AUTOMATED PLANNING DOMAIN INFERENCE FOR ROBOT TASK AND MOTION PLANNING

by

Jinbang Huang

A thesis submitted in conformity with the requirements
for the degree of Master of Science

University of Toronto Institute for Aerospace Studies
University of Toronto

© Copyright 2024 by Jinbang Huang

Automated Planning Domain Inference
for Robot Task and Motion Planning

Jinbang Huang
Master of Science

University of Toronto Institute for Aerospace Studies
University of Toronto
2024

Abstract

Robots excel at completing short-term tasks within structured environments but struggle with longer-horizon tasks in dynamic, unstructured settings, due to the limitations of current motion planning algorithms. Task and motion planning (TAMP) frameworks address this problem by integrating high-level task planning with low-level motion planning. However, existing TAMP methods rely heavily on the manual design of planning domains that specify the preconditions and effects of all high-level actions. This thesis proposes a novel method to automate planning domain inference, reducing the reliance on human design. Our approach incorporates a deep learning-based estimator that predicts the appropriate domain for a new task, and a search algorithm that refines this prediction. Our method is able to generate new domains from minimal demonstrations at test time, enabling robots to handle complex tasks more efficiently. We demonstrate that our approach achieves superior performance and generalization on a variety of tasks compared to behavior cloning baselines.

Acknowledgements

This thesis represents not only my academic endeavours but also a journey of personal growth and discovery, which would not have been possible without the help and support of many individuals. First and foremost, I would like to thank my supervisors, Prof. Jonathan Kelly and Prof. Florian Shkurti, for providing me with this opportunity. I want to thank you not only for the professional guidance but also for the valuable emotional support throughout this program. Your encouragement has been my anchor and motivation during the hardest moments, and your belief in me has never wavered, even when I doubted myself. I am also truly grateful to all my colleagues at the STARS laboratory and RVL group for their valuable feedback and help. I would especially like to express my appreciation to Dr. Miroslav Bogdanovic and Dr. Kourosh Darvish, who always generously shared their expertise throughout this project. I would like to thank my friends, who are always there, providing me with unwavering support, encouragement, and laughter. Lastly, I extend my deepest gratitude to my parents, Jixian Huang and Yan Li, for their unconditional love and support.

Contents

1 Introduction **1**
1.1 Reducing Reliance on Manual Domain Design 2
1.2 Contributions 3

2 Background **4**
2.1 Motion Planning 4
2.2 Task Planning 6
2.3 Task and Motion Planning 8
2.4 Planning Domains and Planning Languages 10
2.4.1 Standardized Planning Languages 10
2.4.2 Complete and Optimal Planning Domains 12
2.5 Graphs and Graph Neural Networks 14
2.5.1 Graph Neural Network Structures 14
2.5.2 Message Passing and Aggregation 16

3 Related Work **18**
3.1 Learning to Plan for Faster Task and Motion Planning 18
3.2 Learning Planning Domains 20
3.3 Research Gap 21

4 Methodology **24**
4.1 Problem Setting 25
4.2 Overview of Inference Framework 26
4.3 Training Predicate and Action Estimators 29
4.3.1 Estimator Design 29
4.3.2 Relevance Scoring 30
4.3.3 Dataset Generation 30
4.4 Precondition and Postcondition Generation 32
4.5 Domain Optimization by Generate-and-Test Search 33

4.6 Example of Domain Inference 35

5 Experiments **39**

5.1 Cube Manipulation 40

5.2 Everyday Tasks 42

5.3 A Puzzle-Solving Task 45

5.4 Composed Task 45

5.5 Training and Testing Dataset 46

5.6 Baseline 48

6 Results **50**

6.1 Planning Success Rate and Generalizability 50

6.2 Computational Cost 52

7 Conclusion **54**

7.1 Contributions 54

7.2 Future Work 55

A Table of Predicates and Actions **56**

Bibliography **58**

List of Tables

3.1 The pros and cons of the learning-based methods reviewed in Section 3.1. . . . 23

A.1 Predicates used in the experiments described in Chapter 5, along with explanations. 56

A.2 Actions used in the experiments described in Chapter 5, along with explanations. 57

List of Figures

2.1 Demonstration of RRT from [13]. . . . 6
2.2 Task and motion planning pipeline. . . . 9
2.3 Example of a 'Pick' action written in PDDL. . . . 12
2.4 Graphic explanation on how redundancy in planning domain affects planning performance. . . . 13
2.5 An example of graph structures. . . . 15

4.1 The overall framework of domain inference. . . . 27
4.2 Formulating predicates as node features. . . . 31
4.3 Formulating predicates as edge connections. . . . 31
4.4 Precondition and postcondition generation. . . . 34
4.5 Example of the domain optimization process of sorting objects. . . . 38

5.1 A graphical view of the cube stacking task. . . . 40
5.2 A graphical view of the cube unstacking task. . . . 41
5.3 A graphical view of sorting experiment. . . . 41
5.4 A graphical view of washing experiment. . . . 42
5.5 A graphical view of grilling experiment. . . . 43
5.6 A graphical view of the cooking experiment. . . . 44
5.7 A graphical view of table cleaning experiment. . . . 44
5.8 A graphical view of painting experiment. . . . 45
5.9 A graphical view of the Towers of Hanoi experiment. . . . 46
5.10 A graphical view of the cook-and-plate experiment. . . . 47
5.11 A graphical view of the unstack-and-cook experiment. . . . 47
5.12 A graphical view of labelling experiment. . . . 48

6.1 Success rates of basic planning tasks for increasing numbers of objects. . . 51
6.2 Success rates of composed planning tasks for increasing numbers of objects. 51
6.3 The number of queries to motion planners during domain optimization. . 52

Chapter 1

Introduction

Humans are able to effortlessly navigate through dynamic environments and to handle most daily chores, but robots struggle to do the same. Simple tasks for humans, such as manipulating typical objects in a kitchen to make a salad, for example, have proven to be very difficult for robots to perform. This difficulty highlights Moravec's paradox, which roughly postulates that what is simple for humans is hard for machines, and vice-versa. Robot autonomy that generalizes to a diverse set of environments requires efficient integration of complex real-time perception, decision-making, and path planning. The growing demand for capable robots has captured significant attention from both industry and academia, leading to rapid advances in automated robotic systems.

Motion planning is a fundamental robotic capability that enables a robot to create a collision-free path to move from a defined start pose to a goal pose, considering any environmental constraints. The motion planning problem is relatively easy to solve in static and structured environments, such as manufacturing plants and industrial packaging settings, enabling robots to navigate and avoid obstacles. These environments have minimal uncertainty and low-dimensional decision spaces, however, making the planning problem straightforward. Existing motion planning reaches its limits in complex and high-dimensional environments [1]–[5] that are frequently encountered in modern robot applications, such as collaborative robots for Industry 4.0 and household service robots [6]. These new scenarios require robots to automatically make decisions, generate complex plans, and adapt to changes in surroundings [7].

To accomplish complex, long-horizon tasks, task planning is necessary in addition to motion planning. Task planning aims to find a sequence of high-level actions to achieve a set of objectives while respecting constraints [8] introduced by intermediate sub-tasks. An assembly task, for example, may involve stepping through a sequence of sub-tasks, such as selecting tools, aligning pieces, and tightening fasteners. These sub-tasks, with

smaller decision spaces and more structured objectives at each step, make it much simpler to find a feasible motion plan. Therefore, task planning offers a way to break down a complex and lengthy planning problem into smaller, simpler sub-problems, which are manageable for motion planning algorithms.

Task and motion planning (TAMP) frameworks combine high-level task planners with low-level motion planners to solve complex planning problems. TAMP employs high-level task planners to discretize a complex planning task into a sequence of manageable sub-tasks for low-level motion planners. Task planners use logic-based reasoning to ensure that the sequence of sub-tasks is feasible and coherent, relying on rules and constraints to guide the decision-making process. A motion planner then generates a motion plan for each sub-task. Finally, the motion plans for each sub-task are sequenced to solve the full planning problem. If the motion planner cannot find a solution to a given sub-task, the task planner backtracks and attempts to identify another sequence of sub-tasks that might accomplish the full task.

1.1 Reducing Reliance on Manual Domain Design

While existing TAMP frameworks have successfully solved many complex, multi-step, long-horizon tasks in robotics [7], [9], task planners rely on human design. In traditional TAMP, a task planner requires a planning domain, that is, a set of human-designed planning rules that encompass the set of objects, actions, and constraints (preconditions and effects) of a particular planning problem. The task planner can only solve problems within the decision space of the planning domain. When faced with a new task, the task planner cannot adapt accordingly. However, human engineers are not able to foresee all possible situations, nor can they pre-design all domains in advance. The reliance on manual input is inefficient and risks introducing human errors, leading to failures and low efficiency.

The significant effort required to engineer a well-structured planning domain motivates us to explore automated solutions for generating planning domains. While directly designing a planning domain is challenging, creating a new planning domain based on existing ones is possible.

We aim to develop a system that learns the relationships between tasks and their respective planning domains from training data, enabling the automatic generation of a new planning domain based on just one or a few demonstrations, obviating the need for manual design of the domain. To do so, we compile each existing domain into structured data and use graph neural networks to extract transferable knowledge. Our method is designed to learn from small domains that are associated with simple plan-

ning tasks and then to generate larger planning domains for longer-horizon tasks, from a small number of additional demonstrations.

1.2 Contributions

The primary contribution of this thesis is a new domain inference method, which automates the generation of planning domains for new TAMP problems in a one-shot or few-shot manner based on demonstrations of continuous (low-level) state-action trajectories. Domain inference has two main components. The first component is a deep learning-based domain estimator that predicts the most likely planning domain for a new task based on knowledge gained from the demonstration dataset. The second component is a search algorithm that adds essential rules to or removes unnecessary rules from the planning domain. The search step ensures that planning is feasible (i.e., solvable) while also enabling fast planning. In summary, the contributions of this thesis are as follows.

- A deep learning-based algorithm to predict the most likely planning domain for a new task based on a small set of demonstrations.
- A search algorithm to refine the predicted planning domain, ensuring that planning is both feasible and efficient.
- A framework to integrate the prediction and search algorithms to efficiently generate the corresponding planning domain for a new task.

Chapter 2

Background

This chapter reviews traditional motion and task planning methods and related concepts and provides a foundation for understanding the thesis. In Section 2.1, we discuss the problem of motion planning and describe the primary approaches that appear in the literature. Section 2.2 reviews classic task planning methods. In Section 2.3, we consider task and motion planning algorithms that enable autonomous robots to effectively plan and execute complex multi-step tasks in the real world. Section 2.4 briefly details the major standardized languages for task planning and defines key concepts related to planning domains, where a *planning domain* encompasses the set of objects, actions, and constraints for a particular planning problem. Finally, in Section 2.5, we cover the mathematical details of graph neural networks, the main deep learning architecture that is used in this thesis.

2.1 Motion Planning

Motion planning, also known as path planning, is the computational problem of finding a sequence of valid configurations that move a robot from a defined start state to a goal state while avoiding collisions with the environment as well as self-collisions according to the robot's geometry. For instance, motion planning for a robotic manipulator involves finding a sequence of arm joint angles that move the end-effector from a start pose to a goal pose along a collision-free path. As a fundamental problem in robotics, motion planning has a long history. Mainstream motion planning algorithms can generally be divided into two categories: optimization-based motion planning and sampling-based motion planning. We discuss each category in turn below.

Algorithms in the optimization-based category formulate planning as an optimization problem. The objective function or cost function is the distance to the goal config-

uration, subject to geometric constraints. Although optimization-based motion planning methods are generally faster at generating solutions, they suffer from often prematurely converging to local optima (minima) and require the planning problem to be well-conditioned [10]. To address these issues, researchers have developed various alternative approaches. Among these efforts, Wang et al. [11] develop a method based on genetic algorithms that is less likely to get stuck in local minima. Their method iteratively 'evolves' potential solutions, leading to highly efficient and effective outcomes. Another notable effort in optimization-based motion planning is the ant colony optimization method proposed by Zhang et al. [12], inspired by the colony behavior of ants.

The other major category is sampling-based motion planning. Methods in this category generate random samples in the configuration space and build a tree structure that represents possible paths. A tree search is then carried out to find a feasible path through the configuration space. Sampling-based motion planning is generally more prevalent than optimization-based planning because sampling does not require the planning problem to be well-conditioned. However, sampling-based planning is usually slower compared to optimization-based methods and is affected by the curse of dimensionality. Therefore, an effective heuristic, which prioritizes feasible paths for search, is typically necessary to ensure efficiency.

A significant advance in sampling-based motion planning came with the development of the rapidly exploring random tree (RRT) algorithm by LaValle [13] in 1998. This algorithm efficiently explores the configuration space by randomly sampling and connecting points (or nodes) to construct a space-filling tree. The RRT algorithm quickly expands the tree in several directions, enabling it to cover large areas and identify feasible paths effectively. The RRT method can be combined with heuristics to guide the tree expansion toward the goal, such as goal biasing, for example.

Notably, RRT does not guarantee that the shortest path will be found. To address this limitation, various RRT variants have been proposed to advance the original idea. One such RRT variant is RRT* [14]. This algorithm modifies RRT by re-evaluating and changing the connections between nodes in the tree to reduce the overall path length (cost) as the tree grows. This continuous reconfiguration ultimately leads to better and better solutions; RRT* is able to produce asymptotically optimal solutions, in fact, given sufficient planning time.

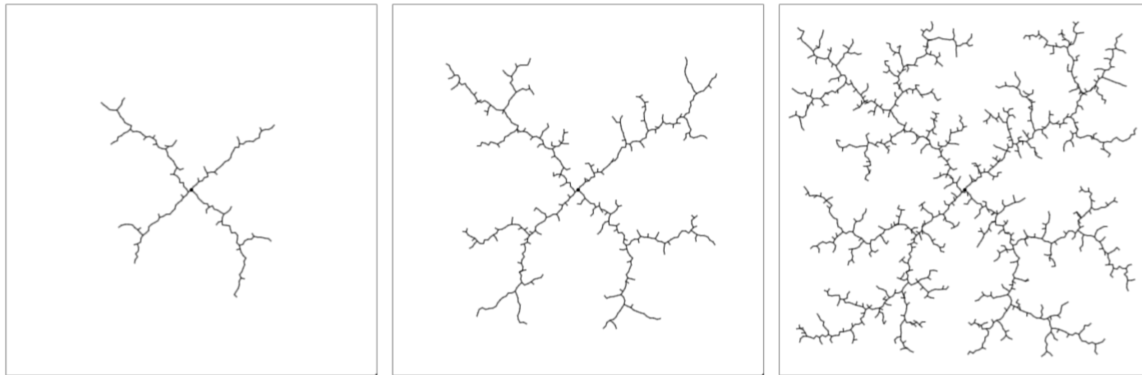


Figure 2.1: Demonstration of RRT from [13]. Left: The tree is rooted at the start state in the center of the figure. Middle: As more sample points are added, the tree covers a greater area and explores a larger number of possible paths. Right: The tree continues to grow until it densely covers the space.

Recently, Gammell et al. [15] proposed the batch-informed trees (BIT*) algorithm, which explores the configuration space like RRT* but uses a smart sampling strategy to focus on promising regions. BIT* further accelerates finding the shortest path in complex environments.

2.2 Task Planning

In contrast to motion planning, robot task planning is a higher-level process that typically involves finding a sequence of discrete sub-goals (to be executed by the motion planner) that accomplish a long-horizon task [16]. Task planning requires knowledge of the effects of the robot's actions and of the abstract state of the environment and objects contained within it. The majority of task planning algorithms involve the application of graph search. Classical AI search algorithms can be divided into three main categories: forward search, backward search, and bidirectional search; we briefly review each category below.

The primary task planning algorithm in robotics is forward search. Forward search begins at the initial state (configuration) and builds a search tree forward toward the goal state (configuration). Forward search is straightforward to implement because it starts from a fully defined initial state. Fast-Forward, developed by Hoffmann [17], is one of the most important and widely used modern task planners due to its outstanding planning efficiency and excellent adaptability to various planning tasks.

Backward search, although less commonly utilized, also plays a significant role in

task planning in robotics. The backward search algorithm starts from the goal state and searches backward until the initial state is reached. Notably, backward search can drastically reduce planning time in many cases by reducing the size of the search space [18].

Forward and backward search algorithms can be applied together in a composite approach called bidirectional search. Bidirectional search runs two searches simultaneously: one forward search from the initial state and one backward search search from the goal state. The algorithm terminates when the two search trees meet at a common intermediate state. Garrett et al. [19] have attempted to combine forward and backward searches to speed up planning. Bidirectional search can be fast but suffers from the problem that the frontiers of the two search trees can fail to meet at a common intermediate state. This failure usually results from difficulties synchronizing the progress of the two searches and ensuring that the trees intersect in complex or irregular search spaces.

A primary challenge with all of the search strategies outlined so far is the curse of dimensionality. Because the algorithms search 'blindly,' they become much less efficient as the number of possible states grows. A heuristic search can be employed to enhance planning efficiency and improve performance. Heuristic search, also known as *informed* search, is a family of tree-searching algorithms that adopts a heuristic function to estimate the cost to reach the goal state from a given intermediate state. In task planning, the cost is usually a measure of resources or effort required to complete the task, or in cases where only a feasible task plan is required, the cost could be the negative estimated feasibility of the task plan. Task planning heuristics essentially inform the branch and bound process, ranking the next high-level actions that will be considered. Heuristic search accelerates problem-solving by prioritizing the least-cost path, especially for NP-hard problems [20]. The A* search algorithm is a representative example from the heuristic search family. A* searches for the lowest-cost path by maintaining a combined estimate of the cost to reach a state (known exactly) and the cost to reach the goal (estimated). At each step, A* expands (i.e., searches forward from) the state with the lowest estimated total cost. An important factor influencing the performance of heuristic search is the informativeness of the heuristic function. Thus, the heuristic functions must be specifically designed for task planning problems. Recent research has also applied machine learning techniques to learn efficient heuristic functions [21].

In most robotics problems, an optimal task planning solution has the lowest possible cost while meeting all objectives and constraints. However, robotic task planning is often a high-dimensional problem, and it is extremely difficult to find an optimal solution within a reasonable time. In this case, robotics researchers have extended the well-known, complete and optimal A* algorithm. Weighted A* search more heav-

ily weights the heuristic function, sacrificing optimality in an effort to find a solution more quickly [22]. Aine et al. [23] describe a multi-heuristic search algorithm that allows multiple heuristic functions to work together to generate feasible plans (i.e., plans that satisfy all constraints) more rapidly. Du et al. [24] extend multi-heuristic A* search to multi-resolution A* search, where the heuristic functions are adjusted based on the complexity of the planning problems.

2.3 Task and Motion Planning

As robotic applications become increasingly complex, the direct solution of the motion planning problem has become a significant challenge. Consequently, research in the hybrid field of task and motion planning (TAMP) has emerged, integrating techniques for task planning with motion planning [7]. Recent advancements in TAMP frameworks reflect substantial progress in this rapidly evolving area. Similar to motion planning, traditional TAMP methods are primarily categorized into optimization-based TAMP and sampling-based TAMP. Algorithms in both categories provide diverse strategies to address the complexities associated with task and motion planning problems.

Optimization-based TAMP treats task and motion planning as a joint optimization problem, using geometric information to determine the problem constraints. Similar to optimization-based motion planning, optimization-based TAMP is faster when finding solutions but may fall into local optima. Logic-geometric programming, for example, is a representative optimization-based method proposed by Toussiant et al. [25] to solve for the robot trajectory at each time step, according to the specific sub-task.

Sampling-based TAMP, which is the method adopted in this thesis, is a more common choice compared to the optimization-based approach. The sampling-based TAMP framework consists of a task planner that discretizes the planning problems into subtasks that are sequenced to reach the goal state [26]. An auxiliary motion planner then later produces a trajectory that satisfies the kinematics, dynamic, and geometric constraints for each sub-task [27]. This joint, bi-level method divides TAMP into task planning, determining *what to do*, and motion planning, choosing *how to do it*. A graphical explanation is shown in Figure 2.2. The high-level, logical task planner breaks down the full task into sub-tasks and sequences them to form a task plan. A low-level motion planner focuses on finding the precise robot trajectories required to complete each sub-task. By combining the two planners, sampling-based TAMP allows robots to execute intricate tasks efficiently and to adapt to changing conditions. Sampling-based methods are (again) typically slower than optimization-based methods.

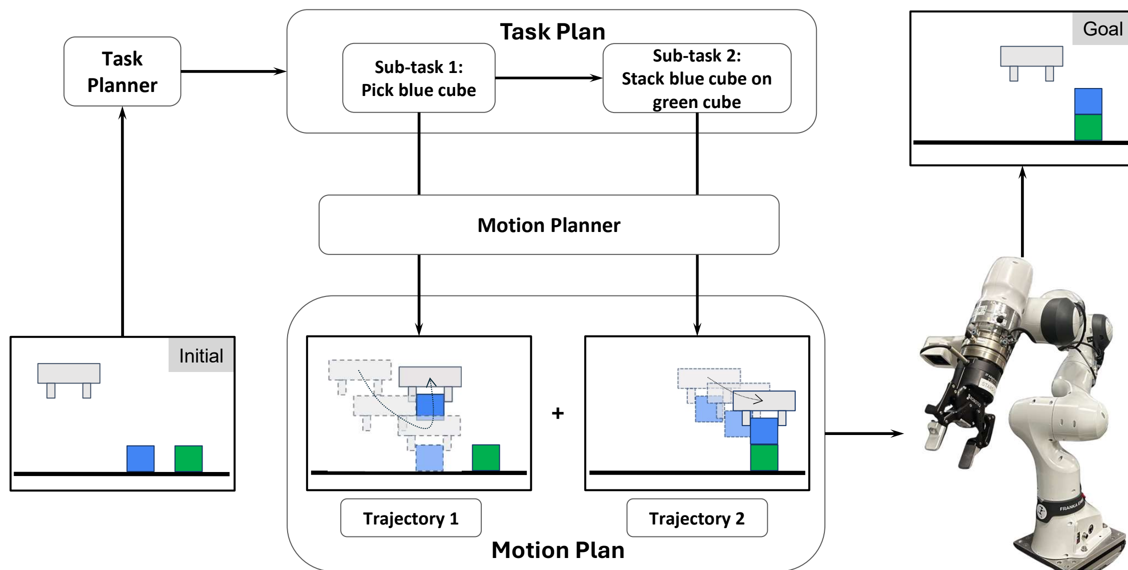


Figure 2.2: An illustration of the task and motion planning pipeline for the task of stacking two cubes. (1) The task planner decomposes the overall task into sub-tasks, 'Pick blue cube' and 'Stack blue cube on green cube.' (2) The motion planner then refines these sub-tasks into specific robot trajectories. (3) The robot trajectories for each sub-task are sequenced together to form the complete motion plan. (4) The robot executes the motion plan to achieve the goal state (stacked cubes).

2.4 Planning Domains and Planning Languages

In the context of task and motion planning, a *planning domain* is a description, typically given in a formal description language, of the problem space in which a robot operates. A concise definition of the concept of a planning domain is given below.

Definition 2.4.1 (Planning Domain)**.** A planning domain is a structured representation encompassing the objects, tasks, actions, and constraints within the working environment of the robot [28].

A planning domain provides the framework for generating a feasible TAMP plan that achieves a specific goal, ensuring that both the high-level task (*what to do*) and the low-level motions (*how to do it*) are effectively integrated. It includes high-level task actions, such as “pick up an object” or “move to a location,” and low-level motion actions, like “grasp.” The planning domain also specifies the objects that the robot is able to interact with, the possible states of the world at any given time, and the constraints governing actions and states, such as collision avoidance and joint limits.

The structured domain representation is crucial to bridge the gap from high-level task requirements to concrete physical actions. TAMP plans are usually discretized and abstracted, simplifying the design and implementation of planning algorithms and enhancing reusability across different tasks and robots.

The design of a planning domain typically involves formulating the logic required to solve a TAMP problem. Logic provides a formal, precise way to carry out automated reasoning in an interpretable way. Depending on the application, planning logic is often a combination of propositional logic and first-order logic. Propositional logic deals with propositions and connectives (i.e., ’and,’ ’or,’ and ’not’) [29]. First-order logic incorporates quantifiers and provides a richer and more expressive language for describing objects, properties, and relationships [30]. The application of either type of logic requires a clear definition and structured domain formulation. Consequently, a well-defined planning domain is essential to efficiently and reliably solving TAMP problems in complex environments. More details are provided in Section 2.4.1.

2.4.1 Standardized Planning Languages

The systematic description of a planning domain is most easily facilitated by the use of a standardized planning language. Early examples of such standardized languages include the Stanford Research Institute Problem Solver (STRIPS) and the Action Description Language (ADL). STRIPS was initially developed as an automated planner by

Richard Fikes et al. [31] at SRI International. Over time, it evolved into what is now known as Action Description Language (ADL), a standardized programming language designed for automated planning and scheduling. These languages provide a structured foundation for defining the rules of task planning, enabling the systematic transformation of planning logic into a format suitable for machine learning and execution.

Inspired by STRIPS and ADL, PDDL (the Planning Domain Definition Language) was developed by McDermott et al. [28] for the International Planning Competition (IPC). PDDL is a human-readable, standardized planning language capable of describing various world logical states; it includes specific constructs for actions (with preconditions and effects) that are not part of first-order logic. However, while expressive enough for planning problems, PDDL is less general than first-order logic and does not support all possible logical constructs or quantifiers. PDDL is now the most widely used planning language and has been interfaced with multiple planning engines. This thesis adopts PDDL for task planning domain formulation.

A PDDL [28] planning domain can formally be written as a tuple $\langle \mathbf{P}, \mathbf{A} \rangle$, where $\mathbf{P}$ is a set of predicates and $\mathbf{A}$ is a set of actions. A *predicate* p is a Boolean function that defines a logical property, condition, or relationship among objects. For example, the predicate (**free** ?gripper) could indicate whether a robot's gripper is holding an object. A ground atom x is obtained by applying the predicate to one or more objects from the set of objects $\mathbf{O}$ in the environment. The objects $o \in \mathbf{O}$ are references to entities in the planning problem [32]. The '?' character in the predicate denotes a variable that can be instantiated with a specific object to produce a ground atom. A logical state $\mathbf{X}$ is a set of ground atoms x describing the environment. An *action* $a = \langle \mathbf{O_a}, \mathbf{Pre}, \mathbf{Eff} \rangle$ can be considered as a logical abstraction of a sub-task. Each action a is a tuple of a set of target parameters $\mathbf{O_a}$, precondition $\mathbf{Pre}$, and postcondition $\mathbf{Eff}$ [33]. The target parameters $\mathbf{O_a}$ specify the objects involved in this action. The precondition $\mathbf{Pre}$ of an action is a set of predicates that must be true to trigger the action. The postcondition $\mathbf{Eff}$ is a set of predicates that indicates the change in the logical state $\mathbf{X}_{\text{after}}$ after the action is executed. The transition of the logical state before and after executing an action is assumed to be deterministic, $\mathbf{X}_{\text{before}} \times a \rightarrow \mathbf{X}_{\text{after}}$.

Figure 2.3 provides one example of an action, 'Pick,' specified in PDDL. The action's parameters are a robot gripper '?gripper', a cube object '?obj', and a table object '?t'. The preconditions of 'Pick' are (**cube** ?obj), (**free** ?gripper), and (**is_on** ?obj ?t). The predicate (**cube** ?obj) indicates that the picked object must be a cube, (**free** ?gripper) indicates the gripper is not holding anything, while (**is_on** ?obj ?t) defines the relative position of the cube and the table. In natural language, the preconditions can be described by the

```
(: (:action Pick
    :parameters (?gripper ?obj ?t)
    :precondition (and (box ?obj) (free ?gripper)
                       (is_on ?obj ?t))
    :effect (and
        (carry ?obj ?gripper)
        (not (free ?gripper))
        (not (is_on ?obj ?t))
    )
))
```

Figure 2.3: Example of a 'Pick' action written in PDDL.

sentence: *The gripper must be free, and the cube must be on the table.* For the postconditions, (**carry** ?obj ?gripper) expresses the fact that the gripper is carrying the object, while (**not** (**is_on** ?obj ?t)) indicates that the cube is no longer on the table, and (**not** (**free** ?gripper)) asserts that the gripper is holding something (and cannot hold anything else). In natural language, the postconditions are given by the sentence: *The gripper is not free because it is holding the cube, and the cube is no longer on the table.*

The state of the robot's operating environment must be fully describable using the logical predicates in the planning domain. The robot can only execute the actions defined in the domain to complete a (full) task. In the context of TAMP, task planning is the process of searching for a sequence of actions that connect the initial logical state to the goal logical state. Once the logical plan is available, a motion planner translates this action sequence into executable robot motion plans.

A *domain set* is the union of the names of the predicates and actions in the planning domain [7]. The domain set contains only the names of predicates and actions, excluding the action target parameters, preconditions, and postconditions. Different planning domains can be constructed from the same domain set by identifying different target parameters, preconditions, and postconditions for each action [34].

2.4.2 Complete and Optimal Planning Domains

Different planning domains can be used to complete the same TAMP task. A domain may contain extra predicates and actions irrelevant to the specific task being considered. In turn, we give two criteria, completeness, and optimality, that can be used to evaluate the expected performance (in terms of computational effort) of planning.

Definition 2.4.2 (Complete Domain)**.** A complete domain is a domain that includes all predicates and actions needed to solve the TAMP problem.

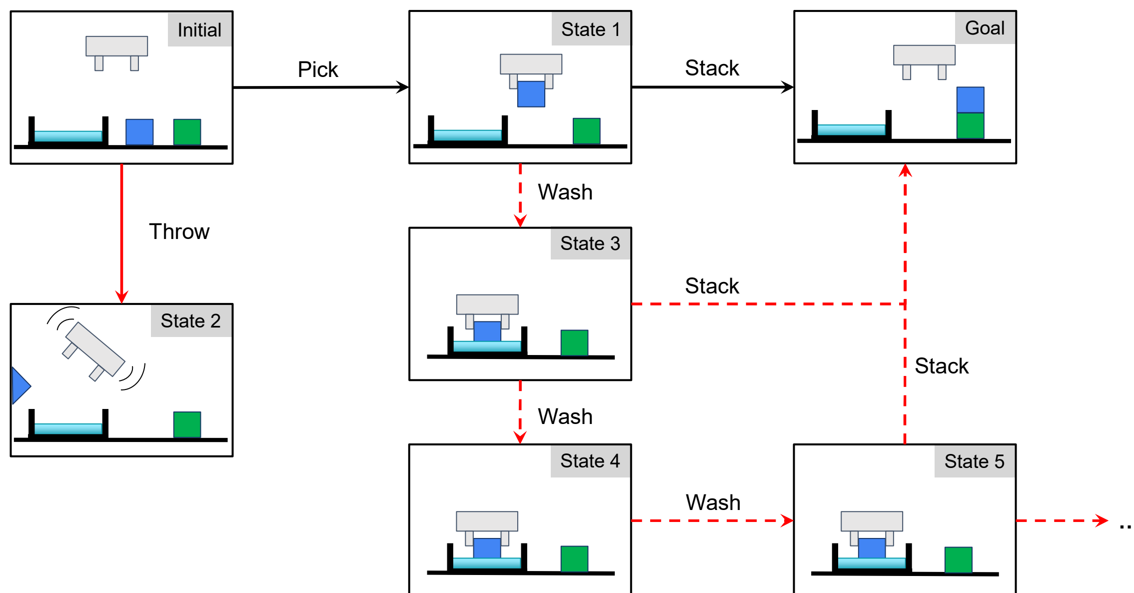


Figure 2.4: A graphic explanation of how redundant actions can negatively impact planning efficiency in a cube-stacking task. The shortest plan, shown with black arrows, includes only the essential actions 'Pick' and 'Stack,' leading directly from the initial state to the goal state. In contrast, the redundant action 'Throw,' indicated by a solid red arrow, leads to a dead end (State 2), while the redundant action 'Wash,' shown with dashed red arrows, causes the exploration of unnecessary states (State 3 and State 4), deviating from the shortest plan and prolonging the planning process.

Definition 2.4.3 (Optimal domain)**.** An optimal domain is a complete domain that contains the least number of predicates and actions.[1]

Clearly, a complete domain is needed in order to successfully find the task plan. The computational effort required to find a task plan depends primarily on the size (predicates and actions) of the domain. As illustrated in Figure 2.4, task planning using PDDL relies on tree search. Starting from an initial state, each action adds a branch to the search tree leading to a successor state. Assuming that every action can be executed in every environment state, we can approximate the complexity of the planning problem. Given the number of actions included in the domain, n_a, and the maximum length of the plan, l, the complexity of a planning problem is $O((n_a)^l)$. Thus, when we include redundant actions, considerable redundant effort may be expended.

Figure 2.4 presents a state-action diagram for a task involving the stacking of two cubes. The planning domain includes both essential actions, such as 'Pick' and 'Stack'

[1] This is the definition of optimality applied in the thesis, but there are other optimality criteria in the literature [7], [23].

(black arrows), which are crucial to reach the goal, and redundant actions, such as 'Throw' (red arrows) and 'Wash' (dashed red arrows), which are not relevant to the goal. Redundant actions may lead to dead ends. For example, as shown in Figure 2.4, the 'Throw' action tosses an object out of the environment, resulting in an irreversible dead end where planning fails. Another example is the 'Wash' action, which increases overall complexity by exploring states that deviate from the goal. The shortest task plan for the stacking of two cubes is 'Pick→Place', while 'Wash' adds unnecessary steps that do not contribute to the goal, thereby reducing planning efficiency.

2.5 Graphs and Graph Neural Networks

A graph is a data structure used to describe objects (i.e., nodes) and their relationships (i.e., edges) [35]. Formally, a graph is a tuple $G = (V, E)$, where V is a set of nodes (or vertices), and E is a set of edges, $E \subseteq \{(u, v) \mid u, v \in V\}$. Edges can be classified mainly into two categories: directional edges, which specify the direction of information flow from the source node to the destination node, and bidirectional edges, which allow information to flow in both directions. For example, in social sciences, directional edges can model relationships, such as one person following another on a social network. Bidirectional edges can represent mutual friendships, where both individuals share the same connection [36]. In chemistry, graphs are widely used to describe the structure of molecules, where nodes represent atoms and edges represent bonds [37]. Directional edges can represent directional bonds. In this thesis, graphs are used to represent the logical state of the environment.

Graph neural networks (GNNs) are designed to work directly with graph-structured data. GNNS are trained (i.e., learn) by passing information between nodes in the graph along the edges, in a process called *message passing*. Past studies have shown that GNNs can be trained to solve TAMP problems by learning the structured relationships between actions and predicates. For example, Silver et al. [38] train a GNN to assess the relevance of each object in the environment to a TAMP problem and to ignore irrelevant objects. Khodeir et al.[21] utilize past plans as demonstrations to train a GNN policy, which predicts the probability that a state will appear in the TAMP plan for a new, related task.

2.5.1 Graph Neural Network Structures

The input to a GNN is a graph, where each node represents an object. Each node has an associated node feature, represented as a vector, that conveys information about the

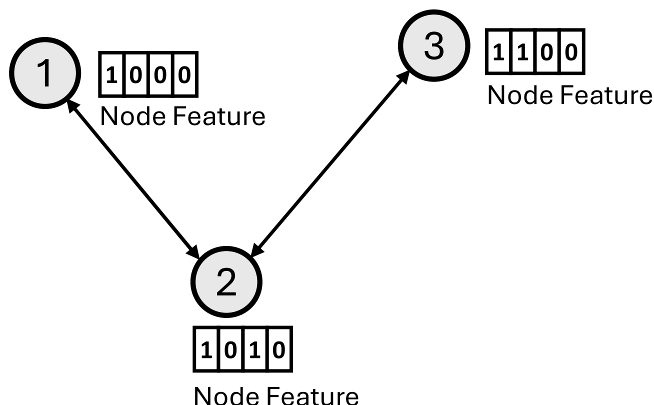


Figure 2.5: An example of graph structures. The grey circles indicate the nodes and the double arrows indicate the bidirectional edges. The node features are indicated by the vector next to each node.

node. The length of the node feature varies depending on the type of information stored for each object. Node features may be numerical or logical values or a complex data format such as an image or a 3D point cloud.

Edges in the input graph connect the nodes and define relationships or interactions between the objects. Similar to node features, edge features can also be defined to encode more information than the edge direction alone. An edge feature is usually defined by a one-hot vector.

Most popular GNN architectures are designed to solve one of three types of problems: node classification, edge classification, or graph classification. Node classification models can predict node properties using existing node attributes [39]. Similarly, edge prediction models, also known as link prediction models, are designed to predict missing properties of edges based on existing node and edge properties [35]. Most edge-prediction tasks involve predicting the existence of edges, similar to binary classification. Finally, graph classification models predict the properties of the entire graph. Graph classification aims to categorize an entire graph based on the node relationships [40].

Scarselli et al. [41] proposed a method to convert a graph and the associated node features into a matrix that serves as a trainable input to a GNN. Node features, expressed as node vectors, are stacked vertically to form the input matrix. For edges, an *adjacency matrix* is proposed in the same paper to represent edge information in matrix form. An adjacency matrix $\mathbf{A}$ is a square matrix. The indices of the rows and columns correspond to the enumeration of the nodes, and the matrix entries indicate (with a one or a zero) whether a connection exists between the corresponding nodes.

As an example, consider the simple graph shown in Figure 2.5, which has three nodes: 1, 2, and 3. Each of the nodes has a corresponding node feature (as indicated). The matrix of node features can be easily found by stacking the node features together,

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}. \tag{2.1}$$

The adjacency matrix for the example graph in Figure 2.5 can be written as

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}. \tag{2.2}$$

The element in the ith row and jth column of the adjacency matrix $\mathbf{A}$ is denoted by $\mathbf{A}_{i,j}$. If $\mathbf{A}_{i,j} = \mathbf{A}_{j,i} = 1$, then the edge is a bidirectional edge. If $\mathbf{A}_{i,j} = 1, \mathbf{A}_{j,i} = 0$, the edge is a directional edge from node i towards node j. Similarly, if $\mathbf{A}_{i,j} = 0, \mathbf{A}_{j,i} = 1$ the edge direction is from node j towards node i. If $\mathbf{A}_{i,j} = \mathbf{A}_{j,i} = 0$, there is no edge between node i and node j.

2.5.2 Message Passing and Aggregation

Graph neural networks use a unique information aggregation process. Considering the stacked node feature matrix $\mathbf{H}$ and the adjacency matrix $\mathbf{A}$, the aggregation rule [41] for a node's connected neighbors can be written as

$$\mathbf{H}' = \sigma\big((\mathbf{A} + \mathbf{I})\,\mathbf{H}\mathbf{W}\big) \tag{2.3}$$

where $\mathbf{W}$ is a trainable (learnable) linear transformation. All nodes share the linear transformation, that is, the same transformation matrix $\mathbf{W}$ is uniformly applied to every node in the network. Unlike sequences, graphs are inherently unordered; the same input graph can have a different adjacency matrix (and a different node feature matrix) by altering the node indices. Therefore, the linear transformation must be invariant to permutations of the node ordering to ensure that the learned representation remains consistent. In Equation (2.3), σ is a nonlinear activation function (with various options available, including the sigmoid and rectified linear unit, or ReLU, functions). The identity matrix $\mathbf{I}$ appears in Equation (2.3) to capture information from a node itself, in addition to its neighbors. Adding the identity matrix is equivalent to adding a loop to each

of the nodes. At each node, $\mathbf{H}'$ is the aggregated information from all connected nodes. The formula given by Equation (2.3) is the basic update rule for GNN learning, but many variations are possible.

The basic updating rule above, while effective, suffers from a problem known as over-smoothing, where the node representations become indistinguishable from each other after multiple layers of graph convolution. Various researchers have developed modified update rules to address this issue. The graph convolution network (GCN) designed by Kipf et al. [42] uses a weighted average of neighboring nodes' features with symmetric normalization to enhance learning stability. GraphSAGE, created by Hamilton et al. [43], employs an inductive approach to generate node embeddings by sampling and aggregating features from local neighborhoods to improve learning scalability. The graph attention network (GAT) proposed by Veličković et al. [40] incorporates an attention mechanism to assign different importance weights to neighboring nodes, allowing for more adaptive and expressive aggregation of node features. Also, GATs allow the use of edge features (encoded as one-hot vectors). GATs and GCNs, both types of GNNs, share a number of similarities but also have important differences. The key difference lies in how they aggregate node information: GCNs aggregate information uniformly from all neighbors, while GATs learn an attention mechanism to assign different weights to neighbors based on their importance. GCNs can be considered as special cases of GATs, in fact, where the importance weights of all neighboring nodes are the same.

Chapter 3

Related Work

For many task and motion planning problems, the logical task planning domain is usually designed in advance by a human being. When faced with simple TAMP problems, manually designing the planning domain is feasible and sufficient. However, it is not possible to manually design all planning domains, particularly for high-dimensional problems. If a domain is not designed properly (e.g., a large number of unnecessary predicates are included), TAMP planners can be extremely slow, as described in Section 2.4.2. Thus, considerable effort has been made to accelerate the planning process and to avoid manually designing planning domains.

This chapter presents an overview of related work on TAMP that is relevant to the thesis. We focus on methods that leverage various machine-learning techniques to enhance planning efficiency and adaptability. In Section 3.1, we discuss how deep learning has been used to accelerate planning, including the application of supervised learning and reinforcement learning. Section 3.2 delves into recent advances related to learning planning domains, including studies of logical predicate learning and automatic planning domain generation.

3.1 Learning to Plan for Faster Task and Motion Planning

Recent advances in machine learning have brought new approaches to task and motion planning. Learning-based TAMP methods often use past plans as demonstration datasets to train policies that either accelerate or replace traditional planners. Recent research highlights the effectiveness of using supervised learning and reinforcement learning to prioritize successful plans.

In supervised learning, the focus is on replicating successful planning experiences from training data. For instance, Kim et al. [44] introduce a relational, logical state rep-

resentation to train graph neural networks (GNNs) for generating TAMP plans. Their method enables robots to move objects while avoiding obstacles by learning from successful demonstrations. Xu et al. [45] employ a convolutional neural network-based encoder to extract geometric information from images and facilitate backward planning. Zhu et al. [18] extend this approach by integrating both geometric and logical information to enable long-horizon manipulation planning. Silver et al. [38] address the challenge of real-world planning involving numerous objects by proposing a GNN architecture that identifies and removes irrelevant objects from the planning process. This reduces computational overhead and simplifies collision checking. McDonald et al. [46] developed a bi-level imitation learning policy with sensory data input to generate TAMP plans. The policy's input includes current environmental observations (RGB images, joint angles, gripper state, and object poses) and logical goals. The high-level policy selects logical actions, while the low-level policy generates continuous control signals to execute these actions. This system is proven capable of handling complex, multi-step robotic tasks. Lin et al. [47] present a GNN policy architecture trained with imitation learning to generate plans for manipulation tasks. Their approach allows for the GNN output's explainability and effectively learns objects' spatial relations in Cartesian space. Yang et al. [48] developed PIGINet, a transformer-based model that assesses the feasibility of a task plan given the task Plan, initial state, and goal state. Their experimental results demonstrate significant improvements in planning speed. Khodeir et al. [21] train a Graph Attention Network (GAT), which assigns attention weights to neighboring nodes in a graph using a set of demonstration plans. The GAT-based policy is integrated with PDDLStream, a variant of the Planning Domain Definition Language (PDDL), to predict the feasibility of actions. This approach demonstrates the potential of deep learning to improve TAMP by leveraging structured data from demonstrations. Dalal et al. [49] utilize visual transformers with image inputs to imitate the behavior of a TAMP planner to generate task and motion plans.

While the methods above have shown that supervised learning can speed up the planning process, they require large amounts of demonstration data. Moreover, the planning policies are conditioned on specific planning environments, meaning that new demonstrations are required for each new environment. As a result, these policies struggle to generalize across different environments or goals. Even though some algorithms can generalize to a larger number of objects within the same environment, their performance declines significantly as the number of objects increases.

Another significant trend in integrating learning with TAMP is the use of reinforcement learning. Chitnis et al. [50] combine an imitation-learned task planner with a

reinforcement-learned motion planner to complete mobile manipulation tasks. This approach uses expert demonstrations to train a task planning policy, which is refined by a reinforcement learning-based motion planner. Paxton et al. [51] use reinforcement learning to train low-level control and high-level task policies to generate safe vehicle motion plans. The high-level policy takes the logical state of the environment and outputs logical actions, such as 'accelerate' and 'stop'. The low-level policy uses the vehicle's continuous state(speed, position, etc.) as input and output control signals(acceleration, steering angle, etc.) directly. Jiang et al. [52] propose Task and Motion Planning Reinforcement Learning (TMP-RL), which integrates TAMP with reinforcement learning to enhance robustness and adaptability in dynamic and uncertain environments. TMP-RL uses a TAMP planner to generate feasible plans and a reinforcement learning loop to learn from execution experiences, improving the planning system's adaptability. Xu et al. [53] develop Deep Affordance Foresight (DAF), a reinforcement learning-based method that introduces a new representation of affordance—defined as the potential success of an action on an object. DAF learns from trial and error to execute multi-step tasks, share affordance knowledge across tasks, and plan using image inputs.

Although reinforcement learning methods often do not require a large training dataset, they are constrained by the need for a well-defined reinforcement learning environment to carry out the learning process [54]. However, creating such environments for complex TAMP problems, in simulation or in the real world, can be challenging, limiting the applicability of reinforcement learning in these scenarios [5]. Most reinforcement learning environments are simplified simulations, and their quality directly impacts policy performance. Constructing highly complex environments is exceedingly difficult, and conducting robot reinforcement learning in the real world is prohibitively expensive. Additionally, reinforcement learning methods often suffer from long training times [55].

3.2 Learning Planning Domains

Recognizing the limitations outlined previously, researchers have increasingly focused on learning planning domains. Traditional methods aimed at learning the behavior of the planner often require a policy to implicitly learn both the planning domain and the search algorithm. However, by focusing solely on learning the planning domain and then combining it with existing search algorithms, the burden on the policy can be reduced. This has led to two main aspects in learning planning domains: logical predicate learning and automatic planning domain generation.

Logical predicate learning often integrates visual information. Kase et al. [56] introduced a framework for learning logical predicates to be used in task planning. Their approach employs a two-level network structure. The high-level network processes a series of images to predict predicates that describe the logical state of the world. These predicates are then used in a pre-designed planning domain to generate task plans. The low-level network takes both the image series and the task plan as input to predict the arm's next joint state. Mukherjee et al. [57] expand on Kase et al.'s work by using 3D point clouds instead of images for predicate learning. However, both methods share a common limitation: they only learn the predicates from visual data, while the planning domain itself remains pre-designed and fixed.

Automatic planning domain generation refers to the automatic creation of planning domains from associated demonstration trajectories. Diehl et al. [33] proposed a method for automatically compiling planning domains from virtual reality (VR) data. However, their approach relies on predefined predicate search trees and does not guarantee the optimality (according to our definition in Section 2.4.2) of the generated domain. Kumar et al. [58] introduce a method for distinguishing necessary preconditions and postconditions through search. Their approach eliminates unnecessary predicates appearing in the preconditions and postconditions of actions by modifying the preconditions and postconditions and then and comparing performance before and after the changes.

Building on Kumar et al.'s research, Silver et al. [34] develop a method to learn state abstractions (predicates) from demonstrations, eliminating the need for manual predicate specification. However, both Kumar et al.'s and Silver et al.'s approaches rely on blind search rather than leveraging past experiences to prioritize likely planning domains. Consequently, their methods require many demonstrations, making the process time-consuming. Additionally, these approaches struggle to generate planning domains for previously unseen environments. Our method is most similar to that of [34], [58], but we train GNN-based domain estimators to accelerate the generation speed of planning domains and reduce the number of demonstrations needed during execution.

3.3 Research Gap

In summary, the related work above shows that learning a planning domain leads to better generalizability and adaptability compared to directly learning a TAMP planner. Although considerable efforts have been made to automate the generation of planning domains, no existing method can produce an optimal planning domain for a new task

from just one or a few demonstrations during execution. Therefore, in this thesis, we combine deep learning with search to find the most likely planning domain for a new task, given one or a few example trajectories that successfully complete the task. Our method thereby enables the rapid generation of an associated, optimal planning domain for the new task.

Method Classification	Method	Mehodologies	Pros	Cons
Supervised Learning	Xu et al. (2019) [45]	CNN-based encoder extracts geometric information from images for backward planning.	Significantly accelerate the planning efficiency in TAMP.	1. Requires a large set of demonstration data. 2. Policies are conditioned on specific planning environments, limiting generalizability.
	Kim et al. (2020) [44]	GNN-based imitation learning to generate task plans considering geometric constrains.		
	Zhu et al. (2021) [18]	Combines geometric and logical data to generate task plan for long-horizon manipulation.		
	Silver et al. (2021) [38]	GNN-based imitation learning to identifies and removes irrelevant objects from the planning process.		
	McDonald et al. (2021) [46]	Bi-level imitation policy uses sensory input to generate TAMP plans for complex multi-step robotic tasks.		
	Lin et al. (2022) [47]	GNN-based imitation learning for manipulation task plan generation, enabling the explainability of the GNN output.		
	Yang et al. (2022) [48]	Transformer-based model speeds up task planning by feasibility assessment.		
	Khodeir et al. (2023) [21]	GAT-based imitation learning to predict action feasibility to accelerate planning.		
	Dalal et al. (2023) [49]	Visual transformers generate TAMP plans by imitating planner behavior from image inputs.		
Reinforcement Learning	Chitnis et al. (2016) [50]	Combine imitation-learned task planning policy with reinforcement-learned motion planning policy.	1. Improve planning efficiency. 2. Enhance the planner's adaptability to dynamic and uncertain environments.	1. Require to construct highly complex environments. 2. The training period is long.
	Paxton et al. (2017) [51]	Reinforcement learning trains high-level task and low-level control policies for safe vehicle motion.		
	Jiang et al. (2019) [52]	Integrate a traditional TAMP planner with reinforcement learning loop to enhance robustness and in dynamic and uncertain environments.		
	Xu et al. (2021) [53]	Reinforcement learning-based method introducing a new representation of affordance for multi-step tasks.		

Table 3.1: The pros and cons of the learning-based methods reviewed in Section 3.1.

Chapter 4

Methodology

Our method aims to automatically generate the planning domain for a new task. We assume that our system is trained with a set of existing TAMP problems that involve different planning domains but that share some predicates and actions. For example, stacking and cooking tasks share the common action 'Pick,' while 'Unstack' only appears in the stacking task, and 'Grill' only appears in the cooking task. Our system has two phases: a training phase and a testing phase. In the training phase, the system is trained to learn the relationships between tasks and their respective planning domains. All predicates and actions used in the planning domains associated with the training TAMP problems are assumed to be known in advance. During the testing phase, we solely utilize the trained system, with no further training conducted. The system is given one or a few examples of continuous state-action trajectories of a robot completing a specific task. Our system searches through all the predicates and actions to select those needed to generate a planning domain capable of reproducing the example robot trajectories and solving TAMP problems in a similar manner.

Previous research has highlighted the high cost of blind search in automatic planning domain generation [33], [58]. The search process can become prohibitively expensive as the number of predicates and actions increases. Studies have shown that leveraging past successful search experience to train graph attention neural networks (GATs) is an effective strategy for accelerating search [21], [38]. Thus, to speed up domain generation, we opt to use GATs to find the most likely planning domain and reduce the search costs.

Our method predicts the most likely planning domain for a new task at test time using a deep learning estimator and then optimizes the domain by adding or removing predicates and actions. We call the complete process of producing a new planning domain for an unseen task *planning domain inference*. In this chapter, we describe our

method and provide one example for a specific robot task. We describe our problem setting in Section 4.1. Then, we introduce our complete domain inference process and provide an overview of each sub-component in Section 4.2. In Section 4.3, we elaborate on the training processes of our predicate and action estimators, which are GAT-based learning algorithms that predict the relevance of a planning domain to the input example trajectories. In Section 4.4, we explain how the preconditions and postconditions of actions are identified. Then, in Section 4.5, we describe our domain optimization strategy, a search algorithm that continuously adjusts the planning domain until the optimal domain associated with the example trajectory is obtained. Finally, in Section 4.6, we present an example of a sorting task that clusters objects into two piles.

4.1 Problem Setting

For a specific task, the object set $\mathbf{O} = \langle o_1, o_2, \ldots, o_n \rangle$ is assumed to include all relevant objects that exist within the environment. The objects $o \in \mathbf{O}$ are references to entities in the planning problem [32]. A known state function provides the continuous state of an object $s : \mathbf{O} \times \mathbf{T} \to \mathbb{R}^{d \times n}$, where $\mathbf{T}$ is the time index used to denote discrete point in time at which the state of an object is evaluated. The continuous state of an object is characterized by its continuous properties, such as pose or temperature, at time t.

Definition 4.1.1 (Full predicate set)**.** The *full predicate set* $\mathbf{P}$ is a set that encompasses all the predicates required to solve a set of TAMP problems from several different environments.

Each predicate $p \in \mathbf{P}$ defines a logical property, condition, or relationship among objects (e.g., (**is_on** ?object1 ?object2), (**cooked** ?object)) via a classifier function that outputs true or false given the continuous state $p(s(o_1, t), s(o_2, t), \ldots) \in \{\text{true}; \text{false}\}$. A ground atom x is a predicate combined with the objects and the classification result (e.g., (**is_on** o_1 o_2), (**not** (**cooked** o)) ...). The logical state of an environment is a collection of all ground atoms for objects in the environment $\mathbf{X} = \langle x_1, x_2, x_3, \ldots, x_n \rangle$.

Definition 4.1.2 (Full action set)**.** The *full action set* $\mathbf{A}$ is a set of actions that logically describe possible actions the robot may execute for a set of TAMP problems.

Each action $a \in \mathbf{A}$ has a corresponding precondition $\mathbf{Pre} = \langle p_1, p_2, \ldots \rangle$, which is a set of predicates that must be true to trigger the action, and a postcondition $\mathbf{Eff} = \langle p_1, p_2, \ldots \rangle$, which is a set of predicates that indicates the change in the logical environmental state after the action is executed. The transition between logical states before and

after the actions is considered to be deterministic, $\mathbf{X}_{\text{before}} \times a \rightarrow \mathbf{X}_{\text{after}}$. A task plan $\pi = \langle a_1, a_2, a_3, ..., a_n \rangle$ is a sequence of logical actions that change the initial logical environmental state to the goal logical environmental state. A (logical) trajectory τ is the sequence of logical environmental state and logical action at each step during task execution, $\tau = \langle (\mathbf{X_1}, a_1), (\mathbf{X_2}, a_2), \ldots, (\mathbf{X_n}, a_n) \rangle$.

Definition 4.1.3 (Full motion planner set)**.** The *full motion planner set* $\mathbf{C}$ is a set of motion planners that produce the corresponding trajectories for actions in $\mathbf{A}$.

For any $a \in \mathbf{A}$, there exists a corresponding $c \in \mathbf{C}$ that can verify the existence of feasible motion plans for the action and produce the motion plan if it exists. For example, the corresponding motion planner for the action 'Pick' will generate a collision-free trajectory for the robot to grasp an object and lift it from the table. The motion planner can be implemented via various methods. For this thesis, we adopt the RRT-based motion planner implemented by Garrett [19].

Using the concept defined above, we can now define a TAMP problem. Any TAMP problem q can be defined by the object set, the full predicate set, the full action set, the initial logical environmental state, and the goal logical environmental state, $q = \langle \mathbf{O}, \mathbf{P}, \mathbf{A}, \mathbf{X}_{\text{init}}, \mathbf{X}_{\text{goal}}, \rangle$. A TAMP problem set $\mathbf{Q}$ is a set of unsolved TAMP problems $\mathbf{Q} = \langle q_1, q_2, \ldots \rangle$.

The planning domain $\mathbf{D}$ associated with a task is characterized by a set of predicates $\mathbf{P_D} \subseteq \mathbf{P}$, a set of actions $\mathbf{A_D} \subseteq \mathbf{A}$, and the corresponding motion planner set $\mathbf{C_D} \subseteq \mathbf{C}$. Formally, the planning domain is defined as $\mathbf{D} = \langle \mathbf{P_D}, \mathbf{A_D}, \mathbf{C_D} \rangle$. The domain set $\omega_\mathbf{D}$ contains all the names of the predicates in $\mathbf{P_D}$ and actions in $\mathbf{A_D}$. The planning domain $\mathbf{D}$ can be constructed from the domain set $\omega_\mathbf{D}$ by designing the preconditions and postconditions for each action and writing the domain into a PDDL file following the PDDL syntax.

4.2 Overview of Inference Framework

Dataset Collection. Our system requires a training dataset in the training phase and two inputs, an example trajectory and a challenge problem set, in the testing phase to produce the planning domain. The training dataset consists of TAMP problems and their associated planning domain sets, with predicates and actions drawn from the full predicate set $\mathbf{P}$ and full action set $\mathbf{A}$. The training dataset is generated through random sampling of various TAMP problems across different experimental planning environments running in the PyBullet simulator. We ensure that each problem is solvable using a traditional TAMP planner and manually label each TAMP problem with the planning

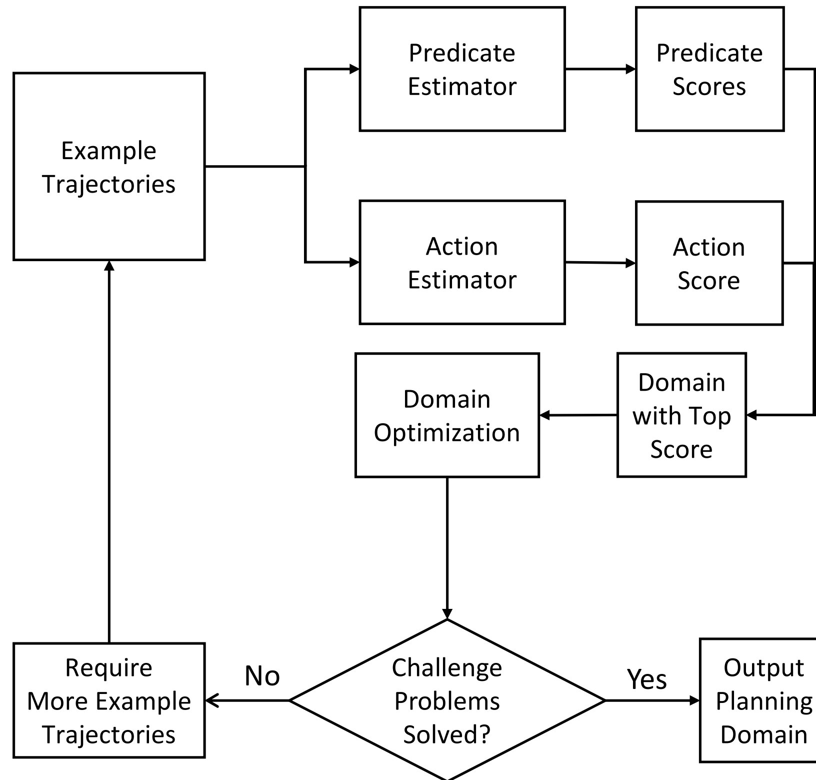


Figure 4.1: The overall framework of domain inference consists of the following steps: (1) The most relevant planning domain, $\mathbf{D}_{\text{top}}$, is predicted by the predicate and action estimators. (2) The predicted domain, $\mathbf{D}_{\text{top}}$, is used as the initial hypothesis for domain optimization, a specific instance of combinatorial optimization, where the problem is to find the smallest subset of predicates and actions associated with the example trajectory. In this process, we keep adding or removing predicates and actions to the planning domain to solve the challenge problem set $\mathbf{Q}_t$. (3) The optimal domain is returned when $\mathbf{Q}_t$ is fully solved; otherwise, more example trajectories from the same type of task are needed.

domains that can solve it. The first input at the testing phase, the example trajectory $\tau_{\mathbf{e}}$, is a logical state-action trajectory of a robot successfully completing a new task. The example trajectory can be collected from a TAMP planner or a human demonstration. The second input, the challenge problem set, is a set of TAMP problems that are solvable by a TAMP planner. The challenge problem set $\mathbf{Q}_t$ contains m unsolved TAMP problems, $\mathbf{Q}_t = \langle q_1, q_2, \ldots, q_m \rangle$. The training dataset is used in the training phase to train our system, while the other two are used in the testing phase to generate the planning domain.

Training Phase. In the training phase, the predicate and action estimators, which are graph attention networks, are trained using the training dataset of TAMP problems and their corresponding planning domain sets. A relevance score $u \in [0,1]$ measures how relevant a predicate(action) is to a specific TAMP problem. A score of 0 indicates that the predicate or action is irrelevant to q, while a score of 1 indicates that it is critical to solving q. The predicate (action) estimator uses the initial state $\mathbf{X}_{\text{init}}$ and the goal state $\mathbf{X}_{\text{goal}}$ of a TAMP problem q to predict a set of relevance scores $\mathbf{U_p}$ ($\mathbf{U_a}$). For each predicate $p \in \mathbf{P}$ (action $a \in \mathbf{A}$), there is a corresponding score $u \in \mathbf{U_p}$ ($u \in \mathbf{U_a}$). The relevance score of a planning domain $u_{\mathbf{D}}$ can be calculated with $\mathbf{U_p}$ and $\mathbf{U_a}$. The relevance score of the domain measures how likely one domain is to be the associated domain for a given TAMP problem. More details in the calculation can be found in Section 4.3. The predicate and action estimators are trained only in the training phase and remain fixed during the testing phase.

Testing Phase. There are two major steps in the testing phase. First, the most likely planning domain for the example trajectory is predicted. The continuous example trajectory is transformed to a logical state-action trajectory $\tau_{\mathbf{e}}$ described by $p \in \mathbf{P}$ and $a \in \mathbf{A}$. The initial logical state $\mathbf{X}_{\text{init}} \in \tau_{\mathbf{e}}$ and the goal logical state $\mathbf{X}_{\text{goal}} \in \tau_{\mathbf{e}}$ are extracted as inputs for the estimators. The estimators predict the relevance score of each predicate and action in the full predicate set $\mathbf{P}$ and the full action set $\mathbf{A}$. The relevance score for the planning domain is then computed and the domain $\mathbf{D}_{\text{top}}$ represents the planning domain with the highest score. Figure 4.1 visualizes the domain inference method at the testing phase.

Second, $\mathbf{D}_{\text{top}}$ is modified by a search algorithm that adjusts the predicates and actions in the planning domain until the optimal domain is obtained. The domain $\mathbf{D}_{\text{top}}$ serves as the initial guess, and at each iteration, a predicate or action is added to or removed from $\mathbf{D}_{\text{top}}$, resulting in a new planning domain $\mathbf{D}_{\text{perturb}}$. We then attempt to solve the challenge problem set $\mathbf{Q}_t$, which is a set of TAMP problems, with $\mathbf{D}_{\text{perturb}}$. If any $\mathbf{D}_{\text{perturb}}$ solves all problems in $\mathbf{Q}_t$, the process continues until $\mathbf{D}_{\text{optm}}$ is obtained. Otherwise, if no $\mathbf{D}_{\text{perturb}}$ can fully solve $\mathbf{Q}_t$ after searching through all possible domains, the

system requires additional example trajectories of the same task to identify the planning domain.

In summary, given **A**, **P**, and **C**, at the testing phase, our system aims at utilizing one or a few example trajectories $\tau_{\mathbf{e}}$ and a challenge problem set $\mathbf{Q}_t$ to produce the optimal planning domain $\mathbf{D}_{\text{optm}} = \langle \mathbf{P}_{\text{optm}}, \mathbf{A}_{\text{optm}}, \mathbf{C}_{\text{optm}} \rangle$ that can reproduce the trajectory $\tau_{\mathbf{e}}$ and generalize to similar TAMP tasks, even with shuffled object poses and varied object numbers.

4.3 Training Predicate and Action Estimators

In this section, we provide a detailed explanation of how the predicate and action estimators are trained. We begin by introducing the architectural design of the GAT-based estimators in section 4.3.1, then elaborate on our method for generating training data in section 4.3.3.

4.3.1 Estimator Design

An important aspect of the GAT-based estimator architecture design is formulating the scene graph, which represents the environment information as graph-structured data. The initial and goal logical states ($\mathbf{X}_{\text{init}}, \mathbf{X}_{\text{goal}}$) of a TAMP problem are represented as scene graphs that are input to our estimators, where each node in the graph represents a single object. Two types of predicates, represented by the node feature and edges, are formulated. The first type, unary predicates, defines the logical states of individual objects $o \in \mathbf{O}$, such as (**top**, ?o) and (**cleaned**, ?o), and is encoded as Boolean elements within the node feature vector. For example, with node feature [(**top**, ?o), (**cleaned**, ?o)], an object on top of a pile but not cleaned will have the node feature $[1,0]$. The second type of predicate, binary predicates, specifies relationships between objects, such as (**is_on**, ?o_1, ?o_2), and is represented as edges connecting the nodes. The edge type is characterized by an edge feature encoded as a one-hot vector.

Figure 4.2 shows the graphical formulation for a cube unstacking problem. The graph on the left is the initial scene graph, and the graph on the right is the goal scene graph. The node feature is [**object index**, (**top**, ?o), (**cleaned**, ?o), (**cooked**, ?o)], where **object index** indicates the type of object with an integer. In this planning problem, the index for Cube is 1. The predicate (**top**, ?o) indicates whether the object o is on top. The predicate (**cleaned**, ?o) indicates whether the object o is cleaned. The predicate (**cooked**, ?o) indicates whether the object o is cooked. Each node is assigned a node feature rep-

resenting the logical state of the corresponding object in the environment. As shown in Figure 4.3, the edges represent the predicates describing relations between the objects such as (**is_on**, $?o_1$, $?o_2$). The type of edge is specified by an edge feature, which is a one-hot vector. The dashed edge is a special edge connecting the same object's initial and goal states.

Each estimator processes the graph input (scene graph of initial and goal logical states) using a series of GAT layers. After the graph data is processed through these GAT layers, the output of the final GAT layer is passed through a multilayer perceptron (MLP) layer. The MLP layer linearly transforms this processed graph information into a vector, and an activation function is applied to bring the range of each output element to between 0 and 1. Each element of the output vector represents the relevance score of a specific predicate or action to the input.

4.3.2 Relevance Scoring

Next, we introduce the method to compute the relevance score of a candidate planning domain. Drawing inspiration from the calculation of the total probability of independent events, we propose a method for computing the domain set score. In this method, the relevance scores for all actions and predicates contribute to the overall domain set score in the following manner:

$$u_{\mathbf{D}} = \prod_{p \in \omega_{\mathbf{D}}} u_{\mathrm{p}} \prod_{p \notin \omega_{\mathbf{D}}} (1 - u_{\mathrm{p}}) \prod_{a \in \omega_{\mathbf{D}}} u_{\mathrm{a}} \prod_{a \notin \omega_{\mathbf{D}}} (1 - u_{\mathrm{a}}) \tag{4.1}$$

In Equation (4.1), u_{p} represents the score for a predicate, while u_{a} represents the score for an action. Similar score computation schemes can be found in many other learning-based planning methodologies [38].

The scores for all possible domain sets can be computed using Equation (4.1). The domain set with the highest relevance score is denoted as $\omega_{\mathbf{D}}^{\mathrm{top}}$. The predicates and actions excluded from $\omega_{\mathbf{D}}^{\mathrm{top}}$ are ranked by their relevance score in a descending manner in a priority list L

4.3.3 Dataset Generation

To effectively train the predicate and action estimator, we need a demonstration dataset of planning problems, each defined by initial and goal environmental logical states. To create this dataset, we employ a sampling-based method that randomly samples a wide range of planning problems in the simulation environment. These TAMP problems are

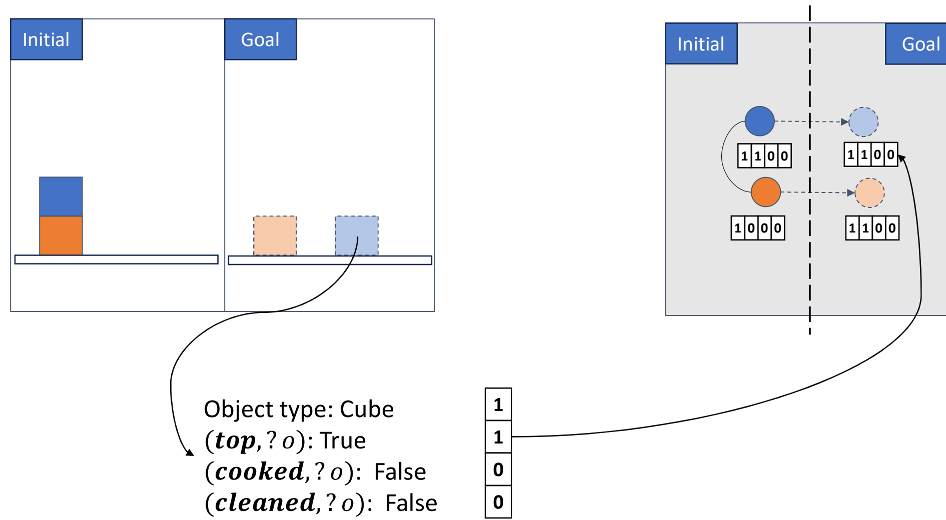


Figure 4.2: Formulating predicates as node features: The dashed blue cube in the goal scene has an object index 1; the cube is on top, so (**top**, ?o) is true. The cube is not cleaned or cooked, so both (**cleaned**, ?o) and (**cooked**, ?o) are false. The resulting node feature vector is $[1,1,0,0]$. We can do the same for other cubes to obtain their node features

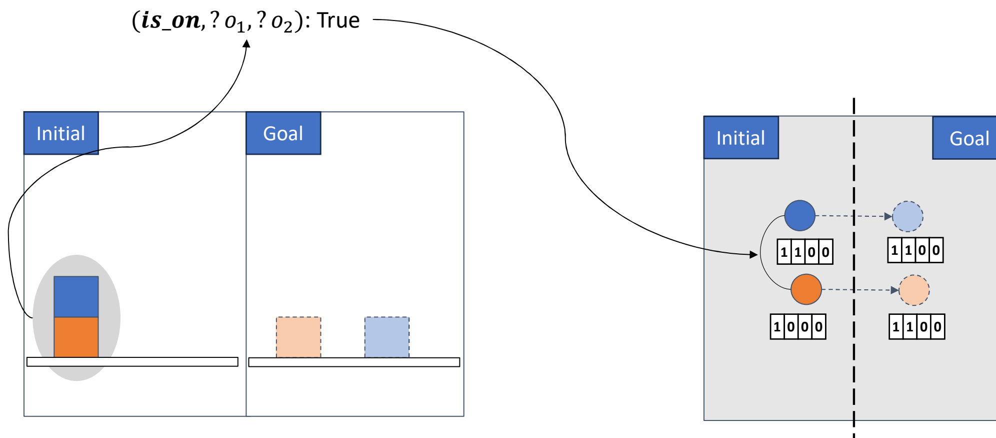


Figure 4.3: Formulating predicates as edge connections: In the initial scene, the predicate (**is_on**, ?o_1, ?o_2) is true for the blue and orange cubes. Thus, a bidirectional edge, represented by a solid line, is formulated to connect them.

labeled with the predicates and actions required for the corresponding planning domain. In this labelling process, predicates and actions crucial for solving the problem are given a score of 1, while irrelevant ones are assigned a score of 0. However, before this dataset can be utilized to train the estimators, it is important to validate the feasibility of solving the sampled problems. This validation step involves verifying if each generated problem has a feasible solution. Including impossible problems in the training data would compromise the performance of the predicate and action estimators.

Existing methodologies for feasibility verification include human verification and automated TAMP planner verification. Human verification has been employed in TAMP research for a long time, but it significantly limits the speed of data generation, and the data quality can be low due to potential human error [59]. An alternative approach involves using traditional TAMP planners to verify the validity of the sample problems. The autonomous nature of these planners allows for massive data generation [21]. Utilizing TAMP planners also results in a cleaner dataset by avoiding human error. Therefore, using traditional TAMP planners is the superior choice for problem verification.

In this thesis we develop a TAMP planner implementation for the Franka Panda arm, which can be parallelized on a compute cluster to check feasibility. Our TAMP planner supports launching multiple planning interfaces simultaneously. The planner reads the input of planning problems and parses them into planning simulation environments. The output of the planner includes information on whether the problem-solving was successful and the complete trajectory of the manipulation (if the TAMP problem is solved). This data generator is specifically designed for the supercomputer operating system. It can be directly cloned online and built on supercomputers automatically.

4.4 Precondition and Postcondition Generation

In the testing phase, we obtain $\omega_{\mathbf{D}}^{\text{top}}$, which is a set of the names of all predicates and actions included in the domain $\mathbf{D}_{\text{top}}$, from the predicate and action estimators. Then, we must convert $\omega_{\mathbf{D}}^{\text{top}}$ into an executable PDDL domain $\mathbf{D}_{\text{top}}$ by determining the target parameters, preconditions, and postconditions for each action. This process depends on finding the commonalities across instances of the same action from the demonstrations.

Definition 4.4.1 (Pre-image)**.** The logical world state for the last time step before the action is triggered.

Definition 4.4.2 (Post-image)**.** The logical world state for the first time step after the

action is completed.

The precondition is determined by identifying the intersections of the pre-images of the same action across multiple instances. As shown in Figure 4.4, we first extract the pre-images for all actions and then group these pre-images by the same action. Next, we find the intersection of all pre-images for the same action, which is then used as the precondition.

Similarly, for generating postconditions, we start by extracting the post-images for all actions and grouping them by the same action. The intersection of the post-images for each action is then calculated. To refine the postcondition, this intersection is compared to the corresponding precondition, and any unchanged predicates are removed. The resulting set of predicates is defined as the postcondition.

Collecting the predicates and actions with preconditions and postconditions, the planning domain is automatically written into a PDDL file, which is ready for task and motion planning.

4.5 Domain Optimization by Generate-and-Test Search

The domain optimization process is based on the generate-and-test search method, a heuristic search technique that involves backtracking. This method ensures that a solution is found if one exists. The technique involves generating all potential solutions and testing them to determine the optimal one [60]. Starting from $\mathbf{D}_{\text{top}}$, the optimization process iterates through all possible domains by adding or removing one predicate or action from $\mathbf{D}_{\text{top}}$ at each iteration until the domain successfully solves the challenge problem set $\mathbf{Q}_t$ with the minimal number of predicates and actions.

There are a few assumptions made to perform the generate and test search:

Assumption 1. If a domain $\mathbf{D}$ is incomplete, then any subdomain $\mathbf{D_s} \subseteq \mathbf{D}$ is incomplete.

Assumption 2. If a domain $\mathbf{D}$ is complete, then the optimal domain $\mathbf{D}_{\text{optm}} \subseteq \mathbf{D}$.

Recall the formal definitions of a complete domain and an optimal domain. A complete domain can help the TAMP planner find a solution whenever a solution exists. An optimal domain is a complete domain that contains the minimal number of predicates and actions.

The optimization process aims to add or remove predicates (actions) from the initial guess until the optimal domain is found. As indicated in Algorithm 1, $\mathbf{D}_{\text{perturb}}$ is initialized as $\mathbf{D}_{\text{top}}$. There are two major steps in the optimization process: expansion and contraction.

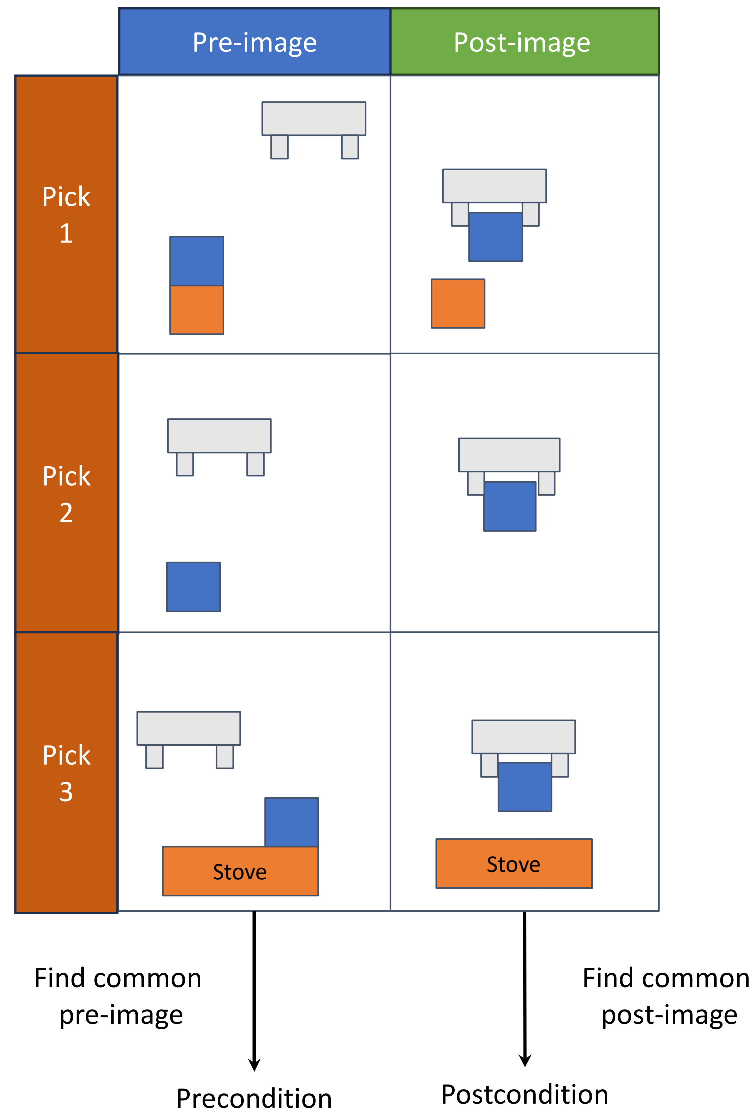


Figure 4.4: Precondition and postcondition generation: (1) The pre-images and post-images are collected and grouped for each action. (2) The intersection of all pre-images (post-image) is formulated as the precondition (postcondition)

Algorithm 1 Domain Optimization Algorithm.

1: $\mathbf{D}_{\text{perturb}} = \mathbf{D}_{\text{top}}$ ▷ Initialization.
2: $is_solved = plan(\mathbf{D}_{\text{perturb}}, \mathbf{Q}_t)$
3: $element_list = \omega_{\mathbf{D}}^{\text{top}}$
4: ▷ $top(L)$ returns the highest-scored element from L.
5: **while** not is_solved **do**
6: $\mathbf{D}_{\text{perturb}} = add(\mathbf{D}_{\text{perturb}}, top(L))$
7: $is_solved = plan(\mathbf{D}_{\text{perturb}}, \mathbf{Q}_t)$
8: $element_list.insert(top(L))$
9: $L.pop(top(L))$
10: **end while**
11: $\mathbf{D}_{\text{optm}} = \mathbf{D}_{\text{expanded}} = \mathbf{D}_{\text{perturb}}$
12: **for** $e_i \in element_list$ **do**
13: $\mathbf{D}_{\text{perturb}} = remove(\mathbf{D}_{\text{optm}}, e_i)$
14: $is_solved = plan(\mathbf{D}_{\text{perturb}}, \mathbf{Q}_t)$
15: **if** is_solved **then**
16: $\mathbf{D}_{\text{optm}} = \mathbf{D}_{\text{perturb}}$
17: **end if**
18: **end for**

In the expansion stage, $\mathbf{D}_{\text{perturb}}$ is expanded until it solves the challenge problem set $\mathbf{Q}_t$. In each iteration, we remove the top element in the priority list L, which has the highest relevance score, and add it to $\mathbf{D}_{\text{perturb}}$. Then $\mathbf{D}_{\text{perturb}}$ is tested with $\mathbf{Q}_t$. We continue this loop until $\mathbf{D}_{\text{perturb}}$ can fully solve $\mathbf{Q}_t$. If $\mathbf{D}_{\text{top}}$ can fully solve $\mathbf{Q}_t$ without any additional predicates or actions, the expansion stage can be skipped.

In the contraction stage, we remove redundant elements from $\mathbf{D}_{\text{optm}}$ until no more elements can be removed while still solving the challenge problem set $\mathbf{Q}_t$. The final domain from the expansion stage, $\mathbf{D}_{\text{expanded}}$, is assigned as the initial $\mathbf{D}_{\text{optm}}$. In each iteration i, we remove one element, either a predicate or an action, $e_i \in \mathbf{D}_{\text{optm}}$, to produce a perturbed domain $\mathbf{D}_{\text{perturb}}$. If $\mathbf{D}_{\text{perturb}}$ maintains the ability to fully solve $\mathbf{Q}_t$, indicating that the removed element e_i is redundant, $\mathbf{D}_{\text{perturb}}$ becomes the new optimal domain $\mathbf{D}_{\text{optm}}$. If $\mathbf{D}_{\text{perturb}}$ fails to solve $\mathbf{Q}_t$, the removed element e_i is critical. In this case, we backtrack to the last domain that fully solved $\mathbf{Q}_t$ as $\mathbf{D}_{\text{optm}}$. This iteration continues until each element in $\mathbf{D}_{\text{expanded}}$ has been verified.

4.6 Example of Domain Inference

Figure 4.5 shows the domain inference process for sorting objects into two clusters. The objects are sparsely placed on the table in the initial state. One example trajectory τ_{sort}

and a challenge problem set $\mathbf{Q}_{\text{sort}}$ are provided to the system. The most relevant domain $\mathbf{d}_{\text{top}}$ and the priority list L are produced by the estimators.

In the first iteration, the perturbed domain $\mathbf{D}^1_{\text{perturb}}$ is the highest-relevance-score domain, $\mathbf{D}_{\text{top}}$. However, $\mathbf{D}^1_{\text{perturb}}$ cannot fully solve $\mathbf{Q}_{\text{sort}}$ because it is missing a critical predicate, (**holding**, ?o, ?r), which prevents the robot ?r from grasping multiple objects simultaneously. Therefore, the system enters the expansion stage. The domain $\mathbf{D}^1_{\text{perturb}}$ is considered incomplete, and no optimal domain is found at this step.

In the second iteration, the domain is perturbed by adding the predicate (**holding**, ?o, ?r), which has the highest relevance score in L. Now, $\mathbf{D}^2_{\text{perturb}}$ can fully solve $\mathbf{Q}_{\text{sort}}$. Thus, the planning domain $\mathbf{D}^2_{\text{perturb}}$ is considered complete and is designated as the current optimal domain, $\mathbf{D}_{\text{optm}}$.

However, at this stage, the optimality of the planning domain is not guaranteed, as it may contain irrelevant predicates. Therefore, from iteration 3 onward, the system enters the contraction stage.

In the third iteration, the predicate (**on_table**, ?o, ?t) is removed from $\mathbf{D}_{\text{optm}}$ to produce $\mathbf{D}^3_{\text{perturb}}$, which cannot fully solve $\mathbf{Q}_{\text{sort}}$ and is considered an incomplete domain. The predicate (**on_table**, ?o, ?t) specifies the vertical relative relation between an object o and a table t. Without this predicate, the planner does not know which table an object should be on. Therefore, the algorithm backtracks to $\mathbf{D}^2_{\text{perturb}}$ to be $\mathbf{D}_{\text{optm}}$.

In the fourth iteration, the predicate (**top**, ?o) is removed from $\mathbf{D}_{\text{optm}}$ to produce $\mathbf{D}^4_{\text{perturb}}$. Since all objects are sparsely placed on the table in the initial state, (**top**, ?o) is unnecessary. The domain $\mathbf{D}^4_{\text{perturb}}$ can fully solve $\mathbf{Q}_{\text{sort}}$ and is considered a complete domain. The predicate (**top**, ?o) is considered redundant, and the current optimal domain, $\mathbf{D}_{\text{optm}}$, is updated to $\mathbf{D}^4_{\text{perturb}}$.

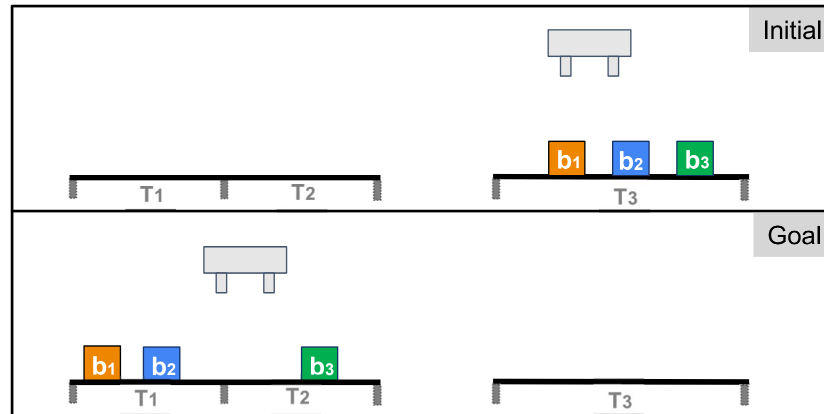
In the fifth iteration, (**is_on**, ?o_1, ?o_2) is removed from $\mathbf{D}_{\text{optm}}$ to produce $\mathbf{D}^5_{\text{perturb}}$, which can fully solve $\mathbf{Q}_{\text{sort}}$ and is considered a complete domain. The predicate (**is_on**, ?o_1, ?o_2) specifies the vertical relative relation of two objects. As objects are sparsely placed on the table, the predicate (**is_on**, ?o_1, ?o_2) is classified as redundant by our system, and the current optimal domain $\mathbf{D}_{\text{optm}}$ is updated with $\mathbf{D}^5_{\text{perturb}}$.

In the sixth iteration, the action 'Pick' is removed from $\mathbf{D}_{\text{optm}}$ to produce $\mathbf{D}^6_{\text{perturb}}$, which cannot fully solve $\mathbf{Q}_{\text{sort}}$ and is considered an incomplete domain. Without 'Pick', no action can be executed from the initial state, and the planner will fail. The action 'Pick' is considered critical, and the algorithm backtracks to $\mathbf{D}^5_{\text{perturb}}$ to be $\mathbf{D}_{\text{optm}}$.

In the last iteration, the action 'Place' is removed from $\mathbf{d}_{\text{optm}}$ to produce $\mathbf{D}^7_{\text{perturb}}$, which cannot fully solve $\mathbf{Q}_{\text{sort}}$ and is considered an incomplete domain. Without 'Place', the planner cannot move an object to its target position. The action 'Place' is considered

to be critical, and the algorithm backtracks to $\mathbf{D}^5_{\text{perturb}}$ to be $\mathbf{d}_{\text{optm}}$.

Eventually, $\mathbf{D}_{\text{optm}}$ ($\mathbf{D}^5_{\text{perturb}}$) is returned since it fully solves $\mathbf{Q}_{\text{sort}}$, and all redundant elements have been removed during the contraction phase.



# iteration	Perturbed domain elements	Q_{sort} solved ?	Failure mode
1	on_table, top, is_on, pick, place	✗	pick b_1→ pick b_2 →…
2	on_table, top, is_on, holding pick, place	✓	N/A
3	top, is_on, holding pick, place	✗	pick b_1→ stack b_1 on ? →…
4	on_table, is_on, holding pick, place	✓	N/A
5	on_table, holding pick, place	✓	N/A
6	on_table, holding place	✗	No action can be applied
7	on_table, holding pick	✗	pick b_1→ ?

Figure 4.5: Example of the domain optimization process of sorting cubes. In initial state, objects b_1, b_2, and b_3 are sparsely placed on table T_3. The robot needs to move the objects from T_3 to T_1, T_2 as specified in the goal state. The elements in the perturbed domain set for each iteration are listed in the table together with its ability to solve $\mathbf{Q}_{\text{sort}}$ and the failure mode. The optimal domain is found at iteration 5, which is highlighted.The detailed explanation of the failure modes can be found in Section 4.6

Chapter 5

Experiments

To demonstrate the advantages of our method, we designed a variety of TAMP tasks, each characterized by a unique planning domain. These tasks include classic cube manipulation, everyday task scenarios, and puzzle solving. This chapter describes each experimental task, with a corresponding graphic showing the initial environment state and the goal environment state. Section 5.1 provides details on our experiments involving classical cube manipulation tasks. Section 5.2 presents a few experiments with everyday tasks such as cleaning. Section 5.3 outlines a puzzle-solving task experiment. The tasks in Section 5.1, Section 5.2, and Section 5.3 are considered to be basic tasks. In Section 5.4, we design several tasks that are composed of sequences of the basic tasks. These composed tasks are used to evaluate whether our approaches can robustly generalize. After explaining our experiment setup, we provide details about our training and testing datasets in Section 5.5 and introduce a baseline method for comparison in Section 5.6. Through our experiments, we aim to answer the following questions:

- How accurately can our system generate the planning domain for a given new task?
- How effective is planning using the generated planning domain? Are we able to generalize to unseen and more complex tasks?
- During the testing phase, how much does our method reduce computational costs compared to blind search?

All experiments rely on the PyBullet simulation platform [61]. The task planner is based on the Fast-Forward library [17] interfaced with PyBullet. This PyBullet task planner library is developed by Garret et al. [7]. The motion planner is an RRT-based algorithm implementation available in the `pybullet-planning` library, also developed by

Garret et al. [19]. For each action $a \in \mathbf{A}$, the corresponding motion planner $c \in \mathbf{C}$ is designed to perform effectively across various environments. GAT training is implemented in PyTorch [62] and PyTorch Geometric [63]. A full explanation of the meaning of predicates and actions is provided in the appendix.

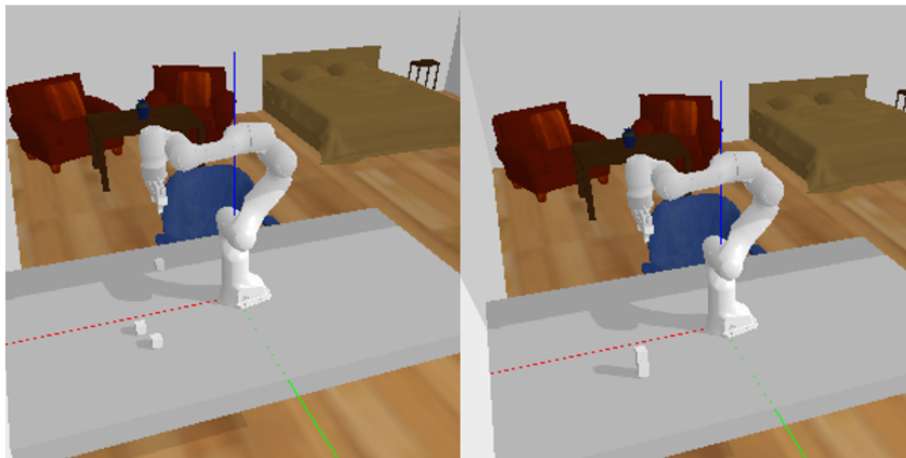


Figure 5.1: A graphical view of our cube stacking experiment: Left: The initial state, where three cubes sit on the table. Right: The goal state, where the cubes are stacked vertically.

5.1 Cube Manipulation

The cube manipulation tasks in this thesis include stacking cubes, unstacking cubes, and sorting cubes into clusters.

As shown in Figure 5.1, the stacking task requires the robot to stack cubes into a tower in a specific order. Initially, the cubes are randomly placed on the table. In the goal state, the cubes are stacked into a tower. The predicates and actions for the domain are:

- $\mathbf{P}_{\text{stack}}$ = {(**on_table** `?obj ?table`), (**top** `?obj`), (**is_on** `?upper ?lower`), (**holding** `?obj ?gripper`)}
- $\mathbf{A}_{\text{stack}}$ = {**Stack**, **Unstack**, **Pick**, **Place**}

The unstacking task is the reverse of the stacking task. As shown in Figure 5.2, the unstacking task involves unstacking a tower of cubes and placing them on a table. The predicates and actions for the domain are:

- $\mathbf{P}_{\text{unstack}}$ = {(**on_table** `?obj ?table`), (**top** `?obj`), (**is_on** `?upper ?lower`), (**holding** `?obj ?gripper`)}
- $\mathbf{A}_{\text{unstack}}$ = {**Stack**, **Unstack**, **Pick**, **Place**}

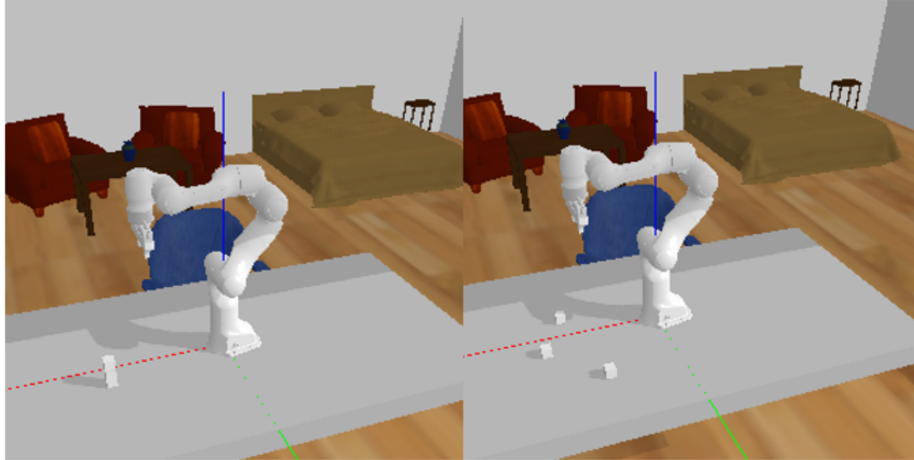


Figure 5.2: A graphical view of the unstacking experiment: Left: The initial state, where the cubes are stacked vertically. Right: The goal state, where three cubes are sitting on the table.

Figure 5.3: A graphical view of sorting experiment: Left: The initial state, where three cubes sparsely sit on the table. Right: The goal state, where the cubes are clustered into two piles.

The sorting task requires the robot to cluster cubes into two groups at different locations. In Figure 5.3, the cubes are randomly placed on the table in the initial scene and are clustered by the robot into two piles when the task is completed. The predicates and actions for the domain are:

- $\mathbf{P}_{\text{sort}}$= {(**on_table** ?obj ?table), (**holding** ?obj ?gripper)}
- $\mathbf{A}_{\text{sort}}$={**Pick**, **Place**}

5.2 Everyday Tasks

This section presents a few experiments involving everyday tasks, including washing, grilling, cooking, table cleaning, and painting.

The washing task requires the robot to wash food ingredients in the kitchen sink. As shown in Figure 5.4, the ingredients are randomly placed on the table in the initial scene, and the robot moves the ingredients into the sink to wash them. The predicates and actions for the domain are:

- $\mathbf{P}_{\text{wash}}$= {(**on_table** ?obj ?table), (**holding** ?obj ?gripper), (**cleaned** ?obj)}
- $\mathbf{A}_{\text{wash}}$={**Pick**, **Place**, **Clean**}

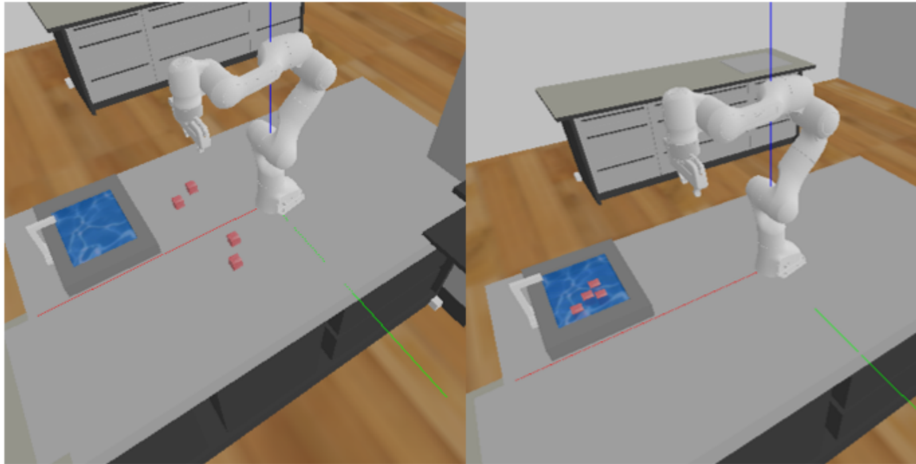


Figure 5.4: A graphical view of washing experiment: Left: The initial state, where the food ingredients sit on the table. Right: The goal state, where the food ingredients are moved into the sink.

The grilling task requires the robot to grill food on the stove. As shown in Figure 5.5, the ingredients are randomly placed on the table in the initial scene, and the robot moves the ingredients to the stove and grills them. The predicates and actions for the domain are:

- $\mathbf{P}_{\text{grill}}$= {(**on_table** ?obj ?table), (**holding** ?obj ?gripper), (**cooked** ?obj)}
- $\mathbf{A}_{\text{grill}}$={**Pick**, **Place**, **Cook**}

The cooking task involves a combination of washing and grilling food. As shown in Figure 5.6, the raw ingredients are initially randomly placed on the kitchen table. The robot is responsible for washing ingredients in the sink, grilling them on the stove, and placing them back on the table. The predicates and actions for the domain are:

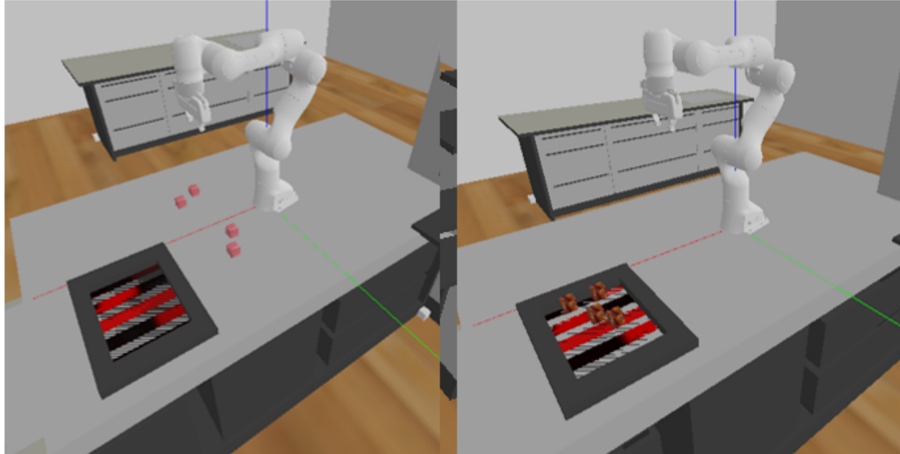


Figure 5.5: A graphical view of grilling experiment: Left: The initial state, where the food ingredients sit on the table. Right: The goal state, where food ingredients are moved on the stove to grill.

- P_{cook}= {(**on_table** ?obj ?table), (**holding** ?obj ?gripper), (**cleaned** ?obj), (**cooked** ?obj)}
- A_{cook}={**Pick**, **Place**, **Cook**, **clean**}

As shown in Figure 5.7, the table cleaning task involves collecting objects from the table and storing them in a bin. The objects are randomly placed on the table in the initial scene and the bin is closed. The robot needs to open the bin, move all the objects on the table into the bin, and then close the bin. The predicates and actions for the domain are:

- P_{table_clean}= {(**on_table** ?obj ?table), (**holding** ?obj ?gripper), (**in** ?obj ?container), (**containable** ?obj), (**is_container** ?obj), (**closed** ?container, ?cover)}
- A_{table_clean}={**Pick**, **Place**, **Open**, **Close**, **Dispose**}

As shown in Figure 5.8, the painting task involves drawing figures on an object or a piece of paper. In the initial scene, the objects to be painted are randomly placed on the table. The robot needs to pick up the pen from the pen box, dip it in the pigment (black ink), and paint the object. The predicates and actions for the domain are:

- P_{paint}= {(**on_table** ?obj ?table), (**holding** ?obj ?gripper), (**colored** ?pen), (**is_pen** ?obj), (**is_pigment** ?obj), (**painted** ?obj, ?pen)}
- A_{paint}={**Pick**, **Dip**, **Paint**}

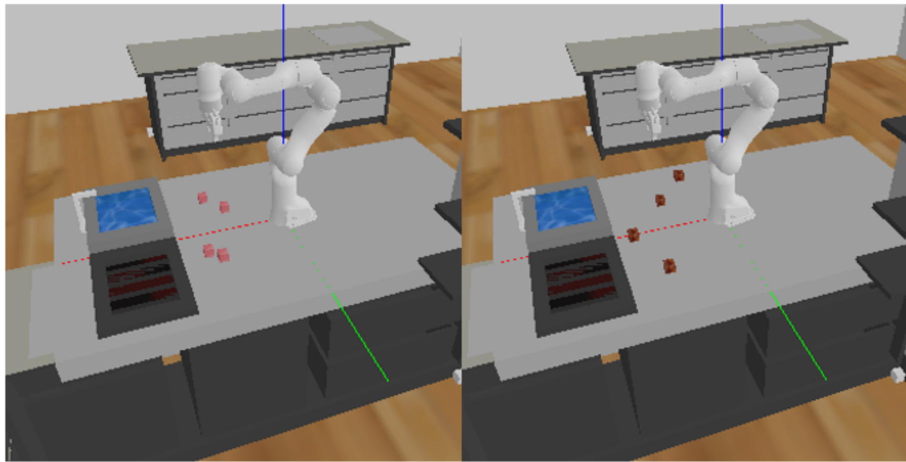


Figure 5.6: A graphical view of the cooking experiment: Left: The initial state, where the raw food ingredients sit on the table. Right: The goal state, where the food ingredients are washed and grilled.

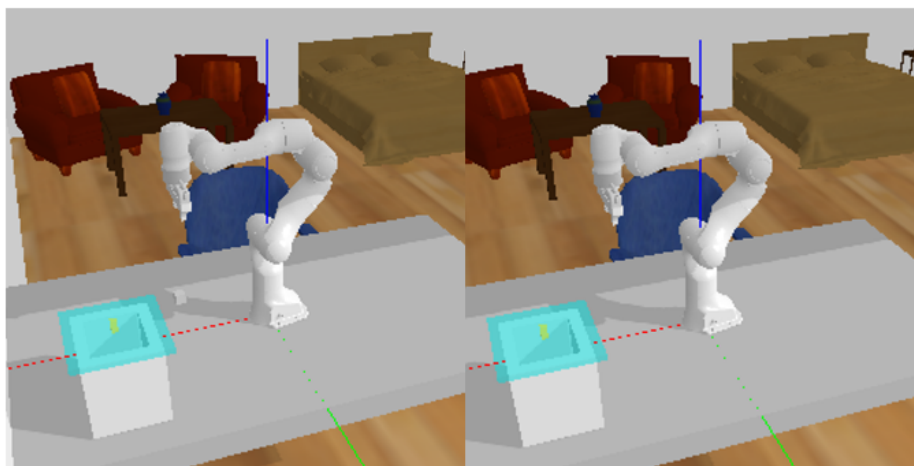


Figure 5.7: A graphical view of table cleaning experiment: Left: The initial state, where one object and the bin sit on the table. Right: The goal state, where the object is moved inside the bin.

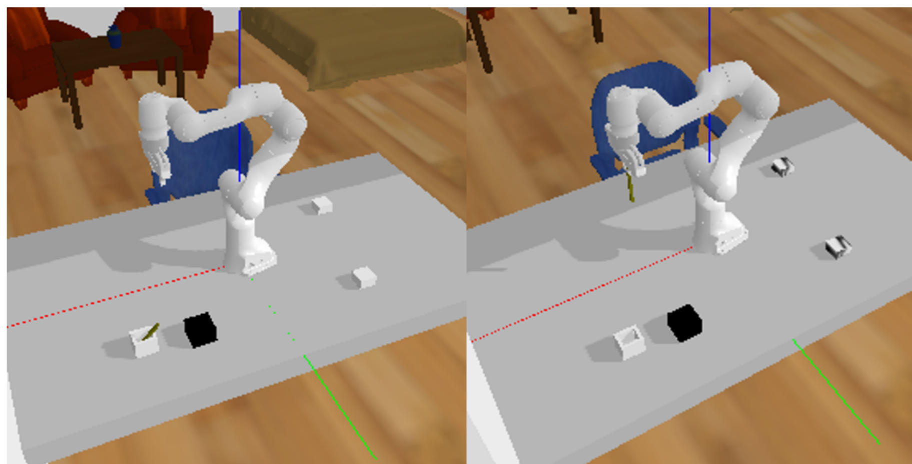


Figure 5.8: A graphical view of painting experiment: Left: The initial state, where the pen is in the pen box, and the pigment and two objects sit on the table. Right: The goal state, where the robot grasps the pen and paints the objects.

5.3 A Puzzle-Solving Task

The puzzle-solving task is to solve the famous 'Towers of Hanoi' puzzle as indicated in Figure 5.9. The Towers of Hanoi environment includes three pegs and a number of disks of different sizes. The goal is to move all the disks from one peg to another while following two rules: only one disk can be moved at a time, and a larger disk can never be placed on top of a smaller one. The predicates and actions for the domain are:

- $\mathbf{P}_{\text{hanoi}}$= {(**smaller** ?obj1 ?obj2), (**top** ?obj), (**is_on** ?upper ?lower), (**holding** ?obj ?gripper)}
- $\mathbf{A}_{\text{hanoi}}$={**stack**, **unstack**}

5.4 Composed Task

Our composed tasks are new tasks created by combining the basic tasks from Section 5.1, Section 5.2, and Section 5.3. Though different, the planning domain of the composed tasks consists of predicates and actions from the same predicate dictionary **P** and action dictionary **A**. Three composed tasks are designed: the cook-and-plate task, the unpack-and-cook task, and the labelling task. We do not design the planning domain for composed tasks because these tasks are only used in the testing phase to verify that our model is able to generate the planning domain for a novel task.

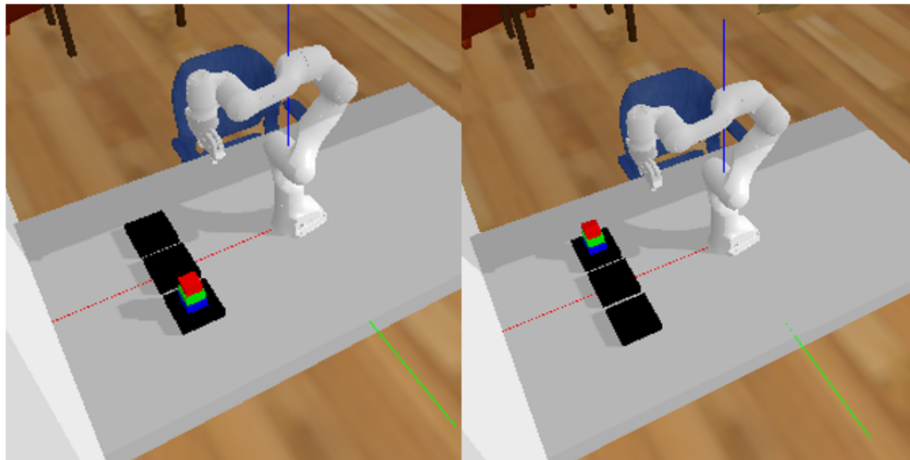


Figure 5.9: A graphical view of the Towers of Hanoi experiment: Left: The initial state, where the tower is on the lower peg. Right: The goal state, where the tower is moved to the upper peg.

The cook-and-plate task involves cooking the raw materials and stacking them together. Composed of the cooking and stacking tasks, as shown in Figure 5.10, the cook-and-plate task is a simplified simulation of the process where a chef washes raw ingredients, grills them, and then arranges them on a plate.

The unpack-and-cook task in our experiment involves unstacking raw materials and cooking them. This task, as shown in Figure 5.11, combines the unstacking and cooking tasks. The unpack-and-cook task is a simplified simulation of how people unpack groceries, wash the raw ingredients, and then grill them.

The last composed task is the labelling task, which requires the robot to unstack piles of objects and then label each item. As shown in Figure 5.12, the labelling task combines the unstacking and painting tasks. This task simulates the process of labelling products in factories.

5.5 Training and Testing Dataset

The training dataset for the predicate and action estimators consists of 30 demonstrations per basic task (no composed task). Each demonstration involves a TAMP task in which the planning environment has two to four objects. During testing, our method is presented with one example trajectory along with a *challenge problem set* containing five unsolved TAMP problems for which the corresponding planning domain needs to be generated. The planning domain is then utilized to solve the test problems.

The test problem set includes both basic and composed tasks. For each task, the test

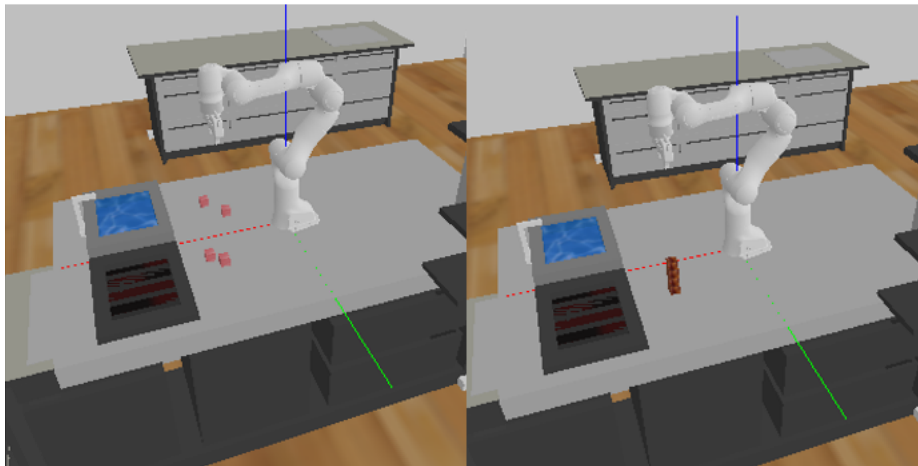


Figure 5.10: A graphical view of the cook-and-plate experiment: Left: The initial state, where the food ingredients sit on the table. Right: The goal state, where the food ingredients are washed, grilled, and stacked vertically.

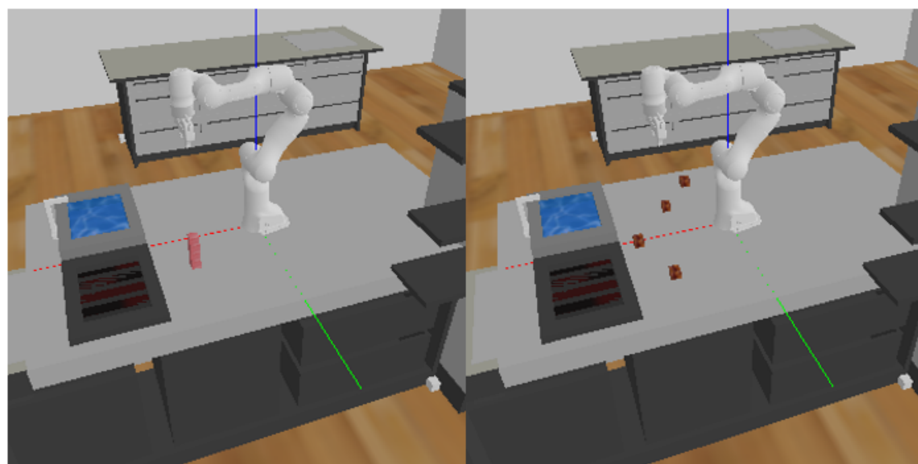


Figure 5.11: A graphical view of the unstack-and-cook experiment: Left: The initial state, where the food ingredients are stacked vertically. Right: The goal state, where the food ingredients are washed, grilled, and placed on the table.

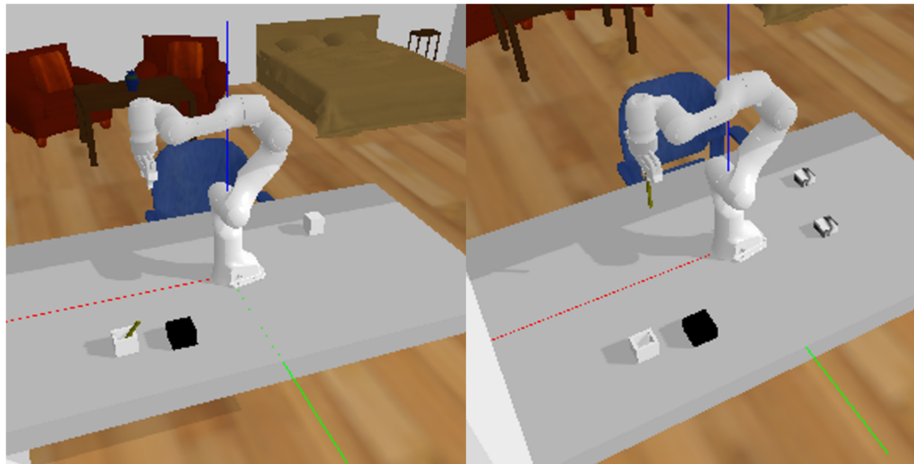


Figure 5.12: A graphical view of labelling experiment: Left: The initial state, where two objects are stacked vertically and the pen is in the pen box. Right: The goal state, where the objects sit on the table and have been painted by the robot.

set includes problems with varying numbers of objects in the planning environment, ranging from two to nine. We generate 10 test problems with randomized object positions for each object count within the specified range. Each planning problem must be solved within a time limit of 30 seconds.

5.6 Baseline

For the baseline, we train a GAT-based behavior-cloning task planner inspired by the work of Lin et al. [47]. The graph-based formulation used in the baseline method is close to ours, with the key difference being that [47] utilizes both predicates and the 3D poses of objects as node features, whereas our approach only uses the predicates. To maintain consistency, we modify their method to use predicates alone as node features.

The inputs to the GAT-based task planning policy are the current logical state $\mathbf{X}_{\text{current}}$ and the goal logical state $\mathbf{X}_{\text{goal}}$, both represented as scene graphs. The policy produces two outputs: a one-hot vector indicating the action a_{next} to execute in the next step and a binary vector where each element corresponds to an object, with 1 indicating a target object and 0 indicating a non-target object. The action and target object are then combined to select the corresponding motion planner $c_{next} \in \mathbf{C}$, which generates the motion plan. As the robot executes the plan, the environment state updates. The updated logical state, along with the goal state, serves as the new input to the task planning policy. This process repeats until the goal is achieved. The baseline method is trained on the same dataset as our method, but trained for and conditioned on each specific task type.

The baseline is trained and evaluated solely on basic tasks. This is because demonstrating the behavior cloning baseline with a single example trajectory for the composed task, as is done in our method, would be trivial. Each planning problem must be solved within 30 seconds.

Chapter 6

Results

In this chapter, we present the experimental results of our method and provide a comparative analysis against the baseline. Section 6.1 presents the experimental planning success rate and compares our method to the behavior cloning baselines. Section 6.2 assesses the computational cost of our method's domain optimization compared to the blind search baseline.

6.1 Planning Success Rate and Generalizability

In Figure 6.1, we compare the performance of our method against the GAT-based behavior-cloning baseline across basic tasks with an increasing number of objects. Each sub-figure presents the change in success rate (y-axis) with the increasing number of objects (x-axis) for each task.

As illustrated in Figure 6.1, our method shows a significant advantage in generalizability, maintaining high success rates even as the number of objects in the tasks increases. Across all the tasks tested, our method consistently outperforms the baseline, maintaining success rates above 90% even as the number of objects increases. In contrast, the baseline method exhibits a sharp decline in success rate as the number of objects increases, showing limited generalizability within a certain threshold for object number. The success rate quickly drops to zero when the number of objects exceeds this threshold. These results indicate that our method has a significant advantage in generalizing to tasks involving more objects.

Moreover, our method exhibits superior performance in generalizing across different task types. It consistently maintains high performance across the nine basic task types shown in the experiment chapter, whereas the baseline's performance varied significantly depending on the task type. The baseline achieved high success rates in sim-

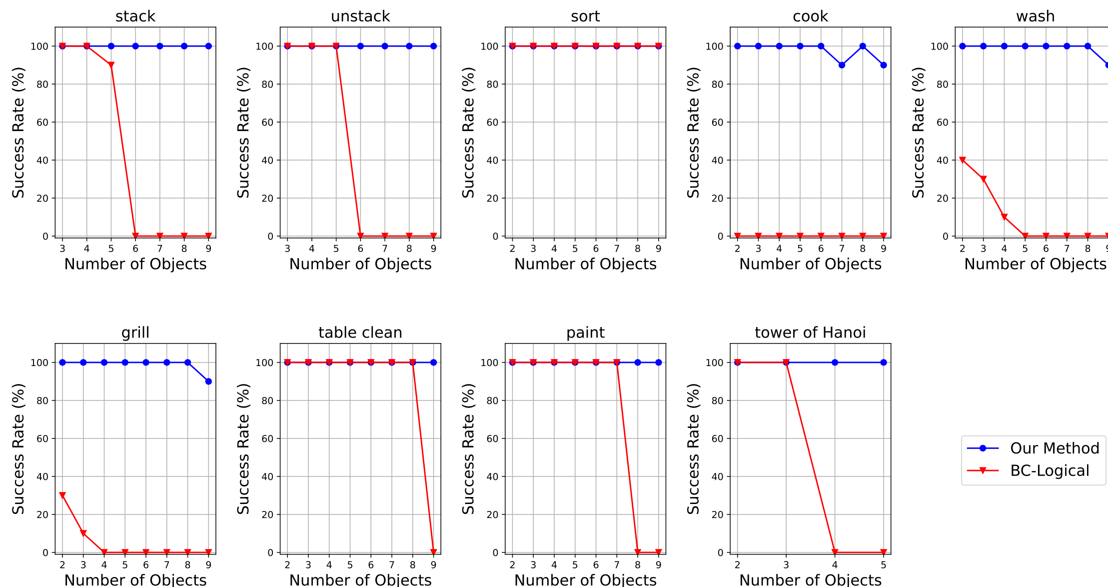


Figure 6.1: Success rates of basic planning tasks for increasing numbers of objects. The planning success rate is the ratio of the number of solved TAMP problems to the total number of TAMP problems attempted. BC-GAT indicates the GAT-based behavior cloning baseline.

ple tasks, such as sorting cubes. However, when faced with more complex tasks, the baseline's performance dropped drastically and sometimes failed to solve the planning problems entirely.

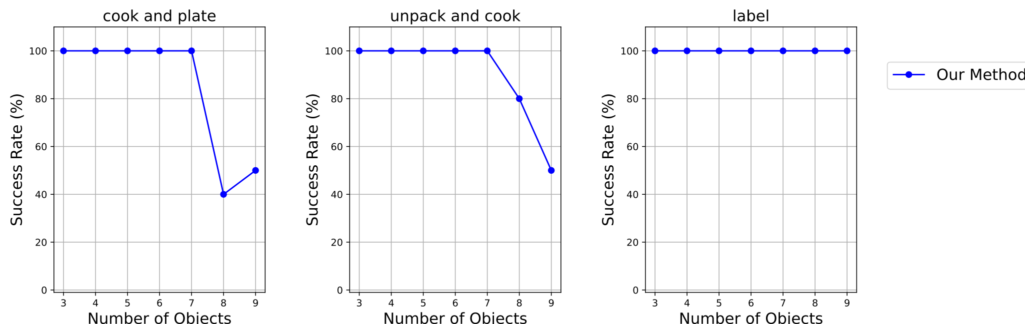


Figure 6.2: Success rates of composed planning tasks for increasing numbers of objects.

In addition to achieving high performance in the basic task types previously seen in the training set, our method is also capable of solving unseen composed tasks in the test problem set. As shown in Figure 6.2, our method consistently maintains a high success rate across all tasks and object counts, demonstrating our system's ability to automati-

cally generate planning domains for new planning tasks in a one-shot or few-shot manner. Compared to the success rate of our method for basic tasks in Figure 6.1, we observe a larger drop in success rate for composed tasks as the number of objects exceeds seven. This is primarily due to the increased difficulty of the composed tasks, which prevents our method from solving the planning problem within the limited time.

6.2 Computational Cost

Another significant advantage of our method is its lower computational cost during the testing phase. Unlike blind search, our approach predicts the most likely planning domain as an initial solution, thus avoiding searching through less likely planning domains at the beginning stage. The computational cost is assessed based on the number of queries made to the motion planner, which is the most time-intensive component of the optimization process. This metric is chosen because it provides a consistent comparison independent of the computer's performance.

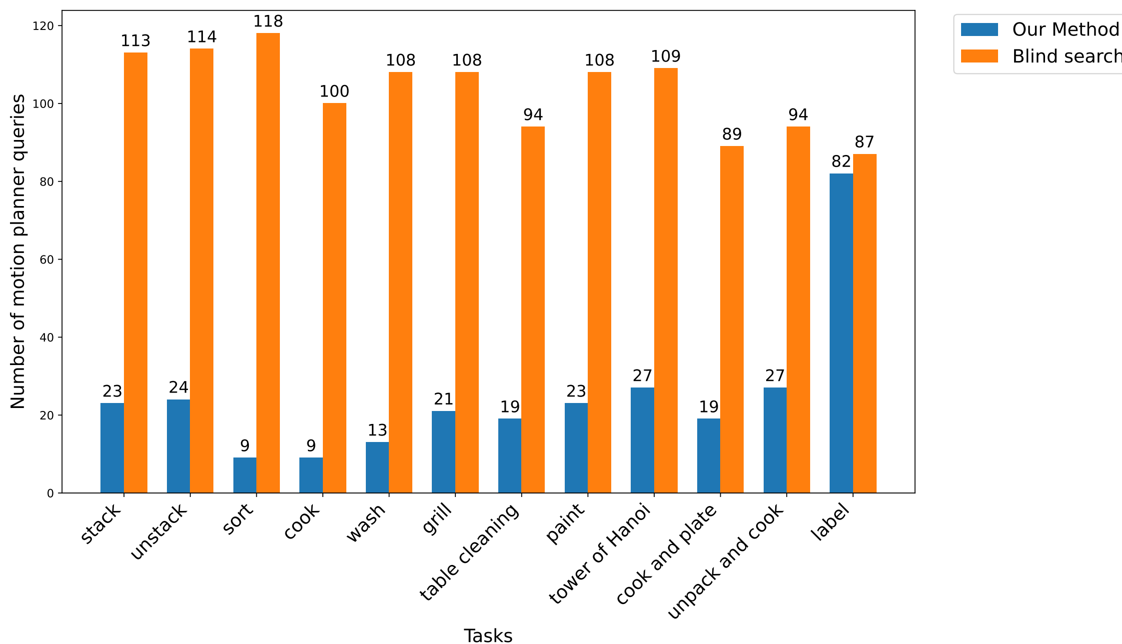


Figure 6.3: The number of queries to motion planners during domain optimization.

As shown in Figure 6.3, we compare the number of queries to motion planners (y-axis) during domain optimization between our method and the blind search baseline. The results show that our method drastically lowers the computational cost in the optimization process. We observe an average reduction of over 74% in the number of motion

planner queries across all test problems. The reduction in computational cost is more apparent for basic task types and attenuates when dealing with composed task types. This phenomenon occurs because the computational cost of our method depends on the accuracy of the most likely domain prediction. When the prediction deviates from the optimal domain, computational costs increase, and the informed search process can become inefficient. The planning domain prediction for composed tasks is more difficult as they are novel to the estimators, leading to increased search efforts. Among the three composed task types, the estimator performs effectively for cook-and-plate tasks and unpack-and-cook tasks, while the domain prediction for the labelling tasks is less accurate. In the domain inference for the labelling task, a critical action, 'Unstack', is predicted to have a very low relevance score. This causes the system to bias other predicates and actions, only including 'Unstack' after expending significant computational resources searching for other predicates and actions.

Overall, our method demonstrates the ability to reduce search effort, even for novel tasks. The reduced search cost is critical for practical applications with limited computational resources, ensuring that our method is efficient and practical for real-world use cases.

Chapter 7

Conclusion

In this thesis, we presented a method to automatically generate planning domains for new task and motion planning problems using deep learning techniques.

Task and motion planning solves complex tasks by integrating a high-level planner, which breaks tasks into sub-tasks, with a low-level planner that computes the corresponding robot trajectories. However, task planning usually requires the manual design of a planning domain, which reduces automation and may introduce errors.

Automatic planning domain generation methods address the issue of dependency on manually-designed domains by searching for predicate and action sets using data from demonstrations. However, they often suffer from poor performance due to the time cost of blind search. In contrast, GNN-based behavior cloning methods effectively accelerate the search process by learning an informative heuristic to help guide the search.

The potential to combine the benefits of heuristic learning for accelerated domain generation led us to develop a method that utilizes GATs, a subclass of GNNs, to prioritize the most likely planning domain for a new task. Our approach, planning domain inference, eliminates the need for manual domain design and significantly accelerates domain generation. To validate the effectiveness of our method, we compared it against a behavior cloning baseline across 870 different tasks (10 tasks evaluated for each combination of task type and object count). Our method achieved an average reduction in domain generation search time of over 74% compared to blind search.

7.1 Contributions

The core contribution of this thesis is the planning domain inference method, which generates the associated planning domain for example robot trajectories in a one-shot

or few-shot manner. We incorporate GAT-based domain estimators with a generate-and-test search process to determine the optimal planning domain. We test the generated domains on various tasks and quantitatively compare the improvements in success rate and reductions in computational cost to existing methods.

There are several important findings from our work. First, our method exhibits better generalization across various environments than direct behavior cloning of the task planner. Our method consistently outperforms behavior cloning, particularly in unseen tasks and with more objects. Second, the computational cost of planning domain generation is significantly reduced compared to blind search. The GAT-based domain estimators provide a reliable initial guess for the most likely planning domain and an efficient heuristic to guide the search. Third, our method is data-efficient when generating new planning domains, since it only requires one or a few demonstrations of a new planning task. In contrast, the behavior cloning baseline requires a large training dataset.

7.2 Future Work

Although the our proposed method improves performance, there are several potential improvements to consider for future work. First, our method assumes the availability of the full predicate and action sets that provide a sufficiently-complete description of the environment state. When the predicates and actions are poorly designed or meaningless, our method is unlikely to produce an effective planning domain. Thus, learning high-quality predicates and actions from the environment could be valuable. Second, the transformation from a continuous state to a logical state in this thesis relies on artificially-designed procedures. Creating such procedures can become increasingly challenging as the number of predicates and actions grows (e.g., managing 1,000 actions and predicates). Learning the transformation directly from sensor data, such as images or 3D point clouds, would make the system more adaptable, more generalizable, and easier to deploy on real robots [56], [57]. Lastly, our method assumes full observability of the environment, which limits its ability to adapt to uncertainty. Past work in TAMP has shown the importance of considering partial observability [64]. Handling partial observability within our system could significantly improve its robustness.

Appendix A

Table of Predicates and Actions

Predicate	Explanation
(**on_table** ?obj ?table)	Indicates whether object is on table.
(**top** ?obj)	Indicate whether one object is on top of a pile.
(**is_on** ?obj1 ?obj2)	Indicates whether object 1 is on object 2.
(**holding** ?obj ?gripper)	Indicates whether one object is held by the robot gripper.
(**cleaned** ?obj)	Indicates whether one object is cleaned.
(**cooked** ?obj)	Indicates whether one object is cooked.
(**in** ?obj ?container)	Indicates whether one object is in a container.
(**containable** ?obj)	Indicates whether one object is a containable.
(**is_pigment** ?obj)	Indicates whether one object is a pigment.
(**is_pen** ?obj)	Indicates whether one object is a pen.
(**colored** ?pen)	Indicates whether a pen has dipped color.
(**painted** ?obj ?pen)	Indicates whether one object is painted by the pen.
(**is_container** ?obj)	Indicates whether one object is a container.
(**closed** ?container ?cover)	Indicates whether a container is closed by the cover.
(**smaller** ?obj1 ?obj2)	Indicates whether object 1 is smaller than object 2.

Table A.1: Predicates used in the experiments described in Chapter 5, along with explanations.

Action	Explanation
Pick	Grasp and lift an object.
Place	Lay an object on the table.
Stack	Put one object on another.
Unstack	Separate piled of objects.
Cook	Heat up an object
Clean	Wash an object.
Open	Uncover the container.
Close	Close the container with its lid.
Dispose	Toss an object in a container.
Dip	Immerse briefly in liquid.
Paint	Apply color to a surface.

Table A.2: Actions used in the experiments described in Chapter 5, along with explanations.

Bibliography

[1] D. González, J. Pérez, V. Milanés, and F. Nashashibi, "A review of motion planning techniques for automated vehicles", *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 4, pp. 1135–1145, 2015.

[2] M. Mohanan and A. Salgoankar, "A survey of robotic motion planning in dynamic environments", *Robotics and Autonomous Systems*, vol. 100, pp. 171–185, 2018.

[3] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles", *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, 2016.

[4] J. T. Schwartz and M. Sharir, "A survey of motion planning and related geometric algorithms", *Artificial Intelligence*, vol. 37, no. 1-3, pp. 157–169, 1988.

[5] C. Zhou, B. Huang, and P. Fränti, "A review of motion planning algorithms for intelligent robots", *Journal of Intelligent Manufacturing*, vol. 33, no. 2, pp. 387–424, 2022.

[6] L. Liu, F. Guo, Z. Zou, and V. G. Duffy, "Application, development and future opportunities of collaborative robots (cobots) in manufacturing: A literature review", *International Journal of Human–Computer Interaction*, vol. 40, no. 4, pp. 915–932, 2024.

[7] C. R. Garrett, R. Chitnis, R. Holladay, *et al.*, "Integrated task and motion planning", *Annual review of control, robotics, and autonomous systems*, vol. 4, no. 1, pp. 265–293, 2021.

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

[9] T. Lozano-Pérez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles", *Communications of the ACM*, vol. 22, no. 10, pp. 560–570, 1979.

[10] M. N. Zafar and J. Mohanta, “Methodology for path planning and optimization of mobile robots: A review”, *Procedia Computer Science*, vol. 133, pp. 141–152, 2018.

[11] W. Jianguo, D. Biao, M. Guijuan, B. Jianwu, and Y. Xuedong, “Path planning of mobile robot based on improving genetic algorithm”, in *Proceedings of the 2011 International Conference on Informatics, Cybernetics, and Computer Engineering (ICCE)*, Springer, 2012, pp. 535–542.

[12] Q. Zhang, J. Ma, and Q. Liu, “Path planning based quadtree representation for mobile robot using hybrid-simulated annealing and ant colony optimization algorithm”, in *Proceedings of the 10th World Congress on Intelligent Control and Automation*, IEEE, 2012, pp. 2537–2542.

[13] S. LaValle, “Rapidly-exploring random trees: A new tool for path planning”, Department of Computer Science, Iowa State University, Tech. Rep., 1998.

[14] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller, “Anytime motion planning using the rrt”, in *2011 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2011, pp. 1478–1483.

[15] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, “Batch informed trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs”, in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2015, pp. 3067–3074.

[16] C. Galindo, J.-A. Fernández-Madrigal, J. González, and A. Saffiotti, “Robot task planning using semantic maps”, *Robotics and Autonomous Systems*, vol. 56, no. 11, pp. 955–966, 2008.

[17] J. Hoffmann, “Ff: The fast-forward planning system”, *AI Magazine*, vol. 22, no. 3, pp. 57–57, 2001.

[18] Y. Zhu, J. Tremblay, S. Birchfield, and Y. Zhu, “Hierarchical planning for long-horizon manipulation with geometric and symbolic scene graphs”, in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021, pp. 6541–6548.

[19] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, “Backward-forward search for manipulation planning”, in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2015, pp. 6366–6373.

[20] D. L. Poole and A. K. Mackworth, *Artificial Intelligence*, 2nd Edition. Cambridge University Press, Sep. 2017.

[21] M. Khodeir, A. Sonwane, R. Hari, and F. Shkurti, "Policy-guided lazy search with feedback for task and motion planning", in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2023, pp. 3743–3749.

[22] R. Ebendt and R. Drechsler, "Weighted a search–unifying view and application", *Artificial Intelligence*, vol. 173, no. 14, pp. 1310–1342, 2009.

[23] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-heuristic a", *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 224–243, 2016.

[24] W. Du, F. Islam, and M. Likhachev, "Multi-resolution a", in *Proceedings of the International Symposium on Combinatorial Search*, vol. 11, 2020, pp. 29–37.

[25] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning.", in *International Joint Conferences on Artificial Intelligence (IJCAI)*, 2015, pp. 1930–1936.

[26] K. Zhang, E. Lucet, J. A. D. Sandretto, S. Kchir, and D. Filliat, "Task and motion planning methods: Applications and limitations", in *International Conference on Informatics in Control, Automation and Robotics*, SCITEPRESS-Science and Technology Publications, 2022, pp. 476–483.

[27] M. Mansouri, F. Pecora, and P. Schüller, "Combining task and motion planning: Challenges and guidelines", *Frontiers in Robotics and AI*, vol. 8, p. 637 888, 2021.

[28] D. McDermott, M. Ghallab, A. E. Howe, *et al.*, "PDDL-the planning domain definition language", 1998. [Online]. Available: https://api.semanticscholar.org/CorpusID:59656859.

[29] C. Talcott, "A computational logic (robert s. boyer and j. strother moore)", *SIAM Review*, vol. 23, no. 2, pp. 264–266, Apr. 1981.

[30] J. A. Robinson, "A machine-oriented logic based on the resolution principle", *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, Jan. 1965.

[31] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving", *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[32] M. Khodeir, B. Agro, and F. Shkurti, "Learning to search in task and motion planning with streams", *IEEE Robotics and Automation Letters*, vol. 8, no. 4, pp. 1983–1990, 2023.

[33] M. Diehl, C. Paxton, and K. Ramirez-Amaro, "Automated generation of robotic planning domains from observations", in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2021, pp. 6732–6738.

[34] T. Silver, R. Chitnis, N. Kumar, *et al.*, "Predicate invention for bilevel planning", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, 2023, pp. 12 120–12 129.

[35] J. Zhou, G. Cui, S. Hu, *et al.*, "Graph neural networks: A review of methods and applications", *AI Open*, vol. 1, pp. 57–81, 2020.

[36] Y. Wu, D. Lian, Y. Xu, L. Wu, and E. Chen, "Graph convolutional networks with markov random field reasoning for social spammer detection", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 1054–1061.

[37] V. Fung, J. Zhang, E. Juarez, and B. G. Sumpter, "Benchmarking graph neural networks for materials chemistry", *NPJ Computational Materials*, vol. 7, no. 1, p. 84, 2021.

[38] T. Silver, R. Chitnis, A. Curtis, J. B. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling, "Planning with learned object importance in large problem instances using graph neural networks", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 11 962–11 971.

[39] C. Sun, C. Li, X. Lin, *et al.*, "Attention-based graph neural networks: A survey", *Artificial Intelligence Review*, vol. 56, no. Suppl 2, pp. 2263–2310, 2023.

[40] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, *et al.*, "Graph attention networks", *Stat Journal*, vol. 1050, no. 20, pp. 10–48 550, 2017.

[41] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model", *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[42] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks", *arXiv preprint arXiv:1609.02907*, 2016.

[43] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs", *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[44] B. Kim and L. Shimanuki, "Learning value functions with relational state representations for guiding task-and-motion planning", in *Conference on Robot Learning*, PMLR, 2020, pp. 955–968.

[45] D. Xu, R. Martín-Martín, D.-A. Huang, Y. Zhu, S. Savarese, and L. F. Fei-Fei, "Regression planning networks", *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[46] M. J. McDonald and D. Hadfield-Menell, "Guided imitation of task and motion planning", in *Conference on Robot Learning*, PMLR, 2022, pp. 630–640.

[47] Y. Lin, A. S. Wang, E. Undersander, and A. Rai, "Efficient and interpretable robot manipulation with graph neural networks", *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 2740–2747, 2022.

[48] Z. Yang, C. R. Garrett, T. Lozano-Pérez, L. Kaelbling, and D. Fox, "Sequence-based plan feasibility prediction for efficient task and motion planning", *arXiv preprint arXiv:2211.01576*, 2022.

[49] M. Dalal, A. Mandlekar, C. Garrett, A. Handa, R. Salakhutdinov, and D. Fox, "Imitating task and motion planning with visuomotor transformers", *arXiv preprint arXiv:2305.16309*, 2023.

[50] 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.

[51] C. Paxton, V. Raman, G. D. Hager, and M. Kobilarov, "Combining neural networks and tree search for task and motion planning in challenging environments", in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 6059–6066.

[52] Y. Jiang, F. Yang, S. Zhang, and P. Stone, "Task-motion planning with reinforcement learning for adaptable mobile service robots", in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2019, pp. 7529–7534.

[53] D. Xu, A. Mandlekar, R. Martín-Martín, Y. Zhu, S. Savarese, and L. Fei-Fei, "Deep affordance foresight: Planning through what can be done in the future", in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021, pp. 6206–6213.

[54] R. Nian, J. Liu, and B. Huang, "A review on reinforcement learning: Introduction and applications in industrial process control", *Computers & Chemical Engineering*, vol. 139, p. 106 886, 2020.

[55] X. Wang, S. Wang, X. Liang, *et al.*, "Deep reinforcement learning: A survey", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 4, pp. 5064–5078, 2022.

[56] K. Kase, C. Paxton, H. Mazhar, T. Ogata, and D. Fox, "Transferable task execution from pixels through deep planning domain learning", in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020, pp. 10 459–10 465.

[57] S. Mukherjee, C. Paxton, A. Mousavian, A. Fishman, M. Likhachev, and D. Fox, "Reactive long horizon task execution via visual skill and precondition models", in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2021, pp. 5717–5724.

[58] N. Kumar, W. McClinton, R. Chitnis, T. Silver, T. Lozano-Pérez, and L. P. Kaelbling, "Learning efficient abstract planning models that choose what to predict", in *Conference on Robot Learning*, PMLR, 2023, pp. 2070–2095.

[59] B. Liu, Y. Zhu, C. Gao, *et al.*, "Libero: Benchmarking knowledge transfer for lifelong robot learning", *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[60] A. R. Mahmood and R. S. Sutton, "Representation search through generate and test", in *Workshops at the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[61] B. Ellenberger, *Pybullet gymperium*, https://github.com/benelot/pybullet-gym, 2018–2019.

[62] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in PyTorch", 2017.

[63] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric", *arXiv preprint arXiv:1903.02428*, 2019.

[64] C. Phiquepal and M. Toussaint, "Combined task and motion planning under partial observability: An optimization-based approach", in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 9000–9006.