Game Trees ;
Searching Techniques
and a Pathological Phenomenon

Agata Muszycka

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

April 1985

ABSTRACT

Game Trees ; Searching Techniques and Pathological Phenomenon

Agata Muszycka

The pruning strategies Branch-and-bound, Alphabeta, Palphabeta, Principal Variation Search, Scout and SSS* are empirically compared on uniform and nonuniform game trees, with four different schemes of assigning static-values to leaf nodes. Results are given discussing the relative performance of these strategies based on the number of nodes created, node-visits and CPU time. Then different methods of speeding-up the tree search are presented. These methods were developed based on the assumption that one wishes to search deeper. Using a probabilistic model of a game, the quality of decision made with deeper searching is examined. The pathological phenomenon is described and the possible causes of it and cures for it are reviewed.

ACKNOWLEDGEMENTS

I would like to thank my teacher and supervisor, Prof. R. Shinghal, for his superb guidance and support throughout this work.

Also, to my friends and family I say thank you for your encouragement and support.

TABLE OF CONTENTS

ABSTRACT

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION.........................1

1.1 Notation For Game Trees............2

2.1 Searching Procedures for Game Trees..............................5

CHAPTER 2. DESCRIPTION OF DIFFERENT PRUNING STRATEGIES..................11

2.1 Branch-and-Bound Algorithm (under the negamax framework).............11

2.2 Alphabeta Algorithm (under the negamax framework)....................13

2.3 Palphabeta Algorithm (under the negamax framework)....................16

2.4 Principal Variation Search (under the negamax framework).............18

2.5 Scout Algorithm (under the negamax framework)..........................21

2.6 SSS* Algorithm (under the minimax framework)..........................24

CHAPTER 3. EMPIRICAL COMPARISON OF PRUNING STRATEGIES..................31

3.1 Criteria used for Performance

Evaluation....................31
3.2 Kinds of Game Trees Simulated......33
3.3 Methods of Assigning Static-Values to Leaf Nodes....................34
3.4 Some Theoretical Results for Complexity of the Pruning Strategies....................35
3.5 Scope of the Experiments..........39
3.6 Results of the Experiments.........40
3.6.1 Comparison Based on the Number of All Nodes Created.................41
3.6.2 Comparison Based on the Number of Leaf Nodes Created................44
3.6.3 Comparison Based on the Number of Node-Visits....................81
3.6.4 Comparison Based on the CPU time Taken....................83
3.7 Overall Remarks on the Pruning Strategies....................101
CHAPTER 4. METHODS OF SPEEDING-UP THE TREE-SEARCH....................103
4.1 Parallel Implementation of Pruning Strategies................103
4.2 Ordering of Nodes in a Game Tree..112
4.3 Use of Transposition Tables.......112
4.4 The Killer Heuristic..............115

CHAPTER 5. PATHOLOGY IN GAME TREES............117

5.1 The Nature of Pathology...........118

5.2 Possible Methods of Overcoming Pathology.........................133

5.3 Experiments Simulated on Pathological and Nonpathological Game Trees....143

5.4 Concluding Remarks................159

CHAPTER 6. CONCLUSIONS.........................160

6.1 Highlights of Results Observed....160

6.2 Suggestions for Further Research..161

REFERENCES....................................163

APPENDIX 1. COMPARISON OF DIFFERENT VERSIONS OF SCOUT ALGORITHM................168

LIST OF FIGURES

FIGURE 1. A specimen game tree....................4
FIGURE 2. A game tree under minimaxing..............7
FIGURE 3. A game tree under negamaxing..............9
FIGURE 4. An example to show that Alphabeta prunes more nodes than Branch-and-bound..15
FIGURE 5. An example in which PVS prunes more nodes than Palphabeta...................20
FIGURE 6. An example in which Scout examines more nodes than Palphabeta or PVS............22
FIGURE 7. Solution trees of a game tree............29
FIGURE 8. A specimen game tree processed by SSS*...30
FIGURE 9. Average number of leaf nodes created for uniform tree of depth 4 with integer-dependent static-values assignment.......45
FIGURE 10. Average number of leaf nodes created for uniform tree of depth 4 with real-dependent static-values assignment.......46
FIGURE 11. Average number of leaf nodes created for uniform tree of depth 4 with unordered-independent static-values assignment.....47
FIGURE 12. Average number of leaf nodes created for uniform tree of depth 4 with 0.2-ordered-independent static-values assignment.....48
FIGURE 13. Average number of leaf nodes created for uniform tree of depth 4 with 0.4-ordered-independent static-values assignment.....49

FIGURE 14. Average number of leaf nodes created for uniform tree of depth 4 with 0.6-ordered-independent static-values assignment.....50
FIGURE 15. Average number of leaf nodes created for uniform tree of depth 4 with 0.8-ordered-independent static-values assignment.....51
FIGURE 16. Average number of leaf nodes created for uniform tree of depth 4 with 1.0-ordered-independent static-values assignment.....52
FIGURE 17. Average number of leaf nodes created for nonuniform tree of depth 4 with integer-dependent static-values assignment.......65
FIGURE 18. Average number of leaf nodes created for nonuniform tree of depth 4 with real-dependent static-values assignment.......66
FIGURE 19. Average number of leaf nodes created for nonuniform tree of depth 4 with unordered-independent static-values assignment.....67
FIGURE 20. Average number of leaf nodes created for nonuniform tree of depth 4 with 0.2-ordered-independent static-values assignment.....68
FIGURE 21. Average number of leaf nodes created for nonuniform tree of depth 4 with 0.4-ordered-independent static-values assignment.....69
FIGURE 22. Average number of leaf nodes created for nonuniform tree of depth 4 with 0.6-ordered-independent static-values assignment.....70

FIGURE 23. Average number of leaf nodes created for nonuniform tree of depth 4 with 0.8-ordered-independent static-values assignment.....71
FIGURE 24. Average number of leaf nodes created for nonuniform tree of depth 4 with 1.0-ordered-independent static-values assignment.....72
FIGURE 25. Average CPU time taken for uniform tree of depth 4 with integer-dependent static-values assignment.......85
FIGURE 26. Average CPU time taken for uniform tree of depth 4 with real-dependent static-values assignment.......86
FIGURE 27. Average CPU time taken for uniform tree of depth 4 with unordered-independent static-values assignment.....87
FIGURE 28. Average CPU time taken for uniform tree of depth 4 with 0.2-ordered-independent static-values assignment.....88
FIGURE 29. Average CPU time taken for uniform tree of depth 4 with 0.4-ordered-independent static-values assignment.....89
FIGURE 30. Average CPU time taken for uniform tree of depth 6 with 0.6-ordered-independent static-values assignment.....90
FIGURE 31. Average CPU time taken for uniform tree of depth 4 with 0.8-ordered-independent static-values assignment.....91

FIGURE 32. Average CPU time taken for uniform tree of depth 4 with 1.0-ordered-independent static-values assignment.....92
FIGURE 33. Average CPU time taken for nonuniform tree of depth 4 with integer-dependent static-values assignment.......93
FIGURE 34. Average CPU time taken for nonuniform tree of depth 4 with real-dependent static-values assignment.......94
FIGURE 35. Average CPU time taken for nonuniform tree of depth 4 with unordered-independent static-values assignment.....95
FIGURE 36. Average CPU time taken for nonuniform tree of depth 4 with 0.2-ordered-independent static-values assignment.....96
FIGURE 37. Average CPU time taken for nonuniform tree of depth 4 with 0.4-ordered-independent static-values assignment.....97
FIGURE 38. Average CPU time taken for nonuniform tree of depth 4 with 0.6-ordered-independent static-values assignment.....98
FIGURE 39. Average CPU time taken for nonuniform tree of depth 4 with 0.8-ordered-independent static-values assignment.....99
FIGURE 40. Average CPU time taken for nonuniform tree of depth 4 with 1.0-ordered-independent static-values assignment....100

FIGURE 41. Distinction made among sons of nodes in a game tree....104

FIGURE 42. Cut-offs which may occur in sequential and parallel Alphabeta....106

FIGURE 43. Static-ordering vs dynamic-ordering of nodes in a game tree....114

FIGURE 44. Subsequent values of ERR for uniform game tree with $err1_d=0.1$, $err2_d=0.1$....121

FIGURE 45. Subsequent values of ERR for uniform game tree with $err1_d=0.1$, $err2_d=0.2$....122

FIGURE 46. Subsequent values of ERR for uniform game tree with $err1_d=0.2$, $err2_d=0.1$....123

FIGURE 47. Subsequent values of ERR for uniform game tree with $err1_d=0.2$, $err2_d=0.2$....124

FIGURE 48. Game tree representing the Pearl-game...128

FIGURE 49. Game tree representing incremental game.131

FIGURE 50. Subsequent values of ERR for nonuniform game tree with $err1_d=0.1$, $err2_d=0.1$....134

FIGURE 51. Subsequent values of ERR for nonuniform game tree with $err1_d=0.1$, $err2_d=0.2$....135

FIGURE 52. Subsequent values of ERR for nonuniform game tree with $err1_d=0.2$, $err2_d=0.1$....136

FIGURE 53. Subsequent values of ERR for nonuniform game tree with $err1_d=0.2$, $err2_d=0.2$....137

FIGURE 54. An example in which minimaxing differs from product-propagation in choosing the move....140

FIGURE 55. The B* algorithm 142

FIGURE 56. An example in which the evaluation function used by Nau is not accurate 144

FIGURE 57. An example in which Scout which uses testm prunes less nodes than Scout presented in section 2.5 172

FIGURE 58. Average CPU time taken by three different versions of Scout 174

LIST OF TABLES

TABLE I. Ranking of pruning strategies under the criterion of nodes created..............42
TABLE II. Newborn's theoretical results for expected number of leaf nodes created by Alphabeta algorithm..................53
TABLE III. Average number of leaf nodes created for uniform tree of depth 4 with integer-dependent static-values assignment.......54
TABLE IV. Average number of leaf nodes created for uniform tree of depth 4 with real-dependent static-values assignment.......55
TABLE V. Average number of leaf nodes created for uniform tree of depth 4 with unordered-independent static-values assignment.....56
TABLE VI. Average number of leaf nodes created for uniform tree of depth 4 with 0.2-ordered-independent static-values assignment.....57
TABLE VII. Average number of leaf nodes created for uniform tree of depth 4 with 0.4-ordered-independent static-values assignment.....58
TABLE VIII. Average number of leaf nodes created for uniform tree of depth 4 with 0.6-ordered-independent static-values assignment.....59
TABLE IX. Average number of leaf nodes created for uniform tree of depth 4 with 0.8-ordered-independent static-values assignment.....60

TABLE X. Average number of leaf nodes created for uniform tree of depth 4 with 1.0-ordered-independent static-values assignment.....61

TABLE XI. Average number of leaf nodes created for nonuniform tree of depth 4 with integer-dependent static-values assignment.......73

TABLE XII. Average number of leaf nodes created for nonuniform tree of depth 4 with real-dependent static-values assignment.......74

TABLE XIII. Average number of leaf nodes created for nonuniform tree of depth 4 with unordered-independent static-values assignment.....75

TABLE XIV. Average number of leaf nodes created for nonuniform tree of depth 4 with 0.2-ordered-independent static-values assignment.....76

TABLE XV. Average number of leaf nodes created for nonuniform tree of depth 4 with 0.4-ordered-independent static-values assignment.....77

TABLE XVI. Average number of leaf nodes created for nonuniform tree of depth 4 with 0.6-ordered-independent static-values assignment.....78

TABLE XVII. Average number of leaf nodes created for nonuniform tree of depth 4 with 0.8-ordered-independent static-values assignment.....79

TABLE XVIII. Average number of leaf nodes created for nonuniform tree of depth 4 with 1.0-ordered-independent static-values assignment.....80

TABLE XIX. Nau's results for approximation of the probability of correct decision for the Pearl-game.......................146

TABLE XX. Results from duplication of Nau's experiment.......................147

TABLE XXI. Nau's results for approximation of the probability of correct decision for the incremental games.......................148

TABLE XXII. Results from duplication of Nau's experiment.......................149

TABLE XXIII. Approximation of the probability of correct decision for uniform trees with integer-dependent static-values assignment.......................152

TABLE XXIV. Approximation of the probability of correct decision for uniform trees with real-dependent static-values assignment.......................153

TABLE XXV. Nau's results for the estimation of the probability of correct decision using product-propagation rule for the Pearl-game.......................155

TABLE XXVI. Nau's results for the estimation of the probability of correct decision using product-propagation rule for incremental game.......................156

TABLE XXVII. Results from duplication of Nau's experiment for the Pearl-game...........157

TABLE XXVIII. Results from duplication of Nau's experiment for the incremental game......158

CHAPTER 1.

INTRODUCTION.

Games, such as chess and checkers, hold an inexplicable fascination for many people. In order to win one has to find the strategy which specifies the best response to every conceivable move of the opponent. Computer programs, which play games, generate moves and choose the best one, based on some estimation of goodness of game positions. Games may be represented by the game trees where the branches are moves, replies etc. Game trees are usually searched by the minimax procedure [21]. To reduce the search effort, researchers have proposed different tree-pruning strategies, which create only a fraction of the game tree, and still allow a game-playing computer program to make an optimal choice for its move. Some of these pruning strategies are known as the Branch-and-bound [12], Alphabeta [12], Scout [23] and SSS* [30] algorithms.

A question arises regarding the comparative performance of the different pruning strategies. In this thesis the exhaustive examination of the known pruning strategies is presented and the results obtained are reported. Then the possible methods of speeding-up the pruning strategies, such as parallel implementation, transposition tables, ordering of nodes and killer heuristic, are discussed.

Another interesting problem for game-playing programs is the relation between the depth of search and the quality of decision made. Until recently there was almost an universal belief that increasing the depth of search increases the correctness of decision made. But the investigations by Beal [5], Bratko and Gams [8], Nau [15,16] and Pearl [25] showed that there exist a class of game trees for which this fact is not true, and such a class of game trees was called pathological. In this thesis the decision quality with deeper searching is analyzed. The models, for which increasing the depth of search is beneficial, and the models for which it is not, are discussed.

This thesis is intended as a contribution to the domain of artificial intelligence.

In the remaininig part of this chapter the notation for the game trees is given and the minimaxing and negamaxing search procedures are reviewed.

1.1. Notation for Game Trees.

In using the word games, we restrict ourselves to two-person, zero-sum, perfect-information games (for example, chess). We have two players, hence we speak of a two-person game. One player wins what the other loses, so the sum of their gains is zero. There is no concealed information for any of the players, hence we speak of the perfect-information game. Any stage in such a game can be represented by a game tree where nodes of a tree correspond to positions in the game. A specimen game tree is shown in Figure 1. If from a position p' one is permited to move to any of the finite number of positions p_1', p_2', ..., p_f', then in the game tree there exists a branch directed from the corresponding node p to node p_1, another branch from p to p_2 and so on. The value of f is called the fan-out of the node p, and p_1, p_2, ..., p_f are called the siblings of one another. Node p_i is the left sibling of the nodes p_{i+1}, p_{i+2}, ..., p_f, and it is the right sibling of nodes p_1, p_2, ..., p_{i-1}. Every node is said to be at level $L \geq 0$. By definition, the root is at level 0. If p is at level k, then nodes p_1, p_2, ..., p_f are each at level k+1. If node s is at a distance of $n \geq 1$ branches from node p and if level of s is greater than level of p, then s is called the successor of p, and p is called the ancestor of s. For the special case when n=1, it is sometimes convenient to refer to s as the son of p, and p as the parent of s. If a node has no sons it is called a leaf node, else it is called a nonleaf node.

In a game, the two players move alternately. We assume that players choose the moves which are the best for them. By convention, player1 moves from nodes at even numbered levels of the tree; i.e., at levels 0, 2, 4, 6.... Nodes at

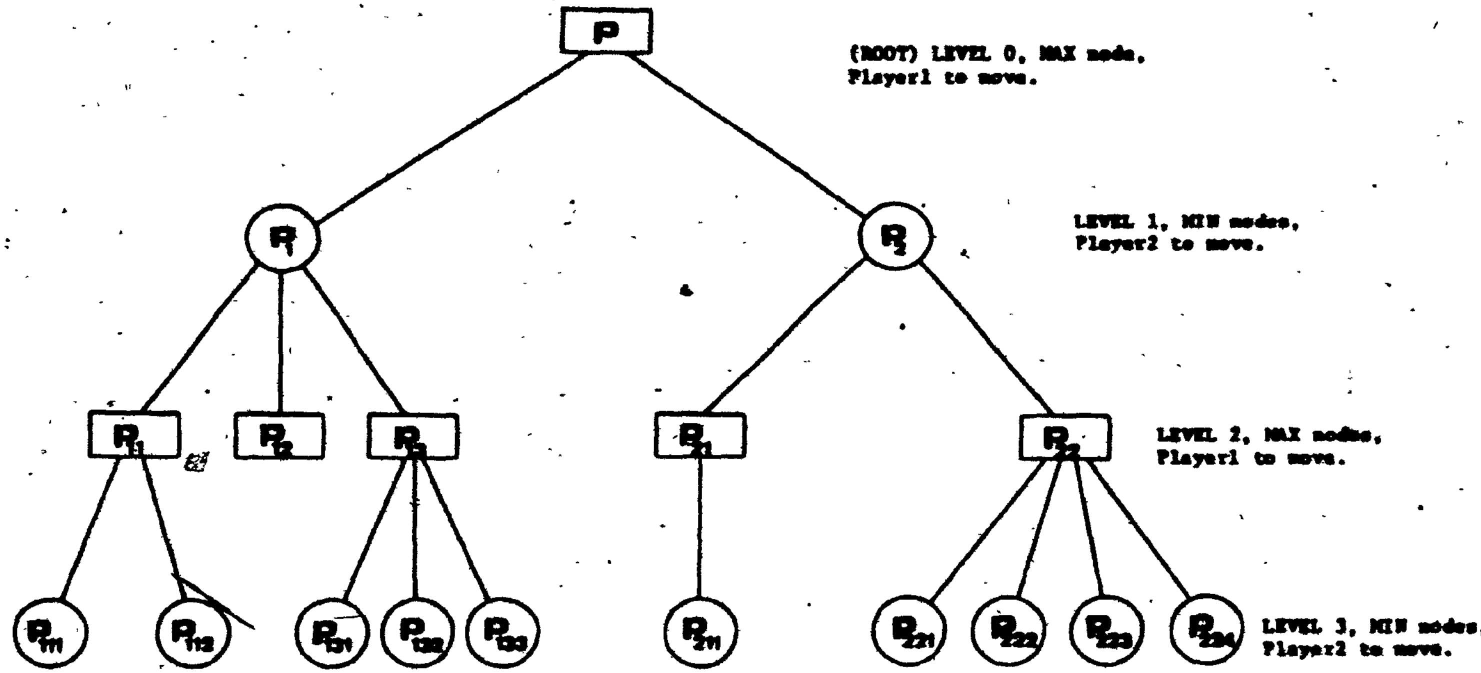


Fig. 1 A specimen game tree.

Below are some examples for the notation defined in the text:

p_1 is a <u>son</u> of p; p is the <u>parent</u> of p_1 .
p_2 is an <u>ancestor</u> of p_{211} ; p_{211} is the <u>successor</u> of p_2 .
The <u>fan-out</u> of node p_{22} is equal to 4.
p_{11} is a <u>left sibling</u> of p_{12} ; p_{12} is a <u>right sibling</u> of p_{11} .
p_{111} , p_{131} , p_{12} , etc. are <u>leaf</u> nodes; p_1 , p_{11} , p_2 , etc. are <u>nonleaf</u> nodes.

these even levels are called MAX nodes. Player2 moves from nodes at odd numbered levels; i.e., at levels 1, 3, 5,.... Nodes at these odd levels are called MIN nodes.

1.2. Searching Procedures for Game Trees.

Any search procedure for the game tree consists of a move-generation procedure, a static evaluation function and a backing-up procedure. The static evaluation function assigns a value to a node without generating any of its sons, hence it assigns values to the leaf nodes of a tree. The value of a leaf node indicates the goodness (or promise) of the corresponding game position from the point of view of one of the players. The move-generating procedure generates all sons for a node. Breadth-first and depth-first are two of many kinds of generation procedures. A breadth-first procedure generates all the nodes at level 1, then at level 2, etc.. A depth-first procedure generates a tree from the left. It starts by generating the leftmost son of a node. If a node is a leaf then all its siblings are generated. Then the procedure generates the next right sibling node for the parent of these siblings, then its leftmost son, and so on. The backing-up procedure assigns to a nonleaf node a value, based on the values of sons of that node.

Game trees are usually searched by the minimax procedure [21]. This procedure combines the depth-first move-generation procedure, the minimax backing-up procedure and an evaluation function. In the minimax search procedure the leaf nodes are assigned values from the point of view of player1. Using this procedure the nonleaf nodes are recursively evaluated; that is, the value of a nonleaf MAX (MIN) node is computed to be the maximum (minimum) value of its sons. If p is a leaf node then its value, calculated by a static evaluation function is denoted as staticvalue(p). The value obtained for player1 at a node p, denoted as BACKVAL(p), is defined as follows :

if p is a MAX node then:

$$\text{BACKVAL}(p) = \begin{cases} \text{staticvalue}(p), & \text{if } p \text{ is a leaf node} \\ \max(\text{BACKVAL}(p_1), \ldots, \text{BACKVAL}(p_r)), & \text{otherwise;} \end{cases}$$

and if p is a MIN node :

$$\text{BACKVAL}(p) = \begin{cases} \text{staticvalue}(p), & \text{if } p \text{ is a leaf node} \\ \min(\text{BACKVAL}(p_1), \ldots, \text{BACKVAL}(p_r)), & \text{otherwise.} \end{cases}$$

Informally, we say that the value of a son is backed-up to its parent node. Thus the value of a nonleaf node p indicates the best that player1 can achieve from the game position corresponding to p. The minimax procedure terminates when it computes the value of the root. The sequence of moves which minimax predicts as optimal for both sides is called the principal continuation. An example of static-values, backed-up values and the principal continuation is presented in Figure 2.

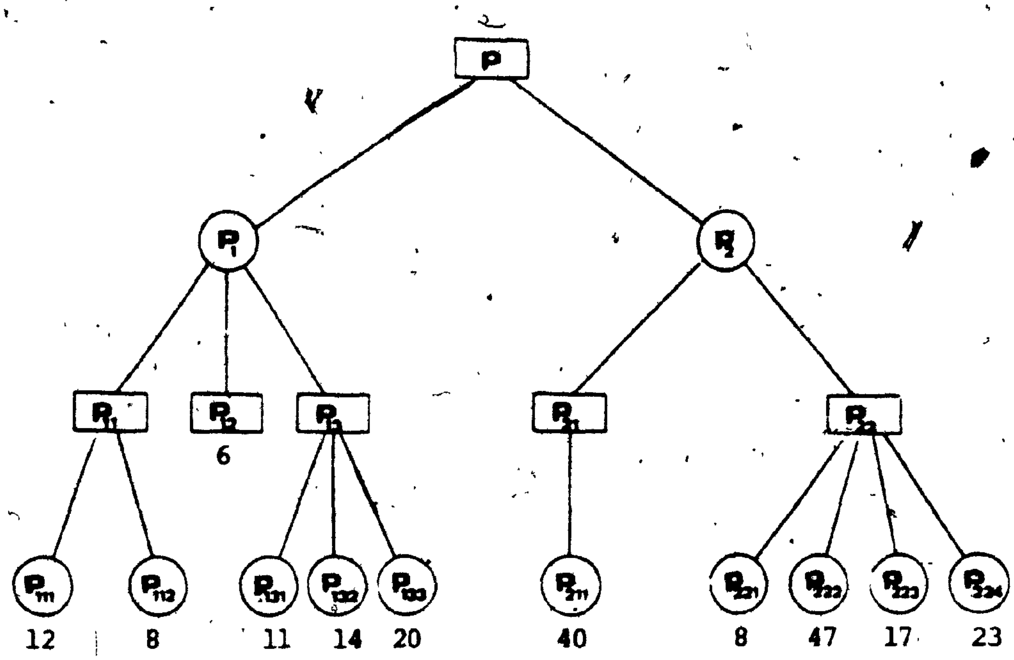


FIGURE 2

An example of static-values, backed-up values and principal continuation.

Let us assume that the leaf nodes have been assigned the example static-values, as shown above :

$p_{111}=12$, $p_{211}=40$,
$p_{112}=8$, $p_{221}=8$,
$p_{12}=6$, $p_{222}=47$,
$p_{131}=11$, $p_{223}=17$,
$p_{132}=14$ $p_{224}=23$.
$p_{133}=20$,

The value of 12 is backed-up to node p_{12} (as maximum of its sons values)
the value of 20 is backed-up to node p_{13} (as maximum of its sons values)
the value of 6 is backed-up to node p_1 (as minimum of its sons values)
the value of 40 is backed-up to node p_{21} (as maximum of its sons values)
the value of 47 is backed-up to node p_{22} (as maximum of its sons values)
the value of 40 is backed-up to node p_2 (as minimum of its sons values).

Finally the value of 40 is backed-up to node p.
The sequence of nodes : p_2, p_{21}, p_{211} represents the principal continuation for this game tree.

A variant of minimax is the negamax procedure, in which the static-value assigned to a leaf node is from the point of view of the player whose turn it is to move. Then, the value computed for any nonleaf node p is the maximum of the negative values of its sons. Alternatively, we can say that the value assigned to a nonleaf node p is the negative of the minimum of its sons values. The value obtained at a node p for player whoes turn it is to move is defined as :

$$BACKVAL(p)=\begin{cases} staticvalue(p), & \text{if } p \text{ is a leaf node} \\ max(-BACKVAL(p_1), \ldots, -BACKVAL(p_f)), & \text{otherwise} \end{cases}$$

or alternatively :

$$BACKVAL(p)=\begin{cases} staticvalue(p), & \text{if } p \text{ is a leaf node} \\ -min(BACKVAL(p_1), \ldots, BACKVAL(p_f)), & \text{otherwise} \end{cases}$$

Negamax does not differentiate between MAX and MIN nodes. The differences between negamax and minimax backed-up values and values assigned to leaf nodes can be seen by comparing Figures 2 and 3. The value computed for the root node is the same by both the minimax and negamax procedures. Thus minimax and negamax are equivalent. For both the minimax and negamax frameworks, we can make the following intuitively understandable statement : if the value of a son p_i is backed-up to its parent p, then p_i represents the best son of p ; that is, the best move for the player at node p is to move to node p_i. The choice of adopting negamax or minimax depends on the pruning strategy and the convenience of implementation selected by the user.

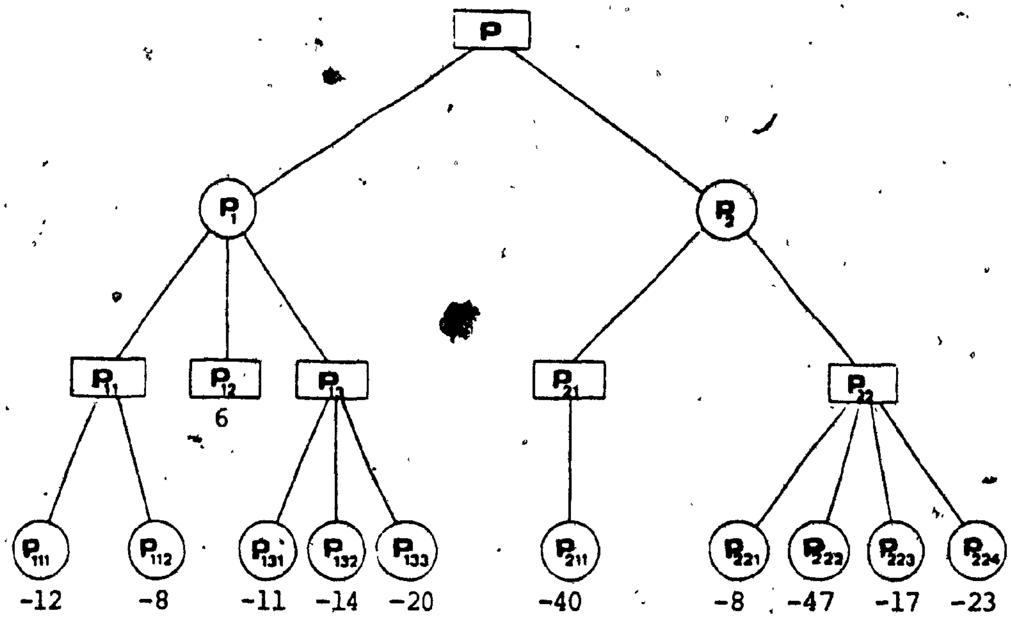


FIGURE 3.

An example of static-values, backed-up values and principal continuation under negamaxing.

The leaf nodes that are at level 3 of a game tree are assigned values which are the negation of the static-values for minimaxing. Because in negamax the values are assigned to leaf nodes from the point of view of the player who makes the move at these nodes, and at level 3 player2 makes the move. Node p_{12} is assigned value of 6, because at level 2 player1 makes the move.

The value of 12 is backed-up to node p_{11} (as -min(-12,-8)),
the value of 20 is backed-up to node p_{13} (as -min(-11,-14,-20)),
the value of -6 is backed-up to node p_1 (as -min(12,6,20)),
the value of 40 is backed-up to node p_{21} (as -min(-40)),
the value of 47 is backed-up to node p_{22} (as -min(-8,-47,-17,-23)),
the value of -40 is backed-up to node p_2 (as -min(40,47)),
finally, the value of 40 is backed-up to node p (as -min(-6,-40)).

Thus negamaxing and minimaxing are equivalent in terms of the value backed-up to the root of a game tree, and in predicting the sequence of nodes in principal continuation. The principal continuation is the sequence: p, p_2, p_{21}, p_{211}.

The game trees have a tendency to become very large, in a sense of total number of nodes in a tree. If the searching procedure is going to do brute-force search, then for example for checkers it has to generate approximately 10^{40} nodes [21]. Thus it is impossible to build a tree representing the whole game. The goal of game-playing programs is to find the best first move, then the next one, and so on. Researchers [12,16,23,30] have made different attempts to devise algorithms which reduce the size of searched trees while still finding the best possible move. In chapter 2 six different pruning strategies are described. All of them aim to create only a fraction of the game tree to compute the value of the root. This value is backed-up from a son towards which a move should be made. The strategies differ in their details but in essence they have one property in common: when it is judged that a node p_i can never change the value of its parent, then further search below p_i can be discontinued; that is, the subtree below p_i may be cut-off. A cut-off may be obtained in many different ways, hence we have different pruning strategies. The way, in which cut-offs are obtained, is described for every pruning strategy presented in chapter 2.

CHAPTER 2.

DESCRIPTION OF DIFFERENT PRUNING STRATEGIES.

In this chapter six different pruning strategies are described. For each strategy an informal description followed by its algorithmic formulation is given and it is also mentioned whether the strategy was implemented under the negamax or minimax framework. According to Kumar and Kanal [13] the pruning strategies can be viewed as special cases of a generalized Branch-and-bound. Comments on this may be found in section 3.7. In the algorithmic formulation, some variables have been declared to be of type NUMERIC. This means that these variables are of a type INTEGER or REAL depending on whether the static-values assigned to leaf nodes are correspondingly integer or real. The following functions: staticvalue(p), which returns a numeric value for the node p, and generate(p), which generates all sons of p and returns the value of fan-out for p, are assumed to exist.

2.1. Branch-and-bound Algorithm (under the negamax framework).

This strategy should in fact be considered as a naive Branch-and-bound, considering Kumar and Kanal's [13] results on generalized Branch-and-bound. However, to be consistent with the name used by Knuth et al. [12], this strategy will be simply called as Branch-and-bound.

In this strategy [12], a provisional value is assigned to a nonleaf node while its sons are being explored. To evaluate a node p, its sons p_1, p_2, ..., p_i are evaluated sequentially. Suppose at a given stage, the values of the nodes p_1, p_2, ..., p_i have been computed. Then we say that the provisional value of their parent p is the maximum of the negative values of p_1, p_2, ..., p_i. The true value of p can only be greater than or equal to its provisional value. If later we observe that the provisional value of node p_{i+1} is $\geq$ the negative of the provisional value of its parent p, then we can safely say that the value of p_{i+1} can never be backed-up to its parent; that is, p_{i+1} can never be the best son of p. So search below p_{i+1} can be cut-off. Thus the provisional value of p acts as a bound for the sons of p. For example a node, say p, has four sons. Two of them have been evaluated and p_1 has value of 3, p_2 has value of 4. So the provisional value of p is -3 after evaluating its two sons. If the provisional value of p_3 is later found to be greater than or equal to -3, then the search below p_3 may be cut-off. Below the recursive algorithmic formulation for the Branch-and-bound strategy is given. It is invoked by calling the function Branchandbound(root,MAXINT), where MAXINT denotes the largest integer value that a computer can store.

```
FUNCTION branchandbound ( p : TREENODE ;
                          bound : NUMERIC ) : NUMERIC ;
VAR  i,f  : INTEGER ; m : NUMERIC ;
BEGIN
  f:=generate(p); /* generate sons p1, p2, ..., pf
                     of node p   */
  IF f=0 THEN return(staticvalue(p)); /* p is leaf node */
  m:=-MAXINT ;
  FOR i:=1  TO  f  DO
   BEGIN
    m:=max(m,-branchandbound(pi ,-m)) ;
    IF m≥ bound THEN return(m); /* cut-off below node pi
                                   and return value of m
                                   as function value */
   END;
  return(m); /* return value of m as  function value */
END.
```

2.2. Alphabeta Algorithm (under the negamax framework).

This strategy [1,4,9,11,12] is an extension of the Branch-and-bound algorithm described above. In the Branch-and-bound algorithm, cut-off took place below a node, when its provisional value was greater than or equal to an upper bound. In Alphabeta algorithm, a cut-off takes place below a node, when its provisional value is $\leq$ a lower bound alpha, or it is $\geq$ an upper bound beta. The interval [alpha,beta] representing the range of values over which the search is to be made is also called the search window. The actual value of the root must lie within the interval (alpha, beta) in order to have a successful search, but with the narrower initial window more cut-offs are obtained. If the value of root is < alpha then we have a case of failing low, if the value of root is > beta then we have a case of failing high. For both cases the search must be repeated, as shown in [12] :

1) IF BACKVAL(root) $\leq$ alpha THEN
   alphabeta(root,alpha,beta) $\leq$ alpha,
2) IF BACKVAL(root) $\geq$ beta THEN
   alphabeta(root,alpha,beta) $\geq$ beta.

Only if alpha < BACKVAL(root) < beta then alphabeta(root,alpha,beta)= BACKVAL(root).

Alphabeta will always examine the same nodes as Branch-and-bound algorithm for the game trees of depth smaller than four [12]. On levels 4, 5, ..., of a game tree Alphabeta is able to make deep cut-offs which can not be obtained by carrying only one bound. The differences between Alphabeta and Branch-and-bound, the deep and shallow cut-offs are shown in Figure 4. The algorithmic formulation

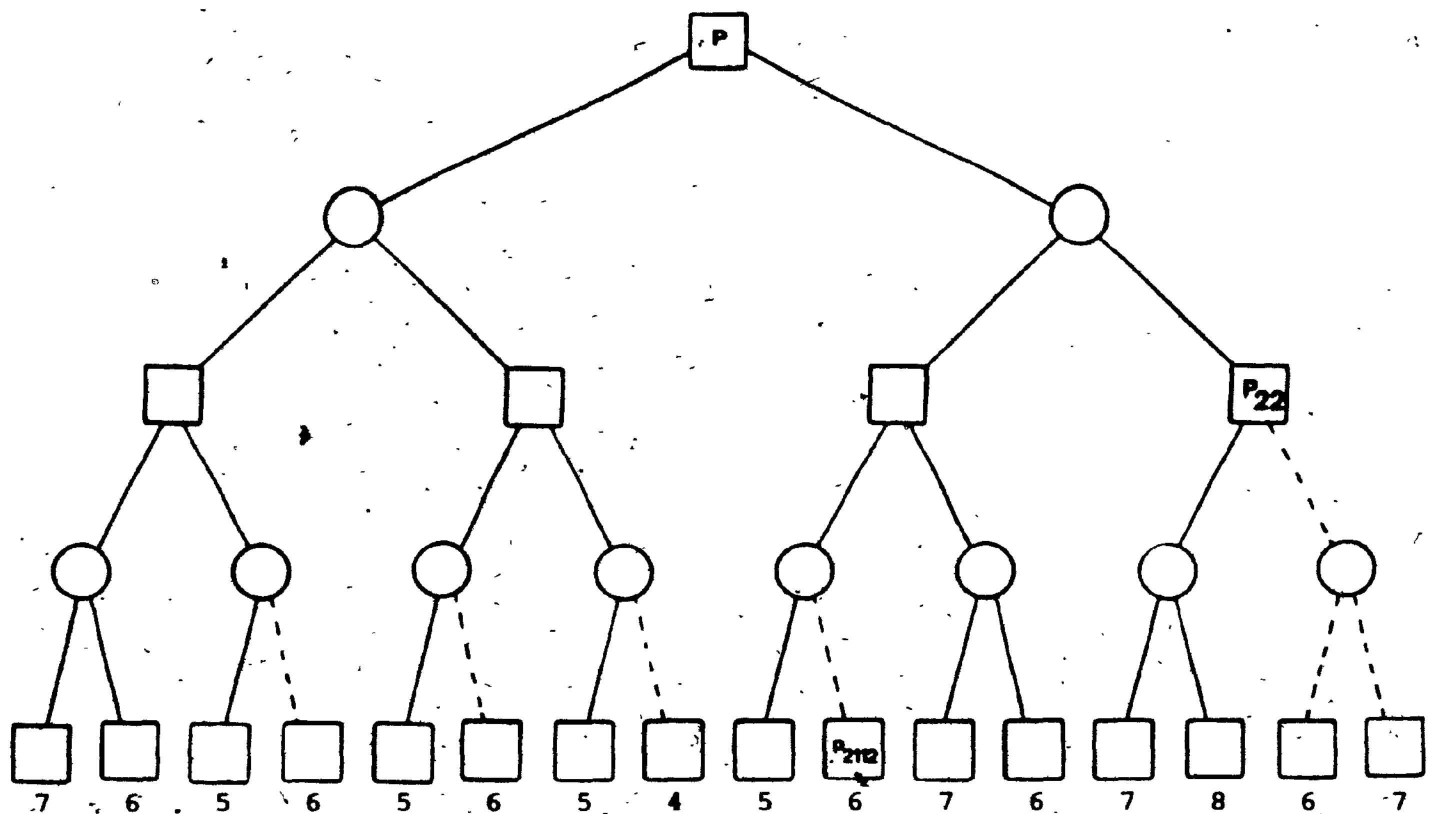


FIGURE 4.

An example to show that Alphabeta prunes more nodes than Branch-and-bound. Leaf node p_{2112} is pruned by Alphabeta, but not by Branch-and-bound. Cut-off obtained at node p_{2112} is called a deep cut-off, cut-off obtained at node p_{22} is called a shallow cut-off.

for the Alphabeta is given below. It is invoked by the function call Alphabeta (root, -MAXINT, MAXINT).

```
1.FUNCTION alphabeta ( p : TREENODE ;
                       alpha, beta : NUMERIC ) : NUMERIC ;
2. VAR  i,f :INTEGER; m : NUMERIC ;
3. BEGIN
4.   f:=generate(p); /* generate sons p1 , p2 , ..., pf
                        of node p  */
5.   IF f=0 THEN return(staticvalue(p)); /* p is leaf node */
6.   m:=alpha;
7.   FOR i:=1 TO f  DO
8.    BEGIN
9.     m:=max(m,-alphabeta(pi ,-beta,-m));
10.    IF m> beta THEN return(m);  /* cut-off below node pi
                                      and return value of m
                                      as function value */
11.   END;
12.  return(m); /* return value of m as  function value */
13.END.
```

<u>2.3. Palphabeta Algorithm (under the negamax framework).</u>

Palphabeta algorithm [9,15] attempts to increase the likelihood of cut-off over the Alphabeta algorithm by tightening the bounds for a node ; that is, by either raising the value of the lower bound or lowering the value of the upper bound. Palphabeta uses a concept of a minimal-window search. First it evaluates the leftmost son p_1 of a node p. Then it invokes, with the window of width 1 (called minimal-window), the function Falphabeta, which indicates whether any of the sibling nodes of p_1 is promising enough to be explored any further; if so, the node is explored under the wider window with the bound returned by Falphabeta. However, each subtree which returns better value then its left siblings must be searched twice. Below the algorithmic formulation for this strategy is given. It is invoked by the function call Palphabeta(root).

```
FUNCTION palphabeta ( p : TREENODE ) : NUMERIC ;
VAR  i,f : INTEGER ; m,t : NUMERIC ;
BEGIN
 f:=generate(p); /* generate sons p_1 , p_2 , ..., p_f
                    of node p    */
 IF f=0 THEN return(staticvalue(p)); /* p is leaf node */
 m:=-palphabeta(p_1);
 FOR i:=2  TO f  DO
  BEGIN
    t:=-falphabeta(p_i ,-m-1,-m);
    IF t>m  THEN  m:=-alphabeta(p_i ,-MAXINT,-t);
  END;
return(m); /* return value of m as  function value */
END.
```

The function falphabeta is similar to the function alphabeta except for two differences :
line 6 of alphabeta becomes m:=-MAXINT, and
line 9 becomes m:=max(m,-falphabeta(p_i ,-beta,-max(m,alpha)))
Falphabeta always examines the same nodes as Alphabeta, but it can give a tighter bound on the true value of the root when the search fails high or low. This is achieved by the bound being taken to be the maximum of the alpha bound and the value of the current best son, as specified in line 9 of the algorithm.

2.4. Principal Variation Algorithm (under the negamax framework).

This strategy, called PVS for short by Marsland [16], is an extension of the Alphabeta algorithm. PVS, just like Palphabeta, also uses the concept of minimal-window search. PVS first evaluates the leftmost son of a node p. Then it explores the other sons under the minimal-window. Note that this window is different than a window used by Palphabeta. It is initialized to the maximum of alpha bound and the value of the current best son. If a son returns promising value (it doesn't fail low) then it is evaluated, but under the tighter bounds than it was done by Palphabeta. The tighter bound for a node is achieved by raising the alpha value of the node p. The game tree processed by the Palphabeta and by PVS is presented in Figure 5. The algorithmic formulation of the PVS is given below. It is invoked by the function call PVS (root,-MAXINT,MAXINT).

```
1.FUNCTION pvs ( p : TREENODE ;
                 alpha, beta : NUMERIC ) : NUMERIC ;
2. VAR  i,f  : INTEGER ; bound,t,m : NUMERIC ;
3. BEGIN
4.   f:=generate(p); /* generate sons p1, p2, ..., pf
                        of node p   */
5.   IF f=0 THEN return(staticvalue(p)); /* p is leaf node */
6.   m:=-pvs(p1,-beta,-alpha);
7.   IF  m < beta  THEN
8.    FOR i:=2 TO f  DO
9.     BEGIN
10.     bound:=max(m,alpha);
11.     t:=-pvs(pi,-bound-1,-bound);
12.     IF  t > m  THEN
13.         m:=-pvs(pi,-beta,-t);
14.     IF m >= beta  THEN
15.        return(m); /* cut-off below node pi and return
                         value of m as function value */
16.    END;
17. return(m); /* return value of m as function value */
18.END.
```

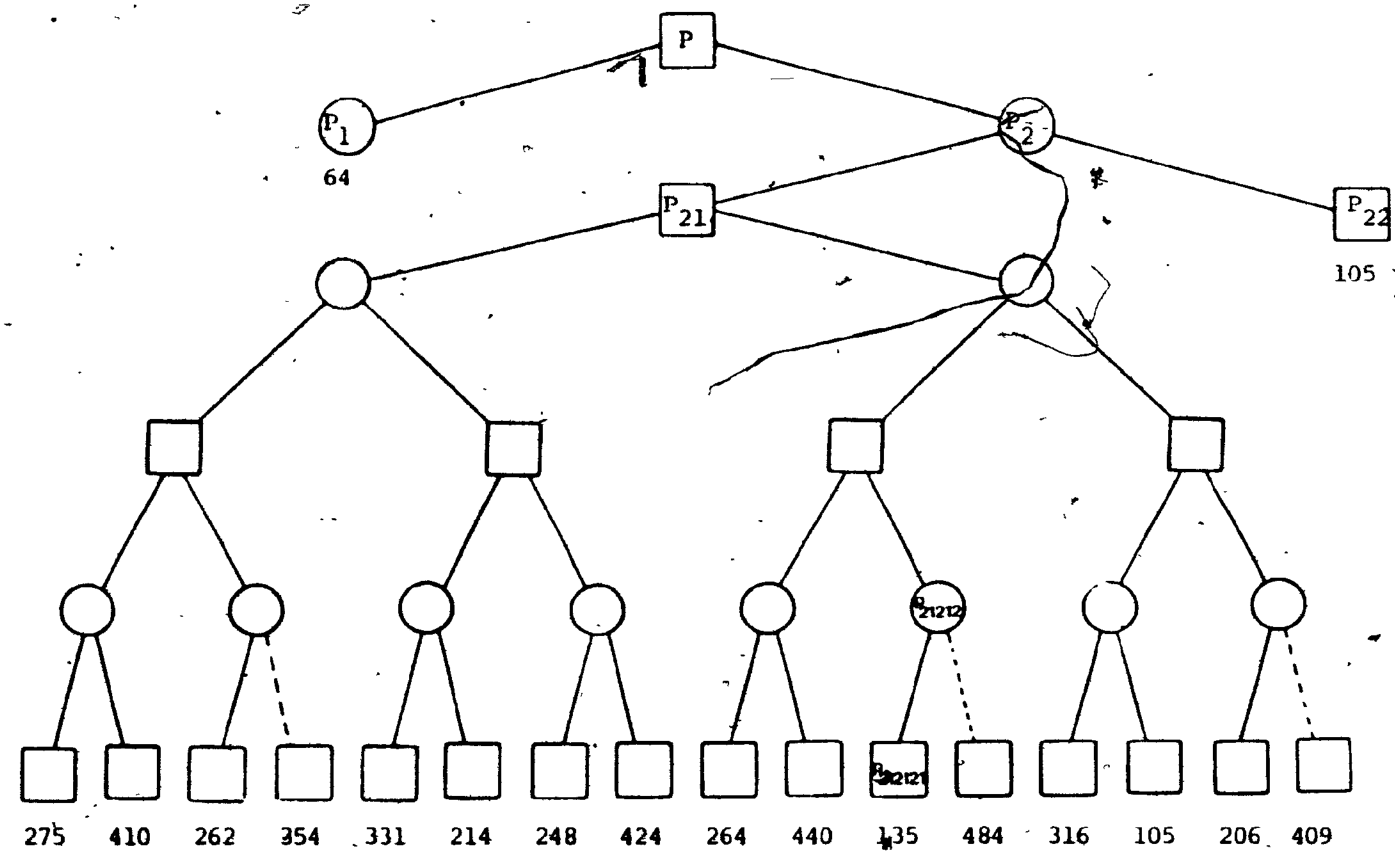


FIGURE 5.

An example in which PVS prunes more nodes than Palphabeta. Nodes p_{21212} and p_{212121} are examined by Palphabeta, but not by PVS because of tighter bounds used by the PVS.

Node p_1 returned the value of 64, and node p_{22}, evaluated after p_{21}, was found to return value of 105.

2.5. Scout Algorithm (under the negamax and minimax framework).

To evaluate a node p, the Scout algorithm [23] first evaluates its son p_1. Node p_1 becomes the current best son. The algorithm then 'scouts' the rest of the sons p_2, p_3, ..., p_f one by one. It invokes Test algorithm, which returns boolean value indicating if a node is worth to be evaluated. If a son p_i does not appear to retun a more promising value than the current best son, search below p_i is cut-off. Otherwise, p_i is evaluated, and p_i becomes the current best node. Test does not return a bound which may be used in further search, so Scout may examine more nodes than Palphabeta or PVS. An example of such a situation is shown in Figure 6. Pearl had initially proposed this algorithm under the minimax framework [23]. Campbell and Marsland [9] used Alphabeta instead of Test algorithm in their negamax version of Scout algorithm. Algorithmic formulation of Scout under negamax framework, Scout which invokes Test, is presented below. Comparison of three different versions of Scout : minimax, negamax and the Campbell-Marsland version is discussed in appendix 1. The presented function is invoked by the call Scout(root).

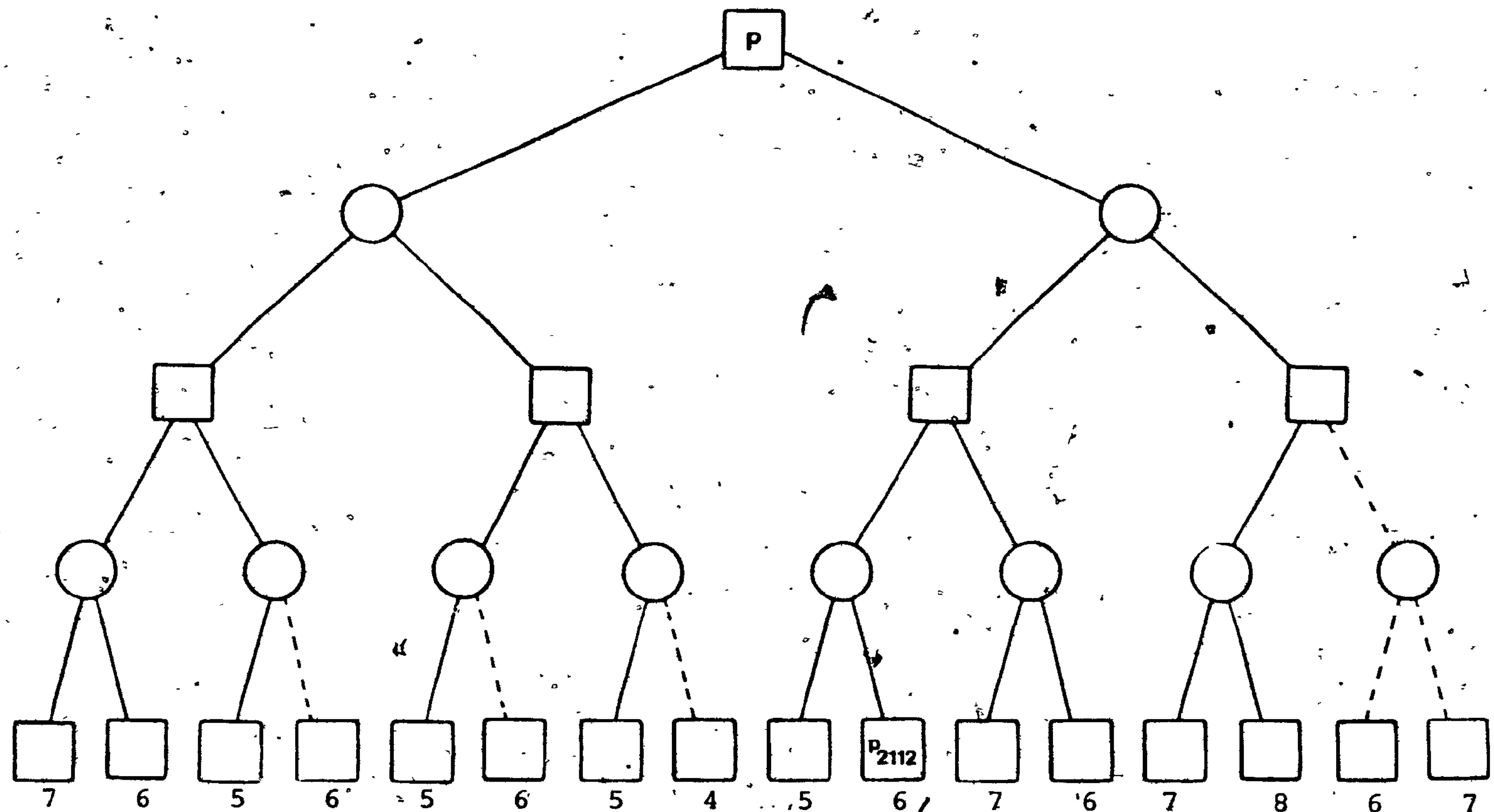


FIGURE 6.

An example in which Scout examines more nodes than Palphabeta or PVS. Node P_{2112} is examined by Scout but not by Palphabeta or PVS, because the Test algorithm, invoked by Scout does not return a bound which may be used in further search, performed by Scout.

```
FUNCTION scout ( p : TREENODE ) : NUMERIC ;
VAR  i,f : INTEGER ; m : NUMERIC ;  op : BOOLEAN ;
BEGIN
  f:=generate(p); /* generate sons p1, p2, ..., pf
                       of node p   */
  IF f=0 THEN return(staticvalue(p)); /* p is leaf node */
  m:=-scout(p1);
  op:=TRUE ; /* parameter used to compare nodes in
                function test invoked by scout   */
  FOR  i:=2  TO  f  DO
     IF (NOT test(pi,-m,not op))  THEN  m:=-scout(pi);
 /* if function test returns false, scout evaluates node pi ,
  else it is not evaluated */
  return(m); /* return value of m as the function value */
END.
```

```
1.FUNCTION test (p :TREENODE ; v :INTEGER; op :BOOLEAN)
              : BOOLEAN;
/* if op is true nodes to be compared are at same level
   of the tree, else at different levels  */
2. VAR i,f : INTEGER ;
3. BEGIN
4. f:=generate(p); /* generate sons p1, p2, ..., pf
                      of node p   */
5. IF f=0  THEN   /*  p is a leaf node  */
6.   IF ((staticvalue(p) >= v)  AND  ( op ))  OR
7.      ((staticvalue(p) > v)  AND  (not op))  THEN
8.        return TRUE /* node p can not be the best son */
9.   ELSE return FALSE; /* node p may become the best son */
10. FOR i:=1  TO  f DO
11.   IF NOT test(pi ,-v,not op)  THEN
12.       return TRUE; /* node pi can not become
                         the best son  */
13. return FALSE;
14.END.
```

2.6. SSS* Algorithm (under the minimax framework).

Stockman [30] developed his SSS* algorithm based on the A* algorithm given by Nilsson [21]. SSS* traverses <u>solution trees</u>, where a solution tree S of a game tree G is defined as follows :

a) the root of G is in S;

b) if a nonleaf MIN node of G is in S, then all of its sons are in S; and

c) if a nonleaf MAX node of G is in S, then exactly one of its sons is in S.

A solution tree S represents the way player1 can play, specifying one response to each of the opponent's moves. VAL(S), the value of S is defined to be minimum value over all the leaf nodes in S. It was shown in [27,30] that the minimax value of the root of the game tree G is equal to the maximum of VAL(S) over all solution trees S in G. In Figure 7 the example of solution trees and their values are presented for the specimen game tree from Figure 1. Associated with every node p in G is a triple <p,s,m>, where s and m are, respectively, called the <u>status</u> and <u>merit</u> of p; s∈{LIVE,SOLVED} and m∈[-INFINITY,+INFINITY]. If p has the status SOLVED, it means p has been evaluated. Otherwise it has status LIVE and it is waiting to be evaluated. The value of merit is defined only for nodes which are examined by SSS*. For evaluated nodes of a certain solution tree S, the value of merit is equal to the VAL(S). The algorithm can then be formulated as follows :

1. Put the triple <root,LIVE,MAXINT> on a list called OPEN.
2. Remove from OPEN the topmost triple <p,s,m>. /* The triples in OPEN are kept in non-decreasing order of merit, such that the triple with the highest merit is at the <u>top</u> of the list OPEN. If two nodes p_i , and p_j have equal merit and if p_i is to the left of p_j in the game tree, then triple <p_i ,s,m> appears above triple <p_j ,s,m> in OPEN. Thus every triple is said to be in its <u>proper sorted position</u> in OPEN. As argued by Campbell and Marsland [9], this ensures that the SSS* dominates the Alphabeta algorithm. */
3. If p=root and s=SOLVED, then terminate the algorithm with m being equal to the minimax value of the root. Otherwise continue.
4. Call Gamma (<p,s,m>). /* this procedure traverses through the solution trees which contain node p */
5. Go to 2.

```
PROCEDURE gamma (p:TREENODE, s:[LIVE,SOLVED], m:NUMERIC);
VAR  i, f : INTEGER;
BEGIN
 IF  s = LIVE   THEN /* p is to be evaluated */
   IF  p  leaf  THEN
     insert <p,SOLVED,min(m,staticvalue(p)> in
     its proper sorted position in  OPEN
   ELSE  /* p is nonleaf  */
     IF  p  is  MIN  node  THEN
       put <leftmost-son-of-p,s,m> at the top of OPEN ;
     ELSE  /* p is  MAX node  */
       FOR i:=f  DOWNTO  1   DO
        put <p_i ,s,m> at the top of OPEN /* f is the
                fan-out of p. Nodes p_1 , p_2 , ...p_f
                are sons of p_i  */
 ELSE  /*  s = SOLVED   */
     IF p is MIN node  THEN
       put <parent-of-p,s,m> at the top of OPEN and
       delete from OPEN all triples associated with
       the successors of the parent-of-p ;
     ELSE  /*  p is  MAX node  */
       IF p is the rightmost sibling  THEN
         put <parent-of-p,s,m> at the top of OPEN
       ELSE  /* p is not the rightmost sibling  */
         put <next-right-sibling-of-p,LIVE,m> at the top
         of OPEN ;
END.
```

The way in which SSS* processes the specimen game tree of Figure 7 is shown in Figure 8.

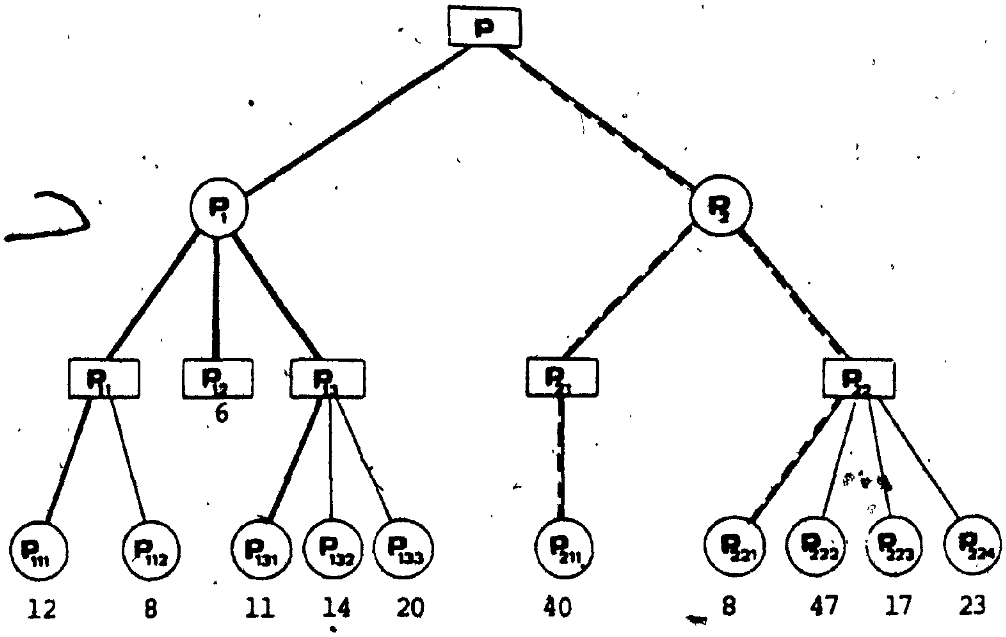


FIGURE 7

For this specimen game tree there are 10 different solution trees. The examples of solution trees are shown one in boldface, another one in broken lines.
Values for the example solution trees are 6 for the first one, and 8 for the second one.
Values for the rest of solution trees are, from the left of a game tree: 6, 6, 6, 6, 6, 40, 17, 23. Based on the theorem stated by Stockman[30] and by Roizen and Pearl[28] the minimax value of this game tree is equal to 40.

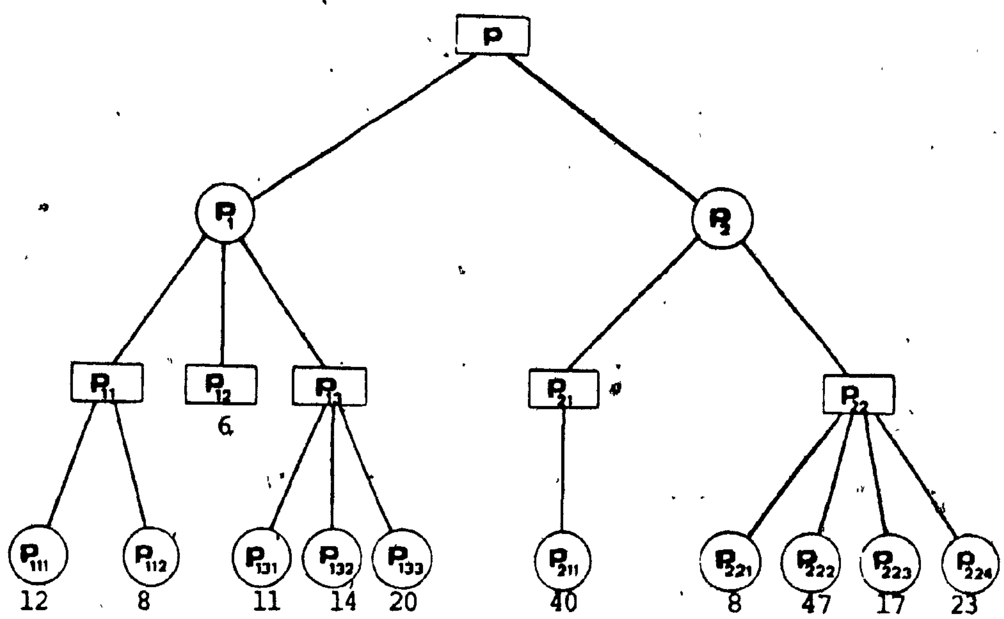


FIGURE 8

A specimen game tree processed by SSS* algorithm.

No.	Changes in the list OPEN
1.	(p,L,∞)
2.	(p_1,L,∞),(p_2,L,∞)
3.	(p_{11},L,∞),(p_2,L,∞)
4.	(p_{111},L,∞),(p_{112},L,∞),(p_2,L,∞)
5.	(p_{112},L,∞),(p_2,L,∞),(p_{111},S,12)
6.	(p_2,L,∞),(p_{111},S,12),(p_{112},S,8)
7.	(p_{21},L,∞),(p_{111},S,12),(p_{112},S,8)
8.	(p_{211},L,∞),(p_{111},S,12),(p_{112},S,8)
9.	(p_{211},S,40),(p_{111},S,12),(p_{112},S,8)
10.	(p_{21},S,40),(p_{111},S,12),(p_{112},S,8)
11.	(p_{22},L,40),(p_{111},S,12),(p_{112},S,8)
12.	(p_{221},L,40),(p_{222},L,40),(p_{223},L,40),(p_{224},L,40),(p_{111},S,12),(p_{112},S,8)
13.	(p_{222},L,40),(p_{223},L,40),(p_{224},L,40),(p_{111},S,12),(p_{112},S,8),(p_{221},S,8)
14.	(p_{222},S,40),(p_{223},L,40),(p_{224},L,40),(p_{111},S,12),(p_{112},S,8),(p_{221},S,8)
15.	(p_{22},S,40),(p_{111},S,12),(p_{112},S,8)
16.	(p_2,S,40),(p_{111},S,12),(p_{112},S,8)
17.	(p,S,40) Minimax value of root is 40.

Nodes cut-off are : p_{12}, p_{13}, (so also p_{131}, p_{132}, p_{133})

CHAPTER 3.

EMPIRICAL COMPARISON OF PRUNING STRATEGIES.

To compare empirically the performance of the different pruning strategies described in chapter 2 above, these strategies were tested on various kinds of simulated game trees using different techniques to assign static-values to leaf nodes. All the programs were coded in Pascal version 3.6 and implemented on a Control Data Cyber 170/835 at Concordia University, Montreal. Below details of the criteria used to compare the performance of the different pruning strategies, the kinds of trees simulated, the methods of assigning static-values to leaf nodes, and the scope of the experiments are given. Some theoretical results for complexity of the tree-pruning strategies are also discussed.

3.1. Criteria Used for Performance Evaluation.

Some researchers [2,9,10,12,21,23,27,27,28] have discussed the comparison of pruning strategies based on the number of leaf nodes created; the fewer the leaf nodes created, the better being the strategy. A node is considered created, if it is not pruned off. One can argue that the criteria of leaf nodes created may not be enough because this places more emphasis on the nodes at the deepest level of the tree. Thus, one could also compare the pruning strategies based on all nodes (leaf and nonleaf) created. However, even this may not be enough. Certain strategies, for example PVS, may prune a larger number of nodes but they may visit the created nodes more than once, thus slowing down the pruning. Hence the number of node-visits can be another criterion. Another important criterion is the CPU time taken by the different pruning strategies. This criterion, however, may be questioned because it depends on the efficiency of program-coding. Under above consideration the empirical comparison of pruning strategies is based on the :

i) average number of all nodes created,

ii) average number of leaf nodes created,

iii) average number of node-visits,

iiii) average CPU time taken.

These criteria were tested for every simulated game tree.

However, the computational effort of game-playing programs is in three basic operations :

i) move generation,

ii) static-evaluation of leaf nodes, and

iii) move selection or minimaxing.

The cost of move generation can vary from game to game. The cost of assigning static-values to leaf nodes depends on the complexity of the function used to assign such values and on the number of leaf nodes created. In our simulation (as discussed in section 3.3), the static-values were assigned by a random number generator. So when we are comparing the different pruning strategies we are restricting ourselves mainly to the cost of minimaxing.

3.2. Kinds of Game Trees Simulated.

Both uniform and nonuniform trees, which will be now defined, were simulated. In a uniform tree U(w,d), the fan-out for every node is equal to w, and all the leaf nodes are at level d. In a nonuniform tree N(w,d), the fan-out of any node can be utmost equal to w, and the level of any leaf node can be utmost equal to d. For both kinds of trees, parameters w and d are, respectively, called the width and depth of the tree. As an illustration, the specimen tree given in Figure 1 is a nonuniform tree of width 4 and depth 3. In simulating a nonuniform trees with the predefined width and depth the actual fan-out of any node was controlled by a uniform random number generator. Nodes with the zero fan-out were considered to be leaf nodes. Constrained by the amount of the computer memory available both uniform and nonuniform trees were simulated with the following parameter values :

depth	width
2	2, 3, 4, 5, 6, 8, 10, 24
3	2, 3, 4, 5, 6, 8, 10
4	2, 3, 4, 5
5	2, 3, 4
6	2, 3

Thus there were 24 different tree-sizes for each of uniform and nonuniform trees, giving 48 classes of trees. In the experiments, 50 trees were simulated for each of the 48 classes.

3.3. Methods of Assigning Static-Values to Leaf Nodes.

In the game-playing programs values at leaf nodes are estimated by some evaluation function. Static-values can be assigned to leaf nodes using a dependent or independent scheme [10,12,20], the details of which will be now given.

Dependent scheme : For this scheme, Newborn [20] discusses two approaches to assign initial values to all nodes in the tree. In the first approach (called integer-dependent approach), sibling nodes p_1, p_2, ...p_f are assigned distinct values from the set $\Theta=\{1, 2, ..., f\}$. In the second approach (called real-dependent approach), set $\Theta=\{1/f^L, 2/f^L, ..., f/f^L\}$, where L is the level of the nodes to which the values are being assigned. As the names of the two approaches imply, the first approach assigns integer values and the second assigns real values. For both approaches, we then compute

static-value-of-a-leaf-node-p = the-assigned-value-of-p + the-summation-of-the-values-assigned-to-all-ancestors-of-p.

Independent scheme : There are two approaches to this scheme. In the first approach [12] (called unordered-independent approach), distinct values from the set {1, 2, ..., M} are assigned as static-values to the leaf nodes, where M is the number of leaf nodes. Thus each of the M! orderings of the values are equally likely. In the second approach (called P-ordered-independent approach), we first arbitrary choose a large positive integer K and a value for P∈[0,1]. Static-values are then assigned to leaf nodes from the range [1,K], such that the probability of any node having its leftmost son as the best son is P. The values of M, K, and P chosen for our experiments are mentioned in section 3.5.

3.4. Some Theoretical Results for Complexity of the Tree-Pruning Strategies.

Many researchers [4,10,12,20,23,24,26,27,28,29,30,31] have analyzed the pruning strategies in order to determine which strategy is optimal over the others (in sense of nodes created), and to test the behaviour of strategies for different schemes of static-values assignment. If by $C_{ST}(w,d)$ we denote the average number of leaf nodes created by a pruning strategy ST for a tree of width w and depth d then the branching factor for this strategy is defined as :

$$R(w) = \lim_{d\to} \sqrt[d]{C_{ST}(w,d)}$$

The average number of leaf node created and the branching factor were used as the criteria for the complexity of a strategy. The most frequently analyzed strategy is the Alphabeta algorithm. Slage and Dixon [29] have shown that the number of leaf nodes created by Alphabeta must lie between two bounds :

$$w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1 \leq C_{AB}(w,d) \leq w^d .$$

Knuth and Moore [12] have shown that there is always a way of ordering nodes such that Alphabeta will not examine more nodes than the lower bound, $w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1$. Alphabeta achieves the lower bound of the number of leaf nodes created for the case when the leftmost son of any nonleaf node is the node's best son. For such ordering, called the perfect-ordering [29], any pruning strategy, except Branch-and-bound, achieves the lower bound of the number of leaf nodes created. Algorithms such as Branch-and-bound or Alphabeta are directional, they never examine a node to the left of one previously examined. Other algorithms, such as SSS*, are non-directional, there is no 'left-to-right' arrangement of the leaf nodes they visit. Pearl [23] has shown that $R^* = \xi_w/(1-\xi_w)$, where ξ_w is the positive root of the $x^w + x - 1 = 0$, is the lower bound for the branching factor of every directional algorithm for uniform trees with continuous static-values. For uniform trees with discrete static-values, the lower bound for the branching factor of any pruning strategy is $w^{1/2}$, as given in [24]. Pearl has shown that his algorithm, Scout, achieves these lower bounds for uniform trees. Baudet [4] and Pearl [24] have independentlly proved that the branching factor of the Alphabeta algorithm also achieves these lower bounds for the uniform trees. Tarsi [31] has later shown that the R* and $w^{1/2}$ are the lower bounds for the branching factor of non-directional algorithms searching uniform game trees.

In [28] the theoretical formulas for the branching factor of SSS* algorithm were given and compared to that of Alphabeta. Roizen et al. [28] have shown that for uniform trees with discrete or continuous static-values the branching factor of the Alphabeta algorithm is equal to that of the SSS* algorithm. In [26] the corollary that the Alphabeta has the lowest branching factor over all algorithms that search uniform game trees with unordered-independent static-values assignment was stated and proved by Pearl. But the optimality of the Alphabeta for searching a real-world game tree is not quaranteed by this corollary because the branching factor quantifies only the rate of growth of $C_{AB}(w,d)$ as d tends to ∞. The analytical results [26, 27, 28] of the average number of leaf nodes created by the Alphabeta, SSS* and Scout algorithms for uniform trees with unordered-independent static-values assignment show that :

$C_{AB}(w,d)=AB(w,d)[R^*(w)]^d$,

$C_{Sc}(w,d)=Sc(w,d)[R^*(w)]^d$,

$C_{SS}(w,d)=SS(w,d)[R^*(w)]^d$.

Over the range $2 \leq w \leq 20$, $2 \leq d \leq 20$ variables AB(w,d) Sc(w,d) and SS(w,d) satisfy :

$4.2 > Sc(w,d) > AB(w,d) > S(w,d) > 1.2$.

So we see from the theoretical analysis above that the SSS*, Alphabeta and Scout algorithms have very similar performance characteristics for the uniform trees with unordered-independent static-values assignment in terms of number of leaf nodes created.

In section 3.6 the performance of six pruning strategies under four different schemes of assigning static-values to leaf nodes for uniform and nonuniform trees will be discussed. The results of comparison regard the number of nodes created, number of node-visits and time of execution. The conjecture which strategy will be popular in game-playing programs is given in section 3.7. The scope of experiment is described in more detail in the following section.

3.5. Scope of the Experiments.

The pruning strategies described in chapter 2 were tested on both uniform and nonuniform trees. For each kind of tree, static-values were assigned to leaf nodes using both the dependent schemes and both the independent schemes. For the P-ordered-independent approach, the value of K was 500 and the probability P ranged from 0.2 to 1.0 in steps of 0.2. For any nonleaf node this probability indicates the likelihood that the node's leftmost son is its best son.

For uniform trees the unordered-independent scheme was implemented as specified in section 3.3. For nonuniform trees the unordered-independent scheme could not be implemented in the manner described in section 3.3 because the number of leaf nodes kept varying. Moreover, the leaf nodes in a given tree occurred at different levels. So in our simulation, the leaf nodes were assigned static values from an arbitrarily selected set {1, 2, ..., 500}.

Dependent schemes for uniform and nonuniform game trees were implemented exactly as specified in section 3.3.

Considering the different kinds of trees and the different approaches to assign static-values to leaf nodes, there were sixteen cases for the experiments. Thus, say, all uniform trees with unordered-independent static-values-assignment approach comprised one case, the trees being of the 24 different sizes mentioned in section 3.2. Note that as mentioned above there were 50 trees of each size. Thus the sample size for each case was 24 * 50 = 1200 trees. The sixteen cases are listed in Table I. Details of Table I are given in the following section, when the experimental results are discussed.

3.6. Results of the Experiments.

To specify that in a given environment, pruning strategy S_i performed better than S_{i+1} for i = 1 , 2, ..., 5, we write them as a list enclosed in parentheses like (S_1 , S_2 , ..., S_6). If within this list notation we write the names of some strategies enclosed in brackets, it means that those strategies performed equally well. Say, we write (S_1, [S_2, S_3], S_4, S_5, S_6). Then it means that S_2 and S_3 performed equally well. They were worse than S_1 but better than S_4.

First the performance of the six pruning strategies based on average number of all nodes created (leaf and nonleaf), the fewer the nodes created the better being the performance, will be compared. Then the results for pruning strategies under the criterion of average number of leaf nodes created are presented and discussed. Next, the results of comparison under the criterion of average number of node-visits and average CPU time taken, are given for every discussed strategy.

3.6.1. Comparison Based on Number of All Nodes Created.

For trees of depth greater than or equal to 4, Table I shows the comparative performance of the pruning strategies for all the sixteen cases, which we mentioned in section 3.5. We see that SSS* consistently creates the least nodes and Branch-and-bound the most. The other strategies fall in between, with PVS among them usually performing the best, except for nonuniform trees with real-dependent static-values assignment, in which case PVS performed slightly worse than Scout. PVS, Palphabeta and Scout usually perormed better than Alphabeta. For example for U(3,6) with 0.2-ordered-independent scheme SSS* creates on average 316.82 nodes, PVS 377.94, Palphabeta 380.60, Scout 399.00, Alphabeta 414.94 and Branch-and-bound creates on average 523.96 nodes. The exception ooccurs for uniform trees when static-values are assigned either by 1.0-ordered-independent scheme or by one of dependent schemes (cases 6, 7 and 8 of Table I). For the 1.0-ordered scheme, all pruning strategies except Branch-and-bound performed equally well, for example for U(3,6) branch-and-bound creates on average 168.00 nodes and all other strategies create 124.00 nodes. For the

Case no.	Type of tree	Method of assigning static values to leaf nodes	Ranking of the pruning strategies in decreasing order of their performance
1.	Uniform	unordered-indep.	(SSS*, PVS, PAB, Scout, AB, BB)
2.	Uniform	0.2-ordered-indep.	
3.	Uniform	0.4-ordered-indep.	
4.	Uniform	0.6-ordered-indep.	
5.	Uniform	0.8-ordered-indep.	
6.	Uniform	1.0-ordered-indep.	([SSS*, PVS, PAB, Scout, AB], BB)
7.	Uniform	integer-dependent	(SSS*, PVS, PAB, AB, Scout, BB)
8.	Uniform	real-dependent	(SSS*, [PVS, PAB, Scout, AB], BB)
9.	Nonuniform	unordered-indep.	(SSS*, PVS, PAB, Scout, AB, BB)
10.	Nonuniform	0.2-ordered-indep.	
11.	Nonuniform	0.4-ordered-indep.	
12.	Nonuniform	0.6-ordered-indep.	
13.	Nonuniform	0.8-ordered-indep.	
14.	Nonuniform	1.0-ordered-indep.	
15.	Nonuniform	integer-dependent	(SSS*, [PVS, PAB], Scout, AB, BB)
16.	Nonuniform	real-dependent	(SSS*, Scout, [PVS, PAB], AB, BB)

TABLE I.

Ranking of pruning strategies under the criterion of all nodes created (leaf and nonleaf) for trees of depth ≥ 4. We observe that SSS* is consistently the best and Branch-and-bound the worst. Strategies enclosed in brackets performed equally well for that case. Note : the ranking of strategies remains same under the criterion of only leaf nodes created.

Legend :

AB - Alphabeta [12]
BB - Branch-and-bound [12].
PAB - Palphabeta [9]
PVS - Marsland's Principal Variation Search [16]
Scout - Pearl's Scout [23]
SSS* - Stockman's State Space Search [30]

real-dependent scheme, PVS, Palphabeta, Scout and Alphabeta performed equally well, with SSS* performing better than all of them. For example for U(3,6) SSS* creates on average 150.80 nodes, Branch-and-bound 346.48 and all other strategies 263.84 nodes. In the case of integer-dependent scheme, Scout performed slightly worse than Alphabeta. For example for U(3,5) Alphabeta created on average 149.94 nodes and Scout 151.02 nodes.

For trees of depth equal to 3, for all sixteen cases except uniform and nonuniform trees with real-dependent static-values assignment, the strategies can be ranked as (SSS*, [PVS, PAB, Scout],[AB, BB]). For example for U(5,3) with unordered-independent static-values assignment SSS* created on average 77.36 nodes, PVS, Palphabeta and Scout 86.32 nodes and Alphabeta and Branch-and-bound 93.18 nodes. For the uniform trees with real-dependent static-values assignment the ranking is (SSS*, [PVS, PAB, Scout, AB, BB]). For example for U(8,3) SSS* created on average 108.14 nodes and all others 159.10 nodes. For the nonuniform trees with real-dependent static-values assignment the ranking is (SSS*, Scout, [PVS, PAB, AB, BB]). For example for N(10,3) SSS* created on average 44.28 nodes, Scout 59.06 nodes and all other pruning strategies 60.82 nodes. Here again, SSS* has always performed the best.

For trees of depth equal to 2, the strategies can be ranked as (SSS*, [PVS, PAB, Scout, AB, BB]), for all of the sixteen cases. For example for N(24,2) with unordered-independent static-values assignment SSS* created on average 131.04 nodes and all other pruning strategies created on average 167.16 nodes.

For all trees of depth equal to or smaller than 3, Alphabeta and Branch-and-bound always performed identically, as expected, based on the discussion by Knuth and Moore [12]. Overall, we notice that SSS* always created the fewest nodes, which confirms the theoretical results of Stockman [30] and Roizen and Pearl [28].

3.6.2. Comparison Based on Number of Leaf Nodes Created.

In Figures 9 to 16, for uniform trees of depth four, we have plotted the average number of leaf nodes created by the different strategies versus the width of tree. In Table II some results from Newborn [20] are given for comparison. Furthermore, in Tables III to X the average number of leaf nodes created in uniform trees by the different strategies are reported. Under unordered-independent and integer-dependent approaches to assigning static-values, Newborn had theoretically estimated the expected number of leaf nodes to be created by the Alphabeta algorithm.

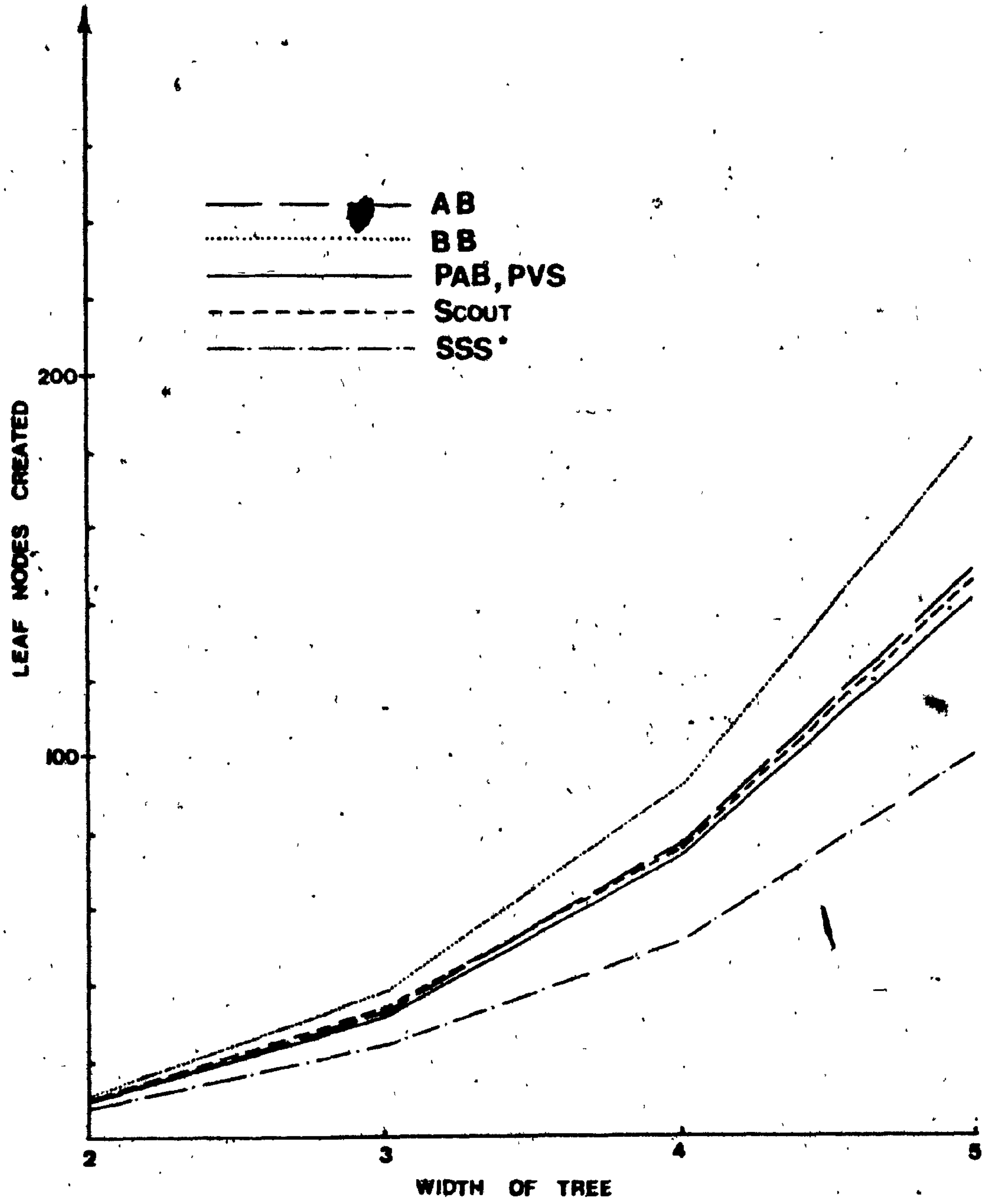


FIGURE 9

Plot of average number of leaf nodes created against width of a uniform tree with depth 4. Static values were assigned to leaf nodes by integer - dependent scheme.

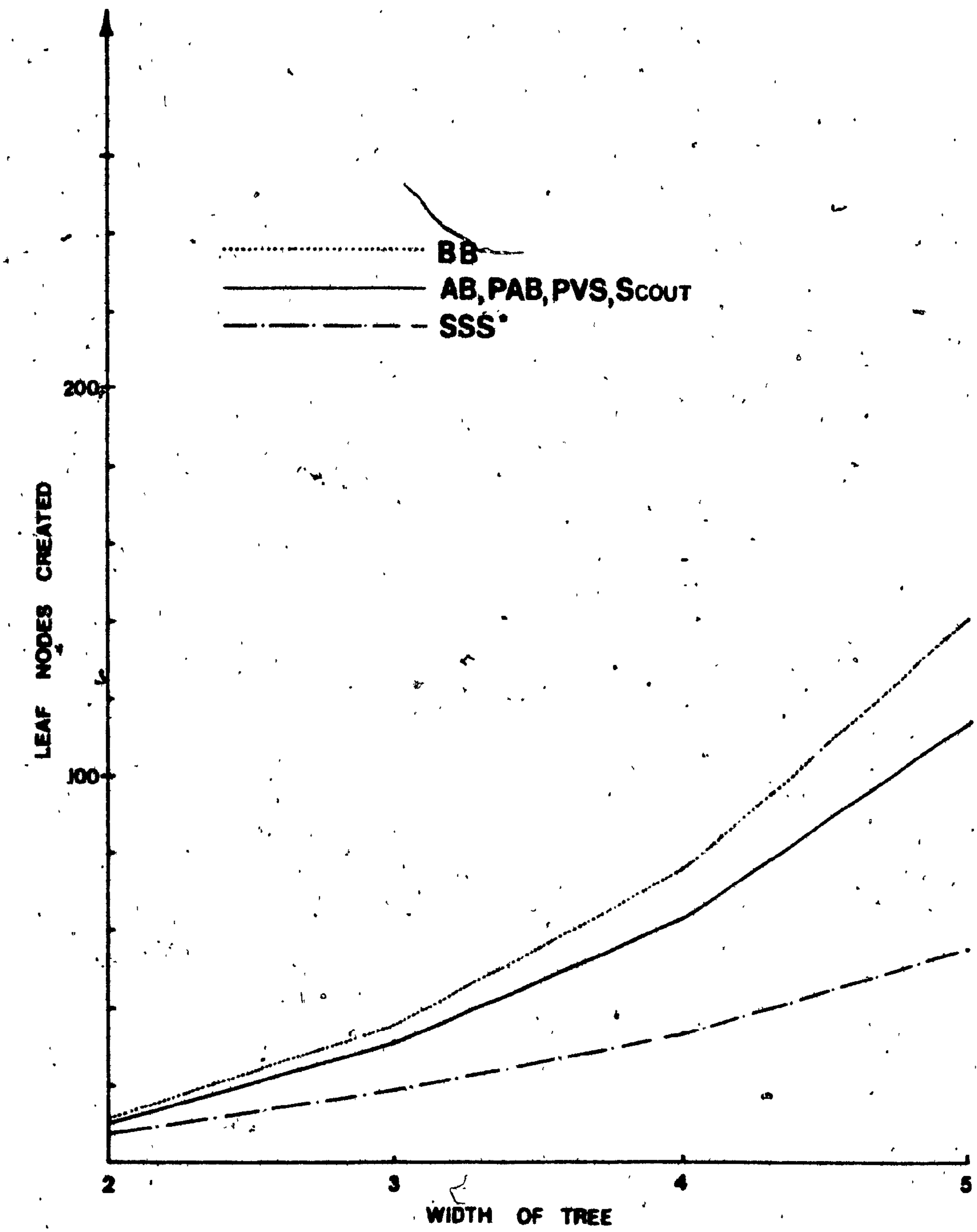


FIGURE 10.

Plot of average number of leaf nodes created against width of a uniform tree with depth 4. Static values were assigned to leaf nodes by real - dependent scheme.

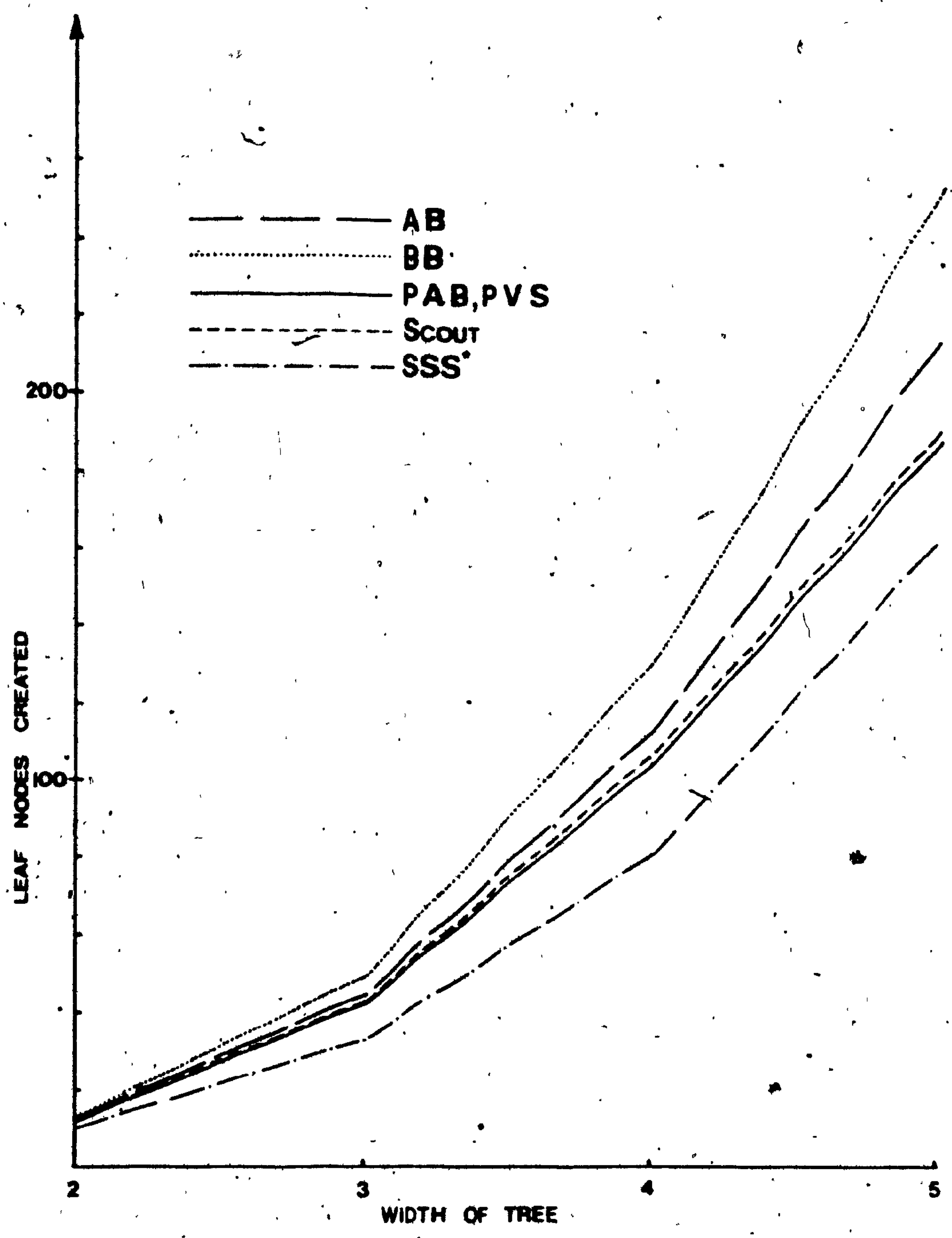


FIGURE 11.

Plot of average number of leaf nodes created against width of a uniform tree with depth 4. Static values were assigned to leaf nodes by unordered-independent scheme.

FIGURE 12.

Plot of average number of leaf nodes created against width of a uniform tree of depth 4. Static-values were assigned to leaf nodes by 0.2-ordered-independent scheme.

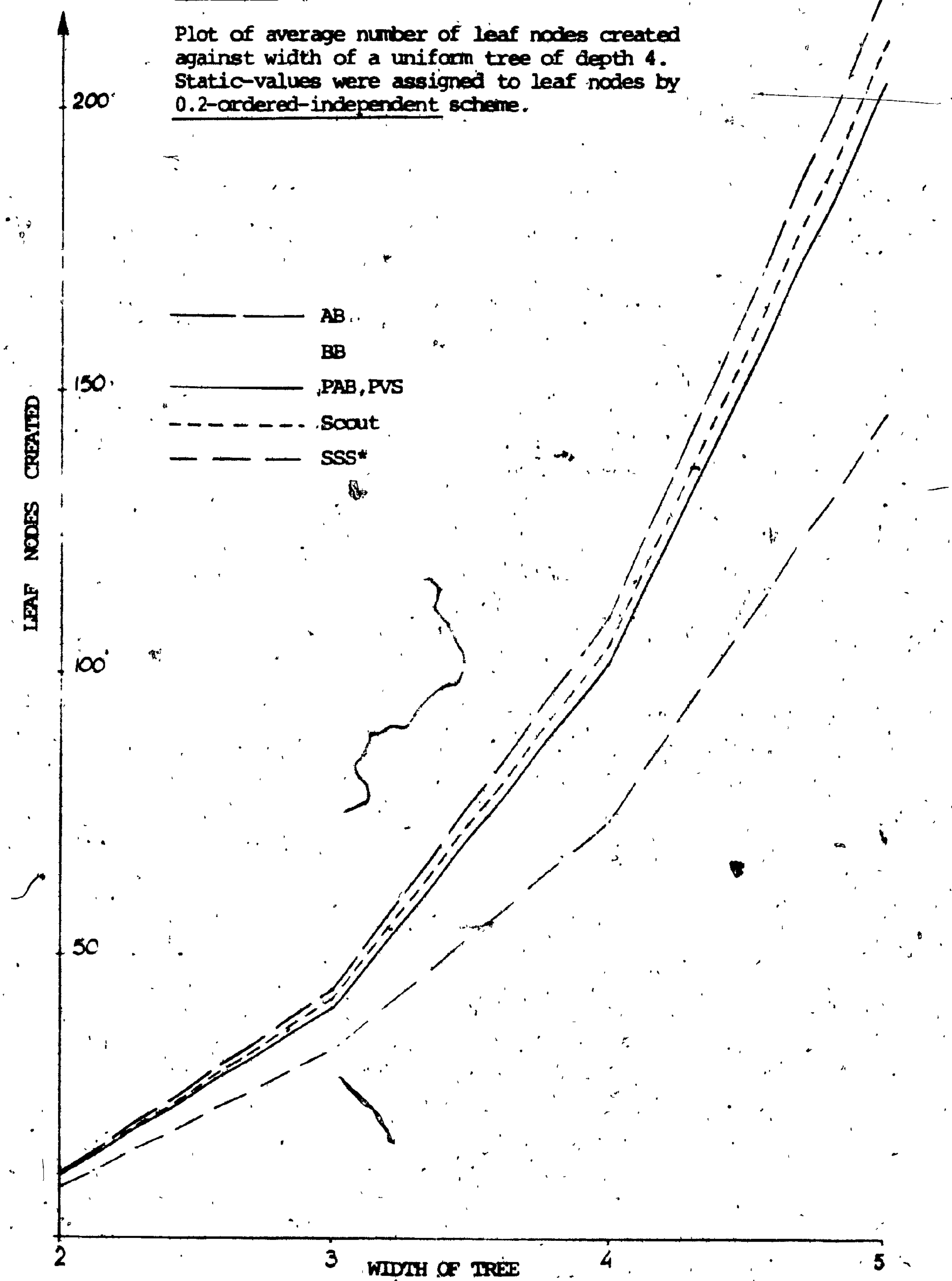


FIGURE 13.

Plot of average number of leaf nodes created against width of a uniform tree with depth 4. Static values were assigned to leaf nodes by 0.4-ordered-independent scheme.

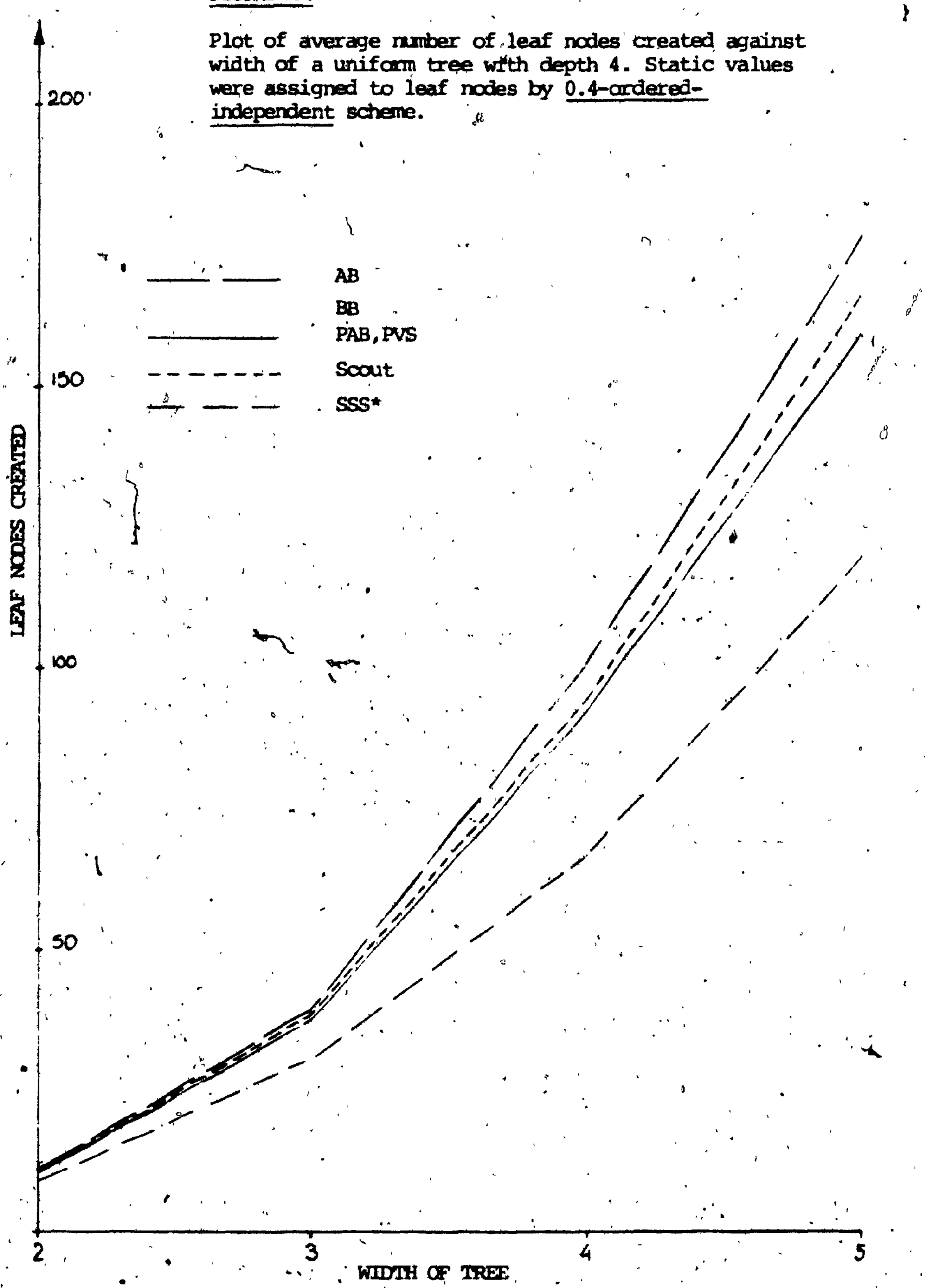


FIGURE 14.

Plot of average number of leaf nodes created against width of a uniform tree of depth 4. Static-values were assigned to leaf nodes by 0.6-ordered-independent scheme.

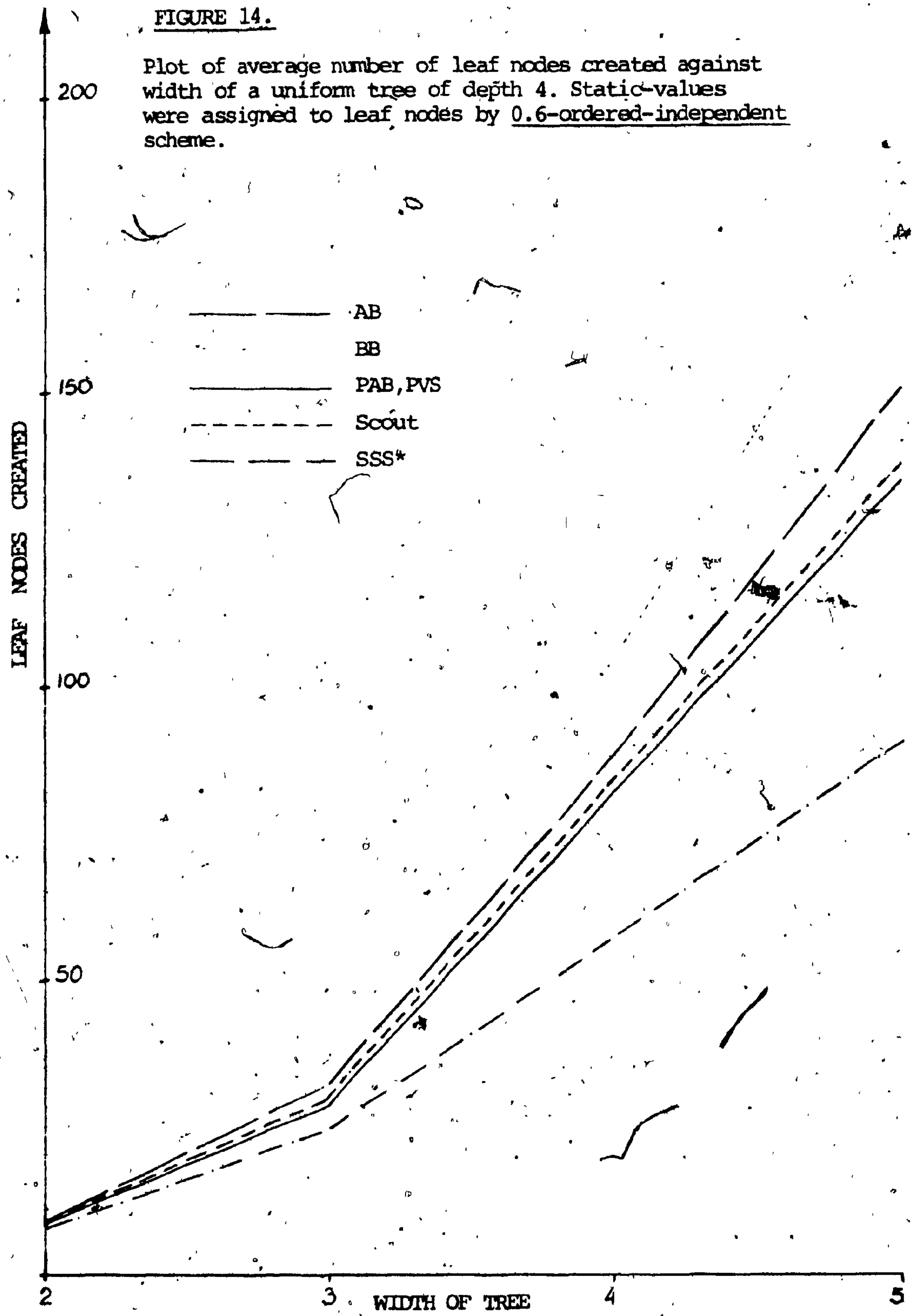
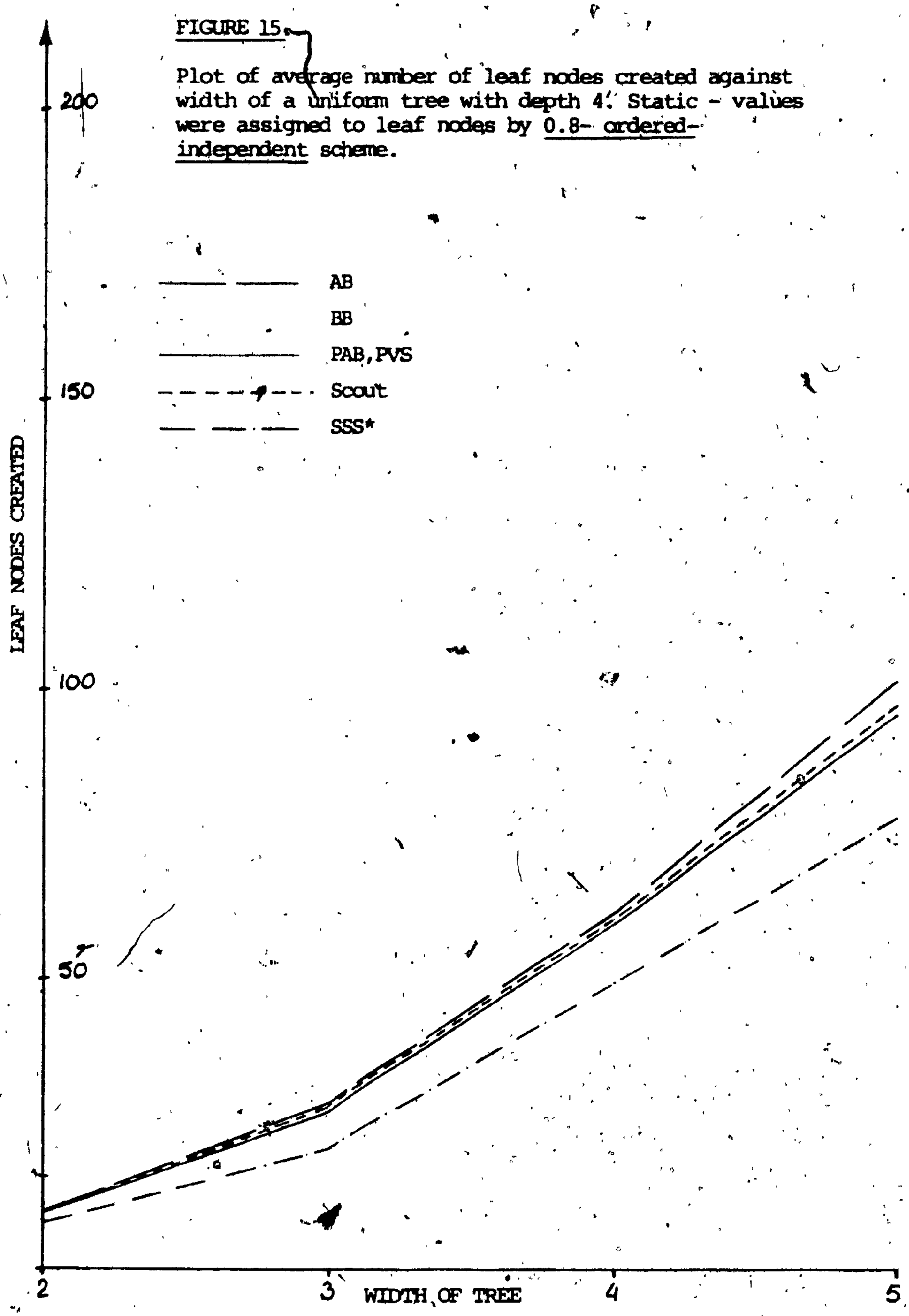


FIGURE 15.

Plot of average number of leaf nodes created against width of a uniform tree with depth 4. Static - values were assigned to leaf nodes by 0.8- ordered-independent scheme.



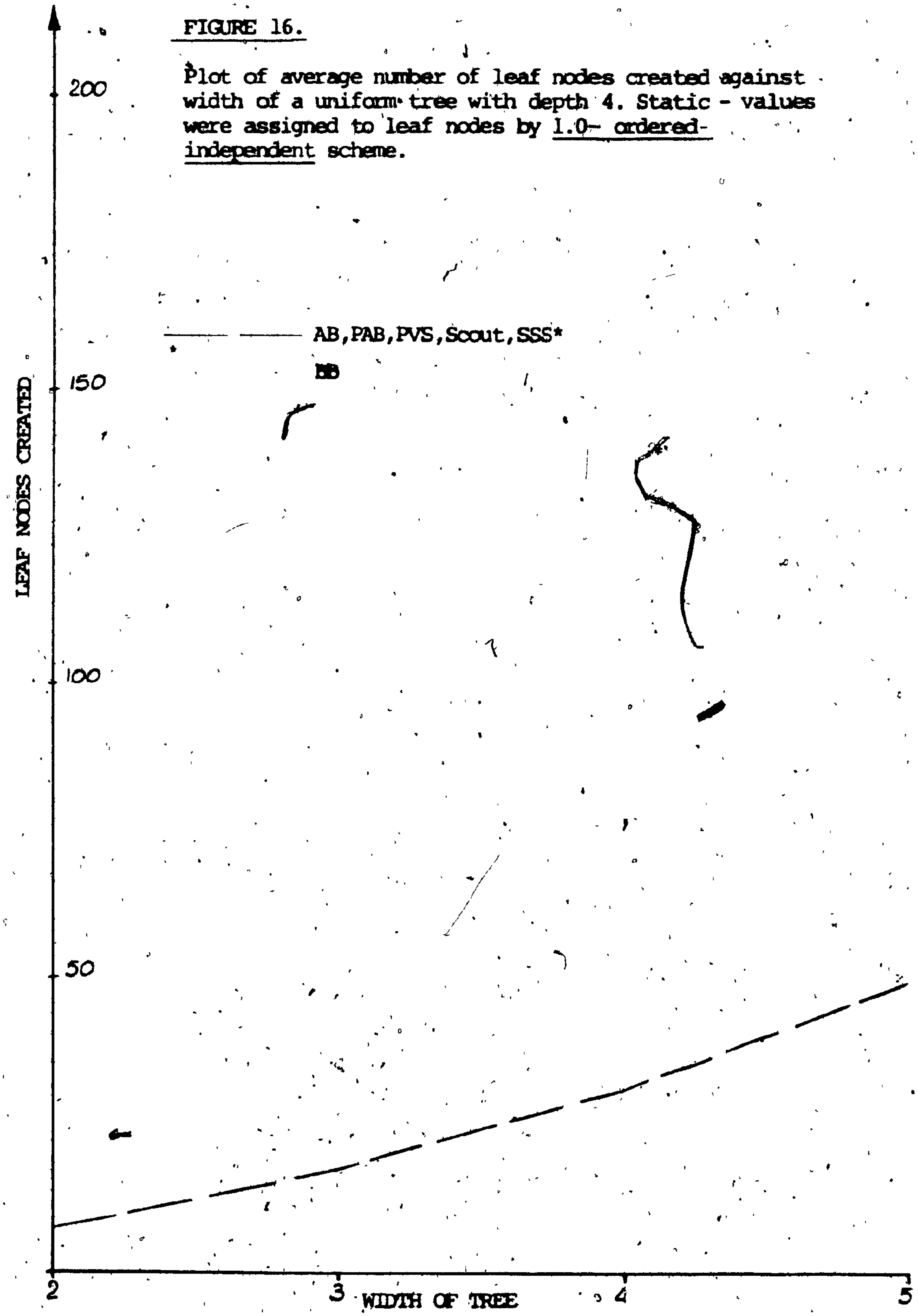


FIGURE 16.

Plot of average number of leaf nodes created against width of a uniform tree with depth 4. Static - values were assigned to leaf nodes by 1.0- ordered-independent scheme.

Tree size	Integer-dependent static-values assignment	Unordered-independent static-values assignment
U(2,2)	3.50	3.67
U(3,2)	6.89	7.44
U(4,2)	10.92	12.14
U(6,2)	20.37	23.96
U(8,2)	31.21	38.65
U(24,2)	143.81	240.29
U(2,3)	6.25	6.84
U(3,3)	16.80	19.45
U(4,3)	32.93	40.11
U(6,3)	82.14	109.61
U(8,3)	153.66	220.37

TABLE II.

Newborn's [20] theoretical results for expected number of leaf nodes created by Alphabeta algorithm. U(3,2) stands for uniform trees of width 3 and depth 2.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
U(2,2)	3.60	3.60	3.60	3.60	3.60	3.22
U(3,2)	7.24	7.24	7.24	7.24	7.24	6.06
U(4,2)	12.38	12.38	12.38	12.38	12.38	10.08
U(5,2)	16.64	16.64	16.64	16.64	16.64	13.08
U(6,2)	25.58	25.58	25.58	25.58	25.58	18.66
U(8,2)	38.24	38.24	38.24	38.24	38.24	28.22
U(10,2)	58.28	58.28	58.28	58.28	58.28	42.82
U(24,2)	241.76	241.76	241.76	241.76	241.76	179.04
U(2,3)	7.04	7.04	6.92	6.92	6.92	6.58
U(3,3)	19.10	19.10	18.34	18.34	18.34	16.73
U(4,3)	39.70	39.70	37.00	37.00	37.00	33.68
U(5,3)	70.32	70.32	63.46	63.46	63.46	56.68
U(6,3)	105.28	105.28	95.26	95.26	95.26	88.10
U(8,3)	220.62	220.62	187.06	187.06	187.06	187.04
U(10,3)	384.34	384.34	322.04	322.04	322.04	321.16
U(2,4)	12.26	12.28	11.96	12.06	11.96	10.20
U(3,4)	43.80	48.78	41.58	42.54	41.54	32.34
U(4,4)	112.86	129.92	103.98	106.18	103.86	80.72
U(5,4)	213.08	253.16	187.40	190.39	187.22	161.30
U(2,5)	21.48	23.32	20.94	21.14	20.88	18.50
U(3,5)	116.90	135.20	103.24	105.82	102.30	93.06
U(4,5)	334.62	408.42	287.04	295.24	283.93	260.36
U(2,6)	35.62	42.72	33.78	34.94	33.70	26.34
U(3,6)	253.56	341.52	226.30	236.28	222.83	173.34

TABLE III.

Average number of leaf nodes created for uniform trees with unordered-independent static-values assignment.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
U(2,2)	3.52	3.52	3.52	3.52	3.52	3.06
U(3,2)	7.04	7.04	7.04	7.04	7.04	5.58
U(4,2)	10.70	10.70	10.70	10.70	10.70	8.18
U(5,2)	15.68	15.68	15.68	15.68	15.68	11.32
U(6,2)	20.06	20.06	20.06	20.06	20.06	14.42
U(8,2)	31.22	31.22	31.22	31.22	31.22	22.16
U(10,2)	42.42	42.42	42.42	42.42	42.42	27.86
U(24,2)	147.62	147.62	147.62	147.62	147.62	84.28
U(2,3)	6.10	6.10	6.10	6.10	6.10	5.50
U(3,3)	16.58	16.58	16.32	16.32	16.32	14.32
U(4,3)	30.52	30.52	29.66	29.66	29.66	26.56
U(5,3)	53.54	53.54	49.70	49.70	49.70	44.98
U(6,3)	81.50	81.50	75.70	75.70	75.70	68.12
U(8,3)	162.44	162.44	146.52	146.52	146.52	128.68
U(10,3)	250.22	250.22	220.98	220.98	220.98	197.50
U(2,4)	10.28	11.32	10.28	10.52	10.28	8.20
U(3,4)	33.62	38.66	33.22	33.94	33.22	25.28
U(4,4)	77.12	92.08	74.46	75.88	74.40	51.44
U(5,4)	150.66	183.24	142.26	146.76	142.20	100.28
U(2,5)	17.74	20.14	17.70	18.10	17.68	14.84
U(3,5)	84.16	102.10	81.66	84.88	81.28	64.18
U(4,5)	229.24	294.00	214.50	222.72	212.60	176.10
U(2,6)	29.38	35.62	29.22	30.20	29.16	23.94
U(3,6)	161.52	237.22	155.74	162.54	155.06	110.44

TABLE IV.

Average number of leaf nodes created for uniform trees with integer-dependent static-values assignment.

Size of tree	PVS, PAB, AB, Scout	BB	SSS*
U(2,2)	3.52	3.52	3.00
U(3,2)	6.88	6.88	5.00
U(4,2)	10.30	10.30	7.00
U(5,2)	14.20	14.20	9.00
U(6,2)	18.00	18.00	11.00
U(8,2)	26.34	26.34	15.00
U(10,2)	35.38	35.38	19.00
U(24,2)	118.30	118.30	47.00
U(2,3)	6.10	6.10	5.42
U(3,3)	15.60	15.60	12.72
U(4,3)	27.46	27.46	22.36
U(5,3)	44.28	44.28	34.28
U(6,3)	66.10	66.10	48.30
U(8,3)	122.38	122.38	85.14
U(10,3)	182.26	182.26	125.38
U(2,4)	10.48	11.32	7.48
U(3,4)	30.96	35.56	18.68
U(4,4)	63.52	76.18	33.52
U(5,4)	115.08	141.06	55.32
U(2,5)	17.90	20.14	13.04
U(3,5)	73.84	88.72	46.12
U(4,5)	177.52	227.92	105.58
U(2,6)	28.88	35.62	17.64
U(3,6)	137.48	199.72	64.36

TABLE V.

Average number of leaf nodes created for uniform trees with real-dependent static-values assignment.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
U(2,2)	3.52	3.52	3.52	3.52	3.52	3.02
U(3,2)	7.44	7.44	7.44	7.44	7.44	5.74
U(4,2)	11.78	11.78	11.78	11.78	11.78	8.84
U(5,2)	16.50	16.50	16.50	16.50	16.50	13.14
U(6,2)	23.36	23.36	23.36	23.36	23.36	15.96
U(8,2)	34.98	34.98	34.98	34.98	34.98	23.02
U(10,2)	49.02	49.02	49.02	49.02	49.02	28.00
U(24,2)	148.26	148.26	148.26	148.26	148.26	64.78
U(2,3)	6.66	6.66	6.56	6.56	6.56	5.98
U(3,3)	19.12	19.12	18.24	18.24	18.24	17.02
U(4,3)	40.30	40.30	37.54	37.54	37.54	34.72
U(5,3)	68.62	68.62	61.74	61.74	61.74	58.80
U(6,3)	106.46	106.46	94.22	94.22	94.22	92.72
U(8,3)	204.62	204.62	169.16	169.16	169.16	165.70
U(10,3)	341.72	341.72	278.02	278.02	278.02	278.00
U(2,4)	11.22	12.14	10.98	11.08	10.98	8.94
U(3,4)	43.80	50.32	40.92	42.10	40.84	33.16
U(4,4)	110.10	129.70	102.06	106.28	102.00	73.86
U(5,4)	222.38	263.74	205.56	213.06	204.96	146.34
U(2,5)	19.66	22.38	18.32	18.66	18.26	17.88
U(3,5)	105.12	126.06	94.96	97.56	94.22	85.68
U(4,5)	345.98	432.38	301.14	313.06	297.46	268.60
U(2,6)	33.56	42.52	31.56	33.10	31.42	24.66
U(3,6)	228.00	319.32	201.48	212.04	199.34	163.46

TABLE VI.

Average number of leaf nodes created for uniform trees with 0.2-ordered-independent scheme.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
U(2,2)	3.50	3.50	3.50	3.50	3.50	3.04
U(3,2)	7.28	7.28	7.28	7.28	7.28	5.76
U(4,2)	10.84	10.84	10.84	10.84	10.84	8.40
U(5,2)	15.20	15.20	15.20	15.20	15.20	10.64
U(6,2)	19.44	19.44	19.44	19.44	19.44	13.68
U(8,2)	29.28	29.28	29.28	29.28	29.28	19.16
U(10,2)	44.94	44.94	44.94	44.94	44.94	26.12
U(24,2)	115.32	115.32	115.32	115.32	115.32	60.68
U(2,3)	6.44	6.44	6.26	6.26	6.26	5.92
U(3,3)	19.16	19.16	18.00	18.00	18.00	16.40
U(4,3)	36.70	36.70	33.44	33.44	33.44	33.20
U(5,3)	65.70	65.70	58.36	58.36	58.36	55.44
U(6,3)	94.86	94.86	82.66	82.66	82.66	82.64
U(8,3)	179.06	179.06	149.52	149.52	149.52	148.52
U(10,3)	306.54	306.54	236.60	236.60	236.60	236.40
U(2,4)	11.16	12.18	10.88	11.06	10.88	8.82
U(3,4)	39.12	46.84	37.44	38.82	37.12	30.88
U(4,4)	100.86	121.56	92.44	94.78	92.32	67.02
U(5,4)	176.90	217.98	160.48	166.46	159.92	120.20
U(2,5)	20.54	22.78	19.56	19.78	19.48	17.72
U(3,5)	95.96	113.36	87.84	89.64	87.58	77.26
U(4,5)	271.76	335.10	236.60	240.26	233.22	208.92
U(2,6)	34.28	43.26	31.90	33.18	31.80	24.00
U(3,6)	208.72	281.38	188.18	195.74	184.86	143.68

TABLE VII.

Average number of leaf nodes created for uniform trees with 0.4-ordered-independent scheme.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
U(2,2)	3.50	3.50	3.50	3.50	3.50	3.02
U(3,2)	6.70	6.70	6.70	6.70	6.70	5.34
U(4,2)	10.16	10.16	10.16	10.16	10.16	7.66
U(5,2)	13.12	13.12	13.12	13.12	13.12	10.26
U(6,2)	16.96	16.96	16.96	16.96	16.96	12.56
U(8,2)	25.72	25.72	25.72	25.72	25.72	18.32
U(10,2)	33.32	33.32	33.32	33.32	33.32	23.80
U(24,2)	89.44	89.44	89.44	89.44	89.44	58.12
U(2,3)	6.24	6.24	6.08	6.08	6.08	5.90
U(3,3)	17.18	17.18	16.74	16.74	16.74	14.96
U(4,3)	33.08	33.08	30.54	30.54	30.54	28.62
U(5,3)	52.82	52.82	47.24	47.24	47.24	44.58
U(6,3)	78.12	78.12	68.74	68.74	68.74	66.02
U(8,3)	156.70	156.70	133.02	133.02	133.02	129.60
U(10,3)	240.98	240.98	199.86	199.86	199.86	199.98
U(2,4)	9.20	10.14	9.14	9.20	9.14	8.44
U(3,4)	32.12	36.84	28.80	29.94	28.64	24.82
U(4,4)	88.36	104.16	81.98	84.20	81.80	57.68
U(5,4)	150.80	183.56	135.92	138.30	135.20	90.88
U(2,5)	19.24	21.96	18.34	18.64	18.28	16.26
U(3,5)	77.02	90.12	74.80	75.63	74.02	62.12
U(4,5)	206.06	255.70	180.58	182.92	178.66	162.34
U(2,6)	32.82	41.16	31.02	31.90	30.82	23.86
U(3,6)	169.64	235.66	156.66	161.82	156.04	8121.62

TABLE VIII.

Average number of leaf nodes created for uniform trees with 0.6-ordered-independent scheme.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
U(2,2)	3.36	3.36	3.36	3.36	3.36	3.02
U(3,2)	6.26	6.26	6.26	6.26	6.26	5.22
U(4,2)	9.26	9.26	9.26	9.26	9.26	7.22
U(5,2)	12.06	12.06	12.06	12.06	12.06	9.70
U(6,2)	15.36	15.36	15.36	15.36	15.36	11.64
U(8,2)	20.06	20.06	20.06	20.06	20.06	16.60
U(10,2)	28.32	28.32	28.32	28.32	28.32	20.72
U(24,2)	78.64	78.64	78.64	78.64	78.64	56.04
U(2,3)	5.76	5.76	5.68	5.68	5.68	5.34
U(3,3)	14.66	14.66	14.38	14.38	14.38	13.12
U(4,3)	27.32	27.32	25.64	25.64	25.64	23.68
U(5,3)	42.36	42.36	39.12	39.12	39.12	37.84
U(6,3)	59.68	59.68	54.98	54.98	54.98	54.88
U(8,3)	112.78	112.78	97.80	97.80	97.80	97.10
U(10,3)	175.52	175.52	153.02	153.02	153.02	153.00
U(2,4)	10.00	10.96	9.74	9.82	9.74	8.10
U(3,4)	28.02	32.24	27.04	27.44	27.04	20.92
U(4,4)	66.68	79.28	61.70	63.18	61.58	49.96
U(5,4)	101.42	126.86	95.42	96.80	95.12	78.12
U(2,5)	17.64	19.68	16.84	17.00	16.76	15.24
U(3,5)	69.96	86.18	63.40	64.16	62.84	56.36
U(4,5)	139.34	173.02	130.06	131.72	129.80	120.28
U(2,6)	28.66	35.56	27.52	28.10	27.42	21.40
U(3,6)	116.98	169.30	110.60	114.14	110.40	90.66

TABLE IX.

Average number of leaf nodes created for uniform trees with 0.8-ordered-independent scheme.

Tree size	AB, PAB, Scout, PVS, SSS*	BB
U(2,2)	3.00	3.00
U(3,2)	5.00	5.00
U(4,2)	7.00	7.00
U(5,2)	9.00	9.00
U(6,2)	11.00	11.00
U(8,2)	15.00	15.00
U(10,2)	19.00	19.00
U(24,2)	47.00	47.00
U(2,3)	5.00	5.00
U(3,3)	11.00	11.00
U(4,3)	19.00	19.00
U(5,3)	29.00	29.00
U(6,3)	41.00	41.00
U(8,3)	71.00	71.00
U(10,3)	109.00	109.00
U(2,4)	7.00	8.00
U(3,4)	17.00	21.00
U(4,4)	31.00	40.00
U(5,4)	49.00	65.00
U(2,5)	11.00	13.00
U(3,5)	35.00	43.00
U(4,5)	79.00	97.00
U(2,6)	15.00	21.00
U(3,6)	53.00	85.00

TABLE X.

Average number of leaf nodes created for uniform trees with 1.0-ordered-independent scheme.

Newborn's results were limited to uniform trees of depths 2 and 3 with widths 2, 3, 4, 6, 8, 12, 16, 20, 24, 28, 32, 36, 48, 64, 80, 96, 128 and 196. Table II shows Newborn's results for those tree-sizes that are common with the tree sizes of the experiments. One can notice that the values for Alphabeta are quite close to the theoretical values estimated by Newborn. For example, for uniform trees of width 8 and depth 3, U(8,3), with unordered-independent static-values assignment, Newborn's value was 220.37 (Table II), whereas our value is 220.62 (Table III). Occasionally, however, Newborn's values and our values are not close: for U(24,2), Newborn gave 143.81 (Table II, integer-dependent static-values assignment); our corresponding value is 147.62 (Table IV). This discrepancy can perhaps be explained by the fact that only 50 trees of any given width and depth were simulated. Having a larger sample-size may have given a value closer to Newborn's.

We now compare Tables III, IV and V. They show that for a given pruning strategy and tree-size, the average number of leaf nodes created were sensitive to the different schemes used to assign static-values. We notice that the highest number of leaf nodes were created for the unordered-independent scheme, fewer for the integer-dependent scheme, and the fewest for the real-dependent scheme. For example, the average number of leaf nodes created by SSS* for U(10,3) is 321.16 in Table

III, 197.50 in Table IV and 125.38 in Table V. This agrees with the remark made by Knuth and Moore [12] that fewer leaf nodes may be created for a dependent static-values scheme as compared to an independent scheme.

In Tables VI to X the average number of leaf nodes created for uniform trees with P-ordered-independent static-values assignment is presented, where value of P varied from 0.2 to 1.0 in steps of 0.2. We notice that as the value of P increased in P-ordered-independent scheme, there was a decrease in the average number of leaf nodes created for every pruning strategy. For example for U(3,6) with 0.2-ordered static-values assignment (Table VI) Alphabeta created on average 232.58 leaf nodes, whereas for trees with 0.8-ordered static-values assignment (Table IX) Alphabeta created on average 10[illegible].98 leaf nodes. This was as expected based on the results obtained by Slagle and Dixon [29]. Thus the least leaf nodes were created for 1.0-ordering. In fact, then the number of leaf nodes created agreed with the theoretical formula $w^{\lceil d/2 \rceil} + w^{\lfloor d/2 \rfloor} - 1$ as given in [12,20].

It was shown analytically by Roizen and Pearl [28] that for uniform trees with unordered-independent static-values assignment the ratio of leaf nodes created by Alphabeta algorithm to the leaf nodes created by SSS* lies in the interval [1.1, 3.0]. Pearl [27] has also proved that the ratio of leaf nodes created by Scout algorithm to the leaf nodes created by Alphabeta remains below 1.275 and tends to unity with increasing search depths. Our experimental results from Table III confirm these two statements. In fact, these results hold even for Tables IV to X, that is for all cases of static-values assignment. We were restricted to trees of depths up to 6 because of limited computer space and time. Up to that depth, the empirical results agree with the theoretical results of Roizen and Pearl [28] and we may conjecture that they would agree even for deeper trees, had we been able to simulate them.

In Figures 17 to 24, for nonuniform trees of depth 4, the average number of leaf nodes created by the different strategies versus the width of tree are plotted. Furthermore, in Tables XI to XVIII the results observed about the average number of leaf nodes created for nonuniform trees with the different kinds of static-values assignments are presented. The trend of results for nonuniform trees is mostly similar to that for the uniform trees. For a given pruning strategy and tree-size, usually the highest number of leaf nodes were created for the unordered-independent scheme, fewer for the integer-dependent scheme and the fewest for the

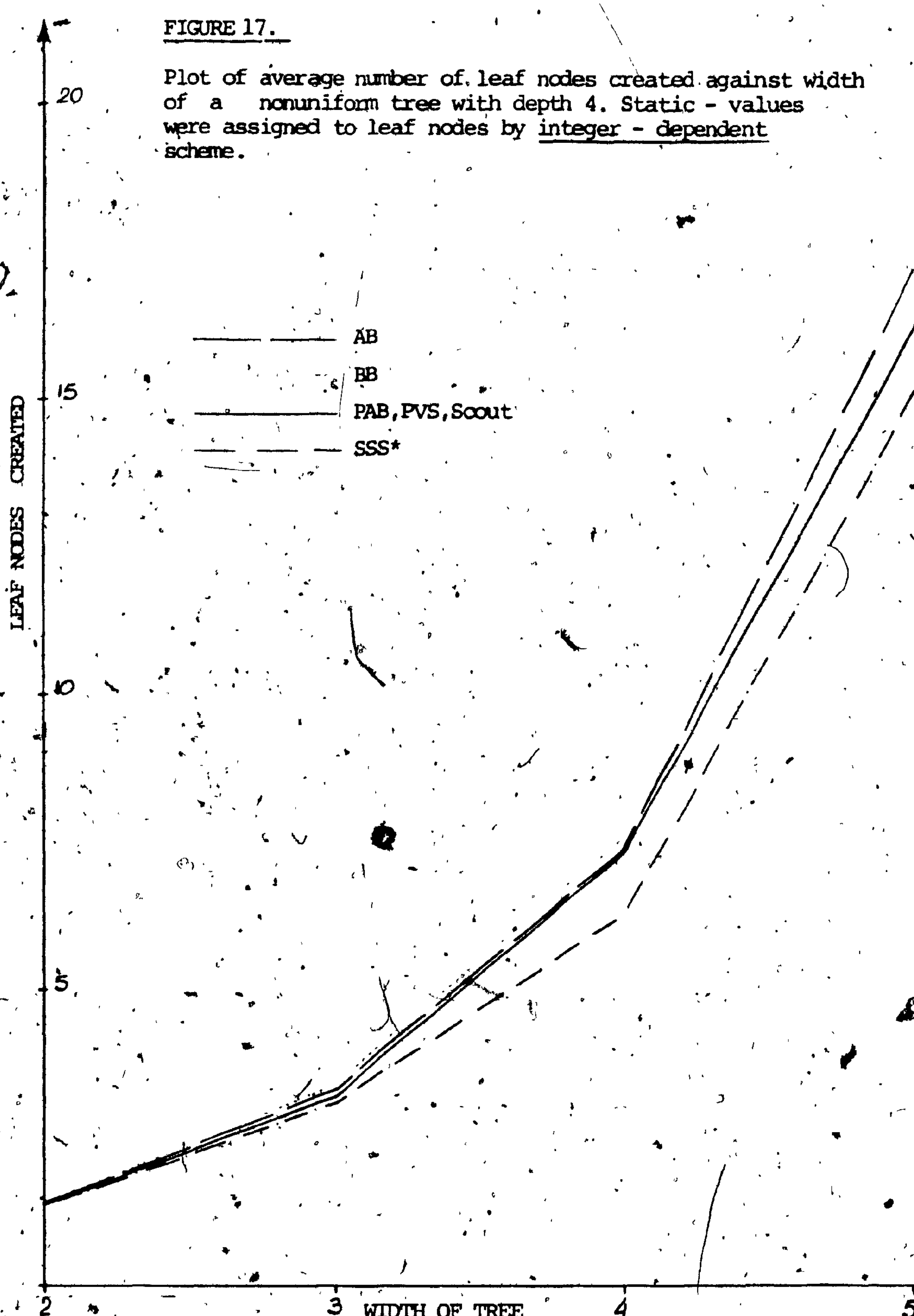


FIGURE 17.
Plot of average number of leaf nodes created against width of a nonuniform tree with depth 4. Static - values were assigned to leaf nodes by integer - dependent scheme.
20
15
10
5
AB
BB
PAB, PVS, Scout
SSS*
LEAF NODES CREATED
2
3
4
5
WIDTH OF TREE

FIGURE 18.

Plot of average number of leaf nodes created against width of a nonuniform tree with depth 4. Static - values were assigned to leaf nodes by real - dependent scheme.

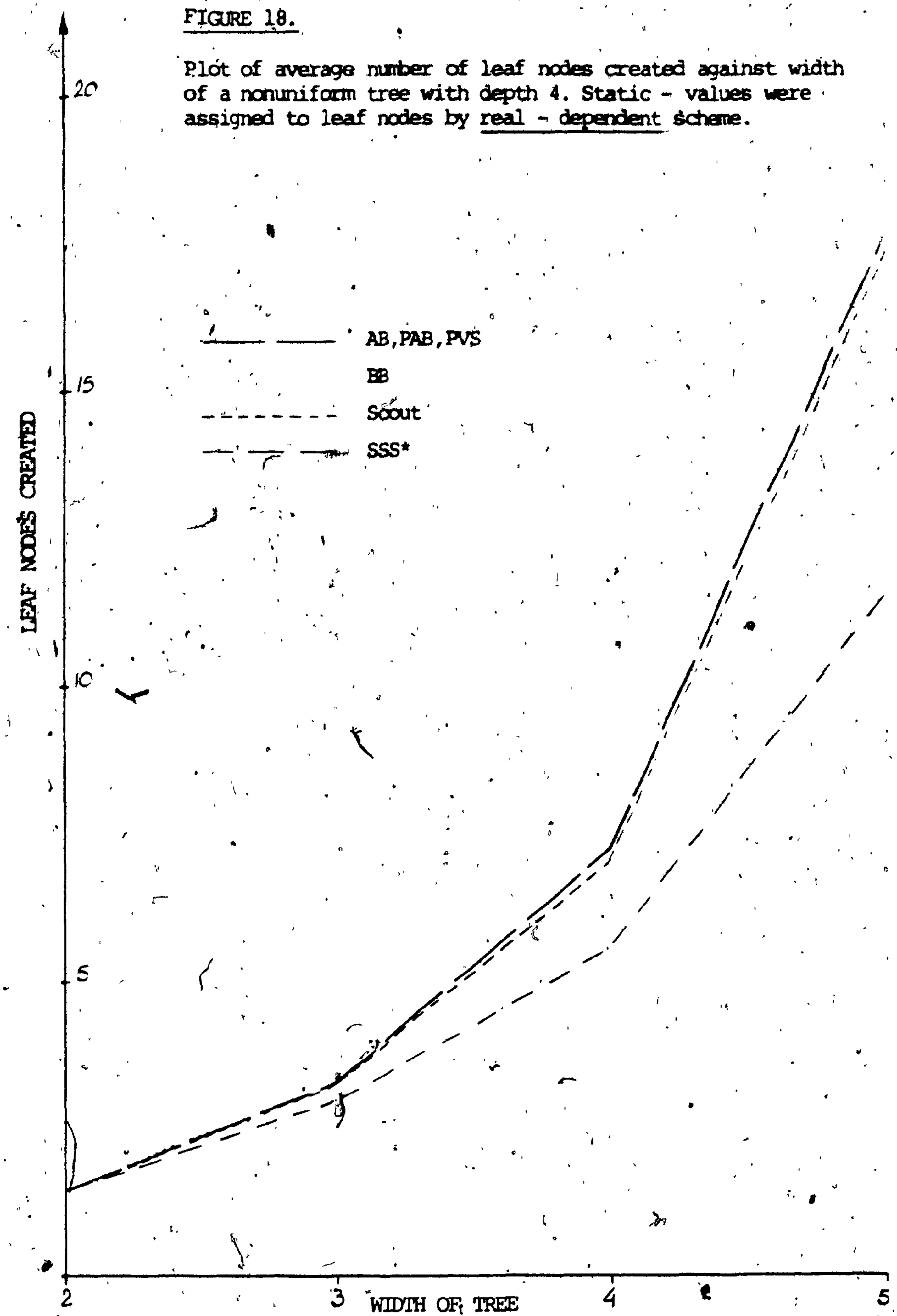


FIGURE 19.

Plot of average number of leaf nodes created against width of a nonuniform tree with depth 4. Static values were assigned to leaf nodes by unordered-independent scheme.

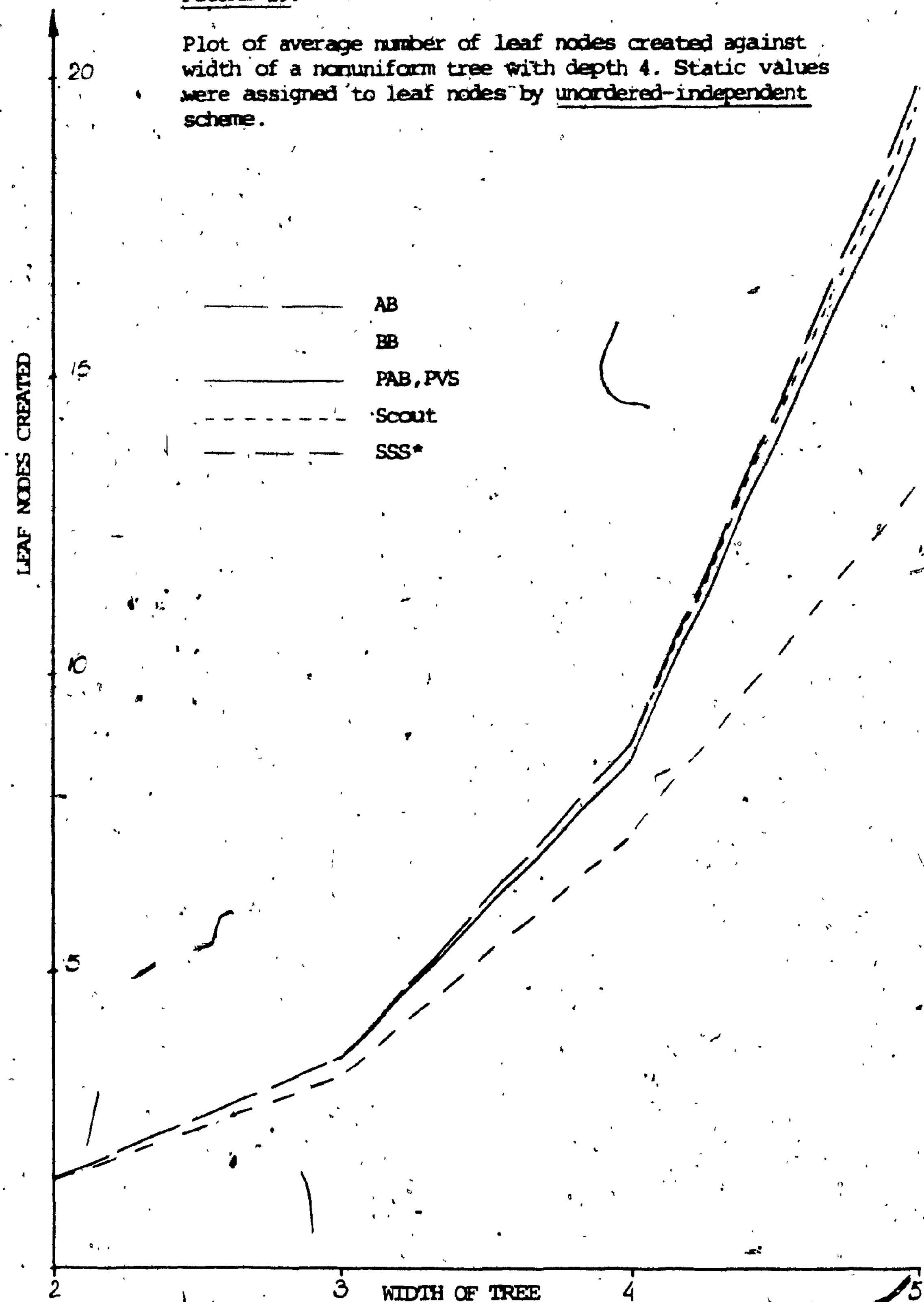


FIGURE 20.

Plot of average number of leaf nodes created against width of a nonuniform tree with depth 4. Static-values were assigned to leaf nodes by 0.2-ordered-independent scheme.

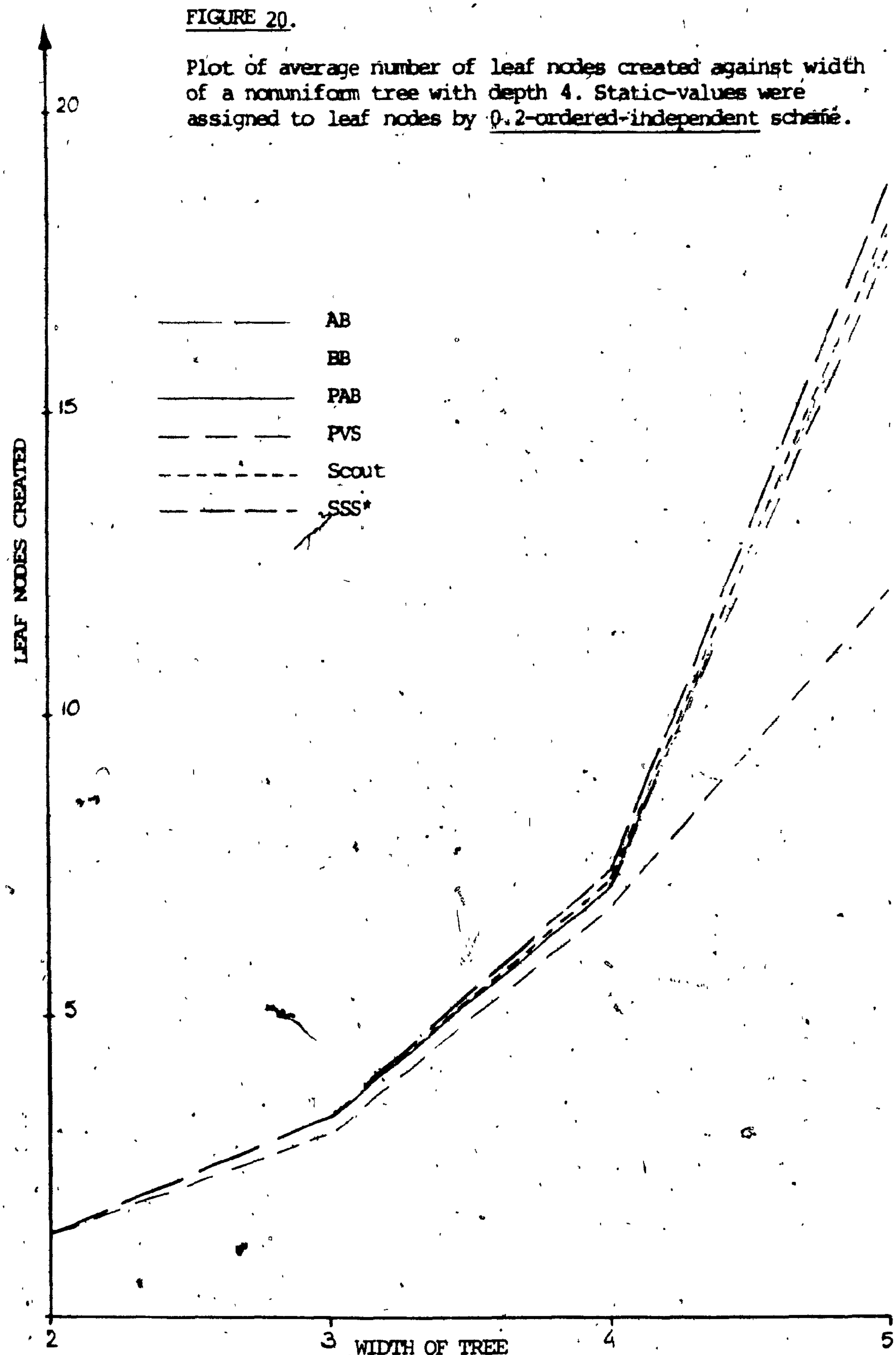


FIGURE 21

Plot of average number of leaf nodes created against width of a Nonuniform tree with depth 4. Static-values were assigned to leaf nodes by 0.4-ordered-independent scheme.

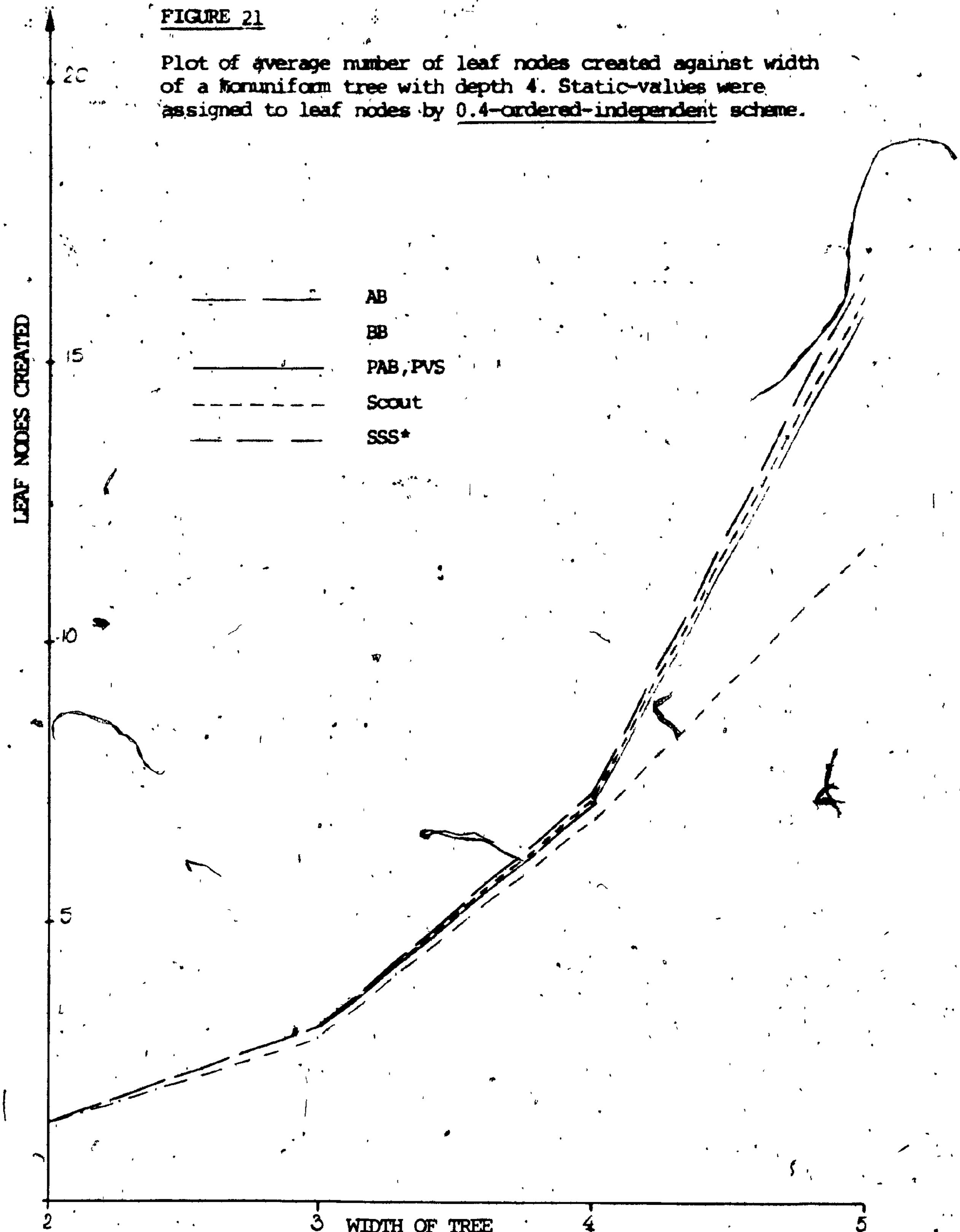


FIGURE 22.

Plot of average number of leaf nodes created against width of a-nonuniform tree of depth 4. Static-values were assigned to leaf nodes by 0.6-ordered-independent scheme.

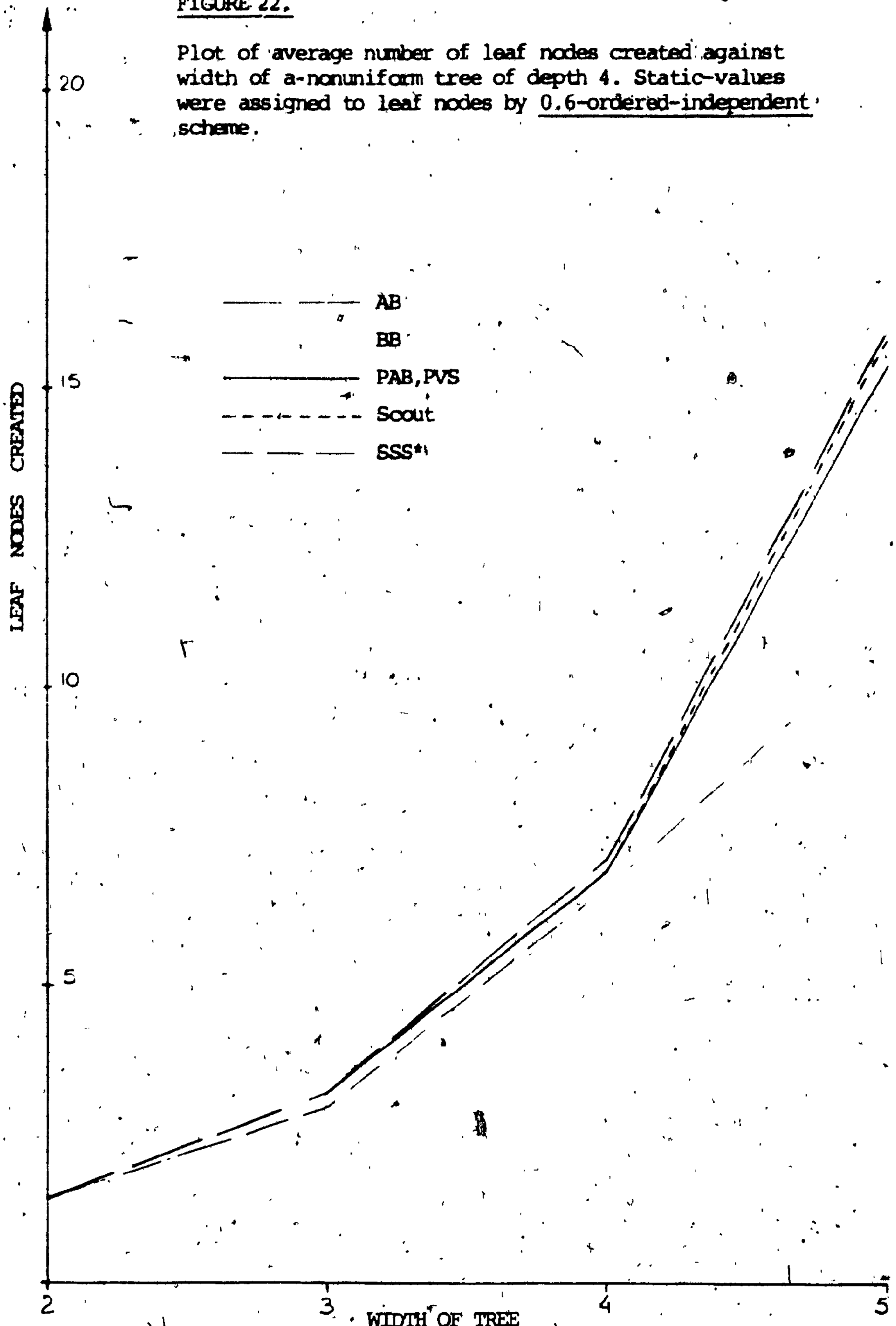
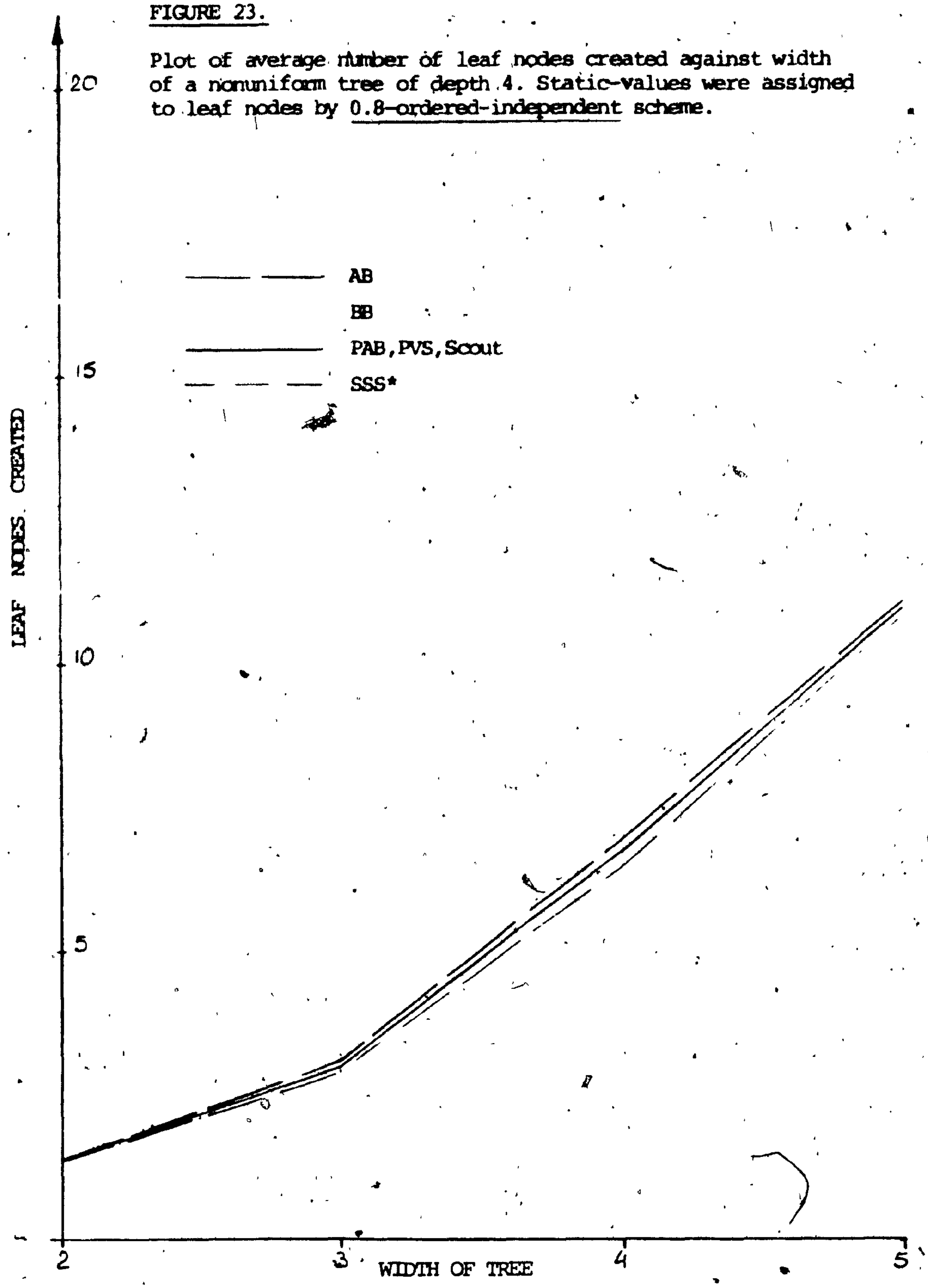


FIGURE 23.

Plot of average number of leaf nodes created against width of a nonuniform tree of depth 4. Static-values were assigned to leaf nodes by 0.8-ordered-independent scheme.



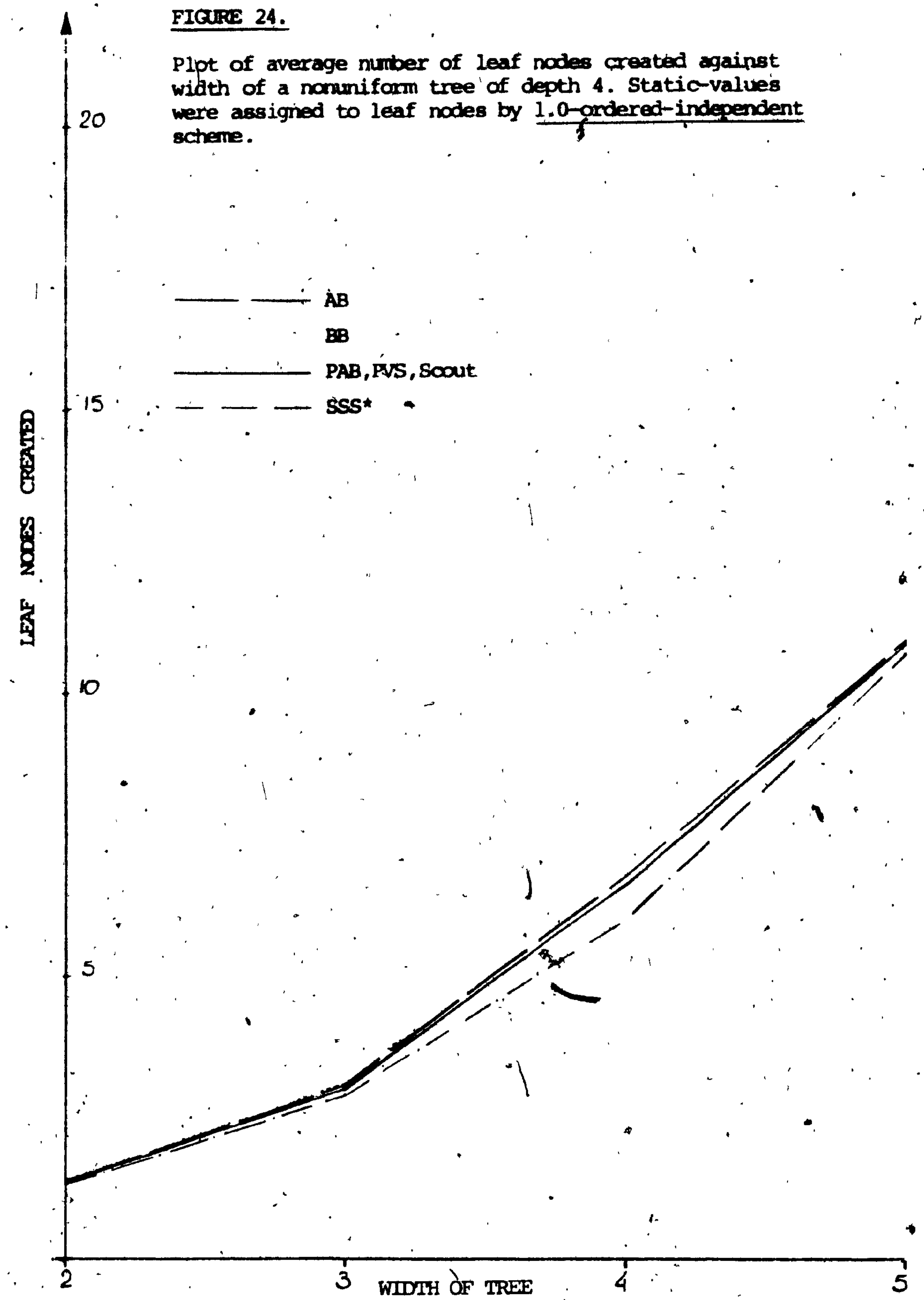


FIGURE 24.
Plot of average number of leaf nodes created against width of a nonuniform tree of depth 4. Static-values were assigned to leaf nodes by 1.0-ordered-independent scheme.
AB
BB
PAB, PVS, Scout
SSS*
LEAF NODES CREATED
20
15
10
5
2
3
4
5
WIDTH OF TREE

Tree size	AB	BB	PAB	Scout	PVS	SSS*
N(2,2)	1.52	1.52	1.52	1.52	1.52	1.52
N(3,2)	2.42	2.42	2.42	2.42	2.42	2.36
N(4,2)	3.74	3.74	3.74	3.74	3.74	3.52
N(5,2)	5.18	5.18	5.18	5.18	5.18	4.46
N(6,2)	7.10	7.10	7.10	7.10	7.10	5.92
N(8,2)	9.28	9.28	9.28	9.28	9.28	7.46
N(10,2)	13.50	13.50	13.50	13.50	13.50	11.28
N(24,2)	45.76	45.76	45.76	45.76	45.76	33.28
N(2,3)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,3)	2.74	2.74	2.72	2.72	2.72	2.68
N(4,3)	5.88	5.88	5.78	5.78	5.78	5.68
N(5,3)	8.40	8.40	8.00	8.00	8.00	7.72
N(6,3)	13.34	13.34	12.64	12.64	12.64	12.36
N(8,3)	28.06	28.06	25.10	25.10	25.10	25.04
N(10,3)	44.82	44.82	36.90	36.90	36.90	36.88
N(2,4)	1.42	1.42	1.42	1.42	1.42	1.42
N(3,4)	3.30	3.32	3.28	3.28	3.27	3.14
N(4,4)	7.40	7.60	7.40	7.40	7.40	6.28
N(5,4)	17.20	18.31	16.29	16.29	16.29	15.20
N(2,5)	1.44	1.44	1.44	1.44	1.44	1.44
N(3,5)	4.00	4.10	3.96	3.98	3.96	3.78
N(4,5)	10.88	12.22	10.38	10.40	10.38	9.36
N(2,6)	1.40	1.40	1.40	1.40	1.40	1.40
N(3,6)	3.70	3.94	3.66	3.68	3.66	3.58

TABLE XI.

Average number of leaf nodes created for nonuniform trees with integer-dependent static-values assignment.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
N(2,2)	1.52	1.52	1.52	1.52	1.52	1.52
N(3,2)	2.38	2.38	2.38	2.38	2.38	2.24
N(4,2)	3.58	3.58	3.58	3.58	3.58	3.32
N(5,2)	4.92	4.92	4.92	4.92	4.92	3.90
N(6,2)	6.70	6.70	6.70	6.70	6.70	4.96
N(8,2)	8.26	8.26	8.26	8.26	8.26	6.16
N(10,2)	10.82	10.82	10.82	10.82	10.82	7.60
N(24,2)	34.17	34.17	34.17	34.17	34.17	29.03
N(2,3)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,3)	2.76	2.76	2.76	2.74	2.76	2.66
N(4,3)	5.94	5.94	5.94	5.94	5.94	5.42
N(5,3)	9.12	9.12	9.12	9.02	9.12	7.62
N(6,3)	14.30	14.30	14.30	14.14	14.30	11.88
N(8,3)	26.26	26.26	26.26	25.98	26.26	24.40
N(10,3)	43.20	43.20	43.20	42.40	43.20	31.74
N(2,4)	1.42	1.42	1.42	1.42	1.42	1.42
N(3,4)	3.30	3.32	3.30	3.30	3.30	2.98
N(4,4)	7.22	7.42	7.22	7.10	7.22	5.52
N(5,4)	17.64	19.60	17.64	17.46	17.64	11.60
N(2,5)	1.44	1.44	1.44	1.44	1.44	1.44
N(3,5)	4.08	4.24	4.08	4.06	4.08	3.76
N(4,5)	11.34	12.68	11.24	10.96	11.24	9.54
N(2,6)	1.40	1.40	1.40	1.40	1.40	1.40
N(3,6)	3.72	4.02	3.70	3.68	3.70	3.42

TABLE XII.

Average number of leaf nodes created for nonuniform trees with real-dependent static-values assignment.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
N(2,2)	1.58	1.58	1.58	1.58	1.58	1.58
N(3,2)	2.54	2.54	2.54	2.54	2.54	2.54
N(4,2)	4.08	4.08	4.08	4.08	4.08	3.78
N(5,2)	6.32	6.32	6.32	6.32	6.32	5.64
N(6,2)	8.44	8.44	8.44	8.44	8.44	7.38
N(8,2)	14.62	14.62	14.62	14.62	14.62	12.16
N(10,2)	25.86	25.86	25.86	25.86	25.86	21.42
N(24,2)	143.68	143.68	143.68	143.68	143.68	107.56
N(2,3)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,3)	3.02	3.02	3.02	3.02	3.02	2.78
N(4,3)	6.62	6.62	6.46	6.46	6.46	6.24
N(5,3)	10.64	10.64	10.48	10.48	10.48	9.04
N(6,3)	23.02	23.02	22.54	22.54	22.54	18.80
N(8,3)	50.82	50.82	49.40	49.40	49.40	39.72
N(10,3)	101.20	101.20	96.20	96.20	96.20	79.60
N(2,4)	1.50	1.50	1.50	1.50	1.50	1.50
N(3,4)	3.50	3.50	3.50	3.50	3.50	3.22
N(4,4)	8.80	8.94	8.56	8.56	8.56	7.32
N(5,4)	19.84	21.50	19.05	19.60	19.00	13.20
N(2,5)	1.52	1.52	1.52	1.52	1.52	1.52
N(3,5)	4.20	4.22	4.18	4.18	4.18	3.98
N(4,5)	17.18	17.98	16.66	16.90	16.54	15.00
N(2,6)	1.50	1.50	1.50	1.50	1.50	1.50
N(3,6)	5.14	5.24	5.06	5.08	5.04	4.90

TABLE XIII.

Average number of leaf nodes created for nonuniform trees with unordered-independent static-values assignment.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
N(2,2)	1.58	1.58	1.58	1.58	1.58	1.58
N(3,2)	2.52	2.52	2.52	2.52	2.52	2.52
N(4,2)	4.08	4.08	4.08	4.08	4.08	3.78
N(5,2)	5.88	5.88	5.88	5.88	5.88	5.06
N(6,2)	7.46	7.46	7.46	7.46	7.46	6.64
N(8,2)	14.52	14.52	14.52	14.52	14.52	12.12
N(10,2)	19.40	19.40	19.40	19.40	19.40	15.46
N(24,2)	93.72	93.72	93.72	93.72	93.72	68.88
N(2,3)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,3)	2.80	2.80	2.80	2.80	2.80	2.68
N(4,3)	6.54	6.54	6.36	6.36	6.36	6.08
N(5,3)	10.52	10.52	9.84	9.84	9.84	8.88
N(6,3)	18.70	18.70	18.34	18.34	18.34	14.92
N(8,3)	44.58	44.58	42.62	42.62	42.62	35.12
N(10,3)	79.94	79.94	74.98	74.98	74.98	60.68
N(2,4)	1.46	1.46	1.46	1.46	1.46	1.46
N(3,4)	3.32	3.32	3.30	3.30	3.30	3.04
N(4,4)	7.40	7.64	7.16	7.26	7.16	6.73
N(5,4)	18.80	20.96	18.00	18.12	17.86	12.08
N(2,5)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,5)	4.18	4.20	4.16	4.16	4.14	3.82
N(4,5)	15.64	16.28	15.00	15.12	14.90	13.16
N(2,6)	1.50	1.50	1.50	1.50	1.50	1.50
N(3,6)	4.70	4.84	4.64	4.64	4.64	4.17

TABLE XIV.

Average number of leaf nodes created for nonuniform trees with 0.2-ordered-independent static-values assignment.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
N(2,2)	1.56	1.56	1.56	1.56	1.56	1.56
N(3,2)	2.39	2.39	2.39	2.39	2.39	2.30
N(4,2)	3.84	3.84	3.84	3.84	3.84	3.50
N(5,2)	6.26	6.26	6.26	6.26	6.26	5.32
N(6,2)	7.40	7.40	7.40	7.40	7.40	6.62
N(8,2)	12.56	12.56	12.56	12.56	12.56	10.50
N(10,2)	19.28	19.28	19.28	19.28	19.28	15.32
N(24,2)	80.70	80.70	80.70	80.70	80.70	65.66
N(2,3)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,3)	2.80	2.80	2.80	2.80	2.80	2.66
N(4,3)	6.22	6.22	5.94	5.94	5.94	5.56
N(5,3)	10.12	10.12	9.82	9.82	9.82	8.70
N(6,3)	17.14	17.14	16.44	16.44	16.44	14.48
N(8,3)	38.10	38.10	35.44	35.44	35.44	30.92
N(10,3)	69.42	69.42	64.52	64.52	64.52	57.42
N(2,4)	1.44	1.44	1.44	1.44	1.44	1.44
N(3,4)	3.16	3.16	3.16	3.16	3.16	3.00
N(4,4)	7.26	7.60	7.12	7.20	7.10	6.80
N(5,4)	16.66	18.24	16.02	16.24	16.00	11.72
N(2,5)	1.47	1.47	1.47	1.47	1.47	1.47
N(3,5)	4.02	4.18	3.99	4.02	3.97	3.78
N(4,5)	14.82	16.04	14.00	14.12	13.96	12.04
N(2,6)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,6)	4.66	4.80	4.29	4.36	4.20	4.02

TABLE XV.

Average number of leaf nodes created for nonuniform trees with 0.4-ordered-independent static-values assignment.

Tree size	AB	BB	PAB	Scout	PVS	SSS*
N(2,2)	1.58	1.58	1.58	1.58	1.58	1.58
N(3,2)	2.32	2.32	2.32	2.32	2.32	2.28
N(4,2)	3.82	3.82	3.82	3.82	3.82	3.50
N(5,2)	5.22	5.22	5.22	5.22	5.22	4.52
N(6,2)	6.38	6.38	6.38	6.38	6.38	5.72
N(8,2)	11.70	11.70	11.70	11.70	11.70	9.26
N(10,2)	17.86	17.86	17.86	17.86	17.86	12.72
N(24,2)	68.94	68.94	68.94	68.94	68.94	42.30
N(2,3)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,3)	2.80	2.80	2.80	2.80	2.80	2.64
N(4,3)	5.66	5.66	5.48	5.48	5.48	5.30
N(5,3)	9.88	9.88	9.82	9.82	9.82	7.90
N(6,3)	13.94	13.94	13.38	13.38	13.38	11.94
N(8,3)	30.56	30.56	28.94	28.94	28.94	26.30
N(10,3)	49.56	49.56	44.48	44.48	44.48	38.80
N(2,4)	1.44	1.44	1.44	1.44	1.44	1.44
N(3,4)	3.14	3.14	3.14	3.14	3.14	2.99
N(4,4)	7.06	7.42	6.98	6.98	6.98	6.66
N(5,4)	16.00	17.62	15.44	15.96	15.40	10.96
N(2,5)	1.46	1.46	1.46	1.46	1.46	1.46
N(3,5)	3.98	4.04	3.80	3.86	3.78	3.72
N(4,5)	12.66	15.00	12.12	12.26	12.06	11.90
N(2,6)	1.46	1.46	1.46	1.46	1.46	1.46
N(3,6)	4.52	4.88	4.20	4.32	4.18	4.00

TABLE XVI.

Average number of leaf nodes created for nonuniform trees with 0.6-ordered-independent static-values assignment

Tree size	AB	BB	PAB	Scout	PVS	SSS*
N(2,2)	1.56	1.56	1.56	1.56	1.56	1.56
N(3,2)	2.48	2.48	2.48	2.48	2.48	2.42
N(4,2)	3.62	3.62	3.62	3.62	3.62	3.22
N(5,2)	4.96	4.96	4.96	4.96	4.96	4.02
N(6,2)	6.32	6.32	6.32	6.32	6.32	5.30
N(8,2)	10.70	10.70	10.70	10.70	10.70	8.46
N(10,2)	14.50	14.50	14.50	14.50	14.50	10.32
N(24,2)	61.28	61.28	61.28	61.28	61.28	36.60
N(2,3)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,3)	2.76	2.76	2.74	2.74	2.74	2.64
N(4,3)	5.70	5.70	5.60	5.60	5.60	5.26
N(5,3)	9.80	9.80	9.72	9.72	9.72	7.98
N(6,3)	13.86	13.86	12.98	12.98	12.98	11.90
N(8,3)	29.78	29.78	26.68	26.68	26.68	25.22
N(10,3)	44.14	44.14	37.98	37.98	37.98	37.10
N(2,4)	1.42	1.42	1.42	1.42	1.42	1.42
N(3,4)	3.10	3.12	3.06	3.08	3.06	2.98
N(4,4)	6.98	7.14	6.80	6.88	6.77	6.50
N(5,4)	11.12	11.98	11.00	11.08	11.00	10.90
N(2,5)	1.44	1.44	1.44	1.44	1.44	1.44
N(3,5)	3.90	3.98	3.76	3.82	3.74	3.68
N(4,5)	10.84	11.26	10.02	10.26	10.00	9.54
N(2,6)	1.44	1.44	1.44	1.44	1.44	1.44
N(3,6)	4.06	4.52	3.98	4.02	3.98	3.96

TABLE XVII.

Average number of leaf nodes created for nonuniform trees with 0.8-ordered-independent static-values assignment

Tree size	AB	BB	PAB	Scout	PVS	SSS*
N(2,2)	1.54	1.54	1.54	1.54	1.54	1.54
N(3,2)	2.44	2.44	2.44	2.44	2.44	2.40
N(4,2)	3.30	3.30	3.30	3.30	3.30	2.90
N(5,2)	4.12	4.12	4.12	4.12	4.12	3.98
N(6,2)	5.42	5.42	5.42	5.42	5.42	4.66
N(8,2)	10.44	10.44	10.44	10.44	10.44	7.42
N(10,2)	13.38	13.38	13.38	13.38	13.38	9.52
N(24,2)	43.44	43.44	43.44	43.44	43.44	23.72
N(2,3)	1.48	1.48	1.48	1.48	1.48	1.48
N(3,3)	2.70	2.70	2.68	2.68	2.68	2.60
N(4,3)	5.62	5.62	5.60	5.60	5.60	5.24
N(5,3)	8.98	8.98	8.88	8.88	8.88	7.48
N(6,3)	12.90	12.90	12.66	12.66	12.66	11.76
N(8,3)	26.98	26.98	24.16	24.16	24.16	24.04
N(10,3)	40.12	40.12	39.04	39.04	39.04	36.60
N(2,4)	1.42	1.42	1.42	1.42	1.42	1.42
N(3,4)	3.06	3.10	3.02	3.02	3.02	2.98
N(4,4)	6.84	7.02	6.60	6.76	6.56	5.56
N(5,4)	10.98	11.24	10.96	10.96	10.94	10.80
N(2,5)	1.44	1.44	1.44	1.44	1.44	1.44
N(3,5)	3.78	3.84	3.70	3.74	3.68	3.62
N(4,5)	10.18	10.96	10.00	10.12	10.00	9.26
N(2,6)	1.42	1.42	1.42	1.42	1.42	1.42
N(3,6)	3.60	3.92	3.52	3.58	3.50	3.46

TABLE XVIII.

Average number of leaf nodes created for nonuniform trees with 1.0-ordered independent static-values assignment real-dependent scheme. For example for N(10,2) with real-dependent static-values assignment (Table XIII) SSS* created on average 7.60 leaf nodes, for the same trees with integer-dependent static-values assignment (Table XII) SSS* created 11.28 leaf nodes on average, and for trees with unordered-independent static-values assignment (Table XIII) SSS* created 21.42 leaf nodes on average. As we see the difference in number of leaf nodes created for the three static-values-assignment schemes was not as sharp as that observed for uniform trees. For the P-ordered-independent schemes, as the value of P increased, there was a decrease in the average number of leaf nodes created. Knuth and Moore [12, page 307] showed that for nonuniform trees ' perfect-ordering is not always the best.' The experimental results show that for a given nonuniform tree, perfect-ordering is at least as good as any other P-ordering ($P < 1$). As an example : for N(5,4) with 0.2-ordered static-values assignment Alphabeta created 18.80 leaf nodes on average (Table XIV) and for trees with 0.8-ordered static-values assignment it created only 11.12 nodes (Table XVIII).

3.6.3. Comparison Based on Number of Node-Visits.

Above the performance of pruning strategies based on the number of nodes created was compared. One could argue that it is not enough, because it has been observed that the pruning strategies which create the fewest nodes are not necessarily the fastest. For example, no other pruning strategy ever created fewer nodes than SSS*, but SSS* was found to be the slowest, because not only does it visit some nodes more than once, it also has to maintain a sorted list. Apart from Alphabeta and Branch-and-bound, all the other strategies visit many nodes more than once, thus slowing down the strategy. But this may not always hold in real games where the greatest amount of work is done in move generation and evaluation, and extra time taken on subsequent visits may be marginal because the move would already have been generated and evaluated. Moreover, the time taken may also depend on the data structures used in the program. The average number of node-visits for the pruning strategies, relative to one another, kept changing with the type of tree, its depth and width, and the scheme adopted to assign static-values to its leaf nodes. So no general statement can be made about comparative performance of pruning strategies under the criterion of node-visits. However, it was noticed that usually Alphabeta visits the least nodes. For trees of depth equal to 2 PVS visits on average same amount of nodes as Palphabeta, less than Scout. For trees of depth equal to 3 PVS visits on average more nodes than Palphabeta but still less than Scout. For trees of depth greater than three performance of PVS under the criterion of node-visits is usually the worst among all tested pruning strategies. For example for U(8,2) with 0.6-ordered-independent static-values assignment PVS visited on average 31.82 nodes and Scout 38.06, but for U(3,5) with 0.4-ordered-independent static-values assignment PVS visited on average 239.34 nodes and Scout visited 213.29 nodes. For trees of depth greater than or equal 4 and width greater than 3, Palphabeta usually visited less nodes than Branch-and-bound. For example for U(2,5) with 0.4-ordered-independent static-values assignment Branch-and-bound visited 43.30 nodes on average and Palphabeta 49.54, but for U(4,5) with same static-values assignment Branch-and-bound visited on average 491.38 nodes and Palphabeta 457.76. Performance of SSS* under the criterion of node-visits had a tendency to vary, its position among other algorithms kept changing from the last one up to the second best. For example for U(2,3) with 0.8-ordered-independent static-values assignment SSS* was the worst, but for U(5,2) with 0.2-ordered-independent static-values assignment SSS* was second best. For nonuniform trees the difference between performance of pruning strategies under the citerion of node-visits was not as sharp as for uniform trees.

3.6.4. Comparison Based on CPU Time Taken.

In this section the CPU time taken by the pruning strategies will be discussed. Comparing algorithms on the

CPU time taken can be questioned because it may depend on efficiency of program coding. We tried to code the programs as intuitively efficient as possible. Apart from SSS* the performance of pruning strategies in terms of node-visits correspond to CPU time taken by the pruning strategy. We found that usually Alphabeta and Branch-and-bound took the least CPU time. For trees with a small total number of nodes Branch-and-bound performs slightly better than Alphabeta, for bigger trees Alphabeta was the best and Branch-and-bound the second best. The other pruning strategies in increasing order of CPU time taken can be listed as Palphabeta, Scout, PVS and SSS*. In Figures 25 to 40 the CPU time taken by the various pruning strategies for uniform and nonuniform trees of depth 4 and widths from 2 to 5, when the static-values were assigned using all the discussed approaches are plotted. As we can see, the least CPU time was taken for trees with 1.0-ordered-independent static-values assignment. As the value of P decreases in P-ordered-independent scheme there was an increase in CPU time taken by any pruning strategy. The most CPU time was taken for trees with 0.2-ordered-independent static-values assignment. For example for U(4,4) with 1.0-ordered static-values assignment PVS had taken on average 4.46 CPU time in milliseconds (Figure 32), whereas for U(4,4) with 0.2-ordered scheme it had taken 23.32 CPU time in milliseconds (Figure 28). For uniform trees as well as for nonuniform trees all pruning strategies performed a little

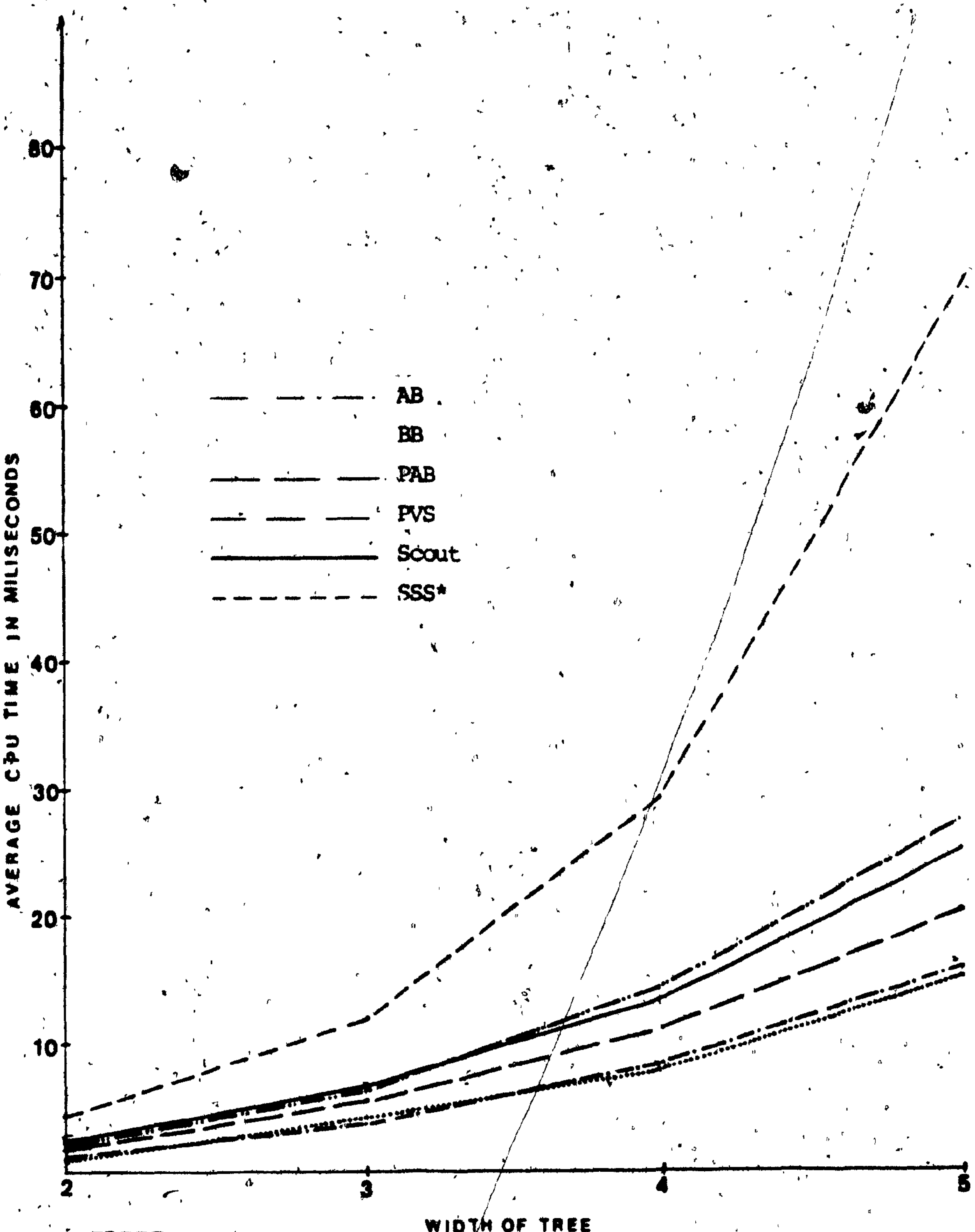


FIGURE 25.

Plot of average CPU time taken against width of a uniform tree with depth 4. Static-values were assigned to leaf nodes by integer-dependent scheme.

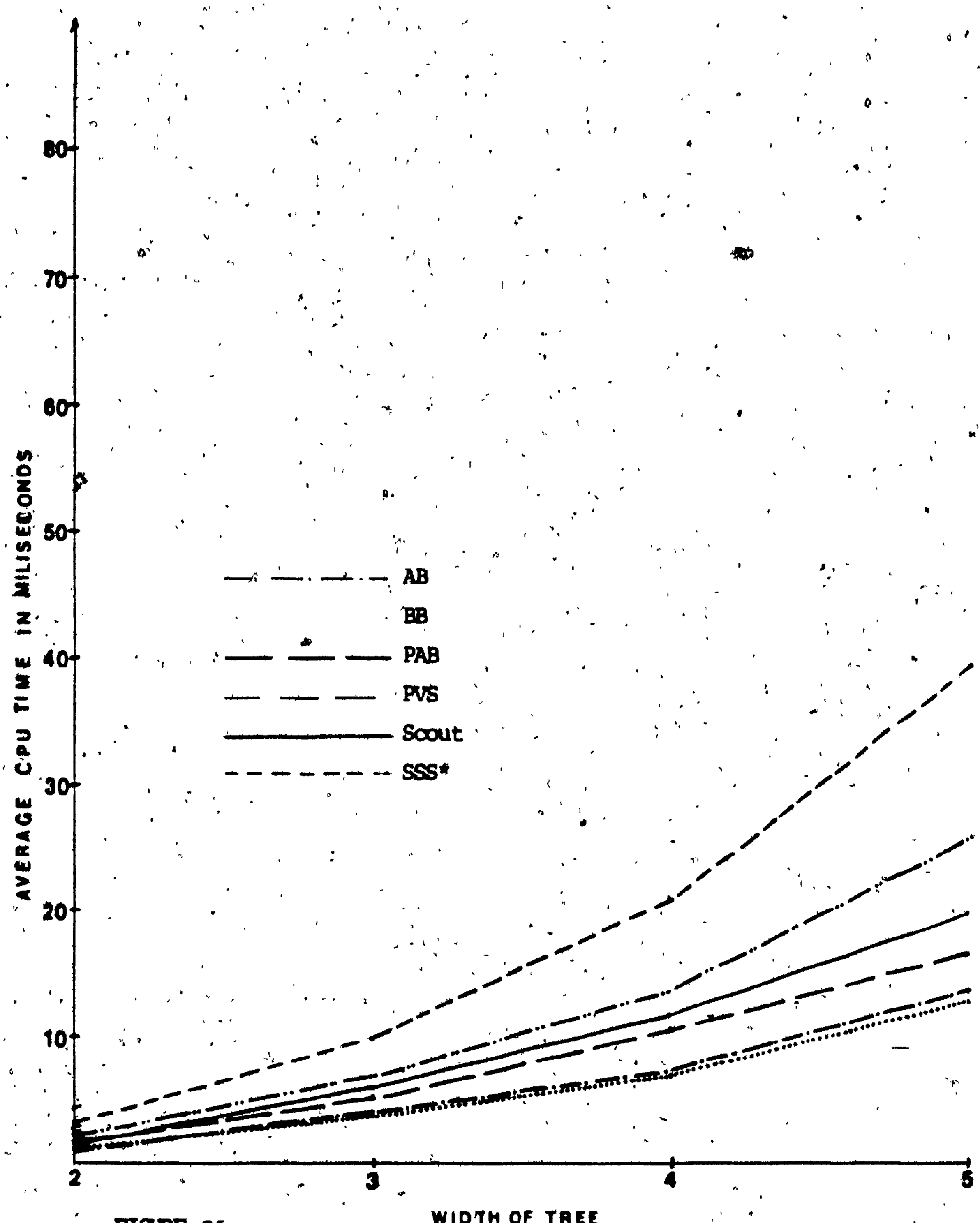


FIGURE 26.

Plot of average CPU time taken against width of a uniform tree with depth 4. Static-values were assigned to leaf nodes by real-dependent scheme.

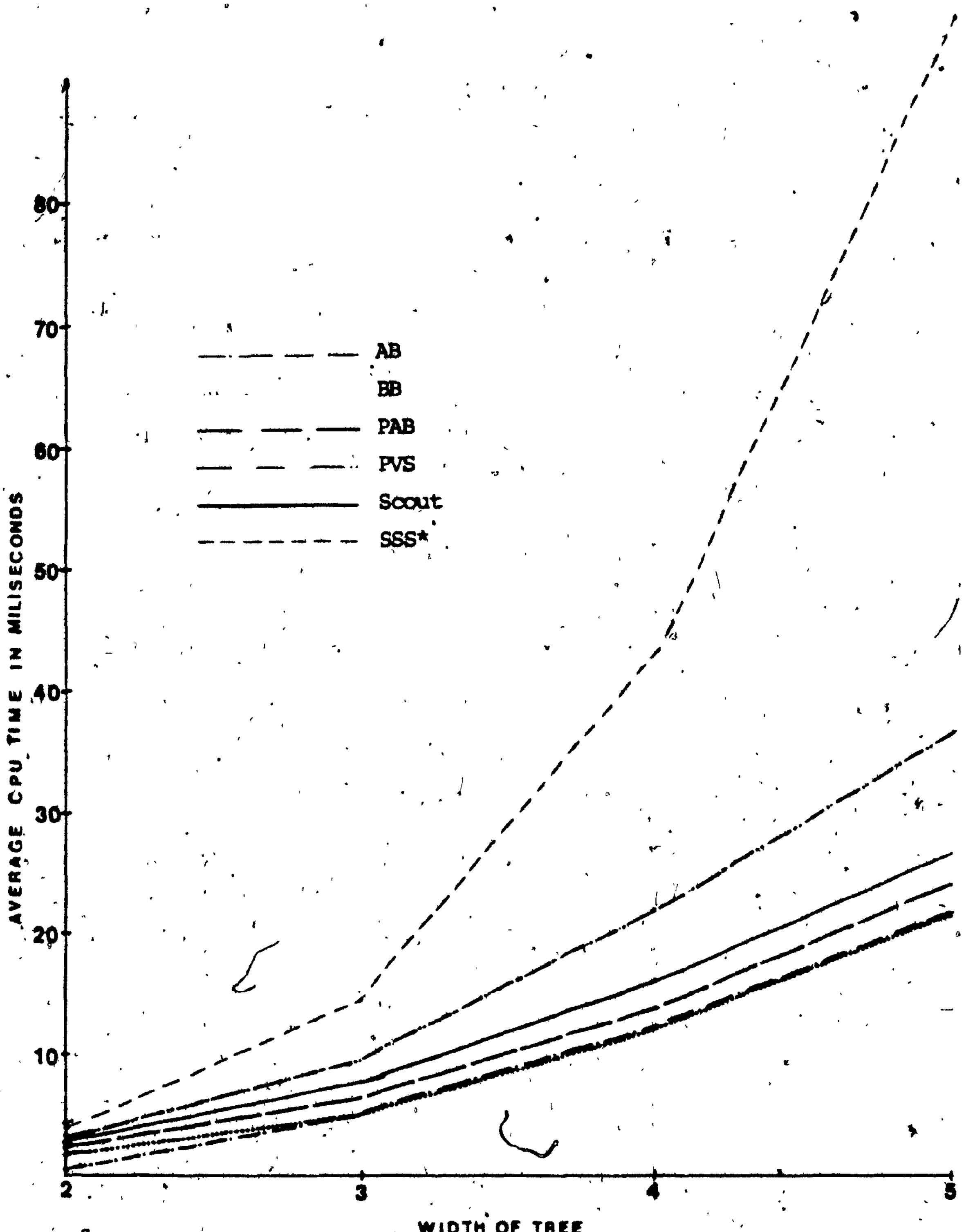


FIGURE 27.

Plot of average CPU time taken against width of a uniform tree with depth 4. Static-values were assigned to leaf nodes by unordered-independent scheme.

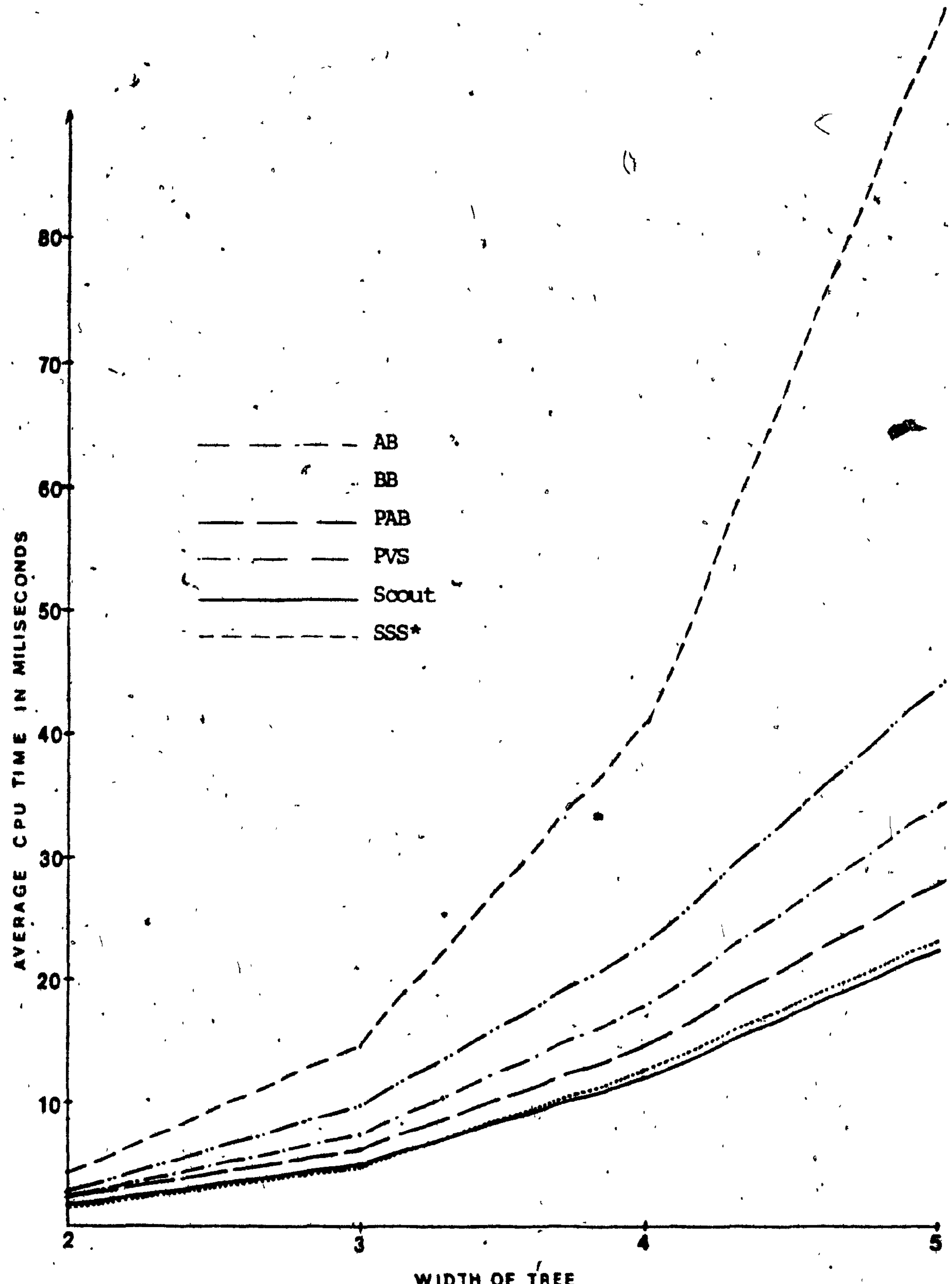


FIGURE 28.

Plot of average CPU time taken against width of a uniform tree with depth 4. Static-values were assigned to leaf nodes by 0.2-ordered-independent scheme.

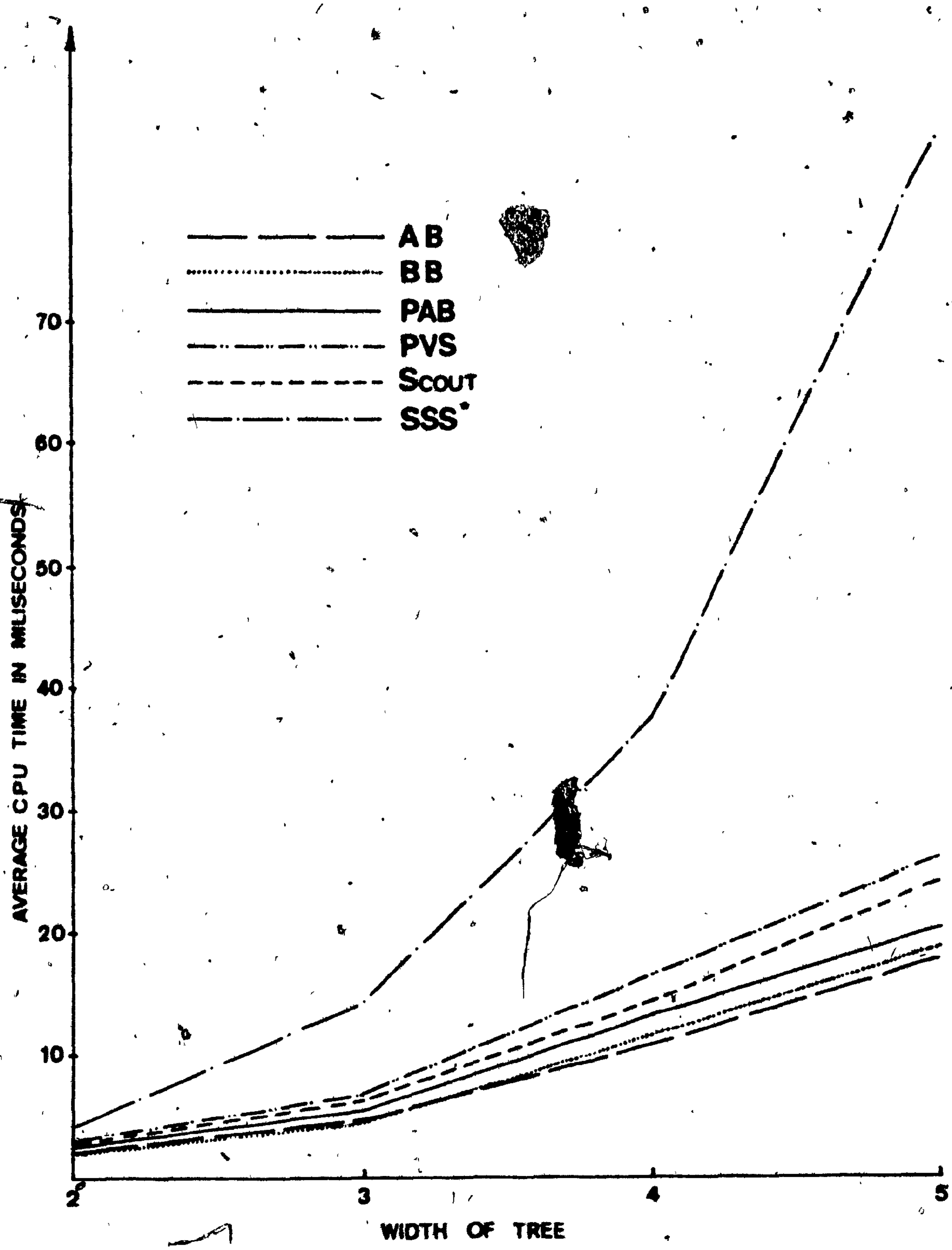


FIGURE 29.

Plot of average CPU time taken against width of a uniform tree with depth 4. Static-values were assigned to leaf nodes by 0.4-ordered-independent scheme.

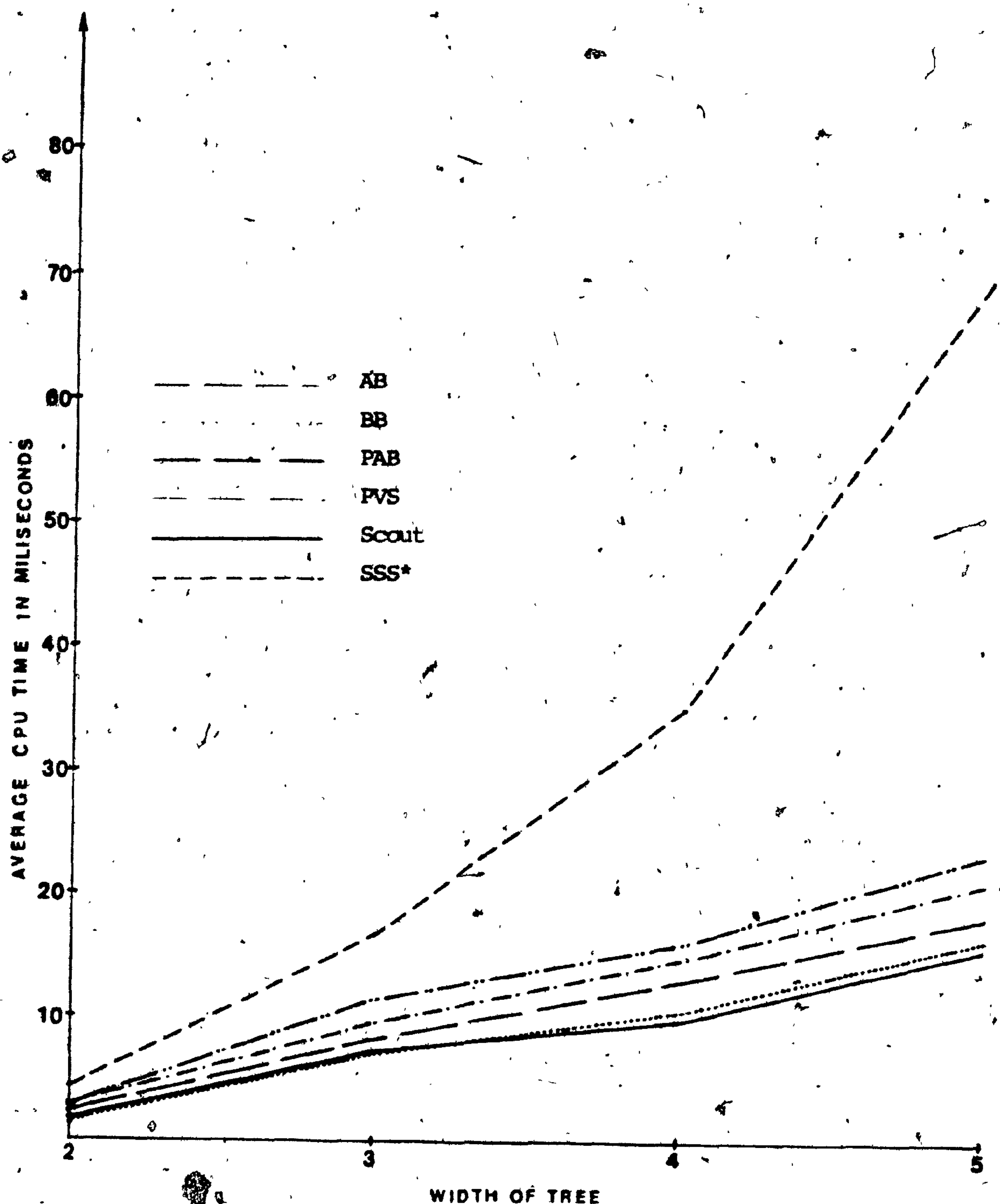


FIGURE 30.

Plot of average CPU time taken against width of a uniform tree with depth 4. Static-values were assigned to leaf nodes by 0.6-ordered-independent scheme.

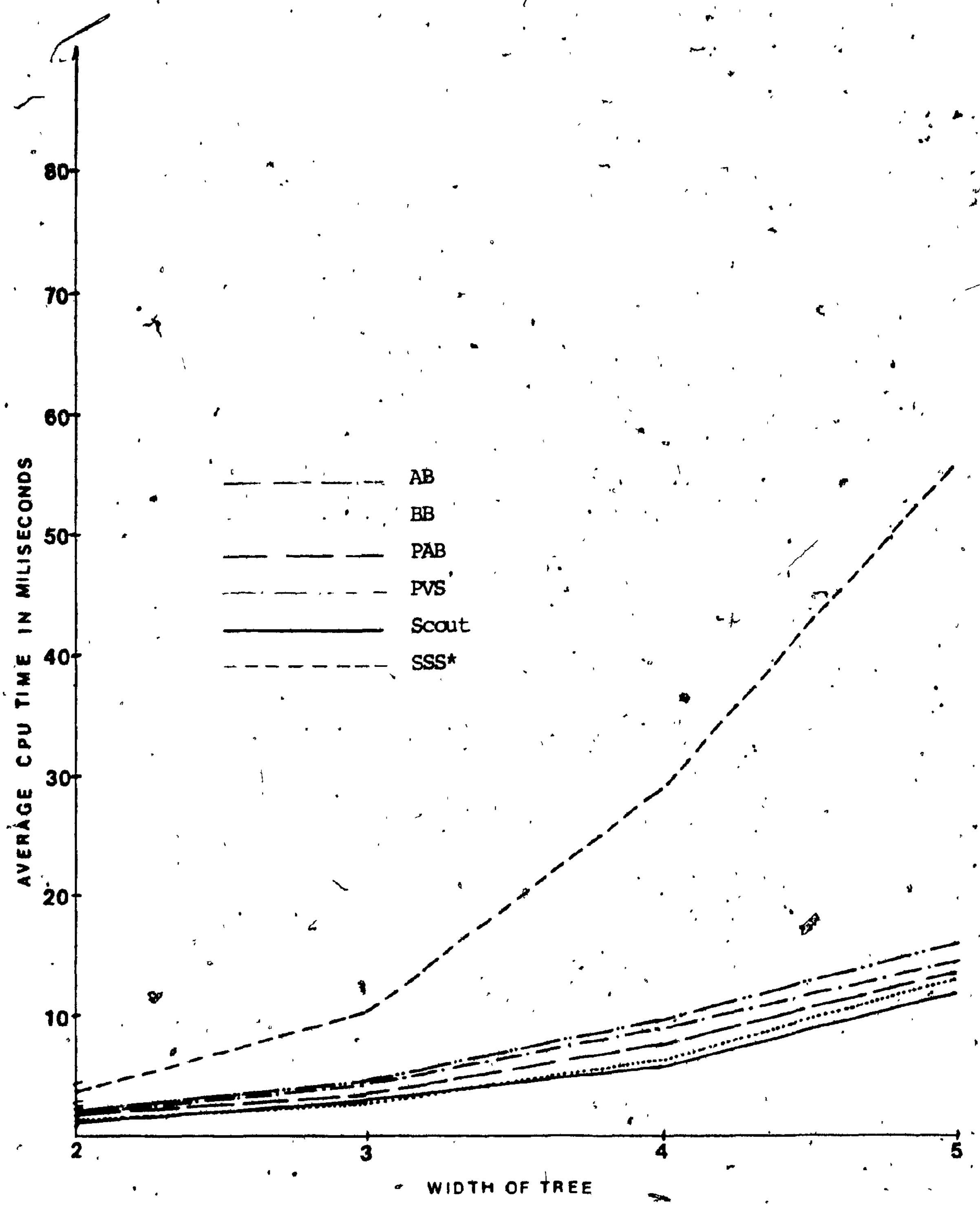


FIGURE 31.

Plot of average CPU time taken against width of a uniform tree with depth 4. Static-values were assigned to leaf nodes by 0.8-ordered-independent scheme.

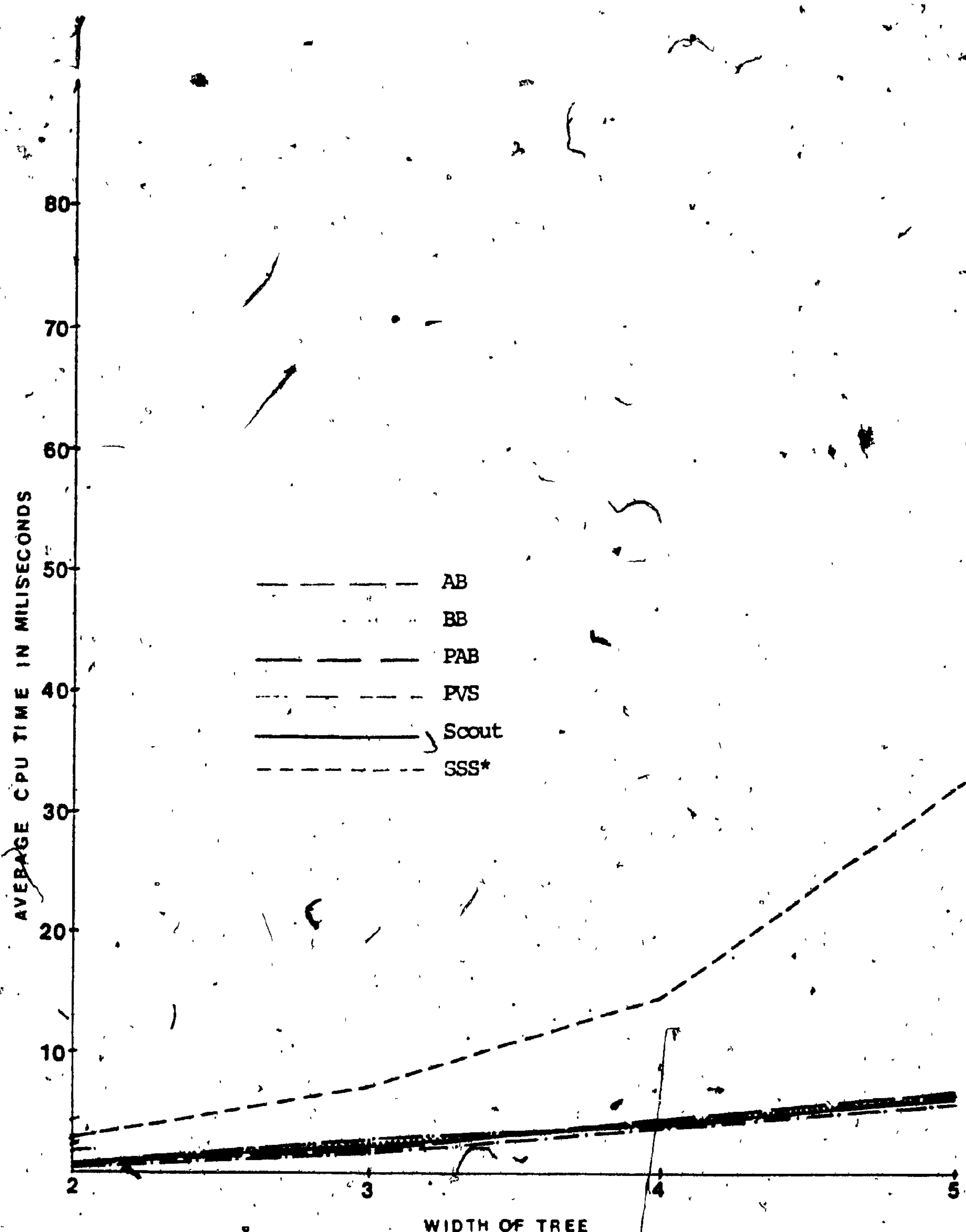


FIGURE 32.

Plot of average CPU time taken against width of a uniform tree with depth 4. Static-values were assigned to leaf nodes by 1.0-ordered-independent scheme.

FIGURE 33.

Plot of average CPU time taken against width of a nonuniform tress with depth 4. Static-values were assigned to leaf nodes by integer-dependent scheme.

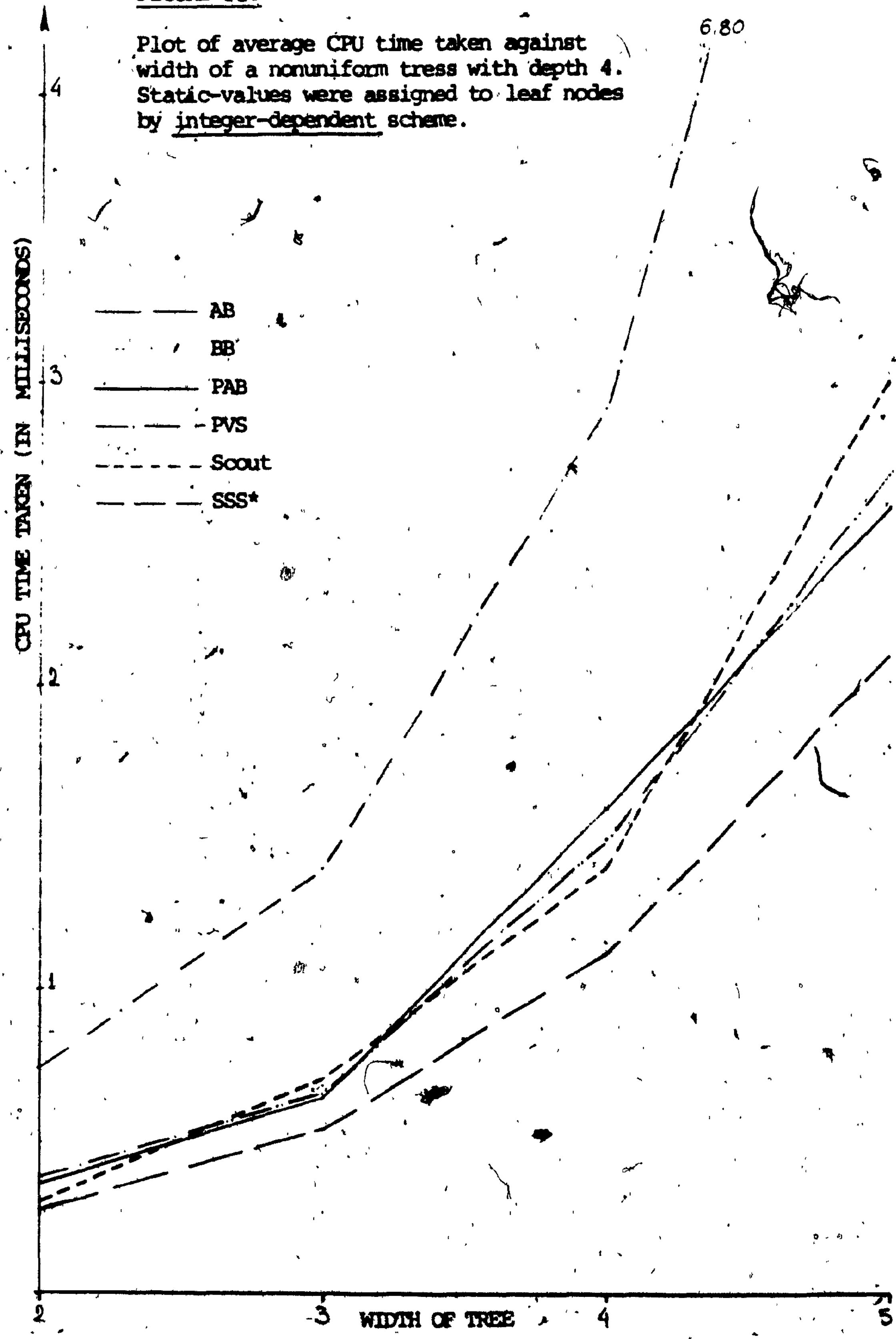


FIGURE 34.

Plot of average CPU time taken against width of a nonuniform tree with depth 4. Static-values were assigned to leaf nodes by real-dependent scheme.

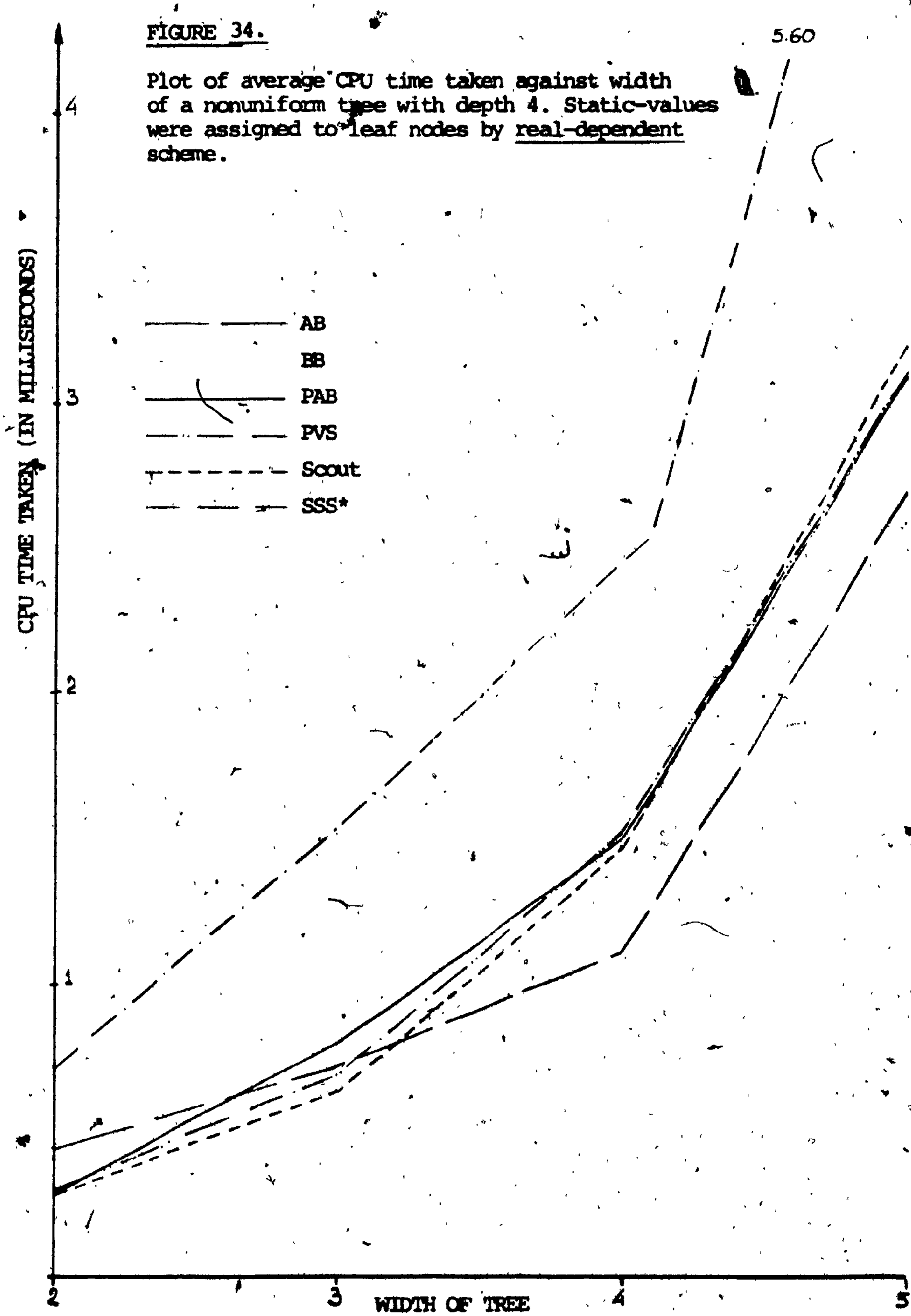


FIGURE 35.

Plot of average CPU time taken against width of à nonuniform tree with depth 4. Static-values were assigned by unordered-independent scheme.

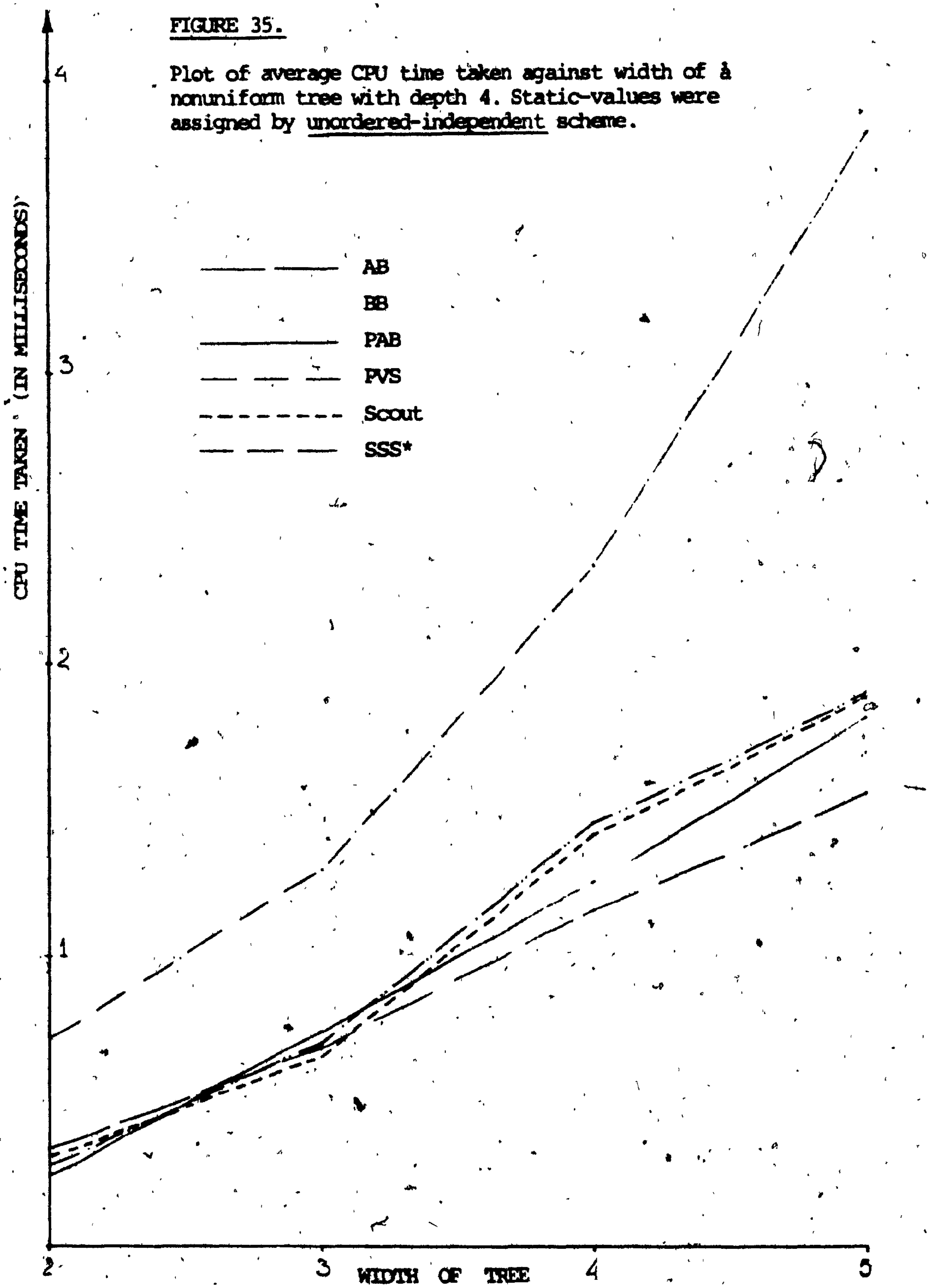
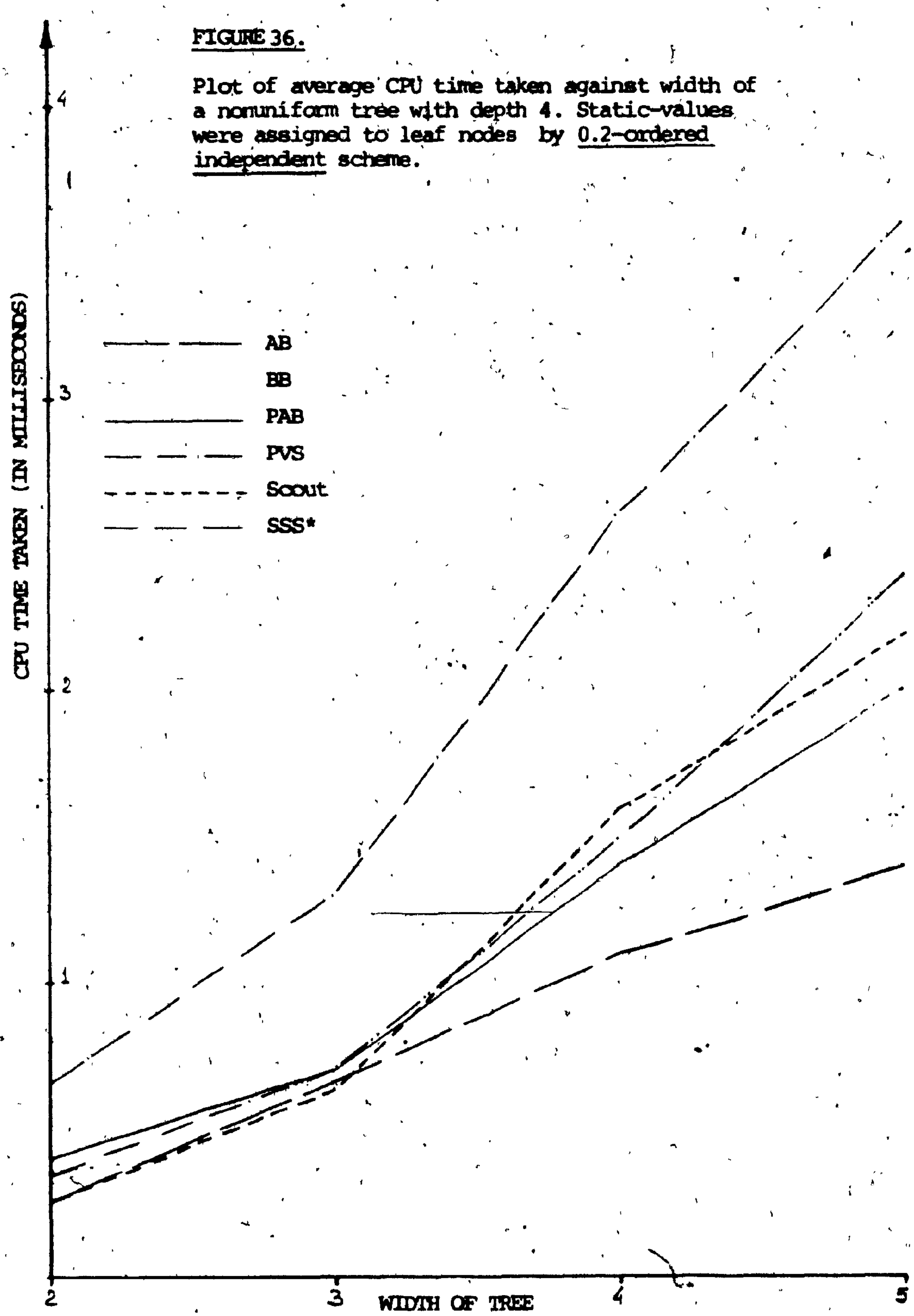


FIGURE 36.

Plot of average CPU time taken against width of a nonuniform tree with depth 4. Static-values were assigned to leaf nodes by 0.2-ordered independent scheme.



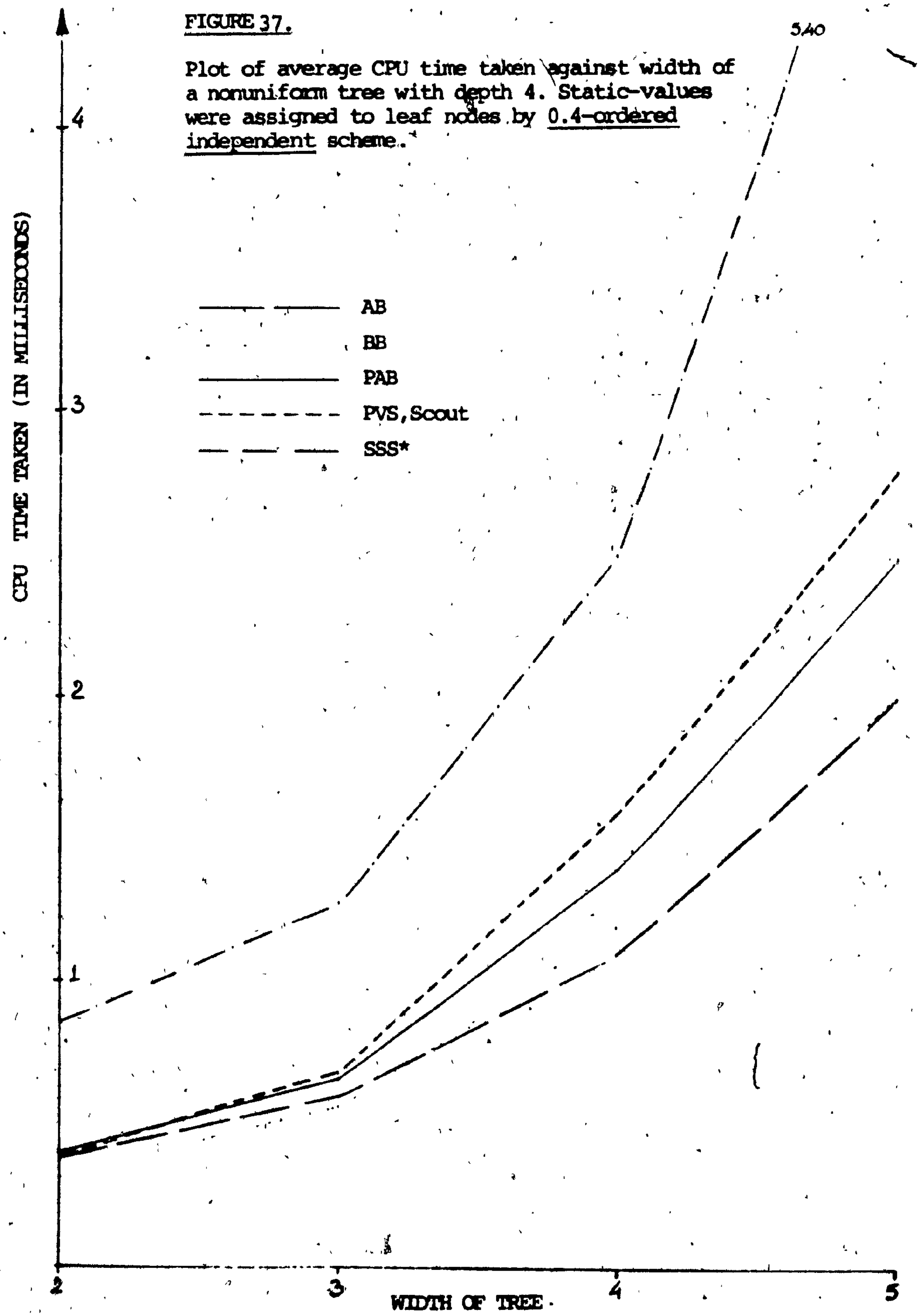


FIGURE 37.

Plot of average CPU time taken against width of a nonuniform tree with depth 4. Static-values were assigned to leaf nodes by <u>0.4-ordered independent</u> scheme.

FIGURE 38.

Plot of average CPU time taken against width of a nonuniform tree with depth 4. Static-values were assigned to leaf nodes by 0.6-ordered-independent scheme.

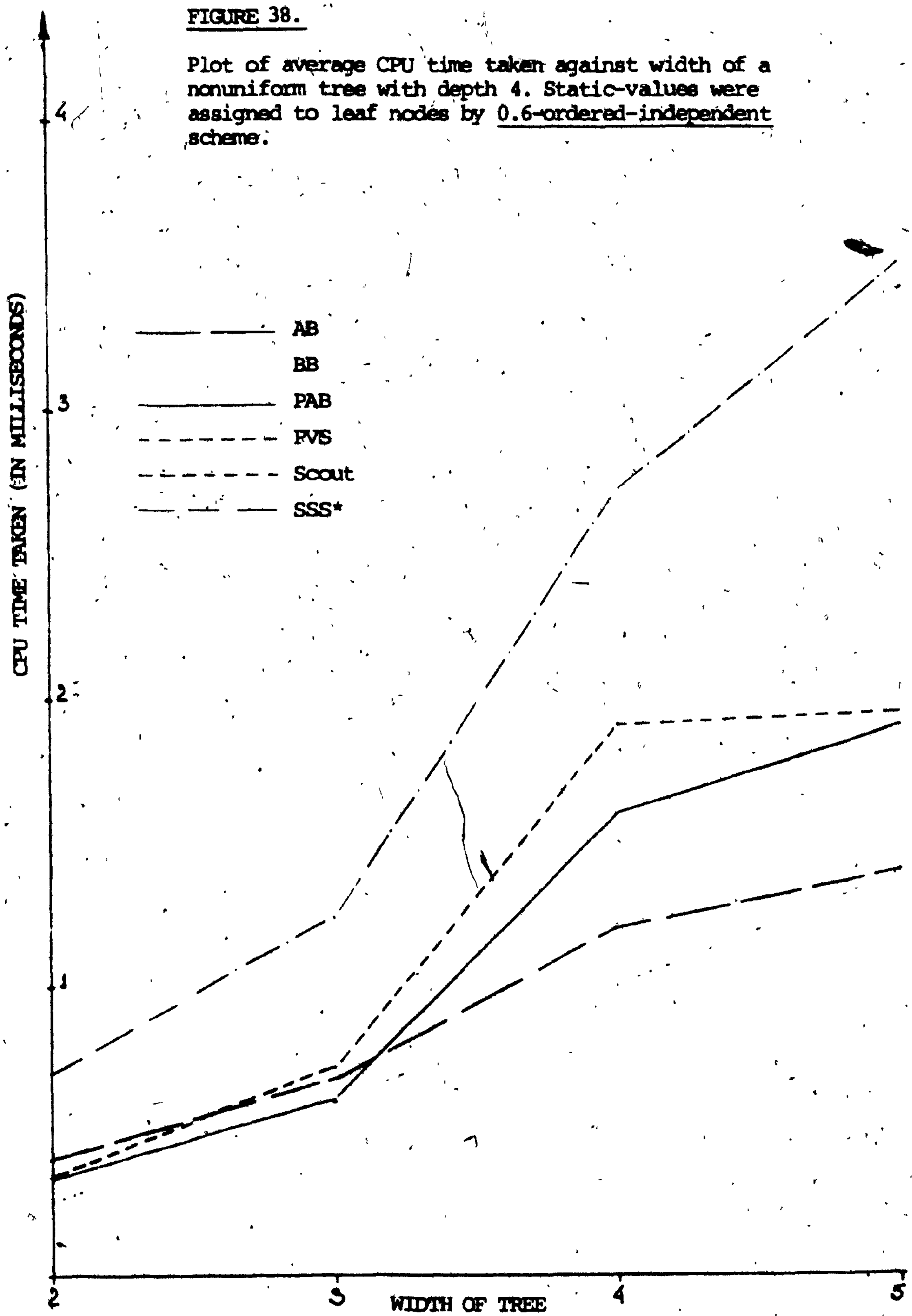


FIGURE 39.

Plot of average CPU time taken against width of a nonuniform tree with depth 4. Static-values were assigned to leaf nodes by <u>0.8-ordered-independent</u> scheme.

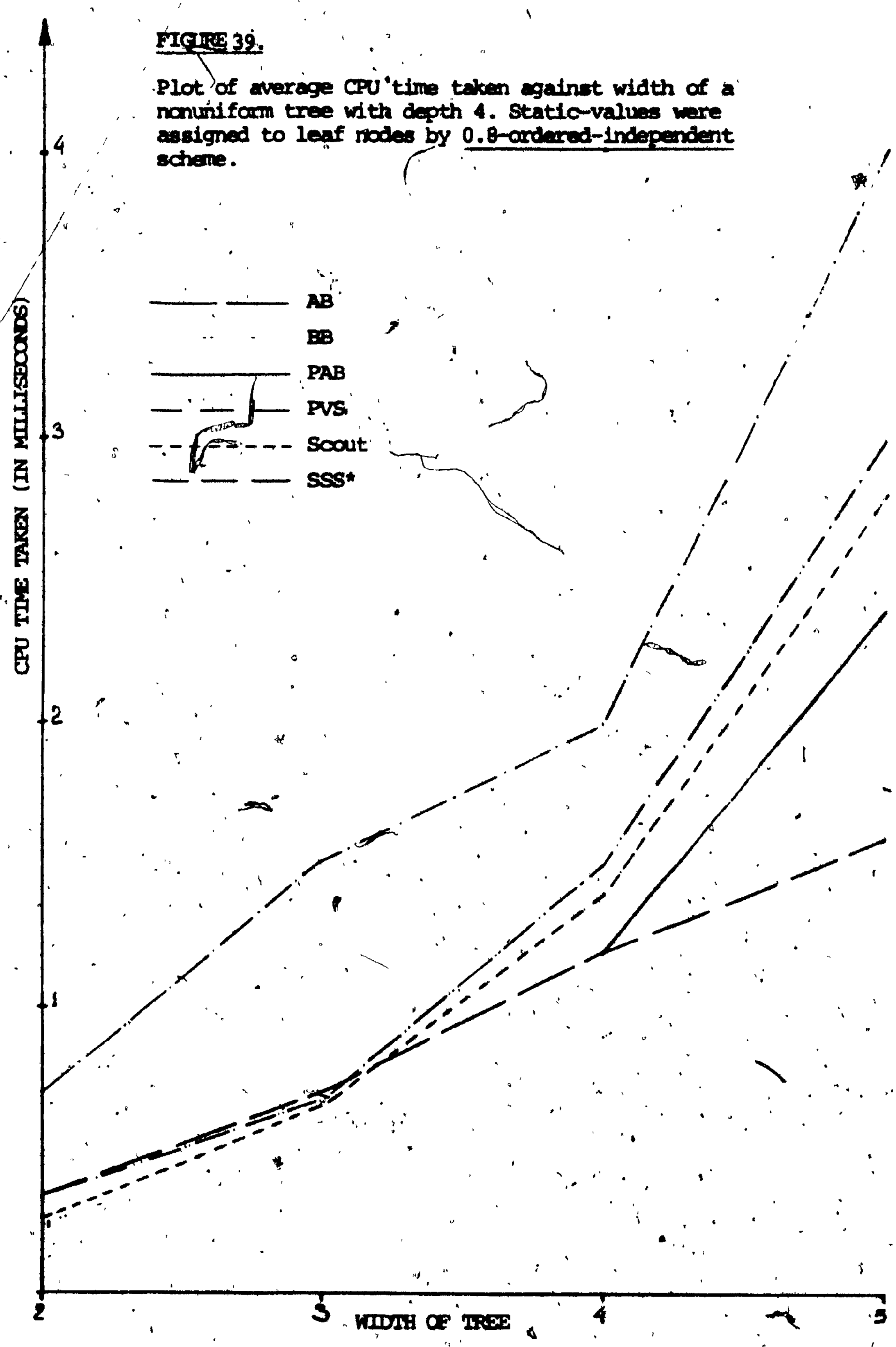
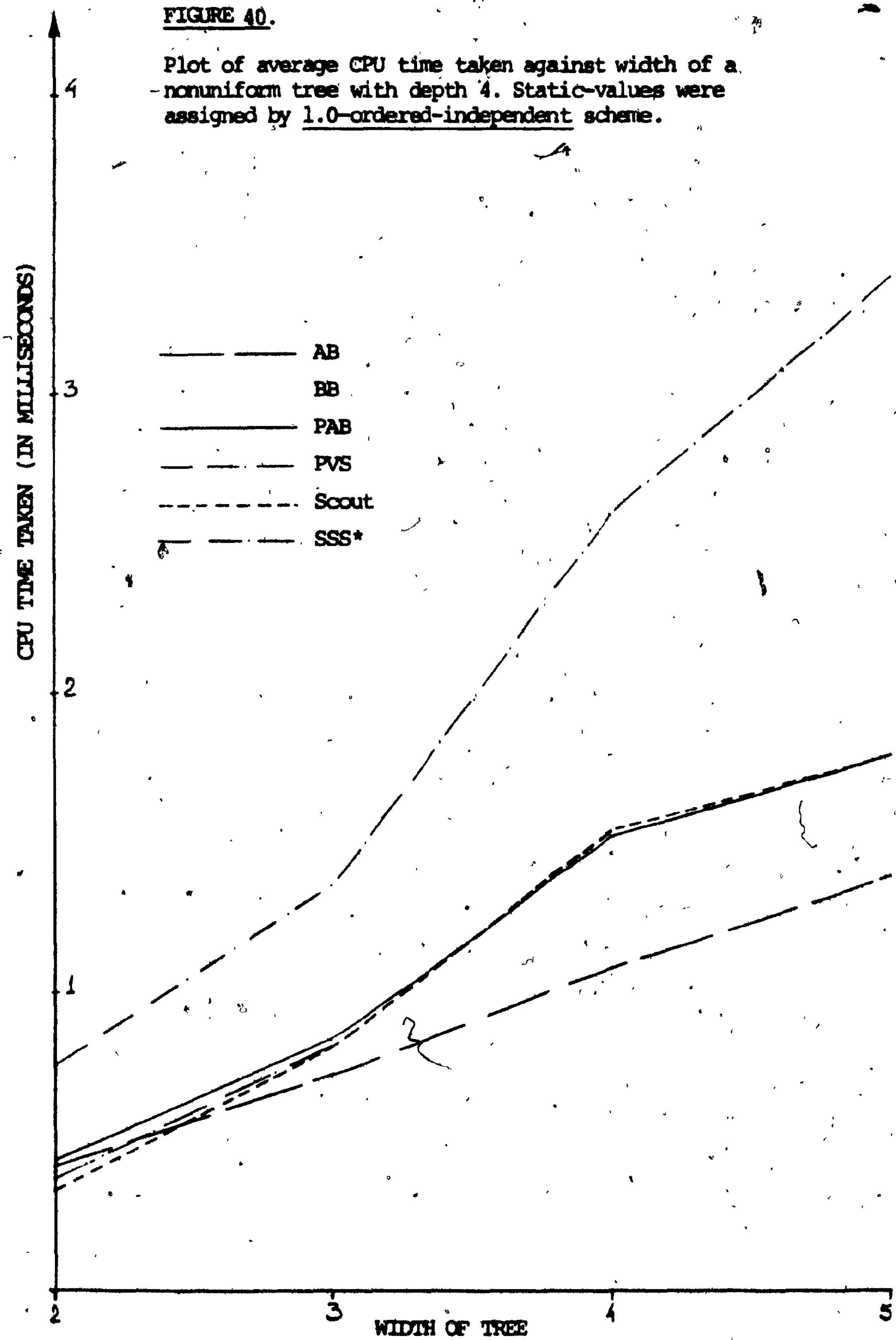


FIGURE 40.

Plot of average CPU time taken against width of a nonuniform tree with depth 4. Static-values were assigned by 1.0-ordered-independent scheme.



worse for trees with real-dependent static-values assignment than for trees with 1.0-ordered scheme under the criterion of average CPU time taken. For trees with integer-dependent static-values assignment pruning strategies performed better than for trees with 0.4-ordered-independent scheme. For unordered-independent scheme pruning strategies performed better than for trees with 0.2-ordered-independent scheme. These results are representative for all kinds of tested trees, although for nonuniform trees the differences were not as sharp as for uniform trees.

3.7. Overall Remarks on the Pruning Strategies.

Six pruning strategies were compared on uniform and nonuniform game trees of twenty-four different sizes, each being assigned leaf-node-static-values under four different schemes. We found that no strategy ever created fewer nodes than SSS*, confirming the theoretical results of [28,30]. However, SSS* was the slowest, mainly because it required maintaining a large sorted list. Moreover, to maintain this list SSS* also required extra storage. Palphabeta, PVS and Scout were slower than Alphabeta and Branch-and-bound because the former three strategies visited many nodes more than once. Kumar and Kanal [13] argue that pruning strategies are special cases of a Generalized Branch-and-bound. In theory one may agree with them completely, but empirically the performance of the pruning strategies varies substantially.

Based on the theoretical results given in [24,31], it has been concluded by Pearl [26,27] that Alphabeta is asymptotically optimal over all algorithms that search uniform game tree with unordered-independent-static-values assignment. Our experiments have shown that Alphabeta usually takes the least CPU time. Considering Pearl's results and our results together with the quick response often required while playing actual games, we conjecture that the Alphabeta algorithm will continue to be popular as a pruning strategy, when used in conjunction with the minimax procedure. We however caution that this conclusion is based on sequential implementation of pruning strategies. The strategies may perform differently under parallel implementation [1,2,11,14].

CHAPTER 4.

METHODS OF SPEEDING-UP THE TREE SEARCH.

In this chapter we will present a survey of some of the known methods developed for speeding-up the pruning strategies. Different parallel implementations of the Alphabeta algorithm, one proposed by Akl et al. [1] and another one by Finkel et al. [11] will be presented. The parallel versions of the Scout [2] and SSS* [14] algorithms will be described. Other methods of speeding-up the game tree search, such as nodes ordering [29], transposition table [15] and the killer heuristic [3] will be also discussed.

4.1. Parallel Implementations of the Pruning Strategies.

To reduce search-time for the game trees the parallel versions of the pruning strategies were introduced [1,11,14]. In [1] a parallel implementation of the Alphabeta algorithm is presented. In this implementation disjoint subtrees are searched concurrently. Assuming that the tree to be searched is perfectly ordered, the nodes that have to be created are visited first. So the distinction is made among the sons of a node. The leftmost son is called left-son, others are called right-sons (as shown in Figure 41). The left subtree of a node is searched by a left process (which

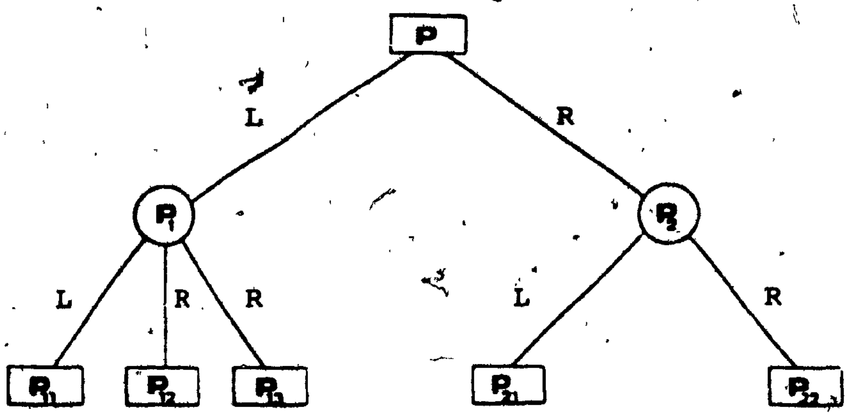
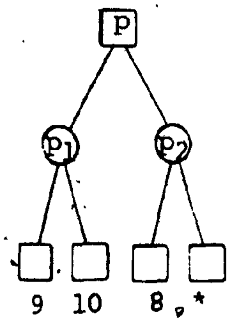


FIGURE 41.

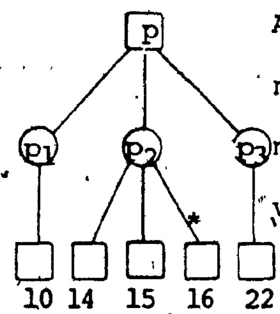
Distinction made among sons of nodes in a game tree. The leftmost sons of every nonleaf node represent the nodes which have to be created. They are visited first in a parallel version of the Alpahabeta algorithm, as described in [1].

is spawned by the parent node) until the value for the left son is backed-up to the parent node. To obtain this value the left-son process spawns processes (left and right) to search all its subtrees. Concurrently a temporary value is obtained for each of the right-sons of the parent node. These values are computed by the right son's spawning a process to search its left subtree. Then these temporary values are compared to the final value of the left-son and all possible cut-offs are made.

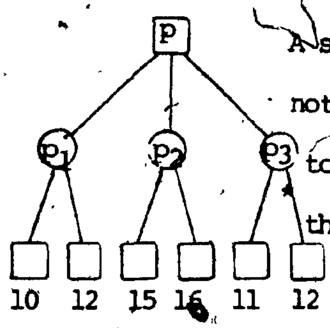
All shallow cut-offs which occur in the sequential search due to the temporary value backed-up to a node from its left-son, will also occur in this parallel search. Some shallow cut-offs may be missed by the parallel search. For example, when a process is generated to search the second-right subtree of the root before the first-right subtree of the root completes its search and update the root's value. Some cut-offs that are missed in sequential search will occur in parallel search. For example, a right subtree that terminates early and causes a change in parent's value may provide cut-offs in other right subtrees. It should be noted that the deep cut-offs allowed by the sequential Alphabeta do not occur in the parallel version of this algorithm because initially the nodes of the right subtree do not 'know' about the values obtained by the left son of the root. In Figure 42 the cut-offs which may occur in the parallel version of Alphabeta are presented.



A shallow cut-off which will occur in sequential and in parallel versions of Alphabeta algorithm. It is shown by *.



A shallow cut-off which will occur in parallel, but not in sequential version of Alphabeta. Search below node p_3 ends earlier and causes a change in the bound value for node p, so search at node p_{23} is cut-off.



A shallow cut-off which will occur in sequential but not in parallel version of Alphabeta. The process to search the subtree below p_3 is generated before the search below p_2 has been completed.

FIGURE 42.

Comparison of cut-offs which occur in sequential and parallel Alphabeta.

Experimental results for this parallel algorithm regard the total run time, number of nodes created and number of nodes-visits for uniform trees U(w,d) where static-values were assigned from a particular probability distribution. The run time decreases sharply with the increasing number of processors. Also the number of created nodes and number of node-visits increase with the increasing number of processors, but these increments are relativelly small and a saturation point is reached quickly (for 5 processors). The behavior of the parallel algorithm remains unchanged for differrent probabilty distributions of the static-values.

The same parallel implementation of Alphabeta was empirically compared with parallel version of the Scout algorithm in [2] for the game tree representing the game of checkers. A sequential version of Scout algorithm uses procedure EVAL to examine the leftmost son of the node and calls the TEST procedure for inequality checking for the other sons. A parallel implemenation of the EVAL procedure evaluates the leftmost son of node while concurrently testing all other sons. This is acomplished by letting the process which calls the EVAL to create an EVAL-process to examine the leftmost son of the node and TEST-processes to test other sons. Synchronisation is required to ensure that the evaluation of the leftmost son is completed before any attempt is made to compare the value of a leaf node with value of the leftmost son. A distinction is made among the sons of a node. A node which is evaluated by EVAL is called E-node while a node that is tested by TEST is called a T-node. E-nodes and T-nodes are searched by EVAL-processes and TEST-processes respectively. A T-node may become an E-node if it is not exempted by the test. The root is an E-node and it generates an EVAL-process to evaluate its leftmost son. The root node also generates TEST-processes to concurrently test all its other sons.

The experimental results in [2] for comparison of the parallel Alphabeta and parallel Scout show the rapid reduction of the run time taken by both algorithms with increasing number of processors used to search the game tree. The saturation point is reached for about 5 or 6 processors. Beyond that point the run time remains relatively constant as the number of processors increases. The total number of created nodes and of node-visits for both algorithms increases slightly with the icreasing number of used processors. The saturation point is also reached for about 5 or 6 processors. It was noticed that the parallel Scout was slightly more efficient than the parallel Alphabeta for the opening checkers game. For the mid-game and for the endgame the Alphabeta algorithm was distinctly superior.

Another version of the parallel Alphabeta algorithm is presented in [11]. In this implementation a game tree is decomposed into several parts which are searched simultaneously. Because of such a decomposition, some subtrees of the game tree which are not searched in sequential version may be searched by the parallel version of Alphabeta, and some cut-offs may be missed by the parallel version of the algorithm. The concurency in search assures, however, the speed-up of execution. Analysis of the parallel Alphabeta is done on a parallel computer built as a tree of the serial computers. A node in this tree is a processor. A processor's parent is its master and its son is its slave. In this parallel implementation of the Alphabeta the root proccessor evaluates the root position. Each nonleaf processor evaluates its assigned position by generating the sons and queuing them for the parallel assignment to its slave processors. A separate process is created for each son and each process attempts to gain exclusive control of a slave processor. As a nonleaf processor receives responses from its slaves it narrows its search window and acknowledges the working slaves about the new alpha and beta bounds. When all sons have been evaluated (or a cut-off has occured) the nonleaf processor is able to compute the value of its position. The leaf processor evaluates it's assigned node using the sequential Alphabeta algorithm. When a processor finishes, it reports the computed value to its master. A cut-off occurs when alpha bound has become greater than or equal to the beta bound.

In [11] the game of checkers was used to generate a game tree and 10 board positions were choosen. All game trees were generated up to depth 8. Processor-trees of depth 1, 2 and 3 and width 2 or 3 were simulated. The speed-up of a parallel version of the algorithm over the sequential version was defined [11] to be the ratio of the CPU time taken by the parallel version to the CPU time taken by sequential one. With increasing number of processors the value of speedup, for this parallel Alphabeta, is also increasing. Theoretical analysis shows that for the worst-case (the rightmost son is the best son) the speedup over the sequential version is equal to number of processors used. For the best-case (the leftmost son of any nonleaf node is its best son) value of speedup is about $k^{1/2}$, where k is the number of processors used. There were no empirical results presented about the number of nodes created or number of node-visits.

In [14] two schemes for performing the game tree search in parallel are discussed. These schemes were implemented for the SSS* algorithm [14]. In the first approach of the parallel implementation of the SSS* algorithm, the multiple processes perform a game tree search. For this approach several searching processes are initialized, one at each processor, with different starting bounds. One process is started with the most pessimistic bound to be sure that the search will be successful. At any time at least one process has the property that if it terminates, it returns an optimum solution. For this approach the speed-up of 25% was achieved when two processors were used for searching uniform trees of different depths and widths, with static-values assigned from an uniform distribution. This method of searching is useful if we have some information about the minimax value of the game tree.

In the second approach of the parallel implementation of the SSS* algorithm, as presented in [14], the game tree is divided into several disjoint parts and each part is searched concurrently in a depth-first manner by a different process; the processes work asynchronously. The parallel version of SSS* algorithm, obtained by this approach, was tested on uniform game trees. For every width and depth 50 trees were simulated, and the static-values were assigned from a uniform distribution. The speed-up was defined as the ratio of leaf nodes created by the sequential version of SSS* to the maximum of the number of nodes created by the separate processes in the parallel SSS*. The average speed-up was observed to vary from 1.71 to 4.95, depending on the width and depth of a simulated tree. It was also noticed that for trees for which the sequential version of SSS* created the largest number of leaf nodes the speed-up was larger than the average one. It means that this parallel implementation is more effective in the situations where it is needed the most.

4.2. Ordering of the Nodes in a Game Tree.

Slagle and Dixon [29] have shown the possibility of improving the Alphabeta algorithm by ordering the successors of a position. They proposed two methods : fixed ordering and dynamic ordering to achieve the speed-up of a tree-search. The fixed ordering procedure is based on the assumption that the static-value of a node is positively correlated with the node's value obtained by backing-up values from the deeper levels of a game tree. Using this method first we estimate the values of the sons of the root by the static-evaluation function. The static-ordering procedure orders the sons, so one with the highest static-value (most likely it will be the best son) is to be searched first. The procedure works in this fashion on subsequent levels of a game tree. Results presented by Slagle et al. [29] show that using the fixed-ordering, in number of leaf nodes created we may expect an improvement of two orders of magnitude over the exhaustive search. This method of ordering may make tree-searching faster, but not always. We may discover that our original estimate based on the static-value is wrong and that the nodes we have chosen to evaluate first has very unpromising backed-up value. So the dynamic ordering was proposed. Nodes at the first level of a game tree are ordered on the basis of their static-values. Then sons of the chosen node are evaluated. If any of them has very unpromising static-value, comparatively to other, then the reordering of the previous choice is done. The comparison of static and dynamic ordering is presented in Figure 43. Slagle and Dixon [29] experimentally showed that the dynamic ordering becomes worthwile for trees of depth greater than six. For the shallower trees the time spent on the reordering of nodes is same as the CPU time taken for searching additional nodes. The number of leaf nodes created for tree searching with the dynamic ordering is very close to the lower possible bound of number of leaf nodes created for a certain game tree.

4.3. Use of Transposition Tables.

When searching a game tree it is common to find nodes corresponding to the same positions of a game. Rather than rebuild the subtrees corresponding to the repeated positions, it is possible to retrieve the results stored by a previous search. The results may be stored in a large hash table, called the transposition table, with each entry representing a position, as described in [15] by Marsland and Campbell. If a node p, reached during the search, matches with the table's entry r then we may do the following :

- if the level of r is less than the level of p, then the search is directed to the best move determined by the previous search on the entry r,
- if the level of r is greater than or equal to the level

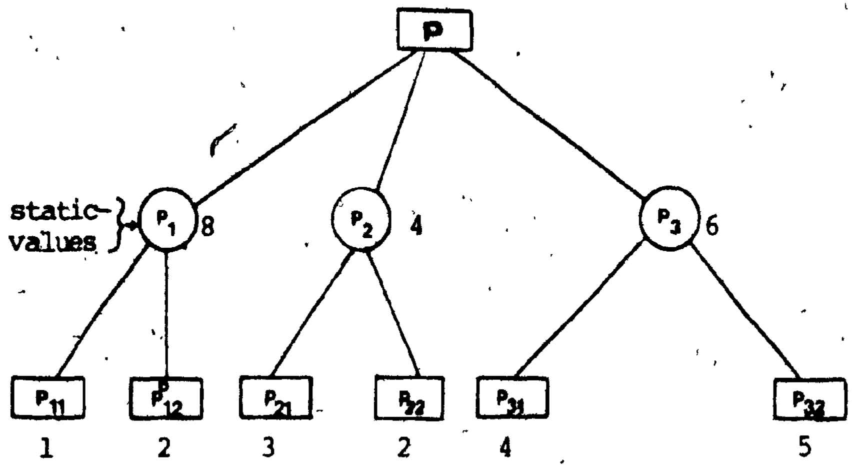


FIGURE 43.

Static-ordering versus dynamic-ordering.

In the static-ordering the search is based on the static-values obtained at level 1 of a tree. For this specimen tree subtree below node p_1 will be searched first, then subtree below node p_3, and at the end subtree below p_2 will be searched.
In the dynamic-ordering nodes at level 1 are ordered on the basis of their static-values. As in static-ordering subtree below p_1 will be searched first, next the subtree below p_3, then the subtree below p_2. But when node p_{11} has been evaluated, its static-value is compared to the value of node p_3 (as the second best on previous level). Because value of 1 is unpromising in comparison with 6, then the program abandons the search below node p_1, and the subtree below p_3 is searched.

Node p_{12} is searched if using static-ordering, but it may be cut off, when the dynamic-ordering is used.

of p, then if the search on r was completed we do not search p at all - value of p is equal to the value obtained for r, otherwise we adjust the current bounds for further search and we direct the search towards the best move found for r.

Transposition tables were described in [15] as being very effective in chess endgames. In [16] the 30% improvement in the number of leaf nodes created was reported when using transposition table for the game of chess with the depth of search equal to six.

4.4. The Killer Heuristic.

In [3] Akl and Newborn analyze the killer heuristic to supplement the Alphabeta algorithm. Any move which causes a cut-off at level L is said to have refuted the move at level L-1. The killer heuristic is based on the assumption that if move A 'refutes' move B then it is more likely that A will be also effective for the other positions. Program which uses a killer heuristic saves on a killer list moves that are refutations, and later it tries to match moves generated at any node with moves from the killer list. If a killer move is found then such a move is examined first. Moves saved while determining the principal continuation may serve as a killer list. Other advantage of the killer heuristic is that it increases the usefulness of the transposition table by continually suggesting the same moves. It may be also used for dynamic reordering. The actual improvements in the tree-search using the killer heuristic over the pure Alphabeta algorithm were analyzed in [3] on the King-pawn chess endgame program developed by Newborn. The presented results show that the percentage improvement of the number of leaf nodes created is oscillating about a fixed value for a given width of tree and different depths of the search. This percantage improvement varied from 15% to 80% depending on the starting position for the search, and on the width of the corresponding game tree.

CHAPTER 5.

PATHOLOGY IN GAME TREES.

Researchers proposed different methods of speeding-up the game tree search based on the assumption that one wishes to search deeper. There was almost universal agreement that increasing the depth of search improves the quality of decision made. Recent investigation by Beal [5], Bratko and Gams [8], Nau [18,19] and Pearl [25,27] showed that there exist a large class of the game trees for which searching deeper will not increase the probability of making the correct decision, and such game trees were called pathological [18]. These researchers analytically and empirically have shown that for this class of the game trees the decision made become random with deeper searching. This phenomenon is not observed in some real-world games, such as chess or checkers, where searching deeper improves the quality of decision. So a major open question has been why pathology occurs in some games and not in others. The review of theoretical analysis and of some experiments, carried by researchers [5,8,18,19,25,27] in order to investigate the causes of pathology will be presented. Then the possible methods of overcoming the pathological phenomenon will be discussed. We will also describe our experiments and we will report the results we have observed.

5.1. The Nature of Pathology.

To illustrate the nature of pathology in game trees, as described in [25], let us consider a uniform tree of width two and depth $d > 0$, where any leaf node may be WIN or LOSS with probability p and 1-p, respectively. It is easy to see that each node in such a game tree is either a forced win or a forced loss node. As we know a static evaluation function provides the estimates of the strength of any leaf node. We may assume that for any leaf node s in such a game tree staticvalue(s) is either 1, which should correspond to WIN, or 0, which should correspond to LOSS. However, this evaluation function might assign value of 1 to a LOSS leaf node, or it might assign value of 0 to a WIN node. The informedness of such a function may be quantified by two error parameters (P stands for probability) :

err1=P(staticvalue(s)=1 | s is LOSS),

err2=P(staticvalue(s)=0 | s is WIN).

For any node of a game tree we may consider its characteristic parameters to be the following probabilities :

1) probability that node is WIN ;

2) probability that the estimated value of the node is 1, and its true-value is LOSS ;

3) probability that the estimated value of the node is 0, and its true-value is WIN.

Any leaf node has an initial state of the characteristic parameters, which may be represented as a vector ($err1_d$, $err2_d$, p_d). For any node at level L-1, where $d \geqslant L > 0$, we may compute these parameters recursively as given in [8,25] :

$$err1_{L-1} = 1-(1-err2_L^2);$$

$$(1) \quad err2_{L-1} = [(1-p_L)err1_L + 2p_L(1-err2_L)]err1_L/(1+p_L);$$

$$p_{L-1} = 1-p_L^2.$$

These equations describe what happens to evaluations of the leaf nodes as these evaluations are backed-up the tree. The probability that at a certain node the decision made is correct depends on the accuracy of the minimax values of node's sons. Let us assume that we have a node with two sons : one corresponding to WIN, one to LOSS. It may happen that the WIN son obtains lower estimate than the LOSS son; then the wrong decision will be made. Probabilty of such a situation, denoted as ERR, is equal to 1/2(err1+err2). Researchers [5,6,8,18,19,25,27] have tested analytically and empirically the behaviour of the ERR with increasing depth of search. If one wants to analyze the benefit of increasing the depth of search by one, two, ... levels, then one has to compare the values of vector

$(err1_d, err2_d, p_d)$ obtained by passing through d iterations of (1) to that obtained by passing through d+1, d+2, ... iterations. Pearl [26] assuming that the static evaluation function is equally informed at all levels of the game, has concluded that the effect of increasing the search depth, equivalent to additional iterations of (1), always increases the value of ERR.

For uniform trees of width two and depth of search varing from 1 to 10, the subsequent values of ERR from (1) for different initial values of $err1_d$, $err2_d$, p_d have been computed. It was assumed that these initial values do not depend on the depth of search. Depending on the value of p_d, we got different values of the vectors $(err1_{d-1}, err2_{d-1})$, $(err1_{d-2}, err2_{d-2})$, ..., $(err1_0, err2_0)$, which results in different values of ERR. For some initial values of $err1_d$, $err2_d$ we have plotted in Figures 44 to 47 the changes in the values of ERR with subsequent iterations of (1), when the value of p_d was varing from 0.2 to 0.9. As we see the value of ERR migrates towards 0.5 with subsequent iterations of (1) for all presented p_d, $err1_d$, $err2_d$. If $(err1_d + err2_d) \geq 1$ then the value of ERR becomes greater than 0.5 which means that the evaluation function used is misleading. The information backed-up is free of error only if initialy err1=0 and err2=0. In such a case the evaluation function perfectly estimates values of the leaf nodes.

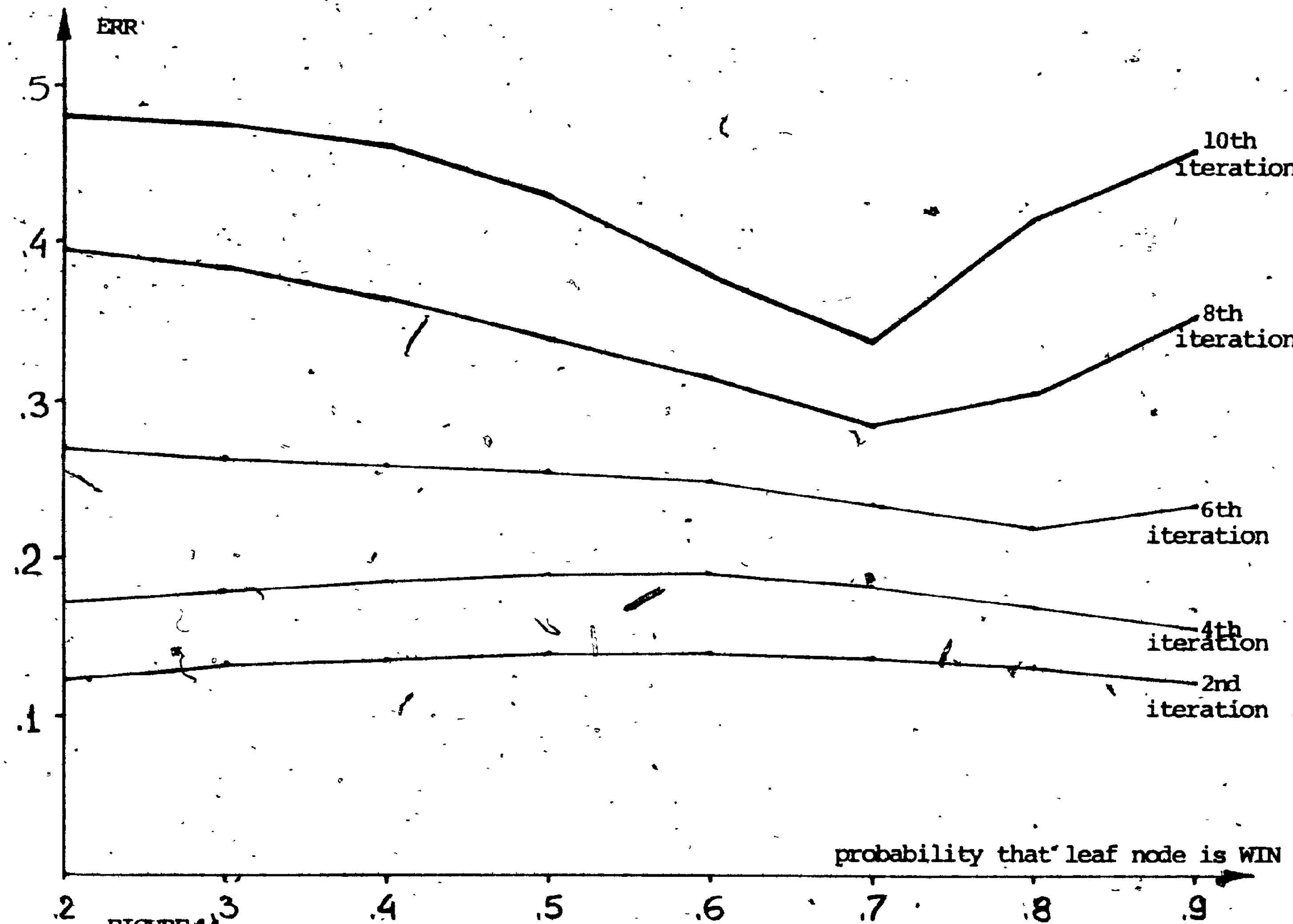


FIGURE 44

Plot of subsequent values of ERR against the probability that leaf node is WIN. Initial values of $err1_d$=0.1, $err2_d$=0.1. Game tree is uniform with width 2, depth 10.

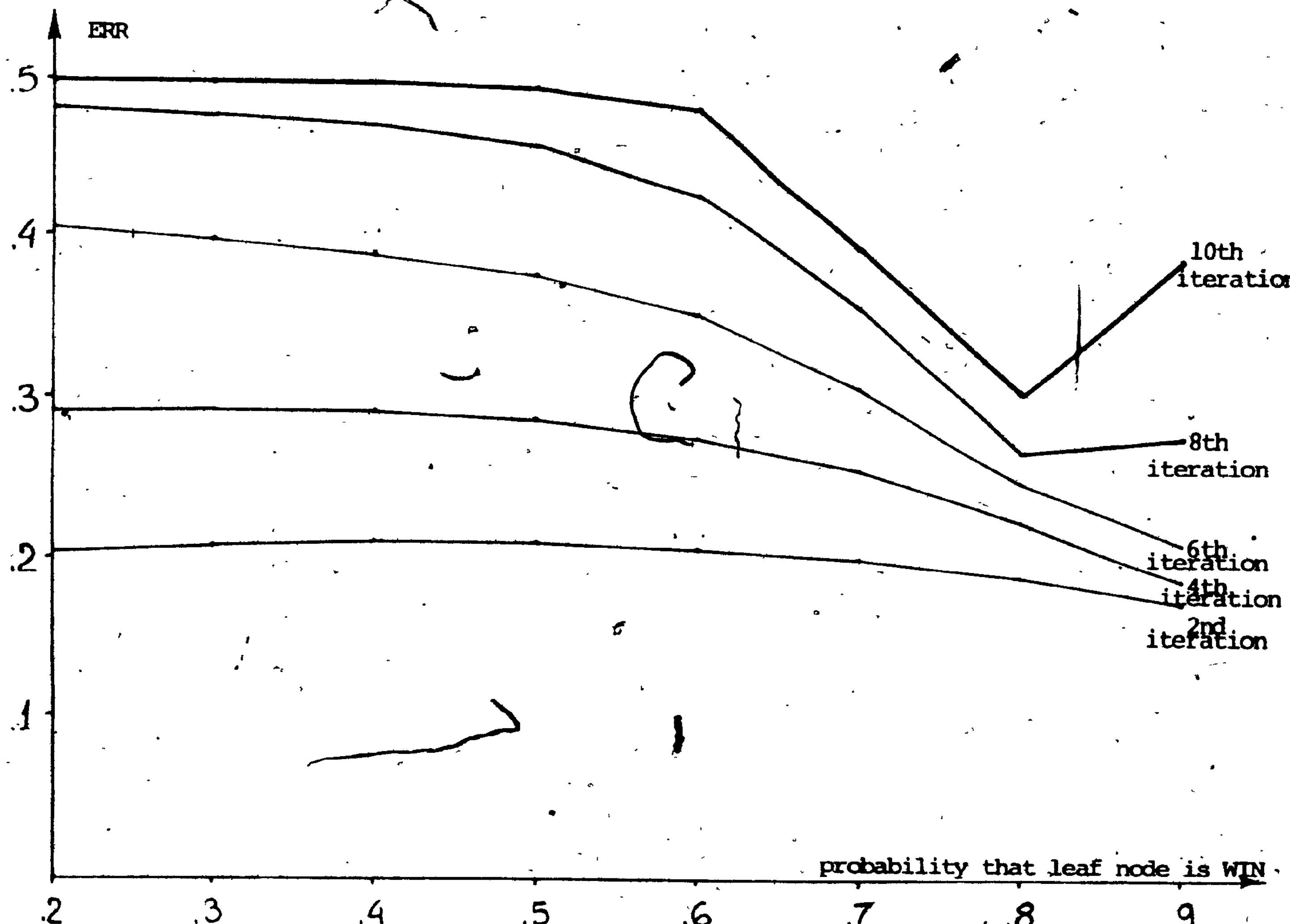


FIGURE 45

Plot of subsequent values of ERR against the probability that leaf node is WIN. Initial values of $err1_d=0.1$, $err2_d=0.2$. Game tree is uniform with width2, depth 10.

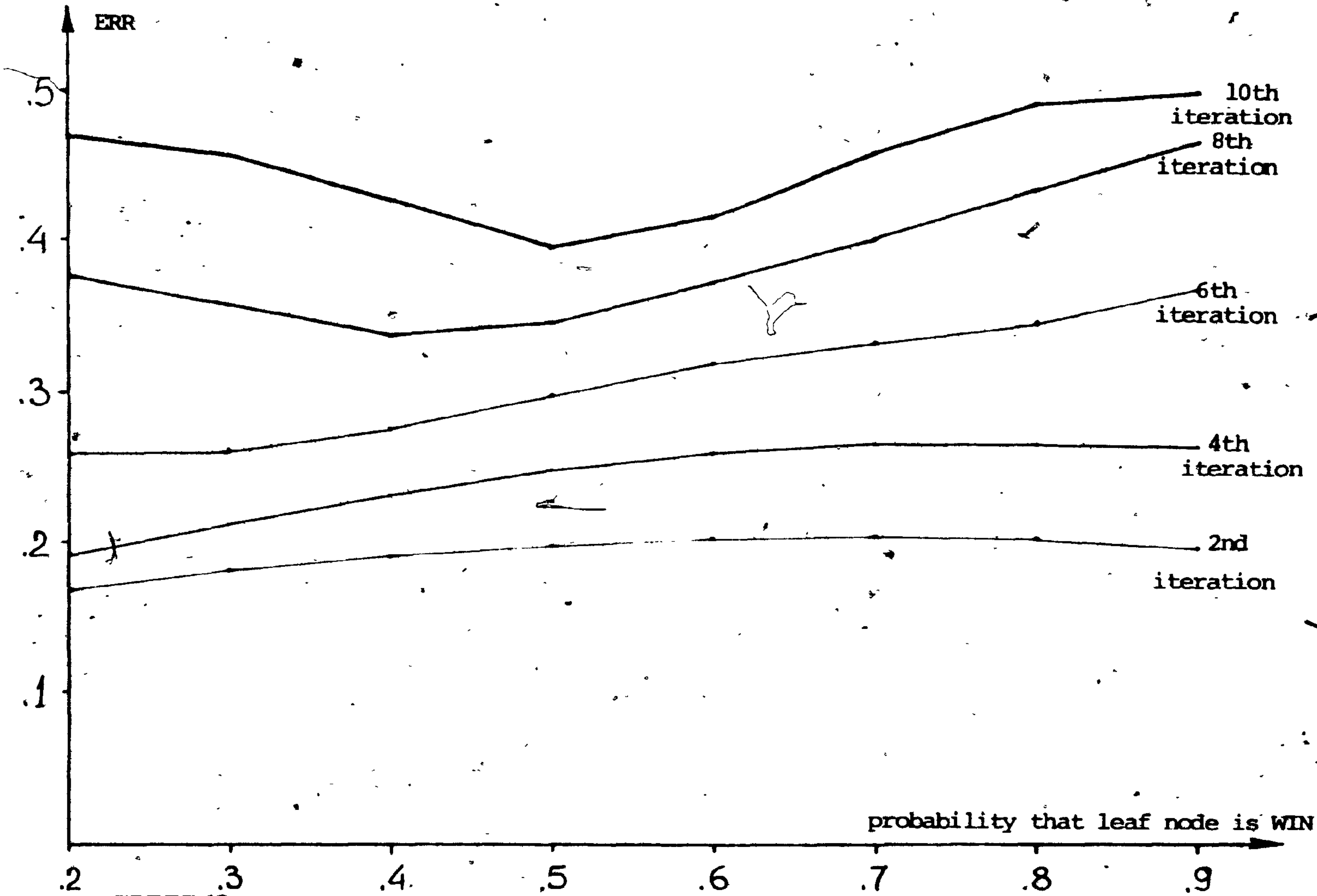


FIGURE 46

Plot of subsequent values of ERR against the probability that leaf node is WIN. Initial values of $err1_d$=0.2, $err2_d$=0.1. Game tree is uniform of width 2, depth 10.

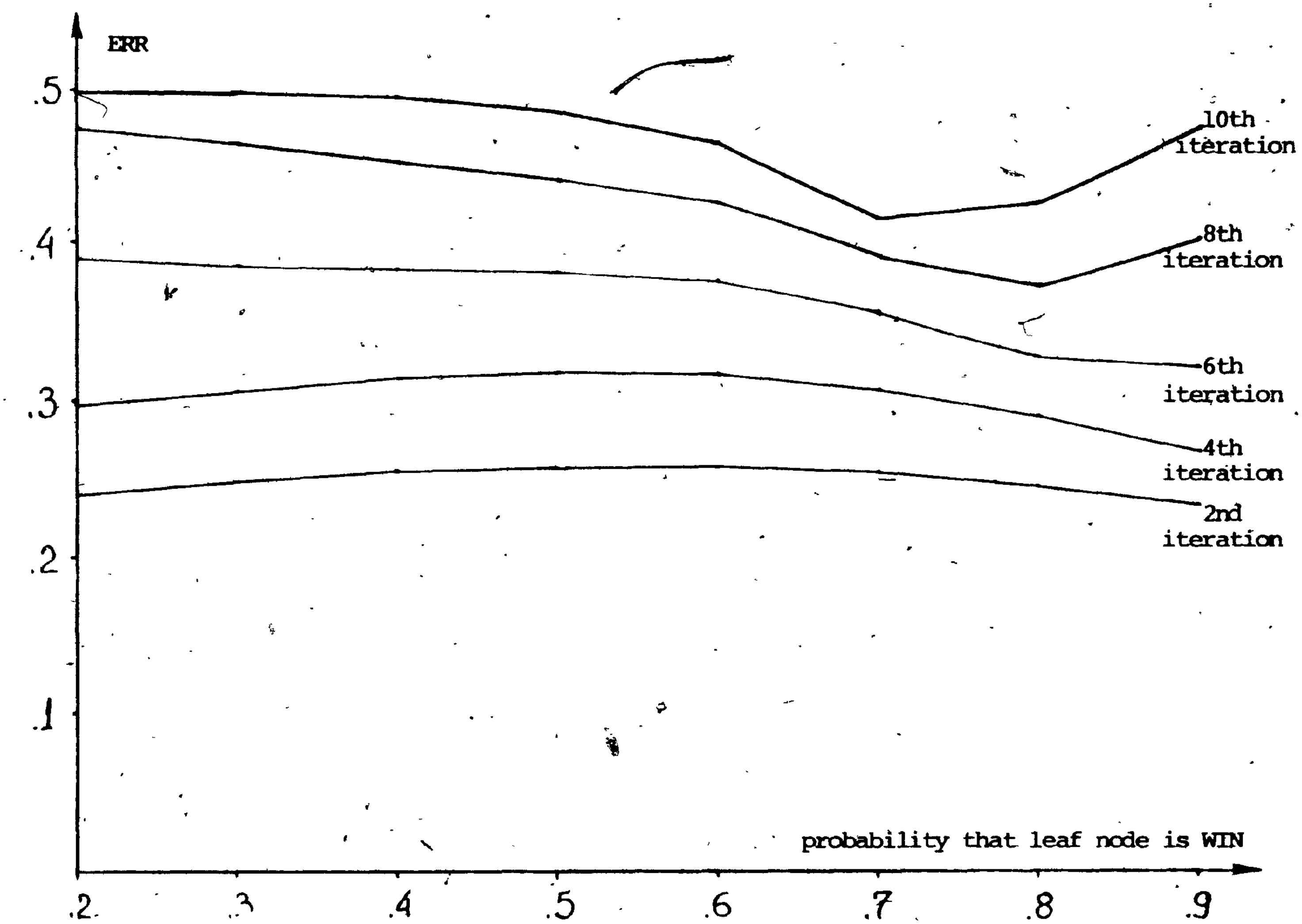


FIGURE 47

Plot of subsequent values of ERR against the probability that leaf node is WIN. Initial values of $err1_d=0.2$, $err2_d=0.2$. Game tree is uniform of width 2, depth 10.

Let us consider a more general case where an evaluation-function takes on multiple discrete or continuous values, as presented in [26]. The connection between magnitude of a static-value and the actual value of any node is characterized by the following pair of distribution functions :

$$F_L = P(\text{staticvalue}(s) \leq x \mid s \text{ is a LOSS node});$$
$$F_W = P(\text{staticvalue}(s) \leq x \mid s \text{ is a WIN node }).$$

For any fixed x, the events staticvalue(s) $\leq$x and staticvalue(s)>x propagate with the same logic as the staticvalue(s)=0 and staticvalue(s)=1 in the bi-valued model. So two functions may be defined :

$$\text{err1}(x)=1-F_L(x), \quad \text{err2}(x)=F_W(x).$$

Equations (1) hold also for this general case, because with each minimax operation the pair (err1(x),err2(x)) undergoes same transformations as (err1,err2) in (1). For the case when the evaluation function returns continuous random values, the probability of making the wrong decision amounts to the area below the curve h, such that err2=h(err1), and may be described as :

$$\text{(2)} \qquad \text{ERR}=\int_x \text{err2}(x)\,d(\text{err1}(x)) = \int_{\text{err1}=0}^{1} h(\text{err1})\,d(\text{err1})$$

Analyzing different initial values of (err1,err2,p) Pearl

[26] has concluded that the value of ERR in (2) will migrate towards 0.5 with icreasing depth of search. Only if $errl_d$ or $err2_d$ initialy has a value of 0 then, depending on value of p_d, the value of ERR may be 0.

If the evaluation function takes on multiple discrete values then the value of ERR will amount to the area enclosed by the polygon connecting points (errl,err2) as their values are obtained passing through iterations of (1). The value of ERR will also migrate towards 0.5 if initially errl $\neq$ 0 and err2 $\neq$ 0.

Similar analysis as above may be done for the uniform trees with width greater than two. For such trees equations (1) become :

$$errl_{L-1} = 1-(1-err2_L)^w;$$

$$err2_{L-1} = [1/(1-p_L)]^w\{[(1-p_L)errl_L + p_L(1-err2_L)]^w - [p_L(1-err2_L)]^w\};$$

$$p_{L-1} = 1-p_L^w.$$

For bi-valued evaluation function the value of ERR will be equal to :

ERR = errl(1-p) + err2*p , as given in [8].

For this case and for the case of multi-valued evaluation function analysis of the value of probability of making wrong decision will be similar to that for binary trees.

From this theoretical analysis it is clear that the whole class of uniform trees with random static-values is pathological, increasing the depth of search for such a game tree degrades the quality of a decision made. The first game proved by Nau [18] to be pathological, is the Pearl-game. The initial playing board for this game is constructed by randomly assigning to each square of the board values of 1 or 0, with probabilities p_d and $1-p_d$ respectively. Player1 divides the board in w, $w \geq 2$, parts vertically and chooses one part, discarding others. Player2 divides what is left horizontally in w parts and chooses one part. The play continues in this manner until only one square is left, this square represents the ending position of a game. Note, that a leaf node usually does not correspond to the terminal position of a game. If the square has a value of 1 then the player1 wins at this position, otherwise his opponent wins. In Figure 48 an example of a game tree which corresponds to the Pearl-game is presented. As originally described in [23] the class of Pearl-games is played on an initial board measuring $w^{d/2} * w^{d/2}$ squares. Since the value of ERR quantifies the probability of erroneous decision we may say that the value of 1-ERR quantifies the probability of making correct decision. Analytical and empirical results for the value of 1-ERR for the Pearl-game, presented by Nau [18,19], show

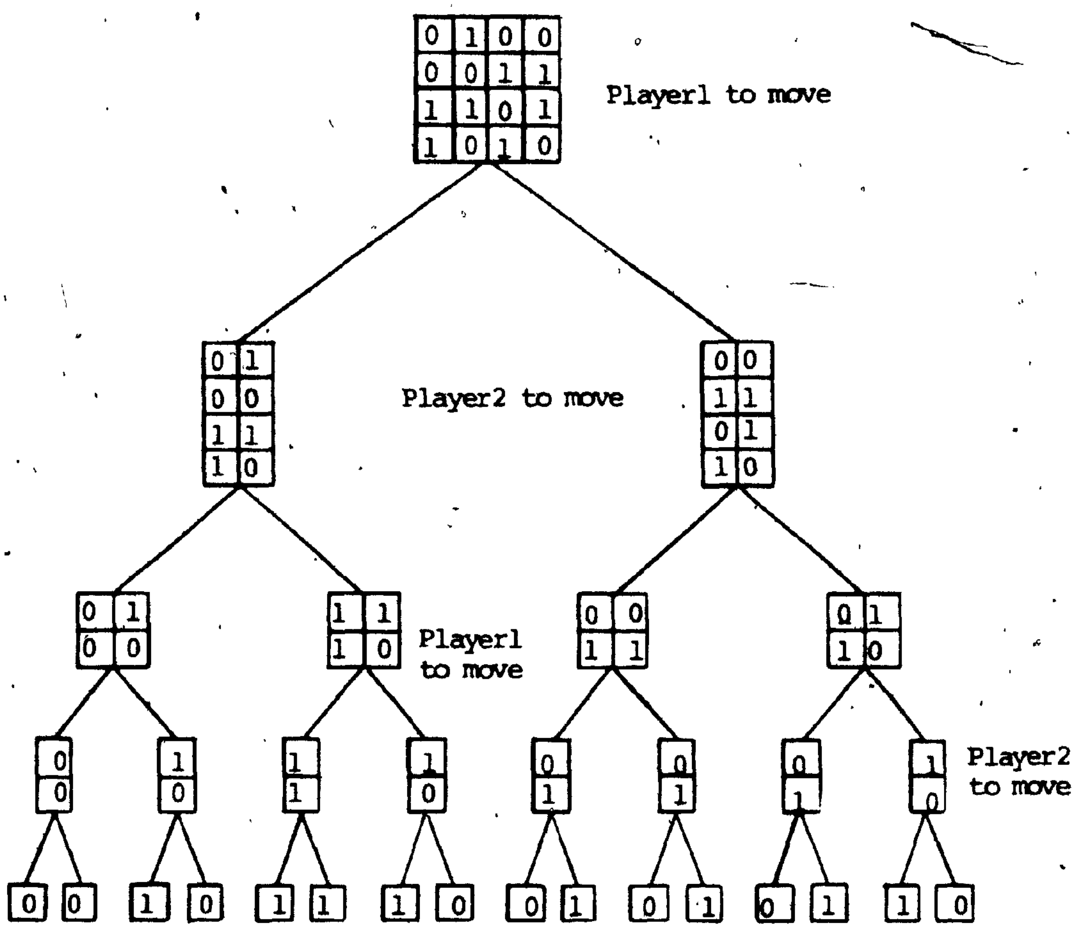


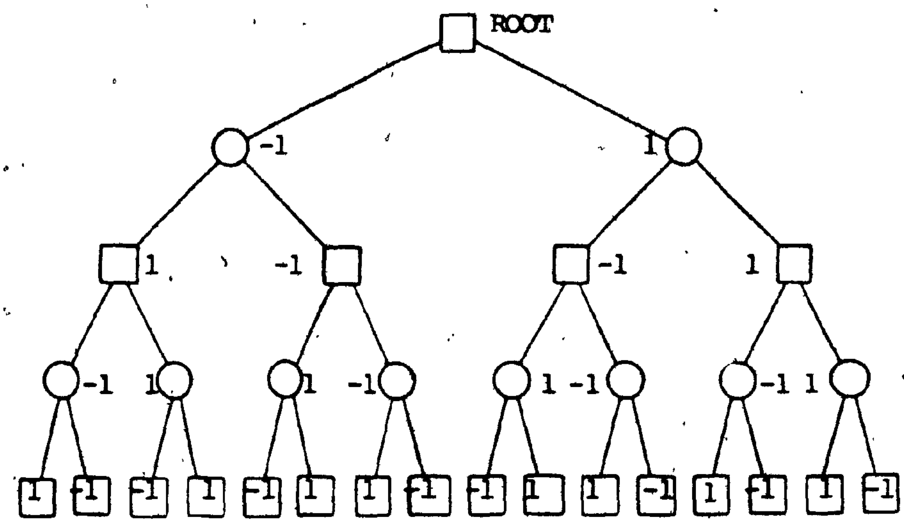
FIGURE 48

An uniform game tree of width two and depth four representing the Pearl-game.

that the probability of making the correct decision for binary trees tends to increase with an increasing depth of search d, but only for trees representing games with terminal positions up to the seventh level. As the level t of the terminal positions becomes greater, the probability of choosing the correct son of the root as the best son decreases with an increasing depth of search. Increasing the depth of search causes the decrease in the value of probability of correct decision even for smaller than seven depths of terminal positions for uniform trees of width greater than two. We simulated similar to Nau's experiments to prove that pathology occurs for the Pearl-games. We comment on the results obtained by Nau and by us in the section 5.3.

A question arises regarding causes of pathology. Nau [18,19] and Pearl [26] question why the Pearl-game is pathological and games such as chess or checkers are not. One of the possible reasons is that in games of chess or checkers the board positions change incrementally. Describing strength of a node as the possibility of winning for the player who moves from the corresponding game position, we may say that a strong node is likely to have strong sons and the values of the sibling nodes are likely to be similar. This property does not occur for the Pearl-game, where the values for any two nodes at the same level of a corresponding game tree are independent of each other as the functions of independent variables. In order to investigate games in which the strength of a node changes incrementally, a class of incremental games was defined by Nau in [18]. While the manner of moving, size of the board and criterion for winning are the same as in the Pearl-game, the assignment of static-values is done differently for incremental game. Each node is independently and randomly given the value 1 with probability p_a and value -1 with probability $1-p_a$. If the terminal node has a positive sum of itself plus the values of all its ancestors, then it is assigned a value of 1, otherwise it is assigned a value of 0. An example of an incremental game is given in Figure 49. Empirical results presented by Nau [18,19] have shown that pathology does not occur for such incremental games. We also simulated experiments to prove that pathology does not occur for incremental games. Discussion and comparison of Nau's and our results obtained for this experiment is presented in the section 5.3.

Bratko and Gams [8], assuming that the nodes of same true-value (WIN or LOSS), are grouped together, have shown that the error parameters err1 and err2 converge towards 0 if initially $err1_d < 1-\xi_w$, and $err2_d < \xi_w$. Beal [6] has shown that the assumption of grouping holds for the King-pawn chess endgame. So we may conjecture that the minimaxing is beneficial for the real-world games because of incremental changes in position values for such games.



SUM : 0 -2 0 2 -2 0 -2 -4 0 2 0 -2 2 0 4 2

FIGURE 49.

Construction of a game tree corresponding to the incremental game of width 2 and depth 4. Every node in a tree is assigned value of 1 with probability p_a, or value of -1 with probability $1-p_a$. For this case p_a was equal to $\frac{1}{2}$. If for a leaf node value of itself plus sum of values of all its ancestors is >0 then such a leaf node has a static-value of 1, otherwise 0.

0	0	0	0
0	1	1	0
0	0	1	1
0	0	0	1

For this specimen incremental game, such an initial playing board appears at the root of the tree.

A real-world game may terminate at any level. The more realistic model representing a game should capture this property. For such a model the leaf nodes may be present at any level of a game tree (if a leaf node is at a level higher than the search depth, then it is the terminal node - node which corresponds to the end of a game). Assuming that each node has a non-zero probability q of being a terminal, and that the evaluation function identifies the terminal nodes without error, we obtain modified version of (1), as given in [26] :

$$
\begin{aligned}
& err1_{L-1} = [1-(1-err2_L^2)][(1-q)p_L^2]/[1-(1-q)(1-p_L^2)], \\
(3) \quad & err2_{L-1} = [(1-p_L)err1 + 2p_L(1-err2_L)]\ err1_L/(1+p_L), \\
& p_{L-1} = (1-q)(1-p_L^2)
\end{aligned}
$$

Analyzing the trajectory of the vector (err1(x),err2(x)) Pearl [26] concludes that the presence of terminal nodes in game trees, even at a low density of 5%, completely eliminates the search-depth pathology for trees where $p_d \neq (\xi - q\xi)/(1-2\xi)$, ξ was defined in section 3.4. For this initial p_d the $(err1_0, err2_0)$ migrates towards the points (0, 0.8089) and (0.842, 0).

For uniform trees of width two and depth of search varying from 1 to 10 we have computed values of ERR for diferent initial values of $err1_d$, $err2_d$, p_d and different values of q. Values of $err1_d$ and $err2_d$ varied from 0.5 to 0.1, of p_d from 0.9 to 0.2 and value of q varied from 0.25 to 0.05. We have noticed that for every tested value of ($err1_d$, $err2_d$, p_d) the value of ERR was decreasing with subsequent iterations of (3). Only for initial p_d=0.6 the value of ERR increases with the subsequent iterations of (3). This was as predicted by Pearl's analysis [25,27]. In Figures 50 to 53 we have plotted the changes in the value of ERR for trees for which q=0.05, for same initial values of $err1_d$, $err2_d$ as plotted for uniform trees, with value of p_d varying from 0.2 to 0.9. As we see the results agree with the Pearl's conclusions. We have noticed that the decrease of the value of ERR with subsequent iterations of (3) is quicker with the smaller initial values of $err1_d$ and $err2_d$.

5.2. Possible Methods of Avoiding Pathology.

As we have seen pathology may disapper if the game tree is nonuniform. Such property is common for real-world games. For games such as Pearl-game, which have a uniform structure, Pearl [25,26,27] has suggested that pathology might be avoided if the evaluation function used would return the probability that a leaf node is forced win, and if for backing-up we replace the minimaxing rule by a

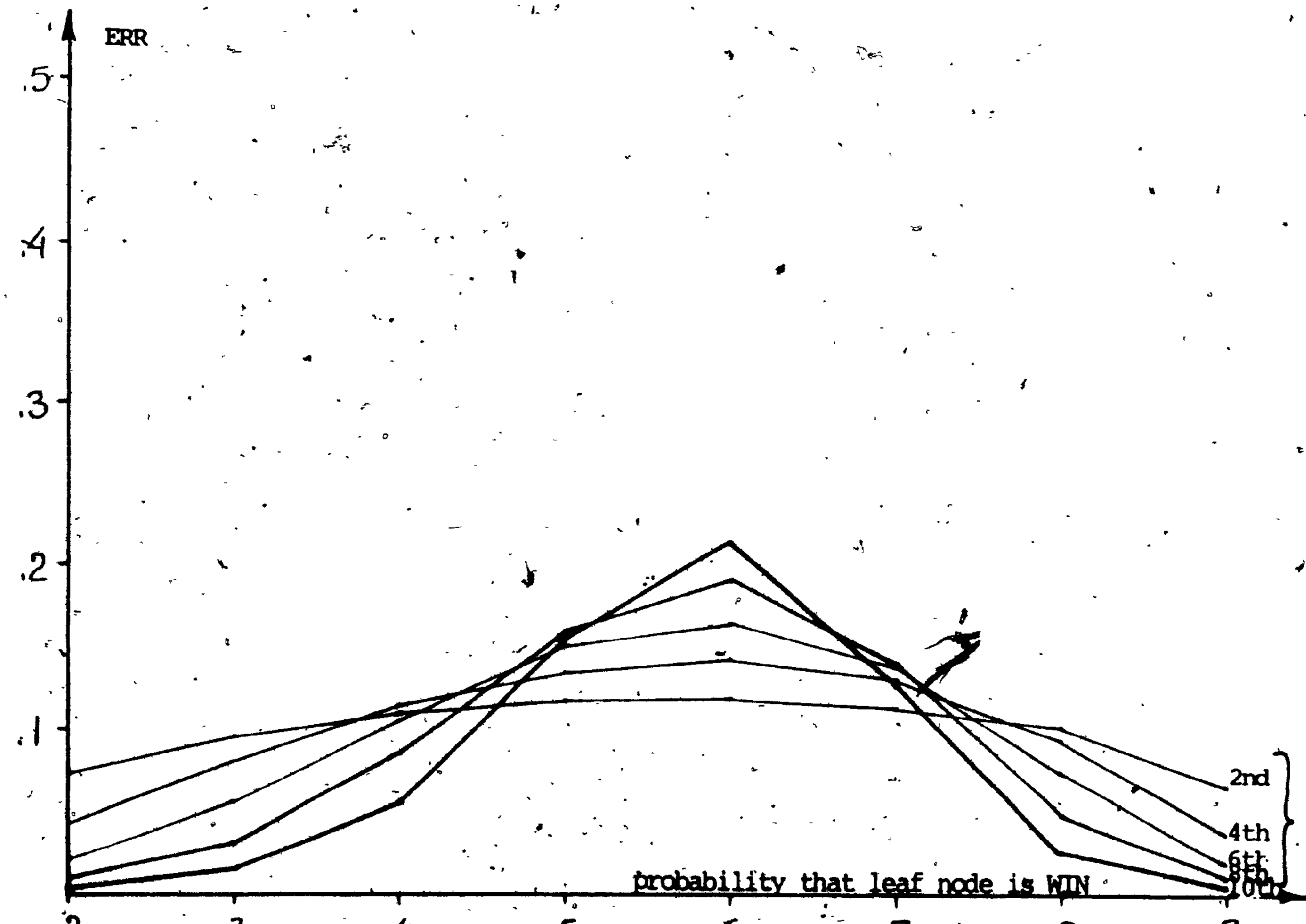


FIGURE 50

Plot of subsequent values of ERR against the probability that leaf node is WIN. Initial values of $err1_d=0.1$, $err2_d=0.1$. Game tree is nonuniform with width 2, depth 10.

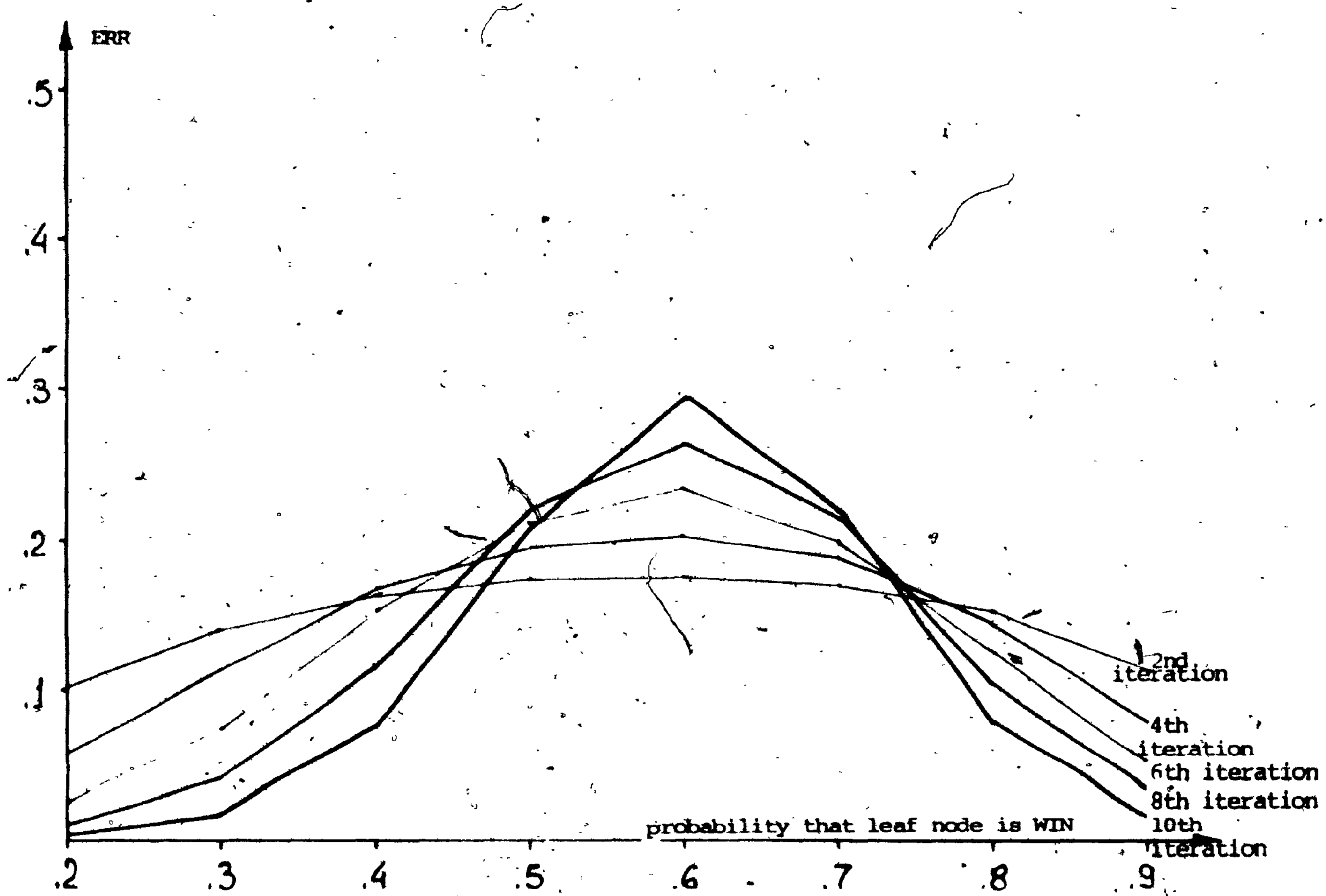


FIGURE 51

Plot of subsequent values of ERR against the probability that leaf node is WIN. Initial values of $err1_d = 0.1$, $err2_d = 0.2$. Game tree is nonuniform with width 2, depth 10.

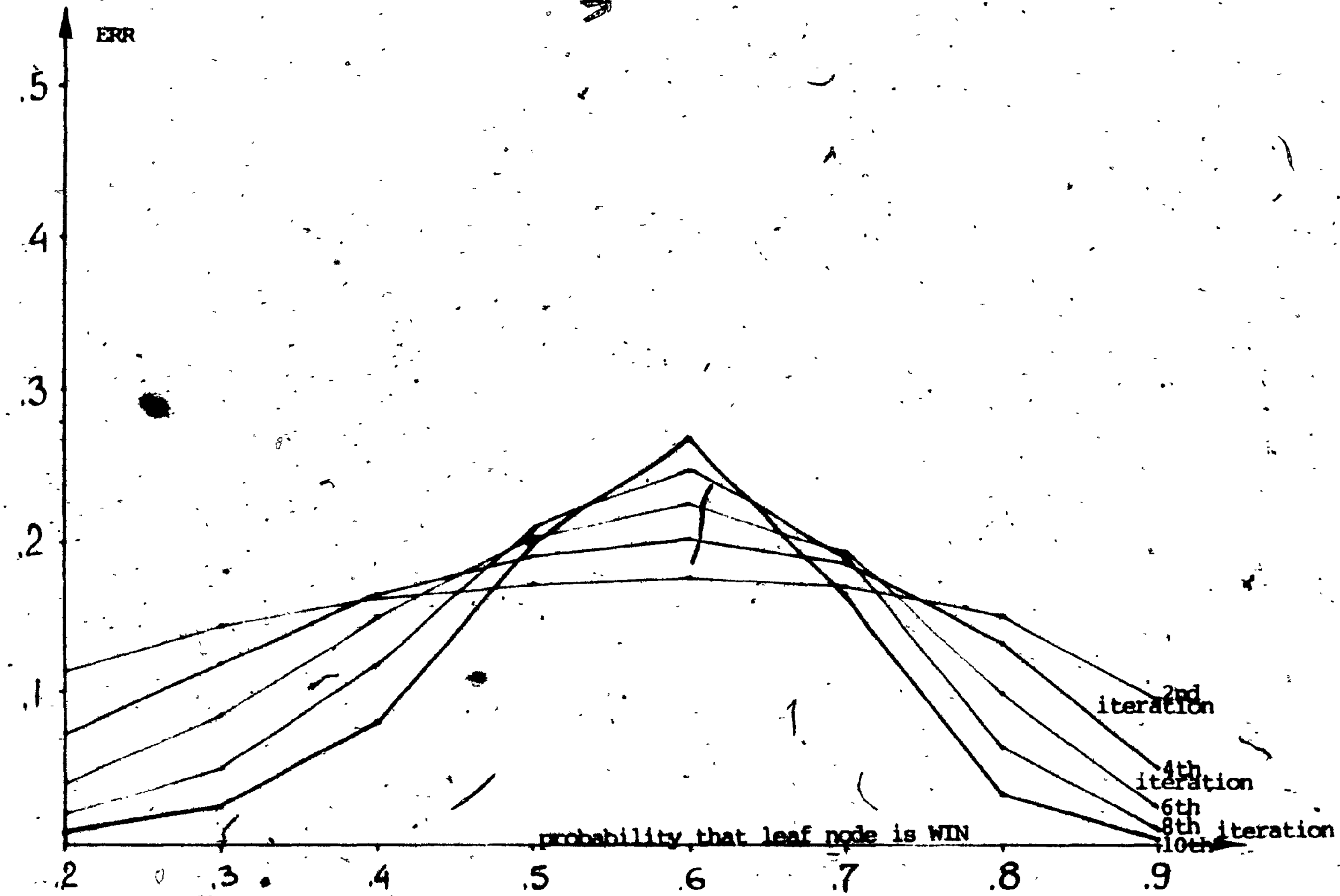


FIGURE 52

Plot of subsequent values of ERR against the probability that leaf node is WIN. Initial values of $err1_d=0.2$, $err2_d=0.1$. Game tree is nonuniform with width 2, depth 10.

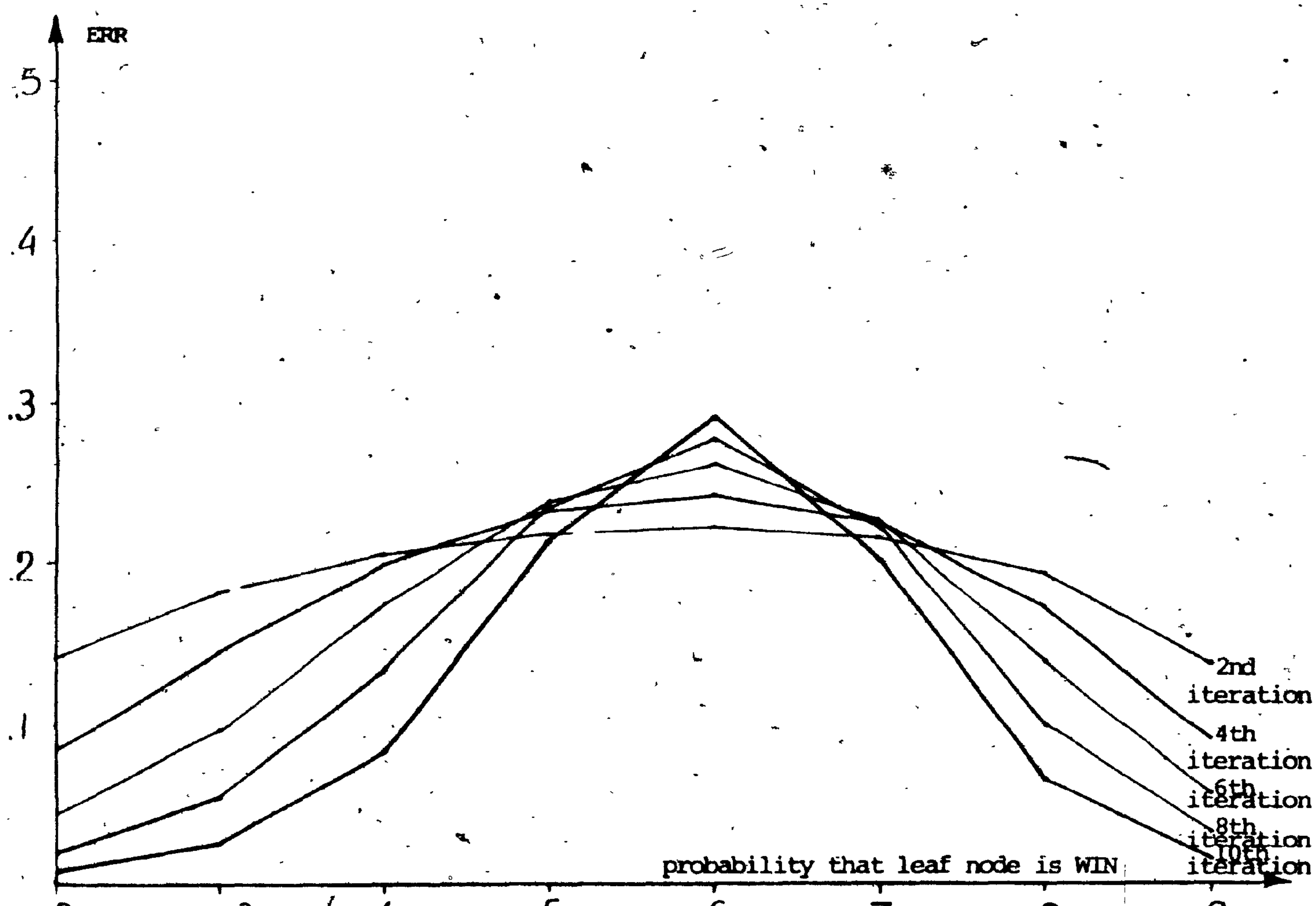


FIGURE 53

Plot of subsequent values of ERR against the probability that leaf node is WIN. Initial values of $err1_d=0.2$, $err2_d=0.2$. Game tree is nonuniform with width 2, depth 10.

product propagation rule. Using this method of searching the game trees, we may say that for every node we compute its winability. The probability estimation function, denoted as probest, assigns a winability values to the leaf nodes. Value obtained for player1 at a node r, denoted as winability(r), is defined as followed :

if r is a MIN node :

$$\text{winability}(r)=\begin{cases}\text{probest}(r), & \text{if } r \text{ is a leaf node ;}\\ \prod_{i=1}^{t}\text{winability}(r_i), & \text{otherwise; } r_i \text{ is son of } r.\end{cases}$$

if r is a MAX node :

$$\text{winability}(r)=\begin{cases}\text{probest}(r), & \text{if } r \text{ is a leaf node;}\\ 1-\prod_{i=1}^{t}(1-\text{winability}(r_i)), & \text{otherwise.}\end{cases}$$

Such a method of searching game trees has a few disadvantages. First is how to estimate the probability that a position is a forced win ? Very likely such an estimation function will be just a mapping of a static evaluation function on the interval [0,1]. Secondly, it is difficult to invent a good pruning algorithm for such a method of backing-up. And, as we well know, it is not practical to perform an exhaustive search of a game tree. Third is that for some instances minimaxing differs from product propagation in predicting which move shall be chosen. Assuming that at a root player1 chooses a move towards the node which returns the highest winability, then for example, for the tree shown in Figure 54 move towards p_2 will be chosen by minimaxing and move towards p_1 will be chosen by a product propagation rule. Which one is correct ?

The first investigation of the probability estimation in conjunction with the product propagation was done by Nau [18,19]. As an estimation of the probability that a leaf node is a forced win Nau used a function which returns the ratio of the number of terminal WIN nodes to the total number of terminal nodes corresponding to this leaf node. Such a function is in fact a mapping of the previously used static evaluation function into the interval (0,1). Nau's empirical results show that for the Pearl-game pathology does not occur when the product propagation is used as the rule for backing-up. Nau's and our experiments performed for such a strategy of game-playing are described in the section 5.3.

It is also possible to improve the decision quality by employing more appropriate backing-up rules [27]. One such rule is represented in Berliner's [7] B* algorithm, where each node is quantified by an optimistic and pessimistic estimate of its strength. These estimates provide a range on the values of the node's successors. We may also say that this range delimits the uncertainty in the evaluation.

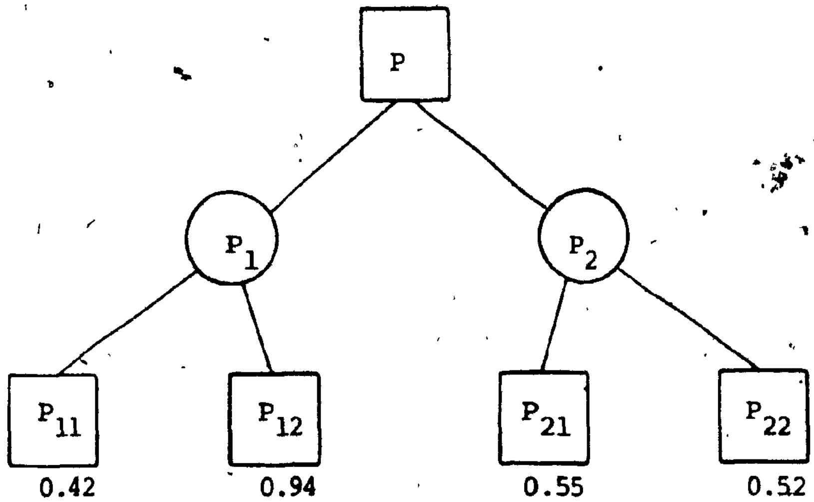


FIGURE 54

An example to show that minimaxing differs from the product propagation in choosing the move.
Assuming that the static-values are equivalent, then for :

- minimaxing :
  a) 0.42 will be backed-up to node p_1
  b) 0.52 will be backed-up to node p_2

  Move towards p_2 will be chosen.

- product propagation :
  a) 0.42*0.94 = 0.3948 will be computed for node p_1
  b) 0.55*0.52 = 0.2860 will be computed for node p_2

  Move towards p_1 will be chosen, as one having the highest probability of being a WIN.

The B* proves that the true value of one of the root's sons is $\geq$ the values of the other sons. This is accomplished by showing that the pessimistic value of one of the sons of the root is $\geq$ ~~the~~ optimistic values of the rest of the root's sons. The initialization of the B* algorithm is shown in Figure 55.

The B* algorithm first expands the root. It has to decide which node to explore and also if it wants to raise the pessimistic value of the current best son above the optimistic values of the other sons, or would it be easier to lower the optimistic value of remaining sons to below pessimistic value of the current best son. The rules for making decisions throught the B* search are based on a simple probabilistic model, as discussed in [22] by Palay. These rules were tested in [22] on simulated game trees and they were shown to provide a saving of 65% of the work required by the exhaustive search. Further studies on B* are needed to conjecture if this strategy is effective for playing the real world-games.

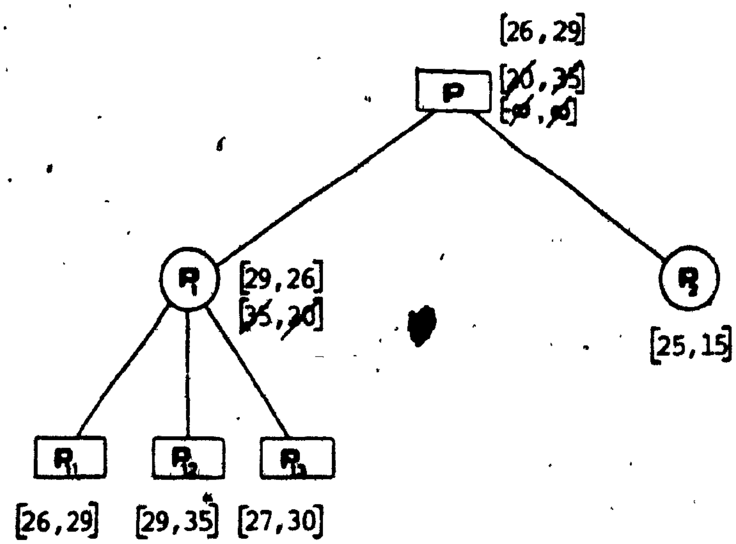


FIGURE 55.

The B* algorithm.

The optimistic and pessimistic values associated with any node are shown in brackets, the optimistic value being the leftmost of the pair. These values are updated as the search progresses.

For this game tree the algorithm tries to raise the lower bound of the node p_1, as the most optimistic, to the value not worse than the upper bound of node p_2.

5.3. Experiments Simulated on Pathological and Nonpathological Game Trees.

To test the changes in the value of the ERR for the Pearl-game, Nau [18,19] has simulated uniform game trees with a certain terminal level t. Such trees will be denoted as U(w,d,t), where d stands for the search-depth and w stands for the width of tree, $d \leq t$. The terminal level of tree represents the end of a game. Nodes at this level were assigned values of 1, corresponding to WIN, or 0, corresponding to LOSS, with probabilities p_d and $1-p_d$, respectively. These true-values were then backed-up to the root, and the forced win son and forced loss son of the root were chosen. Trees, for which the root does not have a forced win son, were disregarded. Then, the nodes at levels t-1, t-2, ..., 1 were considered to be the leaf nodes. The function, which for a certain leaf node, returns the number of corresponding terminal nodes with value 1, was used as an evaluation function. Note, that such an evaluation function is not very accurate on the levels greater than or equal to the level t-2. As an example in Figure 56 we present two cases for which the estimated value of a leaf node is the same, but for the one case the true-value of the node is WIN, and for the other case LOSS. The evaluations of the leaf nodes were then backed-up to the root; the Nau's method of backing-up [19], thought different in notation, is equivalent to negamaxing (or minimaxing). As an

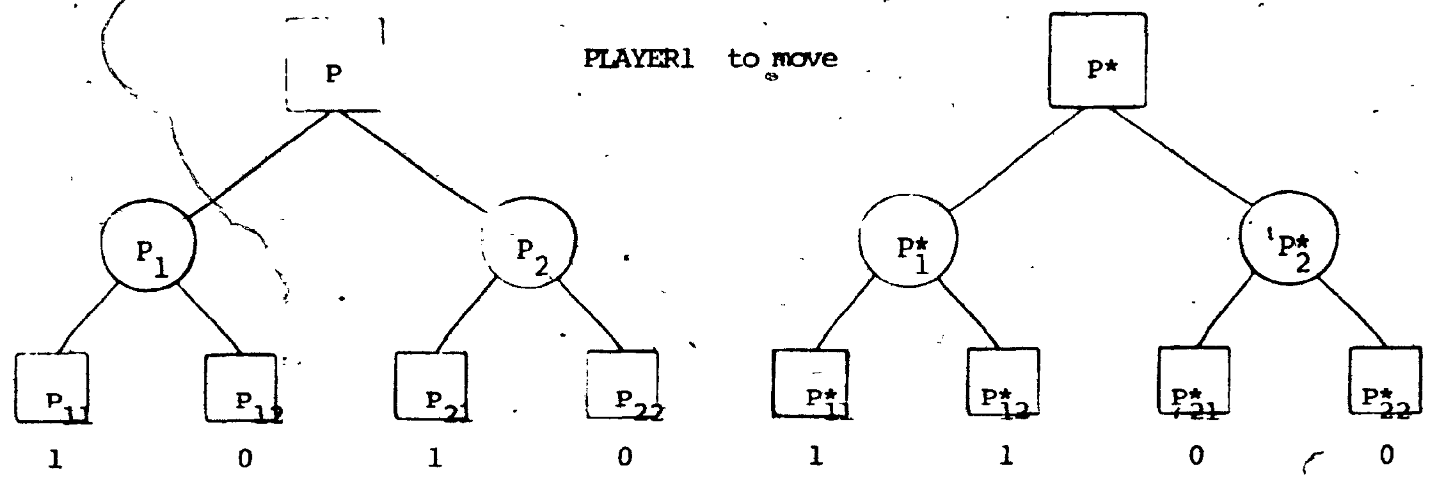


FIGURE 56.

An example to show that the evaluation function used by Nau [19] to prove the existance of pathology is not accurate for the certain situations.

Nodes p and p* have the same estimated value of 2, but the player1 has forced win at node p*, and forced loss at node p.

approximation of the 1-ERR, which quantifies the probability of the correct decision, Nau [18,19] uses the ratio of trees for which the move towards the forced win son was chosen. Results obtained by Nau are presented in Table XIX. We have simulated very similar experiments but we used minimaxing for backing-up. We have also chosen the forced win son and forced loss son of the root node, and than we have compared values returned to those sons with the increasing depth of search. We have simulated this experiment on 20000 trees. Our results are presented in Table XX. They are almost identical with the Nau's results. The few minimal differences may be due to the different sample sizes used by us and by Nau. As we see with increasing width of a tree the pathology occurs for smaller depths of search. This was as expected based on the theorem stated by Nau in [19].

Similar experiments were performed for the incremental games. Nau has simulated different U(w,d,t), where nodes at the terminal level were assigned values of 1 or -1 in a way described in section 5.1. Then the forced win and forced loss sons were chosen. The values returned to these sons were compared with an increasing depth of search. The function which returns the number of corresponding terminal nodes with value 1 was used as the static evaluation function. Nau's results are presented in Table XXI. Our results for this experiment are presented in Table XXII. Both tables show that for every tested U(w,d,t) the value of

Terminal level of the game	Depth of search	Width of tree 2	3	4	5	6
5	1	0.842	0.760	0.711	0.692	0.675
	2	0.867	0.779	0.701	0.671	0.649
	3	0.891	0.777	0.708	0.664	0.638
	4	1.000	1.000	1.000	1.000	1.000
	5	1.000	1.000	1.000	1.000	1.000
6	1	0.809	0.702	0.656	0.637	0.619
	2	0.806	0.699	0.641	0.612	0.602
	3	0.825	0.713	0.632	0.602	0.592
	4	0.833	0.692	0.619	0.577	0.565
	5	1.000	1.000	1.000	1.000	1.000
	6	1.000	1.000	1.000	1.000	1.000
7	1	0.762	0.666	0.613	0.594	not simulated
	2	0.769	0.655	0.602	0.566	
	3	0.777	0.653	0.606	0.559	
	4	0.797	0.640	0.586	0.555	
	5	0.811	0.624	0.569	0.538	
	6	1.000	1.000	1.000	1.000	
	7	1.000	1.000	1.000	1.000	

TABLE XIX.

Nau's results [19] for the approximation of the probability of correct decision for the uniform trees representing the Pearl-game. In the Nau's method of backing-up the value of a node p is computed as follows :

$$\text{BACKVAL}(p) = \begin{cases} \text{staticvalue}(p) & \text{if } p \text{ is a leaf,} \\ -\min(\text{BACKVAL}(p_1), \ldots, \text{BACKVAL}(p_f)) & \text{otherwise} \end{cases}$$

and the static-value is assigned to the leaf node from the point of view of the player who makes the last move.

Terminal level of the game	Depth of search	Width of tree				
		2	3	4	5	6
5	1	0.851	0,741	0.701	0.672	0.663
	2	0.878	0.750	0.692	0.652	0.604
	3	0.891	0.761	0.734	0.653	0.675
	4	1.000	1.000	1.000	1.000	1.000
	5	1.000	1.000	1.000	1.000	1.000
6	1	0.815	0.701	not simulated *		
	2	0.803	0.698			
	3	0.851	0.702			
	4	0.833	0.675			
	5	1.000	1.000			
	6	1.000	1.000			
7	1	0.751	0.671	not simulated		
	2	0.720	0.647			
	3	0.737	0.663			
	4	0,757	0.662			
	5	0.798	0.638			
	6	1,000	1.000			
	7	1.000	1.000			

TABLE XX.

Results from duplication of Nau's experiments. The approximation of probability of correct decision for uniform trees representing the Pearl-game. Minimaxing was used for backing-up.

* these cases were not simulated by us because the trees became to big to fit within the computer memeory available.

Terminal level of the game	Depth of search	Width of tree				
		2	3	4	5	6
5	1	0.941	0.936	0.959	0.968	0.985
	2	0.969	0.967	0.976	0.987	0.997
	3	0.982	0.976	0.987	0.994	0.998
	4	1.000	1.000	1.000	1.000	1.000
	5	1.000	1.000	1.000	1.000	1.000
6	1	0.936	0.941	0.966	0.972	0.985
	2	0.953	0.961	0.978	0.990	0.996
	3	0.976	0.978	0.992	0.996	0.999
	4	0.987	0.983	0.992	0.999	0.999
	5	1.000	1.000	1.000	1.000	1.000
	6	1.000	1.000	1.000	1.000	1.000
7	1	0.924	0.936	0.948	0.969	not simulated
	2	0.955	0.960	0.974	0.992	
	3	0.964	0.964	0.977	0.992	
	4	0.980	0.982	0.988	0.996	
	5	0.985	0.985	0.994	0.998	
	6	1.000	1.000	1.000	1.000	
	7	1.000	1.000	1.000	1.000	

TABLE XXI.

Nau's results [19] for approximation of the pobability of correct decision for uniform trees representing the incremental games. Nau's method of backing-up was used.

Terminal level of the game	Depth of search	Width of tree				
		2	3	4	5	6
5	1	0.960	0.912	0.908	0.916	0.924
	2	0.972	0.923	0.913	0.929	0.937
	3	0.976	0.954	0.927	0.941	0.949
	4	1.000	1.000	1.000	1.000	1.000
	5	1.000	1.000	1.000	1.000	1.000
6	1	0.929	0.930	not simulated *		
	2	0.945	0.942			
	3	0.957	0.959			
	4	0.971	0.979			
	5	1.000	1.000			
	6	1.000	1.000			
7	1	0.931	0.926	not simulated		
	2	0.947	0.931			
	3	0.956	0.937			
	4	0.964	0.959			
	5	0.972	0.984			
	6	1.000	1.000			
	7	1.000	1.000			

TABLE XXII.

Results from duplication of Nau's experiments. The approximation of probability of correct decision for uniform game trees representing the incremental games. Minimaxing was used for backing-up.

* these cases were not simulated by us because the trees became to big to fit within the computer memory available.

estimation of the probability of correct decision increases with increasing depth of search, so there is no pathology for incremental games.

We have also simulated different experiments to examine if pathology occurs for uniform trees with dependent static-values assignment. We simulated different U(w,d,t) trees and we assigned initial values to all nodes in every tree as in dependent scheme described in section 3.3. So for the integer-dependent approach sibling nodes were assigned distinct values from the set $\Theta = \{1, 2, \ldots, f\}$. For the real-dependent approach we used set $\Theta = \{1/f^L, 2/f^L, \ldots, f/f^L\}$, where L is the level of the nodes to which the values were being assigned. For both approaches value of a terminal node was computed as the sum of its assigned value plus the summation of the values of all its ancestors. These computed values were then backed-up to the root of a game tree. Comparing the values obtained for the sons of the root we created a set of best sons. Such set may consist only of one son, if only one node at the first level of a game tree returns the highest value. Trees for which all sons of the root returned the same value were disregarded. Then we performed the minimax searching up to the depth d = t-1, t-2, ..., 1. The value for any leaf node was computed using already assigned initial values. So the static-value was computed as sum of the initial value of a node plus the summation of the values of all node's ancestors. These static-values were backed-up to the root of a tree using the minimax procedure. Then we have compared the highest value obtained for the nodes in the set of best sons, to the highest value obtained for the rest of the root's sons. For every depth of search d we have computed the ratio of trees for which a correct decision was made, in a sense that a son from the set of best sons had obtained the highest value. Our results for the integer-dependent and real-dependent schemes are presented in Tables XXIII and XXIV respectively. For both schemes usually the value of estimation of the probability of correct decision is increasing with increasing depth of search. For some instances, however, the pathology occurs. For example for U(3,2,6) in Table XXIII, the proportion of trees for which a node from the best sons set obtained the highest value is 0.662 but for U(3,3,6) the proportion is 0.654. For dependent schemes of assigning static-values the value of a node does not vary substantially from the value of its father, but sometimes a bigger difference may happen, so for such instances pathology may occur.

Investigating the alternatives to minimaxing Nau [19] has tested the product propagation rule for backing-up when searching the game trees. Assuming that the evaluation function assigns to leaf nodes the probabilities that a node is a win for a certain player, for uniform trees representing the Pearl-games and incremental games, Nau [19]

Terminal level of the game	Depth of search	Width of tree			
		2	3	4	5
4	1	0.812	0.801	0.794	0.721
	2	0.834	0.815	0.799	0.742
	3	0.891	0.819	0.805	0.766
	4	1.000	1.000	1.000	1.000
5	1	0.724	0.717	0.690	not simulated *
	2	0.729	0.732	0.692	
	3	0.762	0.754	0.691	
	4	0.779	0.760	0.699	
	5	1.000	1.000	1.000	
6	1	0.633	0.651	not simulated	
	2	0.641	0.662		
	3	0.684	0.654		
	4	0.709	0.673		
	5	0.712	0.699		
	6	1.000	1.000		

TABLE XXIII.

The approximation of the probability of correct decision for the uniform trees with integer-dependent static-values assignment. Minimaxing was used for backing-up.

* these cases were not simulated by us because the trees became to big to fit within the computer memory available.

Terminal level of the game	Depth of search	Width of tree 2	3	4	5
4	1	0.726	0.697	0.656	0.612
	2	0.738	0.699	0.662	0.612
	3	0.749	0.708	0.687	0.624
	4	1.000	1.000	1.000	1.000
5	1	0.659	0.641	0.612	not simulated *
	2	0.662	0.657	0.624	
	3	0.678	0.649	0.636	
	4	0.691	0.661	0.639	
	5	1.000	1.000	1.000	
6	1	0.610	0.605	not simulated	
	2	0.615	0.612		
	3	0.614	0.624		
	4	0.624	0.629		
	5	0.635	0.629		
	6	1.000	1.000		

TABLE XXIV.

The approximation of the probabilty of correct decision for the uniform trees with real-dependent static-values assignment. Minimaxing was used for backing-up.

* these cases were not simulated by us because the trees became to big to fit within the computer memory available.

performed the same experiments as were performed with minimaxing. Nau's results for the approximation of probability of correct decision for the Pear-game are presented in Table XXV, and results for the incremental games are presented in Table XXVI. Results from Table XXV show that pathology does not occur for any tested value of search-depth in the Pearl-games when the product propagation was used for backing-up. On incremental games the probability of a correct decision was slightly lower for product propagation than for minimaxing. Further Monte-Carlo studies indicate that product propagation performs only marginally better than minimaxing in terms of the number of Pearl-games which were won against the minimax search to the same depth. A possible reason for this disappointing performance of product propagation is that the evaluation functions used were only approximations of the probability that a node is a win.

Our results obtained for searching uniform trees representing Pear-games and incremental games, when product propagation was used for backing-up, are presented in Table XXVII and XXVIII, respectively. Results from Table XXVII show that pathology does not occur for any tested value of serch depth when the product propagation was used for backing-up. For incremental games, however, the results show that the probability of making the correct decision was lower for product propagation than for minimaxing.

Depth of Search	terminal level of the game, t										
	3	4	5	6	7	8	9	10	11	12	13
1	0.947	0.906	0.842	0.809	0.762	0.729	0.694	0.670	0.643	0.621	0.620
2	1.000	0.934	0.876	0.816	0.772	0.732	0.695	0.669	0.641	0.620	0.619
3	1.000	1.000	0.904	0.840	0.779	0.746	0.698	0.673	0.642	0.623	0.619
4		1.000	1.000	0.865	0.815	0.754	0.708	0.676	0.647	0.675	0.622
5			1.000	1.000	0.837	0.782	0.718	0.683	0.643	0.629	0.622
6				1.000	1.000	0.802	0.747	0.684	0.654	0.630	0.623
7					1.000	1.000	0.777	0.710	0.664	0.637	0.628
8						1.000	1.000	0.747	0.695	0.646	0.638
9							1.000	1.000	0.711	0.658	0.637
10								1.000	1.000	0.684	0.660
11									1.000	1.000	0.677
12										1.000	1.000
13											1.000

TABLE XXV

Estimation of the probability of correct decision as a function of search depth using the product-propagation rule for backing-up. Results obtained by Nau [19] for the Pearl-games of width two.

Depth of search	terminal level of the game, t										
	3	4	5	6	7	8	9	10	11	12	13
1	0.982	0.970	0.941	0.936	0.924	0.933	0.914	0.920	0.914	0.913	0.910
2	1.000	0.981	0.963	0.949	0.944	0.945	0.929	0.928	0.929	0.924	0.922
3	1.000	1.000	0.981	0.973	0.959	0.962	0.943	0.940	0.940	0.937	0.931
4		1.000	1.000	0.985	0.972	0.968	0.957	0.948	0.950	0.943	0.936
5			1.000	1.000	0.983	0.978	0.968	0.964	0.957	0.952	0.948
6				1.000	1.000	0.988	0.980	0.973	0.962	0.962	0.952
7					1.000	1.000	0.987	0.984	0.972	0.969	0.960
8						1.000	1.000	0.992	0.979	0.977	0.967
9							1.000	1.000	0.987	0.984	0.976
10								1.000	1.000	0.991	0.978
11									1.000	1.000	0.992
12										1.000	1.000
13											1.000

TABLE XXVI

Estimation of the probability of correct decision, as a function of search depth d. The product-propagation rule was used for backing-up results obtained by Nau [19] for the incremental games of width two.

Depth of Search	terminal level of the game, t										
	3	4	5	6	7	8	9	10	11	12	13
1	0.941	0.902	0.856	0.807	0.790	0.712	0.699	0.690	0.623	0.619	0.618
2	1.000	0.921	0.880	0.818	0.790	0.722	0.701	0.699	0.639	0.625	0.620
3	1.000	1.000	0.912	0.852	0.784	0.739	0.709	0.712	0.647	0.627	0.622
4		1.000	1.000	0.873	0.819	0.760	0.713	0.715	0.651	0.670	0.627
5			1.000	1.000	0.821	0.794	0.716	0.716	0.655	0.671	0.629
6				1.000	1.000	0.812	0.762	0.721	0.664	0.682	0.633
7					1.000	1.000	0.791	0.732	0.671	0.687	0.641
8						1.000	1.000	0.749	0.682	0.689	0.641
9							1.000	1.000	0.705	0.692	0.643
10								1.000	1.000	0.694	0.647
11									1.000	1.000	0.652
12										1.000	1.000
13											1.000

TABLE XXVII

Results from duplication of Nau's experiments. Estimation of the probability of correct decision as a function of search depth d, using the product-propagation rule for backing-up. Results obtained for the Pearl-games of width two. Nau's results are given in [19].

Depth of Search	terminal level of the game, t										
	3	4	5	6	7	8	9	10	11	12	13
1	0.980	0.965	0.931	0.927	0.910	0.927	0.910	0.912	0.908	0.906	0.905
2	1.000	0.972	0.935	0.935	0.932	0.931	0.921	0.918	0.917	0.914	0.912
3	1.000	1.000	0.970	0.958	0.936	0.949	0.935	0.931	0.932	0.921	0.920
4		1.000	1.000	0.973	0.951	0.957	0.949	0.940	0.941	0.931	0.928
5			1.000	1.000	0.970	0.971	0.951	0.952	0.950	0.935	0.932
6				1.000	1.000	0.980	0.962	0.959	0.953	0.942	0.941
7					1.000	1.000	0.979	0.964	0.961	0.948	0.950
8						1.000	1.000	0.971	0.969	0.953	0.958
9							1.000	1.000	0.977	0.955	0.961
10								1.000	1.000	0.959	0.966
11									1.000	1.000	0.971
12										1.000	1.000
13											1.000

TABLE XXVIII

Results from duplication of Nau's experiments. Estimation of the probability of correct decision, as a function of search depth, using product-propagation rule for backing-up. Results obtained for the incremental games of width two. Nau's results are given in [19].

5.4. Concluding remarks.

The fact that increasing the depth of search is beneficial in real-world games does not mean that we may ignore the pathological phenomenon. The absence of pathology in games such as chess or checkers means only that the degradation of the decision quality is masked by some other processes. Further studies are needed to discover such processes. More analysis is required to understand the nature of dependencies which exist in real-world games. But it is possible to improve the decision quality using backing-up rules which are more appropriate for evaluating the game positions. It is also possible to estimate probabilities of win for a certain position and to use the product propagation rule, instead of minimaxing, for backing-up. This method proved to be effective for pathological games, but it can not be used for real-world games unless an appropriate pruning strategy is developed.

CHAPTER 6.

CONCLUSIONS

6.1. Highlights of Results Observed.

In this thesis different problems araising for the game-playing computer programs were discussed, we restrict ourselves to two-person, zero-sum games of perfect-information. Six different pruning strategies : Alphabeta [12], Branch-and-bound [12], Palphabeta [9], PVS [16], Scout [23] and SSS* [30] were reviewed. The theoretical analysis of some of these strategies was presented, following [12,27,28], showing that Alphabeta, Scout and SSS* have very similar performance characteristics. The empirical comparison of these six pruning strategies on different kinds of simulated game trees was then presented. It was shown that the performance of these strategies varies substantially. It has been concluded that the Alphabeta algorithm will remain the most popular strategy for playing a real-world game. Then the different methods of speeding-up the tree search were reviewed. But the need of searching deeper, which is the goal of these methods, may be questioned because of the pathological phenomenon, which was also described in this work. It was shown that there exist a large class of game trees for which searching deeper does not improve the quality of decision made, following [5,7,18,19,25,27]. The product propagation rule for backing-up and the B* procedure were discussed as the possible methods of overcoming pathology. On the nonuniform trees and on game trees representing the incremental games the pathology was not observed. So, the existance of terminal nodes at any level of a game tree and the dependencies between parent and son nodes may be the possible reasons why pathology is not observed on the real-world games.

6.2. Suggestions for the Further Research.

In this work different pruning strategies were compared on the simulated game trees. It will be interesting to compare performance of these strategies on real-world games. For such comparison different evaluation functions may be tested to see how much the pruning depends on the function used. When analyzing different games, it may be possible to find a model of a game in which the dependencies between node and its succesors are described by a mathematical formula.

Since the product propagation rule was found to be a cure for pathology, it will be worthwhile to develop a pruning strategy for this approach. For example such a strategy may be based on the fact that lower and upper bounds on the value of a node can be derived by examining one or more of node's sons. These bounds may be calculated using the upper and lower bounds for the value of probability (0 and 1). It will be also worthwhile to analyze the performance of B* procedure on the pathological game trees to see if using this algorithm we may overcome pathology.

REFERENCES.

[1] S.G. Akl, D.T. Barnard and R.J. Doran, ' Design, Analysis and Implementation of a Parallel Tree Search Algorithm,' IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-4, no.2, pp 192-203, March 1982.

[2] S.G. Akl and R.J. Doran, ' A Comparison of Parallel Implementation of the Alphabeta and Scout Tree Search Algorithms Using the Game of Checkers,' Sigart Newsletter, vol.80, April 1982, pp 77-83.

[3] S.G. Akl and M.M. Newborn, ' The Principal Continuation and the Killer Heuristic,' in Proceeding of the 32nd Annual ACM Conference, Seatle Washigton, pp 466-473, 1977.

[4] G.M. Baudet, ' On the Branching Factor of the Alphabeta Pruning Algorithm,' Artificial Intelligence, vol.10, no.2, pp 173-199, 1978.

[5] D.F Beal, ' An Analysis of Minimax,' Advances in Computer Chess 2, editor : M.R.B. Clarke, Edinburgh, University Press, pp 103-109, 1980.

[6] D.F. Beal, ' Benefits of Minimax Search,' Advances in Computer Chess 3, editor : M.R.B. Clarke, Pergamon Press, New York, pp 17-24, 1982.

[7] H. Berliner, ' The B* Tree Search Algorithm : A Best-First Proof Procedure,' Artificial Intelligence, vol:12, no.1, pp 23-40, 1979.

[8] I. Bratko and M. Gams, ' Error Analysis of the Minimax Principle,' Advances in Computer Chess 3, editor : M.R.B. Clarke, Pergamon Press, New York, pp 1-15, 1982.

[9] M.S. Campbell and T.A. Marsland, ' Comparison of Minimax Searching Algorithms,' Artificial Intelligence, vol.20, no.4, pp 347-367, 1983.

[10] N.M. Darwish, ' A Quantitive Analysis of Alpha-Beta Pruning Algorithm,' Artificial Intelligence, vol.21, no.5, pp 405-433, 1983.

[11] R.A. Finkel and J.P. Fishburn, ' Parallelism in Alpha-Beta Search,' Artificial Intelligence, vol.19, no.1, pp 89-106, 1982.

[12] D.E. Knuth and R.W. Moore, ' An Analysis of Alpha-Beta Pruning,' Artificial Intelligence, vol.6, no.9, pp 293-326, 1975.

[13] V. Kumar and L.N. Kanal, ' A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures,' Artificial Intelligence, vol.21, no.2, pp 179-198, 1983.

[14] V. Kumar and L.N. Kanal, ' Parallel Branch-and-Bound Formulation for AND/OR Tree Search,' IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-6, no.6, pp. 768-778, November 1984.

[15] T.A. Marsland and M.Campbell, ' A Survey on Enhancements to the Alpha-Beta Algorithm,' in Proceeding of the 36th Annual ACM Conference, Los Angeles, California, 1981.

[16] T.A. Marsland, ' Relative Efficiency of the Alpha-Beta Implementations,' in Proceeding of the 8th International Joint Conference on Artificial Intelligence, August 1983, Karlsruhe, West Germany, pp 763-766.

[17] D.S. Nau, ' The Last Player Theorem,' Artificial Intelligence, vol. 18, no.2, pp 53-65, 1982.

[18] D.S. Nau, ' An Investigation of the Causes of Pathology in Games,' Artificial Intelligence, vol. 19, no.2, pp 257-278, 1982.

[19] D.S. Nau, ' Pathology in Game Trees Revisited and an Alternative to Minimaxing,' Artificial Intelligence, vol. 21, no.2, pp 221-244, 1983.

[20] M.M. Newborn, ' The Efficiency of the Alpha-Beta Search on Trees with Branch-dependent Terminal Node Scores,' Artificial Intelligence, vol.8, no.2, pp 137-153, 1977.

[21] N.J. Nilsson, Principles of Artificial Intelligence, Paolo Alto, California: TIOGA, 1980.

[22] A.J. Palay, ' An Experimental Analysis of the B* Tree Search Algorithm,' Department of Computer Science, Carnegia-Mellon University, Pittsburgh, Report CMU-CS-80-106, 1980.

[23] J. Pearl, ' Asymptotic Properties of Minimax Trees and Game-searching Procedures,' Artificial Intelligence, vol.14, no.1, pp 113-138, 1980.

[24] J. Pearl, ' The Solution for the Branching Factor of the Alphabeta,' Communication of the ACM, vol.25, no.8, pp 559-564, August 1982.

[25] J. Pearl, ' On the Nature of Pathology in Game Searching,' Artificial Intelligence, vol. 20, no.3, pp 427 - 453, 1983.

[26] J. Pearl, ' Some Recent Results in Heuristic Search Theory,' IEEE Transaction on Pattern Analysis and Machine Intelligence, vol. PAMI-6, no.1, pp 1-12, January 1984.

[27] J. Pearl, Heuristics. Reading, MA: Addison-Wesley, 1984.

[28] I. Roizen and J. Pearl, ' A Minimax Algorithm Better than Alphabeta ? Yes and No,' Artificial Intelligence, vol.21, no.2, pp 199-220, 1983.

[29] J. Slagle and J. Dixon, ' Experiments with Some Programs that Search Game Trees,' Journal of the Association for Computing Machinery, vol.16, no.2, pp 198-207, April 1969.

[30] G. Stockman, ' A Minimax Algorithm Better than Alpha-Beta ?,' Artificial Intelligence, vol.12, pp 179-196, 1979.

[31] M. Tarsi, ' Optimal Searching of Some Game Trees,' Journal of the Association for Computing Machinery, vol.30, no.3, pp 384-396, 1983.

APPENDIX 1.

COMPARISON OF DIFFERENT VERSIONS OF SCOUT ALGORITHM.

As we have mentioned in section 2.5, we have also compared three different versions of the Scout algorithm : minimax, negamax and the Campbell-Marsland version. In section 2.5 the algorithmic formulation of the Scout under the negamax framework was presented. Below the algorithmic formulation of the Scout under the minimax framework is presented, as given by Pearl in [23]. The function is invoked by calling scout (root).

```
1.FUNCTION scout ( p : TREENODE ) : NUMERIC ;
2. VAR  i,f : INTEGER ; m : NUMERIC ;  op : BOOLEAN ;
3. BEGIN
4.   f:=generate(p); /* generate sons p1 , p2, ..., pf
                       of node p   */
5.   IF f=0 THEN return(staticvalue(p)); /* p is leaf node */
6.   m:=scout(p1 );
7.   FOR i:=2 TO  f  DO
8.    IF p  is MAX node THEN op:=FALSE;
       /* op is a parameter used to compare nodes in
        function test invoked by scout  */
9.     IF (NOT test(pi ,m,op))  THEN  m:=scout(pi );
 /* if function test returns false, scout evaluates node pi ,
   else it is not evaluated */
10.  ELSE /* pi  is MIN node  */
```

```
11.   op:=TRUE;
12.    IF test(p_i ,m,op) THEN m:=scout(p_i );
 /* if function test returns true, scout evaluates node p_i ,
    else it is not evaluated  */
13. return(m); /* return value of m as the function value */
14.END.

1.FUNCTION test (p :TREENODE ; v :INTEGER; op :BOOLEAN)
                  : BOOLEAN;
/* if op is true nodes to be compared are at same level
    of the tree, else at different levels  */
2. VAR i,f : INTEGER ;
3. BEGIN
4. f:=generate(p); /* generate sons p_1 , p_2 , ..., p_f
                             of node p   */
5. IF f=0  THEN   /*  p is a leaf node  */
6.   IF ((staticvalue(p) >= v)  AND  (not op ))  OR
7.      ((staticvalue(p) > v)  AND  (  op))  THEN
8.         return TRUE /* node p can not be the best son */
9.   ELSE return FALSE; /* node p may become the best son */
10. FOR i:=1  TO  f DO
11.  BEGIN
12.     IF (p_i  is MAX node) AND (test(p_i ,v,op)) THEN
13.         return TRUE; /* node p_i  can not become
                              the best son  */
14.     IF (p_i  is MIN node) AND (NOT test(p_i ,v,op)) THEN
15.        return FALSE;
```

16. END;

17. IF (p_i is MAX node) THEN return TRUE

18. ELSE return FALSE;

19. END.

The Campbell-Marsland [9] version of Scout uses Alphabeta instead of Test for inequality checking, and it is very similar to the Scout presented in section 2.5, the only difference is that line 9 becomes :

t:=-alphabeta(p_i,-m-1,-m); if (t>m) then m:=-scout(p_i).

Campbell and Marsland [9] have also presented the negamax version of Test. Their version is different than the one described in section 2.5. Below their version of Test algorithm is presented. It may be invoked by the Scout as described in section 2.5, Scout which uses Test for inequality checking.

1. FUNCTION testm (p : TREENODE ; v : INTEGER)
                    : BOOLEAN;

2. VAR i,f : INTEGER;

3. BEGIN

4. f:=generate(p); /* generate sons p_1, p_2, ..., p_f
                    of node p */

5. IF f=0 THEN /* p is a leaf node */

6.   IF (staticvalue(p) > v) THEN

7.   return TRUE /* node p can not be the best son */

8.   ELSE return FALSE; /* node p may become the best son */

9. FOR i:=1 to f DO

10.   IF NOT(testm(p_i,-v) THEN

11.   return TRUE; /* node p_i can not become

the best son */

12. return FALSE;

13. END.

As we see testm does not use two kinds of operators (> or ≥) for nodes at different levels of a tree. Because of this Scout algorithm which invokes testm function will prune less nodes. In Figure 57 an example of such a situation is shown.

Under the criterion of nodes created (all or only leaf) the three tested versions of Scout algorithm always performed identically. Under the criterion of node-visits for all sixteen cases except for nonuniform trees with real-dependent static-values assignment they have also performed identically. For nonuniform trees with real-dependent static-values assignment the two versions which call Test algorithm, outperform the Campbell-Marsland version, one which calls Alphabeta for inequality checking. For example for N(3,5) with real-dependent static-value assignment minimax and negamax versions visited on average 78.54 nodes, 61.26 leaf nodes. The Campbell-Marsland version of Scout visited on average 79.30 nodes, 62.06 leaf nodes. Under the criterion of CPU time taken the negamax and the Campbell-Marsland versions of Scout performed very similary and the minimax version was the slowest. As an example, for uniform trees with 0.4-ordered-independent

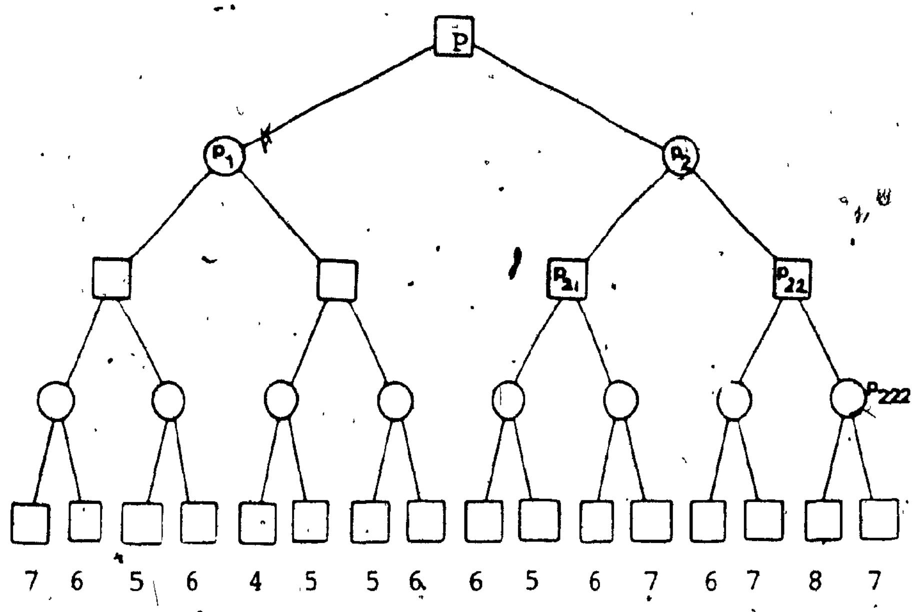


FIGURE 57.

An example to show that Scout which uses the Campbell-Marsland [9] version of Test will examine more nodes than Scout which invokes Test presented in section 2.5.
Since value of node p_{21} is 6, and the provisional value of node p_{22} is 6 the node p_{222} does not have to be examined. Because the Campbell-Marsland function testm uses only > operator, then Scout which invokes such a Test procedure examines nodes p_{222}, p_{2221}, and p_{2222}.

static-values assignment in Figure 58 the average CPU time taken by these three different versions of Scout algorithm versus the width of tree has been plotted.

FIGURE 58

Plot of average CPU time taken by three different versions of Scout algorithm against width of a uniform tree of depth 4. Static-values were assigned to leaf nodes by 0.4-ordered-independent scheme.