Optimization in the Now: Dynamic Peephole Optimization for Hierarchical Planning

Dylan Hadfield-Menell, Leslie Pack Kaelbling and Tomás Lozano-Pérez*

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 recent advances using hierarchical planning [1] have been able to provide leverage on this problem. Unfortunately, this approach makes no effort to account for the execution cost of an abstract plan and often arrives at poor quality plans. This paper outlines a method for dynamically improving a hierarchical plan 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 ran experiments in challenging domains and observed up to 30% reduction in execution cost when compared with a standard hierarchical planner.**

I. INTRODUCTION

We want to design robots that can perform complex tasks in real-world domains, such as households and hospitals, over increasingly long time horizons. This involves solving large instances of problems that are, in general, PSPACE-complete. Inspired by human ability to cope with this seemingly intractable problem, we believe that there is some underlying structure or simplicity in these problems that provides a mechanism for reducing complexity.

One way to reduce complexity in long-horizon problems is to use temporal hierarchy to decompose the problem into multiple shorter-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 [1]. HPN provides an interleaved planning and execution strategy that can be shown to be correct and complete for a large class of hierarchical system specifications.

HPN 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 sub-goal.

The fundamental difficulty is that, at the upper levels of the hierarchical planning process, the model does not specify the cost of taking actions. Specifying the cost of abstract operators can be very difficult: it may be highly variable depending on the situation in which the operator is executed, and determining the 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 packages currently in their cargo holds, and the package of interest's current location. It also depends on which abstract operations are sequenced before it. 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 robotic planning that addresses plan quality by dynamically reordering and grouping subgoals in an abstract plan. We re-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.

In addition, 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 HPN. Furthermore, on problems with little room for optimization, we find that our approach results in only a negligible increase in planning time.

II. BACKGROUND

We use a relatively standard symbolic representation for planning operators, derived from STRIPS [2] 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

*Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 32 Vassar Street, Cambridge, MA 02139, USA {djhm, lpk, tlp}@mit.edu. This work was supported in part by the NSF under Grant No. 1117325. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We also gratefully acknowledge support from ONR MURI grant N00014-09-1-1051, from AFOSR grant FA2386-10-1-4135 and from the Singapore Ministry of Education under a grant to the Singapore-MIT International Design Center.

be possible to use real continuous representations of object and robot poses instead [1].

A. Domain representation

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 $In(package1, truck3)$.
- **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. $PickUp(package2)$). Each operator, *op*, is characterized by: *preconditions*, $pre(op)$, a conjunction of fluents that describes when this operator is applicable; result, $res(op, s)$, a conjunction of fluents whose value changes as a result of applying *op* in state s; and *cost*, a real valued cost of applying *op*.

A planning problem, Π, is a tuple; $\Pi = \langle F, O, I, G \rangle$ 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 Π is a sequence of operators $p = \{op_1, op_2, \ldots, op_n\}$ such that $I \in pre(op_1)$, $res(op_i) \in pre(op_{i+1})$, and $res(op_n) \in G$. Each result function has an implicit argument that is the state that resulted from the application of the previous operator. To discuss the quality of a solution we say that $cost(p) = \sum_{i=1}^{n} cost(op_i)$. In realistic domains, fully specifying a truth value for each possible fluent is usually difficult and, as in the case of 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.

B. Abstraction

An *abstraction method* is a function, $f : \langle F, O, I, G \rangle \rightarrow \langle F', O', I', G' \rangle$ 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 [3].

There are many strategies for constructing abstractions. We 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 $abs(op, i), 0 \leq i < n$. The preconditions of $abs(op, i)$ 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. *Place(package1, truck1)* could map to a different abstraction level than *Place(package2, truck2)*.

C. 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.

Algorithm 1 The HPN planning and execution algorithm
```
1: procedure HPN($s, \gamma, \alpha, world$)
2:     $p$ =Plan($s, \gamma, \alpha$)
3:     for ($op_i, g_i$) in $p$ do
4:         if IsConcrete($op_i$) then
5:             $world$.execute($op_i, s$)
6:         else
7:             HPN($s, g_i$, NextLevel($\alpha, op_i$), $world$)
8:         end if
9:     end for
10: end procedure
```

The HPN algorithm is shown in Algorithm 1. It takes as inputs the current state of the environment, s; the goal to be achieved, γ; the current abstraction level, α; and the *world*, which is generally an interface to a real or simulated robotic actuation and perception system. Initially α 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.

Then, 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, then HPN is called recursively, with a more concrete abstraction level for that step. The procedure NextLevel takes a level of abstraction α and an operator *op*, and returns a new level of abstraction β that is more concrete than α.

The strategy of committing to the plan at the abstract level and beginning to execute it 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 general hierarchical planning algorithm must be capable of backtracking across abstraction levels if planning fails on a subgoal. An alternative 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 [4] describe several conditions under which this assumption holds.

III. 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. Consider a robot that must cause several objects to be placed in a room and cause the door to that room to be closed. It is entirely possible to make a plan at the abstract level that closes the door first and then places the objects; or one that places several small objects first, in the only space that a particularly large object can fit. For hierarchies that possess the DRP, HPN will ultimately achieve the goal, but at the cost of achieving, violating, and re-achieving subgoals unnecessarily.

If a context-based cost of each abstract action were available at planning time, these problems could be avoided. Unfortunately, that is hard to achieve without, essentially, solving all possible subproblems. Problems of this kind could possibly be addressed on a case-by-case basis by rewriting or reordering the preconditions in the operator descriptions. However, there is a fundamental contradiction: abstract planning is efficient *because* it ignores details; adding those details to make the resulting plans shorter will increase planning time and negate the advantages of the hierarchy.

A. Optimization in the now

We propose a strategy that retains the efficiency of aggressive hierarchical planning and execution while offering the opportunity to arrange subgoals in an order that will lead to shorter plans without significantly increasing planning time. In many cases, abstract operators are independent of one another: many orderings of them are equally valid with respect to the abstract preconditions and there is no clear reason to prefer one over another. To take advantage of this, we propose to use information in the current state of the world at plan execution time to perform peephole optimization. We heuristically select the next subgoal to achieve when there are several possible next abstract plan steps that are components of valid plans. This process can take advantage of more complex properties of the domain and do more expensive computation because it is not a bottleneck in the search for an abstract solution.

In the process of refining and executing an abstract plan, each subgoal is achieved sequentially, which means that the system is unable to take advantage of parallelism that might exist in subtasks. For example, consider the previous situation but with a robot that is able to carry multiple objects simultaneously. Serializing the subtasks of putting away each object might result in the robot making two long trips when it could have made just one. 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. Deciding this in a post-processing step, however, fits naturally into our framework. Thus, in addition to considering a reordering of an abstract plan, our refinement process considers achieving some of the subgoals jointly. 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 dynamically considers re-ordering and re-grouping plan steps. Before describing the crucial SelectGap and SelectParallelOps procedures, we describe the ordering preference information they will rely on.

Algorithm 2 Reordering and combining hierarchical planning and execution algorithm

```
    procedure RCHPN($s, \gamma, \alpha, world$)
2:      $p$ =Plan($s, \gamma, \alpha$)
        while $p \neq \emptyset$ do
4:          $(op, g)$ = SelectGap($s, p, \alpha$)
            if IsConcrete($op$) then
6:              $world$.execute($op, s$)
            else
8:              $sg$ = SelectParallelOps($s, op, g, p, \alpha$)
                $cur\_index = p$.index($g$)
10:             $sg\_index = p$.index($sg$)
                for $(op', g')$ in $p[cur\_index : sg\_index]$ do
12:                 $\alpha$ = NextLevel($\alpha, op'$)
                end for
14:             RCHPN($s, sg$, NextLevel($\alpha, op$), $world$)
            end if
16:     end while
    end procedure
```

B. Context-sensitive ordering

RCHPN depends on the specification of a context-sensitive comparison function, *arrange*(g_1, g_2, s), which takes as arguments two subgoals and an initial state and returns 0 if g_1 should 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. 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?

- Evaluation takes place "in the now": the algorithm knows the current state and does not need to consider the large number of possible ways in which the preconditions for an operator could have been realized.
- The task is simply to determine an ordering, not to estimate the actual costs, which would generally be much more difficult to do accurately.
- 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.

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 solve large instances of such problems effectively.

Assuming the existence of the *arrange* function, we now describe the peephole optimizations in RCHPN.

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 [2]. 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, we repeat this process with the g_j as the new "first" subgoal. 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.

SelectParallelOps, shown in Algorithm 4, proceeds in a similar fashion. It keeps track of a subgoal, *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 g_i 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.

Algorithm 3 Reordering an Abstract Plan

```
    procedure SELECTGAP($s, p, \alpha$)
2:      $next\_subgoal$ =HighestApplicableSubgoal($p, s$)
        $next\_index = p$.index($next\_subgoal$)
4:      $highest\_checked = next\_index$
        while $highest\_checked <$ len($p$) do
6:          for ($op, sg$) in $p[next\_index :]$ do
                if $arrange(next\_subgoal, sg, s) = 1$ then
8:                  $new\_p = p$.move(($op, sg$), $next\_index$)
                    if IsValid($new\_p$) then
10:                     $p = new\_p$
                        $next\_subgoal = sg$
12:                     $highest\_checked = next\_index$
                        break
14:                 end if
                end if
16:             $highest\_checked = p$.index($sg$)
            end for
18:     end while
    end procedure return $sg$
```

Algorithm 4 Combining Subgoals of an Abstract Plan

```
    procedure SELECTPARALLELOPS($s, op, sg, p, \alpha$)
2:      $next\_sg = sg$
        $next\_index = p$.index($next\_sg$) $+ 1$
4:      for ($op, sg$) in $p[next\_index :]$ do
            if $arrange(next\_sg, sg, s) = 2$ then
6:              $new\_p = p$.move(($op, sg$), $next\_index$)
                if IsValid($new\_p$) then
8:                  $p = new\_p$
                    $next\_sg = CombineGoals(next\_sg, sg)$
10:                 $next\_index = next\_index + 1$
                    if MaxComplexity($next\_sg$) then
12:                     return $next\_sg$
                    end if
14:             end if
            end if
16:     end for
    end procedure return $next\_sg$
```

C. Ordering-preference heuristics

Now we consider some principles that can guide the specification of the *arrange* function for particular domains, based on a concept of resource use. We can frame the problem in terms of shared resource consumption. Recall the robot that must put several objects in a room and close the door. In this case, the resource being used is free space. Both the act of placing an object inside the room and the act of closing the door consume the space in the doorway. Our difficulty arises because closing the door does not free up the resource after it is accomplished, but rather consumes the resource in perpetuity. The combination of tasks can be viewed in a similar light: moving each object needs to use the robot. 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 is *shareable* with respect to a goal if, while it is being used to accomplish that goal, it does not become unavailable. A

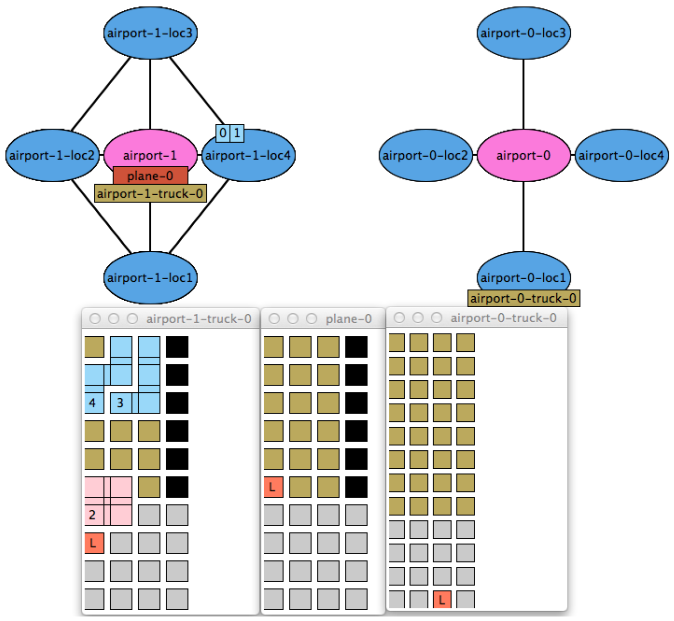


Fig. 1. Visualization of the Marsupial Logistics Domain. Circles are locations and pink circles are airports. The additional windows represent the loading and storage areas of the vehicles. The red squares represent a marsupial robot which takes care of storing packages for transit. In order for vehicles 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.

resource 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 is *continual* with respect to a goal if, so long as that goal is true, that resource will be unavailable. This now 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. 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 of plans and the opportunity for cost savings. If the common resource 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 violates 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. However, 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 include 0^{th} order estimates of resources consumed by operators used for that computation. This can be thought of as 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.

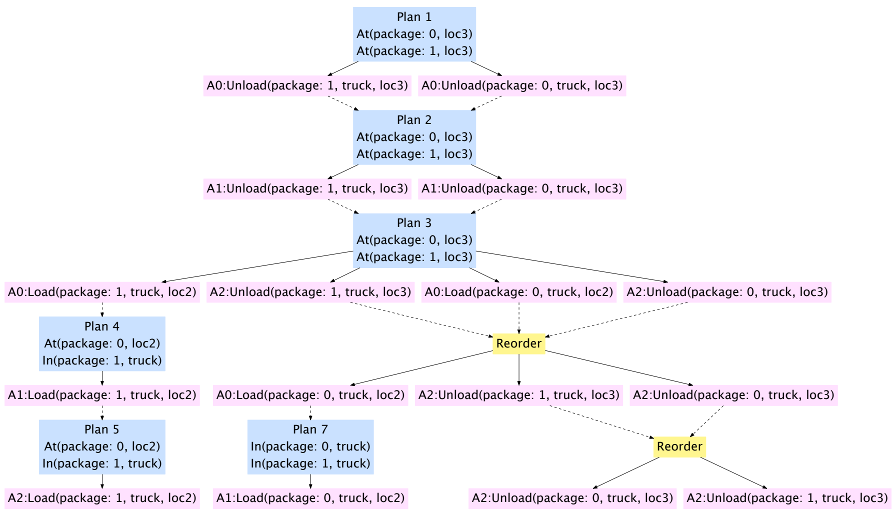
IV. Related Work

The notion of dropping preconditions to build hierarchies was first introduced by Sacerdoti [3]. He proposed that certain preconditions of operators could be considered details and deferred to a later point in planning, defining a hierarchy of abstraction spaces. Knoblock [5] provides a more formal definition of a refinement and a system for automatically generating hierarchies. His definition of refinement explicitly states that ordering relations between operators must be preserved in the refinement of a plan, but he does not consider the problem of reordering an abstract plan. Bacchus and Yang [4] analyzed the effectiveness of hierarchical planning with respect to the probability that abstract plans can be refined. They find that the *downward refinement property* (DRP) is a key factor in speedups from hierarchy.

Nau et al. [6] use a hierarchical task network (HTN) to hierarchically solve planning problems. 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. [7] suggest a view of abstract actions centered around upper and lower bounds on reachable sets of states. 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



Plan 1
At(package: 0, loc3)
At(package: 1, loc3)
A0:Unload(package: 1, truck, loc3)
A0:Unload(package: 0, truck, loc3)
Plan 2
At(package: 0, loc3)
At(package: 1, loc3)
A1:Unload(package: 1, truck, loc3)
A1:Unload(package: 0, truck, loc3)
Plan 3
At(package: 0, loc3)
At(package: 1, loc3)
A0:Load(package: 1, truck, loc2)
A2:Unload(package: 1, truck, loc3)
A0:Load(package: 0, truck, loc2)
A2:Unload(package: 0, truck, loc3)
Plan 4
At(package: 0, loc2)
In(package: 1, truck)
Reorder
A1:Load(package: 1, truck, loc2)
A0:Load(package: 0, truck, loc2)
A2:Unload(package: 1, truck, loc3)
A2:Unload(package: 0, truck, loc3)
Plan 5
At(package: 0, loc2)
In(package: 1, truck)
Plan 7
In(package: 0, truck)
In(package: 1, truck)
Reorder
A2:Load(package: 1, truck, loc2)
A1:Load(package: 0, truck, loc2)
A2:Unload(package: 0, truck, loc3)
A2:Unload(package: 1, truck, loc3)

Fig. 2. The root of a planning tree for a simple problem 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 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.

are solved on their own, treating the problems solvable by the 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 [8]. 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 [9] 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. Minh and Kambhampati [10] use post-processing and reordering to do makespan optimization for temporal planning. Their system uses a standard planner to find an initial, totally ordered, plan and post-processes that into a partially ordered plan, which is optimally scheduled. These approaches are similar to ours in that our heuristics take advantage of 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.

V. EXPERIMENTS

A. *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 [11]. 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 the 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 the package is loaded onto or removed from a truck, there will be a detailed motion plan for a forklift, which might include moving other packages within the truck to accommodate the target package. Fig. 1 depicts a graphical representation of this domain.

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 [12]). We can also see it as an instance of the *navigation among movable obstacles* (NAMO) problem in a discrete space [13]. 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.

The domain is formalized with operators shown in the appendix. In addition to results and preconditions, with their associated criticalities, each operator declares its resource dependencies and cost. Vehicle operators (Load, Unload, Travel) have cost 10. Robot operators (LoaderMove, LoaderPick, LoaderPlace) have cost 1. Inference operators (SameCity, Pack) have cost 0.

B. *Experiments and results*

We designed experiments to compare a classical non-hierarchical planner called FF [14], HPN, and RCHPN. FF is a fast, easy-to-use classical planning algorithm. 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 had no shape (i.e. 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. There have been improvements in this class of planners [15], but they cannot ultimately address the fundamental problem that very long plans are required 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. 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; in this work, it is an implementation of A^* in the discretized geometry of a truck.

In designing the *arrange* function, we are tasked with determining the resources used by abstract versions of operators and categorizing 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 in the vehicle, including packages mentioned in the goal, such that the vehicle could be packed. 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. 2.

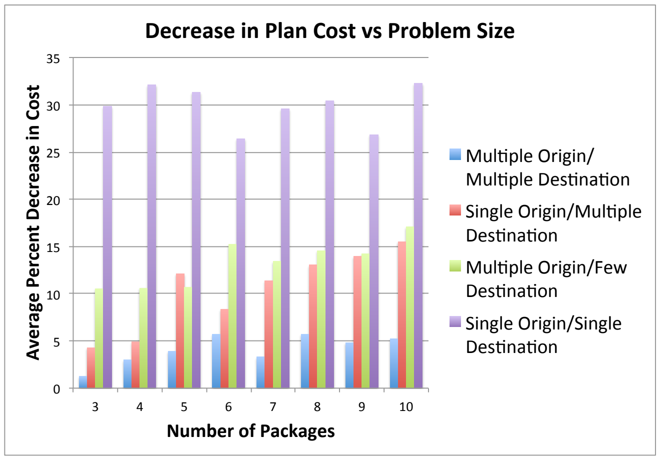


Fig. 3. Average percent decrease in plan cost vs. problem size for RCHPN vs. HPN

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 truck to do the routing within each city and a single plane to route between the airports. For the vehicles, the cargo area for packages was randomly selected to be either small (3x6) or large (4x8). Packages were randomly selected from a set of 6 shapes.

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. 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

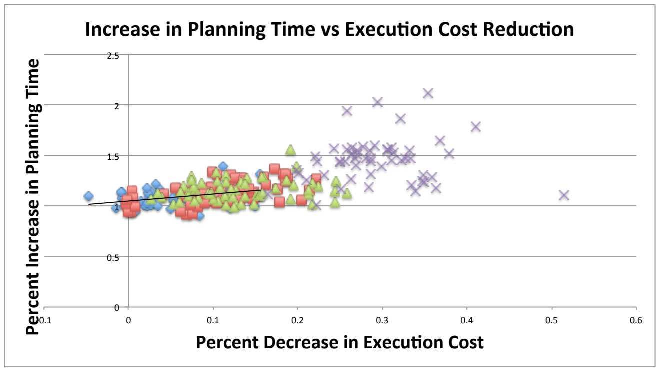


Fig. 4. Plot of percent decrease in execution cost vs. percent increase in planning time. The positive correlation between the two highlights a useful property of optimizing as a post-processing step: on problems where there is little parallel structure, our modifications have little to no effect on 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.

the planner will be able to take advantage of. Fig. 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. 4 depicts this relationship in detail.

APPENDIX

Operator Descriptions for the Marsupial Logistics Domain

- *Load(package, vehicle, location)*:
  result: *In(package, vehicle*,
  PkgLoc(package, vehicle, vehicle.loadLoc)
  preconditions: *Reachable(location, vehicle)*, *At(package, location*, *At(vehicle, location)*,
  Clear(vehicle, loadRegion)
  consumes: *vehicle, vehicle.loadRegion, vehicle.location*
- *Unload(package, vehicle, location)*:
  result:*At(package, location)*
  preconditions: *Reachable(location, vehicle),*
  At(vehicle, location), In(package, vehicle),
  In(package, vehicle), PkgLoc(package, vehicle, loadLoc)
  consumes: *vehicle, vehicle.loadRegion, vehicle.location*
- *Travel(vehicle, startLoc, resultLoc)*:
  result:*At(vehicle, resultLoc)*
  preconditions: *At(vehicle, startLoc),*
  Connected(startLoc, resultLoc, vehicle), Packed(vehicle)
  consumes: *vehicle, vehicle.location*
- *LoaderGrasp(vehicle, package, grasp, gridLoc)*:
  result: *LoaderHolding(vehicle, package, grasp)*
  choose: *loaderLoc* $\in$ *GraspLocations(gridLoc, grasp)*
  preconditions:*LegalGrasp(package, grasp, gridLoc), LoaderLoc(vehicle, loaderLoc), LoaderHolding(vehicle, None, None), PkgLoc(vehicle, package, gridLoc)*
  consumes: *vehicle.loader, loaderLoc*
- *LoaderPlace(vehicle, package, gridLoc, grasp)*:
  result: *PkgLoc(package, gridLoc)*
  choose: *loaderLoc* $\in$ *GraspLocations(gridLoc, grasp)*
  preconditions: *LegalGrasp(package, grasp, gridLoc), LoaderLoc(vehicle, loaderLoc), LoaderHolding(vehicle, package, grasp), LoaderHolding(vehicle, package, grasp)*
  consumes: *vehicle.loader, gridLoc, loaderLoc*
- *LoaderMove(vehicle, targetLoc)*:
  result: *LoaderLoc(vehicle, targetLoc)*
  choose: *p* $\in$ *Paths(vehicle.loaderLoc, targetLoc)*
  preconditions: Clear(p)
  consumes: *vehicle.loader, p*
- *SameCity(package, vehicle)*:
  result: *SameCity(package, vehicle)*
  choose: *loc* $\in$ *ReachableLocs(vehicle)*
  preconditions: *At(packge, loc)*
- *Pack(vehicle)*:
  result: *Packed(vehicle)*
  choose: *loc[pkg]* $\in$ *vehicle.storageRegion* $\forall$ *pkg s.t. In(pkg, vehicle)*
  preconditions: *PkgLoc(pkg, vehicle, loc[pkg])* $\forall$ *pkg s.t. In(pkg, vehicle)*
  consumes: *vehicle, vehicle.storageRegion*

REFERENCES

[1] L. Kaelbling and T. Lozano-Perez, "Hierarchical planning in the now," *IEEE Conference on Robotics and Automation*, 2011.
[2] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, pp. 189–208, 1971.
[3] E. D. Sacerdoti, "Planning in a hierarchy of abstraction spaces," *Artificial Intelligence*, vol. 5, no. 2, pp. 115 – 135, 1974.
[4] F. Bacchus and Q. Yang, "Downward refinement and the efficiency of hierarchical problem solving," *Artificial Intelligence*, vol. 71, no. 1, pp. 43 – 100, 1994.
[5] C. A. Knoblock, "Automatically generating abstractions for planning," *Artificial Intelligence*, vol. 68, no. 2, pp. 243 – 302, 1994.
[6] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *JAIR*, vol. 20, pp. 379–404, 2003.
[7] B. Marthi, S. Russell, and J. Wolfe, "Angelic hierarchical planning: Optimal and online algorithms," in *Proceedings of ICAPS 2008*, 2008.
[8] E. Amir and B. Engelhardt, "Factored planning," in *Proceeding of IJCAI 2003*, 2003, pp. 929–935.
[9] B. Srivastava and S. Kambhampati, *Scaling up Planning by Teasing Out Resource Scheduling*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2000, vol. 1809, pp. 172–186.
[10] M. B. Do and S. Kambhampati, "Improving the temporal flexibility of position constrained metric temporal plans," in *Proceedings of ICAPS 2003*, 2003.
[11] M. M. Veloso, *Planning and Learning by Analogical Reasoning*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1994.
[12] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell, "Tetris is hard, even to approximate," *CoRR*, vol. cs.CC/0210020, 2002.
[13] M. Stilman and J. J. Kuffner, "Planning among movable obstacles with artificial constraints," in *Proceedings of WAFR 2006*, 2006.
[14] Jörg Hoffman, "FF: The fast-forward planning system," *AI Magazine*, vol. 22, pp. 57–62, 2001.
[15] S. Richter and M. Westphal, "The LAMA planner: Guiding cost-based anytime planning with landmarks," *Journal of Artificial Intelligence Research*, vol. 39, pp. 127–177, 2010.