TR-444

Hierarchical Representation for Dependency-Directed Search

by
K. Inoue

December, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 - 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Hierarchical Representation for Dependency-Directed Search

Katsumi Inoue

ICOT Research Center,
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, Japan
phone: +81-3-456-2511, telex: ICOT J 32964
csnet: inoue%icot.jp@relay.cs.net
uucp: {enea,inria,kddlab,mit-eddie,ukc}!icot!inoue

ICOT Technical Report TR-444
December 8, 1988

ABSTRACT

This paper describes a general search algorithm for multiple contexts. Dependency-directed search is desirable for searching multiple contexts when good ordering heuristics are not available. However, previous work on dependency-directed search fails to model or capture the incremental construction of hierarchical structure in various levels of complex and/or large scale problem solving. Our idea is based on AND/OR tree search with underlying assumption-based reasoning. In assumption-based reasoning, a context can be characterized by a combination of assumptions, while in AND/OR tree search procedures, it can be characterized by a partial solution tree. A context deriving a contradiction is checked in a mechanism of truth maintenance, making the search efficient. A search algorithm called GDDS improves the search efficiency and the expressive power more than those of various truth maintenance systems.

1. Introduction

Search is an integral element of all AI systems. In past AI systems, problem solving frameworks were considered from the viewpoint of efficiency in search processes. In general, the problem solving process may be represented by a model in which an AND/OR graph is constructed incrementally at the same time that the search is conducted. In searching such a graph, if there are alternatives among items of knowledge, so that deterministic choices on OR branches are not possible, a single branch must be chosen and reasoning processed further from there regarded as *assumptions*. When such an inference does not result in a satisfactory solution, it may be necessary to backtrack and try a different choice or choices. For this purpose, *dependency-directed search* (DDS) is a good way to avoid redundant computing and rediscovering failures involved by chronological backtracking, and it is very common for recent AI systems to control *assumption-based reasoning*. Many problems require this kind of reasoning in *multiple contexts*. Although DDS itself is a powerful way to handle OR parts intelligently, previously proposed methods have some problems of efficiency and 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 a synthesis task in addition to analysis and simulation tasks. In design process, first, the structures of the design object are determined, then the attribute parameters of structures of the design object are refined. Each design decision is regarded as assumptions. However, we are not interested in all combinations because one decision depends on decisions made earlier. Therefore, decisions should be represented in a hierarchy. It is not natural to treat assumptions on such different levels as a single combination of assumptions, as in the ATMS [de Kleer 86a].

Recently, to tackle this problem, control mechanisms for assumption-based reasoning have been proposed from the viewpoint of applications, such as the *implied-by* strategy in ATMoSphere [Forbus & de Kleer 88] and *GSEARCH* algorithm [Inoue 87] in the APRICOT system [Inoue 88]. This paper focuses on the search for multiple contexts with hierarchy. The exact search procedure realizing DDS called *GDDS*, which is an improved version of GSEARCH, will be shown in section 4. In GDDS, not every hypothesis is handled concurrently; and the incremental addition of hypotheses according to their contexts is possible. GDDS improves the search efficiency and the expressive power more than previously proposed algorithms of DDS with the ATMS.

2. Dependency-Directed Search

DDS plays an important role in *truth maintenance systems* (TMSs), whose main task is to maintain consistency of dynamic knowledge bases. We assume that our framework has a TMS which maintains multiple contexts simultaneously like de Kleer's *assumption-based truth maintenance system* (ATMS) [de Kleer 86a]. The TMS is generic while the problem solver reasons dependently on the problem domain. In APRICOT, the DDS procedure is considered to be a generic interface which can give a guide for problem solving between the TMS and the problem solver, but is considered to be domain-dependent in the sense of utilizing domain heuristics [Inoue 88].

2.1 Searching Multiple Contexts with the ATMS

In the following, a set of assumptions is called an *environment*, and a set of data followed by an environment and a set of axioms (or *justifications*) is called a *context* [de Kleer 86a]. The ATMS maintains a global, concurrent representation of all contexts by labeling each item of data with environments. It allows multiple contexts to be compared, switched, or synthesized as needed. In the ATMS, only an environment identifies a context, avoiding redundant computations and duplication of conclusions in different contexts. Therefore, the method proposed in this paper follows this approach.

With the ATMS, however, there is still a major problem of controlling the inference during search for multiple contexts. Search in the ATMS is based on *interpretation construction*. During interpretation construction, the earlier the most general *nogoods*, that is, inconsistent environments, are found, the more steps concerning ultimately inconsistent environments are reduced. For these purposes, a problem solver focuses breadth-first on environments with fewer assumptions first through a specialized interface, or *consumer architecture* [de Kleer 86b]. The scheduler needs backtracking when only part of the search space should be explored for the purpose of the characteristics

of tasks, such that not all solutions are required at once, or requirement of efficiency. Therefore, a simple method, called *assumption-based dependency-directed backtracking* (ADDB), that combines depth-first search of DDB [Doyle 79] and breadth-first search of consumer architecture is proposed in [de Kleer & Williams 86].

2.2 ADDB Heuristics and Intelligent Backtracker

One of theoretical goals of DDS is to minimize the area to explore the search space before obtaining solutions. Unfortunately, this goal has not been fully achieved. Intuitively, to obtain all consistent solutions, any search procedure must search at least the union of the *union of power sets of all solution environments* and the *union of power sets of all nogoods*. The first half is required to check the consistency of each solution and the last half to prune all inconsistent environments. Nevertheless, we can reduce the search by exploiting the following *ADDB heuristics* and an intelligent backtracker.

The contributions of the idea of ADDB to reduce the search are as follows. (1) By introducing *control disjunctions*, the ATMS need not identify explicitly the trivial nogoods of exclusive disjunctive relations. 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. This enlarges the effect of pruning search areas by smaller nogoods. (3) ADDB keeps exploring *depth-first* on a consistent environment as long as the 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 DDB 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 *hyperresolution* of the extended ATMS. By hyperresolution, if all choices in a disjunction failed and no failure depended on a choice, say P, then we can ignore other choices for P. Similar mechanisms were introduced in the dependency analysis in SCHEMER [Zabih *et al.* 87].

3. Hierarchical Representation

3.1 Problems of Handling Hierarchy

In section 1, we stated that design tasks should be represented in a hierarchy. In the field of *nonmonotonic logics*, several methods to handle hierarchies have recently been proposed. However, they mainly focus on only *taxonomic hierarchy* (e.g., [Konolige 88]), and problem-subproblem decomposition hierarchy has not been considered so far. To handle this kind of hierarchy, it is more natural to regard the process as controlling assumption-based reasoning with DDS. The idea of ADDB heuristics is very clear and effective for controlling the ATMS. However, the ADDB algorithm still has the following major problems: (1) In ADDB, the current environment must be flatly constructed from all control assumptions. Therefore, the task that some choices are dependent on other contexts and some are not cannot be dealt with by ADDB directly. (2) Hyperresolution by the ATMS requires enormous tasks and reduces efficiency of the ATMS. In [de Kleer 88], a new *label* updating algorithm which does not require hyper-

resolution was proposed, but it does not appear to be easily controllable. Moreover, for hierarchical tasks, hyperresolution or the new labeling algorithm are not available to produce nogoods properly in intermediate levels. To solve these problems, GSEARCH [Inoue 87] controls reasoning based on the AND/OR tree search procedure, and the implied-by strategy has been proposed independently by [Forbus & de Kleer 88]. The control by implied-by strategy is domain-dependent, so that the user must specify how to switch contexts explicitly, while our method selects the next context automatically by a generic controlling mechanism. A more detailed discussion will be given in section 6.1.

3.2 Partial Solution Trees

Our main goal 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 *sons* are called *nonterminal*, and those with no son are called *terminal*. Nonterminal nodes with sons of type AND are called *AND nodes*, and their sons correspond to conjunctive partial problems. Nonterminal nodes with sons of type OR are called *OR nodes*, and their sons correspond to disjunctions as possibilities of implementation. As already discussed in section 1, multiple contexts can be basically characterized by an AND/OR tree or an AND/OR graph. Although the search space for nondeterministic problems has been dealt with by OR trees in such as SCHEMER, the representation with AND/OR trees has several advantages over the one with OR trees, because with AND/OR trees, it is more natural to describe a problem and more compact to represent a problem avoiding duplication.

Given an AND/OR tree representation of assumptions, we can identify its different solutions, each one representing a possible 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 sons are in T, and if an OR node of G is in T, then exactly one of its sons is in T. A solution tree may be expressed by its *tip* nodes. An example of an AND/OR tree is shown in Figure 1. In the figure, the solution tree $\{A, C, F\}$ is shown by the bold line. We assume that the search tree G should be *incrementally* constructed, expanded, and traversed during problem solving, so that it is not necessary to explore all assumptions as a whole search tree. This point is very important for practical problem solving because not all assumptions or their relations are represented explicitly before any inference starts. We shall use a representation for a set of solution trees, that is, a *partial solution tree*. 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 sons is in T'. It is possible to say that T' is a set of T and that it is an incomplete solution tree which may be extended. A partial solution tree is called *active* when it characterizes a consistent environment. An example of a partial solution tree is illustrated in Figure 1 by the dotted line, representing $\{A, P_2\}$.

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

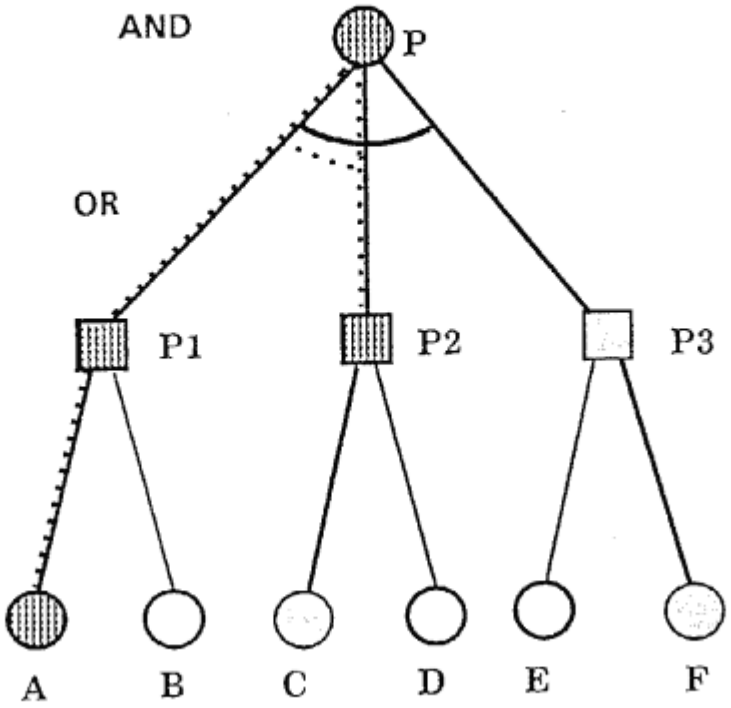

Figure 1. An AND/OR search tree with depth 2.

only on environments consisting of one assumption from each control disjunction.[1] In the next section, we propose a new search algorithm based on a generalization of ADDB heuristics given in section 2.2. For this purpose, instead of handling a simple list of alternatives, hierarchical structures for disjunctions are introduced, and ADDB heuristics are generalized to handle AND/OR trees with more than depth 3 as follows. (1) Each OR relation corresponds to disjunctions, so that we need not create an environment which combines two different partial solution trees. In other words, any *consumers* of such environments are not executed. (2) When a new partial solution tree is examined, environments with fewer assumptions are focused on breadth-first. (3) An active partial solution tree is explored depth-first to obtain a solution tree as long as it is consistent.

An important improvement is made because of the hierarchy; enormous tasks of hyperresolution can be reduced if we can give some consumers detecting inconsistency in an intermediate level. For example, in Figure 1, ADDB can derive $nogood\{A\}$ by hyperresolution if $nogood\{A, E\}$ and $nogood\{A, F\}$ are found, so that environment $\{A, D\}$ is not examined to check consistency. However, this pruning is not possible for trees with more than depth 3. The exact pruning rule may be given as:

`From` $nogood\{A, E\}$ `and` $nogood\{A, F\}$, `derive` $nogood\{A, P_3\}$.

because P may not be the root of the whole problem. This is why hyperresolution is

[1] Moreover, ADDB requires that all assumptions appearing in a control disjunction must be defined before the disjunction is asserted. In our method, each node appearing in a disjunction is incrementally generated instead of all of them being defined, and it may not be a control assumption.

not available for hierarchical structures. However, 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, A means `P1=0`, P_3 means a variable `P3`, and `divide_by_zero` consumers can prohibit the computation of `P3/P1` marking the environment as a nogood. Note that this kind of constraint in intermediate levels cannot be expressed by ADDB.

4. General Dependency-Directed Search

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 **GDDS** context search algorithm maintains three sets: $OPEN$, $NOGOOD$ and $SOLUTION$. $OPEN$ is a set of active partial solution trees, each of which represents a state of traversal corresponding to a consistent environment. $NOGOOD$ is a set of maximally general nogoods that have been found. $SOLUTION$ is a set of complete consistent solution trees. In the GDDS algorithm, the basic loop consists of picking one active partial solution tree from $OPEN$, checking its consistency, executing the problem solving procedures attached to its environment 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 $NOGOOD$ through the **NOGOOD** procedure. An active partial solution tree in $OPEN$ can be represented by a pair, $(P.E)$, of a node, P, to be expanded and a consistent environment, E, to be combined with P. The corresponding environment of (P, E) can be represented by a set of tip nodes, $< E, P >$, where the $<>$ operator concatenates, flattens, and merges all elements. for example, $< \{S,T\}, A, \{T,X\} >= \{S, T, A, X\}$ and $< \phi >= \phi$. In the algorithm, `Pick`(a, A) means to pick the first element, a, from A and remove it from A, and `Push`(a, A) means to add a to the head of A. `Schedule`(X) means to produce the scheduled power set of X, for example, when $X = \{A, B, C\}$, `Schedule`(X) $= \{\phi, \{A\}, \{B\}, \{A, B\}, \{C\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}$. In GDDS, at the beginning, a problem, P_0, to be solved is assigned to $OPEN$, and all consumers attached to the overall environment $\{\}$ which consists of no assumptions are executed.

procedure GDDS:

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

1. Let $OPEN := \{(P_0, < \phi >)\}$, $NOGOOD := \phi$, and $SOLUTION := \phi$.
2. Halt if a *termination condition* is satisfied; consistent solutions are given by $SOLUTION$. A termination condition is either of the following: (a) when all consistent solutions are desired, $OPEN = \phi$, or (b) when the number of elements of $SOLUTION$ is equal to the given number of solutions.
3. `Pick`$(S.OPEN)$, where $S = (P, T)$.
4. Generate the next son, P_i, of P. $\Omega(T) :=$ `Schedule`(T).
5. If $\Omega(T) = \phi$ holds, then $EXPAND(S, P_i, \phi)$ and return to 2.
6. `Pick`$(t_i.\Omega(T))$. $T_i :=< t_i, P_i >$.
7. Execute consumers attached to T_i. If an inconsistency is found in executing consumers, then $EXPAND(S, P_i, t_i)$ and return to 2. Otherwise, return to 5. □

procedure EXPAND$((P,T),P_i,t_i)$**:**
Remarks. `Exhaust`(P) means that all sons of P have been examined. `Notexhaust`(P) means that there is at least one unexamined son of P. `And`(P) (`Or`(P)) means that P is an AND (OR) node. `Terminal`(P) (`Nonterminal`(P)) means that P is terminal (nonterminal). `Unexpanded`(T,P') means that there is at least one unexpanded nonterminal node in T and returns the left-most such node, P'. `Expanded`(T) means that all nodes in T have been expanded. `Firstson`(P) returns the left-most son of P.
Case 1: $t_i = \phi$, `Notexhaust`(P), `And`(P) $\Rightarrow$ `Push`$((P,< T,P_i >),OPEN)$.
Case 2: $t_i = \phi$, `Notexhaust`(P), `Or`(P)
$\Rightarrow$ `Push`$((P,T),OPEN)$, `Push`$((P_i,T),OPEN)$.
Case 3: $t_i = \phi$, `Exhaust`(P), `Nonterminal`(P_i), `And`(P)
$\Rightarrow$ `Push`$((\text{Firstson}(P),<< T,P_i > -\text{Firstson}(P)>),OPEN)$.
Case 4: $t_i = \phi$, `Exhaust`(P), `Nonterminal`(P_i), `Or`(P) $\Rightarrow$ `Push`$((P_i,T),OPEN)$.
Case 5: $t_i = \phi$, `Exhaust`(P), `Terminal`(P_i), `Unexpanded`(T,P')
$\Rightarrow$ `Push`$((P',< T-P',P_i >),OPEN)$.
Case 6: $t_i = \phi$, `Exhaust`(P), `Terminal`(P_i), `Allexpanded`(T)
$\Rightarrow$ `Push`$(< T,P_i >,SOLUTION)$.
Case 7: $t_i \neq \phi$, `And`(P) $\Rightarrow$ $NOGOOD(< t_i,P_i >)$.
Case 8: $t_i \neq \phi$, `Or`(P), `Notexhaust`(P)
$\Rightarrow$ `Push`$((P,T),OPEN)$, $NOGOOD(< t_i,P_i >)$.
Case 9: $t_i \neq \phi$, `Or`(P), `Exhaust`(P) $\Rightarrow$ $NOGOOD(< t_i,P_i >)$. □

procedure NOGOOD(T)**:**
Definition. A partial solution tree, T, is a *specialization* of another partial solution tree, T', if and only 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'.

1. If T is a specialization of any other element in $NOGOOD$, then return. Otherwise, add T to $NOGOOD$.
2. Delete each element in $NOGOOD$ which is a specialization of T from $NOGOOD$.
3. Delete each active partial solution tree in $OPEN$, whose corresponding environment is a specialization of T, from $OPEN$. Return. □

Lemma 1. When an active partial solution tree, $S = (P,T)$, is picked from $OPEN$ in step 3 of GDDS, T is consistent.
Remark. From this property, GDDS checks only the consistency of combinations of T and new state P_i, so that the consistency of the power set of T need not be reexamined.

Proof. No partial solution tree in $OPEN$ includes any element of $NOGOOD$ by step 3 of NOGOOD. S must be constructed with the environment, T, in EXPAND, where the consistency of T has already been checked in steps 5 to 7 of GDDS. □

Theorem 2. At the end of GDDS, any solution tree in $SOLUTION$ is consistent, that is, GDDS is *sound*, and when all consistent solutions are desired, $SOLUTION$ holds all of them, that is, GDDS is *complete*.

Sketch of proof. By using Lemma 1 inductively, the soundness follows from the fact that any solution is added to $SOLUTION$ in EXPAND (case 6) after its consistency has been checked. To prove the completeness, we assume the contrary and obtain a

contradiction. Suppose GDDS misses the solution, S. Since every consistent partial solution tree is added to $OPEN$ and picked from it unless it becomes contradictory, there exists an environment, S', which is a partial solution tree of 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 $SOLUTION$. Both cases contradict the supposition. □

GDDS has several advantages as it searches an AND/OR tree constructed with hierarchy incrementally. With GDDS, problem solving can proceed efficiently with *compiled knowledge* because a contradiction in an intermediate level can be found so that a kind of compilation of a condition on a set of low-level knowledge can be represented. This is our main solution for the problems of ADDB given in section 3.1. As described in section 3.2, in many cases no intelligent backtracker is needed because compiled constraints can find upper-level contradictions before lower-level contradictions are found. Nevertheless, if we want to derive nogoods from $NOGOOD$ in the similar way to hyperresolution, a partial solution tree, T, can be pruned when its specializations are inconsistent and they cover exhaustively the expanded sons of an OR node of T. In GDDS, this can be implemented more simply and can be embedded not in the ATMS itself, but in the search algorithm, so that the performance of the basic ATMS is not reduced at all. For this purpose, case 9 of EXPAND can be changed as follows:

Case 9_A: $t_i \neq \phi$,`Or`(P), `Exhaust`(P),`OnesonIN`$(P) \Rightarrow NOGOOD(< t_i, P_i >)$.
Case 9_B: $t_i \neq \phi$,`Or`(P), `Exhaust`(P),`AllsonsOUT`$(P, N) \Rightarrow NOGOOD(< N, t_i, P >)$.
Remarks. `AllsonsOUT`(P, N) means that all partial solution trees containing each son, P_j $(1 \leq j \leq i-1)$, of P are inconsistent, and that all nogoods are gathered as $N :=< t_1, \ldots, t_{i-1} >$, where for each P_j, $< t_j, P_j >$ is nogood. `OnesonIN`(P) means that at least one expanded partial solution tree containing a son of P remains consistent.

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 GDDS 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 GDDS or the expansion rule in EXPAND may be altered so that the most preferred environment is selected and expanded. The termination condition, however, is left as in the all solution search of GDDS to exclude local optimization. The resulting procedure performs *best-first search* like AO^* or GBF [Pearl 84], or it supports a *branch and bound* procedure [Ibaraki 77]. It seems to be rationally efficient for assumption-based reasoning that best-first search is employed to elicit the advantages of concurrent representation. In best-first search, context switching or backtracking happens not only when a context becomes contradictory, but also when there is a more preferred active context.

5. Application to Design Tasks

The context search algorithm described in the previous section must be used with a problem solver dependent on a problem domain. Here, the working of the GDDS 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, but practically, the problem in parametric design can be considered to be partially structured and constraint networks are not explicitly given. Because of this property, various kinds of ordering heuristics or network-based heuristics (e.g., [Mackworth 77]) are not readily available. Moreover, it takes a lot of time and space to execute each consumer involving an analysis or simulation task. This is why *dependency analysis* is useful for this kind of CSP. *Generate and test* (G&T) is the most simple technique, where the constraints are used to test the consistency of the assignments made by a generator. When the constraints can be applied to partial assignments, partial solution trees can be pruned by *hierarchical G&T*.

The GDDS 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 is an assignment of a value to a variable at a terminal. GDDS can assign values for variables like *constraint propagation*, through the consumer architecture. Note that these techniques can also be utilized by ADDB. The important advantage of GDDS is, however, that hierarchical G&T can be supplied to make the search more efficient. An assumption in an intermediate level can represent compiled knowledge or an abstraction of low-level parameters, so that a partial solution tree can be pruned by the constraints. This way of representation is reported to be very useful for CSPs in the independent research of [Mittal & Frayman 87]. It should be also noted that GDDS 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 models. A *design model* can be expressed by a set of constraints relating the desire, intention, specification given by the user forming a context, and can be represented by a hierarchical structure of an AND/OR tree, where its lower-level parameters can be attached below the model.

6. Related Work

6.1 Comparison with Other DDS Strategies

Much research on DDS has concentrated on controlling forward reasoning efficiently, so that redundant computing and rediscovering failures are avoided by *bottom-up* strategies. However, forward reasoning tends to generate consistent but irrelevant computation to solve the problem. GDDS eliminates this weak point by introducing *top-down* monitoring into bottom-up consumer architecture. The advantage of top-down strategies is that only the part relevant to the given goal is solved.

Although the idea of GDDS is based on a generalization of ADDB heuristics, GDDS improves the expressive power by extension to handle hierarchy[2] and reduces search by pruning upper-level nodes without utilizing hyperresolution, as stated earlier. There are some similarities between GDDS and the implied-by strategy [Forbus & de Kleer 88]: (1) assumptions are only created when needed and need not be control assumptions, (2) both use AND/OR tree search schemes allowing for some infinite domain, and

[2] While conditional control disjunctions are used to model the interactions between choices in [de Kleer & Williams 86], they are still handled within the framework of AND/OR trees with depth 2.

(3) consumers are executed only in the current focus environment. The differences are: (1) control by implied-by strategy is strongly domain-dependent, so that the user must specify how to switch contexts using contradiction consumers or some scoring mechanism, while GDDS selects the next context automatically by a generic controlling mechanism;[3] and (2) ATMoSphere's AO-nodes themselves represent environments, while GDDS schedules partial solution trees as environments, so that GDDS creates fewer nodes than ATMoSphere. As a result, if the user cannot specify how to switch contexts, search by ATMoSphere will be inefficient. As stated earlier, many problems that require dependency analysis cannot be given good ordering heuristics.

6.2 Upper and Lower Bounds

Informed search embedded in GDDS was introduced in section 4.2. However, even the concept of generalized ADDB heuristics without preference relations described in section 3 and 4.1 is very close to the notion of pruning trees in conventional search techniques in AI. [Stockman 79] regarded minimax game tree search as AND/OR tree search and proposed a procedure called *SSS**, and [Ibaraki 86] 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 partial solution tree 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 partial solution trees. Our GDDS is similar in this point. Each partial solution tree, T, in $OPEN$ may have its upper bound, $U(T)$, as:

$$U(T) = \bigwedge_{P \in Tip(T)} U_T(P), \quad \text{where } Tip(T) \text{ is the set of all tip nodes of } T, \text{ and}$$

$$U_T(P) = \begin{cases} \Gamma_P, & \text{if P is a terminal node and can be consistently assumed;} \\ \mathbf{F}, & \text{if P is contradictory (terminal or nonterminal);} \\ \mathbf{T}, & \text{elsewhere.} \end{cases}$$

SSS*'s strategy corresponds to the ADDB breadth-first heuristics. However, because two different Boolean values, say, Γ_{P_1} and Γ_{P_2}, cannot be compared, GDDS prefers the left-most active (i.e., $U(T) \neq 0$) partial solution tree. Moreover, SSS* generates all sons of a node at one time, while GDDS generates them one by one, so that it can handle the case where sons are infinitely many but are expected to be contradictory as a whole.

7. Conclusion

This paper introduced a general technique to control reasoning with DDS. The resulting search procedure, GDDS, models the incremental construction of hierarchical structure in assumption-based reasoning. 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 the TMS along their contexts incrementally rather than added to every possible world concurrently in a flat structure like the ATMS. 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.

[3] The scoring algorithm of ATMoSphere, however, corresponds to our informed search in section 4.2.

ACKNOWLEDGMENTS

I would like to thank Koichi Furukawa, Yasuo Nagai, Ryuzo Hasegawa, Yuichi Fujii and LANRS Group of ICOT for their useful comments and helpful discussions. I would also like to thank Professor Toshihide Ibaraki of Kyoto University for discussions on comparison with conventional search methods. Finally, I wish to express my thanks to Dr. Kazuhiro Fuchi, Director of ICOT Research Center, who provided me with insight into merging top-down and bottom-up search procedures and with the opportunity to conduct this research in the Fifth Generation Computer Systems Project.

REFERENCES

[de Kleer 86a] de Kleer, J., "An Assumption-based TMS", *Artificial Intelligence* **28** (1986), pp.127–162.

[de Kleer 86b] de Kleer, J., "Problem Solving with the ATMS", *Artificial Intelligence* **28** (1986), pp.197–224.

[de Kleer 88] de Kleer, J., "A General Labeling Algorithm for Assumption-based Truth maintenance", *Proc. AAAI-88* (1988), pp.188–192.

[de Kleer & Williams 86] de Kleer, J. and Williams, B. C., "Back to Backtracking: Controlling the ATMS", *Proc. AAAI-86* (1986), pp.910–917.

[Doyle 79] Doyle, J., "A Truth Maintenance System", *Artificial Intelligence* **12** (1986), pp.231–272.

[Forbus & de Kleer 88] Forbus, K. D. and de Kleer, J., "Focusing the ATMS", *Proc. AAAI-88* (1988). pp.193–198.

[Ibaraki 77] Ibaraki, T., "The Power of Dominance Relations in Branch and Bound Algorithms", *J. ACM* **24** (1977), pp.264–279.

[Ibaraki 86] Ibaraki, T., "Generalization of Alpha-Beta and SSS* Search Procedures", *Artificial Intelligence* **29** (1986), pp.73–117.

[Inoue 87] Inoue, K., "Pruning Search Trees in Assumption-based Reasoning", Technical Report TR-333, ICOT, 1987; also in *Proc. Avignon '88: The 8th International Workshop on Expert Systems & their Applications* (1988), pp.133–151.

[Inoue 88] Inoue, K., "Problem Solving with Hypothetical Reasoning", *Proc. FGCS '88: The 3rd International Conference on Fifth Generation Computer Systems* (1988), pp.1275–1281.

[Konolige 88] Konolige, K., "Hierarchic Autoepistemic Theories for Nonmonotonic Reasoning", *Proc. AAAI-88* (1988), pp.439–443.

[Mackworth 77] Mackworth, A. K., "Consistency in Networks of Relations", *Artificial Intelligence* **8** (1977), pp.99–118.

[Mittal & Frayman 87] Mittal, S. and Frayman F., "Making Partial Choices in Constraint Reasoning Problems", *Proc. AAAI-87* (1987), pp.631–636.

[Pearl 84] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA, 1984.

[Stockman 79] Stockman, G. C., "A Minimax Algorithm Better Than Alpha-Beta?", *Artificial Intelligence* **12** (1979), pp.179–196.

[Zabih *et al.* 87] Zabih, R., McAllester, D. and Chapman, D., "Non-Deterministic Lisp with Dependency-Directed Backtracking", *Proc. AAAI-87* (1987), pp.59–64.