BACK TO BACKTRACKING: CONTROLLING THE ATMS

Johan de Kleer

Intelligent Systems Laboratory
XEROX Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

and

Brian C. Williams

M.I.T. Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts, 02139

ABSTRACT

The ATMS (Assumption-Based Truth Maintenance System) provides a very general facility for all types of default reasoning. One of the principal advantages of the ATMS is that all of the possible (usually mutually inconsistent) solutions or partial solutions are directly available to the problem solver. By exploiting this capability of the ATMS, the problem solver can efficiently work on all solutions simultaneously and avoid the computational expense of backtracking. However, for some applications this ATMS capability is more of a hindrance than a help and some form of backtracking is necessary. This paper first outlines some of the reasons why backtracking is still necessary, and presents a powerful backtracking algorithm which we have implemented which backtracks more efficiently than other approaches.

1. Introduction

The complexity of a problem-solving task is a function of both the number of rules executed and number of contexts considered in the search. Many techniques have been developed to minimize this complexity for various types of tasks. In this paper we show how two such approaches, the ATMS and conventional dependency-directed backtracking (DDB), can be combined to produce a control strategy more efficient than either.

An ATMS-controlled problem-solver tends to be more efficient for problems where all the solutions are needed. It achieves this efficiency by organizing the search to find the most general inferences first. Thus, the number of rules executed and contexts examined are significantly reduced. A DDB-controlled problem-solver tends to be more efficient for problems where only one of a number of the possible solutions are required. It achieves this efficiency by organizing the search to find a single specific solution first. By combining them we get the advantages of both.

2. The ATMS

A TMS-based problem solver consists of two components: an inference engine and the TMS. The inference engine deduces new data from old (usually by the application of rules, or *consumers* in ATMS terminology). Associated with each consumer is a set of data, referred to as the consumer's *antecedents*. A consumer is invoked on its antecedents when all of them are believed in the current context. Every inference resulting from a consumer invocation is recorded as a *justification* using the TMS and must include the antecedents of the consumer. The TMS's task is to determine what data is believed in each context given the justifications produced thus far in the problem-solving effort.

Justification-based TMS's force the problem solver to focus on a single consistent data base at a time. This has many disadvantages (see [4,7]). The ATMS permits the problem solver to operate simultaneously in several mutually inconsistent contexts. To achieve this, the ATMS augments the conventional TMS data structures in a several of ways. The ATMS introduces the notion of a primitive *assumption*. Unlike other data, which are believed only if belief in them can be justified, assumptions are believed unless there is evidence to the contrary. By tracing backwards through supporting justifications for a datum the ATMS identifies the set(s) of assumptions upon which the datum ultimately depends. Such a set of assumptions is

called an *environment*. Typically a datum can be derived in a variety of ways and so a datum can follow from a variety of environments. The complete set of environments from a which a datum can be derived is called its *label*. The set of data derivable from an environment is called its *context*. Any environment from which false ($\perp$) can be derived is inconsistent and is called a *nogood*. The ATMS data base achieves its efficiency by (a) optimizing the representation of nogoods, (b) exploiting the fact that if a datum follows from an environment it follows in all the environment's supersets and thus it is only necessary to explicitly store the minimal environments of a label, and (c) recognizing that all ATMS operations reduce to set operations for which good algorithms are available.

3. Three Reasons for Backtracking

ATMS-based problem solvers suffer from three kinds of difficulties related to backtracking: (a) the task may require that only a small fraction of the search space be explored, (b) even for problems where all solutions are desired, they often search more than necessary, and (c) they are inherently more difficult to debug.

First, conventional backtracking searches depth-first, producing solutions one at a time. Without exercising external control, the ATMS-based problem solver identifies all solutions at once. If the overall goal of the problem solver is to find only one solution which satisfies the goal, then the ATMS-based problem solver can be hopelessly inefficient.

Consider an example (adapted from [6] and [10]) where the task is to construct an n-bit number with odd parity (i.e., a string of bits with an odd number of ones). For each bit position two variables are created. Let b_i indicate the ith bit's value, and p_i the parity for all the bits up to and including b_i. Each b_i can be zero or one ($\Gamma_{b_i \equiv 0}$ represents the assumption that b_i is 0). Parity is defined recursively as ($\Rightarrow$ indicates a justification):

$$p_{-1} = 0$$

$$p_{i-1} = 0, \Gamma_{b_i \equiv 1} \Rightarrow p_i = 1$$

$$p_{i-1} = 1, \Gamma_{b_i \equiv 0} \Rightarrow p_i = 1$$

$$p_{i-1} = 0, \Gamma_{b_i \equiv 0} \Rightarrow p_i = 0$$

$$p_{i-1} = 1, \Gamma_{b_i \equiv 1} \Rightarrow p_i = 0.$$

The ATMS, which finds all solutions, discovers all 2^i bit strings having odd (and even) parity. If the goal was to find only one odd-parity bit string, then the simple ATMS-based problem solver has done 2^i more work than necessary.

Second, even if the goal of the problem solving is to identify all solutions, the ATMS-based problem solver

may still search more than necessary. The problem solver may examine regions of the search space which are never reached by dependency-directed backtracking.

Consider the familiar n-queens problem. (For simplicity assume that there are 3 queens and that queen Q_i is placed on row i.) A set of rules is constructed which checks that no two queens are placed in attacking positions. The problem is then to find all placements of the three queens which are consistent with these rules.

The ATMS-based problem solver would try the placement of all singletons, pairs and then triples of queens in search of a consistent solution. For example, it would check whether placing Q_3 in column 3 was consistent with placing Q_2 in column 2 and find that they lay on the same diagonal. Within a backtracking problem solver one would first attempt to place Q_1, then Q_2, and only then Q_3. As there is no way of placing a queen in row 1 and Q_2 in column 2, the above situation (Q_3 in column 3 and Q_2 in column 2) never arises.
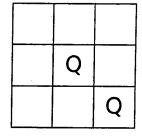

Fig. 1 : Unreachable situation for DDB.

Third, debugging an ATMS-based problem solver can be difficult. Most of the effort of constructing a problem solver involves debugging the knowledge base and inference rules. In our experience, by far the most common error of ATMS users is failing to specify all the ways a context can be inconsistent. Thus, during debugging far more solutions are found than expected. Commonly, so many solutions exist that problem solving cannot terminate within a reasonable time. With the ATMS, all solutions are explored at once. If the solver is halted prematurely, no solution is found, leaving the user with little information about what knowledge he failed to incorporate into his knowledge base or inference engine. A common and related problem is that the user has framed a problem such that one of the solutions contains an infinite amount of data. If so the problem solver will never terminate.

These debugging difficulties pose far less of a problem for a solver based on conventional backtracking where the problem solver finds one solution at a time. When the first solution appears inadequate, the implementor immediately notices the missing knowledge. In addition, when a particular solution seems to take excess computational resources, the computation can be interrupted and the current state examined. This is difficult in an ATMS-based

problem solver because the intermediate states always represent pieces of many solutions, any of which could be causing a problem.

Dependency-directed backtracking is typically applied to problems satisfying the following constraints: 1) given a finite set of choices, each solution must select exactly one choice from each set, 2) a solution is confirmed by checking for inconsistencies, 3) testing for inconsistencies must take a finite amount of time, and 4) the solution does not depend on the order in which choices are made.

In the next two sections we discuss ATMS-guided and DDB-guided problem solving, respectively, for this class of problems. In Section 6 we compare the approaches, and then propose a hybrid approach combining the features of both (without the limitations) in Section 7.

4. ATMS-guided Problem Solving

For most tasks, the majority of the problem solving resources are involved with executing the consumers.[1] Therefore, a central goal is to minimize the number of consumers that must be executed to solve a particular problem.

The ATMS is typically used for tasks where multiple solutions are required. For such tasks, the best approach is to ensure that no consumer is ever executed unnecessarily. There are two types of situations where a consumer could be uselessly invoked. First, a consumer can be executed in an environment which is later discovered to be contradictory, making the consumer's execution useless (unless it applied to another environment as well). Second, a consumer can deduce some datum in some environment, but some other consumer may later deduce the same datum in a more general environment; therefore, the first consumer's execution was superfluous. To avoid such inefficiencies the consumers are scheduled such that consumers in smaller environments are executed first and within each set of consumers for a particular environment, consumers directed towards detecting inconsistencies are executed first.

Within this framework the consumer scheduler repeatedly picks the smallest consistent environment with non-executed consumers and runs one of its consumers until all consumers of all consistent environments are executed. For example, suppose the problem solver must search through a space of possibilities where it must pick one from each of the sets: $\{A, B\}$, $\{\alpha, \beta\}$, and $\{1, 2\}$. Fig 2 illustrates the lattice of possibly consistent environments. The consumer scheduler first executes any pending consumers of the first row ($\{A\}$, $\{B\}$, $\{\alpha\}$, $\{\beta\}$, $\{1\}$, and $\{2\}$), then

the second row ($\{A, \alpha\}$, $\{A, \beta\}$,...), and then the third row ($\{A, \alpha, 1\}$,...).

If the ATMS finds an inconsistent environment, then the problem solver stops exploring that environment and any superset of it. Thus the ATMS may explore some of the consequences of inconsistent environments with a minimal set of assumptions; however, it will not explore an inconsistent environment which is not minimal (i.e., has a subset environment which is also inconsistent). The set of minimally inconsistent environments can be pictured as a line, dividing the the search space into two parts; environments above the line are consistent and those below are inconsistent. If we associate each consumer with the smallest environment it runs in, then 1) all consumers associated with environments above the line will be executed, 2) some consumers associated with minimally inconsistent environments will be run (possibly all), and 3) any consumer associated with environments below the line are guaranteed *not* to be run. Returning to the example, assume that among all the rules three inconsistencies are ultimately detected ($\mapsto$ indicates a consumer):

$$A, \beta \mapsto \perp,$$

$$B \mapsto \perp,$$

$$\alpha, 1 \mapsto \perp.$$

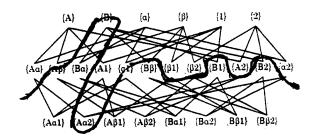The resulting lattice is shown in Fig. 2:

Fig. 2 : ATMS-controlled search.

5. DDB-guided Problem Solving

As illustrated in Section 3, for tasks requiring only a small number of the set of possible solutions the simple ATMS-controlled approach can be extremely inefficient. Instead, some form of backtracking control could be exploited. Even in those cases where all solutions are being explored, the ATMS may have to explore portions of the search space which DDB would avoid.

---

[1] An ATMS-based problem solver only examines contexts which result in the execution of consumers. Thus the number of consumers executed must be at least as large as the number of contexts examined. As a result we only need to evaluate the system's performance with respect to rule invocations, not contexts examined.

There is a wide variety of backtracking techniques. For the sake of discussion we shall use one of the best: the general dependency-directed backtracking embodied in a conventional justification-based TMS (such as [9]). The crucial characteristic of this backtracker upon which our argument depends is that the backtracker can test a particular context for any inconsistencies previously encountered. Consumers are executed only after a context has been determined not to contain any known inconsistency.
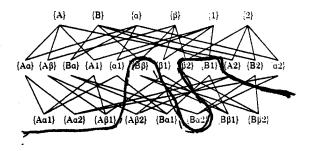

Fig. 3 : DDB-controlled search.

Backtracking exploits the fact that any solution must pick exactly one element from each set of choices. In our example, any solution must contain one assumption from each of $\{A, B\}$, $\{\alpha, \beta\}$, and $\{1, 2\}$. Backtracking enumerates the space through depth-first search, selecting one assumption from each set of choices in the order given and executing the consumers only when a terminal environment is reached (i.e., environments which contain exactly one assumption from every set of choices). When this occurs the consumers of that environment, as well as those in any subset of that environment, are invoked until a contradiction occurs or no inconsistency remains. No ordering is placed on the order of consumer invocation. The TMS ensures that no consumer will ever be executed in a terminal environment which contains a previously discovered inconsistency. For example, if some consumer determines that $\{\alpha, 1\}$ is inconsistent while analyzing the terminal environment $\{A, \alpha, 1\}$, no consumer will ever be executed in the terminal environment $\{B, \alpha, 1\}$. (Note that here as in Section 5 we presume that every inference performed is recorded as a justification and thus no consumer need ever be executed twice.)

Given the same consumers as Section 4., DDB explores the search space as shown in Fig. 3. For presentation we associate consumers with their minimal environments although without the ATMS these cannot be explicitly computed. The DDB-controlled search runs consumers only in terminal environments which is equivalent to running all consumers in all subsets of the terminal environment. Fig. 3 illustrates the DDB-guided search. Rules

have been run in every environment above this line[2].

6. A Comparison of the Approaches

Each of the approaches explored above has its own advantages and disadvantages. These are described below.

The primary advantage of the ATMS is that it is guaranteed not to explore any inconsistent environment which is not minimally inconsistent. To accomplish this the ATMS organizes the search to find the most general inferences first. The ATMS has the additional feature that it only examines environments which contain at least one pending consumer. Thus for those problems with a sparse number of pending consumers and a very large set of environments the ATMS will have a significant performance advantage over those approaches where every environment is examined.

The primary disadvantage of the ATMS is that it essentially works in a breadth-first manner, working on all solutions simultaneously. As pointed out in Section 3, for those problems where only a few of a large number of solutions is desired the ATMS can have significant drawbacks. In addition, the ATMS' unfocused behavior makes it difficult to debug.

These, however, are exactly the advantages of dependency-directed backtracking. DDB focuses on one solution at a time, making its inferences easy to follow (by the implementor). In those cases where one or a few solutions are desired, DDB does not waste effort exploring additional solutions never used. The primary disadvantage of DDB is that, unlike the ATMS, it can explore inconsistent environments which are not minimal. This happens because, given a terminal environment, there is no ordering placed on the subset environments being explored. Thus an environment may be explored before its subset is shown to be inconsistent. If these environments contain a number of consumers using enormous computational resources then the consumers are invoked needlessly and the performance of DDB will be clearly inferior. For example, $\{B, \alpha\}$ is an example of an inconsistent environment which was uselessly explored by DDB in the example of section 5 but not explored by the ATMS in section 4.

DDB has another advantage over the ATMS approach. Even when looking for all solutions there are environments explored by the ATMS, which are not explored using DDB. The reason for this is subtle, and depends on two key observations: First, each solution must select one assumption from *every set of choices*, any incomplete set of choices is not a solution. Second, the order in which these choices are made is irrelevant. For any problem with more than two sets of choices, there are several orders in which the

---

[2] As with the ATMS-guided search, depending on the order in which the rules are executed, few or many of the rules of an inconsistent environment may be run.

choices could be made. Each sequence of choices corresponds to a path moving from the root of the environment lattice to one of the solutions. DDB makes the observation that ordering is irrelevant and thus explores only one path to a particular solution, (based on the ordering in which choices are supplied). The ATMS, on the other hand, explores all paths to the same solution in parallel. This is clearly wasteful. Thus it is not surprising that the ATMS stumbles across environments which need not be explored using dependency-directed backtracking. $\{\beta, 2\}$ is an environment which was explored by the ATMS in the example of section 4 but not explored by DDB in section 5.

7. Assumption-based DDB

Given the analysis above, our goal is to combine the features of each approach without inheriting any of their disadvantages. Figs. 2 and 3 highlight the differences between the two approaches. Note that each approach ignores certain environments explored by the other. Thus ideally we would like to construct an approach which explores only the intersection of the environments explored by the two approaches taken alone. This is depicted in Fig. 4. More precisely we would like an approach which explores an environment if 1) it is a subset of a terminal environment explored by DDB, and 2) it is either a consistent environment or a minimally inconsistent environment. To accomplish this task we construct an algorithm called *assumption-based DDB* which takes the DDB algorithm and embeds the ATMS within it. When DDB decides to explore a terminal environment, we use the ATMS scheduler to explore the subsets of the terminal environment, smallest first (again only examining environments which have pending consumers). Thus DDB provides the search strategy with a coarse focus, while the ATMS provides an additional level of discrimination.
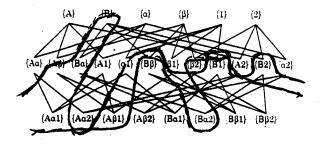

Fig. 4 : Assumption-based DDB controlled search.

Because this approach explores the intersection of the environments explored by DDB and the ATMS, we are guaranteed that its worst case performance with respect to the number of consumer invocations will be at least as

good if not better than the two approaches taken separately. This statement is true *independent* of the number of solutions being explored. Even if the problem solver is only interested in one solution![3]

A more detailed description of the algorithm follows. The consumer scheduler for assumption-based DDB maintains an ordered set of choices, each referred to as a *control disjunction*:

$$control\{C_1, C_2, ...\}$$

A control disjunction consists of an ordered set of assumptions, called *control assumptions*, which are pairwise inconsistent. The system also supports assumptions which are not part of any control disjunction; these are handled by the traditional ATMS mechanism. The scheduler also maintains a single current environment consisting of a set of assumptions, one from each control disjunction. A consumer is not run unless the union of one of its antecedent environments and the current environment is consistent.[4] When a contradiction is encountered, the scheduler finds the next (in chronological backtracking order) environment free from any known contradiction.

Initially we start with the empty current environment E and a stack of control disjunctions S. The backtracker can be implemented by the procedure $backtrack(E, S)$:

1. If S is empty, $schedule(E)$, and return.
2. Let D be the first control disjunction of S, S the remainder.
3. If no remaining assumptions in D, return.
4. Let a be the first control assumption of D, D the remainder.
5. $E' = \{a\} \cup E$.
6. If E' is consistent, $backtrack(E', S)$.
7. If E is now inconsistent, return.
8. Go to 3.

$Schedule(E)$ executes consumers in the smallest environments first. For example, $schedule(\{A, \alpha, 1\})$ executes the consumers in the following order: $\{A\}, \{\alpha\}, \{A, \alpha\}, \{1\}, \{A, 1\}, \{\alpha, 1\}, \{A, \alpha, 1\}$.

We must place two conditions on the problem solver to ensure the correct operation of this backtracking scheme. First, *all* problem-solving operations are performed by the consumers. In particular, the user may not add new consumers, assumptions, justifications, data, etc. during the search. The reason for this is that the new information could create new contexts which have already been implicitly examined by the backtracking mechanism. Thus,

---

[3] In addition, the same claim holds for the number of environments explored, since environments are only explored if they have consumers associated with them.

[4] The set of assumptions in an environment can be broken into two sets: control assumptions and non-control assumptions. The environment will be consistent with the current environment only if 1) the set of control assumptions is a subset of the current environment, and 2) the addition of the non-control assumptions does not cause the current environment to become inconsistent with the current environment.

if new knowledge is added externally during the problem-solving activity, the backtracker must start searching the space from the beginning. In the ATMS framework this is relatively inexpensive (but not free) as the consumer scheduler guarantees no work is ever done twice and remembers all contradictions ever encountered. Second, every action of a consumer must itself depend on all its antecedents (a stipulation already imposed by the ATMS scheduler itself). If a consumer were permitted to perform arbitrary actions, then this would be the same as adding external knowledge.

8. Minimizing Consistency Checks

Prior to running any consumers in an environment, the backtracker first checks it for previously discovered inconsistencies. Although limiting the number of consumers executed is of primary importance, it is also important to limit the number of contexts the backtracking procedure is forced to examine for possible inconsistencies. Intrinsic to the ATMS are two capabilities which reduce the number of contexts tested for consistency. Although these two capabilities do not reduce the number of consumers executed, they do reduce the number of contexts the backtracker must examine.

Unlike a conventional justification-based TMS, the ATMS is guaranteed to explicitly identify *all* nogoods which follow from the current set of justifications. A conventional TMS will only identify the first nogood it comes across (which necessarily identifies the current environment as inconsistent). As a result, a conventional TMS may have to consider environments which would have been excluded using nogoods explicitly identified by the ATMS. For example, suppose we already have the justifications:

$$A, z \Rightarrow \perp,$$

$$1, z \Rightarrow \perp,$$

$$\beta \Rightarrow \perp,$$

and in the current environment $\{A, \alpha, 1\}$ a consumer produces:

$$\alpha \Rightarrow z.$$

In a conventional TMS only one of the two contradictions may be explicitly noted, say $\{\alpha, 1\}$ while an ATMS detects $\{A, \alpha\}$ immediately as well. As a result a conventional TMS tests $\{A, \alpha, 2\}$ while the ATMS never considers it. In addition, $\{\beta\}$ is marked nogood immediately, so $\{A, \beta, 1\}$, $\{A, \beta, 2\}$ will never be considered by the assumption-based backtracker either.

The ATMS also takes advantage of the fact that the set of choices in each control disjunction is exhaustive. Consider the example ($\mapsto$ indicates a consumer):

$$control\{A, B\},$$

$$control\{C, D, E, ...\},$$

$$control\{X, Y\},$$

$$A \mapsto x = 2,$$

$$X \mapsto x = 1,$$

$$Y \mapsto x = 1.$$

The backtracker first explores the environment $\{A, C, X\}$. It is found that $\{A, X\}$ is inconsistent and the corresponding nogood is recorded. Next it explores the context $\{A, C, Y\}$ noticing $\{A, Y\}$ is nogood as well. Traditional dependency directed backtracking would then try exploring the environments $\{A, D, X\}$, $\{A, D, Y\}$, $\{A, E, X\}$, $\{A, E, Y\}, ...$ In each case the backtracker will realize that the environment is a superset of one of the explicit nogoods before running any consumers. Nevertheless, time is wasted switching to each context. Using the ATMS the backtracker is able to infer from the two nogoods, plus the exhaustivity of the third control disjunction that $\{A\}$ is also a nogood. Thus no superset environment of $\{A\}$ is considered further.

To accomplish this the ATMS contains the following hyperresolution rule for disjunctions (of which control disjunctions are an instance).

$control\{A_1, A_2, ...\}$
$nogood\ \alpha_i$ where $A_i \in \alpha_i$ and $A_{j \neq i} \notin \alpha_i$ for all i

$nogood \bigcup_i [\alpha_i - \{A_i\}]$

In this case, the ATMS infers:

$control\{X, Y\}$
$nogood\{A, X\}$
$nogood\{A, Y\}$

$nogood\{A\}$

Therefore, the ATMS-controlled backtracker considers the environment $\{B, C, X\}$ next. It is important to note that the backtracking algorithm is unchanged: all the necessary nogoods are detected by the ATMS itself in its normal operation and are indistinguishable from the nogoods detected explicitly by consumers.

9. Generalized Assumption-based DDB

The ATMS backtracking scheme outlined in the previous section presumes the set of control disjunctions is fixed at the beginning of problem solving and that each set of choices is completely independent. Neither is the case in practice. During problem solving a new set of choices may become of interest in addition to the ones already known. Furthermore, some choice sets are logically dependent on others. Together these make it difficult for the problem solver to reason about its own control. We follow the ideas

of [8] and allow control decisions to have justifications as well. For example, we write:

$$x \Rightarrow control\{A, B\}$$

to state that if x holds, then $control\{A, B\}$ is an *active* control disjunction and has the full force of a normal control disjunction. If a control disjunction has no valid justification, then it is *passive* and is ignored completely. When a control disjunction is passive, its assumptions (unless they appear in other control disjunctions), will never be part of the current environment and hence no consumer which solely depends on them will be executed. The execution condition for consumers is the same as that for the assumption-based DDB of Section 7.

The control disjunction is designed to be used within a schema which ensures that any inferences following from one of its control assumptions also depends on the control disjunction's antecedents. For example, the proposition (if $a_1, a_2,...$ hold, explore c_1 first, then $c_2,...$),

$$a_1 \wedge a_2 \wedge ... \rightarrow c_1 \vee c_2 \vee ...$$

is encoded by the control disjunction,

$$a_1, a_2, ... \Rightarrow control\{C_1, C_2, ...\},$$

and justifications,

$$a_1, a_2, ..., C_i \Rightarrow c_i.$$

This encoding ensures that some c_i will hold only in a context in which $a_1, a_2, ...$ hold as well as the specified control assumption being active. A control disjunction may be believed in some context, but in order to be active it must be consistent with the current environment.

This formulation of dependency-directed backtracking is extremely powerful and allows the problem solver to dynamically manipulate the shape of its search space without requiring any change of terminology or representation. The backtracking algorithm is, however, rather complicated and we briefly outline it here. The generalized backtracker follows the simpler version by depending heavily on the nogood data-base of the ATMS. The backtracker maintains a single stack of the currently active control disjunctions. There are three significant events: a passive control disjunction becomes active, an active control disjunction becomes passive, and the current environment becomes inconsistent.

When a passive control disjunction becomes active add it to the active stack, select the first control assumption from it that can be consistently added to the current environment, and make the resulting environment the new current environment. If no such assumption exists, the hyperresolution rule will have determined that the current environment is inconsistent. As long as the active stack does not change, search continues in chronological order. More than one control disjunction can become active simultaneously. In such cases if desired the control disjunctions can be sorted, oldest first, and an assumption chosen from each in turn. The sorting guarantees that the search space is explored in the order intended by the problem solver.

The case of an active control disjunction becoming passive is more complex. Although far more complex strategies are possible, the simplest technique is to temporarily unwind the control disjunctions on the active stack up to and including the affected control disjunction. As each control disjunction is removed, the active control assumption for that disjunction is removed from the current context. After the affected disjunction is removed the remaining temporarily removed control disjunctions still active are pushed back on the control stack (selecting their first consistent control assumption for the current context).

A more complex strategy would only reexamine those environments whose exploration was blocked (i.e., the environment was inconsistent with the current environment) by nogoods containing a control assumption of the newly passive control disjunction. Such a scheme would avoid examining some environments twice. Fortunately, the ATMS scheduler explicitly records all pending consumers with environments, thus it is almost free to reexamine an environment.

When the current context becomes inconsistent, but the active stack is unchanged, the backtracking proceeds as in the simple assumption-based DDB scheme. However, when more than one of these three conditions occurs simultaneously, these operations must be interleaved to avoid needless thrashing.

10. Related Work

Our approach to backtracking is dependent on the ATMS [4,5,6] but exploits the ideas of explicit control of reasoning [8] and previous approaches to backtracking [9,12].

Our approach is also strongly related to the use of intelligent backtracking in PROLOG[1,2,3]. The backtracking scheme of [1] is similar to the case where the control disjunctions are fixed (our control disjunctions correspond to [1]'s value generators). When a contradiction is encountered, the stack is unwound to the first generator contributing to the contradiction. When a generator is exhausted, the reasons for each eliminated value are combined to form a contradiction which is used to guide the backtracking. This corresponds to the ATMS' use of hyperresolution. The basic difference (as far as backtracking is concerned) is that the assumption-based backtracker records all nogoods permanently while the PROLOG intelligent backtrackers throw away contradiction records when the stack

is unwound. As a consequence some contradictions must be continually rediscovered. [1] argues that the cost of recording and checking for possible contradictions is not worth the computational overhead incurred.

11. Conclusions

Problem solvers built on the ATMS are often very difficult to control. Some problems are inherently ill-suited to the ATMS, but for many others the addition of a simple backtracking control structure resolves many of the difficulties. This paper has presented a simple backtracking scheme which exploits some of the unique properties of both the ATMS and DDB, resulting in a hybrid algorithm whose worst case performance based on rule invocations is superior to the two approaches individually. This performance holds from problems demanding anywhere from a single solution to all solutions.

Non-control assumptions are treated as conventional ATMS assumptions, while control assumptions are treated much like DDB in a justification-based TMS. In addition, the system supports conditional control disjunctions used to model the interactions between "not quite independent" choices. The result is an overall problem solver with the advantages of both an ATMS and dependency-directed backtracking.

ACKNOWLEDGMENTS

BIBLIOGRAPHY

1. Bruynooghe, M., Solving combinatorial search problems by intelligent backtracking, *Information Processing Letters* **12** (1981) 36–39.
2. Bruynooghe, M. and Pereira, L.M., Deduction revision by intelligent backtracking, in: J.A. Campbell (Ed.), *Current Issues in Prolog Implementation*, (Wiley, New York, 1984) 194–215.
3. Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, (Springer-Verlag, New York, 1981).
4. de Kleer, J., An assumption-based truth maintenance system, *Artificial Intelligence* **28** (1986) 127–162.
5. de Kleer, J., Extending the ATMS, *Artificial Intelligence* **28** (1986) 163–196.
6. de Kleer, J., Problem solving with the ATMS, *Artificial Intelligence* **28** (1986) 197–224.
7. de Kleer, J., Choices without backtracking, *Proceedings of the National Conference on Artificial Intelligence*, Austin, TX (August 1984), 79–85.
8. de Kleer, J., Doyle, J., Steele, G.L. and Sussman, G.J., Explicit control of reasoning, in *Artificial Intelligence: An MIT Perspective,* edited by P.H. Winston and R.H. Brown, 1979. Also in: *Proceedings of the Symposium on Artificial Intelligence and Programming Languages,* 1977, Also in: *Readings in Knowledge Representation,* edited by R.J. Brachman and H.J. Levesque, (Morgan Kaufman, 1985).
9. Doyle, J., A truth maintenance system, *Artificial Intelligence* **24** (1979).
10. McAllester, D., A widely used truth maintenance system, unpublished, 1985.
11. Stallman, R. and Sussman, G.J., Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence* **9** (1977) 135–196.
12. Steele, G.L., The definition and implementation of a computer programming language based on constraints, AI Technical Report 595, MIT, Cambridge, MA, 1979.