Discovering and Exploiting Synergy Between Hierarchical Planning Agents

Jeffrey S. Cox and Edmund H. Durfee
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109
{jeffcox, durfee} @umich.edu

Abstract

Agents interacting in a multiagent environment not only have to be wary of interfering with each other when carrying out their tasks, but also should capitalize on opportunities for synergy. Finding overlapping effects between agents' plans can allow some agents to drop tasks made unnecessary by others' actions, to reduce the cost of execution and improve the overall efficiency of the multiagent system, thus creating synergy. In this paper we define criteria for finding synergy and develop algorithms for discovering synergy between planning agents that exploit hierarchical plan representations. Our results show that our approach not only can dramatically reduce the costs of finding synergies compared to non-hierarchical strategies, but can also find synergies that might otherwise be missed.

Introduction

Agents interacting in a multiagent environment have the ability to act in ways that are both felicitous and deleterious to the goals of other agents. For example, an agent responsible for cleaning a kitchen in a house and an agent responsible for cleaning the floors of the whole house can help or hinder each other's efforts. The floor-cleaning agent could sweep the floor of the kitchen, thus allowing the kitchen-cleaning agent to mop the floor without having to sweep it first itself. On the other hand, the floor-cleaning agent could take the shared broom off to other rooms first, leaving the kitchen-cleaning agent waiting for the broom.

Various researchers have studied strategies for identifying and resolving unintended conflicts between agents' plans (see Related Work section). Recently, Clement has shown that utilizing summary information in agents' plan hierarchies is an efficient and reliable means for conflict detection and resolution between multiagent plans (Clement, 1999a). In this paper, we build on Clement's approach to also efficiently detect and exploit unexpected synergies (overlapping objectives) to the benefit of the agents.

A synergy arises when a plan or subplan of an agent happens to achieve effects that another agent also planned to achieve. For example, in the case of the cleaning agents, both agents want to make the kitchen floor is clean, and left to themselves would redundantly clean it. Rather than both achieving the same results, the agent whose plan's effects (also called postconditions) are subsumed by those of the other agent can drop that plan, and instead depend on the other agent to establish those conditions. We say in this case that the subsuming and subsumed plans are "merged" to reduce the overall cost of execution.

Agents that are inherently benevolent might choose to merge plans to minimize the aggregate effort among them, even if one agent ends up doing most of the work. When agents' plans are mutually subsuming (have the same effects), however, the criteria for deciding which will drop its plan could be based on some form of compensation for agents who are self-interested. Alternatively, agents might assign responsibilities to balance the costs accrued by the agents, or to support parallelism among activities to minimize the time-to-completion for the slowest of the agents. Plan merging is not without its disadvantages, however. It introduces dependencies among agents that decrease their individual autonomy. It also can introduce delays as agents wait for others to achieve results that might have been done faster locally.

Finding and exploiting serendipitous synergies can also be computationally and communicatively intensive, to the point where the effort spent synergizing exceeds the savings accrued by the synergies! Agents operating in complex environments might consider numerous alternative plans, and the number of ways each of these plans could interleave execution with others' plans leads to exponential computation. Our contributions in this paper are that we use insights from hierarchical planning (Korf, 1987; Knoblock, 1991) and hierarchical coordination (Clement, 1999b) to develop a top-down approach to multiagent plan merging at abstract levels using agents' hierarchical plan representations. Our techniques help reduce the problem complexity and permit the discovery of additional merging opportunities to improve agents' joint plan execution.

In the remainder of this paper, we begin by briefly summarizing related prior work. We then present the specifics of our hierarchical plan representation, and review Clement's hierarchical plan summarization process, which we rely on to discover and perform abstract merges. We

then formalize the concept of plan merging, or synergy, between plans in hierarchical plans, and describe a top-down algorithm that we have implemented that is capable of solving a restricted case of the synergy problem presented here. This is followed by an analysis of the benefits of our techniques along with experimental evaluation. We conclude the paper by outlining our future work.

Related Work

The idea of merging plans to increase efficiency is not a new one. Yang has done work on plan step merging for a single agent with conjunctive goals with the idea of removing redundancy between solutions to individual conjuncts of the larger goal (Yang, 1997). The rationale, based on work first done by Korf (Korf, 1987), was that dividing a larger, more complex problem into several separate (and simpler) subproblems and then merging the resultant plans into a single plan for a single agent to execute could be more efficient than simply solving the complex problem.

Yang gives a formal definition for what it means for a set of plan steps to merge with another set: a set of grouped plan steps Σ can be merged with a plan step μ (meaning μ replaces Σ in the plan) if the preconditions of Σ subsume those of μ, the postconditions of μ subsume those of Σ, and the cost of μ is less than the cost of Σ. In other words, μ achieves more effects and needs fewer preconditions when executed, and is cheaper to carry out as well. More recent work by Horty and Pollack (Horty et. al., 2000) on evaluating new goal options given an existing plan context relies on Yang's definition of plan step merging, as does our own work (though currently we ignore the precondition requirement). When we refer in this paper to two plan steps merging, what we mean is that one plan step will replace the other in the resultant multiagent plan.

Wilkins' SIPE system (Wilkins, 1988) is capable of goal phantomization, where existing operators are grounded or constrained so as to achieve new goals (preventing redundant operator instantiation). This process is analogous to our work, though we operate in a multiagent context and remove redundant steps from plans rather than not adding them in the first place. We also (as yet) do not handle operators with variables. Ephrati has extended Yang's work on plan merging to the multiagent context by farming out subgoals of a single goal to different agents to solve and then integrate (Ephrati et. al., 1994). Ephrati's system was able to handle both positive and negative interactions between plans, but did not deal with agents with their own, independent goals. Georgeff did early work on resolving conflicts and potential clobbering actions between non-hierarchical plans of different agents with different goals by grouping plan steps and inserting synchronization actions, but did not handle positive interactions (Georgeff, 1983).

Most recently, Clement has examined ways of coordinating hierarchical plans between multiple agents and using summary information to speed his process, but his work also avoided handling positive interactions (Clement, 1999a). Goldman and Rosenschein have researched the benefit of implementing general rules of cooperative behavior in multiagent systems that, because they are cooperative, has more in common with Yang's plan merging efforts than multiagent coordination (Goldman et. al, 1994). Unlike Goldman and Rosenschein's work on cooperative behavior, we discover explicit opportunities for synergy that are specific to the plans of the executing agents rather than simply general environmental rules. Von Martial (Von Martial, 1990) presents a relationship taxonomy of both positive and negative interactions between autonomous planning agents that is relevant to our own plan step merging criteria. The system we present here handles both Von Martial's equality relationships and consequence relationships, but not favor relationships.

Hierarchical Planning and Computing Plan Summary Information

For individual agents, hierarchical plans offer several advantages over traditional, STRIPS style "flat" plans. Hierarchical planning has been recognized as an efficient form of planning that focuses the search through plan space for an agent carrying out the planning problem (Knoblock, 1991). Instead of having to try out a large number of possible plan orderings, plan hierarchies limit the ways in which an agent can select and order its primitive operators, and thus reduce the size of the search space. In multiagent systems, agents need to worry not only about the internal flexibility of their own plan representations, but also need flexible ways in which to interact with other agents as well. We have built a system capable of taking advantage of positive interactions between abstract plans within different agents' plan hierarchies that require less specific commitments between agents to carry out, and thus still take advantage of the innate flexibility offered by the hierarchical plan representation.

Before exploring our multiagent plan synergy system, we should outline our definition and representation of hierarchical plans that our constructed system is capable of merging. A plan hierarchy is a hierarchy comprised of plans; at the higher (more abstract) levels of the hierarchy, each plan achieves more effects than a plan at a lower level, and the bottom of the hierarchy is composed of primitive plans that are the operators that an agent can directly execute. In essence, a plan hierarchy represents a set of possible plans that all achieve the same overall goal. Each possible complete refinement of a hierarchical plan represents one possible non-hierarchical set of primitive plans capable of achieving the goal of the hierarchical plan. A partial refinement is still an abstract plan, just with some expansion decisions already made (and thus representing a smaller set of possible complete refinements).

Formally, we define a plan *p* (an element of a plan hierarchy) as a tuple {*pre*, *in*, *post*, *type*, *subplans*, *order*, *cost*}. *Pre*, *in*, and *post* are sets of conditions corresponding to the preconditions, inconditions, and postconditions of the plan. Preconditions are conditions that must hold

prior to the plan's execution, inconditions are conditions that must or may hold during execution, and postconditions are conditions that will hold after execution (also known as the effects of the plan). The type of plan *p*, *type*(*p*), has one of three values: *primitive, and,* or *or*. An *and* plan is a non-primitive plan that is accomplished by carrying out all of its *subplans*, and an *or* plan is a non-primitive plan that is accomplished by carrying out any one of its *subplans*. An *or* plan is one in which the agent executing the plan has a choice as to how it will accomplish the goal of the plan. A *primitive* plan has no *subplans.* *Subplans* is a set of pointers to the subplans of the plan. *Order* is a set of temporal ordering constraints represented using precedence

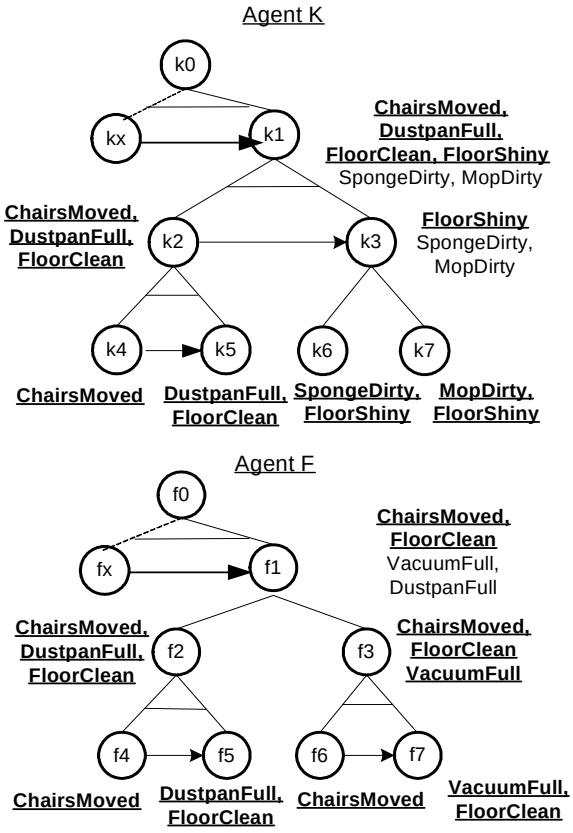

Figure 1

relations between time-points, where each plan is an interval represented by both a start and an end point (Vilain, et. al., 1986). All ordering constraints in *Order* are between time points of plan *p*'s *Subplans*. Constraints can be of forms **before(*a, x, b, y*)**, **after(*a, x, b, y*)**, and **same(*a, x, b, y*)** where *a* and *b* are plan steps in *Subplans*, and *x* and *y* are either *start* or *end* (indicating whether the constraint is between the start point or the end point of the plan step). Finally, *cost* is a real number representing the cost of executing the plan. For a non-primitive plan *p*, *pre, in, post,* and *cost* are implicitly encoded in the conditions and costs of *p*'s primitive subplans. The summarization process we describe next makes this implicit information explicit.

Figure 1 shows a partial plan hierarchy for the Kitchen-cleaning agent (K) and the Floor-cleaning agent (F) described earlier. For the sake of simplicity, only the post-conditions are shown. A horizontal line connecting the edges going to a plan's children indicates the plan is of type *and*, else it is of type *or*. The directional arrows between plans indicate temporal ordering constraints. ***k3*** in the kitchen agent's hierarchy is an example of an *or* plan, where the agent can either clean the kitchen with a mop or a sponge. ***k2,*** on the other hand, is an *and* plan because both of its subplans (***k4*** and ***k5)*** need to be taken. The arrow between ***k2*** and ***k3*** shows an ordering constraint, **before(*k2, end, k3, start*)**, where the kitchen agent must clean the floor before using a sponge or a mop to make the floor shiny.

Reasoning about potential plan merges between abstract plan steps across different agents' hierarchies can be difficult because often information about the preconditions and effects of an abstract plan are only represented in the primitives of the plan, and not in the abstract plans themselves. Fortunately, Clement has designed a mechanism for propagating such information up the plan hierarchy. This enables reasoning about interactions between abstract plan steps of different agents (Clement, 1999b). Clement's Multilevel Coordination Agent (MCA) is capable of resolving negative interactions between the hierarchical plans of multiple agents by building up this plan *summary information* to allow coordination between hierarchical plans at any of the abstract or primitive levels of expansion.

To represent this summary information in the plan hierarchy once it is calculated, Clement relies on the existing plan tuple we have presented, with two modifications. First, a plan condition *c* (*c* ∈ *pre*, *in* or *post*) is now represented as a tuple, {*l, existence*}. *l* is the condition literal, the actual condition that will hold, and *l*(*c*) refers to the literal of condition *c*. The *existence* of *c* can be *must* or *may*. If it is *must*, then *c* is called a *must* condition because *l* holds for every successful plan execution. If *c* is *may*, it is called a *may* condition because it may hold, but is not guaranteed to hold for every successful plan execution. This introduction of *must* and *may* conditions is necessary as, unlike a primitive plan step whose conditions are always necessary (i.e., always *musts*) an abstract plan step with *or* plan step descendents will have different conditions depending on how it is decomposed. In Figure 1, *must* conditions are underlined, such as ChairsMoved for ***k1,*** as it will be accomplished no matter how the plan is decomposed***.*** *May* conditions are not underlined, such as MopDirty for **k1,** as the mop will only be dirty if the agent uses the mop (and not the sponge).

Clement's second change is to replace *cost* with *mincost* and *maxcost*, thus specifying a range of possible costs of the plan step. This is done for reasons similar to the ones outlined above. *maxcost* of an *or* plan is the largest *maxcost* of its *subplans*, while *mincost* of an *or* plan is the smallest *mincost* of its *subplans*. *maxcost* and *mincost* of an *and* plan are the sum of the *maxcosts* or *mincosts* of the members of its *subplans*¸ respectively. Summary informa-

tion is calculated by recursively propagating information from the leaves of the hierarchy to the root plan.

Discovering Multiagent Plan Step Merges

To discover possible merges of plans between different agents, we have constructed a *Synergy Agent* (SA) capable of finding and implementing plan step merges between different agent plans. Agents wishing to have their plans synergized send their plan hierarchies to the SA (which itself could be one of the synergizing agents) via messages passed over the CoABS grid (CoABS, 2002). The SA identifies and implements sets of merges by removing redundant plan steps and sending to the other agents additional constraints on how or when they should execute the plan steps in their plan hierarchies to take advantage of the discovered synergy (the following section will explore this in more detail.)

The SA currently finds only pairwise merges; if three or more agents can merge plans such that only one of them needs to execute a plan step to satisfy them all, the SA can find such a merge through repeated discovery and adoption of pairwise merges. The SA can determine if any two plans in two different plan hierarchies can merge by first calculating the summary information for each of the two hierarchical plans using Clement's techniques. Then the SA examines the summarized postconditions of the two plans. If one of the two plans has a set of summarized *must* postconditions that subsumes the summarized *must* postconditions of the other plan, then they can be merged (this is half of Yang's original definition). More formally, we say that plan step p_i can merge with plan step p_j (meaning p_j replaces p_i in the new plan) if and only if

$$\forall c_i \left(\begin{array}{l} \left(c_i \in post(p_i) \,\&\, existence(c_i) = must\right) \rightarrow \\ \left(\begin{array}{l} \exists c_j, c_j \in post(p_j) \,\&\, existence(p_j) = must \\ \&\, l(c_i) = l(c_j) \end{array} \right) \end{array} \right)$$

Intuitively, the SA should be able to merge on *may* conditions as well. Since *may* conditions at a higher level of abstraction are *must* conditions at a lower level, to take advantage of possible merges between *may* conditions the SA must further decompose the hierarchy to the point that the *may* conditions become *must*. In Figure 1, one possible merge would be between ***k2*** and ***f2***, as they share the same *must* postconditions (ChairsMoved**,** DustpanFull**,** FloorClean**)**. Other merges include ***k1*** and ***f2***, and ***k1*** and ***f1***. The best merge(s) to perform will depend on the comparative cost reduction(s) achieved by the merge(s), and the time required to discover the merge(s) in the hierarchies.

Performing Plan Merging between Different Agent Hierarchies

Once the SA has identified a pair of plan steps within different agent hierarchies that can merge, it must perform a series of operations to implement the merge. First, the plan step whose postconditions are subsumed must be removed from the agent's hierarchy. If the subsuming plan step is a descendant of a plan of type *or*, then the SA must request that the executing agent constrain its plan hierarchy so that it is guaranteed to execute the subsuming plan. The same follows for the agent with the plan step being subsumed (though this agent doesn't actually execute the subsumed plan). The SA requests this by partially expanding the plan hierarchy to the point that this selection has been made (thus limiting the possible executions of the hierarchical plan) and returning this partial expansion to the waiting agents. In addition, if the plan step being removed is part of a larger partial order of plans in the plan hierarchy, the SA also requests that the requisite inter-agent temporal ordering constraints be implemented so that the subsuming plan step of the two being merged is substituted in the partial order for the one being removed and the agent having its plan removed is constrained to wait for the agent with the subsuming plan step to execute it. Previous work has shown that it is possible to map from this solution representation to an agent executable representation, such as one used by UM-PRS (Cox, 2001), where all inter-agent plan-ordering constraints are represented as wait conditions and synchronization messages in the agent's internal plan representation.

Synergy Algorithm

Given the formalization of merging plan steps from different agents' plan hierarchies (presented earlier) and the approach to implementing individual merges (outlined in the previous section), the challenge is to find effective *sets* of plan step merges that reduce the cost of their execution. To do this, the SA uses a top-down search algorithm. The search is called a top-down search because the first plan step merges that are discovered are ones between plan steps at an abstract level within the agents' hierarchies. Our current implementation assumes that there are no remaining conflicts between agents' plans (that is, any conflicts have been resolved prior to coordination with Clement's techniques). We assume that the goal of the SA is to find merges that help minimize the total cost expended by all agents, and thus the SA values solutions based on the total cost of the plan hierarchies.

Figure 2 shows a series of top-down expansions of the agents' plan hierarchies that gives a sense of how the SA uses this top-down search to discover plan step merges. The SA starts at the root of the hierarchies and expands downwards until it finds plan steps that can be merged. In the figure, the merge it has discovered is between **k2** and **f2,** where **f2** replaces **k2**, (though in the actual algorithm it would have found other merges along the way as well). The merge requires the floor-cleaning agent to commit to an execution of its plan hierarchy in which it guarantees to select **f2** (and thus sweep the floor) when expanding **f1**. In addition, to ensure temporal consistency, the SA has added an ordering constraint between the endpoint of ***f2*** and the

starting point of ***k3*** in place of the ordering constraint between the endpoint of ***k2*** and the starting point of ***k3*** (as the kitchen-cleaning agent still has to wait until the floor is swept to mop it). In our algorithm, we refer to the most abstract plan steps of a partially expanded plan hierarchy as a *frontier* (similar to the frontier of a search process through a search tree). A frontier is only valid if the total set of summary postconditions of all plan steps on the frontier is equivalent to the set of summary postconditions in the original root plan step of the hierarchy. In Figure 2, each set of plan steps divided by the solid lines depicts the two current plan frontiers of the two cleaning agents.

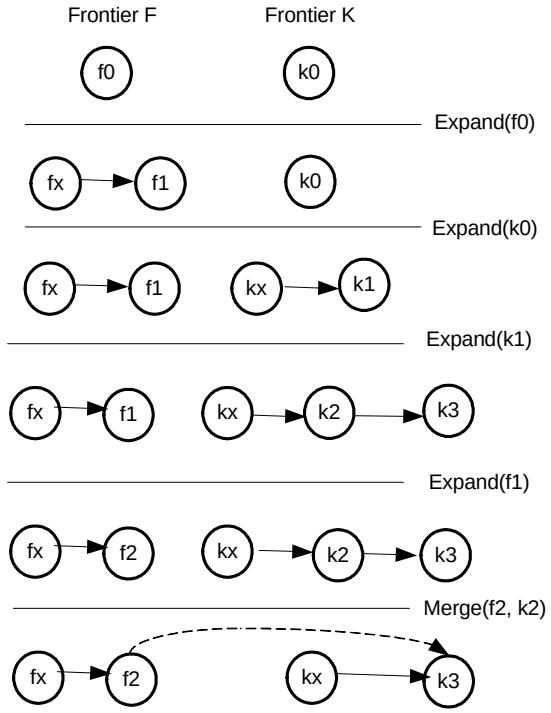

Figure 2

The inputs to the algorithm are the top-level plans of each of the two agents being merged. In the example above, these were **k0** and **f0**. (Note that our algorithm supports the merging of any number of agent plans, but without loss of generality, we present how it functions with only two agents being merged.) A *state* in our algorithm is a tuple {*frontA*, *frontB*, *orderings*}. *frontA* and *frontB* are the current plan frontiers of the coordinating agents. Finally, *Orderings* is the set of inter-agent and intra-agent causal relations added by merging and expanding plans. An example state from Figure 2 (and one that represents the bottom-most set of plan steps in the figure) has *frontA* = {***kx***, ***k3***}, *frontB* = {***fx***, ***f2***} and *orderings* = {**before**(***f2, end***, ***k3, start***)}.

Figure 3 shows our synergy search algorithm, the heart of the SA. The algorithm begins by de-queuing the first

```
Search Algorithm

s = newSearchState();
s.frontA = top-level plan of agent A
s.frontB = top-level plan of agent B
s.orderings = NULL;
Push(SearchQueue, s)

Begin Loop
If SearchQueue empty, exit

  s := Pop(SearchQueue)

  If s ∈ ClosedList, skip state and continue

  Insert(s, ClosedList)

  PruneFrontiers(s);

   ;;merge all possible plans
  ∀x, y (x ∈  s.frontA, y ∈  s.frontB){
   if postcondSubsume(x, y)
      s2 = MergePlans(x, y, s)

   else if poscondSubsume(y, x)
      s2 = MergePlans(x, y, s)

    if(!CycleDetect(n2)){
      Push(s2, SearchQueue)
      Insert(s2, SearchQueue)
      }
  };;end for all x, y

  ;;generate all possible frontier expansions
  ∀x (x ∈  s.frontA x OR  x∈  s.frontB,
  x.type ≠ primitive, x.pruned = false)
  {
      s2 = ExpandPlan(x,  s)
      Insert(s2, SearchQueue)
   } ;;end for all x
End Loop

MergePlans(a,b,n){
s2 = Copy(s)
Remove(b, s)
UpdatePartialOrderings(a, b, s)
Return s2
}

ExpandPlan(a, n){

s2 = Copy(s)
Remove(a, s)
Insert(a.subplans, s)
SubstPartialOrderings(a, a.subplans, s)
Return s2
}
End Algorithm
```

Figure 3

search state from the search queue. If the de-queued state is not already on the closed list of states, it is inserted onto the closed list and the loop proceeds. Otherwise, the state is discarded and a new state is de-queued. The algorithm then generates successor states created by merging plans. In the *MergePlans* function, states are tested to determine if plans on one agent's frontier (*frontA*) could merge with plans on the other agent's frontier (*frontB*). For each pair of plans that are found to be able to merge between the two frontiers (based on the criterion described earlier), the SA

generates a new search state in which the plan that was subsumed is removed from its corresponding frontier.

The algorithm also generates successor states by expanding existing plan steps. For each plan of type *and* on each frontier, it generates an additional search state in which the plan step has been replaced on the frontier by its subplans. For each plan of type *or* on either frontier, the algorithm enqueues a new state for each subplan of the plan step where the *or* plan step has been replaced by a subplan. This expansion of *or* plan steps allows for potential merging of children of an *or* plan by committing the executing agent to a particular subplan. Newly generated states are pushed on to the end of the queue, making the search breadth-first. After all successor states are enqueued on the queue, the loop repeats.

When the algorithm generates a successor state in which a merge has been performed, it marks it as a potential solution. Marked states are converted to solution candidates if the total cost of the state is less than all previously seen solution candidates. Though the SA could conceivably just ship back to the waiting agents the required modifications to be made to implement a set of discovered merges, allowing the waiting agents to make the changes themselves, it currently creates modified plan hierarchies for each solution candidate and returns them in whole to the waiting agents. To do this, it first implements the inter-agent ordering constraints in *Orderings* as wait conditions and signal messages to be passed between agents. To convert the frontiers and intra-agent ordering constraints in *Orderings* to plan hierarchies the SA creates a new root plan for each agent, assigns its frontier to the new root plan's *subplans* list, and adds all the agent's intra-agent ordering constraints stored in the search state's *orderings* to the root plan's *orderings*.

Plan Frontier Pruning

To avoid the unnecessary generation of search states, before a de-queued state generates its successor states based on the contents of its frontiers, the SA checks each plan on one frontier against the plans on the other agent's frontier to determine if the plan has postconditions that overlap with those of any plan on the other frontier. Plans that the SA identifies as having no overlapping postconditions are marked as "pruned," meaning that the plans are not expanded to generate new search states, since a plan with no overlapping postconditions will not have any children with postconditions that overlap either. More generally, different agents' hierarchies that predominantly affect different aspects of the world will have many pruned plan steps, allowing the SA to find the few actual merges much more quickly than it could otherwise. This feature can significantly reduce the search space in problems where there is little potential for synergy, as it eliminates a large number of plan comparisons and possible search states. It is implemented as the *PruneFrontiers* function in Figure 3.

Maintaining Partial Orders and Detecting Cycles

Each time the SA generates a new search state from a merge, it must modify the partial orderings over agents' plans in the search state to implement this merge and ensure that dependencies on the plan being removed are replaced with dependencies on the plan replacing it. To modify the ordering constraints the SA must carry out carry out three steps: First, the SA removes all ordering constraints between either the start point or end point of the plan step ***a*** being removed and the start or end point of some other plan ***b*** of form **before(*a, x, b, y*)** (or **after(*b, y, a, x*)**) from the constraints list. Second, for each of the previously removed constraints, a new constraint is added between ***b*** and the equivalent point of the plan step ***c*** replacing the removed plan of form **before(*c, x, b, y*)**. Finally, for all points ***d*** of plans in constraints of form **before(*d, z, a, x*)** (or **after(*a, x, d, z*)**) and points ***b*** (defined above), we add a constraint of form **before(*d, z, b, y*)**. This ensures that all existing orderings are preserved. This merging is accomplished by the *UpdatePartialOrderings* function in the *MergePlans* function in Figure 3. Once the SA has implemented these ordering constraint changes, the *CycleDetect* function in Figure 3 determines if the transitive closure of the partial orderings between the plans in both frontiers contains any cycles and thus must not be added to the solutions list or the search queue. The final step of Figure 2 shows an example of how the ordering constraints would be modified if **f2** were merged with **k2**.

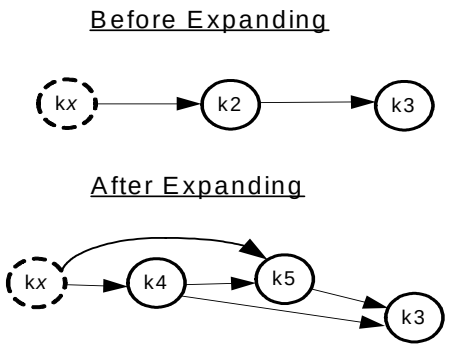

Figure 4

To maintain temporal consistency, the SA also modifies partial orderings when it expands plan steps in search states as well. When a child or the children of a plan replace it on a frontier, all constraints between it and other plans on either frontier are removed by the SA, and it creates new constraints between these other plans and its children, such that the existing temporal orderings are ensured. To do this, for each constraint between the expanded plan step ***p*** and another plan step ***q***, the same constraint is added between ***q*** and all of ***p***'s children (if ***p*** is an *and* plan step) or the selected child (if the *p* is an or step). This functional-

ity is implemented in the *SubstPartialOrderings* function in the *ExpandPlan* function in Figure 3. Figure 4 shows how the ordering constraints would be modified if the algorithm expanded **k2**. For example, before being expanded, **kx** is ordered before **k3**, so after the expansion, **kx** must also be ordered before **k3**.

Evaluation

Our top-down approach to merging between hierarchical plans of multiple agents has several advantages over more traditional, primitive-level approaches. One of the key advantages of a hierarchical approach arises when agents have different primitives for accomplishing actions, but their abstract plans accomplish the same (or subsumed) effects. For example, as a modification to Figure 1, let us say that agent K cleans floors using a two-step process where the first step accomplishes (ChairsMoved and DogMoved) and the second step achieves (FloorClean and CupboardsShut). That is, K clears out the room and then shuts cupboards while moving around sweeping. On the other hand, agent F gets the room set first by achieving (ChairsMoved and CupboardsShut) and then chases the dog out while cleaning for the effects (FloorCleaned and DogMoved). In this case, none of the primitive steps can be merged, but the abstract steps that combine each of the pairs of primitives can.

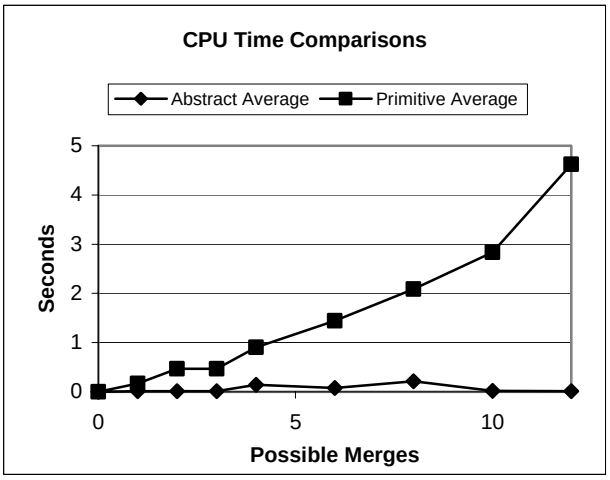

Figure 5

A bottom-up approach where primitives were collected into groups and then groups were in turn merged could identify larger merges. This approach, while able to identify abstract merges akin to the ones the top-down approach does (and potentially others that the top-down approach would not), suffers from the problem of not knowing what sets of primitives to group together. Given n primitives at the bottom of a tree, the number of possible subsets of primitive plans is $\sum_{k=1}^{n} n!/(n-k)!k! = O(2^n)$. Yang's complexity analysis of this method reveals the same exponential result (Yang, 1997). Yang presents an O($n$log(n)) approximation algorithm to merging, although this algorithm relies on a total order over the individual plans, which is an assumption not made of the plans handled by our algorithm. So, just computing all possible subsets of primitives leads to an exponential time complexity, let alone having to compare one exponential set of primitive subsets to the set of subsets of another agent's plan. The top-down approach relies on the pre-existing structure and organization of primitives into more abstract tasks (just one of many possible subsets), and thus avoids such complexity issues. That is, the hierarchy represents a single possible grouping of primitives, and thus limits the possible comparisons between subsets of primitives.

Empirical evidence also shows that using a top-down approach to find merges is potentially faster than finding the set of possible merges at the primitive level. Figure 5 presents an experiment showing the number of plan comparisons performed (checking postconditions against postconditions) to arrive at a merged solution. We ran both our top-down system and a baseline system (that simply checks the primitives of the two different agent plan hierarchies against each other for merges) on a set of hierarchies with a uniform-depth of five, a branching factor of two, and a total ordering over the primitives, where the two hierarchies had varying numbers of possible merges between plans (both abstract and primitive). We performed experiments on an 850 MHz Pentium III with 400 megabytes of RAM. The data points are averages of thirty independent runs, all with standard deviations less than 0.1 seconds. The Abstract Average line represents the average CPU time needed to arrive at the cheapest solution possible given the set of possible plan-step merges, while the Primitive Average line shows the time required to find this merge at the primitive level. As is clear from the graph, as the number of potential merges increases, the time to find an abstract solution using our top-down method stays relatively constant, while computing the solution at the primitive level increases linearly. This is because as the primitive-only algorithm merges more plan steps, the algorithm must spend more time integrating the partial orderings of the plans and checking for potential cycles. When there are more possible merges, our top-down approach tends to identify merges at higher levels of abstraction, requiring less time to integrate partial orderings of the two hierarchies because of the fewer plan steps on the frontiers.

Though the graph in Figure 5 does show some benefit to utilizing a top-down approach to finding synergy, the actual speedup is not as significant as it could potentially be, because the time to perform individual plan comparisons at the abstract level can be significantly longer due to the larger number of summarized conditions at the abstract

level. We still see some improvement at the abstract level because not all summarized *must* conditions at the primitive level are *must* conditions at the abstract level (some are *mays* and thus are not compared by the top-down approach), and some conditions appear more than once at lower levels and are not duplicated at higher levels. Furthermore, by extending our algorithm to handle merges between hierarchies with both positive and negative interactions (rather than only positive ones), we expect to see even faster performance gains from finding solutions at the abstract level, because of the time required to integrate the additional ordering constraints required to resolve the negative interactions. Clement has presented a similar conclusion with regard threat resolution (Clement, 2001).

Another advantage of the top-down approach versus simply merging plan steps at the primitive level hinges on the increased flexibility abstract merges provide agents when they go to execute their plans. To merge at the primitive level, an agent is required to fully decompose its plan hierarchy, committing to a set of choices of how to decompose its plan (at *or* branches) prior to execution. Performing merges at abstract levels leaves an agent with options as to how it will choose to decompose its plan at runtime, leading to more robust behavior. For example, if plan step ***k3*** (figure 2) were to replace another plan step in another agent's hierarchy, the kitchen-cleaning agent would still have the option to either use a sponge or a mop to clean the floor, whereas merging at the primitive level will require a complete decomposition and selection of primitives, forcing the kitchen-cleaning agent to commit to either using the sponge or the mop prior to execution.

Finally, our top-down approach suffers from its own thoroughness, since it effectively generates all possible pairs of frontiers to check for synergy. This completeness results in a large branching factor in the search, causing significant slowdown. We hope to address this issue by adding heuristics to guide the search to solutions faster in future versions of the algorithm.

Conclusions and Future Work

We have presented a top-down methodology for discovering and implementing synergy between agents' hierarchical plans in order to reduce redundant execution. To improve the efficiency of the algorithm, we hope to develop better state space pruning mechanisms or to integrate branch-and-bound techniques limiting the number of search states explored. We currently support partially ordered plans, but would like to support explicit causal links between pre- and post-conditions of plans. Our most recent work has involved the integration of this merging mechanism with Clement's plan conflict resolution mechanism (Clement, 1999b), allowing hierarchical plans to be coordinated and merged in a single process, though this work has yet to be evaluated. We look forward to researching the possibility of merging the incomplete plans of agents, where individually agents would be unable to accomplish their plans, but with the help of others it would be possible. Ultimately, our research goal is to construct a mechanism for determining when it is worthwhile for agents to engage in plan merging (and at what level of abstraction) given the tradeoffs between computation time and execution time.

Acknowledgements

We wish to thank the CoABS Grid development team for making the grid software available to us, and helping us integrate our code with the grid. We would also like to thank Brad Clement for providing us his summarization mechanism as well as other useful supporting code, and Thom Bartold for his help with our grid integration efforts. This research was supported by DARPA (F30602-98-2-0142).

References

Clement, C., and Durfee, E. 1999a. Top-Down Search for Coordinating the Hierarchical Plans of Multiple Agents. *Proceedings of the Third International Conference on Autonomous Agents*, 252-259.

Clement, C., and Durfee, E. 1999b. Theory for Coordinating Concurrent Hierarchical Planning Agents Using Summary Information. *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, 495-502.

Clement, C., 2001. Abstract Reasoning for Multiagent Coordination and Planning. Ph.D. diss., Dept. of Computer Science, University of Michigan.

Cox, Jeffrey, et. al., 2001. "Integrating Multiagent Coordination with Reactive Plan Execution." (abstract) *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-01)*.

DARPA Control of Agent Based Systems Program (CoABS). http://coabs.globalinfotek.com/

Ephrati E., Rosenschein, J 1994. Divide and Conquer in Multi-Agent Planning. *Proceedings of American Association for Artificial Intelligence*, 375-380.

Georgeff, M. P. 1983. Communication and interaction in multi-agent planning. In *Proceedings of American Association for Artificial Intelligence*, 125-129.

Goldman, C. and Rosenschein, J. 1994. Emergent Coordination through the Use of Cooperative State Changing Rules. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 408-413.

Horty, J. and Pollack, M. 2000. Evaluating new options in the context of existing plans. *Artificial Intelligence*, 207:199-220.

Knoblock, C. 1991. Search reduction in hierarchical problem solving. *Proceedings of American Association for Artificial Intelligence*, 686–691.

Korf, R. 1987. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65-88.

Vilain, M. and Kautz, H., 1986. Constraint Propagation Algorithms for Temporal Reasoning. *Proceedings of American Association for Artificial Intelligence*, 377-382.

Von Martial, F., 1990. Interactions Among Autonomous Planning Agents. *Distributed AI*. North Holland: Elsevier.

Yang, Q. 1997. *Intelligent Planning*. New York: Springer-Verlag.