UC Irvine
ICS Technical Reports

Title
Performance analysis of a message-oriented knowledge-base

Permalink
https://escholarship.org/uc/item/7w0054z7

Authors
Wong, Wang-chan
Suda, Tatsuya
Bic, Lubomir

Publication Date
1987-06-10

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ARCHIVES
Z
699
C3
no. 87-11

Performance Analysis of A Message-Oriented Knowledge-Base

Wang-chan Wong

Tatsuya Suda

Lubomir Bic

Technical Report #87-11

Dept. of Information and Computer Science

University of California, Irvine

Irvine, CA 92717

June 10, 1987

Contents

1. Introduction 2

2. Horn Knowledge-Based Systems and Resolution 3

3. The Message-Driven Knowledge-Base Model 5

3.1 The Extensional Database (EDB) 7

3.2 The Intensional Database (IDB) 9

3.3 Inference Engine – Message Propagation Scheme for SL-Resolution 10

4. Assumptions 10

5. Performance Analysis of the Uniprocessor Backtracking Scheme 12

5.1 Case 1: The Number of Correct Answers Is Known 14

5.2 Case 2: The Number of Correct Answers Is Unknown 17

6. Performance Analysis of Message-Propagation Scheme 18

6.1 Case 1: The Number of Correct Answers Is Known 19

6.2 Case 2: The Number of Correct Answers Is Unknown 23

7. Numerical Results 25

7.1 Speedup factor in terms of the t_u/t_f ratio 25

7.2 Irregular Trees, Varying Number of Answers 26

7.3 Best and Worst Cases Comparisons 28

7.4 The Structure of the Knowledge-Base: Bushy vs Skinny 28

7.5 Comparisons with Large Search Trees 30

8. Concluding Remarks 31

References 38

Abstract

First-order Horn logic is a useful formalism to design knowledge-based systems. When implemented on a sequential von Neumann computer, the main limitation of such systems is performance. We present a message-driven model for function-free Horn logic, where the knowledge base is represented as a network of logical processing elements communicating with one another exclusively through messages. The lack of centralized control and centralized memory makes this model suitable to implementation on a highly-parallel asynchronous computer architecture.

The primary contribution of this paper is a performance analysis of this message-driven system and a comparison with a sequential resolution scheme using backtracking. For both approaches, closed form expressions for the performance results are derived and compared.

Index Terms – non von Neumann computing, deductive knowledge base, parallel processing, message-driven computation, performance analysis.

1. Introduction

There is a number of possible ways to represent and process knowledge. One formalism that has attracted much attention by the research community in recent years is first-order predicate logic. When limited to Horn clauses, i.e., clauses with only one positive literal, search strategies can be devised to make the systems suitable to automatic processing by electronic computers. By far the most popular search strategy is to scan the clauses from left to right and from top to bottom. Whenever a goal fails, the search backtracks to the immediately preceding goal and tries to find an alternate solution. This results in a depth-first search of the solution space. PROLOG is the most prominent representative of a programming language based on this strategy termed SL-resolution.

The main limitation of this approach, especially in the realm of knowledge base applications where significant numbers of unit clauses (facts) must be examined, is its speed of execution [Kow79]. In an attempt to alleviate this problem, we have developed a model for parallel processing of a subset of Horn clause logic [BiLe87]. This subset is restricted to function-free binary predicates, which are generally suitable to the development of a knowledge base. The model is based on the principles of asynchronous message-driven computation found in dataflow systems [Com82, TBH82] or actor-based models [Agha85]. The fundamental idea underlying the model is to view the knowledge base not as a passive data structure stored and manipulated in memory but as a network of *active* nodes. Each node represents a single constant (object) of the knowledge base and is capable of communicating with other nodes by exchanging messages via the network links. This network will be called the extentional database.

To extract information from such a knowledge base, each query is viewed as a template for which a match must be found in the knowledge base network. This template is formed by searching the rules of the knowledge base; these will be termed the intentional database. The search for possible matches of a given query is performed by messages carrying the query template. These are injected into selected nodes and propagated through the network by being passed from one node to another. The message propagation does not require any centralized control; each message carries all the necessary information for a node to determine if and where to propagate it. This lack of centralized control permits the model to be implemented on a computer architecture consisting of a very large number of independent processing elements. Potentially, as many physical elements could be used as there are nodes in the knowledge base network.

The primary objective of the present paper is to model and analyze the performance of the proposed message-oriented model. To have a reference point for comparisons, we also model and analyze the conventional Horn database with backtracking. For both models, if the search trees are regular, closed form expressions for the performance results can be derived.

The organization of this paper is as follows. In section 2, we describe the conventional Horn database and explain how information is retrieved. In section 3, we present our deductive message-driven model. Assumptions used in the analysis are described in section 4. In section 5, we analyze the performance of the Horn knowledge-base system that uses backtracking. The performance of our message-driven model is analyzed in section 6. In section 7, the numerical results are shown, and performance of our deductive model is compared with that of the Horn knowledge-base with a backtracking mechanism.

2. Horn Knowledge-Based Systems and Resolution

In this section, we describe a knowledge-based system that is based on first-order function-free Horn logic and show how resolution is applied to retrieve answers from such a system.

A *Horn Knowledge-based system* is a collection of *clauses*, each taking the form of:

$$\neg p_0 \vee \neg p_1 \vee ... \vee \neg p_{n-1} \vee p_n$$

Each p_i is called a *literal* and has the form $p_i(t_1, ..., t_m)$, where p_i is a *predicate* symbol and $t_1, ..., t_m$ are *terms*. Terms may be constants or variables. The literal p_n, which is the only positive literal in the clause, is called the *head* of the clause; the remaining literals form its *body*. A clause with an empty body is called an *assertion* and is used to represent explicit facts. Clauses with a non-empty body are called *rules*. The head of each rule defines a *relation*; the predicate gives the relation name; terms are interpreted as attributes.

Relations defined by assertions are called *base* relations, while relations defined as the head of a rule will be referred to as *virtual* relations. For example, a predicate $supervisor(alex, bill)$ is a base relation which can be interpreted as *"alex is the supervisor of bill"*. With this base relation *supervisor*, we can define a virtual relation *boss* as $\{\neg supervisor(X, Y) \vee boss(X, Y)\}$, which means that every supervisor is also a boss.

A *query* or *goal* is viewed as a theorem which can be proven or disproven. In order to prove a theorem (i.e. a query), we first negate the theorem and then prove it "unsatisfiable" by deriving an empty clause. In other words, by disproving the negated theorem, we prove the original theorem. The renowned Resolution Principle proposed by Robinson [ROB79] is a procedure that makes this proof technique mechanical. The following example illustrates how this procedure works. Suppose we have the following two propositional clauses:

(1) P

(2) $\neg P \vee Q$

To see if Q is true with these clauses, we first negate the Q into $\neg Q$ and see if we can derive an empty clause. Clauses (1) and (2) can be combined because clause (1) has a P and clause (2) has $\neg P$. Consequently, we obtain a new clause with a single proposition Q which is called a *resolvent* of clauses (1) and (2). By combining this resolvent Q with the negated proposition $\neg Q$, we derive an empty clause. We then conclude that Q is true.

The resolution principle is also applicable to predicate logic by using the *unification* process. The unification process is a pattern matching procedure that finds the most general substitutions among the terms of predicates in order to make the predicates resolvable. For example, literals $P(X)$ and $\neg P(Y)$ can be resolved by the substitution $\{X/Y\}$.

Note that the resolution process neither specifies any order nor imposes any restriction on resolving clauses. Thus, it is possible to generate redundant and irrelevant resolvents. The combinatorial explosion of generating these redundant and irrelevant resolvents makes the proof technique difficult to program. To solve this problem, a number of refinements of the resolution process have been proposed. One of these refinements is the *binary resolution* [ChLe73] which specifies that, at any time, two clauses can be combined by "resolving" a literal in each of these two clauses only.

Linear input resolution [ChLe73], a special case of *binary resolution*, is another example of refinement of the resolution process. It requires that, during each resolution step, at least one of the two clauses being resolved must come from the original input set of clauses, which comprises all the inference rules and data. In a linear input resolution, since any resolvent may contain a number of literals, the way to choose which one of these literals as the next goal to be resolved may affect the efficiency of the resolution process.

SL-resolution (linear resolution with selection function) [KoKu71] is a special case of linear input resolution, in which the resolution process specifies a computation rule to select the next subgoal from a resolvent. If the computation rule is to select the leftmost literal first, the procedure is called left-to-right SL-resolution.

Consider the Horn knowledge-base in Example 1. If query $boss(alex, ?)$ is asked, the SL-resolution method will generate a proof tree as shown in Figure 1. In this proof tree, the root is the negated goal $(\neg boss(alex, ?))$, and the numbered edges indicate the corresponding unifiable clauses of Example 1. Starting from the root and evaluating clauses from left to right, there exist four different paths. Two of them lead to failures because there is no match for the current subgoal. The other two lead to an answer (i.e. an empty clause is derived).

Algorithm A summarizes the procedure to retrieve data from the proof tree.

1) $supervisor(alex, bill)$.

2) $supervisor(bill, charles)$.

3) $\neg supervisor(X, A) \vee \neg boss(A, Y) \vee boss(X, Y)$.

4) $\neg supervisor(X, Y) \vee boss(X, Y)$.

5) *negated query* $\neg boss(alex, ?)$.

Example of a Horn Knowledge-Base

Example 1

(1) Negate query as the goal G to be resolved.
(2) Scan through the input clauses;
  if an unused input clause R matches goal G;
  then mark the R as used; (goto step (3);)
  else **if** G is the root
    then terminate the process
    else fail the goal G;
      goto step (2); (the backtracking step)
(3) Resolve the R and the clause containing G as immediate goal;
  if an empty clause is derived
  then succeed and return bindings
    fail the last subgoal; (in order to collect the remaining answers)
    goto step (2); (the backtracking step)
  else the residual literals of R are placed at the left-most
    position of the resolvent;
(4) Set goal G equal to the left-most literal of the current resolvent;
  Re-initialize all input clauses that are marked in step (2);
  Goto step (2);

The Depth-First SL-resolution for Information Retrieval
Algorithm A

3. The Message-Driven Knowledge-Base Model

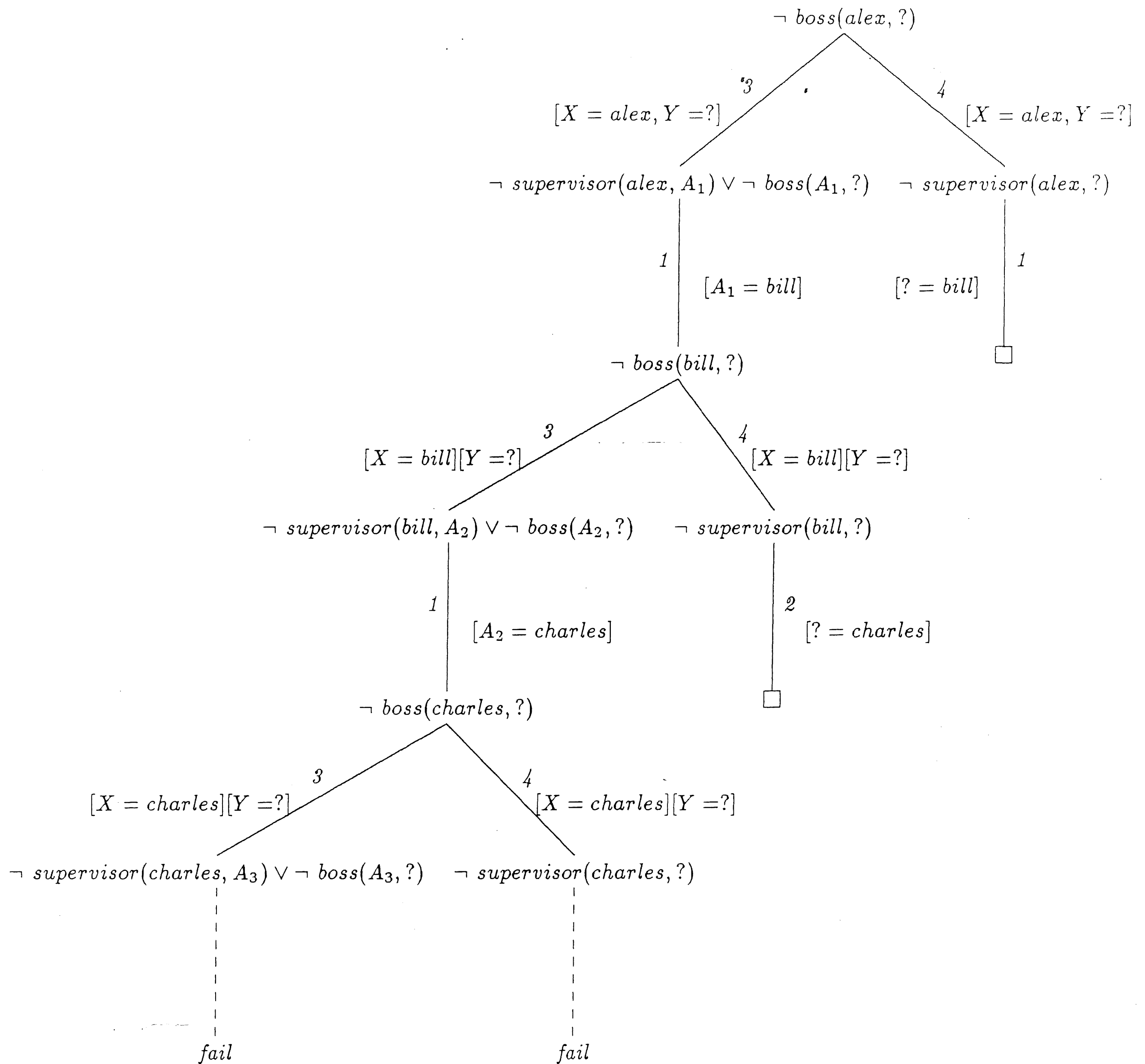

Figure 1
Example of a Preorder SL-resolution Proof Tree

Our message-driven knowledge-base model follows the same conventions as the first-order function-free Horn knowledge-base described in section 2 but is capable of exploiting parallel processing. The overall structure of our system is depicted in Figure 2. Queries from the users are processed by the query processors at Layer 3. These query processors have complete knowledge of the Layer 2 knowledge-based model that comprises two major components. The first component is a model of knowledge representation that describes the facts and rules of the knowledge-base.

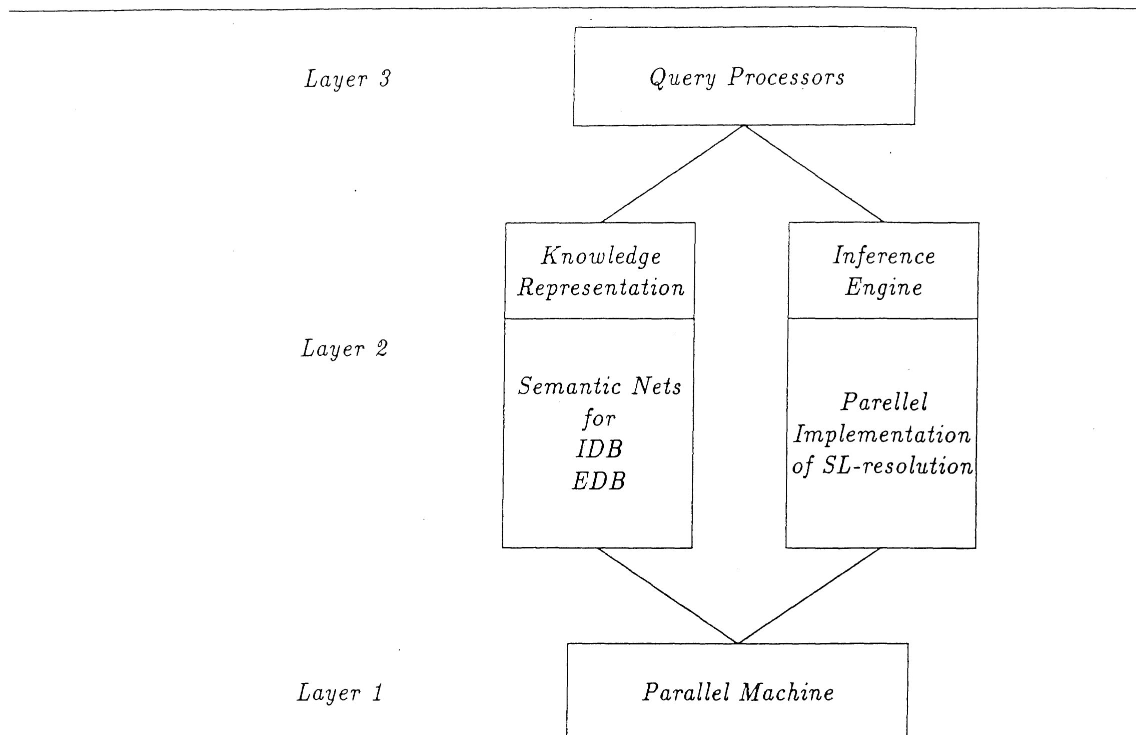

Figure 2
Overall Structure of The Message-Driven Knowledge-Base Model

The facts and rules are represented by two logically separated sets of graphs. One set contains all assertions and is called the Extensional Database (EDB). The other set comprises all inference rules and is called the Intensional Database (IDB). The second component at Layer 2 is the inference engine that implements a parallel SL-resolution by means of message propagation. Layer 1 is a parallel machine (actual hardware), which the IDB and EDB are mapped onto. The machine may contain a large number of processing elements (PEs) without any central controller. These PEs can communicate with one another asynchronously via messages. There exist many possible instances of such a machine. The design of the actual architecture suitable of our model is beyond the scope of this paper. Here, we are only interested in Layer 2. Details of this layer will be discussed in the following sections.

3.1. The Extensional Database (EDB)

The Extensional Database (EDB) is a graph with labeled arcs. Each node of the graph represents a "set" or "entity". The arcs are relations between sets. Figure 3 shows an example. The node *employee* is a set node which defines the set of all employees. Individual elements of the set are connected to the set node via the "*instance*" arcs (depicted as *inst* in Figure 3). The labeled arc *belong* connects the sets *employee* and *dept*. The direction of the arc shows the asymmetry of the relation (e.g. *employee* belongs to *dept*, but not vice versa). Moreover, the labeled

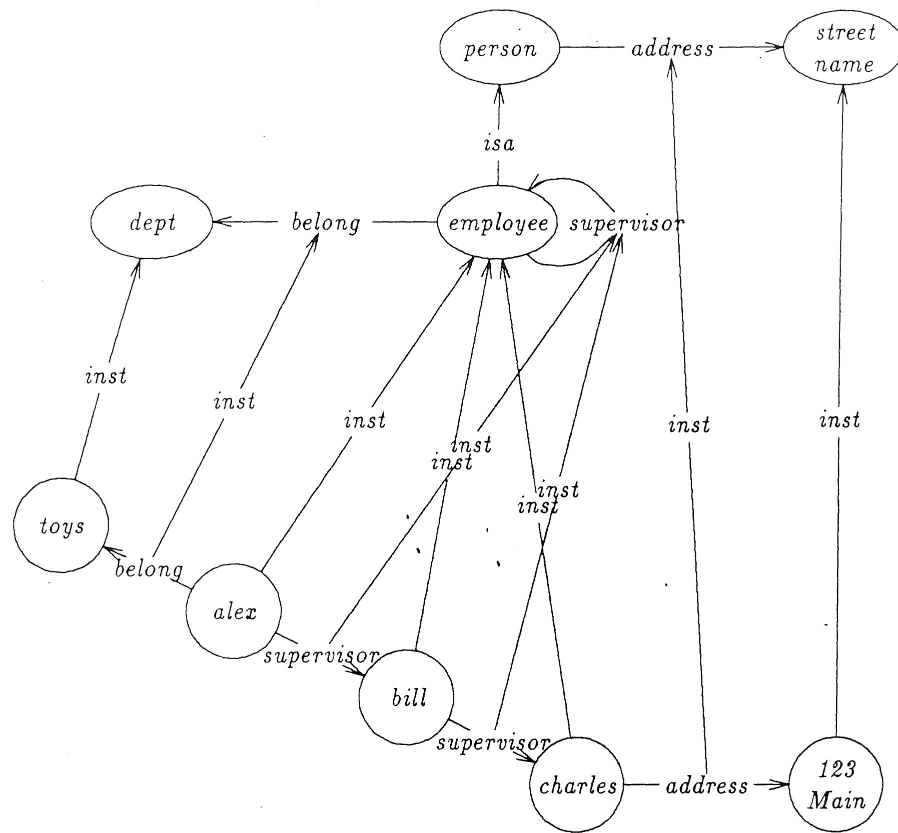

Figure 3
An Example of an EDB

arc between two set nodes is also a set which contains many elements of same type of relation (one may implement the labeled arcs as nodes of the graph). For example, there exist several instances of the relation between the *employee* and the *dept.*

There is a special type of arc called "*isa*" arc that connects two set nodes. The *isa* arc helps to form an inheritance hierarchy of the set nodes. For instance, the node *employee* is a *person*. Therefore, an employee should also have an address. The *isa* arc can further be applied to the labeled arcs such that relations also form a hierarchy.

Our data model can be easily described by the function-free binary logic in which the predicate names are arcs (relations) and terms are set nodes. For example, in the predicate *belong(employee, dept)*, the relation *belong* is the predicate name and the sets *employee* and *dept* are the terms. Instances of a relation are also represented by binary predicates. For example, *belong(alex,toys)* is an instance of the relation *belong(employee, dept)*. We assume only binary predicates in our data model. There are three reasons to use binary predicates rather than n-ary predicates. First, an n-ary predicate can always be represented by $(n + 1)$ binary predicates [DeKo79]. Second, we can represent binary assertions easily as a

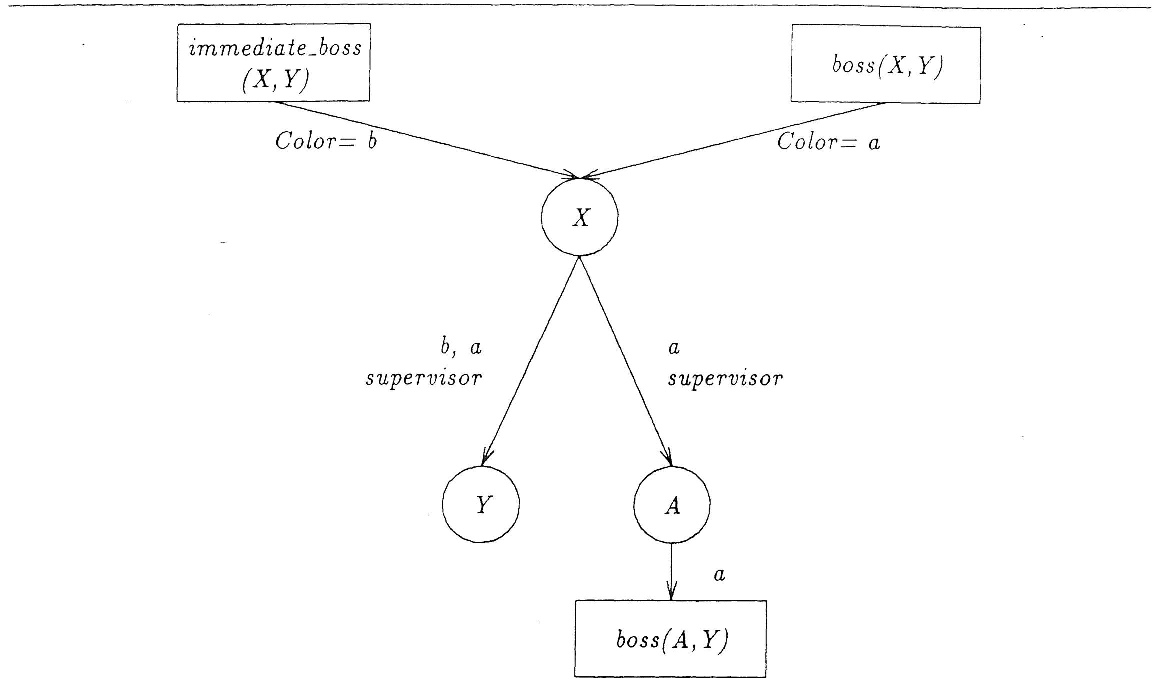

Figure 4
An Example of an IDB

connected graph in the EDB. Third, we can represent "missing" data better than with an n-ary predicate.

3.2. The Intensional Database (IDB)

The intensional database (IDB) consists of all the inference rules, i.e. the rules whose heads are virtual relations. For instance, there are two rules defining the virtual relation *boss* in Example 1. Suppose we have a rule to define another virtual relation *immediate_boss* as follows:

$$\neg supervisor(X, Y) \vee immediate_boss(X, Y)$$

A portion of the IDB containing the virtual relations *boss* and *immediate_boss* is shown in Figure 4. The rectangles of the graph represent the virtual relations. The definitions of these virtual relations (e.g. the bodies of the rules) form a tree in which nodes are terms and edges are predicate names. The tree is then connected to the corresponding virtual relations. For example, a tree is formed for the rules that define relation *boss(X,Y)* and is attached to the rectangular node *boss(X,Y)*. Since the rules in the IDB will share the structures of some base relations (for instance, the virtual relations *boss* and *immediate_boss* share the base relation *supervisor*), we need to separate them from tangling with one another. In order to do so, each virtual relation is associated with a *color*. For example, the virtual relation *immediate_boss* has color *b*. If a message is injected into the node

immediate_boss(X, Y), only those arcs marked with color b will allow the message to go through.

3.3. Inference Engine – Message Propagation Scheme for SL-Resolution

To process queries against the EDB, *search* messages containing the query are injected into specific nodes of the EDB network. From each injection point, the *search* messages replicate asynchronously into many directions in search of answers. If a *search* message refers to a rule in the IDB, a *collect* message is generated and injected into the IDB to collect the definitions of that particular rule. After the content of the requesting *search* message is updated with these definitions, propagation is resumed in the EDB. Thus, it allows dereferencing. Moreover, the graph representation of the IDB allows *rule sharing*, since *collect* messages can be injected into the IDB concurrently.

The way that messages are being propagated in both IDB and EDB corresponds to the left-to-right SL-resolution described in section 2. To be specific, we are implementing a parallel left-to-right SL-resolution by means of message propagation. Consider the proof tree in Figure 1. In our model, a *collect* message is injected into the IDB as shown in Figure 4. Two separate *search* messages are generated. One holds the definition path $(\neg supervisor(alex, ?))$; the other contains the definition path $(\neg\ supervisor(alex, A_1) \vee boss(A_1, ?))$. These two messages are then injected to the node *alex* in the EDB. To resolve the literals $(\neg supervisor(alex, ?))$ and $(\neg supervisor(alex, A_1))$ in a conventional manner, the resolution process will search for assertions with predicate name *supervisor* in the knowledge-base. In our model, to search for such assertions in the EDB is equivalent to see if there are any arcs labeled as *supervisor* leaving from the node *alex*. Since an arc *supervisor* connects the node *alex* to the node *bill* (as shown in Figure 3), the messages are propagated to the node *bill*, and both variables '?' and A_1 in the definition paths of these two messages are bound to *bill*. The way that message propagation is done is equivalent to the unification process but the search and the bindings of '?' and A_1 are done in parallel. After receiving the messages, the node *bill* further propagates them separately according to the remaining definition paths until either the definition paths are completed successfully or they are terminated with failure. Since the messages are propagated independently and not related to one another, the way to propagate messages in our model is equivalent to search the proof tree in parallel.

4. Assumptions

We have so far described the conventional Horn knowledge-base model (section 2) and our message-driven knowledge-base model (section 3). We will analyze their performance in sections 5 and 6. In this section, we first state the assumptions made in our analysis.

The performance measure to be obtained is the time to complete the search for all correct answers. The analysis can further be divided into two cases depending

on our knowledge of the knowledge-base. If we have no knowledge of the number of correct answers in the knowledge-base, the trees have to be searched *exhaustively*. This is very much alike the *worst* case analysis in tree search algorithms. We also analyze the case, in which the number of correct answers is known. For this case, as soon as all the expected answers are retrieved, the search is terminated. We assume that, once the answers are found, there are some other mechanisms that will convey the answers to the users, and hence, the time to report answers for both cases is not included.

We further make the following assumptions for both the backtracking and the message-driven models:

(1) All data reside in main memory. It is observed that the declining costs of memory will make it feasible to store the entire knowledge-base in main memory. Hence, the conventional I/O bottleneck between processor and secondary memory will disappear. Therefore, data retrieval from secondary storage is not considered in this study.

(2) Unless otherwise stated, the search trees are irregular and may have multiple correct answers.

(3) We have complete knowledge of the search trees such as the connectivity of the tree.

(4) To make our message propagation scheme and the backtracking scheme compatible, we extend the depth of the search tree in order to account for the fact that we use three nodes to represent one binary predicate. For example, if we implement the labeled arc in our message-driven model as a node, a binary predicate $p(a, b)$ is represented as a path of $a - p - b$, which has a length of 2. Therefore, if a proof tree in the backtracking scheme has a depth of w, the corresponding message-driven search tree should have a depth of $2(w + 1)$. (A tree with one single node has depth of zero.)

(5) We assume all queries will terminate eventually. In other words, we assume that there are algorithms to eliminate left-recursion and terminate repeated goals [WB86, WB87A].

In the analysis of our message-driven model we further assume that:

(6) There are sufficient numbers of processing elements so that messages will not form a queue at each PE of the underlying architecture.

(7) All PEs are connected through a point-to-point connection network (but it is not a fully-connected topology; a message may have to go through several PE's to reach its destination PE). Moreover, sending copies of a message is done sequentially. For example, when a sending node PE_i sends messages to its immediate neighbors PE_j and PE_k, it replicates and then sends the message to one of these neighbors, say, PE_j. Once this is done, another copy of the message is sent to PE_k in the same manner.

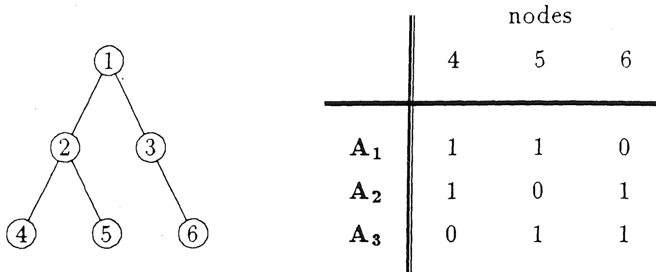

Figure 5
An Example Search Tree and Its Assignments

(8) As described in section 3.2, the rules of the IDB are represented by simple OR trees. All search trees generated by these rules are purely OR trees and each node is distinct in the tree.

5. Performance Analysis of the Uniprocessor Backtracking Scheme

In this section, we shall look at the performance of the backtracking scheme implemented in a uniprocessor environment. The following notations are defined for the subsequent analysis:

Unification Times	Edges	Backtracking Times	Edges
t_u^1	1-2	t_b^1	4-2
t_u^2	2-4	t_b^2	5-2
t_u^3	2-5	t_b^3	2-1
t_u^4	1-3	t_b^4	6-3
t_u^5	3-6	t_b^5	3-1

Table 1
Unification and Backtracking Times Among Nodes in Figure 5

(1) The unification time includes the time to search for unifiable clauses and to bind terms of predicates of the matched goal. The time required for a goal to unify with another goal may be different from node to node. If the tree

is traversed in a preorder manner, it is possible to enumerate the individual unification time. t_u^i denotes the *i-th* unification time (by preorder traversal) of the given proof tree. To illustrate, for the proof tree in Figure 5, the enumerations of the unification time are shown in Table 1.

(2) The backtracking time includes the time to undo the bindings of the failed goal and backtrack to its parent. Similar to the unification time, backtracking times may be different from node to node. t_b^i denotes the *i-th* backtracking time for a given proof tree.[1] For the example in Figure 5, its backtracking times are also shown in Table 1. These backtracking times are enumerated in the order as if we are carrying out the unification process. For instance, if node 5 is the answer, we have to first traverse to nodes 2 and 4. Since node 4 is not the answer, we backtrack to node 2 before arriving at node 5. Hence, the time required to backtrack from node 4 to node 2 is the first enumerated backtracking time of the tree in Figure 5. The rest of the backtracking times in Table 1 are enumerated in this manner.

(3) l is the total number of leaves of the proof tree. c is the number of correct answers in the tree. A binary variable a^i is associated with the ith leaf. a^i equals 1 if a correct answer is assigned to the ith leaf; otherwise, a^i equals 0. Vector $\mathbf{A} = [a^1,\ a^2,\ ...,\ a^l]$ represents the possible assignments of correct answers among leaves. Since there are l leaves and c correct answers, we have $n = \binom{l}{c}$ number of possible assignments. To distinguish them, we use $\mathbf{A_j}$ to refer to the jth assignment ($j = 1, 2, ..., n$). Accordingly, a_j^i refers to the value of the binary variable assigned to the ith leaf of the jth assignment. For an assingment $\mathbf{A_j} = [a_j^1, ..a_j^k, ...a_j^l]$, the position of the leaf to which the last correct answer is assigned is denoted by p_j. ($p_j = k$, if $a_j^k = 1$ and $a_j^h = 0$ for $k < h \leq l$.)

For example, the tree in Figure 5 has three leaves 4, 5 and 6 ($l = 3$). If there exists two correct answers ($c = 2$), the total number of possible assignments $n = \binom{3}{2} = 3$. These assignments are:

$$\mathbf{A_1} = [a_1^1,\ a_1^2,\ a_1^3] = [1,\ 1,\ 0]$$
$$\mathbf{A_2} = [a_2^1,\ a_2^2,\ a_2^3] = [1,\ 0,\ 1]$$
$$\mathbf{A_3} = [a_3^1,\ a_3^2,\ a_3^3] = [0,\ 1,\ 1]$$

These assignments are also shown in Figure 5.

Furthermore, values of p_j for $j = 1, 2, 3$ are equal to 2, 3, 3, respectively.

[1] The depth-first left-to-right search of the proof tree naturally suggests a stack implementation. In fact, most of the SL-resolution implementations such as Prolog utilize stacks to make the backtracking more efficient. If we adopt the depth-first left-to-right search, t_u^i is the time to manipulate these stacks and the time to search for unifiable rules. t_b^i is the time to backtrack, which involves simple operations on these stacks.

(4) D_i, or *preorder distance* between the root and the ith leaf, is the enumeration of nodes traversed in a *preorder* manner from the root to the ith leaf. d_i is the *direct distance* between the root and the ith leaf. For example, in Figure 5, D_2, the preorder distance from the root to the second leaf (i.e. node 5), is equal to 3 since it has to traverse from node 1 to nodes 2 and 4 before it arrives at node 5. On the other hand, d_2, the direct distance from the root to the second leaf (i.e. node 5), is equal to 2 since the length of direct path from node 1 to node 5 (i.e. $1-2-5$) is 2.

(5) If the tree is regular, let b be the branching factor and w be the depth of the tree. The total number of nodes v is, then, equal to $(b^{w+1}-1)/(b-1)$.

5.1. Case 1: The Number of Correct Answers Is Known

In this section, we assume the number of correct answers is known and examine how well the backtracking model behaves. The major characteristic of this case lies in the search termination condition: whenever the foreknown number of answers are retrieved, the search is terminated. Therefore, the tree does not have to be completely searched. Since we have knowledge on both the proof tree and the number of correct answers, the possible assignments of these answers to the leaves are known. With these assignments, we can further obtain the average and best completion times.

(I) Average Case

Let t_{avg} be the average completion time to search for all answers from the root. t_{avg} is given by:

$$t_{avg} = \sum_{j=1}^{n} P(\mathbf{A_j}) \times \{ \sum_{i=1}^{D_{p_j}|\mathbf{A_j}} t_u^i + \sum_{i=1}^{D_{p_j}|\mathbf{A_j}-d_{p_j}|\mathbf{A_j}} t_b^i \} \qquad (5.1.1)$$

where $P(\mathbf{A_j})$ is the probability of assignment $\mathbf{A_j}$ being selected, $D_{p_j}|\mathbf{A_j}$ is the preorder distance from the root to the p_j-*th* leaf to which the rightmost correct answer in $\mathbf{A_j}$ is assigned, and $d_{p_j}|\mathbf{A_j}$ is the direct distance from the root to the p_j-*th* leaf.

Eq. 5.1.1 is derived in the following manner. Since the rightmost correct answer is assigned to the p_j-*th* leaf in assignment $\mathbf{A_j}$, D_{p_j} equals the number of attempts to unify goals. Therefore, the term $\sum_{i=1}^{D_{p_j}|\mathbf{A_j}} t_u^i$ is a sum of the individual unification times starting from the root to the p_j-*th* leaf. In order to arrive at the p_j-*th* leaf, there are a total of $[D_{p_j} - d_{p_j}]$ backtracks. Similarly, the term $\sum_{i=1}^{D_{p_j}|\mathbf{A_j}-d_{p_j}|\mathbf{A_j}} t_b^i$ adds up all the individual backtracking times prior to arriving at p_j-*th* leaf. These two terms compose the completion time of assignment $\mathbf{A_j}$.

If the values of t_u^i and t_b^i are constant and are equal to t_u and t_b, respectively, *e.q.* 5.1.1 takes a much simpler form. In this case, the term $\sum_{i=1}^{D_{p_j}|\mathbf{A_j}} t_u^i$ becomes

$(D_{p_j}|\mathbf{A_j}) \times t_u$ and the term $\sum_{i=1}^{D_{p_j}|\mathbf{A_j}-d_{p_j}|\mathbf{A_j}} t_b^i$ becomes $(D_{p_j}|\mathbf{A_j} - d_{p_j}|\mathbf{A_j}) \times t_b$. Since there are n possible assignments, the probability $P(\mathbf{A_j})$ of an assignment being chosen is equal to $\frac{1}{n}$. Hence, the average completion time is equal to the total of the completion time for each assignment multiplied by its probability of being chosen. For constant t_u and t_b, *Eq.* 5.1.1 becomes:

$$t_{avg} = \frac{1}{n}\sum_{j=1}^{n}[(D_{p_j}|\mathbf{A_j}) \times t_u + (D_{p_j}|\mathbf{A_j} - d_{p_j}|\mathbf{A_j}) \times t_b] \qquad (5.1.2)$$

Let us illustrate *eq.* 5.1.2 with the example in Figure 5. If there exists two correct answers among the three leaves, there exist six possible assignments of answers to leaves. These different assignments are shown in Figure 5. In the first assignment $\mathbf{A_1}$, the rightmost correct answer is assigned to the second leaf (i.e. $p = 2$). In the second and third assignments ($\mathbf{A_2}$ and $\mathbf{A_3}$), the rightmost correct answer is assigned to the third leaf (i.e. $p = 3$). Hence, we have:

for assignment $\mathbf{A_1}$, the completion time $= [3t_u + (3-2)t_b]$
for assignment $\mathbf{A_2}$, the completion time $= [5t_u + (5-2)t_b]$
for assignment $\mathbf{A_3}$, the completion time $= [5t_u + (5-2)t_b]$

Moreover, each of these three assignments has the same probability of $\frac{1}{3}$ to be chosen. Therefore, we have:

$$t_{avg} = \frac{1}{3}\{[3t_u + t_b] + [5t_u + 3t_b] + [5t_u + 3t_b]\} = \frac{1}{3}[13t_u + 7t_b]$$

If the tree is a regular tree with depth w, branching factor b and with a single answer, a closed form solution is derived as follows. In this case, the number of leaves is equal to the number of possible assignments (e.g. $n = b^w$). The locations of the correct answer in the assignments $\mathbf{A_1}, \mathbf{A_2}, ..., \mathbf{A_n}$ are equal to the $1^{th}, 2^{nd}, ..., n^{th}$ leaves. The direct distances from the root to each of these leaves are equal to the depth (w) of the tree. Therefore, the total direct distances of all assignments is equal to:

$$\sum_{j=1}^{n} d_{p_j}|\mathbf{A_j} = n \times w$$

Since the correct answers are located at the $1^{th}, 2^{nd}, ..., n^{th}$ leaves of assignments $\mathbf{A_1}, \mathbf{A_2}, ..., \mathbf{A_n}$, respectively, the average preorder distance from the root to these leaves is equal to the average of the preorder distances of the first and last leaves. By definition, the preorder distance of the first leaf is equal to the depth of the tree, and the preorder distance of the last leaf is $(v-1)$. Therefore, we obtain the

average preorder distance of all assignments as follows:

$$\frac{1}{n}\sum_{j=1}^{n} D_{p_j}|\mathbf{A_j} = \frac{w+(v-1)}{2}$$

Hence, for the regular tree, we have, from *eq* 5.1.2:

$$\begin{aligned}
t_{avg} &= \frac{1}{n}\{\sum_{j=1}^{n}[(D_{p_j}|\mathbf{A_j}) \times t_u + (D_{p_j}|\mathbf{A_j} - d_{p_j}|\mathbf{A_j}) \times t_b]\} \\
&= \frac{1}{n}\{(\sum_{j=1}^{n}[(D_{p_j}|\mathbf{A_j}) \times (t_u + t_b)]) - (\sum_{i=1}^{n}(d_{p_j}|\mathbf{A_j}) \times t_b)\} \\
&= (t_u + t_b) \times \frac{1}{n}\sum_{j=1}^{n}(D_{p_j}|\mathbf{A_j}) - \frac{t_b}{n} \times (\sum_{i=1}^{n}(d_{p_j}|\mathbf{A_j})) \\
&= (t_u + t_b) \times \frac{w+(v-1)}{2} - \frac{t_b}{n} \times n \times w \\
&= \frac{1}{2}[(w+v-1) \times t_u + (v-1) \times t_b]
\end{aligned} \tag{5.1.3}$$

(II) Best Case

The best case is the shortest completion time to search from the root to the pth node (i.e. the rightmost correct answer) among all possible assignments (i.e. $\mathbf{A_j}$'s). From the similar argument used to derived *eq.* 5.1.1, the shortest completion time t_{best} becomes:

$$t_{best} = \min_{1 \le j \le n}\{\sum_{i=1}^{D_{p_j}|\mathbf{A_j}} t_u^i + \sum_{i=1}^{D_{p_j}|\mathbf{A_j} - d_{p_j}|\mathbf{A_j}} t_b^i\} \tag{5.1.4}$$

If the unification and backtracking times are constants (i.e. equal to t_u and t_b, respectively), *eq.* 5.1.4 is simplified to:

$$t_{best} = \min_{1 \le j \le n}\{(D_{p_j}|\mathbf{A_j}) \times t_u + (D_{p_j}|\mathbf{A_j} - d_{p_j}|\mathbf{A_j}) \times t_b\} \tag{5.1.5}$$

Consider the example in Figure 5. From the table, we obtain the values of p_j's for assignments $\mathbf{A_j}$ (j=1,...,n). The values of D_{p_j} and d_{p_j} are then computed accordingly. From *eq.* 5.1.5, the best time becomes:

$$t_{best} = min\{3t_u + (3-2)t_b,\ 5t_u + (5-2)t_b,\ 5t_u + (5-2)t_b\} = 3t_u + t_b$$

If the tree is regular with one single answer and with depth w, the best case occurs when the answer is at the leftmost leaf. At each level of the tree, the search

process finds the correct branch (the leftmost branch) immediately. There is no need to backtrack at all. Therefore, *eq.* 5.1.5 is reduced to:

$$t_{best} = D_1 \times t_u = w \times t_u \tag{5.1.6}$$

where the D_1 denotes the direct distance to the leftmost leaf of the tree and is equal to w.

5.2. Case 2: The Number of Correct Answers Is Unknown

If we do not have any idea of how many answers are in the knowledge-base, we would have no choice but to search the whole proof tree. In this case, it is equivalent to analyze the worst case to search the proof tree. Let t_{worst} be the completion time to search the complete proof tree. We obtain:

$$t_{worst} = \sum_{i=1}^{D_r} t_u^i + \sum_{i=1}^{D_r - d_r} t_b^i \tag{5.2.1}$$

where D_r and d_r are the preorder distance and direct distance from the root to the rightmost leaf, respectively.

Eq. 5.2.1 is derived in the following manner. To search the complete proof tree, it will take all the unification times from the root up to the rightmost leaf ($\sum_{i=1}^{D_r} t_u^i$). It also has to backtrack $\sum_{i=1}^{D_r - d_r} t_b^i$ units of time before it reaches the rightmost leaf.

If the unification and backtracking times are constants, we reduce *Eq.* 5.2.1 to:

$$t_{worst} = D_r \times t_u + (D_r - d_r) \times t_b \tag{5.2.2}$$

For the example in Figure 5, the worst time becomes:

$$t_{worst} = 5t_u + (5-2)t_b = 5t_u + 3t_b$$

If a tree is regular with single answer, *eq.* 5.2.2 is reduced to:

$$t_{worst} = (v-1) \times t_u + (v - 1 - w) \times t_b \tag{5.2.3}$$

where the preorder distance D_r equals $(v-1)$, and the direct distance d_r equals w.

	nodes 2	3
$\mathbf{R}_1^1$	1	0
$\mathbf{R}_2^1$	1	1
$\mathbf{R}_3^1$	1	1

(a)

	nodes 4	5
$\mathbf{R}_1^2$	1	1
$\mathbf{R}_2^2$	1	0
$\mathbf{R}_3^2$	0	1

(b)

	node 6
$\mathbf{R}_1^3$	0
$\mathbf{R}_2^3$	1
$\mathbf{R}_3^3$	1

(c)

Table 2

Assignment Tables for The Tree in Figure 5

6. Performance Analysis of Message-Propagation Scheme

In this section, we show the performance of our model with message propagation described in section 3. We first define the following notations:

(1) As in section 5, notations l, c, n denote the total number of leaves, number of correct answers and total number of possible assignments of the answers in the leaves, respectively. $\{\mathbf{A}_1, ..., \mathbf{A}_j, ..., \mathbf{A}_n\}$ is the set of all possible assignments of correct answers in leaves. If the tree is regular, b,w and v denote the branching factor, depth and total number of nodes of the tree, respectively.

(2) Starting from the root, an ID number is assigned to each node by enumerating nodes from left to right on each level of the tree;

(3) b^i is the branching factor of node i;

(4) Vector $\mathbf{S^i} = \{S_1^i, S_2^i, ..., S_{b^i}^i\}$ denotes the ID numbers of the children of node i. For example, in Figure 5, $\mathbf{S^1}$ is $\{2,\ 3\}$ and $\mathbf{S^3}$ is $\{6\}$.

(5) t_f^i is the fabrication time for node i; the time it takes to receive and process an incoming message, plus the time to fabricate new outgoing messages.

(6) $t_p^{s_h^i}$ is the communication delay from node i to its h-th child; t_p includes the start-up time to communicate and the propagation delay for a message to travel between PE's;

(7) For a given assignment $\mathbf{A_j}$, a vector $\mathbf{R_j^i} = [R_j^i(1), ..., R_j^i(h), ..., R_j^i(b_i)]$ is associated with node i. An element $R_j^i(h)$ is a binary variable and is equal to 1, if the subtree rooted at the hth child of node i contains at least one leaf with a correct answer; otherwise, $R_j^i(h)$ is equal to 0.

For example, consider the tree in Figure 5. For assignment $\mathbf{A}_1$, $\mathbf{R}_1^1$ equals $[1,0]$, since the subtree rooted from the first child of node 1 (i.e.

node 2) contains two correct answers and the second child of node 1 (i.e. node 3) contains no correct answer in its subtree. Similarly, $\mathbf{R}_2^1$ and $\mathbf{R}_3^1$ are equal to [1, 1] and [1, 1], respectively. These values are shown in Tables 2 (a), (b) and (c).

6.1. Case 1: The Number of Correct Answers Is Known

If we assume that the number of correct answers is known, the search will be terminated as soon as the foreknown number of answers have been retrieved. With the knowledge of the search tree and the number of answers, we are able to derive all possible assignments of answers to the leaves as described previously. With these assignments, we can further analyze the average and best behavior of our model.

(I) Average Case

For the assignment $\mathbf{A}_j$, the completion time of node i becomes:

$$T_j^i = \begin{cases} 0, & \text{if node } i \text{ is a leaf;} \\ \max\{(t_f^i + t_p^{s_1^i} + T_j^{s_1^i}) \times R_j^i(1), ..., \\ \qquad (h \times t_f^i + t_p^{s_h^i} + T_j^{s_h^i}) \times R_j^i(h), ..., \\ \qquad (b^i \times t_f^i + t_p^{s_{b^i}^i} + T_j^{s_{b^i}^i}) \times R_j^i(b^i)\}, & \text{otherwise} \end{cases} \tag{6.1.1}$$

Eq. 6.1.1 can be derived in the following way. Since the message propagation is done sequentially from the leftmost to the rightmost child, a node i takes $h \times t_f^i$ units of time to prepare a copy of the message to the *h-th* child. Note that the time to process an incoming message at node i is the same (i.e. t_f^i) since it only involves processing of the message within the PE. It further takes $t_p^{s_h^i}$ units of time for a message to propagate from node i to its *h-th* child. Once the *h-th* child receives the message, it will propagate the message in the same fashion. It will then take $T_j^{s_1^i}$ time to find an answer. However, not every branch will lead to an answer. If the *h-th* branch leads to an answer, the corresponding element $R_j^i(h)$ is 1; otherwise, it is 0. By multiplying the $R_j^i(h)$'s to the time discussed above (e.g. $h \times t_f^i + t_p^{s_h^i} + T_j^{s_h^i}$), we eliminate the nodes which lead to no correct answers. Since there are multiple correct answers, the completion time is the longest time among the various paths that lead to an answer. Hence, a function *max* is applied to these paths to obtain the overall completion time.

If we assume that the t_f^i and $t_p^{s_h^i}$ are constants and are equal to t_f and t_p, respectively, *eq.* 6.1.1 becomes:

$$T_j^i = \begin{cases} 0, & \text{if node } i \text{ is a leaf;} \\ \max\{(t_f + t_p + T_j^{s_1^i}) \times R_j^i(1), ..., \\ \quad (h \times t_f + t_p + T_j^{s_h^i}) \times R_j^i(h), ..., \\ \quad (b^i \times t_f + t_p + T_j^{s_{b^i}^i}) \times R_j^i(b^i)\}, & \text{otherwise} \end{cases} \tag{6.1.2}$$

By recursively solving *eq.* 6.1.2, we can derive T_j^1, the time to complete the search for a given assignment of correct answers $\mathbf{A_j}$ from the root. Then, the average completion time, T_{avg}, becomes:

$$T_{avg} = \sum_{j=1}^{n} P(\mathbf{A_j}) \cdot T_j^1 \tag{6.1.3}$$

where $P(\mathbf{A_j})$ is the probability of assignment $\mathbf{A_j}$ being chosen. Since $P(\mathbf{A_j}) = \frac{1}{n}$ for all j, *eq.* 6.1.3 becomes:

$$T_{avg} = \frac{1}{n} \cdot \sum_{j=1}^{n} T_j^1 \tag{6.1.4}$$

Let us illustrate *eq.* 6.1.4 with the example given in Figure 5, assuming the t_f and t_p are constants. The answer assingments and the vector values of the $\mathbf{R_j^i}$ are shown in Figure 5 and in Tables 2(a), 2(b) and 2(c). From *eq.* 6.1.4, we obtain:

$$\begin{aligned} T_{avg} = & \frac{1}{3} \times \{\max\{(t_f + t_p + T_1^2) \times R_1^1(1), (2 \times t_f + t_p + T_1^3) \times R_1^1(2)\} \\ & + \max\{(t_f + t_p + T_2^2) \times R_2^1(1), (2 \times t_f + t_p + T_2^3) \times R_2^1(2)\} \\ & + \max\{(t_f + t_p + T_3^2) \times R_3^1(1), (2 \times t_f + t_p + T_3^3) \times R_3^1(2)\}\} \end{aligned}$$

By substituting the values of the $R_j^i(h)$ from Table 2, we obtain:

$$\begin{aligned} T_{avg} = & \frac{1}{3} \times \{\max\{(t_f + t_p + T_1^2) \times 1, (2 \times t_f + t_p + T_1^3) \times 0\} \\ & + \max\{(t_f + t_p + T_2^2) \times 1, (2 \times t_f + t_p + T_2^3) \times 1\} \\ & + \max\{(t_f + t_p + T_3^2) \times 1, (2 \times t_f + t_p + T_3^3) \times 1\}\} \end{aligned} \tag{6.1.5}$$

In order to solve *eq.* 6.1.5, we have to apply *eq.* 6.1.2 recursively to obtain T_j^2 and T_j^3 ($j = 1, 2, 3$). From *eq.* 6.1.2 and with the values of $\mathbf{R_j^2}$ in Table 2(b), we obtain:

$$\begin{aligned} T_1^2 &= \max\{(t_f + t_p + T_1^4) \times 1, (2 \times t_f + t_p + T_1^5) \times 1\} \\ T_2^2 &= \max\{(t_f + t_p + T_2^4) \times 1, (2 \times t_f + t_p + T_2^5) \times 0\} \\ T_3^2 &= \max\{(t_f + t_p + T_3^4) \times 0, (2 \times t_f + t_p + T_3^5) \times 1\} \end{aligned} \tag{6.1.6}$$

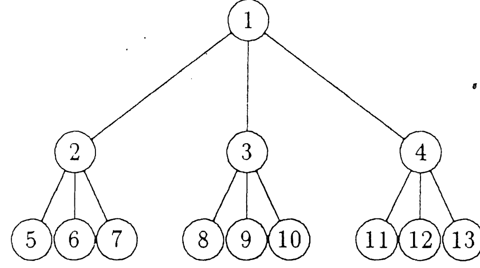

Figure 6
An Example of a Regular Tree

Similarly, with the values in Table 2(c), we obtain:

$$\begin{aligned} T_1^3 &= \max\{(t_f + t_p + T_1^6) \times 0\} \\ T_2^3 &= \max\{(t_f + t_p + T_2^6) \times 1\} \\ T_3^3 &= \max\{(t_f + t_p + T_3^6) \times 1\} \end{aligned} \tag{6.1.7}$$

Since nodes 4, 5 and 6 are leaves, we have:

$$T_j^4 = T_j^5 = T_j^6 = 0 \quad (j = 1, 2, 3) \tag{6.1.8}$$

By substituting (6.1.8), (6.1.7) and (6.1.6) to *eq.* 6.1.5, we obtain:

$$\begin{aligned} T_{avg} &= \frac{1}{3} \times \{\max\{(t_f + t_p + 2 \cdot t_f + t_p) \times 1, (2 \cdot t_f + t_p + 0) \times 0\} \\ &\quad + \max\{(t_f + t_p + t_f + t_p) \times 1, (2 \cdot t_f + t_p + t_f + t_p) \times 1\} \\ &\quad + \max\{(t_f + t_p + 2 \cdot t_f + t_p) \times 1, (2 \cdot t_f + t_p + t_f + t_p) \times 1\}\} \\ &= \frac{1}{3} \times \{(t_f + t_p + 2 \cdot t_f + t_p) + (t_f + t_p + 2 \cdot t_f + t_p) + (t_f + t_p + 2 \cdot t_f + t_p)\} \\ &= \frac{1}{3} \times \{9 \cdot t_f + 6 \cdot t_p\} \\ &= 3 \cdot t_f + 2 \cdot t_p \end{aligned} \tag{6.1.9}$$

If the tree is regular with depth w and branching factor b, and if there is only one answer, we can derive a closed form solution for T_{avg} in the following manner:

For a regular tree with only one answer, we observe the following characteristics: (1) if the root is at level 0, each subtree of a node at level l has $b^{(w-l)}$ leaves ($0 \leq l \leq w$). (2) each subtree at a given level l has the same number of possible assignments that lead to an answer from that subtree. The number of such assignments is equal to $b^{(w-l)}$. For example, at level 1 of the tree in Figure 6, each subtree rooting from nodes 2, 3 and 4 has ($3^{(2-1)} = 3$) leaves. Of the total ($3^2 = 9$) possible leaf assignments, each of these subtrees has 3 possible assignments that yield an answer.

With respect to the node 1 (the root) at level 0, the total time to complete the search for answers of all possible assignments has two components: (1) the time it takes to propagate messages to its immediate children at level 1 ($b^{w-1} \cdot \{(t_f + t_p) + (2 \cdot t_f + t_p) ... + (b \cdot t_f + t_p)\}$), and (2) the time it requires to search for the answer from its children at level 1. Since the search time of each node at the same level is the same, we obtain the term $b \times \sum_{j=1}^{n} T_j^{s_1^1}$. Therefore, the term $\sum_{j=1}^{n} T_j^1$ in *eq.* 6.1.4 becomes:

$$\sum_{j=1}^{n} T_j^1 = b^{w-1} \cdot [(t_f + t_p) + (2 \cdot t_f + t_p) ... + (b \cdot t_f + t_p)] + b \times \sum_{j=1}^{n} T_j^{s_1^1}$$

Hence, from *eq.* 6.1.4, T_{avg} becomes:

$$\begin{aligned} T_{avg} &= \frac{1}{n} \cdot \sum_{j=1}^{n} T_j^1 \\ &= \frac{1}{n} \cdot \{b^{w-1} \cdot [(t_f + t_p) + (2 \cdot t_f + t_p) ... + (b \cdot t_f + t_p)] + b \times \sum_{j=1}^{n} T_j^{s_1^1}\} \end{aligned}$$

For illustration purpose, let Y_1 be the term $\sum_{j=1}^{n} T_j^{s_1^1}$. Then, Y_1 is the completion time of a node at level 1 of the tree. Therefore, we obtain:

$$\begin{aligned} \sum_{j=1}^{n} T_j^1 &= b^{w-1} \cdot (t_f + t_p + ... + b \cdot t_f + t_p) + b \times Y_1 \\ Y_1 &= b^{w-2} \cdot (t_f + t_p + ... + b \cdot t_f + t_p) + b \times Y_2 \\ Y_2 &= b^{w-3} \cdot (t_f + t_p + ... + b \cdot t_f + t_p) + b \times Y_3 \\ &\vdots \qquad\qquad \vdots \\ Y_{w-1} &= b^{w-w} \cdot (t_f + t_p + ... b \cdot t_f + t_p) + b \times Y_w \end{aligned}$$

where Y_w is at the leaf level and is equal to 0. By substituting and simplifying the above equations, we have:

$$\sum_{j=1}^{n} T_j^1 = w \cdot b^{w-1} \cdot (t_f + t_p + ... + b \cdot t_f + t_p)$$

Therefore, the average completion becomes:

$$
\begin{aligned}
T_{avg} &= \frac{1}{n} \cdot \sum_{j=1}^{n} T_j^1 \\
&= \frac{1}{b^w} \cdot w \cdot b^{w-1} \cdot (t_f + t_p + 2 \cdot t_f + t_p + \ldots + b \cdot t_f + t_p) \\
&= \frac{w}{b} \cdot (t_f + t_p + 2 \cdot t_f + t_p + \ldots + b \cdot t_f + t_p) \\
&= \frac{w}{b}[(1 + 2 + \ldots + b) \times t_f + b \times t_p] \\
&= \frac{w}{b}[\frac{b(b+1)}{2} \times t_f + b \times t_p] \\
&= w[\frac{(b+1)}{2} t_f + t_p] \qquad (6.1.10)
\end{aligned}
$$

(II) Best Case

The best case is defined as the shortest completion time to search from the root for all answers among the possible assignments. Let T_{best} be the best completion time. We have:

$$T_{best} = \min_{1 \leq j \leq n} T_j^1 \qquad (6.1.11)$$

where the term T_j^1 is obtained from *eq.* 6.1.1.

However, if t_f and t_p are constants, the term T_j^1 is derived from *eq.* 6.1.2. For instance, consider the irregular tree in Figure 5, which contains two correct answers distributed among the three leaves. The T_j^1 is obtained in the same manner as in *eqs.* 6.1.5 to 6.1.8. Thus, we obtain:

$$
\begin{aligned}
T_{best} &= \min\{(t_f + t_p + 2t_f + t_p),\ (t_f + t_p + 2t_f + t_p),\ (t_f + t_p + 2t_f + t_p),\} \\
&= 3t_f + 2t_p
\end{aligned}
$$

If the tree is a regular tree with depth w, branching factor b and with one answer, the best case occurs if the answer is the leftmost leaf. Since, at each level, the first generated message is propagated into the correct path immediately. T_{best} becomes:

$$T_{best} = w \times (t_f + t_p) \qquad (6.1.12)$$

6.2. Case 2: The Number of Correct Answers Is Unknown

In the case that we have no knowledge regarding the number of correct answers in the search tree, the completion time will equal to the longest time it takes to traverse all paths until it reaches the leaves. For any node i, its completion time T^i becomes:

$$T^i = \begin{cases} 0, & \text{if node } i \text{ is a leaf;} \\ \max\{(t_f^i + t_p^{s_1^i} + T^{s_1^i}), ..., \\ \quad (h \times t_f^i + t_p^{s_h^i} + T^{s_h^i}), ..., \\ \quad (b^i \times t_f^i + t_p^{s_{b^i}^i} + T^{s_{b^i}^i})\}, & \text{otherwise} \end{cases} \quad (6.2.1)$$

The completion time starting from the root is equivalent to the worst case of the parallel tree search. Therefore, we obtain:

$$T_{worst} = T^1 \quad (6.2.2)$$

where T^1 is solved recursively with *eq.* 6.2.1

The difference between this equation and *Eq.* 6.1.1 is as follows. Since we do not know the number of correct answers, and we have to traverse all branches of the tree in any case, the binary variables $R_j^i(1), ..., R_j^i(h), ..., R_j^i(b^i)$ as in *eq.* 6.1.1 are no longer needed.

Similarly, if t_f and t_p are constants, *eq.* 6.2.2 is simplified to:

$$T_{worst} = \begin{cases} 0, & \text{if node 1 is a leaf;} \\ \max\{(t_f + t_p + T^{s_1^1}), ..., \\ \quad (h \times t_f + t_p + T^{s_h^1}), ..., \\ \quad (b^1 \times t_f + t_p + T^{s_{b^1}^1})\}, & \text{otherwise} \end{cases} \quad (6.2.3)$$

For the example given in Figure 5, we derive the worst case as follows:

$$T_{worst} = \max\{(t_f + t_p + T^2),\ (2t_f + t_p + T^3)\}$$

We solve T^2 and T^3 recursively and obtain:

$$\begin{aligned} T^2 &= \max\{(t_f + t_p + T^4),\ (2t_f + t_p + T^5)\} \\ T^3 &= \max\{(t_f + t_p + T^6)\} \end{aligned}$$

where T^4, T^5, T^6 are equal to zero since they are leaves. Therefore, T_{worst} becomes:

$$\begin{aligned} T_{worst} &= \max\{(t_f + t_p + 2t_f + t_p),\ (2t_f + t_p + t_f + t_p)\} \\ &= 3t_f + 2t_p \end{aligned}$$

Again, if the tree is a regular tree with depth w, branching factor b and with a single answer, the worst case occurs when the answer is the rightmost leaf. Since there is only one assignment and the maximum time equals $(b \times t_f + t_p)$ at each level of the tree, we obtain the closed form solution for the worst case as follows:

$$T_{worst} = w \times (b \times t_f + t_p) \tag{6.2.4}$$

7. Numerical Results

We shall first compare the backtracking scheme described in section 2 with the message propagation scheme described in section 3. We assume the values of t_u^i, t_b^i, t_f^i and $t_p^{s_h^i}$ to be constants throughout the numerical analysis. Through the numerical results, we demonstrate that our message propagation scheme will perform better than the backtracking scheme, especially when the knowledge-base is large.

7.1. Speedup factor in terms of the t_u/t_f ratio

In this section, we would demonstrate the speed up in performance of our model over the backtracking scheme. Let us define the speedup factor S as:

$$S = \frac{\textit{average completion time of backtracking scheme}}{\textit{average completion time of message propagation scheme}}$$

$S > 1$, if the message propagation scheme performs better than the backtracking scheme. We consider the speedup for the regular tree with single answer because the closed form solutions enable us to have general observations. From *eqs.* 5.1.3 and 6.1.10, the speedup factor becomes:

$$S = \frac{\frac{1}{2}[(w+v-1)t_u + (v-1)t_b]}{w_1[\frac{(b+1)}{2}t_f + t_p]}$$

where w and w_1 are the depth of the regular tree of the backtracking scheme and that of the message propagation scheme, respectively. From our assumption (4) described in section 4, in order to make sensible comparisons, we let $w_1 = 2 \times (w+1)$. Thus, the speedup factor S becomes:

$$S = \frac{\frac{1}{2}[(w+v-1)t_u + (v-1)t_b]}{2(w+1)[\frac{(b+1)}{2}t_f + t_p]}$$

Let $t_b = c \times t_u$ and $t_p = k \times t_f$, where c and k are constants. Thus S becomes:

$$S = \frac{(w+v-1)t_u + c(v-1)t_u}{4(w+1)[\frac{(b+1)}{2}t_f + k \times t_f]} = \frac{w + (c+1)(v-1)}{2(w+1)(b+2k+1)} \times \frac{t_u}{t_f}$$

Therefore, the speedup factor S is a function of the $(\frac{t_u}{t_f})$ ratio. In order to study the speedup of our model, we vary the values of k against a range of c. With $c = 0.1, 0.01$ and different values of k, we show the speedups of our message propagation scheme in Figure 11.

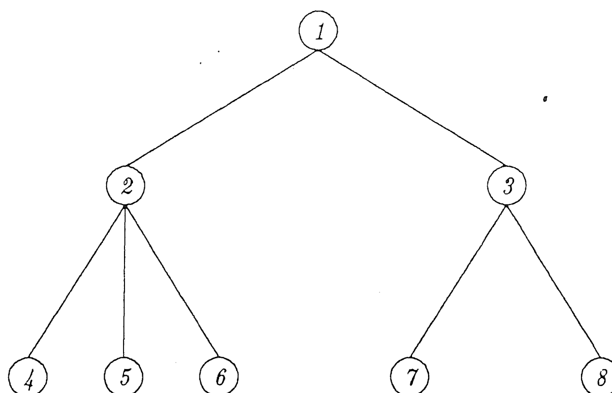

Figure 7
An example of a Proof Tree

Two observations can be made from this figure. First, the speedup factor increases as the size of the tree grows larger. In other words, the speedup is greater for a larger tree than that for a smaller tree. We can attribute this result to the fact that our message propagation model can exploit a higher degree of parallelism as the size of the tree increases. If the tree is very small, or when the t_u/t_f ratio is small, the backtracking scheme performs better. For example, consider lines 1,3,5, and 7 with very small t_u/t_f ratios. This is due to the fact that in our propagation model, the overhead in doing the parallel search (i.e. the communication time t_p) may be relatively higher. However, for a practical knowledge-base, the size of the search tree is usually large. Therefore, our proposed model is good for such applications.

Second, for the same size of tree, the constant k (sometimes it is called the *coupling factor*) has significant effect on the speedup. For example, consider lines 2 and 6. The speedup increases as the coupling factor k decreases. The coupling factor k is a measure of how much time we spend in communication between PEs. If k is a fraction of t_f (e.g. $t_f > t_p$), then we spend relatively more time in processing and fabricating messages in the PE than in communication. If the tree becomes larger, the cost of communication becomes larger. Therefore, in designing our message-driven knowledge-base machine, we prefer k to be relatively small in order to have better speedup.

7.2. Irregular Trees, Varying Number of Answers

In this section, we compare the average completion times of irregular trees with different number of answers. Consider the proof tree in Figure 7. We obtain the corresponding search tree by extending each node in Figure 7 with three nodes (as discussed in assumption 4 of section 4.) The resulting tree is depicted in Figure 8. The numerical results obtained from *eqs.* 5.1.2 and 6.1.4 are plotted in Figure 12 against different numbers of correct answers in the trees.

We notice that:

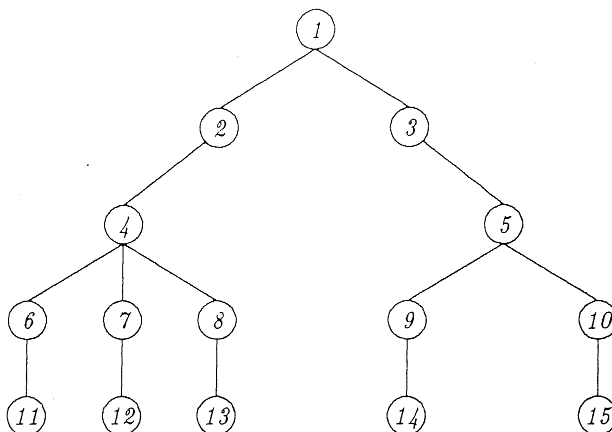

Figure 8
The Corresponding Search Tree of Figure 7

(1) the slope of the message propagation scheme is so flat that the extra time required to search for an additional answer is relatively small.

(2) in the backtracking scheme, the extra time required to search for an additional answer tends to be very high when the number of correct answers is small. It then levels off as the number of correct answers increases.

(3) the coupling factor k has a dominant effect in the performance in our message-driven model. For example, when k is increased from 0.5 to 1.5, the performance of the model degrades sharply.

The first observation is due to the fact that propagation in our model is done in parallel. Once a message is injected to the tree, it propagates asynchronously to search for all possible answers. The time difference to search for different number of answers is then very small.

For the second observation, we notice that, with a backtracking model, to search for an additional answer would mean to fail the currently found answer and to backtrack in order to get to the remaining branches of the tree. This backtracking time (t_b) is a dominant factor in determining the average completion time. We notice that the increase in the backtracking time to find additional answers diminishes as the total number of answers increases. For the example given in Figure 7, the average backtracking times (the second term of *eq.* 5.1.2, $\frac{1}{n}\sum_{j=1}^{n}(D_{p_j}|\mathbf{A_j} - d_{p_j}|\mathbf{A_j}) \times t_b$) are 4.8, 7.4, 8.8, 9.5 and 10.0 units of time for the number of answers ranging from 1 to 5, respectively. If the number of answers is changed from 1 to 2, the average backtracking time increases 2.6 (7.4 − 4.8) units of time. However, the average backtracking time only increases 0.5 (10.0 − 9.5) units of time for the case from 4 to 5 answers. Therefore, the additional time required to search for an answer levels off gradually as the number of answers is approaching to the number of leaves. The last observation just confirms our previous discussion in section 7.1.

7.3. Best and Worst Cases Comparisons

In this section, we show how our message-driven model compares to the conventional backtracking model in the best and worst cases. From Figure 12, we notice that, with parameters $t_f = 2$ $t_p = 1.0$ $k = 0.5$ and $t_u = 2$ $t_b = 0.6$ $c = 0.3$ for our model and the backtracking model, respectively, their average performances are very similar. Therefore, we use these parameters to analyze the corresponding best and worst cases for both models. The numerical results obtained from equations 5.1.5, 5.2.2, 6.1.11 and 6.2.3 are plotted in Figure 13 for the examples given in Figures 7 and 8. As described previously, the worst case analysis assumes no knowledge of the number of correct answers in the tree. In this case, the tree has to be searched completely. Therefore, its completion time is a constant for a given tree regardless of the number of answers to be retrieved. Thus, its completion time is plotted as a horizontal line in Figure 13. We observe that:

(1) for the worst case, our model has better performance than the backtracking model.

(2) for the best case, we see the same performance characteristic found in the average case. For example, the performance of the backtracking is better when the number of answers is less, which is indicated in Figure 12.

(3) if the number of answers is equal to the number of leaves, the best and worst cases are equivalent.

The first observation is due to the fact that, in the worst case, the backtracking model needs to backtrack every time when it reaches the leaves until it arrives at the rightmost leaf. In our model, the message propagation allows to search for different paths asynchronously. Therefore, it behaves better in the worst case.

If the backtracking model performs better in the average case, it is obvious that it will also perform better in the best case. This is because most of the inefficiency of such a model lies in the backtracking mechanism while our model has the fixed overhead of propagation. If the overhead is less costly, our performance will definitely be better.

For the last observation, it is obvious that if the number of answers equals the number of leaves, the best case still has to traverse all branches to retrieve all the answers.

7.4. The Structure of the Knowledge-Base: Bushy vs Skinny

In this section, we show how the structure of the knowledge-base may affect the performance of our model. Consider the tree in Figure 9. The tree has a total of ten nodes with one level. With the same number of nodes, we arbitrarily restructure the bushy tree with three levels as in Figure 10. We vary the number of correct answers and the values of k in both cases. Their average completion times are plotted in Figure 14. We make the following observations:

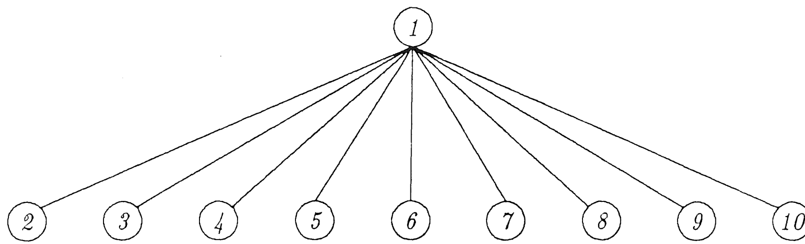

Figure 9
Example of a Bushy Tree

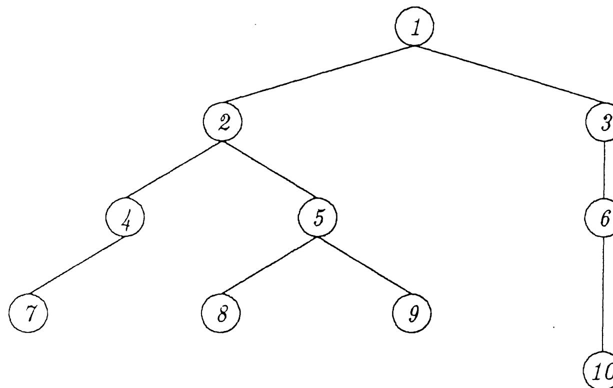

Figure 10
A Skinny Tree Restructured From Figure 9

(1) if k (the communication delay) is relatively large (for a certain size of tree), a bushy structure (i.e. shallow hierarchy with many branches as in Figure 9) is more desirable than a skinny structure (i.e. deep hierarchy with fewer branches as in Figure 10). For example, if $k = 1.5$, the average completion time of the bushy tree is smaller than that of the skinny tree.

(2) in the case that k (the communication delay) is small, a skinny structure (such as Figure 10) becomes more desirable. For instance, if $k = 0.5$, the average completion time of the skinny tree is smaller than that of the bushy tree.

(3) even though a skinnier structure may have worse performance when k is large, the extra time to search for an additional answer seems less when compared to that of a bushy structure. For example, the slopes of all curves of the skinny trees are relatively flat.

The reasons of observations (1) and (2) can be explained as follows. Since a skinny structure has deeper hierarchy than a bushy structure, a message has to pass through each level prior to reaching the leaves. Hence, the time required is affected significantly by the depth of the structure. If the communication delay (i.e. t_p) is

relatively large (when compared to t_f), it will be more costly to propagate a message in a skinny structure than in a bushy structure. This result suggests that we may improve the performance by manipulating the structure of the knowledge-base: if we have *a priori* knowledge of the communication delay, we may re-structure the knowledge-base to be bushy if k is relatively large; otherwise, we may organize it to be skinnier.

For the last observation, we realize that a skinny structure has fewer branches than a bushy structure. Moreover, our message propagation scheme generates copies of message sequentially (as described in section 6.1). Therefore, the time difference of generating the first and last copies of the outgoing messages is less than that in a bushy structure. Therefore, the curve of a skinny structure tends to have a smaller slope.

7.5. Comparisons with Large Search Trees

In the previous sections, the size of the search tree was quite small. In this section, we consider cases of larger search trees.

For simplicity, we adopt the analyses of regular trees with a single answer. The closed form equations 5.1.3 and 6.1.10 are used. From our previous analysis in subsection 7.1, we observed that our model has better performance than the backtracking model in large search trees. In order to verify this perspective, we intentionally choose the values of the parameters that are favorable to the backtracking model in a small tree. However, as we shall demonstrate, if the tree grows very large, our message propagation model reverses the situation: it performs far much better than the backtracking model. For this reason, parameters chosen for the analysis are: $t_f = 2$, $k = 1.5$ for the message propagation model and $t_u = 2$, $c = 0.1$ for the backtracking model. Note that, with these parameters for small trees, the backtracking model performs better as shown in Figure 12.

For a bushy structure (i.e. with branching factor from 30 to 150 and depth of 2 or 4), the performance of both models are shown in Table 5. Note that we have extended the depth of the search tree for the message-driven model as we did in section 7.1. As shown in Table 5, the time required in the message propagation model is much less than in the backtracking model. For example, when the branching factor is 150 and the depth is 4, the backtracking scheme requires 1.1×10^9 units of time while the message propagation scheme only takes 3.1×10^3 units of time. This confirms the observation in subsection 7.1: as the size of the knowledge-base grows, our model performs better because the message propagation scheme achieves a higher degree of parallelism. Furthermore, as the size of the tree grows, the times that the backtracking model takes to complete the search increase sharply (e.g. from 2.1×10^3 to 1.1×10^9). On the other hand, the completion times that the message propagation model takes show a relatively flatter slope (e.g. from 4.1×10^2 to 3.1×10^3).

When the structure of the knowledge-base is skinny rather than bushy (for instance, the depth has a range from 4 to 10 while the branching factor is either 5 or 8), the results are shown in Table 6. The results again confirm the observation that our model can achieve better performance as the size of the knowledge-base increases. Consider the case where the depth is 10 and the branching factor is 8. The backtracking model requires 2.7×10^9 units of time, while the message propagation model only needs 5.3×10^2 units of time to complete the search.

Furthermore, the results show that the structure of the knowledge-base does not have much effect on the backtracking model. In Tables 5 and 6, for a tree with similar total number of nodes, we notice that the backtracking model shows similar performance regardless of the structure of the tree. Consider the case where the size of the tree (i.e. n) equals $14,521$ and $19,531$ in Table 5 and Table 6, respectively. On the other hand, the results of the message-driven model show that the structure of the knowledge-base is a salient factor of performance. For example, consider the case that the branching factor is 150 and depth is 4 in Table 5 and the case that the branching factor is 8 and the depth is 10 in Table 6. The first case represents a bushy tree. The latter case is a skinnier tree with a size *twice that* of the first tree. However, the time needed in the skinnier tree is even less than that of the bushy tree (e.g. 5.3×10^2 versus 3.1×10^3). Note that this is not contradictory to the observation in section 7.4. It shows that the coupling factor k is relative to the size of the tree.

8. Concluding Remarks

In this paper, we have presented a message-oriented knowledge-based model. The major components of our model are the semantic networks representation of the IDB and EDB, and the implementation of the SL-resolution by means of message propagation. Since, the message propagation is done asynchronously, the resolution done in this manner is a set at a time instead of using the backtracking method which carries out the resolution a tuple at a time.

Both our model and the conventional backtracking model were analyzed and compared. We demonstrated that our model outperforms the backtracking model in most cases.

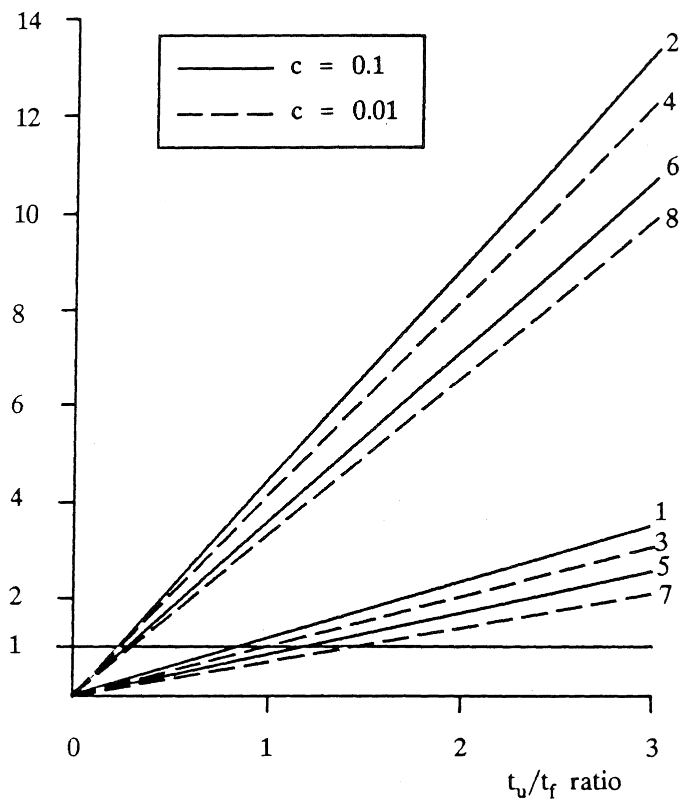

Figure 11

Speedup Factor versus t_u/t_f ratio

line 1 : v = 40, w = 3, b = 3, k = 0.5
line 3 : v = 40, w = 3, b = 3, k = 0.5
line 5 : v = 40, w = 3, b = 3, k = 1.5
line 7 : v = 40, w = 3, b = 3, k = 1.5

line 2 : v = 259, w = 3, b = 6, k = 0.5
line 4 : v = 259, w = 3, b = 6, k = 0.5
line 6 : v = 259, w = 3, b = 6, k = 1.5
line 8 : v = 259, w = 3, b = 6, k = 1.5

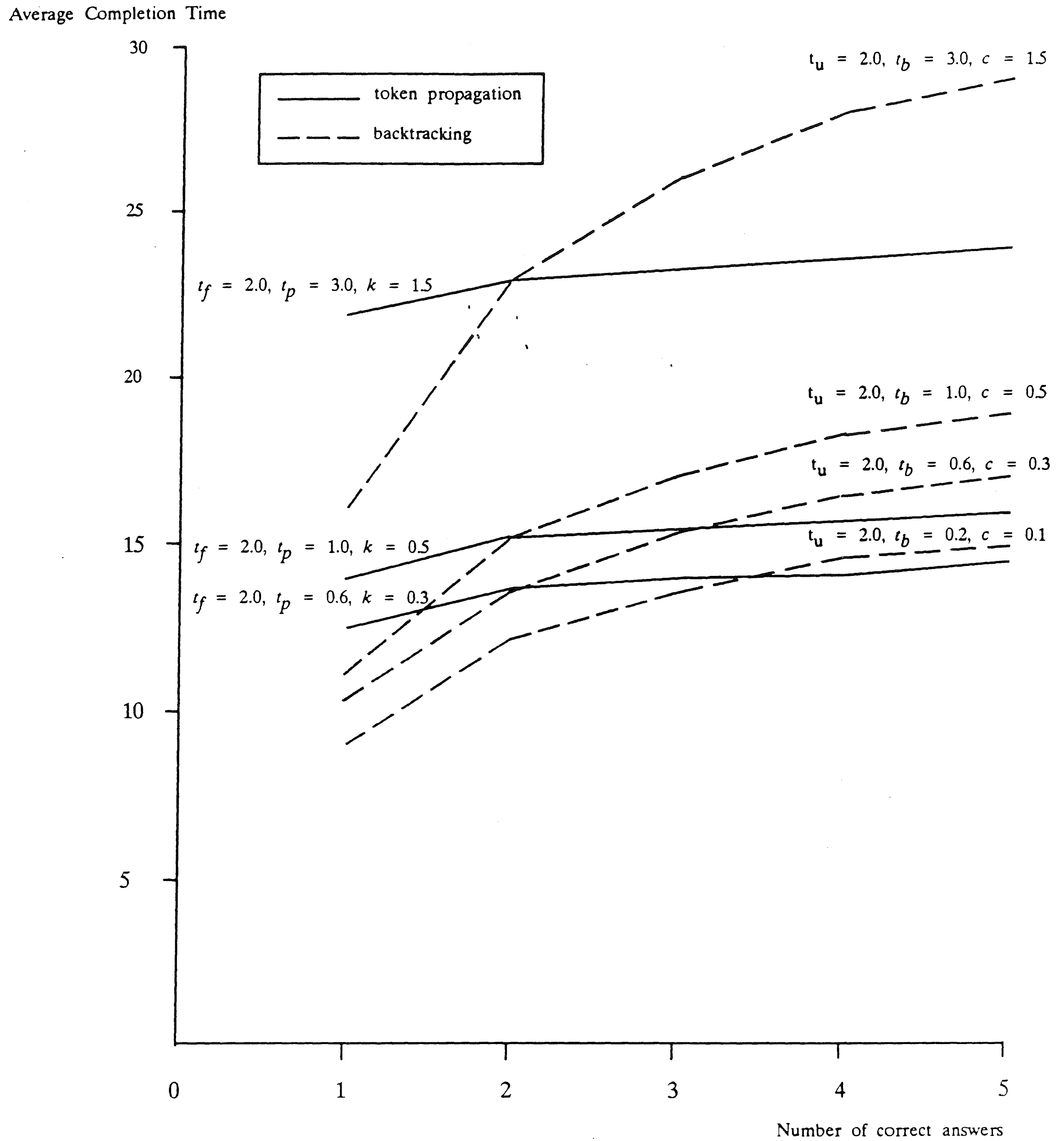

Figure 12

Irregular tree with different number of correct answers

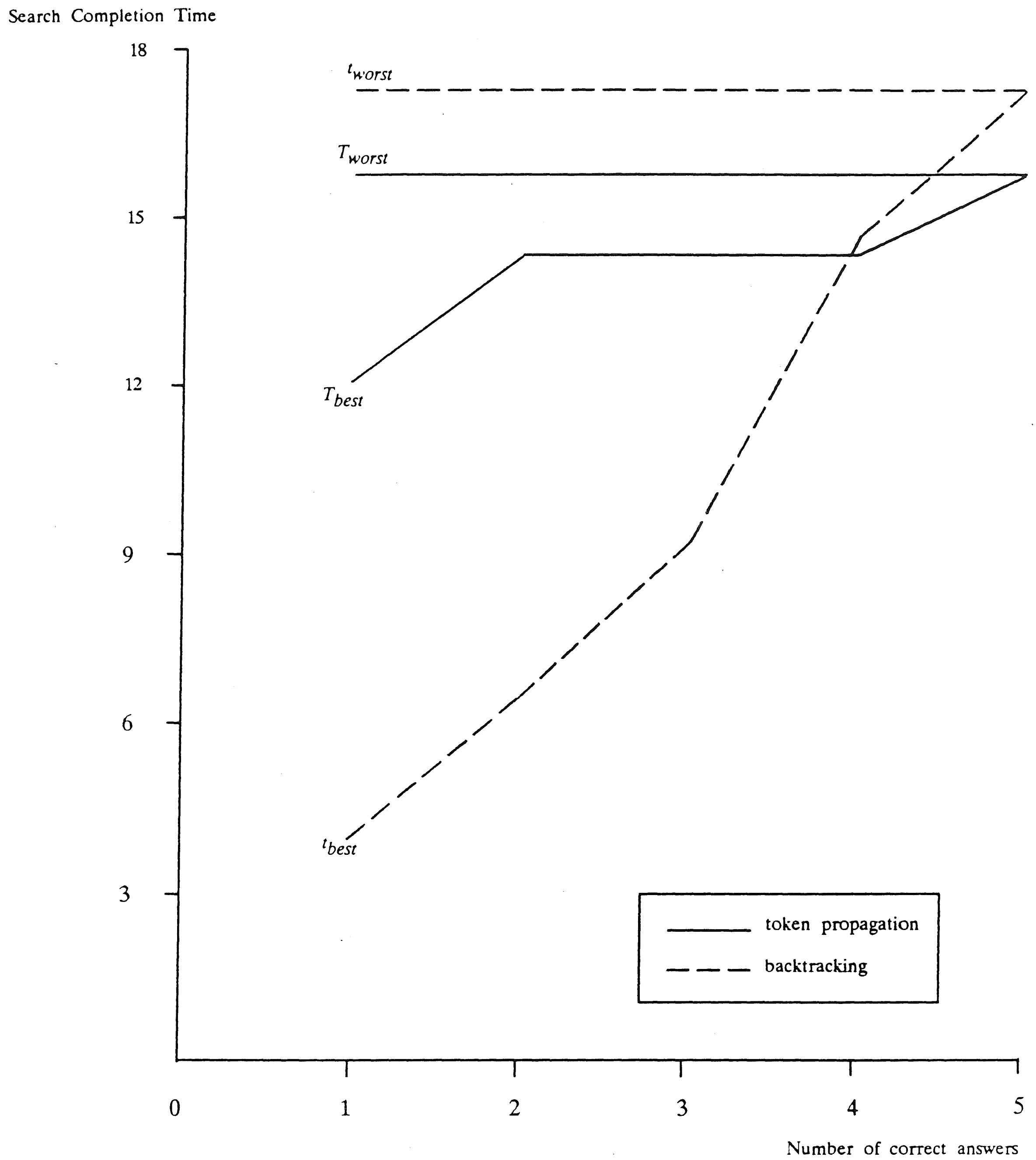

Figure 13

Best and Worst Cases Analysis

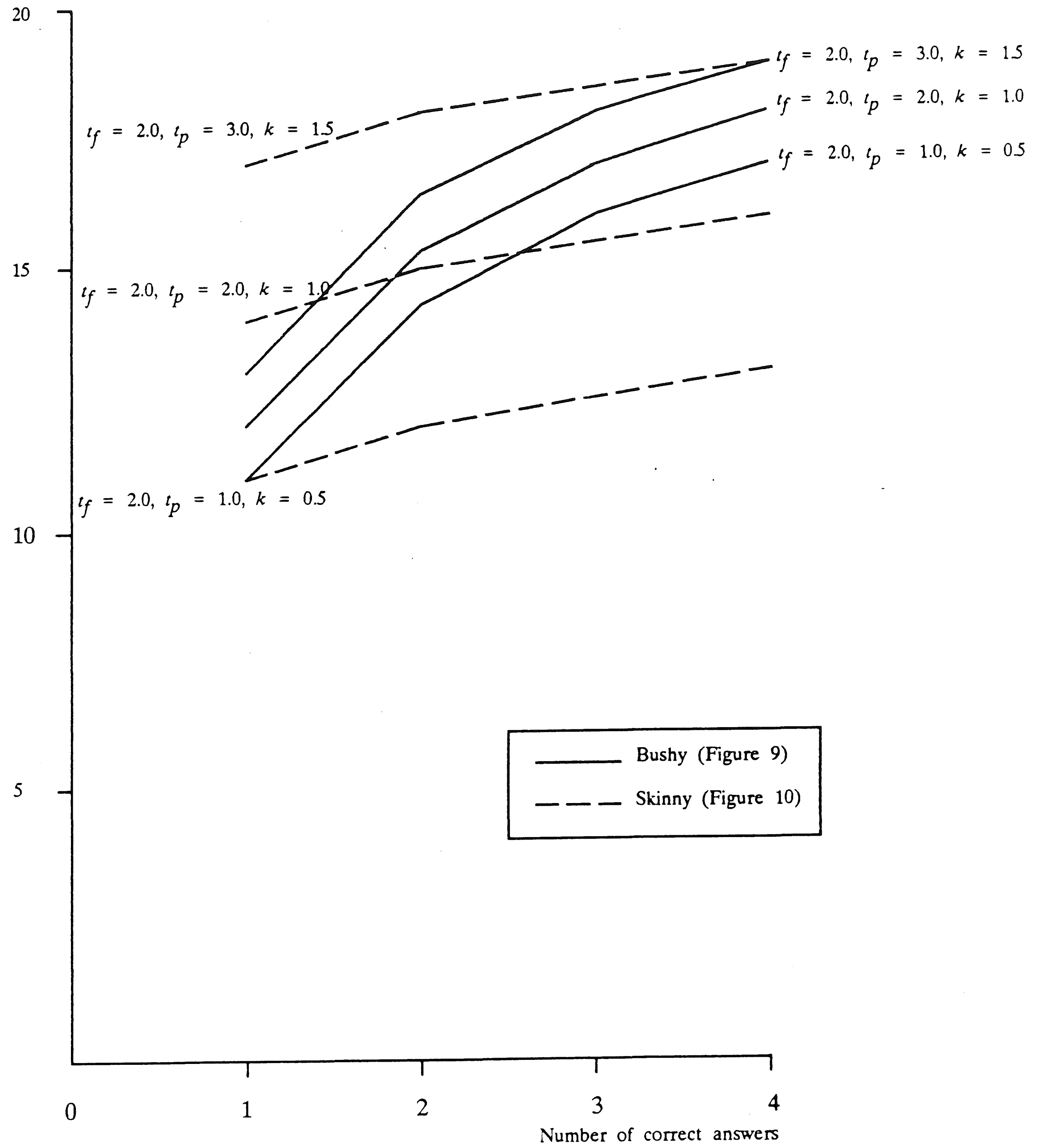

Figure 14

Bushy Versus Skinny Trees

depth \ branching	30	60	90	120	150
2	n=931 l=900 $t_{avg}=2.1\times10^3$ $T_{avg}=4.1\times10^2$	n=3,661 l=3,600 $t_{avg}=8.1\times10^3$ $T_{avg}=7.7\times10^2$	n=8,191 l=8,100 $t_{avg}=1.8\times10^4$ $T_{avg}=1.1\times10^3$	n=14,521 l=14,400 $t_{avg}=3.2\times10^4$ $T_{avg}=1.5\times10^3$	n=22,651 l=22,500 $t_{avg}=5.0\times10^4$ $T_{avg}=1.8\times10^3$
4	n=837,931 l=810,000 $t_{avg}=1.8\times10^6$ $T_{avg}=6.8\times10^2$	n=13,179,661 l=12,960,000 $t_{avg}=2.9\times10^7$ $T_{avg}=1.3\times10^3$	n=66,347,191 l=65,610,000 $t_{avg}=1.5\times10^8$ $T_{avg}=1.9\times10^3$	n=209,102,521 l=207,360,000 $t_{avg}=4.6\times10^8$ $T_{avg}=2.5\times10^3$	n=509,647,651 l=506,250,000 $t_{avg}=1.1\times10^9$ $T_{avg}=3.1\times10^3$

$t_u=2,\ c=0.1;\ t_f=2,\ k=1.5$
n= total number of nodes
l= total number of leaves
t_{avg}= the time for backtracking scheme
T_{avg}= the time for token propagation scheme

Table 5

Large Bushy Trees

depth / branching	4	6	8	10
5	n = 781 l = 625 $t_{avg} = 1.7\times10^3$ $T_{avg} = 1.8\times10^2$	n = 19,531 l = 15,625 $t_{avg} = 4.3\times10^4$ $T_{avg} = 2.5\times10^2$	n = 488,281 l = 390,625 $t_{avg} = 1.1\times10^6$ $T_{avg} = 3.2\times10^2$	n = 12,207,031 l = 9,756,625 $t_{avg} = 2.7\times10^7$ $T_{avg} = 4.0\times10^2$
8	n = 4,681 l = 4,096 $t_{avg} = 1.0\times10^4$ $T_{avg} = 2.4\times10^2$	n = 299,593 l = 262,144 $t_{avg} = 6.6\times10^5$ $T_{avg} = 3.4\times10^2$	n = 19,173,961 l = 16,777,216 $t_{avg} = 4.2\times10^7$ $T_{avg} = 4.3\times10^2$	n = 1,227,133,513 l = 1,073,741,824 $t_{avg} = 2.7\times10^9$ $T_{avg} = 5.3\times10^2$

$t_u = 2,\ c = 0.1;\ t_f = 2,\ k = 1.5$
n = total number of nodes
l = total number of leaves
t_{avg} = the time for backtracking scheme
T_{avg} = the time for token propagation scheme

Table 6

Large Skinny Trees

References

[Agha85] Agha, G.A. Actors: A Model of Concurrent Computation In Distributed Systems. Tech. Rep. No. 844. MIT Artificial Intelligence Lab., MIT, Cambridge, Mass..

[BiLe87] Bic, L., Lee, C. A Data-Driven Model for a Subset of Logic Programming. *To appear in TOPLAS* (1987).

[ChLe73] Chang, Chin-Liang and Lee, Richard Char-Tung *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[Com82] *Computer*, Special Issue on Dataflow Systems *V15*, n2 (Feb., 1982).

[DeKo79] Deliyanni, Amaryllis and Robert A. Kowalski Logic and Semantic Networks. In *CACM*, Vol. *22-3*, 1979, pp. 184–192.

[Kow79] Kowalski, R. *Logic for Problem Solving*, North-Holland, New York, 1979.

[KoKu71] Kowalski, R.A. and Kuehner, D., Linear Resolution with Selection Function. In *Artificial Intelligence, 2(1971)*, pp. 227-260.

[Rob79] Robinson, J.A. *Logic: Form and Function*, North-Holland, New York, 1979.

[TBH82] Treleaven, P.C., Brownbridge, D.R., Hopkins, R.P Data-Driven and Demand-Driven Computer Architecture. *Computing Surveys V14*, n1 (March, 1982), 93–143.

[WB86] Wong, Wang-chan and Bic, Lubomir, Efficient Recursion Termination For Function-Free Horn Logic. TR 86-26. ICS Dept, University of California, Irvine (1986).

[WB87a] Wong, Wang-chan and Bic, Lubomir A Tagging Scheme to Prevent Infinite Recursion in First-Order Databases Function-Free Horn Database. *To appear in Proc of the Second Int'l Conf on Computer Applications* (June, 1987), IEEE.