Nonlinear Problem Solving Using Intelligent Casual-Commitment

Manuela M. Veloso

December 1989
CMU-CS-89-210 3

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Complex interactions among conjunctive (simultaneous) goals motivate the need for nonlinear planners. Whereas the literature addresses least-commitment approaches that require breadth-first search and theorem proving style-reasoning to seek a possible answer, we advocate a casual-commitment approach that finds viable plans incrementally. In essence, all decision points (operator selections, goal orderings, backtracking points, etc.) are open to introspection and reconsideration. However, in the presence of background knowledge - heuristic or definitive - only the most promising parts of the search space will be explored in satisficing mode to produce a solution plan efficiently. In the limiting case, however, casual commitment can backtrack, explore the entire space subsuming all goal orderings, and generate partial orders guaranteeing synthesis of all possible plans including the optimal one. This paper reports on the full implementation of the efficient, casual-commitment nonlinear problem solver of the PRODIGY architecture. The principles of nonlinear planning are discussed, the algorithms in the implementation are described in some detail, and the use of knowledge (if present) to focus search is considered.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), United States Air Force, Wright-Patterson AFB, Ohio 45433-6543.

Contents

1. Introduction

A *nonlinear* problem solver is able to explore and exploit interactions among multiple conjunctive goals, whereas a *linear* one must address each goal in sequence, independent of all the others. Hence, nonlinear problem solving is desired when there are strong interactions among simultaneous goals and subgoals in the problem space. We explore a method to solve problems nonlinearly, that generates and tests 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 [8,10,12], where decisions are deferred until all possible interactions are recognized. This approach has been implemented as the kernel of the PRODIGY architecture [6], and we refer to this system as **NoLimit**, standing for **No**nlinear problem solver using casual commit**m**ent. NoLimit outputs a set of partially ordered solutions assembled from totally ordered solutions produced by the search process.

The paper is organized in six sections. In section 2 we motivate our approach with a discussion on the issues that differentiate linear and nonlinear problem solving. Section 3 describes the general search procedure used by NoLimit which we instantiate with a complete example. Section 4 discusses the use of control rules, heuristics to guide the search mechanism. In section 5 we present the algorithms that translate totally ordered solutions into partially ordered ones and vice versa. Finally, in section 6, we draw conclusions and summarize the research contributions of NoLimit. Throughout the paper the terms *problem solving* and *planning* are used interchangeably as are the terms *solution* and *plan*.

2. Motivation

Consider the following idealized planning problem: given a formal description of an initial state of the world, a set of operators that can be executed to transition from one world state to another, and a goal statement, find a **plan**, to *transform* the initial state into a final state in which the goal statement is true. Consider that the goal statement is defined as a conjunction of goals. This raises the issue of how to deal with possible interactions among the conjuncts [3]. A simple approach, followed by early planners, such as STRIPS [4], is to solve one goal at a time. A final solution to a problem with initial goals $G_1, \ldots, G_k$ is a sequence of complete subsolutions to each one of the goals. Note that there is an underlying assumption of independence among the goals $G_1, \ldots, G_k$. This approach can be slightly improved by allowing any permutation of the goals to be considered. Thus, if the plan for G_j deletes G_i, for $j > i$, then the planner will reorder G_j before G_i. Another level of improvement is obtained if the problem solver can reconsider a goal that was achieved once and then deleted while working on a different goal. The planner reaches a solution when all the goals $G_1, \ldots, G_k$ are true in some world state. For now, we refer to this approach as **linear** planning. Later we return to this issue and try to identify what specific characteristics of this approach have been identified with the term **linear** planning.

2.1. Linear problem solving

Linear planning suffers from both non-optimality and incompleteness: non-optimality in terms of finding solutions that involve doing and undoing operators unnecessarily; incompleteness in terms of missing a solution to problems when one exists. Both these problems are due to the fact, mentioned above, that linear planning works on one goal at a time. We present two examples below that illustrate these problems.

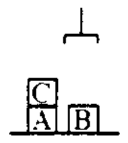
2.1.1. An example on the non-optimal character of linear planning

The problem described below is known as the *Sussman anomaly* as it was identified by Sussman in [9]. Consider the blocksworld with the following operator:

- *MOVE(x,y,z)* moves block x from the top of y to the top of z. y and z can be either the table or another block. *MOVE* is applicable only if x and z are clear, and x is on y. The table always has clear space. A block is clear if it does not have any other block on top of it.

The problem is shown in Figure 1. Note that the goal statement is expressed as a conjunction of two literals. The goal statement does not fully describe the final desired state. Instead it specifies only the conditions that must be met in order to consider the problem solved.

Initial state:
(on B table)
(on A table)
(on C A)
(clear C)
(clear B)



Goal statement:
(and
(on A B)
(on B C))

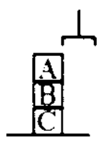
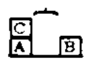
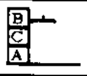
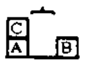
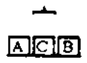
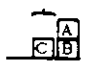
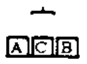
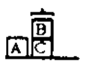
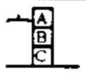


Figure 1 - The Sussman anomaly: Find a plan to transform the initial state to achieve the goal statement.

In Figure 2 we show two plans generated by a linear planner augmented to consider different permutations of the conjunctive goals, and work on a single goal more than once, i.e., admitting that a goal might need to be reachieved. For both plans, the initial state and the goal statement are the ones shown in Figure 1.

The two plans differ in the choice of the first goal considered. It is clear that both plans are non-optimal, as both have actions that are done and undone unnecessarily. For example, in the first plan, B is moved from the table to the top of C, to achieve the goal *(on B C)*. It is then moved back to the table to clear C so that the goal *(on A B)* may be achieved. These inefficiencies arise because the linear planner *forgets* about the other goals while trying to achieve a particular goal in the conjunctive set. More formally, this means that, if the goal statement is the conjunction of goals $G_1, \ldots, G_k$, the linear planner does not consider any of the goals $G_j, j \neq i$, when working on goal G_i. An optimal solution to the Sussman anomaly is the three-step plan: *(MOVE C A table)*, *(MOVE B table C)*, *(MOVE A table B)*.

Goal	Step of the Plan	State
		
(on B C)	(MOVE B table C)	
(on A B)	(MOVE B C table)	
	(MOVE C A table)	
	(MOVE A table B)	
(on B C)	(MOVE A B table)	
	(MOVE B table C)	
(on A B)	(MOVE A table B)	

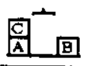
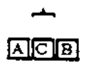
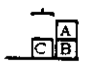
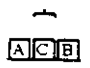
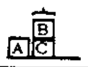
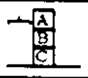
Goal	Step of the Plan	State
		
(on A B)	(MOVE C A table)	
	(MOVE A table B)	
(on B C)	(MOVE A B table)	
	(MOVE B table C)	
(on A B)	(MOVE A table B)	

Figure 2 - Two linear plans that solve the Sussman anomaly inefficiently.

Non-optimality is a problem that could, however, be overcome by a post-processing module that removes unnecessary steps after the planning is completed [7]. It is not straightforward to think of a general way to deal with arbitrary repetitions of the same goal and other suboptimal plan steps. Detecting loops in the state is not a guaranteed mechanism, as a situation could occur where an operator would always change the state but in *irrelevant* details regarding the goals. We can say that in this particular example of the Sussman anomaly the linear planner is lucky to find a solution, even if non-optimal, by working repeatedly on the same goals. In general, however, linear planners may fail drastically, as we discuss below.

2.1.2. An example on the incompleteness of linear planning

A much more serious problem occurs when a linear planner fails to solve a problem that could be solved if goal interactions were properly considered through interleaving of subgoals. In the example we show below, the linear planner fails to produce any solution at all. Consider the set of operators given in Figure 3 that define the *ONE-WAY-ROCKET* domain. Variables in the operators are represented by framing their name with the signs "<" and ">". In this world, there are variable objects and locations, and one constant ROCKET. An object can be loaded into the ROCKET at any location by applying the operator LOAD-ROCKET. Similarly, an object can be unloaded from the ROCKET at any location by using the operator UNLOAD-ROCKET. The operator MOVE-ROCKET shows that the ROCKET can move only from the constant location *locA* to the constant location *locB*. Although NoLimit will solve much more complex and general versions of this problem, the present minimal form suffices to illustrate the need for nonlinear planning.

```
(LOAD-ROCKET                  (UNLOAD-ROCKET                  (MOVE-ROCKET
 (preconds                     (preconds                       (preconds
  (and                          (and                            (at ROCKET locA))
   (at <obj> <loc>)              (inside <obj> ROCKET)         (effects
   (at ROCKET <loc>)))           (at ROCKET <loc>)))            (add (at ROCKET locB))
 (effects                      (effects                         (del (at ROCKET locA))))
  (add (inside <obj> ROCKET))   (add (at <obj> <loc>))
  (del (at <obj> <loc>))))      (del (inside <obj> ROCKET))))
```

Figure 3 - The three operators defining the *ONE-WAY-ROCKET* domain.

Initial state:	Goal statement:
(at obj1 locA)	(and
(at obj2 locA)	(at obj1 locB)
(at ROCKET locA)	(at obj2 locB))

Figure 4 - A problem in the *ONE-WAY-ROCKET* world.

The problem we want to solve consists in moving two given objects *obj*1 and *obj*2 from the location *locA* to the location *locB* as expressed in Figure 4. In Figure 5 we show the two incomplete plans that a linear planner produces before failing. The two possible permutations of the conjunctive goals are tried without success. Accomplishing either goal individually inhibits the accomplishment of the other goal as a precondition of the operator LOAD-ROCKET cannot be achieved. The ROCKET cannot be moved back to the object's initial position. An example of a solution to this problem is the following plan: (LOAD-ROCKET obj1 locA), (LOAD-ROCKET obj2 locA), (MOVE-ROCKET), (UNLOAD-ROCKET obj1 locB), (UNLOAD-ROCKET obj2 locB).

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA) (MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	*failure*

Goal	*Plan*
(at obj2 locB)	(LOAD-ROCKET obj2 locA) (MOVE-ROCKET) (UNLOAD-ROCKET obj2 locB)
(at obj1 locB)	*failure*

Figure 5 - Two failed linear plans for the *ONE-WAY-ROCKET* problem. The second conjunctive goal cannot be achieved because the ROCKET cannot return to pick up the remaining object.

The failure presented is due to the *irreversibility* of the operator MOVE-ROCKET, combined with the linear strategy used. We introduce below a general categorization of operators with respect to reversibility. Suppose that an operator O applies to a state of the world S_{old} and produces a new state of the world S_{new}.

- An operator is **reversible** if there is a sequence of operators $O_1 \dots O_k$ that produce the state S_{old} when applied to S_{new}, i.e., $O_k(O_{k-1}(\dots(O_1(S_{new})\dots) = S_{old}$.
  - An operator is **easily reversible** if both the number k of operators and the cost of applying the k operators are *reasonable* within an accepted metric. In the simplest case, there exists a single operator O_j such that $O_j(S_{new}) = S_{old}$.

- An operator is **not easily reversible** otherwise.

- An operator is **irreversible** if there is no sequence of operators that apply to the state S_{new} producing the state S_{old} again.

Linear planners may generate non-optimal solutions in the presence of reversible operators and may fail to find solutions in the presence of irreversible operators. Planning with irreversible operators requires special mechanisms to *avoid* artificial deadends. We will show later how NoLimit deals with these problems due to its nonlinear character.

2.2. Nonlinear problem solving

There has been some ambiguity in previous work in the use of the terms **linear** and **nonlinear** planning. Linear planning has been used in the context of planners that generate totally ordered plans. We discuss below why we think total ordering is not specific to linear planners.

We claim that linear planning refers to the following *interdependence* characteristics:

- searching using a *stack* of goals, not allowing therefore interleaving of goals at different depths of search,
- generating solutions as sequential concatenation of complete subsolutions for conjunctive goals, and, recursively, for conjunctive subgoals.

The notion of nonlinear planning was motivated by recognizing problems like the Sussman anomaly in a linear planner such as STRIPS [9]. The approach proposed to face this anomaly consisted of deferring taking decisions while building the plan [8]. The result of a planner that follows this least-commitment strategy is a partially ordered plan as opposed to a totally ordered one, and consequently the term *nonlinear plan* is used. However, the essence of the *nonlinearity* is not in the fact that the plan is partially ordered, but in the fact that a plan need not be a linear concatenation of complete subplans. NoLimit can generate *totally* ordered plans that are *nonlinear*, i.e., they cannot be decomposed into a sequence of complete subplans for the conjunctive goal set. Therefore generating totally ordered plans is not, per se, a true characteristic of a linear planner. (In fact a totally ordered plan is itself a *degenerate* partially ordered one.)

Summarizing, we believe that nonlinear planning refers to the following characteristics:

- searching using a *set* of goals, allowing therefore interleaving of goals and subgoals at different depths of search,
- generating solutions that are not necessarily a sequence of complete subsolutions for the conjunctive goals.

In both linear and nonlinear planning, the final solution can be presented as a partially ordered plan, as one can be built from a set of totally ordered plans. We present later in this paper the algorithm that we use to accomplish this transformation and show how it supports this claim. To conclude our general discussion about linear and nonlinear planning , we next discuss the complexity of using a least-commitment strategy and that of an intelligent casual-commitment one.

2.2.1. Least-commitment and intelligent casual-commitment

In a least-commitment planning strategy, decisions are deferred until forced by constraints. Typically what happens is that conjunctive goals are assumed to be independent and worked separately, producing unordered sets of actions to achieve the goals. From time to time, the planner fires some plan critics that check for interactions among the individual subplans. If conflicting interactions are found, the planner commits to a specific partial ordering that avoids conflicts. There may be cases for which actions stay unordered during the whole planning process, leading to a final partially ordered plan. In this strategy, it is NP-hard [3] to determine if a given literal is true at a particular instant of time while planning, when actions are dependent on the state of the world, as all paths through the partial order must be verified. To avoid this combinatorial explosion, planners that follow this least-commitment strategy use *heuristics* to reduce the search space to determine the truth of a proposition.

A casual-commitment strategy corresponds to searching for a solution by generating and testing alternatives in both the ordering of goals and possible operators to apply. The planner commits to the most promising goal order and operator selection, backtracking to test other orderings and selections, if and only if a failure is reached. Using this approach, there is no problem in determining the truth of a proposition at a certain time, as a *mental* state of the world is kept along the search. However, in the worst case, the method involves an exponential search over the space of solutions. Like the previous approach, NoLimit uses *heuristics* to reduce this exponential search. *Smart* heuristics, in this context, transform a simple casual-commitment strategy into an *intelligent* casual-commitment one, leading to an intelligent exploration of the different alternatives.

In a nutshell, least commitment corresponds to breadth-first search over the space of possible plans, and casual commitment corresponds to best-first heuristic search. The former derives some benefit from structure sharing among alternative plans (the partial order) and the latter benefits from any intelligence that can be applied at decision points - and the direct computation of the world state when necessary.

2.2.2. Completeness

We define the *completeness* of a planner along three different criteria:

1. Ability to reach a solution whenever one *exists*.
2. Ability to find the set of *all* possible solutions.
3. Ability to find the *optimal* solution.

We discuss these criteria in function of the specification of the search algorithms independently of the corresponding computational tractability.

We showed before, through an example, that it is easy to write a domain theory in which there are solvable problems that a linear planner cannot solve, even by searching its entire solution space. The search space of a linear planner is only a subspace of the complete search space and therefore linear planners are not complete according to the first criterion mentioned above.

Other implemented nonlinear planners [8,9,10,12] are complete, in the sense that they are able to explore interactions among conjunctive goals, and hence they reach a solution when there is one. These systems generate one partially ordered plan that corresponds therefore to a set of individually totally ordered executable solutions. However they reason about at most *one* partially ordered plan (backtracking among different partial orders). This prevents these planners from being complete in terms of the two last criteria referred above, as the set of all the solutions to a problem is, in the general case, a *set* of partially ordered plans, not mergeable into a single one. Note, for instance, that different solutions may contain alternative instantiated operators which achieve successfully the same goal. These cannot be merged into the same partial order, as a partial order does not express disjunctions.

NoLimit is complete with respect to all three criteria. Like the other classic nonlinear planners, it reaches a solution whenever there is one. Furthermore, it reasons about a *set* of partially ordered plans (see section 5), having therefore the ability to find the set of all the possible solutions to a problem, including the optimal one.

Control knowledge is usually used to guide and eventually prune the search space to reduce exponential explosion, allowing the implementations to be computationally tractable. However, if completeness is a desired characteristic of a nonlinear planner, special care has to be taken with respect to the heuristics used. To guarantee completeness, in any of the three senses defined, heuristics cannot be used to definitely eliminate portions of the search space unless there are provably no solutions in the discarded subspaces. In general, heuristics should be used to *prefer* some alternatives or choices over others while still keeping the entire domain solution space as a fallback position. On the other hand, if there are other priorities besides completeness (like speed, efficiency, etc), then control knowledge can and should reject parts of the search space. Completeness may be desirable, but can also be very expensive. The ideal situation is to produce a nonlinear planning architecture that is in the default complete - that exploits preferential heuristics when available - and that accepts constraints and definite rule-out heuristics, if the user or application demands their use for computational tractability in large problem spaces.

3. NoLimit - The search algorithm

NoLimit is a nonlinear problem solver that uses means-ends analysis. The representation used to define the domain theory is an extension of the PRODIGY description language [6,11] - a kind of structured first-order-logic with finite-extent metapredicates. For the purpose of this paper, one can just assume that an operator is a simple conjunction of *preconditions* that state the necessary conditions to apply the operator, and a list of *effects* of the operator on the world, stated as predicates to be added and/or deleted. A basic means-ends analysis module tries to apply operators that reduce the differences between the current world and the final desired goal state (a partial description of the world). Basically, in a backward chaining mode, given a goal predicate not true in the current world, the planner selects one operator that adds (in case of a positive goal, or deletes, in case of a negative goal) that goal to the world. We say that this operator is *relevant* to the given goal. If the preconditions of the chosen operator are true, the operator can be *applied*. If this is not the case, i.e., some preconditions are not true in the *state*, then these preconditions become *subgoals*, i.e., new goals to be achieved.

NoLimit proceeds in this apparently simple way too. Its nonlinear character stems from working with a **set** of goals in this cycle, as opposed to the top goal in a goal stack. The skeleton of NoLimit's search algorithm is shown in Figure 6. The algorithm describes the basic cycle of a *mental* planner. *Applying* an operator means *executing* it in the *internal* world of the problem solver, which we refer to, simply by *world* or *state.* A more complete (and real) version of the algorithm can be obtained by adding, to this basic cycle, the several details we discuss along the paper.

1. Check if the goal statement is true in the current state.

   If yes, then show the formulated plan (and take appropriate action, namely stop or continue searching).

   else continue.

2. Compute the *set* of *pending goals* $\mathcal{G}$, and the possible *applicable operator* A.
3. Choose a goal G from $\mathcal{G}$ or select the operator 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 2.
5. If the operator A has been selected as directly applicable, then
   - *apply* A,
   - go to step 1.

Figure 6 - A skeleton of NoLimit's search algorithm.

In step 1 of the algorithm, we check whether the top level goal statement is true in the current state. If this is the case, then we have reached a solution to the problem. We can run NoLimit in a *multiple-solutions* mode that allows us to have the option of searching for more than one solution to a problem. In this mode, NoLimit shows the solution found and continues searching for more solutions, which it groups into *buckets* of solutions. Each *bucket* has different solutions that use the same set of plan steps (instantiated operators). The set of different totally ordered solutions within a bucket form a potential partially ordered solution (see section 5).

We continue to step 2, in case the goal statement is not yet true in the current state. In step 2, we compute the set of pending goals.

A goal is *pending*, iff it is a precondition of a *chosen* operator that is not true in the state.

Let's continue the *subgoaling* branch of the algorithm, by choosing, at step 3, a goal from the set of pending goals. The problem solver *expands* this goal, by getting the set of *instantiated operators* that are relevant to it (step 4). NoLimit now *commits* to a relevant operator. This means that the goal just being expanded is to be achieved by applying this *chosen* operator.

At step 2, we further check for an *applicable* chosen operator.

An operator is **applicable**, iff all its preconditions are true in the state.

The operator considered to be applicable is the last chosen operator not applied yet in the current search path. Note that we can apply several operators in sequence by repeatedly choosing step 5 in case there are applicable operators. The *applying* branch continues by choosing to apply this operator at step 3, and applying it at step 5, i.e., performing the effects of the operator into the state.

A search path is therefore defined by the following regular expression

$$(\textit{goal} \;\; \textit{chosen-op} \;\; \textit{applied-op}^{*})^{*}.$$

It is a sequence of a goal, followed by the selection of a relevant operator to this goal, followed by an eventual sequence of applied operators. As an example, a path might be: *(goal1 chosen-op1 goal2 chosen-op2 applied-op2 applied-op1)*. The path *(goal1 chosen-op1 goal2 chosen-op2 applied-op1 applied-op2)* is not possible as the operator *op1* is to be applied after the operator *op2* according to the commitments made at choice time. To get operator *op1* to be applied before operator *op2* then the corresponding goals *goal1* and *goal2* should be interchanged. As NoLimit reasons with a set of goals, this is easily realized, and is important in obtaining nonlinear behavior by interleaving subgoal trees.

3.1. An example: solving the *ONE-WAY-ROCKET* problem

We now show how NoLimit searches for a solution to the *ONE-WAY-ROCKET* problem introduced earlier (see Figures 3 and 4) following its subgoaling structure in the planning graph. The graph is stored during the search process in an AND/OR *conceptual tree*, for short *ctree*, structured as follows:

- A conceptual AND-node (AND-cnode) is a literal corresponding to an instantiated goal and a conceptual OR-node (OR-cnode) is an instantiated operator.
- The children of an AND-cnode are OR-cnodes and the children of OR-cnodes are AND-cnodes.
- An OR-cnode is an instantiated operator that is relevant to the goal at its parent AND-cnode.
- An AND-cnode is either the internal top-level goal *(done)* or is a precondition of its parent OR-cnode.

For the sake of having a unique root in the conceptual tree, and following the same convention used in the linear problem solver of PRODIGY [6], NoLimit has an internal goal *(done)* that is added to the state by an internal operator **finish** whose preconditions are set initially to the externally provided goal statement. This is an implementation detail, but has the nice effect of transforming a forest of trees each rooted at a different user-given goal conjunct, into a unique tree rooted at the goal *(done)*. Note that the same goal or instantiated operator may be repeated in several places in the conceptual tree, as the subgoaling structure is really a graph.

Let us trace NoLimit in solving the *ONE-WAY-ROCKET* problem, given the conjunctive goal *(and (at obj1 locB) (at obj2 locB))*. Figure 7 (a) shows the corresponding initial ctree. AND-cnodes are represented as rectangles and OR-cnodes as ovals. Shaded nodes correspond to the choices to which the problem solver has already committed.

Table 1 summarizes the search process as successful partial paths. We write operators in upper-case when they are applied. In this table the search steps are directly related to the choices identified in the search algorithm described above. The indentation captures roughly new search cycles and operator applications. Changes in the state are also represented when an operator is applied. To follow the search cycling, the reader should follow the search steps in the sequence presented in the table. The resulting ctree after each step is illustrated in the figure referred in the corresponding column. The "choices left" at each step correspond to the figure referred in the immediately previous step. As an example, consider the third search step in the table, i.e., where the choice *(at obj1 locB)* is taken. The immediately previous ctree is the one in Figure 7 (a), where we can see that there is one choice left, namely the goal *(at obj2 locB)*. Figure 7 (b) illustrates the choices made at the third and fourth step in the table.

Search steps	*Choices left*	*Figure*	*State*
(done)		7 (a)	(at obj1 locA) (at obj2 locA) (at rocket locA)
finish			
(at obj1 locB)	(at obj2 locB)	7 (b)	
(unload-rocket obj1 locB)			
(inside obj1 rocket)	(at rocket locB) (at obj2 locB)	7 (c)	
(load-rocket obj1 locA)	(load-rocket obj1 locB)		
(LOAD-ROCKET obj1 locA)	(at rocket locB)		(inside obj1 rocket) (at obj2 locA) (at rocket locA)
(at obj2 locB)	(at rocket locB)	7 (d)	
(unload-rocket obj2 locB)			
(inside obj2 rocket)	(at rocket locB)	7 (e)	
(load-rocket obj2 locA)	(load-rocket obj2 locB)		
(LOAD-ROCKET obj2 locA)	(at rocket locB)		(inside obj1 rocket) (inside obj2 rocket) (at rocket locA)
(at rocket locB)		7 (f)	
(move-rocket)			
(MOVE-ROCKET)			(inside obj1 rocket) (inside obj2 rocket) (at rocket locB)
(UNLOAD-ROCKET obj2 locB)			(inside obj1 rocket) (at obj2 locB) (at rocket locB)
(UNLOAD-ROCKET obj1 locB)			(at obj1 locB) (at obj2 locB) (at rocket locB)
FINISH			(done) (at obj1 locB) (at obj2 locB) (at rocket locB)

Table 1 - Tracing NoLimit solving the *ONE-WAY-ROCKET* problem.

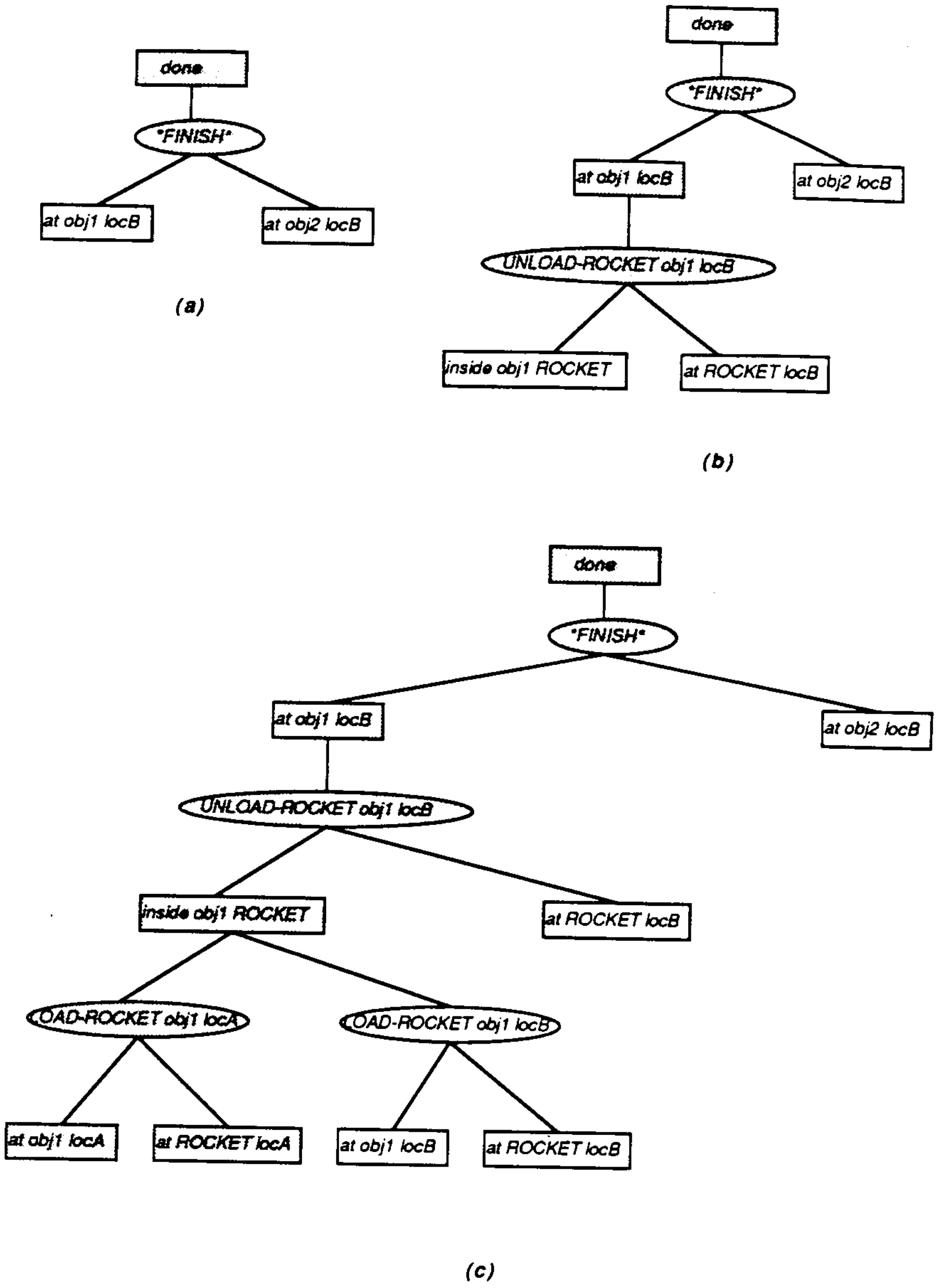
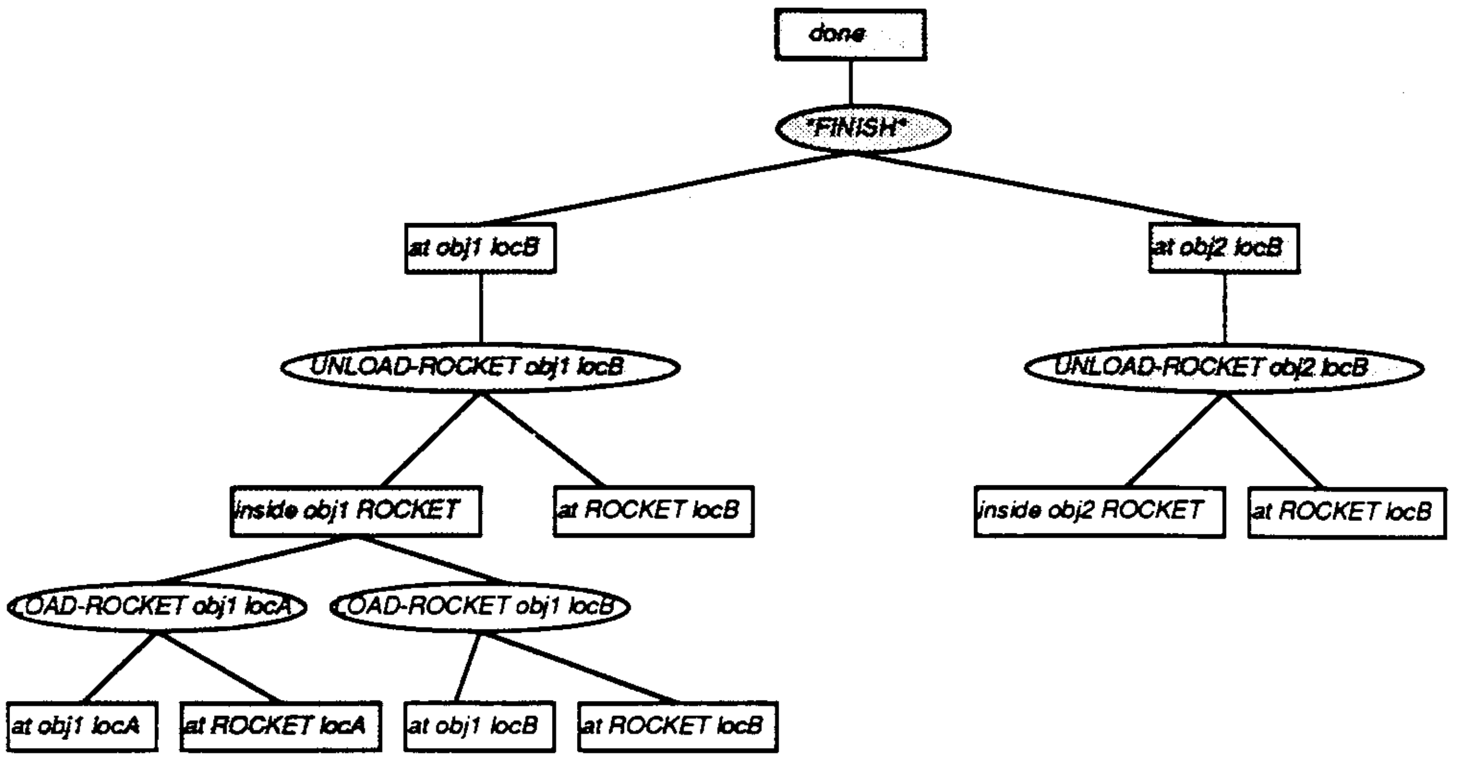
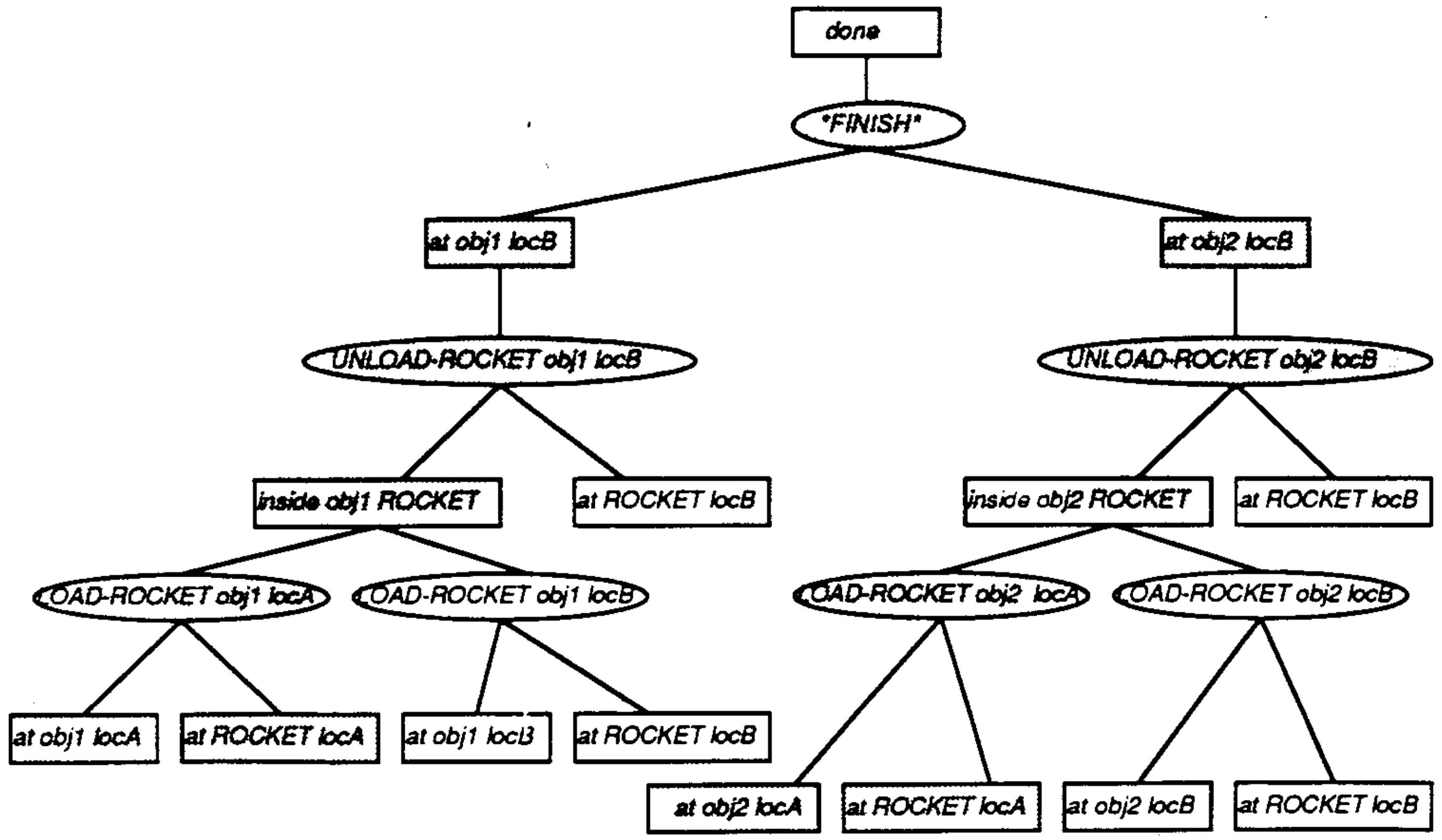


Figure 7 - Conceptual trees generated by NoLimit in successive planning steps corresponding to Table 1 for the *ONE-WAY-ROCKET* problem.



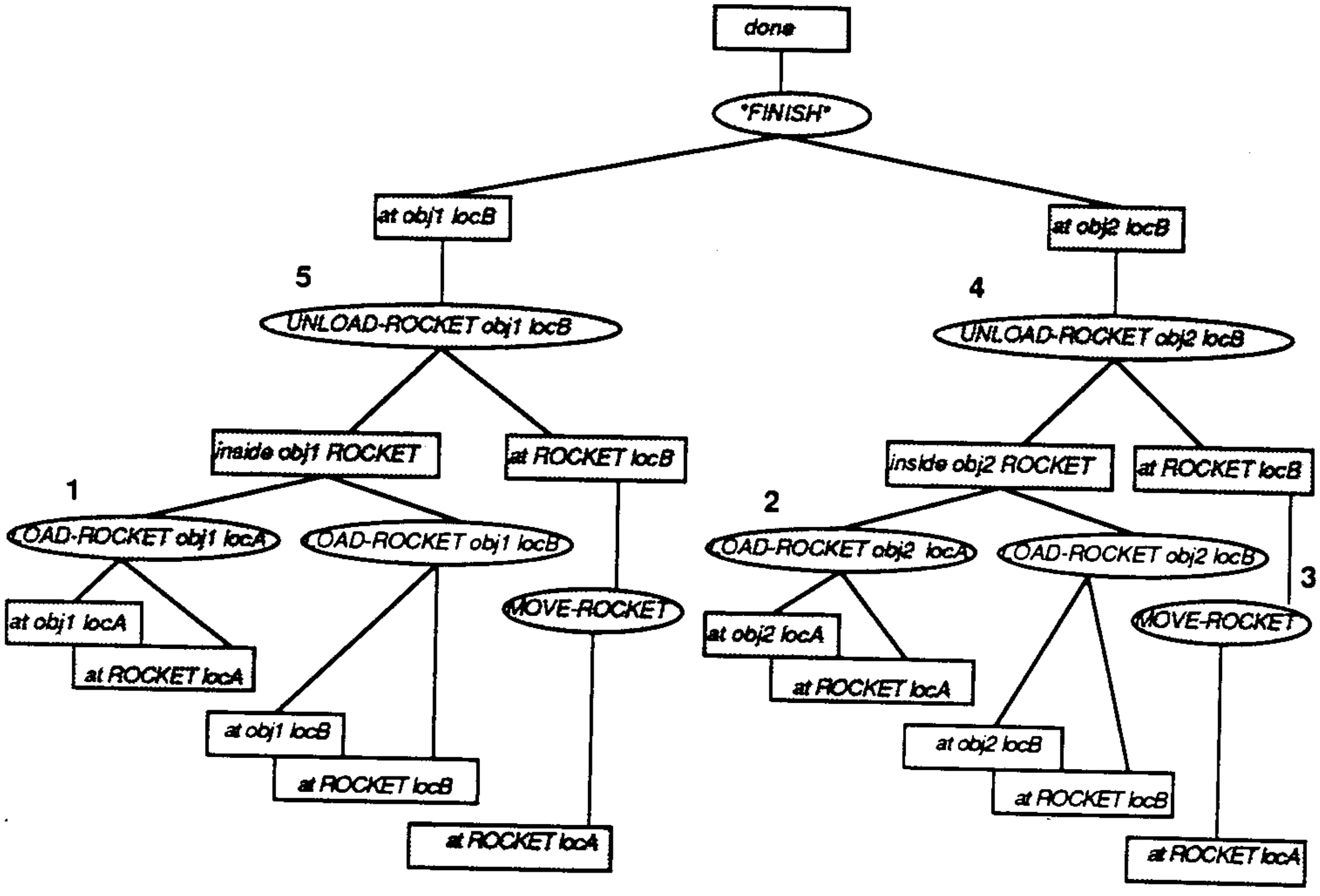
(d)



(e)

Figure 7 - (continued)

Figure 7 (e) shows the complete conceptual tree for the successful path traced in Table 1. The numbers at the operator nodes refer to the final plan steps in chronological order. NoLimit therefore outputs the plan: (LOAD-ROCKET obj1 locA), (LOAD-ROCKET obj2 locA), (MOVE-ROCKET), (UNLOAD-ROCKET obj2 locB), (UNLOAD-ROCKET obj1 locB). NoLimit solves this problem, where linear planners fail, because it switches attention to the conjunctive goal *(at obj2 locB)* before completing the first conjunct *(at obj1 locB)*. This is shown in Figure 7 (e) by noting that, after the plan step 1 where the operator (LOAD-ROCKET obj1 locA) is applied as relevant to a subgoal of the top-level goal *(at obj1 locB)*, NoLimit changes its focus of attention to the other top-level goal and applies, at plan step 2, the operator (LOAD-ROCKET obj2 locA) which is relevant to a subgoal of the goal *(at obj2 locB)*. Therefore the complete subplans for the each of those goals are interleaved and cannot be organized in strict linear sequence.



(f)

Figure 7 - (continued) The complete conceptual tree for a successful solution path.
The numbers at the nodes show the execution order of the plan steps.
Shaded nodes correspond to the choices to which the problem solver committed.

NoLimit can find all the possible solutions to planning problems. In this case the other three possible plans are obtained quite easily by NoLimit, without necessitating three more complete searches. Instead it tries successfully to vary the solution found, considering the alternatives left unexplored. By using the algorithm described in section 5 to convert a set of totally ordered plans into a set of partially ordered ones that cover it, NoLimit is also able to *propose* different solutions that are then simply tested. In this particular case, NoLimit returns the partial order represented in Figure 8.

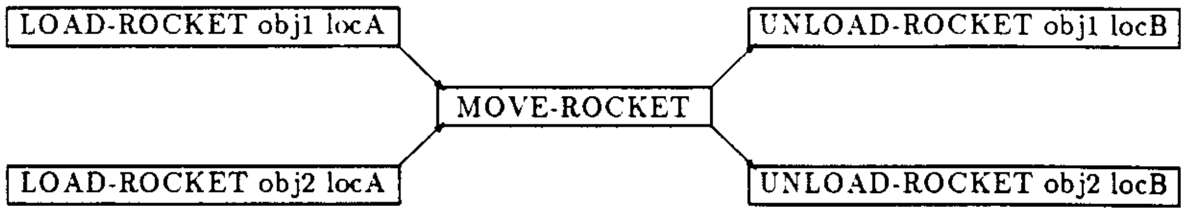


Figure 8 - Partially ordered plan for the *ONE-WAY-ROCKET* problem, corresponding to the four totally ordered plans consistent with the partial order.

3.2. Failing and backtracking

The only true cause for failure is reaching a subgoal that is unachievable for lack of any relevant operators, in which case the path fails and is abandoned.

NoLimit has several other heuristics that propose *suspending* a path, under various conditions when a path becomes unpromising. The two main ones follow:

- Goal loop - If a subgoal is generated that is still pending earlier in the path, then a goal loop is recognized.
- State loop - If applying an operator generates a state in the world that was previously visited, then a state loop is recognized. Note that suspending processing on state-loop detection is truly a heuristic - progress could still be made working for instance, on other goals first. A complete planner cannot totally abandon search in this situation.

Upon failure, NoLimit backtracks chronologically to the previous choice points. However it has the ability to call *backtracking* control rules (see section below) that accept (or reject) a particular backtracking point as a good (or bad) one, thus performing more intelligent allocation of resources and permitting dependency-directed backtracking or other disciplines that override the chronological backtracking default. When a backtracking choice point is found, the next choice left is considered and the search proceeds through another path exploring a new alternative.

4. Control knowledge

The search algorithm described in the previous section involves several choice points, to wit:

- What *operator* to choose to achieve a particular goal?
- What *bindings* to choose in order to instantiate the chosen operator?
- What *goal* to select from the set of pending goals and subgoals?
- *Apply* an applicable operator or continue *subgoaling*?
- Should the search path being explored be *suspended*?
- Upon failure, what alternative *instantiated operator* to choose to achieve a pending goal?
- Upon failure, what alternative *goal* to subgoal on?
- Upon failure, *apply* an applicable goal or *subgoal* on a goal left to work on?
- Upon exhaustion of alternatives, what *suspended path* to consider for further search?

Decisions at all these choices are taken based on user-given or learned *control rules* to guide the casual commitment search. Control rules can *select*, *reject*, *prefer*, or *decide* alternatives [11]. They guide the search process and help to reduce the exponential explosion in the size of the search space. All these choice points have been largely recognized as the crucial decisions in the problem solving paradigms. Previous work in the linear planner of PRODIGY used explanation based learning techniques [5] to extract from a problem solving trace the explanation chain responsible for a success or failure and compile search control rules therefrom. We are now developing a case-based approach that consists in storing individual problems solved in the past to guide the several decision choice points [1,2] when solving similar new problems. The machine learning and knowledge acquisition work supports the NoLimit casual-commitment method, as it assumes there is *intelligent* control knowledge, exterior to its search cycle, that it can rely upon to take decisions.

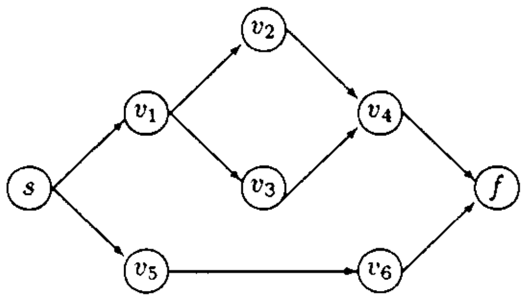
For a complete description of the syntax and functionality of control rules, the reader is referred to [11]. For illustrative purposes, here we show a simple example of a backtracking control rule using a pseudo-code language. Failure occurs, when the problem solver encounters a goal that cannot be achieved, i.e., a goal that does not have any relevant operators. It then makes sense to backtrack to a node where the operator that has this goal as a precondition was chosen and select an alternative operator (rather than exploring another path requiring the same operator, and guaranteed to fail for the same reason).

```
(CONTROL-RULE
   (preconditions (and (is-failure-reason 'no-relevant-operators)
                       (was-working-for-goal <goal>)
                       (is-precondition-at <the-current-node> <goal>)))
   (effects (select for-backtracking <the-current-node>)))
```

The system has the ability to consider all the coexisting control rules and take action as soon as some control knowledge applies. The control rule shown above is domain independent. For examples on domain dependent control rules see [5,6,11].

5. Partial and total orders

A partially ordered graph is convenient 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 steps (instantiated operators) of the plan, and E is the set of *edges* (temporal constraints) in the partial order. Let $V = \{v_0, v_2, \ldots, v_{n-1}\}$. We represent the graph as a square array P, where $P[i,j] = 1$, if there is an edge from vertex v_i to vertex v_j. There is an edge from v_i to v_j, if the operator corresponding to v_i must be executed **before** the one corresponding to v_j, referred from now on as simply v_j. We say that v_i **precedes** v_j and denote it as $v_i \prec v_j$. The inverse of this statement does not necessarily hold, i.e. there may be the case that $v_i \prec v_j$ and there is not an edge from v_i to v_j. This is due to the fact that the relation $\prec$ is the *transitive closure* of the relation represented in the partial order. We show a simple example of a partial order in Figure 9. Without loss of generality consider operators v_0 and v_{n-1} of any plan to be the *fictitious* operators named *start* and *finish*, represented in Figure 9 as s and f.



$$\begin{tikzcd}
 & & v_2 \arrow[dr] & & \\
 & v_1 \arrow[ur] \arrow[dr] & & v_4 \arrow[dr] & \\
s \arrow[ur] \arrow[drr] & & v_3 \arrow[ur] & & f \\
 & & v_5 \arrow[r] & v_6 \arrow[ur] &
\end{tikzcd}$$

Figure 9 - An example of a partially ordered plan.

Legal orderings are, for example, $(s, v_1, v_2, v_3, v_4, v_5, v_6, f)$, or $(s, v_1, v_5, v_2, v_6, v_3, v_4, f)$, or - $(s, v_1, v_5, v_3, v_2, v_6, v_4, f)$. The ordering $(s, v_5, v_6, v_3, v_2, v_4, v_1, f)$ is not legal as v_1 has to come before v_2, v_3, and v_4. We can enumerate all the total orders that can be generated from a partial order. We say that a set of partial orders $\mathcal{P}$ **covers** a set of total orders $\mathcal{T}$ if every element of $\mathcal{T}$ can be generated from at least one element of $\mathcal{P}$. If the complete set of total orders generated from $\mathcal{P}$ is equal to $\mathcal{T}$, then we say that $\mathcal{P}$ **covers exactly** $\mathcal{T}$.

5.1. Converting total orders into a set of partial orders

NoLimit internally generates totally ordered plans. It can generate as many solutions as we want until the search space is exhausted. The complete set of solutions is a **set** of partially ordered solutions. Note that in general a set of total orders **cannot** be represented by a single partial order (as assumed in least-commitment planning) but rather by a (smaller) set of partial orders. Further note that a total order is itself a *degenerate* partial order. We are interested however in generating a *better* set of partial orders than the enumeration of all the total orders. We now present the algorithm that NoLimit uses to create the set of partial orders that covers a set of total orders. As an initial case consider that the set of total orders given is covered exactly by **one** partial order. Let $\mathcal{T}$ be this set of total orders, $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$. Each total order T_i is a plan, i.e., a sequence of n instantiated operators from a set $V = \{s, v_1, \ldots, v_{n-2}, f\}$. Then the algorithm described below in Figure 10 builds the partial order by first creating a graph with the transitive edges also present, and then removing all edges that can be generated by the transitive closure of the remaining ones.

1. Create a square array $\mathcal{P}$ of size n, where $\mathcal{P}[i,j] = 1$, iff $\forall \alpha \in \{1,\ldots,k\}$ $v_i \prec v_j$ *in* T_α.
2. Remove transitive edges in $\mathcal{P}$ in the following way: set $\mathcal{P}[i,j] = 0$, iff $\mathcal{P}[i,j] = 1$, and *there is a path to go from* v_i *to* v_j *that does not use the edge* (i,j).

Figure 10 - Algorithm to generate one partial order covering a set of total orders.

In Figure 11 we illustrate this algorithm with a simple example.

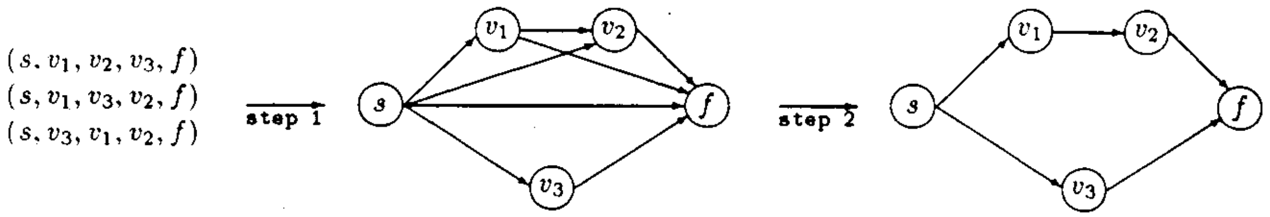


Figure 11 - Converting a complete set of total orders into one partial order.

Consider now the case where we are given a complete set of total orders that is covered by a set of partial orders. In this case the algorithm, as shown in Figure 10, *overgeneralizes* and returns one partial order that covers a *super* set of the given set of total orders. We rewrite the algorithm to be able to generate a set of partial orders that covers exactly a given set of total orders (see Figure 12). Let again $\mathcal{T}$ be the set of total orders, $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$. Each total order T_i is some sequence of n operators from a set $V = \{s, v_1, \ldots, v_{n-2}, f\}$. The set of partial orders returned is $\overline{\mathcal{P}} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_p\}$.

1. Create a square array $\mathcal{P}$ of size n, where $\mathcal{P}[i,j] = 1$, iff $\forall \alpha \in \{1,\ldots,k\}$ $v_i \prec v_j$ *in* T_α.
2. Remove transitive edges in $\mathcal{P}$ in the following way: set $\mathcal{P}[i,j] = 0$, iff $\mathcal{P}[i,j] = 1$, and *there is a path to go from* v_i *to* v_j *that does not use the edge* (i,j).
3. Set $\overline{\mathcal{P}} = \{\mathcal{P}\}$.
4. Calculate the set $\mathcal{T}'$ of the total orders generated by $\overline{\mathcal{P}}$. Compare $\mathcal{T}'$ with $\mathcal{T}$.
   - If $\mathcal{T}' = \mathcal{T}$ then return $\overline{\mathcal{P}}$.
   - If $\mathcal{T} \subset \mathcal{T}'$, go to step 5. Assume $T_i \in (\mathcal{T}' - \mathcal{T})$.
   - If $\mathcal{T} \supset \mathcal{T}'$, then go to step 6. Let $\mathcal{T}'' = \mathcal{T} - \mathcal{T}'$.
5. Let $\mathcal{P}_j$ cover the total order T_i. Break $\mathcal{P}_j$ into the union of two partial orders $\mathcal{P}_1, \mathcal{P}_2$, such that T_i is not covered by $\mathcal{P}_1 \cup \mathcal{P}_2$. Set $\overline{\mathcal{P}} = (\overline{\mathcal{P}} - \mathcal{P}_j) \cup \mathcal{P}_1 \cup \mathcal{P}_2$. Go to step 4.
6. Calculate $\overline{\mathcal{P}}'$ that covers the set $\mathcal{T}''$ by running the algorithm with input set $\mathcal{T}''$. Return $\overline{\mathcal{P}} = \overline{\mathcal{P}} \cup \overline{\mathcal{P}}'$.

Figure 12 - Algorithm to generate a set of partial orders covering a set of total orders.

In Figure 13 we illustrate the new version of the algorithm with an example.

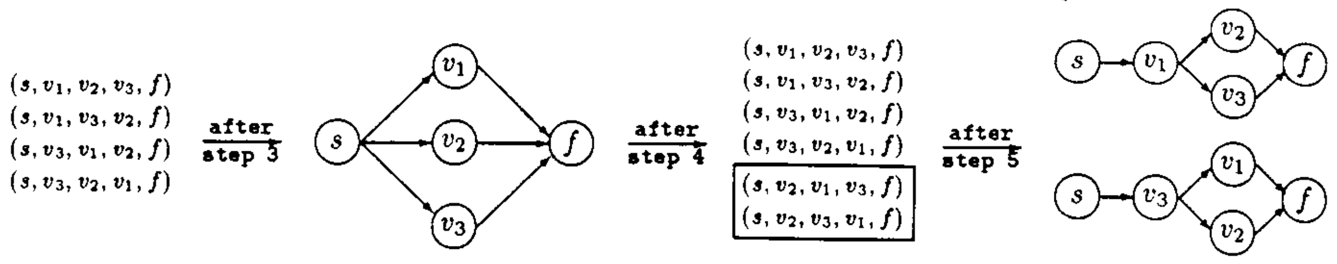


Figure 13 - Converting a set of total orders into a set of partial orders.

The method used in step 5 of breaking a partial order into a set of partial orders does not guarantee that the final set of partial orders is the **minimal** set of partial orders that covers the given set of total orders. This is acceptable within the context of our work. If t and n are the cardinalities of the sets T and V respectively, then the algorithm runs in the worst case in $O(tn^2)$.

5.2. Generating total orders from a set of partial orders

Step 4 of the algorithm presented in Figure 12 requires calculating the set of total orders generated from a set of partial orders. The algorithm to achieve this, works by committing to a particular branch, and recursively considering the partial order that results from this commitment.

As an illustration, consider the partial order of Figure 14. The algorithm commits to both possible branches out of the vertex s. Committing to v_1 leads to (s, v_1) followed by the total orders of the partial order in Figure 15 (a). Committing to v_3 leads to (s, v_3) followed by the total orders of the partial order in Figure 15 (b). This process goes on recursively until exhaustion of the alternative branches. The final set of total orders in this case is the expected (s, v_1, v_2, v_3, f), (s, v_1, v_3, v_2, f), and (s, v_3, v_1, v_2, f).

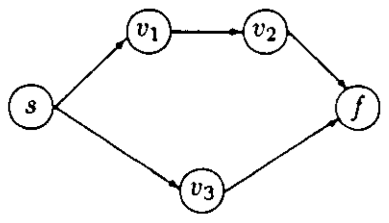
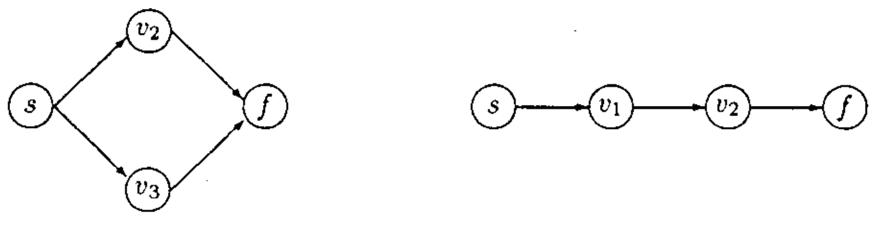


Figure 14 - An example of a partial order from which we generate the corresponding total orders.



(a) (b)

Figure 15 - (a) Removing the edge v_1, (b) Removing the edge v_3.

The algorithm described above in Figure 12 is very useful due to its inherent *overgeneralization*. The process of creating a totally ordered solution has an associated cost due to the search process, although the cost may be less than complete replanning if the full search tree is kept to explore untried alternatives. With this algorithm, from a small set of totally ordered solutions, we can create a set of partial orders that covers more than the total orders given. Then NoLimit simply *tests* the new total orders implicit in the partial order instead of searching for them. The search effort is then balanced through the generation of a super-covering partial order and subsequent testing of individual solutions generated by this partial order.

6. Conclusion

NoLimit is a completely implemented nonlinear planner that uses an intelligent casual-commitment strategy to guide its search process. The casual-commitment method used to achieve its nonlinear character is in marked contrast to the least-commitment strategy used in other nonlinear planners. NoLimit has the ability to call user-given or automatically learned control rules in all its choice points. Work in progress includes: combining hierarchical planning with NoLimit's search mechanism, and making NoLimit able to replay previous traces of solutions.

The system has additional features not reported here, such as a sophisticated TMS that enables deduction and control of beliefs, and a type hierarchy to organize the objects of the world. The reader is referred to [11] for a complete description of NoLimit's implementation and additional features.

7. Acknowledgments

The author would like to thank Jaime Carbonell for motivation and continuous helpful discussions on problem solving in general, and nonlinear planning in particular that culminated in this work. The author is also in debt to Steve Minton and Craig Knoblock in several ways, namely, for early discussions on the fundamental characteristics of linear and nonlinear problem solving, and for detailed descriptions of the implementation of the linear planner of the PRODIGY architecture. Daniel Sleator and Ravi Kannan provided important suggestions on how to address the equivalence of partial and total orders. In particular, the first algorithm described in section 5 resulted from a fruitful discussion with Daniel Sleator. My special thanks to Daniel Borrajo for a major part of the underlying implementation. Without him, it would have been very difficult to include many powerful features in NoLimit. Craig Knoblock, Robert Frederking, Alicia Perez, Yolanda Gil, and Daniel Borrajo provided numerous comments on early versions of this paper. Jaime Carbonell assisted with the final revision. Finally the author thanks the whole PRODIGY research group for constantly providing helpful suggestions during the implementation process.

References

[1] J. G. Carbonell. Derivational analogy: a theory of reconstructive problem solving and expertise acquisition. In *Machine Learning, An Artificial Intelligence Approach, Volume II*, Morgan Kaufman, 1986.

[2] J. G. Carbonell and M. M. Veloso. Integrating derivational analogy into a general problem solving architecture. In *Proceedings of the First Workshop on Case-Based Reasoning*, Morgan Kaufmann, May 1988.

[3] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[4] R. E. Fikes and N. J. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[5] S. Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, Computer Science Department, Carnegie Mellon University, 1988.

[6] S. Minton, C. A. Knoblock, D. R. Kuokka, Y. Gil, and J. G. Carbonell. *PRODIGY 2.0: The Manual and Tutorial.* Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.

[7] E. Rich. *Artificial Intelligence.* McGraw-Hill Inc., 1983.

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

[9] G. J. Sussman. *A Computational Model of Skill Acquisition.* Technical Report AI-TR-297, Artificial Intelligence Laboratory, MIT, 1973.

[10] A. Tate. Generating project networks. In *Proceedings of IJCAI-77*, pages 888–893, 1977.

[11] M. Veloso, D. Borrajo, and A. Perez. *NoLimit - The nonlinear problem solver for Prodigy: User's and programmer's manual.* Technical Report, School of Computer Science, Carnegie Mellon University, 1990.

[12] D. E. Wilkins. *Can AI Planners Solve Practical Problems?* Technical Note 468R, SRI International, 1989.