In Proceedings of the DARPA Workshop of Innovative Approaches to
Planning, Scheduling and Control, Morgan Kaufmann, November 1990

Nonlinear Planning with Parallel Resource Allocation

Manuela M. Veloso **M. Alicia Pérez** **Jaime G. Carbonell**

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Most nonlinear problem solvers use a least-commitment search strategy, reasoning about partially ordered plans. Although partial orders are useful for exploiting parallelism in execution, least-commitment is NP-hard for complex domain descriptions with conditional effects. Instead, a casual-commitment strategy is developed, as a natural framework to reason and learn about control decisions in planning. This paper describes (i) how NoLimit reasons about totally ordered plans using a casual-commitment strategy, (ii) how it generates a partially ordered solution from a totally ordered one by analyzing the dependencies among the plan steps, and (iii) finally how resources are allocated by exploiting the parallelism embedded in the partial order. We illustrate our claims with the implemented algorithms and several examples. This work has been done in the context of the PRODIGY architecture that incorporates NoLimit, a nonlinear problem solver.

1 Introduction

Nonlinear problem solving is desired when there are strong interactions among simultaneous goals and subgoals in the problem space. NoLimit, the nonlinear problem solver of the PRODIGY architecture [Carbonell *et al.*, 1990, Veloso, 1989], develops a method to solve problems nonlinearly that explores different alternatives at the operator and at the goal ordering levels. Commitments are made during the search process, in contrast to a least-commitment strategy [Sacerdoti, 1975, Tate, 1977, Wilkins, 1989], where decisions are deferred until all possible interactions are recognized. With the casual-commitment approach [Minton *et al.*, 1989], background knowledge, whether hand-coded expertise, learned control rules, or heuristic evaluation functions, guides the efficient exploration of the most promising parts of the search space. Provably incorrect alternatives are eliminated and heuristically preferred ones are explored first. Casual commitment is crucial because it provides a framework in which it is natural to reason and *learn* about the control decisions of the problem solver.

The immediate output of a problem solver that searches using a casual-commitment strategy is a totally ordered plan. It is advantageous to know the solution in terms of the least constrained partial ordering of its steps, which NoLimit generates by analyzing the dependencies among the different operators. The algorithm implemented constructs a directed acyclic graph that relates preconditions and effects of operators and then translates this graph into a partial order.

The independent actions shown in the partially ordered graph may not directly correspond to parallel executable actions due to resource contention. We show how a resource allocation module further analyzes the partial order and generates the final parallel plan.

This paper is organized in five sections. Section 2 briefly presents the casual-commitment search algorithm discussing its motivation and claims. In section 3, we introduce the algorithm that generates the partially ordered plan from the totally ordered one. In section 4 we describe the method to allocate resources, by analyzing the parallelism of the partially ordered solution. Finally, in section 5, we draw conclusions on this work. We illustrate our concepts, claims, and algorithms with several examples throughout the paper.

2 Nonlinear Problem Solving using Casual Commitment

NoLimit reasons about *totally* ordered plans that are *nonlinear*, i.e., the plans cannot be decomposed into a sequence of complete subplans for the conjunctive goal set. All decision points (operator selections, goal orderings, backtracking points, etc.) are open to introspection and reconsideration. In the presence of background knowledge – heuristic or definitive – only the most promising parts of the search space are explored to produce a solution plan efficiently [Veloso, 1989]. The skeleton of NoLimit's search algorithm, shown in Table 1, describes the basic cycle of the nonlinear planner.

In step 1 of the algorithm, the planner checks whether the goal is true in the current state. If so, the planner has found a solution to the problem. In step 2, it computes both the *set of pending goals* and the *set of applicable operators*. A goal is *pending*, if it is a precondition of a *chosen* operator that is not true in the state. An operator is *applicable*, if all its preconditions are true in the state. In step 3, the planner selects a goal to work on or an operator to apply. If a goal is chosen, the problem solver *expands* the goal in step 4, by generating and selecting a relevant *instantiated operator*. If an applicable operator is selected, then, in step 5, it is applied, i.e. executed in the *internal* current state to produce a new state.

PRODIGY provides a rich action representation language

1. Check if the goal statement is true in the current state, or there is a reason to suspend the current search path.
   If yes, then either return the final plan or backtrack.
2. Compute the *set* of *pending goals* $\mathcal{G}$, and the set of possible *applicable operators* $\mathcal{A}$.
3. Choose a goal G from $\mathcal{G}$ or select an operator A from $\mathcal{A}$ that is directly applicable.
4. If G has been chosen, then
   - *expand goal* G, i.e., get the set $\mathcal{O}$ of *relevant instantiated operators* for the goal G,
   - choose an operator O from $\mathcal{O}$,
   - go to step 1.
5. If an operator A has been selected as directly applicable, then
   - *apply* A,
   - go to step 1.

Table 1: A Skeleton of NoLimit's Search Algorithm.

coupled with an expressive control language. Preconditions in the operators can contain conjunctions, disjunctions, negations, and both existential and universal quantifiers with typed variables. Effects in the operators can contain conditional effects, which depend on the state in which the operator is applied. The control language allows the problem solver to represent and learn control information about the various problem solving decisions, such as selecting which goal/subgoal to address next, which operator to apply, what bindings to select for the operator or where to backtrack in case of failure. Different disciplines for controlling decisions can be incorporated [Drummond and Currie, 1989, Anderson and Farley, 1990]. In PRODIGY, there is a clear division between the declarative domain knowledge (operators and inference rules) and the more procedural control knowledge. This simplifies both the initial specification of a domain and the incremental learning of the control knowledge [Minton, 1988, Veloso and Carbonell, 1990].

Previous work in the linear planner of PRODIGY used explanation-based learning techniques [Minton, 1988] to extract from a problem solving trace the explanation chain responsible for a success or failure and compile search control rules. We are now extending this work to NoLimit, as well as developing a derivational-analogy approach to acquire control knowledge [Carbonell, 1986, Veloso and Carbonell, 1990]. The machine learning and knowledge acquisition work supports NoLimit's casual-commitment method, as it assumes there is *intelligent* control knowledge, exterior to its search cycle, that it can rely upon to make decisions.

2.1 Example

Consider a generic transportation domain with three simple operators that load, unload, or move a carrier, as shown in Figure 1 (variables in the operators are shown in bold face).

Suppose that the operator MOVE a carrier has *constant* locations $locA$ and $locB$. This transforms the current general domain into a *one-way* carrier domain. The problem we want to solve consists in moving two given objects $obj1$ and $obj2$ from the location $locA$ to the location $locB$ using a ROCKET as the carrier, for example. Without any control knowledge the problem solver searches for the goal ordering that enables the problem to be solved. Accomplishing either goal individually, as a linear planner would do, in-

```
(LOAD                        (UNLOAD                      (MOVE
 (preconds                    (preconds                    (preconds
  (and                         (and                         (at carrier locA))
   (at obj loc)                 (inside obj carrier)       (effects
   (at carrier loc)))           (at carrier loc)))          (add (at carrier locB))
 (effects                     (effects                      (del (at carrier locA))))
  (add (inside obj carr))      (add (at obj loc))
  (del (at obj loc))))         (del (inside obj carrier))))
```

Figure 1: A Transportation Domain.

hibits the accomplishment of the other goal, as a precondition of the operator LOAD cannot be achieved: the ROCKET cannot be moved back to the object's initial position. So interleaving of goals and subgoals at different levels of the search is needed to find a solution. NoLimit solves this problem, where linear planners fail (but where, of course, other least-commitment planners also succeed), because it switches attention to the conjunctive goal *(at obj2 locB)* before completing the first conjunct *(at obj1 locB)*. This is shown in Figure 2 by noting that, after the plan step 1 where the operator (LOAD ROCKET obj1 locA) is applied, NoLimit changes its focus of attention to the other top-level goal and applies the operator (LOAD ROCKET obj2 locA). NoLimit returns the totally ordered solution (LOAD ROCKET obj1 locA), (LOAD ROCKET obj2 locA), (MOVE ROCKET), (UNLOAD ROCKET obj1 locB), (UNLOAD ROCKET obj2 locB).

Figure 2: The Complete Conceptual Tree for a Successful Solution Path. The numbers at the nodes show the execution order of the plan steps.

Clearly, NoLimit solves much more complex and general versions of this problem. The present minimal form was used to illustrate the casual-commitment strategy in nonlinear planning, allowing full interleaving of goals and subgoals. We present below examples with a complex logistics domain.

3 Total and Partial Orders

A partially ordered graph is a convenient way to represent the ordering constraints that exist among the steps of the plan. Consider the partial order as a directed graph (V, E), where V, the set of vertices, is the set of steps (instantiated operators) of the plan, and E is the set of edges (ordering constraints) in the partial order. Let $V = \{op_0, op_2, \ldots, op_{n+1}\}$. We represent the graph as a square matrix P, where $P[i,j] = 1$, if there is an edge from op_i to op_j. There is an edge from op_i to op_j, if op_i must *precede* op_j, i.e. $op_i \prec op_j$. The inverse of this statement does not necessarily hold, i.e. there may be the case where $op_i \prec op_j$ and there is not an edge from op_i to op_j. The relation $\prec$ is the *transitive closure* of the relation represented in the graph for the partial order. Without loss of generality consider operators op_0 and op_{n+1} of any plan to be the *additional* operators named *start* and *finish*, represented in the Figures below as s and f.

3.1 Transforming a Total Order into a Partial Order

A plan step op_i necessarily precedes another plan step op_j if and only if op_i adds a precondition of op_j, or op_j deletes a precondition of op_i. For each problem, the start operator s adds all the literals in the initial state. The preconditions of the finish operator f are set to the user-given goal statement. Let the totally ordered plan $\mathcal{T}$ be the sequence $op_1, \ldots, op_n$ returned by NoLimit as the solution to a problem. In Table 2, we show the algorithm to generate the partially ordered plan from this totally ordered one, $\mathcal{T}$.

Input: A totally ordered plan $\mathcal{T} = op_1, op_2, \ldots, op_n$, and the start operator s with preconditions set to the initial state.
Output: A partially ordered plan shown as a directed graph $\mathcal{P}$.

```
procedure Build_Partial_Order($\mathcal{T}$, $s$):
1.  for $i \leftarrow$ n down-to 1 do
2.      for each precond in Preconditions_of($op_i$) do
3.          supporting_operator ←
                ← Last_Op_Adding_Precond(precond,i)
4.          Add_Directed_Edge(supporting_operator,$op_i$,$\mathcal{P}$)
5.      for each del in Delete_Effects($op_i$) do
6.          supported_operators ←
                ← All_Ops_Needing_Effect(del,i)
7.          for each supported_operator do
8.              Add_Directed_Edge(supported_operator,$op_i$,$\mathcal{P}$)
9.      for each add in Primary_Adds($op_i$) do
10.         adversary_operators ←
                ← Ops_Deleting_Primary_Add(add,i)
11.         for each adversary_operator do
12.             Add_Directed_Edge(adversary_operator,$op_i$,$\mathcal{P}$)
13. $\mathcal{P}$ ← Remove_Transitive_Edges($\mathcal{P}$)
```

Table 2: Building a Partial Order from a Total Order

Step 1 loops through the plan steps in the *reverse* of the execution order. Lines 2-4 loop through each of the preconditions of the operator, i.e. plan step. The procedure Last_Op_Adding_Precond (not shown) searches from the operator op_i back to, at most the operator s, for the first operator (supporting_operator) that has the effect of adding the precondition in consideration. Note that one such operator must be found as the given $\mathcal{T}$ is a solution to the problem (in particular the initial state is added by the operator s). All the supporting_operators of an operator op_i must precede it. The algorithm sets therefore a directed edge from each of the former into the latter. Lines 5-8 similarly loop through each of the delete effects of the operator. The procedure All_Ops_Needing_Effect (not shown) searches for all the earlier operators that need, i.e. have as a precondition, each delete effect of the operator. We call such operators, supported_operators. Lines 7-8 capture the precedence relationships by adding directed edges from each supported_operator to the operator that deletes some of their preconditions. Lines 9-12 guarantee that the primary adds of this operator are kept in the partial order. An add effect is primary if it in the subgoaling chain of a user given goal conjunct. The procedure Ops_Deleting_Primary_Add identifies the the adversary_operators that, earlier in the plan, delete a primary add. Any such operator cannot be performed after the current operator. Hence line 12 sets a directed edge from each adversary_operator to the operator under consideration. Finally, line 13 removes all the transitive edges of the resulting graph to produce the partial order. Every directed edge e connecting operator op_i to op_j is removed, if there is another path that connects the two vertices. The procedure Remove_Transitive_Edges tentatively removes e from the graph and then checks to see whether vertex op_j is reachable from op_i. If this is the case, then e is removed definitively, otherwise e is set back in the graph.

If n is the number of operators in the plan, p is the average number of preconditions, d is the average number of delete effects, and a is the average number of add effects of an operator, then steps 1-12 of the algorithm Build_Partial_Order run in $O((p + d + a)n^2)$. Note that the algorithm takes advantage of the given total ordering of the plan, by visiting, at each step, only earlier plan steps. The final procedure Remove_Transitive_Edges runs in $O(e)$, for a resulting graph with e edges [Aho *et al.*, 1974]. Empirical experience with test problems shows that the algorithm Build_Partial_Order runs in meaningless time compared to the search time to generate the input totally ordered plan.

We now illustrate the algorithm in the simple *one-way* rocket problem introduced in the previous section. NoLimit returned the totally ordered plan $\mathcal{T}$ = (LOAD ROCKET obj1 locA), (LOAD ROCKET obj2 locA), (MOVE ROCKET), (UNLOAD ROCKET obj1 locB), (UNLOAD ROCKET obj2 locB). Let op_i be the ith operator in $\mathcal{T}$. In Figure 3 we show the partial order generated by the algorithm, before removing the transitive edges. As previously seen, the goal of the problem we solved is the conjunction *(and (at obj1 locB) (at obj2 locB))*. These two predicates are added by the UNLOAD steps, namely by op_4 and op_5 respectively. The edges labelled "g" show the precedence requirement between op_4 and op_5, and the finish operator f. The numbers at the other edges in Figure 3 represent the order by which the algorithm introduces them into the graph.

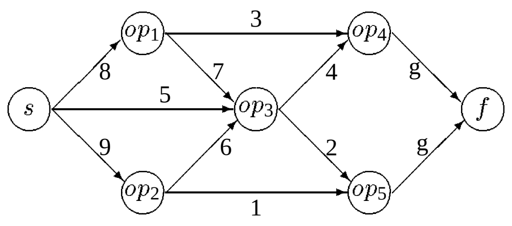
$$\begin{tikzcd}
 & op_1 \arrow[rr, "3"] \arrow[dr, "7"] & & op_4 \arrow[dr, "g"] & \\
s \arrow[ur, "8"] \arrow[rr, "5"] \arrow[dr, "9"] & & op_3 \arrow[ur, "4"] \arrow[dr, "2"] & & f \\
 & op_2 \arrow[rr, "1"] \arrow[ur, "6"] & & op_5 \arrow[ur, "g"] &
\end{tikzcd}$$

Figure 3: Partial Order with Transitive Edges.

As an example, while processing op_5 (UNLOAD ROCKET obj2 locB), it sets the edges 1 and 2, as the preconditions of op_5, namely *(inside obj1 ROCKET)* and *(at ROCKET locB)* (see Figure 1), are added by op_2 and op_3 respectively. When processing op_3 (MOVE ROCKET), edge 5 is set because op_3's precondition *(at ROCKET locA)* is in the initial state. The edges 6 and 7 are further set, because op_3 deletes *(at ROCKET locA)* that is needed (as a precondition) by the earlier steps op_1 and op_2. Removing the transitive edges, namely edges 1, 3, and 5, in this graph results in the final partial order.

4 Exploiting Parallelism in the Plan Steps

When there are multiple execution-time agents in a domain, they must be able to organize their activities so that they can cooperate with one another (e.g. to push a very heavy block) and avoid conflicts (e.g. not to tyr to use the same tool at the same time).

Our approach for doing multiagent planning is a centralized one [Georgeff, 1983, Lansky and Fogelsong, 1987]. An initial planning phase produces a plan as parallel as possible by reasoning about a presumably infinite number of resources. Real available resources are then assigned to obtain the final parallel plan [Wilkins, 1989]. A problem is first solved creating generic instances of the resources. In this context, "resources" refer to agents, such as robots, or trucks or airplanes in a logistics transportation domain, or machines in a process planning domain. Control rules assign different resources to different unrelated goals to obtain a plan as parallel as possible. In some cases the same resource can be used to solve different unrelated goals. For example, it is better to load different objects in the same truck if they have the same destination), if minimization of resources usage is preferred by the control knowledge.

Let $\mathcal{T}$ be the resulting plan and s the start operator. Table 3 outlines the algorithm for resource allocation.

1. Generate the partial order graph $\mathcal{P}$ using the algorithm in Table 2 with inputs $\mathcal{T}$ and s.
2. Insert parallel split and join nodes in the partial order graph $\mathcal{P}$ obtaining a graph $\mathcal{P}'$.
3. Recursively analyze in $\mathcal{P}'$ the parallel branches inside a split-join pair. If some of the parallel branches are in conflict insert sequential split and join nodes. If all the parallel branches are in conflict, transform the parallel split-join pair into a sequential one. Let $\mathcal{P}''$ be the resulting graph.
4. From $\mathcal{P}''$, assign real resources to the generic instances.
5. Assign plans to the individual resources and monitor their execution to avoid conflicts.

Table 3: Algorithm for Resource Allocation.

In step 1 the algorithm section 3.1 generates the partial order graph from $\mathcal{T}$. Step 2 extends this graph with nodes that are not associated with steps in the plan. They only serve as guidelines to determine which actions can be executed in parallel. If a node op_i has several successors op_{i_1}, ..., op_{i_n}, a *parallel split node* is inserted having op_i as a predecessor and op_{i_1}, ..., op_{i_n} as successors. The edges between op_i and op_{i_1}, ..., op_{i_n} are removed. Similarly, if a node op_j has several predecessors op_{j_1}, ...,op_{j_n}, a *parallel join node* is inserted having op_j as only successor and op_{j_1}, ..., op_{j_n} as predecessors. The edges between op_{j_1}, ..., op_{j_n} and op_j are removed.

Step 3 analyzes the parallel branches. It may be necessary to add *sequential split* and *join nodes* to the graph, or replace some of the parallel ones. The branches inside a sequential split-join pair must be executed sequentially although any order is allowed.

A class of objects C can be declared as a possible reason for conflict. Two *actions* are in conflict if they use the same instance of C, and hence they are not allowed to occur simultaneously. A conflict between two *branches* is detected when there is not a pair of actions, one of each branch, that can be executed at the same time. If *all* the actions of the two branches are in conflict, they are enclosed in a sequential split-join pair. If only *some* of them are, the parallel split-join remains. Committing to executing the branches in sequence would constrain the parallelism in the plan, as the actions not in conflict could still be done simultaneously. As we describe below, an execution monitor is responsible for avoiding that the conflicting actions are performed simultaneously. This analysis is done recursively to deal with nested split-join pairs.

Step 4 assigns real resources to the generic instances, by recursively analyzing the branches inside a split-join pair. If enough resources are available, the algorithm assigns different ones to each branch. Otherwise the available resources are shared by several branches. These branches are put inside a sequential split-join pair so the monitor can execute them without conflicts. The planner may have to be called again to obtain the actions that situate the real resource in the same initial state as the generic one it replaces.

From the global parallel plan obtained so far, step 5 generate plans for each of the agents or resources. A monitor module is responsible for synchronizing the execution of the different plans (for example, in the case when two or more agents are necessary to perform an action). It uses the sequential split and join nodes to deal with conflicts or resource sharing among different branches. Those conflicts can be considered as critical regions. Standard operating systems methods can be used to enforce synchronization in the plans so the conflicting critical regions are not entered at the same time [Georgeff, 1983].

4.1 Example in the Extended-STRIPS Domain

To illustrate this we will consider a simple example where two robots, R1 and R2, have to move two blocks, a heavy one H, and a light one L. The two robots have to cooperate to push H. The domain is an extension of the STRIPS domain; the operators include going to locations, going through doors and pushing objects to locations. There are also "team" operators that require the cooperation of two robots to perform an action (e.g. t-push-to-location). Only one robot can go through a door at a time, therefore doors are considered reasons for potential conflicts. Figures 4 (a) and (b) show the initial state and goal statement, and (c) shows the initial state using generic robots GR1, GR2 and GR3.

The problem is first solved with generic robots. Their initial situation was decided based on domain dependent heuristics such as the initial situation of the available robots and of the objects that have to be pushed. The solution is:

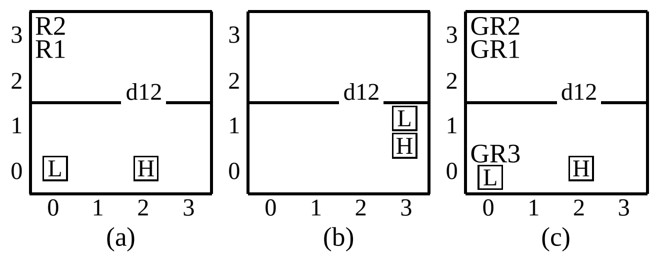

Figure 4: Initial State, Goal Statement, and Initial State with Generic Resources for the Example Problem. Coordinates represent the locations within the rooms.

1 (goto-loc GR1 3 0 2 2)
2 (go-thru-door GR1 door12 2 2 2 1)
3 (goto-loc GR1 2 1 2 0)
4 (goto-loc GR2 3 0 2 2)
5 (go-thru-door GR2 door12 2 2 2 1)
6 (goto-loc GR2 2 1 2 0)
7 (t-push-to-loc GR1 GR2 heavy-block 2 0 3 1)
8 (push-to-loc GR3 light-block 0 0 3 1)

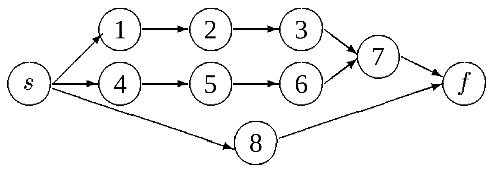

Figure 5: Partial Order Graph for the Example Problem.

Figure 5 shows the partial order generated by the algorithm in section 3. The only conflict is between 2 and 5 when GR1 and GR2 try to go through the door at the same time. As the other actions (1, 3, 4, 6) in the parallel branches do not conflict, these branches are not put inside a sequential split join pair. The resource assignment step assigns R1 to GR1, and R2 to both GR2 and GR3. After this step the graph looks like in Figure 6.

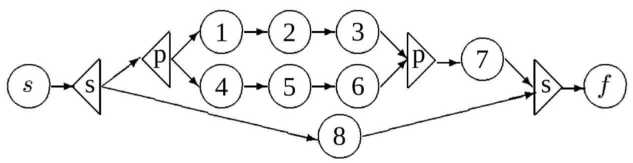

Figure 6: Graph after Assigning Resources.

Now the task of the monitor is to control the plan execution avoiding the conflict at the door and deciding which of the two branches will be executed first. The planner is called to plan the actions of R2 to join the end of branch 1-2-3 with the beginning of branch 4-5-6. A resulting parallel plan is the one shown below, where branch 1-2-3, and branch 4-5-6 are monitored to be executed in parallel avoiding the conflict between steps 2 and 5. Step 7' is added to the plan.

monitored-parallel-split
1 (goto-loc R1 3 0 2 2)
2 (go-thru-door R1 door12 2 2 2 1)
3 (goto-loc R1 2 1 2 0)

4 (goto-loc R2 3 0 2 2)
5 (go-thru-door R2 door12 2 2 2 1)
6 (goto-loc R2 2 1 2 0)
monitored-parallel-join

7 (t-push-to-loc R1 R2 heavy-block 2 0 3 1)
7' (goto-loc R2 3 1 0 0)
9 (push-to-loc R2 light-block 0 0 3 1)

4.2 Example in the Logistics Domain

We are currently implementing a complex logistics planning domain. In this domain, packages are to be moved among different cities. Packages are carried within the same city in trucks and across cities in airplanes. Trucks and airplanes may have limited capacity. At each city there are several locations, e.g. post offices (po) and airports (ap). This domain (without introducing the capacity of carriers) is an extension of the generic transportation domain (see Figure 1). Consider carriers of type TRUCK and AIRPLANE. The logistics domain consists of the operators LOAD TRUCK (LT), LOAD AIRPLANE (LA), UNLOAD TRUCK (UT), UNLOAD AIRPLANE (UA), DRIVE TRUCK (DT), FLY AIRPLANE (FA). Consider the problem shown in Figure 7 where bo, pg and sf stand for Boston, Pittsburgh and San Francisco respectively. There are three packages (*p1, p2, p3*), two airplanes (*a1, a2*), and four trucks (*tbo1, tbo2, tsf, tpg*). NoLimit returns the plan in Figure 8, and Figure 9 shows the partial order generated by the algorithm in Table 2.

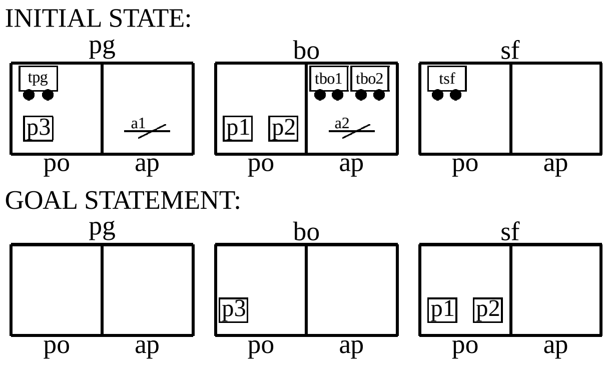

Figure 7: A Problem in the Logistics Domain.

1.(LT p3 tpg pg-po)
2.(DT tsf sf-po sf-ap)
3.(DT tpg pg-po pg-ap)
4.(UT p3 tpg pg-ap)
5.(LA p3 a1 pg-ap)
6.(FA a1 pg-ap bo-ap)
7.(UA p3 a1 bo-ap)
8.(LT p3 tbo1 bo-ap)
9.(DT tbo1 bo-ap bo-po)
10.(UT p3 tbo1 bo-po)
11.(DT tbo2 bo-ap bo-po)
12.(LT p2 tbo2 bo-po)
13.(LT p1 tbo2 bo-po)
14.(DT tbo2 bo-po bo-ap)
15.(UT p2 tbo2 bo-ap)
16.(UT p1 tbo2 bo-ap)
17.(LA p2 a2 bo-ap)
18.(LA p1 a2 bo-ap)
19.(FA a2 bo-ap sf-ap)
20.(UA p2 a2 sf-ap)
21.(UA p1 a2 sf-ap)
22.(LT p2 tsf sf-ap)
23.(LT p1 tsf sf-ap)
24.(DT tsf sf-ap sf-po)
25.(UT p2 tsf sf-po)
26.(UT p1 tsf sf-po)

Figure 8: Totally Ordered Plan - Logistics Domain.

Suppose now that when executing this plan, there is available only one airplane (a) and only one truck in Boston (tbo). The resource allocation algorithm assigns a to both $a1$ and $a2$, and tbo to both $tbo1$ and $tbo2$ after generating the parallel serial graph. Figure 10 shows a possible solution for the plans of a and tbo. Using the information on the graph built by

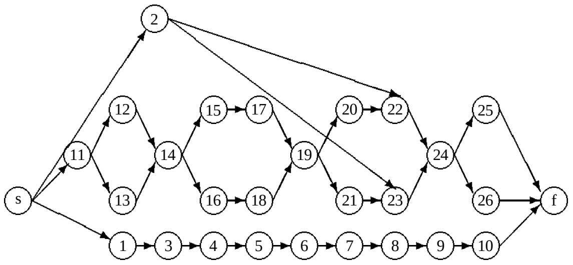

Figure 9: Partially Ordered Plan - Logistics Domain.

the algorithm, the monitor synchronizes the execution of the plans for the different agents, without violating the constraints discovered by the algorithm.

Plan for airplane a:	*Plan for truck tbo:*
(LA p3 a pg-ap)	(DT tbo bo-ap bo-po)
(FA a pg-ap bo-ap)	(LT p2 tbo bo-po)
(UA p3 a bo-ap)	(LT p1 tbo bo-po)
(LA p2 a bo-ap)	(DT tbo bo-po bo-ap)
(LA p1 a bo-ap)	(UT p2 tbo bo-ap)
(FA a bo-ap sf-ap)	(UT p1 tbo bo-ap)
(UA p2 a sf-ap)	(LT p3 tbo bo-ap)
(UA p1 a sf-ap)	(DT tbo bo-ap bo-po)
	(UT p3 tbo bo-po)

Figure 10: Plans for Each Resource.

We are refining the monitor synchronization mechanism to deal with more complex conflict constraints, by using domain dependent heuristics.

5 Conclusion

In this paper, we first discuss the use of a casual-commitment strategy to generate plans for nonlinear problems. This strategy provides a natural framework to learn and reason about control decisions during the planning process. The method becomes increasingly efficient as the planner learns control knowledge from experience. Committing while searching generates a totally ordered solution. As it is advantageous to know the least constrained partial ordering of the plan steps, we then discuss how we efficiently generate a partial order from the total order returned by the casual-committing problem solver. Finally, we show a resource allocation strategy that reasons about the partially ordered plan to convert it into a parallel executable graph.

This work has been done in the context of the PRODIGY architecture that is designed as a testbed for machine learning research. Casual commitment relies upon learned control knowledge to efficiently make decisions. The resource allocation module is an ongoing research effort to address multiagent (or multi-resource) planning and execution.

Acknowledgements

Our special thanks to Daniel Borrajo for a major part of NoLimit's implementation. Without him, it would have been very difficult to include many powerful features in NoLimit. We acknowledge Craig Knoblock and Yolanda Gil for providing useful comments on a draft. The authors thank the whole PRODIGY research group for helpful discussions.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OH 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the U.S. Government. The second author was supported by a Fellowship from the Ministerio de Educación y Ciencia of Spain.

References

[Aho *et al.*, 1974] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[Anderson and Farley, 1990] John S. Anderson and Arthur M. Farley. Partial commitment in plan composition. Technical Report TR-90-11, Computer Science Department, University of Oregon, 1990.

[Carbonell *et al.*, 1990] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. Prodigy: An integrated architecture for planning and learning. In K. VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990. Also Technical Report CMU-CS-89-189.

[Carbonell, 1986] Jaime G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach, Volume II*, pages 371–392. Morgan Kaufman, 1986.

[Drummond and Currie, 1989] Mark Drummond and Ken Currie. Goal ordering in partially ordered plans. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 960–965, Detroit, MI, 1989.

[Georgeff, 1983] M. Georgeff. Communication and interaction in multi-agent planning. In *Proceedings of the National Conference of the American Association for Artificial Intelligence*, pages 125–129, Washington, DC, August 1983.

[Lansky and Fogelsong, 1987] A. L. Lansky and D. S. Fogelsong. Localized representation and planning methods for parallel domains. In *Proceedings of the National Conference of the American Association for Artificial Intelligence*, pages 240–245, Seattle, Washington, August 1987.

[Minton *et al.*, 1989] Steven Minton, Craig A. Knoblock, Dan R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, Boston, MA, 1988.

[Sacerdoti, 1975] Earl D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of IJCAI-75*, pages 206–213, 1975.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–900, 1977.

[Veloso and Carbonell, 1990] Manuela M. Veloso and Jaime G. Carbonell. Integrating analogy into a general problem-solving architecture. In Maria Zemankova and Zbigniew Ras, editors, *Intelligent Systems*, pages 29–51. Ellis Horwood, Chichester, England, 1990.

[Veloso, 1989] Manuela M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.

[Wilkins, 1989] David E. Wilkins. Can AI planners solve practical problems? Technical Note 468R, SRI International, 1989.