Learn2Plan: Learning variable ordering heuristics for scalable planning

Ashwin Misra

CMU-RI-TR-23-14

May 2023

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Robotics.

To my parents, family and friends.

Abstract

Historically, the combinatorics of various search-based approaches to planning, even in single-agent planning contexts, have presented a solid barrier to scalable performance. These approaches quickly become complicated and overwhelming as the number of goals to be achieved (or tasks to be included in the plan) increases. It is because the number of tasks to be included adds more nodes in the search tree and, with more constraints, increases the constraint satisfaction times. However, we believe there is a way forward. We propose that the combinatorics of a planning problem can be overcome by learning an abstract model of the planner's search that utilizes the characteristics of the current state to learn the relative quality of various search decisions extending the plan. A substantial computational speedup can be achieved by learning on example runs of the planner in a given domain, and the learned model can be used as a surrogate for the explicit combinatorial search. We focus specifically on a framework for multi-agent planning and scheduling [14] recently developed to support the continual management of a team of humans, robotic agents, and autonomous control systems. Within this framework, a new task request triggers a search for the best HTN decomposition of the request into constituent actions to be added to the plan, the best assignment of resources to those actions, and the best time slots over the planning horizon. The search initialization generates and evaluates all feasible options concerning a specified objective and selects the best option. The computational cost of performing the search increases proportionally with the number of pre-existing actions in the plan, the number of agents (resources) available to carry out these actions, and the number of alternative decompositions associated with a high-level task request.

This thesis proposes two solutions. First, it presents an efficient learning framework to aid the planner in accelerating the time-consuming search process by learning an abstract representation of the search space. This learning framework consists of an LSTM memory network with an MLP baseline that learns from example runs of the planner. Second, it describes a simulation pipeline for enhancing the real-world physical understanding of the reasoning used by the planner. Both of these solutions form a part of a robust, efficient, and interpretable planning system that can also be used in varied domains outside of a space habitat.

Acknowledgments

I am very grateful to Dr. Stephen F. Smith and Dr. Zachary B. Rubinstein for their continuous guidance and support throughout my Master's Degree. They have always guided me with the utmost patience and enlightened me with various foreign concepts that proved very crucial to this thesis. Their support helped me in exploring new ideas and their experience helped me convert these ideas to methods. I would like to immensely thank them for providing me with this opportunity to work with them on various pressing issues in the field of Robotics. I would like to express my gratitude to Dr. Jean Oh for serving on my committee and providing me with valuable feedback and helping me refine my thesis. I would also like to thank Sha Yi for being on my committee and asking me valuable questions.

I would like to thank Mrs. Varsha Misra and Mr. Anupam Misra for being my two pillars of strength through thick and thin. To my brother, Vihaan Misra for always making me push my boundaries and strive to improve. To Mrs. Smita Shukla and Mr. Gaurav Shukla for being my home away from home and to my grandparents, Mrs. Kusum Shukla and Mr. Kamal Kant Shukla. I would also like to thank Mansi Agarwal for being the most supportive friend and providing me with constant motivation, much-needed help, and critical comments. To Mononito Goswami and Yves Georgy Daoud for being my best friends throughout my Degree, and Viraj Parimi for his help, explanations, and thought-provoking comments. To all other friends that helped me through code reviews, coffee chats and long conversations. Last but not least, to Aaron Berger for code support and git conversion of the simulation pipeline. I would also like to express my gratitude to Barbara Fecich for making my experience at CMU and RI fulfilling and memorable.

Funding

This research was funded in part by the NASA Space Technology Research Institute for Deep Space Habitat Design under grant #80NSSC19K1052, and the CMU Robotics Institute.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Controlling the operations of a spacecraft orbiting a planet millions of miles away from Earth cannot be considered a trivial task. Due to the brevity of these tasks, most of these tasks are manual and human-controlled in near-earth habitats. However, transmission can only happen when a deep-space satellite is visible in deep-space habitats. These transmissions consume much energy to coordinate activities in a space habitat. Apart from energy requirements, a communication delay (NASA reports 5 to 20 minutes for Mars communications) prohibits real-time control of spacecraft activities from the ground, especially during unmanned phases. Hence, autonomy proves helpful in deep space habitats, integrating into repair or task workflows. An essential part of this effort is devoted to integrating planning and scheduling technologies to provide an adequate basis for multi-agent planning. This effort led to Timeline-based planning approaches first proposed by Remote Agents [13]. One of the critical features of the Remote Agent system is its ability to operate on long timelines, spanning days, weeks, or even months. The system uses a hierarchical approach to planning and scheduling, with high-level plans generated by a mission planner and lower-level plans generated by individual subsystems on the spacecraft.

Space habitats are complex. For example, The International Space Station has

16 pressurized modules. Each module has more than ten systems ranging from Environment Control and Life Support System (ECLSS) to Guidance, Control, and Navigation system, each with multiple subsystems. Autonomous deep space habitats add more complexity due to the absence of a crew that adds robustness, safety, and computation checks. The subsystems in such habitats must identify, plan, and execute various tasks in an ordered sequence. For example, a filter change requires detection, cutting off the supply, replacing the filter, and then turning on the supply. Most tasks in space can be divided into a sequence of such sub-tasks that are generally executed in coordination with various agents - robots, rovers, and free flyers. Optimal planning to solve such tasks requires scheduling these agents with system constraints. These constraints ensure that the order of the sub-tasks is met without conflicts. These constraints can be spatial constraints that ensure no collision between agents or temporal constraints that ensure the agents are free to assign tasks.

Historically, such tasks are scheduled on multi-agent systems through a search-based approach. Search-based planning typically works by exploring the space of possible plans or action sequences and evaluating them based on some objective function or cost metric. The search process involves generating candidate plans by considering the possible actions that can be taken in each state and evaluating the resulting states until a plan is found that meets the specified goal criteria. As we increase the number of tasks that must be included (or goals to be achieved), the combinatorics of search-based planning methods grow exponentially and quickly become overwhelmed. The complexity of these approaches poses a barrier to scalable and efficient performance.

We focus specifically on a recent framework for multi-agent planning and scheduling T-HTN [14] that was developed to support the continual management of a team of agents. The framework combines the structure of a Hierarchical Task Network planning with the temporal flexibility of timeline-based planners. This helps maintain a global schedule of activities on a habitat over time and increases robustness to uncertain and unexpected outcomes and tasks. To plan for a new request, a decomposition of the request into constituent actions is carried out. These actions should be added to the evolving plan with the best assignment of resources to those actions over the best-possible time slots over the horizon. Once this search process is initiated, all feasible options are generated and evaluated based on an objective function (e.g., minimizing makespan/setup time), and the best option is selected. The computational cost of performing the search increases proportionally with the number of actions preexisting in the plan, the number of agents (resources) available to carry out these actions, and the number of alternative decompositions associated with specified higher-level task requests.

However, we believe this complexity can be mitigated by learning search heuristics as a variable ordering problem. We hypothesize that an abstract model of the planner's search can be learned that utilizes state descriptions of the planning state to recognize the relative importance of various search decisions for iterative planning. This model can aid the planner as a surrogate for the explicit combinatorial search and substantially speed up the computation time required to generate a partial solution.

1.2 Contributions

We propose Learn2Plan, a novel memory-driven learning framework that supports hierarchical planners to enable multi-agent coordination efficiently. We designed Learn2Plan to learn constraints from spatiotemporal descriptions of the world state and understand the characteristic behavior of hierarchical planners. A physics-based simulation environment and the pipeline are also proposed to aid planning frameworks to be interpretable. With this pipeline, we propose better-capturing black-box planners' physical uncertainties and real-world awareness. Specifically, our contributions are the following:

- A hybrid planning system, as shown in Figure 1.1, combines learning and planning to overcome the traditional complexity of planners to select optimum slots. The learning method learns the decision structure of the planner through examples, takes candidate slots as input, and predicts the best slot ordering to minimize makespan. This system includes:
  - A Multi-layer neural network exploratory comparative baseline

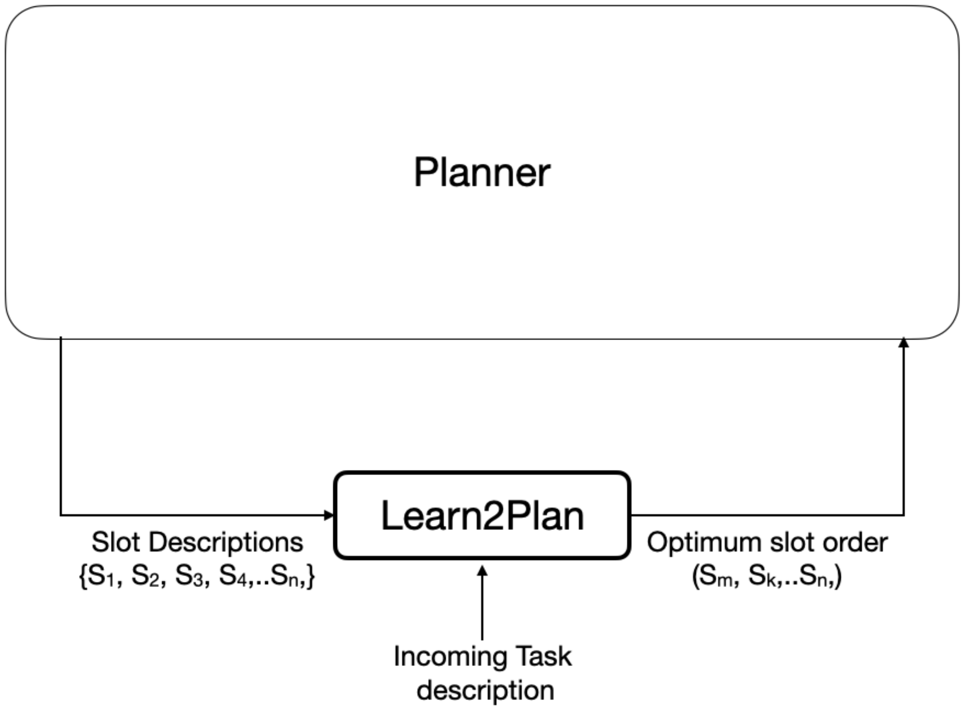

Figure 1.1: Learn2Plan hybrid planning system

  - A Long-Short Term Memory Network is aware of the inherent sequence of requests on a horizon.

- A simulation pipeline to convert plan outputs to domain simulations built on ROS and MuJoCo.

1.3 Organization

The thesis starts with essential concepts in constraint-based scheduling, resource allocation, and search techniques in task-based planning and symbolic planning methods in Chapter 2. Chapter 3 discusses recent literature on optimization techniques and learning frameworks, mainly focused on the search process. Chapter 4 introduces

Learn2Plan, the search modeling technique used, and each framework component in detail. The empirical results and experiments are discussed in Chapter 5, demonstrating the efficient nature of specific multi-agent scenarios. The simulation pipeline and its components are discussed in Chapter 6. Lastly, Chapter 6 is the conclusion of this thesis that provides future research directions based on the current limitations.

Chapter 2

Background

Before understanding the concepts, let us understand the domain environment (as shown in Figure 2.1) used for this thesis. The domain is a rail-mounted system with a rail further divided into several rail blocks. There are two rail-mounted UR5 robots - UR5a and UR5b, with Robotiq grippers. There are custom boxes that can be added depending on the problem size and complexity. The problem class we follow for this thesis is pick-and-place operations that must be coordinated in the system.

2.1 Constraint-based Scheduling and Planning

Scheduling is the process of allocation of a set of resources to a set of activities over time [3]. In a standard scheduling problem, resources are constrained in various ways. These constraints can be pertinent to optimizing the capacity of a resource as well as the order of the activities. An optimal schedule is defined as one that satisfies all such constraints for specific criteria. These criteria can be broad, like the length of the schedule, or specific, like the capacity of a resource.

Constraint-based scheduling allows for a much larger variety of constraints and changes in the model dynamically without changing the algorithms that enable the reuse of the model for other tasks, such as planning. It achieves this by separating the model (the activities, resources, constraints, and objectives) from the solving

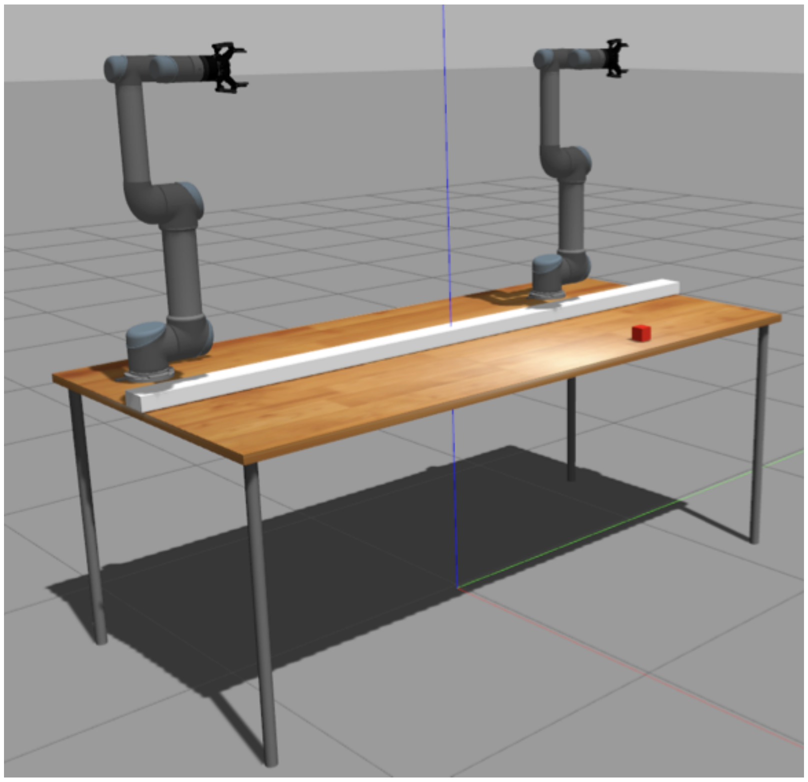

Figure 2.1: Domain setup

algorithms. Combinatorics of the problem is heavily leveraged for solving the problems, leading to improved performance characteristics. Constraint-solving methods like Constraint Propagation, Domain Reduction, and Backtracking search have proved well in various industries.

Constraint-based planning follows the same pattern, i.e., given a set of desired products, a *planner* determines the sequence of tasks (the plan) that will deliver the products. The *scheduler* decides specific resources, operations, and timing to perform the tasks. This information then goes to an executor, ensuring the products are produced [8]. In certain domains, however, they both can be embedded and used as an integrated control system. A general description of the planning problems defines three inputs:

- A description of the initial state and expectation of changes in the world
- A description of the goal of the agent
- A description of the possible actions that can be performed (Domain Theory)

The output of a planner is a feasible sequence of actions referred to as a plan, a partial ordering of actions that will achieve the agent's goal starting from the initial state. In terms of generating multi-agent plans that satisfy a set of complex constraints, there are two broad approaches. Action-based temporal planners (like HTNs [15]) typically achieve success by constraining the plan generation process to forward state-space search (i.e., expanding plans in a time-forward order). On the other hand, timeline-based approaches [13] provide flexibility for actions to be inserted at different time points on an agent's horizon to better optimize the global parameters of the plan.

Constraint-Based Scheduling models are highly configurable regarding new constraints and objectives and are general in representational assumptions. By definition, they are incremental decision-making procedures. Hence, they provide a direct basis for managing solutions change over time and minimizing undue disruption to the current schedule.

2.2 Constraint-based Search

Constraint-based search is a problem-solving technique that involves defining a set of constraints on the possible solutions to a problem and then searching for a solution that satisfies these constraints. The basic idea behind constraint-based search is to reduce the search space by using constraints that rule out certain solutions that are not feasible. This is done by defining a set of rules or constraints that specify the allowable combinations of values for the variables that make up the problem. As the search proceeds, the constraints eliminate possible solutions that violate the rules. The remaining solutions are then checked for feasibility, and the search continues until a satisfactory solution is found or all possible solutions have been exhausted.

Constraint-based search models combine principles of constraint programming and heuristic search in the formulation and solution of scheduling problems [17]. Constraint propagation routines are triggered as scheduling decisions are posted or retracted from a solution representation. Propagation computes the consequences of any change for related scheduling decisions, possibly winnowing (or enlarging) the set of feasible values for other decision variables or detecting a constraint conflict (which signifies an infeasible state). Commitment and retraction strategies/heuristics are two essential components that guide the search process in moving forward (e.g., allocating resources or assigning start and end times to an as yet unassigned mission) or moving backward (e.g., unassigning a previously assigned mission) in the underlying search space.

To better understand the search process, let us understand graph representations of planners. Planning graphs provide a graphical representation of a planning problem and allow for efficient search and inference to find a solution.

A planning graph consists of two levels: the state level and the action level. The state-level represents the possible states that the planning problem can be in at any given time, while the action level represents the possible actions that can be taken to move from one state to another. At the state level, the planning graph captures the current state of the problem, as well as all possible future states that can be reached through the application of actions. Each level of the state level represents a snapshot of the problem's state, with each node representing a set of state variables and their corresponding values. At the action level, the planning graph captures the set of actions that can be applied to change the problem's state. Each node in the action level represents a specific action that can be taken, and the edges between nodes represent the preconditions and effects of those actions. Such planning graphs can also accommodate temporal constraints through Task Constraint Networks [6]. A TCN is a graph-based model that represents the constraints between temporal events. The graph nodes represent the events, and the edges represent the temporal constraints between them. These constraints can be specified regarding the relative ordering or duration of events.

Returning graph is constructed by iteratively adding levels to the state and action levels until a fixed point is reached, where no new information is added. At this point, the graph can search for a plan that achieves the desired goal using techniques such as heuristic search or graph traversal algorithms. Constraint-Based Planning utilizes search nodes representing partial plans with time intervals connected by constraints. These partial plans may not be fully complete, as constraints may not be fully satisfied, and decisions may still need to be made. The planning process aims to transform these partial plans into valid and complete plans. To achieve this, traditional search-based methods attempt various possibilities to complete the partial plans and backtrack when constraints are breached. Constraint reasoning methods, such as propagation and consistency checks, can aid this process.

These possibilities, which are feasible for the constraints, are then compared with an objective function like minimizing the makespan. The best possibility is chosen as the final result of the search.

2.3 T-HTN

T-HTN [14] is a current framework that combines the structural strength of Hierarchical Task Networks with the flexibility of Simple Temporal Networks (Timeline-based planners). It is a planning system that differs from traditional planning methods by explicitly representing the evolution of the current state over time rather than just a collection of facts. This leads to a more extensive search space as states across the planning horizon need to be considered. However, T-HTN takes advantage of problem structure by designating particular objects as resources and assuming that all actions will be allocated and scheduled on specific resources. Each resource has a timeline representing its availability over time, and when an action is scheduled on a resource, an action token is inserted onto the resource's timeline. This token indicates the state aspects of the action that are true during the scheduled interval and assumes that they will only change with the insertion of new actions. The T-HTN system uses an extended version of the Hierarchical Domain Definition Language (HDDL), used in planning problems in the past, to represent the domain and problem inputs with this additional resource structure.

Plan Generation Search Procedure

Let us understand the Plan Generation Search procedure by an example. Building on the setup from Figure 2.1, the example task is to move the red cube from the given location to the table's edge. Considering this incoming task request, the first step is to process this task request to map it to a known task model to create a path decomposition tree for all the potential possible alternatives. As the request maps to a high-level task request, it is decomposed into low-level tasks with an AND/OR tree variant.

All the alternative decompositions are collected through recursively expanding OR nodes and depth-first search. After this, for each possible decomposition, a task network is instantiated for each possible decomposition with an underlying STN composed of tokens corresponding to high-level tasks. Each token constitutes a start and end time for a particular task and consists of potential parameter assignments. This task network forms the fundamental structure, wherein the unassigned variables are grounded and then utilized to search for "slots" on the required resource timelines where the instantiated tokens can be inserted.

Returning to our example, Figure 2.2 shows the generated path decomposition tree. Let us consider the first node and leaf of the grasp item, and try to schedule

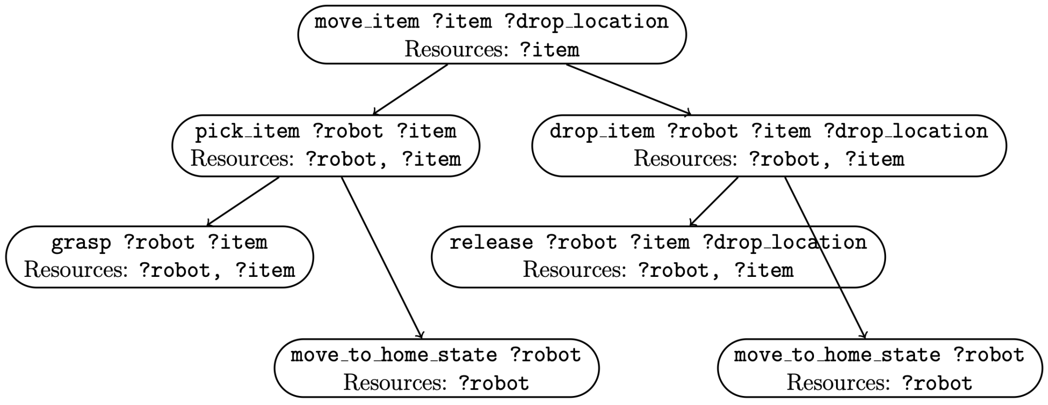

Figure 2.2: Path Decomposition tree

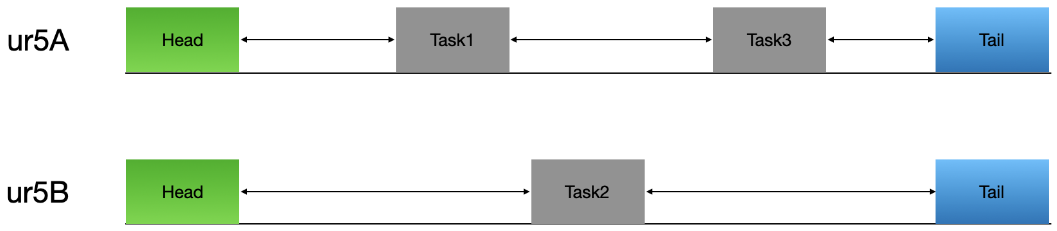

Figure 2.3: Initial resource timelines before an incoming/new request

this particular item. In our domain, we have two robots as our resources. Before the example task request goes in, let us assume that the resource timelines are previously instantiated with tokens. Shown in Figure 2.3 are the timelines for both of the resources. As we can see in these timelines, there are two pre-existing tokens on the timeline of ur5A and one on ur5B that correspond to earlier already scheduled tasks. The scheduled tokens have durations, start and end locations, and other parameters that factor in temporal and sequencing constraints for further scheduling requests.

When our example item goes into the planner, the "slots" are the empty spaces on the timeline. These slots have early and late bounds on start and end times to capture flexibility. Let us assume that the incoming item has a release time (how soon

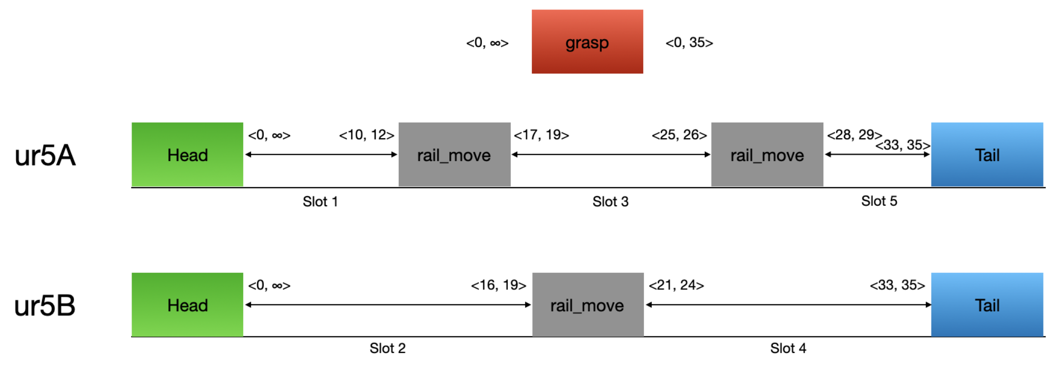

Figure 2.4: Generated slots triggered by an incoming item

the item can start) of 0 units and a due date (how late the request has to end) of 35 units. The incoming item triggers the generation of all slots, known as candidate slots, where the incoming item can be scheduled. Let us parameterize the incoming item by having a duration of 6 units, as shown in Figure 2.4. Assuming all sequencing and spatial constraints are met for all five candidate slots, we can see that the fifth slot does not meet the duration constraint of 6 units, which removes this candidate slot as infeasible. Hence the feasible slots are slots 1,2,3,4, in which all of the constraints are met for scheduling the incoming item.

Moving on to the matching process, the planner checks all of these feasible slots to optimize an objective function, in this case, minimizing the makespan. The planner will do so by scheduling the grasping task on each of these slots one by one and keeping track of the makespan of each schedule. After all the slots have been checked, the planner will choose the slot which accounts for the least makespan. This search process is called backtracking search, as it backtracks after scheduling each slot and then noting the final makespan. The order in which you try these slots can be seen as a variable ordering heuristics problem. As discussed, there are various techniques like "fail-first," in which the slot that is most likely to fail is checked first. However, Backtracking and exhaustive search on all slots is computationally expensive and time-consuming, and the computation grows linearly with the number of slots available.

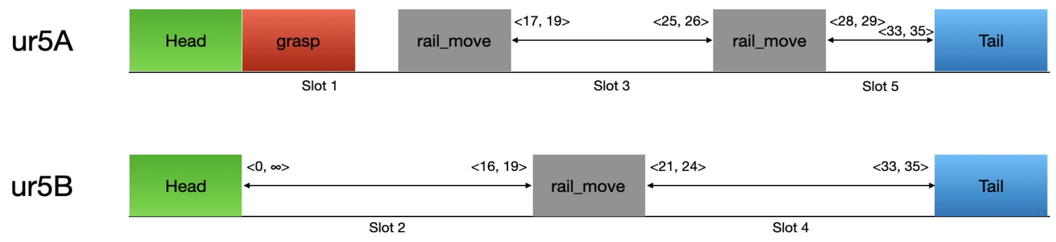

Figure 2.5: (a) Scheduling grasp action on slot 1 corresponds to a total makespan of 32

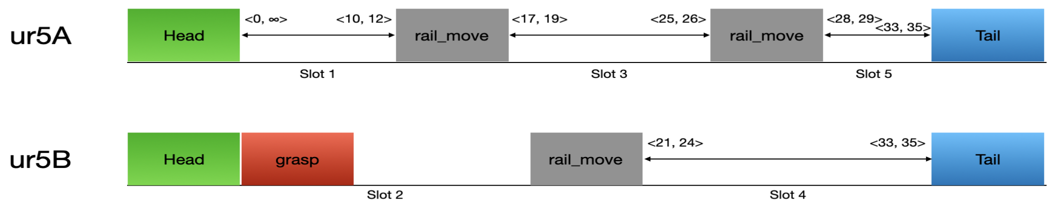

Figure 2.6: (b) Scheduling grasp action on slot 2 corresponds to a total makespan of 27

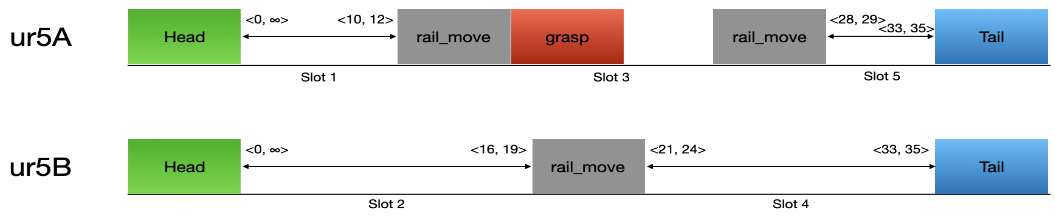

Figure 2.7: (c) Scheduling grasp action on slot 3 corresponds to a total makespan of 28

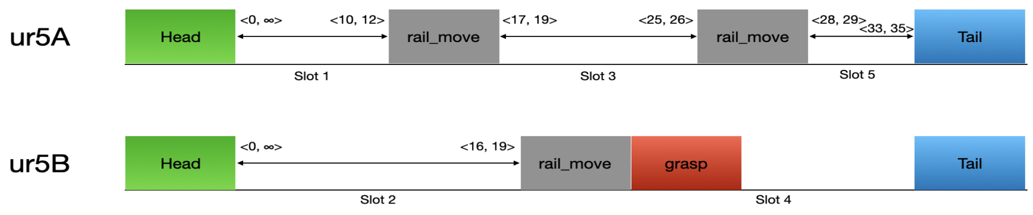

Figure 2.8: (d) Scheduling grasp action on slot 4 corresponds to a total makespan of 26

Figure 2.9: Search procedure, (a) case 1: 2.5, (b) case 2: 2.6, (c) case 3: 2.7 , (d) case 4: 2.8

If we can visualize by our example through 2.9, and cases 1-4, we can see that the least makespan is observed in case 4, i.e., Slot 4, which is seen after trying out slots 1-3. As we optimize for minimum makespan, the optimum slot order in our example is 4 - 2 - 3 - 1. However, to find the best slot, the planner checks all slots from 1 to 4, which uses computation time that ultimately does not lead to the final plan. This area has much potential for improvement and can lead to substantial speedups. In our example, directly going to the best slot can result in a 75% computational speedup.

2.4 Long Short-Term Memory Networks

Long Short-Term Memory (LSTM) [10] is a type of recurrent neural network (RNN) architecture that is designed to overcome the limitations of traditional RNNs, such as the vanishing gradient problem. It has been widely used in various domains, including natural language processing, speech recognition, and time series prediction.

An LSTM cell is shown in figure

The input gate determines which information from the current input should be stored in the memory cell. It is implemented as a sigmoid layer that takes the current input and the previous hidden state as inputs and outputs a number between 0 and 1 for each input dimension. A value of 0 means the information should be discarded, while a value of 1 means the data should be stored in the memory cell.

The forget gate determines which information from the previous hidden state should be discarded. It is also implemented as a sigmoid layer that takes the previous hidden state and the current input as inputs and outputs a number between 0 and 1 for each dimension of the hidden state. A value of 0 means the information should be discarded, while a value of 1 means the data should be retained.

The output gate determines which information from the memory cell should be output. It is implemented as a sigmoid layer that takes the current input and the previous hidden state as inputs and outputs a number between 0 and 1 for each dimension of the memory cell. A value of 0 means the information should not be output, while a value of 1 means the data should be output.

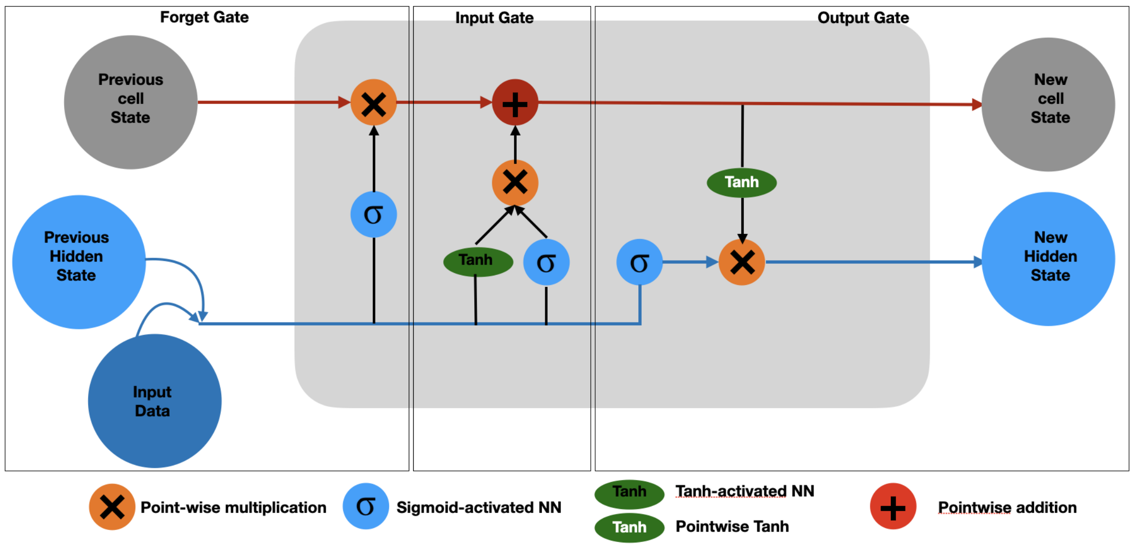

Figure 2.10: LSTM Cell

In addition to the three gates, LSTM has a cell state that runs through the entire sequence and is updated by the input and forget gates. The current input modifies the cell state, and the forget gate decides which information should be discarded from the previous cell state. The input gate then decides which data from the modified cell state should be added to the current cell state.

The final output of LSTM is obtained by multiplying the output gate and the updated cell state and passing the result through a tanh activation function to squash the values between -1 and 1.

Overall, LSTM is a robust architecture that allows the model to selectively forget or remember information, making it well-suited for tasks that require long-term dependencies. It has been shown to outperform traditional RNNs on many benchmark datasets and is considered a standard baseline for sequential data modeling.

We have used these LSTM cells to capture the sequentiality in our data. As we know that the requests are sequential, the slot output for the first request is captured in the hidden state and entered as input for the subsequent request. Hence, we use an LSTM cell per request that cascades the sequentiality from the initial request to the final request.

Chapter 3

Literature Review

3.1 Variable ordering heuristics

Variable ordering heuristics are techniques used to guide the search for solutions to constraint satisfaction problems (CSPs). CSPs are computational problems that aim to find a solution that satisfies constraints. Variable ordering heuristics determine the order in which variables are assigned values during the search process. These heuristics can effectively reduce the branching factor of the search tree and quickly identify variables that are highly constrained or more likely to cause a failure.

The fail-first heuristic is a problem-solving strategy [9] that aims to quickly identify and eliminate the most likely causes of a problem. In the context of constraint satisfaction problems (CSPs), the fail-first heuristic can improve the efficiency of variable ordering heuristics. The basic idea behind the fail-first heuristic in variable ordering is prioritizing variables most likely to fail early in the search process. By selecting variables that are more likely to fail, the search algorithm can quickly prune large portions of the search space and reduce the overall search time.

Dynamic Variable Ordering is investigated in conjunction with tree-search algorithms in the context of binary CSPs [2]. With DVO, the order in which the variables are instantiated during tree search can vary from branch to branch in the search tree, not sequential instantiation. The tree-search algorithm calls the Minimum Remaining

Value procedure when it has descended to the next level and needs to choose a variable to assign at this level. When invoked, the MRV procedure can check each value in the domain of each uninstantiated variable to see if that value is consistent with the current set of assignments. By cycling through all the values in the variable domains, it can count the number of remaining compatible values each variable has and return a variable with the minimum number. DVO heuristics, however, add complexity to the planners and extra overheads. The solver is also very domain-dependent, which reduces the generality of this approach. Another approach is the Degree Heuristic [1]; it attempts to reduce the branching factor on future choices by selecting the variable involved in the most significant number of constraints on other unassigned variables. The authors found that degree heuristics were highly effective in reducing CSP search space, leading to faster convergence and fewer backtracks. In particular, they found that the ”max” degree heuristic, which selects the variable with the most constraints on other unassigned variables, performed the best overall. However, computing the degree of each variable can be computationally expensive, especially if the problem has many variables and constraints. The degree of heuristics' computational complexity can become prohibitive as the problem grows.

Just like these approaches to Variable Ordering Heuristics, Learn2Plan learns a heuristic that is a probability-based priority method that prioritizes slots for a given planner's tree search. This ultimately prunes the search tree by leveraging the planner's experience.

3.2 Learning Frameworks

Recent research has combined learning frameworks in planners to increase backtracking efficiency regarding variable ordering heuristics. Learning variable ordering heuristics for Constraint-based satisfaction Problems has been explored in recent years through Reinforcement Learning, Deep Learning, and Meta-Learning.

Muchg and learning initiated from Adaptive Tree Search [16], in which the search order is adjusted according to heuristic information available during the search for a given problem. With Adaptive Probing, the tree is traversed stochastically, and the model will guide the child's choice at each node learned iteratively by a cost on every leaf. However, this work focuses on solving a given problem instance which limits its approach.

Adaptive Entropy Tree Search (AETS)[19] is a planning and learning algorithm that combines tree search with online learning to solve planning problems efficiently. The algorithm uses an entropy-based heuristic to guide the search and adaptively adjust the exploration-exploitation trade-off during the search process. The basic idea behind AETS is to use a decision tree to represent the search space of possible actions and outcomes, then use an adaptive entropy-based heuristic to guide the search. The entropy heuristic measures the uncertainty of the outcome of each action and is used to balance exploration and exploitation of the search space. The algorithm uses online learning to update the entropy estimates and adaptively adjust the heuristic during the search. While Adaptive Entropy Tree Search (AETS) is a robust planning and learning algorithm, there are some drawbacks to consider. Firstly, the performance of AETS can be sensitive to the choice of parameters, such as the depth of the decision tree and the learning rate for the entropy estimates. Poor choices of these parameters can result in suboptimal or even inefficient search behavior. Secondly, using online learning to update the entropy estimates can be computationally expensive, particularly in large search spaces. This can make AETS less efficient than other planning and learning algorithms in specific domains.

Much research on integrating learning and planning has focused on conventional action-based planners, in which the plan tree traces are used to learn heuristics. This research, as formulated in Learning Search Control [12], aims to speed up the plan generation process or improve the quality of generated plans by learning search control rules. These rules give the planner the knowledge to guide decision-making at choice points. They can include selection rules (recommendations to use an operator in a specific situation), rejection rules (suggestions not to use an operator in a particular situation or avoid a world state), and preference rules (indicating that certain operators are preferable in specific cases). In the article [11], a theoretical basis for formally defining algorithms that learn preconditions for HTN methods is proposed. The algorithm consists of three main steps: preprocessing the plan traces, learning preconditions from the preprocessed plan traces, and using the learned preconditions to improve planning efficiency. In the first step, the plan traces are preprocessed to extract the action sequences performed in each plan trace and the corresponding HTN structure. In the second step, an association rule mining algorithm extracts frequent patterns of operators and preconditions from the preprocessed plan traces. These patterns are used to construct preconditions for the operators in the HTN structure. Finally, in the third step, the learned preconditions improve planning efficiency by guiding the planner to select operators more likely to apply in a given state. The results showed that the learned preconditions outperformed the baselines regarding the number of expanded nodes, search time, and plan quality. Specifically, the learned preconditions reduced the number of expanded nodes by up to 73% and the search time by up to 90% compared to the random baseline. Compared to the CaMeL preconditions, the learned preconditions reduced the number of expanded nodes by up to 33% and the search time by up to 62%. However, this work did not consider the timelines or any time constraints. It also relies on the assumption that the preconditions for operators are independent of each other. This assumption may not hold in some domains, and in such cases, the learned preconditions may not accurately capture the actual dependencies between the preconditions. The authors also need to provide a detailed analysis of the algorithm's computational complexity, and it is unclear how scalable the approach is to larger datasets or more complex domains. The algorithm requires a large set of plan traces to learn the preconditions accurately, which may be computationally expensive or infeasible for some problems. We believe that even higher performance can be achieved with better uncertainty accommodation by considering time bounds.

Another approach to learning from plan traces was to incrementally learn control knowledge to improve the efficiency and quality of planning in domains with large and complex search spaces [4]. The algorithm generates a search tree and analyzes it to identify areas where the search is inefficient. Relevant sub-trees are identified, and control knowledge is defined in each sub-tree selection, rejection, and preference rules. Control knowledge is incrementally learned during the planning process and used to guide the search through relevant sub-trees and to avoid revisiting already explored areas of the search space. By doing so, the algorithm can significantly improve the efficiency and quality of planning in complex domains. The approach relies on analyzing the search space of a planning problem, which can be computationally expensive, especially for complex domains. Additionally, the approach requires domain-specific knowledge to identify the relevant parts of the search space and to define the control knowledge to be learned.

There has been an effort to discover new variable ordering heuristics to boost the efficiency of planners. Deep Reinforcement Learning [18] has been used to automatically find new variable ordering heuristics for a given class of CSP instances. To accommodate different sizes and complex constraints and variables, they represent CSP as a Graph Neural Network. The authors use Double Deep Q-Network to minimize the search effort of reaching the leaf nodes. A GNN-based scheme to describe the internal search states, based on which a deep Q network is designed to output the Q-value of each candidate variable end-to-end from raw state features of variables and constraints. Variables are given updated embeddings based on iterations of embedding aggregations. Experimental results on random CSP instances show that the learned policies outperform classical hand-crafted heuristics with smaller search trees (up to 10.36 reduction) on small and medium-sized instances. Computationally, the DRL approach is complex and does not significantly reduce computation times.

Another paper [20] presents a reinforcement learning approach for project scheduling under resource constraints, where the objective is to minimize the total project duration subject to resource availability constraints. The proposed method uses a variant of Q-learning called S-learning, which incorporates resource constraints into the state and action spaces of the learning agent. The state space includes information on the project schedule, resource utilization, and remaining work, while the action space includes the set of possible task assignments for a given time. The reward function is based on the total project duration, with penalties for resource over-utilization and incomplete tasks. The authors demonstrate the effectiveness of their approach on a set of project scheduling instances with different resource constraints and problem sizes. They compare the S-learning approach with a baseline heuristic and show that S-learning achieves better solutions in less time for most instances, significantly as the problem size increases. The paper concludes that reinforcement learning is a promising approach for project scheduling under resource constraints and suggests several directions for future research.

However, RL-based scheduling techniques require a large amount of data to train the agent and can be computationally expensive. In addition, the performance of RL algorithms can be sensitive to the choice of hyperparameters, reward function, and exploration strategy.

The paper[7] presents a method for improving the efficiency of constraint optimization algorithms by learning and adapting variable ordering heuristics. The authors propose an online learning framework that uses past performance data to update the heuristics in real time during the search process. The approach is based on a variant of the multi-armed bandit problem, where the learner must balance exploration and exploitation to find the best ordering heuristic. The authors evaluate their approach to several benchmark problems from the constraint optimization literature and show that their method consistently outperforms several state-of-the-art algorithms. They also analyze the impact of different exploration strategies and show that their approach is robust to problem instances and search algorithm changes. For example, in some instances of the Minimum Vertex Cover problem, their approach improved by up to 30 percent over the best-performing baseline algorithm. In instances of the Maximum Independent Set problem, their approach enhanced by up to 13 percent over the best-performing baseline algorithm. Overall, the paper provides evidence that the approach can substantially improve the efficiency of constraint optimization algorithms. A major drawback is that the approach requires significant computational resources to maintain and update the heuristics in real-time. This may be a challenge in cases where computational resources are limited, or the search process is constrained by time or memory limitations.

Chapter 4

Methodology

The main goal of this thesis is to develop a hybrid planning system with an integrated learning technique that can reason for task assignments for coordination and management of activities on board space habitats while being scalable, efficient, and robust. However, the application of this system goes well beyond deep-space habitats; it can work with different planners in different domains as it is highly domain-independent. At the core of this system is an LSTM-based learning architecture- Learn2Plan, which models the task planning process as a sequence-to-sequence model. Building on the search process explained in Chapter 4, this framework provides a substantial computational speedup to the search process of hierarchical planners by providing the optimum slot order for an incoming task. Learn2Plan learns the slot matching process from the spatial description of the world state on data obtained from example data from the planner. Suppose the slot and incoming request descriptions (including time bounds and locations) collected for a wide range of problem instances with high variance are combined with the makespan and the selected slot for a request. This data can be used to learn a surrogate model of the planner's decision-making structure. This high-accuracy framework aids the planner in catalyzing the search by suggesting the best slot iteratively to the planner from the optimum slot order as the output.

4.0.1 Problem Instances

We consider pick-and-place operations as tasks for this given domain. This is because these operations encompass a wide range of activities, including cargo stowage, object retrieval, and manipulation of parts on board. Each request corresponds to picking a box up and keeping it in the desired position. The number of requests (i.e., boxes) with corresponding locations and the number of divisions of the rail blocks are specified in the problem file. Each move action by the robot is constituted by a constant time interval specified in the problem file. For this thesis and to establish a proof of concept and build on our domain setup of rail-mounted robots and rail block divisions, we limit training to problems with five requests and five rail block divisions, with only two slots per request investigated.

4.1 World state

Problem instances are converted to problem files, then used as planner input to run examples. As the planner is run, features are collected for each feasible slot generated during the planner search. Each such collected slot provides data for each request as an example. As we established before, the combinatorics of a multi-agent scheduling problem is the primary source of complexity increase; we try to describe the world state only through spatiotemporal features. These features range from the spatial start and end places on the domain, with temporal bounds of each slot. An incoming task is also described with the same parameters to ensure uniformity. The features are a surrogate for constraints, both temporal and spatial. Spatial constraints are captured by the start and end location of the slot and the temporal constraints by the bounds of the start and end times. Similarly, we have similar features for an incoming task to understand how the constraints play in selecting a slot. All of these features are summarized in Table 4.1

Our training data follows this structure; every request has multiple candidate slots. The ground truth results from the planner after it checks all the slots and

Item	Spatial Features	Temporal Features
Slot	Start location End location	Early Start Time, Late Start Time Early Finish Time, Late Finish Time
Incoming Task	Pick-up location Drop-off location	Duration

Table 4.1: Slot and Task Description

outputs the chosen slot and makespan for one request.

$$S_1, S_2, S_3, ..., S_n \Rightarrow S_{optimum}, makespan_{optimum} \tag{4.1}$$

where S_i is a feasible slot, and n is the number of feasible slots per request.

4.2 Data

To generate training and testing data, we run T-HTN on a set of representative problems. We developed an automated script to query the planner to create problem files in the Extended Hierarchical Domain Definition Language supported by T-HTN.

Through the script, we develop 100,000 problem files for the planner. These problem files consist of 5 rail blocks and five requests, and the planner outputs the state descriptions and the ground truths as discussed. We have focused on high variance in our data set to capture the different nature of tasks with different spatial and temporal constraints to prepare a holistic dataset for learning. For every request, we consider two slots.

Dataset $S = \{\boldsymbol{z}_i\}_{i=1}^n = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$ is sampled from a distribution $\mathcal{D}$ over a domain $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$. This distribution consists of 100,000 tasks with descriptions defined in Table 4.1, with all slots per request and corresponding makespan.

$\mathcal{X}$ is the instance domain (a set), $\mathcal{Y}$ is the label domain (a set), and $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$ is the example domain (a set).

$$\begin{aligned} \boldsymbol{x}_i &= \{RequestID, description_{state}, description_{task}, slots, makespan\} \\ RequestID &\in requestA, requestB, requestC, requestD, requestE \\ slots &= \text{All feasible slots per RequestID} \end{aligned}$$

Usually, $\mathcal{X}$ is a subset of $\mathbb{R}^d$ and $\mathcal{Y}$ is a subset of $\mathbb{R}^{d_o}$, where d is the input dimension (10), d_o is the output dimension (1). The label for our data is the selected slot and the corresponding makespan.

$$\boldsymbol{y}_i = \{SelectedSlot, makespan\}$$

$n = 100,000$ is the number of samples. Without specification, S and n are for the training set, where each sample contains the sequence of all requests required to complete a task. We use a 70/30 split between training and testing. We do not keep a cross-validation set because the data is well-curated and structured by the planner.

Loss function

A loss function, denoted by $\ell : \mathcal{H} \times \mathcal{Z} \to \mathbb{R}_+ := [0, +\infty)$, measures the difference between a predicted label and a true label. In our case, we take one primary loss and an auxiliary loss. The primary loss is a standard classification loss, a stable request-wise Binary Cross Entropy with Logits Loss. The auxiliary loss is a Root-Mean Squared Error for our linear variable of makespan.

$$\begin{aligned}
\ell(f_{\boldsymbol{\theta}}, \boldsymbol{z}) &= \ell_{slot} + \ell_{makespan} \\
\ell_{slot} = L_{BCE} &= -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i) \\
\ell_{makespan} = RMSE &= \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}
\end{aligned}$$

where ℓ_{slot} is the classification loss for the predicted slot, and $\ell_{makespan}$ is a regression loss for the predicted makespan. $\hat{y}_i$ is the ground truth, which is the slot chosen by the planner and the makespan value computed by the planner, y_i is the prediction by our learner. We predict the makespan but do not report it because it is not a part of the variable ordering process for slot matching. If we select the optimum slot for the planner to validate, it will automatically calculate the makespan for that slot and action. However, we consider it in the loss function because that is the objective function in our case and gives useful information about guiding the search process.

Empirical risk or training loss for a set $S = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{n}$ is denoted by $L_S(\boldsymbol{\theta})$ or $L_n(\boldsymbol{\theta})$ or $R_n(\boldsymbol{\theta})$ or $R_S(\boldsymbol{\theta})$,

$$L_S(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i). \tag{4.2}$$

The population risk or expected loss is denoted by $L_{\mathcal{D}}(\boldsymbol{\theta})$ or $R_{\mathcal{D}}(\boldsymbol{\theta})$

$$L_{\mathcal{D}}(\boldsymbol{\theta}) = \mathbb{E}_{\mathcal{D}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}), \tag{4.3}$$

where $\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{y})$ follows the distribution $\mathcal{D}$.

4.2.1 Multi-Layer Perceptron baseline

The purpose of developing an MLP (multi-layer perceptron) is to provide a proof-of-concept and validate learning on the data set for simplicity and a much lower computation cost. The model is used to establish a starting point for evaluating the performance of our proposed complex model. An MLP is a type of neural network with multiple layers of nodes, where each node in a layer is connected to every node in the previous layer. The connections between nodes have associated weights that are adjusted during training.

The MLP baseline is trained on the previously explained data set, and its performance is evaluated on the same metrics for our proposed LSTM model. The primary goal of using an MLP baseline is to demonstrate whether the proposed model improves upon the performance of a simpler model that could be easily implemented with fewer computational resources.

Architecture

As seen in Figure 4.1, The neural network consists of hundred input dimensions and two hidden layers with fifty and twenty-five nodes, respectively. We pass ten tasks (batch size) for each run through this network with two requests each. Hence there are five nodes, one for each request. The input dimension is calculated by multiplying ten tasks with five requests, generating two slots, resulting in a vector size of hundred. The hidden layer nodes are decided on an ad-hoc basis by trading off accuracy with computation. The final layer has one node for the best slot per request, a sigmoid layer that also gives the probability output for all slots per request and pushes the best slot per request iteratively to the planner. All layers are linear with ReLU activation functions. The total input vector size is 100, and the output vector size is 50, generating the best slot per request for ten tasks.

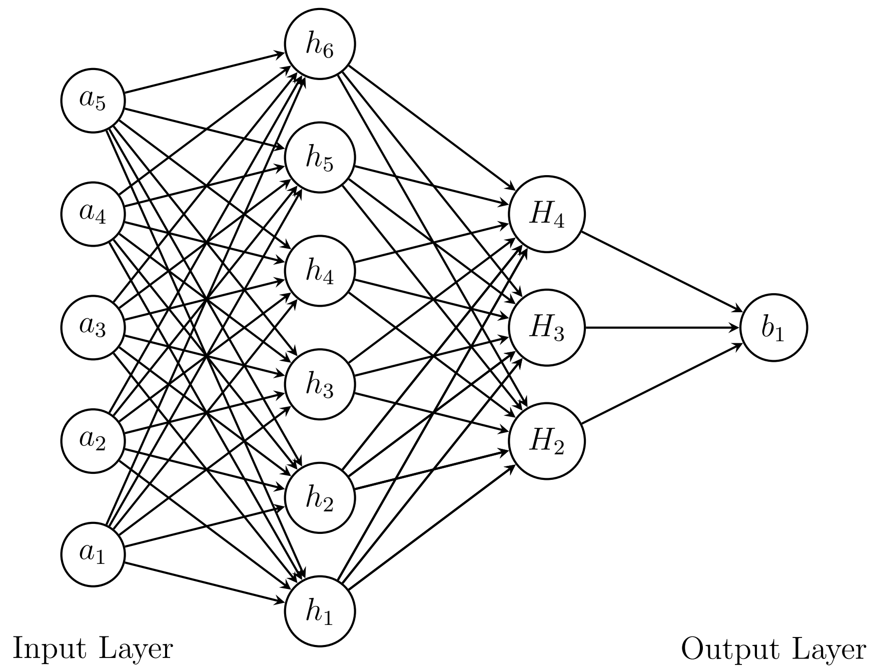

Figure 4.1: MLP baseline architecture with slots per request as input and the best slot as the output

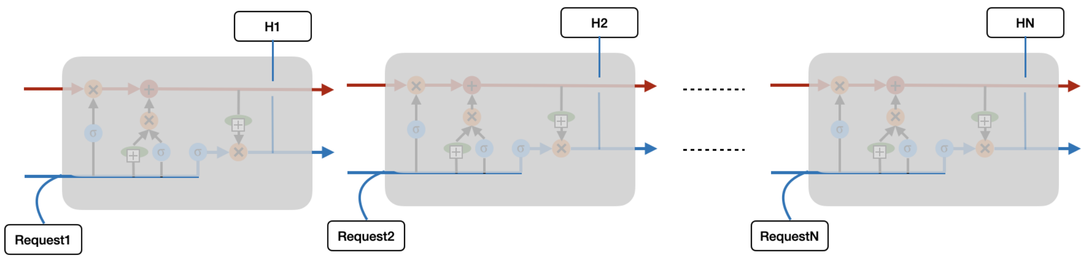

Figure 4.2: Data Modelling through LSTM cells. As we can see that the sequence2sequence structure understands the sequence in requests and keeps track of it with the hidden state H

4.2.2 LSTM

Similar to the input and output modeling of the MLP baseline, the LSTM model is modeled with the same data and loss function. LSTMs have a chain-like structure, and the repeating module is an LSTM cell, as described earlier. Every cell corresponds to one request and stores a hidden state at every request for the whole task, as shown in figure 4.2. As we increase complexity, we have to add zero padded cells for variable input sizes, but our formulation will still be one cell per request.

Architecture

The encoder-decoder layers in the LSTM cell have five layers each, as in the internal architecture of MLP. The input and output description, sizes, and batch size remain the same to ensure uniform metrics. After the LSTM layer, there is a downstream linear neural network layer with 64 nodes in the hidden layer and ReLU activation functions.

Algorithm 1 Learn2Plan

```
1:  procedure SlotOrderHeuristic($T, S$)
2:      $S \leftarrow$ Initialize an empty slot list
3:      for all $t \in T$ do
4:          $D_a \leftarrow S_a, t$
5:          $S_{order} = \text{LSTM}(D_a)$
6:          if $S_{order} \neq \varnothing$ then
7:              for all $s \in S_{order}$ do
8:                  if $s$ is validated by the planner and consistent with constraints then
9:                      $s \leftarrow$ task t
10:                     Add $s$ to $TaskNetwork$
11:                 end if
12:             end for
13:         end if
14:     end for
15:     return $S$
16: end procedure
```

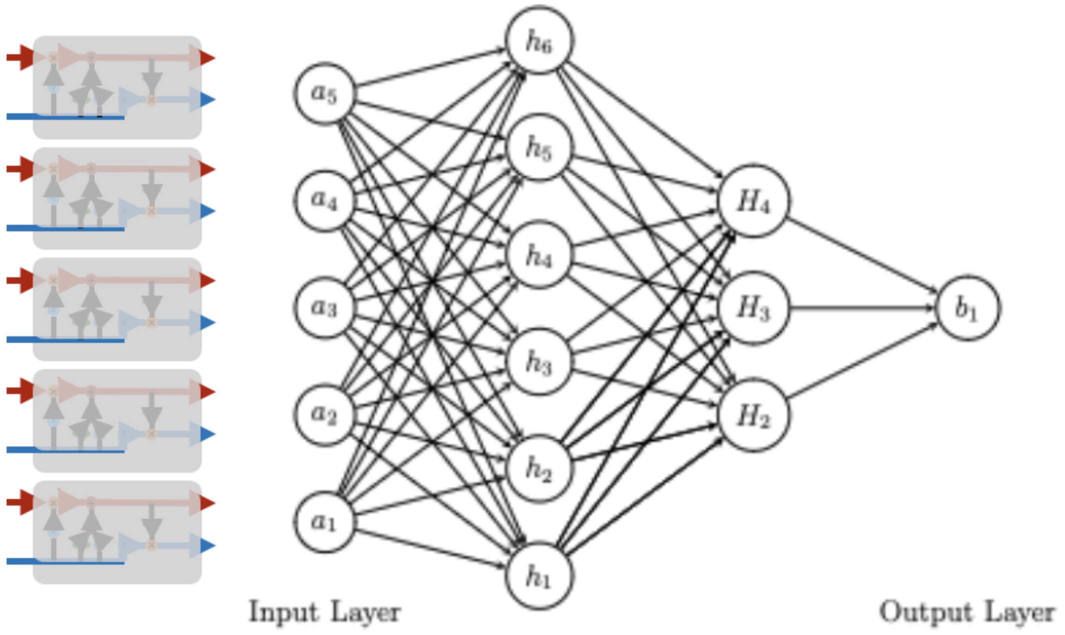

Figure 4.3: LSTM and Downstream Neural Network

Chapter 5

Experiments

For evaluation, a data set of five requests and five blocks is used, with randomly sampled data consisting of ten rail blocks to test generalization. We use an Apple Mac M1 CPU with 16GB of RAM for computation, and all the times quoted from here on are from the same machine.

Setup

We will use the rail-mounted robot domain we defined earlier for our experiments. Every request is a pick-and-place operation, picking a box up from a rail block and placing it on another rail block.

5.1 MLP

We train the MLP baseline model for 300 epochs until we find convergence, as seen in 5.1. We can see expected trends in the loss function as the test loss converges to zero with the training loss, which means that the model is learning well. We calculate accuracy as slot accuracy. It measures how often the MLP model predicts the same slot as the T-HTN planner. We can see that even without sequential awareness, the MLP baseline achieves reasonable accuracy rates.

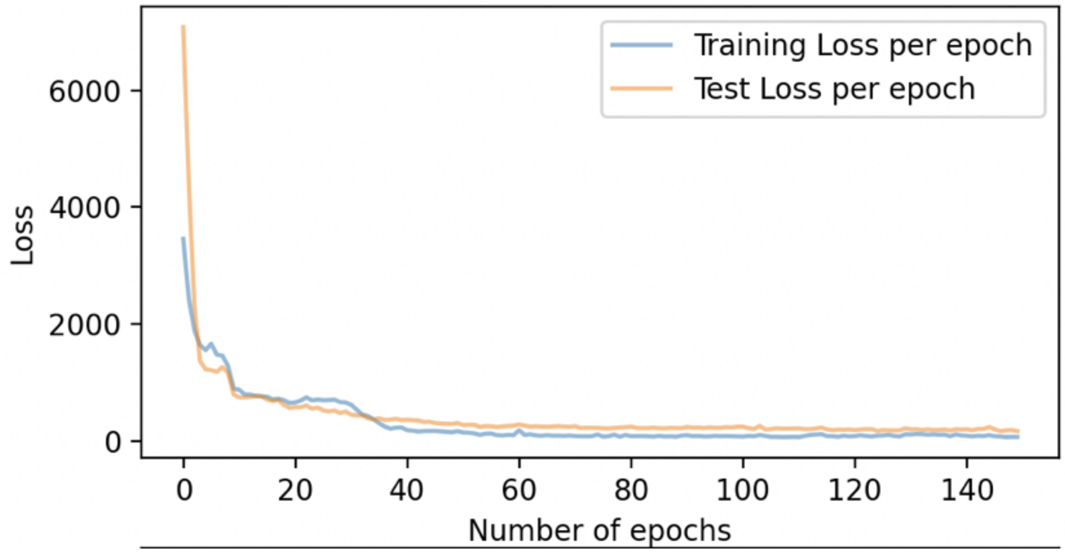

Figure 5.1: Loss curve for the train and test

Accuracy

We get a good test accuracy value around 100 epochs, with high accuracy rates achieved in the early stages of the training as shown by Figure 5.2. This means that the model is very accurate and, around 85% of the time, predicts the same output as the planner.

ROC-Curve and Precision-Recall

To measure the model's performance, we also calculate other metrics as shown in Figure 5.3. ROC (Receiver Operating Characteristic) curve, AUC (Area Under the ROC Curve), and F1 score are standard evaluation metrics used in machine learning models to measure the model's ability to distinguish between positive and negative classes for classification problems.

The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) at different classification thresholds. TPR is the proportion of actual positive cases correctly identified as positive, while FPR is the proportion of actual negative

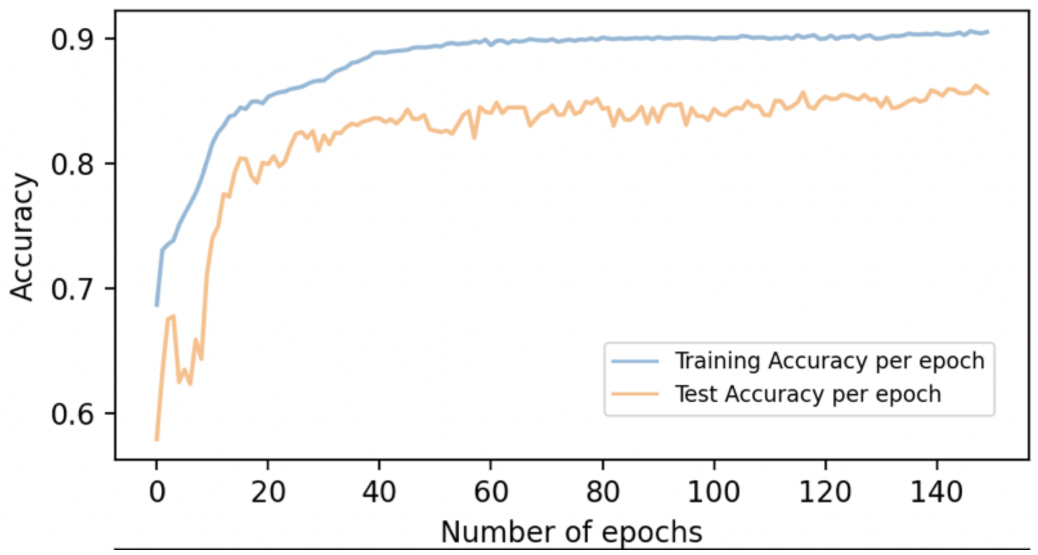

Figure 5.2: Accuracy curve for the training and testing of the MLP baseline

cases incorrectly classified as positive. The ROC curve visualizes how well a model can distinguish between the two classes by showing how the TPR and FPR change as the classification threshold varies. The AUC score is a measure of the area under the ROC curve. It ranges from 0 to 1, with higher values indicating better performance. An AUC score of 1 represents a perfect classifier, while a score of 0.5 represents a random classifier. We see an AUC score of 0.956, as shown in Table 5.3, which signifies that the model can differentiate between classes.

The Precision-Recall curve is another evaluation metric used in machine learning models, particularly in binary classification problems. It is a plot of precision against recall at different classification thresholds. Precision is the proportion of true positive predictions (correctly identified positive cases) among all positive predictions. It measures how many of the positive predictions made by the model are correct. Conversely, recall is the proportion of true positive predictions among all positive cases. It measures how many actual positive cases are correctly identified by the model. The Precision-Recall curve visualizes the trade-off between precision and recalls at different classification thresholds. As the threshold increases, the model

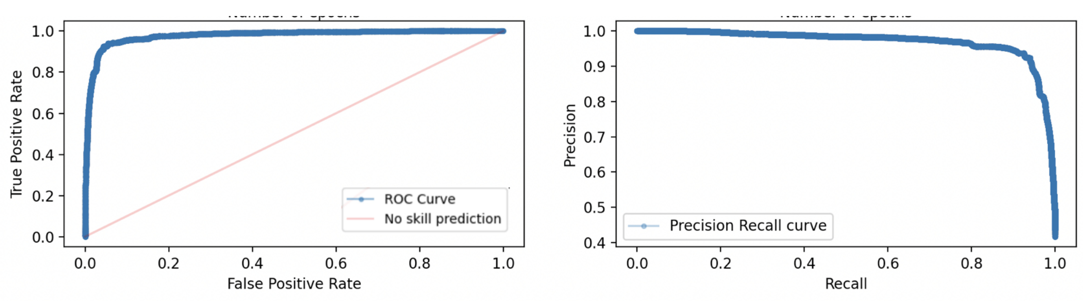

Figure 5.3: ROC; Precision-Recall Curve for the MLP baseline

Table 5.1: Evaluation Metrics

Model	**Accuracy**	**F1-Score**	**AUC-Score**
MLP baseline	86.3%	0.801	0.956

becomes more conservative in its predictions, leading to higher precision but lower recall. Conversely, as the threshold decreases, the model becomes more liberal in its predictions, leading to higher recall but lower precision. A good classifier has both high precision and high recall. However, in practice, there is often a trade-off between precision and recall, and the appropriate balance depends on the problem. This can also be seen in the curve from Figure 5.3, that the model reaches high precision and high recall at different thresholds.

All metrics show a positive trend, with a good F1 and AUC score. As shown in the Table 5.1:

Feature Importance

To understand the working behind the MLP and the neural networks, we plot the correlation matrices on the data set with the predicted slot probabilities in Figure5.4. This would give us the importance of our spatial and temporal descriptions in the network and how much they contribute to the final slot selection.

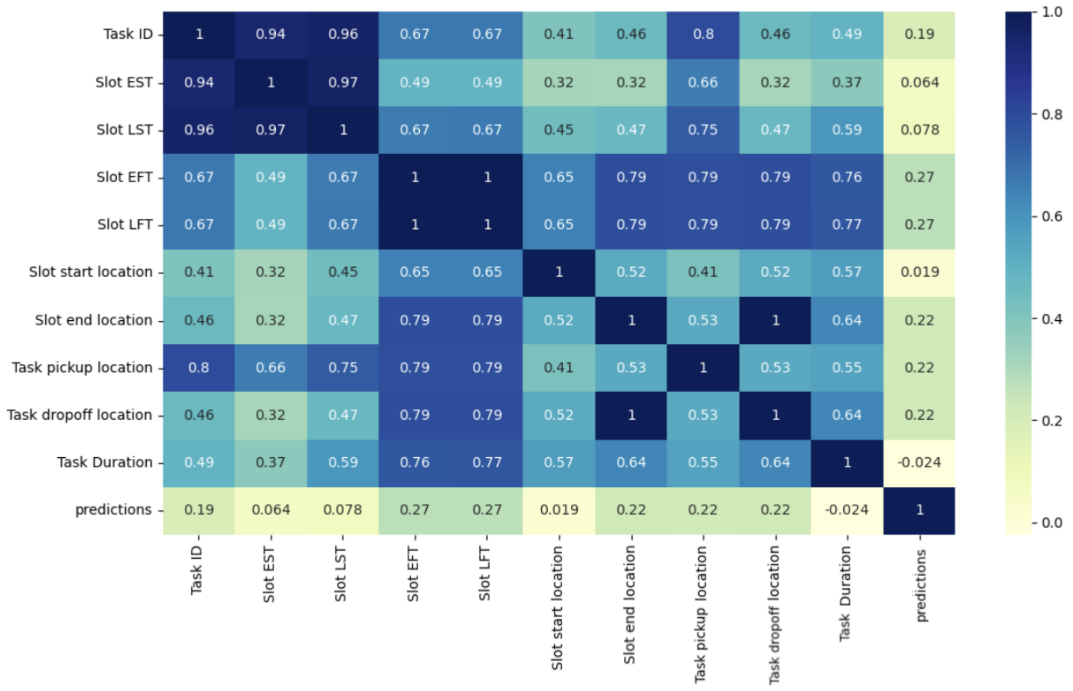

Figure 5.4: Correlation Matrix for MLP baseline with the predicted probabilities

As we can see from the correlation matrices, the spatial and temporal constraints, such as the pickup location, drop location, and the time bounds of the incoming item and the slots, have a higher correlation. There is a drop in start times because the first two requests generate slots from the start with a relaxed constraint. From this figure's last column, we can imply that the model gives almost equal weights to all features and does not overfit to any; this signifies that the learning problem is valid without the learner single-handedly learning the relation/heuristic between two vital components.

5.2 LSTM

We train the LSTM model for 1200 epochs until we find convergence. As shown in figure 5.5, we get expected trends in the loss function. It converges around 500 epochs

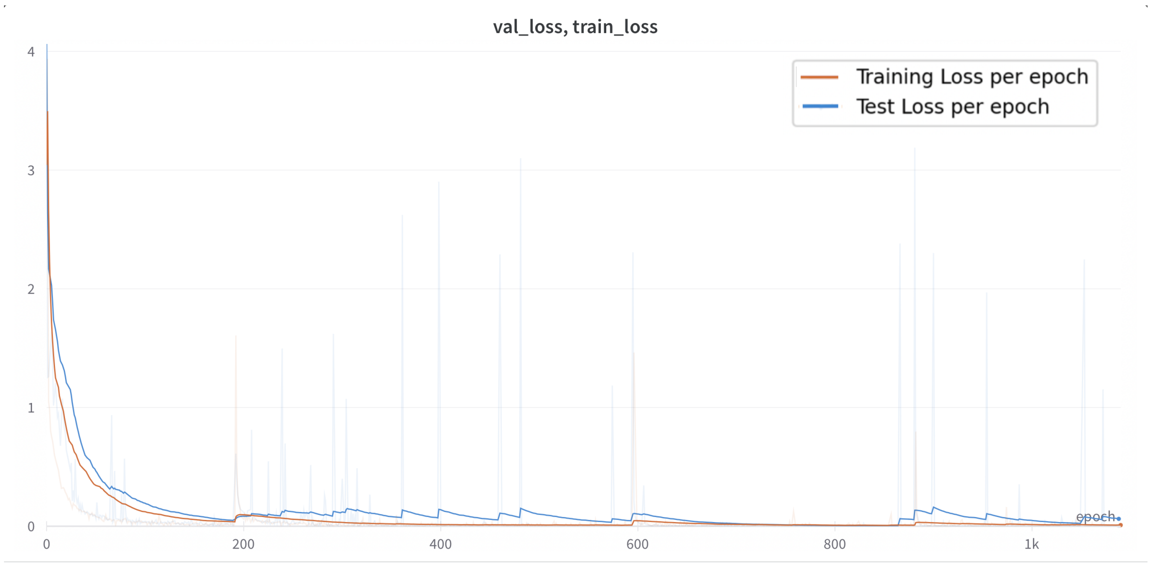

Figure 5.5: Loss curve for the training and testing of the LSTM model

and continues to perform well.

Accuracy

We calculate accuracy as slot accuracy. It measures how often the LSTM model predicts the same slot as the T-HTN planner. As shown by Figure 5.6, we get a good test accuracy value of 90.2% around 1200 epochs, with high accuracy rates achieved in the early stages of the training. With the seq2seq modeling, the LSTM achieves better accuracy rates than the baseline.

ROC Curve and Precision-Recall

We also calculate metrics similar to the MLP baseline to measure the model's performance, as shown in figure 5.7. ROC (Receiver Operating Characteristic) curve, AUC (Area Under the ROC Curve), and F1 score are standard evaluation metrics used in machine learning models to measure the model's ability to distinguish between positive and negative classes for classification problems.

All metrics show a positive trend, with an excellent F1 and AUC score. As shown

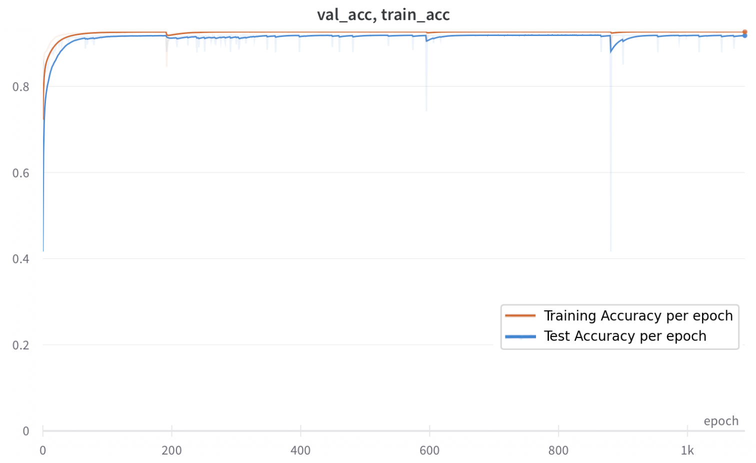

Figure 5.6: Accuracy curve for the train and test

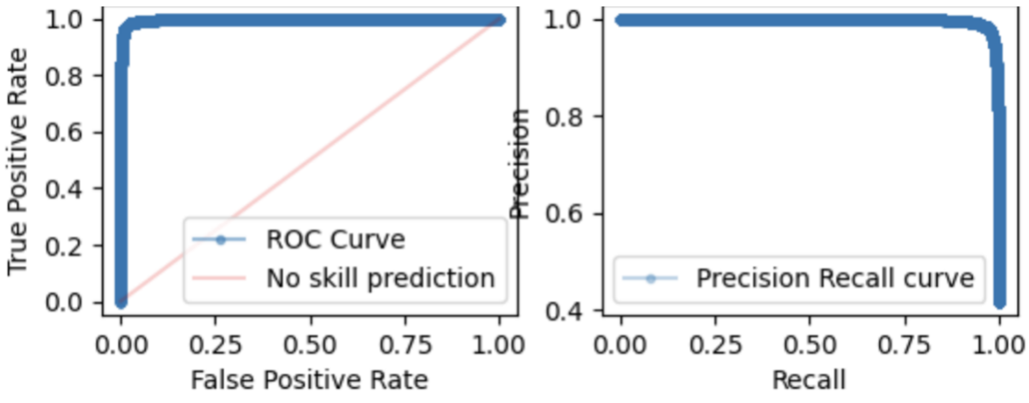

Figure 5.7: ROC and Precision-Recall Curve

Table 5.2: Evaluation Metrics - Learn2Plan

Model	**Accuracy**	**F1-Score**	**AUC-Score**
Learn2Plan	90.2%	0.893	0.996

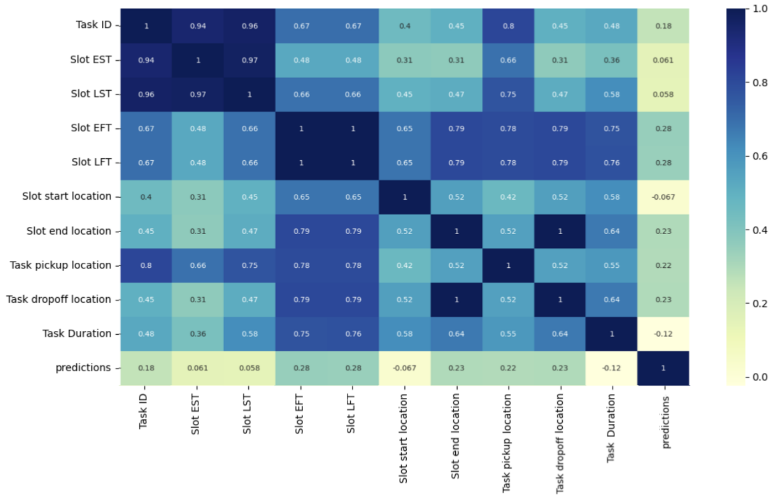

Figure 5.8: Correlation Matrix

in the Table 5.2. A high F1 score says our predictions have low false negatives and positives, and the model classifies well.

Feature Importance

Similar to the baseline, we calculate feature importance in Figure 5.8

As we can see from the correlation matrices and much like the MLP correlation matrix, the spatial and temporal constraints, such as the pickup location, drop location, and the time bounds of the incoming item and the slots, have a high correlation.

Much like the baseline, the LSTM model is also not dominated by specific features and understands the interplay of these features equitably, as seen in the last column.

Table 5.3: Computation Metrics

5 requests	**T-HTN**	**POPF**	**MLP**	**Learn2Plan**
5 rail blocks	0.150	0.143	0.052	0.128

5.3 Computation Times

Our mission is to reduce the computation time for the slot-matching process. The novel thing about our approach is that no matter how many candidate slots are presented, the learner will take the same amount of time. However, the planner computation time will grow linearly with the number of candidate slots. For example, for a particular request, if ten slots have to be evaluated by the planner, following the Learn2Plan Heuristic will take approximately 1/10th of the computation time that the planner will take.

We compare Learn2Plan with the planner T-HTN from which we have generated our training data, and POP-F [5] a forward chaining temporal planner used in task-planning and scheduling benchmarks. For every request, we compute the computation time improvement for our model (both LSTM and baseline) by the equation:

$$t_{improvement} = (t_{Planner})_{allslots} - t_{model}$$
$$t_{model} = (t_{model})_{modelinference} - (t_{Planner})_{modeloutputslot}$$

Where t is computation time. As we can see in the table 5.3, we get computational speedups compared to both planners. The results are statistically significant, with p-scores of 0.012 and 0.014 for MLP and Learn2Plan, respectively, from a one-sample t-test. The MLP baseline achieves a 65.3% and 63.6% computational speedup compared to T-HTN and POP-F, respectively. The LSTM also fairs well with 14.6% and 15.3%, respectively, due to higher inherent complexity. However, we will achieve substantial improvement as we increase the number of slots per request, requests, and rail blocks. As accuracy is a feature for computational speedup, as it requires computing even to validate sub-optimal slots, LSTM will work reasonably well for complex cases with multiple slots.

Table 5.4: Generalization accuracy

5 requests	**MLP**	**Learn2Plan**
10 rail blocks	76.11%	78.21%

Table 5.5: Computation Metrics

5 requests	**T-HTN**	**POPF**	**MLP**	**Learn2Plan**
5 rail blocks	0.150	0.143	0.052	0.128
10 rail blocks	0.150	0.143	0.06	0.135

5.4 Generalization

We generated sample data with five requests and ten rail blocks to evaluate performance on unseen data. 10 Tasks were randomly sampled from this dataset and used with Learn2Plan to evaluate the generalizability of this approach. First, the mean accuracy is calculated in Table 5.4. We believe this is due to the difference in rail block locations, as it is from a different statistical distribution than that of 5 rail blocks.

The computation time is also measured to compare with the ten rail block test set that shows the numbers are relatively similar due to similar input and output sizes as shown in Table 5.5.

These results show that the model can reasonably generalize for uncertain requests.

Chapter 6

Simulation

Moving on to the MuJoCo-based simulation framework with real physical and joint dynamics. We have ported previous simulation approaches in planning problems from ROS Gazebo to MuJoCo because of a lot of reasons, including:

- Performance: MuJoCo is designed to be highly optimized for fast and efficient simulations, making it more suitable for real-time control applications. On the other hand, ROS Gazebo may be slower due to the overhead of ROS middleware.
- Physics Engine: MuJoCo uses a proprietary physics engine that is highly accurate and provides a realistic simulation of complex robotic systems. In comparison, ROS Gazebo uses open-source ODE or Bullet physics engines that may not be as accurate or reliable for some applications.
- Ease of use: MuJoCo has a more straightforward and user-friendly interface, making it easier for researchers to set up and run simulations. In comparison, ROS Gazebo has a steeper learning curve and may require more programming and simulation setup expertise.

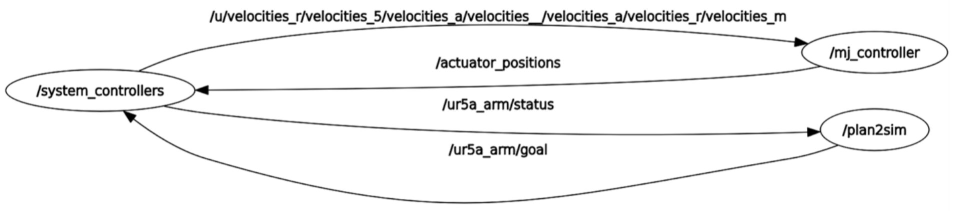
/u/velocities_r/velocities_5/velocities_a/velocities__/velocities_a/velocities_r/velocities_m
/mj_controller
/actuator_positions
/system_controllers
/ur5a_arm/status
/plan2sim
/ur5a_arm/goal

Figure 6.1: ROS Pipeline

6.1 ROS Pipeline

The ROS simulation pipeline in figure 6.1 consists of three modules - Plan2Sim, System_Control, and mujoco_control. Let us see what all three modules do-

- Plan2Sim: Parses an output plan, a JSON file, to identify coordinates from symbolic locations such as rail_block5. After interpreting, actions are assigned to subsystems.
- system_controllers: The robot and the rail are modeled as a subsystem in this module with relevant URDF files. This module handles actuation.
- mujoco_control: Handles the difference between desired and current values, interfaces with MuJoCo, and sets limits, checks, and timeouts.

The example environment for a sorbent bed replacement task with a free flyer modeled as a drone can be seen here:

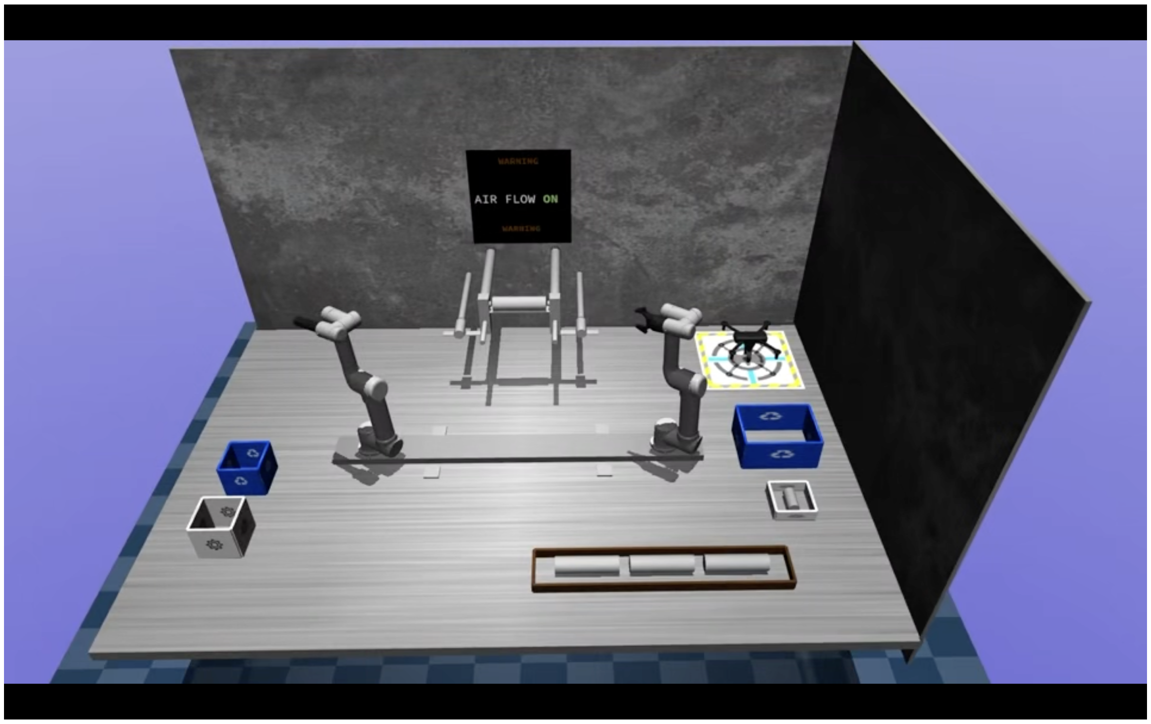

Figure 6.2: Simulation Environment

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we propose Learn2Plan, a novel and comprehensive LSTM-based memory framework that combines the robustness and accuracy of a hierarchical task planner and the efficiency of a shallow learning neural network model. Further, this thesis also puts forward a method of a surrogate learning model that can aid multiple HTN planners in scalability by speeding up their search process. It also opens up a new research topic of learning heuristics and decision structures of hierarchical networks through plan search tree traces. The simulation pipeline developed also proposes a step towards physically-aware closed-loop planning to enhance the dimension of interpretability in task-based planners.

In this framework, examples of planner runs are used to understand the spatial and temporal reasoning behind the working of a planner without using any constraints for the learning framework. For a given domain, the time bounds and the operating locations of the tasks and slots generated by the planner are used. In planners, the slot generation process is followed with this learning heuristic and bypasses the lengthy and time-intensive slot-matching process. The learner takes all the candidate slots as input and, through learning, outputs the best slot for a particular request optimizing for makespan. The learning framework uses an LSTM model that understands the

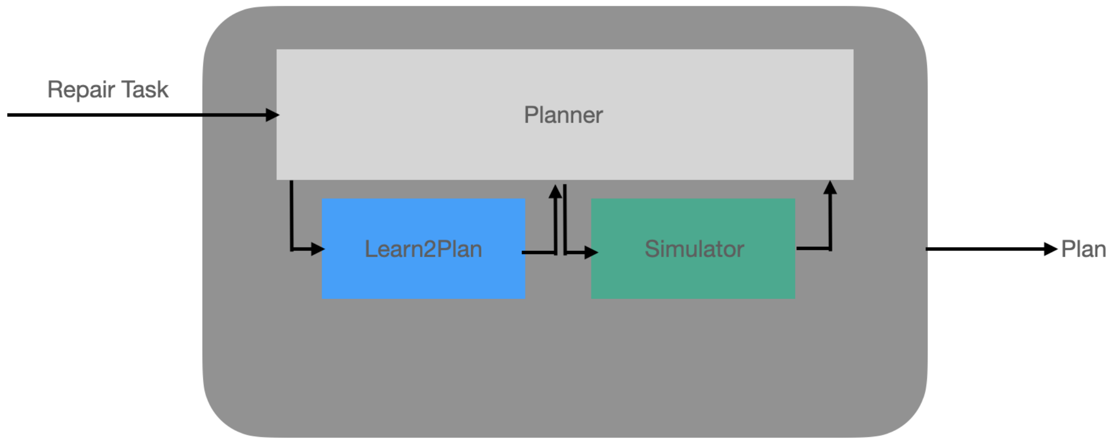

Figure 7.1: Planning system

sequentiality in the nature of requests and uses memory to reason for different slots for different requests.

Finally, we conducted initial testing and proof-of-concept on a limited data set that shows substantial computational speed-up compared to existing planners and benchmarks. This speed-up is compared with almost 90% accuracy rates, guaranteeing robustness in generated plans. Learn2Plan learns variable ordering heuristics through planning tree traces and demonstrates effectiveness as a scalable add-on to HTN planners. This thesis proposes a planning system that is robust, scalable, and interpretable with the proposed solutions, as shown in Figure 7.1. Such planning systems will enable efficient autonomous coordination in deep space habitats by using a robust planner with a Learn2Plan integration to make it much more efficient. This planner will also be integrated with the simulator to execute the plan in a physics-aware environment and replan for failures.

7.2 Future Work

Learn2Plan experiment results show trends in the positive direction and achieve excellent results with our initial tests. However, there are three future research directions for this work:

First, in the area of extensive testing and evaluation. This work's true essence and advantage will be seen when it is applied on multiple slots per request and many requests because this will linearly increase the computation time for planners but will always stay the same for Learn2Plan. Taking one step further to assess domain independence, use different domains and problem instances than the one discussed in the thesis. Ultimately, we want this approach to be domain-independent to a large extent, if not entirely, and work with different hierarchical planners with minimal modifications.

Second, Learn2Plan will work better if we convert the input values to world embedding. World embedding is defined in a latent space that adds continuity to discrete data and helps achieve higher accuracy rates by developing an upstream neural network to convert variable input sizes to same-size embedding. Such embedding would further go in the LSTM model instead of currently zero-padding cells.

Lastly, we want to integrate the simulation pipeline with plan execution and replanning. As we know, planners work offline, and if some uncertainty or failure is crept in the simulation due to physical features like friction, there should be a replan trigger from the simulator. This would require sending the world state and the reason for failure back to the planning system to replan from that state to achieve the final state.

Bibliography

[1] Maryam Adineh and Mostafa Nouri-Baygi. High quality degree based heuristics for the influence maximization problem, 2019.

[2] Fahiem Bacchus and Paul van Run. Dynamic variable ordering in csps. In *International Conference on Principles and Practice of Constraint Programming*, 1995.

[3] K.R. Baker. *Introduction to Sequencing and Scheduling [By] Kenneth R. Baker*. Wiley, 1974. URL https://books.google.com/books?id=dYgYcgAACAAJ.

[4] Daniel Borrajo and Manuela Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *Artificial Intelligence Review*, 11(1):371–405, 02 1997. ISSN 1573-7462. doi: 10.1023/A:1006549800144. URL https://doi.org/10.1023/A:1006549800144.

[5] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 20(1):42–49, May 2021. doi: 10.1609/icaps.v20i1.13403. URL https://ojs.aaai.org/index.php/ICAPS/article/view/13403.

[6] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1):61–95, 1991. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(91)90006-6. URL https://www.sciencedirect.com/science/article/pii/0004370291900066.

[7] Floris Doolaard and Neil Yorke-Smith. Online learning of variable ordering heuristics for constraint optimisation problems. *Annals of Mathematics and Artificial Intelligence*, 94(1-2):1–21, 2022. doi: 10.1007/s10472-022-09816-z. URL https://doi.org/10.1007/s10472-022-09816-z.

[8] Markus Fromherz. Constraint-based scheduling. volume 4, pages 3231 – 3244 vol.4, 02 2001. ISBN 0-7803-6495-3. doi: 10.1109/ACC.2001.946421.

[9] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for

constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(80)90051-X. URL https://www.sciencedirect.com/science/article/pii/000437028090051X.

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

[11] Okhtay Ilghami, Dana Nau, Héctor Muñoz-Avila, and W. Aha. Learning preconditions for planning from plan traces and htn structure. *Computational Intelligence*, 21:388–413, 11 2005. doi: 10.1111/j.1467-8640.2005.00279.x.

[12] Steve Minton. Learning search control knowledge: An explanation-based approach. In *Proceedings of the 7th International Workshop on Machine Learning*, pages 281–287. Morgan Kaufmann Publishers, 1990.

[13] Nicola Muscettola, P.Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote agent: to boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1):5–47, 1998. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(98)00068-X. URL https://www.sciencedirect.com/science/article/pii/S000437029800068X. Artificial Intelligence 40 years later.

[14] Viraj Parimi, Zachary B Rubinstein, and Stephen F Smith. T-htn: Timeline based htn planning for multiple robots. *HPlan 2022*, page 59.

[15] Chao Qi, Dan Wang, Héctor Muñoz-Avila, Peng Zhao, and Hongwei Wang. Hierarchical task network planning with resources and temporal constraints. *Knowledge-Based Systems*, 133:17–32, 2017. ISSN 0950-7051. doi: https://doi.org/10.1016/j.knosys.2017.06.036. URL https://www.sciencedirect.com/science/article/pii/S0950705117303167.

[16] Stuart M. Shieber and Wheeler Ruml. Adaptive tree search. 2002.

[17] S.F Smith, M.A Becker, and L.A Kramer. Continuous management of airlift and tanker resources: A constraint-based approach. *Mathematical and Computer Modelling*, 39(6):581–598, 2004. ISSN 0895-7177. doi: https://doi.org/10.1016/S0895-7177(04)90542-0. URL https://www.sciencedirect.com/science/article/pii/S0895717704905420. Defense transportation: Algorithms, models, and applications for the 21st century.

[18] Wen Song, Zhiguang Cao, Jie Zhang, Chi Xu, and Andrew Lim. Learning variable ordering heuristics for solving constraint satisfaction problems. *Engineering Applications of Artificial Intelligence*, 109:104603, 2022. ISSN 0952-1976. doi: https://doi.org/10.1016/j.engappai.2021.104603. URL https://www.sciencedirect.com/science/article/pii/S0952197621004255.

[19] Zachary N Sunberg, Zichuan Yang, Andreas Mueller, Felix Berkenkamp, and Angela Schoellig. Adaptive entropy tree search for planning and learning. In *Proceedings of the 35th International Conference on Machine Learning*, pages 4793–4802, 2018.

[20] Inkyung Sung, Bongjun Choi, and Peter Nielsen. Reinforcement learning for resource constrained project scheduling problem with activity iterations and crashing. *IFAC-PapersOnLine*, 53(2):10493–10497, 2020. ISSN 2405-8963. doi: https://doi.org/10.1016/j.ifacol.2020.12.2794. URL https://www.sciencedirect.com/science/article/pii/S2405896320335588. 21st IFAC World Congress.