UNIT 4 TREE SEARCH

CONTENTS

1.0 INTRODUCTION

Tree search algorithms are specialised versions of graph search algorithms, which take the properties of trees into account. An example of tree search is the game tree of multiple-player games, such as chess or backgammon, whose nodes consist of all possible game situations that could result from the current situation. The goal in these problems is to find the move that provides the best chance of a win, taking into account all possible moves of the opponent(s). Similar problems occur when humans or machines have to make successive decisions whose outcomes are not entirely under one's control, such as in robot guidance or in marketing, financial or military strategy planning. This kind of problems has been extensively studied in the context of artificial intelligence. Examples of algorithms for this class are the minimax algorithm, alpha-beta pruning, and the A* algorithm.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

describe a game tree
describe some two-player games search algorithms
explain intelligent backtracking
solve some simple problems on tree search.

3.0 MAIN CONTENT

3.1 Game Tree

A game tree is a directed graph whose nodes are positions in a game and whose edges are moves. The complete game tree for a game is the game tree starting at the initial position and containing all possible moves from each position; the complete tree is the same tree as that obtained from the extensive-form game representation.

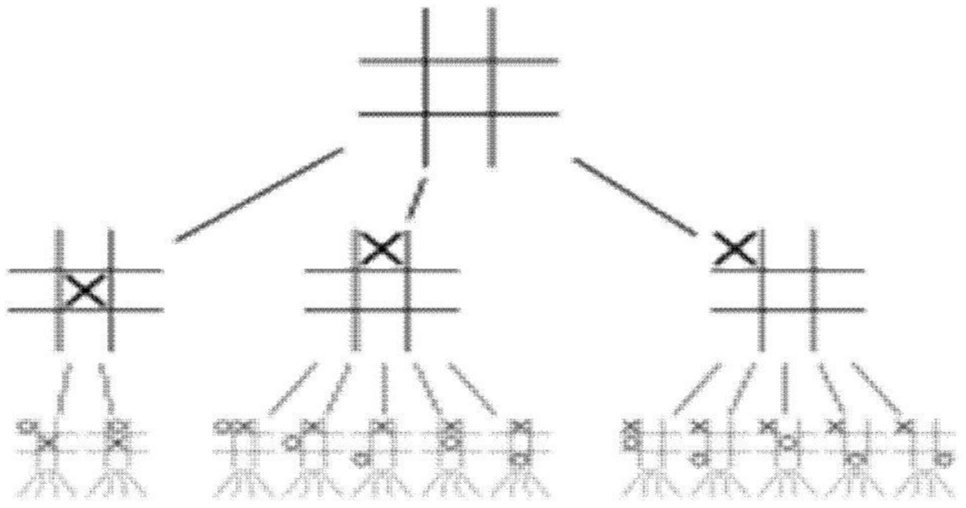

Figure 1: Game tree for tic-tac-toe

The first two plies of the game tree for tic-tac-toe.

The diagram shows the first two levels, or *plies*, in the game tree for tic-tac-toe. We consider all the rotations and reflections of positions as being equivalent, so the first player has three choices of move: in the center, at the edge, or in the corner. The second player has two choices for the reply if the first player played in the center, otherwise five choices. And so on.

The number of leaf nodes in the complete game tree is the number of possible different ways the game can be played. For example, the game tree for tic-tac-toe has 26,830 leaf nodes.

Game trees are important in artificial intelligence because one way to pick the best move in a game is to search the game tree using the minimax algorithm or its variants. The game tree for tic-tac-toe is easily searchable, but the complete game trees for larger games like chess are much too large to search. Instead, a chess-playing program searches a partial game tree: typically as many plies from the current position as it can search in the time available.

Except for the case of "pathological" game trees (which seem to be quite rare in practice), increasing the search depth (i.e., the number of plies searched) generally improves the chance of picking the best move.

Two-person games can also be represented as and-or trees. For the first player to win a game there must be a winning move for all moves of the second player. This is represented in the and-or tree by using disjunction to represent the first player's alternative moves and using conjunction to represent all of the second player's moves.

3.2 Two-Player Games Search Algorithms

The second major application of heuristic search algorithms in Artificial Intelligence is two-player games. One1 of the original challenges of AI, which in fact predates the term, Artificial Intelligence, was to build a program that could play chess at the level of the best human players, a goal recently achieved.

Following are the algorithms meant to solve this problem.

Minimax Search
Alpha-Beta Pruning
Quiecence
Transposition Tables
Limited Discrepancy Search
Intelligent Backtracking

3.2.1 Minimax Search Algorithm

The standard algorithm for two-player perfect-information games such as chess, checkers or Othello is minimax search with heuristic static evaluation. The minimax search algorithm searches forward to a fixed depth in the game tree, limited by the amount of time available per move. At this search horizon, a heuristic function is applied to the frontier nodes. In this case, a heuristic evaluation is a function that takes a board position and returns a number that indicates how favourable that position is for one player relative to the other. For example, a very simple heuristic evaluator for chess would count the total number of pieces on the board for one player, appropriately weighted by their relative strength, and subtract the weighted sum of the opponent's places. Thus, large positive values would correspond to strange positions for one player called *MAX*, whereas large negative values would represent advantageous situation for the opponent called *MIN*.

Given the heuristic evaluations of the frontier nodes, minimax search algorithm recursively computes the values for the interior nodes in the tree according to the *maximum rule*. The value of a node where it is MAX's turn to move is the maximum of the values of its children, while the value of the node where MIN is to move is the minimum of the values of its children. Thus at alternative levels of the tree, the maximum values of the children are backed up. This continues until the values of the immediate children of the current position are computed at which point one move to the child with the maximum or minimum value is made depending on whose turn it is to move.

3.2.2 Alpha-Beta Pruning

One of the most elegant of all AI search algorithms is alpha-beta pruning. The idea, similar to branch-and-bound, is that the minimax value of the root of a game tree can be determined without examining all the nodes at the search frontier.
Only the labeled nodes are generated by the algorithm, with the heavy black lines indicating pruning. At the square node MAX is to move, while at the circular nodes it MIN's turn. The search proceeds depth-first to minimize the memory required, and only evaluates a node when necessary. First node and f are statically evaluated at 4 and 5, respectively, and their minimum value, 4 is backed up to their parent node d. Node h is then evaluated at 3, and hence the value of its parent node g must be less than or equal to 3, since it is the minimum of 3 and the unknown value of its right child. Thus, we level node g as <=3. The value of node c must be 4 then, because it is the maximum of 4 and a value that is less than or equal to 3. Since we have determined the minimax value of node c, we do not need to evaluate or even generate the brother of node h.

Similarly, after evaluating nodes k and l at 6 and 7 respectively, the backed up value of their parent node j is 6, the minimum of these values. This tells us that the minimax value of node I must be greater than or equal to 6 since it is the maximum of 6 and the unknown value of its right child. Since the value of node b is the minimum of 4 and a value that is greater than or equal to 6, it must be 4 and hence we achieve another cut off.

The right half of the tree shows an example of deep pruning. After evaluating the left half of the tree, we know that the value of the root node a is greater than or equal to four, the minimax value of node b. Once node q is equated at 1, the value of its parent node nine must be less than or equal to 1. Since the value of the root is greater than or equal to two.

Moreover, since the value of node m is the minimum of the value of node n and its brother, and node n has a value less than or equal to two, the value of node m must also be less than or equal to two. This causes the brother of node n to be pruned, since the value of the root node a is greater than or equal to four. Thus, we computed the minimax value of the root of the tree to be four, by generating only seven of sixteen leaf nodes in this area.

Since alpha-beta pruning performs a minimax search while pruning much of the tree, its effect is to allow a deeper search with the same amount of computation. This raises the question of how much does alpha-beta improve performance. The best way to characterize the efficiency of a pruning algorithm is in terms of its effective branching factor. The effective branching factor is the dth root of the frontier nodes that must be evaluated in a search to depth d, in the limit of large d.
The efficiency of alpha-beta pruning depends upon the order in which nodes are encountered at the search frontier. For any set of frontier node values, there exists same ordering of the values such that alpha-beta will not perform any cut offs at all. In that case, the effective branching factor is reduced from b to b^1/2., the square root of the brute-force branching factor. Another way of viewing the perfect ordering case is that for the same amount of computation, one can search twice as deep as with alpha-beta pruning as without since the search tree grows exponentially with depth, doubling the search horizon is a dramatic improvement.

In between worst-possible ordering and perfect ordering is random ordering, which is the average case. Under random ordering of the frontier nodes, alpha-beta pruning reduces the effective branching factor approximately b3/4. This means that one can search 4/3 as deep with alpha-beta, yielding as 33% improvement in search depth.

In practice, however, the effective branching factor of alpha-beta is closer to the best case of b1/2 due to node ordering. The idea of node ordering is that instead of generating the tree left to right, we can reorder the tree based on static evaluations of the interior nodes. In other words, the children of MAX nodes are expanded in decreasing order of their static values, while the children of MIN nodes are expanded in increasing order of their static values.

3.2.3 Quiescence Search

The idea of quiescence is that the static evaluator should not be applied to positions whose values are unstable, such as those occurring in the middle of the piece trade. In those positions, a small secondary search is conducted until the static evaluation becomes more stable. In games such as chess or checkers, this can be achieved by always exploring any capture moves one level deeper. This extra search is called quiescence search. Applying quiescence search to capture moves quickly will resolve the uncertainties in the position.

3.2.4 What is Transposition Table?

A transposition table is a table of previously encountered game states, together with their backed-up minimax values. Whenever a new state is generated, if it is stored in the transposition table, its stored value is used instead of searching the tree below the node. Transposition table can be used very effectively so that reachable search depth in chess, for example, can be doubled.

3.2.5 Limited Discrepancy Search (LDS)

Limited Discrepancy Search (LDS) is a completely general tree-search algorithm, but is most useful in the context of constraint satisfaction problems in which the entire tree is too large to search exhaustively. In that case, we would like to search that subset of the tree that is most likely to yield a solution in the time available. Assume that we can heuristically order a binary tree so that at any node, the left branch is more likely to lead to a solution than the right branch. LDS then proceeds in a series of depth-first iterations. The first iteration explores just the left-most path in the tree. The second iteration explores those root-to-leaf paths with exactly one right branch, or discrepancy in them. In general, each iteration explores those paths with exactly k discrepancies, with k ranging from zero to the depth of the tree. The last iteration explores just the right most branch. Under certain assumptions, one can show that LDS is likely to find a solution sooner than a strict left-to-right depth-first search.

3.2.6 What is Intelligent Backtracking?

Performance of brute force backtracking can be improved by using a number of techniques such as variable ordering, value ordering, back jumping, and forward checking.

The order in which variables are instantiated can have a large effect on the size of the search tree. The idea of variable ordering is to order the variables form most constrained to least constrained. For example, if a variable has only a single value remaining that is consistent with the previously instantiated variable, it should be assigned that value immediately.

In general, the variables should be instantiated in increasing order of the size of their remaining domains. This can either be done statically at the beginning of the search or dynamically, reordering the remaining variables each time a variable is assigned a new value.

The order in which the value of a given variable is chosen determines the order in which the tree is searched. Since it does not affect the size of the tree, it makes no difference if all solutions are to be found. If only a single solution is required, however, value ordering can decrease the time required to find a solution. In general, one should order the values from least constraining to most constraining in order to minimize the time required to find a first solution.

An important idea, originally called back jumping, is that when an impasse is reached, instead of simply undoing the last decision made, the decision that actually caused the failure should be modified. For example, consider the three-variable problem where the variables are instantiated in the order x,y,z. Assume that values have been chosen for both x and y, but that all possible values for z conflict with the value chosen for x. In chronological backtracking, the value chosen for y would be changed, and then all the possible values for z would be tested again, to no avail. A better strategy in this case is to go back to the source of the failure, and change the value of x before trying different values for y.

When a variable is assigned a value, the idea of forward checking is to check each remaining uninstantiated variable to make sure that there is at least one assignment for each of them that is consistent with the previous assignments. If not, the original variable is assigned its next value.

4.0 CONCLUSION

In computer science, a search tree is a binary tree data structure in whose nodes data values are stored from some ordered set, in such a way that in-order traversal of the tree visits the nodes in ascending order of the stored values. This means that for any internal node containing a value v, the values x stored in its left sub tree satisfy $x \leq v$, and the values y stored in its right sub tree satisfy $v \leq y$. Each sub tree of a search tree is by itself again a search tree.

5.0 SUMMARY

In this unit, you learnt that:

A game tree is a directed graph whose nodes are positions in a game and whose edges are moves
The second major application of heuristic search algorithms in Artificial Intelligence is two-player games
The standard algorithm for two-player perfect-information games such as chess, checkers or Othello is minimax search with heuristic static evaluation
One of the most elegant of all AI search algorithms is alpha-beta pruning.
The idea of quiescence is that the static evaluator should not be applied to positions whose values are unstable, such as those occurring in the middle of the piece trade.

6.0 TUTOR-MARKED ASSIGNMENT

Answer the following questions on informed search and heuristics:

1. Which of the following are admissible, given admissible heuristics $h1$, $h2$? Which of the following are consistent, given consistent heuristics $h1$, $h2$?
2. $h(n) = min\{h1(n), h2(n)\}$
3. $h(n) = wh1(n) + (1 - w)h2(n)$, where 0 w 1
4. $h(n) = max\{h1(n), h2(n)\}$
5. The heuristic path algorithm is a best-first search in which the objective function is $f(n) = (2 - w)g(n) + wh(n)$. For what values of w is this algorithm guaranteed to be optimal when h is admissible? What kind of search does this perform when $w = 0$? When $w = 1$? When $w = 2$?

7.0 REFERENCES/FURTHER READING

Christopher, D. Manning & Schutze, H. *Foundations of Statistical Natural Language Processing*. The MIT Press.

Hu, Te Chiang. Shing, M. (2002). *Combinatorial Algorithms*. Courier Dover Publications. ISBN 0486419622. http://books.google.com/?id=BF5_bCN72EUC. Retrieved 2007-04-02.

Nau, D. (1982). "An investigation of the causes of pathology in games". *Artificial Intelligence* 19: 257–278. doi:10.1016/0004-3702(82) 90002-9.

Allis, V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands. ISBN 9090074880. http://fragrieu.free.fr/ SearchingForSolutions.pdf.