ICOT Technical Report: TR-593

TR-593

Making Dependency-Directed Search Hierarchical

by

K. Inoue & Y. Ohta

September, 1990

Making Dependency-Directed Search Hierarchical

Katsumi Inoue and **Yoshihiko Ohta**

ICOT Research Center
Institute for New Generation Computer Technology
Mita Kokusai Bldg. 21F
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
phone: +81-3-456-2514
email: {inoue,ohta}@icot.jp

August 31, 1990

Abstract

This paper concerns search algorithms for combinatorial problems including constraint satisfaction problems (CSPs). When good ordering heuristics are not available for a CSP, dependency-directed search is desirable for computing all solutions in multiple contexts to avoid redundant computation. However, previous works on dependency-directed search fail to model or capture the incremental construction of hierarchical structure with various levels of complex and/or large-scale problem solving. Our idea is based on AND/OR tree search underlying the assumption-based truth maintenance system (ATMS). In the ATMS, a context is characterized by a combination of choices (assumptions), without information of hierarchy. In AND/OR tree search procedures, a context can be characterized by a partial solution tree and thus reflects a tree structure. We show the connection between these two approaches. A proposed search algorithm called HDDS improves the search efficiency and the expressive power more than previous methods.

Keywords: Dependency-Directed Search, ATMS, Constraint Satisfaction

1 Introduction

When problems require making lots of choices, decisions or assumptions from items of alternative knowledge, *dependency-directed search* (DDS) [Stallman and Sussman, 1977] is a good way to avoid redundant computation and to prevent rediscovering failures. DDS also plays an important role in problem solving with the *assumption-based truth maintenance system* (ATMS) [de Kleer, 1986a], one of whose tasks is to maintain consistency of multiple contexts of dynamically constructed knowledge bases. By exploiting DDS in problem solving with the ATMS, we can characterize an intelligent search procedure for obtaining all solutions of non-deterministic problems as a generic interface which can give a guide for problem solving between the ATMS and a domain-dependent problem solver [de Kleer, 1986b; de Kleer and Williams, 1986; Inoue, 1988b]. However, DDS has the following problems of efficiency and expressive power, which are dependent on each other.

Problem of Efficiency: One of theoretical goals of DDS is to minimize the area of the search space that must be explored to obtain consistent solutions. Unfortunately, this goal has not been fully achieved. The difficulty partially comes from the computational complexity of finding the maximal consistent sets of assumptions in the ATMS, which is at least as hard as an NP-complete problem; the bound for the number of assumption sets to be examined is impractically large even for a moderate number of assumptions [Provan, 1987]. This is a significant limitation, which simple heuristics of weak methods cannot improve upon without partitioning the problem and knowledge bases. Some information of the problem structure such as hierarchy should be utilized in search.

Problem of Expressive Power: When large-scale and/or complex problems are handled, assumptions must be considered at various levels of problem solving. For example, design problems can be regarded as complicated tasks that contain synthesis tasks as well as analysis and simulation tasks. In design process, first, the structures of the design objects are determined, then the attribute parameters of these structures are refined. Each design decision can be regarded as an assumption, but we are not interested in all combinations of assumptions because many decisions depend on decisions made earlier. Therefore, decisions should be represented in a hierarchy. However, the ATMS identifies a context only by a set of assumptions and cannot utilize a structure of assumptions in such different levels.

This paper focuses on the search for multiple contexts with hierarchy. The exact search procedure realizing DDS called *HDDS*, which is an improved version of the *GSEARCH* algorithm [Inoue, 1988a] and can handle hierarchy better, will be shown in section 4. HDDS improves the search efficiency and the expressive power more than previously proposed algorithms of DDS with the ATMS.

2 Dependency-Directed Search

2.1 ATMS and Interpretation Construction

In the ATMS [de Kleer, 1986a], every datum is expressed as a proposition, and a proposition representing a choice between alternatives is called an *assumption*. A domain-dependent problem solver transmits information of dependencies between propositions, which are represented by propositional formulas (called *justifications*), to the ATMS. A set of assumptions is called an *environment*, and the set of propositions followed by an environment and a set of justifications is called a *context*. When a context is inconsistent, its characterizing environment is called *nogood*. For two environments, E and E', we say E is *more general than* E' (and E' is *less general than* E) if $E \subseteq E'$. Note that if an environment E is nogood, every environment which is less general than E is nogood. The ATMS maintains a global, concurrent representation of all contexts by labeling each proposition with all its most general consistent environments whose contexts contain the proposition. This ATMS technique allows multiple contexts to be compared, switched, or synthesized as needed. In the ATMS, only an environment identifies a context, which avoids redundant computation and duplication of conclusions in different contexts.

Search with the ATMS is based on *interpretation construction*: building the maximal consistent environments. During interpretation construction, the earlier the most general nogoods are found, the more steps concerning ultimately inconsistent environments are reduced. For this purpose, a problem solver focuses breadth-first on environments with fewer assumptions first through a specialized interface, or *consumer architecture* [de Kleer, 1986b]. A *consumer* is a problem-solving procedure attached to its environment. When a contradiction occurs in executing a consumer, we can immediately notice that the corresponding environment is nogood, and thus can ignore the execution of consumers of all of its less general environments. It is by the number of consumers or even by the number of environments whose consumers are executed that the efficiency of a search algorithm is evaluated. Now, for a set S of assumptions, let $\mathcal{P}(S)$ be the power set of S. Let SOL be the set of all consistent solution environments, and NG be the set of all most general nogood environments. Formally, to obtain all consistent solutions, any search procedure must search at least the environments:

$$\bigcup_{S \in SOL \cup NG} \mathcal{P}(S) ,$$

that is, environments to be checked for the consistency of each solution and environments to be pruned of all most general inconsistent environments.

2.2 ADDB Heuristics with Hyperresolution

When only part of the search space should be explored for the purpose of the task, because not all solutions are required at once, or because efficiency requirements demand it, a backtracking mechanism can be used for DDS to improve the efficiency of search. A simple method, called *assumption-based dependency-directed backtracking* (ADDB) [de Kleer and Williams, 1986], that incorporates depth-first search into breadth-first search of consumer architecture can reduce the search by exploiting the following *ADDB heuristics* and an *intelligent backtracker*:

1. By introducing explicit statements of exclusive disjunctions (called *control disjunctions*), the ATMS need not identify any of the trivial nogoods which consist of different items in the same control disjunctions. Without them, the ATMS must make all combinations of assumptions and then prune vast areas of the search space.

2. ADDB focuses *breadth-first* on the power set of the current environment, with fewer assumptions first like consumer architecture. This enlarges the effect of pruning search areas by most general nogoods.

3. ADDB keeps on exploring the current environment *depth-first* as long as its context is consistent, until a solution environment is obtained. This enables the problem solver to find a solution as fast as possible and to perform intelligent backtracking with the problem-solving tasks reduced more than in the simple consumer architecture.

The effect of introducing both the first and the third heuristics of ADDB enables us to utilize an intelligent backtracker, which directly backtracks the choosing point to cause a failure. Intelligent backtracking by ADDB is done by the following *hyperresolution* rule:

$$\frac{\begin{array}{l} P_1 \vee \ldots \vee P_k \\ \mathit{nogood}\ E_i\,,\ \ \text{where } P_i \in E_i \text{ and } P_{j \neq i} \notin E_i\,,\ \text{for all } 1 \leq i, j \leq k \end{array}}{\mathit{nogood} \bigcup_{1 \leq i \leq k} [\, E_i - \{P_i\}\,]\,.}$$

Intuitively, if all choices in a disjunction failed and no failure depended on a choice, say P, then by hyperresolution we can ignore other choices for P.

2.3 Problems of Handling Hierarchy

While the idea of ADDB heuristics is very clear and effective for controlling the ATMS, the ADDB algorithm still has the following big problems for handling hierarchy:

- In ADDB, the current environment must be flatly constructed by a set of assumptions, each of which is exclusively taken from a control disjunction. Therefore, the task that some choices are dependent on other contexts but some are not cannot be dealt with by ADDB directly.

- Hyperresolution requires enormous tasks and reduces efficiency of the ATMS. Moreover, for hierarchical tasks, hyperresolution is not available to produce nogoods in intermediate levels properly.

3 Hierarchical Representation

3.1 Partial Solution Trees

Our main goal now is to formalize a general search algorithm for multiple contexts so as to overcome the problems analyzed in the previous section. Our method for controlling search is via an *AND/OR tree*, where the *root* represents an overall problem to be solved, and *arcs* in it indicate logical dependencies between *nodes* representing decomposition processes or relations of assumptions. Nodes with *children* are called *nonterminal*, and those with no child are called *terminal*. Each nonterminal node has immediate children either of type AND or of type OR. A nonterminal node P with children of type AND is called an *AND node*, and each child corresponds to one of P's conjunctive partial problems:

$$P \rightarrow \bigwedge_{P_i \in Child(P)} P_i ,$$

where $Child(P)$ is the set of all children of P [1]. A nonterminal node P with children of type OR is called an *OR nodes*, and its children correspond to the disjunction representing the possibilities of P's implementation:

$$P \rightarrow \bigvee_{P_i \in Child(P)} P_i .$$

An example of an AND/OR tree is shown in Figure 1.

Given an AND/OR tree representation of assumptions, we can identify its different solutions, each one representing a consistent environment, by a *solution tree*. A solution tree T of an AND/OR tree G is a subtree of G with the following two properties: (i) the root of G is the root of T; and (ii) if an AND node of G is in T, then all of its children are in T, and if an OR node of G is in T, then exactly one of its children is in T. A solution tree may be expressed by its terminal nodes; for example, in Figure 1, $\{A, C, F\}$ is a solution tree shown by the bold line.

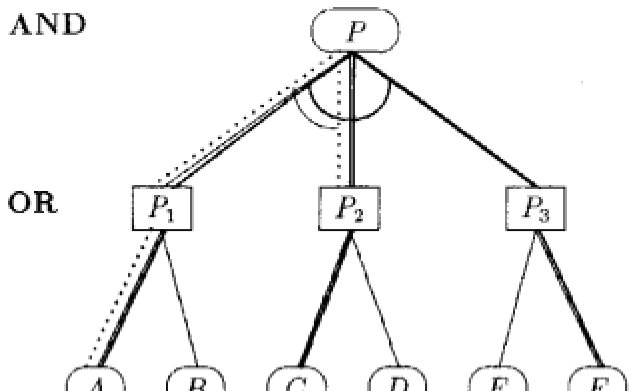

Figure 1. An AND/OR tree with depth 2.

We assume that the search tree G should be *incrementally* constructed, expanded, and traversed during problem solving. This point is very important for practical problem solving

[1] We assume here that problem solving is proceeded *top-down*. In some cases, the meaning of the relation between a node and its children would be "$\equiv$" or "$\leftarrow$" instead of "$\rightarrow$". These distinctions are not essential here.

because not all assumptions and their relations are represented explicitly before any inference starts. Therefore, we shall use a representation for an incomplete solution tree that may be extended, called a *partial solution tree* (PST). A partial solution tree T' of an AND/OR tree G is a subtree of G with the following three properties: (i) the root of G is the root of T'; (ii) if any node other than the root of G is in T', then its ancestors are also in T'; and (iii) if an OR node of G is in T', then at most one of its children is in T'. It is possible to say that a PST is an intersection of solution trees. A PST can be represented by its *tip* nodes, and thus can be associated with an environment. A PST is called *active* when it characterizes a consistent environment. An example of a PST is illustrated in Figure 1 by the dotted line, representing $\{A, P_2\}$.

3.2 Generalized ADDB Heuristics without Hyperresolution

ADDB can be characterized as a search algorithm for an *AND/OR tree with depth 2* whose root is an AND node (such as that shown in Figure 1), because ADDB focuses only on environments consisting of assumptions, each of which is exclusively chosen from a control disjunction. To propose a new search algorithm (in the next section), instead of handling a simple list of alternatives, hierarchical structures for disjunctions are introduced, and ADDB heuristics given in section 2.2 are generalized to handle AND/OR trees with a depth greater than 3 as follows:

1. Two different PSTs are disjunctive, so that we need not create any environment by merging them. Therefore, none of the consumers of such an environment is executed.

2. When a current PST T is examined, environments in $\mathcal{P}(T)$ are focused breadth-first.

3. The current PST keeps on being expanded depth-first to obtain a solution tree, as long as it is active.

An important improvement is made due to the introduction of hierarchy; enormous tasks of hyperresolution can be reduced if we can give consumers detecting inconsistencies in intermediate levels. For example, in Figure 1, when *nogood*$\{A, E\}$ and *nogood*$\{A, F\}$ are found, ADDB can derive *nogood*$\{A\}$ by hyperresolution so that environment $\{A, D\}$ is neither examined to check consistency nor executed its consumers. However, this pruning is not useful when we consider trees with a depth greater than 3 because logical dependencies may be very complex. Instead, if we can give a consumer detecting inconsistency when exploring an intermediate environment $\{A, P_3\}$, we do not need to examine even environments $\{A, E\}$ and $\{A, F\}$. This kind of consumer is used in practice; for example, let P_1 stand for a variable `P1`, P_3 for a variable `P3`, and let A mean an assignment 0 for `P1`, "`P1=0`"; and suppose that the "`prohibit_division_by_zero`" consumer can warn of the illegal computation of `P3/P1`, marking the environment $\{A, P_3\}$ as a nogood, without computing two possible assignments for `P3` which correspond to E and F. Note that this kind of constraint in intermediate levels cannot be handled by ADDB directly.

4 Hierarchical DDS

We now present search procedures for all or some numbers of logically consistent solutions, and for an optimal solution by an estimate.

4.1 Searching Consistent Contexts

The HDDS context search algorithm maintains three sets: $OPEN$, NG and SOL. $OPEN$ is a set of active PSTs, each of which represents a state of traversal corresponding to a consistent environment. NG is a set of most general nogoods that have been found. The "generality" of environments will be redefined to reflect their hierarchical structures. SOL is a set of complete consistent solution trees. In the HDDS algorithm, at the beginning, a problem P_0 to be solved is placed in $OPEN$. The basic loop consists of picking one active PST from $OPEN$, checking its consistency, executing its consumers if the test is all right, and then decomposing it or traversing a search by the EXPAND procedure. If a contradiction occurs in a context, the corresponding environment is added to NG through the NOGOOD procedure.

Procedure HDDS:
Remark. Once a consumer is executed, it is discarded and never executed again.

Notations. An active PST in $OPEN$ can be represented by a pair, (P, E), where P is a node and E is a consistent environment to be combined with P. The corresponding environment of (P, E) can be represented by the set $E \cup \{P\}$ of tip nodes, which is denoted $\langle E, P \rangle$, where the $\langle\ \rangle$ operator concatenates, flattens, and merges all elements; for example, $\langle \{S, T\}, A, \{T, X\} \rangle = \{S, T, A, X\}$ and $\langle \phi \rangle = \phi$. `Pick`$(a, A)$ means to pick the first element a from A and remove it from A. `Push`(a, A) means to add a to the head of A. `Schedule`(X) means to produce and sort the power set $\mathcal{P}(X)$ of X scheduled in the bit-vector ascending order; for example, when $X = \{A, B, C\}$, `Schedule`$(X) = \{\phi, \{A\}, \{B\}, \{A, B\}, \{C\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}$. `Examine`$(E)$ executes all consumers attached to E.

1. Let $OPEN := \{(P_0, \phi)\}$, $NG := \phi$, and $SOL := \phi$.
Execute all consumers attached to the overall environment $\{\}(= \phi)$ which consists of no assumption.

2. Halt if a *termination condition* is satisfied; consistent solutions are given by SOL. A termination condition is either of the following: (a) $OPEN = \phi$ (to get all solutions), or (b) the number of elements of SOL is equal to the given number of solutions.

3. `Pick`$(S, OPEN)$, and let $S = (P, T)$.

4. Generate the next child, P_i, of P.
$\Omega(T) :=$ `Schedule`(T).

5. If $\Omega(T) = \phi$, then $EXPAND(S, P_i)$ and return to 2.

6. `Pick`$(t_j, \Omega(T))$. $T_{ij} := \langle t_j, P_i \rangle$.

7. `Examine`(T_{ij}). If an inconsistency is found in executing consumers, then $NOGOOD(S, T_{ij})$ and return to 2. Otherwise, return to 5. □

Procedure EXPAND$((P, T), P_i)$:
Remark. This procedure performs *depth-first* search for examining PSTs.

Notations. `Exhaust`(P) means that all children of P have been **examined**. **Notexhaust**(P) means that there is at least one un-**examined** child of P. `And`(P) (`Or`(P)) means that P is an AND (OR) node. `Terminal`(P) (`Nonterminal`(P)) means that P is terminal (nonterminal). **Allexpanded**(T) means that all nodes in T have been expanded, that is, every tip node in T is terminal. `Unexpanded`(T, I) means that there is at least one nonterminal node in T and returns the left-most such node I. `LMC`(P) is the left-most child of P.

Apply one of the following rules according to the status of P, T, and P_i, then return.

Case 1: `Notexhaust`(P), `And`(P)
$\Rightarrow$ `Push`$((P, \langle T, P_i \rangle), OPEN)$.

Case 2: `Notexhaust`(P), `Or`(P)
$\Rightarrow$ `Push`$((P, T), OPEN)$, `Push`$((P_i, T), OPEN)$.

Case 3: `Exhaust`(P), `Nonterminal`(P_i), `And`(P)
$\Rightarrow$ `Push`$((\texttt{LMC}(P), \langle T, P_i \rangle \backslash \texttt{LMC}(P)), OPEN)$.

Case 4: `Exhaust`(P), `Nonterminal`(P_i), `Or`(P)
$\Rightarrow$ `Push`$((P_i, T), OPEN)$.

Case 5: `Exhaust`(P), `Terminal`(P_i), `Allexpanded`(T)
$\Rightarrow$ `Push`$(\langle T, P_i \rangle, SOL)$.

Case 6: `Exhaust`(P), `Terminal`(P_i), `Unexpanded`(T, I)
$\Rightarrow$ `Push`$((I, \langle T \setminus I, P_i \rangle), OPEN)$. □

Procedure NOGOOD$((P, T), T_{ij})$:
Definition. A PST, T, is a *specialization* of another PST, T', if for each element, $t' \in T'$, there is an element, $t \in T$, such that $t = t'$ or t is a descendant of t'. In this case, T' is said to be *more general than* T.

1. `Or`(P), `Notexhaust`(P) $\Rightarrow$ `Push`$((P, T), OPEN)$.

2. If T_{ij} is a specialization of any other element in NG, then return. Otherwise, add T_{ij} to NG.

3. Delete each element in NG which is a specialization of T_{ij} from NG.

4. **(Prune)** Delete each active PST in $OPEN$ whose corresponding environment is a specialization of T_{ij}, from $OPEN$. Return. □

Lemma 1 When an active PST, $S = (P, T)$, is picked from $OPEN$ in step 3 of HDDS, T is consistent.

> **Proof:** *(Sketch)* No PST in $OPEN$ includes any element of NG by step 4 of NO-GOOD. S must be constructed in such a way that the consistency of T has already been checked in steps 5 to 7 of HDDS. □

By Lemma 1, HDDS needs to check only the consistency of every combination of new state P_i with each element of $\mathcal{P}(T)$, so that the consistency of neither element of $\mathcal{P}(T)$ need be reexamined.

Theorem 2 (*Admissibility of HDDS*) At the end of HDDS, any solution tree in SOL is consistent, and when all consistent solutions are desired, SOL holds all of them.

> **Proof:** *(Sketch)* By using Lemma 1 inductively, the soundness follows from the fact that any solution is added to SOL in EXPAND (case 5) after its consistency has been checked. To prove the completeness, suppose to the contrary that HDDS misses a solution, say S. Since every consistent PST is added to $OPEN$ and picked from it unless it becomes contradictory, there exists an environment, S', which is more general than S such that S' is selected from $OPEN$ and is not to be inserted in $OPEN$ again. Then S' is either found to be inconsistent, or added to SOL. Both cases contradict the supposition. □

HDDS has several advantages as it searches an AND/OR tree constructed with hierarchy incrementally. With HDDS, problem solving can proceed efficiently with *compiled knowledge* so that a kind of compilation of a condition on a set of lower-level knowledge can be represented. This is our main solution for the problems of ADDB given in section 2.3. As described in section 3.2, in many cases no intelligent backtracker is necessary because upper-level contradictions can be found through compiled constraints before lower-level contradictions are found.

4.2 Informed Search

When an estimate for assumptions or environments is available, we can expect to improve the search performance. We can order environments by comparing them with some *preference* relation, and an optimal solution can be gained. For this purpose, we may change HDDS in section 4.1 slightly. The concept of checking consistency may be altered to *feasibility*, or possibility for optimality. The selection rule in step 3 of HDDS or the expansion rule in EXPAND may be altered so that the most preferred environment is selected and expanded. The resulting procedure performs *best-first search* like AO^* [Nilsson, 1980], or it supports a *branch and bound* procedure [Ibaraki, 1977]. It seems to be rationally efficient for multiple context handling that best-first search is employed to elicit its advantage of the concurrency.

5 Application to Design

In this section, the working of the HDDS algorithm on *constraint satisfaction problems* (CSPs) in *parametric design* is illustrated. In CSPs, consistent assignments of values for a set of variables which satisfy all constraints are to be found. In practice, however, constraint networks for parametric design problems can not be explicitly given in advance. Moreover, it takes lots of time and space to execute each consumer involving an analysis or simulation task. Because of these properties, various kinds of ordering heuristics or network-based heuristics (for example, [Mackworth, 1977; Dechter and Pearl, 1988]) for CSPs are not readily available. Instead, *dependency analysis* is useful for this kind of CSP[2].

The HDDS procedure can be applied to this type of CSP as follows. The simple CSP can be characterized as an AND/OR tree with depth 2 (like Figure 1) whose root is an AND node and whose nodes in level 1 are OR nodes representing variables, and terminal nodes represent domains of their parent nodes. An assumption at a terminal is an assignment of a value to a variable. An important advantage of HDDS is that *hierarchical generate and test* can be supplied to make the search more efficient: an assumption in an intermediate level can represent compiled knowledge or an abstraction of lower-level parameters, so that PSTs can be pruned by these constraints. This way of representation is reported to be very useful for CSPs in the independent research of [Mittal and Frayman, 1987]. It should be also noted that HDDS can be applied to design tasks other than simple CSPs, where the problems need to be selected for their design models as well as for the values of the variables for those models. A *design model* can be expressed by a set of constraints, and can be represented by a hierarchical structure, below which its lower-level parameters can be attached.

6 Related Work

6.1 Implied-by Strategy

HDDS can be compared with the independent research of the *implied-by strategy* of `ATMoSphere` [Forbus and de Kleer, 1988]. There are some similarities between HDDS and the implied-by strategy: (1) assumptions are only created when needed; (2) both use AND/OR tree search schemes allowing for some infinite domain; and (3) consumers are executed only in the focus environment. The differences are: (1) control by `ATMoSphere` is strongly domain-dependent, so that the user must specify how to switch contexts by using contradiction consumers or some scoring mechanism, while HDDS selects the next context automatically by a generic controlling mechanism; and (2) in `ATMoSphere` an AO-node itself represents an environment, while in HDDS a PST represents an environment, so that HDDS may create fewer nodes than `ATMoSphere`. As a result, if the user cannot specify how to switch contexts, search by `ATMoSphere` will be inefficient. As stated in section 5, many problems that require dependency analysis cannot be given good ordering heuristics.

6.2 SSS*: Best-First AND/OR Tree Search

Although informed search embedded in HDDS was introduced in section 4.2, even the concept of generalized ADDB heuristics without preference relations described in sections 3.2 and 4.1 is very close to the notion of pruning trees in conventional search techniques in AI. Stockman [1979] regarded minimax game tree search as AND/OR tree search and proposed a procedure called

[2]Another claim on the relationship between CSP and ATMS techniques is also given in [de Kleer, 1989].

*SSS**, and Ibaraki [1986] generalized the idea by analyzing the usage of heuristic information pertaining to nonterminal nodes, such as *upper* and *lower bounds* of the exact values. SSS* can be regarded as a best-first search procedure preferring the PST whose upper bound of the exact value, which can be computed as the minimum value in upper bounds of all its tip nodes, is the highest in all active PSTs. Our HDDS is similar in this point. For a node P, let Γ_P denote the assumption associated with P. Then, each PST T in $OPEN$ has its upper bound $U(T)$, as follows:

$$U(T) = \bigwedge_{P \in Tip(T)} U_T(P),$$

where $Tip(T)$ is the set of all tip nodes of T, and

$$U_T(P) = \begin{cases} \Gamma_P, & \text{if } P \text{ is a terminal node and} \\ & \text{can be consistently assumed;} \\ \textbf{False}, & \text{if } P \text{ is contradictory} \\ & \text{(terminal or nonterminal);} \\ \textbf{True}, & \text{otherwise.} \end{cases}$$

A PST T is *active* if $U(T) \neq$ **False**. For two PSTs, T_1 and T_2, their upper bounds of Boolean values are ordered by the subsumption relation:

$$U(T_1) \leq U(T_2) \quad \text{if} \quad \models U(T_1) \rightarrow U(T_2).$$

SSS*'s strategy corresponds to the ADDB breadth-first heuristics; preferring an active PST whose upper bound is maximal. However, if (more than) two different maximal Boolean values cannot be compared (that is, neither one subsumes the other), HDDS prefers the left-most active PST. Moreover, SSS* generates all children of a node at one time, while HDDS generates them one by one, so that it may handle the case where children are infinitely many but are expected to be contradictory as a whole.

7 Conclusion

This paper introduced a novel technique to control reasoning with DDS. The resulting search procedure, HDDS, models the incremental construction of hierarchical structure in DDS. The main characteristics of the proposed method are that reasoning is controlled by an AND/OR tree search mechanism, and that assumptions can be added to environments along their contexts incrementally rather than added to every possible environment concurrently in a flat structure. Informed search can be incorporated into this method to make searching more efficient. This mechanism can solve complex and hierarchical problems such as design tasks. We are currently applying HDDS to an extended version of the `APRICOT/0` system and its application to logic design [Ohta and Inoue, 1990].

Acknowledgment

We are grateful to Koichi Furukawa of ICOT and Yasuo Nagai of Toshiba for their useful comments, and to Prof. Toshihide Ibaraki of Kyoto University for discussions on comparison with conventional search methods.

References

[Dechter and Pearl, 1988] Rina Dechter and Judea Pearl. Network-Based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence* **34** (1988) 1–38.

[de Kleer, 1986a] Johan de Kleer. An Assumption-based TMS. *Artificial Intelligence* **28** (1986) 127–162.

[de Kleer, 1986b] Johan de Kleer. Problem Solving with the ATMS. *Artificial Intelligence* **28** (1986) 197–224.

[de Kleer, 1989] Johan de Kleer. A Comparison of ATMS and CSP Techniques. *Proc. IJCAI-89*, Detroit (1989) 290–296.

[de Kleer and Williams, 1986] Johan de Kleer and Brian C. Williams. Back to Backtracking: Controlling the ATMS. *Proc. AAAI-86*, Philadelphia (1986) 910–917.

[Forbus and de Kleer, 1988] Kenneth D. Forbus and Johan de Kleer. Focusing the ATMS. *Proc. AAAI-88*, St. Paul (1988) 193–198.

[Ibaraki, 1977] Toshihide Ibaraki. The Power of Dominance Relations in Branch and Bound Algorithms. *J. ACM* **24** (1977) 264–279.

[Ibaraki, 1986] Toshihide Ibaraki. Generalization of Alpha-Beta and SSS* Search Procedures. *Artificial Intelligence* **29** (1986) 73–117.

[Inoue, 1988a] Katsumi Inoue. Pruning Search Trees in Assumption-based Reasoning. *Proceedings of the 8th International Workshop on Expert Systems & their Applications*, Avignon (1988) 133–151.

[Inoue, 1988b] Katsumi Inoue. Problem Solving with Hypothetical Reasoning. *Proceedings of the 3rd International Conference on Fifth Generation Computer Systems*, Tokyo (1988) 1275–1281.

[Mackworth, 1977] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence* **8** (1977) 99–118.

[Mittal and Frayman, 1987] Sanjay Mittal and Felix Frayman. Making Partial Choices in Constraint Reasoning Problems. *Proc. AAAI-87*, Seattle (1987) 631–636.

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, Calif., 1980.

[Ohta and Inoue, 1990] Yoshihiko Ohta and Katsumi Inoue. A Forward-Chaining Multiple Context Reasoner and Its Application to Logic Design. *The 2nd IEEE International Conference on Tools for Artificial Intelligence*, Washington D.C. (1990) to appear.

[Provan, 1987] Gregory M. Provan. Efficiency Analysis of Multiple-Context TMSs in Scene Representation. *Proc. AAAI-87*, Seattle (1987) 173–177.

[Stallman and Sussman, 1977] Richard M. Stallman and Gerald J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence* **9** (1977) 135–196.

[Stockman, 1979] G. C. Stockman. A Minimax Algorithm Better Than Alpha-Beta? *Artificial Intelligence* **12** (1979) 179–196.