EFFICIENTLY MANIPULATING CLUTTER VIA LEARNING AND SEARCH-BASED REASONING

By

BAICHUAN HUANG

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Jingjin Yu

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May 2025

ABSTRACT OF THE DISSERTATION

Efficiently Manipulating Clutter via Learning and Search-Based Reasoning

by **BAICHUAN HUANG**

Dissertation Director: Jingjin Yu

Object rearrangement is a fundamental and highly challenging problem in robotic manipulation, encompassing a diverse set of tasks such as clutter removal and object retrieval. These tasks require robots to intelligently plan and execute sequences of manipulation actions to reorganize objects or extract specific targets from cluttered environments. The significance of object rearrangement extends to numerous real-world applications, including warehouse automation, household assistance, and industrial manufacturing. However, solving this problem efficiently remains difficult due to the high-dimensional configuration spaces, intricate object interactions, and long planning horizons involved. The ability to develop more advanced solutions to these challenges is critical for enabling robots to operate autonomously in unstructured real-world settings.

This dissertation is motivated by the need for more efficient and robust manipulation planning approaches that can be adapted to dynamic and complex environments. To address these challenges, we propose a set of novel algorithms that leverage the strengths of search-based planning, deep learning, and parallelized computation. Our work focuses on improving the prediction of object interactions, integrating these predictions into tree search algorithms, and utilizing high-performance parallel computing to significantly accelerate the planning process.

Our research begins with the development of the Deep Interaction Prediction Network

(DIPN), which enables accurate predictions of object motions when subjected to pushing actions. DIPN is trained to model object interactions with high precision, achieving over 90% accuracy in predicting the final poses of objects after a push. This significantly surpasses existing baseline methods and allows for more reliable decision-making in cluttered environments. Building on this capability, we integrate DIPN with Monte Carlo Tree Search (MCTS) to optimize the planning of non-prehensile actions for object retrieval tasks. This integrated approach enables robots to autonomously determine effective sequences of push actions, leading to a 100% completion rate in specific, well-defined challenging scenarios where heuristic-based solutions previously struggled.

To further improve computational efficiency, we introduce the Parallel Monte Carlo Tree Search with Batched Simulations (PMBS) framework. This framework leverages GPU-accelerated physics simulations to parallelize planning computations, achieving more than a $30\times$ speed-up compared to traditional serial implementations. Importantly, this acceleration does not compromise solution quality; PMBS maintains or even improves it, demonstrating its effectiveness for real-time robotic planning. Additionally, we combine different manipulation techniques, such as pick-and-place and push, to make our approach more flexible and adaptable to various tasks. By integrating diverse manipulation techniques, our system can tackle a wider range of object rearrangement challenges more effectively.

Extensive experiments conducted in both simulated environments and real-world robotic systems validate the efficacy of our proposed methods. Our findings demonstrate state-of-the-art performance in terms of success rates, solution quality, and computational efficiency across a variety of complex rearrangement tasks. By pushing the boundaries of robotic manipulation capabilities, this work contributes to the advancement of autonomous robotic systems, bringing us closer to deploying intelligent robots capable of handling complex object rearrangement tasks in real-world, unstructured environments.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Prof. Jingjin Yu, for his inspiration and guidance throughout my research journey. Your expertise and mentorship were instrumental in shaping every aspect of my work during my five-year Ph.D. study, not only in academic matters but also in practical aspects such as building robust robotic systems and in guiding my career planning. I am also profoundly thankful to my dissertation committee members. In particular, I would like to thank Prof. Abdeslam Boularias, who contributed significantly to much of my works and provided insightful ideas that greatly enhanced my research. My sincere appreciation goes to all my committee members, not only for their valuable comments and suggestions on this dissertation but also for their broader impact on my academic growth. I am especially honored to have served as a teaching assistant for Prof. Kostas Bekris, from whom I learned invaluable lessons about effective teaching. I extend my heartfelt gratitude to Dr. Bowen Wen, a member of my committee, whose research findings have deeply influenced my own work and this dissertation.

I have been fortunate to collaborate, both formally and informally, with several of my lab members. I would particularly like to thank Shuai Han for his contributions to my first paper at Rutgers. To all my lab mates: thank you for your companionship and unwavering support during times of need. I am grateful to the funding agencies that supported my work, as well as to my colleagues during my internships. Special thanks to Dr. Siddarth Jain and the team at Mitsubishi Electric Research Laboratories, where I gained invaluable industrial experience. I also appreciate the opportunities provided by Coupang, which further broadened my professional horizons.

Last but certainly not least, I want to express my heartfelt thanks to my parents for their unconditional love and support throughout this journey. Your encouragement has been my foundation and inspiration.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

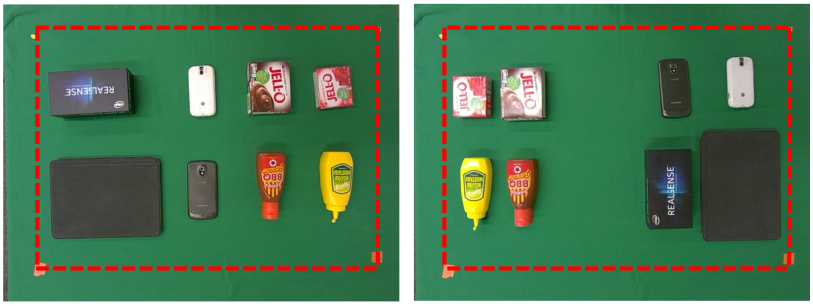
Robotic manipulation [1] is a fundamental capability that enables various applications across many industries. In warehouse automation, robots are revolutionizing order fulfillment and inventory management [2]. In healthcare, robotic systems assist in surgeries, patient care, and laboratory tasks [3]. Human-robot collaboration is enhancing productivity in manufacturing and assembly lines [4]. In disaster response scenarios, search and rescue robots navigate hazardous environments to locate and assist survivors [5]. These diverse applications highlight the critical role of robotic manipulation.

Among these applications, clutter removal is an important task involving the grasping and extracting of objects from cluttered environments such as bins or conveyor belts [6]. Object retrieval is a closely related but distinct challenge, where the goal is to extract a specific target object from a cluttered scene. This capability is crucial for household robots tasked with retrieving items from a pile, where specific items need to be identified and retrieved [7]. Beyond removal and retrieval, object rearrangement [8] presents its own challenges, requiring robots to reorganize objects within a given space. Furthermore, the ability to manipulate objects in dynamic environments, where items may be moving or rolling, adds another layer of complexity to these tasks [9]. These challenges, illustrated in Figure 1.1, all fall under the broader category of robot manipulation tasks and represent active areas of research.

The field of robotic manipulation has a rich history of research and development. Approaches to solving these problems have generally fallen into two main categories: learning-based methods and analytical algorithms. Learning-based methods, which rely on neural networks to determine the next action, show promise in performing specific tasks. However, they currently face limitations regarding stability and generalization due to the

(a) Grasping all from Clutter

(b) Grasping target from Clutter



(c) Object Rearrangement: From start (left) to goal (right)

Figure 1.1: Examples of robot manipulation tasks. (a) Grasping objects from clutter: This can involve grasping all objects or targeting a specific item. The challenge lies in creating sufficient space for the gripper to access objects. (b) Dynamic grasping: Manipulating moving objects, such as receiving an item from a human hand. (c) Object rearrangement: Reorganizing objects to achieve a desired layout, similar to housekeeping. This task requires both high-level planning and precise motion control.

scarcity of training data in the robotics domain [10]. On the other hand, analytical algorithms, while often more stable, struggle with defining metrics and rules from visual inputs, face challenges in exploring vast solution spaces, and frequently lack the flexibility to adapt to varied problems or changes in the scene. These methods are often sensitive to environmental variations and struggle to generalize across different scenarios [11].

Given these challenges, developing robust and versatile algorithms for real-world robotic manipulation applications is paramount. Such algorithms must be capable of planning solutions, reasoning about physical interactions between objects, and controlling the robot to complete tasks successfully. We aim to create a general framework supporting similar but distinct applications, balancing solution quality and computational efficiency. This

dissertation aims to address these challenges by proposing approaches combining the strengths of learning-based methods and analytical algorithms in robotics. We explore using deep learning to predict object interactions, integrate these predictions with tree search algorithms for efficient planning, and leverage parallel computing to accelerate decision-making processes. Our work spans various aspects of robotic manipulation, from object retrieval in cluttered environments to dynamic grasping of moving objects, with the overarching goal of advancing the capabilities of robotic systems in handling complex, real-world manipulation tasks. Through a series of interconnected studies, we demonstrate how our proposed methods achieve state-of-the-art performance in various challenging manipulation scenarios. By focusing on both the theoretical foundations and practical implementations of these algorithms, we contribute to the broader goal of deploying versatile and efficient robotic systems in unstructured real-world environments.

This dissertation is structured to align with the planned chapters, each corresponding to published research work [12]–[23].

1.1 DIPN: Deep Interaction Prediction Network with Application to Clutter Removal

Chapter 3 introduces the Deep Interaction Prediction Network (DIPN), a neural model designed to predict the effects of push actions in cluttered environments. DIPN leverages deep learning to estimate object interactions when a robot manipulator executes push actions, generating accurate synthetic images of potential outcomes. This predictive capability enables the system to make intelligent push-versus-grasp decisions, ultimately facilitating efficient clutter removal. By integrating DIPN with a grasp prediction network, the system achieves robust self-supervised learning, significantly outperforming previous state-of-the-art methods. Remarkably, DIPN demonstrates superior generalization on real hardware compared to simulation, underscoring its practical utility in robotic manipulation tasks.

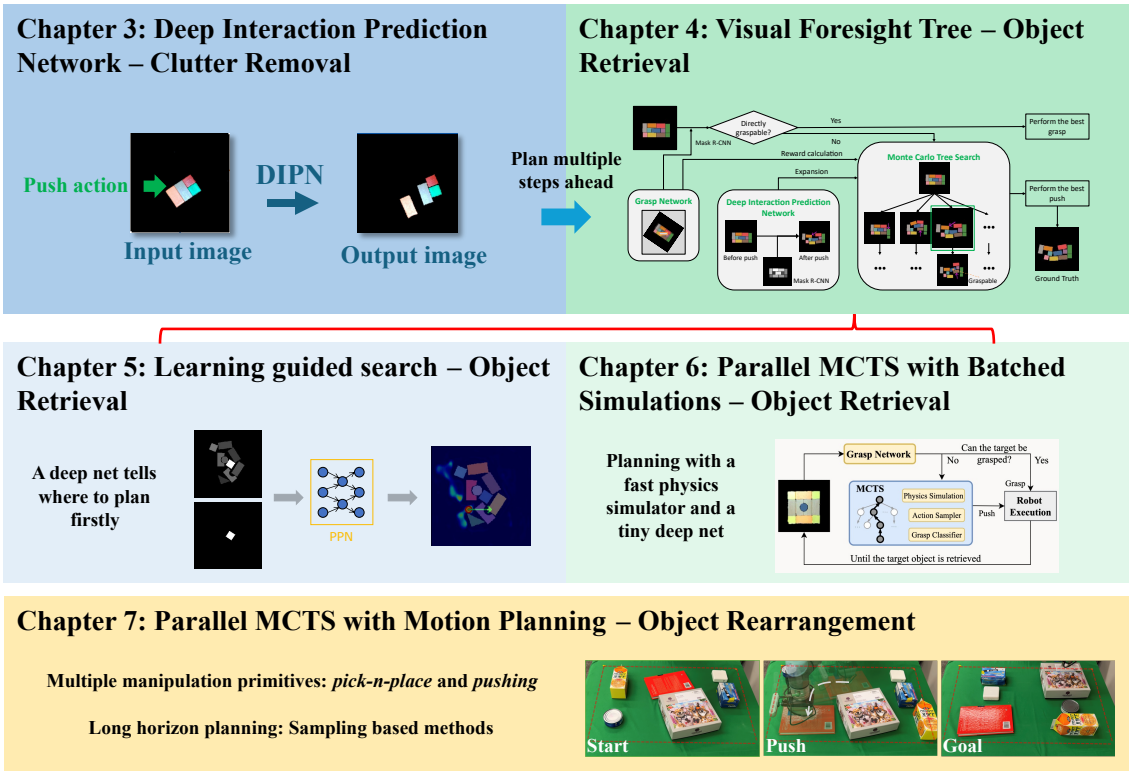


Figure 1.2: Structure of the dissertation. Chapter 3 introduces the Deep Interaction Prediction Network for one-step push prediction in clutter removal. Chapter 4 extends to multi-step planning for efficient object retrieval using push actions. Chapters 5 and 6 explore GPU-accelerated Monte Carlo tree search: Chapter 5 focuses on learning a strategic network to guide tree search, while Chapter 6 utilizes Isaac Gym for parallel simulations in real robot execution. Chapter 7 applies similar concepts to object rearrangement in constrained spaces, incorporating motion planning.

1.2 Visual Foresight Trees for Object Retrieval from Clutter with Nonprehensile Rearrangement

Building on single-step push predictions, Chapter 4 extends the scope to multi-step planning for object retrieval in clutter. This chapter presents the Visual Foresight Trees (VFT) framework, which employs a deep predictive model (DIPN) to anticipate object movements resulting from sequential pushing actions. By integrating a tree search algorithm, VFT evaluates various push sequences to determine the optimal strategy for rearranging the environment before grasping a target object. The method significantly improves retrieval success rates and reduces the number of required actions compared to the baseline. Experiments in

both simulation and real-world settings confirm that VFT effectively balances prediction accuracy with computational efficiency, paving the way for more intelligent robotic planning.

1.3 Interleaving Monte Carlo Tree Search and Self-Supervised Learning for Object Retrieval in Clutter

Chapter 5 explores a deeper integration of Monte Carlo Tree Search with learning-based strategies for object retrieval in cluttered environments. The Monte Carlo tree search and learning for Object REtrieval (MORE) framework follows a self-supervised approach inspired by Kahneman's System 2 $\rightarrow$ System 1 learning paradigm. Initially, MCTS enables a deep neural network to understand object interactions and predict optimal push actions. Once trained, the network is incorporated into MCTS to accelerate decision-making, significantly reducing computational overhead while maintaining or improving solution quality. MORE represents a key step in closing the loop between classical planning and deep learning for efficient robotic manipulation.

1.4 Parallel Monte Carlo Tree Search with Batched Rigid-body Simulations for Speeding up Long-Horizon Episodic Robot Planning

Unlike Chapter 5, which focuses on learning-guided Monte Carlo Tree Search (MCTS), PMBS takes a different approach by introducing large-scale parallel simulations to enhance planning efficiency. The Parallel Monte Carlo Tree Search with Batched Simulations (PMBS) method leverages GPU-based parallelism to evaluate multiple action trajectories simultaneously using Isaac Gym. By executing a vast number of physics simulations in parallel with strategic sampling, PMBS significantly accelerates long-horizon planning tasks, such as object retrieval from clutter, achieving over $30\times$ speedups compared to conventional MCTS. PMBS achieves significant computational efficiency and maintains high solution quality while minimizing planning latency, thus making real-time robotic decision-making more viable. Experimental results further demonstrate that PMBS can be

seamlessly deployed on real hardware with minimal sim-to-real discrepancies.

1.5 Toward Optimal Tabletop Rearrangement with Multiple Manipulation Primitives

Finally, Chapter 7 applies these principles to a broader class of robotic manipulation tasks. This chapter presents the Rearrangement with Multiple Manipulation Primitives (REMP) problem, which involves coordinating pick-and-place and push actions to optimally organize objects in constrained spaces. Two complementary algorithms are developed: hierarchical best-first search (HBFS) for fast heuristic planning and parallel MCTS for multi-primitive rearrangement (PMMR) for high-quality, human-like solutions. The integration of push and pick-and-place strategies allows robots to efficiently solve complex rearrangement tasks that require diverse manipulation skills. Extensive evaluations in both simulation and real-world environments highlight the effectiveness of these approaches in optimizing object placement while ensuring task success.

CHAPTER 2
RELATED WORKS

Robotic manipulation encompasses a broad range of methods and tasks, including grasping, singulation, retrieval, rearrangement, pushing, and combined push-grasping strategies. The following sections review the relevant literature along these dimensions, highlighting both classical (analytical) approaches and more recent learning-based methods. We also highlight how Task and Motion Planning (TAMP), Monte Carlo Tree Search (MCTS), and other advanced approaches have been used to tackle long-horizon challenges such as clutter removal, object retrieval, and object rearrangement.

2.1 Prehensile vs. Non-Prehensile Manipulation

Manipulation actions can be broadly classified into prehensile and non-prehensile actions. Prehensile (or "grasping") actions involve lifting or holding objects using a gripper or suction, while non-prehensile actions (e.g., pushing, dragging, toppling) manipulate objects by applying forces without grasping them. Often, these two families of actions are studied separately [24]–[26]. However, there is an increasing interest in leveraging both types to tackle challenging tasks more effectively [27]–[29]. Despite being limited in variety, prehensile actions allow a robot to secure an object and move it in tandem with the end-effector. Non-prehensile actions, on the other hand, use contact with the environment to control or guide objects on a surface [1], [30], [31]. As we will discuss below, combining the strengths of prehensile and non-prehensile actions (e.g., pushing to create space or orient an object prior to grasping) can significantly improve performance in cluttered or complex environments.

2.2 Object Grasping

2.2.1 Analytical vs. Data-Driven Approaches

Robotic grasping methods can generally be divided into analytical and data-driven categories. Analytical approaches rely on precise object models (often 3D) and mechanical properties (e.g., friction coefficients and mass distributions) to evaluate force-closure or form-closure conditions for grasp stability [11], [24]. However, building these exact models can be prohibitively difficult in real-world settings due to incomplete scans of objects, unknown friction coefficients, and inaccurate mass estimates.

To address these challenges, data-driven methods [32] learn grasp success probabilities directly from observations. Early data-driven techniques often focused on isolated single objects [33]–[37]. More recent work has shifted to grasping in clutter, exploring how learned models can effectively handle occlusions, unexpected contacts, and multi-object interactions [38]–[44]. A common approach trains convolutional neural networks (CNNs) to propose candidate grasp actions, sometimes producing 6-DoF grasp poses in point clouds, such as Dex-Net [45], [46]. Dex-Net and other approaches also combine suction grippers with fingered jaws to maximize grasp stability [6], [47].

2.2.2 Grasping in Dense Environments

When objects are densely packed, direct grasping may fail due to collisions or partial occlusions. Recent techniques incorporate additional manipulation primitives (e.g., pushing, poking, or top-sliding) to move obstacles or improve grasp accessibility [27], [48]. This synergy motivates us to combine grasping with auxiliary actions.

2.3 Pushing

Pushing is an example of non-prehensile actions that robots can apply on objects. As with grasping, there are two main categories of methods that predict the effect of a push

action [49]. *Analytical* methods rely on mechanical and geometric models of the objects and utilize physics simulations to predict the motion of an object [50]–[55]. Notably, Mason [51] derived the voting theorem to predict the rotation and translation of an object pushed by a point contact. A stable pushing technique when objects remain in contact was also proposed in [52]. These methods often make strong assumptions such as quasi-static motion and uniform friction coefficients and mass distributions. To deal with non-uniform frictions, a regression method was proposed in [56] for identifying the support points of a pushed object by dividing the support surface into a grid.

The *limit surface* plays a crucial role in the mechanical models of pushing. It is a convex set of all friction forces and torques that can be applied to an object in quasi-static pushing. The limit surface is often approximated as an ellipsoid [57], or a higher-order convex polynomial [58]–[60]. An ellipsoid approximation was also used to simulate the motion of a pushed object to perform a push-grasp [61]. To overcome the rigid assumptions of analytical methods, *statistical learning* techniques predict how new objects behave under various pushing forces by generalizing observed motions in training examples. For example, a Gaussian process was used to solve this problem in [25], but was limited to isolated single objects. Most recent push prediction techniques rely on deep learning [62]–[64], which can capture a wider range of physical interactions from vision. Deep RL was also used for learning pushing strategies from images [65]–[68].

2.4 Push-Grasping

Combining pushing and grasping in a single framework has led to more robust performance in cluttered settings. Two common paradigms are pre-grasp push, where non-prehensile actions reposition or uncover a target object for an ensuing grasp, and push-assisted grasping, where the push directly aids in achieving a stable grasp. Several techniques implement a push-then-grasp sequence to reduce clutter around a target object [28], [29], [69]–[71]. Others focus on 'push-grasp' actions that simultaneously slide and lift the object [61]. The

Visual Pushing and Grasping (VPG) framework [27] is a well-known example that uses a model-free Q-learning approach to select push or grasp actions. Extensions to VPG further incorporate predictive models of how objects move under push, allowing for 'look-ahead' planning rather than purely reactive decisions. In such model-based setups, the robot can simulate pushing outcomes before deciding whether and where to push, thereby avoiding unnecessary actions in cluttered environments.

2.5 Singulation

2.5.1 Definition and Techniques

Singulation refers to isolating one or more specific objects from a cluttered collection [72]. By clearing the target's immediate surroundings, singulation can make grasping or retrieval more straightforward. A common approach is to use a combination of pushing and grasping actions to move objects that are obstructing the move. The methods in [73]–[75] often rely on model-free reinforcement learning (RL) policies that reactively push objects away until the target is exposed.

2.5.2 Limitations and Extensions

These reactive techniques can be effective in lightly cluttered or moderate-density scenes, where a single push often suffices to create sufficient clearance [72]. However, for more densely packed scenarios or targets that require repeated, strategically chosen pushes, short-sighted or purely reactive methods may underperform. This limitation has spurred research on longer-horizon or model-based push planning to enable more intelligent singulation strategies.

2.6 Object Retrieval

2.6.1 Problem Setting and Challenges

Object retrieval tasks aim to locate and extract a target object from clutter. In many real-world applications (e.g., warehouse order fulfillment, household assistance), objects are stacked or partially occluded, necessitating a sequence of deliberate actions to uncover and grasp the target. This process often involves pushing, rearranging, or even removing other objects from the workspace.

2.6.2 Model-Free vs. Model-Based Approaches

Early methods treat object retrieval as an online planning challenge under partial observability [61], sometimes relying on search heuristics to reduce the solution space. Others explore model-free RL to select among push, poke, or grasp actions [76]–[78]. While these methods can learn effective strategies for moderate clutter, long-horizon reasoning is often limited. More recent work integrates predictive models to anticipate how objects will move under certain pushes [12], or uses MCTS to systematically explore multi-step action sequences [79]. Explicitly modeling future states is especially beneficial in tightly packed scenes, where small changes can substantially affect the feasibility of extracting the target.

2.7 Rearrangement Planning

Rearrangement planning extends object retrieval to more general problems of reconfiguring multiple objects from an initial to a goal arrangement [79]–[87]. In tabletop scenarios, rearranging objects often requires carefully allocating free space, deciding which objects to move first, and determining how to move them. Many methods rely on pick-and-place only, treating pushing as secondary or ignoring it entirely [18], [88], [89]. However, pick-and-place can be inefficient or infeasible when objects are heavy, large, or extremely cluttered. Some approaches reduce complexity by removing objects from the workspace

altogether [86] or by temporarily using external space to hold objects. When dealing with tightly packed configurations, heuristic-guided search has been used [18], [88]. Graph-based formulations can capture dependencies among objects that block one another [80], [81]. Beyond classic motion planning frameworks, data-driven or learning-based rearrangement methods have emerged, harnessing deep neural networks or reinforcement learning to guide action selection [79].

2.8 Task and Motion Planning (TAMP)

Task and motion planning (TAMP) deals with orchestrating high-level actions (tasks) while simultaneously ensuring geometric and kinematic feasibility (motion planning) [90]–[93]. Compared to discrete, rule-based domains (e.g., board games), TAMP operates over continuous state and action spaces. Traditional TAMP approaches often combine symbolic reasoning with sampling-based motion planning [94], [95]. More recent work leverages learning—such as learning to guide search, predict outcomes of actions, or estimate feasibility—to navigate the large search space [12], [96]–[99]. In the context of object retrieval or rearrangement, TAMP can formalize the problem: the task layer decides which object to move and how, while the motion planner ensures a collision-free trajectory for each manipulation primitive. When clutter is dense, the space of feasible moves can be large, making efficient search strategies or learned heuristics crucial.

2.9 Monte Carlo Tree Search (MCTS) for Manipulation

Monte Carlo Tree Search (MCTS) has shown promise in multi-step decision-making for manipulation, particularly in cluttered scenes [79]. MCTS incrementally expands a lookahead search tree, simulating potential action sequences to estimate their outcomes (e.g., clearing clutter to expose the target). Often, these simulations rely on predictive models—either analytical or learned—to approximate object dynamics, collisions, and future states [25], [62]–[64], [79]. When integrated with data-driven push or grasp predictors, MCTS can

efficiently explore extended action sequences, balancing exploration of different moves with exploitation of promising trajectories [78], [79]. Recent work also explores network-based predictions of multi-object collisions under pushing to accelerate the simulation phase [65]–[68]. Nevertheless, designing accurate predictive networks remains challenging in highly cluttered or diverse object sets [100]. Overall, MCTS-based approaches hold substantial potential for manipulation tasks that require long-horizon reasoning, such as object retrieval or complex rearrangement, by combining learned predictive models with systematic search to reduce trial-and-error in the physical environment.

By unifying insights from these diverse research areas—prehensile and non-prehensile actions, pushing and grasping, singulation, object retrieval, rearrangement, TAMP, and MCTS-based planning—we see that integrated strategies are critical for addressing the challenges posed by dense clutter. The remainder of this dissertation builds on these findings, focusing on how to combine model-based predictions, data-driven methods, and efficient planning techniques to enable robust, long-horizon manipulation in real-world settings.

CHAPTER 3
DIPN: DEEP INTERACTION PREDICTION NETWORK WITH APPLICATION TO CLUTTER REMOVAL

3.1 Introduction

We propose a Deep Interaction Prediction Network (DIPN) for learning object interactions directly from examples and using the trained network for accurately predicting the poses of the objects after an arbitrary push action (Figure 3.1). To demonstrate its effectiveness, We integrate DIPN with a deep Grasp Network (GN) for completing challenging clutter removal manipulation tasks. Given grasp and push actions to choose from, the objective is to remove all objects from the scene/workspace with a minimum number of actions. In an iteration of push/pick selection (Figure 3.1c), the system examines the scene and samples a large number of candidate grasp and push actions. Grasps are immediately scored by GN, whereas for each candidate push action, DIPN generates an image corresponding to the predicted outcome. In a sense, DIPN "imagines" what happens to the current scene if the robot executes a certain push. The predicted future images are also scored by GN; the action with the highest expected score, either a push or a grasp, is then executed.

Our extensive evaluation demonstrates that DIPN can accurately predict objects' poses after a push action with collisions, resulting in less than 10% average single object pose error in terms of IoU (Intersection-over-Union), a significant improvement over the compared baselines. Push prediction by DIPN generates clear synthetic images that can be used by GN to evaluate grasp actions in future states. Together with GN, our entire pipeline achieves 34% higher completion rate, 20.9% higher grasp success rate, and 30.4% higher action efficiency in comparison to [27] on challenging clutter removal scenarios. Moreover, experiments suggest that DIPN can learn from randomly generated scenarios with the learned policy

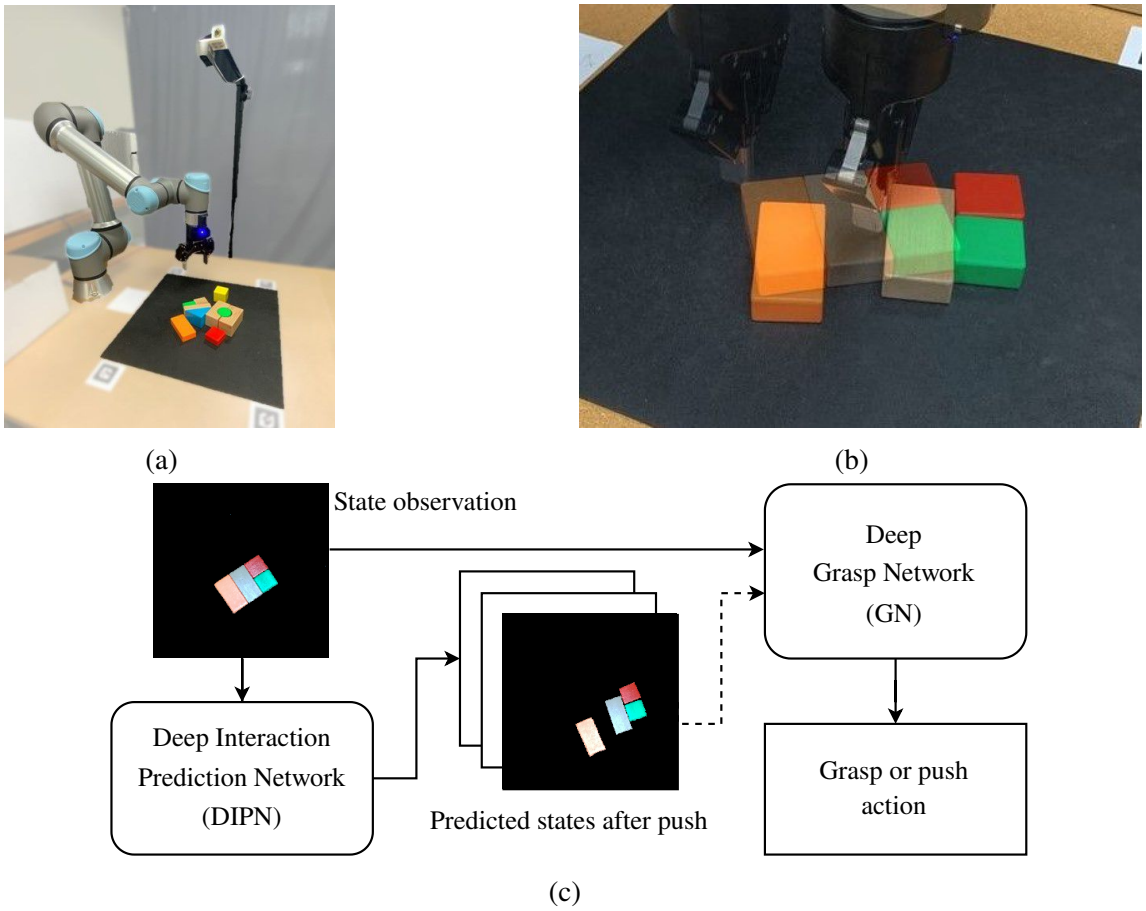


Figure 3.1: (a) The system setup includes a workspace with objects to remove, a Universal Robots UR-5e manipulator with a Robotiq 2F-85 two-finger gripper, and an Intel RealSense D435 RGB-D camera. (b) An example push action and superimposed images of scenes before and after the push. (c) System architecture of our pipeline, and one predicted image that DIPN can generate for the push shown in (b). Notice the similarity between the predicted synthetic image and the real image resulting from the push action.

maintaining high levels of performance on challenging tasks involving previously unseen objects. Remarkably, DIPN+GN achieves even better performance on real robotic hardware than in the simulation environment where it was developed.

3.2 Problem Formulation

We formulate the clutter removal problem (Figure 3.1a) as Pushing Assisted Grasping (PAG). In a PAG, the workspace of the manipulator is a square region containing multiple objects and the volume directly above it. A camera is placed on top of the workspace for state

observation. Given camera images, all objects must be removed using two basic motion primitives, grasp, and push, with a minimum number of actions.

In our experimental setup, the workspace has a uniform background color and the objects have different shapes, sizes, and colors. The end-effector is a two-finger gripper with a narrow stroke that is slightly larger than the smallest dimension of individual objects. Objects are removed one by one, which requires a sequence of push and grasp actions. When deciding on the next action, a state observation is given as an RGB-D image, re-projected orthographically, cropped to the workspace's boundary, and down-sampled to 224×224.

From the down-sampled image, a large set of candidate actions is generated by considering each pixel in the image as a potential center of a grasp action or initial contact point of a push action. A grasp action $a^{\text{grasp}} = (x, y, \theta)$ is a vertical top-down grasp centered at pixel position (x, y) with the end-effector rotation set to θ around the vertical axis of the workspace; a grasped object is subsequently transferred outside of the workspace and removed from the scene. Similarly, a push action $a^{\text{push}} = (x, y, \theta)$ is a horizontal sweep motion that starts at (x, y) and proceeds along θ direction for a fixed distance. The orientation θ can be one of 16 values evenly distributed between 0 and 360 degrees. That is, the entire action space includes $2 \times 224 \times 224 \times 16$ different grasp/push actions.

The problem studied in this paper is defined as:

Problem 1 Pushing Assisted Grasping (PAG). Given objects in clutter within the described system setup, determine a sequence of push and grasp actions, using only visual input from the workspace, to remove all objects while minimizing the total number of actions executed.

3.3 Methodology

We describe the Deep Interaction Prediction Network (DIPN), the Grasp Network (GN), and the integrated pipeline for solving PAG challenges.

3.3.1 Deep Interaction Prediction Network (DIPN)

The architecture of our proposed DIPN is outlined in Figure 3.2. At a high level, given an image and a candidate push action as inputs, DIPN segments the image and then predicts 2D transformations (translations and rotations) for all objects, and particularly for those affected by the push action, directly or indirectly through a cascade of object-object interactions. A predicted image of the post-push scene is synthesized by applying the predicted transformations on the segments. We opted against an end-to-end, pixel-to-pixel method as such methods (e.g., [62]) often lead to blurry or fragmented images, which are not conducive to predicting the quality of a potential future grasp action.

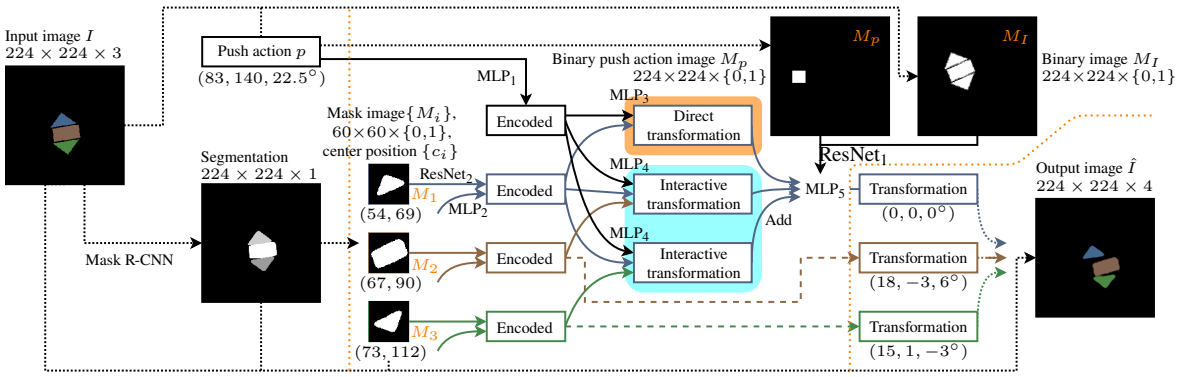


Figure 3.2: DIPN flow with an example. The network components dedicated to an object are color-coded to match the object. We only show the full network for the blue triangle object; the instance-specific structures for the other objects share the same weights and are simplified as dashed lines. Components inside the orange dotted line are the core of the DIPN. The output image is synthesized by applying the predicted transformations to the object segments.

Segmentation. DIPN employs Mask R-CNN [101] for object segmentation (instance level only, without semantic segmentation). The resulting binary masks (m_i) and their centers (c_i), one per object, serve as the input to the push prediction module of DIPN. Our Mask R-CNN setup has two classes, one for the background and one for the objects. The network is trained from scratch in a *self-supervised* manner without any human intervention: objects are randomly dropped into the workspace, and data is automatically collected. Images that can be easily segmented into separate instances based on color/depth information (distinct

color blobs) are automatically labeled by the system as single instances for training the Mask R-CNN. The self-trained Mask R-CNN can then accurately find edges in images of tightly packed scenes and even in scenes with novel objects. Note that the data used for training the segmentation module are also counted in our evaluation of the data efficiency of our technique and the comparisons to alternative techniques.

Push sampling. Based on foreground segmentation, candidate push actions are generated

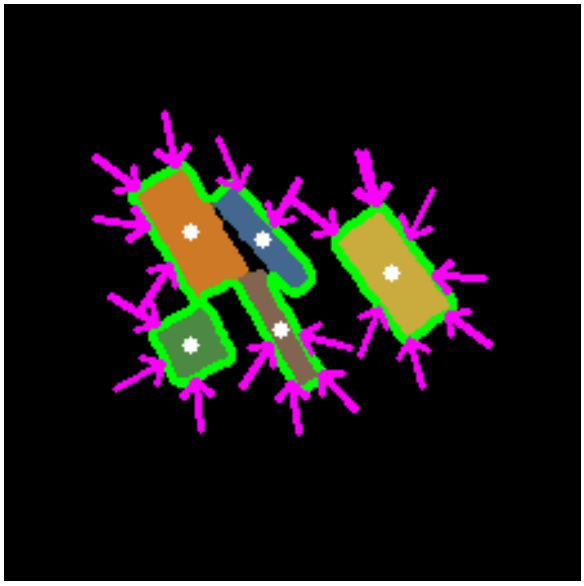


Figure 3.3: Sampled action in purple arrows around each object.

by uniformly sampling push contact locations on the contour of object clusters (see Figure 3.3). Push directions point to the centers of the objects. Pushes that cannot be performed physically, e.g., from the inside of an object bundle or in narrow spaces between objects, are filtered out based on the masks returned by R-CNN. In the figure, for example, samples between the two object clusters are removed. A sampled push is defined as $p = (x, y, \theta) \in SE(2)$ where x, y are the start location of the push and θ indicates the horizontal push direction. The push distance is fixed.

Input to push prediction. The initial scene image I, scene object masks and centers (m_i, c_i), and a sampled push action $p = (x, y, \theta)$ are the main inputs to the push prediction module. To reduce redundancy, we transform all inputs using a 2D homogeneous transfor-

mation matrix T such that $pT = (40, 112, 0)$. The position of the push is normalized for easier learning such that a push will always go from left to right in the middle of the left side of the workspace. From here, it is understood that all inputs are with respect to this updated coordinate frame defined by T, i.e., $p \leftarrow pT, c_i \leftarrow c_i T$, and so on. Apart from the inputs mentioned so far, we also generate: (1) one binary *push action* image M_p with all pixels black except in a small square with top-left corner $(40, 100)$ and bottom-right corner $(65, 124)$ which is the finger movement space, (2) one 224×224 binary image M_I with the foreground of I set to white, and (3) one 60×60 binary mask image M_i for each object mask m_i, centered at c_i. Despite being constant relative to the image transformed by T, push image M_p is used as an input because we noticed from our experiments that it helps the network focus more on the pushing area.

Push prediction. With global (binary images M_p, M_I) and local (mask image M_i and the center c_i of each object) information, DIPN proceeds to predict objects' transformations. To start, a Multi-Layer Perceptron (MLP) and a ResNet [102] (with no pre-training) are used to encode the push action and the global information, respectively:

$$e_p = \text{MLP}_1(p), \quad e_{AB} = \text{ResNet}_1(M_p, M_I).$$

A similar procedure is applied to individual objects. For each object o_i, its center c_i and mask image M_i are encoded using ResNet (again, with no pre-training) and MLP as:

$$e_i = (\text{ResNet}_2(M_i), \text{MLP}_2(c_i)).$$

Adopting the design philosophy from [64], the encoded information is then passed to a *direct transformation* (DT) MLP module (blocks in Figure 3.2 with orange background) and multiple *interactive transformation* (IT) MLP modules (blocks in Figure 3.2 with cyan background):

$$\forall 1 \leq i \leq n: \ \text{DT}_i = \text{MLP}_3(e_p, e_i),$$

$$\forall 1 \leq i, j \leq n, j \neq i: \ \text{IT}_{ij} = \text{MLP}_4(e_p, e_i, e_j).$$

Here, the direct transformation modules capture the effect of the robot *directly* touching the objects (if any), while the interactive transformation modules consider collision between an object and every other object (if any). Then, all aforementioned encoding is put together to a decoding MLP to derive the output 2D transformation for each object o_i in the push action's frame: $\forall 1 \leq i \leq n$,

$$(\hat{x}_i, \hat{y}_i, \hat{\theta}_i) = \text{MLP}_5(e_{AB}, \text{DT}_i + \sum_{1 \leq j \leq n, j \neq i} \text{IT}_{ij}),$$

which can be mapped back to the original coordinate frame via T^{-1}. This yields predicted poses of objects. From these, an "imagined" push prediction image is readily generated.

In our implementation, both ResNet_1 and ResNet_2 are ResNet-50. MLP_1 and MLP_2, encoding the push action and single object position, both have two (hidden) layers with sizes 8 and 16. MLP_3, connecting encoded and direct transformations, has two layers of a uniform size 128. MLP_4, connecting encoded and interactive transformations, has three layers with a size of 128 each. The final decoder MLP_5 has five layers with sizes [256, 64, 32, 16, 3]. The number of objects n varies across scenes. The network handles a variable number of objects because the same weight-shared networks ($\text{MLP}_{2\text{-}5}$ and ResNet_2) process each object and object pair.

Training. For training in simulation and for real experiments, objects are randomly dropped onto the workspace. The robot then executes random pushes to collect training data. *SmoothL1Loss* (Huber Loss) is used as the loss function. Given each object's true post-push transformation (x_i, y_i, θ_i) and the predicted $(\hat{x}_i, \hat{y}_i, \hat{\theta}_i)$, the loss is computed as the sum of coordinate-wise SmoothL1Loss between the two. DIPN performs well on unseen objects and can be completely trained in simulation and transferred to the real-world (Sim-to-Real). It is also robust with respect to changes in objects' physical properties, e.g., variations in mass and friction coefficients.

3.3.2 The Grasp Network (GN)

We briefly describe GN, which shares a similar architecture to the DQN used in [27]. Given an observed image and candidate grasp actions, GN finds the optimal policy for maximizing a single-step grasp reward, defined as 1 for a successful grasp and 0 otherwise. GN focuses its *attention* on local regions that are relevant to each single grasp and uses image-based self-supervised pre-training to achieve a good initialization of network parameters.

The proposed modified network's architecture is illustrated in Fig. Figure 3.4. It takes an input image and outputs a score for each candidate grasp centered at each pixel. The input image is rotated to align it with the end-effector frame (see the left image in Fig. Figure 3.4). ResNet-50 FPN [102] is used as the backbone; we replace the last layer with our own customized head structure shown in Fig. Figure 3.4. We observe that our structure leads to faster training and inference time without loss of accuracy. Given that the network computes pixel-wise values in favor of a local grasp region, at the end of the network, we place two convolutional layers with kernel size 11×57, which was determined based on the clearance of the gripper.

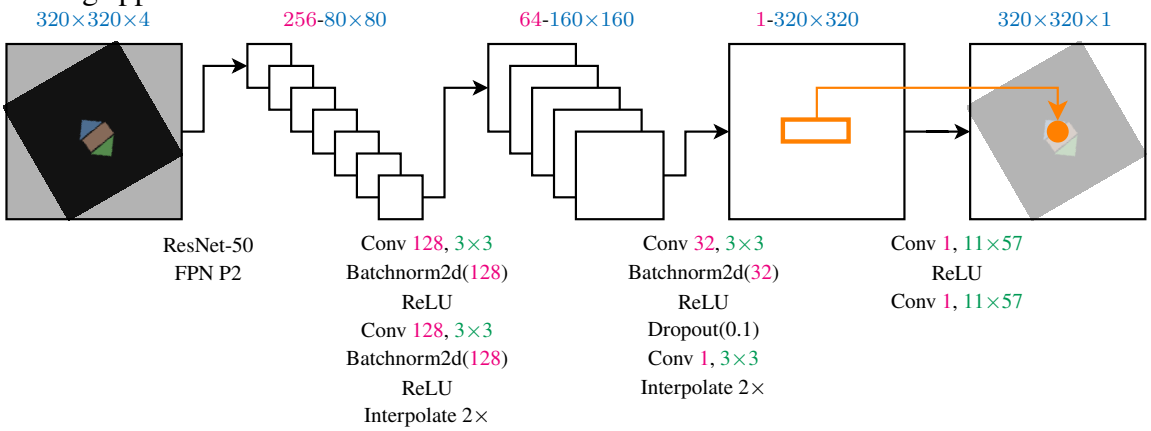


Figure 3.4: Architecture of GN. Pink, blue, and green text are used for channel count, image size, and kernel size, respectively.

In training GN, image-based pre-training [103] was employed. The pre-training process treats pixel-wise grasping as a vision task to obtain a good network initialization. The process automatically labels with 0 or 1 all the pixels in a small set of arbitrary images,

depending on whether grasps centered at each pixel would lead to a finger collision with an object, based only on color/depth and without actually simulating or executing the grasps physically. The pre-training data set does not include objects used for testing.

3.3.3 The Complete Algorithmic Pipeline

The training process of DIPN+GN is outlined in Algorithm 1. In line 1, an image data set is collected for training Mask R-CNN (i.e., for push prediction segmentation) and initializing (i.e., pre-training) GN. Note that training data for Mask R-CNN and pre-training data for GN are essentially free, with no physics involved. After pre-training, the training process for push (line 2) and grasp (line 3) predictions can be executed on PAG scenes in any order.

Algorithm 1: Training DIPN+GN

Output: trained DIPN+GN.

1 GN, Mask R-CNN$\leftarrow$GetImageDataSetAndPre-Train ()
2 DIPN $\leftarrow$ TrainOnPAGPushOnly (Mask R-CNN)
3 GN $\leftarrow$ TrainOnPAGGraspOnly (GN)

The high-level workflow of our framework on PAG is described in Algorithm 2. When working on an instance, at every decision-making step t, an image M_t is first obtained (line 2). Then, the image M_t, along with sampled push actions A^{push}, are sent to the trained DIPN to generate predicted synthetic images $\hat{M}_{t+1}$ after each imagined push a (line 3-line 4). With A^{grasp} denoting the set of all grasp actions, their discounted average reward on the predicted next image $\hat{M}_{t+1}$ is then compared with the average of grasping rewards in the current image (line 6): recall that GN takes an image and a grasp action as input, and outputs a scalar grasp reward value. If there exists a push action with a higher expected average grasping reward in the predicted next image, the best push action is then selected and executed (line 7); otherwise, the best grasp action is selected and executed (line 8). Because it is desirable to have a single push action that simultaneously renders multiple objects graspable, the average grasp reward is used instead of only the maximum.

The framework contains two hyperparameters. The first one, γ, is the discount factor of

Algorithm 2: Executing DIPN+GN

Input: trained GN and DIPN, discount factor γ

while there are objects in workspace **do**
  $A^{\text{push}} \leftarrow \varnothing$, $M_t \leftarrow$ GetImage ();
  for a in SamplePushActions (M_t) **do**
    $A^{\text{push}} \leftarrow A^{\text{push}} \cup \{a\}$; $\hat{M}_{t+1} \leftarrow$ DIPN (M_t, a);
    $Q(M_t, a) = \frac{\gamma}{|A^{\text{grasp}}|} \sum_{a' \in A^{\text{grasp}}} \text{GN}(\hat{M}_{t+1}, a')$;
  if $\max_{a \in A^{\text{push}}} Q(M_t, a) > \frac{1}{|A^{\text{grasp}}|} \sum_{a' \in A^{\text{grasp}}} \text{GN}(M_t, a')$ **then**
    Execute $\arg\max_{a \in A^{\text{push}}} Q(M_t, a)$;
  else Execute $\arg\max_{a \in A^{\text{grasp}}} \text{GN}(M_t, a)$;

the Markov Decision Process. For a push action to be selected, the estimated discounted grasp reward after a push must be larger than grasping without a push, since the push and then grasp takes two actions. In our implementation, we set γ to be 0.9. The other *optional* hyperparameter is used for accelerating inference: if the maximum grasp reward is higher than a threshold, we directly execute the grasp action without calling the push prediction. This hyperparameter requires tuning. In our implementation, the threshold value is set to be 0.7. Note that the maximum reward for a single grasp is 1.

3.4 Experimental Evaluation

We first evaluate GN and DIPN separately and then compare the full system's performance with the state-of-the-art model-free RL technique presented in [27], which is the closest work to ours. Apart from evaluating our approach on a real robotic system, we also conducted extensive evaluations in the CoppeliaSim [104] simulator. We use an Nvidia GeForce RTX 2080 Ti graphics card to train and test the algorithms. All simulation experiments are repeated 30 times; all real experiments are repeated for 5 times to get the mean metrics.

3.4.1 Deep Interaction Prediction Network (DIPN)

To evaluate how accurately DIPN can predict the next image after a push action, DIPN is first trained on randomly generated PAG instances in simulation. At the start of each episode,

randomly generated objects with random colors and shapes are randomly dropped from mid-air to construct the scene. Up to 7 objects are generated per scene. We calculate the prediction accuracy by measuring the *Intersection-over-Union* (IoU) between a predicted image and the corresponding ground-truth after pushing. The IoU calculation is performed at the object level and then averaged.

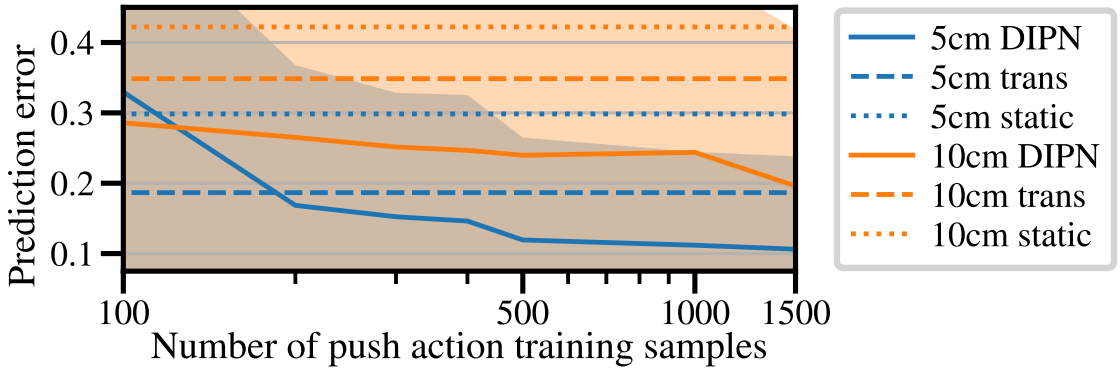


Figure 3.5: DIPN learning curve with standard deviation shown as shaded regions. The x-axis is the number of pushes for training DIPN. The y-axis is the prediction error: $1 - \text{IoU}$. The dotted and dashed lines are baselines.

DIPN is compared with two baselines: the first one, called *static*, assumes that all objects stay still. The second one, called *trans*, always assumes that only the pushed object moves, and that it moves exactly by the push distance along the push direction. Both baselines are engineered methods that do not require training. The push prediction errors ($1 - \text{IoU}$) are illustrated in Figure 3.5 as learning curves in simulation. Mask R-CNN is trained (i.e., Algorithm 1, line 1) using an additional 100 images, which is why Figure 3.5 starts from 100. We observe, for different push distances, that DIPN outperforms the baselines with a large margin after sufficient training. After convergence, the prediction error for DIPN is less than 0.1 for a 5cm push, which indicates that the predicted pose of an object overlaps $90\%+$ with the ground truth. As expected, DIPN is more accurate and more sample efficient with a shorter push distance. On the other hand, longer push distances generally result in better overall performance for PAG challenges even though push predictions become less accurate, since larger actions are more effective in terms of changing the scene.

Figure 3.6 shows typical predictions by DIPN. The network is learned in simulation with randomly shaped and colored objects, and directly transferred to the real system. We observe that DIPN can accurately predict the state after a push, with good accuracy on object orientation and translation.

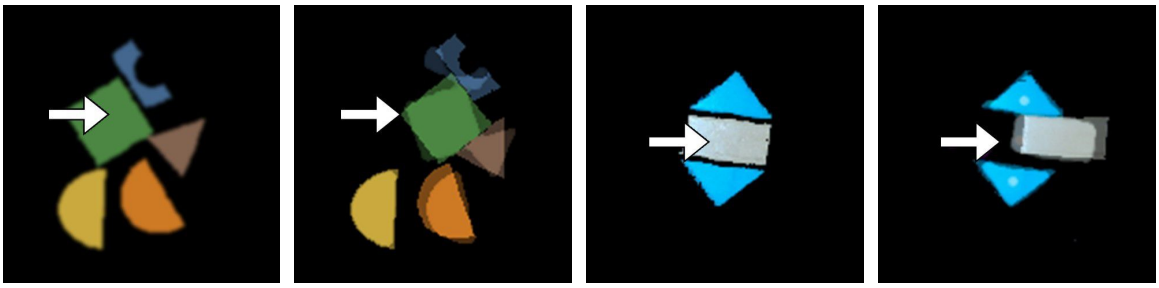


Figure 3.6: Typical DIPN results. The figures from left to right are: original and predicted images in simulation, and original and predicted images in a real experiment. The ground truth images after a push are overlaid on the predicted images with transparency. The arrows visualize the push actions.

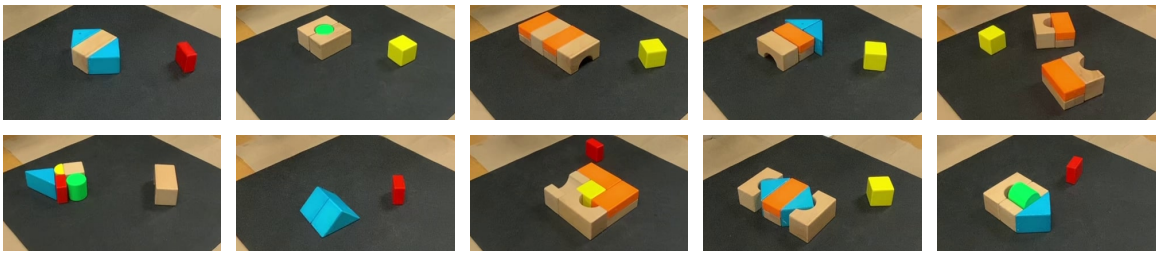


Figure 3.7: Manually generated hard instances largely similar to the ones in [27]. The cases are used in both simulation and real experiment.

3.4.2 Grasp Network (GN)

We train and evaluate GN as a standalone module and compare it with the state-of-the-art DQN-based method known as Visual Pushing and Grasping (VPG) [27], which learns both grasp and push at the same time. Since GN is only trained on grasp actions, and for a fair comparison, we also tested a third method that learns both grasp and push actions: this method, denoted by DQN+GN, uses GN for learning grasp actions and the DQN structure of [27] to learn push actions. The algorithms are compared using the grasp success rate

metric, i.e., *the number of objects removed* divided by *the total number of grasps*. We train all algorithms directly on randomly generated PAG instances with 10 objects.

The learning curve in simulation is provided in Figure 3.8. The pre-training process (Algorithm 1, line 1, which is also self-supervised) for GN takes 100 offline images that are not reported in the plot. Comparing DQN+GN which reaches $> 90\%$ success rate with less than 300 (grasp and push) samples, and baseline VPG, which converges at 82% success rate with more than 2000 (grasp and push) samples, it is clear that GN has significantly higher grasp success rate and sample efficiency than the baseline VPG. As shown by the comparison between GN and DQN+GN, when training using only grasp actions, GN can be more sample efficient without sacrificing success rate. The result also indicates that for randomly generated PAG, pushing is often *unnecessary*.

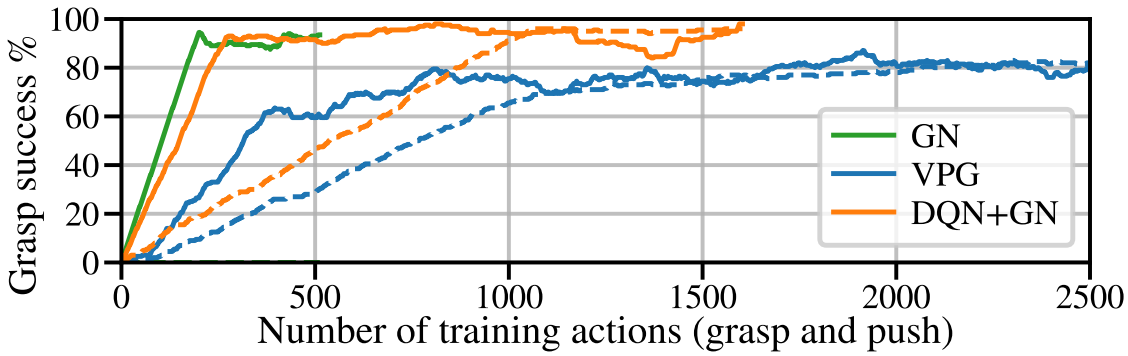


Figure 3.8: Grasp learning curves of algorithms for PAG in simulation. The x-axis is the total number of training steps, i.e., number of actions taken, including push and grasp. The y-axis is the grasp success rate. The dashed lines denote the success rate for a grasp right after a push action.

3.4.3 Evaluation of the Complete Pipeline

We evaluate the learned policies on PAG with up to 30 objects, first in a simulation, then on a real system. Four algorithms are tested: VPG [27], DQN+GN, REA+DIPN, and DIPN+GN (our full pipeline). Here, REA+DIPN follows Algorithm 2 but uses the reactive grasp network from [27] instead of GN for grasp reward estimation. We use three metrics

for comparison: *(i) Completion*, calculated as *the number of PAG instances where all objects got removed* divided by *the total number of instances*; incomplete tasks typically occur if objects are pushed out of the workspace, or if the task is not completed within a predefined action limit (e.g., three times the number of objects). *(ii) Grasp success*, calculated as *the total number of objects grasped* divided by *the total number of grasp actions*. *(iii) Action efficiency*, calculated as *the total number of objects removed* divided by *the total number of actions (grasp and push)*. For grasp success rate and action efficiency, we use two formulations: one does not count incomplete tasks (reported in gray text), which is the same as the one used in [27], and the other one counts incomplete tasks, which we believe is more reflective. The robot could grasp more than one object at a time. We consider it as a successful grasp, as the goal is to clear all objects from the table.

Table 3.1 reports simulation results on 30 randomly generated PAG instances and 10 manually placed hard instances (illustrated in Figure 3.7) where push actions are necessary. The algorithms were not trained on these hard instances. The number of training samples for each algorithm are: 2500 actions (grasp and push) for VPG [27], 1500 grasp actions and 2000 push actions for REA+DIPN, 1500 actions (grasp and push) for DQN+GN, and 500 grasp actions and 1500 push actions for DIPN+GN (our full pipeline). The results show that DIPN and GN both are sample efficient in comparison with the baseline and provide significant improvement in PAG metrics; when combined, DIPN+GN reaches the highest performance on all metrics.

We repeated the evaluation on a real system (see. Figure 3.1a). Each random instance contains 10 randomly selected objects; the hard instances are shown in Figure 3.7. Figure 3.9 shows grasp learning curve. We compare VPG [27] (trained with 2000 grasp and push actions) and the proposed DIPN+GN pipeline (pre-trained with 100 unlabeled RGB-D images for segmentation, trained GN with 500 grasp actions and DIPN with 1500 simulated push actions). The evaluation result is reported in Table 3.2. Remarkably, our networks, while being developed using only simulation based training, perform even better when

trained/evaluated only on real hardware.

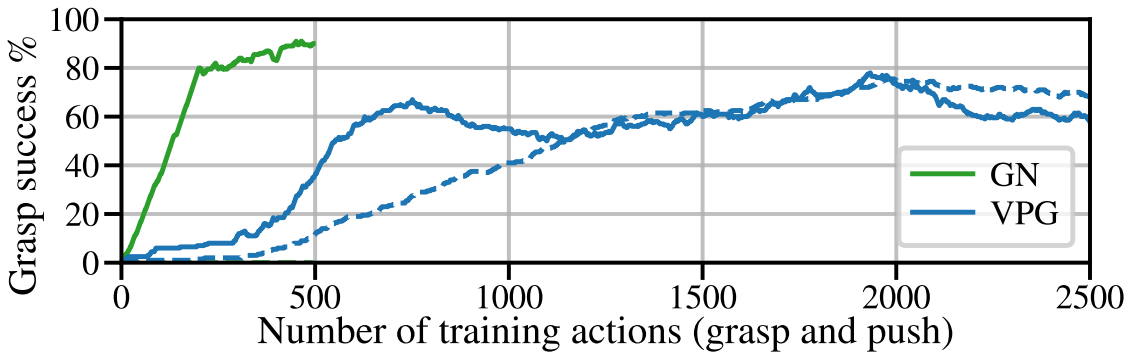


Figure 3.9: Grasp learning curves for PAG in real experiment. Solid lines indicate grasp success rate and dotted lines indicate push-then-grasp success rates over training steps. The GN is trained in a grasp only manner.

Table 3.1: Simulation, random and hard instances (mean %)

	Method	Completion	Grasp success		Action efficiency	
Rand	VPG[27]	20.0[1]	69.0	52.6	66.3	52.6
	REA[27]+DIPN	83.3	79.5	77.9	77.4	76.3
	DQN[27]+GN	46.7	85.2	83.9	83.4	81.7
	DIPN+GN	**83.3**	**86.7**	**85.2**	**84.4**	**83.3**
Hard	VPG[27]	77.7	67.4	60.0	60.8	57.6
	REA[27]+DIPN	90.3	81.5	76.6	64.7	62.6
	DQN[27]+GN	86.0	91.1	87.1	70.2	67.9
	DIPN+GN	**100.0**	**93.3**	**93.3**	**74.4**	**74.4**

Table 3.2: Real system, random and hard instances (mean %)

	Method	Completion	Grasp success		Action efficiency	
Rand	VPG[27]	80.0	85.5	79.0	75.3	67.9
	DIPN+GN	**100.0**	**94.0**	**94.0**	**98.2**	**98.2**
Hard	VPG[27]	64.0	75.1	69.0	51.9	47.8
	DIPN+GN	**98.0[2]**	**89.9**	**89.9**	**77.6**	**78.2**

With DIPN and GN outperforming the corresponding components from VPG [27], it is unsurprising that DIPN+GN does much better. In particular, DIPN architecture allows it to

[1]The low completion rate is primarily due to pushing objects outside of the workspace.

[2]The single failure was due to an object that was successfully grasped but slipped out of the gripper before the transfer was complete.

learn intelligent, graded push behavior efficiently. In contrast, VPG [27] has a fixed 0.5 push reward, which sometimes negatively impacts performance: VPG could push unnecessarily for many times without a grasp when it is not confident enough to grasp. It also risks pushing objects outside of the workspace. Following [27], we tested DIPN+GN with previously unseen objects, such as soapboxes and plastic bottles. Our method maintained a similar level of performance to that reported in Table 3.2.

3.5 Summary

In this work, we have developed a Deep Interaction Prediction Network (DIPN) for learning to predict the complex interactions that occur as a robot manipulator pushes objects in clutter. Unlike most existing end-to-end techniques, DIPN is capable of generating accurate predictions in the form of clearly legible synthetic images that can be fed as inputs to a deep Grasp Network (GN), which can then predict successes of future grasps. We demonstrated that DIPN, GN, and DIPN+GN all have excellent sample efficiency and significantly outperform the previous state-of-the-art learning-based method for PAG challenges, while using only a fraction of the interaction data used by the alternative. Our networks are trained in a fully self-supervised manner, without any manual labeling or human inputs, and exhibit high levels of generalizability. The proposed system, initially developed in simulation, also performs effectively when trained and deployed on real hardware with physical objects. DIPN+GN demonstrates high robustness to variations in object properties such as shape, size, color, and friction.

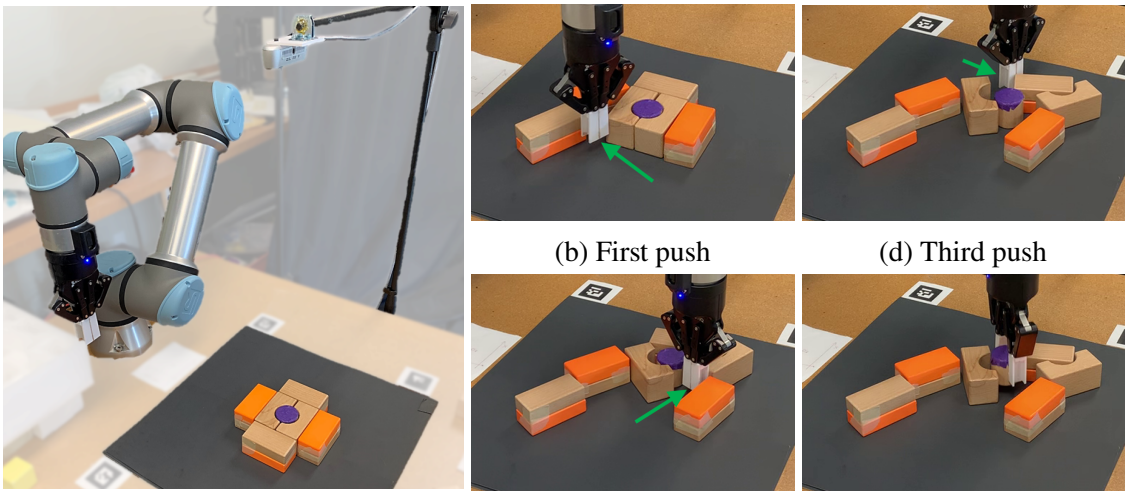
CHAPTER 4
VISUAL FORESIGHT TREES FOR OBJECT RETRIEVAL FROM CLUTTER WITH NONPREHENSILE REARRANGEMENT

4.1 Introduction

In many application domains, robots are tasked with retrieving objects that are surrounded by multiple tightly packed objects. To enable the grasping of target object(s), a robot needs to rearrange the scene to create sufficient clearance before attempting a grasp. Scene rearrangement can be achieved through nested sequential push actions, each moving multiple objects simultaneously. In this paper, we address the problem of finding the minimum number of push actions to create a scene where the target object can be grasped and retrieved.

To solve the object retrieval problem, the robot must imagine how the scene would look after any given sequence of pushing actions, and select the shortest sequence that leads to a state where the target object can be grasped. The huge combinatorial search space makes this problem computationally challenging, hence the need for efficient planning algorithms, as well as fast predictive models that can return the predicted future states in a few milliseconds. Moreover, objects in clutter typically have unknown physical properties such as mass and friction coefficients. While it is possible to utilize off-the-shelf physics engines to simulate contacts and collisions of rigid objects in clutter, simulation is highly sensitive to the accuracy of the provided mechanical parameters. To overcome the problem of manually specifying these parameters, and to enable full autonomy of the robot, most recent works on object manipulation utilize machine learning techniques to train predictive models from data [105]–[107]. The predictive models take the state of the robot's environment a control action as inputs and predict the state after applying the control action.

In this work, we propose to employ *visual foresight trees* (VFT) to address the compu-



(a) Hardware setup (b) First push (c) Second push (d) Third push (e) Grasp

Figure 4.1: (a) The hardware setup for object retrieval in a clutter includes a Universal Robots UR-5e manipulator with a Robotiq 2F-85 two-finger gripper, and an Intel RealSense D435 RGB-D camera. The objects are placed in a square workspace. (b), (c), (d) Three sequential push actions (green arrows) create space to access the target (purple) object. The push directions are toward top-left, top-right, and bottom-right, respectively. (e) The target object is successfully grasped and retrieved.

tational and modeling challenges related to the object retrieval problem. A key building block of VFT is a Convolutional Neural Network (CNN) extending DIPN [12], capable of predicting multi-step push outcomes involving multiple objects. A second CNN evaluates the graspability of the target object in predicted future images. A Monte Carlo Tree Search utilizes the two CNNs to obtain the shortest sequence of pushing actions that lead to an arrangement where the target can be grasped.

To our knowledge, the proposed technique is the first model-based learning solution to the object retrieval problem. Extensive experiments on a real robot with physical objects, as exemplified in Figure 4.1, demonstrate that the proposed approach succeeds in retrieving target objects with manipulation sequences that are shorter than model-free reinforcement learning techniques and a limited-horizon planning technique.

4.2 Problem Formulation

4.2.1 Problem Statement

The Object Retrieval from Clutter (ORC) challenge asks a robot manipulator to retrieve a target object from a set of objects densely packed together. The objects may have different shapes, sizes, and colors.

Objects other than the target object are unknown a prior. Focusing on a mostly planar setup, the following assumptions are made:

1. The hardware setup (Figure 4.1a) contains a manipulator, a planar workspace with a uniform background color, and a camera on top of the workspace.

2. The objects are rigid and are amenable to the gripper's prehensile and non-prehensile capabilities, limited to straight-line planar push actions and top-down grasp actions.

3. The objects are confined to the workspace without overlapping. As a result, the objects are visible to the camera.

4. The target object, to be retrieved, is visually distinguishable from the others.

Under these assumptions, the *objective* is to retrieve only the target object, while minimizing the number of pushing/grasping actions that are used. Each grasp or push is considered as one atomic action. While a mostly planar setup is assumed in our experiments, the proposed data-driven solution is general and can be applied to arbitrary object shapes and arrangements. In the experiments, we mainly work with woodblocks; we also evaluate the proposed approach on novel objects such as soapboxes, which are challenging as their widths are close to the maximum distances between the gripper's fingers.

4.2.2 Manipulation Motion Primitives

Similar to studies closely related to the ORC challenge, e.g., [12], [27], [48], we employ a set of pre-defined and parameterized pushing/grasping manipulation primitives. The

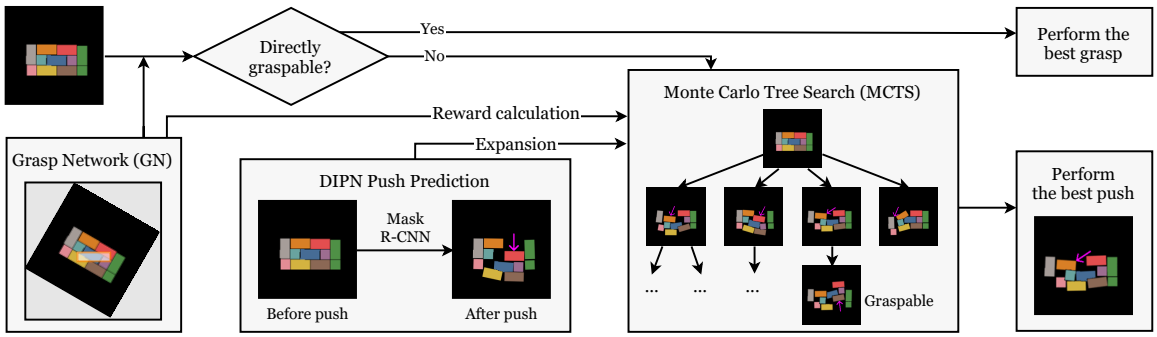


Figure 4.2: Overview of the proposed technique for object retrieval from clutter with nonprehensile rearrangement. The problem is iteratively solved by observing the environment at each time step, taking the current state as input, and returning the best action. It is repeated until the object is retrieved.

decision-making problem then entails the search for the optimal order and parameters of these primitives. A grasp action $a^{\text{grasp}} = (x, y, \theta)$ is defined as a top-down overhead grasp motion at image pixel location (x, y), with the end-effector rotated along with the world z-axis by θ degrees. In our implementation, a grasp center (x, y) can be any pixel in a down-sampled 224×224 image of the planar scene, while rotation angle θ can be one of 16 values evenly distributed between 0 and 2π. To perform a complete grasp action, the manipulator moves the open gripper above the specified location, then moves the gripper downwards until a contact with the target object is detected, closes the fingers, and transfers the grasped object outside of the workspace.

When objects are densely packed, the target object is generally not directly graspable due to collisions between the gripper and surrounding objects. When this happens, non-prehensile push actions can be used to create opportunities for grasping. For a push action $a^{\text{push}} = (x_0, y_0, x_1, y_1)$, the gripper performs a quasi-static horizontal motion. Here, (x_0, y_0) and (x_1, y_1) are the start and end location of the gripper center, respectively. The gripper's orientation is fixed along the motion direction during a push maneuver.

4.3 Methodology

4.3.1 Overview of the Proposed Approach

When objects are tightly packed, the robot needs to carefully select an appropriate sequence of pushes that create a sufficient volume of empty space around the target object before attempting to grasp it. In this work, we are interested in challenging scenarios where multiple push actions may be necessary to de-clutter the surroundings of the target, and where the location, direction, and duration of each push action should be carefully optimized to minimize the total number of actions. Collisions among multiple objects often occur while pushing a single object, further complicating the matter. To address the challenge, we propose a solution that uses a neural network to forecast the outcome of a sequence of push actions in the future, and estimates the probability of succeeding in grasping the target object in the resulting scene. The optimal push sequence is selected based on the forecasts.

A high-level description of the proposed solution pipeline is depicted in Figure 4.2. At the start of a planning iteration, an RGB-D image of the scene is taken, and the objects are detected and classified as *unknown clutter* or *target object*. With the target object located, a second network called Grasp Network (GN) predicts the probability of grasping the target. GN is a Deep Q-Network (DQN) [108] adopted from prior works [12], [27] for ORC. It takes the image input, and outputs the estimated grasp success probability for each grasp action. The target object is considered directly graspable if the maximum estimated grasp success probability is larger than a threshold. The robot executes the corresponding optimal grasp action; otherwise, push actions must be performed to create space for grasping.

When push actions are needed, the next action is selected using Monte-Carlo Tree Search (MCTS). In our implementation, which we call the Visual Foresight Tree (VFT), each search state corresponds to an image observation of the workspace. Given a push action and a state, VFT uses the Deep Interaction Prediction Network (DIPN) [12] as the state transition function. Here, DIPN is a network that predicts the motions of multiple objects and generates

a synthetic image corresponding to the scene after the imagined push. VFT uses GN to obtain a reward value for each search node and detect whether the search terminates. Both DIPN and GN are trained offline on different objects.

4.3.2 Visual Foresight Trees

This section discusses the three main components of VFT: GN, DIPN, and Monte-Carlo Tree Search (MCTS).

4.3.3 Grasp Network

The Grasp Network (GN), adapted from [12], takes the image s_t as input, and outputs a pixel-wise reward prediction $R(s_t) = [R(s_t, a^1), \ldots, R(s_t, a^n)]$ for grasps $a^1, \ldots, a^n$. The output is a 2D map with the same size as the input image, and where each point contains the predicted reward of performing a grasp at the corresponding input pixel. Table $R(s_t)$ is a single-channel image with the same size as input image s_t (224×224 in our experiments), and a value $R(s_t, a^i)$ represents the expected reward of the grasp at the corresponding action. To train GN, we set the reward to be 1 for grasps where the robot successfully picks up only the target object, and 0 otherwise. GN is the reward estimator for states in VFT (subsection 4.3.5).

A grasp action $a^{\text{grasp}} = (x, y, \theta)$ specifies the grasp location and the end-effector angle. GN is trained while keeping the orientation of the end-effector fixed relative to the support surface, while randomly varying the poses of the objects. Therefore, GN assumes that the grasps are aligned to the principal axis of the input image. To compute reward R for grasps with $\theta \neq 0$, the input image is rotated by θ before passing it to GN. As a result, for each input image, GN generates 16 different grasp R reward tables.

The training process of the GN used in this work is based on previous works [12], [27] but differs in terms of objectives, which requires a significant modification, explained in the following. The objective in previous works is to grasp all the objects; the goal of ORC

is to retrieve a specific target among a large number of obstacles. We noticed from our experiments that if GN is trained to grasp all the objects, then a greedy policy will be learned, and it will always select the most accessible object to grasp. In contrast, all other objects that can also be directly grasped are ignored because they have low predicted rewards. This causes the problem that GN cannot correctly predict the grasp success rate of a specific target object. One straightforward adaptation to this new objective is only to give reward when the grasp center is inside the target object, which is the approach that was followed in [48]. However, we found that we can achieve a higher sample efficiency by providing a reward for successfully grasping any object. The proposed training approach is similar in spirit to Hindsight Experience Replay (HER) [109]. To balance between exploration and exploitation, grasp actions are randomly sampled from $P(s, a^{\text{grasp}}) \propto bR(s, a^{\text{grasp}})^{b-1}$ where b is set to $3/2$ in the experiments.

After training, GN can be used for selecting grasping actions in new scenes. Since the network returns reward R for all possible grasps, and not only for the target object, the first post-processing step consists in selecting a small set of grasps that overlap with the target object. This is achieved by computing the overlap between the surface of the target object and the projected footprint of the robotic hand, and keeping only grasps that maximize the overlap. Then, grasps with the highest predicted values obtained from the trained network are ranked, and the best choice without incurring collisions is selected for execution.

4.3.4 Push Prediction Network

DIPN [12] is a network that takes an RGB-D image, 2D masks of objects, center positions of objects, and a vector of the starting and endpoints of a push action. It outputs predicted translations and rotations for each passed object. The predicted poses of objects are then used to create a synthetic image. Effectively, DIPN imagines what happens to the clutter if the robot executes a certain push.

The de-cluttering tasks considered in [12] required only single-step predictions. The

ORC challenge requires highly accurate predictions for multiple consecutive pushes in the future. To adapt DIPN for ORC, we fine-tuned its architecture, replacing ResNet-18 with ResNet-10 [110] while increasing the output feature dimension from 256 to 512 to predict motions of more objects simultaneously and efficiently. The number of decoder MLP layers is also increased to six, with sizes $[768, 256, 64, 16, 3, 3]$. Other augmentations are reported in section of experiments. Finally, we trained the network with $200,000$ random push actions applied on various objects. This number is higher than the $1,500$ actions used in [12] as we aim for the accuracy needed for long-horizon visual foresight. Given a sequence of candidate push actions, the fine-tuned DIPN predicts complex interactions, e.g., Figure 4.3.

4.3.5 Visual Foresight Tree Search (VFT)

We introduce DIPN for predicting single-step push outcome and GN for generating/rating grasps as building blocks for a multi-step procedure capable of long-horizon planning. A natural choice is Monte-Carlo Tree Search (MCTS) [111], which balances scalability and optimality. In essence, VFT fuses MCTS and DIPN to generate an optimal multi-step push prediction, as graded by GN. A search node in VFT corresponds to an input scene or one imagined by DIPN. MCTS prioritizes the most promising states when expanding the search tree; in VFT, such states are the ones leading to a successful target retrieval in the least number of pushes. In a basic search iteration, MCTS has four essential steps: selection, expansion, simulation, and back-propagation. First, the *selection* stage samples a search node and a push action based on a selection function. Then, the *expansion* stage creates a child node of the selected node. After that, the reward value of the new child node is determined by a *simulation* from the node to an end state. Finally, the *back-propagation* stage updates the estimated Q-values of the parent nodes.

For describing MCTS with visual foresight, let $N(n)$ be the number of visits to a node n and $Q(n) = \{r_1, \ldots, r_{N(n)}\}$ as the estimated Q-values of each visit. We use N_{max} to denote the number of iterations the MCTS performed; we may also use an alternative computational

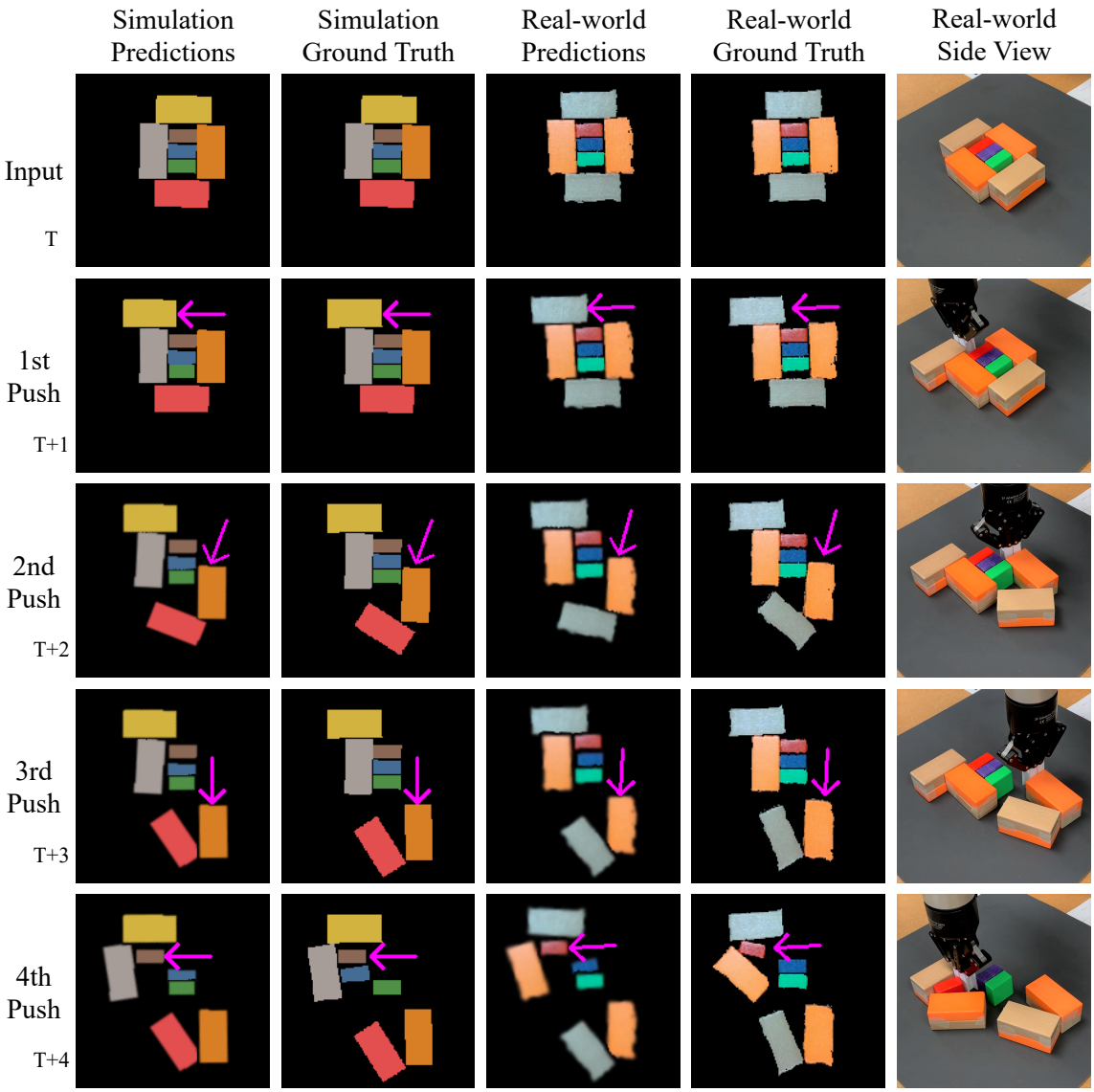


Figure 4.3: Example of 4 consecutive pushes showing that DIPN can accurately predict push outcomes over a long horizon. We use purple arrows to illustrate push actions. The first and second columns are the predictions and ground truth (objects' positions after executing the pushes) in simulation. The third and fourth columns show results on a real system. The last column is the side view of the push result. Each row represents the push outcome with the previous row as the input observation.

budget to stop the search [111]. The high-level workflow of our algorithm is depicted in Algorithm 3, and illustrated in Figure 4.2. We will describe one iteration (line 11-line 29) of MCTS in VFT along with the pseudo-code in the remaining of this section.

Selection. The first step of MCTS is to select an *expandable* search node (line 12-line 13)

Algorithm 3: Visual Foresight Tree Search

Function `VFT(`s_t`)`
  while there is a target object in workspace **do**
    $R(s_t) \leftarrow \text{GN}(s_t)$
    if $\max_{a^{\text{grasp}}} R(s_t, a^{\text{grasp}}) > R^*_g$ **then**
      Execute $\arg\max_{a^{\text{grasp}}} R(s_t, a^{\text{grasp}})$ // Grasp
    else Execute `MCTS(`s_t`)` // Push

Function `MCTS(`s_t`):`
  Create root node n_0 with state s_t
  $N(\cdot) \leftarrow 0, Q(\cdot) \leftarrow \varnothing$ // Default N, Q for a search node
  for $i \leftarrow 1, 2, \ldots, N_{max}$ **do**
    $n_c \leftarrow n_0$
    ▷ **Selection and Expansion**
    while n_c is not expandable **do**
      $n_c \leftarrow \pi_{\text{tree}}(n_c)$ // Use (Equation 4.1) to find a child node
    $a^{\text{push}} \leftarrow$ sample from untried push actions in n_c
    $n_c \leftarrow \text{DIPN}(n_c, a^{\text{push}})$ // Generate node by push prediction
    ▷ **Simulation**
    $r \leftarrow 0, d \leftarrow 1, s \leftarrow n_c.\text{state}$ // s is the state of n_c
    while s is not a terminal state **do**
      $a^{\text{push}} \leftarrow$ randomly select a push action in s
      $s \leftarrow \text{DIPN}(s, a^{\text{push}})$ // Simulate to next state
      $R(s) \leftarrow \text{GN}(s)$
      $r \leftarrow \max\{r, \gamma^d \max_{a^{\text{grasp}}} R(s, a^{\text{grasp}})\}$
      $d \leftarrow d + 1$
    ▷ **Back-propagation**
    while n_c is not root **do**
      $N(n_c) \leftarrow N(n_c) + 1$
      $R(n_c.\text{state}) \leftarrow \text{GN}(n_c.\text{state})$
      $r \leftarrow \max\{r, \max_{a^{\text{grasp}}} R(n_c.\text{state}, a^{\text{grasp}})\}$
      $Q(n_c) \leftarrow Q(n_c) \cup \{r\}$ // Record the reward
      $r \leftarrow r \cdot \gamma$
      $n_c \leftarrow$ parent of n_c
  $n_{\text{best}} \leftarrow \arg\max_{n_i \in \text{children of } n_0}(\text{UCT}(n_i, n_0))$
  return push action a^{push} that leads to n_{best} from the root

using a tree policy π_{tree}. Here, *expandable* means the node has some push actions that are not tried via selection-expansion; more details of the push action space will be discussed later in the expansion part. To balance between exploration and exploitation, when the current node n_c is already fully expanded, π_{tree} uses Upper Confidence Bounds for Trees (UCT) [111] to rank its child node n_i. We customize UCT as

$$\text{UCT}(n_i, n_c) = \frac{Q^m(n_i)}{\min\{N(n_i), m\}} + C\sqrt{\frac{\ln N(n_c)}{N(n_i)}}. \tag{4.1}$$

Here, C is an exploration weight. In the first term of (Equation 4.1), unlike typical UCT that favours the child node that maximizes $Q(n_i)$, we keep only the most promising rollouts of n_i and denote by $Q^m(n_i)$ the average returns of the top m rollouts of n_i. In our implementation, $m = 3$ and $C = 2$. We also use (Equation 4.1) with parameters $m = 1$ and $C = 0$ to find the best node, and thus the best push action to execute, after the search is completed, as shown in line 30.

Expansion. Given a selected node n, we use DIPN to generate a child node by randomly choosing an untried push action a^{push} (line 14-line 15). The action a^{push} is uniformly sampled at random from the selected node's action space, which contains two types of push actions:

1. For each object, we apply principal component analysis to compute its feature axis. For example, for a rectangle object, the feature axis will be parallel to its long side. Four push actions are then sampled with directions perpendicular or parallel to the feature axis, pushing the object from the outside to its center.

2. To build a more complete action space, eight additional actions are evenly distributed on each object's contour, with push direction also towards the object's center.

Simulation. After we generated a new node via expansion, in line 16-line 22, we estimate the node's Q-value by uniformly randomly select push actions at random (line 18) and use DIPN to predict future states (line 19) until one of the following two termination criteria is met:

1. The total number of push actions used to reach a simulated state is larger than a constant D^*.

2. The maximum predicted reward value of a simulated state exceeds a threshold R^*_{gp}.

In line 21, when calculating r, a discount factor γ is used to penalize a long sequence of action. Here, we use $\max$ GN to reference the maximum value in a grasp reward table. In our implementation, GN is only called once for each unique state and the output is saved by a hashmap.

Back-propagation. After simulation, the terminal grasp reward is back-propagated (line 23-line 29) through its parent nodes to update their $N(n)$ and $Q(n)$. Denote by r_0 the max grasp reward of a newly expanded node n_0, and $n_1, n_2, \ldots, n_k$ as the sequence of n_0's parents in the ascending order up to node n_k. With $Q(n_0) = \{r_0\}$, the Q-value of n_k in this iteration is then $\max_{0 \leq j < k} \gamma^{k-j} \max Q(n_j)$, which corresponds to the max reward of states along the path [79]. Here, γ is a discount factor to penalize a long sequence of actions. As a result, for each parent n_k, $N(n_k)$ increases by 1, and $\max_{0 \leq j < k} \gamma^{k-j} \max Q(n_j)$ is added to $Q(n_k)$.

4.4 Experimental Evaluation

We performed an extensive evaluation of the proposed method, VFT, in simulation and on the real hardware system illustrated in Figure 4.1. VFT is compared with multiple state-of-the-art approaches [27], [48] and DIPN in Chapter 3, with necessary modifications for solving ORC, i.e., minimizing the number of actions in retrieving a target. The results convincingly demonstrate VFT to be robust and more efficient than the compared approaches.

Both training and inference are performed on a machine with an Nvidia GeForce RTX 2080 Ti graphics card, an Intel i7-9700K CPU, and 32GB of memory.

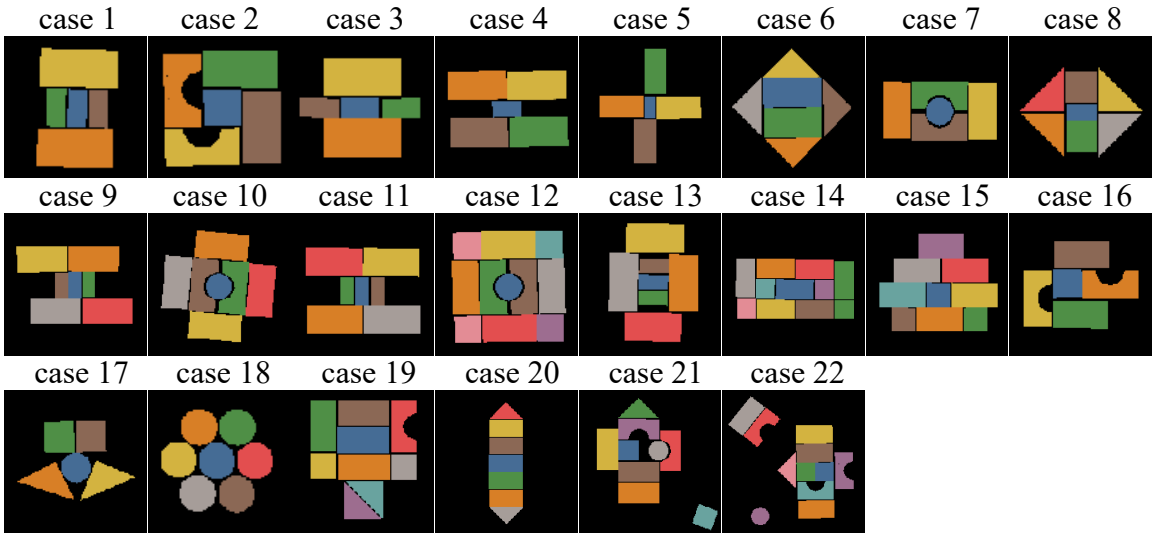


Figure 4.4: 22 Test cases used in both simulation and real world experiments. The target objects are blue. Images are zoomed in for better visualization.

4.4.1 Experiment Setup

The complete test case set includes

1. the full set of 14 test cases from [48]

2. 18 hand-designed and more challenging test cases where the objects are tightly packed.

All test cases are constructed using wood blocks with different shapes, colors, and sizes. We set the workspace's dimensions to 44.8cm $\times$ 44.8cm. The size of the images is 224×224. Push actions have a minimum 5cm *effective push distance*, defined as the end-effector's moving distance after object contact. Multiple planned push actions may be concatenated if they are in the same direction and each action's end location is the same as the next action's start location. In all scenes, the target object is roughly at the center of the scene.

The hyperparameters for VFT are set as follows. The number of iterations $N_{max} = 150$. The discount factor $\gamma = 0.8$. The maximum depth D^* of the tree is capped at 4. The terminal threshold of grasp reward $R^*_{gp} = 1.0$. Threshold R^*_g that decides to grasp or to push is 0.8 in the simulation experiments and 0.7 in the real hardware experiments. Such thresholds can potentially be fully optimized for a production system; it is not carried out in this work as

reasonably good values are easily obtained while it is prohibitively time-consuming to carry out a full-scale optimization.

4.4.2 Network Training Process

VFT contains two deep neural networks: GN and DIPN. Both are trained in simulation with the same objects as used in real experiments to capture the physical properties and dynamics of the environment. No prior knowledge is given to the networks except the dimensions of the gripper fingers.

GN is trained on-policy with 20 000 grasp actions. Similar to [12], [27], [48], randomly-shaped objects are uniformly dropped onto the workspace to construct the training scenarios. A successful grasp is decided by checking the distance between grippers, which should be greater than 0. A Huber loss on the pixel where the robot performed the grasp action is used. All other pixels do not contribute to the loss during back-propagation. Image-based pre-training [12], [103] was employed to initialize the training parameters. We then train the GN by stochastic gradient descent with the momentum of 0.9, weight decay of 10^{-4}, and batch size of 12. The learning rate is set to 5×10^{-5} and by half every 2000 iteration.

DIPN [12] is trained in a supervised manner with 200 000 random push actions from simulation. In the push data set, 20% of the scenes contain randomly placed objects, and 80% contain densely packed objects. The push distance for DIPN is fixed to 7.4 cm (effective touch distance is 5 cm). In the DIPN (Chapter 3), the distance was 5 cm and 10 cm without considering the effective range.

We note that a total of 2000 actions (500 grasps and 1500 pushes) are sufficient for the networks to achieve fairly accurate results (see, Chapter 3). Because training samples are readily available from simulation, it is not necessary to skimp on training data. We thus opted to train with more data to evaluate the full potential of VFT.

A Smooth L1 Loss with beta equals to 2 is used instead of 1 in Chapter 3. We train the DIPN by stochastic gradient descent with the momentum of 0.9, weight decay of 10^{-4}, and

using cosine annealing schedule [112] with learning rates of learning rate of 10^{-3} for 76 epochs, and the batch size is 128.

4.4.3 Compared Methods and Evaluation Metrics

Goal-Conditioned VPG (gc-VPG). Goal-conditioned VPG (gc-VPG) is a modified version of Visual Pushing Grasping (VPG) [27], which uses two DQNs [108] for pushing and grasping predictions. VPG by itself does not focus on specific objects; it was conditioned [48] to focus on the target object to serve as a comparison point, yielding gc-VPG.

Goal-Oriented Push-Grasping. In [48], many modifications are applied to VPG to render the resulting network more suitable for solving ORC, including adopting a three-stage training strategy and an efficient labeling method [113]. For convenience, we refer to this method as go-PGN (the authors of [48] did not provide a short name for the method).

DIPN. As an ablation baseline for evaluating the utility of employing deep tree search, we replace MCTS from VFT with a search tree of depth one. In this baseline, DIPN is used to evaluate all candidate push actions. The push action whose predicted next state has the highest grasp reward for the target object is then chosen. This is similar to how DIPN is used in Chapter 3; we thus refer to it simply as DIPN.

In our evaluation, the main metric is the total number of push and grasp actions used to retrieve the target object. For a complete comparison to [27], [48], we also list VFT's grasp success rate, which is the ratio of successful grasps in the total number of grasps during testing. The completion rate, i.e., the chance of eventually grasping the target object, is also reported. Similar to Chapter 3, when DIPN is used, a 100% completion rate often reached.

We only collected evaluation data on DIPN and VFT. For the other two baselines, gc-VPG and go-PGN, results are directly quoted from [48] (at the time of our submission, we could not obtain the trained model or the information necessary for the reproduction of gc-VPG and go-PGN). While our hardware setup is identical to that of [48], and the poses of objects are also identical, we note that there are some small differences between the

evaluation setups:

1. We use PyBullet [114] for simulation, while [48] uses CoppeliaSim; the physics engine is the same (Bullet).

2. [48] uses an RD2 gripper in simulation and a Robotiq 2F-85 gripper for real experiment; all of our experiments use 2F-85.

3. [48] has a 13cm push distance, while we only use a 5cm effective distance (the distance where fingers touch the objects)

4. [48] uses extra top-sliding pushes which expand the push action set.

We believe these relatively minor differences in experimental setup do not provide our algorithm an unfair advantage.

4.4.4 Simulation Studies

Figure 4.5 and Table 4.1 show the evaluation results of all algorithms on the 10 simulation test cases from [48]. Each experiment is repeated 30 times, and the average number of actions until task completion in each experiment is reported. Our proposed method, VFT, which uses an average of 2.00 actions, significantly outperforms the compared methods. Specifically, VFT uses one push action and one grasp action to solve the majority of cases, except for one instance with a half-cylinder shaped object, which is not included during the training of the networks. Interestingly, when only one push is necessary, VFT, with its main advantage as multi-step prediction, still outperforms DIPN due to its extra simulation steps. The algorithms with push prediction perform better than gc-VPG and go-PGN in all metrics.

To probe the limit of VFT's capability, we evaluated the methods on harder cases demanding multiple pushes. The test set includes 18 manually designed instances and 4 cases from [48] (see Figure 4.4). As shown in Figure 4.6 and Table 4.2, VFT uses fewer actions than DIPN as VFT looks further into the future. Though we could not evaluate the

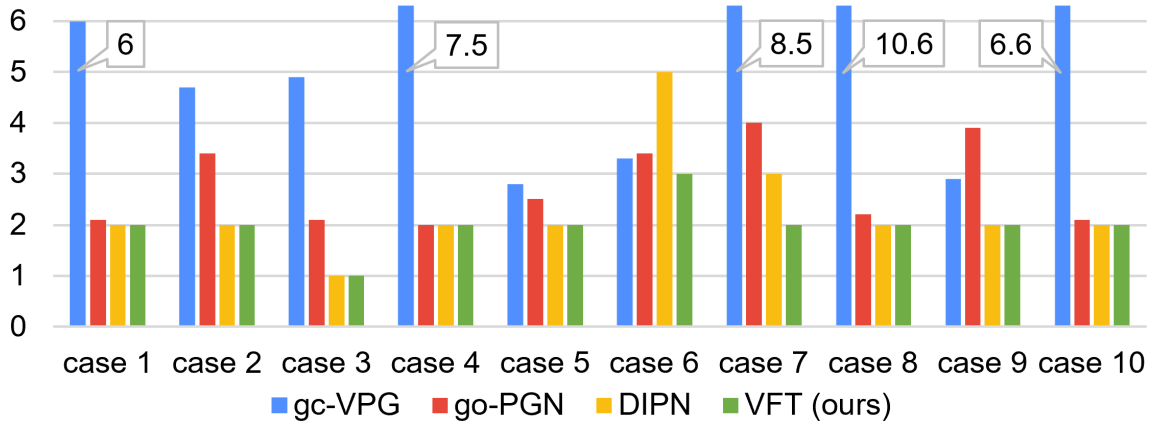


Figure 4.5: Simulation results per test case for the 10 problems from [48]. The horizontal axis shows the average number of actions used to solve a problem instance: the lower, the better.

	Completion	Grasp Success	Number of Actions
gc-VPG [48]	89.3%	41.7%	5.78
go-PGN [48]	99.0%	90.2%	2.77
DIPN [12] (Chapter 3)	100%	100%	2.30
VFT (Chapter 4)	100%	100%	**2.00**

Table 4.1: Simulation results for the 10 test cases from [48].

performance of gc-VPG and go-PGN on these settings for direct comparison because we could not obtain the information necessary for the reproduction of these systems, notably, the average number of actions (2.45) used by VFT on harder instances is even smaller than the number of actions (2.77) go-PGN used on the 10 simpler cases.

	Completion	Grasp Success	Num. of Actions
DIPN [12] (Chapter 3)	100%	98.3%	4.31
VFT (Chapter 4)	100%	98.8%	**2.45**

Table 4.2: Simulation result for the 22 test cases in Figure 4.4.

4.4.5 Evaluation on a Real System

We repeated the 22 hard test cases on a real robot system (Figure 4.1a). Both VFT and DIPN are evaluated. We also bring the experiment result from [48] on its 4 real test cases for comparison. All cases are repeated at least 5 times to get the mean metrics. The result,

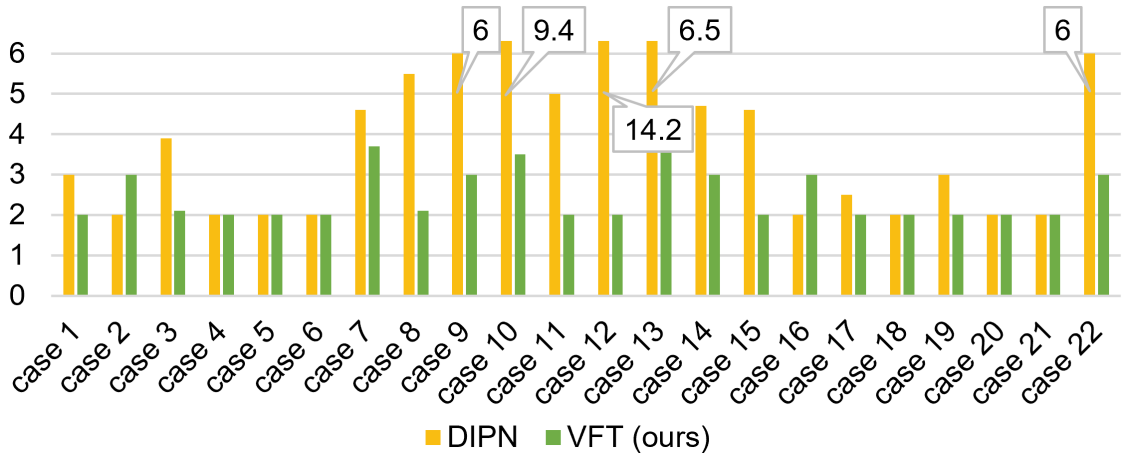


Figure 4.6: Simulation result per test case for the 22 harder problems (Figure 4.4). The horizontal axis shows the average number of actions used to solve a problem instance: the lower, the better.

shown in Figure 4.7, Table 4.3, and Table 4.4 closely matches the results from simulation. We observe a slightly lower grasp success rate due to the more noisy depth image on the real system. The real workspace's surface friction is also different from simulation. However, VFT and DIPN can still generate accurate foresight.

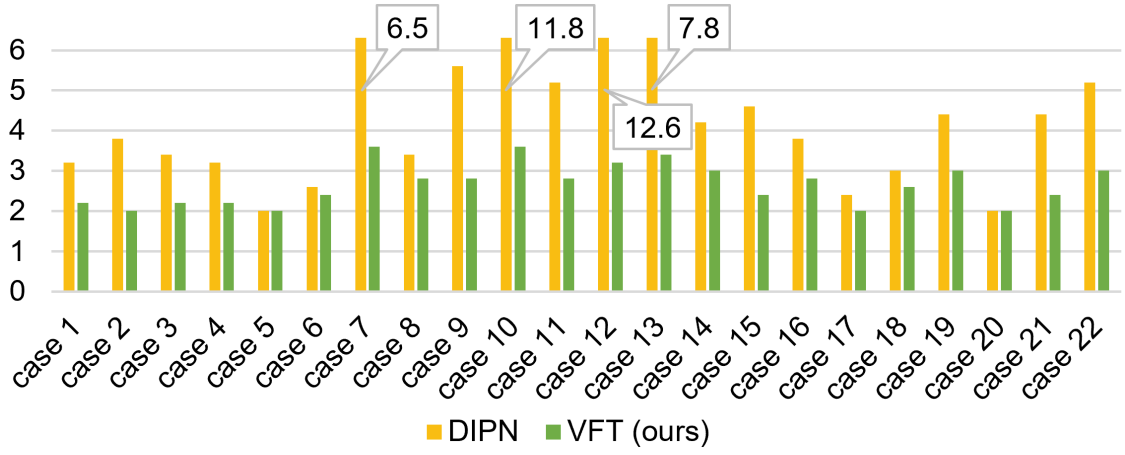


Figure 4.7: Real experiment results per test case for the 22 harder problems (Figure 4.4). The horizontal axis shows the average number of actions used to solve a problem instance: the lower, the better.

We also explored our system on everyday objects (Figure 4.8), where we want to retrieve a small robotic vehicle surrounded by soapboxes. Although the soapboxes and the small vehicles are unseen types of objects during training, the robot is able to strategically push

	Completion	Grasp Success	Num. of Actions
DIPN [12] (Chapter 3)	100%	97.0%	4.78
VFT (Chapter 4)	100%	98.5%	**2.65**

Table 4.3: Real experiment results for the 22 Test cases in Figure 4.4.

	Completion	Grasp Success	Num. of Actions
go-PGN [48]	95.0%	86.6%	4.62
DIPN [12] (Chapter 3)	100%	100%	4.00
VFT (Chapter 4)	100%	100%	**2.60**

Table 4.4: Real experiment results for cases 19 to 22 in Figure 4.4.

the soapboxes away in two moves only and retrieve the vehicle.

Figure 4.8: Test scenario with soap boxes and masked (in purple) 3D printed vehicle. Two push actions and one grasp action.

We report that the running time to decide one push action is around 3 minutes on average when the number of MCTS iterations is set to be 150. A single push prediction of DIPN took 30 milliseconds. While using the simulator as the transition function in MCTS under a similar criterion would take 8 minutes on average to decide one push action. In this chapter, our primary focus is action optimization.

4.5 Summary

In conclusion, through an organic fusion of Deep Interaction Prediction Network (DIPN) and MCTS, the proposed Visual Foresight Trees (VFT) can make a high-quality multi-horizon prediction for optimized object retrieval from dense clutter. The effectiveness of VFT is convincingly demonstrated with extensive evaluation. As to the limitations of VFT, the time required is relatively long because of the large MCTS tree that needs to be computed. This

can be improved with multi-threading because the rollouts have sufficient independence. Currently, only a single thread is used to complete the MCTS. It would also be interesting to develop a learned heuristic or value function to guide or truncate the rollout phase, potentially reducing inference time. This technique would be similar in spirit to the MuZero algorithm [115], which has been shown to be efficient by combining Monte Carlo tree search and learning by self-playing in an end-to-end manner. The learned rollout policy could lead to better performance, which will be covered in Chapter 5. One issue related to end-to-end training is data efficiency, which is why this type of technique has been limited to games. Improving the data efficiency of end-to-end techniques is crucial to the deployment of these techniques on robotic tasks.

CHAPTER 5
INTERLEAVING MONTE CARLO TREE SEARCH AND SELF-SUPERVISED LEARNING FOR OBJECT RETRIEVAL IN CLUTTER

5.1 Introduction

Kahneman [116] proposed a thought-provoking hypothesis of human intelligence: in solving real-world problems, humans engage fast or "System 1" (S1) type of thinking for making split-second decisions, e.g., speech, driving, and so on. For other decision-making processes, e.g., playing chess, a slow or "System 2" (S2) approach is taken, where the brain would perform a search over some structured domain for the best actions to take. After repeatedly using S2 thinking to solve a given problem, patterns can be distilled over time and burned into S1 to accelerate the overall process. In playing chess, for example, good chess players can instinctively identify good candidate moves. First-time or beginner drivers rely heavily on S2 and gradually converge to S1 as they gain more experience. This S2→S1 thinking has gained significant attention and has been explored in many directions in machine learning, including attempts at building machines with consciousness [117]. Perhaps the most prominent line of work in reinforcement learning [118] that closely aligns with this paradigm is the application of Monte Carlo Tree Search (MCTS) for carrying out self-supervised learning in games [115], [119], where an "understanding" of a game emerges from a lifelong self-play and is gradually distilled so that it significantly reduces the search effort. Gradually, the overall system learns enough useful information that allows it to play perfect games with much less time and computing resources.

Inspired by [115], [119] that show a search-and-learn approach for realizing S2→S1 applies well to game-like settings with relatively well-defined rules, we set out to find out whether we could build a similar framework that enables real robots to interact with real-world

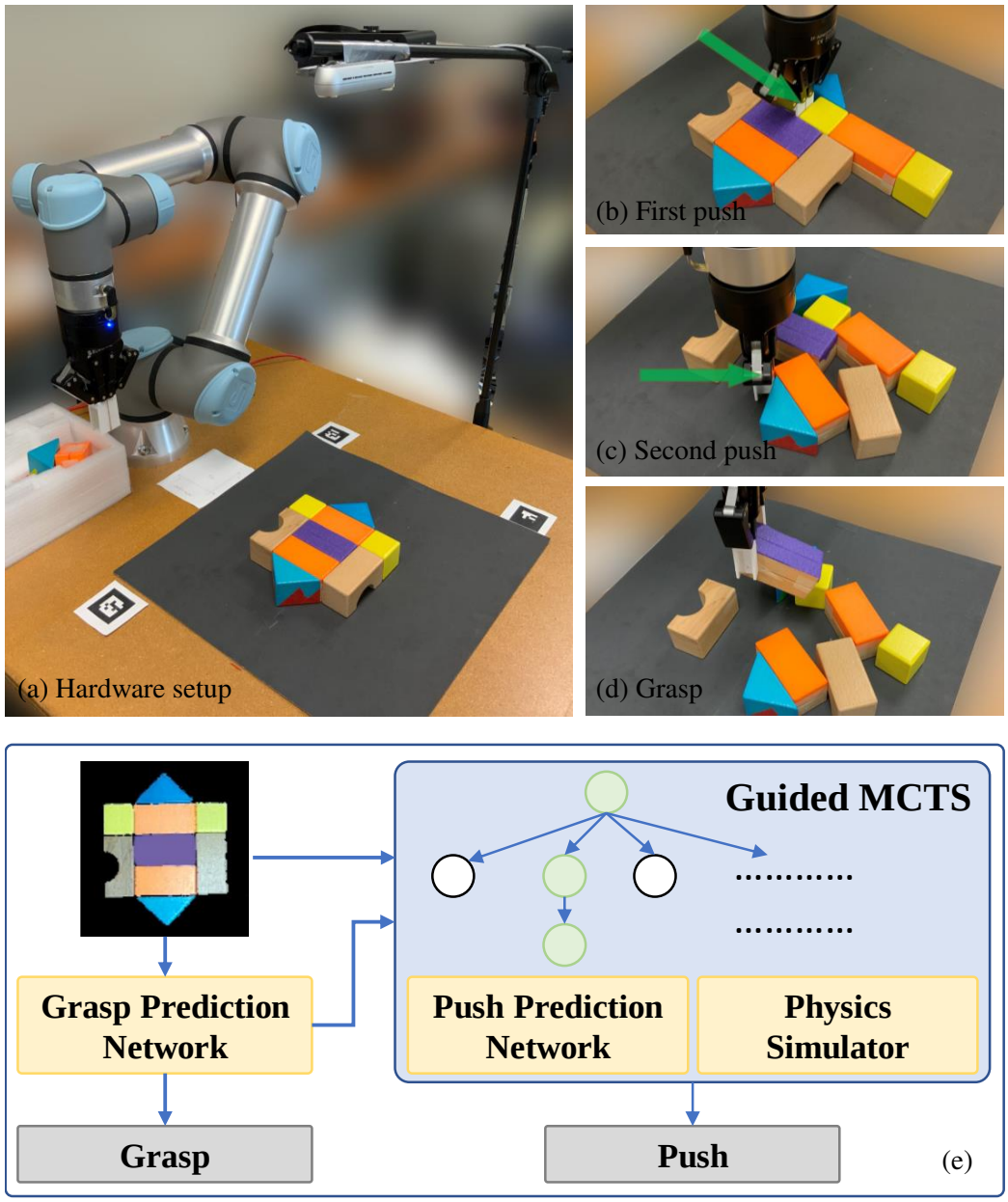


Figure 5.1: (a) The hardware setup for object-retrieval-from-clutter includes a Universal Robots UR5e manipulator with a Robotiq 2F-85 two-finger gripper, and an Intel RealSense D455 RGB-D camera. The objects are placed in a square workspace and the target object is masked in purple. (b)(c) Two push actions (shown with green arrows) are used to enable the grasping of the target (purple) object. (d) The target object is successfully grasped and retrieved. (e) The overview of our overall system.

physics and uncertainties to perform physical tasks, somewhat akin to [120]. Specifically, we focus on the task of retrieving an object enclosed in clutter using non-prehensile actions, such as pushing and poking, followed by prehensile two-finger grasping. The goal is to

obtain a computationally efficient system and produce high-quality solutions (i.e., using the minimum number of actions).

As pointed out by Valpola [121], due to the difficulty in exploring the landscape of the state space of real-world problems, in addition to uncertainty, naive applications of the S2→S1 paradigm often lead to undesirable behavior. Non-trivial design as well as engineering efforts are needed to build such S2→S1 systems. In the object-retrieval-from-clutter setting, the challenge lies in the difficulty of predicting the outcome of push actions, with the tip of the gripper, when many objects are involved. This is due to discontinuities inherent in object interactions; for example, while a certain pushing action might move a given object, a slightly different push direction could miss that same object entirely.

The main contribution of this work is proving the feasibility of applying the S2→S1 philosophy to build a self-supervised robotic object retrieval system capable of continuously improving its computational efficiency, through *cloning* the behavior of the time-consuming initial MCTS phase. Through the careful design and integration of two Deep Neural Networks (DNNs) with MCTS, our proposed self-supervised method, named **M**onte Carlo tree search and learning for **O**bject **RE**trieval (MORE), enables a DNN to learn from the manipulation strategies discovered by MCTS. Then, learned DNNs are fed back to the MCTS process to guide the search. MORE significantly reduces MCTS computation load and achieves identical or better outcomes, i.e., retrieving the object using very few strategic push actions. In other words, our method "closes the loop". This contrasts with [120], which only learns to replace the rollout function of MCTS.

5.2 Problem Formulation

The Object Retrieval from Clutter task consists in using a robot manipulator to retrieve a hard-to-reach target object (Figure 5.1). Objects are rigid bodies with various shapes, sizes, and colors; the target object is assigned a unique color. Similar to Chapter 3, a top-down fixed camera is installed to observe the workspace. The camera takes an RGB-D image of

the workspace (e.g., the top-left image of Figure 5.1), which serves as the only input to our system.

Pushing and *grasping* actions are allowed, the execution of each is considered as one *atomic action*. A grasp action is defined as a top-down overhead grasp motion $a^g = (x, y, \theta)$, corresponding to the gripper's target location and orientation, based on a coordinate system defined over the input image. A push action is defined as a quasi-static planar motion $a^p = (x_0, y_0, x_1, y_1)$ where (x_0, y_0) and (x_1, y_1) are the start and the end locations of the gripper tip. The horizontal push distance is fixed and it is 10cm in our experiments. Each primitive action is transformed to the real-world coordinates for execution, but all the planning and reasoning are in image coordinates. The robotic arm keeps pushing objects until the target object can be grasped or until the target object is pushed outside of the workspace, in which case the task is considered a failure. The problem is to find a policy that maximizes the frequency of successfully grasping the target object, while also minimizing the number of pre-grasp pushing actions.

5.3 Methodology

The MORE framework consists of three components: a Grasp Network (GN), a Monte Carlo Tree Search (MCTS) routine, and a Push Prediction Network (PPN). GN is a neural network that predicts the success probabilities of grasp actions. It is trained online similarly to Chapter 4. The success probabilities can be interpreted as immediate rewards. MCTS uses a physics engine as a transition function to simulate long sequences of consecutive push actions that end with a terminal grasp action. Each branch in MCTS is composed of push actions as internal nodes, and a grasp action as a leaf. Grasp actions are evaluated with GN, and the returned rewards are back-propagated to evaluate their corresponding branches. The branch with the highest discounted reward, or *Q-value*, is selected for execution by the robot.

While highly effective in finding near-optimal paths, MCTS suffers from a high computation time that makes it impractical. To solve this, MORE employs a second neural network,

PPN, to prioritize the action selection in the rollout policy. The robot starts by relying entirely on MCTS (S2 type of thinking) to solve various instances of the object-retrieval problem. Instead of throwing away the computation performed by MCTS for solving the various instances, we use the computed Q-values as ground-truth to train PPN. Note that this computation data is free, since it is generated by the simulations performed by MCTS as a byproduct of solving the actual problem. PPN is a neural network that learns to imitate MCTS and clone its behavior, while avoiding heavy computation and physics simulations by MCTS. As PPN becomes more accurate in predicting the outcome of MCTS, the robot starts relying on both MCTS and PPN for action selection. In a nutshell, PPN is used for orienting the search in MCTS toward more promising push actions that rearranges the scene and renders the target object graspable. After a long experience, PPN's accuracy in predicting the Q-values of push actions matches that of MCTS, and the robot switches entirely to PPN to make decisions in a few milliseconds (S1 type of thinking).

5.3.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [122] is used in MORE for both decision-making and training PPN. A typical MCTS routine has four steps: selection, expansion, simulation, and back-propagation. In our case, the goal of the search is to find the shortest action sequence; we can stop the search as soon as the best solution is found without exploring the rest. The search stops in two cases:

1. the number of iterations n exceeds a pre-set budget N_{max}, or
2. the expanded node with state that the target object can be grasped, and all nodes in parent level are expanded.

A node is considered as a leaf if $\max_{i,j,\theta} R_{gm}(s_t)[i, j, \theta] > R_{g*}$ where $R_{gm}(s_t)$ is obtained from GN and R_{g*} is a pre-defined high probability. The maximum depth of the tree is limited to d, where d is set to 4 in our experiments.

In the selection phase, we find an expandable node starting from the root according to the search policy

$$\pi_n(s) = \arg\max_{a^p}(Q(s, a^p) + C\sqrt{\frac{\ln N(s)}{N(s, a^p)}}), \tag{5.1}$$

where $N(s)$ is the number of visits to node (state) s and $N(s, a^p)$ is the number of times push action a^p has been selected at node (state) s. The Q-value is calculated as

$$Q(s, a^p) = \frac{\sum_{i=1}^{m} r_i(s, a^p)}{\min\{N(n_i), m\}}, \tag{5.2}$$

where $r(s, a^p)$ is the returned long-term reward and m is a pre-set maximum. Only the best m terms $r_i(s, a^p)$ are used to compute the Q-value in the equation above. m is set to 10 when expanding nodes and 1 when selecting the best solution. C is the coefficient of the exploration term, and it is set to 2 when expanding nodes and 0 when selecting the best solution. In the expansion phase, we use a physics simulator to execute the chosen push action a^p at state s_i and predict new state s_{i+1}. Then, a random policy is used to sample actions to simulate until a grasp is possible or a failure is encountered. The reward r is predicted by GN at a terminal state s_t. Reward r is set to 1 if $\max_{i,j,\theta} R_{gm}(s_t)[i, j, \theta] > R_{g*}$, and 0 otherwise. One additional term $\delta \max(R_{gm}(s_t))$ is added to r, to distinguish between good and bad push actions. We set δ to be 0.2. In the last step, reward r is propagated back to its parent nodes to update their Q-values with a discount factor $\gamma = 0.5$.

As the push action space is enormous even after discretization, we further sample a subset of actions such that all push actions start around the contour of an object and point to the center of the object (Figure 5.2). This action sampling method has been discussed in VFT (Chapter 4) and was empirically proven efficient for a similar setup of object retrieval.

In our implementation, N_{max} is set to 300 when MCTS is used to collect data to train PPN. The second and the third conditions for stopping the search are only activated after at least 50 roll-outs, so that the number of visits to a state is statistically significant and to reduce the variance of PPN.

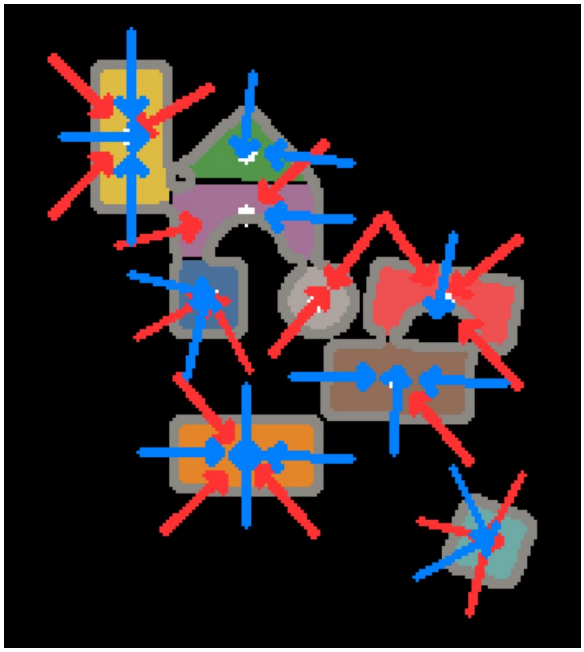


Figure 5.2: Sampled push actions.

5.3.2 Push Prediction Network (PPN)

As previously mentioned, PPN learns to imitate MCTS. PPN is a deep neural network with ResNet-34 FPN [102], [110] as the backbone, where the P2 level of the FPN connects to the head. It takes a two-channel input and outputs a single channel pixel-wise push Q-value map, similar to the reward map produced by GN. An example input is shown in Figure 5.3, where the first channel is a segmented image of all objects and the second channel is a binary image of the target object. The output is the image on the right of Figure 5.3, where the arrow shows a push action with the highest Q-value. PPN estimates the Q-value (discounted rewards) $Q_p(s_t)$ of executing push actions at the corresponding pixel, where the action is assumed to push 10cm to the right. $\max(Q_p(s))$ is limited to the range $[0, \eta]$, where η is the maximum reward of a terminal state.

When MCTS is used to generate training cases, it builds a tree and saves the transitions for each case: the state (image) s, the push action a^p, the Q-value $Q(s, a^p)$, and the visited number $N(s, a^p)$. As such, PPN is trained in a self-supervised manner. The input image is rotated based on a push action so that the corresponding push action points to the right.

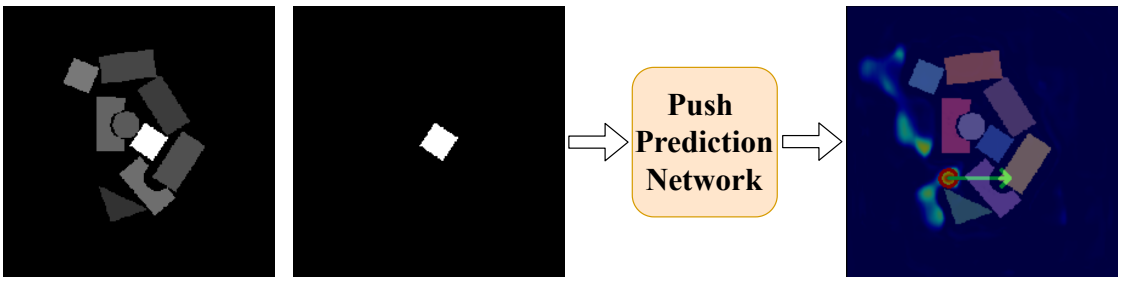


Figure 5.3: The left two figures are the input to PPN. The first is a segmentation of objects; the second is the mask of the target object. The image on the right is the output from the PPN. We use Jet colormap to represent the reward value, where the value ranges from red (high) to blue (low). The pixel with the highest Q-value is plotted with a circle and attached with an arrow on the right image, representing pushing action starting at the circle and moving to the right with a distance of 10cm.

Because a single action is generated by MCTS (i.e., a δ signal over the entire input), which is not conducive to training PPN, we "expand" the Q-value over a 3×3 patch centered around MCTS action but set invalid pushes (e.g., if part of the patch is inside an object) to be zero. Now, the label is relatively dense compared to a one-hot pixel, so we can use Smooth L1 loss from Pytorch [123] with β equals to 0.8 to regress. Only gradients on the labeled pixels are used. Loss weighting is also applied: label values from the MCTS are weighted based on $N(s, a^p)$, label values (zero Q-value) from void push actions are weighted with a small number, 0.001 for collision and 0.0001 for pushing thin air. We observe that PPN has difficulty learning to create clearance around the target object. Data augmentation is applied here so that for each training case, we also randomly choose the target object for the MCTS to solve; so each arrangement becomes many training cases. It mitigates over-fitting; given similar visual information, it could learn different strategies, as the target object could be anywhere.

The head model is an FCN with four layers, where the first two layers have a kernel size of 3, the last two 1, and the strides of four layers are all 1. Batch normalization is used at each layer of the head model except the last. Bilinear interpolation ($\times 2$) is applied interleaved between the last three layers of the head model to scale up the hidden state to the same size as the input image. The training process has two stages, one to train the network

with a batch size of 8, learning rate starts at $1\mathrm{e}{-4}$, for 50 epochs. The learning rate decays with cosine annealing [112], where the maximum number of iterations is set to be the same as the epoch number 50 and the minimum learning rate is $1\mathrm{e}{-8}$. The second is a fine-tuning stage; we increase the batch size to 28 and the learning rate to $1\mathrm{e}{-5}$ with an epoch of 20.

5.3.3 Guided Monte-Carlo Tree Search

With the trained GN and PPN, a guided MCTS is implemented to accelerate the search process, cutting cost from time-consuming expansion and simulation phases. GN is again used to determine the terminal state and if so, calculate its estimated reward, as discussed in subsection 5.3.1. PPN, trained with data from MCTS, can estimate how much reward can be gained from taking a push action at a certain state.

For this combination of MCTS with PPN, some additional updates are made (compared to subsection 5.3.1) to incorporate the guidance from PPN. The exploration term is removed from the search policy, so C in equation Equation 5.1 is set to 0. Similar to [124], we use the estimated reward from PPN as a prior, so the Q-value is calculated as follows

$$Q_{guide}(s, a^p) = \frac{\max(Q_p(s)) + \sum_{i=1}^{m} r_i(s, a^p)}{N(s, a^p)}, \tag{5.3}$$

where m is set to 3 when expanding nodes and $N(s, a^p)$ is initialized to 1 for all state-action pairs. Instead of computing an average as standard MCTS, only best m of Q_p are considered, this is due to the number of rollout is small, a good action could be averaged out. To select the best action as the next step solution, the Q-value is calculated without the denominator

$$Q_{best}(s, a^p) = \max(Q_p(s)) + \max(r_i(s, a^p)), \tag{5.4}$$

where only the best explored solution is considered.

The push action space of the guided MCTS is limited to a subset (like Figure 5.2) so that the estimated reward from PPN is more accurate and the branching factor of the tree is of a

reasonable size. To make the selection mimic the training data, we rotate the image for each sampled push action such that the push action in the rotated image is always pointing to the right. Then, we only use the estimated Q-value at the corresponding pixel (push action) of the output Q-value map. An example of guided MCTS is given in Figure 5.4. The expansion of the tree is prioritized by PPN, where the push action with higher Q-value is sampled earlier, and the rollout policy is also prioritized. The maximum depth of the tree is limited to 3 instead of 4 as used in the earlier version of MCTS for collecting data to train PPN.

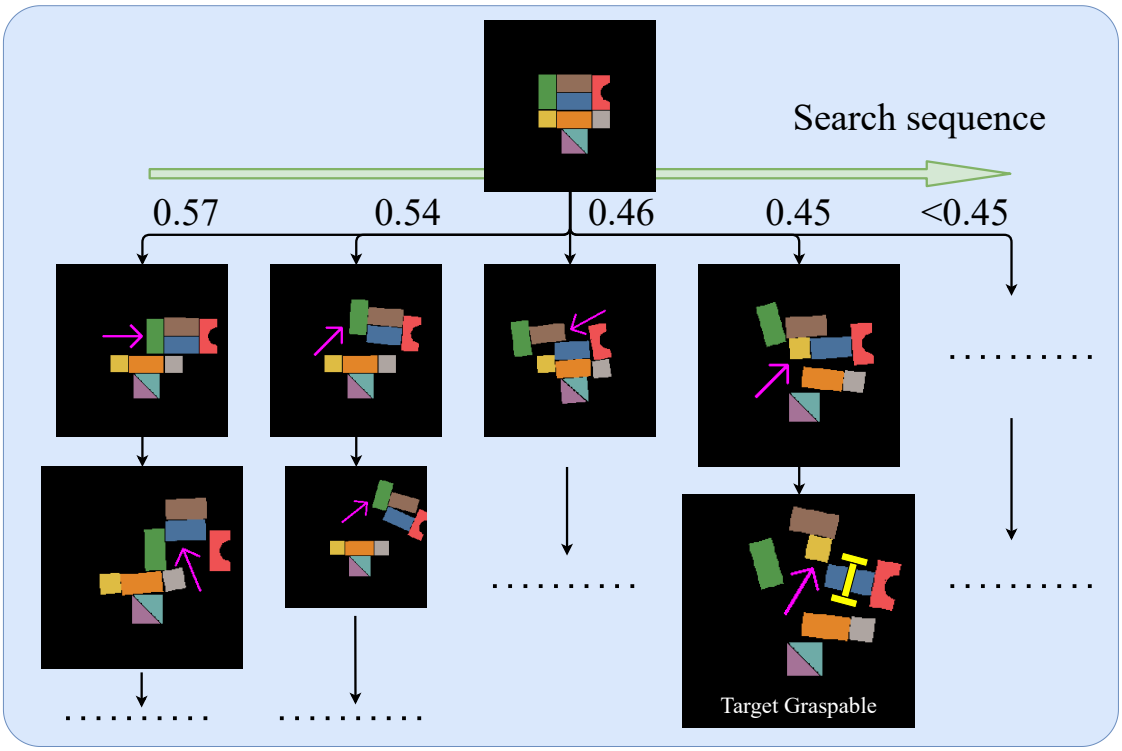


Figure 5.4: An example of the guided MCTS with a budget of 10 iterations. State with larger image have higher estimated Q-values. All expanded nodes are plotted. The numbers in the first levels represent the estimated Q-value returned by PPN for corresponding push action. These values, together with the reward returned from simulation, guide the tree search.

5.4 Experimental Evaluation

We evaluated the proposed technique both in simulation (PyBullet [125]) and on adversarial test cases on a UR5e robot with a Robotiq 2F-85 gripper using real objects. The robot,

workspace, objects, and camera are the same in simulation and real-world experiments, so that we can seamlessly transfer from simulation to the real setup. The workspace is limited to a square with a side length of 0.448m; it is discretized as a grid of 224×224 cells during the image processing step. The friction of objects and table surface cannot be accurately measured; nevertheless, high-fidelity physical properties do not seem to be needed for this particular application. The results demonstrate that the proposed method significantly outperforms MCTS in Chapter 4 [16] in terms of time efficiency while returning plans of equal quality. The plans returned by the proposed technique contain fewer actions and yield higher success rates than those returned by the purely learning-based solution presented in [48]. Training and evaluation are completed on a machine with an Intel i7-9700K CPU and an Nvidia GeForce RTX 2080 Ti.

5.4.1 Simulation experiments

Tasks. Given an arrangement of heterogeneous and tightly packed objects, a target object is to be retrieved using push and grasp actions from a two-finger gripper. In simulation, we benchmark on 22 adversarial test cases from Chapter 4 (Figure 4.4) and 10 from [27], [48]. Here "adversarial" means that at least one push action has to be executed for a grasp action to be feasible (insert gripper without collision). Random cases, which are too easy from Chapter 3 and Chapter 4 [12], [16], are not discussed here.

Metrics. We use four metrics:

1. the number of actions used to retrieve the target object,
2. the total time used for retrieving the target object, which includes both planning time and execution time for simulation results,
3. the completion rate, failures occur when the target object is pushed out of the workspace, and

4. the grasp success rate, which is the number of successful grasps divided by the total number of grasping attempts.

The number of re-arrangement actions that are needed to make the target object graspable and time are the two main metrics. The completion and grasp success rates are also reported but are not the main focus as they are often close to 100%.

Baseline Methods. We compare with three methods:

1. A self-supervised reinforcement learning method denoted as go-PGN [48], which trains a grasp DQN and a push DQN then selects an action with the highest Q-value out of the two networks to execute.

2. MCTS as described in Section subsection 5.3.1. This is adapted from [16], but we use here a simulator to predict the next state instead of the originally used learned model, for fair comparisons.

3. PPN as described in Section subsection 5.3.2. PPN proposes push actions based on their predicted Q-values and the robot executes those actions until the target object can be grasped according to GN.

Simulation Studies. We ran our method and the three alternative methods on 22 cases Chapter 4 and 10 cases [27], [48], in simulation first. Table 5.1 and Table 5.2 show the overall performance of the four methods, where MCTS based methods are limited to a budget of 50 iterations per test case. In this paper, we denote the tree search methods with different budgets of search iterations as MCTS-10/20/50 and MORE-10/20/50, where the suffix denotes the iterations limit. The 22 cases are generally harder to solve than the 10 cases, where the target object can be retrieved after one push action. The time metric records the average time (out of 5 trials) for retrieving the object, including planning and execution times.

For the baseline go-PGN, results on 10 cases are directly quoted from the paper (at the time of our submission, we could not obtain the trained model or the information necessary

for fully reproducing go-PGN). MORE uses the fewest number of actions to solve the task. Performance details on 22 cases can be found in Figure 5.5 for the number of actions and Figure 5.6 for the running time. PPN is fast as it is a one-stage DNNs solution. It learned a policy that creates free spaces around the target object, but it is less consistent and less stable than the tree search solutions. From our observation, PPN can propose non-prevailing pushing actions. MCTS provides a consistent and good quality solution, but requires a much longer planning time. MORE, combining the benefits of both, reduces the planning time and delivers high-quality solutions.

	Num. of Actions	Time	Completion	Grasp Success
MORE-50	**2.61**	82s	100%	99.2%
MCTS-50 [16]	2.69	208s	100%	99.1%
PPN	3.68	8s	100%	97.7%

Table 5.1: Simulate experiment results for 22 cases Chapter 4. Budgets of MCTS and MORE are limited up to 50 iterations.

	Num. of Actions	Time	Completion	Grasp Success
MORE-50	**2.10**	16s	100%	100%
MCTS-50 [16]	2.20	32s	100%	93.4%
PPN	2.70	4s	100%	95.0%
go-PGN [48]	2.77	–	99.0%	90.0%

Table 5.2: Simulate experiment results for 10 cases [48]. Budgets of MCTS and MORE are limited up to 50 iterations.

Ablation Studies Although the data generated by MCTS for training PPN is free because it is collected fully automatically in simulation, we set to explore data efficiency in training, which can be important for building larger models in practice. For this purpose, we collected 243 training cases (65384 transitions in 30 hours with PyBullet) with MCTS as described in subsection 5.3.1. Training on PPN on all data took approximately 22 hours. As shown in Figure 5.7, we tested MCTS and MORE with different budgets. Also, MORE is trained on different numbers of training data. Clearly, the problem can be solved by all tested methods with fewer actions when the search iteration limits are increased. But the time for solving

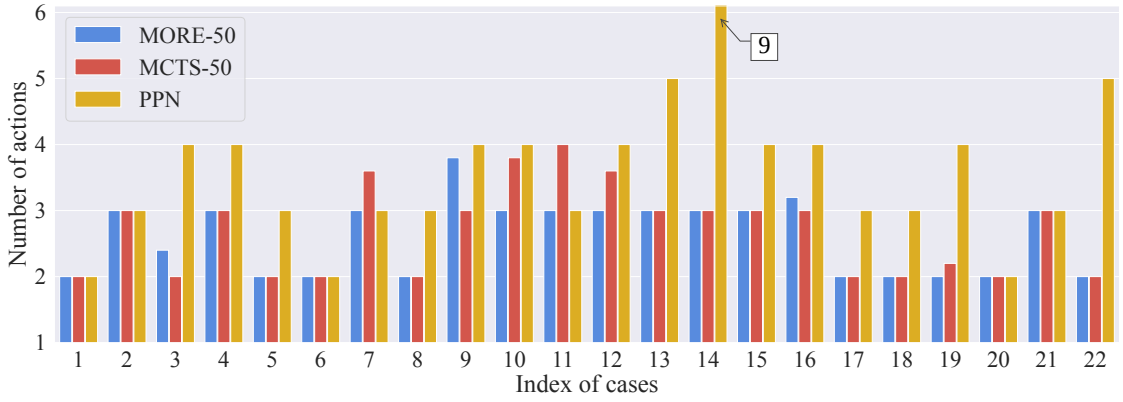


Figure 5.5: The average number (out of 5 trials) of action used to solve one case for 22 cases.

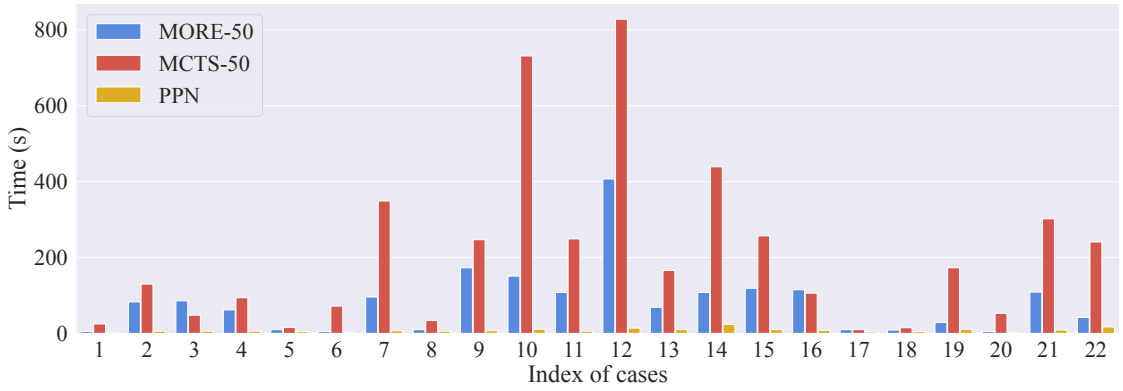


Figure 5.6: The average time (of 5 trials) used to solve one case for 22 cases.

the problem also increases as a consequence. The proposed MORE technique can retrieve target objects with only 2.8 executed actions and using only 10 iterations of MCTS that last 36 seconds on average. This is close to the best that MCTS without PPN can achieve, 2.69 actions, after 50 iterations that last 208 seconds. When we limit the number of iterations of MCTS (without MORE) to 10, the number of executed actions increases to 3.19, and the search time remains relatively high (127 seconds). This clearly demonstrates the superior performance of the proposed approach in terms of both time and action efficiency.

5.4.2 Robot Experiments

We evaluated the four methods on six real test cases (four from [48] and two from Chapter 4). These six test cases are representative in that they contain more objects and often require at

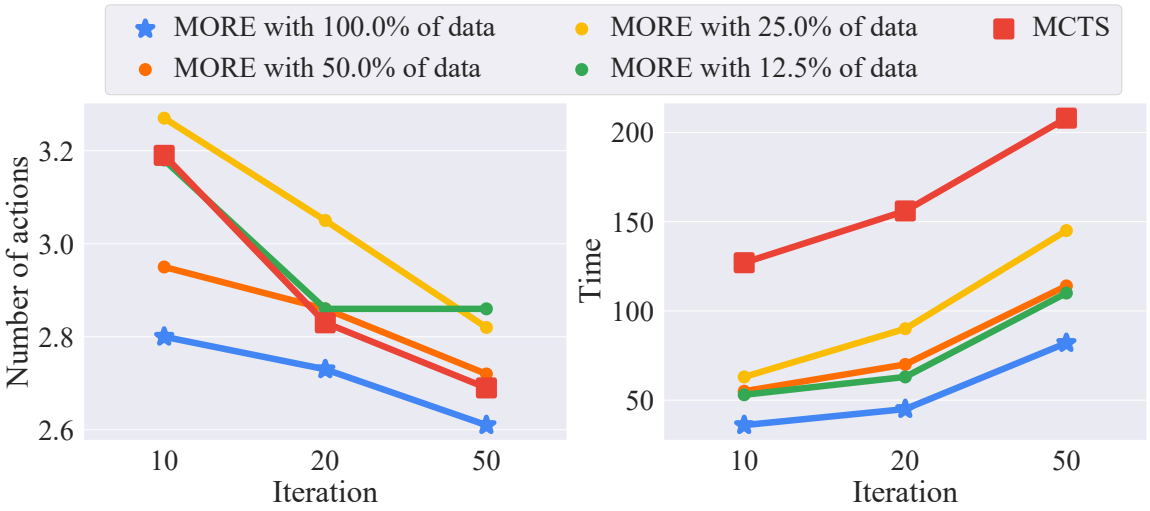


Figure 5.7: Different amounts of training data are used to train PPN, which are evaluated on MORE with different budgets (iteration). This is the evaluation of the 22 cases.

least two push actions to solve. For these real experiments, the results are shown in Table 5.3 and Figure 5.9. The budget of MCTS and MORE is limited to 10 iterations. We note that the results for go-PGN are taken from [48]. The execution time of PPN is not listed in Table 5.3 as it is a near-constant small value as we had in the simulation experiments. From the result, we observe only negligible performance degradation in comparison to simulation, which may be due to differences in friction, slight differences in the dimensions of the objects between simulation and real world, statistical error, or a combination of these. Overall, the sim-to-real transfer was very successful and showed that MORE can learn in simulation and directly apply the learned skill to real-world tasks. We assume models of objects are known, such that simple pose estimation can be used to locate objects in the real world and placed in simulation for planning. We could also use sophisticated tracking systems [126]–[128] for general purpose.

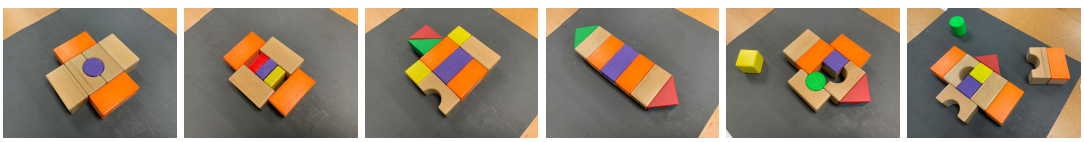


Figure 5.8: Manually generated cases similar to [48] and Chapter 4. The target object is masked in purple. These cases are used also in simulation experiments as shown in Figure 4.4.

	Num. of Actions	Time	Completion	Grasp Success
MORE-10	**2.83**	$36s$	100%	100%
MCTS-10 [16]	3.67	$190s$	100%	95.8%
PPN	3.72	$3s$	94.5%	95.8%
go-PGN [48]	4.62	–	95.0%	86.6%

Table 5.3: Real experiment results for six cases as shown in Figure 5.8. The budget of MCTS and MORE is limited to 10 iterations. For go-PGN, only the first four cases apply, and results are from [48]. Only planning time is recorded (robot execution was intentionally slowed down for safety). The computation time for PPN to solve a task is 3 seconds on average (estimated).

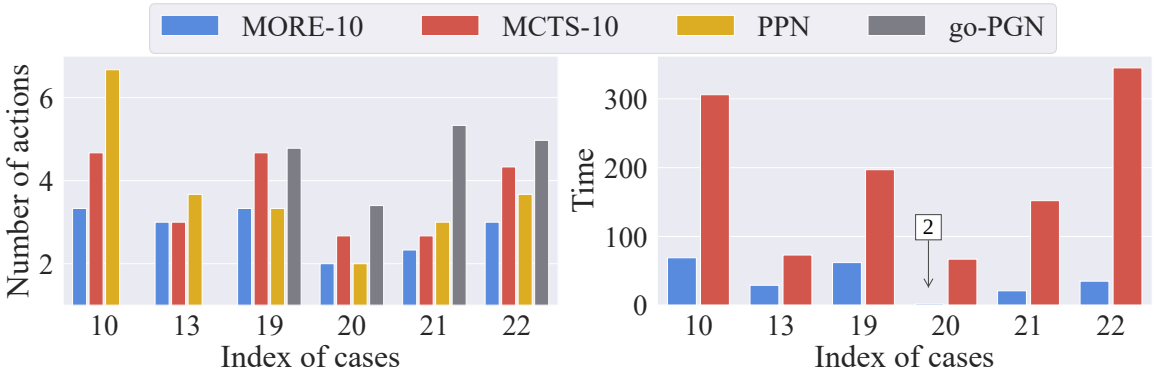


Figure 5.9: The number of action and time used on solving six cases. The budget is up to 10 iterations for MCTS and MORE.

5.5 Summary

The main limitation of this work is that we need to know the models of the objects to do the planning. One possible solution is instead of using an explicit simulator, we can use a learned model Chapter 3 to simulate the push results. Generalization to novel objects could then be possible. We can further utilize the Push Prediction Network to estimate the simulation (rollout) result instead of using a physics engine. However, this can introduce additional uncertainties that typically result from using DNNs, which can cause unexpected behaviors such as pushing objects out of the workspace. Building on the know-how gained from developing MORE, we are exploring other real-world robotic manipulation tasks that would benefit from the S2→S1 search-and-learn philosophy. We point out that MORE can be further sped up by implementing a parallel version of MCTS, as we only utilized a single

CPU thread in our implementation and PPN (on GPU) is not being used most of the time.

CHAPTER 6
PARALLEL MONTE CARLO TREE SEARCH WITH BATCHED RIGID-BODY SIMULATIONS FOR SPEEDING UP LONG-HORIZON EPISODIC ROBOT PLANNING

6.1 Introduction

The past decade has witnessed dramatic leaps in robot motion planning for solving problems that involve sophisticated interaction between the robot and its environment, with milestones including teaching quadrupeds to perform impressive tricks [129], [130] and navigate challenging terrains [131], enabling high-DOF robot hands to solve the Rubik's cube [10], and so on. While some of the success can be attributed to the rapid advancement in deep learning [132] and deep reinforcement learning [133], another undeniable factor is the availability of fast, high-fidelity physics engines, including PyBullet [125] and MoJuCo [134]. These physics engines allow the simulation of the physics of complex rigid-body systems, sometimes faster than real-time, which enables the collection of large amounts of realistic system behavior data without even touching the actual robot hardware. Nevertheless, most physics simulators are CPU-based, which can only simulate a limited number of robots simultaneously; this has led to some studies seeking parallelism by using a massive amount of computing resources. For example, the OpenAI hand study [10] used a total of $6,144$ CPU cores to train their model for over 40 hours, which is costly and time-consuming.

As physics simulation starts to become a bottleneck in solving robotic tasks, GPU-based physics engines have recently begun to emerge, including Isaac Gym [135] and Brax [136], to address the issue by enabling large-scale rigid body simulation. Early results are fairly promising; for example, the training of the OpenAI hand using Isaac Gym can be done on a single GPU in one hour, translating to a combined resource-time saving of several orders of

magnitude. Similar success has also been realized in applying reinforcement learning on quadrupeds, robotic arms, and so on [135].

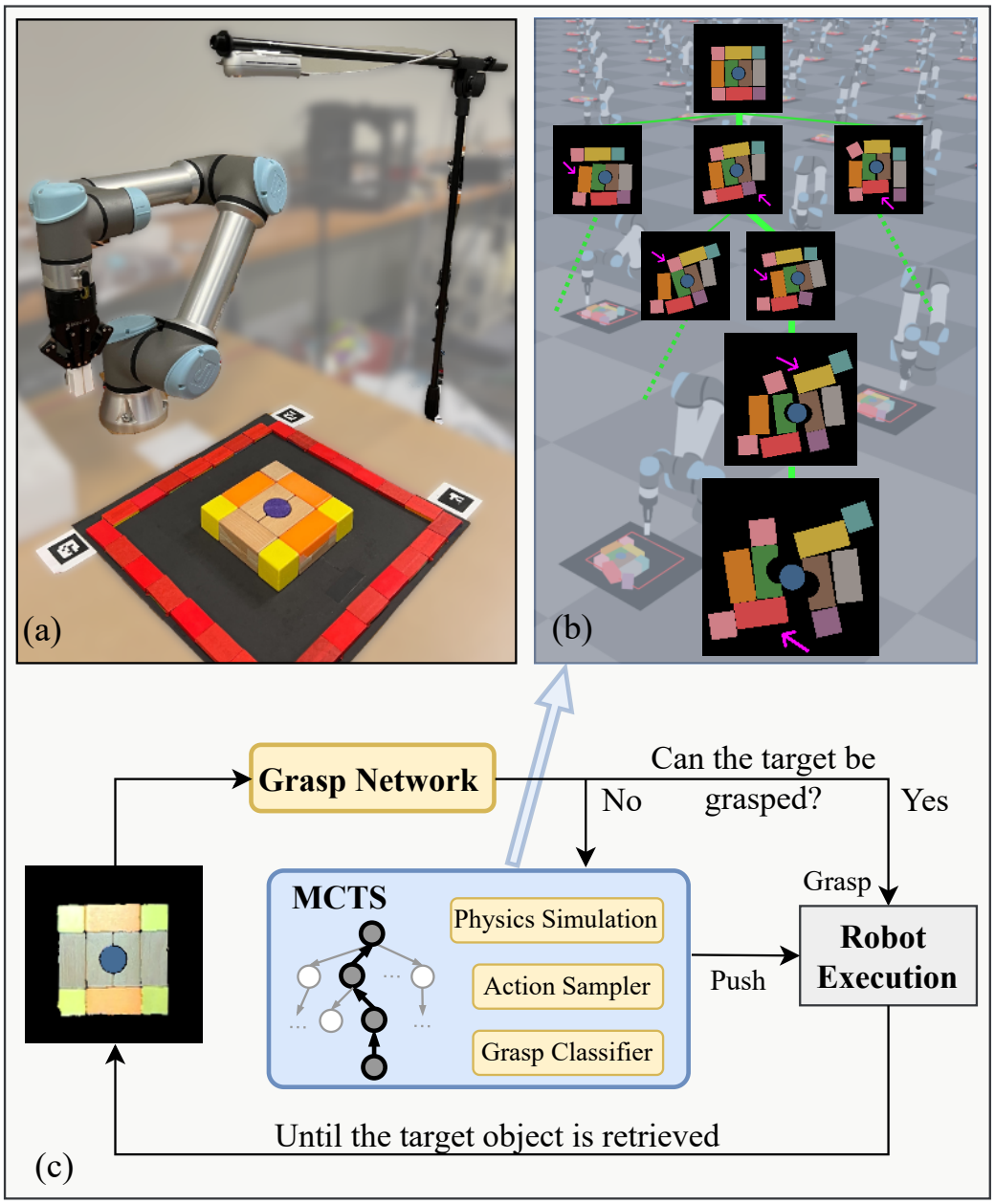


Figure 6.1: (a) The hardware setup includes a Universal Robots UR-5e with a Robotiq 2F-85 two-finger gripper and an Intel RealSense D455 RGB-D camera. (b) Planning and simulation carried out in physics simulator where thousands of virtual robots operate in parallel. (c) Overview of our system; the small blue cylinder at the center is the target object to be retrieved.

In this work, we exploit the power of large-scale rigid body simulation for optimally

solving long-horizon episodic robotic planning tasks, such as multi-step object retrieval from clutter, leveraging the strength of another powerful tool that has attracted a great deal of attention – Monte Carlo tree search (MCTS) [122]. MCTS demonstrates clear advantages in solving long-horizon optimization problems without the need for significant domain knowledge [137], and was already employed for solving challenging manipulation tasks Chapter 4 [16] and Chapter 5 [19]. However, even with significant guidance using domain knowledge Chapter 5, MCTS incurs fairly long planning times due to its need of carrying out numerous rounds of sequential *selection-expansion-simulation-backpropagation* cycles. The long planning time, sometimes several minutes per decision step, limits the applicability of the methods from Chapter 4 and Chapter 5 toward real-time decision making.

Through combining MCTS and large-scale rigid-body simulation with Isaac Gym [135], and carefully introducing parallelism into the mix, we have developed a new line of parallel MCTS algorithms for efficiently solving long-horizon episodic robotic planning tasks. The development of the large-scale rigid-body simulation enabled parallel MCTS is the key contribution of this research, which is highly non-trivial. This is because MCTS has an inherently serial characteristic; as will be explained in more detail, the *selection* phase of an MCTS iteration depends on the completion of the previous selection-expansion-simulation-backpropagation iteration. Fusing MCTS and Isaac Gym for solving long-horizon manipulation planning tasks also brings significant technical integration challenges because many computational bottlenecks must be addressed for the parallel MCTS implementation to be efficient.

We call our algorithm **P**arallel **M**onte Carlo tree search with **B**atched **S**imulation (PMBS). As its name suggests, PMBS realizes parallel MCTS computation through batched rigid-body simulation enabled by Isaac Gym. Efficiently combining MCTS and Isaac Gym, PMBS achieves over $30\times$ speedups in planning efficiency for solving the task of object retrieval in clutter, while still achieving better solution quality, as compared to an optimized serial MCTS implementation, using identical computing hardware. PMBS drops the single-step

decision making time to a few seconds on average, which is close to being able to solve the task in real-time. We further demonstrate that PMBS can be directly applied to real robot hardware with negligible sim-to-real differences.

6.2 Problem Formulation

In this paper, we task a robot equipped with a camera and a two-finger gripper to grasp a desired object from a densely packed clutter, as a concrete instance of long-horizon episodic robot planning problems. The workspace is a confined planar surface. Two types of primitive actions are allowed: pushing and grasping. All objects are rigid; the target object has a different color to facilitate its detection. The only observation available to the robot is an RGB-D image that is taken by a top-down fixed camera, as shown in Figure 6.1. Every time the robot executes a push or a grasp action, a new image is taken. A similar problem has been previously defined in Chapter 3 [12], Chapter 4 [16], and Chapter 5 [19]. Compared to [16], [19], the problem addressed in the present work is significantly more challenging to solve because the workspace is confined to a substantially smaller area, while keeping the number and sizes of objects the same. Consequently, the free space between the objects is reduced, and the robot needs to find a larger number of shorter surgical push actions in order to free the target object and grasp it. In fact, we found from our experiments (section 6.4) that the original setup considered in [16], [19] can be solved using a brute-force parallel search in a GPU-based physics simulator, without a Monte Carlo tree search.

6.3 Methodology

MCTS builds a search tree, balancing exploration and exploitation, by iteratively performing *selection-expansion-simulation-backpropagation* operations. In the *selection* phase, MCTS selects a best node to grow the tree. A popular node selection criterion is based on the *upper*

confidence bound (UCB) [138], [139],

$$\underset{n' \in \text{children of } n}{\arg\max} \frac{Q(n')}{N(n')} + c\sqrt{\frac{2 \ln N(n)}{N(n')}}, \tag{6.1}$$

where $Q(n)$ is the sum of rewards collected starting from the state corresponding to node n, $N(n)$ is the number of times n was selected so far. The selection process continues until it finds a node that corresponds to a terminal state or a node that has never-explored children. We note that a node n is always associated with a state s and an observation o; sometimes a node n and the corresponding state s are used interchangeably. After a node n is selected, if it is not a terminal node, it is *expanded*, and its new child, say n', is added to the search tree. Subsequently, a *simulation* will be carried out at n'. This *selection-expansion-simulation* process is repeated until a terminal state (or a stopping condition) is reached, which yields a reward. The obtained terminal reward is *propagated back* from n' all the way to the root node, while updating the sum of reward ($Q(n)$) and incrementing the number of visits ($N(n)$) for all the nodes along the path.

Effectively employing MCTS to tackle long-horizon episodic robot planning requires a highly non-trivial adaptation of MCTS. In this section, we first describe the necessary preparation for integrating MCTS and physics simulation for object retrieval, then describe augmentations to the architecture for GPU-based processing, and finally outline our key ideas and design choices in our parallelization effort.

6.3.1 Serial MCTS for Object Retrieval from Clutter

To use MCTS for the object retrieval task and solve real instances, we integrate it in a process that alternates between search in simulation and execution on the real system. Our MCTS process takes in a scene that is segmented into objects, and uses physics simulation to reason about the proper push actions to facilitate the final retrieval of the target object. An overview of the MCTS process is provided in Figure 6.1. In other words, we first replicate in the

simulator the real perceived scene at the beginning of each episode, perform computation and simulation, and then execute with the real robot the action that results from the simulation to guide the resolution of the retrieval task on the real objects.

We now describe the details of our basic serial MCTS adaptation. For the selection step, the standard UCB formula is used. For the expansion step, for a selected node n that has not been expanded, we sample many potential push directions by examining the contour of the objects. These sampled pushes become the candidate actions under n for expansion. After a sampled push action is chosen, the action is executed in the physics simulation and a new node is added to the tree. The MCTS simulation step is then carried out with additional consecutive random pushes to obtain a reward for the newly added node. Note that for each simulation step, we must decide whether the resulting state is a terminal state; this is done using a *grasp classifier*, to be explained later.

An important design decision we make here, to render MCTS computation more tractable, is to limit the depth of the tree. We limit the depth of the overall tree to be no more than some d_T. The simulation can be carried out for at least d_s steps. This means that the maximum depth reached by MCTS does not exceed $d_T + d_s$. If expansion happens at depth d_T, we allow the state to be simulated further until $d_T + d_s$. Given our goal of finding the least number of pushes for retrieving the target object, d_T and d_s can potentially be dynamically updated when an identified terminal node has a depth d smaller than d_T. In this case, we set $d_T = d$ and $d_s = 0$. We terminate an MCTS process if: (1) the elapsed time exceeds a preset budget T_{max}, (2) the tree is fully explored, or (3) the target can be grasped in an explored node and all nodes at its parent's level have been explored.

After each full MCTS run, we execute the best action it returns on the actual scene (simulated or real), and then use a *grasp network* (GN) from Chapter 5 to tell us whether the target object is retrievable. If it is, GN further tells us how to grasp it; the task is then completed. Details of GN, for replication purposes, can be found in the online supplementary material.

6.3.2 Adaptions for GPU

Besides simulation, which can be sped up using GPU-based physics engines, there are three additional bottlenecks in the process to parallelize MCTS for object retrieval. One of these is the parallelization of MCTS itself and the other two are specific to the object retrieval problem: action sampling and grasp feasibility prediction. We leave the first bottleneck for subsection 6.3.3 and address the latter two here.

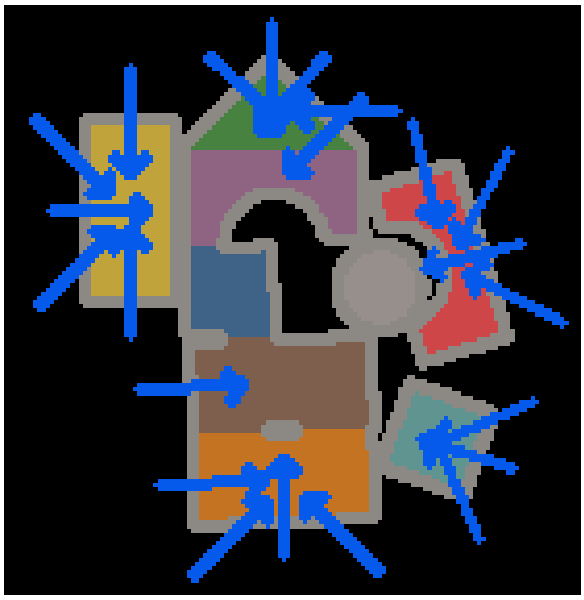


Figure 6.2: Sampled push actions.

Speeding up Action Sampling. Because the number of push action choices is uncountably infinite, action sampling is necessary. We modified the action sampler from Chapter 4 and Chapter 5 with slight changes and a more efficient implementation. As shown in Figure 6.2, for a given o_t, actions a_t^p are sampled around the clutter. N_a actions are evenly sampled around the contour of each object, from edge to center. Actions that cannot be executed due to collisions are discarded. Further speedups are obtained by pre-computing the sampled actions for each object and only performing collision checking between the robot's start pose of push and objects at runtime.

Grasp Evaluation. Previous learning-based methods for object retrieval use a *grasp*

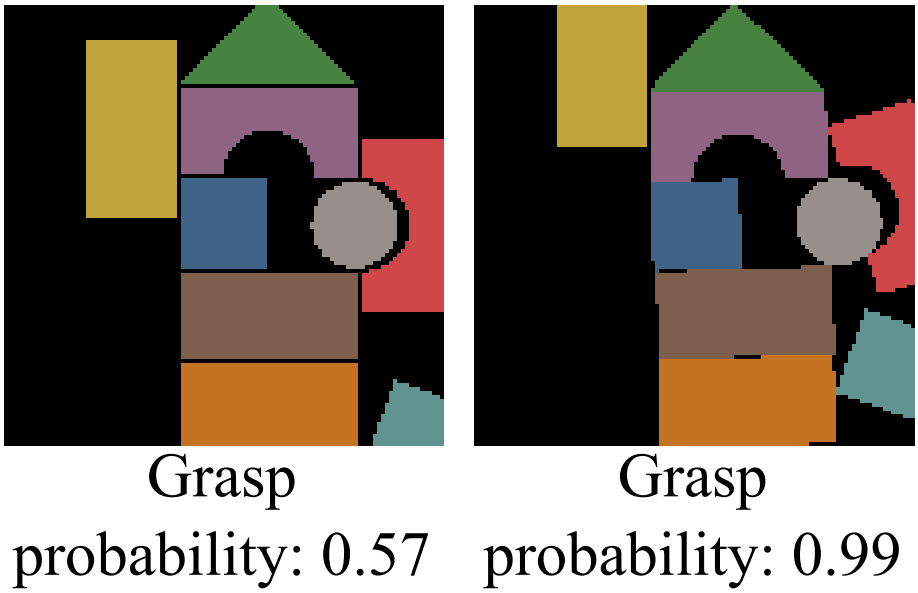


Figure 6.3: Examples of using the *grasp classifier* to produce probabilities to grasp the object at center (blue in this case). Here we used an RGB image for illustration purpose (input should be a depth image).

network (GN) to evaluate the feasibility of grasping the target object (Chapter 4 and Chapter 5), which becomes a time-consuming bottleneck when parallelized directly. GN is relatively slow because it evaluates a large number of possible grasp poses. However, knowing the best grasp pose is unnecessary if we only want to know how 'graspable' a state is. Given this observation, we develop a simplified *grasp classifier* (GC) that only returns a grasp probability (Figure 6.3). GC is an EfficientNet-b0 [140] that takes a depth image as input and outputs a probability between 0 and 1. Given a depth image and a target object, we can query GC whether the target object is graspable by comparing its output to a preset threshold R_c^*. Details about GC's implementation and training in simulation can be found in the online supplementary material. Note that GN is still used after each full MCTS run for potentially grasping the target object, as described in subsection 6.3.1.

6.3.3 Parallel MCTS with Batched Simulation

Given the availability of powerful GPU-based physics simulators including Isaac Gym [135] and Brax [136], which enables the simulation of a large number of systems independently

and simultaneously, a natural route for speeding up long-horizon episodic robot planning tasks is to introduce parallelism into the MCTS pipeline outlined in subsection 6.3.1, to perform many simultaneous simulations. However, it is challenging to introduce parallelism into MCTS because optimal node selection depends on the reward of all previous rounds. To enable parallelism in MCTS for object retrieval from clutter and harness GPU-based simulation, we introduce the following modifications to the MCTS procedure outlined in subsection 6.3.1. We assume the number of parallel environments in the simulator is fixed to some number N_e. Each parallel environment contains an identical virtual robot and objects.

Selection with Virtual Loss. By observing the operations of MCTS, it is not difficult to see that the parallelism of MCTS requires modifying the UCB formula. Otherwise, the same node in a search tree will be selected for expansion in multiple parallel environments, leading to redundancy and poor performance. To address this issue, we adopt the idea of *virtual loss* [141], which has shown to give good results in multiple application domains [119], [142], [143]. Virtual loss is used to adjust the calculation of UCB values for the nodes that have been selected but not yet expanded [111], [141],

$$\underset{n' \in \text{children of } n}{\arg\max} \frac{Q(n')}{N(n') + \hat{N}(n')} + c\sqrt{\frac{2\ln\left(N(n) + \hat{N}(n)\right)}{N(n') + \hat{N}(n')}}, \tag{6.2}$$

where the $\hat{N}(n)$ is the number of selected but not yet expanded nodes under node n. $\hat{N}(n)$ will be reset to zero once the selection phase of parallel MCTS is completed. Basically, Equation 6.2 penalizes selecting nodes that have already been selected in some other parallel environment but for which expansion and simulation have not yet been completed. With Eq. (Equation 6.2), it is still possible for a node n to be selected multiple times, which may lead to redundant simulations. To avoid this and ensure that no redundant simulations are carried out, we mark all selected actions of node n and share this information across all the parallel environments.

A collection of state-action pairs is returned from the selection phase. The same state

could be selected many times, but all state-action pairs in the selected collection are unique. For example, in Figure 6.4, upper left, $(s^1_{t+2}, a^1), \ldots, (s^4_{t+1}, a^4)$ are four such state-action pairs. Batch-mode expansion/simulation on this collection is then performed in parallel using GPU.

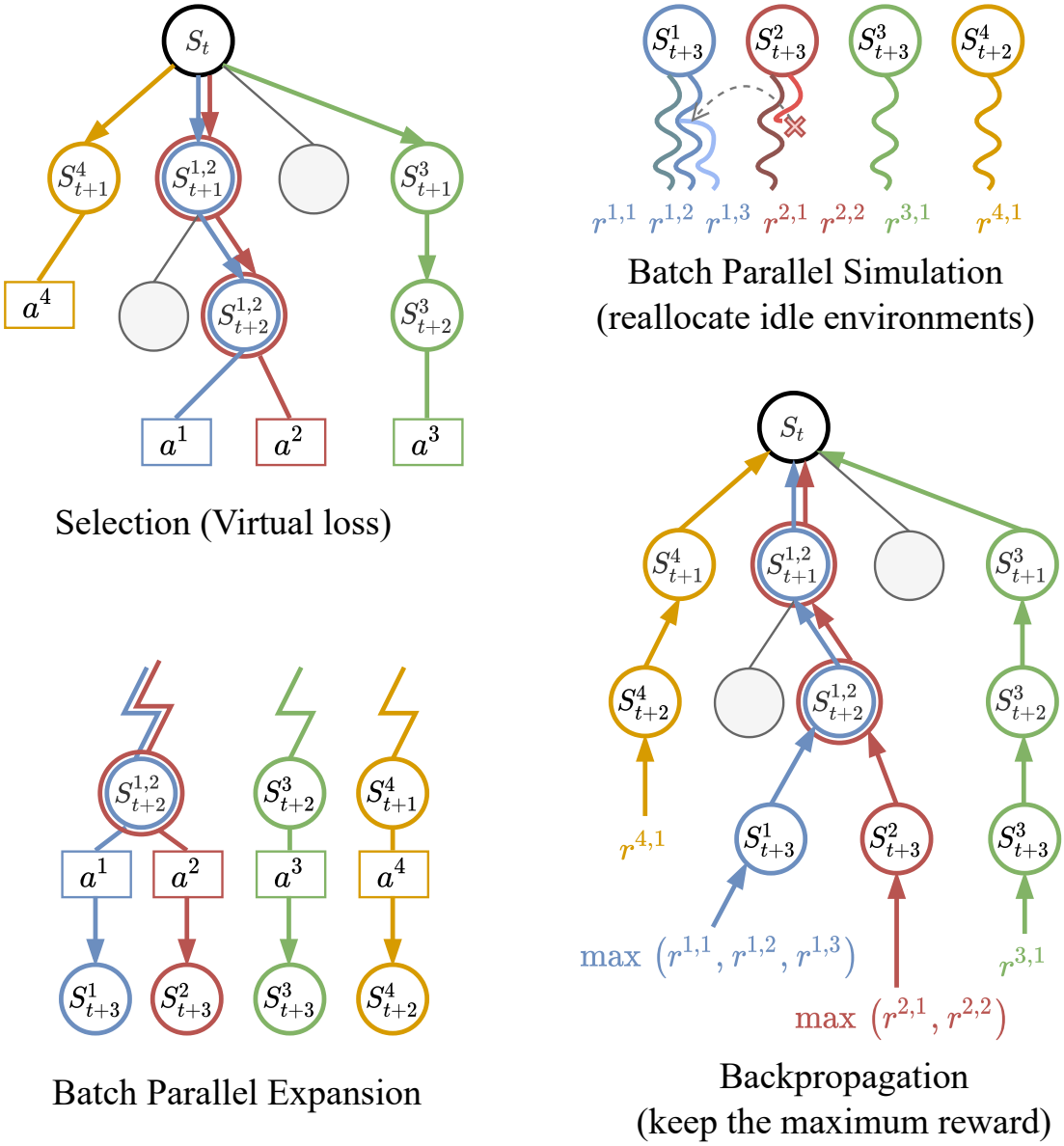


Figure 6.4: Steps in PMBS, our parallel MCTS with batched operation.

Batch Expansion. After a batch of state-action pairs ($\{(s, a)\}$) has been selected, the expansion step is carried out for all of these pairs simultaneously. For this purpose,

environments in the simulator (Isaac Gym) will be loaded with the appropriately selected states, after which expansion (transition) is carried out in parallel. A batch expansion creates a set of new nodes added to the tree, each of which is different. In Figure 6.4, lower left, $s^1_{t+3}, \ldots, s^4_{t+2}$ are the result of expanding $(s^1_{t+2}, a^1), \ldots, (s^4_{t+1}, a^4)$, respectively.

Algorithm 4: Parallel MCTS with Batched Simulation

Function `Main`(s_t, o_t)
  while there is a target object in workspace **do**
    if the target object can be grasped (query GN) **then**
      Execute grasp of the target object
    else Execute `Parallel-MCTS`(s_t) // Push

Function `Parallel-MCTS`(s)
  Create root node n_0 with state s
  es_level $\leftarrow 1$ // Early stop level
  graspable_nodes $\leftarrow \varnothing$
  while (within time budget) and (depths of all graspable_nodes are greater than es_level) **do**
    $[(n^1, a^1), \ldots, (n^{N_e}, a^{N_e})] \leftarrow$ `Selection`(n_0)
    Reset all $\hat{N}(n)$ to 0
    $[n'^1, \ldots, n'^{N_e}] \leftarrow$ `Expansion`$([(n^1, a^1), \ldots, (n^{N_e}, a^{N_e})])$
    for n' in $[n'^1, \ldots, n'^{N_e}]$ **do**
      if $GC(n'(o)) > R^*_c$ **then**
        graspable_nodes $\leftarrow$ graspable_nodes $\cup \{n'\}$
    if all nodes at es_level $- 1$ are fully expanded or terminal **then**
      es_level $\leftarrow$ es_level $+ 1$
    $[r^1, \ldots, r^{N_e}] \leftarrow$ `Simulation`$([n'_1, \ldots, n'_{N_e}])$
    `Backpropagation`$([(n'^1, r^1), \ldots, (n'^{N_e}, r^{N_e})])$
    return the a^p that leads to best child node of root, ranked by Equation 6.2

Batch Simulation. The batch simulation step of our parallel MCTS implementation is similar to the batch expansion step, but with additional steps inserted before and after. Before a push simulation, random actions must first be selected, using the action sampling method outlined in subsection 6.3.2. After each push simulation, GC, as described in subsection 6.3.2 is applied to evaluate the outcome. As we can see, to best exploit the parallelism from the simulator, action sampling and GC should be carried out as efficiently as possible, so that they do not become significant computational bottlenecks.

During simulation, we also perform *leaf parallelization* [141] when the number of simulation environments is more than the number of states for which MCTS simulations are to be carried out. This is reflected in Figure 6.4, upper right, where the first two states each are simulated twice initially. If some environments, after a push, are predicted by GC as graspable, then further simulation on these environments will not be carried out, and these environments can be re-purposed. For example, in Figure 6.4, upper right, a simulation under the second state terminates early, and the associated environment can be used to perform additional simulation for the first state.

Backpropagation. The backpropagation phase is straightforward to execute, as it simply backpropagates the rewards to the root of the tree. We note that, for a single state for which multiple simulations are carried out, it is natural to select the maximum reward obtained instead of taking averages (see Figure 6.4, lower right).

The pseudo-code of PMBS is given in Algorithm 4 with the selection subroutine given in Algorithm 5. Other subroutines of PMBS are mostly straightforward.

Algorithm 5: Selection with Virtual Loss

```
Function Selection(n_0)
    Pairs ← ∅
    while len(Pairs) < N_e do
        n^i ← traverse tree until a leaf node using Equation 6.2
        a^i ← one sampled action of n^i
        Remove a^i from the sampled actions of n^i
        Pairs ← Pairs ∪ {(n^i, a^i)}
        N̂(n^i) ← N̂(n^i) + 1
        increment virtual counts of ancestor nodes of n^i
    return Pairs
```

6.4 Experimental Evaluation

We evaluated the proposed system (PMBS) in a physics simulator (Isaac Gym) and on a real robot on adversarial test cases. In comparisons to baseline and ablation studies, we observe significant improvements using the GPU-based physics simulator together with

parallel MCTS, which brings episodic decision making for real robots closer to real-time, i.e., a single complex decision is made in a few seconds. All experiments were conducted on a desktop with an Nvidia RTX 2080Ti GPU, an Intel i7-9700K CPU, and 32GB of memory.

6.4.1 Simulation Studies

In this work, the simulated environment is built with Isaac Gym [135], consisting of a Universal Robot UR5e with a two-finger gripper Robotiq 2F-85, and an Intel RealSense D455 RGB-D camera overlooking a tabletop workspace as shown in Figure 6.1. The robot is in position-control mode; push and grasp actions control the end-effector's position, and Inverse Kinematics (IK) [144], [145] is applied to convert these to joint space commands. The effective workspace is at a size of 0.288×0.288m, discretized as a grid of 144×144 cells where each cell is one pixel in the image (orthographic projection) taken by the camera. The workspace, in comparison to previous Chapter 4 and Chapter 5, is significantly smaller (only about 45% in terms of area), making the setting much more challenging. We intentionally selected the setting to demonstrate the power of PMBS.

All objects should reside in the workspace. 20 cases from Chapter 4 used for evaluation can be found in Figure 6.5, where the red lines denote the boundary to which objects centers must be confined at all times.

The push distance of a push action is fixed at 5cm (10cm in previous Chapter 4 and Chapter 5, the effective push distance is around 3cm (the distance objects are moved).

Metrics. Four metrics are used to evaluate our systems:

1. the number of actions used to retrieve the target object
2. the total planning time used for retrieving the target object (build the tree)
3. the completion rate in retrieving the target object within 16 actions
4. the grasp success rate, which is the number of successful grasps divided by the total number of grasping attempts

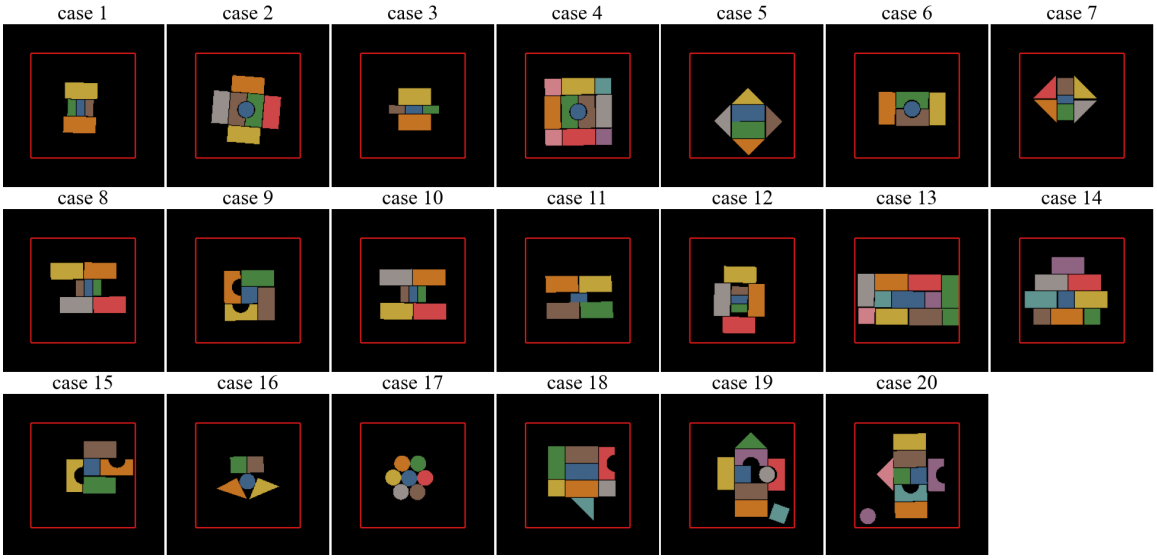


Figure 6.5: 20 cases from Chapter 4 used in simulation experiments, where the target object has a blue mask. No object should exceed the boundary (red lines).

Baseline. We use an optimized serial MCTS implementation as the baseline, where the number of environments used for MCTS is one. The following hyperparameters are used across all methods in benchmark unless otherwise mentioned. The discount factor $\gamma = 0.8$. The default maximum tree depth is $d_T = 7$, and the default simulation (rollout) depth is $d_s = 3$. The threshold of GC is $R_c^* = 0.9$. The UCB exploration term c in Equation 6.1 and Equation 6.2 is 0.3. The time limit (budget) T_{max} for one step planning is 60 seconds. 1000 robots (environments) in Isaac Gym are used in our PMBS; it takes around 2.2 seconds for all robots to complete one push action.

We evaluate the performance of PMBS and the baseline serial MCTS over all 20 cases, running each case five times. For the evaluation, we set a time budget $T_{\text{max}} = 60$ s and denoted the two methods as PMBS-60 and MCTS-60, respectively. The summary benchmark for these two methods can be found in the first two rows of Table 6.1; individual results for each case can be found in Figure 6.6 and Figure 6.7.

We make some observations based on the results. First, PMBS outperforms the serial MCTS version in terms of number of actions and computation time across all cases, which is as expected because PMBS engages many environments to facilitate its search effort.

On the other hand, when we view the solution quality and computation time together, the advantage of PMBS over serial MCTS is significant: PMBS-60 uses 35 seconds on average for planning, whereas MCTS-60 uses over 300 seconds. This along translates to an $8.6\times$ speedup. At the same time, PMBS-60 uses 70% fewer actions in solving the tasks. A further data point regarding the speedup at the same solution is given a bit later in Figure 6.8.

A second observation is that, despite the fact that we are dealing with a difficult long-horizon planning problem, PMBS is able to achieve planning that is close to being able to perform reasoning in real-time, as it takes an average of $35/3.91 < 9$ seconds to make a single decision. With further optimization and/or better hardware, we believe that PMBS will achieve real-time decision-making capability for the current set of object retrieval tasks.

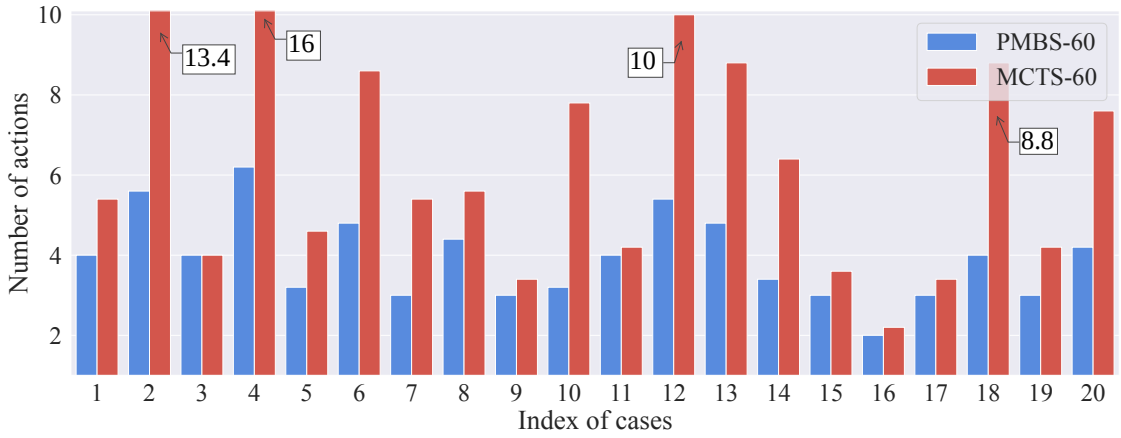


Figure 6.6: The average number (over five independent trials) of actions per case needed for solving the twenty cases, given a time budget of 60 seconds.

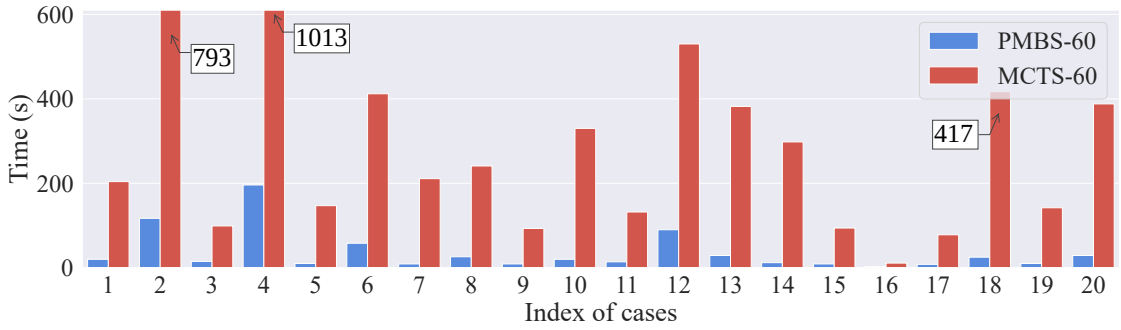


Figure 6.7: The average time (over five independent trials) per case needed for solving the twenty cases, given a time budget of 60 seconds.

	Num. of Actions	Time	Completion	Grasp Success
PMBS-60	**3.91**	**35**s	100%	98.3%
MCTS-60	6.67	301s	93.0%	96.4%
PMBS-60 ($c = 0$)	4.03	113s	100.0%	99.2%
PMBS-60 ($c = \infty$)	12.71	147s	42.0%	96.7%

Table 6.1: Simulation experiment results for 20 cases. Time budgets are limited up to 60 seconds.

Ablation Study. The time budget T_{max} is one of the main factors that influence the solution quality and planning time. To understand its role, several time budgets are used to evaluate our method, as shown in Figure 6.8. Given more time for tree search, serial MCTS and PMBS could improve the solution quality, leading to fewer required actions. The trends of serial MCTS (number of action) are steep, as it is highly possible that it could not find a solution given a limited time. The trend of PMBS (planning time) is more gradual, as the most time-consuming search happens in the first few iterations, which usually uses all the time budget. While serial MCTS never achieves the same solution quality as PMBS, comparing the first PMBS data point and the last serial MCTS data point, we observe a $855/28 = 30\times$ speedup with PMBS still has some quality advantage.

On the flip side, we note that the speed-up of $30\times$ seems small considering that we used 1000 environments. This is due to two factors. First, MCTS is itself a serial process; parallelization will incur performance loss. Second, while we have improved many bottlenecks, e.g., on action sampling and grasp classification, the object retrieval task contains many elements that cannot be readily parallelized.

We also evaluated the impact of the exploration and exploit trade-off on PMBS. If the c in Equation 6.2 is set to 0, i.e., pure exploitation, PMBS-60 uses 4.03 actions and 113 seconds (planning time) on average on 20 cases. The performance is worse than when $c = 0.3$, as shown in Table 6.1. This is expected as the greedy approach could be stuck in a local optimum. PMBS-60 is also tested by setting c to be a large number in Equation 6.2, i.e., pure exploration, which uses 12.71 actions and 147 seconds (planning time) on averages; the completion rate has a steep drop to 42.0%.

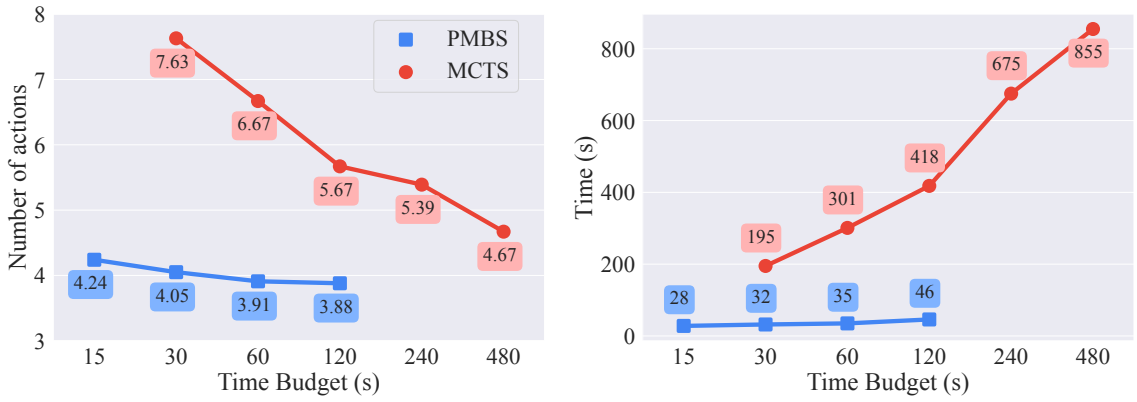


Figure 6.8: PMBS and serial MCTS evaluated with different time budgets. The reported values are averages over all 20 cases.

6.4.2 Real Robot Experiments

For experiments on the physical UR-5e, the input to PMBS is a single RGB-D image. A 1280×720 RGB-D image is taken, then it is orthogonally projected over the workspace of resolution of 144×144 (with cropping). Since the same robot and objects are used in both simulation and the real world, we can observe and act on a real robot but plan in a simulator. For each object, simple pose estimation is performed to transfer the perceived scene from real images to the physics simulator environments.

The pose estimation is done by firstly extracting masks for objects from the image, then a brute-force matching between detected mask and recorded mask is performed for each object. We could achieve it at 0.15 seconds for one image (around 10 objects). Serial MCTS and PMBS were evaluated the same way as in simulation experiments, except we only run tests on the six most challenging cases. Individual benchmarks on six cases can be found in Figure 6.9. Average statistics are listed in Table 6.2. We observe minimal sim-to-real performance loss; A small gap exists between the real and the simulation experiments, mainly due to pose estimation errors and mismatch of physics properties.

Additional Experimental Details. Curious readers may find in the online supplementary material additional experimental details including complete, actual execution snapshots of PMBS and MCTS for all 20 cases, as well as the execution snapshots for real robot

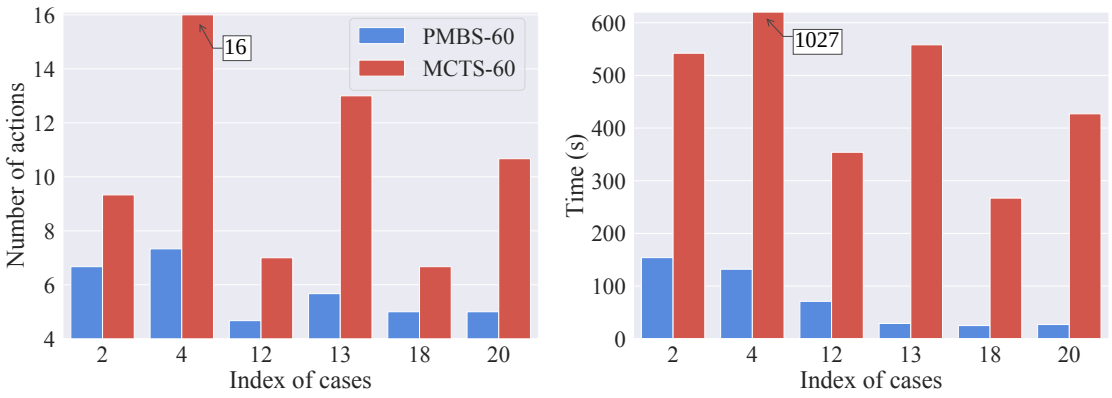


Figure 6.9: The number of actions and time used for solving the six most challenging cases on the physical robot. The time budget is 60 seconds.

	Num. of Actions	Time	Completion	Grasp Success
PMBS-60	**5.72**	**73**s	100%	100%
MCTS-60	10.45	529s	83.3%	87.0%
PMBS-60 (sim)	5.03	81s	100%	97.2%
MCTS-60 (sim)	10.77	587s	76.7%	96.6%

Table 6.2: Real robot experiment results on the six most difficult cases. Time budgets are limited to 60 seconds per case.

experiments.

6.5 Summary

In this chapter, we proposed PMBS, a novel parallel Monte Carlo tree search technique with GPU-enabled batched simulations for accelerating long-horizon, episodic robotic planning tasks. Through a series of careful design choices to overcome major parallelization bottlenecks, PMBS achieves an over $30\times$ speedup compared to an optimized serial MCTS implementation while also delivering better solution quality, using identical computing hardware. Real robot experiments show that PMBS directly transfers from simulation to the real physical world to achieve near real-time planning performance in solving complex long-horizon episodic robot planning tasks.

CHAPTER 7
TOWARD OPTIMAL TABLETOP REARRANGEMENT WITH MULTIPLE MANIPULATION PRIMITIVES

7.1 Introduction

Real-world manipulation tasks, e.g., rearranging a messy tabletop or furniture in the house, often require multiple manipulation primitives (e.g., pick-n-place, pushing, toppling, etc.) to accomplish. When rearranging small/light objects, e.g., a cell phone on a table or a small chair in a room, it is convenient to do a *pick-n-place*, i.e., to pick up the object, lift it above other objects, move it across the space to above its destination on the table, and then place it. On the other hand, for handling large/heavy objects, e.g., a thick book or a heavy couch, *pushing* or *dragging* close to the space's surface is more commonly adopted, executed with added caution. In this case, planning the object's motion trajectory must consider avoiding colliding with other objects more carefully. Solving such long-horizon task-and-motion planning tasks efficiently and optimally is highly challenging, as it involves not only an extended horizon but also selecting among multiple types of manipulation primitives at each step, both of which add to the combinatorial explosion of the search space.

Toward quickly and optimally solving rearrangement tasks using multiple manipulation primitives, we focus on a tabletop setting where both *pick-n-place* and *pushing* are employed to rearrange objects (see Figure 7.1). Many objects, such as those shown in Figure 7.1(e), cannot be easily picked up and moved around without damaging or disassembling the object. For example, as shown in Figure 7.1(f)(g), books and certain boxes cannot be moved around using suction-based pick-n-place manipulation primitive (note that it is also difficult to do pick-n-place using fingered grippers). However, these objects can be effectively rearranged using a *pushing* manipulation primitive in which the suction-based end-effector holds the

Figure 7.1: (a) Overview of system setup, a camera is mounted on the end-effector for perception. (b)-(d) An example case and an intermediate step in solving it. (e) Example objects requiring a *push*. (f) *pick-n-place* may break the book. (g) *pick-n-place* will separate a box, failing to pick it up.

object on or close to the tabletop and pushes/drags the object around (see Figure 7.1(c)). We call the frequently encountered yet largely unaddressed problem *rearrangement with multiple manipulation primitives* (REMP). This study on REMP brings the following contributions:

- With the formulation of REMP, we propose a first formal study of solving long-horizon rearrangement tasks utilizing multiple distinctive precision manipulation primitives with the goal of computing an optimized manipulation sequence. Due to its high practical relevance, REMP constitutes an important specialized task and motion planning problem.
- We developed two novel algorithms for REMP, the first of which is a fast rule-based solution capable of effectively and quickly solving non-trivial REMP instances. The second, leveraging Monte Carlo tree search (MCTS) [122] and parallelism to look further into the planning horizon, delivers a much higher success rate for more challenging tasks, providing higher-quality solutions simultaneously.
- We thoroughly evaluate our methods in simulation and extensive real robot experiments. In particular, our real robot experiments with integrated vision solutions, demonstrate that our algorithms can be readily applied to interact with everyday household objects in real-world scenarios.

7.2 Problem Formulation

7.2.1 Rearrangement with Multiple Manipulation Primitives

We now specify the concrete *rearrangement with multiple manipulation primitives* (REMP) studied in this work. Let the workspace be $\mathcal{W}$ a 2D rectangle. The robot is provided with a start image (state) s_s and a goal image (state) s_g containing the initial and desired object arrangements. The robot must rearrange the objects to match the configurations specified in s_g. Two manipulation primitives are permitted: *pick-n-place* $\mathcal{A}_{pp}$ and *push* (from top) $\mathcal{A}_{pt}$. The robot's objective is to complete the task efficiently in terms of the *execution time*. The start and goal states and the objects' transportation should be collision-free, and all objects should remain within the workspace. It is assumed that all tasks are feasible, i.e., there is always a viable solution $\mathcal{P} = \{a_1, a_2, ..., a_n\}$ leading from the start state s_s to the goal state s_g, where $a \in \{\mathcal{A}_{pp}, \mathcal{A}_{pt}\}$.

A state s_t represents the pose of objects at time t. A pick-n-place action is specified by a pick pose (x_0, y_0, θ_0) and a place pose (x_1, y_1, θ_1). A push action is specified by trajectories $\{(x_0, y_0, \theta_0), ..., (x_n, y_n, \theta_n)\}$, where the robot holds the object against the tabletop at the initial pose, and then pushes it following the waypoints, ending at the final pose.

One assumption is that the object should be capable of being stably positioned on the workspace, as all primitives are considered to be quasi-static.

7.2.2 Monte Carlo Tree Search

The Monte Carlo tree search (MCTS) algorithm has broad applications. It is prevalent in turn-based tasks such as the game of Go [146], but its usage extends beyond such contexts. MCTS plays a crucial role in solving rearrangement tasks [21], [147], [148]. As an *anytime* tree search algorithm, MCTS is designed to run for a fixed amount of time, each consisting of four stages: selection, expansion, simulation, and backpropagation. Fundamentally, MCTS preferentially exploits nodes that yield superior outcomes. To strike a balance between exploration and exploitation in the selection phase, an *upper confidence bound* (UCB) [138] formula is utilized (see Equation 6.2), where n is the parent node of n', and $Q(n')$ is the total reward n' received after $N(n')$ visits.

7.3 Methodology

This work addresses the primary challenge of synergistic integration of pick-n-place and push actions. Because planning a push involves collision-free path planning in $SE(2)$, which is time-consuming, it poses a significant challenge if many push actions are explored. MCTS offers a solution capable of elegantly negotiating between the two disparate actions while maintaining optimality, given ample planning time.

7.3.1 Action Space Design

Planning requires searching through candidate manipulation actions, which must first be *sampled*. Action space design refers to action sampling, which plays a critical role in dictating the expansion of the tree search because there are an uncountably infinite number of possible manipulation actions. In sampling pick-n-place actions, we must ensure the place pose is collision-free. The same applies to a push action's final pose (though, in addition, the entire trajectory connecting all waypoints for a push action must be collision-free). Four criteria are applied to sample the place/final poses for pick-n-place/push actions at the current state s_t:

1. **Random**. A naive approach randomly samples collision-free place/final poses for pick-n-place/push actions.

2. **Around Current and Goal**. Random sampling is not always efficient. For object o_i, favoring regions around the current and goal poses of o_i in s_g can be helpful.

3. **Grid**. Additionally, we adopt a grid-based sampling strategy. By modeling an object as a 2D polygon, we encapsulate it within a rotated bounding box. This allows us to generate a tiled representation in the workspace, denoted as $\mathcal{W}$, resembling a grid structure. This method proves advantageous for covering boundary areas, which can be hard to sample through random methods.

4. **Direct to Goal**. If o_i can be directly placed at its goal, this pose will be prioritized over the above three samplings in the search process.

Once the place/final poses have been sampled, a $\mathcal{A}_{pp}$ sample is obtained. However, $\mathcal{A}_{pt}$ requires an additional step - generating a trajectory from the current pose to the place pose for o_i. We employ RRT-connect [149] during the tree search and LazyRRT [150] for robot execution to produce such collision-free trajectories within a set time limit. An example of sampling the final pose for a push action is shown in Figure 7.2. The criteria for sampling are

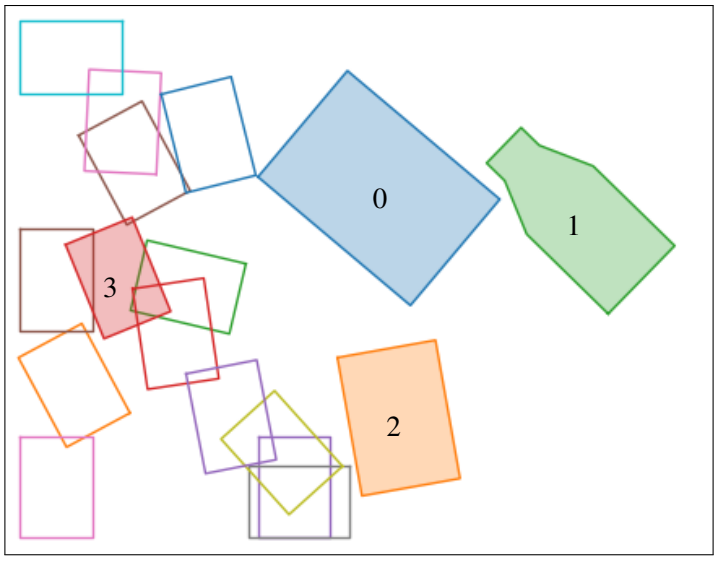


Figure 7.2: Consider action sampling for labeled 3 to be manipulated using push (there are a total of four objects). The absence of sampled actions in the right region is attributed to obstructions posed by objects 0, 1, and 2, preventing the movement of object 3 to that area.

crucial in solving the problem. Relying solely on standard uniform sampling often proves inefficient, particularly when sampling around boundaries and certain edge cases. While one might consider increasing the sample size to cover these edge cases, this inadvertently leads to many redundant actions that are time-consuming to process.

7.3.2 Hierarchical Best-First Search

The first algorithm we designed for REMP is a (greedy) best-first search algorithm called *hierarchical best-first search* (HBFS). HBFS is outlined in Algorithm 6 and operates according to the following sequence of steps:

- (Lines 3-4) When objects can be directly moved to their goal poses, an action cost is computed for each. The action yielding the smallest cost is then applied.
- (Lines 5-9) For each object o_i, HBFS identifies which objects occupy o_i's goal and attempts to displace these obstructing objects in the direction of their respective goals. If no actions are feasible in this direction, a random action is sampled. Again, the action associated with the smallest cost is selected and implemented.

- (Line 10) If the above steps do not yield a viable action, an action is randomly selected for execution.

The above three phases of HBFS may best be viewed as a three-level hierarchical search. To boost its performance and solution optimality, HBFS is implemented by leveraging multi-core capabilities of modern CPUs. This is realized by executing multiple HBFS in parallel and choosing the best action among the returned solutions.

Algorithm 6: Hierarchical Best-First Search (HBFS)

```
1  Function HBFS(s_s, s_g)
2      s ← s_s, A ← ∅
3      for o_i in s do A ← A ∪ {move o_i to its goal}
4      if |A| > 0 then return the lowest cost action from A
5      for o_i in s do
6          obs ← objects occupy the goal pose of o_i
7          for o_j in obs do
8              A ← A ∪ {move o_j towards its goal, otherwise at random}
9      if |A| > 0 then return the lowest cost action from A
10     return a randomly sampled action
```

7.3.3 Speeding up MCTS with Parallelism

Standard MCTS is more straightforward to implement, but it does not fully utilize multi-core processing capabilities of modern hardware. We introduce parallelism to the expansion and simulation stages of MCTS, leveraging tree parallelization techniques [141]. The application of parallelism allows for decoupling the select and expand stages from the simulation stage in an MCTS iteration. The decoupling allows multiple MCTS iterations to be carried out simultaneously, limited only by the number of CPU cores. The standard UCB formula used in MCTS is updated as Equation 6.2, where the idea of virtual loss [141] is applied by adding one extra virtual visit counts $\hat{N}$ that indicates a node has been selected but not yet simulated and backpropagated. Since the simulation and subsequent backpropagation stages are not yet completed, the tree search algorithm must be notified to update the Q and N of the node. This adjustment minimizes the likelihood of revisiting the node in the next

iteration, implementing a conservative approach in anticipation of a potentially poor reward. Once the simulation stage has concluded, Q is updated in backpropagation with returned reward from simulation result, N increments and $\hat{N}$ decrements.

7.3.4 Adapting MCTS for REMP

Given REMP's extremely large search space due to push actions' trajectory planning requirements, modifications are introduced to best apply MCTS to REMP.

Action Space Bias. The action space used in the simulation stage is a subset of that used in the expansion stage. Given that one iteration of the simulation stage constitutes a coarse estimation of action and state, reducing the number of actions leads to a larger number of total iterations.

Biased Simulation. A straightforward implementation of the simulation stage in MCTS is a random policy that indiscriminately selects an action for execution, ultimately obtaining a reward at the terminal state reached. In our implementation, we adopt a heuristic to guide the action selection in the simulation stage towards the ultimate goal of a given object. Specifically, with a probability of θ_{sim} (dynamically changed based on depth of the search), a random action is selected; otherwise, an attempt is made to select an action that will move an object towards its goal pose. This introduces a bias in the simulation stage, which offsets the drawback of limited iterations due to the time-consuming nature of motion planning and collision checking. We note that we did not adopt recent advancements in MCTS for long-horizon planning that injects a data-driven element to partially learn the reward, e.g., a neural network can be trained to evaluate the quality of an action-state pair [19], [79], [151].

Reward Shaping. We structure the reward to favor the goal state but without introducing undue bias. The reward function plays a critical role as it steers the tree search and is composed of three components. Firstly, if the task is accomplished, with all objects placed at their goal poses, a reward of R_g is awarded. Secondly, if an object o_i is located at its goal pose, a reward r_o is given. The cumulative reward from all objects, denoted as

R_o, is computed as $R_o = \sum_i r_{o_i}$. Lastly, the reward structure also takes into account the cost associated with the movement of objects. For the pick-n-place action ($\mathcal{A}_{pp}$), the cost corresponds to the Euclidean distance between the pick-and-place poses, with an additional fixed cost factored in. For the push action ($\mathcal{A}_{pt}$), the cost is determined by the Euclidean distance of the path, also supplemented by a fixed cost. Additionally, a base reward is computed $R_b = R_o(s_0)$ from initial state s_0, which is used to normalize the final reward during the search. For each iteration, a reward is returned by the simulation stage and is updated during the backpropagation stage.

$$R_i = \begin{cases} \max(0, R_g - \textit{cost} - R_b), & \text{if } s_i \text{ is the goal state} \\ \max(0, R_o(s_i) - \textit{cost} - R_b), & \text{otherwise} \end{cases}$$

Traditionally, Q retains the average reward values derived from simulation results, providing a robust estimation for action over millions of iterations. However, in our case, we aim to maintain the planning time within reasonable limits. Therefore, we introduced a priority queue to store simulation results, which serves as the Q value in the algorithm. For the purpose of calculation in Equation 6.2, we only preserve the top k rewards, similarly in the Chapter 5. There is a possibility that during a simulation, a subsequent action may transition the state to one with lower rewards, thereby negating the benefits of a preceding beneficial action within that simulation. To limit the search time, we choose to return the maximum reward encountered at the intermediate steps during the simulation instead of the final reward. Therefore, the returned reward from a simulation is given by

$$\max(\beta \cdot \max_{i \in m-1}(R_i), R_m) \cdot \gamma^m,$$

where $\beta \in (0, 1]$ is a scaling parameter and m is the total steps used in the simulation. γ is the discount factor, encouraging the problem to be solved in the early stage if possible.

Due to limited space, we omit the pseudo-code of the parallel MCTS algorithm but note

that all details for reproducing the algorithm have been fully specified. We call the resulting algorithm *parallel Monte Carlo tree search for multi-primitive rearrangement* or PMMR.

7.4 Experimental Evaluation

We evaluated HBFS and PMMR methods for REMP in simulated environments and on a real robot. Regardless of whether it is a simulation or a real robot experiment, both algorithms perform *percept-plan-act* loops until the task is solved or the budgeted time or maximum number of actions is exhausted. All experiments were conducted on an Intel i9-10900K (10 CPU cores) desktop PC and implemented in Python. As a note, limited testing shows that using Intel i9-13900K (24 CPU cores) reduces the planning time by roughly half, demonstrating the effectiveness and scalability of employing parallelism (code in Python).

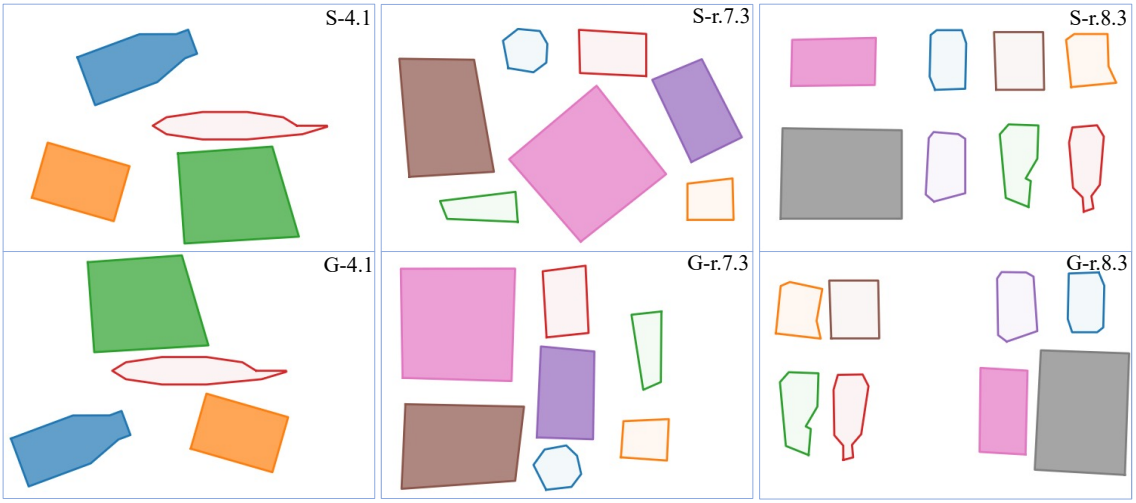


Figure 7.3: Example cases. The top row shows the start states and the bottom goal states. Lightly shaded objects can be pick-n-placed; heavily shaded objects must be manipulated using push. Cases 4.1, r.7.3, and r.8.3 are evaluated and presented in Figure 7.4. Objects are distinguished by color.

7.4.1 Simulation Studies

Simulations are conducted in PyBullet [152]. A real robot setup, consisting of a Universal Robot UR5e + OnRobot VGC-10 vacuum gripper, is replicated. The robot operates under

end-effector position control; the workspace measures $0.78 \times 0.52\mathrm{m}^2$. 25 feasible scenarios are created where all objects are confined within the workspace. Object sizes, shapes, and poses are randomly determined in each scenario (see Figure 7.3 for some examples). Cases that are trivial to solve (e.g., objects that happen to be mostly small) are filtered. The number of objects ranges from four to eight; five distinct cases are generated for each specified number of objects.

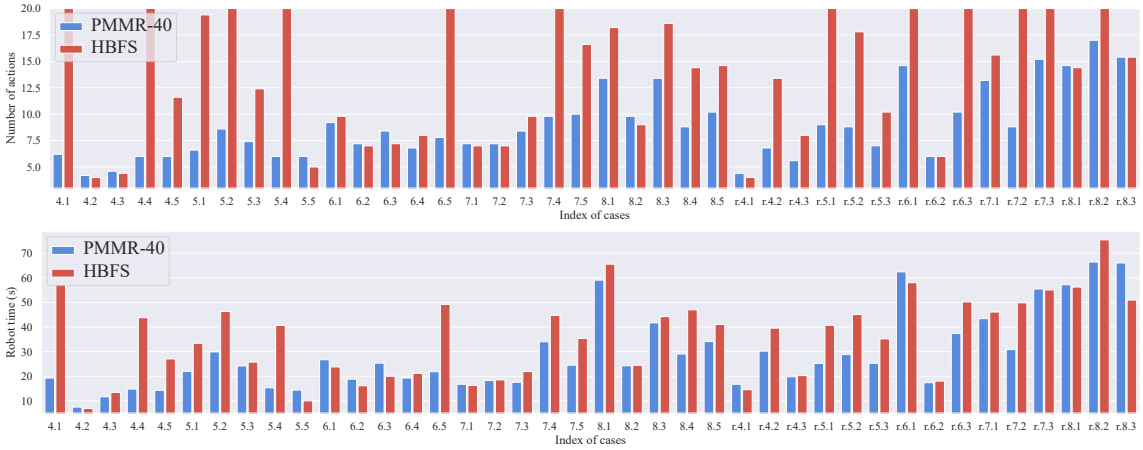


Figure 7.4: As an expanded illustration of Table 7.1, the upper plot lists the number of actions the robot executes to resolve individual cases. The lower plot lists the robot's execution times in solving the individual cases following the computed plan. For the labels on the horizontal axis, the first digit indicates the number of objects contained within each case, while the second digit represents the index of the cases. Cases beginning with the prefix 'r' are the ones that are constructed for and executed by the real-robot setup.

In evaluating PMMR, each *percept-plan-act* loop runs for a predetermined duration to identify the best next action until the problem is resolved or the maximum number of actions has been exhausted. If the latter occurs, it is treated as a failed case. We denote the corresponding PMMR method as PMMR-X, where X is the maximum number of seconds allowed in a single iteration of the loop. We settled on PMMR-40 as the main PMMR method used in the evaluation. In both simulation and real robot experiments, for PMMR-40, we keep the top $k = 100$ for Q value, $c = 1.5$ in Equation 6.2. The maximum depth of the MCTS tree D is based on the number of objects $\mathcal{N}$: $D = 2\mathcal{N} + 2$. θ_{sim} is based on the depth d of the node $\theta_{sim} = \max(-0.106 + 0.231d - 0.013d^2, 0.2)$. These numbers are

handpicked, representing that as the tree goes deeper, the probability of selecting a random action should be increased. $r_o = 0.7$ for object can be operated by $\mathcal{A}_{pp}$, the $R_g = 2r_0\mathcal{N}$. For an object that can be operated by $\mathcal{A}_{pt}$, the reward is given to $1.1r_o$. We set $\beta = 0.5$ and $\gamma = 0.9$ to scale the reward.

	Robot Time	Completion	Num. of Actions	Plan Time
PMMR-40	29.17s	98.00%	8.90	264.99s
HBFS	36.22s	54.50%	13.72	30.04s

Table 7.1: Summary of simulation results (25 cases) and real-robot experiments (15 cases) for HBFS and PMMR-40.

Individual experiment results for all cases are shown in Figure 7.4 (which also includes cases used for real-robot experiments, to be detailed in subsection 7.4.3). Detailed experiment results are presented in Table 7.1. Here, *robot time* refers to the cumulative time required for the robot to execute all actions, while *completion* refers to the success rate. The number of actions quantifies the execution of atomic actions, represented by $\mathcal{A}_{pp}, \mathcal{A}_{pt}$. The *plan time* is the total planning time. Each case underwent five independent trials.

Failure often happens because the case requires more than 15 actions to solve (even for humans). A failure may be due to sampled actions not containing a solution or the search not being deep enough. Sometimes, the algorithm can recover from an early bad choice, but not always since the number of iterations is capped. We observe that, while HBFS runs relatively fast in comparison to PMMR-40, it frequently fails (55% success vs. 98% for PMMR-40) and uses many more actions (13.7 vs. 8.9 for PMMR-40). Visually, as can be seen in the accompanying video, the actions generated by PMMR-40 are much more human-like than those by HBFS (the same holds for real robot experiments).

7.4.2 Ablation Studies

We investigated the impact of time budgets, the depth of tree search, and the base reward R_b on solving REMP. The time budget is critical; an extended search duration tends to yield better results. However, it is necessary to balance planning time and solution quality. An

insufficient search might sample highly suboptimal paths, leading to locally optimal actions. As depicted in Figure 7.5 and Table 7.2 shows the correlation between planning time and solution quality, leading us to select PMMR-40 for our main evaluation.

A shallow MCTS (PMMR-40 ($D = 3$), max MCTS tree depth of 3) was included specifically to compare with HBFS, which has three "depth levels" per iteration.

In terms of reward design, as detailed in section 7.3, we introduced a base reward R_b, which serves to normalize the reward to 0 at root. Without this adjustment, the tree search might commence with a non-zero reward signal, where a disadvantageous branch may still return a reward, causing the search to frequently explore such branches. By comparing PMMR-40 (no-R_b) in Table 7.2 with PMMR-40 in Table 7.1, we observe that the introduction of R_b aids the tree search.

	Robot Time	Completion	Num. of Actions	Plan Time
PMMR-10	35.60s	94.00%	10.51	103.61s
PMMR-20	32.46s	96.00%	9.86	155.68s
PMMR-60	27.93s	99.50%	8.53	358.44s
PMMR-40 ($D = 3$)	57.83s	32.50%	16.62	641.76s
PMMR-40 (no-R_b)	32.92s	94.00%	9.83	270.65s

Table 7.2: Ablation study results (averaged over 40 cases), for comparison with Table 7.1.

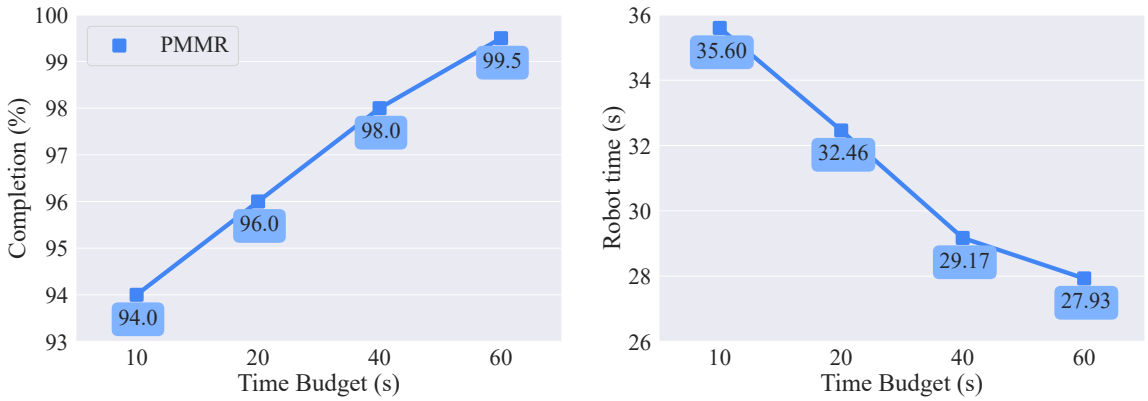


Figure 7.5: PMMR is evaluated with different time budgets. The reported values are averaged over 40 cases.

7.4.3 Real Robot Experiments

In simulation, we emulate the vacuum function by attaching the object to the end-effector using an extra link via a fixed joint. We employ two vacuum cups to provide sufficient suction power to ensure a robust connection in the real-world setup. The point of suction on the object is taken as its center, assuming this central area is flat. Similar to simulation studies, the number of objects ranges from four to eight, and three distinct cases are generated for each number of objects.

Figure 7.6: The full set of objects used in our real-robot experiments.

A RealSense D455 camera is affixed to the robot's wrist, capturing the scene from a top-down perspective, and an orthogonal view is rendered from the point cloud. We employ the Segment Anything Model [153] to extract masks of the objects present on the table. Subsequently, OpenCV [154] is applied to determine the contours and approximate them into polygons, which are then used for planning. An additional step determines each object's $SE(2)$ poses. For each case, the experiment is repeated at least three times.

Due to the small sim-to-real gap, we let the algorithm plan the entire sequence of

manipulation actions at the beginning, which generally works well. If no solution is found using 20 actions, we mark it as a failure; otherwise, the robot executes the planned actions. Robot time is not recorded for failure cases, hence its absence in Table 7.3. For completeness, in cases where both PMMR and HBFS succeed at least once, HBFS averages 15.15 actions and PMMR 9.56, with robot (execution) times of 96.62 seconds and 95.75 seconds, respectively (but note that HBFS fails much more frequently). Results from individual benchmarks across 15 cases are presented in Figure 7.7. Each case was subjected to three independent trials. In real-robot experiments, the cases were intentionally designed to be challenging. A greedy action may exacerbate the problem, making it even more difficult to resolve. Consequently, the completion rate of HBFS is significantly reduced.

	Robot Time	Completion	Num. of Actions	Plan Time
PMMR-40	95.75s*	96.44%	9.56	292.02s
HBFS	96.62s*	38.33%	15.15	29.05s
PMMR-40 (Sim)	–	94.67%	10.44	306.41s
HBFS (Sim)	–	45.33%	14.99	22.18s

Table 7.3: Experiment results of real robot trials across 15 cases, with time budgets constrained to a maximum of 40 seconds for a single MCTS run. The robot time is only considered in cases where both methods succeed at least once. Additionally, benchmarks from simulations covering 15 cases are included for sim-to-real gap comparisons. The robot time for PMMR-40 and HBFS, denoted with an asterisk, is recorded only for successful cases.

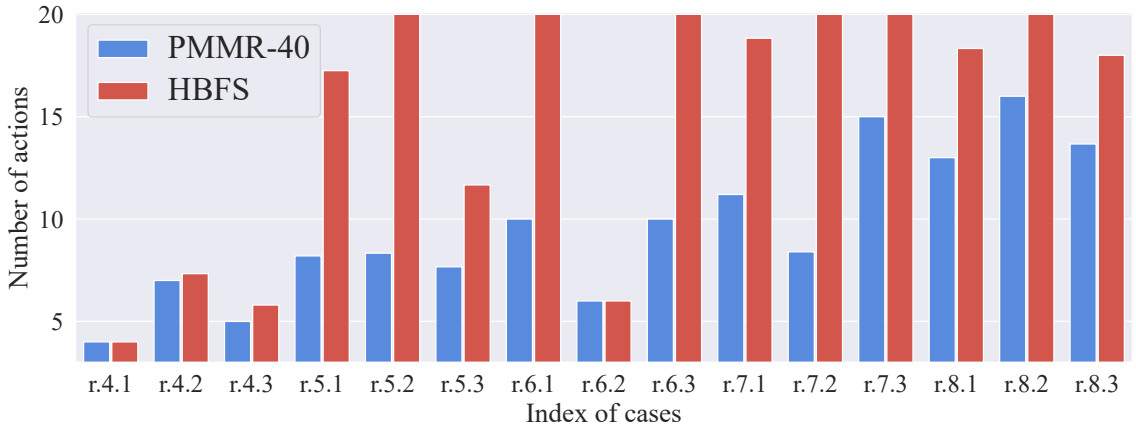


Figure 7.7: As an expanded illustration of Table 7.3, this plot illustrates the number of actions the robot executes to resolve individual cases.

7.5 Summary

We make the observation that humans frequently solve manipulation challenges using multiple types of manipulation actions. In contrast, there has been relatively limited research tackling planning high-quality resolutions for long-horizon manipulation tasks exploring the synergy of multiple manipulation actions. Inspired by how humans solve everyday manipulation tasks, in this paper, we proposed and studied the *Rearrangement with Multiple Manipulation Primitives* (REMP) problem. To optimally solve REMP, we developed two effective methods, HBFS and PMMR. PMMR is especially adept at solving difficult REMP instances with high success rates and producing high-quality solution sequences, capabilities confirmed through thorough simulation and real-robot experiments that included full percept-plan-act loops.

CHAPTER 8
CONCLUSION

In this dissertation, we have explored a spectrum of approaches for robotic manipulation in cluttered and long-horizon scenarios, with an emphasis on efficiently combining learned predictive models, search-based planning, and parallelization techniques. A central theme across the presented work is the pursuit of accurate interaction prediction and high-quality planning under uncertainty, grounded in both simulation and real-world experimentation.

With the introduction of Deep Interaction Prediction Network (DIPN) for push-and-grasp challenges, we established the importance of generating clear and reliable intermediate predictions that can be effectively used by downstream networks such as the Grasp Network (GN). This integrated method (DIPN+GN) exhibits strong generalization capabilities, demonstrates excellent sample efficiency, and outperforms prior state-of-the-art learning-based approaches. Notably, the methods train in a self-supervised manner, require no manual labeling or human input, and are robust to variations in object size, shape, color, and friction.

Building on these learned predictive models, the Visual Foresight Trees (VFT) framework introduces a synergy between DIPN and Monte Carlo Tree Search (MCTS) for long-horizon planning in object retrieval tasks from dense clutter. VFT has been shown to generate high-quality multi-step plans, though its performance is constrained by the substantial computational overhead. This drawback can be mitigated by parallelization—an approach that has been partially addressed by developing parallel MCTS strategies and will be essential for achieving real-time performance in robotic applications.

In parallel, we explored techniques for model-based simulation, including MORE (a simulator-driven approach), which uses a search-and-learn philosophy. Despite showing promising results, MORE and similar planning algorithms require either explicit object models or learned surrogates to simulate push outcomes. These requirements impose

potential limitations on generalization to novel objects and introduce additional uncertainties when learned components are used in place of a physics engine.

To address the computational bottleneck inherent in MCTS for robotic tasks, we introduced PMBS, a novel parallel MCTS method that leverages GPU-enabled batched simulations, achieving an over $30\times$ speedup relative to a strong serial MCTS baseline. Real-world robot experiments confirm that PMBS transfers effectively from simulation to physical systems, enabling near real-time performance in complex long-horizon planning.

Finally, we proposed the rearrangement with multiple manipulation primitives (REMP) framework, inspired by how humans tackle daily manipulation tasks using diverse actions. Our methods, HBFS and PMMR, facilitate planning that encompasses multiple action types (e.g., pushing, grasping) and achieve high success rates with high-quality solution sequences in both simulation and real-robot trials.

Across all these methods, a unifying message is the feasibility and effectiveness of integrating learning, search, and parallelization to solve challenging robotic manipulation tasks. Nonetheless, data efficiency, robustness to model inaccuracies, and computational scalability remain ongoing concerns that warrant further attention.

In future research, we will (1) extend the range of manipulation actions to include non-horizontal pushes and non-vertical grasps (arbitrary 6D end-effector poses), enabling more versatile rearrangement; (2) further optimize parallelization to achieve near real-time planning, focusing on multi-threaded or GPU-accelerated strategies; and (3) pursue integrated learning-based frameworks for rollout policies and reward estimation, aiming to reduce reliance on explicit simulation and improve overall efficiency and robustness. These directions promise to enhance the adaptability and performance of robotic systems in increasingly complex, real-world scenarios.

Appendices

APPENDIX A
CHAPTER 6 - PMBS SUPPLEMENTARY

A.0.1 Grasp Classifier Implementation Details

For our implementation of the grasp classifier (GC), we used Isaac Gym to collect grasp training data. Random objects are first sampled on the workspace, and then we discretize the workspace into a grid, where each point is the (x, y) of a grasp action a^g. We also discretize rotation into K angles uniformly. All robots in simulator will pick one grasp action and check the distance between two fingers as a signal of successful grasping. For each depth image, it is associated with hundreds of grasps. If the target can be grasped in at least n attempts, then the label is 1, and 0 otherwise (n = 5). We used two days of generating 20000 training data (a depth image focus centered on the target object and a label of can it be grasped) without human annotation. It is evaluated on test data with 93.45% accuracy if the R_c^* equals 0.7. The batch size is 256, learning rate is 0.1, epochs is 90, momentum is 0.9. We have successfully reduced the grasp evaluation time from 0.26 to 0.003 per image, making the parallel MCTS possible.

A.0.2 Grasp Network Implementation Details

For deciding whether to perform further push actions or to make an attempt to retrieve the target object, we resort to a *grasp network* (GN) that is fast and amenable to parallelization. GN is based on *fully convolutional networks* (FCNs) [12], [27] and customized to estimate the grasp probability for the target object [16], [19]. It takes an RGB-D image o_t as input and outputs dense pixel-wise values $P(o_t) \in [0, 1]^{H \times W \times K}$. H and W are the height and width of the o_t. To account the gripper orientation, we discretize θ of a^g into $K = 16$ angles, in multiples of (22.5°), so o_t has been rotated K times. GN presented in [12] is trained to estimate the grasp success rate for all objects. Further, binary mask of target object M is

imposed using Mask R-CNN [101], to truncate the values as $P_m(o_t) = P(o_t) \cap M(o_t)$. In proposed system, we only interested the highest grasp probability. If $\max_{h,w,k} P_m(o_t)$ is greater than the preset threshold P^* (it is 0.75 in our case), then, the robot should grasp at location h, w with k as orientation of gripper. The backbone of GN is ResNet-18 FPN [102], [110], with convolution layers and bilinear-upsampling layers as described in [16], [19]. We used the pre-trained Grasp Network from [19]. The Grasp Network is a plug-in component for the system, can be replaced by other advanced methods as if a grasp probability and grasp action can be provided.

A.0.3 Real-to-Sim-to-Real Comparison

We solve the task using a physics simulator. While there exists a real-to-sim and sim-to-real gaps, it is sufficiently small for this type of task, even when considering pose estimation errors that affect object localization in both the real and simulated environments.

From the Figure A.1 and Figure A.2, in the first row and first column, we capture an image of the real-world scene, perform pose estimation, and reconstruct the scene in the simulator (first column, second row). Planning is conducted within the simulator, where the estimated push action and its resulting state are shown in the first-row images. This process is iteratively repeated until the target object is successfully grasped. The discrepancy between the third-row images and the first-row images in the next column illustrates the difference between the planned/estimated results and the actual execution results of the real robot.

The simulator is capable of providing highly accurate physics simulations. However, in some cases, it may yield non-accurate yet reasonable physics approximations, which are still useful for task execution.

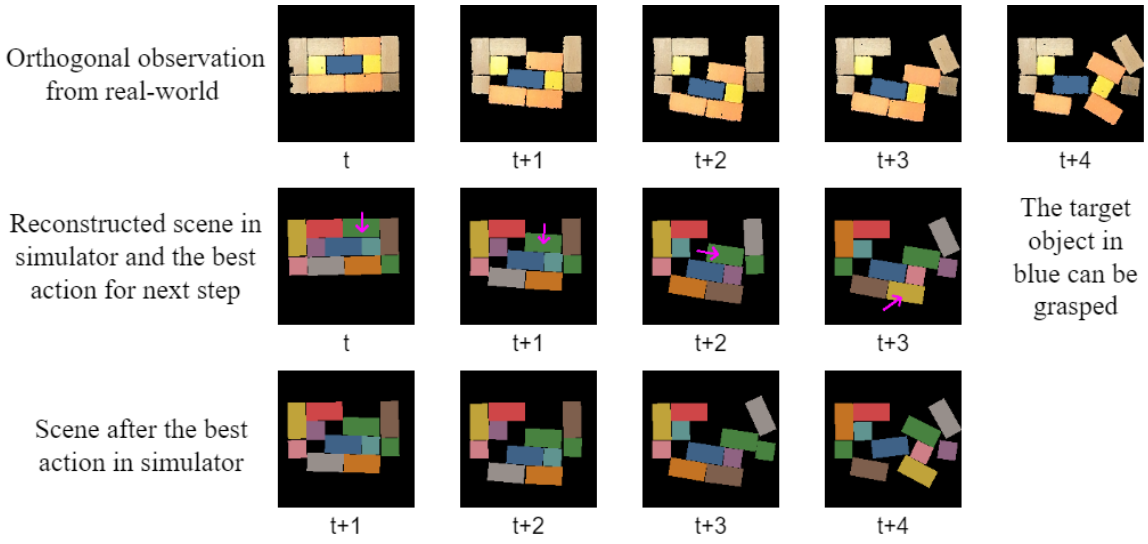


Figure A.1: Case study one of real to sim to real gap. Simulator provides accurate physics simulations.

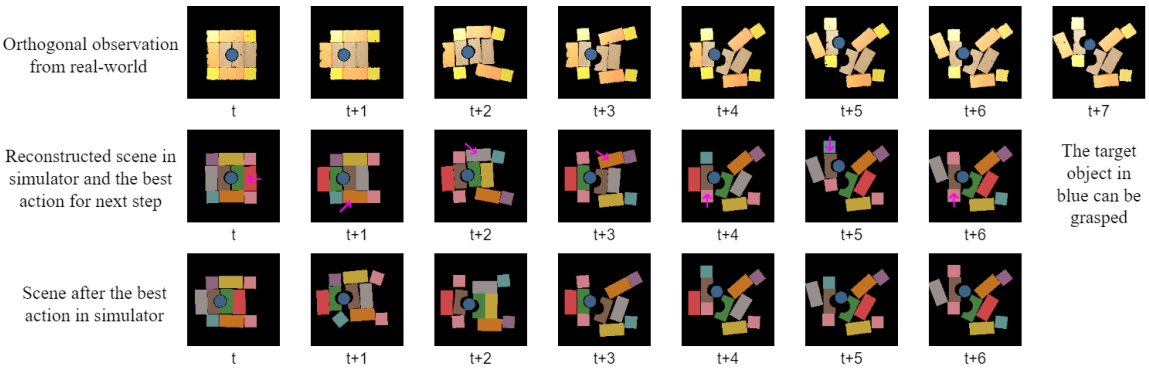


Figure A.2: Case study two of real to sim to real gap. Simulator provide non-accurate but reasonable physics simulations.

APPENDIX B

CHAPTER 7 - REMP SUPPLEMENTARY

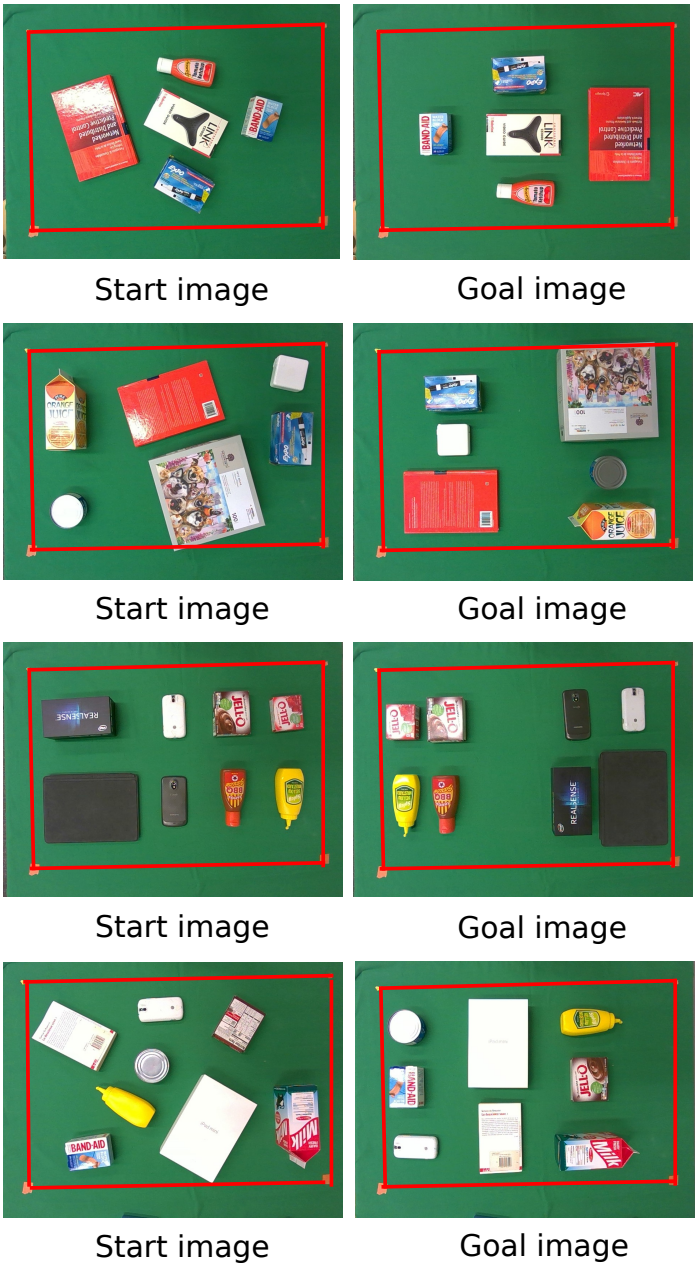


Figure B.1: Some cases in real world setup.

ACKNOWLEDGMENT OF PREVIOUS PUBLICATIONS

P1 **Baichuan Huang**, Shuai D. Han, Abdeslam Boularias, and Jingjin Yu. "Dipn: Deep interaction prediction network with application to clutter removal." In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4694–4701, 2021. IEEE.

P2 **Baichuan Huang**, Shuai D. Han, Jingjin Yu, and Abdeslam Boularias. "Visual foresight trees for object retrieval from clutter with nonprehensile rearrangement." *IEEE Robotics and Automation Letters*, vol. 7, no. 1, pp. 231–238, 2021. IEEE.

P3 **Baichuan Huang**, Teng Guo, Abdeslam Boularias, and Jingjin Yu. "Interleaving monte carlo tree search and self-supervised learning for object retrieval in clutter." In *2022 International Conference on Robotics and Automation (ICRA)*, pp. 625–632, 2022. IEEE.

P4 **Baichuan Huang**, Abdeslam Boularias, and Jingjin Yu. "Parallel monte carlo tree search with batched rigid-body simulations for speeding up long-horizon episodic robot planning." In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1153–1160, 2022. IEEE.

P5 **Baichuan Huang**, Xujia Zhang, and Jingjin Yu. "Toward optimal tabletop rearrangement with multiple manipulation primitives." In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10860–10866, 2024. IEEE.

P6 **Baichuan Huang**, Jingjin Yu, and Siddarth Jain. "EARL: Eye-on-hand reinforcement learner for dynamic grasping with active pose estimation." In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2963–2970, 2023.

IEEE. This work, while part of the author's PhD research efforts, is not discussed in detail within this dissertation.

REFERENCES

[1] M. T. Mason, "Toward robotic manipulation," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 1, no. 1, pp. 1–28, 2018.

[2] K. Azadeh, R. De Koster, and D. Roy, "Robotized and automated warehouse systems: Review and recent developments," *Transportation Science*, vol. 53, no. 4, pp. 917–945, 2019.

[3] R. H. Taylor, A. Menciassi, G. Fichtinger, P. Fiorini, and P. Dario, "Medical robotics and computer-integrated surgery," *Springer handbook of robotics*, pp. 1657–1684, 2016.

[4] A. Ajoudani, A. M. Zanchettin, S. Ivaldi, A. Albu-Schäffer, K. Kosuge, and O. Khatib, "Progress and prospects of the human–robot collaboration," *Autonomous robots*, vol. 42, pp. 957–975, 2018.

[5] J. S. Jennings, G. Whelan, and W. F. Evans, "Cooperative search and rescue with a team of mobile robots," in *1997 8th International Conference on Advanced Robotics. Proceedings. ICAR'97*, IEEE, 1997, pp. 193–200.

[6] J. Mahler, M. Matl, X. Liu, A. Li, D. Gealy, and K. Goldberg, "Dex-net 3.0: Computing robust vacuum suction grasp targets in point clouds using a new analytic model and deep learning," in *2018 IEEE International Conference on robotics and automation (ICRA)*, IEEE, 2018, pp. 5620–5627.

[7] T. Boroushaki, L. Dodds, N. Naeem, and F. Adib, "Fusebot: Rf-visual mechanical search," *Robotics: Science and Systems 2022*, 2022.

[8] A. Krontiris and K. E. Bekris, "Dealing with difficult instances of object rearrangement.," in *Robotics: Science and Systems*, vol. 1123, 2015.

[9] N. Marturi, M. Kopicki, A. Rastegarpanah, *et al.*, "Dynamic grasp and trajectory planning for moving objects," *Autonomous Robots*, vol. 43, pp. 1241–1256, 2019.

[10] O. M. Andrychowicz, B. Baker, M. Chociej, *et al.*, "Learning dexterous in-hand manipulation," *The International Journal of Robotics Research*, vol. 39, no. 1, pp. 3–20, 2020.

[11] A. Rodriguez, M. T. Mason, and S. Ferry, "From caging to grasping," *The International Journal of Robotics Research*, vol. 31, no. 7, pp. 886–900, 2012.

[12] B. Huang, S. D. Han, A. Boularias, and J. Yu, "Dipn: Deep interaction prediction network with application to clutter removal," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021, pp. 4694–4701.

[13] K. Gao, S. W. Feng, B. Huang, and J. Yu, "Minimizing running buffers for tabletop object rearrangement: Complexity, fast algorithms, and applications," *The International Journal of Robotics Research*, vol. 42, no. 10, pp. 755–776, 2023.

[14] B. Huang, X. Zhang, and J. Yu, "Toward optimal tabletop rearrangement with multiple manipulation primitives," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2024, pp. 10 860–10 866.

[15] H. Chang, K. Gao, K. Boyalakuntla, *et al.*, "Lgmcts: Language-guided monte-carlo tree search for executable semantic object rearrangement," in *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2024, pp. 13 607–13 612.

[16] B. Huang, S. D. Han, J. Yu, and A. Boularias, "Visual foresight trees for object retrieval from clutter with nonprehensile rearrangement," *IEEE Robotics and Automation Letters*, vol. 7, no. 1, pp. 231–238, 2021.

[17] S. D. Han, B. Huang, S. Ding, *et al.*, "Toward fully automated metal recycling using computer vision and non-prehensile manipulation," in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, IEEE, 2021, pp. 891–898.

[18] K. Gao, D. Lau, B. Huang, K. E. Bekris, and J. Yu, "Fast high-quality tabletop rearrangement in bounded workspace," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 1961–1967.

[19] B. Huang, T. Guo, A. Boularias, and J. Yu, "Interleaving monte carlo tree search and self-supervised learning for object retrieval in clutter," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 625–632.

[20] Y. Zhao, B. Huang, J. Yu, and Q. Zhu, "Stackelberg strategic guidance for heterogeneous robots collaboration," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 4922–4928.

[21] B. Huang, A. Boularias, and J. Yu, "Parallel monte carlo tree search with batched rigid-body simulations for speeding up long-horizon episodic robot planning," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2022, pp. 1153–1160.

[22] B. Huang, J. Yu, and S. Jain, "Earl: Eye-on-hand reinforcement learner for dynamic grasping with active pose estimation," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2023, pp. 2963–2970.

[23] X. Zhang, S. Jain, B. Huang, M. Tomizuka, and D. Romeres, "Learning generalizable pivoting skills," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2023, pp. 5865–5871.

[24] A. Bicchi and V. Kumar, "Robotic grasping and contact: A review," in *Proceedings 2000 ICRA. Millennium conference. IEEE international conference on robotics and automation. Symposia proceedings (Cat. No. 00CH37065)*, IEEE, vol. 1, 2000, pp. 348–353.

[25] M. Bauza and A. Rodriguez, "A probabilistic data-driven model for planar pushing," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 3008–3015.

[26] R. Shome, W. N. Tang, C. Song, *et al.*, "Towards robust product packing with a minimalistic end-effector," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 9007–9013.

[27] A. Zeng, S. Song, S. Welker, J. Lee, A. Rodriguez, and T. Funkhouser, "Learning synergies between pushing and grasping with self-supervised deep reinforcement learning," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2018, pp. 4238–4245.

[28] K. Hang, A. S. Morgan, and A. M. Dollar, "Pre-grasp sliding manipulation of thin objects using soft, compliant, or underactuated hands," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 662–669, 2019.

[29] M. R. Dogar, M. C. Koval, A. Tallavajhula, and S. S. Srinivasa, "Object search by manipulation," *Autonomous Robots*, vol. 36, pp. 153–167, 2014.

[30] Y. Hou, Z. Jia, A. M. Johnson, and M. T. Mason, "Robust planar dynamic pivoting by regulating inertial and grip forces," in *Algorithmic Foundations of Robotics XII: Proceedings of the Twelfth Workshop on the Algorithmic Foundations of Robotics*, Springer, 2020, pp. 464–479.

[31] N. Doshi, O. Taylor, and A. Rodriguez, "Manipulation of unknown objects via contact configuration regulation," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 2693–2699.

[32] J. Bohg, A. Morales, T. Asfour, and D. Kragic, "Data-driven grasp synthesis—a survey," *IEEE Transactions on robotics*, vol. 30, no. 2, pp. 289–309, 2013.

[33] R. Detry, C. H. Ek, M. Madry, and D. Kragic, "Learning a dictionary of prototypical grasp-predicting parts from grasping experience," in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 601–608.

[34] I. Lenz, H. Lee, and A. Saxena, "Deep learning for detecting robotic grasps," *The International Journal of Robotics Research*, vol. 34, no. 4-5, pp. 705–724, 2015.

[35] D. Kappler, J. Bohg, and S. Schaal, "Leveraging big data for grasp planning," in *2015 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2015, pp. 4304–4311.

[36] X. Yan, J. Hsu, M. Khansari, *et al.*, "Learning 6-dof grasping interaction via deep 3d geometry-aware representations," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA 2018)*, 2018.

[37] A. Mousavian, C. Eppner, and D. Fox, "6-dof graspnet: Variational grasp generation for object manipulation," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 2901–2910.

[38] H. Liang, X. Ma, S. Li, *et al.*, "Pointnetgpd: Detecting grasp configurations from point sets," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 3629–3635.

[39] A. Ten Pas and R. Platt, "Using geometry to detect grasp poses in 3d point clouds," *Robotics Research: Volume 1*, pp. 307–324, 2018.

[40] L. Pinto and A. Gupta, "Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours," in *2016 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2016, pp. 3406–3413.

[41] J. Mahler and K. Goldberg, "Learning deep policies for robot bin picking by simulating robust grasping sequences," in *Conference on robot learning*, PMLR, 2017, pp. 515–524.

[42] H.-S. Fang, C. Wang, M. Gou, and C. Lu, "Graspnet-1billion: A large-scale benchmark for general object grasping," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 11 444–11 453.

[43] A. Boularias, J. Bagnell, and A. Stentz, "Efficient optimization for autonomous robotic manipulation of natural objects," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, 2014.

[44] B. Wen, W. Lian, K. Bekris, and S. Schaal, "Catgrasp: Learning category-level task-relevant grasping in clutter from simulation," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 6401–6408.

[45] J. Mahler, J. Liang, S. Niyaz, *et al.*, "Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics," *arXiv preprint arXiv:1703.09312*, 2017.

[46] J. Mahler, M. Matl, V. Satish, *et al.*, "Learning ambidextrous robot grasping policies," *Science Robotics*, vol. 4, no. 26, eaau4984, 2019.

[47] Y. Deng, X. Guo, Y. Wei, *et al.*, "Deep reinforcement learning for robotic pushing and picking in cluttered environment," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Ieee, 2019, pp. 619–626.

[48] K. Xu, H. Yu, Q. Lai, Y. Wang, and R. Xiong, "Efficient learning of goal-oriented push-grasping synergy in clutter," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 6337–6344, 2021.

[49] J. Stüber, C. Zito, and R. Stolkin, "Let's push things forward: A survey on robot pushing," *Frontiers in Robotics and AI*, vol. 7, p. 8, 2020.

[50] K. M. Lynch, "Estimating the friction parameters of pushed objects," in *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'93)*, IEEE, vol. 1, 1993, pp. 186–193.

[51] M. T. Mason, "Mechanics and planning of manipulator pushing operations," *The International Journal of Robotics Research*, vol. 5, no. 3, pp. 53–71, 1986.

[52] K. M. Lynch and M. T. Mason, "Stable pushing: Mechanics, controllability, and planning," *The international journal of robotics research*, vol. 15, no. 6, pp. 533–556, 1996.

[53] K. M. Lynch and M. T. Mason, "Dynamic nonprehensile manipulation: Controllability, planning, and experiments," *The International Journal of Robotics Research*, vol. 18, no. 1, pp. 64–92, 1999.

[54] S. Akella and M. T. Mason, "Posing polygonal objects in the plane by pushing," *The International Journal of Robotics Research*, vol. 17, no. 1, pp. 70–88, 1998.

[55] M. T. Mason, "On the scope of quasi-static pushing," in *International Symposium on Robotics Research, 1986*, 1986, pp. 229–233.

[56] T. Yoshikawa and M. Kurisu, "Indentification of the center of friction from pushing an object by a mobile robot," in *Proceedings IROS'91: IEEE/RSJ International Workshop on Intelligent Robots and Systems' 91*, IEEE, 1991, pp. 449–454.

[57] R. D. Howe and M. R. Cutkosky, "Practical force-motion models for sliding manipulation," *The International Journal of Robotics Research*, vol. 15, no. 6, pp. 557–572, 1996.

[58] J. Zhou, R. Paolini, J. A. Bagnell, and M. T. Mason, "A convex polynomial force-motion model for planar sliding: Identification and application," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2016, pp. 372–377.

[59] J. Zhou, M. T. Mason, R. Paolini, and D. Bagnell, "A convex polynomial model for planar sliding mechanics: Theory, application, and experimental validation," *The International Journal of Robotics Research*, vol. 37, no. 2-3, pp. 249–265, 2018.

[60] J. Zhou, Y. Hou, and M. T. Mason, "Pushing revisited: Differential flatness, trajectory planning, and stabilization," *The International Journal of Robotics Research*, vol. 38, no. 12-13, pp. 1477–1489, 2019.

[61] M. R. Dogar and S. S. Srinivasa, "A framework for push-grasping in clutter.," in *Robotics: Science and systems*, vol. 2, 2011.

[62] C. Finn, I. Goodfellow, and S. Levine, "Unsupervised learning for physical interaction through video prediction," in *Advances in neural information processing systems*, 2016, pp. 64–72.

[63] A. Byravan and D. Fox, "Se3-nets: Learning rigid body motion using deep neural networks," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 173–180.

[64] N. Watters, D. Zoran, T. Weber, P. Battaglia, R. Pascanu, and A. Tacchetti, "Visual interaction networks: Learning a physics simulator from video," in *Advances in neural information processing systems*, 2017, pp. 4539–4547.

[65] L. Sergey, N. Wagener, and P. Abbeel, "Learning contact-rich manipulation skills with guided policy search," in *Proceedings of the 2015 IEEE International Conference on Robotics and Automation (ICRA), Seattle, WA, USA*, 2015, pp. 26–30.

[66] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *Journal of Machine Learning Research*, vol. 17, no. 39, pp. 1–40, 2016.

[67] C. Finn and S. Levine, "Deep visual foresight for planning robot motion," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 2786–2793.

[68] A. Ghadirzadeh, A. Maki, D. Kragic, and M. Björkman, "Deep predictive policy training using reinforcement learning," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 2351–2358.

[69] M. R. Dogar and S. S. Srinivasa, "Push-grasping with dexterous hands: Mechanics and a method," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2010, pp. 2123–2130.

[70] M. R. Dogar, K. Hsiao, M. T. Ciocarlie, and S. S. Srinivasa, "Physics-based grasp planning through clutter.," in *Robotics: Science and systems*, vol. 8, 2012, pp. 57–64.

[71] J. E. King, M. Klingensmith, C. M. Dellin, *et al.*, "Pregrasp manipulation as trajectory optimization.," in *Robotics: Science and Systems*, Berlin, 2013.

[72] L. Chang, J. R. Smith, and D. Fox, "Interactive singulation of objects from a pile," in *2012 IEEE International Conference on Robotics and Automation*, IEEE, 2012, pp. 3875–3882.

[73] A. Eitel, N. Hauff, and W. Burgard, "Learning to singulate objects using a push proposal network," in *Robotics Research: The 18th International Symposium ISRR*, Springer, 2020, pp. 405–419.

[74] M. Danielczuk, J. Mahler, C. Correa, and K. Goldberg, "Linear push policies to increase grasp access for robot bin picking," in *2018 IEEE 14th international conference on automation science and engineering (CASE)*, IEEE, 2018, pp. 1249–1256.

[75] B. Tang, M. Corsaro, G. Konidaris, S. Nikolaidis, and S. Tellex, "Learning collaborative pushing and grasping policies in dense clutter," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021, pp. 6177–6184.

[76] Y. Xiao, S. Katt, A. ten Pas, S. Chen, and C. Amato, "Online planning for target object search in clutter under partial observability," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 8241–8247.

[77] M. Danielczuk, A. Kurenkov, A. Balakrishna, *et al.*, "Mechanical search: Multi-step retrieval of a target object occluded by clutter," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 1614–1621.

[78] A. Kurenkov, J. Taglic, R. Kulkarni, *et al.*, "Visuomotor mechanical search: Learning to retrieve target objects in clutter," in *IEEE/RSJ Int. Conference. on Intelligent Robots and Systems (IROS)*, 2020.

[79] H. Song, J. A. Haustein, W. Yuan, *et al.*, "Multi-object rearrangement with monte carlo tree search: A case study on planar nonprehensile sorting," in *2020 IEEE/RSJ*

international conference on intelligent robots and systems (IROS), IEEE, 2020, pp. 9433–9440.

[80] S. D. Han, N. M. Stiffler, A. Krontiris, K. E. Bekris, and J. Yu, "High-quality tabletop rearrangement with overhand grasps: Hardness results and fast methods," in *Robotics: Sciences and Systems*, 2017.

[81] S. D. Han, N. M. Stiffler, A. Krontiris, K. E. Bekris, and J. Yu, "Complexity results and fast methods for optimal tabletop rearrangement with overhand grasps," *The International Journal of Robotics Research*, vol. 37, no. 13-14, pp. 1775–1795, 2018.

[82] K. Gao, S. W. Feng, and J. Yu, "On minimizing the number of running buffers for tabletop rearrangement," in *Robotics: Sciences and Systems*, 2021.

[83] J. Yu, "Rearrangement on lattices with swaps: Optimality structures and efficient algorithms," in *Robotics: Sciences and Systems*, 2021.

[84] J. Yu, "Rearrangement on lattices with pick-n-swaps: Optimality structures and efficient algorithms," *The International Journal of Robotics Research*, vol. 42, no. 10, pp. 957–973, 2023.

[85] K. Gao, S. W. Feng, B. Huang, and J. Yu, "Minimizing running buffers for tabletop object rearrangement: Complexity, fast algorithms, and applications," *The International Journal of Robotics Research*, vol. 42, no. 10, pp. 755–776, 2023.

[86] B. Tang and G. S. Sukhatme, "Selective object rearrangement in clutter," in *Conference on Robot Learning*, PMLR, 2023, pp. 1001–1010.

[87] J. Ahn, C. Kim, and C. Nam, "Coordination of two robotic manipulators for object retrieval in clutter," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 1039–1045.

[88] M. Moll, L. Kavraki, J. Rosell, *et al.*, "Randomized physics-based motion planning for grasping in cluttered and uncertain environments," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 712–719, 2017.

[89] R. Wang, Y. Miao, and K. E. Bekris, "Efficient and high-quality prehensile rearrangement in cluttered and confined spaces," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 1968–1975.

[90] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *2011 IEEE International Conference on Robotics and Automation*, IEEE, 2011, pp. 1470–1477.

[91] 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," in *2014 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2014, pp. 639–646.

[92] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning.," in *IJCAI*, 2015, pp. 1930–1936.

[93] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "Incremental task and motion planning: A constraint-based approach.," in *Robotics: Science and systems*, Ann Arbor, MI, USA, vol. 12, 2016, p. 00 052.

[94] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *Proceedings of the international conference on automated planning and scheduling*, vol. 30, 2020, pp. 440–448.

[95] T. Migimatsu and J. Bohg, "Object-centric task and motion planning in dynamic environments," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 844–851, 2020.

[96] R. Chitnis, D. Hadfield-Menell, A. Gupta, *et al.*, "Guided search for task and motion plans using learned heuristics," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2016, pp. 447–454.

[97] B. Kim, Z. Wang, L. P. Kaelbling, and T. Lozano-Pérez, "Learning to guide task and motion planning using score-space representation," *The International Journal of Robotics Research*, vol. 38, no. 7, pp. 793–812, 2019.

[98] D. Driess, J.-S. Ha, and M. Toussaint, "Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image," *arXiv preprint arXiv:2006.05398*, 2020.

[99] A. M. Wells, N. T. Dantam, A. Shrivastava, and L. E. Kavraki, "Learning feasibility for task and motion planning in tabletop environments," *IEEE robotics and automation letters*, vol. 4(2), pp. 1255–1262, 2019.

[100] N. Dengler, D. Großklaus, and M. Bennewitz, "Learning goal-oriented non-prehensile pushing in cluttered scenes," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2022, pp. 1116–1122.

[101] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

[102] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature pyramid networks for object detection," *CoRR*, vol. abs/1612.03144, 2016. arXiv: 1612.03144.

[103] L. Yen-Chen, A. Zeng, S. Song, P. Isola, and T.-Y. Lin, "Learning to see before learning to act: Visual pre-training for manipulation," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020, pp. 7286–7293.

[104] E. Rohmer, S. P. Singh, and M. Freese, "V-rep: A versatile and scalable robot simulation framework," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2013, pp. 1321–1326.

[105] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, "Dream to control: Learning behaviors by latent imagination," in *International Conference on Learning Representations*, 2020.

[106] F. Ebert, C. Finn, S. Dasari, A. Xie, A. X. Lee, and S. Levine, "Visual foresight: Model-based deep reinforcement learning for vision-based robotic control," *CoRR*, vol. abs/1812.00568, 2018. arXiv: 1812.00568.

[107] Muhayyuddin, M. Moll, L. Kavraki, and J. Rosell, "Randomized physics-based motion planning for grasping in cluttered and uncertain environments," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 712–719, Apr. 2018.

[108] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[109] M. Andrychowicz, F. Wolski, A. Ray, *et al.*, "Hindsight experience replay," *Advances in neural information processing systems*, vol. 30, 2017.

[110] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[111] C. B. Browne, E. Powley, D. Whitehouse, *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[112] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.

[113] M. Andrychowicz, F. Wolski, A. Ray, *et al.*, "Hindsight experience replay," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, 2017.

[114] E. Coumans and Y. Bai, *Pybullet, a python module for physics simulation for games, robotics and machine learning*, http://pybullet.org, 2016–2019.

[115] J. Schrittwieser, I. Antonoglou, T. Hubert, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[116] D. Kahneman, *Thinking, fast and slow*. Macmillan, 2011.

[117] Y. Bengio, "The consciousness prior," *arXiv preprint arXiv:1709.08568*, 2017.

[118] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[119] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[120] H. Song, J. A. Haustein, W. Yuan, *et al.*, "Multi-object rearrangement with monte carlo tree search: A case study on planar nonprehensile sorting," *CoRR*, vol. abs/1912.07024, 2019. arXiv: 1912.07024.

[121] R. Boney, N. Di Palo, M. Berglund, *et al.*, "Regularizing trajectory optimization with denoising autoencoders," *Advances in Neural Information Processing Systems*, vol. 32, pp. 2859–2869, 2019.

[122] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*, Springer, 2006, pp. 72–83.

[123] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[124] J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, *et al.*, "Combining q-learning and search with amortized value estimates," in *International Conference on Learning Representations ICLR*, 2019.

[125] E. Coumans and Y. Bai, *Pybullet, a python module for physics simulation for games, robotics and machine learning*, http://pybullet.org, 2016–2021.

[126] B. Wen, C. Mitash, B. Ren, and K. E. Bekris, "Se (3)-tracknet: Data-driven 6d pose tracking by calibrating image residuals in synthetic domains," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2020, pp. 10 367–10 373.

[127] B. Wen and K. Bekris, "Bundletrack: 6d pose tracking for novel objects without instance or category-level 3d models," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2021, pp. 8067–8074.

[128] C. Mitash, B. Wen, K. Bekris, and A. Boularias, "Scene-level pose estimation for multiple instances of densely packed objects," in *Conference on Robot Learning*, PMLR, 2020, pp. 1133–1145.

[129] X. B. Peng, E. Coumans, T. Zhang, T.-W. E. Lee, J. Tan, and S. Levine, "Learning agile robotic locomotion skills by imitating animals," in *Robotics: Science and Systems*, Jul. 2020.

[130] J. Hwangbo, J. Lee, A. Dosovitskiy, *et al.*, "Learning agile and dynamic motor skills for legged robots," *Science Robotics*, vol. 4, no. 26, eaau5872, 2019.

[131] A. Kumar, Z. Fu, D. Pathak, and J. Malik, "RMA: Rapid Motor Adaptation for Legged Robots," in *Proceedings of Robotics: Science and Systems*, Virtual, Jul. 2021.

[132] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[133] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[134] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *2012 IEEE/RSJ international conference on intelligent robots and systems*, IEEE, 2012, pp. 5026–5033.

[135] V. Makoviychuk, L. Wawrzyniak, Y. Guo, *et al.*, "Isaac gym: High performance gpu-based physics simulation for robot learning," *arXiv preprint arXiv:2108.10470*, 2021.

[136] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem, *Brax - a differentiable physics engine for large scale rigid body simulation*, version 0.0.10, 2021.

[137] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[138] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[139] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*, Springer, 2006, pp. 282–293.

[140] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, PMLR, 2019, pp. 6105–6114.

[141] G. M.-B. Chaslot, M. H. Winands, and H. Herik, "Parallel monte-carlo tree search," in *International Conference on Computers and Games*, Springer, 2008, pp. 60–71.

[142] A. Liu, J. Chen, M. Yu, Y. Zhai, X. Zhou, and J. Liu, "Watch the unobserved: A simple approach to parallelizing monte carlo tree search," in *International Conference on Learning Representations*, 2020.

[143] X. Yang, T. Aasawat, and K. Yoshizoe, "Practical massively parallel monte-carlo tree search applied to molecular design," in *International Conference on Learning Representations*, 2021.

[144] K. P. Hawkins, "Analytic inverse kinematics for the universal robots ur-5/ur-10 arms," Georgia Institute of Technology, Tech. Rep., 2013.

[145] S. W. Feng, T. Guo, K. E. Bekris, and J. Yu, "Team rubot's experiences and lessons from the ariac," *Robotics and computer-integrated manufacturing*, vol. 70, p. 102 126, 2021.

[146] D. Silver, A. Huang, C. J. Maddison, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[147] Y. Labbé, S. Zagoruyko, I. Kalevatykh, *et al.*, "Monte-carlo tree search for efficient visually guided rearrangement planning," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3715–3722, 2020.

[148] K. Gao, J. Yu, T. S. Punjabi, and J. Yu, "Effectively rearranging heterogeneous objects on cluttered tabletops," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2023, pp. 2057–2064.

[149] J. Kuffner and S. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 2, 2000, 995–1001 vol.2.

[150] R. Bohlin and L. E. Kavraki, "A randomized approach to robot path planning based on lazy evaluation," *COMBINATORIAL OPTIMIZATION-DORDRECHT-*, vol. 9, no. 1, pp. 221–249, 2001.

[151] F. Bai, F. Meng, J. Liu, J. Wang, and M. Q.-H. Meng, "Hierarchical policy with deep-reinforcement learning for nonprehensile multiobject rearrangement," *Biomimetic Intelligence and Robotics*, vol. 2, no. 3, p. 100 047, 2022.

[152] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," 2016.

[153] A. Kirillov, E. Mintun, N. Ravi, *et al.*, "Segment anything," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2023, pp. 4015–4026.

[154] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.