Execution Cost Optimization for Hierarchical Planning in the Now

by

Dylan Hadfield-Menell

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

© Massachusetts Institute of Technology 2013. All rights reserved.
Execution Cost Optimization for Hierarchical Planning in the Now

by

Dylan Hadfield-Menell

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 2013, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

Abstract

For robots to effectively interact with the real world, they will need to perform complex tasks over long time horizons. This is a daunting challenge, but human ability to routinely solve these problems leads us to believe that there is underlying structure we can leverage to find solutions. Recent advances using hierarchical planning [19] have been able to solve these problems by breaking a single long-horizon problem into several short-horizon problems. While this approach is able to effectively solve real world robotics planning problems, it makes no effort to account for the execution cost of an abstract plan and often arrives at poor quality plans. In this thesis, we analyze situations that lead to execution cost inefficiencies in hierarchical planners. We argue that standard optimization techniques from flat planning or search are likely to be ineffective in addressing these issues. We outline an algorithm, RCHPN, that improves a hierarchical plan by considering peephole optimizations during execution. We frame the underlying question as one of evaluating the resource needs of an abstract operator and propose a general way to approach estimating them. We introduce the marsupial logistics domain to study the effectiveness of this approach. We present experiments in large problem instances from marsupial logistics and observed up to 30% reduction in execution cost when compared with a standard hierarchical planner.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor
Acknowledgments

First and foremost, I would like to thank my advisors, Leslie Kaelbling and Tomás Lozano-Pérez. Taking 6.01 with them in my freshman year spring inspired me to choose computer science as my major and their insights, encouragement, and prodding have been indispensable in writing this thesis. I would like to thank the members of the LIS lab for helping me get started with research and providing a great environment to learn how to present. Finally, I'd like to acknowledge my parents who have been helpful and supportive throughout this process and thank my friends for providing welcome distractions when they were needed.
Contents

1 Introduction ................................................................. 11

2 Background ................................................................. 15
   2.1 Domain representation .............................................. 15
   2.2 Abstraction in Planning ............................................ 17
   2.3 Hierarchical Planning in the Now ............................... 18

3 Optimizing Hierarchical Planning ...................................... 21
   3.1 Execution cost inefficiencies in hierarchical planning .... 21
      3.1.1 Incorrect Ordering of Abstract Operators ............. 22
      3.1.2 Missed Parallel Structure ................................. 25
   3.2 Optimization in the now ........................................... 28
      3.2.1 Context-sensitive ordering ................................. 29
      3.2.2 Leveraging pairwise ordering information ............. 31
   3.3 Ordering-preference heuristics .................................. 33

4 Related Work .............................................................. 37
   4.1 Symbolic planning .................................................. 37
   4.2 Hierarchical planning .............................................. 39
   4.3 Partial orders and planning ...................................... 41

5 Evaluation & Experiments .............................................. 43
   5.1 Transportation domain with marsupial robots ............... 43
      5.1.1 Fluent specification .......................................... 46
## List of Figures

3-1 Caricature of situations in which incorrectly ordering abstract tasks results in poor execution cost. .............................................. 23

3-2 Illustration of situation where combing subgoals can reduce execution cost. ................................................................. 26

5-1 Visualization of the Marsupial Logistics Domain. ................................................. 44

5-2 Example planning tree for marsupial logistics. ................................................................. 45

5-3 Average percent decrease in plan cost vs. problem size for RCHPN vs. HPN. ................................................................. 52

5-4 Plot of percent decrease in execution cost vs. percent increase in planning time. ................................................................. 54
Chapter 1

Introduction

A longstanding goal of robotics research is the development of machines that can accomplish complex tasks in unstructured real-world settings. This is inspired by a desire to build robots that can perform household tasks, assist in hospitals, or take part in search and rescue operations.

Since the 1960’s there have been significant advances in many of the component modules for these robots. The release of the Kinect RGBD sensor has enabled cheap high-quality perception. Hardware advances embodied in the Willow Garage PR2 robot, combined with improved motion planning methods, are beginning to enable complex and interesting manipulation [4]. State estimation techniques and probabilistic methods have given us tools to reason about uncertainty in the world [6]. Symbolic planning has made great strides through the discovery of effective domain independent heuristics [18, 15, 26]. Forty years of processor improvements according to Moore’s Law have given us the computing power to leverage these techniques.

While these improvements have led to dramatic advances in the ability of robots to perform primitive actions, such as picking up a plate, the ability to combine these actions to perform complex and novel tasks, such as clearing a table, remains beyond the current state of the art. This is not without reason, planning problems faced by a household robot are characterized by long horizons, partial observability, and continuous variables. Planning is PSPACE-complete in the discrete, fully-observable case, so the difficulty in applying it to real world settings is not surprising. Inspired
by human ability to routinely solve these seemingly intractable problems, we believe that there is some underlying structure or simplicity in these problems that provides a mechanism for reducing complexity in typical problem instances.

One way to reduce complexity, for certain classes of long-horizon problems, is to use temporal hierarchy to decompose a problem into multiple short-horizon problems. A method that has been shown to be effective in robotic mobile-manipulation problems is the Hierarchical Planning in the Now (HPN) architecture [19]. HPN makes use of an aggressive hierarchical strategy. It commits to an abstract plan and interleaves planning and execution to obviate the need to reason about all of the ways an abstract action can be executed. It has been shown to be correct and complete for a class of hierarchical system specifications that is suitable for modeling household robotic tasks and mobile manipulation problems.

While HPN is able to find solutions to many large planning problems, it makes no claims about the quality of the behavior it produces, even when an optimizing algorithm (e.g. A*) is used to solve the individual subproblems. The resulting behavior can be short-sighted, with the robot achieving one subgoal, only to have to undo it, fix something else, and then re-achieve the original subgoal.

The fundamental difficulty is that, at the upper levels of the hierarchical planning process, the models used do not account for the cost of taking abstract actions. From the point of view of the abstract planner, all actions will take the same amount of time to execute. This is clearly not the case, as different subtasks will result in different sequences of primitive operations. However, specifying this cost can be difficult: it may be highly variable and depend on details of the situation in which the operator is executed. Determining this cost is generally as difficult as finding a fully grounded plan.

For example, consider delivering a package to some destination in a distributed robotic transportation system. We can consider operations of forklifts for loading, unloading, and arranging packages within a truck or airplane, as well as operations that drive and fly the transportation vehicles. The cost of delivering that package depends on the initial locations of trucks and planes, the arrangements of other pack-
ages currently in their cargo holds, and the package of interest's current location. Furthermore, these values depend on the initial state and on abstract operations that are executed before the operator whose cost we are evaluating. As a result, two plans that look similar at the abstract level (i.e., the execution order for two abstract operations is swapped) may result in large differences in the quality of the behavior that the system can generate.

We propose a strategy for tackling the problem of optimization in hierarchical planning that addresses plan quality by dynamically reordering and grouping the subgoals in an abstract plan. Our approach lets us frame the cost estimation problem as one in which, given two subgoals $G_1$ and $G_2$, we must estimate which of the following strategies will be most efficient: planning for and executing $G_1$ first, planning for and executing $G_2$ first, or planning for them jointly and interleaving their execution. Given the ability to answer that query, we will be able to perform "peephole optimization" of the plan at execution time, taking advantage of immediate knowledge of the current state of the world to select the best next action to take.

We propose general principles, based on concepts of shared and constrained resource use, for the design of heuristics to answer the ordering-preference queries. The overall utility of this approach is demonstrated in very large instances of a multi-robot transportation problem that cannot be solved through classical non-hierarchical methods. We show up to 30% improvement in plan quality over the non-optimizing version of HPN. Furthermore, on problems with little room for optimization, we find that our approach results in only a negligible increase in planning time.

The remainder of this thesis is organized as follows. Chapter 2 provides an introduction to the representations and formalisms we use for planning, the use of general use of abstraction in planning, and the HPN architecture. Chapter 3 describes the execution cost optimization problem for hierarchical planning in detail and presents our peephole optimization solution. Chapter 4 summarizes the related work in this area. Chapter 5 describes the experiments performed to evaluate the effectiveness of our system using hand-coded heuristics. Chapter 6 summarizes the contributions of this thesis and concludes with a discussion of directions for future research.
Chapter 2

Background

This chapter provides an introduction to the problem of task planning, the representations used, and the solution techniques this work relies on. The first section describes the planning problem. It uses an example of a package delivery problem to illustrate the aspects a planning problems we would like to model. The second section describes the use of abstraction in planning and motivates its need. The final section describes the Hierarchical Planning in the Now (HPN) framework and describes its advantages and disadvantages over other planning setups.

2.1 Domain representation

We use a relatively standard symbolic representation for planning operators, derived from STRIPS [12] but embedded in Python to allow more freedom in specifying preconditions and effects. In the domains considered in this paper, the geometric aspects of loading trucks are discretized; it would be possible to use real continuous representations of object and robot poses instead [19].

A domain is characterized by:

- **Entities**: names of individual objects in the domain; for example, trucks, packages, planes, forklifts, etc.

- **Fluents**: logical relationships between entities in the world that can change over time; for example, $\text{In(package1, truck3)}$. 
• **Initial state:** a conjunction of logical fluents known to be true initially.

• **Goal:** a conjunction of logical fluents specifying a set of desired world states.

• **Operators:** actions that are parameterized by objects (e.g. \( \text{PickUp(package2)} \)). Each operator, \( op \), is characterized by:
  
  - **preconditions:** \( \text{pre}(op) \), a conjunction of fluents that describes when this operator is applicable
  
  - **result:** \( \text{res}(op, s) \), a conjunction of fluents whose value changes as a result of applying \( op \) in state \( s \)
  
  - **choose:** a list of variables names and values they can take on
  
  - **cost:** a real valued cost of applying \( op \).

A planning problem, \( \Pi \), is a tuple; \( \Pi = (F, O, I, G) \), where \( F \) is a set of fluents, \( O \) is set a operators, \( I \) is an initial state, and \( G \) is a goal. A solution to \( \Pi \) is a sequence of operators \( p = \{op_1, op_2, \ldots, op_n\} \). A plan is **feasible** if the preconditions for \( op_1 \) are satisfied in the initial state and the preconditions for each other operator, \( op_i \), in the state that results from applying \( op_{i-1} \): \( I \in \text{pre}(op_1) \) and \( s_i = \text{res}(op_i, s_{i-1}) \in \text{pre}(op_{i+1}) \). A plan achieves a goal if the goal formula is satisfied in the final state: \( s_n \in G \). The cost of a plan, \( p \), is the sum of of the costs of its operators: \( \text{cost}(p) = \sum_{i=1}^{n} \text{cost}(op_i) \).

In realistic domains, specifying a truth value for every possible fluent is usually difficult and, as is the case for continuous domains, can even be impossible. We address this problem by performing backward search from the goal set and computing **preimages** of subgoals under operators until we reach a subgoal that contains the initial state. This method of chaining preimages is known as **goal regression.** This approach allows us to avoid representing the initial state completely in the language of fluents and instead provide a function for each fluent that allows its truth to be tested in the initial state.

The primary difference between formalism presented here and standard formalisms is the **choose** attribute of operators. When the possible values for a variable are fully enumerated, this is analogous to including extra parameters for an operator. However, when planning in large or continuous state spaces, we can, and do, generate a small
number of potential bindings based on the current state. This is similar to the effect applicator modules from semantic attachment approaches to planning [11].

### 2.2 Abstraction in Planning

The complexity class of algorithms to find solutions to planning problems using the domain representation we have described is exponential in the length of the solution [5]. In many real-world problems, such as planning for an entire day’s worth of actions, this may be hundreds or thousands of actions and will require unacceptable amounts of computation time. One way to mitigate this is through the use of abstract planning.

An abstraction method is a function, \( f : (F, O, I, G) \rightarrow (F', O', I', G') \) that maps a planning problem into a simplified version that is easier to solve. In this work, we will focus on temporal abstractions, where the goal is to map problems into abstract versions that have shorter solutions. The central concept is to use a solution to the abstract problem to help find a solution to the original, concrete, problem. This process of converting an abstract plan into a concrete one is known as refinement. An abstraction method can be applied recursively in order to define a hierarchy of abstraction spaces [27].

There are many strategies for constructing abstractions. We will demonstrate optimization methods in the context of temporal abstraction hierarchies of the type used in HPN, but the techniques are general and could be applied to other types of hierarchies.

We construct a hierarchy of temporal abstractions by assigning a criticality in the form of an integer to each precondition of an operator, \( op \). If the largest criticality in \( op \) is \( n \), then we have \( n \) abstract operators, denoted \( \text{abs}(op, i), 0 \leq i < n \). The preconditions of \( \text{abs}(op, i) \) are the preconditions of \( op \) which have criticality \( k > i \). This defines a hierarchy of abstractions for a particular operator, as more abstract versions ignore more preconditions. An abstraction level for the whole space is a mapping \( \alpha : O \rightarrow \{1, \ldots, n\} \) which specifies the abstraction level for each operator.
Note that this depends on the particular way an operator’s variables are bound to entities. \( \text{Place}(\text{package1}, \text{truck1}) \) could map to a different abstraction level than \( \text{Place}(\text{package2}, \text{truck2}) \).

2.3 Hierarchical Planning in the Now

Most hierarchical planning methods construct an entire plan at the concrete level, prior to execution, using the hierarchy to control the search process. The HPN method, in contrast, performs an online interleaving of planning and execution. This allows it to be robust to uncertainty: it avoids planning for subgoals in the far future at a fine level of detail because it is likely that those details may change. In addition, it can choose to delay detailed planning because the information necessary to support that planning has not yet been acquired.

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure{HPN}{s, \gamma, \alpha, \text{world}}
\State \( p = \text{Plan}(s, \gamma, \alpha) \)
\For{(op\textsubscript{i}, gi) in p}
\If{IsConcrete(op\textsubscript{i})}
\State \text{world.execute}(op\textsubscript{i}, s)
\Else
\State HPN(s, gi, NextLevel(\alpha, op\textsubscript{i}), world)
\EndIf
\EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

The HPN algorithm is shown in Algorithm 1. It takes as inputs the current state of the environment, \( s \); the goal to be achieved, \( \gamma \); the current abstraction level, \( \alpha \); and the \textit{world}, which is generally an interface to a real or simulated robotic actuation and perception system. Initially \( \alpha \) is set to the most abstract version of every operator.

HPN starts by calling the regression-based Plan procedure, which returns a plan at the specified level of abstraction,

\[ p = ((-, g_0), (op_1, g_1), ..., (op_n, g_n)) , \]
where the $op_i$ are operator instances, $g_n = \gamma$, $g_i$ is the preimage of $g_{i+1}$ under $op_i$, and $s \in g_0$. The preimages, $g_i$, will serve as the goals for the planning problems at the next level down in the hierarchy.

HPN executes the plan steps, starting with action $op_1$, side-effecting $s$ so that the resulting state will be available when control is returned to the calling instance of HPN. If an action is a primitive, then it is executed in the world, which causes $s$ to be changed; if not, HPN is called recursively, with a more concrete abstraction level for that step. The procedure NextLevel takes a level of abstraction $\alpha$ and an operator $op$, and returns a new level of abstraction, $\beta$, that is more concrete than $\alpha$.

The strategy of committing to the plan at the abstract level and beginning to execute it before finding a full concrete plan is potentially dangerous. If it is not, in fact, possible to make a plan for a subgoal at a more concrete level of the hierarchy, then the entire process will fail. In order to be complete, a completely general hierarchical planning algorithm must be capable of backtracking across abstraction levels if planning fails on a subgoal. An alternative, which we adopt in this work, is to require hierarchical structures that have the downward refinement property (DRP), which requires that any abstract plan that reaches a goal has a valid refinement that reaches that goal. Bacchus and Yang [2] describe several conditions under which this assumption holds.
Chapter 3

Optimizing Hierarchical Planning

A fundamental difficulty of hierarchical planning with downward refinement is that the costs of abstract actions are not available when planning at the high level, so that even if completeness is guaranteed, the resulting trajectories through the space can be very inefficient. In this chapter, we analyze common cases where execution cost inefficiencies arise in hierarchical planning. We argue that standard cost-sensitive search is unlikely to effectively address these concerns. We present RCHPN to cope with these issues via online peephole optimization and characterize the domain specific information it requires.

3.1 Execution cost inefficiencies in hierarchical planning

To illustrate execution cost issues for hierarchical planning, consider a domain with a single robot and multiple boxes. This robot can carry up to two boxes at a time and must transport them to goal locations, avoiding obstacles. Our examples will make heavy use of the Place operator so we give a full specification of its operator schema. In the following, the function $IK$ stands for inverse-kinematics and, given a location for an object and a grasp, computes the corresponding robot pose to hold that object at that location with that grasp.
Place(obj, loc, grasp):

res: \( \text{ObjLoc}(obj, loc) \land \lnot \text{Holding}(obj, grasp) \)

pre: 1. LegalLoc(IK(loc, grasp)), LegalLoc(loc)
2. ClearPath(CurrentLoc(obj), loc)
3. Holding(obj, grasp)
4. RobotLoc(IK(loc, grasp))

cost: 10

Remember that, at each level, our precondition formula is the conjunction of fluents at that level and the levels above it. For example, the precondition for \( \text{Place}(obj, loc, grasp) \) at abstraction level 2 is

\[
\text{LegalLoc}(IK(loc, grasp)) \land \text{LegalLoc}(loc) \land \text{ClearPath}(\text{CurrentLoc}(obj), loc)
\]

At the highest level, we require that the target location and a configuration from which to place that object are collision free. In planning at the next level, we require that there exist a path that will enable the robot to transfer the object from its current location to its goal. At the next level we need to be holding the object. At the most concrete level we require that the robot also be in the correct position to place the object. This is a reasonable precondition hierarchy and can be used to find solutions to mobile-manipulation problems. It mirrors hierarchies used in [19]. We now present two example problems and explore the behavior this hierarchical specification elicits. The execution cost issues we observe are indicative of broad classes of inefficiencies we observe in hierarchical planning.

3.1.1 Incorrect Ordering of Abstract Operators

A common failure mode of temporal hierarchy, with respect to execution cost, comes from the incorrect ordering of abstract tasks. Consider a robot tasked with placing 3 boxes, call them \( box_1, box_2, \) and \( box_3 \), in an enclosed space, where placing one of the
Figure 3-1: Caricature of situations in which incorrectly ordering abstract tasks results in poor execution cost. The task is to transport the boxes from their initial locations on the left to their goal locations on the right. Our issue arises because all orderings of the operators \{\text{Place}(\text{box}_1), \text{Place}(\text{box}_2), \text{Place}(\text{box}_3)\} are valid abstract plans yet placing \text{box}_3 first will require us to do extra work. If the abstract plan has the operator sequence \[ \text{Place}(\text{box}_3), \text{Place}(\text{box}_2), \text{Place}(\text{box}_1) \], we will have to move \text{box}_3 away from its goal in order to accomplish each additional subgoal and replace it after. By comparison, if our planner orders the operators correctly it avoids this work and roughly halves the resulting execution cost. It is hard to address this issue by specifying cost estimates for abstract actions or by directly modifying the hierarchy to preclude this behavior.

Boxes blocks entry to that space. This is a common characteristic of manipulation problems because of the conservative estimates used to avoid collisions. An example scenario is depicted in Fig. 3-1.

Using the precondition hierarchy described above, the realizable abstract operator sequences are permutations of

\[
\text{Place}(\text{box}_1, \text{goal}_1, \text{grasp}_1), \ \text{Place}(\text{box}_2, \text{goal}_2, \text{grasp}_2), \ \text{Place}(\text{box}_3, \text{goal}_3, \text{grasp}_3).
\]

Where \text{goal}_i is the goal location of \text{box}_i and \text{grasp}_i is a feasible grasp for placing \text{box}_i in \text{goal}_i. Our issue arises because the execution cost of the corresponding concrete plans exhibits a large variation over this set.

At the two different extremes of execution cost are the following two plans, shown here with only \text{ObjLoc} fluents from subgoals to conserve space.
The abstract plans we generate specify serializations of our goal: we will accomplish each fluent in the goal sequentially and attempt to keep achieved fluents true. Plan 3.1 will have lower execution cost than Plan 3.2 because there is no concrete plan in which \( \text{ObjLoc}(\text{box}_3, \text{goal}_3) \) becomes true before the other goal fluents and remains true until the full goal is achieved. Placing \( \text{box}_3 \) in its goal location blocks access to the goal locations for the other boxes. As a result achieving the second and third subgoals in Plan 3.2 will consist of moving \( \text{box}_3 \) out of the way, placing the appropriate box, and replacing \( \text{box}_3 \). This increases the number of place operations required from 3 to 7 and essentially doubles the execution cost of Plan 3.2 over Plan 3.1, depending on the cost of other required operators.

We ultimately achieve the goal, but at a much higher cost than is necessary. A context-sensitive cost would enable us to avoid this issue and select the cheaper option. Unfortunately, although we have costs for primitive actions, it is difficult to determine the cost for an abstract operator at planning time. This cost depends on solving many subsequent planning problems and is not purely a function of the operator, parameters and abstraction level. For example, Plan 3.1 and Plan 3.2 use the same operators with the same parameters at the same level of abstraction but have very different execution costs. We cannot simply include this value as a part of the domain description, as we do for primitive operators.

Computing cost estimates during execution is difficult to do without incurring a
large increase in planning time. In our example, the cost for \( \text{Place}(\text{box}_1, \text{goal}_1, \text{grasp}_1) \) depends on the locations of the other boxes, the location of the robot, and the clear paths from \( \text{box}_1 \)'s location to \( \text{goal}_1 \). Furthermore, it relies on these values at the time we plan for and execute that particular operator as opposed to their values when we search for an abstract plan. The values we need depend on the results of the other operators in the abstract plan and are not known at planning time. Even oracle access to these values leaves much to be desired as the computations required can be prohibitively expensive to evaluate at each node expansion during our search. Unless careful attention is paid, this approach can result in worse performance than simply solving the problem without hierarchy.

An alternative approach is to alter the hierarchy such that only plans which place \( \text{box}_3 \) last, in this problem instance, are valid. A example solution of this type might combine the preconditions for \( \text{Place} \) from levels 1 and 2 so that our first plan must consider the \( \text{ClearPath} \) fluent when placing. This approach does not scale well. Other ordering issues will require adding different preconditions to more abstract levels. The likely outcome is the reintroduction of most, if not all, preconditions at the most abstract level and our agent is faced with the original, long-horizon, intractable planning problem. There is a fundamental contradiction in this strategy: abstract planning is efficient because it ignores details; adding those details to reduce execution cost will increase planning time and negate the advantages of the hierarchy.

### 3.1.2 Missed Parallel Structure

In the process of refining and executing an abstract plan, each subgoal is achieved sequentially. This is an important feature of hierarchical planning, as it keeps the planning horizon for subproblems short. It also prevents hierarchical system from leveraging parallelism in subtasks to reduce execution cost. Flexibility in serializing subgoals can enable a hierarchical planner to find shorter plans. This section will provide an example of the execution cost savings this enables and discuss the difficulties in leveraging these savings while maintaining efficiency. We consider a simplification of the the example from Section 3.1.1 that ignores \( \text{box}_3 \), but is otherwise identical, to
Figure 3-2: Illustration of situation where combing subgoals can reduce execution cost. The roots of two different planning trees to accomplish the goal \( \text{ObjLoc}(\text{box}_1, \text{goal}_1) \wedge \text{ObjLoc}(\text{box}_2, \text{goal}_2) \). 3-2(a) represents the types of plans that HPN can find for this goal. Because we serialize every subgoal in the abstract plan we will always plan for placing the two boxes independently and will not be able to take advantage of similar structure in the plans. 3-2(b) illustrates a solution which does not serialize these subgoals. The resulting subproblem has a short horizon so we can still solve it efficiently. Combining subgoals would enable an agent to avoid traveling extra distance while incurring a small computational cost, in this scenario. Introducing preconditions such the first solution found includes picking as well as placing will increase the planning horizon to a point where only simple problems can be solved. Augmenting the original planning problem with joint operators to enable this behavior will increase the branching factor and detrimentally affect performance. Note that these plans omit the level of planning that introduces the \( \text{ClearPath} \) fluent to improve clarity.
illustrate these concerns.

Even for this simple scenario, our hierarchical planner will perform substantially worse than optimal. There are two realizable abstract plans, one that places box$_1$, then box$_2$ and one that places box$_2$, then box$_1$. Suppose we get the first option as our abstract plan. In executing this plan the robot will travel to box$_1$’s initial location, pick up box$_1$, travel to goal$_1$, place box$_1$ at goal$_1$, then repeat this process for box$_2$. Recall that the robot in this example is capable of holding two boxes at once and the initial locations of the boxes are close to each other. There exists a plan with less execution cost that achieves this goal by picking up both boxes before transporting both of them to their corresponding goal locations. While finding the optimal plan is likely impossible while preserving efficiency, we should be able to take advantage of this parallel structure in subproblems to find better plans. Fig. 3-2 depicts example planning trees for this problem.

In order to take advantage of this parallelism we need to an abstract plan that considers picking and placing for both objects at the same time: this lets us to interleave the Pick and Place operators. Perhaps the simplest way to enable this is to include preconditions that relate to this structure at the highest level. In this example, that amounts to including the Holding precondition in the most abstract space so that plans will include Pick operators. This creates similar issues to modifying our hierarchy to deal with reordering. We collapse the hierarchy and make all but the simplest problems intractable; imagine planning for picking and placing 10 objects as a single planing problem.

Another option is to augment the planning problem with ‘joint’ operators: operators which represent the application of several operators at the same time. These would enable us to plan jointly for these operators more concrete levels of the hierarchy. On the surface, this is a reasonable approach; it avoids increasing the planning horizon. However, this solution runs into two issues. The first is that we are increasing the branching factor of the planning problem exponentially. If we want to consider doing $j$ of $n$ operators at the same time, we need to add $O(n^j)$ joint operators. This will certainly have a negative impact on planning time.
Furthermore, simply adding these operators is not enough; we need to enable our planner to intelligently select when it is appropriate to use a joint operator instead of the corresponding sequential operators. The standard way to do this is to include cost estimates for our new operators. We have already argued that finding cost estimates for abstract operators in planning is difficult; the problem compounds with joint operators, as the corresponding subproblems are more complicated. In order to leverage shared structure in subtasks to find cost savings without reducing our capacity to solve hard problems we will need a different approach.

3.2 Optimization in the now

In this section, we outline the central contribution of this thesis: a novel refinement strategy that enables execution cost optimization but retains the efficiency of aggressive hierarchical planning. It offers the opportunity to arrange or combine subgoals such that planning for and executing them sequentially will result in shorter plans without significantly increasing planning time.

The ordering problems discussed in the previous section arise from the fact that there are many orders of an abstract plan that are equivalent with respect to the abstract preconditions but not with respect to the ensuing execution cost. We argued that an abstract planner is ill-equipped to select the correct ordering without incurring unacceptable computational cost. Yet, the low cost options make use of the same operators found in each abstract plan. With this in mind, we can draw inspiration from motion planning, where cost-sensitive planning frequently proceeds by first finding a solution and then improving on that solution in a latter process [17].

We propose to use information from the current state of the world at plan execution time to perform peephole optimization. We find an initial plan using the same abstract planning process as before. We modify the refinement process to heuristically select the next subgoal to achieve from the unachieved subgoals in our plan. We restrict the subgoals considered to be subgoals for which the corresponding preconditions are true in the world. We also ensure that there is a valid plan, with respect
to the abstract preconditions, that executes this subgoal followed by some ordering of the remaining unachieved subgoals.

Because we select the next subgoals from a small set of options at execution time, this process can take advantage of more complex properties and details of the domain. It can also perform more expensive computation because we do not need to do cost estimation at each node expansion along our search. This will enable our planner to consider cost optimization, with respect to the reordering of operators in our abstract plan, without dramatically increasing computation time.

Similar analysis applies to the problem of deciding when to jointly achieve subgoals. An abstract planner does not generally have enough information to determine whether groups of subtasks should be addressed jointly at a lower level of abstraction, but good solutions can usually be found by considering combinations of subgoals in the original plan. Treating these options in a post-processing step fits naturally into our refinement strategy. In addition to reordering an abstract plan, our refinement process considers achieving some of the subgoals jointly. We ensure correctness in combining operators $A$ and $B$ by finding a valid plan in which $A$ and $B$ are planned for sequentially. Then we use $\text{res}(A, \text{res}(B, s))$ as the goal for our next subproblem, where $s$ is the current state. This increases the computational difficulty of the subsequent planning problem in the hope that it will generate a better quality plan.

Algorithm 2 shows an extension of HPN, called RCHPN, that implements these modifications. RCHPN relies on two functions to make ordering or combining decisions: SelectGap, which heuristically selects the next subtask to plan for, and SelectParallelOps, which will combine subgoals that should be considered jointly to expose parallel structure. Both rely on a context-sensitive function, $\text{arrange}$, to find situations in which reordering or combining would be beneficial. Before describing these procedures, we describe the ordering preference information they rely on.

### 3.2.1 Context-sensitive ordering

RCHPN depends on the specification of a context-sensitive comparison function, $\text{arrange}(g_1, g_2, s)$, which takes as arguments two subgoals and an initial state. It returns 0 if $g_1$ should
Algorithm 2 Reordering and combining hierarchical planning and execution algorithm

```
procedure RCHPN(s, γ, α, world)
2:    p = Plan(s, γ, α)
    while p ≠ ∅ do
4:        (op, g) = SelectGap(s, p, α)
        if IsConcrete(op) then
6:            world.execute(op, s)
        else
8:            sg = SelectParallelOps(s, op, p, α)
            cur_index = p.index(g)
            sg_index = p.index(sg)
            for (op’, g’) in p[cur_index : sg_index] do
12:                α = NextLevel(α, op’)
            end for
14:        RCHPN(s, sg, NextLevel(α, op), world)
        end if
16:    end while
end procedure
```

be serialized before $g_2$, returns 1 if $g_2$ should be serialized before $g_1$, and returns 2 if they should be combined into a single subgoal and solved jointly. These correspond to the subgoal sequences $(g_1; g_1 \land g_2), (g_2; g_1 \land g_2), \text{and } (g_1 \land g_2)$ accordingly. It might seem that in order to be effective, arrange will have to perform some sort of cost estimation for an abstract task; so, why do we believe that it will be easier to specify than a traditional cost function?

1. Evaluation takes place “in the now”: the algorithm knows the current world state and does not need to consider the many ways preconditions for an operator could have been realized.

2. The task is simply to determine an ordering, not to estimate the actual costs, which would generally be much more difficult to do accurately.

3. We only have to compute ordering preferences for the operators that actually appear in the plan, rather than computing a cost for every operator that is considered during the search.

The first property arises because our refinement procedure interleaves optimization
with planning. Thinking back to our boxes example, we argued that estimating the abstract cost for placing a box was hard, in part, because we did not know the initial location of the robot or the boxes when we do our cost evaluation. By considering our options in a post-processing step, we can interleave re-ordering with planning and give \textit{arrange} direct access to these values.

The second property stems from the fact that we know what the alternative options are. In evaluating costs during a general search we do not know what other operators we will need to compare to. Thus, we need a common criterion to compare this choice with any alternatives. This forces us to find an actual cost estimate because specifying pairwise orders with all other options is infeasible. In contrast, \textit{arrange} knows what the different alternatives are; we do not need the results of this computation to apply beyond the comparison of these two subgoals.

Our final property is due to the restriction of our final abstract plan to plans that contain operators from the initial solution. This enables us to do more complex and costly computation for each call to \textit{arrange} without unacceptably increasing the total amount of computation. For example, determining if placing a box at its goal will block all paths to place another is too costly to do at each node expansion. However, performing that computation once for placements we are committed to performing is computationally reasonable.

Of course, the risk remains that the particular plan chosen has no room for improvement, but there is an alternative plan with different subgoals that is much better. We know of no way to do cost optimization for large instances of such problems effectively.

### 3.2.2 Leveraging pairwise ordering information

Assuming the existence of the \textit{arrange} function, we now describe the peephole optimizations in RCHPN.

\textbf{SelectGap}, shown in Algorithm 3, takes a greedy approach to plan reordering. To select the plan step to execute, it finds the preimage, \( g_i \), with the highest index \( i \) such that \( s \in g_i \). This is the plan step that is closest to the end of the plan such that,
were we to begin plan execution from that step, a state satisfying the goal condition would hold. This strategy is similar to idea of executing the “highest true kernel” from the STRIPS system [12]. SelectGap iterates through the rest of the plan calling arrange\((g_i, g_j, s)\) for \(j\) ranging from \(i + 1\) to \(n\). If it returns 1, then we attempt to move \(g_j\) to be directly before \(g_i\). If the resulting plan is valid with respect to the abstract operators’ preconditions, we accept the move and repeat this process with the \(g_j\) as the new “first” subgoal. Otherwise, we undo the change and continue as before. This process terminates when we have checked all the way through the plan without moving any operators. As long as arrange does not have cycles, the process will terminate. In the worst case, we have to do \(O(n^2)\) checks but this is negligible when compared to the complexity of planning, which is exponential in \(n\).

Algorithm 3 Reordering an Abstract Plan

```plaintext
procedure SELECTGAP(s, p, α)
2:   next_subgoal = HighestApplicableSubgoal(p, s)
3:   next_index = p.index(next_subgoal)
4:   highest_checked = next_index
5:   while highest_checked < len(p) do
6:     for (op, sg) in p[next_index:] do
7:       if arrange(next_subgoal, sg, s) = 1 then
8:         new_p = p.move((op, sg), next_index)
9:         if IsValid(new_p) then
10:            p = new_p
11:            next_subgoal = sg
12:            highest_checked = next_index
13:            break
14:       end if
15:     end if
16:   end for
17:   end while
end procedure return sg
```

SelectParallelOps, shown in Algorithm 4, proceeds in a similar fashion. It maintains the next subgoal we will plan for, \(sg\), which is initialized to be the result of SelectGap. It iterates through the rest of the plan, calling arrange\((sg, g_i, s)\) for \(i\) ranging from the index of \(sg\) to \(n\). If arrange returns 2, then we attempt to move
to be directly after \( sg \) in the plan. If the result is a valid plan we combine the \( g_i \) with \( sg \) and set \( sg \) to be the result. To ensure that planning problems considered at the next level are not so large that we cannot solve them, we terminate this process when we have checked through all subgoals or reach a complexity limit on \( sg \). This represents the trade-off between the complexity of planning and quality of the solutions we can hope to achieve. At the moment we do this by placing a cap on the number of tasks we can jointly plan and determined this value empirically for our experiments. Exploring better ways to make this trade-off is an interesting avenue for further research.

\[ \text{Algorithm 4 Combining Subgoals of an Abstract Plan} \]

\[
\text{procedure } \text{SELECTPARALLELOPS}(s, op, sg, p, \alpha) \\
2: \quad \text{next} \_ \text{sg} = sg \\
3: \quad \text{next} \_ \text{index} = p.\text{index}(\text{next} \_ \text{sg}) + 1 \\
4: \quad \text{for } (op, sg) \text{ in } p[\text{next} \_ \text{index}:] \text{ do} \\
5: \quad \quad \text{if } \text{arrange}(\text{next} \_ \text{sg}, \text{sg}, s) = 2 \text{ then} \\
6: \quad \quad \quad \text{new} \_ \text{p} = p.\text{move}((op, sg), \text{next} \_ \text{index}) \\
7: \quad \quad \quad \quad \text{if } \text{IsValid(new} \_ \text{p) then} \\
8: \quad \quad \quad \quad \quad p = \text{new} \_ \text{p} \\
9: \quad \quad \quad \quad \quad \text{next} \_ \text{sg} = \text{CombineGoals(next} \_ \text{sg}, \text{sg}) \\
10: \quad \quad \quad \quad \quad \text{next} \_ \text{index} = \text{next} \_ \text{index} + 1 \\
11: \quad \quad \quad \text{if } \text{MaxComplexity(next} \_ \text{sg) then} \\
12: \quad \quad \quad \quad \quad \text{return next} \_ \text{sg} \\
13: \quad \quad \quad \quad \quad \text{end if} \\
14: \quad \quad \quad \text{end if} \\
15: \quad \text{end for} \\
16: \text{end procedure return next} \_ \text{sg} \\
\]

3.3 Ordering-preference heuristics

Now we consider some principles that can guide the specification of the \text{arrange} function for particular domains. We can frame this task in terms of shared resource consumption. Recall the robot that must put several boxes in a room. In this example, we can treat free space as the important resource. Placing each box uses the space in the entry to that enclosed region. Our difficulty arises because placing box3 does
not free up the resource when this task is complete, but rather consumes the resource in perpetuity. The only way to enable subsequent subtasks to use this resource is to undo that subgoal, which forces us to re-achieve it later. Combining tasks can be viewed in a similar light: moving each box needs to use the robot resource. In this case, the resource in question is shareable so combining these subgoals allows us to take advantage of parallel structure in the sub-plans.

Generalizing from these examples, we can divide the resource use associated with achieving a subgoal into three categories: shareable, contained, and continual. A resource’s use is shareable with respect to a goal if, while it is being used to accomplish that goal, it does not become unavailable. A resource’s use is contained with respect to a goal if it becomes unavailable during the course of achieving that goal but becomes available again after the goal has been achieved. Finally, a resource’s use is continual with respect to a goal if, so long as that goal is true, that resource will be unavailable.

This reduces arrange to two steps: computing an estimate of the resources consumed by achieving each subgoal and classifying the overlapping resource use as shareable, contained, or continual. After this classification is done, determining the correct output from arrange is simple. If two subgoals need the same resource and it is shareable, then they should be combined with the hope that this shared resource will result in parallel structure in the plans and the opportunity for cost savings. If a common resource’s use is continual for one goal and contained for the other, the one with the contained use should be ordered first. If tasks have contained use of all shared resources, then any serialization is acceptable. Note that we should never arrive at a situation where two subgoals require continual use of the same resource as this implies that there is no refinement of this plan and that our hierarchy does not possess the DRP.

There are several strategies for estimating the resources consumed by an operator at abstraction level \(i\). The first is simply to use the resources required by the associated concrete operator. We will refer to this as the 0\(^{th}\) order estimate. In many situations this may be enough. If we wish to make a more informed estimate, we can include the resources required by the hidden preconditions. We can compute a
preimage of the preconditions for level $i - 1$ and keep track of the operators used in that computation. We add to our additional resource estimate by including $0^{th}$ order estimates of resources consumed by those operators. We will consider this a $1^{st}$ order estimate.

We can extend this by going further back at level $i - 1$ and by considering preconditions at level $i - 2$. Thus, a $2^{nd}$ order estimate would use a $0^{th}$ order estimate for a preimage of preconditions at level $i - 2$ and a $1^{st}$ and $0^{th}$ order estimate for operators in the first and second preimages, respectively, of preconditions at level $i - 1$. Note that in calculating these estimates we are doing a limited search for a plan. Trying to compute increasingly complex preimages eventually boils down to solving the full planning problem and will negate any computational savings from hierarchy. We found that $2^{nd}$ order estimates were sufficient for our purposes.
Chapter 4

Related Work

This chapter provides a brief overview of the related work from the planning literature.

4.1 Symbolic planning

The notion of serializable subgoals is due to Korf [23]. He analyzed planning as a knowledge-guided search problem and explored the utility of subgoals in the planning process. He defines serializable subgoals: subgoals that can be planned for sequentially without undoing previous subgoals. In Korf’s terminology our refinement procedure is trying to find orders of operators such the we can serialize the corresponding subgoals. Barrett and Weld explore this issue further and introduce the concept of laboriously serializable subgoals: subgoals for which a non-trivial number of orderings do not serialize [3]. This characterization shares many similarities with the situations in which abstract planning can be inefficient.

Beginning in 2001, the discovery of high-quality domain-independent heuristics for symbolic planning has led to rapid increase in the abilities of symbolic planners to solve classic benchmark problems. Hoffman’s Fast-Forward system, and corresponding heuristic, was the first heuristic planner to show reasonable performance across a wide number of problems [18]. FF makes use of a relaxed planning graph, a planning problem in which no fluents become false as the result of an operator and multiple operators can be applied in parallel, to estimate distances to a goal for forward search.
The forward search algorithm used is a greedy hill-climbing algorithm. To maintain completeness they resort to a more standard backtracking search if that fails.

Fast-Downward improved on the state of heuristic search by doing small searches in an abstract space to create heuristic estimates [15]. It uses a causal graph heuristic to automatically generate abstractions which are not accurate enough for direct hierarchical search, but provide estimates which serve as good heuristics. This system shares the use of abstraction to reduce search with HPN approaches, but we use abstraction for search control rather than using it to get heuristic estimates. It would be interesting to see if the Fast-Downward heuristic, using the already existing hierarchy, could be used to speed up planning at a particular level of abstraction within HPN.

The most recent advance in symbolic planning comes in the form of the LAMA planner and is due to Richter and Westphal [26]. LAMA makes use of ordered landmarks, formulas which must become true at some point along any solution to a planning problem, to define a pseudo-heuristic. The pseudo-heuristic counts the number of landmarks that have not been achieved on this plan and is not a true heuristic because it depends on the search path as well as the state being evaluated. LAMA also integrates cost-optimization into their search. An initial solution is found through greedy hill climbing. Then, a series of weighted A* searches, which find solutions with increasing optimality guarantees, are run until a set time limit expires. LAMA introduces multi-queue heuristic search to use heuristic information from multiple heuristic functions to guide search. While these algorithms have proved quite effective on IPC (International Planning Competition) benchmarks, they do not scale up to the long-horizon problems faced by a robotic agent.

Dornhege et al. attempt to extend classical planning to more complicated domains by using external modules called semantic attachments [11]. These semantic attachments allow designers to specify arbitrary code to test whether a fluent is true in a world state or compute the effects of an action. The effect applicator modules are analogous to our choose functions in operators, except that ours is used for regression planning and theirs for forward chaining. Semantic attachments enable the
planners they consider to avoid fully enumerating complicated effects or fluents for each possible world state. They use these modules to consider a variant of the logistics domain that, similar to the domain used in our experiments, accounts for the geometry of packages. This domain differs from ours in that they do not consider the task of actually placing objects in vehicles and instead only check that there is a feasible packing for the objects being considered.

4.2 Hierarchical planning

Precondition-dropping abstractions in hierarchical planning were first studied by Sacerdoti in his system, ABSTRIPS [27]. Preconditions with lower criticalities were considered details and dropped from initial planning problems. Sacerdoti’s criterion for determining which preconditions were details was the ability to find a short plan to achieve them without violating preconditions from higher levels. ABSTRIPS differs from RCHIPN in that it finds a full plan at each level before refining and does not consider reordering or combining subgoals.

Knoblock provided a more formal definition and analysis of refinement as well as a system, ALPINE, to automatically derive hierarchies [22]. His definition requires that ordering relations between operators must be preserved when refining a plan. He defines an ordered monotonic (OM) refinement as one where new operators do not change any fluents used in the abstract plan. He argues that hierarchies for which all refinements are OM will be effective in problem solving and describes a system which can find OM hierarchies. The drawback of this approach is that, while OM hierarchies are effective, this property can be overly restrictive and many problems may not admit an OM hierarchy.

Bacchus and Yang modeled a hierarchical planner as a branching probabilistic process and analyzed the expected amount of computation as the probability that a particular subproblem could be refined [2]. Their model predicts that abstract planning should be efficient if all subproblems can be refined (i.e. no backtracking across levels of the hierarchy) or if the probability of refinement is very small, as bad plans
are quickly ruled out. They define hierarchies with the *downward refinement property* as hierarchies where every abstract plan can be refined. They define conditions under which this can be achieved and prevent a systems which uses these conditions to improve on APLINE hierarchies.

Nau et al. use a hierarchical task network (HTN) to hierarchically solve planning problems [25]. In their setting, the goals are tasks, which have preconditions and effects, but also specify the possible refinements. The components of these refinements can themselves be abstract tasks. Nau et al. attempt to deal with optimality in several ways. The most prevalent of these does a branch and bound search through the space of task refinements. However, the costs used must be fully specified beforehand, which requires a large amount of work on the part of the system designer. They attempt to interleave abstract tasks, but do so in a blind, non-deterministic way.

Marthi et al. suggest a view of abstract actions centered around upper and lower bounds on reachable sets of states [25]. They use angelic nondeterminism in addition to upper and lower bounds on costs to find optimal plans. They do this both in offline and realtime settings, providing hierarchical versions of A* and LRTA*. These searches amount to heuristic search through the possible refinements of a high level action. Their most effective algorithm, Hierarchical Satisficing Search, is similar to the approach taken in HPN in that it commits to the best high level action which can provably reach the goal within a cost bound. This is beneficial in that execution will only begin if there is a proof that the task can be accomplished within the bound. However, if the abstract level is ambiguous between several plans (i.e. different orderings of the same HLAs), then they may miss an opportunity to reduce cost.

Factored planning generalizes hierarchical planning to decompose a planning problem into several factors. Factors are solved on their own, treating the problems solvable by other factors as abstract actions. A solution for a problem is frequently computed in a bottom-up manner, with factors computing preconditions and effects that they publicize to other factors [1]. These planners exhibit local optimality in that plans within a factor are optimal with respect to that factor but do not make any attempt at global optimality. Furthermore, they have not been shown to scale
up to problems of the size necessary for a real robotics problem.

Srivastava and Kambhampati [29] decompose planning into causal reasoning and resource scheduling. They plan initially in an abstract space where similar entities are treated as the same and are scheduled in a later phase. This decomposition enables them to scale standard planning domains and take advantage of similar objects in a domain (e.g. two different robot hands) without increasing planning time. These approaches are similar to ours in that our heuristics use a similar decomposition. However, our system uses the decomposition to do online execution cost optimization while their system uses this knowledge in order to scale up or optimize a classical planner.

4.3 Partial orders and planning

The use of partial orders in planning is an old idea and dates back to Sacerdoti's NOAH system [28]. Most uses of partial-order planning can be viewed as alternative, non-hierarchical, planning algorithms where the goal is simply to find another plan. The partial order planner that shares the most with our solution is the final version of Prodigy [31]. Prodigy searches by maintaining a totally ordered ‘head’ and partially ordered ‘tail’ for a plan. The state that results from executing the head of the plan is the ‘current’ state. Planning proceeds by adding an operator to the tail or by adding an operator from the tail, whose preconditions are satisfied in the current state, to the head plan. This is similar to interleaving planning with execution because, although the it is simulated, the current state can be used to guide planning for the tail. However, Prodigy solves problems in a single planning step and falls prey to the same types of issues as other non-hierarchical planners.

Bäkström studied the problem of de-ordering or re-ordering a plan [8]. He considers modifying plans to find solutions with fewer constraints or to reduce parallel execution time. He proposes several definitions for an optimal re-ordering and shows that only the simplest of these is tractable to achieve. However, he finds a class of plans for which determining an optimal de-ordering is efficient. Our work implicitly
relies on the de-ordered plan, but does not explicitly compute it. We do re-ordering, but our goal is to minimize the execution cost of a hierarchical planner, which is not a case Bäkström considers.

The closest use of partial orders in planning to RCHPN is due to Hoffman, Porteous, and Sebastia [16]. They use partial orders between landmarks to guide search. Hoffman et al. introduce landmarks and provides techniques for automatically finding landmarks for a planning problem using a planning graph. They define several types of ordering relations between landmarks, one of which, reasonable orders, deals with landmarks that have to be undone and redone if achieved out of order. Subgoals in an abstract plan become landmarks when we consider search at the next level. One way to view the ordering issues we see in hierarchical planning is as violations of reasonable orders. Hoffman et al. treat the landmarks as a partially ordered abstract plan and greedily plan for the closest unachieved landmark. This form of search control is similar to ours, but we use heuristics to select a good subgoal to plan for next.
Chapter 5

Evaluation & Experiments

This chapter defines the marsupial logistics domain and lays out a candidate hierarchical decomposition of this domain. It presents experiments to evaluate the usefulness of RCHPN in marsupial logistics. It concludes with a discussion of the issues associated with learning ordering rules for marsupial logistics and presents some results for learning in a simple context.

5.1 Transportation domain with marsupial robots

We tested the RCHPN approach in a complex transportation domain, which is an extension of a classical abstract logistics domain [32]. The goal is to transport several packages to destination locations. The locations are grouped into cities: trucks can move among locations within a city. Some locations in a city are airports: planes can move among airports. Each truck has a geometrically constrained cargo area and carries a “marsupial” robot. This robot can be thought of as an idealized forklift that can move packages within the cargo area and onto and off of the truck. A plan for transporting a package to a goal location will typically consist of transporting it (in a truck) to an airport, flying it to the correct city, and then transporting it to the goal location. Each time a package is loaded onto or removed from a truck, there will be a detailed motion plan for a forklift. Fig. 5-1 depicts a graphical representation of this domain.
Figure 5-1: Visualization of the Marsupial Logistics Domain. Circles are locations and pink circles are airports. The additional windows represent the loading and storage areas of the vehicles. The red squares represent a marsupial robot which takes care of storing packages for transit. In order for vehicles to move, all packages, as well as the loader, must be on one of the beige squares. Package 2 is about to be unloaded at airport-1 so it can be flown to a destination.
Figure 5-2: The root of a planning tree for a simple problem in the marsupial logistics domain that involves transporting two packages to another location within the same city. At the high level, the Unload operators are recognized as overlapping on a shareable resource (truck) and are combined. In refining Plan 3, the Load operator is determined to overlap with the Unload operator on both the shareable resource of the truck and the contained resource of the truck’s location. It is reordered to be before the first Unload because it is estimated, greedily, as being easier to achieve from the current state. If there was not enough space in the truck, then the truck would not be considered shareable and the ordering would remain unchanged.
The HPN framework supports using real robot kinematics and continuous geometry for managing objects inside the trucks. For efficiency in these experiments, however, we use a simplified version of the geometry in which the cargo hold is discretized into a grid of locations; the robot occupies one grid location and can move in the four cardinal directions. Each "package" takes up multiple cells and is shaped like a Tetris piece. This model retains the critical aspects of reasoning about the details and order of operations within the truck (even determining whether a set of objects can be packed into a truck is, in general, NP-complete [10]). We can also see it as an instance of the navigation among movable obstacles (NAMO) problem in a discrete space [30]. To load a package onto a truck, for example, it might be necessary to move, or even unload and reload other packages that are currently in the truck.

5.1.1 Fluent specification

This section provides a formal description of the fluents used in marsupial logistics. Each fluent specifies a test function which will enable us to determine its truth value in a given world model. SweptVolume is a function that takes a path, package, and grasp as arguments and computes the region that must be clear for a loader to traverse that path holding that package with that grasp.

- \textit{In}(package, vehicle)
  \textit{test}: \text{package} \in \text{vehicle.objects}

- \textit{At}(vehicle, location)
  \textit{test}: \text{vehicle.location} = \text{location}

- \textit{PkgLoc}(package, vehicle, gridLoc)
  \textit{test}: \text{vehicle.objLoc}[\text{package}] = \text{gridLoc}

- \textit{LoaderLoc}(vehicle, gridLoc)
  \textit{test}: \text{vehicle.loaderLoc} = \text{gridLoc}

- \textit{LoaderHolding}(vehicle, package, grasp)
  \textit{test}: \text{vehicle.heldObject} = \text{package} \land \text{vehicle.loaderGrasp} = \text{grasp}

- \textit{ClearPath}(path, grasp, package, vehicle)
  \textit{test}: \forall \text{gridLoc} \in \text{sweptVolume}(\text{path}, \text{grasp}, \text{package}), \neg \text{blocked}(\text{gridLoc}, \text{vehicle})
5.1.2 Operator specification

This section formalizes the operator schemas used for marsupial logistics. Operators are divided into 3 categories: logistics operators, marsupial operators, and inference operators. Logistics operators describe actions for loading and unloading packages into vehicles, as well as moving vehicles between locations. Marsupial operators describe actions for manipulating packages within a vehicle. Inference operators enumerate preconditions for derived predicates; e.g., locations for objects such that a vehicle is packed. They serve to enable our regression based planner to create subgoals for derived predicates. Logistics operators have cost 10, marsupial operators have cost 1, and inference operators have cost 0. We list the resources that primitive operators consume. This listing does not classify resource use as contained, shareable, or continual because those classifications are done with respect to abstract operators and are left up to the arrange function. Operator schemas also include the precondition criticalities that define the hierarchy we used for this domain.

- Load(package, vehicle, location):
  res: In(package, vehicle), PkgLoc(package, vehicle, vehicle.loadLoc)
  pre:
    1. Reachable(location, vehicle), At(package, location)
    2. At(vehicle, location)
    3. Clear(vehicle, loadRegion)
  cost: 10
  consumes: vehicle, vehicle.loadRegion, vehicle.location

- Unload(package, vehicle, location):
  res: At(package, location)
  pre:
1. Reachable(location, vehicle)
2. SameCity(package, vehicle)
3. In(package, vehicle)
4. At(vehicle, location)
5. PkgLoc(package, vehicle, vehicle.loadLoc)

**cost:** 10

**consumes:** vehicle, vehicle.loadRegion, vehicle.location

- **Travel(vehicle, startLoc, resultLoc):**
  
  **res:** At(vehicle, resultLoc)
  
  **pre:**
  
  1. At(vehicle, startLoc), Connected(startLoc, resultLoc, vehicle)
  2. Packed(vehicle)

  **cost:** 10

  **consumes:** vehicle, vehicle.location

- **LoaderGrasp(vehicle, package, grasp, gridLoc):**
  
  **res:** LoaderHolding(vehicle, package, grasp)
  
  **choose:** loaderLoc ∈ GraspLocations(gridLoc, grasp), pickPath ∈ Paths(vehicle.loaderHome, targetLoc)

  **pre:**
  
  1. LegalGrasp(package, grasp, gridLoc, vehicle), ClearPath(pickPath, grasp, package, vehicle), PkgLoc(vehicle, package, gridLoc)
  2. LoaderHolding(vehicle, None, None)
  3. LoaderLoc(vehicle, loaderLoc)

  **cost:** 1

  **consumes:** vehicle.loader, loaderLoc, pickPath

- **LoaderPlace(vehicle, package, gridLoc, grasp):**
  
  **res:** PkgLoc(package, gridLoc)
  
  **choose:** loaderLoc ∈ GraspLocations(gridLoc, grasp), placePath ∈ Paths(vehicle.loaderHome, targetLoc)

  **pre:**
  
  1. LegalGrasp(package, grasp, gridLoc, vehicle), In(package, vehicle)
  2. ClearPath(placePath, gridLoc, grasp, package, vehicle)
  3. LoaderHolding(vehicle, package, grasp)
  4. LoaderLoc(vehicle, loaderLoc)
cost: 1
costumes: vehicle.loader, gridLoc, loaderLoc, placePath

- LoaderMove(vehicle, targetLoc, package, grasp):
  res: LoaderLoc(vehicle, targetLoc)
  choose: path ∈ Paths(vehicle.loaderHome, targetLoc)
  pre:
  1. LegalGrasp(package, grasp, targetLoc, vehicle)
  2. ClearPath(path, grasp, package, vehicle)
  3. LoaderHolding(package, grasp, vehicle)

consumes: vehicle.loader, p

- SameCity(package, vehicle):
  res: SameCity(package, vehicle)
  choose: loc ∈ ReachableLocs(vehicle)
  pre:
  1. ∅
  2. At(package, loc)

  cost: 0
  consumes: vehicle, package

- Pack(vehicle):
  res: Packed(vehicle)
  choose: loc[0] ∈ vehicle.storageRegion ∀ pkg s.t. In(pkg, vehicle),
  loaderLoc ∈ vehicle.storageRegion
  pre:
  1. ∅
  2. PkgLoc(pkg, vehicle, loc[pkg]) ∀ pkg s.t. In(pkg, vehicle)
  3. LoaderLoc(vehicle, loaderLoc)

  cost: 0
  consumes: vehicle, vehicle.storageRegion

- ClearPath(path, grasp, package, vehicle)
  res: ClearPath(path, grasp, package, vehicle)
  choose: loc[pkg] ∈ vehicle.storageRegion ∀ pkg s.t. overlaps(pkg, path)
  pre:
  1. ∅
  2. PkgLoc(pkg, vehicle, loc[pkg]) ∀ pkg s.t. In(pkg, vehicle)

  cost: 0
  consumes: vehicle.loader
5.2 Experiments and results

We designed experiments to compare a classical non-hierarchical planner called FF [18], HPN, and RCHPN. FF is a fast, easy-to-use classical planning algorithm. However, even small instances of the marsupial transportation domain are intractable for FF. To demonstrate this, we ran FF on an instance with 8 locations, 2 of which were airports; a single truck per airport; one plane; and a single package which occupied a single location on the grid. The package needed to be transported from a location to the airport it was not connected to. Even on this problem, FF took slightly less than 7.5 hours to find a solution of length 62. The pddl domain and problem files, as well as the solution FF found, are shown in Appendix A. There have been improvements in this class of planners [26], but they cannot ultimately address the fundamental problem that we need to search over a long horizon with a large branching factor to solve even the simplest problems in this domain.

We altered the basic HPN algorithm so that it solves easy problems more quickly at the cost of a small increase in computation time on other problems. Given a conjunctive goal, we first check for the existence of a plan for a random serialization of the fluents; this will succeed very quickly in problems with many goals that are independent at the current level of abstraction and usually fails quickly otherwise. If it fails, we search for a monotonic plan (one that never causes a goal fluent that is already true to be made false). Should we fail to find a monotonic plan, we execute a standard backward search. These are standard modifications to backchaining planners and do not affect the overall correctness of the algorithm [12].

At the lowest levels of abstraction, we use a motion planner to determine detailed placements of packages and motions of the robot. The motion planner could be something like an RRT in the continuous configuration space of robot and packages [24]; in this work, it is an implementation of $A^*$ in the discretized geometry of a truck. The ability to elegantly use a specialized planner to solve sub-problems is a benefit of the HPN approach and is key to its ability to tractably solve problems in this domain. Work on integrating modules into standard symbolic architectures attempts
to enable similar benefits for standard symbolic planners, but places restrictions on the types of planners that can be used [11].

In designing the arrange function, we must determine the resources used by abstract versions of operators and categorize those resources as shareable, contained, or continual. Our implementation considers rearranging abstract versions of Load, Unload, and SameCity. Other operators only appear lower in the hierarchy and the plans they appeared in were frequently quite constrained; the computational effort to reorder them is not worth it.

We estimated the abstract resource use of Load with a \(0^{th}\) order estimate. For SameCity, we used a \(1^{st}\) order estimate. This allowed us to expose the resources used to unload a package in this particular city. We used a \(2^{nd}\) order estimate for abstract Unloads. We estimated the resource use to include the implicit SameCity precondition. At lower levels in the hierarchy, we consider the free space resource used in placing a package in the load region. We do not consider free space earlier because, unless we know where a package is within the vehicle, it is hard to make any useful assessment of this resource. This illustrates the utility of optimizing in the now; we can postpone optimization as well as planning.

We adopted the convention that a vehicle resource was shareable for two goals if there was an arrangement of packages, including packages mentioned in the goal that fit in the vehicle. We estimated this with a greedy method that iteratively placed packages as far towards the back of the vehicle as possible, preferring placements towards the sides as a tiebreaker. An example execution of a simple plan with reordering and combining of subgoals is illustrated in Fig. 5-2.

We defined a distribution over planning problems within this domain and tested on samples from that distribution. Each instance had 5 airports, with 4 locations connected to each airport. The layout of each airport and connected locations was randomly selected from a class of layouts: circular (roads between locations are connected in a circle), radial (each location is directly connected to the airport, but not to other locations), linear (the same as circular with one connection dropped), and connected (each location was connected to each other location). There was a single
Figure 5-3: Average percent decrease in plan cost vs. problem size for RCHPN vs. HPN. This figure depicts results across four experimental conditions. 'Multiple Origin/Multiple Destination' shows the execution cost savings when there is little room for improvement. As expected it shows very little difference between HPN and RCHPN. 'Single Origin/Single Destination' depicts execution cost reductions when there is a large potential for execution cost savings. In this setting we see up to 30% improvements. The 'Single Origin/Multiple Destination' and 'Multiple Origin/Few Destination' conditions show an intermediate between the other two conditions. As expected, they also show an intermediate reduction in cost.

We ran experiments in four different regimes: maximal parallel structure among tasks, parallel structure in the destination of packages only, parallel structure in the origin of packages only, and finally little to no parallel structure. This enables us to evaluate the performance of RCHPN in situations where there is a large opportunity for execution cost savings and in situations where there is little to no room for improve-
ment. To do this, we varied the number of potential start locations and destinations for packages from a single option to a uniform selection from all locations in the domain. We collected data for tasks with 3 to 10 packages and averaged results across 10 trials. For a particular problem we ran both HPN and RCHPN and computed the ratio of the costs (the sum of the costs of the all of the primitive operators executed during the run) and averaged these ratios across 10 independent runs.

For problems with a single origin and destination, RCHPN achieved an average of 30% improvement, roughly independent of problem size. When either the origin or destination was dispersed, the average improvement dropped to about 15%. In this case, smaller problems typically saw less improvement than larger ones. This is because the more packages, the more likely it is that there is some structure the planner will be able to take advantage of. Fig. 5-3 depicts average decrease in execution cost vs. problem size for our testing regimes.

Our heuristics only apply to packages going from and to similar locations, so when both package origins and package destinations are distributed widely we expect to see little improvement. This was borne out in our results, as the multiple origin, multiple destination experiments saw 5% improvement. However, while we saw little to no improvement in execution cost on those runs, we also saw little to no increase in planning time. This illustrates the utility of doing peephole optimization outside of the main planning loop. Our solution will spend a small amount of time at the abstract level looking for parallel structure, but if none is found, it proceeds with planning as normal and incurs a modest overhead. Fig. 5-4 shows this relationship in detail.

5.3 Learning ordering and combining rules

One of the upsides of RCHPN is that the heuristics used to perform the optimization, though domain specific, are usually quite simple. This creates a hope that it will be possible to learn these rules from experience. This amounts to learning the arrange function, as actual control decisions can be deduced from that. This task is made
Figure 5-4: Plot of percent decrease in execution cost vs. percent increase in planning time. Points are color coded to correspond to the data series from Fig. 5-3. The positive correlation between the two highlights a useful property of optimizing in a post-processing step: on problems where there is little parallel structure, our modifications have little to no effect on the planning time. As more parallel structure is introduced, more planning time is spent utilizing that structure. In some cases planning time decreased slightly. This is a result of non-determinism in the planner.

difficulty by the need to learn rules at multiple levels in the hierarchy and issues with effectively representing world state.

Interaction between rules at different levels in our hierarchy is one of the key challenges in approaching this learning problem. We would like to learn rules independently or at least sequentially, learning first at either the highest or lowest levels and progressing appropriately. Unfortunately, this decomposition is likely to lead to issues. Many control rules at the high level will only result in good performance if the appropriate ordering rules can be applied at lower levels. For example, the rule ‘unload packages going to the same destination’ is only a good idea if lower levels in the hierarchy know to load both packages before unloading them. However, the problem distribution we train with at more concrete levels is defined by the control rules at more abstract levels. We only know to train lower levels to deal with this type of rule if we already have that rule at higher levels. A possible way to approach
this is through use of coordinate ascent, alternating learning at multiple levels.

Our other key issue is one of representation. The underlying world models for HPN can be quite complex and large. Representing this compactly while capturing enough information to enable learning is a difficult task. One approach is to make use of the logical formalism used for planning. There is some work done on performing statistical learning with logical representations and we already need to create this representation for planning [13]. However, these learning techniques usually assume that we are always learning in the same logical world, so objects in the world are the same across training instances. We would like to be able to learn control rules for a domain, where the problem instance can vary according to an arbitrary probability distribution. Enabling this in general requires finding correspondences between objects in different, although similar, logical worlds. There are some kernels, such as the pyramid match kernel, which may be able to cope with this [14].

5.3.1 Learning Experiments

We did some initial experiments with learning for this setting. We focused on input representation issues and restricted ourselves to learning control rules at the highest level of the hierarchy, using hand-coded rules for other abstraction levels. We kept the road networks and locations constant across training instances, with 2 airports and 3 locations per airport, but allowed the number and types of packages to vary. We re-named packages in descending order of distance (in the road network) to the goal to avoid the issue of determining object correspondences. We used the truth values of logical fluents as our basic features, and augmented our features was a binary matrix that contained contradiction information about each pair of features.

We employed a two-tiered classification strategy with a dual SVM as the underlying classification method and used the libSVM software package to run our experiments [7, 9]. We first trained a classifier to determine when combining subgoals was beneficial, and trained a different classifier to make subsequent ordering decisions. We trained on 108 different problem instances. We found that a polynomial kernel of degree 3 gave the best performance.
We evaluated our learned rules through leave one out cross-validation and compared to our hand-coded heuristics and an optimal decision strategy that plans for each valid ordering or combination separately and selects the option that minimizes the execution cost. This strategy is computationally infeasible for any reasonably sized task but gives insight into the best we could hope to do. Overall, the results were positive. Across the test set, selecting the best option every time results in a total execution cost of 20173 units. Our hand-coded heuristics incur an additional 572 units of cost over the optimal solution: a 2.8% increase. Our learned control function saw an increase of 247 units compared to the optimal solution. Although it remains to be seen if these results generalize to more complicated scenarios, this is a 56% improvement in the additional cost incurred and is a promising result. Note that we are able to get this close to ‘optimal’ because we are only comparing control strategies at the highest level of abstraction; all of the solutions being compared use the same planning rules and heuristics at more concrete levels.
Chapter 6

Conclusion and Future Directions

If our goal is to build agents that can execute tasks in real-world long-horizon settings we need a way to efficiently select actions despite the large problem size and PSPACE-complete nature of this problem. Aggressive hierarchical planning proposes that we solve this by breaking up a single large problem into many small problems that can be solved sequentially.

Hierarchical Planning in the Now (HPN) provides a way to use temporal hierarchy for large planning problems. It interleaves planning with execution in order to reduce the complexity associated with estimating the results of an abstract action. This property also lets HPN extend naturally to the partially-observable case [20]. Interleaved planning and execution is done by committing to an initial abstract plan and incrementally refining it, executing the first step in the abstract plan before refining the next.

While this strategy is effective at finding solutions, it make no claims to the quality of solutions produced. There is no cost model at abstract planning levels, so abstract solutions can commit HPN-style planners to executing plans with very poor execution cost. A common failure mode comes from incorrectly ordering subgoals so that they do not serialize: subgoals are achieved, only to be undone and re-achieved in planning and executing subsequent subgoals. Additionally, HPN-style planners commit to executing each subgoal in a abstract plan sequentially. This keeps the planning horizon short, but can also hide potential cost savings if there is parallel
structure in subtasks. Considering subtasks jointly, in certain scenarios, enables a hierarchical planner to find shorter solutions and leverage this structure.

A hierarchical planning process does not generally have enough information to make these ordering and combining decisions. Building cost estimation into the planning loop will likely result in unacceptable slowdown. We presented RCHPN, a modification of HPN that considers re-ordering and combining subtasks in a post-processing step, to address these concerns and perform execution cost optimization. RCHPN uses a pairwise comparison function to make ordering decisions 'in the now' interleaving optimization with planning and execution. This lets optimization procedures utilize more details about the current world state and perform more expensive computation when compared with optimization done purely at planning time.

We provided guidelines for writing heuristics to control optimization. We frame the underlying issue as one of shared resource use and classify the resource use of an abstract operator into one of three categories: shareable, contained, or continual. From this classification it is straightforward to make the corresponding combining or re-ordering decision.

To evaluate RCHPN we introduced the marsupial logistics domain: a modification of the IPC logistics domain that introduces operators for manipulating and packing packages. This modification results in a large increase in the state space and branching factor for similar problems and renders even simple problems intractable for state-of-the-art symbolic planning algorithms. We presented a precondition dropping hierarchy that lets us to solve large problems in this domain.

We implemented re-ordering and combining heuristics for this domains and compared the performance of RCHPN and HPN with respect to planning time and execution cost across a wide variety of large marsupial logistics problems. We found that, when the opportunity is there, RCHPN is able to leverage parallel structure in subtasks and make ordering decisions to reduce execution cost by up to 30%. When there is little opportunity for cost savings, we find that the running time of RCHPN stays very close to the running time of non-optimizing HPN.

We examined the problem of learning control rules for RCHPN. We found that two
key obstacles in this problem are dependence between learning tasks at different levels of the hierarchy and input representation. We performed experiments training an SVM to make control decisions at the highest level of abstraction for small problems and were able to outperform hand-coded heuristics. However, as these experiments did not deal with learning at multiple levels and had a fairly simple problem distribution, it is still an open problem to determine if learning control rules is tractable in this setting. In addition to applying learning, there are several interesting research directions this work points toward. We conclude with a brief discussion of some of these.

6.1 Avenues for future research

One interesting direction for further research is investigating the task of deriving ordering or combining relations automatically. The inspiration for this line of research comes from the similarity between reasonable orderings for landmarks and the ordering failure modes we see in hierarchical planners. There are automated mechanisms to extract landmarks and corresponding orderings from planning problems. One of the main tasks here is in determining the relationship between abstract subgoals and landmarks. Once we commit to abstract subgoals, they will become true at some point in any plan we can find, so the concepts are likely related. If we could apply automated techniques to determine orderings between subgoals we would obviate the need for learning, or hand-coding, heuristics.

In order to apply landmark extraction techniques from the literature we would need to build a compact purely symbolic abstraction of our subproblems. This is not a simple task and is an interesting research avenue in itself. Being able to build these symbolic abstractions for complex and large domains would enable more flexibility in the planner used to solve subproblems within HPN as well. This would enable HPN-style planners to benefit from positive results in forward heuristic search, embodied in planners like FF, Fast-Downward, and LAMA. Such a representation does exist for any problem HPN can solve; simply write down all actions considered by the
back-chaining planner. The challenge is to determine this \textit{without} first solving the planning problem. This is analogous to the approaches taken in sampling-based motion planning, where a small, discrete, representation, which is sufficient for planning, is extracted from a large continuous domain [24, 21].

A final direction for future research is the extension of the class of plans we consider in our optimization. In addition to combining and reordering subgoals, there are potential cost savings to be found in rebinding variables in an abstract plan. An example application might be jointly optimizing the placement of two objects within a region subject to maintaining the correctness of the corresponding plan. Similar arguments to those made about re-ordering and combining apply to the case of finding a good binding of a variable with respect to execution cost. Adding this capability to hierarchical planners makes the job of a system designer easier; the variable binding that occurs during planning need only concern itself with finding a solution and can pay less attention to the execution cost ramifications of that selection.
Appendix A

PDDL for Marsupial Logistics

The appendix contains the pddl domain and problem file used to test FF's performance on the marsupial logistics problem. It also includes FF's output from running on these files.

A.1 Domain

;; marsupial-logistics domain
;;
;; ;; Dylan Hadfield-Menell 9/2012

(define (domain marsupial-logistics)
  (:requirements :strips :typing)
  (:types PACKAGE TRUCK LOCATION AIRPLANE AIRPORT PLACE DIR)
  (:constants N E W S - DIR
   ;; grid places
   p00 p10 p20 p30 p40
   p01 p11 p21 p31 p41
   p02 p12 p22 p32 p42
   p03 p13 p23 p33 p43
   p04 p14 p24 p34 p44
   p05 p15 p25 p35 p45
   p06 p16 p26 p36 p46
   p07 p17 p27 p37 p47
   p08 p18 p28 p38 p48
   p09 p19 p29 p39 p49 - place)

  (:predicates
(at ?obj ?loc)
(in ?obj ?vehicle)
(exists-road ?loc1 ?loc2)
(at-grid ?x ?v ?y)
(loader-at ?p ?v)
(adj ?d ?x ?y ?v); ?y is to the ?dir of ?x in ?v
(holding ?x ?v)
(clear ?x ?v)
(free ?v)
)

;;; Marsupial actions
(:action
  move
  :parameters (?d - dir ?rp1 ?rp2 - place ?truck - (either TRUCK AIRPLANE))
  :precondition
  (and (loader-at ?rp1 ?truck)
   (adj ?d ?rp1 ?rp2 ?truck)
   (free ?truck)
   (clear ?rp2 ?truck)
  )
  :effect (and (loader-at ?rp2 ?truck) (not (loader-at ?rp1 ?truck))
          (not (clear ?rp2 ?truck)) (clear ?rp1 ?truck))
  )

(:action
  grasp
  :parameters (?h - PACKAGE ?d - dir ?rp ?hp - place
               ?truck - (either TRUCK AIRPLANE))
  :precondition
  (and
   (free ?truck)
   (in ?h ?truck)
   (at-grid ?h ?hp ?truck)
   (loader-at ?rp ?truck)
   (adj ?d ?rp ?hp ?truck))
  :effect (and (holding ?h ?truck) (not (free ?truck)))
  )

(:action
  ungrasp
  :parameters (?h - PACKAGE ?truck - (either TRUCK AIRPLANE))
  :precondition
  (holding ?h ?truck)
  :effect (and (not (holding ?h ?truck)) (free ?truck))
  )

(:action

62
move_1_H
?truck - (either TRUCK AIRPLANE))
:precondition
(and (holding ?h ?truck)
(loader-at ?rp1 ?truck)
(adj ?d ?rp1 ?rp2 ?truck)
(at-grid ?h ?hp1 ?truck)
(adj ?d ?hp1 ?hp2 ?truck)
(clear ?rp2 ?truck)
(clear ?hp2 ?truck)
)
:effect (and (loader-at ?rp2 ?truck) (not (loader-at ?rp1 ?truck))
(clear ?hp1 ?truck) (not (clear ?hp2 ?truck))
(clear ?rp1 ?truck) (not (clear ?rp2 ?truck)))

;; Move when ?hp1 = ?rp2 (e.g. move N when grasp N)
;; Pushing the grasped block ahead the robot
(:action
move_2_H
?truck - (either TRUCK AIRPLANE))
:precondition
(and (holding ?h ?truck)
(loader-at ?rp1 ?truck)
(adj ?d ?rp1 ?rp2 ?truck)
(at-grid ?h ?rp2 ?truck)
(adj ?d ?rp2 ?hp2 ?truck)
(clear ?hp2 ?truck)
)
:effect (and (loader-at ?rp2 ?truck) (not (loader-at ?rp1 ?truck))
(clear ?rp1 ?truck) (not (clear ?hp2 ?truck)))
)

;;;Logistics actions
(:action LOAD-TRUCK
:parameters
(?obj - PACKAGE
?truck - TRUCK
?loc - (either LOCATION AIRPORT))
:precondition
(and (at ?truck ?loc)
(at ?obj ?loc)
(clear p00 ?truck))
:effect
(and (not (at ?obj ?loc))
(in ?obj ?truck)
(at-grid ?obj p00 ?truck)
(not (clear p00 ?truck))))

(:action LOAD-AIRPLANE
:parameters
(?obj - PACKAGE
 ?airplane - AIRPLANE
 ?loc - AIRPORT)
:precondition
(and
(at ?obj ?loc)
(at ?airplane ?loc)
(clear p00 ?airplane))
:effect
(and (not (at ?obj ?loc))
(in ?obj ?airplane)
(at-grid ?obj p00 ?airplane)
(not (clear p00 ?airplane))))

(:action UNLOAD-TRUCK
:parameters
(?obj - PACKAGE
 ?truck - TRUCK
 ?loc - (either LOCATION AIRPORT))
:precondition
(and (at ?truck ?loc)
(in ?obj ?truck)
(at-grid ?obj p00 ?truck)
(not (holding ?obj ?truck)))
:effect
(and (not (in ?obj ?truck))
(at ?obj ?loc)
(clear p00 ?truck)
(not (at-grid ?obj p00 ?truck))))

(:action UNLOAD-AIRPLANE
:parameters
(?obj - PACKAGE
 ?airplane - AIRPLANE
 ?loc - AIRPORT)
:precondition
(and (in ?obj ?airplane)
(at ?airplane ?loc)
(at-grid ?obj p00 ?airplane)
(not (holding ?obj ?airplane)))
:effect
(and
(not (in ?obj ?airplane))
(at ?obj ?loc)
(clear p00 ?airplane)
(not (at-grid ?obj p00 ?airplane)))

(:action DRIVE-TRUCK
:parameters
(?truck - TRUCK
?loc-from - (either LOCATION AIRPORT)
?loc-to - (either LOCATION AIRPORT))
:precondition
(and (at ?truck ?loc-from)
(or (exists-road ?loc-from ?loc-to) (exists-road ?loc-to ?loc-from))
(clear p00, ?truck)(clear p10, ?truck)(clear p20, ?truck)
(clear p02, ?truck)(clear p12, ?truck)(clear p22, ?truck)
:effect
(and (not (at ?truck ?loc-from))
(at ?truck ?loc-to)))

(:action FLY-AIRPLANE
:parameters
(?airplane - AIRPLANE
?loc-from - AIRPORT
?loc-to - AIRPORT)
:precondition
(and (at ?airplane ?loc-from)
:effect
(and (not (at ?airplane ?loc-from))
(at ?airplane ?loc-to)))
)

A.2 Problem instance

(define (problem augmented-strips)
 (:domain marsupial-logistics)

(65)
(:objects truck1 - TRUCK
boston-loc0 - LOCATION
boston-loc1 - LOCATION
boston-loc2 - LOCATION
boston-loc3 - LOCATION
boston-loc4 - LOCATION
boston-loc5 - LOCATION
boston-arpt - AIRPORT
boston-truck - TRUCK
sf-loc5 - LOCATION
sf-truck - TRUCK
sf-arpt - AIRPORT

plane - AIRPLANE
pkg - PACKAGE)

(:init
(at boston-truck boston-arpt)
(at pkg boston-loc5)
(at plane boston-arpt)

(loader-at p00 boston-truck)(clear p10 boston-truck)(clear p20 boston-truck)
(clear p01 boston-truck)(clear p11 boston-truck)(clear p21 boston-truck)
(clear p02 boston-truck)(clear p12 boston-truck)(clear p22 boston-truck)
(clear p03 boston-truck)(clear p13 boston-truck)(clear p23 boston-truck)
(clear p04 boston-truck)(clear p14 boston-truck)(clear p24 boston-truck)
(clear p05 boston-truck)(clear p15 boston-truck)(clear p25 boston-truck)
(clear p06 boston-truck)(clear p16 boston-truck)(clear p26 boston-truck)
(clear p07 boston-truck)(clear p17 boston-truck)(clear p27 boston-truck)
(clear p08 boston-truck)(clear p18 boston-truck)(clear p28 boston-truck)

(adj E p00 p10 boston-truck)(adj N p00 p01 boston-truck);p00
(adj E p01 p11 boston-truck)(adj N p01 p02 boston-truck)
(adj S p01 p00 boston-truck);p01
(adj E p02 p12 boston-truck)(adj N p02 p03 boston-truck)
(adj S p02 p01 boston-truck);p02
(adj E p03 p13 boston-truck)(adj N p03 p04 boston-truck)
(adj S p03 p02 boston-truck);p03
(adj E p04 p14 boston-truck)(adj N p04 p05 boston-truck)
(adj S p04 p03 boston-truck);p04
(adj E p05 p15 boston-truck)(adj N p05 p06 boston-truck)
(adj S p05 p04 boston-truck);p05
(adj E p06 p16 boston-truck)(adj N p06 p07 boston-truck)
(adj S p06 p05 boston-truck);p06
(adj E p07 p17 boston-truck)(adj N p07 p08 boston-truck)
(exists-road boston-loc0 boston-loc1)
(exists-road boston-loc1 boston-loc2)
(exists-road boston-loc2 boston-loc3)
(exists-road boston-loc3 boston-loc4)
(exists-road boston-loc4 boston-loc5)
(exists-road boston-loc5 boston-arpt)

(free boston-truck)

(at sf-truck sf-arpt)
(loader-at p00 sf-truck)(clear p10 sf-truck)(clear p20 sf-truck)
(adj W p23 p13 sf-truck)(adj N p23 p24 sf-truck)
(adj S p23 p22 sf-truck);p23
(adj S p24 p23 sf-truck);p24
(adj S p25 p24 sf-truck);p25
(adj W p26 p16 sf-truck)(adj N p26 p27 sf-truck)
(adj S p26 p25 sf-truck);p26
(adj W p27 p17 sf-truck)(adj N p27 p28 sf-truck)
(adj S p27 p26 sf-truck);p27
(adj W p28 p18 sf-truck)(adj S p28 p27 sf-truck);p28
(free sf-truck)

(loader-at p00 plane)(clear p10 plane)(clear p20 plane)(clear p30 plane)
(clear p01 plane)(clear p11 plane)(clear p21 plane)(clear p31 plane)
(clear p02 plane)(clear p12 plane)(clear p22 plane)(clear p32 plane)
(clear p03 plane)(clear p13 plane)(clear p23 plane)(clear p33 plane)
(clear p04 plane)(clear p14 plane)(clear p24 plane)(clear p34 plane)
(clear p05 plane)(clear p15 plane)(clear p25 plane)(clear p35 plane)
(clear p06 plane)(clear p16 plane)(clear p26 plane)(clear p36 plane)
(clear p07 plane)(clear p17 plane)(clear p27 plane)(clear p37 plane)
(clear p08 plane)(clear p18 plane)(clear p28 plane)(clear p38 plane)
(clear p09 plane)(clear p19 plane)(clear p29 plane)(clear p39 plane)
(free plane)

(adj E p00 p10 plane)(adj N p00 p01 plane);p00
(adj E p01 p11 plane)(adj N p01 p02 plane)
(adj S p01 p00 plane);p01
(adj E p02 p12 plane)(adj N p02 p03 plane)
(adj S p02 p01 plane);p02
(adj E p03 p13 plane)(adj N p03 p04 plane)
(adj S p03 p02 plane);p03
(adj E p04 p14 plane)(adj N p04 p05 plane)
(adj S p04 p03 plane);p04
(adj E p05 p15 plane)(adj N p05 p06 plane)
(adj S p05 p04 plane);p05
(adj E p06 p16 plane)(adj N p06 p07 plane)
(adj S p06 p05 plane);p06
(adj E p07 p17 plane)(adj N p07 p08 plane)
(adj S p07 p06 plane);p07
(adj E p08 p18 plane)(adj S p08 p07 plane);p08
(adj E p10 p20 plane)(adj W p10 p00 plane)
(adj N p10 p11 plane);p10
(adj E p11 p21 plane)(adj W p11 p01 plane)
(adj N p11 p12 plane)(adj S p11 p10 plane);p11
(adj E p12 p22 plane)(adj W p12 p02 plane)
(adj N p12 p13 plane)(adj S p12 p11 plane);p12
(adj E p13 p23 plane)(adj W p13 p03 plane)
(adj N p13 p14 plane)(adj S p13 p12 plane);p13
(adj E p14 p24 plane)(adj W p14 p04 plane)
(adj N p14 p15 plane)(adj S p14 p13 plane);p14
(adj E p15 p25 plane)(adj W p15 p05 plane)
(adj N p15 p16 plane)(adj S p15 p14 plane);p15
(adj E p16 p26 plane)(adj W p16 p06 plane)
(adj N p16 p17 plane)(adj S p16 p15 plane);p16
(adj E p17 p27 plane)(adj W p17 p07 plane)
(adj N p17 p18 plane)(adj S p17 p16 plane);p17
(adj E p18 p28 plane)(adj W p18 p08 plane)
(adj S p18 p17 plane);p18
(adj W p20 p10 plane)(adj N p20 p21 plane);p20
(adj W p21 p11 plane)(adj N p21 p22 plane)
(adj S p21 p20 plane);p21
(adj W p22 p12 plane)(adj N p22 p23 plane)
(adj S p22 p21 plane);p22
(adj W p23 p13 plane)(adj N p23 p24 plane)
(adj S p23 p22 plane);p23
(adj W p24 p14 plane)(adj N p24 p25 plane)
(adj S p24 p23 plane);p24
(adj W p25 p15 plane)(adj N p25 p26 plane)
(adj S p25 p24 plane);p25
(adj W p26 p16 plane)(adj N p26 p27 plane)
(adj S p26 p25 plane);p26
(adj W p27 p17 plane)(adj N p27 p28 plane)
(adj S p27 p26 plane);p27
(adj W p28 p18 plane)(adj S p28 p27 plane);p28
)

(:goal (at pkg sf-arpt))
)

A.3 FF output

step 0: MOVE N P00 P01 PLANE
1: MOVE N P01 P02 PLANE
2: MOVE N P02 P03 PLANE
3: MOVE N P03 P04 PLANE
4: MOVE N P04 P05 PLANE
5: MOVE N P00 P01 BOSTON-TRUCK
6: MOVE N P01 P02 BOSTON-TRUCK
7: MOVE N P02 P03 BOSTON-TRUCK
8: MOVE N P03 P04 BOSTON-TRUCK
9: MOVE N P04 P05 BOSTON-TRUCK
10: MOVE E P05 P15 BOSTON-TRUCK
11: DRIVE-TRUCK BOSTON-TRUCK BOSTON-ARPT BOSTON-LOC5
12: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC5 BOSTON-LOC4
13: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC4 BOSTON-LOC3
14: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC3 BOSTON-LOC2
15: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC2 BOSTON-LOC1
16: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC1 BOSTON-LOC0
17: LOAD-TRUCK PKG BOSTON-TRUCK BOSTON-LOC0
18: MOVE S P15 P14 BOSTON-TRUCK
19: MOVE S P14 P13 BOSTON-TRUCK
20: MOVE S P13 P12 BOSTON-TRUCK
21: MOVE S P12 P11 BOSTON-TRUCK
22: MOVE S P11 P10 BOSTON-TRUCK
23: GRASP PKG W P10 P00 BOSTON-TRUCK
24: MOVE_1_H N PKG P10 P11 P00 P01 BOSTON-TRUCK
25: MOVE_1_H N PKG P11 P12 P01 P02 BOSTON-TRUCK
26: MOVE_1_H N PKG P12 P13 P02 P03 BOSTON-TRUCK
27: MOVE_1_H N PKG P13 P14 P03 P04 BOSTON-TRUCK
28: MOVE_1_H N PKG P14 P15 P04 P05 BOSTON-TRUCK
29: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC0 BOSTON-LOC1
30: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC1 BOSTON-LOC2
31: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC2 BOSTON-LOC3
32: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC3 BOSTON-LOC4
33: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC4 BOSTON-LOC5
34: DRIVE-TRUCK BOSTON-TRUCK BOSTON-LOC5 BOSTON-ARPT
35: MOVE_1_H S PKG P15 P14 P05 P04 BOSTON-TRUCK
36: MOVE_1_H S PKG P14 P13 P04 P03 BOSTON-TRUCK
37: MOVE_1_H S PKG P13 P12 P03 P02 BOSTON-TRUCK
38: MOVE_1_H S PKG P12 P11 P02 P01 BOSTON-TRUCK
39: MOVE_1_H S PKG P11 P10 P01 P00 BOSTON-TRUCK
40: UNGRASP PKG BOSTON-TRUCK
41: UNLOAD-TRUCK PKG BOSTON-TRUCK BOSTON-ARPT
42: MOVE E P05 P15 PLANE
43: LOAD-AIRPLANE PKG PLANE BOSTON-ARPT
44: MOVE S P15 P14 PLANE
45: MOVE S P14 P13 PLANE
46: MOVE S P13 P12 PLANE
47: MOVE S P12 P11 PLANE
48: MOVE S P11 P10 PLANE
49: GRASP PKG W P10 P00 PLANE
50: MOVE_1_H N PKG P10 P11 P00 P01 PLANE
51: MOVE_1_H N PKG P11 P12 P01 P02 PLANE
52: MOVE_1_H N PKG P12 P13 P02 P03 PLANE
53: MOVE_1_H N PKG P13 P14 P03 P04 PLANE
54: MOVE_1_H N PKG P14 P15 P04 P05 PLANE
55: FLY-AIRPLANE PLANE BOSTON-ARPT SF-ARPT
56: MOVE_1_H S PKG P15 P14 P05 P04 PLANE
57: MOVE_1_H S PKG P14 P13 P04 P03 PLANE
58: MOVE_1_H S PKG P13 P12 P03 P02 PLANE
59: MOVE_1_H S PKG P12 P11 P02 P01 PLANE
60: MOVE_1_H S PKG P11 P10 P01 P00 PLANE
61: UNGRASP PKG PLANE
62: UNLOAD-AIRPLANE PKG PLANE SF-ARPT

0.04 seconds instantiating 6144 easy, 42 hard action templates
0.18 seconds reachability analysis, yielding 4879 facts and 4177 actions
0.00 seconds creating final representation with 243 relevant facts
0.01 seconds building connectivity graph
26839.73 seconds searching, evaluating 3621458 states, to a max depth of 8
26839.96 seconds total time
Bibliography


