Engineering and Computer Science
on July 23, 2020, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

How can we enable robots to efficiently reason both at the discrete task-level and the continuous motion-level to achieve high-level goals such as tidying up a room or constructing a building? This is a challenging problem that requires integrated reasoning about the combinatoric aspects of the problem, such as deciding which object to manipulate, and continuous aspects of the problem, such as finding collision-free manipulation motions, to achieve goals.

The classical robotics approach is to design a planner that, given an initial state, goal, and transition model, computes a plan. The advantage of this approach is its immense generalization capability. For any given state and goal, a planner will find a solution if there is one. The inherent drawback, however, is that a planner does not typically make use of planning experience, and computes a plan from scratch every time it encounters a new problem. For complex problems, this renders planners extremely inefficient. Alternatively, we can take a pure learning approach where the system learns, from either reinforcement signals or demonstrations, a policy that maps states to actions. The advantage of this approach is that computing the next action to execute becomes much cheaper than pure planning because it is simply making a prediction using a function approximator. The drawback, however, is that it is brittle. If a policy encounters a state that is very different from the ones seen in the training set, then it is likely to make mistakes and might get into a situation from which it does not know how to proceed.

Our approach is to take the middle ground between these two extremes. More concretely, this thesis introduces several algorithms that learn to *guide* a planner from planning experience. We propose state representations, neural network architectures, and data-efficient algorithms for learning to perform both task and motion level reasoning using neural networks. We then use these neural networks to guide a planner and show that it performs more efficiently than pure planning and pure learning algorithms.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor of Computer Science and Engineering

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor of Computer Science and Engineering

Acknowledgement

It has been tremendous five years. Thanks to my friends, mentors, and colleagues that I met along this journey, I have learned and grown a lot both as a person and as a researcher during my Ph.D.

First and foremost, I am extremely grateful to my advisors, Leslie and Tomás, who have been just fantastic mentors. They taught me some of the most important skills and mindsets for becoming a good researcher – writing, speaking, formulating problems, organizing and critiquing research ideas, and knowing when to slow down, to name a few – and have been extremely patient during this process. As I also embark on the journey of starting my research lab, I will try my best to give back the love, care, and patience that I have received.

I am grateful to my comrades, the LIS lab members, for their friendship. I am thankful to Caelan, who was always there to have a discussion; to Rachel, for making me realize the importance of having a *good* presentation; to Rohan and Tom for a collaboration filled with humor; to Ferran for being a source of motivation; to Zi and Zelda for making 414 so much fun to be in; to Luke for reminding me of the importance of get-things-done attitude; and to Yoon for pondering together about what an ideal lab and research advisor might be.

Outside of research, I am grateful to Moosun and Byongjun who have enriched my life. Our late-night snacks and drinks, marathons on Havard Bridge, and visits to Newbury street kept my life on balance.

Last but not least, I am deeply grateful to my parents. About 20 years ago, my parents decided to immigrate to Canada. They had to sacrifice their jobs, families, and friends. They had to struggle to learn a new language and adapt to a new country. And they this all these for one thing, and one thing only: better education for their children. I will be forever thankful for their unconditional love. This thesis is dedicated to them.

Table of contents

1
Introduction

We are facing an increasing demand for intelligent robots that can perform a wide range of tasks that are mundane, dangerous, and demanding for humans. Unlike traditional robots that mostly operate in factories, these robots must operate in unstructured environments such as construction sites, disaster areas, or homes, and manipulate various objects efficiently and robustly to accomplish the given tasks.

We can formalize many such complex tasks as *task-and-motion planning* (TAMP) problems. TAMP involves combined planning of task and motion level decisions to achieve a high-level goal, such as building a structure or putting away groceries. It is an extremely difficult planning problem that involves a large number of objects, hybrid search space, long planning horizon, and expensive action feasibility checking.

The typical approach to TAMP problems is to design relatively general-purpose planning algorithms with domain-independent heuristics for search guidance [118, 53, 43, 15, 124, 37, 36]. The advantage of such pure planning approaches is that a planner can, most of the time, guarantee a form of *completeness*, which means that for any given initial state and goal pair, it will eventually find a solution if a solution exists. The major drawback, however, is that it is computationally inefficient. TAMP planners do not typically have the ability to learn from past planning experience, and must solve difficult TAMP problems from scratch even when the current problem is similar to the ones it has solved in the past.

Alternatively, we can take a pure learning approach where we learn a policy that maps a state of the environment to an action using reinforcement or imitation learning algorithms [122, 2]. The benefit of this approach is the execution-time computational efficiency: computing the next action to execute comes down to making a prediction from a function approximator, rather than performing an expensive search procedure. Its disadvantage, however, is that learned policies are often very fragile. If a policy encounters a state that is very differ-

ent from the ones seen in training, it may make mistakes and might get into a situation from which it does not know how to proceed. Collecting a large amount of data could be a solution, but data tends to be expensive in robot manipulation problems.

Based on these observations, we take the middle ground between these two extremes. We propose a framework that, given a set of past planning experiences, learns to *guide* a planner by learning search guidance predictors. This approach combines the best features of both pure planning and pure learning. It is more efficient than the pure planning approach because the predictors guide the search to a more promising region of the search space; because it can rely on the planner to correct for mistakes the predictors may make, it is far less fragile than the pure learning approach. Developing such a framework, however, raises three fundamental challenges:

Representation How can we design a representation of a problem state that consists of poses, shapes, and symbolic attributes of objects that can generalize across different environments and tasks? A typical fixed-sized vector representation cannot directly be applied because different problems may involve different numbers of objects. Moreover, such a representation cannot encode the relational information among objects, such as which object is occluding which other objects, that is essential in evaluating the feasibility of robot actions.

Learning How can we design a data-efficient algorithm that can learn from planning experience? Unlike reinforcement learning experience with a simulator or imitation learning dataset, planning experience is a set of search trees. Each tree involves a single *positive* trajectory that led to a goal, and several *neutral* trajectories which may have lead to the goal have we put in more search efforts. While it is possible to learn just from the positive trajectories, We need learning algorithms that can use both types of data for data efficiency.

Exploration vs. exploitation The central argument for learning to *guide* planning is that we can recover from prediction mistakes on search guidance by choosing actions that are not necessarily suggested by the predictors. How can we design a planning algorithm that efficiently balances between exploiting the prior search guidance knowledge and exploring new choices? Can give a performance guarantee for such a planning algorithm?

To address these challenges, we first consider an important subclass of TAMP called geometric task-and-motion planning (G-TAMP)

in Part I. We then consider the full TAMP problem in Part II. The contributions of this thesis are follows:

- Chapter 2 formulates the G-TAMP problem class and presents a planning algorithm. The algorithm is an extension of heuristic graph search to G-TAMP [64] and handles the expensive feasibility checking and hybrid search space.
- Chapter 3 presents a strategy for guiding the discrete search in G-TAMP problems by learning a ranking function that ranks discrete choices at each state. We present a novel abstract state representation that encodes relational information such as object occlusion and reachability that is sufficient for making discrete decisions. Using this representation, we represent our ranking function as a graph neural network that takes the graph of the scene which encodes this relational information as an input [64].
- Chapter 4 presents a strategy for guiding the continuous search in G-TAMP problems by learning biased samplers. We propose a novel *key-configuration* state representation that approximates the true configuration space (c-space) obstacles using a sparse set of important configurations, which enables the sampler to reason in the c-space. We propose two adversarial learning algorithms that can use both neutral and positive trajectories [61, 62] to learn samplers.
- Chapter 6 presents a planning algorithm that balances exploiting the prior search guidance knowledge and exploring other choices to more efficiently find a solution. It is an extension of Monte Carlo Tree Search to continuous action spaces based on a novel budgeted-black-box function optimization algorithm. We provide performance guarantees on this algorithm [63].
- Chapter 7 changes focus and presents a strategy for guiding the search for general TAMP problems, which may involve purely symbolic attributes of objects in addition to geometric attributes. To deal with such diverse attributes, we propose a *score-space* representation that directly reasons with the scores of potential solutions. We assume that we are given a *plan skeleton*, and the goal is to find the continuous parameters of the skeleton. To deal with mistakes in prediction, we propose an exploration vs. exploitation algorithm that suggests the next potential continuous parameters based on the scores of the ones tried so far [60, 66].
- Chapter 8 makes a connection between the representation and prediction algorithm presented in the previous chapter to Bayesian Optimization (BO). We propose a new class of algorithms called

meta-BO, which have access to past optimization experience, and the goal is to more efficiently optimize the current function based on the experience. We provide a theoretical guarantee on our algorithm based on the regret analysis tools in BO [65].

- Chapter 9 presents conclusions and directions for future work.

Part I

Geometric task-and-motion planning

2

A planning algorithm for G-TAMP

In this chapter, we focus on a sub-class of TAMP problems that we call *geometric task-and-motion planning* (G-TAMP), in which we are interested in moving a set of objects from one region to another among movable obstacles. In particular, we will present a planning algorithm called SAmpling-based Priority Search (SAPS) that can handle hybrid search space and expensive action feasibility checking.

G-TAMP is a particularly important sub-class of TAMP problems because it occurs as a subproblem of every TAMP problem: whether it is cooking a meal, constructing a building, or simply putting away groceries, a robot must efficiently reason about how to arrange obstacles to move objects to desired regions. Therefore, we believe if we can solve G-TAMP problems more efficiently, then we can solve TAMP problems more efficiently as well. Examples of G-TAMP problems are shown in Figure 2.1.

We will defer all the experimental results of Part I to chapter 5.

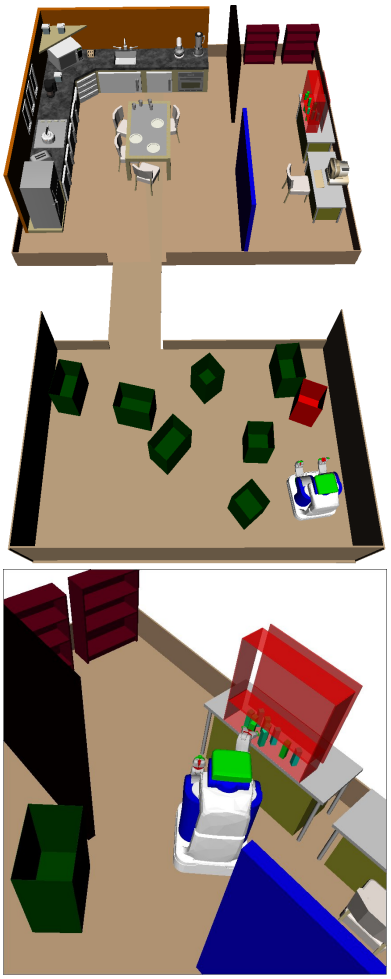


Figure 2.1: Examples of G-TAMP problems

2.1 *Geometric task-and-motion planning problem formulation*

We assume that the environment of a G-TAMP problem consists of a set of fixed rigid objects $\mathbf{O}^{(F)} = \{o_i^{(F)}\}_{i=1}^{n_1}$, a set of movable rigid objects $\mathbf{O}^{(M)} = \{o_i^{(M)}\}_{i=1}^{n_2}$, and a set of workspace regions $\mathbf{R} = \{r_i\}_{i=1}^{n_3}$.

A state of the system is determined by the poses of the movable objects, where the pose of i^{th} object is denoted $\mathrm{P}_i^{(M)}$, and the configuration $c \in \mathcal{C}$ of the robot. The poses of the objects and regions are defined relative to a *parent* object, which can be a movable object such as a tray, or a fixed object, such as a floor. We denote a state as $s \in \mathcal{S}$ where $s = (\mathrm{P}_1^{(M)}, \cdots, \mathrm{P}_{n_1}^{(M)}, c)$. All objects and regions have known and fixed shapes. We assume that states are fully observable.

The action space consists of a set of n_o manipulation operators, $\mathcal{O} = \{\mathfrak{o}_1, \cdots, \mathfrak{o}_{n_o}\}$. An operator can cause collision-free motion

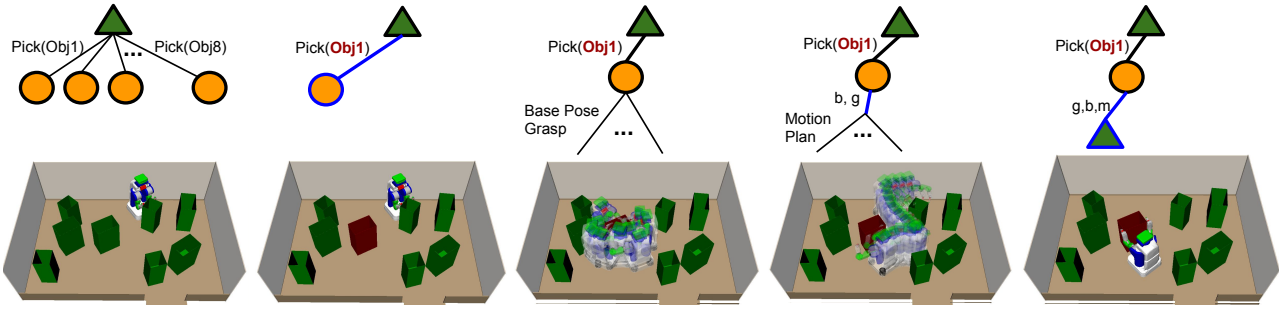


Figure 2.2: Part of a search tree for SAPS. Green triangle nodes denote nodes in which abstract action choices are available, and orange circle nodes denote the ones where continuous parameter choices are available. In the initial state shown in the left-most figure, the robot has 8 available abstract actions. It chooses to explore abstract action Pick(obj_1). The continuous parameters of Pick manipulation operator, which consists of base pose and grasp, are sampled using random sampling; ones that are in collision or does not have a feasible IK solution are rejected. Once feasible base pose b and grasp g are sampled, SAPS then calls a motion planner to see if there exists a motion plan to the selected base pose, and finds feasible motion m. A new node is then added to the search tree, in which the robot is at a new configuration, holding object the object with manipulation operation Pick(obj_1, (g,b)).

of a single object. Each operator takes as inputs a fixed number of operation-specific continuous parameters $\kappa \in K_o$, and discrete parameters $\delta \in \mathbf{O}^{(M)} \times \mathbf{R}$. The robot might have one or more such operators available, such picking-and-placing, pushing, or throwing.

For instance, the PICKANDPLACE operator had discrete parameters specifying the object to move and the region to place it into. As continuous parameters, it takes a grasp and placement pose for the object. Each of the operators has a set of feasibility constraints, and we are given a set of external feasibility checker functions that can check the feasibility of the given parameters. Examples of such feasibility checkers could be an inverse-kinematics (IK) solver and a motion planner, which can be thought of as checking the feasibility of a constraint of the existence of a collision-free path between two configurations

We will call the pair of operator type and its discrete parameters as an *abstract action*, which we denote with $o(\delta)$. For instance, an abstract action

$$\text{PICKANDPLACE}(\delta = (obj_1, table\_top))$$

specifies the robot moves obj_1 to $table\_top$ using pick-and-place. We will define the continuous parameters associated with an abstract action as a *continuous action*. An *action*, denoted $o(\delta, \kappa)$, is a concrete operation that can be executed by the robot. For instance, we may have

$$\text{PICKANDPLACE}\,(\delta = (obj_1, table\_top), \kappa = (g, p))$$

for pick-and-placing obj_1 to $table\_top$ using the continuous action that consists of grasp g and the placement pose p on $table\_top$. Associated with an action is a low-level robot motion expressed as a sequence of configurations, which is obtained by calling a motion planner, if the operator instance is feasible.

Each action $o(\delta, \kappa)$ induces a mapping $T(\cdot, o(\delta, \kappa))$ from a world state s, in which it is executed, to a resulting world state $s' \in \mathcal{S}$. If

the operation cannot be legally executed in s, we let $s' = s$ or an absorbing "failure" state.

We specify a goal set $\mathcal{G}$ as a conjunction of statements of the form $\textsc{InRegion}(o, r)$, where $o \in \mathbf{O}^{(M)}$ and $r \in \mathbf{R}$, which are true if o is contained entirely in region r. It is also possible to specify the final robot configuration as part of the goal for NAMO problems [120], or final poses of objects for specifying rearrangement problems [76, 67].

A G-TAMP *planning problem* is characterized by $(\mathbf{O}^{(M)}, \mathbf{O}^{(F)}, \mathbf{R}, s_0, \mathcal{O}, \mathcal{G}, T)$, where $(\mathbf{O}^{(M)}, \mathbf{O}^{(F)}, \mathbf{R})$ defines the *environment* and s_0 is the initial state. The objective is to find a sequence of operator instances that changes the state from s_0 to a state that satisfies $\mathcal{G}$.

2.2 *A planning algorithm for geometric task-and-motion planning*

There are two key distinctions between G-TAMP problems and a typical graph search problem. The first is that G-TAMP problems involve a hybrid search space that consists of discrete task-level and continuous motion-level decisions. The second is that they involve expensive action feasibility checking: finding a feasible pick-and-place action, for example, requires a call to a motion planner and an inverse kinematics solver, making the generation of successor states expensive.

To deal with these problems, we propose SAPS. Given a problem instance, SAPS searches forward, first branching on the choice of an *abstract action*, which specifies the type of a manipulation operator and its discrete parameters, such as *pick* $object_1$. Then, given the choice of abstract action, it branches on a *continuous parameters* of the operator, such as the grasp and base pose to pick the object, using random sampling. The planner calls feasibility checker, such as motion planners and inverse kinematics solver on the sampled parameters, and the next state is generated. Figure 2.2 shows an example of a search tree of a G-TAMP problem.

Unlike traditional heuristic search, in which a state is expanded and its successors are added to the queue, SAPS maintains a priority queue of state-and-abstract-action pairs. If an effective priority function and continuous parameter samplers are available, then we can find a plan more efficiently, by prioritizing expensive feasibility checking on promising abstract actions and continuous parameters first.

We begin by introducing our abstract state representation based on geometric predicates. The non-learning version of SAPS uses a hand-designed priority function based on this abstract state representation.

2.2.1 *Abstract state representation*

For G-TAMP problems, the important information that must be captured in an abstract state representation is which objects are occluding which other objects and regions. Once we have this information, the planner can use it to efficiently plan operator instances that clears necessary obstacles, and move the goal objects into goal regions.

To capture this information in a state, we make use of the assumption that each action manipulates a single object to move it to a region or to a different pose within its region. Under this assumption, we can characterize the preconditions of executing $\mathfrak{o}(\delta, \kappa)$ with two volumes of workspace, where $\delta = (o, r)$ for some object o and for some region r, and κ is chosen so that if the robot uses action $\mathfrak{o}(\delta, \kappa)$, then it can move o to region r.

The first is the volume $V_{\text{pre}}(q, \mathfrak{o}(\delta, \kappa))$, which is the swept volume that the robot must move through from its current configuration q to a configuration q' in which o can be reached. The second is the volume $V_{\text{manip}}(\mathfrak{o}(\delta, \kappa))$, which is the swept volume that the robot and object must move through, from their configurations at the beginning of the operation, (P_o, q'), to their configurations at the end of the operation, as determined by continuous parameters κ. Figure 2.3 shows examples of these two swept volumes.

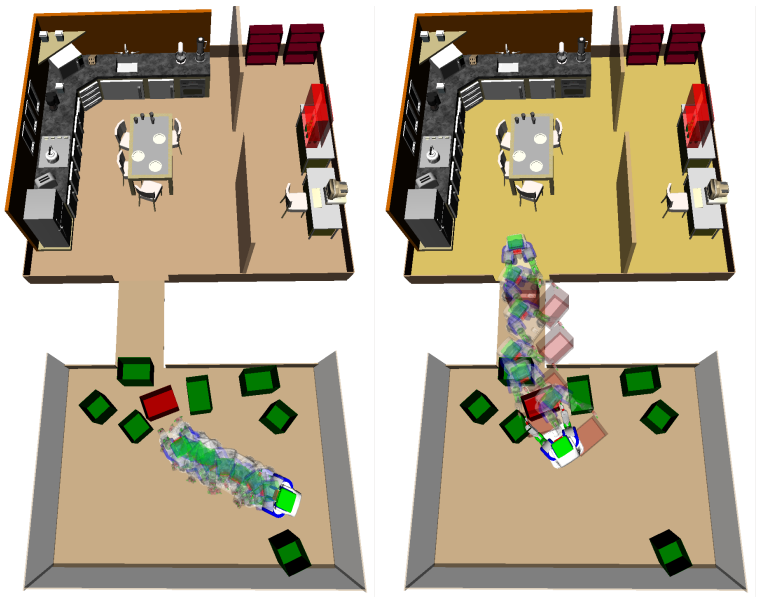


Figure 2.3: Left: $V_{\text{pre}}(q, \mathfrak{o}(\delta, \kappa))$ where $\mathfrak{o}$ is pick-and-place, δ is the red box, q is configuration of the solid robot. κ is not shown. Right:$V_{\text{manip}}(\mathfrak{o}(\delta, \kappa))$ where $\mathfrak{o}$ is pick-and-place, δ is the red box and home region marked yellow, and κ is the robot's base pose at the end of the trajectory inside the home region

We construct a relational abstract representation of the state $s = (\mathrm{P}_{o_1^{(M)}}, \ldots, \mathrm{P}_{o_n^{(M)}}, q)$ and goal $\mathcal{G}$, denoted $\alpha(s, \mathcal{G})$, as a conjunction of all true instances of the following relations, applied to *entities* $e \in \mathbf{O}^{(M)} \cup \mathbf{R}$:

- IsRegion(e), true if e is a region;
- IsObject(e), true if e is an object;
- IsGoal(e), true if e is mentioned in the goal specification $\mathcal{G}$;
- InRegion(o, r), true if object o is currently in region r;
- PreFree(o), true if $\exists \kappa$ such that $V_{\text{pre}}(q, \mathfrak{o}((o, r), \kappa))$ is collision-free;
- ManipFree(o, r), true if $\exists \kappa$ such that $V_{\text{manip}}(\mathfrak{o}((o, r), \kappa))$ is collision free;
- OccludesPre(o_1, o_2), true if o_1 is an object that overlaps the swept volume $V_{\text{pre}}(q, \mathfrak{o}((o_2, r), \kappa))$, where κ is chosen to avoid collisions if possible; and
- OccludesManip(o_1, o_2, r), true if o_1 is an object that overlaps the swept volume $V_{\text{manip}}(\mathfrak{o}((o_2, r), \kappa))$, where κ is chosen to avoid collisions if possible.

The detailed implementations for the last four relations are specific to an operator *type*, such as pick-and-place or pull. The value of any of these predicates, if applied to arguments that are clearly the wrong type, is false.

Given a state s and goal $\mathcal{G}$, we must compute values for all instances of these predicates in that domain. The last four require non-trivial computation, including finding feasible κ values and computing the motion plans to obtain the necessary swept volumes. Ideally, we would find κ and associated trajectories so that V_{manip} and V_{pre} had a minimum number of collisions with obstacles in the world, which can be very costly in the general case [45].

We compute them only approximately, in two stages: first, we attempt to find a collision-free κ and trajectory. If that fails, then we simply find κ and a trajectory that are collision-free with respect to the fixed obstacles, but may collide with movable obstacles.

Evaluating each predicate is usually a very expensive operation, but there is a lot of information that can be retained across different calls, or across different iterations within the same call, to the predicate evaluation function. Therefore, we use caching extensively to make the repeated computation of the predicates efficient. Implementation details of this caching can be found in our original conference paper.

Sampling-based abstract-edge heuristic search algorithm The key distinction between G-TAMP problems and most graph search problems is that the feasibility of a transition is very expensive to evaluate. To

check whether an operation is feasible, we must first sample continuous values, and call an inverse kinematic solver and a motion planner to ensure the existence of of a robot configuration and a collision-free path for performing the operation.

So, instead of the traditional discrete state-based search, in which a state is expanded and its successors are added to the queue, we maintain a priority queue of state-and-abstract-action pairs, which we call *abstract edges*. We then use the *priority function*, denoted $\mathbf{p}(\alpha(s, \mathcal{G}), \mathfrak{o}(\delta))$, that ranks the abstract edges in the queue to determine which abstract edge to explore first.

As a non-learning version of SAPS, we propose the following heuristic function based on our abstract state representation,

$$H(s, \mathcal{G}, o, r) = |\mathrm{M}| - |O_{\text{achieved}}| + \mathbb{1}_{\textsc{InRegion}(o,r) \wedge (o,r) \in \mathcal{G}} \tag{2.2.1}$$

where M is a set of objects that need to be moved, O_{achieved} is a set of goal objects already in their specified goal regions, and the indicator function evaluates to 1 if the given object is a goal object already in its goal region, otherwise 0. Intuitively, this function has high values on states which have more objects to move, and discourages moving objects that have already been placed in its goal region.

Specifically, the set M is computed using following steps:

1. $M = \{o_G | (o_G, r_G) \in \mathcal{G} \wedge \neg \textsc{InRegion}(o_G, r_G)\}$,
2. $M = M \cup \{o | \exists o_m \in M, \textsc{OccludesPre}(o, o_m) = \text{True}\}$
3. $M = M \cup \{o | \exists o_m \in M, r \in \mathbf{R}, \textsc{OccludesManip}(o_m, r, r) = \text{True}\}$

Intuitively, the set is built recursively by first including goal objects not in their goal regions. Then, we add objects that obstruct the pre-manipulation robot configuration for the objects already in M, and then do the same for the objects that occlude the manipulation of objects in M.

Algorithm 1 defines SAPS, which is a greedy strategy with respect to the priority function. It takes in as an input the initial state, the set of goal states, the priority function, and the hyper-parameters for sampling continuous parameters of operators, N_{mp} and N_{smpl}.

The algorithm begins by creating a priority queue and adding abstract edges in the initial state to the queue. At each iteration, the algorithm selects the abstract edge with the highest priority, and attempts to construct a successor state by sampling feasible continuous parameters for the abstract action in the associated state using the function SmplCont.

Algorithm 1 SAPS($s_0, \mathcal{G}, N_{\text{smpl}}, N_{\text{mp}}, \mathbf{p}(\cdot,\cdot)$)

queue = *PriorityQueue*()
for $\delta \in \mathbf{O}^{(M)} \times \mathbf{R}, \mathfrak{o} \in \mathcal{O}$
    queue.add($(s_0, \mathfrak{o}(\delta)), \mathbf{p}(\alpha(s, \mathcal{G}), \mathfrak{o}(\delta))$)
end for
while not *time_limit_reached*
    $s, \mathfrak{o}(\delta)$ = *queue.pop*()
    κ = SMPLCONT($s, \mathfrak{o}(\delta), N_{\text{smpl}}, N_{\text{mp}}$)
    if κ is feasible
        $s' = T(s, \mathfrak{o}(\delta, \kappa))$
        $s'.path = s.path + \mathfrak{o}(\delta, \kappa)$
        if $s' \in \mathcal{G}$
            return $s'.path$
        end if
        for $\delta' \in \mathbf{O}^{(M)} \times \mathbf{R}, \mathfrak{o} \in \mathcal{O}$
            queue.add($(s', \mathfrak{o}(\delta')), \mathbf{p}(\alpha(s, \mathcal{G}), \mathfrak{o}(\delta))$)
        end for
    end if
    if *queue.empty*
        for $\delta \in \mathbf{O}^{(M)} \times \mathbf{R}, \mathfrak{o} \in \mathcal{O}$
            queue.add($(s_0, \mathfrak{o}(\delta)), \mathbf{p}(\alpha(s, \mathcal{G}), \mathfrak{o}(\delta))$)
        end for
    end if
end while

This function operates as follows. We first generate N_{mp} samples by checking whether the constraints are satisfied at the pre-manipulation and post-manipulation states without calling a motion planner to check the existence of a feasible motion. If we cannot sample a value within N_{mp} attempts, then we return. Otherwise, we call the motion planner to using the sampled values to see if there exists a feasible motion to any one of the sampled values. If there is, then we return that value.

If SMPLCONT returns feasible continuous parameters, then abstract edges based on the new state are added to the queue. If the next state is in the goal set, it returns the plan by retracing the path to the root. If it fails to sample feasible continuous parameters, it moves onto the next tuple on the queue. Unlike discrete graph search, our search space involves continuous values, so when the queue is empty we add the initial abstract edges back to the queue to maintain completeness.

2.3 *Related work*

There are several pure planning algorithms pure-planning algorithms for TAMP. Many approaches use define separate strategies for abstract and continuous actions. Typically, the task plans are determined using a task planner, often assuming that the low-level motion plans would be valid. Motion plans are then determined using optimization or sampling-based algorithms [37, 36, 124, 15, 118]. Then, they verify whether the abstract and continuous actions can together achieve the goal. If not, they try a different task plan, and the process repeats until the goal is found.

One of the difficulties of using learning algorithms in these planners is that at the task-level, they only have access to a "relaxed state" where the low level geometric details are not completely determined (for example, they may not know the poses of objects). This makes it difficult to predict the ranks of abstract actions, which requires occlusion and reachability information. Similarly, for continuous actions, it is difficult for the learned action sampler to suggest high-quality actions if the low-level geometric details are not completely determined. For these reasons, our algorithm, SAPS performs a search with complete states.

G-TAMP problem is very closely related to manipulation among movable obstacles [119]. G-TAMP significantly generalizes this class of problems by allowing to move multiple goal objects to goal regions, and lifting the assumption, in that particular method, that the robot must touch each object once. Rearrangement planning [76, 67] can also be considered as a subclass of G-TAMP problem, where we are given goal object poses for all objects instead of goal regions for some objects.

3
Learning to guide discrete search

If the geometric predicates are computed exactly, the heuristic function proposed in section 2.2.3 would be sufficient to find the object that needs to be moved at each state. However, because the predicates are *approximated* using a PRM and samples of continuous parameters, they often have errors. For instance, PREFREE(o) may evaluate to False when o is actually reachable. While we can hand-design how we might respond to such errors, this is difficult and tedious.

Therefore, we propose a learning algorithm that learns from planning experience a ranking function that can be used to augment the heuristic function to increase the efficiency of finding a plan. The fundamental challenge in designing such learning algorithm is representational: in guiding the decisions on which object to manipulate, the learning algorithm must reason about the reachability of objects, regions, and robot configurations. So, we must design a representation that (1) has sufficient information to infer reachability, and (2) achieves generalization across environments with different numbers and types of objects.

We observe that, for the purposes of guiding the search for abstract actions, it is not necessary to represent the state in complete geometric detail. Instead, we wish to capture the essential geometric aspects of a scenario but encode them abstractly in a way that affords generalization even across environments. More concretely, we use the set of geometric predicates proposed in Chapter 2 to represent a state.

Encoding such a relational state representation in a neural network, however, is not straight-forward. A simple feed-forward network cannot generalize to different numbers of objects. Our approach is to use graph neural networks (GNNs) which take as input a graph. We represent the relational state using a graph, where each node represents an object; unary predicates are stored at nodes and binary predicate at edges between nodes. GNNs are parameterized by a fixed set of weights, but can be applied to state representations with

any number of objects; *i.e.*, graphs with different numbers of nodes and edges. Using this representation and architecture, we learn a ranking function based on a large-margin objective [125].

The work presented in this chapter is based on our conference paper [1] which was a work done in collaboration with Luke Shimanuki.

[1] B. Kim and L. Shimanuki. Learning value functions with relational state representations for guiding task-and-motion planning. *Conference on Robot Learning*, 2019

3.1 *Related work*

Learning to guide planning The most famous system for learning to guide planning is AlphaZero, which was developed for the game of Go [112]. In AlphaZero, Monte Carlo tree search (MCTS) is integrated with a value function and policy that are learned from past instances of the game of Go. These learned predictors then guide MCTS into promising regions of the search space for the future instances of the game. Our framework can be seen as a version of AlphaZero for G-TAMP problems.

Given this observation, one may wonder whether we can simply use methods from AlphaGo and apply it to G-TAMP problems. There are few key differences between G-TAMP and Go that make such direct application non-trivial. First, Go has a discrete action space where each action is placing a stone on the board. In contrast, G-TAMP problems have a hybrid action space where each abstract action specifies which object to manipulate using which robot manipulation operator, and each continuous parameters specify which motion the robot should use to manipulate the chosen object. Further, generating the successor state for each action in the search requires an expensive feasibility check by calling a motion planner. In our framework, we resolve this challenge by learning a ranking function for abstract actions and samplers for continuous parameters. To minimize the number of feasibility checks, we propose a novel heuristic forward search algorithm that, instead of queuing the nodes of the graph, queues the abstract actions of the graph. This enables it to check feasibility of promising abstract actions first.

Another key difference is in the dimensionality of the state space. In the game of Go, you may use a board image as a state representation across different instances of the game. In contrast, states of G-TAMP problems consist of objects and their attributes resulting in state descriptions with different numbers of dimensions.

Our component for guiding the abstract action search is closely related to substantial body of work on learning to guide discrete planning. One line of work learns a heuristic function on states to be used in planners based on heuristic search. This is typically formulated as supervised learning from planning experience on related problems. Learning the heuristic function directly has proven chal-

lenging. Perhaps the most successful of these methods [134] learns domain-dependent corrections to an existing domain-independent heuristic. Many approaches learn how to best combine a variety of domain independent heuristics [32, 25]. Other approaches [38], like us, learn to rank actions directly, or learn a Q-value function for ranking actions [97].

Another approach that is similar to ours [18] also addresses TAMP problems by learning Q-values. However, their search is over plan refinements, rather than abstract actions and they provide no guidance on the choice of state-representation. [11] aim to learn a heuristic to guide greedy search in grid-based motion-planning problems. They also learn Q-values by imitating an oracle (a standard graph search) but they exploit the fact that they can easily get many training examples by solving all-sources shortest path problems. Also, a state of their search is specified by a set of user-defined features on the whole state of the search, including the search queues.

Graph Neural Networks For guiding the search for abstract actions, we use an abstract state representation based on geometric predicates. This naturally defines a graph, where each node encodes information about an entity in the scene, and each edge encodes relationship between entities. To handle this graph-based input, we use graph neural nets (GNNs). GNNs [115, 42, 106] (see surveys of [7, 135, 132]) incorporate a *relational inductive bias*: a set of entities and relations between them. In particular, we build on the framework of *message-passing neural networks* (MPNNs) [39], similar to *graph convolutional networks* (GCNs) [69, 7]. A key advantage of GNNs is that they learn a fixed-size set of parameters from problem instances with different numbers of entities. After learning, the GNN can be applied to arbitrarily large sets of objects and relations. This is crucial for G-TAMP problems where the number of objects varies widely.

3.2 *Learning the ranking function for abstract actions*

Our objective is to learn a ranking function that takes in as input the abstract state representation, and outputs ranking among the abstract actions. While we could, in principle, learn a action-value function for the abstract actions, it has been shown that learning a ranking function is sufficient to act optimally [88], and it is more data-efficient to do so [38]. We first begin with describing how we represent the ranking function using a graph neural network (GNN).

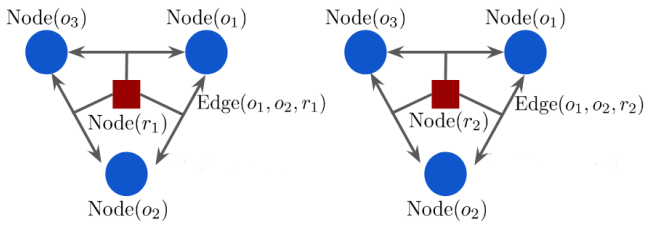


Figure 3.1: The graph encoding the geometric predicates of a scene with three objects o_1, o_2, and o_3 and two regions r_1 and r_2. We represent a scene with a disconnected graph where each component of the graph is a fully connected graph associated with a region. On the left is a component associated with region r_1 and right is the one with associated with r_2. Each Edge(o_i,o_j,r_k) encodes the binary predicates associated between two objects o_i and o_j and ternary predicates associated with the two objects and region r_k

3.2.1 *Representing an abstract rank function*

For each operator $\mathfrak{o}$, such as PickAndPlace, we define a GNN $\widehat{F}_{\mathfrak{o}}(\alpha(s, \mathcal{G}), \delta)$ that ranks discrete parameter choices for all $\delta \in \mathbf{O}^{(M)} \times \mathbf{R}$. We will refer to an element of $\mathbf{O}^{(M)} \cup \mathbf{R}$ as an *entity*, and reserve *object* and *region* to refer an element of $\mathbf{O}^{(M)}$ or $\mathbf{R}$ respectively. We begin by describing the input to the network, which is an encoding of the relational abstract state, $\alpha(s, \mathcal{G})$.

For each entity $e_i \in \mathbf{O}^{(M)} \cup \mathbf{R}$, we define x_{e_i} as a vector of unary predicate values

$$x_{e_i} = [\textsc{IsObject}(e_i), \textsc{IsRegion}(e_i), \textsc{IsGoal}(e_i), \textsc{PreFree}(e_i)]$$

For all ordered pairs of entities e_i, e_j, we define $x_{e_i e_j}$ as a vector of binary predicate values,

$$x_{e_i e_j} = \left[\textsc{InRegion}(e_i, e_j), \textsc{OccludesPre}(e_i, e_j), \textsc{ManipFree}(e_i, e_j)\right]$$

For all entities e_i, e_j and region r_k, $x_{e_i e_j r_k}$ is a single ternary relation value

$$x_{e_i e_j r_k} = \left[\textsc{OccludesManip}(e_i, e_j, r_k)\right]$$

The input to the ranking function is a graph representation of the abstract state. The graph is a disconnected graph with $|\mathbf{R}|$ components, each of which is a fully connected graph with $|\mathbf{O}^{(M)}| + 1$ number of nodes. We represent an entity with a node vector that encodes the unary predicates of the entity, which we denote with Node(e_i), where

$$\text{Node}(e_i) = x_{e_i}$$

At an edge between objects o_i and o_j on the component associated with region r_k, we have an edge vector Edge(o_i, o_j, r_k) that encodes binary and ternary predicates among o_i, o_j and r_k

$$\text{Edge}(o_i, o_j, r_k) = [x_{o_i}, x_{o_j}, x_{o_i o_j}, x_{o_j o_i}, x_{o_i r_k}, x_{o_j r_k}, x_{o_i o_j r_k} x_{o_j o_i r_k}]$$

An example of the graph representation of abstract state is shown in Figure 3.1.

Our ranking function takes this graph as an input, and outputs a $|\mathbf{O}^{(M)}|$ by $|\mathbf{R}|$ matrix where each value in the matrix indicates the rank of moving an object to a region.

To compute the rank among abstract actions, we perform two rounds of message passing. To compute the message from object o_i to object o_j, we first initialize the values at each node with

$$u_{o_i}^{(0)} = f(\text{Node}(o_i), \theta_1) \quad \text{and} \quad v_{o_i}^{(0)} = f(\text{Node}(o_i), \theta_2)$$

for all $o_i, o_j \in \mathbf{O}^{(M)}$ where the superscript denotes the round of message passing.

At each edge, we compute an edge embedding with

$$c_{o_i o_j r_k} = f(\text{Edge}(o_i, o_j, r_k); \theta_3) \ .$$

We compute the message from o_i to o_j for region r_k, $m_{o_i o_j r_k}$, as

$$m_{o_i o_j r_k}^{(0)} = f(u_{o_i}^{(0)}, v_{o_j}^{(0)}, c_{o_i o_j r_k}; \theta_4)$$

We aggregate these messages using averaging,

$$m_{o_j}^{(0)} = \frac{1}{|\mathbf{O}^{(M)}| + |\mathbf{R}|} \sum_{o_i, r_k} m_{o_i o_j r_k}^{(0)}$$

We perform one more round of message passing. The values at nodes are computed again using $f(\cdot; \theta_1)$ and $f(\cdot; \theta_2)$ as

$$u_{o_j}^{(1)} = f(m_{o_j}^{(0)}; \theta_1) \quad \text{and} \quad v_{o_j}^{(1)} = f(m_{o_j}^{(0)}; \theta_2)$$

The new messages are computed with the updated node values and the edge embedding

$$m_{o_i o_j r_k}^{(1)} = f(u_{o_j r_k}^{(1)}, v_{o_j r_k}^{(1)}, c_{o_i o_j r_k}, \theta_4).$$

We then aggregate the messages using averaging, only with respect to objects this time:

$$m_{o_j r_k}^{(1)} = \frac{1}{|\mathbf{O}^{(M)}|} \sum_{o_i} m_{o_i o_j r_k}^{(0)}.$$

Finally, we compute the ranking matrix, where each entry is

$$\widehat{F}_{\mathfrak{o}}(\alpha(s, \mathcal{G}), (o_j, r_k); \theta) = f(m_{o_j, r_k}^{(1)}; \theta_5),$$

where $\theta = \{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$.

3.2.2 *Learning a rank function from planning experience*

We now describe the training data and loss function for training the ranking function. Because we know the transition model for the domain, we can find plans by solving past problem instances using an existing planning algorithm.

Each such plan is a sequence of state-action pairs

$$[(s_0, \mathfrak{o}_0(\delta_0, \kappa_0)), \ldots, (s_{T-1}, \mathfrak{o}_{T-1}(\delta_{T-1}, \kappa_{T-1})), (s_T, \emptyset)]$$

in which $s_T \in \mathcal{G}$, where $\mathcal{G}$ is the goal for which that plan was made. From this, we can construct $T + 1$ supervised training examples of the form $(\alpha(s_t, \mathcal{G}), \mathfrak{o}_t(\delta_t))$. Aggregating this data from multiple start-goal pairs, and partitioning it according to the abstract action types $\mathfrak{o}$, we end up with a data set $\mathcal{D}_\mathfrak{o}$ for each $\mathfrak{o}$, with entries of the form $(\alpha(s_t, \mathcal{G}), \delta)$.

We wish to rank the abstract actions such that the actions that appeared in past plans have higher ranks than those that have not appeared. This is implemented using the following large-margin loss:

$$\mathcal{L}_{LM}(\theta) = \sum_{(s,\mathcal{G},\delta,d)\in\mathcal{D}_\mathfrak{o}} \max(0, 1 - M_Q(\alpha(s, \mathcal{G}), \delta; \theta)),$$

where

$$\begin{aligned} &M_Q(\alpha(s, \mathcal{G}), \delta; \theta) \\ &\quad = \widehat{F}_\mathfrak{o}(\alpha(s, \mathcal{G}), \delta; \theta) - \max_{\delta' \in \Delta \setminus \{\delta\}} \widehat{F}_\mathfrak{o}(\alpha(s, \mathcal{G}), \delta'; \theta). \end{aligned}$$

Intuitively, the margin term $M_Q(\alpha(s, \mathcal{G}), \delta; \theta)$ is penalizing the ranking function until the value of δ that appeared in the dataset in the given state is higher than the values of all the other abstract actions by 1. If this is satisfied, the margin term evaluates to zero.

To guide the search algorithm SAPS, we use the priority function, which combines the learned ranking and the heuristic function given in Eqn 2.2.1,

$$\begin{aligned} \mathbf{p}(\alpha(s, \mathcal{G}), \mathfrak{o}(\delta)) = &- H(\alpha(s, \mathcal{G})) \\ &+ \lambda \cdot \frac{exp\left(\widehat{F}_\mathfrak{o}(\alpha(s, \mathcal{G}), \delta; \theta)\right)}{exp\left(\sum_{\mathfrak{o}', \delta'} \widehat{F}_{\mathfrak{o}'}(\alpha(s, \mathcal{G}), \delta; \theta)\right)}. \end{aligned} \tag{3.2.1}$$

We use this priority function $\mathbf{p}$ as an input to SAPS, as shown in Algorithm 1. The reason we integrate with the hand-designed heuristic function, H, is that H is a good estimate of cost-to-go from the current state, but by definition it does tell which actions are better within the state. Therefore, we use a ranking function to prioritize actions within a state; to make sure that ranking does not override the heuristic function, we make sure that the rank of an action is between 0 and 1, using the Boltzmann function.

4
Learning to guide continuous search

In Chapter 3, we presented a framework for guiding the discrete search in G-TAMP problems. In this chapter, we describe the representation and learning algorithms for guiding the continuous search in G-TAMP problems.

We first observe that these continuous parameters often specify, implicitly or explicitly, a goal for a low-level motion planner, such as a grasp and base pose for picking up an object. So, it is important to predict a continuous action that induces a feasible motion plan, because infeasible motion planning calls are extremely expensive. One naive method would be to encode object poses and shapes into a vector, but that would not generalize across different numbers of objects. Moreover, using, such workspace occupancy information it is very difficult to determine reachability. Instead, what we need is the collision information in the configuration space (c-space) of the robot.

Therefore, we introduce a novel state representation called *key configuration obstacles* that encodes collision information at essential robot configurations. This enables the learned predictor to reason about the feasibility of continuous actions because key configuration obstacles approximate the true c-space obstacles in the current scene.

Using this representation, one simple method for learning to predict continuous parameters given a state is to use regression to learn a mapping from a state to an action. However, this method has several problems. First, it outputs an average of the actions taken in a state, which often leads to undesirable results. Second, it only predicts a single value, which means that the planner cannot query for other values if the first value leads the search astray.

To deal with these issues, we can learn a generative model for conditional distribution over actions given a state using Kernel Density Estimation (KDE). This approach can handle multi-modality in a state, and can generate multiple values. However, this method cannot scale to high-dimensional states because it relies on a kernel, such as squared exponential function, which does not perform well in high-

dimensional spaces. Also, the computational complexity of inference in KDE increases linearly with data, which is undesirable especially because we wish to speed up planning.

We instead use generative adversarial nets (GANs), which can handle high-dimensional inputs using neural networks and whose inference time is independent of a number of training data. Moreover, GANs learn a data-dependent objective function that can handle multi-modality, rather than using predefined objective function such as MSE that leads to undesirable behavior such as averaging.

One problem with a standard GAN is that it is data-inefficient for learning from planning experience. In a successful episode of search, there is a large number of state and action pairs that are considered but were not on a trajectory to the goal in the episode, which we call *neutral data*, and only a relatively small number of samples that were on a trajectory to the goal, which we call *positive data*. While we could just use the small number of positive data, learning would be much more data-efficient if we could use the abundant neutral data as well.

To use both types of data, we extend the basic GAN algorithm under two different setups. The first setup is where we have positive and neutral labels on the trajectories encountered during search. We introduce a new generative model learning algorithm, called Generative Adversarial Network with Direct Importance estimation (GANDI) [61], that first estimates the importance-ratio between neutral and positive data distributions. Using the estimated ratio, GANDI can use both types of data but train the sampler to approximate the positive data distribution. We theoretically analyze how the importance-ratio estimation and the difference between positive and neutral data distributions affect the quality of the resulting approximate distribution.

The second setup we consider is when we have access to rewards along the trajectory encountered during search. We propose an actor-critic method that learns a sampler by simultaneously maximizing the sum of rewards and imitating the planning experience [62]. Our algorithm, Adversarial Monte-Carlo (ADMON), penalizes actions if they are too different from the planning experience, while learning the Q-function of the sequences using past search trees. As a consequence of integrating critic into GAN training, we can emphasize imitating actions that have higher values.

4.1 Related work

Two basic methods for in generative-model learning, GANs [41] and Variational Auto Encoders (VAEs) [68], are appealing choices for the purpose of learning action samplers because an inference is simply a

feed-forward pass through a network. GANs are especially appealing, because for generic action spaces, we do not have any metric information available. VAEs require a Euclidean metric on the action space in order to compute the distance between a decoded sample and a true sample. For our purpose, which involves predicting poses of objects and the robot base, this is problematic because a small deviation in parameter values can mean the difference between in feasibility and infeasibility. For this reason, we use GANs to represent our sampler.

A major drawback of the original GAN [41], which minimizes Shannon-divergence, is that its instability during training. WGANs [3], which minimizes the Earth-mover's distance, have shown to be more stable, but led to difficulty in optimization due to hard gradient clipping. We use WGAN with Gradient Penalty (WGAN-GP) [44], which improves the training of WGAN by using a soft-enforced constraint on the gradient of the discriminator.

There is a long history of work that uses importance sampling to approximate desired statistics for a target distribution p using samples from another distribution q in various problems [71, 101, 122]. In these cases, we have a surrogate distribution q that is cheaper to sample than the target distribution p. Our work shares the same motivation as these problems, in the sense that in search experience data, samples that are on successful trajectories are expensive to obtain, while neutral samples are relatively cheaper and more abundant.

Recently, importance-ratio estimation has been studied for the problem of covariate shift adaptation, which closely resembles our setting. Covariate shift refers to the situation where we have samples from a training distribution that is different from the target distribution. Usually, an importance-ratio estimation method [54, 121, 51] is employed to re-weight the samples from the training distribution, so that it matches the target distribution in expectation for supervised learning. We list some prominent approaches here. In kernel-based methods for estimating the importance [51], authors try to match the mean of samples of q and p in a feature space, by re-weighting the samples from q. In the direct estimation approach, Sugiyama et al. [121] try to minimize the KL divergence between the distributions q and p by re-weighting q. In another approach, Kanamori et al. [54] directly minimize the least squares objective between the approximate importance-ratios and the target importance-ratios. All these formulations yield convex optimization problems, where the decision variables are parameters of a linear function that computes the importance weight for a given random variable. Our algorithm extends the direct estimation approach using a deep neural network, and then applies it for learning an action sampler using both neutral and

positive data.

ADMON can be seen as a variant of an actor-critic algorithm that uses extra data from past planning experience in addition to reward signals. In a standard actor-critic algorithm [72], a value function is first trained that evaluates the current policy, and a policy is trained by maximizing this value function. Actor-critic algorithms have traditionally been applied to problems with discrete action spaces, but recently they have been successfully extended to continuous action space problems. For example, in Proximal Policy Optimization (PPO) [109], an off-policy actor-critic method, a value function is trained with Monte-Carlo roll-outs from the current policy. Then, the policy is updated based on an advantage function computed using this value function, with a clipping operator that prohibits a large changes between iterations. Deep Deterministic Policy Gradient (DDPG) [83] is another actor-critic algorithm that extends deep Q-learning to continuous action space by using a deterministic policy gradient. These methods have been applied to learning low-level control tasks such as locomotion, and rely solely on the given reward signal. Our method uses both past search trees and reward signals, and is applied to the problem of learning a high-level operator policy that maps a state to continuous parameters of the operators.

A number of other algorithms also use data from another source besides rewards to inform policy search. Guided policy search (GPS) [81] treats data obtained from trajectory optimization as demonstrations, trains a policy via supervised learning, and enforces a constraint that makes the policy visit similar states as those seen in the trajectory optimization data. The key difference from our work is that we use search trees as guiding samples - in a search tree, not all actions are optimal, so we cannot simply use supervised learning. We instead propose an objective that simultaneously maximizes the sum of rewards while softly imitating the trajectories in the search tree. Another important difference is that trajectory optimization requires smooth reward functions while the problems of interest to us have discontinuous reward functions. In approximate policy iteration from demonstrations (APID) [59], suboptimal demonstrations are provided in addition to reward signals, and the discrete action-space problems are solved using an objective that mixes large-margin supervised loss and policy evaluation. ADMON can be seen as an extension of this work, where the suboptimal demonstrations are provided by the past search trees, to continuous action spaces.

4.2 State representation using key configurations

The continuous actions predicted by a learned sampler should, with high probability, satisfy the feasibility constraints and achieve the goal. Typically, these continuous actions implicitly represent the goal for the low-level motion planner, such as a base pose for picking an object. So, for our biased sampler to predict promising values for the continuous parameters, we need a representation of the scene that enables a neural network to infer the existence of a collision-free motion.

One naive approach is to use the poses and shapes of the movable objects. However, since this approach only gives us the obstacle information in the workspace, it is insufficient to reason about collision-free constraints, especially for objects and robots with complex shapes. Moreover, this representation cannot generalize across different numbers and types of objects. What we need instead is the obstacle information in the configuration space of the robot, but this is extremely expensive to construct exactly.

Our approach is to approximate the collision information at essential regions of the configuration space, using a set of *key configurations*. Key configurations are a set of configurations that we construct by first collecting from our planning experience a large set of robot configurations that were used in the planning solutions, and then sparsely sub-sampling from that set using a threshold on the distances among them. Given a scene, we represent the state using the collision information at these key configurations. Our insight is that we do not need to completely construct the configuration space obstacles, but only represent them for essential regions of the c-space that the robot is likely to re-use in the future problem instances.

Another essential type of information that we need to capture is the set of swept-volumes for manipulating goal objects to their associated goal regions, which we call *goal-swept-volumes*. If there are no obstacles in collision with these goal-swept-volumes, then we would have to move just the goal objects to goal regions, without manipulating any other objects. So, it is important information that we should encode.

To do this, we consider $V\text{manip}, \mathfrak{o}(o_G, r_G)$ for goal objects and their associated goal regions that we computed for evaluating geometric predicates in an abstract state. While this may not be the actual swept-volume for moving the goal object to the goal region, it represents an approximation of it.

We then encode the collision and goal-swept-volume information using a binary vector ϕ. Formally, ϕ is a binary vector of shape n_k by 2, where n_k is the number of key configurations. The first column of

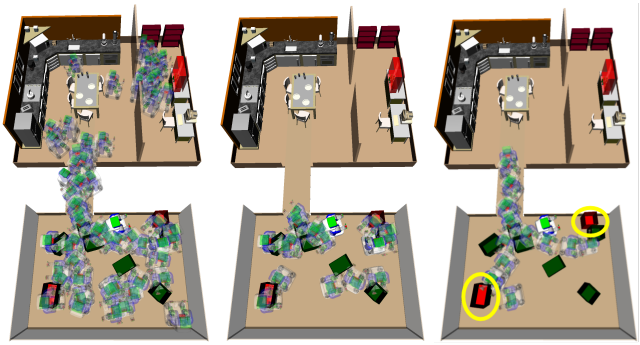


Figure 4.1: Left: a subset of key configurations used for this domain. Middle: key configurations that are in collision. Right: key-configuration-representation of goal-swept-volumes for the two goal red objects marked yellow circles. The state is r epresented by ϕ, whose first column encodes collision information shown in the middle figure, and the second column encodes goal-swept-volumes shown on the right-most figure

ϕ encodes collision information and the second column encodes goal swept volumes. For each key configuration, we check its collision. If the key configuration is in collision, then the corresponding row and the first column of ϕ is assigned with a value of 1, otherwise to 0. Likewise, for goal-swept-volumes, We go through each configuration in the swept volume, find the key configuration that is closest to it, and assign a value of 1. For all the key configurations that have not been selected, their values are set to 0.

Figure 4.1 shows an example of a state representation based on key configurations that encode both the key configuration collisions and goal-swept-volumes.

4.3 *Learning a biased sampler from planning experience*

We have so far described a state representation that approximates the c-space obstacles. We now describe a set of learning algorithms for learning a sampler. We begin with a problem formulation.

Suppose that we are given a planning experience dataset $\mathbf{D}_{\text{pl}} = \{\tau^{(i)}\}_{i=1}^{L}$, where each *operator sequence*, τ, from a search tree is a tuple $\{s_t, \mathfrak{o}_t(\delta_t, \kappa_t), y_t\}_{t=1}^{T}$ where $\mathfrak{o}_t \in \mathcal{O}$ and y_t indicates whether $(s_t, \mathfrak{o}_t(\delta_t, \kappa_t))$ is a positive or neutral data. We assume that the positive actions in state s follow an unknown conditional distribution $p_{K|S=s}$.

Give this dataset, our objective is to learn an *operator sampler* associated with each operator that maps a state to the continuous parameters of the operator, $\{\pi_{\theta_i}\}_{i=1}^{m}$, where θ_i is the parameters of the sampler for the operator $\mathfrak{o}^{(i)}$, $\pi_{\theta_i} : \mathcal{S} \rightarrow K_{\mathfrak{o}^{(i)}}$, such that $\pi_{\theta_i}(s)$ is similar to $p_{K|S=s}$.

4.3.1 *Wasserstein GANs with gradient penalty*

We first present a sampler learning algorithm that can only use the positive data from our planning experience that consists of trajecto-

ries that got to the goal. We will denote to the subset of out planning experience data that only contains positive data as $\mathbf{D}_{\text{pl}}^{(p)} \subseteq \mathbf{D}_{\text{pl}}$.

We use Wasserstein GANs (WGANs) [3], which has proven to be more stable training behavior than GANs. Our objective function, in the case of infinite samples, is

$$\max_{D \in ||D||_L \leq 1} \mathbb{E}_{s \sim P_S} \left[\mathbb{E}_{\kappa \sim p_{K|S=s}} \left[D(s, \kappa) \right] - \mathbb{E}_{\kappa \sim \pi_\theta} \left[D(s, \kappa) \right] \right].$$

where π_θ represents our learned distribution, P_S represents the distribution over states in our dataset, and $p_{K|S}$ represents the conditional distribution over continuous action given a state that we wish to learn, and $||D||_L \leq 1$ indicates the set of all 1-Lipschitz functions $D : \mathcal{S} \times K \to \mathbb{R}$.

Function D is commonly referred to as a *discriminator* that discriminate the samples of $p_{K|S}$ from the samples of π_θ. We represent the function class $||D||_L \leq 1$ with neural networks parameterized by α, which we denote D_α. To enforce the 1-Lipschitz constraint on these neural networks, the original WGAN used hard gradient clipping which leads to difficulty in optimization. Instead, we use WGAN with Gradient Penalty (WGAN-GP) [44]. This method uses soft-constraint the norm of the gradient of the function D_α based on the observation that a differential function is 1-Lipschitz if and only if its gradients have norms of at most 1 everywhere. Our objective for training a discriminator is

$$\begin{aligned} &\min_\alpha \mathbb{E}_{s \sim P_S} \Big[\mathbb{E}_{\kappa \sim p_{K|S=s}} \left[D_\alpha(s, \kappa) \right] + \mathbb{E}_{\kappa \sim \pi_\theta(K|s)} \left[D_\alpha(s, \kappa) \right] + \\ &\lambda \cdot \mathbb{E}_{\hat{\kappa} \sim p_{\hat{K}}} \left[\left(||\nabla_{\hat{\kappa}} D_\alpha(s, \hat{\kappa})||_2 - 1 \right)^2 \right] \Big] \end{aligned} \tag{4.3.1}$$

Here, the last term is responsible for approximately enforcing Lipschitz constraint. Since enforcing the constraint everywhere is intractable, WGAN-GP enforces it only on the samples from $p_{\hat{K}}$, where $p_{\hat{K}}$ is defined as a uniform distribution on a straight line between a pair of samples from $p_{K|S}$ and π_θ.

To train our sampler, we use the discriminator as our optimization criterion. Its objective is defined as

$$\max_\theta \mathbb{E}_{\kappa \sim \pi_\theta(K|s)} \left[D_\alpha(s, \kappa) \right]. \tag{4.3.2}$$

In practice, these two neural networks are trained in an alternating fashion. The pseudo-code is shown in Algorithm 2.

The algorithm takes as inputs the training dataset $\mathbf{D}_{\text{pl}}^{(p)}$, gradient penalty scale term λ, total number of iterations, n_{tot}, number of gradient steps for discriminator training at each iteration, n_c, the batch size n_b, and learning rates for the sampler and discriminator, lr_θ and

Algorithm 2 WGAN-GP($\mathbf{D}_{\text{pl}}^{(p)}, \lambda, n_{tot}, n_c, n_b, lr_\theta, lr_\alpha$)

for $t = 0$ **to** n_{tot}
  for $t_c = 0$ **to** n_c
    for $t_m = 0$ **to** n_b
      $\left(s^{(i)}, \kappa^{(i)}\right) \sim \mathbf{D}_{\text{pl}}^{(p)}$
      $z^{(i)} \sim P_Z(z)$, $\kappa_\theta^{(i)} \sim \pi_\theta(s^{(i)}, z^{(i)})$
      $\epsilon^{(i)} \sim U[0,1], \hat{\kappa}^{(i)} = \epsilon\kappa^{(i)} + (1-\epsilon)\hat{\kappa}^{(i)}$
      $L^{(i)} = D_\alpha(s^{(i)}, \kappa^{(i)}) - D_\alpha(s^{(i)}, \kappa_\theta^{(i)}) + \lambda \left(||\nabla_{\hat{\kappa}^{(i)}} D(s^{(i)}, \hat{\kappa}^{(i)})||_2 - 1\right)^2$
    end for
    $\alpha = \alpha + \text{Adam}(lr_\alpha, \nabla_\alpha \frac{1}{n_b} \sum_{i=1}^{n_b} L^{(i)})$
  end for
  $\{z^{(i)}\}_{i=1}^{n_b} \sim P_Z(z)$
  $\theta = \theta + \text{Adam}(lr_\theta, \nabla_\theta \frac{1}{n_b} \sum_{i=1}^{n_b} f(s^{(i)}, \pi(s^{(i)}, z^{(i)})))$
end for
return π_θ

lr_α, respectively. It begins by training the discriminator. At each iteration of discriminator training, it creates a batch of $\hat{\kappa}$ values, by sampling a point from $\mathbf{D}_{\text{pl}}^{(p)}$, and generating a point from π_θ. It then samples a random number uniformly between 0 and 1 and uses this as a weight to mix the point from $\mathbf{D}_{\text{pl}}^{(p)}$ and point from π_θ. These are used to compute our objective function for each point in our batch, $L^{(i)}$. Once all of these values are computed, then we take a gradient step with respect to α using the Adam optimizer. We repeat these steps n_c times and then update the parameters of sampler, θ. The entire process is repeated n_{tot} number of times.

One of the fundamental challenges of GANs is evaluating quality of trained models. We can use it to guide the planner select the best model that achieves highest improvement in planning efficiency, but this is expensive. So, we use Kernel Density Estimation to evaluate the quality of trained models. To do this, for each state in our dataset, we generate 100 samples and then fit it with Kernel Density Estimation (KDE). We then measure the likelihood of the continuous parameters for that state using KDE. We average the likelihood across all states in the dataset, and discard the trained weights and restart training if the average likelihood value are too low.

4.3.2 *Generative adversarial network with direct importance estimation (*GANDI*)*

The downside of Algorithm 2 is it can only use the positive data, whereas planning experience consists of positive and neutral datasets. We now describe an adversarial training algorithm called Generative

adversarial network with direct importance estimation (GANDI) that uses both types of data.

The way it works is by estimating the importance-ratio between the neutral and positive data distributions using the least squares approach proposed in [54], and then use the estimated ratios to train the model to imitate the positive data distribution $p_{K|S}$ while using positive and neutral data. More concretely, we denote the distribution over neutral data as $q_{K|S}$ and the samples from $q_{K|S}$ as κ_q, samples from $p_{K|S}$ as κ_p, and the number of samples we have from positive and neutral data distributions as n_p and n_q respectively. The goal in importance-ratio estimation is to estimate the ratio $w(\kappa; s) = p_{K|S}(\kappa|s)/q_{K|S}(\kappa|s)$ using only samples from distributions $p_{K|S}$ and $q_{K|S}$. We henceforth omit the conditioning on states to avoid clutter unless it is ambiguous.

We use the least squares approach proposed in [54]. In this method, the importance-ratio is approximated by minimizing

$$J(\widehat{w}) = \int_{\kappa} (\widehat{w}(\kappa) - w(\kappa))^2 q(\kappa) d\kappa \,.$$

In practice, we optimize its sample approximation version, which gives

$$\hat{w} = \arg\min_{\widehat{w}} \sum_{i=1}^{n_q} \widehat{w}^2(\kappa_q^{(i)}) - 2 \sum_{i=1}^{n_p} \widehat{w}(\kappa_p^{(i)}), \text{ s.t } \widehat{w}(\kappa) \geq 0 \tag{4.3.3}$$

The method was originally proposed to be used with a linear architecture, in which $\widehat{w}(\kappa) = \theta^T \phi(\kappa)$; this implies there is a unique global optimum as a function of θ, but requires a hand-designed feature representation $\phi(\cdot)$. For robot planning problems, however, manually designing features is difficult, while special types of DNN architectures, such as convolutional neural networks, may effectively learn a good representation. In light of this, we represent $\widehat{w}$ with a DNN. The downside of this strategy is the loss of convexity with respect to the free parameters θ, but we have found that the flexibility of representation offsets this problem.

We now introduce our algorithm, GANDI, which can use neutral samples from q_K using importance-ratios. We first describe how to formulate the objective for training GANs with importance weights. For the purpose of exposition, we begin by assuming we are given $w(\kappa; s)$, the true importance-ratio between q_K and p_K, for all κ_q distributed according to q_K, and we only have neutral samples, and none from the positive data distribution. We denote the neutral samples as $\mathbf{D}_{\text{pl}}^{(q)} \subseteq \mathbf{D}_{\text{pl}}$. By the definition of importance weights $w(\kappa)$, the

WGAN-objective for the discriminator, Eqn. (4.3.1), becomes

$$\min_{D \in ||D||_L \leq 1} \mathbb{E}_{\kappa_q \sim q_K} \left[w(\kappa_q) D_\alpha(\kappa_q) \right] + \mathbb{E}_{\kappa_\theta \sim \pi_\theta} \left[D_\alpha(\kappa_\theta) \right] + \lambda \cdot \mathbb{E}_{\hat{\kappa} \sim q_{\hat{K}}} \left[(||\nabla_{\hat{\kappa}} D_\alpha(\hat{\kappa})||_2 - 1)^2 \right] \tag{4.3.4}$$

Notice that this objective is now with respect to q_K instead of p_K. The distribution $q_{\hat{K}}$ is analogous to $p_{\hat{K}}$, but we create samples $\hat{\kappa}$ using q_K and π_θ.

In trying to solve the equation (4.3.4), it is critical to have balanced training set sizes n_q and n_g in order to prevent the class imbalance problem for D_α. In the importance weighted version of the GAN shown in equation 4.3.4, the sum of the weights $c = \sum_{i=1}^{n_q} w(\kappa_q^{(i)})$, serves as an *effective sample size* for the data $\mathbf{D}_{\text{pl}}^{(q)}$. To achieve a balance the number of samples from each class, we might then select n_g to be equal to some constant c. Taking this approach, however, would require adjusting the GAN objective function and making sure that in every batch, the number of generated samples is equal to the sum of the importance weights of the real samples in that batch, which is tedious.

Instead, we develop a method for bootstrapping $\mathbf{D}_{\text{pl}}^{(q)}$ that allows us to use existing mini-batch gradient descent without modification. Specifically, instead of multiplying each neutral sample by its importance weight, we bootstrap (i.e. re-sample the data from $\mathbf{D}_{\text{pl}}^{(q)}$ with replacement), with probability $p_w(\kappa)$, where $p_w(\kappa) = \frac{w(\kappa)}{\sum_{i=1}^{n_q} w(\kappa_q^{(i)})}$. This method allows us to generate a dataset $\widehat{\mathbf{D}}_{\text{pl}}$ in which the expected number of instances of each data in $\mathbf{D}_{\text{pl}}^{(q)}$ is proportional to its importance weight. Moreover, since we bootstrap, the amount of training data remains the same, and discriminator D_α now sees a balanced number of samples effectively drawn from $p_K(\kappa) = w(\kappa) q_K(\kappa)$ and π_θ. One can also show that p_w is actually proportional to p.

Proposition 1. *For* $\kappa \in \mathbf{D}_{pl}^{(q)}$,

$$p_w(\kappa) = k \cdot p(\kappa) \text{ where } k = \frac{1}{\sum_{i=1}^{n_q} w(\kappa_q^{(i)})}.$$

We now describe some practical details for approximating $w(\kappa)$ with $\widehat{w}(\kappa)$, whose architecture is a DNN. Equation 4.3.3 can be solved by a mini-batch gradient-descent algorithm implemented using any readily available NN software package. The non-negativity constraint can also be straight-forwardly incorporated by simply using the rectified linear activation function at the output layer. In practice, this often lead gradients to shrink to 0 due to saturation. Although this can be avoided with a careful initialization method, we found

that it is effective to just use linear activation functions, and then set $w(\kappa) = 0$ if $w(\kappa) < 0$.

Now, with estimated importance weights and bootstrapped samples, the sample-approximation of objective for the discriminator shown in equation 4.3.4 is

$$\hat{D}_\alpha = \arg\min_D \sum_{i=1}^{n_q} D(\kappa_w^{(i)}) + \sum_{i=1}^{n_g} \log(1 - D(\kappa_g^{(i)})) + \lambda \cdot \mathbb{E}_{\hat{\kappa} \sim q_{\hat{K}}} \sum_{i=1}^{n_q} \left[\left(||\nabla_{\hat{\kappa}} D(\hat{\kappa}^{(i)})||_2 - 1 \right)^2 \right] \tag{4.3.5}$$

where $\kappa_w^{(i)}$ denotes a bootstrapped sample from $\mathbf{D}_{\text{pl}}$, and $n_q = n_g$. Algorithm 3 contains the code for GANDI.

We illustrate the result of the bootstrapping with a simple example, shown in Figure 4.2, where we have a Gaussian mixture model for both positive and neutral data distributions p and q, where p is a mixture of two Gaussians centered at $(1,1)$ and $(3,1)$, and q is a mixture of three Gaussians at $(1,1)$, $(3,1)$, and $(2,2)$ with larger variances than those of p.

Figure 4.2 (a) shows the true distributions of positive and neutral data distributions, p and q denoted with blue and red, respectively. Figure 4.2 (b) shows training data points from each of these distributions, $\mathbf{D}_{\text{pl}}^{(p)}$ and $\mathbf{D}_{\text{pl}}^{(q)}$. Figure 4.2 (c) shows the estimated importance weights, $\widehat{w}$, using objective (4.3.3), where darker color indicates higher $\widehat{w}(a)$ value. We can see that $\widehat{w}$ is almost zero in regions where $\mathbf{D}_{\text{pl}}^{(p)}$ and $\mathbf{D}_{\text{pl}}^{(q)}$ do not overlap, especially around (2,2). Figure 4.2 (d) shows our bootstrapped samples, $\widehat{\mathbf{D}}_{\text{pl}}$, sampled from our bootstrap probability distribution p_w in green, and $\mathbf{D}_{\text{pl}}^{(p)}$ again in red. We can see that it reflects the values of $\widehat{w}$. Lastly, the right-most plot shows the result of training GANDI using $\widehat{\mathbf{D}}_{\text{pl}}$, from which we can see it is quite similar to p.

Algorithm 3 GANDI($\mathbf{D}_{\text{pl}}^{(p)}, \mathbf{D}_{\text{pl}}^{(q)}$)

$\widehat{w} \leftarrow$ EstimateImportanceWeights($\mathbf{D}_{\text{pl}}^{(p)}, \mathbf{D}_{\text{pl}}^{(q)}$) // obj. (4.3.3)

$p_w(\kappa) := \frac{\widehat{w}(\kappa)}{\sum_{i=1}^{n_q} \widehat{w}(\kappa_q^{(i)}) + \sum_{i=1}^{n_p} \widehat{w}(\kappa_p^{(i)})}$ // bootstrap p.m.f

$\mathbf{D}_{\text{pl}} \leftarrow \mathbf{D}_{\text{pl}}^{(p)} \cup \mathbf{D}_{\text{pl}}^{(q)}$

$\widehat{\mathbf{D}}_{\text{pl}} \leftarrow$ Bootstrap($\mathbf{D}_{\text{pl}}, p_w$) // sample $\mathbf{D}_{\text{pl}} \sim p_w$ with replacement

$\theta \leftarrow$ TrainWGAN($\widehat{\mathbf{D}}_{\text{pl}}$) // Algorithm 2 with $\widehat{\mathbf{D}}_{\text{pl}}$ as data

return θ

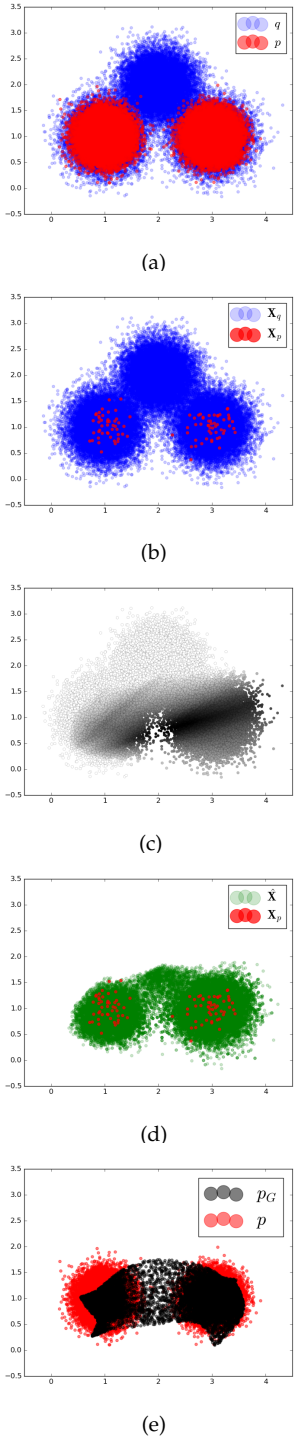


Figure 4.2: (a) positive and neutral target distributions, (b) training data points $\kappa_p^{(i)}$ and $\kappa_q^{(i)}$, (c) importance weight estimation, (d) bootstrapping result and (e) learned distribution π_θ and target distribution p.

Theoretical analysis

In this section, we analyze how the error in importance estimation affects the performance of π_θ in approximating p. The basic result on GANs, shown in the limit of infinite data, representational and computational capacity, is that π_θ converges to p [41]. The proofs for the theorems presented in this section can be found in our conference paper [61].

Now, under the same assumptions, we consider the effect of using importance weighted off-target data. If w is exact, then $p(\kappa) = w(\kappa)q(\kappa)$ and the GAN objective is unchanged. If, however, we use an estimation of importance weighting function $\widehat{w}$, then the objective of $\hat{D}_\alpha$, the importance-weight corrected discriminator, differs from D_α and they achieve different solutions.

We wish to analyze the effect of importance estimation error on KL and reverse-KL divergence between p and π_θ. First, define $\rho = \sup_{\kappa \in \mathcal{K}_p} q(\kappa)/p(\kappa)$, where $\mathcal{K}_p$ is the support of p. We can see that $\rho >= 1$, with equality occurring when $p(\kappa) = q(\kappa)$ for all κ.

For the KL divergence, we have the following theorem.

Theorem 1. *If* $w(\kappa) \geq \epsilon \; \forall \kappa \in \mathcal{K}_q$, $\epsilon \geq 0$, *and* $J(\widehat{w}) \leq \epsilon^2$, *then*

$$\mathrm{KL}(p||\pi_\theta) \leq \log\left(\frac{1}{1-\epsilon\rho}\right) \; .$$

Note that $0 \leq \epsilon\rho \leq 1$ due to the condition $w(\kappa) \geq \epsilon$. For reverse KL we have:

Theorem 2. *If* $J(\widehat{w}) \leq \epsilon^2$, $KL(\pi_\theta||p) \leq (1+\epsilon)\log(1+\epsilon\rho)$.

These theorems imply three things: (1) If $w = \widehat{w}$, then $\epsilon = 0$, and both divergences go to 0, regardless of ρ; (2) If $p = q$, then the error in importance weight estimation is the only source of error in modeling p with π_θ. This error can be arbitrarily large, as ϵ becomes large; and (3) If $p \neq q$ then $\rho > 1$, and it contributes to the error in modeling p with π_θ.

4.3.3 *Adversarial Monte Carlo (*ADMON*)*

We now consider an alternative setup where we are given rewards along the trajectories that have been explored in an episode of search. This allow us to make use of neutral samples by augmenting the adversarial training with an actor-critic objective.

In this setup, we are given a planning experience dataset $\mathbf{D}_{\mathrm{pl}} = \{\tau^{(i)}\}_{i=1}^{L}$, where each operator sequence, τ, from a search tree is a tuple $\{s_t, \mathfrak{o}_t(\delta_t, \kappa_t), r_t, s_{t+1}\}_{t=1}^{T}$, where we have reward r_t for using $\mathfrak{o}_t(\delta_t, \kappa_t)$ in state s_t instead of label y_t. Our objective is to learn an

operator sampler associated with each operator that maps a state to the continuous parameters of the operator, $\{\pi_{\theta_i}\}_{i=1}^{m}$, where θ_i is the parameters of the sampler for the operator $\mathfrak{o}^{(i)}$, $\pi_{\theta_i} : \mathcal{S} \to K_{\mathfrak{o}^{(i)}}$, that maximizes the expected sum of the rewards

$$\max_{\theta_1,\cdots,\theta_m} \mathbb{E}_{s_0 \sim P_0}\Big[\sum_{t=0}^{H} r(s_t, \kappa_t)\Big|\theta_1, \cdots, \theta_m\Big]$$

where $r(s_t, \kappa_t) = r(s_t, \mathfrak{o}_t(\delta_t, \kappa_t))$, and P_0 is the initial state distribution. Given a problem instance, we assume a task-level planner has given the operators and discrete parameters, and our goal is to predict the continuous parameters.

One way to formulate an objective for imitating the planning experience dataset $\mathbf{D}_{\text{pl}}$ is by using the adversarial training scheme [41], where we learn a discriminator function $\hat{Q}_\alpha$ that assigns high values to operator instances from $\mathbf{D}_{\text{pl}}$ and low-values to operator instances generated by the sampler. We will, for the purpose of exposition, consider a single operator setting to avoid the notational clutter. We have

$$\max_{\alpha} \sum_{s_i,\kappa_i \in \mathbf{D}_{\text{pl}}} \hat{Q}_\alpha(s_i, \kappa_i) - \hat{Q}_\alpha(s_i, \pi_\theta(s_i)) \tag{4.3.6}$$

$$\max_{\theta} \sum_{s_i,\kappa_i \in \mathbf{D}_{\text{pl}}} \hat{Q}_\alpha(s_i, \pi_\theta(s_i)) \tag{4.3.7}$$

These two objectives are optimized in an alternating fashion to train the sampler that imitates the operator sequences in $\mathbf{D}_{\text{pl}}$. This can be seen as an application of Wasserstein-GAN [3] to an imitation learning problem.

The trouble with this approach is that not all sequences in the search trees are equally desirable: we would like to generate operator instances that yield high values. So, we propose the following regularized sampler evaluation objective that learns the value function from sequences in the search trees, but simultaneously penalizes the sampler in an adversarial manner, in order to imitate the planner's operator sequences. We have

$$\min_{\alpha} \sum_{s_i,\kappa_i \in \mathbf{D}_{\text{pl}}} ||Q(s_i, \kappa_i) - \hat{Q}_\alpha(s_i, \kappa_i)||^2 + \lambda \cdot [\hat{Q}_\alpha(s_i, \pi_\theta(s_i))] \tag{4.3.8}$$

where $Q(s_i, \kappa_i) = r(s_i, \kappa_i) + \sum_{t=i+1}^{T} r(s_t, k_t)$ is the sum of the rewards of operator sequences in the search tree. We treat this as we would a value obtained from a Monte-Carlo rollout, and λ is used to trade off adversarial regularization versus accuracy in value-function estimation.

The pseudocode for our algorithm, Adversarial Monte-Carlo (ADMON), is given in Algorithm 4. The algorithm takes as inputs the

Algorithm 4 ADMON($\mathbf{D}_{\text{pl}}, \lambda, T_S, lr_\alpha, lr_\theta, n$)

for $t_s = 0$ **to** T_s
  // Train Q-value
  Sample $\{s_i, \kappa_i\}_{i=1}^n \sim \mathbf{D}_{\text{pl}}$ // a batch of data
  $dq = \nabla_\alpha \sum_{i=1}^n \left[(Q_i - \hat{Q}_\alpha(s_i, \kappa_i))^2 + \lambda \hat{Q}_\alpha(s_i, \pi_\theta(s_i))\right]$
  $\alpha = \alpha - \text{Adam}(lr_\alpha, dq)$
  // Train sampler
  $dp = \nabla_\theta \left[\sum_{i=1}^n \hat{Q}_\alpha(s_i, \pi_\theta(s_i))\right]$
  $\theta = \theta + \text{Adam}(lr_\theta, dp)$
  $J_{t_s} = \textit{Evaluate}(\pi_\theta)$
end for
return $\hat{Q}_\alpha, \pi_\theta$ **with** $\max J_0, \cdots, J_{T_s}$

planning experience dataset $\mathbf{D}_{\text{pl}}$, the parameter for ADMON, λ, the number of iterations, T_s, the learning rates for the Q-function, lr_α, and the sampler, lr_θ, , and batch size n. It then takes a batch gradient descent step with the parameters of the Q-function, α, and then takes a batch gradient step with those of the sampler, θ.

Adversarial training is known to have stability problems in its typical application of generating images, since evaluating image-generation policies is not simple. This is not an issue in our case. In ADMON, after each update of the sampler parameters, we evaluate its performance using the *Evaluate* function, which executes the given sampler for a fixed number of time steps and returns the sum of the rewards. We then return the best performing sampler.

4.3.4 Cleaning the training dataset

Since the planning algorithm that we use to generate training data is sampling-based, the continuous parameter data tends to noisy, we often get non-optimal state-continuous-parameter pairs. For example, the robot might place an object at an unhelpful pose only to move it again later. If we were to use this dataset directly, then we would end up with a sampler that is very similar to a uniform sampler, because there is not enough useful regularity to capture.

To deal with this problem, we use a strategy for cleaning the dataset such that each training pair makes a progress towards a goal. The idea is to check whether the object that we move at each step decreases the number of objects in collision with goal-swept-volumes.

For each tuple $(s_t, \delta_t, \kappa_t, s_{t+1})$ from $\mathbf{D}_K$, δ_t is the object that the robot moves at time step t, κ_t is the continuous parameter that moves δ_t, and s_{t+1} is the resulting state.

To determine whether to include this training example, first recall that each training example is from a problem instance for a goal $\mathcal{G}$. So for each $o_g \in \mathcal{G}$, we check if δ_t is in collision with

$V_{\text{pre}}(o_g) \cup V_{\text{manip}}(o_g, r_g)$ at s_t. We add the number of times that δ_t was in collision, and denote the summation as m_t. The maximum m_t would be number of goal objects in $\mathcal{G}$. We repeat these steps to count the collisions in in state s_{t+1}, and compute m_{t+1}. We include κ_t in $\mathbf{D}_K$ only if $m_{t+1} - m_t > 0$. Intuitively, this method makes sure that each example we include in our data moves an object out of the goal-swept-volumes.

5
Experiments in geometric task-and-motion planning problems

We now evaluate our planning and guidance algorithms developed in Chapters 3 and 4 by applying it to challenging G-TAMP problems. We first present the results on guiding the continuous search, and then describe the results on using guidance on both discrete and continuous search.

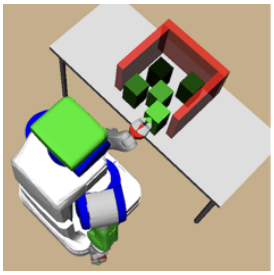
5.1 Results in guiding continuous search

For this section, we assume that the abstract actions are given by an oracle, and the robot just has to find the continuous decision variables that satisfy the feasibility constraints and achieve the goal. As a planner, we use Algorithm 1 but assume we know the correct abstract actions to take.

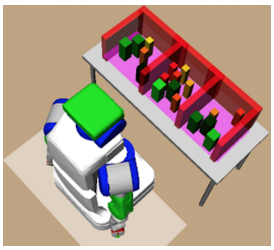
5.1.1 Results using GANDI

We validate GANDI on three different robot planning tasks that involve continuous state and action spaces and finite depths. These experiments have two purposes: first, to verify the hypothesis that learning an action sampler improves planning efficiency relative to a standard uniform sampler and second, GANDI is more data efficient than a standard GAN that is only trained with positive data.

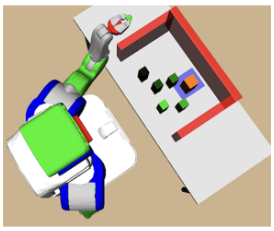
We have three tasks that might occur in a warehouse. The first is a bin-packing task, shown in Figure 5.1a, where a robot has to plan to pack different numbers of objects of different sizes into a relatively tight bin. The second task is planning to stow eight objects into crowded bins, shown in Figure 5.1b where there already are obstacles. The final task is a reconfiguration task, shown in Figure 5.1c where a robot needs to reconfigure five randomly placed moveable objects in order to make room to reach a target object.



(a)



(b)



(c)

Figure 5.1: (a) Bin packing. The color of objects indicate the order of placement. The darker the earlier. (b) Stowing task. The goal is to put all green objects into the crowded bins. (c) Reconfiguration task. The movable obstacles are colored green, and the target object is colored with orange. For all tasks, the robot can only grasp objects from the side.

These are highly intricate problems that require long-term geometric reasoning about object placements and robot configurations; for each object placement, we require that there is a collision-free path and an inverse kinematics (IK) solution to place the object. For the placement of an object, we verify an existence of the IK solution by solving for IK solutions for a set of predefined grasps, and then check for the existence of a collision-free path from the initial state to the IK solution using a linear path. A more sophisticated motion planners, such as RRT, can be used instead but we found linear path to be sufficient in these tasks.

In the bin packing and stowing tasks, the robot is not allowed to move the objects once they are placed, which leads to a large volume of dead-end states that cause wasted computational effort for a planner with a uniform action sampler. In the reconfiguration task, we have no dead-end states, but a planner could potentially waste computational effort in trying no-progress actions that does not clear a volume to reach the target object. For all tasks, the robot is only allowed to grasp objects from the side; this is to simulate a common scenario in a warehouse environment, with objects in a place covered on top, such as a shelf. For the first two experiments, we use a heuristic that estimates the cost-to-go to be the number of objects remaining to be placed, since we cannot move objects once they are placed. For the last experiment we use breadth-first-search with no heuristic. For all cases, instead of discarding the node when an infeasible action is sampled, we try to sample 4 feasible actions.

In each task, we compare three different action sampler in terms of success rate within a given time limit: one that uniformly samples an action from the action space, a standard GAN trained only with positive samples, and GANDI, which is trained with both neutral and positive samples. We use the same architecture for both the standard GAN and GANDI, and perform 100 repetitions to obtain the 95% confidence intervals for all the plots.

A crucial drawback of generative adversarial networks is that they lack an evaluation metric; thus it is difficult to know when to stop training. We deal with this by testing weights from all epochs on 10 trials, and then picking the weights with the best performance, with which we performed 100 additional repetitions to compute the success rates.

5.1.2 Bin packing problem

In this task, a robot has to move 5, 6, 7 or 8 objects into a region of size 0.3m by 1m. The number of objects is chosen uniform randomly. The size of each object is uniformly random between 0.05m to 0.11m,

depending on how many objects the robot has to pack. A problem instance is defined by the number of objects and the size of each object, $\omega = [n_{obj}, O_{size}]$. A state is defined by the object placements. For a given problem instance, all objects have the same size. An example of a solved problem instance with $n_{obj} = 5$ and $O_{size} = 0.11$m is given in Figure 5.1a.

The action space consists of the two dimensional (x,y) locations of objects inside the bin, and a uniform action sampler uniformly samples these values from this region. The robot base is fixed. The planning depth varies from 5 to 8, depending on how many objects need to be placed. This means that plans consist of 10 to 16 decision variables.

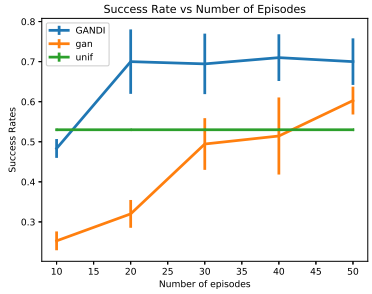
Figure 5.2a plots, for each method, the success rate when given 5.0 seconds to solve a problem instance. We can see the data efficiency of GANDI: with 20 training episodes, it outperforms the uniform sampler, while a standard GAN requires 50 training episodes to do so. The uniform sampler can only solve about 50% of the problem instances within this time limit, while GANDI can solve more than 70%.

We also compare the action samplers trained using GAN and GANDI when the same number of training data are given. Figures 5.3a and 5.3b show 1000 samples from GAN and GANDI for packing 5 objects. While GANDI learns to avoid the front-middle locations, GAN is still close to a uniform action sampler, and has a lot of samples in this region which lead to dead-end states. GANDI focuses its samples on the corners at the back or the front so that it has spaces for all 5 objects.

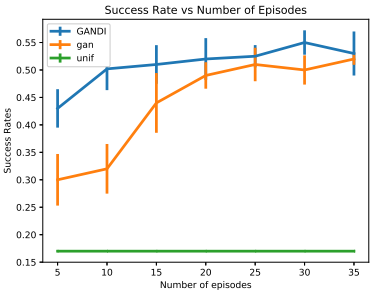
5.1.3 *Stowing objects into crowded bins*

In this task, a robot has to stow 8 objects into three different bins, where there already are 10 obstacles. A bin is of size 0.4m by 0.3m, an obstacle is of size 0.05m by 0.05m, and the objects to be placed down are all of size 0.07m by 0.07m. A problem instance is defined by the (x,y) locations of 10 obstacles, each of which is randomly distributed in one of the bins. Figure 5.1b shows an instance of a solved stow problem.

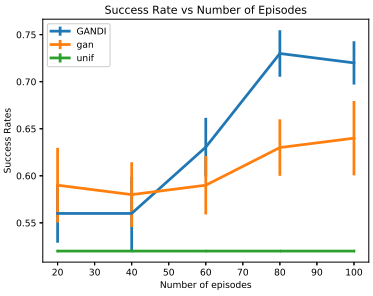
The action space for this problem consists of (x,y) locations of an object to be placed, and the robot's (x,y) base pose. This makes a 4 dimensional continuous action-space. The planning depth is always 8, for placing 8 objects. Thus plans consist of 36 continuous decision variables. Again, there is a large volume of dead-end actions, similarly to the previous problem: putting objects down without consideration of poses of later objects can potentially block collision-free



(a)



(b)



(c)

Figure 5.2: Plots of success rate vs. number of training episodes for the (a) bin packing, (b) stow, and (c) reconfiguration domains

paths for placing them.

Figure 5.2b compares the success rates of the algorithms with a 30-seconds time limit for planning. For the uniform sampler, we sample first an object placement pose, and then sample a base pose that can reach the object at its new location without colliding with other objects. Unlike the previous task, learning-based approaches significantly outperform the uniform sampling approach for this task. This is because there is a small volume of action space that will lead to a goal, and a large volume of search space. Again, we can observe the data efficiency of GANDI compared to GAN. When the number of training data points is small, it outperforms it.

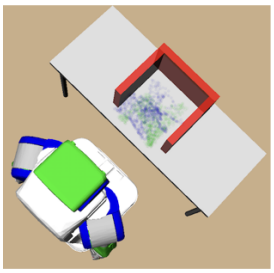
5.1.4 *Reconfiguration planning in a tight space*

In this task, a robot has to reconfigure movable obstacles out of the way in order to find a collision-free IK solution for its left-arm to reach the target object. There are five movable obstacles in this problem, each with size 0.05m by 0.05m, and the target object of size 0.07m by 0.07m, and the reconfiguration must happen within a bin, which is of size 0.7m by 0.4m. A problem instance is defined by (x,y) locations of the movable obstacles and the target object. The movable obstacles are randomly distributed within the bin; the target object location is distributed along the back of the bin. Figure 5.3c shows an example of a rearrangement problem instance at its initial state, with the black region indicating the distribution of target object locations.

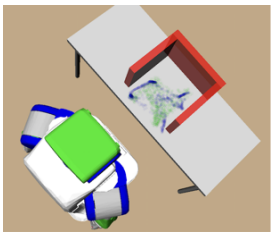
An action specification consists of one discrete value and two continuous values: what object to move and the (x,y) placement pose of the object being moved. There is no fixed depth. For both the uniform random sampler and the learned sampler, we uniformly at random choose an object to move. The robot base is fixed, and the robot is not allowed to use its right arm.

Figure 5.2c compares the success rates of the algorithms with a 10-seconds time limit for planning. In this problem, the learning-based approaches outperform the uniform sampler even with a small number of training data points. The relationship between GANDI and GAN is similar to the previous experiment, except that GANDI and GAN are within the each other's confidence interval when a small number of training points are used. Eventually, GANDI comes to clearly outperform GAN.

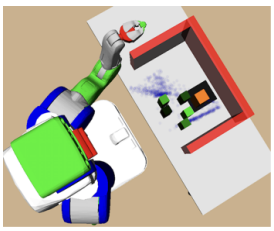
We would like to know if GANDI's distribution indeed assigns low probabilities to no-progress actions. In Figure 5.3c, we show GANDI's distribution of object placements after training on 35 episodes. The left top corner of the bin is empty because there are no collision-free IK solutions for that region[1]. As the figure shows, there are no



(a)



(b)



(c)

Figure 5.3: Figures (a) and (b) respectively show actions sampled from action samplers trained with GAN and GANDI from the bin packing domain when 20 episodes of training data are used. Green indicates the positive samples, and blue indicates the learned samplers. Figure (c) shows an action distribution for the reconfiguration domain when given 35 training episodes.

[1] The robot's left arm will collide with the bin

placement samples in front of the target object, but only on the sides that would contribute to clearing space for the robot's left arm to reach the target object.

5.1.5 *Results using* ADMON

We evaluate ADMON in two practical and challenging G-TAMP problems, and compare against three benchmarks: PPO [109], DDPG [83], which are actor-critic algorithms for continuous action spaces, and Generative Adversarial Imitation Learning (GAIL) [48], a state-of-the-art inverse reinforcement learning algorithm that treats the planning experience dataset as optimal demonstrations, and uses PPO to find a policy that maximizes the learned rewards. For DDPG, we use an episodic variant that defers updates of the policy and replay buffer to the end of each episode, which makes it perform better in our inherently episodic domain. It is important to keep some level of stochasticity in any policy we learn, because there is a large volume of infeasible operator instances for which no transition occurs. So, we use a Gaussian policy with a fixed variance of 0.25, and use the learned policies to predict only the mean of the Gaussian.

Our hypotheses are that (a) ADMON, by using the planning experience dataset $\mathbf{D}_{\text{pl}}$, can learn more data efficiently than the benchmarks, and (b) learning these policies can improve planning efficiency. To test the first hypothesis, we show two plots. The first is the learning curve as a function of the size of $\mathbf{D}_{\text{pl}}$, with a fixed number of interactions with the simulated environment for the RL methods. For the RL methods, $\mathbf{D}_{\text{pl}}$ is used as an initial training set. For ADMON, simulations are only used for the evaluation of the current policy. For this plot, we fix the amount of RL experience at 30000 for the conveyor belt domain, and 15000 for the object fetching domain; these are obtained from 300 updates of the policy and value functions of each algorithm, where for each update, we do 5 roll-outs, each of which is 20 steps long for the first domain and 10 steps long for the second domain. We report the performance of the best policy from these 300 updates, averaged over four different random seeds. Second is the learning curve with increasing amount of simulated RL experience, with fixed $\mathbf{D}_{\text{pl}}$ size. We fix its size at 100 for the first domain and 90 for the second domain. For testing hypothesis (b), we show the improvement in planning efficiency when we use the best policy out of all the ones used to generate the first two plots to guide a planner.

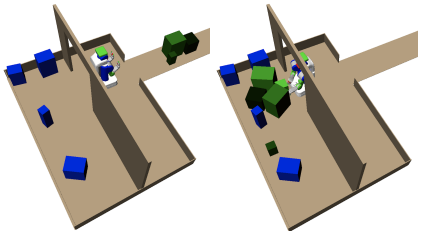
Domain overview Our objective is to test the generalization capability of the learned policy across the changes in the poses and shapes of

different number of objects in the environment, while the shape of the environment stays the same. We have two test domains. In the first conveyor belt domain shown in Figure 5.4a, the robot's objective is to receive either four or five box-shaped objects with various sizes from a conveyor belt and pack them into a room with a narrow entrance, already containing some immovable obstacles. A problem instance is defined by the number of objects in the room, their shapes and poses, and the order of the objects that arrive on the conveyor belt. The robot must make a plan for handling all the boxes, including a grasp for each box, a placement for it in the room, and all of the robot trajectories. The initial base configuration is always fixed at the conveyor belt. After deciding the object placement, which determines the robot base configuration, a call to an RRT motion planner is made to find, if possible, a collision-free path from its fixed initial configuration at the conveyor belt to the selected placement configuration. The robot cannot move an object once it has placed it. This is a difficult problem that involves trying a large number of infeasible motion planning problems, especially if poor sampling is used to sample continuous operator parameters.

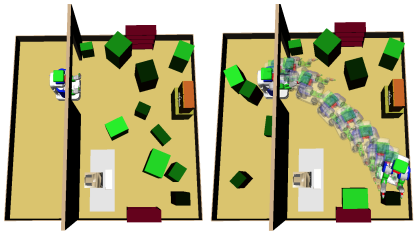
In the second object-fetch domain shown in Figure 5.4b, the robot must move to fetch a target object and bring it back to the robot's initial location. The target object is randomly placed in the bottom half of the room, which is packed with both movable and immovable obstacles. A problem instance is defined by the poses and shapes of the target and movable obstacles. The robot's initial configuration is always fixed, but it needs to plan paths from various configurations as it picks and places objects. To reach the target object, the robot must pick and place movable obstacles along the way, each of which involves a call to the motion planner in a relatively tight environment. We have set the problem instance distribution so that the robot must move five to eight objects to create a collision-free path to fetch the target object.

In both domains, if the selected operator instance is infeasible due to collision or kinematics, the state does not change. Otherwise, the robot picks or places the selected object with the given parameters. The reward function for the conveyor belt domain is 1 if we successfully place an object into the room and 0 otherwise. For the fetching domain, the reward function is 0 if we successfully pick an object, -1 if we try an infeasible pick or place, and 1 if we successfully move an object out of the way. For moving all the objects out of the way, the robot receives a reward of 10.

Operator description The robot is given two manipulation operators are given to the planner, *pick* and *place*, each of which uses two arms



(a) Conveyor belt domain: green objects must be packed in room.



(b) Object-fetch domain: objects must be removed from path to target object.

Figure 5.4: Examples of initial (left column) and goal states (right column) in two domains.

Operators	Continuous parameters	Inputs to π_θ (conv belt)	Inputs to π_θ (obj fetch)
Pick	(x_r^o, y_r^o, ψ_r^o) (d, h, χ)	-	ϕ_{fetch}, ϕ $(x_o, y_o, \psi_o), (l, w, h)$
Place	(x, y, ψ)	ϕ	$\phi_{fetch}, \phi, (x_o, y_o, \psi_o)$

Table 5.1: Operator descriptions. (x, y, ψ) refers to a robot base pose, at (x,y) location and rotation ψ in the global frame, (x_r^o, y_r^o, ψ_r^o) refers to the relative robot base pose with respect to the pose of an object o, whose pose in global frame is (x_o, y_o, ψ_o). (d, h, χ) is a grasp represented by a depth, as a portion of size of object in the pushing direction, height, as a portion of object height, and angle in the pushing direction, respectively, and (l, w, h) represents the length, width, and height of object being picked. ϕ is key configuration obstacles, and ϕ_{fetch} is a fetching path represented with key configurations.

to grasp a large object. Table 5.1 summarizes the parameters of each operator.

To provide guidance, we use the policies trained with different learning algorithms. The inputs to the policy are described in the table as well. Each operator will generally require a call to a motion planner to find a collision-free path from the previous configuration.

The oracle that gives task-level plan is implemented as follows. For both domains, the task-plan is the sequence of objects to be pick-and-placed. For the conveyor domain, this is given by the problem instance definition: the objects arrive in the order to be packed. For the fetching domain, we implement a swept-volume approach similar to [24, 119]. We first plan a fetching motion to the target object assuming there are no movable obstacles in the scene. Then, we check the collision between this path and the moveable obstacles to identify objects that need to be moved.

To sample parameters for the *pick* operation using the default uniform policy:

1. Sample a collision-free base configuration, (x_r^o, y_r^o, ψ_r^o), uniformly from a circular region of free configuration space, with radius equal to the length of the robot's arm, centered at the location of the object.

2. With the base configuration fixed at (x_r^o, y_r^o, ψ_r^o) from the object, sample (d, h, χ), where d and z has a range $[0.5, 1]$, and χ has a range $[\frac{\pi}{4}, \pi]$, uniformly. If an inverse kinematics (IK) solution exists for both arms for this grasp, proceed to step 3, otherwise restart.

3. Use bidirectional RRT (biRRT), or any motion planner, to get a path from the current robot base configuration to (x_r^o, y_r^o, ψ_r^o) from the pose of the object. A linear path from the current arm configuration to the IK solution found in step 2 is then planned.

If a collision is detected at any stage, the procedure restarts. When we use the learned policy π_θ, we simply draw a sample from it, and then check for IK solution and path existence with the predicted grasp and base pose.

For the conveyor belt domain, we assume that the conveyor belt drops objects into the same pose, and the robot can always reach

them from its initial configuration near the conveyor belt, so we do not check for reachability. For the object fetch domain, we do all three steps.

From a state in which the robot is holding an object, it can place it at a feasible location in a particular region. To sample parameters for *place* using the default uniform policy:

1. Sample a collision-free base configuration, (x, y, ψ), uniformly from a desired region.
2. Use biRRT from the current robot base configuration to (x, y, ψ).

To use π_θ, we sample base configurations from it in step 1.

For heuristic function for the continuous-space graph search in the fetching domain, we use the number of objects to be moved out of the way as a heuristic. For the conveyor belt domain, we use the remaining number of objects to be packed as a heuristic.

To collect a dataset $\mathbf{D}_{\text{pl}}$, we use search trees constructed while solving previous planning problems. To create a operator sequence τ from a search tree, we begin at the root and collect state, action, and rewards up to each leaf node.

Results for the conveyor belt domain Figure 5.5 (top) shows the learning curve as we increase the number of search trees. Each search tree from a problem instance adds at most 50 (state, reward, operator instance, next state) tuples. The RL algorithms, DDPG and PPO, have rather flat learning curves. This is because they treat the planning experience dataset $\mathbf{D}_{\text{pl}}$ as just another set of roll-outs; even with 100 episodes of planning experience, this is only about 5000 transitions. Typically, these methods require tens of thousands of data to work well. ADMON, on the other hand, makes special use of $\mathbf{D}_{\text{pl}}$ by trying to imitate the sequences. On the other hand, the results from GAIL show that it is ineffective to treat $\mathbf{D}_{\text{pl}}$ as optimal demonstrations and simply do imitation. ADMON, which uses reward signals to learn a Q-function in addition to imitating $\mathbf{D}_{\text{pl}}$, does better.

Figure 5.5 (middle) shows the learning curves as we increase the amount of transition data, while fixing the number of search trees at 100. Again, ADMON outperforms the RL algorithms. Note that the RL approaches, DDPG and PPO, are inefficient in their use of the highly-rewarding $\mathbf{D}_{\text{pl}}$ dataset. For instance, PPO, being an on-policy algorithm, discards $\mathbf{D}_{\text{pl}}$ after an update. Even though the transition data collected after that is much less informative, since it consists mostly of zero-reward transitions, it makes an update based solely on them. As a result, it tends to fall into bad local optima, and the learning curve saturates around 3000 steps. The situation is similar for the off-policy algorithm DDPG. It initially only has $\mathbf{D}_{\text{pl}}$ in its replay

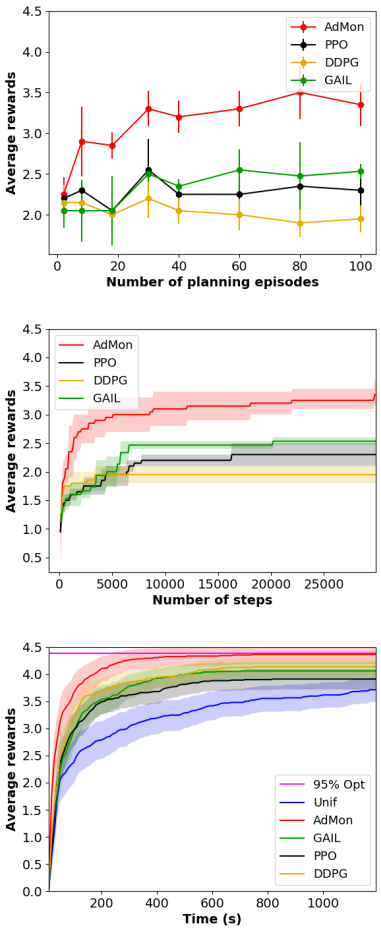


Figure 5.5: Conveyor belt plots

buffer, but as it collects more data it fills the buffer with zero-reward sequences, slowing the learning significantly after around 5000 steps. ADMON, on the other hand, is able to better exploit the planning experience dataset to end up at a better (local) optima. GAIL shows slightly better performance than PPO and DDPG. It tends to escape bad local optima by learning a reward function that assigns high rewards to the planning experience dataset $\mathbf{D}_{\text{pl}}$, but still performs worse than ADMON because it treats $\mathbf{D}_{\text{pl}}$ as optimal demonstrations.

Figure 5.5 (bottom) shows the reduction in planning time achieved by different learning algorithms. We can see that, after about 400 seconds, ADMON achieves 95% optimal performance, whereas the uniform policy still have not achieved that performance after 1200 seconds, indicating a speed up of at least 3.

Results for object fetch domain Figure 5.6 (left) shows the learning curve as we increase the number of search trees. Each search tree adds at most 50 (state, operator instance, next state) tuples, up to 25 of which use pick operator instances, and the remaining are place operator instances. Again, the RL methods show weaker performance than ADMON although this time they are closer. The poor performance of GAIL is due to $\mathbf{D}_{\text{pl}}$ containing many more off-target operator sequences than before, due the longer horizon. Since most of these sequences are not similar to the solution sequence, treating these data points as optimal demonstrations hurts the learning.

Figure 5.6 (middle) shows the learning curve as we increase the number of transitions, while fixing the number of search trees at 90. PPO shows large variance with respect to different random seeds, and on average, shows a very steep learning curve at the beginning, but it gets stuck at a bad local optima. DDPG shows good performance, but still performs worse than ADMON. GAIL fails to learn anything meaningful due to the previously stated reasons.

Figure 5.6 (right) shows the impact on the planning efficiency when trained policies are used to choose the continuous parameters. This time, we plot the progress, measured by the number of objects cleared from the fetching path for different time limits. We can see that ADMON clears the optimal number of objects at around 1500 seconds, and the uniform policy takes 3500 seconds, an improvement in planning efficiency by a factor of more than 2.3.

DDPG initially performs just as well as ADMON until it clears 3 objects, but its improvement stops after this point. This is due to the current-policy-roll-out exploration strategy used by DDPG. It is very unlikely with this strategy to encounter an episode where it clears more than 3 objects. When used with the planner, which uses a heuristic, the policy starts encountering states that have more

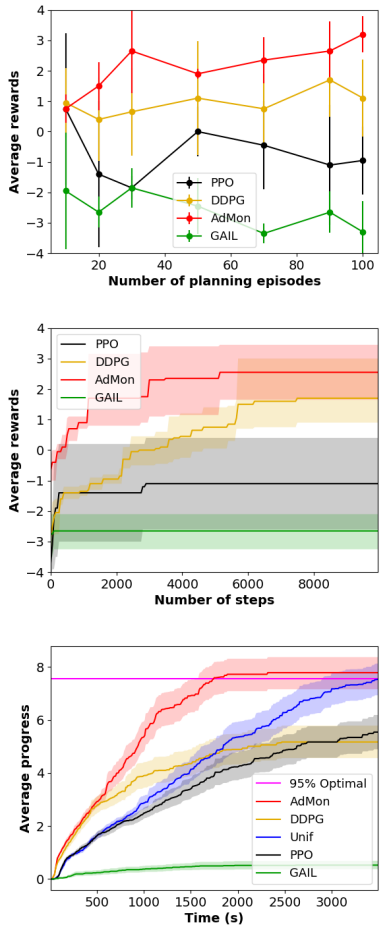


Figure 5.6: Plots for object fetch domain

than three objects cleared, leading to poor performance. This phenomenon, where there is a discrepancy between the distribution of states encountered during training and testing, is also noted in [105]. ADMON on the other hand does not have this problem because it is trained with the search trees produced by the planner. Note also the decrease in planning efficiency when using poor policies.

5.2 Results on combined guidance on discrete and continuous search

Previous sections empirically demonstrated that ADMON and GANDI can guide the search for continuous search in G-TAMP to improve planning efficiency by several factors compared to unguided planners, and does so using far less data than state-of-the-art learning algorithms. This was possible because ADMON and GANDI uses both neutral and positive datasets. We now present results on guiding the discrete search by using the learned ranking function for abstract actions. We then integrate learned sampler, ranking function, and planning algorithm, SAPS, in a single framework, and present the its results.

We consider two different environments. One is the box-moving domain shown in shown in Figure 2.1 (left) where the objective is to move a set of boxes from their current locations to the kitchen region. The second environment is the cupboard domain where the robot has to move a target object from the cupboard to the packing box shown in Figure 2.1 (right). A problem instance is defined by the poses of movable objects, where the poses of objects are randomly chosen from a distribution.

To make sure that each problem instance is non-trivial (i.e. cannot be solved by simply moving just the goal objects to goal regions), we define a distribution over pose such that the robot must manipulate at least 2 objects. In the box-moving domain, we do this by randomly placing at least 3 objects at the exit and around the robot. Similarly in the cupboard domain, we randomly place the goal object at the back of the cupboard to ensure the robot must rearrange at least 2 objects. Some example problems are shown in Figure 5.7.

We again focus on the pick-and-place operator in this section. For the box-moving domain, we have two-arm pick-and-place; for the cupboard environment, we have a one-arm pick-and-place. Both of these domains use the same sampling scheme as we have described in the section 5.1.5, except that for the one-arm case the parameters specify the midpoint of the right arm's gripper fingers.

The inputs to the learned samplers are the key configuration obstacles and the pick-and-place motion for moving the goal objects into the goal region, expressed using key configurations. The abstract

ranking function takes in the graph representation of the scene as described in section 3.2.1 based on geometric predicates.

Given a grasp and base pose for pick, an inverse kinematics solver is used to generate (collision-free) arm configurations for picking at the specified base pose and grasp parameters. In the box-moving environment, once we determine the pick-and-place continuous parameters, we plan the base-motion plans for both picking and placing the objects. In the cupboard environment, we omit motion planning and simply check collisions at the pick and place configurations, which is sufficient in this domain. We use Rapidly exploring Randomized Trees as our motion planner, and IKFast [23] as our inverse kinematics solver.

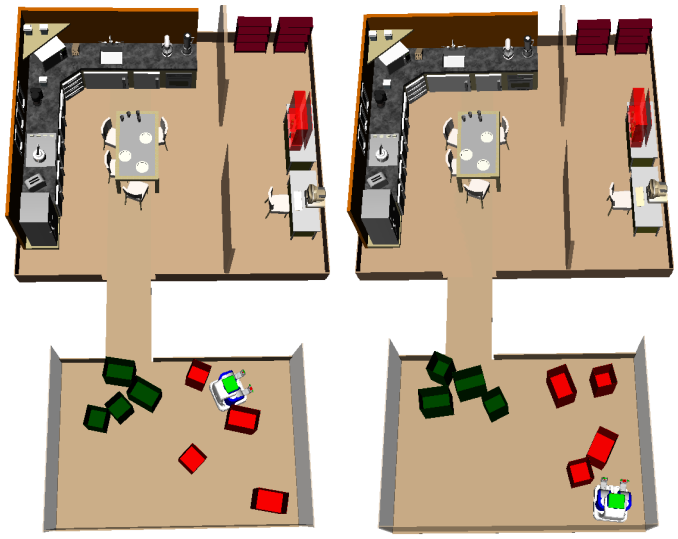


Figure 5.7: Examples of problems in the box-moving domain, where the robot has to move all the red boxes to the kitchen region at the top. Note that there are objects blocking the exit, and some objects are surrounding the robot.

The central claim in this work is that learning to guide a planner is more computationally efficient and reliable than using pure learning or pure planning algorithms. To support this claim, we compare against the following benchmarks:

- PureLearning: A method that simply uses the abstract action that has the highest rank as predicted by our ranking function $\widehat{F}$ and the first set of continuous parameters generated by our learned sampler π_θ. It resets to the initial state if it samples an infeasible action.

- i-rsc: Iterative Resolve Spatial Constraint. In i-rsc, we extend rsc [119], which is the state-of-the-art algorithm for manipulation among movable obstacles, to moving multiple objects to a goal region. To do this, we first plan pick-and-place motions for moving goal objects to goal regions with checking collisions only at object placements and initial and final robot configurations. From this,

we get an order to pack objects into the goal regions. Each single-object packing sub-problem is solved by an application of RSC. If, after some number of iterations, RSC does not find a solution, we modify the object ordering and try randomly permuting the unplaced boxes. The algorithm will eventually try all orderings if given enough time.

- SAPS-HCOUNT: SAPS with a hand-designed state-based-heuristic function shown in Eqn 2.2.1.

PURELEARNING is a pure-learning method, while I-RSC and SAPS-HCOUNT are pure-planning methods. For our guided-planner, we have two versions:

- SAPS-RANK: uses the priority function in equation 3.2.1 in SAPS, which includes the learned ranking term, but does not use the learned sampler.

- SAPS-RANK-SAMPLER: uses the priority function in equation 3.2.1 and the learned sampler inside SMPLCONT function in SAPS.

PURELEARNING is a pure-learning method, while I-RSC and SAPS-HCOUNT are pure-planning methods. For our guided-planner, we have two versions:

- SAPS-RANK: uses the priority function in equation 3.2.1 in SAPS, which includes the learned ranking term, but does not use the learned sampler.

- SAPS-RANK-SAMPLER: uses the priority function in equation 3.2.1 and the learned sampler inside SMPLCONT function in SAPS.

For the parameters of SAPS, we use $N_{smpl} = 2000$, $N_{mp} = 5$ in the box-moving domain, and $N_{smpl} = 50$ for the shelf domain. The priority function in 3.2.1 uses $\lambda = 1$. For training the sampler, we use $n_{tot} = 100000, n_c = 5, n_b = 32, lr_\theta = 1e-4, lr_\alpha = 1e-4$, and use $\mathcal{N}(0,1)$ for P_Z.

We train separate samplers for pick parameters and place parameters. For generating place parameters, we use the output from pick sampler as an input to place sampler. We also train separate place samplers for different regions, because different regions have different distributions over placements. We use 1000 planning episodes for training samplers in the cupboard domain, and 1500 planning episodes for training samplers in the box-moving domain. For training the ranking function, we use 250 planning episodes.

To generate our training data, we first solve problems using I-RSC and use the planning experience to train the ranking function. Then, we use SAPS-RANK to solve additional problems and use this planning

experience to train our sampler. To build the set of key configurations, we use the motion plans from I-RSC planning experience, and then sparsely sub-sample them by discarding the ones that are too close. To test the ranking function's generalization capability, we only collect planning experience from the box-moving domain, where the goal is to move a single box to the kitchen. Then, we test its performance in (1) the same box-moving domain, but where the goal is move four boxes to the kitchen, to demonstrate its capability to generalize to harder problems, and (2) the cupboard domain, where the goal is to move smaller objects into a packing box using one-arm pick-and-place, to demonstrate its capability to generalize to harder problems.

The learned samplers, which operate on lower geometric details, however, must be trained in each environment. For this reason, its planning experience data consists of a mixture of moving 1 and 4 boxes into the kitchen in the box-moving domain, and packing 1 object in the cupboard domain. We train separate samplers for these environments.

To evaluate each of these algorithms, we measure two quantities. One is the planning time, and the other is the success rate, both within time limits. To measure these, we test the algorithms on 25 problem instances in each setup. For algorithms that involve planning, we use 5 different planning seeds. For algorithms that involve learning we use 5 different training seeds.

Figure 5.8 (left), shows the planning time results for moving a single box in the box-moving domain. We see that the median of SAPS-RANK is 3 times faster than that of I-RSC, and about 1.5 times faster than that of SAPS-HCOUNT. Further, the median of SAPS-RANK-SAMPLER is 6 times faster than I-RSC, and 3 times faster than SAPS-HCOUNT. The guidance-based approaches have much lower 90th percentiles as well, indicating that they are better even for harder problem instances. I-RSC performs badly because it makes the *monotonicity* assumption, which states that problems can be solved by touching each object only once. This does not hold in this problem. PURELEARNING performs the worst among all the methods, due to its inability to overcome its prediction mistakes. This also evident in Table 5.2 (left, second column). Because of their inherent flaws, pure-learning and pure-planning methods only solve about half of the problems, compared to guidance-based approaches which solve more than 90% of the problems.

Figure 5.8 (left) shows the results for problems where the robot has to pack four boxes. We see that the results support our claim about generalizing to harder problem instances when learning only from easier instances. Even without retraining, our ranking function

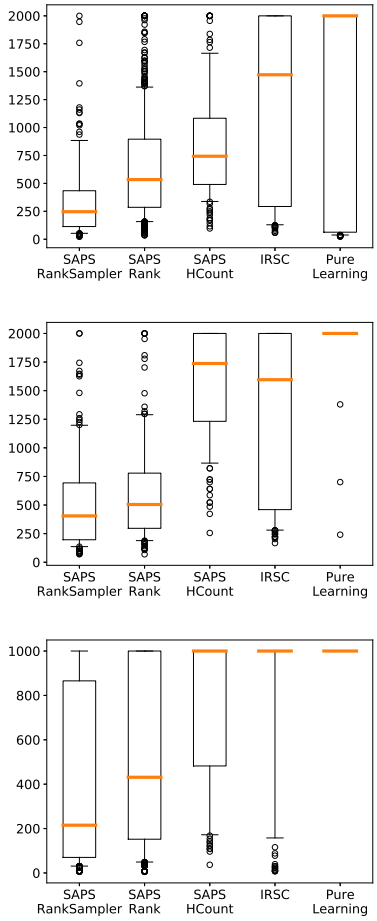


Figure 5.8: Whisker plots for planning times for moving one box in the box-moving domain (top) for moving four boxes in the same domain (middle), and for moving the target object in the cupboard domain (bottom). The box-moving domain had 2000 seconds time limit, and the cupboard domain has 1000 seconds time limit. Whiskers indicate 10th and 90th percentiles.

and the sampler can guide SAPS to have a significant advantage over the pure planning and pure-learning approaches. The median of SAPS-RANK-SAMPLER is almost 5.8 times faster than those of SAPS-HCOUNT and I-RSC, and SAPS-RANK is about 3.5 times faster. The 90th percentile is again significantly lower than the benchmarks. It is also worth noting the significant drop in the success rate of the pure-learning approach in Table 5.2 (left, third column). The main reason is that, because this is a longer-horizon problem than the one-box-moving scenario, there is more room for making prediction mistakes. In terms of success rates, the guidance based algorithms outperform the benchmarks.

Algorithm	Success rate (1 box, 2000s)	Success rate (4 boxes, 2000s)
I-RSC	0.53	0.51
PURELEARNING	0.43	0.03
SAPS-HCOUNT	0.94	0.56
SAPS-RANK	0.94	0.96
SAPS-RANK-SAMPLER	**0.99**	**0.97**

Algorithm	Success rate (1 obj, 1000s)
I-RSC	0.21
PURELEARNING	0.00
SAPS-HCOUNT	0.44
SAPS-RANK	0.75
SAPS-RANK-SAMPLER	**0.81**

Table 5.2: Success rates of different algorithms in the box-moving domain with 2000s time limit (left) and cupboard domain 1000s time limit (right)

Lastly, we consider the cupboard domain. Here, we retrain our sampler by collecting more data in this domain, but not our ranking function, and apply the same ranking function that we learned from the box-moving domain. We again see the significant improvement in the medians of the guidance-based approaches compared to pure planning and pure learning approaches. I-RSC especially suffers in this domain because the environment is tighter than the box-moving domain, making more instances of non-monotonic problems. Similarly, the tightness in the environment requires longer horizon plans, making PURELEARNING to suffer. The success rates in Table 5.2 (right) indicates that guidance-based algorithms, SAPS-RANK-SAMPLER and SAPS-RANK, significantly outperform the benchmarks and improves the success rate by a factor of 1.8.

6

Learning on-line to guide planning: Voronoi Optimistic Optimization applied to Trees

So far, we presented algorithms that learn from past off-line experience a biased sampler and abstract ranking function to guide planning. However, naively following the suggestions made by these predictors would lead to poor performance, because they might make prediction mistakes. In this chapter, we present a novel planning algorithm that balances between exploring new choices and exploiting the prior search guidance knowledge by learning the values of actions *on-line*. The work presented in this chapter is based on our conference paper [63] which was a work done in collaboration with Kyungjae Lee and Sungbin Lim.

More concretely, we consider a setup in which we are given a reward function, deterministic transition model, and planning horizon, and the objective is to find a plan that maximizes the sum of rewards. Our algorithm is a variant of Monte Carlo Tree Search (MCTS) that learns the action-value function on-line based on the Monte Carlo simulations of trajectories. This on-line learning capability enables the algorithm to correct the prior heuristic function or biased sampler to explore more efficiently.

In discrete action spaces, MCTS is a well-studied algorithm [70], and we can use them at the nodes where we have to make abstract action choices. However, the main challenge lies in how to efficiently make continuous action choices when applying MCTS to G-TAMP problems. In particular, the parameters of manipulation operators are high dimensional, consisting of multiple pick or placement configurations of the robot.

Therefore, the main contribution in this work is extending MCTS to continuous action space problems. There are several challenges involved in this. First, we have a high-dimensional continuous action spaces with possibly a discontinuous objective function. For example, consider the sequential robot mobile-manipulation planning problem

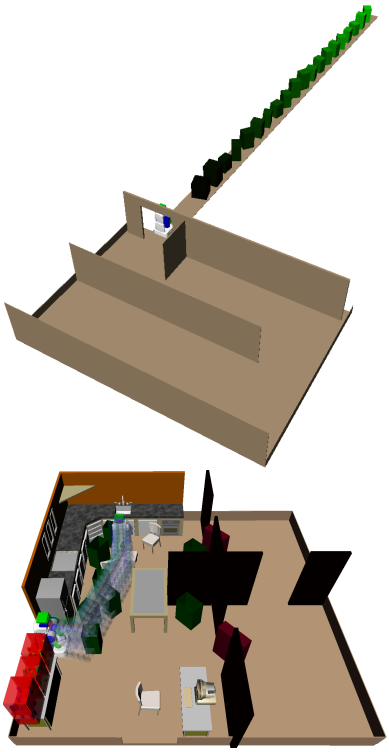


Figure 6.1: Packing domain: the task is to pack as many objects coming from a conveyor belt into the room (left). Object-clearing domain: obstacles must be cleared from the swept-volume of a path to the sink (right). In both domains, the robot needs to minimize the overall trajectory length.

shown in Figure 6.1 (left). In this domain, the objective function is defined to be the number of objects that the robot packs into the storage room while satisfying feasibility conditions, such as collision-free motions, and minimizing the total length of its trajectory. Another example is shown in Figure 6.1 (right), where the task is to clear obstacles from a region, and the objective is a function of the number of obstacles cleared and trajectory length. In both cases, the robot's action space is high dimensional, consisting of multiple pick or placement configurations of the robot.

More generally, such discontinuous objective functions are the sum of a finite set of step functions in a high-dimensional state-action space, where each step corresponds to the occurrence of an important event, such as placing an object. For classes of functions of this kind, standard gradient-based optimization techniques are not directly applicable, and even if we smooth the objective function, the solution is prone to local optima.

Recently, several gradient-free approaches to continuous-space planning problems have been proposed [14, 92, 131, 87], some of which have been proven to asymptotically find a globally optimal solution. These approaches either frame the problem as simultaneously optimizing a whole action sequence [14, 131] or treat the action space in each node of a tree search [87] as the search space for a budgeted-black-box function optimization (BBFO) algorithm, and use hierarchical-partitioning-based optimization algorithms [91, 13] to approximately find the globally optimal solution.

While these hierarchical-partitioning algorithms handle a richer class of objective functions than traditional methods [96], their main drawback is poor scalability to high-dimensional search spaces: to optimize efficiently, these algorithms sequentially construct partitions of the search space where, at each iteration, they create a finer-resolution partition inside the most promising cell of the current partition. The problem is that constructing a partition requires deciding the optimal dimension to cut, which is a difficult combinatorial problem especially in a high-dimensional space. Figure 6.2 (Top) illustrates this issue with one of the algorithms, DOO [91].

We propose a new BBFO algorithm called Voronoi Optimistic Optimization (VOO) which, unlike the previous approaches, only implicitly constructs partitions, and so scales to high-dimensional search spaces more effectively. Specifically, partitions in VOO are Voronoi partitions whose cells are implicitly defined as the set of all the points that are closer to the generator than to any other evaluated point. Figure 6.2 (right) shows an example.

Given as inputs a semi-metric, a bounded search space, and an exploration probability ω, VOO operates similarly to the previous

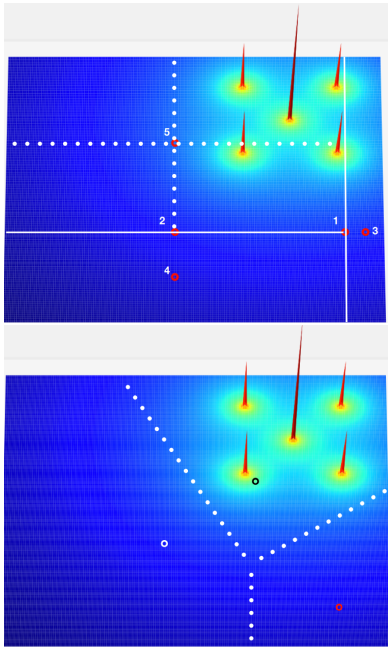


Figure 6.2: Top: Illustrations of a partition made by DOO when five points are evaluated to optimize a 2D Shekel function. Each solid line shows the partitions made by the point that is on it. Numbers indicate the order of evaluations. The dotted lines indicate the two possible partitions that can be made by the fifth point, and depending on this choice, the performance differs. Bottom: Illustration of the Voronoi partition implicitly constructed by VOO. We can sample from the best Voronoi cell (defined by the black point) by random-sampling points, and rejecting them until we obtain one that is closer to the black point than the other points. We can sample a point with Voronoi bias by uniformly sampling from the entire search space; the cell defined by the white point is most likely to be selected.

partition-based methods: at each iteration, it selects (implicitly) a Voronoi cell based on a simple exploration-exploitation scheme, samples a point from the cell, and (implicitly) makes finer-resolution cells inside the selected cell based on the sampled point. The selection of a Voronoi cell is based on the given exploration probability: with probability ω, it explores by selecting a cell with probability proportional to the volume of the cell; with probability $1 - \omega$, it exploits by selecting the cell that contains the current best point. Unlike the previous methods, however, VOO never explicitly constructs the partitions: by using the definition of Voronoi partition and the given semi-metric, sampling from the best cell is implemented simply using rejection sampling. Sampling a point based on the volumes of the cells, which is also known as the *Voronoi bias* [77], is also simply implemented by sampling uniformly at random from the search space. Figure 2.1 (right) demonstrates this point. We prove the regret bound of VOO which shows that under some mild assumptions, the regret goes to zero.

Using VOO, we propose a novel continuous state-action-space Monte Carlo tree search (MCTS) algorithm, *Voronoi optimistic optimization applied to trees* (VOOT) that uses VOO at each node of the search tree to select the optimal action, in a similar fashion to HOOT [87]. HOOT, however, does not come with performance guarantees; we are able to prove a performance guarantee for VOOT, which is derived from a bound on the regret of VOO. The key challenge in showing this result is that, when VOO is used to optimize the state-action value function of a node in the tree, the value function is non-stationary, so that even when the environment is deterministic, its value changes as the policy at the sub-tree below the action changes. We address this problem by using the regret of VOO at the leaf nodes, whose value function is stationary, and computing how many re-evaluations at each depth is required to maintain the same regret at the root node as at the leaf node. We show this regret can be made arbitrarily small.

We compare VOO to several algorithms on a set of standard functions for evaluating black-box function optimization algorithms in which the number of dimensions of the search space is as high as 20, and show that VOO significantly outperforms the benchmarks, especially in high dimensions. To evaluate VOOT, we compare it to other continuous-space MCTS algorithms in the two sequential robot mobile-manipulation problems shown in Figure 2.1, and show that VOO computes significantly better quality plans than the benchmarks, within a much smaller number of iterations.

6.1 Related work

There are several planning methods that use black-box function optimization algorithms in continuous-space problems. We first give an overview of the BBFO algorithms, and then describe planning algorithms that use them. We then give an overview of progressive-widening approaches, which are continuous-space MCTS algorithms that do not use black-box function optimization methods.

Global optimization of black-box functions with budget Several partition-based algorithms have been proposed [91, 13, 92]. In [91], two algorithms are proposed. The first algorithm is DOO, which requires as inputs a semi-metric and the Lipschitz constant for the objective function. It sequentially constructs partitions of the search space, where a cell in the partition has a representative point, on which the objective function is evaluated. Using the local-smoothness assumption, it builds an upper-bound on the un-evaluated points in each cell using the distance from the representative point. It chooses the cell with the highest-upper bound, and creates a finer-resolution cell inside of it, and repeats. The second algorithm proposed in [91] is SOO, which does not require a Lipschitz constant, and evaluates all cells that might contain the global optimum. In [13], Hierarchical Optimistic Optimization (HOO) is proposed. Unlike SOO and DOO, HOO can be applied to optimize a noisy function, and can be seen as the stochastic counterpart of DOO. So far, these algorithms have been applied to problems with low-dimensional search spaces, because solving for the optimal sequence of dimensions to cut at each iteration is difficult. VOO gets around this problem by not explicitly building the partitions.

Alternatively, we may use Bayesian optimization (BO) algorithms, such as GP-UCB [116]. A typical BO algorithm takes as inputs a kernel function, and an exploration parameter, and assumes that the objective function is a sample from a Gaussian Process (GP). It builds an acquisition function, such as upper-confidence-bound function in GP-UCB [116], and it chooses to evaluate, at every iteration, the point that has the highest acquisition function value, updates the parameters of the GP, and repeats. The trouble with these approaches is that at every iteration, they require finding the global optimum of the acquisition function, which is expensive in high dimensions. In contrast, VOO does not require an auxiliary optimization step.

There have been several attempts to extend BO to high-dimensional search spaces [128, 55]. However, they make a rather strong assumption on the objective function, such as that it lies on a low-dimensional manifold, or that it can be represented by a linear combination of functions of sub-dimensions, which are unlikely to hold

in domains such as robotics, where all of the action dimensions contribute to its value. Also, these methods require extra hyperparameters that define the lower-dimensional search space that are tricky to tune. VOO requires neither the assumption or the hyperparameters for defining the low-dimensional search space.

There are also methods that try to combine BO and hierarchical partitioning methods, such as [126, 57]. The idea is to use hierarchical partitioning methods to optimize the acquisition function of BO; unfortunately, for the same reason as hierarchical partitioning methods, they tend to perform poorly in higher dimensional spaces.

Optimal planning in continuous spaces using BBFO There are two approaches to continuous-space planning problems that use black-box function-optimization (BBFO) algorithms. In the first group of approaches, the entire sequence of actions is treated as a single search space for optimization. In [131], the authors propose *hierarchical open-loop optimistic planning* (HOLOP), which uses HOO for finding finite-horizon plans in stochastic environments with continuous action space. In [14], the authors propose an algorithm called *simultaneous optimistic optimization for planning* (SOOP), that uses SOO to find a plan when the environment is deterministic. These methods become very expensive as the length of the action sequence increases.

The second group of approaches, where our method belongs, performs a sample-based tree search with a form of continuous-space optimizer at each node. Our work most closely resembles *hierarchical optimistic optimization applied to trees* (HOOT) [87], which applies *hierarchical optimistic optimization* (HOO) at every node in MCTS for the action-optimization problem, but does not provide any performance guarantees. These algorithms have been limited to problems with low-dimensional action space, such as the inverted pendulum. Our experiments demonstrate VOOT can solve problems with higher-dimensional action spaces much more efficiently than these algorithms.

Widening techniques for MCTS in continuous action spaces There are progressive-widening (PW) algorithms that extend MCTS to continuous action spaces [19, 5], but unlike the approaches above, their main concern is deciding when to sample a new action, instead of which action to sample. The action-sampler in these PW algorithms is assumed to be an external function that has a non-zero probability of sampling a near-optimal action, such as a uniform-random sampler.

Typically, a PW technique [19] ensures that the ratio between the number of sampled actions in a node to the number of visits to the node is above a given threshold. In [5], the authors show that a form of PW can guarantee that each state's estimated value approaches

the optimal value asymptotically. However, this analysis does not take into consideration the regret of the action sampler, and assumes that the probability of sampling a near-optimal action is the same in every visit to the node. So, if an efficient action-sampler, whose regret reduces quickly at each visit, is used, their error bound would be very loose. Our analysis shows how the regret of voo affects the planning performance.

6.2 *Monte Carlo planning in continuous state-action spaces*

We have a continuous state space $\mathcal{S}$, a continuous action space A, a deterministic transition model of the environment, $T : \mathcal{S} \times A \to \mathcal{S}$, a deterministic reward function $R : \mathcal{S} \times A \to \mathbb{R}$, and a discount factor $\gamma \in [0, 1)$. Our objective is to find a sequence of actions with planning horizon H that maximizes the sum of the discounted rewards $\max_{a_0, \cdots, a_{H-1}} \sum_{t=0}^{H-1} \gamma^t r(s_t, a_t)$ where $s_{t+1} = T(s_t, a_t)$. Our approach to this problem is to use MCTS with an action-optimization agent, which is an instance of a black-box function-optimization algorithm, at each node in the tree.

We now describe the general MCTS algorithm for continuous state-action spaces, which is given in Algorithm 5. The algorithm takes as inputs an initial state s_0, an action-optimization algorithm $\mathcal{A}$, the total number of iterations N_{iter}, the re-evaluation parameter $N_r \in [0, N_{iter}]$, and its decaying factor $\kappa_r \in [0, 1]$. It begins by initializing the necessary data in the root node. U denotes the set of actions that have been tried at the initial node, $\hat{Q}$ denotes the estimated state-action value of the sampled actions, and n_r denotes the number of times we re-evaluated the last-sampled action. It then performs N_{iter} Monte Carlo simulations, after which it returns the apparently best action, the one with the highest estimated state-action value. This action is executed, and we re-plan in the resulting state.

Algorithm 5 MCTS($s_0, \mathcal{A}, N_{iter}, N_r, \kappa_r, H, \gamma$)

1: global variables: $T, R, H, \gamma, \mathcal{A}, N_{iter}, \kappa_r, H, \gamma$
2: $\mathcal{T}(s_0) = \{A = \emptyset, \hat{Q}(s_0, \cdot) = -\infty, n_r = 0\}$
3: **for** $i = 1 \to N_{iter}$
4: SIMULATE($s_0, 0, N_r$)
5: **end for**
6: **return** $\text{argmax}_{a \in \mathcal{T}(s_0).A} \mathcal{T}(s_0).\hat{Q}(s_0, a)$

Procedure SIMULATE is shown in Algorithm 6. It is a recursive function whose termination condition is either encountering an infeasible state or reaching a depth limit. At the current node $\mathcal{T}(s)$, it either selects the action that was most recently sampled, if it has not yet been evaluated N_r times and we are not in the last layer of the

Algorithm 6 SIMULATE(s, h, N_r)

global variables: $T, R, H, \gamma, \mathcal{A}, N_{iter}, \kappa_r, H, \gamma$
if $s ==$ infeasible or $h == H$
    return 0
end if
if $(|\mathcal{T}(s).U| > 0) \wedge (\mathcal{T}(s).n_r < N_r) \wedge (h \neq H - 1)$
    // re-evaluate the last added action
    $a = \mathcal{T}.U$.get_last_added_element()
    $\mathcal{T}(s).n_r = \mathcal{T}(s).n_r + 1$
else
    // Perform action optimization
    $a \sim \mathcal{A}(\mathcal{T}(s).\hat{Q})$
    $\mathcal{T}(s).U = \mathcal{T}(s).U \cup \{u\}$
    $\mathcal{T}(s).n_r = 1$
end if
$s' = T(s, a)$
$r = R(s, a)$
$\hat{Q}_{new} = r + \gamma \cdot$ SIMULATE$(s', h + 1, N_r \cdot \kappa_r)$
if $\hat{Q}_{new} > \mathcal{T}(s).\hat{Q}(s, a)$
    $\mathcal{T}(s).\hat{Q}(s, a) = \hat{Q}_{new}$
end if
return $\mathcal{T}(s).\hat{Q}(s, a)$

tree, or it samples a new action. To sample a new action, it calls $\mathcal{A}$ with estimated Q-values of the previously sampled actions, $\mathcal{T}(s).\hat{Q}$. A transition is simulated based on the selected action, and the process repeats until a leaf is reached; Q-value updates are performed on a backward pass up the tree if a new solution with higher value has been found (note that, because the transition model is deterministic, the update only requires maximization.)

The purpose of the re-evaluations is to mitigate the problem of non-stationarity: an optimization algorithm $\mathcal{A}$ assumes it is given evaluations of a stationary underlying function, but it is actually given $\hat{Q}(s, a_t)$, whose value changes as more actions are explored in the child sub-tree. This problem is also noted in [87]. So, we make sure that $\hat{Q}(s, a_t) \approx Q^*(s, a_t)$ before adding an action a_{t+1} in state s by sampling more actions at the sub-tree associated with a_t. Since at the leaf node $Q^*(s, a_t) = R(s, a_t)$, we do not need to re-evaluate actions in leaf nodes. In section 5, we analyze the impact of the estimation error in $\hat{Q}$ on the performance at the root node.

One may wonder if it is worth it to evaluate the sampled actions same number of times, instead of more sophisticated methods such as Upper Confidence Bound (UCB), for the purpose of using an action-optimization algorithm $\mathcal{A}$. Typical continuous-action tree search methods perform progressive widening (PW) [19, 5], in which they sample new actions from the action space uniformly at random,

but use UCB-like strategies for selecting which of the previously-sampled actions to explore further. In this case, the objective for allocating trials is to find the highest-value action among a discrete set, not to obtain accurate estimates of the values of all the actions.

VOOT operates in continuous action spaces but performs much more sophisticated value-driven sampling of the continuous actions than PW methods. To do this, it needs accurate estimates of the values of the actions it has already sampled, and so we have to allocate trials even to actions that may currently "seem" suboptimal. Our empirical results show that this trade-off is worth making, especially in high-dimensional action spaces.

6.3 Voronoi optimistic optimization

Given a bounded search space $\mathcal{X}$, a deterministic objective function $f : \mathcal{X} \to \mathbb{R}$ and a numerical function evaluation budget n, our goal is to devise an exploration strategy over $\mathcal{X}$ that, after n evaluations, minimizes the *simple regret* defined as $f(x_\star) - \max_{t\in[n]} f(x_t)$, where $f(x_\star) = \max_{x\in\mathcal{X}} f(x)$, x_t is a point evaluated at iteration t, and $[n]$ is shorthand for $\{1, \cdots n\}$. Since our algorithm is probabilistic, we will analyze its expected behavior. We define the simple regret of a probabilistic optimization algorithm $\mathcal{A}$ as

$$\mathcal{R}_n = f(x_\star) - \mathbb{E}_{x_{1:t}\sim\mathcal{A}}\left[\max_{t\in[n]} f(x_t)\right]$$

Our algorithm, voo (Algorithm 7), operates by implicitly constructing a Voronoi partition of the search space $\mathcal{X}$ at each iteration: with probability ω, it samples from the entire search space, to sample from a Voronoi cell with probability proportional to its volume; with probability $1 - \omega$, it samples from the *best Voronoi cell*, which is the one induced by the current best point, $x_t^* = \arg\max_{i\in[t]} f(x_i)$.

Algorithm 7 voo$(\mathcal{X}, \omega, d(\cdot,\cdot), n)$

for $t = 0 \to n-1$
  Sample $\nu \sim Unif[0,1]$
  if $\nu \leq \omega$ **or** $t == 0$
    x_{t+1} =UnifSample$(\mathcal{X})$
  else
    x_{t+1} =SampleBestVCell$(d(\cdot,\cdot))$
  end if
  Evaluate $f_{t+1} = f(x_{t+1})$
end for
return $\arg\max_{t\in\{0,\ldots,n-1\}} f_t$

It takes as inputs the bounded search space $\mathcal{X}$, the exploration

probability ω, a semi-metric $d(\cdot,\cdot)$, and the budget n. The algorithm has two sub-procedures. The first one is UnifSample, which samples a point from $\mathcal{X}$ uniformly at random, and SampleBestVCell, which samples from the best Voronoi cell uniformly at random. The former implements exploration using the Voronoi bias, and the latter implements exploitation of the current knowledge of the function. Procedure SampleBestVCell can be implemented using a form of rejection sampling, where we sample a point x at random from $\mathcal{X}$ and reject samples until $d(x, x_t^*)$ is the minimum among all the distances to the evaluated points. Efficiency can be increased by sampling from a Gaussian centered at x_t^*, which we found to be effective in our experiments.

To use voo as an action optimizer in Algorithm 6, we simply let A be the search space, and use the semi-metric $d(\cdot,\cdot)$. $f(\cdot)$ is now the value function $Q^*(s,\cdot)$ at each node of the tree, whose estimation is $\hat{Q}(s,\cdot)$. The consequence of having access only to $\hat{Q}$ instead of the true optimal state-action value function Q^* will be analyzed in the next section.

6.4 *Analysis of* voo *and* voot

We begin with definitions. We denote the set of all global optima as $\mathcal{X}^\star$, the Voronoi cell generated by a point x as $\mathcal{C}(x)$. We define the *diameter* of $\mathcal{C}(x)$ as $\sup_{y\in\mathcal{C}(x)} d(x,y)$ where $d(\cdot,\cdot)$ is the semi-metric on $\mathcal{X}$.

Suppose that we have a Voronoi cell generated by x, $\mathcal{C}_0(x)$. When we randomly sample a point z from $\mathcal{C}_0(x)$, this will create two new cells, one generated by x, which we denote with $\mathcal{C}_1(x)$, and the other generated by z, denoted $\mathcal{C}_1(z)$. The diameters of these new cells would be random variables, because z was sampled randomly. Now suppose that we have sampled a sequence of n_0 points from the sequence of Voronoi cells generated by x, $\{\mathcal{C}_0(x), \mathcal{C}_1(x), \mathcal{C}_2(x), \cdots, \mathcal{C}_{n_0}(x)\}$. Then, we define the *expected diameter* of a Voronoi cell generated by x as the expected value of the diameter of the last cell, $\mathbb{E}[\sup_{y\in\mathcal{C}_{n_0}(x)} d(x,y)]$.

We write δ_{max} for the largest distance between two points in $\mathcal{X}$, $B_r(x)$ to denote a ball with radius r centered at point x, and $\bar{\mu}_B(r) = \frac{\mu(B_r(\cdot))}{\mu(\mathcal{X})}$ where $\mu(\cdot)$ is a Borel measure defined on $\mathcal{X}$. We make the following assumptions:

A 1. *(Translation-invariant semi-metric) $d : \mathcal{X}\times\mathcal{X} \to \mathbb{R}^+$ is such that $\forall x,y,z \in \mathcal{X}$, $d(x,y) = d(y,x)$, $d(x,y) = 0$ if and only if $x = y$, and $d(x+z, y+z) = d(x,y)$.*

A 2. *(Local smoothness of f) There exists at least one global optimum $x_\star \in \mathcal{X}$ of f such that $\forall x \in \mathcal{X}, f(x_\star) - f(x) \leq L \cdot d(x, x_\star)$ for some*

$L > 0$.

A 3. *(Shrinkage ratio of the Voronoi cells) Consider any point y inside the Voronoi cell $\mathcal{C}$ generated by the point x_0, and denote $d_0 = d(y, x_0)$. If we randomly sample a point x_1 from $\mathcal{C}$, we have $\mathbb{E}[\min(d_0, d(y, x_1))] \leq \lambda d_0$ for $\lambda \in (0, 1)$.*

A 4. *(Well-shaped Voronoi cells) There exists $\eta > 0$ such that for any Voronoi cell generated by x with expected diameter d_0 contains a ball of radius ηd_0 centered at x.*

A 5. *(Local symmetry near optimum) $\mathcal{X}_\star$ consists of finite number of disjoint and connected components $\{\mathcal{X}_\star^{(\ell)}\}_{\ell=1}^k$, $k < \infty$. For each component, there exists an open ball $B_{\nu_\ell}(x_\star^{(\ell)})$ for some $x_\star^{(\ell)} \in \mathcal{X}_\star^{(\ell)}$ such that $d(x, x_\star^{(\ell)}) \leq d(y, x_\star^{(\ell)})$ implies $f(x) \geq f(y)$ for any $x, y \in B_{\nu_\ell}(x_\star^{(\ell)})$.*

We now describe the relationship between these assumptions and those used in the previous literature. A1 and A2 are assumptions also made in [91]. These make the weaker version of the Lipschitz assumption applied only to the global optima, instead of every pair of points in $\mathcal{X}$. A3 and A4 are also very similar to the assumptions made in [91]. In [91], the author assumes that cells decrease in diameter as more points are evaluated inside of them and that each shell is well-shaped, in that it always contains a ball. Our assumption is similar, except that in our case, A3 and A4 are stated in terms of expectation, because VOO is a probabilistic algorithm.

A5 is an additional assumption that previous literature has not made. It assumes the existence of a ball inside of which, as you get closer to an optimum, the function values increase. It is possible to drastically relax this assumption to the existence of a sequence of open sets, instead of a ball, whose values increase as you get closer to an optimum. In our proof, we prove the regret of VOO in this general case, and Theorem 3 holds as the special case when A5 is assumed. We present this particular version for the purpose of brevity and comprehensibility, at the expense of generality.

Define $\nu_{min} = \min_{\ell \in [k]} \nu_\ell$. We have the following regret bound for VOO. All the proofs can be found in our conference paper [63].

Theorem 3. *Let n be the total number of evaluations. If $\frac{1-\lambda^{1/k}}{\bar{\mu}_B(\nu_{min})+1-\bar{\mu}_B(\eta \cdot \lambda \delta_{max})} < \omega$, we have*

$$\begin{aligned}\mathcal{R}_n \leq & L\delta_{max} C_1 \left[\lambda^{1/k} + \omega(1 - \bar{\mu}_B(\eta \cdot \lambda^n \delta_{max}))\right]^n \\ & + L\delta_{max} C_2 [(1 - \omega k \bar{\mu}_B(\nu_{min})) \cdot (1 + \lambda^{1/k})]^n\end{aligned}$$

where C_1 and C_2 are constants as follows

$$C_1 := \frac{1}{1-\rho(\lambda^{1/k}+1-[1-\omega+\omega\bar{\mu}_B(\eta\cdot\lambda\delta_{max})])^{-1}},$$
$$\rho := 1-\omega\bar{\mu}_B(\nu_{min}),$$
$$\textit{and } C_2 := \frac{\lambda^{-1/k}+1}{(\lambda^{-1/k}+1)-(1-\omega\bar{\mu}_B(\nu_{min})^{-1}}$$

Some remarks are in order. Define an *optimal cell* as the the cell that contains a global optimum. Intuitively speaking, when our best cell is an optimal cell, the regret should reduce quickly because when we sample from the best cell with probability $1-\omega$, we always sample from the optimal cell, and we can reduce our expected distance to an optimum by λ. And because of A5, the best cell is an optimal cell if we have a sample inside one of $B_{\nu_\ell}(x_\star)$.

Our regret bound verifies this intuition: the first term decreases quickly if λ is close to 0, meaning that if we sample from an optimal cell, then we can get close to the optimum very quickly. The second term says that, if $\bar{\mu}_B(\nu_{min})$, the minimum probability that the best cell is an optimal cell, is large, then the regret reduces quickly. We now have the following corollary showing that VOO is no-regret under certain conditions on λ and $\bar{\mu}_B(\nu_{min})$.

Corollary 6.1. *If $\frac{\lambda^{1/k}}{(1+\lambda^{1/k})k\bar{\mu}_B(\nu_{min})} < \omega < 1-\lambda^{1/k}$ and $\frac{\lambda^{1/k}}{1-\lambda^{2/k}} < k\bar{\mu}_B(\nu_{min})$, then $\lim_{n\to\infty}\mathcal{R}_n = 0$.*

The regret bound of VOOT makes use of the regret bound of VOO. We have the following theorem.

Theorem 4. *Define $C_{max} = max\{C_1, C_2\}$. Given a decreasing sequence $\eta(h)$ with respect to h, $\eta(h) > 0$, $h \in \{0\cdots H-1\}$ and the range of ω as in Theorem 3, if $N_{iter} = \prod_{h=0}^{H-1} N_r(h)$ is used, where*

$$N_r(h) \geq \log\Big(\frac{\eta(h)-\gamma\eta(h+1)}{2L\delta_{max}C_{max}}\Big)\cdot\min(G_{\lambda,\omega}, K_{\nu,\omega,\lambda})$$

$G_{\lambda,\omega} = (\log\Big(\lambda^{1/k}+\omega\Big))^{-1}$, and $K_{\nu,\omega,\lambda} = (\log([(1-\omega\bar{\mu}_B(\nu_{min}))(1+\lambda^{1/k})]))^{-1}$, then for any state s traversed in the search tree we have

$$V_\star^{(h)}(s) - \hat{V}_{N_r(h)}^{(h)}(s) \leq \eta(h) \quad \forall h \in \{0,\cdots,H-1\}$$

This theorem states that if we wish to guarantee a regret of $\eta(h)$ at each height of the search tree, then we should use N_{iter} number of iterations, with $N_r(h)$ number of iterations at each node of height h.

To get an intuitive understanding of this, we can view the action optimization problem at each node as a BBFO problem that takes

account of the regret of the next state. To see this more concretely, suppose that $H = 2$. First consider a leaf node, where the problem reduces to a BBFO problem because there is no next state, and the regret of the node is equivalent to the regret of VOO. We can verify that by substituting $N_r(H-1)$ to the bound in Theorem 3 the regret of $\eta(H-1)$ is guaranteed. Now suppose that we are at the root node at height $H-2$. There are two factors that contribute to the regret at this node: the regret at the next state in height $H-1$, and the regret that stems from sampling non-optimal actions in this node, which is the regret of VOO. Because all nodes at height $H-1$ have a regret of $\eta(H-1)$, to obtain the regret of $\eta(H-2)$, the regret of VOO at the node at height $H-2$ must be $\eta(H-2) - \gamma N_r(H-1)$. Again, by substituting $N_r(H-2)$ to the bound in Theorem 3, we can verify that that it would yield the regret of $\eta(H-2) - \gamma N_r(H-1)$ as desired.

Now, we have the following remark that relates the desired constant regret at each node and the total number of iterations.

Remark 1. *If we set* $\eta(h) = \eta, \ \ \forall h \in \{0 \cdots H-1\}$, *and* $N_{iter} = (N_r)^H$ *where*

$$N_r = \log\Big(\frac{\eta(1-\gamma)}{2L\delta_{max}C_{max}}\Big) \cdot \min(G_{\lambda,\omega}, K_{\nu,\omega,\lambda})$$

then, for any state s *traversed in the search tree we have*

$$V_\star^{(h)}(s) - \hat{V}_{N_r(h)}^{(h)}(s) \leq \eta \quad \forall h \in \{0, \cdots, H-1\}$$

We draw a connection to the case of discrete action space with b number of actions. In this case, we can guarantee zero-regret at the root node if we explore all b^H number of possible paths from the root node to leaf nodes. In the continuous case, with assumptions A1-A5, it would require sampling infinite number of actions at a leaf node to guarantee zero-regret, rendering achieving zero-regret in problems with $H > 0$ impossible. So, this remark considers a positive expected regret of η. It show that to guarantee this, we need to explore at least $(N_r)^H$ paths from the root to leaf nodes, where N_r is determined by the regret-bound of our action-optimization algorithm VOO. Alternatively, if some other action-optimization algorithm such as DOO, SOO, or GP-UCB is used, then its regret bound can be readily used by computing the respective $N_r(h)$ values in Theorem 3, and its own N_r value in Remark 1. It is possible to prove a similar remark in an undiscounted case. Please see Remark 2 in the appendix of our paper [63].

6.5 *Experiments*

We designed a set of experiments with two goals: (1) test the performance of VOO on high-dimensional functions in comparison to other black-box function optimizers and (2) test the performance of VOOT on deterministic planning problems with high-dimensional action spaces in comparison to other continuous-space MCTS algorithms. All plots show mean and 95% confidence intervals (CIs) resulting from multiple executions with different random seeds.

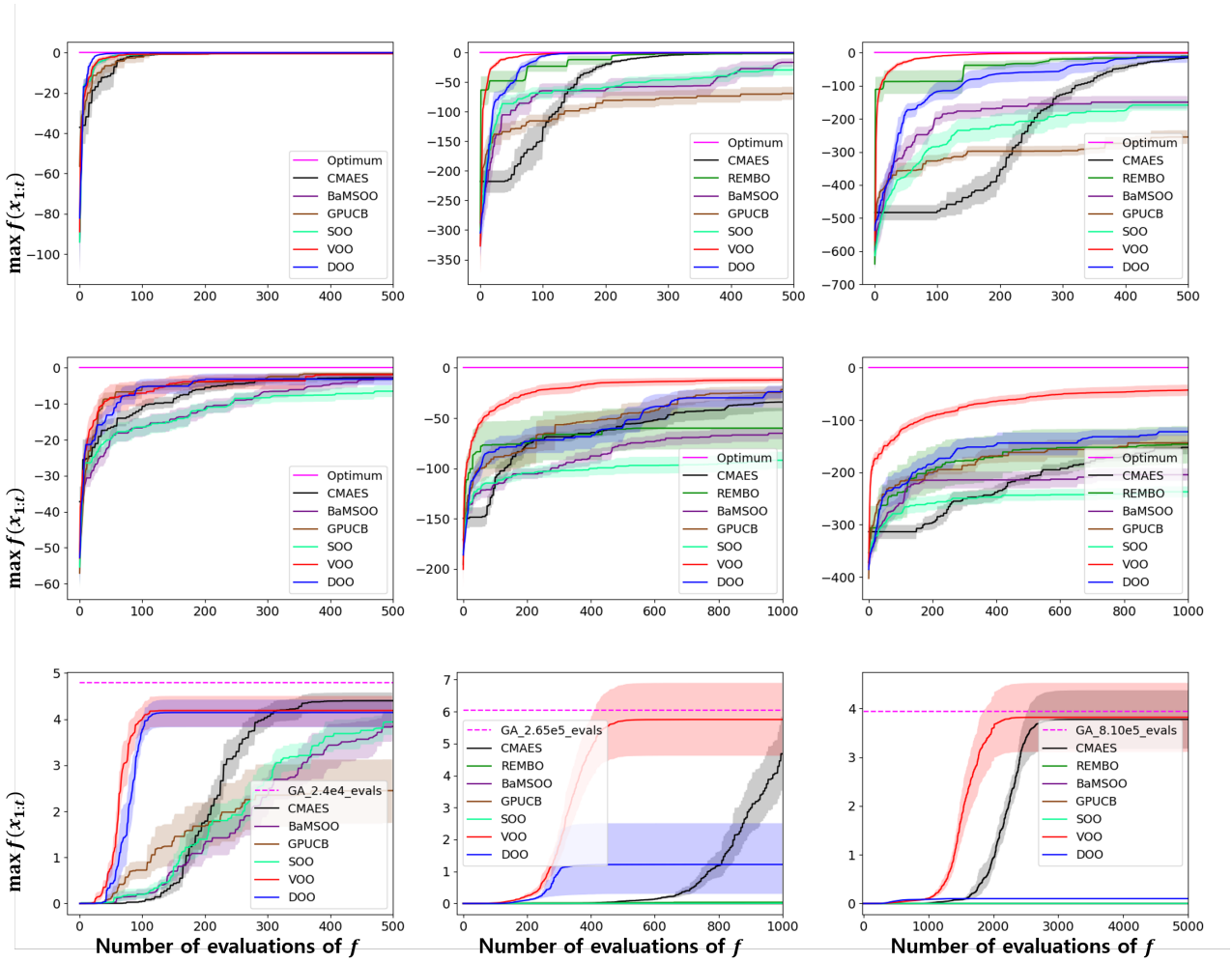


Figure 6.3: Griewank, Rastrigin, and Shekel functions (top to bottom) in 3, 10, and 20 dimensions (left to right)

Budgeted-black-box function optimization We evaluate VOO on three commonly studied objective functions from the DEAP [34] library: Griewank, Rastrigin, and Shekel. They are highly non-linear, with many local optima, and can extend to high-dimensional spaces. The true optimum of the Shekel function is not known; to gauge the

optimality of our solutions, we attempted to find the optimum for our instances by using a genetic algorithm (GA) [102] with a very large budget of function evaluations.

We compare VOO to GP-UCB, DOO, SOO, CMA-ES, an evolutionary algorithm [10], REMBO, the BO algorithm for high-dimensional space that works by projecting the function into a lower-dimensional manifold [128], and BAMSOO, which combines BO and hierarchical partitioning [126]. All algorithms evaluate the same initial point. We ran each of them with 20 different random seeds. We omit the comparison to HOO, which reduces to DOO on deterministic functions. We also omit testing REMBO in problems with 3-dimensional search spaces. Detailed descriptions of the implementations and extensive parameter choice studies are in the appendix of our paper [63].

Results are shown in Figure 6.3. In the 3-dimensional cases, most algorithms work fairly well with VOO and DOO performing similarly. But, as the number of dimensions increases, VOO is significantly better than all other methods. Purely hierarchical partitioning methods, DOO and SOO suffers because it is difficult to make the optimal partition, and SOOsuffers more than DOO because it does not take advantage of the semi-metric; the mixed approach of BO and hierarchical partitioning, BAMSOO, tends to do better than SOO, but still is inefficient in high dimensions for the same reason as SOO. GP-UCB suffers because in higher dimensions it becomes difficult to globally optimize the acquisition function. REMBO assumes that the objective function varies mostly in a lower-dimensional manifold, and there are negligible changes in the remaining dimensions, but these assumptions are not satisfied in our test functions, and VOO, which doesn't make this assumption, outperforms it. CMA-ES performs a large number of function evaluations to sustain its population, making it less suitable for *budgeted*-optimization problems where function evaluations are expensive.

This trend is more pronounced in the Shekel function, which is flat over most of its domain, but does increase near the optimum (see the 2D version in Figure 6.2). DOO, SOO, and BAMSOO perform poorly because they allocate samples to large flat regions. GP-UCB performs poorly because in addition to the difficulty of optimizing the acquisition function, the function is not well modeled by a GP with a typical kernel, and the same goes for REMBO. VOO has neither of these problems; as soon as VOO gets a sample that has a slightly better value, it can concentrate its sampling to that region, which drives it more quickly to the optimum. We do note that CMA-ES is the only method besides VOO to perform at all well in high-dimensions.

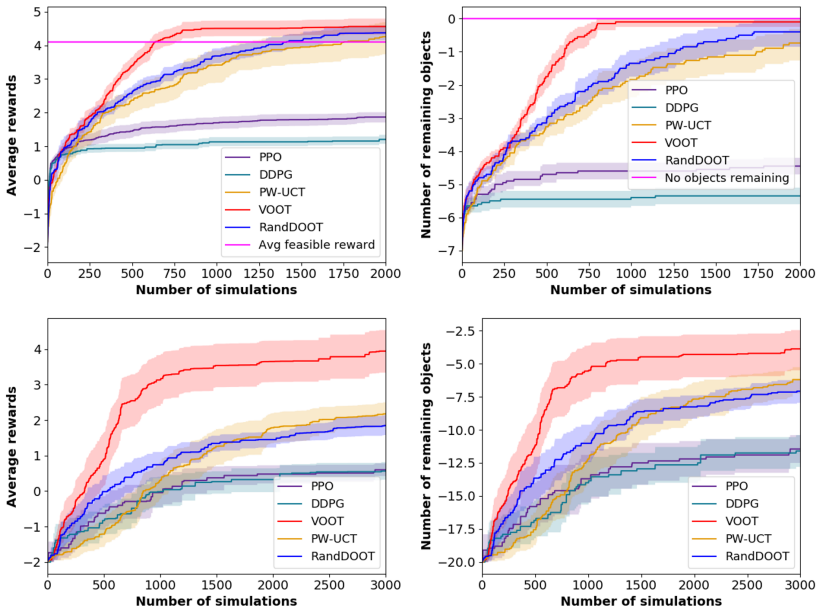


Figure 6.4: (Top-left) max sum of rewards vs. N_{iter} for the object clearing domain (Bottom-left) that for the packing domain. (Top-right) minus the number of remaining objects that need to be moved vs. N_{iter} in the object clearing domain (Bottom-right) that for the packing domain.

Sequential mobile manipulation planning problems We now study two realistic robotic planning problems. We compare VOOT to DOOT, which respectively use VOO and DOO and as its action-optimizer in Algorithm 6, and a single-progressive-widening algorithm that uses UCT (PW-UCT) [19]. But to make DOOT work in these problems, we consider a randomized variant called RAND-DOOT which samples an action uniformly in the cell to be evaluated next, instead of always selecting the mid-point, which could not solve any of these problems.

The objective of comparing to PW-UCT is to verify our claim that using an efficient action-optimizer, at the expense of uniform re-evaluations of the sampled actions, is better evaluating sampled actions with UCB at the expense of sampling new actions uniformly. The objective of comparing to RAND-DOOT is to verify our claim that VOOT can scale to higher dimensional problems for which RAND-DOOT does not.

In addition to the state-of-the-art continuous MCTS methods, we compare VOO to the representative policy search methods typically used for continuous-action space problems, PPO [109] and DDPG [83]. We train the stochastic policy using the same amount of simulated experience that the tree-search algorithms use to find a solution, and report the performance of the best trajectory obtained.

The action-space dimensions are 6 and 9 in the object-clearing and packing domains, respectively. The detailed action-space and reward function definitions, and extensive hyper-parameter value studies

are given in the appendix of our paper [63]. The plots in this section are obtained with 20 and 50 random seeds for object-clearing and packing problems, respectively.

We first consider the object-clearing problem (s_0 is shown in Figure 2.1 (right)). Roughly, the reward function penalizes infeasible actions and actions that move an obstacle but do not clear it from the path; it rewards actions that clear an object, but with value inversely proportional to the length of the clearing motion. The challenging aspect of this problem is that, to the right of the kitchen area, there are two large rooms that are unreachable by the robot; object placements in those rooms will be infeasible. So, the robot must clear obstacles within the relatively tight space of the kitchen.

Figure 6.4 (Top-left) shows the results. In this case, PW-UCT samples from the whole space, concentrating far too many of them in the unreachable empty rooms. RAND-DOOT also spends time partitioning the big unreachable regions, due to its large exploration bonus; however it performs better than PW-UCT because once the cells it makes in the unreachable region get small enough, it starts concentrating in the kitchen region. However, it performs worse than VOOT for similar reasons as in the Shekel problems: as soon as VOOT finds the first placement inside the kitchen (i.e. first positive reward), it immediately focuses its sampling effort near this area with probability $1 - \omega$. This phenomenon is illustrated in Figure 6.5, which shows the values of placements. We can also observe from Figure 6.4 (Bottom-left) that VOOT clears obstacles much faster than the other methods; it clears almost all of them with 750 simulations, while others require more than 1700, which is about a factor of 2.3 speed-up.

The reinforcement learning algorithms, PPO and DDPG, perform poorly compared to the tree-search methods. We can see that within the first 250 simulations, their rewards grow just as quickly as for the search algorithms, but they seem to get stuck at local optima, clearing only one or two obstacles. This is because the problem has two challenging characteristics: large future delayed rewards and sparse rewards.

The problem has sparse rewards because most of the actions are unreachable placements, or kinematically infeasible picks. It has large delayed rewards because the reward function is inversely proportional to the length of the clearing motion, but the first few objects need to be moved far away from their initial locations to make the subsequent objects accessible. Unfortunately, the RL methods come with an ineffective exploration strategy for long-horizon planning problems: Gaussian random actions[1]. This strategy could not discover the delayed future rewards, and the policies fell into a local optima in which they try to clear the first two objects with the least

[1] In order to get the RL methods to perform at all well, we had to tailor the exploration strategy to compensate for the fact that many of the action choices are completely infeasible. Details are in the appendix of our paper [63].

possible cost, but blocking the way to the subsequent objects.

We now consider the conveyor belt problem (s_0 shown in Figure 2.1 (left)). The challenge is the significant interdependence among the actions at different time steps: the first two boxes are too big to go through the door that leads to the bigger rooms, so the robot must place them in the small first room, so that there is still room to move the rest of the objects into the bigger rooms. Figure 6.4 (row 5, left) shows the results. VOOT achieves the reward of a little more than 3 with 1000 simulations, while other methods achieve below 1 ; even with 3000 simulations, their rewards are below 2, whereas that of VOOT goes up to approximately 4. Figure 6.4 (row 5, right) shows that VOOT finds a way to place as many as 15 objects within 1000 simulations, whereas the alternative methods have only found plans for placing 12 or 13 objects after 3000 simulations. We view each action-optimization problem (line 10 of Alg. 6) as a BBFO problem, since we only have access to the values of the actions that have been simulated, and the number of simulations is limited to N_{iter}. The RL approaches suffer in this problem as well, packing at most 8 boxes, while the worst search-based method packs 13 boxes. Again, the reason is the same as in the previous domain: sparse and delayed long-term rewards.

Figure 6.5: $\hat{Q}(s,a)$ of PW-UCT, RAND-DOOT, and VOOT (left to right) after 50 visits to the place node for the first object. Blue and purple bars indicate values of infeasible and feasible placements, respectively. Solid robot indicates the current state of the robot, and the transparent robots indicate the placements sampled. Notice VOOT has far fewer samples in infeasible regions.

6.6 *Discussion and future work*

We proposed a continuous MCTS algorithm in deterministic environments that scales to higher-dimensional spaces, which is based on a novel and efficient BBFO VOO. We proved a bound on the regret for VOO, and used it to derive a performance guarantee on VOOT. The tree performance guarantee is the first of its kind for search methods with BBFO-type algorithms at the nodes. We demonstrated that both VOO and VOOT significantly outperform previous methods within a small number of iterations in challenging higher-dimensional synthetic BBFO and practical robotics problems.

We believe there is a strong potential for combining learning and VOOT to tackle more challenging tasks in continuous domains, much like combining learning and Polynomial UCT has done in the game of Go [111]. We can learn from previous planning experience a policy π_θ, which assigns high probabilities to promising actions, using a reinforcement-learning algorithm. We can then use VOO with π_θ, instead of uniform sampling.

Another extension is combining the PW techniques [19, 5] and VOOT. The PW-algorithms primarily studies *when* to add a new action to the existing set of actions, and uses rather primitive uniform sampling to sample a new action. VOOT, on the other hand, is primarily about *which action* to sample, and uses VOO to sample a new action, and it is rather primitive when to add an action. It simply uses a fixed number of times to try an action. We believe we can devise an algorithm by integrating these two techniques with complementary strengths.

Part II

Task-and-motion planning

7
Learning to guide TAMP

TAMP encompasses a broader class of problems than G-TAMP in that it includes non-geometric and purely symbolic aspects, such as whether an object is cooked. Such generality makes it difficult to design a vector representation of problems because each one involves different sets of task constraints and symbolic attributes. To address this issue, we propose the *score-space* representation, which abstracts away from such variations [60]. In a score-space, a problem is represented in terms of the scores of representative solutions. The similarity between two potential solutions is measured by the covariance of their scores across different problem instances, which is estimated by the estimator that we propose. Using this representation, we propose a provably-efficient sequential prediction algorithm that predicts guidance, which speeds up planning time by several factors.

7.1 Related work

There is a substantial body of work aimed at improving motion planning performance on new problem instances based on previous experience on similar problem instances [9, 50, 94, 52, 82, 95]. The typical approach is to store a large set of solutions to earlier instances so that, when presented with a new problem instance, one can (a) retrieve the most relevant previous solution and (b) adapt it to the new situation. These methods differ in the way that they find the most relevant previous solution and how it is adapted.

Several of these approaches define a similarity metric between problem instances and retrieve solutions based on this metric. For example, [50] use the distance between the start and goal pairs as the metric, whereas, [94] based their metric on descriptions of quickly generated low-quality solutions for the current and previous instance. [52] use a mapping into a task-relevant space and measure similarity in that space, with a learned metric.

Instead of defining solution similarity, [9] define relevance of ear-

lier solutions by measuring the degree of constraint violation, for example collision, in the current situation. A related idea is developed by [95], where the search graph of past solutions is saved and the search for the current problem instance is biased towards the part of this past graph that is still feasible.

For more complex robotic planning, such as for mobile manipulation in cluttered environments, complete solutions are more difficult to adapt to new problem instances. In particular, the length of the plans is highly variable and they contain both discrete and continuous parameters.

Some earlier approaches have also focused on predicting partial solutions, in the form of a goal state or subgoals, instead of a complete solution. For instance, in the work of [26], the objective is to learn from previous examples a classifier (or regressor) that, given a hand-designed feature representation of a planning problem instance, enables choosing a goal that leads to a good locally optimal trajectory. In the approach of [33], the goal is to learn a model that predicts partial paths or subgoals, from a given parametric representation of a planning problem instance, aimed at enabling a randomized motion planner to navigate through narrow passages.

Several approaches have been proposed for learning representations for robot manipulation skills with varying set of objects [74, 75]. Specifically, [74] proposes a feature selection method that, given a certain basic features of objects in a scene, such as bounding boxes describing their shapes, predicts relevance of each feature for the given task. The relevant features are then used with a low-level robot motion skill represented with a dynamic motion primitive. While this approach is quite general, it still requires a human to design the basic set of features that are relevant for the tasks that the robot will encounter.

[136] proposed an approach to learn a representation for learning a complete policy from RGB images. Authors designed an architecture and defined a set of discrete high-level actions that allows an agent to accomplish different tasks. Similar to our approach, they learn a representation from planning experience. However, they learn a complete policy, whereas our approach learns to predict solution constraints. Moreover, their method does not take a robot into account - for manipulation tasks, however, visual representation greatly suffers from occlusion by the end-effectors of the robot. Our score-space representation abstracts away from this problem by directly representing a planning scene with how well the predicted constraints works.

Our approach can be seen as a method for choosing actions from a library; several methods have been proposed for this problem.

[21] propose a method that finds a fixed ordering of the actions in a library that optimizes a user-defined submodular function, for example, the probability that a sequence of candidate grasps will contain a successful one. Unlike our work, this method produces a static list, which does not change across different problem instances. Later, [22], generalized the approach by producing an ordered list of classifiers (operating over environment features) that select actions for a given problem instance. This approach again requires hand-designed features for the problem instances.

We formulate our problem as a black-box function optimization problem. In particular, BOX is motivated by the *principle of optimism in the face of uncertainty*, which is well surveyed by [92]. The main idea is to select the most "optimistic" item from the given set of items, by constructing an upper bound on the values of un-evaluated items. The performances of these algorithms are heavily influenced by how the upper-bounds are constructed. For instance, DOO (Deterministic Optimistic Optimization), developed by [91], first constructs the upper bound on the target function using manually specified semi-metric and a smoothness assumption on the target function. It then chooses the next point to evaluate based on this upper bound in order to balance exploration and exploitation.

Another suite of algorithms for black-box function optimization problems are Bayesian optimization algorithms [116, 127, 114]. In Bayesian optimization, an upper bound of the target function is constructed based on a hand-designed covariance matrix and the Gaussian process assumption. For instance, [116] constructs an upper bound using the confidence interval in an algorithm called Gaussian Process Upper Confidence Bound (GP-UCB). At each time step, GP-UCB evaluates the point that has the highest UCB value, observes the function value, and updates the Gaussian process. It returns the query point that resulted in the highest value.

The problem with these approaches is that, in general, they require a human to design a similarity function on problem instances and planning solutions: DOO requires a semi-metric, and Bayesian optimization algorithms require a kernel. This introduces human bias in constructing the upper-bounds on the target function, which strongly influences the performance of these algorithms. Moreover, as we have shown in the introduction, designing such a function is not trivial: even a small change in a problem instance may induce a large change in scores.

Rather than relying on a hand-designed similarity function over problem instances and planning solutions, our algorithm, BOX, operates in a vector space of scores of possible solutions where the correlation information among them can be computed easily.

7.2 Problem Formulation

Our premise is that calling the planner with a solution constraint, although much more efficient than the completely unconstrained problem, takes a significant amount of time and may generate significantly suboptimal plans if the solution constraint used is not a good match for the problem instance. Given a new instance we will call the planner with a fixed number of solution constraints and return the best plan obtained. So, our problem is which solution constraints should be tried, and in what order.

We formulate the problem as a black-box function optimization problem over a discrete space of candidate solution constraints, and use upper bounds constructed from experience on previous problem instances as well as the accumulated experience on this instance to determine which constraint to try next.

Formally, we have a sample space for problem instances, Ω, whose elements ω are distributed according to $P(\omega)$; a space of possible planning solutions, $\mathcal{X}$; and a space of solution constraints. The plan solution space includes all possible assignments to all of the decision variables, and the solution constraint space includes all possible assignments to a *subset* of decision variables for the given planning problem. The function $J(\omega, x)$ specifies the score of a solution $x \in \mathcal{X}$ on problem instance $\omega \in \Omega$. We assume a planner $\pi : \Omega \rightarrow \mathcal{X}$ that, given a problem instance ω can return a solution $\pi(\omega) \in \mathcal{X}$ that is either feasible or near-optimal depending on the nature of the problem. In addition, we assume that, given a solution constraint θ, the planner π will return $\pi(\omega, \theta) \in \mathcal{X}$,which is a plan subject to the solution constraint θ; in general this solution will not be optimal (so in general $J(\omega, \pi(\omega)) > J(\omega, \pi(\omega, \theta))$), unless θ was perfectly suited to the problem instance ω, but constraining the plan to satisfy θ will make it significantly more efficient to compute. With a slight abuse of notation, we will denote $J(\omega, \theta) = J(\omega, \pi(\omega, \theta))$, the evaluation of the solution constraint.

Let $\Theta = \{\theta_1, \ldots, \theta_m\}$ be a set of samples from the space of solution constraints. Now, we formulate our problem as follows: given a "training set" of example problem instances $\omega_1, \ldots, \omega_n$ sampled identically and independently from $P(\omega)$, a discrete set of solution constraints $\hat{\Theta}$, and the score function $J(\cdot, \cdot)$, generate a high-scoring solution to the "test" problem instance ω_{n+1}.

An interesting problem that we do not explicitly address in this paper is how to select the subset of decision variables for specifying constraints θ. In this paper, we manually choose a subset of decision variables as solution constraints, and take the simple approach of solving the training problem instances and then extracting the θ

values corresponding to these chosen constraints. The details of constructing Θ are provided in Algorithm 9.

7.2.1 *Black-box function optimization with experience*

Instead of designing a problem-dependent representation for problem instances, we represent a problem instance with a vector of scores of solution constraints Θ, where

$$\Phi(\omega) = [J(\omega, \theta_1), \cdots, J(\omega, \theta_m)]$$

$\Phi(\omega)$ here is a random vector that maps a sample from the sample space of problem instances to $\mathbb{R}^m$. Using this representation, our training data constructed from n problem instances can be represented with a $n \times m$ matrix

$$\mathbf{D} = \begin{bmatrix} \Phi(\omega_1) \\ \Phi(\omega_2) \\ \vdots \\ \Phi(\omega_n) \end{bmatrix}$$

that we call the *score matrix*. Now, given a new problem instance, ω, our goal is to take advantage of one or more solution constraints in Θ to find a high scoring plan without evaluating all of solution constraints in Θ. To do this, we will develop a procedure that evaluates $J(\omega, \theta)$ by computing a plan $\pi(\omega, \theta)$ for $k << m$ values of θ.

We begin by making use of the intuition that some solution constraints (via the plans they generate) are inherently more useful than others, independent of the problem instance. This leads to a naive score-space approach, STATIC, that tries solution constraints in Θ in a static order according to the empirical mean scores in the $1 \times m$ vector computed by

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{D}_i \tag{7.2.1}$$

where i indicates the row of the score matrix, and then returns the highest scoring plan obtained from trying the top k solution constraints.

This simple approach does not take advantage of the fact that there are correlations among the scores of solution constraints across problem instances; that is, the score of a solution constraint that has been already tried on this problem instance can inform us about the scores of other untried but correlated solution constraints. In order to exploit correlation, we assume that the random vector Φ is distributed according to a multivariate Gaussian distribution, $\mathcal{N}(\mu, \Sigma)$.

Now the score matrix is used to estimate the parameters of the prior distribution of Φ, $\hat{\mu}$ and $\hat{\Sigma}$, where $\hat{\mu}$ is defined in equation 7.2.1 and

$$\hat{\Sigma} = \frac{1}{n-1}\sum_{i=1}^{n}(\mathbf{D}_i - \hat{\mu})^T(\mathbf{D}_i - \hat{\mu}) \tag{7.2.2}$$

is a $m \times m$ covariance matrix. This prior distribution is updated given evidence about a new problem instance, in the form of score values. Algorithm 8 contains detailed pseudo-code for an algorithm based on these ideas, called BOX, which stands for Blackbox Optimization with eXperience. It takes as input: ω_{n+1}, the "test" planning problem instance; ζ, a constant governing the magnitude of the exploration; k, the number of solution constraints to evaluate; Θ, the set of solution constraints in the training set; $\hat{\mu}$ and $\hat{\Sigma}$, the parameters for the prior distribution of $\Phi(\omega_{n+1})$; J, the scoring function; and π, the planner.

The algorithm first estimates the parameters of the prior distribution. Then, it iterates over solution constraints: first it selects a solution constraint, then it uses the chosen constraint to construct a new plan, and the score of that plan combined with the prior computed from $\mathbf{D}$ is used to determine the next solution constraint to evaluate.

We will use Θ_t to denote the constraints that have been tried up to time t, $\bar{\Theta}_t = \Theta \setminus \Theta_t$ to denote the ones not tried, $\theta^{(t)}$ to denote the index of the solution constraint chosen at time t, $x^{(t)}$ to denote the associated plan, $J^{(t)}$ to denote the score of that plan on the given problem instance, and $J^{1:t}$ and $J^{\overline{1:t}}$ to denote the scores of tried and untried solution constraints up to time t, respectively.

We will use this constraint notation to refer to corresponding rows and columns in the mean vector and empirical covariance matrix. For instance, at time t, we can rearrange the covariance matrix $\hat{\Sigma}$ as

$$\begin{bmatrix} \hat{\Sigma}_{\bar{\Theta}_t,\bar{\Theta}_t} & \hat{\Sigma}_{\bar{\Theta}_t,\Theta_t} \\ \hat{\Sigma}_{\Theta_t,\bar{\Theta}_t} & \hat{\Sigma}_{\Theta_t,\Theta_t} \end{bmatrix}$$

where the subscript represents a set of rows and columns of the matrix $\hat{\Sigma}$. This way, the top-left block matrix is the covariance among untried solution constraints, the top-right and bottom-left represent covariance among tried and untried solution constraints, and the bottom-right represents the covariance among the tried solution constraints.

In line 1 of Algorithm 8, we first estimate the prior distribution of the score function for a new problem instance ω_{n+1}. For the consistency of notations we assume $\hat{\mu}^{(0)} = \hat{\mu}$ and $\hat{\Sigma}^{(0)} = \hat{\Sigma}$. Line 2 selects the next solution constraint to try based on the principle of *optimism in the face of uncertainty*, by selecting the one with the maximum upper confidence bound (UCB). The next three lines generate a plan

Algorithm 8 BOX($\omega_{n+1}, C, k, \Theta, \mathbf{D}, \mathbf{J}, ß$)

Compute $\hat{\mu}^{(0)}$ and $\hat{\Sigma}^{(0)}$ according to Eqns 7.2.1 and 7.2.2
for $t = 1$ **to** k
  $\theta^{(t)} = \arg\max_{i \in \bar{\Theta}_t} \hat{\mu}_i^{(t-1)} + \zeta \cdot \sqrt{\hat{\Sigma}_{ii}^{(t-1)}}$ // *i^{th} entry and i^{th} diagonal entry*
  $x^{(t)} = \pi(\omega_{n+1}, \theta^{(t)})$
  $J^{(t)} = J(\omega_{n+1}, x^{(t)})$
  Compute $\hat{\mu}^{(t)}$ and $\hat{\Sigma}^{(t)}$ using eqn. 7.2.3
end for
$t^* = \arg\max_{t \in \{1, \cdots, k\}} J^{(t)}$
return $x^{(t^*)}$

Algorithm 9 GenerateTrainingData($n, \pi, J, [\omega_1, \cdots, \omega_n]$)

for ω **in** $[\omega_1, \cdots, \omega_n]$
  $x_i = \pi(\omega)$ // *repeat to get multiple solutions if desired*
  $\theta_i = extractConstraint(x_i)$ // elements of Θ
end for
for ω **in** $[\omega_1, \cdots, \omega_n]$
  for θ **in** Θ
    $J(\omega, \theta) = J(\omega, \pi(\omega, \theta))$ // *elements of* $\mathbf{D}$
  end for
end for
return $\mathbf{D}, \Theta$

using the chosen solution constraint, and then evaluate it. At iteration t, given the experience of trying $\Theta_t = [\theta^{(1)}, \cdots, \theta^{(t)}]$ and getting scores $J^{1:t} := [J^{(1)}, \cdots, J^{(t)}]$, our posterior on the scores of the untried solution constraints, denoted $J^{\overline{1:t}}$, is

$$J^{\overline{1:t}} | J^{1:t} \sim \mathcal{N}(\hat{\mu}^{(t)}_{\bar{\Theta}_t}, \hat{\Sigma}^{(t)}_{\bar{\Theta}_t, \bar{\Theta}_t})$$

where

$$\begin{aligned} \hat{\mu}^{(t)}_{\bar{\Theta}_t} &= \hat{\mu}_{\bar{\Theta}_t} + \hat{\Sigma}_{\bar{\Theta}_t, \Theta_t} (\hat{\Sigma}_{\Theta_t, \Theta_t})^{-1} (J^{1:t} - \hat{\mu}_{\Theta_t}) \\ \hat{\Sigma}^{(t)}_{\bar{\Theta}_t} &= \hat{\Sigma}_{\bar{\Theta}_t, \bar{\Theta}_t} - \hat{\Sigma}_{\bar{\Theta}_t, \Theta_t} (\hat{\Sigma}_{\Theta_t, \Theta_t})^{-1} \hat{\Sigma}_{\Theta_t, \bar{\Theta}_t} \end{aligned} \tag{7.2.3}$$

The constant ζ governs the size of the confidence interval on the scores. We show how the constant can be set through theoretical analysis of regret bounds in the next section. The number of evaluations k should be chosen based on the desired trade-off between computation time and solution quality.

In order to create the score matrix $\mathbf{D}$ and solution constraints Θ, we run Algorithm 9. This algorithm takes as input n, the number of training problem instances, π a planning algorithm that can solve problem instances ω_{n+1} without additional constraints, J, the scoring function for a plan, and $[\omega_1, \cdots, \omega_n]$, a set of training sample problem instances drawn iid from $P(\omega)$. For each problem instance, a solution is generated using π, and a constraint is extracted from

the solution and added to set Θ. The process of extracting constraints is domain-dependent; several examples are illustrated in the experiment section. Each new solution constraint is used to generate a solution $\pi(\omega_{n+1}, \theta)$ whose score $J(\omega_{n+1}, \theta)$ is stored in the **D** matrix.

7.2.2 *Illustrative examples*

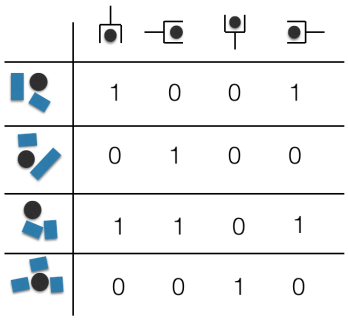
In Figure 7.1a, we show an example of a score matrix **D** obtained by running Algorithm 9. In this figure, the target object is represented with a black circle, and the blue rectangular objects represent obstacles. We have four training problem instances, shown across the rows of the score matrix, and the four constraints across the columns.

For illustrative purposes, we assume a simple binary score function, which outputs one if a constraint is feasible for the given problem instance, and zero otherwise. For example, for the first training problem instance, the top-approaching direction is feasible because there is no obstacle blocking the object in that direction, whereas the left-approaching direction is blocked with an obstacle. For such binary score function, other prior assumption on the target function, such as Bernoulli distribution, might be more suited; however, we will in general consider score functions that take on real numbers, as we will demonstrate in our experiment section.

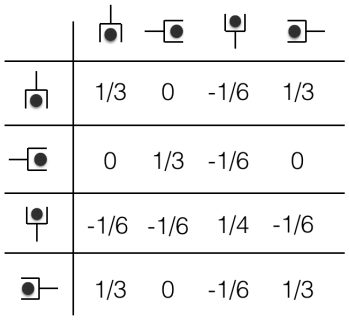
We now provide concrete examples of running BOX on some simple examples. Suppose that our constraint is a set of four different grasps, defined by approach vectors, from the top, left, bottom, and right. Our problem is to plan a collision-free path to grasp a target object. The planner is constrained to use the chosen grasp approach direction to grasp the target object. We arbitrarily choose $\zeta = 1.96$ to ensure 95% confidence interval in our experiments, but it could be tuned via cross validation for better performance.

In Figure 7.1b, we show the result of computing the covariance matrix $\hat{\Sigma}$ using **D** and equation 7.2.2. In order to understand BOX more thoroughly, we note some salient correlation information in $\hat{\Sigma}$. First, the top-approaching constraint is positively correlated with the right-approaching direction, whereas it is negatively correlated with the bottom-approaching direction. So, in a new problem instance, if we find that the top-approaching constraint fails, then it will increase the UCB value of the score for the bottom-approaching direction while decreasing the UCB value of the right-approaching direction.

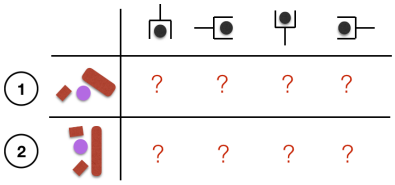
We will illustrate these types of behaviors of using two problem instances shown in Figure 7.1c, where the task is to plan a collision-free path to grab the circular magenta object, which is occluded by red obstacles. Clearly, for the problem shown in the first row, the only constraint that would work is the bottom-approaching direc-



(a) Score matrix, **D**



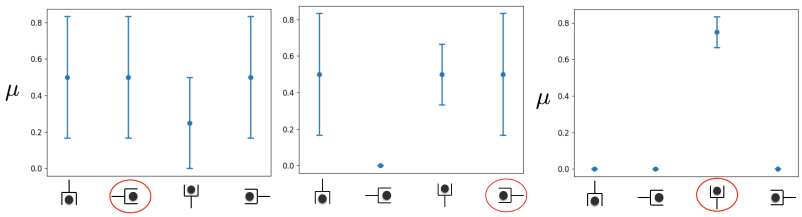
(b) Covariance matrix, $\hat{\Sigma}$



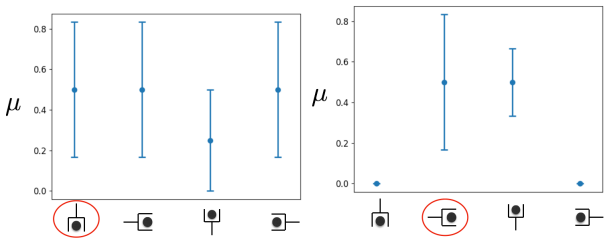
(c) Two new problem instances

Figure 7.1: Score and covariance matrices for running BOX, and two new problem instances

tion. For the second problem instance, the left-approaching direction would be the only feasible constraint.



(a) Evolution of μ and UCBs for the first problem instance



(b) Evolution of μ and UCBs for the second problem instance

Figure 7.2: Illustration of how UCBs change as BOX uses constraints in two different problem instances

Figure 7.2 shows the evolution of UCB values,

$$\hat{\mu}_i^{(t)} + \zeta \cdot \sqrt{\hat{\Sigma}_{ii}^{(t)}}$$

of the different constraints we denoted with i, as BOX suggests constraints and receives feedback from the planner and the environment. For example, from the score matrix shown in Figure 7.1a, we can see that the average values of the scores for top-,left-, and right-approaching directions are 0.5. The elements in the diagonal of the covariance matrix, which are variances of the scores of different constraints, are approximately 0.33 for these three constraints. These give approximate UCB values of 0.83 for these constraints.

The first plot in Figure 7.2 (a) shows the UCB values when $t = 1$. There is a tie among UCB values of the first, second, and the last constraints, so we randomly break the tie and select the second constraint, marked with the red circle. After trying to plan a path with this constraint, we see that it is infeasible. The second plot shows the updated UCB values after observing that the second constraint has a score of zero, using Eqn 7.2.3. We see that the UCB values of the first and the last constraint remained unchanged, while the third constraint has increased. This is because the second constraint has zero correlation with the first and last constraints, but has negative correlation with the third constraint, as shown in the Figure 7.1b. After this, the last and the first constraints have the same UCB values, and we again randomly break the tie; unfortunately, we chose the fourth one, but from this we can update our UCBs such that all other constraints except the third one are infeasible.

This example is a particularly hard for BOX, because from our prior experience the third constraint was feasible only $1/4$ of the time with the lowest variance. Therefore, our belief about its score was quite low. We now consider the second problem instance in Figure 7.1c, which is more favorable. Figure 7.2 (b) shows the evolution of UCB values. At $t = 1$, shown in the first plot, we randomly break the tie, and chose the first constraint. After observing this is infeasible, at $t = 2$, the UCB value of the fourth constraint, which has positive correlation with the first constraint, also reduces to almost 0. The third constraint was negatively correlated with the first constraint, so its UCB value increased; however, the second constraint, which is the correct constraint for this problem instance, has zero correlation with the first constraint, and its UCB value is higher than the third one even after the update. Hence BOX ends up choosing the feasible constraint after just a single mistake.

7.3 Constructing a minimal set

In this section, we propose an algorithm for BOX which tries to reduces the cardinality of Θ while maintaining important properties, such as the probability that the set will contain a constraint that is applicable to a new problem instance.

We begin with the problem formulation. We wish that, for all problem instances, there is at least one solution in our set. We define a constraint to be *feasible* for a problem instance if we can find a solution that satisfies it. We will say a constraint *covers* a problem instance if it is feasible for a problem instance. Given a set of constraints, Θ, we are interested in a minimum cardinality subset of the original constraint set that covers all the problem instances in the training data. We will call such subset a *minimal set*, and denote it with Θ_{min}.

Out of all the minimal sets, we are interested in those whose probability of success is maximized, so that the set can be applied to a wide range of problem instances. The probability of success is measured by

$$P(\Theta_{min} \text{ succeeds on } \omega) = 1 - \prod_{\theta \in \Theta_{min}}^{k} (1 - p_\theta)$$

where k is the cardinality of the minimal set, and p_θ is the probability of constraint θ_i in the minimal set being feasible. Notice that p_θ can be approximated by empirical counts of successes divided by the number of problem instances from our training data. We will call a minimal set whose probability of success is maximum a *maximally successful minimal set*.

Out of all the maximally successful minimal sets, we are interested in those that give maximum information on the values of the scores of other constraints, in order to minimize the number of evaluations. First note that the differential entropy h of the multivariate Gaussian distribution $\mathcal{N}(\mu, \Sigma_{\Theta_{min}})$ over a random vector Θ_{min} is defined by

$$h(\Theta_{min}) = \frac{|\Theta_{min}|}{2}\Big[1 + \log(2\pi)\Big] + \frac{1}{2}\log\det\Sigma_{\Theta_{min}}$$

where n is $|\Theta_{min}|$ and $|\Sigma_{\Theta_{min}}|$ is a determinant of the covariance matrix $\Sigma_{\Theta_{min}}$. Now, using this, we can define the *gain function*, g, which characterizes the information gain for scores of other constraints given an evaluation of a constraint θ_i

$$\begin{aligned} g(\Sigma_{\Theta_{min}}, \theta_i) &= h(\Theta_{min}) - h(\Theta_{min}|\theta_i) \\ &= \log(|\Sigma_{\Theta_{min}}|) - \log(|\Sigma_{\Theta_{min}|\theta_i}|) \end{aligned}$$

where $|\Sigma_{\Theta_{min}}|$ is the determinant of the covariance matrix of the minimal set Θ_{min}, and $|\Sigma_{\Theta_{min}|\theta_i}|$ is the determinant of the covariance matrix of the minimal set after evaluating the constraint θ_i in Θ_{min}. We will call a maximally successful minimal set that maximizes the gain function an *optimally minimal set (OMS)* and denote it with Θ^*_{min}.

We now formulate the optimally minimal set construction problem as follows. Given an original constraint set Θ, construct a minimal set Θ_{min} that maximizes the function

$$c(\Theta_{min}) = \sum_{\theta \in \Theta_{min}} p_\theta + \lambda \cdot g(\Sigma_{\Theta_{min}}, \theta)$$

The first term is responsible for maximizing the probability of success for Θ_{min}, and the second term is responsible for maximizing the sum of information gain of each constraint in Θ_{min}. Now the optimally minimal set is defined as

$$\Theta^*_{min} = \arg\max_{\Theta_{min} \in 2^\Theta} c(\Theta_{min})$$

Clearly, constructing Θ^*_{min} is an NP-complete problem[1] This motivates us to devise a greedy approach that approximately optimizes the function c, as described in Algorithm 10.

[1] We can make a polynomial-time reduction to the minimum set-cover problem.

This algorithm takes as inputs: the experience matrix $\mathbf{D}$, the original constraint set Θ, the approximated parameters of a distribution of score vectors $\hat{\mu}$ and $\hat{\Sigma}$, the number of problem instances n and the number of constraints of the original set m, and outputs an approximation of an OMS, L.

It operates by progressively constructing a list of constraints, L, using the gain function and probability of success of a constraint, which is measured by the mean score function μ.

Algorithm 10 ConstructOMS($\mathbf{D}, \Theta, \hat{\mu}, \hat{\Sigma}, n, m$)

$\theta_{next} = \arg\max_{\theta \in \Theta} \mu_\theta$
$L = CreateList(\theta_{next})$
$C_{\max} = nCoveredBy(L, \{1, \cdots, n\})$
while $C_{\max} \neq n$
  $U = \{1, \cdots, n\}) \setminus CoveredBy(L, \{1, \cdots, n\})$
  $\Theta_{\text{cand}} = \{i | \max_{i \in 1, \cdots, m} len(CoveredBy(\{L, i\}, U))\}$
  $\Theta_{\text{cand}} = \{i | i = \arg\max_{i \in I_{\text{cand}}} \mu_i\}$
  $\theta_{next} = \arg\max_{\theta \in \Theta_{\text{cand}}} g(\Sigma_L, \theta)\}$
  $L = \{L, \theta_{next}\}$
  $C_{\max} = nCoveredBy(L, \{1, \cdots, n\})$
end while
return L

It begins by first adding the index of the constraint that has the maximum mean score value. Then, it checks the number of problem instances covered by the current list of constraints L, using the helper function $nCoveredBy$. It then computes the uncovered problem instance indices, U. The algorithm then computes the set of next candidate constraint to add to L, Θ_{cand}, by taking the constraint that maximally covers the currently uncovered indices U. This maximal coverage step is to ensure that we are minimizing the cardinality of L.

From this set, the algorithm updates Θ_{cand} by considering only those constraints whose mean score is the maximum; it is still a set, since there may be a tie in mean scores. From this set, the algorithm chooses the next constraint to add to L, by taking the one that has the maximum gain function value. We update the number of problem instances covered, and repeat until we cover all the problem instances. Lastly the algorithm returns L, the set of constraints that approximates the OMS.

7.4 Experiments

We demonstrate the effectiveness of score-space algorithms STATIC and BOX in four robotic planning domains: grasp-selection, grasp-and-base-selection, pick-and-place, and conveyor-belt unloading. Each of these domains has several decision variables and different types of solution constraints. For all the problems, a problem instance varies in the sizes of the objects being manipulated, and the poses of obstacles.

In each of these domains, $\pi(\cdot, \theta)$ finds values of decision variables that are not specified in θ. For example, for the grasp-and-base-selection domain $\pi(\cdot, \theta)$ consists of an IK solver and a path planner, and θ specifies values such as a robot base pose or a grasp for picking an object that is relevant for achieving a goal. To implement RAW-

PLANNER, $\pi(\omega)$, we first uniformly sample θ from their original space, such as $\mathbb{R}^2$ for robot base pose, instead of from Θ, and then use $\pi(\omega, \theta)$ with the sampled constraints.

We are interested in both running time and solution quality. We compare score-space algorithms, STATIC and BOX, with RAWPLANNER as well as two other methods that generate plans by selecting a subset of size k of the solution constraints from Θ and return the highest scoring one. As previously mentioned, STATIC sequentially selects constraints based on their average score values, without considering their correlation information. RAND selects k of the θ_i values at random from $\hat{\Theta}$; DOO is an adaptation of DOO [91] to optimization of a black-box function over a discrete set, which is Θ in our case. Like BOX, it alternates between evaluating θ_j and constructing upper bounds on the unevaluated θ_i for k rounds. It assumes that the function is Lipschitz continuous with constant λ, and uses the bound $J^\omega(\theta_i) \leq J^\omega(\theta_j) + \lambda \cdot l(\theta_i, \theta_j)$ for some semi-metric l, $\lambda \in \mathbb{R}$. We use the Euclidean metric for l, and $\lambda = 1$.

To show that score-space algorithms can work with different planners, we show results using two different planners: bidirectional RRT with path smoothing implemented in OpenRAVE [23], seeded with a fixed randomization seed value, and Trajopt [108]. In the pick-and-place and conveyor-belt unloading domains, where there is a narrow-passage path planning problem, Trajopt cannot find feasible paths without being given a good initial solution, so we omit it.

In each domain, we report the results using two plots, the first showing the time to find the first feasible solution and the second showing how the solution quality improves as the algorithms are given more time. Each data point on each plot is produced using leave-one-out cross-validation. That is, given a total data set of n problem instances and associated solutions, we report the average of n experiments, in which $n - 1$ of the instances are used as training data and the remaining one is used as a test problem instance.

Grasp-selection, grasp-and-base-selection, and pick-and-place problems are satisficing problems in which we are mainly interested in finding a feasible solution. So, a binary score function that specifies the feasibility of a given plan would be sufficient to use BOX. However, in many problems, we want to find a low-cost plan, rather than just a feasible one, using BOX's ability to seek optimal solutions from its library of constraints.

To do this, we design a score function that measures the trajectory length for a feasible plan, and that assigns a large cost if the plan is infeasible in the given problem instance. So, given a plan $\pi(\omega) = (q_1, \cdots, q_l)$ where q_i denotes a configuration of the robot, our score

function is

$$J^{\omega}(x) = \begin{cases} -\sum_{i=1}^{l-1} ||q_{i+1} - q_i|| \text{ if } x \text{ feasible in } \omega \\ d, \text{ otherwise} \end{cases} \tag{7.4.1}$$

where $||\cdot||$ denotes a suitable distance metric between configurations and $d = min(\mathbf{D}) - mean(\mathbf{D})$. This is our strategy for finding a domain dependent minimum score for failing to solve a problem. Conveyor-belt unloading domain, on the other hand, is not a satisficing problem: we are interested in maximizing the number of objects that a robot packs into a tight room. The Conveyor-belt unloading domain, is not a satisficing problem: we are interested in maximizing the number of objects that the robot packs into a tight room. Therefore, naturally, our score function is defined as the number of objects packed by a plan $\pi(\omega)$.

7.4.1 Grasp-selection domain

Our first problem domain is to find an arm motion to grasp an object that lies randomly either on a desk or a bookshelf, where there also are randomly placed obstacles. Neither the grasp of the object nor the final configuration of the robot is specified, so the complete planning problem includes choosing a grasp, performing inverse kinematics to find a pre-grasp configuration for the chosen grasp, and then solving a motion planning problem to the computed pre-grasp configuration.

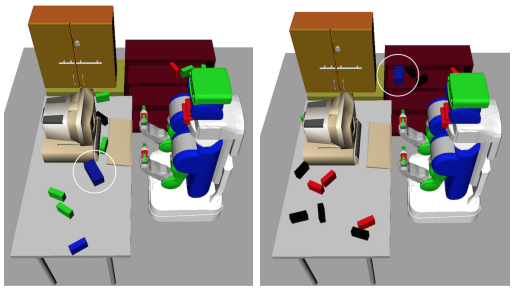
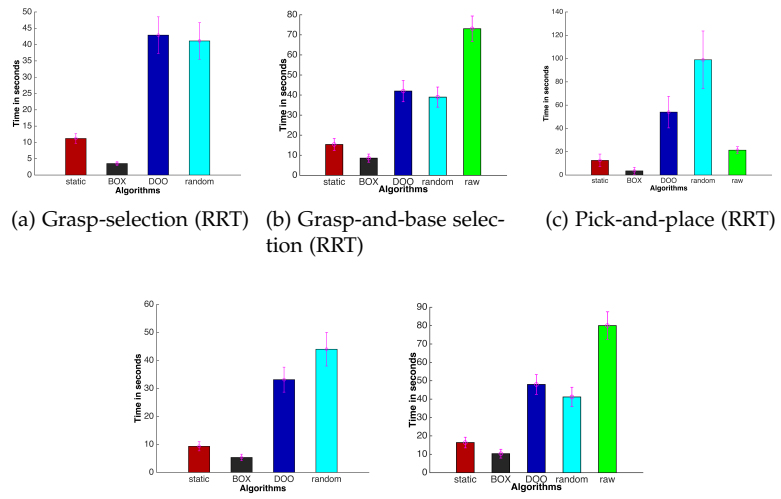


Figure 7.3: Two instances of the grasp selection domain. The arrangement and number of obstacles vary randomly across different planning problem instances. The objective is to find an arm trajectory to a pre-grasp pose for the blue box, marked with a circle, whose pose is randomly determined in each problem instance

A planning problem instance for this domain is defined by an arrangement of several objects on a table. Figure 7.3 shows two instances of this problem, which are also part of the training data. There are up to 20 obstacles in each problem instance. The robot's active degrees of freedom (DOF) are its left and right arms, each of which has 7 DOFs, and torso height with 1 DOF, for a total of 15 DOF. Θ consists of 81 different grasps per each arm, computed using OpenRAVE's grasp model function. Θ would be all possible grasps for an object. Notice that since our search space for solution constraints is discrete, RAWPLANNER is equivalent to RAND.

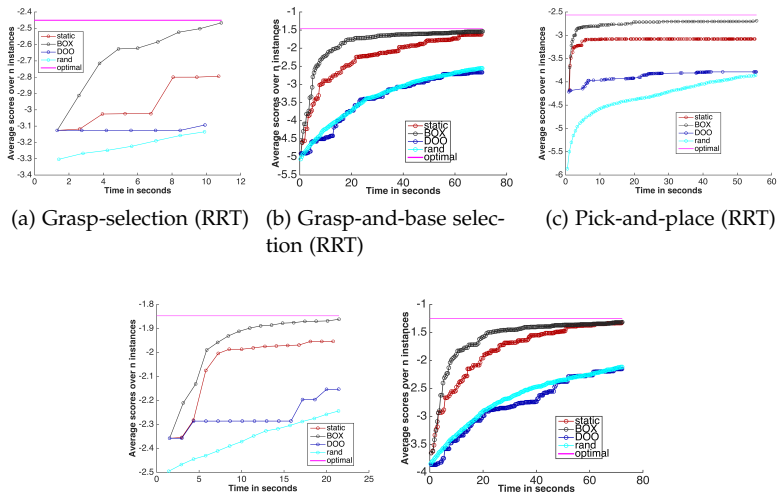
Given a solution constraint θ, which is a grasp (pose of robot hand with respect to the object) and an arm to pick the object with, it remains for $\pi(\omega, \theta)$ to find an IK solution and motion plan, which can be expensive, but predicting a good grasp makes the overall process much more efficient. The trajectory of the arm to the pre-grasp configuration, with the base fixed, is scored according to eqn. 7.4.1, with a score of d assigned to problem instances and constraints for which no solution is found within a fixed amount of computation.



(a) Grasp-selection (RRT) (b) Grasp-and-base selection (RRT) (c) Pick-and-place (RRT)

(d) Grasp-selection (TrajOpt) (e) Grasp-and-base selection (TrajOpt)

Figure 7.4: LOOCV estimate of time to find first feasible solution, for each method in different domains. Whiskers indicate 95% confidence interval on mean. The top row uses RRT and the bottom row uses TrajOpt



(a) Grasp-selection (RRT) (b) Grasp-and-base selection (RRT) (c) Pick-and-place (RRT)

(d) Grasp-selection (TrajOpt) (e) Grasp-and-base (TrajOpt)

Figure 7.5: Solution score versus run time for different algorithms in various domains. The time axis goes until the first algorithm reaches 95% of the optimal score, marked with magenta. This optimal line is obtained by taking the θ from Θ that achieved maximum score for each problem instance. The top row uses RRT and the bottom row uses TrajOpt.

The experiments were run on a data set of 1800 problem instances. Figure 7.4 (a) compares the time required by each method to find

the first feasible plan with RRT as the path planner, and Figure 7.4 (d) compares the time with TrajOpt as the path planner. In both of the plots, we can observe that the score-space algorithms STATIC and BOX outperform all other algorithms in terms of finding a good solution with a given amount of time. BOX performs about three times faster than STATIC, showing the advantage of using the correlation information. Compared to DOO and RAND, BOX is more than nine times faster. DOO does only slightly better than RAND, which illustrates that in the space of grasps, the Euclidean metric is not effective.

Figure 7.5 (a) compares the solution quality vs time when RRT is used; figure 7.5 (d) compares the same quantities when TrajOpt is used. Here, the score-space algorithms again outperform the other algorithms, with BOX outperforming STATIC.

7.4.2 *Grasp-and-base selection domain*

In this experiment, we evaluate how the score-space algorithms perform when we construct the matrix Θ by sampling from a continuous space. Here, the robot needs to search for a base configuration, a left arm pre-grasp configuration, and a feasible path between these configurations to pick an object.

A planning problem instance is again defined by the arrangement of objects. Figure 7.6 shows three different training problem instances. We have 20 rectangular boxes as obstacles, all resting on the two tables both of which remain fixed in all instances. For each of the red obstacles and the blue target object, the (x, y) location and orientation in the plane of the table are randomly chosen subject to the constraint that they are not in collision. It is possible that the problem instances will be infeasible (the target object is too occluded or kinematically unreachable by the robot). The robot always starts at the same initial configuration.

The robot's active DOFs include its base configuration, torso height, and left arm configuration, for a total of 11 DOF. A solution constraint for this domain consists of the robot base configuration to pick the target object, (x, y, ψ), where ψ is an orientation of the robot, as well as one of 81 grasps from the previous section.

The solution constraints in this case are the grasp g, and the base configuration k. Given a planning problem instance with no constraints, the RAWPLANNER for this domain performs three sampling procedures, using a uniform random sampler, backtracking among them as needed to find a feasible solution:

1. Sample a base configuration, $k = (x, y, \psi)$, from a circular region of free configuration space, with radius equal to the length of the robot's arm, centered at the location of the object.

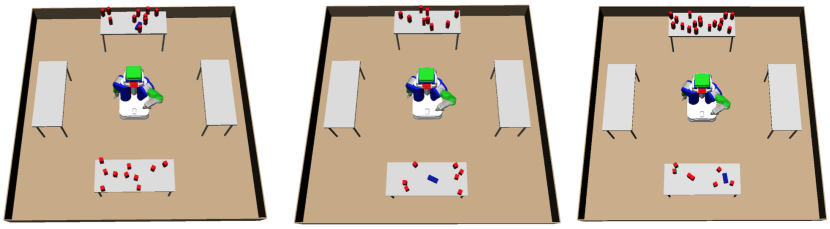


Figure 7.6: Three instances in which the robot must select base configuration, grasp, and paths, to pick the target object (blue). The poses of the objects are randomly varied between instances.

2. Sample, without replacement, from the 81 grasps until a legal one is found, i.e. one for which there is an IK solution in which the robot is holding the target object using that grasp in a collision-free configuration.

3. Use bidirectional RRT or TrajOpt to find a path for the arm and torso between the configurations found in steps 1 and 2.

We assume that the configuration from step 1 is reachable from the initial configuration. To extract a solution constraint from the resulting plan, we simply return the base configuration from step 1 and the grasp from step 2.

Unlike RAWPLANNER, which has to search for k and g, $\pi(\omega, \theta)$ simply solves the inverse kinematics and motion planning problems as in the previous example. The trajectory of the arm to the pre-grasp configuration, with the base fixed according to the constraint, is scored according to equation 7.4.1, with a score of d assigned to problem instances and constraints for which no feasible solution is found within a fixed number of iterations of the RRT. The experiments were run on a data set of 1000 problem instances. The set Θ contained 1000 pairs of grasp and robot base configuration, each extracted from a different problem instance.

Figures 7.4 (b) and 7.4 (e) show the time required by each method to find the first feasible plan, using RRT and TrajOpt as the planner. The score-space algorithms perform orders of magnitude better than the other algorithms, with BOX again outperforming STATIC. DOO and RAND do provide some advantage by using previously stored solution constraints compared to RAWPLANNER. RAWPLANNER has to sample in the continuous space of base configurations and check whether an IK solution and feasible path exist by running IK and path planning. This causes a significant increase in time to find a solution.

Figure 7.5 (b) compares the solution quality vs time when RRT is used and Figure 7.5 (e) compares the same quantities when TrajOpt is used. Again, the score-space approaches outperform all other methods, with BOX performing better than STATIC, by using the correlation

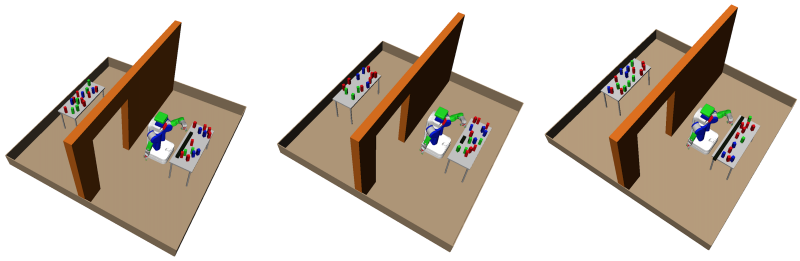


Figure 7.7: Three problem instances from the pick-and-place domain. The robot's initial configuration and the black object's initial pose are fixed across different planning scenes, but other objects' poses and the black object's length vary.

information from the score space. DOO and RAND perform similarly, mainly because that simple Euclidean distance is not effective for the hybrid space of base configuration and grasps.

7.4.3 *Pick-and-place domain*

In this experiment, with problem instances as shown in figure 7.7, we introduce solution constraints involving the placements of objects. Here, the robot needs to pick a large object (shown in black) up off of a table in one room, carry it through a narrow door, and place the object on a table. The initial poses of the target object and the robot are fixed, but problem instances vary in terms of the initial poses of 28 obstacles on both the starting and final tables, which are chosen uniformly at random on the table-tops subject to non-collision constraints, and the length of the target object, which is chosen at random from three fixed sizes.

The robot's active DOFs are the same 11 DOFs as in the previous problem domain. The solution constraints in this domain consists of grasp g to pick the object, o, the placement pose of the object on the table in the back room, k_b, the pre-placement base configuration of the robot for placing the object at pose o, and k_{sg}, the subgoal base configuration for path planning through the narrow passage to k_b from the initial configuration.

Given a problem instance with no constraint, RAWPLANNER performs six sampling procedures, similarly to the previous domain, using a uniform random sampler:

1. Sample a grasp g, without replacement, from the 81 grasps until a legal one is found.
2. Plan a path for the arm and torso to the pre-grasp configuration found in step 1. If none is found, choose another grasp.
3. Sample a collision-free object pose o on the table in the other room.
4. Sample k_b, the pre-placement base configuration, from a circular region of free configuration space around o, with radius equal to

the length of the robot's arms. If none is found, go back to the previous step.

5. Plan a path from the initial configuration to k_b. If none is found, go back to the previous step.

6. Plan a path from k_b to a place configuration for putting the object down at o. If none is found, go back to the previous step.

In contrast, given a solution constraint, $\pi(\omega, \theta)$ simply solves for inverse kinematics and path plans.

The experiments were run on a data set of 1500 problem instances, with 500 instances per rod size. The set $\hat{\Theta}$ contained 1000 tuples of solution-constraint values, obtained first by running Algorithm 9 and then randomly subsampling them to reduce the size to 1000.

Figure 7.4 (c) shows the time required by each method to find the first feasible plan. Again, the score-space algorithms significantly outperform the other algorithms and BOX outperforms STATIC. One noticeable difference between this domain and the previous two is that an ineffective solution constraint takes a long time to evaluate, because computing a path plan or IK solution for an infeasible constraint is computationally expensive. This is evident in performance of RAND and DOO which perform worse than RAWPLANNER as they tend to choose solution constraints that are infeasible and expensive to evaluate.

Figure 7.5 (c) shows the average solution score as a function of computation time. The graphs show a similar trend as in the previous experiments, with score-space algorithms outperfoming the other algorithms, and BOX performing better than STATIC. The fact that this domain requires a significant amount of time to try an ineffective solution constraint is again evident in DOO's plot, where consecutive dots have a large gap between them. BOX and STATIC are able to avoid this problem by exploiting the score-space information.

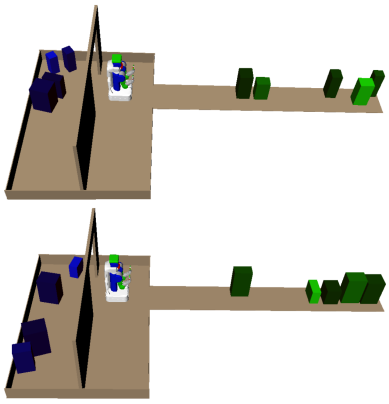


Figure 7.8: Two instances of the conveyor belt domain

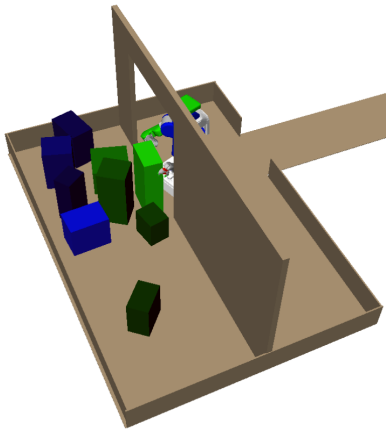


Figure 7.9: A solution for the conveyor belt domain

7.4.4 *Conveyor belt unloading domain*

In this domain, the robot has to manipulate box-shaped objects using two-handed grasps. The robot's objective is to receive five box-shaped objects with various sizes from a conveyor belt and pack them into a room with a narrow entrance. A problem instance is defined by the shapes and order of the objects that arrive on the conveyor belt. Examples of problem instances are shown in Figure 7.8, and a solved problem instance is shown in Figure 7.9.

The robot must make a plan for handling all the boxes, including a grasp for each box, a placement for it in the room, and all of the robot trajectories. The initial base configuration is always fixed at the

conveyor belt. After it decides the object placement, which uniquely determines the robot base configuration, a call to an RRT is made to find, if possible, a collision-free path from its fixed initial configuration at the conveyor belt to the selected placement configuration. The robot cannot move an object once it has placed it.

The three previous problems involve an infinite branching factor, but a relatively shorter planning horizon than this problem: if we assume that a call to a motion planner is a "step" in our plan, since we are using it as a primitive planner, the grasp-selection domain has a horizon of one, the grasp-base-selection domain is a has a horizon of two (for base planning and arm planning for picking an object), and the pick-and-place domain has a horizon of three (for picking an object, moving robot's base, and then to place the held object). The conveyor-belt domain requires a horizon of ten, for picking and placing five objects in total.

Further, this domain is particularly challenging compared to the previous experiments for two reasons. First, the robot is operating in an environment with tight free-space, in which there may or may not be a collision-free path from one robot configuration to another. If the object placements are not carefully chosen, calls to the motion planner will be extremely expensive, either because they are infeasible and will have to run until a time-out is reached, or because the tolerances are tight and so even if the problem is feasible, it may run for a long time or time out. Second, they contain a large volume of "dead-end" states that require the task-level planner to backtrack. For example, if the planner greedily places early objects near the door, then it will eventually find that it is infeasible to place the rest of the objects and will have to backtrack to find different placements of those objects.

As mentioned, we have a significantly longer horizon planning problem than the previous domains. Therefore, we use graph-search with the sampled operators as our RAWPLANNER. It proceeds as follows:

1. Place the root node on the search agenda.

2. Pop the node from the agenda with the lowest heuristic value (estimated cost to reach the goal)

3. Expand the popped node by generating three operator instances by sampling their parameters, and add their successor states on the search agenda.

4. If the popped node is a root node, add it back to the queue after expansion

5. If at the current node we cannot sample any feasible operators, then we discard the node and continue with the next node in the agenda.

6. Go to step 2 and repeat until we arrive at a goal state.

We have two operators: *pick* and *place*. To sample parameters for the *pick* operation, the raw planner RAWPLANNER executes the following steps:

1. Sample a collision-free base configuration, (x, y, ψ), uniformly from a circular region of free configuration space, with radius equal to the length of the robot's arm, centered at the location of the object, using a uniform sampler.

2. With the base configuration fixed at (x, y, ψ), sample (d, h, γ), where d and h are in the range $[0.5, 1]$, and γ is in the range $[\frac{\pi}{4}, \pi]$, uniformly. If an inverse kinematics (IK) solution exists for both arms for this grasp, proceed to step 3, otherwise restart.

3. A linear path from the current arm configuration to the IK solution found in step 2 is planned.

At each stage, if a collision is detected, this means that the sampled parameters are infeasible, so sampling proceeds from the step 1

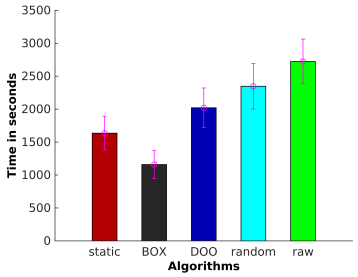
We assume that the conveyor belt drops objects into the same pose, and the robot can always reach them from its initial configuration near the conveyor belt, so we omit step 3. From a state in which the robot is holding an object, it can place it at a feasible location in a particular region. To sample parameters for *place*, RAWPLANNER executes following steps:

1. Sample a collision-free base configuration, (x, y, ψ), uniformly from a desired region R.

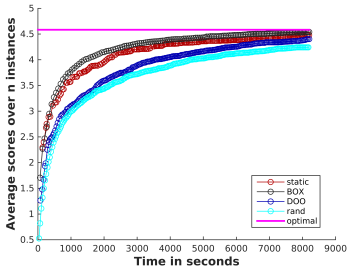
2. Use bidirectional RRT from the current robot base configuration to (x, y, ψ).

The experiments were run on a dataset of 468 problem instances. The solution constraints in this domain consists of the base configuration for the *pick* operator, and the base configuration for *place* operator. The size of Θ was 400, obtained by running Algorithm 9 on total of 468 problem instances and then randomly subsampling them to reduce the size to 400.

Figure 7.10a shows the time required by each method to pack four objects. Again, the score-space algorithms significantly outperform the other algorithms and BOX outperforms STATIC. In this domain, each evaluation of a constraint is significantly longer than in the previous domains, because trying a constraint involves calls to an RRT



(a) Time to pack four objects for the conveyor belt domain using RRT



(b) Solution score vs run time for the conveyor belt domain using RRT

Figure 7.10: LOOCV estimates of different performance metrics for the conveyor belt domain

up to ten times, some of which may be an infeasible motion planning problem. We also notice that DOO does not perform as badly as in the previous domains, because the constraints are defined on the base poses of robots, in which Euclidean distance is a reasonable metric.

Figure 7.10b shows the average solution score as a function of computation time. The graphs show a similar trend as in the previous experiments, with score-space algorithms outperfoming the other algorithms, and BOX performing better than STATIC. Compared to the previous domains, the difference between BOX and STATIC is much smaller. This is because in this domain, any strategy that packs objects inside first and then gradually towards the narrow entrance will tend to have high scores. Therefore, STATIC, which uses the mean of the scores, works reasonably well, although BOX generally does better.

7.4.5 *Experiments with optimally minimal set*

The purpose of this experiment is to verify our hypothesis that reducing the size of the constraint set reduces the time for matrix inversion involved in updating the mean and covariance matrix in BOX, as well as the time to find the first feasible solution. We test Algorithm 10 in the first three problems previously solved using BOX with full constraints, the grasp-and-base-selection domain, pick-and-place domain, and conveyor-belt domain, and provide comparisons. We omit the grasp-selection domain because it already has a small constraint set of size 162, compared to 1500,1000, and 500 for the other three domains.

Using Algorithm 10, we were able to reduce the constraint set size significantly. For the grasp-and-base-selection domain, the algorithm reduced it to 41 from 1500 for the pick-and-place domain it reduced the constraint set size to 69 from 1000, and for the conveyor-belt domain it reduced from 500 to 76.

Figure 7.11 shows the times to find a first-feasible solution for these domains using the reduced constraint set. As we can see, it reduces both planning time and BOX time, which confirms our hypothesis. In fact, in all of the domains, it reduced the BOX time, which mostly consists of covariance matrix inversion time, to almost zero. In terms of reduction in planning times, the reduction was approximately a factor of around 3 for the grasp-and-base-selection domain. The reduction is smaller in the pick-and-place and conveyor belt domains, although there is a notable reduction. The reason that the reduction is smaller in these two domains is because there generally is a smaller reduction in variance of scores of other constraints compared to the grasp-and-base-selection domain when one constraint is evaluated.

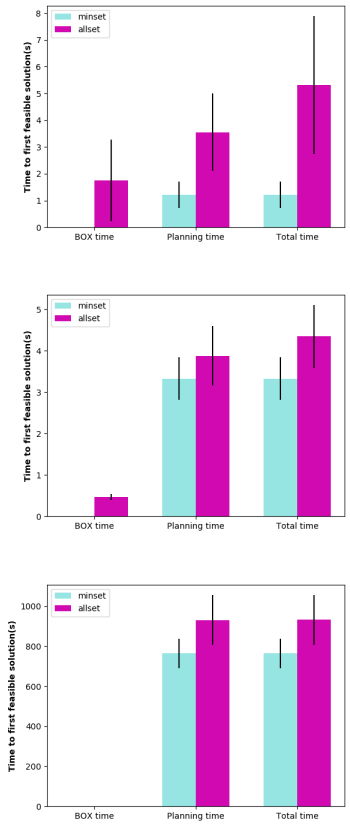


Figure 7.11: A comparison of time to find a first feasible solution for grasp-and-base-selection domain (left), pick-and-place domain (middle), and conveyor belt domain (right). *minset* refers to running BOX with the constraint set found by Algorithm 10, and *allset* refers to running BOX using the original set Θ. BOX time refers to time spent (mostly) inverting the covariance matrix, and planning time refers to time spent on planning using chosen constraints.

7.5 Discussion and future work

In this chapter, we proposed an algorithm for learning to guide a planner for task-and-motion planning problems by addressing three important questions: what to predict, how to represent a planning problem instance, and how to transfer planning knowledge from one instance to another.

In order to trade-off between the burden on the learning algorithm and the planner, we proposed to predict constraints on the planning process rather than a complete solution. To eliminate the bias and cumbersome feature design, we introduced a score-space representation, in which we construct a representation of a problem instance using a set of scores of plans that satisfy a pre-built discrete set of constraints on-line. To transfer knowledge, we proposed BOX, an algorithm that tries to both accurately construct a score-space representation of the given problem instance and choosing a constraint to try next from the given set.

As an extension to our original work, we also proposed an approach for reducing the constraint-set size. This is motivated by the fact that BOX requires inverting a matrix, which gets larger as the size of the constraint-set increases. This algorithm effectively reduced the constraint-set size, which lead to reduced covariance matrix inversion time and reduced number of evaluations of constraints. We demonstrated effectiveness of these algorithms in four challenging TAMPdomains. We now discuss limitations of the current approach and future work.

7.5.1 Fixed plan skeletons

In this work, we focused on TAMPproblems such that even for a fixed sequence of operators, also known as plan skeletons [84], the planner would yield a solution even for different problem instances. For example for the grasp-and-base selection domain, the sequence *MoveBase,Pick* was sufficient, and for the pick-and-place domain the sequence *Pick,MoveBase,Place* was sufficient for different problem instances. As noted by [84], the same plan skeleton can solve a large number of problem instances for some TAMPproblems.

However, there are more general TAMPproblems in which a fixed plan skeleton would not work. For example, consider the problem of making a cup of coffee, where a problem instance is defined by the number of spoons of sugar to put in. For such variation in problem instances, we would need different plan skeletons depending on the request.

To deal with this limitation, we are currently working on learning

high-level constraints that constrains the search space of plan skeletons from planning experience. Since the number of plan skeletons is discrete, if we can find a good set of constraints that reduces the space of plan skeletons to a small but promising set, then we can construct Θ for each skeleton, and then use BOX appropriately.

7.5.2 Discrete constraints

To make use of the correlation information among scores of constraints, our approach builds a discrete set of constraints and evaluates them on training problem instances during the training phase. For some applications, however, finding a solution that conforms to one of the constraints from a selected discrete set might be insufficient to cope with changes in problem instances. For instance, consider the task of moving objects to clear a path to the target object. Depending on the arrangement of moveable obstacles, we would need different object placements each time, and covering all possible such placements in a discrete set would be difficult.

For this problem, we are currently looking into generative models for generating promising constraints from the original space Θ. The idea is to use the recent advancement in generative model learning [41] to generate constraints with high scores, by training a generative model for constraints using successful plans. The main challenge would be how to incorporate score information appropriately to generative adversarial network.

8
Connection to Bayesian optimization

It turns out that the algorithm that we have presented in Chapter 7, BOX, has a strong connection to Bayesian Optimization (BO). In particular, BOX can be seen as a *meta-Bayesian optimization* algorithm that learns parameters of a Gaussian Process (GP) from which the objective function is drawn from, using previous optimization experience on functions that are also sampled from the same GP. This chapter presents this connection, based on a conference paper with Zi Wang [65], and proposes a novel variant of BOX, called Point Estimate Meta Bayesian Optimization (PEM-BO), with a principled method for determining the exploration parameter. We analyze its regret bound using tools from the BO literature. We show that by learning the parameters of the GP from prior optimization experience, we can improve the optimization efficiency, measured by the best function value obtained within the given number of function evaluations.

8.1 Background and related work

BO optimizes a black-box objective function through sequential queries. We usually assume knowledge of a Gaussian process [103] prior on the function, though other priors such as Bayesian neural networks and their variants [35, 79] are applicable too. Then, given possibly noisy observations and the prior distribution, we can do Bayesian posterior inference and construct acquisition functions [78, 90, 4] to search for the function optimizer.

However, in practice, we do not know the prior and it must be estimated. One of the most popular methods of prior estimation in BO is to optimize mean/kernel hyper-parameters by maximizing data-likelihood of the current observations [103, 46]. Another popular approach is to put a prior on the mean/kernel hyper-parameters and obtain a distribution of such hyper-parameters to adapt the model given observations [47, 114]. These methods require a predetermined form of the mean function and the kernel function. In the existing lit-

erature, mean functions are usually set to be 0 or linear and the popular kernel functions include Matérn kernels, Gaussian kernels, linear kernels [103] or additive/product combinations of the above [27, 56].

Meta BO aims to improve the optimization of a given objective function by learning from past experiences with other similar functions. Meta BO can be viewed as a special case of transfer learning or multi-task learning. One well-studied instance of meta BO is the machine learning (ML) hyper-parameter tuning problem on a dataset, where, typically, the validation errors are the functions to optimize [29]. The key question is how to transfer the knowledge from previous experiments on other datasets to the selection of ML hyper-parameters for the current dataset.

To determine the similarity between validation error functions on different datasets, meta-features of datasets are often used [12]. With those meta-features of datasets, one can use contextual Bayesian optimization approaches [73] that operate with a probabilistic functional model on both the dataset meta-features and ML hyper-parameters [6]. Feurer et al. [31], on the other hand, used meta-features of datasets to construct a distance metric, and to sort hyper-parameters that are known to work for similar datasets according to their distances to the current dataset. The best k hyper-parameters are then used to initialize a vanilla BO algorithm. If the function meta-features are not given, one can estimate the meta-features, such as the mean and variance of all observations, using Monte Carlo methods [123], maximum likelihood estimates [133] or maximum *a posteriori* estimates [100, 99].

As an alternative to using meta-features of functions, one can construct a kernel between functions. For functions that are represented by GPs, Malkomes et al. [86] studied a "kernel kernel", a kernel for kernels, such that one can use BO with a "kernel kernel" to select which kernel to use to model or optimize an objective function [85] in a Bayesian way. However, [86] requires an initial set of kernels to select from. Instead, Golovin et al. [40] introduced a setting where the functions come in sequence and the posterior of the former function becomes the prior of the current function. Removing the assumption that functions come sequentially, Feurer et al. [30] proposed a method to learn an additive ensemble of GPs that are known to fit all of those past "training functions".

Theoretically, it has been shown that meta BO methods that use information from similar functions may result in an improvement for the cumulative regret bound [73, 110] or the simple regret bound [99] with the assumptions that the GP priors are given. If the form of the GP kernel is given and the prior mean function is 0 but the kernel hyper-parameters are unknown, it is possible to obtain a regret

bound given a range of these hyper-parameters [130]. In this paper, we prove a regret bound for meta BO where the GP prior is unknown; this means, neither the range of GP hyper-parameters nor the form of the kernel or mean function is given.

A more ambitious approach to solving meta BO is to train an end-to-end system, such as a recurrent neural network [49], that takes the history of observations as an input and outputs the next point to evaluate [17]. Though it has been demonstrated that the method in [17] can learn to trade-off exploration and exploitation for a short horizon, it is unclear how many "training instances", in the form of observations of BO performed on similar functions, are necessary to learn the optimization strategies for any given horizon of optimization. In this paper, we show both theoretically and empirically how the number of "training instances" in our method affects the performance of BO.

Our methods are most similar to the BOX algorithm [60], which uses evaluations of previous functions to make point estimates of a mean and covariance matrix on the values over a discrete domain. Our methods for the discrete setting (described in Sec. 8.3.1) directly improve on BOX by choosing the exploration parameters in GP-UCB more effectively. This general strategy is extended to the continuous-domain setting in Sec. 8.3.2, in which we extend a method for learning the GP prior [98] and the use the learned prior in GP-UCB and PI.

Learning how to learn, or "meta learning", has a long history in machine learning [107]. It was argued that learning how to learn is "learning the prior" [8] with "point sets" [89], a set of iid sets of potentially non-iid points. We follow this simple intuition and present a meta BO approach that learns its GP prior from the data collected on functions that are assumed to have been drawn from the same prior distribution.

Empirical Bayes [104, 58] is a standard methodology for estimating unknown parameters of a Bayesian model. Our approach is a variant of empirical Bayes. We can view our computations as the construction of a sequence of estimators for a Bayesian model. The key difference from traditional empirical Bayes methods is that we are able to prove a regret bound for a BO method that uses estimated parameters to construct priors and posteriors. In particular, we use frequentist concentration bounds to analyze Bayesian procedures, which is one way to certify empirical Bayes in statistics [113, 28].

8.2 Problem formulation and notations

Unlike the standard BO setting, we do not assume knowledge of the mean or covariance in the GP prior, but we do assume the availability of a dataset of iid sets of potentially non-iid observations on functions sampled from the same GP prior. Then, given a new, unknown function sampled from that same distribution, we would like to find its maximizer.

More formally, we assume there exists a distribution $GP(\mu, k)$, and both the mean $\mu : \mathfrak{X} \to \mathbb{R}$ and the kernel $k : \mathfrak{X} \times \mathfrak{X} \to \mathbb{R}$ are unknown. Nevertheless, we are given a dataset $\bar{D}_N = \{[(\bar{x}_{ij}, \bar{y}_{ij})]_{j=1}^{M_i}\}_{i=1}^{N}$, where $\bar{y}_{ij}$ is drawn independently from $\mathcal{N}(f_i(\bar{x}_{ij}), \sigma^2)$ and $f_i : \mathfrak{X} \to \mathbb{R}$ is drawn independently from $GP(\mu, k)$. The noise level σ is unknown as well. We will specify inputs $\bar{x}_{ij}$ in Sec. 8.3.1 and Sec. 8.3.2.

Given a new function f sampled from $GP(\mu, k)$, our goal is to maximize it by sequentially querying the function and constructing $D_T = [(x_t, y_t)]_{t=1}^{T}$, $y_t \sim \mathcal{N}(f(x_t), \sigma^2)$. We study two evaluation criteria: (1) the *best-sample simple regret* $r_T = \max_{x \in \mathfrak{X}} f(x) - \max_{t \in [T]} f(x_t)$ which indicates the value of the best query in hindsight, and (2) the *simple regret*, $R_T = \max_{x \in \mathfrak{X}} f(x) - f(\hat{x}_T^*)$ which measures how good the inferred maximizer $\hat{x}_T^*$ is.

Notation We use $\mathcal{N}(u, V)$ to denote a multivariate Gaussian distribution with mean u and variance V and use $\mathcal{W}(V, n)$ to denote a Wishart distribution with n degrees of freedom and scale matrix V. We also use $[n]$ to denote $[1, \cdots, n], \forall n \in \mathbb{Z}^+$. We overload function notation for evaluations on vectors $\boldsymbol{x} = [x_i]_{i=1}^{n}, \boldsymbol{x}' = [x_j]_{j=1}^{n'}$ by denoting the output column vector as $\mu(\boldsymbol{x}) = [\mu(x_i)]_{i=1}^{n}$, and the output matrix as $k(\boldsymbol{x}, \boldsymbol{x}') = [k(x_i, x_j')]_{i \in [n], j \in [n']}$, and we overload the kernel function $k(\boldsymbol{x}) = k(\boldsymbol{x}, \boldsymbol{x})$.

8.3 Meta BO and its theoretical guarantees

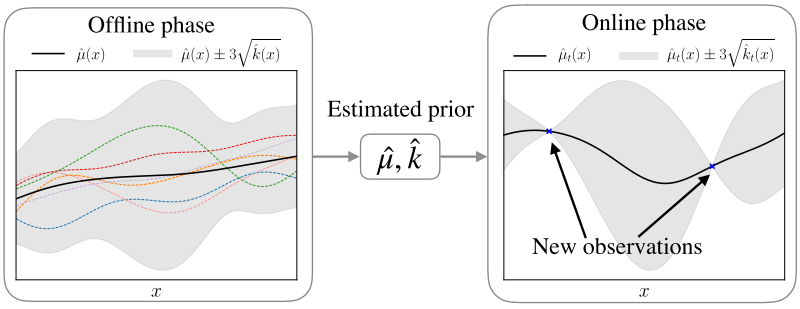


Figure 8.1: Our approach estimates the mean function $\hat{\mu}$ and kernel $\hat{k}$ from functions sampled from $GP(\mu, k)$ in the offline phase. Those sampled functions are illustrated by colored lines. In the online phase, a new function f sampled from the same $GP(\mu, k)$ is given and we can estimate its posterior mean function $\hat{\mu}_t$ and covariance function $\hat{k}_t$ which will be used for Bayesian optimization.

Instead of hand-crafting the mean μ and kernel k, we estimate them using the training dataset $\bar{D}_N$. Our approach is fairly straightforward: in the offline phase, the training dataset $\bar{D}_N$ is collected and we obtain estimates of the mean function $\hat{\mu}$ and kernel $\hat{k}$; in the online phase, we treat $GP(\hat{\mu}, \hat{k})$ as the Bayesian "prior" to do Bayesian optimization. We illustrate the two phases in Fig. 8.1. In Alg. 11, we depict our algorithm, assuming the dataset $\bar{D}_N$ has been collected. We use $\textsc{Estimate}(\bar{D}_N)$ to denote the "prior" estimation and $\textsc{Infer}(D_t; \hat{\mu}, \hat{k})$ the "posterior" inference, both of which we will introduce in Sec. 8.3.1 and Sec. 8.3.2. For acquisition functions, we consider special cases of *probability of improvement* (PI) [129, 78] and *upper confidence bound* (GP-UCB) [117, 4]:

$$\alpha_{t-1}^{\text{PI}}(x) = \frac{\hat{\mu}_{t-1}(x) - \hat{f}^*}{\hat{k}_{t-1}(x)^{\frac{1}{2}}}, \quad \alpha_{t-1}^{\text{GP-UCB}}(x) = \hat{\mu}_{t-1}(x) + \zeta_t \hat{k}_{t-1}(x)^{\frac{1}{2}}.$$

Here, PI assumes additional information in the form of the upper bound on function value $\hat{f}^* \geq \max_{x \in \mathfrak{X}} f(x)$; alternatively, an upper bound $\hat{f}^*$ can be estimated adaptively [129]. Note that here we are maximizing the PI acquisition function and hence $\alpha_{t-1}^{\text{PI}}(x)$ is a negative version of what was defined in [129]. For GP-UCB, we set its hyperparameter ζ_t to be

$$\zeta_t = \frac{\left(6(N-3+t+2\sqrt{t\log\frac{6}{\delta}}+2\log\frac{6}{\delta})/(\delta N(N-t-1))\right)^{\frac{1}{2}} + (2\log(\frac{3}{\delta}))^{\frac{1}{2}}}{(1-2(\frac{1}{N-t}\log\frac{6}{\delta})^{\frac{1}{2}})^{\frac{1}{2}}},$$

where N is the size of the dataset $\bar{D}_N$ and $\delta \in (0,1)$. With probability $1-\delta$, the regret bound in Thm. 2 or Thm. 4 holds with these special cases of GP-UCB and PI. Under two different settings of the search space $\mathfrak{X}$, finite $\mathfrak{X}$ and compact $\mathfrak{X} \in \mathbb{R}^d$, we show how our algorithm works in detail and why it works via regret analyses on the best-sample simple regret. Finally in Sec. 8.3.3 we show how the simple regret can be bounded. The proofs of the analyses can be found in the appendix of our original paper [65].

8.3.1 $\mathfrak{X}$ is a finite set

We first study the simplest case, where the function domain $\mathfrak{X} = [\bar{x}_j]_{j=1}^M$ is a finite set with cardinality $|\mathfrak{X}| = M \in \mathbb{Z}^+$. For convenience, we treat this set as an ordered vector of items indexed by $j \in [M]$. We collect the training dataset $\bar{D}_N = \{[(\bar{x}_j, \bar{\delta}_{ij}\bar{y}_{ij})]_{j=1}^M\}_{i=1}^N$, where $\bar{y}_{ij}$ are independently drawn from $\mathcal{N}(f_i(\bar{x}_j), \sigma^2)$, f_i are drawn independently from $GP(\mu, k)$ and $\bar{\delta}_{ij} \in \{0,1\}$. Because the training data can be collected offline by querying the functions $\{f_i\}_{i=1}^N$ in parallel, it is not unreasonable to assume that such a dataset $\bar{D}_N$ is available. If $\bar{\delta}_{ij} = 0$,

Algorithm 11 Meta Bayesian optimization

function META-BO($\bar{D}_N, f$)
  $\hat{\mu}(\cdot), \hat{k}(\cdot,\cdot) \leftarrow \textsc{Estimate}(\bar{D}_N)$
  return BO($f, \hat{\mu}, \hat{k}$)
end function

function BO ($f, \hat{\mu}, \hat{k}$)
  $D_0 \leftarrow \emptyset$
  for $t = 1, \cdots, T$
    $\hat{\mu}_{t-1}(\cdot), \hat{k}_{t-1}(\cdot) \leftarrow \textsc{Infer}(D_{t-1}; \hat{\mu}, \hat{k})$
    $\alpha_{t-1}(\cdot) \leftarrow \textsc{Acquisition}\,(\hat{\mu}_{t-1}, \hat{k}_{t-1})$
    $x_t \leftarrow \arg\max_{x \in \mathfrak{X}} \alpha_{t-1}(x)$
    $y_t \leftarrow \textsc{Observe}(f(x_t))$
    $D_t \leftarrow D_{t-1} \cup [(x_t, y_t)]$
  end for
  return D_T
end function

it means the (i,j)-th entry of the dataset $\bar{D}_N$ is missing, perhaps as a result of a failed experiment.

Estimating GP parameters If $\bar{\delta}_{ij} < 1$, we have missing entries in the observation matrix $\bar{Y} = [\bar{\delta}_{ij}\bar{y}_{ij}]_{i\in[N],j\in[M]} \in \mathbb{R}^{N\times M}$. Under additional assumptions specified in [16], including that $\operatorname{rank}(Y) = r$ and the total number of valid observations $\sum_{i=1}^{N}\sum_{j=1}^{M}\bar{\delta}_{ij} \geq O(rN^{\frac{6}{5}}\log N)$, we can use matrix completion [16] to fully recover the matrix $\bar{Y}$ with high probability. In the following, we proceed by considering completed observations only.

Let the completed observation matrix be $Y = [\bar{y}_{ij}]_{i\in[N],j\in[M]}$. We use an unbiased sample mean and covariance estimator for μ and k; that is, $\hat{\mu}(\mathfrak{X}) = \frac{1}{N}Y^{\mathrm{T}}1_N$ and $\hat{k}(\mathfrak{X}) = \frac{1}{N-1}(Y - 1_N\hat{\mu}(\mathfrak{X})^{\mathrm{T}})^{\mathrm{T}}(Y - 1_N\hat{\mu}(\mathfrak{X})^{\mathrm{T}})$, where 1_N is an N by 1 vector of ones. It is well known that $\hat{\mu}$ and $\hat{k}$ are independent and $\hat{\mu}(\mathfrak{X}) \sim \mathcal{N}(\mu(\mathfrak{X}), \frac{1}{N}(k(\mathfrak{X}) + \sigma^2\boldsymbol{I}))$, $\hat{k}(\mathfrak{X}) \sim \mathcal{W}(\frac{1}{N-1}(k(\mathfrak{X}) + \sigma^2\boldsymbol{I}), N-1)$ [1].

Constructing estimators of the posterior Given noisy observations $D_t = \{(x_\tau, y_\tau)\}_{\tau=1}^{t}$, we can do Bayesian posterior inference to obtain $f \sim GP(\mu_t, k_t)$. By the GP assumption, we get

$$\mu_t(x) = \mu(x) + k(x, \boldsymbol{x}_t)(k(\boldsymbol{x}_t) + \sigma^2\boldsymbol{I})^{-1}(\boldsymbol{y}_t - \mu(\boldsymbol{x}_t)), \quad \forall x \in \mathfrak{X} \tag{8.3.1}$$

$$k_t(x, x') = k(x, x') - k(x, \boldsymbol{x}_t)(k(\boldsymbol{x}_t) + \sigma^2\boldsymbol{I})^{-1}k(\boldsymbol{x}_t, x'), \quad \forall x, x' \in \mathfrak{X}, \tag{8.3.2}$$

where $\boldsymbol{y}_t = [y_\tau]_{\tau=1}^T$, $\boldsymbol{x}_t = [x_\tau]_{\tau=1}^T$ [103]. The problem is that neither the posterior mean μ_t nor the covariance k_t are computable because the Bayesian prior mean μ, the kernel k and the noise parameter σ are all unknown. How to estimate μ_t and k_t without knowing those prior parameters?

We introduce the following unbiased estimators for the posterior mean and covariance,

$$\hat{\mu}_t(x) = \hat{\mu}(x) + \hat{k}(x, \boldsymbol{x}_t)\hat{k}(\boldsymbol{x}_t, \boldsymbol{x}_t)^{-1}(\boldsymbol{y}_t - \hat{\mu}(\boldsymbol{x}_t)), \quad \forall x \in \mathfrak{X}, \tag{8.3.3}$$

$$\hat{k}_t(x, x') = \frac{N-1}{N-t-1}\left(\hat{k}(x, x') - \hat{k}(x, \boldsymbol{x}_t)\hat{k}(\boldsymbol{x}_t, \boldsymbol{x}_t)^{-1}\hat{k}(\boldsymbol{x}_t, x')\right), \quad \forall x, x' \in \mathfrak{X}. \tag{8.3.4}$$

Notice that unlike Eq. (8.3.1) and Eq. (8.3.2), our estimators $\hat{\mu}_t$ and $\hat{k}_t$ do not depend on any unknown values or an additional estimate of the noise parameter σ. In Lemma 1, we show that our estimators are indeed unbiased and we derive their concentration bounds.

Lemma 1. *Pick probability $\delta \in (0, 1)$. For any nonnegative integer $t < T$, conditioned on the observations $D_t = \{(x_\tau, y_\tau)\}_{\tau=1}^t$, the estimators in Eq. (8.3.3) and Eq. (8.3.4) satisfy $\mathbb{E}[\hat{\mu}_t(\mathfrak{X})] = \mu_t(\mathfrak{X}), \mathbb{E}[\hat{k}_t(\mathfrak{X})] = k_t(\mathfrak{X}) + \sigma^2\boldsymbol{I}$.*

Moreover, if the size of the training dataset satisfies $N \geq T + 2$, then for any input $x \in \mathfrak{X}$, with probability at least $1 - \delta$, both

$$|\hat{\mu}_t(x) - \mu_t(x)|^2 < a_t(k_t(x) + \sigma^2) \;\; \textit{and} \;\; 1 - 2\sqrt{b_t} < \hat{k}_t(x)/(k_t(x) + \sigma^2) < 1 + 2\sqrt{b_t} + 2b_t$$

hold, where $a_t = \frac{4\left(N-2+t+2\sqrt{t\log(4/\delta)}+2\log(4/\delta)\right)}{\delta N(N-t-2)}$ and $b_t = \frac{1}{N-t-1}\log\frac{4}{\delta}$.

Regret bounds We show a near-zero upper bound on the best-sample simple regret of meta BO with GP-UCB and PI that uses specific parameter settings in Thm. 2. In particular, for both GP-UCB and PI, the regret bound converges to a residual whose scale depends on the noise level σ in the observations.

Theorem 2. *Assume there exists constant $c \geq \max_{x\in\mathfrak{X}} k(x)$ and a training dataset is available whose size is $N \geq 4\log\frac{6}{\delta} + T + 2$. Then, with probability at least $1 - \delta$, the best-sample simple regret in T iterations of meta BO with special cases of either GP-UCB or PI satisfies*

$$r_T^{UCB} < \eta_T^{UCB}(N)\lambda_T, \;\; r_T^{PI} < \eta_T^{PI}(N)\lambda_T, \;\; \lambda_T^2 = O(\rho_T/T) + \sigma^2,$$

where $\eta_T^{UCB}(N) = (m + C_1)(\frac{\sqrt{1+m}}{\sqrt{1-m}} + 1)$, $\eta_T^{PI}(N) = (m + C_2)(\frac{\sqrt{1+m}}{\sqrt{1-m}} + 1) + C_3$, $m = O(\sqrt{\frac{1}{N-T}})$, $C_1, C_2, C_3 > 0$ are constants, and $\rho_T = \max_{A\in\mathfrak{X},|A|=T} \frac{1}{2}\log|\boldsymbol{I} + \sigma^{-2}k(A)|$.

This bound reflects how training instances N and BO iterations T affect the best-sample simple regret. The coefficients η_T^{UCB} and η_T^{PI} both converge to constants (for more details, please refer to the appendix of our original paper [65]), with components converging at rate $O(1/(N-T)^{\frac{1}{2}})$. The convergence of the shared term λ_T depends on ρ_T, the maximum information gain between function f and up to T observations $\boldsymbol{y}_T$. If, for example, each input has dimension $\mathbb{R}^d$ and $k(x, x') = x^{\mathrm{T}}x'$, then $\rho_T = O(d\log(T))$ [117], in which case λ_T converges to the observational noise level σ at rate $O(\sqrt{\frac{d\log(T)}{T}})$. Together, the bounds indicate that the best-sample simple regret of both our settings of GP-UCB and PI decreases to a constant proportional to noise level σ.

8.3.2 $\mathfrak{X} \subset \mathbb{R}^d$ is compact

For compact $\mathfrak{X} \subset \mathbb{R}^d$, we consider the primal form of GPs. We further assume that there exist basis functions $\Phi = [\phi_s]_{s=1}^K : \mathfrak{X} \to \mathbb{R}^K$, mean parameter $\boldsymbol{u} \in \mathbb{R}^K$ and covariance parameter $\Sigma \in \mathbb{R}^{K\times K}$ such that $\mu(x) = \Phi(x)^{\mathrm{T}}\boldsymbol{u}$ and $k(x, x') = \Phi(x)^{\mathrm{T}}\Sigma\Phi(x')$. Notice that $\Phi(x) \in \mathbb{R}^K$ is a column vector and $\Phi(\boldsymbol{x}_t) \in \mathbb{R}^{K\times t}$ for any $\boldsymbol{x}_t = [x_\tau]_{\tau=1}^t$. This means, for any input $x \in \mathfrak{X}$, the observation satisfies $y \sim \mathcal{N}(f(x), \sigma^2)$, where $f = \Phi(x)^{\mathrm{T}}W \sim GP(\mu, k)$ and the linear operator $W \sim \mathcal{N}(\boldsymbol{u}, \Sigma)$ [93]. In the following analyses, we assume the basis functions Φ are given.

We assume that a training dataset $\bar{D}_N = \{[(\bar{x}_j, \bar{y}_{ij})]_{j=1}^M\}_{i=1}^N$ is given, where $\bar{x}_j \in \mathfrak{X} \subset \mathbb{R}^d$, y_{ij} are independently drawn from $\mathcal{N}(f_i(\bar{x}_j), \sigma^2)$, f_i are drawn independently from $GP(\mu, k)$ and $M \geq K$.

Estimating GP parameters Because the basis functions Φ are given, learning the mean function μ and the kernel k in the GP is equivalent to learning the mean parameter $\boldsymbol{u}$ and the covariance parameter Σ that parameterize distribution of the linear operator W. Notice that $\forall i \in [N]$,

$$\bar{\boldsymbol{y}}_i = \Phi(\bar{\boldsymbol{x}})^{\mathrm{T}}W_i + \bar{\boldsymbol{\epsilon}}_i \sim \mathcal{N}(\Phi(\bar{\boldsymbol{x}})^{\mathrm{T}}\boldsymbol{u}, \Phi(\bar{\boldsymbol{x}})^{\mathrm{T}}\Sigma\Phi(\bar{\boldsymbol{x}}) + \sigma^2\boldsymbol{I}),$$

where $\bar{\boldsymbol{y}}_i = [\bar{y}_{ij}]_{j=1}^M \in \mathbb{R}^M$, $\bar{\boldsymbol{x}} = [\bar{x}_j]_{j=1}^M \in \mathbb{R}^{M\times d}$ and $\bar{\boldsymbol{\epsilon}}_i = [\bar{\epsilon}_{ij}]_{j=1}^M \in \mathbb{R}^M$. If the matrix $\Phi(\bar{\boldsymbol{x}}) \in \mathbb{R}^{K\times M}$ has linearly independent rows, one unbiased estimator of W_i is

$$\hat{W}_i = (\Phi(\bar{\boldsymbol{x}})^{\mathrm{T}})^{+}\bar{\boldsymbol{y}}_i = (\Phi(\bar{\boldsymbol{x}})\Phi(\bar{\boldsymbol{x}})^{\mathrm{T}})^{-1}\Phi(\bar{\boldsymbol{x}})\bar{\boldsymbol{y}}_i \sim \mathcal{N}(\boldsymbol{u}, \Sigma + \sigma^2(\Phi(\bar{\boldsymbol{x}})\Phi(\bar{\boldsymbol{x}})^{\mathrm{T}})^{-1}).$$

Let $\mathrm{W} = [\hat{W}_i]_{i=1}^N \in \mathbb{R}^{N\times K}$. We use the estimator $\hat{\boldsymbol{u}} = \frac{1}{N}\mathrm{W}^{\mathrm{T}}1_N$ and $\hat{\Sigma} = \frac{1}{N-1}(\mathrm{W} - 1_N\hat{\boldsymbol{u}})^{\mathrm{T}}(\mathrm{W} - 1_N\hat{\boldsymbol{u}})$ to the estimate GP parameters. Again, $\hat{\boldsymbol{u}}$ and $\hat{\Sigma}$ are independent and

$\hat{\boldsymbol{u}} \sim \mathcal{N}\left(\boldsymbol{u}, \frac{1}{N}(\Sigma + \sigma^2(\Phi(\bar{\boldsymbol{x}})\Phi(\bar{\boldsymbol{x}})^{\mathrm{T}})^{-1})\right), \hat{\Sigma} \sim \mathcal{W}\left(\frac{1}{N-1}\left(\Sigma + \sigma^2(\Phi(\bar{\boldsymbol{x}})\Phi(\bar{\boldsymbol{x}})^{\mathrm{T}})^{-1}\right), N-1\right)$ [1].

Constructing estimators of the posterior We assume the total number of evaluations $T < K$. Given noisy observations $D_t = \{(x_\tau, y_\tau)\}_{\tau=1}^t$, we have $\mu_t(x) = \Phi(x)^{\mathrm{T}} \boldsymbol{u}_t$ and $k_t(x, x') = \Phi(x)^{\mathrm{T}} \Sigma_t \Phi(x')$, where the posterior of $W \sim \mathcal{N}(\boldsymbol{u}_t, \Sigma_t)$ satisfies

$$\boldsymbol{u}_t = \boldsymbol{u} + \Sigma\Phi(\boldsymbol{x}_t)(\Phi(\boldsymbol{x}_t)^{\mathrm{T}}\Sigma\Phi(\boldsymbol{x}_t) + \sigma^2\boldsymbol{I})^{-1}(\boldsymbol{y}_t - \Phi(\boldsymbol{x}_t)^{\mathrm{T}}\boldsymbol{u}), \tag{8.3.5}$$

$$\Sigma_t = \Sigma - \Sigma\Phi(\boldsymbol{x}_t)(\Phi(\boldsymbol{x}_t)^{\mathrm{T}}\Sigma\Phi(\boldsymbol{x}_t) + \sigma^2\boldsymbol{I})^{-1}\Phi(\boldsymbol{x}_t)^{\mathrm{T}}\Sigma. \tag{8.3.6}$$

Similar to the strategy used in Sec. 8.3.1, we construct an estimator for the posterior of W to be

$$\hat{\boldsymbol{u}}_t = \hat{\boldsymbol{u}} + \hat{\Sigma}\Phi(\boldsymbol{x}_t)(\Phi(\boldsymbol{x}_t)^{\mathrm{T}}\hat{\Sigma}\Phi(\boldsymbol{x}_t))^{-1}(\boldsymbol{y}_t - \Phi(\boldsymbol{x}_t)^{\mathrm{T}}\boldsymbol{u}), \tag{8.3.7}$$

$$\hat{\Sigma}_t = \frac{N-1}{N-t-1}\left(\hat{\Sigma} - \hat{\Sigma}\Phi(\boldsymbol{x}_t)(\Phi(\boldsymbol{x}_t)^{\mathrm{T}}\hat{\Sigma}\Phi(\boldsymbol{x}_t))^{-1}\Phi(\boldsymbol{x}_t)^{\mathrm{T}}\hat{\Sigma}\right). \tag{8.3.8}$$

We can compute the conditional mean and variance of the observation on $x \in \mathfrak{X}$ to be $\hat{\mu}_t(x) = \Phi(x)^{\mathrm{T}}\hat{\boldsymbol{u}}_t$ and $\hat{k}_t(x) = \Phi(x)^{\mathrm{T}}\hat{\Sigma}_t\Phi(x)$. For convenience of notation, we define $\bar{\sigma}^2(x) = \sigma^2\Phi(x)^{\mathrm{T}}(\Phi(\bar{\boldsymbol{x}})\Phi(\bar{\boldsymbol{x}})^{\mathrm{T}})^{-1}\Phi(x)$.

Lemma 3. *Pick probability $\delta \in (0,1)$. Assume $\Phi(\bar{\boldsymbol{x}})$ has full row rank. For any nonnegative integer $t < T$, $T \le K$, conditioned on the observations $D_t = \{(x_\tau, y_\tau)\}_{\tau=1}^t$, $\mathbb{E}[\hat{\mu}_t(x)] = \mu_t(x), \mathbb{E}[\hat{k}_t(x)] = k_t(x) + \bar{\sigma}^2(x)$. Moreover, if the size of the training dataset satisfies $N \ge T + 2$, then for any input $x \in \mathfrak{X}$, with probability at least $1-\delta$, both*

$$|\hat{\mu}_t(x) - \mu_t(x)|^2 < a_t(k_t(x) + \bar{\sigma}^2(x)) \text{ and } 1 - 2\sqrt{b_t} < \hat{k}_t(x)/(k_t(x) + \bar{\sigma}^2(x)) < 1 + 2\sqrt{b_t} + 2b_t$$

hold, where $a_t = \frac{4\left(N-2+t+2\sqrt{t\log(4/\delta)}+2\log(4/\delta)\right)}{\delta N(N-t-2)}$ *and* $b_t = \frac{1}{N-t-1}\log\frac{4}{\delta}$.

Regret bounds Similar to the finite $\mathfrak{X}$ case, we can also show a near-zero regret bound for compact $\mathfrak{X} \in \mathbb{R}^d$. The following theorem clarifies our results. The convergence rates are the same as Thm. 2. Note that λ_T^2 converges to $\bar{\sigma}^2(\cdot)$ instead of σ^2 in Thm. 2 and $\bar{\sigma}^2(\cdot)$ is proportional to σ^2 .

Theorem 4. *Assume all the assumptions in Thm. 2 and that $\Phi(\bar{\boldsymbol{x}})$ has full row rank. With probability at least $1-\delta$, the best-sample simple regret in T iterations of meta BO with either GP-UCB or PI satisfies*

$$r_T^{UCB} < \eta_T^{UCB}(N)\lambda_T, \;\; r_T^{PI} < \eta_T^{PI}(N)\lambda_T, \;\; \lambda_T^2 = O(\rho_T/T) + \bar{\sigma}(x_\tau)^2,$$

where $\eta_T^{UCB}(N) = (m + C_1)(\frac{\sqrt{1+m}}{\sqrt{1-m}} + 1)$, $\eta_T^{PI}(N) = (m + C_2)(\frac{\sqrt{1+m}}{\sqrt{1-m}} + 1) + C_3$, $m = O(\sqrt{\frac{1}{N-T}})$, $C_1, C_2, C_3 > 0$ *are constants,* $\tau = \arg\min_{t\in[T]} k_{t-1}(x_t)$ *and* $\rho_T = \max_{A\in\mathfrak{X}, |A|=T} \frac{1}{2}\log|\boldsymbol{I} + \sigma^{-2}k(A)|$.

8.3.3 Bounding the simple regret by the best-sample simple regret

Once we have the observations $D_T = \{(x_t, y_t)\}_{t=1}^T$, we can infer where the arg max of the function is. For all the cases in which $\mathfrak{X}$ is discrete or compact and the acquisition function is GP-UCB or PI, we choose the inferred arg max to be $\hat{x}_T^* = x_\tau$ where $\tau = \arg\max_{\tau \in [T]} y_\tau$. We show in Lemma 5 that with high probability, the difference between the simple regret R_T and the best-sample simple regret r_T is proportional to the observation noise σ.

Lemma 5. *With probability at least* $1 - \delta$, $R_T \leq r_T + 2(2 \log \frac{1}{\delta})^{\frac{1}{2}} \sigma$.

Together with the bounds on the best-sample simple regret from Thm. 2 and Thm. 4, our result shows that, with high probability, the simple regret decreases to a constant proportional to the noise level σ as the number of iterations and training functions increases.

8.4 Experiments

We evaluate our algorithm in four different black-box function optimization problems, involving discrete or continuous function domains. One problem is optimizing a synthetic function in $\mathbb{R}^2$, and the rest are optimizing decision variables in robotic task and motion planning problems that were used in [60][1].

[1] Our code is available at `https://github.com/beomjoonkim/MetaLearnBO`.

At a high level, our task and motion planning benchmarks involve computing kinematically feasible collision-free motions for picking and placing objects in a scene cluttered with obstacles. This problem has a similar setup to experimental design: the robot can "experiment" by assigning values to decision variables including grasps, base poses, and object placements until it finds a feasible plan. Given the assigned values for these variables, the robot program makes a call to a planner[2] which then attempts to find a sequence of motions that achieve these grasps and placements. We score the variable assignment based on the results of planning, assigning a very low score if the problem was infeasible and otherwise scoring based on plan length or obstacle clearance.

[2] We use Rapidly-exploring random tree (RRT) [80] with predefined random seed, but other choices are possible.

Planning problem instances are characterized by arrangements of obstacles in the scene and the shape of the target object to be manipulated, and each problem instance defines a different score function. Our objective is to optimize the score function for a new problem instance, given sets of decision-variable and score pairs from a set of previous planning problem instances as training data.

In two robotics domains, we discretize the original function domain using samples from the past planning experience, by extracting the values of the decision variables and their scores from successful plans. This is inspired by the previous successful use of BO in a

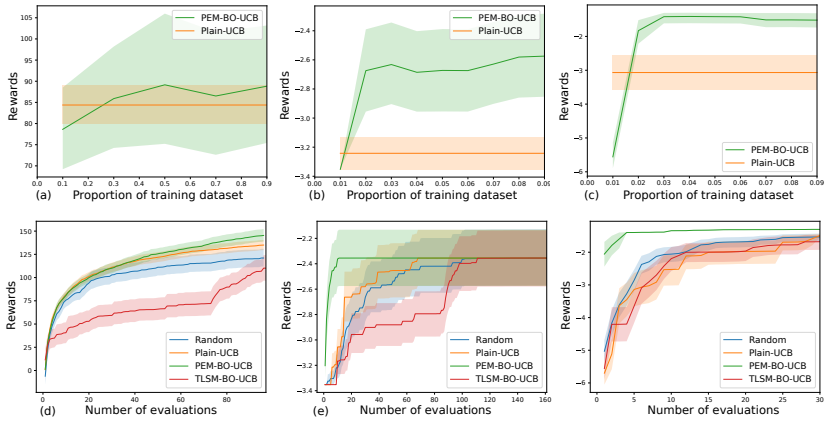


Figure 8.2: Learning curves (top) and rewards vs number of iterations (bottom) for optimizing synthetic functions sampled from a GP and two scoring functions from.

discretized domain [20] to efficiently solve an adaptive locomotion problem.

We compare our approach, called *point estimate meta Bayesian optimization* (PEM-BO), to three baseline methods. The first is a plain Bayesian optimization method that uses a kernel function to represent the covariance matrix, which we call PLAIN. PLAIN optimizes its GP hyperparameters by maximizing the data likelihood. The second is a *transfer learning sequential model-based optimization* [133] method, that, like PEM-BO, uses past function evaluations, but assumes that functions sampled from the same GP have similar response surface values. We call this method TLSM-BO. The third is random selection, which we call RANDOM.

In all domains, we use the ζ_t value as specified in Sec. 8.2. For continuous domains, we use $\Phi(x) = [\cos(x^T \beta^{(i)} + \beta_0^{(i)})]_{i=1}^K$ as our basis functions. In order to train the weights $W_i, \beta^{(i)}$, and $\beta_0^{(i)}$, we represent the function $\Phi(x)^T W_i$ with a 1-hidden-layer neural network with cosine activation function and a linear output layer with function-specific weights W_i. We then train this network on the entire dataset $\bar{D}_N$. Then, fixing $\Phi(x)$, for each set of pairs $(\bar{\boldsymbol{y}}_i, \bar{\boldsymbol{x}}_i), i = \{1 \cdots N\}$, we analytically solve the linear regression problem $\boldsymbol{y}_i \approx \Phi(\boldsymbol{x}_i)^T W_i$ as described in Sec. 8.3.2.

Optimizing a continuous synthetic function In this problem, the objective is to optimize a black-box function sampled from a GP, whose domain is $\mathbb{R}^2$, given a set of evaluations of different functions from the same GP. Specifically, we consider a GP with a squared exponential kernel function. The purpose of this problem is to show that PEM-BO, which estimates mean and covariance matrix based on $\bar{D}_N$, would perform similarly to BO methods that start with an appropriate prior. We have training data from $N = 100$ functions with $M = 1000$ sample points each.

Figure 8.2(a) shows the learning curve, when we have different portions of data. The x-axis represents the percentage of the dataset used to train the basis functions, $\boldsymbol{u}$, and W from the training dataset, and the y-axis represents the best function value found after 10 evaluations on a new function. We can see that even with just ten percent of the training data points, PEM-BO performs just as well as PLAIN, which uses the appropriate kernel for this particular problem. Compared to PEM-BO, which can efficiently use all of the dataset, we had to limit the number of training data points for TLSM-BO to 1000, because even performing inference requires $O(NM)$ time. This leads to its noticeably worse performance than PLAIN and PEM-BO.

Figure 8.2(d) shows the how $\max_{t \in [T]} y_t$ evolves, where $T \in [1, 100]$. As we can see, PEM-BO using the UCB acquisition function performs similarly to PLAIN with the same acquisition function. TLSM-BO again suffers because we had to limit the number of training data points.

Optimizing a grasp In the robot-planning problem shown in Figure 7.3, the robot has to choose a grasp for picking the target object in a cluttered scene. A planning problem instance is defined by the poses of obstacles and the target objects, which changes the feasibility of a grasp across different instances.

The reward function is the negative of the length of the picking motion if the motion is feasible, and $-k \in \mathbb{R}$ otherwise, where $-k$ is a suitably lower number than the lengths of possible trajectories. We construct the discrete set of grasps by using grasps that worked in the past planning problem instances. The original space of grasps is $\mathbb{R}^{58}$, which describes position, direction, roll, and depth of a robot gripper with respect to the object, as used in [23]. For both PLAIN and TLSM-BO, we use squared exponential kernel function on this original grasp space to represent the covariance matrix. We note that this is a poor choice of kernel, because the grasp space includes angles, making it a non-vector space. These methods also choose a grasp from the discrete set. We train on dataset with $N = 1800$ previous problems, and let $M = 162$.

Figure 8.2(b) shows the learning curve with $T = 5$. The x-axis is the percentage of the dataset used for training, ranging from one percent to ten percent. Initially, when we just use one percent of the training data points, PEM-BO performs as poorly as TLSM-BO, which again, had only 1000 training data points. However, PEM-BO outperforms both TLSM-BO and PLAIN after that. The main reason that PEM-BO outperforms these approaches is because their prior, which is defined by the squared exponential kernel, is not suitable for this problem. PEM-BO, on the other hand, was able to avoid this problem

by estimating a distribution over values at the discrete sample points that commits only to their joint normality, but not to any metric on the underlying space. These trends are also shown in Figure 8.2(e), where we plot $\max_{t \in [T]} y_t$ for $T \in [1, 100]$. PEM-BO outperforms the baselines significantly.

Optimizing a grasp, base pose, and placement We now consider a more difficult task that involves both picking and placing objects in a cluttered scene. A planning problem instance is defined by the poses of obstacles and the poses and shapes of the target object to be pick and placed. The reward function is again the negative of the length of the picking motion if the motion is feasible, and $-k \in \mathbb{R}$ otherwise. For both PLAIN and TLSM-BO, we use three different squared exponential kernels on the original spaces of grasp, base pose, and object placement pose respectively and then add them together to define the kernel for the whole set. For this domain, $N = 1500$, and $M = 1000$.

Figure 8.2(c) shows the learning curve, when $T = 5$. The x-axis is the percentage of the dataset used for training, ranging from one percent to ten percent. Initially, when we just use one percent of the training data points, PEM-BO does not perform well. Similar to the previous domain, it then significantly outperforms both TLSM-BO and PLAIN after increasing the training data. This is also reflected in Figure 8.2(f), where we plot $\max_{t \in [T]} y_t$ for $T \in [1, 100]$. PEM-BO outperforms baselines. Notice that PLAIN and TLSM-BO perform worse than RANDOM, as a result of making inappropriate assumptions on the form of the kernel.

8.5 Discussion

This chapter established the connection between BOX, presented in the Chapter 7 to meta BO. We proposed a new framework for meta BO that estimates its Gaussian process prior based on past experience with functions sampled from the same prior. We established regret bounds for our approach without the reliance on a known prior and showed its good performance on task and motion planning benchmark problems.

9
Conclusion

We presented a framework for learning to *guide* task-and-motion planning, which combines the benefits of pure planning and pure learning. We have identified the three fundamental challenges in designing a framework for learning to guide planning – designing representation, learning, and exploration vs. exploitation algorithms – and addressed them in various setups. We first considered a smaller yet important subclass of TAMP problems, called G-TAMP, and introduced several representation, learning, and planning algorithms. We then moved onto TAMP, and presented a framework that can guide the search given a plan skeleton. We have demonstrated, both theoretically and empirically, that by leveraging planning experience to guide a planner, we can obtain a solution more efficiently than using pure learning or pure planning algorithm alone.

The main idea that runs through all of the algorithms developed in this thesis is using learning to guide the computations of manually-designed algorithms, so that the entire system can be more efficient and adaptive. For instance, by learning samplers, we learned the patterns of placements of objects that are likely to lead to a goal. We believe that this approach can be applied to make other aspects of intelligent robots more efficient, such as perception, state-estimation, and control.

This is a crucial and exciting time for robotics. Advancements in data, algorithms, and hardware are hinting at general-purpose robots that can reliably and efficiently operate in diverse domains. For the next few decades, we believe a tight integration of machine learning and classical robotics algorithms will be invaluable in realizing this vision.

Bibliography

[1] Theodore Wilbur Anderson. *An Introduction to Multivariate Statistical Analysis*. Wiley New York, 1958.

[2] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 2009.

[3] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. *Proceedings of the International Conference on Machine Learning*, 2017.

[4] Peter Auer. Using confidence bounds for exploitation-exploration tradeoffs. *Journal of Machine Learning Research*, 3:397–422, 2002.

[5] D. Auger, A. Couëtoux, and Olivier Teytaud. Continuous upper confidence trees with polynomial exploration - consistency. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2013.

[6] Rémi Bardenet, Mátyás Brendel, Balázs Kégl, and Michele Sebag. Collaborative hyperparameter tuning. In *Proceedings of the International Conference on Machine Learning*, 2013.

[7] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[8] J Baxter. A Bayesian/information theoretic model of bias learning. In *Conference on Learning Theory*, 1996.

[9] D. Berenson, P. Abbeel, and K. Goldberg. A robot path planning framework that learns from experience. *IEEE Conference on Robotics and Automation*, 2012.

[10] H-G. Beyer and H-P Schwefel. Evolution strategies – a comprehensive introduction. *Natural Computing*, 2002.

[11] Mohak Bhardwaj, Sanjiban Choudhury, and Sebastian Scherer. Learning heuristic search via imitation. In *Conference on Robot Learning*, 2017.

[12] Pavel Brazdil, João Gama, and Bob Henery. Characterizing the applicability of classification algorithms using meta-level learning. In *European Conference on Machine Learning*, 1994.

[13] S. Bubeck, R. Munos, G. Stoltz, and C. Szepesvári. X-armed bandits. *Journal of Machine Learning Research*, 2011.

[14] Buşoniu, A Daniels, R. Munos, and R. Babuška. Optimistic planning for continuous-action deterministic systems. *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 2011.

[15] S. Cambon, R. Alami, and F. Gravot. A hybrid approach to intricate motion, manipulation, and task planning. *International Journal of Robotics Research*, 2009.

[16] Emmanuel J Candès and Benjamin Recht. Exact matrix completion via convex optimization. *Foundations of Computational mathematics*, 9(6):717, 2009.

[17] Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando de Freitas. Learning to learn without gradient descent by gradient descent. In *Proceedings of the International Conference on Machine Learning*, 2017.

[18] Rohan Chitnis, Dylan Hadfield-Menell, Abhishek Gupta, Siddharth Srivastava, Edward Groshev, Christopher Lin, and Pieter Abbeel. Guided search for task and motion plans using learned heuristics. In *IEEE International Conference on Robotics and Automation*, 2016.

[19] A. Couëtoux, J-B Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard. Continuous upper confidence trees. *International Conference on Learning and Intelligent Optimization*, 2011.

[20] A. Cully, J. Clune, D. Tarapore, and J. Mouret. Robots that adapt like animals. *Nature*, 2015.

[21] D. Dey, T. Y. Liu, B. Sofman, and J. A. Bagnell. Efficient optimization of control libraries. *AAAI Conference on Artificial Intelligence*, 2012.

[22] D. Dey, T. Y. Liu, B. Sofman, M. Hebert, and J. A. Bagnell. Contextual sequence prediction with application to control library optimization. *Robotics: Science and Systems*, 2012.

[23] R. Diankov. *Automated Construction of Robotic Manipulation Programs*. PhD thesis, CMU Robotics Institute, August 2010.

[24] M. Dogar and S. Srinivasa. A framework for push-grasping in clutter. *Robotics: Science and systems*, 2011.

[25] Carmel Domshlak, Erez Karpas, and Shaul Markovitch. To max or not to max: Online learning for speeding up optimal planning. In *AAAI Conference on Artificial Intelligence*, 2010.

[26] A. Dragan, G. J. Gordon, and S. S. Srinivasa. Learning from experience in manipulation planning: Setting the right goals. *International Symposium on Robotics Research*, 2011.

[27] David K Duvenaud, Hannes Nickisch, and Carl E Rasmussen. Additive Gaussian processes. In *Advances in Neural Information Processing Systems*, 2011.

[28] Bradley Efron. Bayes, oracle Bayes, and empirical Bayes. *Technical Report, Stanford University*, 2017.

[29] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, 2015.

[30] Matthias Feurer, Benjamin Letham, and Eytan Bakshy. Scalable meta-learning for Bayesian optimization. *arXiv preprint arXiv:1802.02219*, 2018.

[31] Matthias Feurer, Jost Springenberg, and Frank Hutter. Initializing Bayesian hyperparameter optimization via meta-learning. In *AAAI Conference on Artificial Intelligence*, 2015.

[32] Michael Fink. Online learning of search heuristics. In *Artificial Intelligence and Statistics*, 2007.

[33] S. Finney, L. P. Kaelbling, and T. Lozano-Pérez. Predicting partial paths from planning problem parameters. *Robotics: Science and Systems*, 2007.

[34] F-A Fortin, F-M De Rainville, M-A Gardner, M Parizeau, and C Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 2012.

[35] Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the International Conference on Machine Learning*, 2016.

[36] C. R. Garrett, L. P. Kaelbling, and T. Lozano-Pérez. Sample-based methods for factored task and motion planning. *Robotics: Science and Systems*, 2017.

[37] C. R. Garrett, T. Lozano-Peréz, and L P. Kaelbling. Ffrob: Leveraging symbolic planning for efficient task and motion planning. *International Journal of Robotics Research*, 2014.

[38] Caelan Reed Garrett, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning to rank for synthesizing planning heuristics. In *International Joint Conference on Artificial Intelligence*, 2016.

[39] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *Proceedings of the International Conference on Machine Learning*, 2017.

[40] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *SIGKDD Conference on Knowledge Discovery and Data Mining*, 2017.

[41] I. Goodfellow, J. Pouget-Abadie, M Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Advances in Neural Information Processing Systems*, 2014.

[42] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *IEEE International Joint Conference on Neural Networks*, 2005.

[43] F. Gravot, S. Cambon, and R. Alami. asymov: A planner that deals with intricate symbolic and geometric problems. *International Symposium on Robotics Research*, 2005.

[44] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of Wasserstein gans. *Proceedings of the International Conference on Machine Learning*, 2017.

[45] Kris Hauser. The minimum constraint removal problem with three robotics applications. *The International Journal of Robotics Research*, 33(1), 2014.

[46] Philipp Hennig and Christian J Schuler. Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*, 13:1809–1837, 2012.

[47] José Miguel Hernández-Lobato, Matthew W Hoffman, and Zoubin Ghahramani. Predictive entropy search for efficient global optimization of black-box functions. In *Advances in Neural Information Processing Systems*, 2014.

[48] J. Ho and S. Ermon. Generative adversarial imitation learning. *Advances in Neural Information Processing Systems*, 2016.

[49] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[50] J. Hodál and J. Dvořák. Using case-based reasoning for mobile robot path planning. *Journal of Engineering Mechanics*, 2008.

[51] J. Huang, A.J. Smola, A. Gretton, K.M. Borgwardt, and B. Schölkopf. Correcting sample selection bias by unlabeled data. In *Advances in Neural Information Processing Systems*, 2007.

[52] N. Jetchev and M. Toussaint. Fast motion planning from experience: trajectory prediction for speeding up movement generation. *Autonomous Robots*, 2013.

[53] Leslie Pack Kaelbling and Tomas Lozano-Pérez. Hierarchical task and motion planning in the now. *IEEE Conference on Robotics and Automation*, 2011.

[54] T. Kanamori, S. Hido, and M. Sugiyama. A least-squares approach to direct importance estimation. *Journal of Machine Learning Research*, 10, 2009.

[55] K. Kandasamy, J. Schneider, and B. Poczos. High dimensional bayesian optimisation and bandits via additive models. *Proceedings of the International Conference on Machine Learning*, 2015.

[56] Kirthevasan Kandasamy, Jeff Schneider, and Barnabas Poczos. High dimensional Bayesian optimisation and bandits via additive models. In *Proceedings of the International Conference on Machine Learning*, 2015.

[57] K. Kawaguchi, L.P. Kaelbling, and T. Lozano-Pérez. Bayesian optimization with exponential convergence. In *Advances in Neural Information Processing Systems*, 2015.

[58] Robert W Keener. *Theoretical Statistics: Topics for a Core Course*. Springer, 2011.

[59] B. Kim, A-M Farahmand, J. Pineau, and D. Precup. Learning from limited demonstrations. *Advances in Neural Information Processing Systems*, 2013.

[60] B. Kim, L. P. Kaelbling, and T. Lozano-Pérez. Learning to guide task and motion planning using score-space representation. *IEEE Conference on Robotics and Automation*, 2017.

[61] B. Kim, L. P. Kaelbling, and T. Lozáno-Pérez. Guiding search in continuous state-action spaces by learning an action sampler from off-target search experience. *AAAI Conference on Artificial Intelligence*, 2018.

[62] B. Kim, L. P. Kaelbling, and T. Lozáno-Pérez. Adversarial actor-critic method for task and motion planning problems using planning experience. *AAAI Conference on Artificial Intelligence*, 2019.

[63] B. Kim, K. Lee, S. Lim, L. P. Kaelbling, and T. Lozáno-Pérez. Monte Carlo tree search in continuous spaces using Voronoi optimistic optimization with regret bounds. *AAAI Conference on Artificial Intelligence*, 2020.

[64] B. Kim and L. Shimanuki. Learning value functions with relational state representations for guiding task-and-motion planning. *Conference on Robot Learning*, 2019.

[65] B. Kim, Z. Wang, and L. P. Kaelbling. Regret bounds for meta Bayesian optimization with an unknown Gaussian process prior. In *Advances in Neural Information Processing Systems*, 2018.

[66] B. Kim, Z. Wang, T. Lozano-Pérez, and L. P. Kaelbling. Learning to guide task and motion planning using score-space representation. In *International Journal of Robotics Research*, 2019.

[67] Jennifer E. King, Marco Cognetti, and Siddhartha S. Srinivasa. Rearrangement planning using object-centric and robot-centric action spaces. *IEEE International Conference on Robotics and Automation*, 2016.

[68] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In *International Conference on Learning Representations*, 2014.

[69] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations*, 2017.

[70] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pages 282–293. Springer, 2006.

[71] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[72] V. R. Konda and J. N. Tsitsiklis. On actor-critic algorithms. *SIAM Journal on Control and Optimization*, 2003.

[73] Andreas Krause and Cheng S Ong. Contextual Gaussian process bandit optimization. In *Advances in Neural Information Processing Systems*, 2011.

[74] O. Kroemer and G. S. Sukhatme. Meta-level priors for learning manipulation skills with sparse features. *International Symposium on Experimental Robotics*, 2016.

[75] O. Kroemer and G. S. Sukhatme. Feature selection for learning versatile manipulation skills based on observed and desired trajectories. *IEEE International Conference on Robotics and Automation*, 2017.

[76] Athanasios Krontiris and Kostas E Bekris. Dealing with difficult instances of object rearrangement. In *Robotics: Science and Systems*, 2015.

[77] J.J Kuffner and S.M LaValle. RRT-connect: An efficient approach to single-query path planning. In *International Conference on Robotics and Automation*, 2000.

[78] Harold J Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Fluids Engineering*, 86(1):97–106, 1964.

[79] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems*, 2017.

[80] Steven M LaValle and James J Kuffner Jr. Rapidly-exploring random trees: Progress and prospects. In *Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2000.

[81] S. Levine and P. Abbeel. Learning neural network policies with guided policy search under unknown dynamics. *Advances in Neural Information Processing Systems*, 2014.

[82] J. Lien and Y. Lu. Planning motion in environments with similar obstacles. *Robotics: Science and Systems*, 2009.

[83] T. P. Lillicrap, A. Pritzel J. J. Hunt, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *International Conference on Learning Representations*, 2016.

[84] T. Lozano-Pérez and L.P. Kaelbling. A constraint-based method for solving sequential manipulation planning problems. *International Conference on Intelligent Robots and Systems*, 2014.

[85] Gustavo Malkomes and Roman Garnett. Towards automated Bayesian optimization. In *ICML AutoML Workshop*, 2017.

[86] Gustavo Malkomes, Charles Schaff, and Roman Garnett. Bayesian optimization for automated model selection. In *Advances in Neural Information Processing Systems*, 2016.

[87] Mansley, A. Weinstein, and M. Littman. Sample-based planning for continuous action Markov Decision Processes. *International Conference on Automated Planning and Scheduling*, 2011.

[88] Amir massoud Farahmand. Action-gap phenomenon in reinforcement learning. In *Advances in Neural Information Processing Systems*, 2011.

[89] T P Minka and R W Picard. Learning how to learn is learning with point sets. Technical report, MIT Media Lab, 1997.

[90] J. Močkus. On Bayesian methods for seeking the extremum. In *Optimization Techniques International Federation for Information Processing Technical Conference*, 1974.

[91] R. Munos. Optimistic optimization of a deterministic function without the knowledge of its smoothness. *Advances in Neural Information Processing Systems*, 2011.

[92] R. Munos. From bandits to Monte-carlo Tree Search: the optimistic principle applied to optimization and planning. *Foundations and Trends in Machine Learning*, 2014.

[93] R.M. Neal. *Bayesian Learning for Neural Networks*. Lecture Notes in Statistics 118. Springer, 1996.

[94] S. Pandya and S. Hutchinson. A case-based approach to robot motion planning. *IEEE International Conference on Systems, Man, and Cybernetics*, 1992.

[95] M. Phillips, B. Cohen, S. Chita, and M. Likhachev. E-graphs: Bootstrapping planning with experience graphs. *Robotics: Science and Systems*, 2012.

[96] J.D Pintér. *Global Optimization in Action (Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications)*. Springer US, 1996.

[97] Jervis Pinto and Alan Fern. Learning partial policies to speedup MDP tree search via reduction to I.I.D. learning. *Journal of Machine Learning Research*, 2017.

[98] John C Platt, Christopher JC Burges, Steven Swenson, Christopher Weare, and Alice Zheng. Learning a Gaussian process prior for automatically generating music playlists. In *Advances in Neural Information Processing Systems*, 2002.

[99] Matthias Poloczek, Jialei Wang, and Peter Frazier. Multi-information source optimization. In *Advances in Neural Information Processing Systems*, 2017.

[100] Matthias Poloczek, Jialei Wang, and Peter I Frazier. Warm starting Bayesian optimization. In *Winter Simulation Conference*, 2016.

[101] D. Precup, R.S. Sutton, and S. Dasgupta. Off-policy temporal-difference learning with function approximation. In *Proceedings of the International Conference on Machine Learning*, 2001.

[102] A.K. Qin and P.N. Suganthan. Self-adaptive differential evolution algorithm for numerical optimization. *IEEE Congress on Evolutionary Computation*, 2005.

[103] Carl Edward Rasmussen and Christopher KI Williams. Gaussian processes for machine learning. *The MIT Press*, 2006.

[104] Herbert Robbins. An empirical Bayes approach to statistics. In *Third Berkeley Symposium on Mathematical Statistics and Probability*, 1956.

[105] S. Ross, G.J. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. *International Conference on Artificial Intelligence and Statistics*, 2011.

[106] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 2009.

[107] J Schmidhuber. On learning how to learn learning strategies. Technical report, 1995.

[108] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel. Motion planning with sequential convex optimization and convex collision checking. *International Journal of Robotics Research*, 2014.

[109] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv*, 2017.

[110] Alistair Shilton, Sunil Gupta, Santu Rana, and Svetha Venkatesh. Regret bounds for transfer learning in Bayesian optimisation. In *International Conference on Artificial Intelligence and Statistics*, 2017.

[111] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 2016.

[112] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 2017.

[113] Suzanne Sniekers, Aad van der Vaart, et al. Adaptive Bayesian credible sets in regression with a Gaussian process prior. *Electronic Journal of Statistics*, 9(2):2475–2527, 2015.

[114] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems*, 2012.

[115] Alessandro Sperduti and Antonina Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 1997.

[116] N. Srinivas, A. Krause, S. Kakade, and M. Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *Proceedings of the International Conference on Machine Learning*, 2010.

[117] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the International Conference on Machine Learning*, 2010.

[118] S. Srivastava, E. Fang, L Riano, R. Chitnis, S. Russell, and P. Abbeel. Combined task and motion planning through an extensible planner-independent interface layer. *IEEE Conference on Robotics and Automation*, 2014.

[119] M. Stilman, J-U. Schamburek, J. Kuffner, and T. Asfour. Manipulation planning among movable obstacles. *IEEE International Conference on Robotics and Automation*, 2007.

[120] Mike Stilman and James J Kuffner. Navigation among movable obstacles: Real-time reasoning in complex environments. *International Journal of Humanoid Robotics*, 2005.

[121] M. Sugiyama, S. Nakajima, P.V. Buenau H. Kashima, and M. Kawananbe. Direct importance estimation with model selection and its application to covariate shift adaptation. In *Advances in Neural Information Processing Systems*, 2008.

[122] R.S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[123] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task Bayesian optimization. In *Advances in Neural Information Processing Systems*, 2013.

[124] Marc Toussaint. Logic-geometric programming: An optimization-based approach to combined task and motion planning. *International Joint Conference on Artificial Intelligence*, 2015.

[125] I. Tsochantaridis, T. Joachims, T. Hofmann, Y. Altun, and Y. Singer. Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research*, 2006.

[126] Z. Wang, B. Shakibi, L. Jin, and N. Freitas. Bayesian multi-scale optimistic optimization. *International Conference on Artificial Intelligence and Statistics*, 2014.

[127] Z. Wang, B. Zhou, and S. Jegelka. Optimization as estimation with Gaussian processes in bandit settings. *International Conference on Artificial Intelligence and Statistics*, 2017.

[128] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. Freitas. Bayesian optimization in high dimensions via random embeddings. *International Conference on Artificial Intelligence and Statistics*, 2013.

[129] Zi Wang and Stefanie Jegelka. Max-value entropy search for efficient Bayesian optimization. In *Proceedings of the International Conference on Machine Learning*, 2017.

[130] Ziyu Wang and Nando de Freitas. Theoretical analysis of Bayesian optimisation with unknown Gaussian process hyperparameters. In *NIPS Workshop on Bayesian Optimization*, 2014.

[131] A. Weinstein and M. Littman. Bandit-based planning and learning in continuous-action Markov Decision Proceses. *International Conference on Automated Planning and Scheduling*, 2012.

[132] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[133] Dani Yogatama and Gideon Mann. Efficient transfer learning method for automatic hyperparameter tuning. In *International Conference on Artificial Intelligence and Statistics*, 2014.

[134] Sung Wook Yoon, Alan Fern, and Robert Givan. Learning heuristic functions from relaxed plans. In *International Conference on Automated Planning and Scheduling*, 2006.

[135] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[136] Y. Zhu, D. Gordon, E. Kolve, D. Fox, L. Fei-Fei, A. Gupta, R. Mottaghi, and A. Farhadi. Visual semantic planning using deep successor representations. *International Conference on Computer Vision*, 2017.