3.4.1. THE MINIMAX PROCEDURE

Many simple games (as well as some "ending" sequences of more complex games) can be handled by search techniques that are analogous to those used for finding AND/OR solution graphs. The solution graph, then, represents a complete playing strategy. Grundy's game, tic-tac-toe (naughts and crosses), various versions of nim, and some chess and checker end-games are examples of simple games in which AND/OR search to termination is feasible. A gross estimate of the size of the tic-tac-toe game tree, for example, can be obtained by noting that the start node has nine successors, these in turn have eight, etc., yielding 9! (or 362,880) nodes at the bottom of the tree. Many of the paths end in terminal nodes at shallower levels, however, and further reductions in the size of the tree result if symmetries are acknowledged.

For more complex games, such as complete chess and checker games, AND/OR search to termination is wholly out of the question. It has been estimated that the complete game tree for checkers has approximately 10^{40} nodes and the chess tree has approximately 10^{120} nodes. (It would take about 10^{21} centuries to generate the complete checker tree, even assuming that a successor could be generated in 1/3 of a nanosecond.) Furthermore, heuristic search techniques do not reduce the effective branching factor sufficiently to be of much help. Therefore, for complex games, we must accept the fact that search to termination is impossible; that is, we must abandon the idea of using this method to prove that a win or draw can be obtained (except perhaps during the end-game).

Our goal in searching a game tree might be, instead, merely to find a good first move. We could then make the indicated move, await the opponent's reply, and search again to find a good first move from this new position. We can use either breadth-first, depth-first, or heuristic methods, except that the termination conditions must now be modified. Several artificial termination conditions can be specified based on such factors as a time limit, a storage-space limit, and the depth of the deepest node in the search tree. It is also usual in chess, for example, not to terminate if any of the tip nodes represent "live" positions, that is, positions in which there is an immediate advantageous swap.

After search terminates, we must extract from the search graph an estimate of the "best" first move. This estimate can be made by applying a *static* evaluation function to the leaf nodes of the search graph. The evaluation function measures the "worth" of a leaf node position. The

measurement is based on various features thought to influence this worth; for example, in checkers some useful features measure the relative piece advantage, control of the center, control of the center by kings, and so forth. It is customary in analyzing game trees to adopt the convention that game positions favorable to *MAX* cause the evaluation function to have a positive value, while positions favorable to *MIN* cause the evaluation function to have a negative value; values near zero correspond to game positions not particularly favorable to either *MAX* or *MIN*.

A good first move can be extracted by a procedure called the *minimax* procedure. (For simplicity we explain this procedure and others depending on it as if the game graph were really just a game tree.) We assume that were *MAX* to choose among tip nodes, he would choose that node having the largest evaluation. Therefore, the (*MAX* node) parent of *MIN* tip nodes is assigned a *backed-up* value equal to the maximum of the evaluations of the tip nodes. On the other hand, if *MIN* were to choose among tip nodes, he would presumably choose that node having the smallest evaluation (that is, the most negative). Therefore, the (*MIN* node) parent of *MAX* tip nodes is assigned a backed-up value equal to the minimum of the evaluations of the tip nodes. After the parents of all tip nodes have been assigned backed-up values, we back up values another level, assuming that *MAX* would choose that node with the largest backed-up value while *MIN* would choose that node with the smallest backed-up value.

We continue to back up values, level by level, until, finally, the successors of the start node are assigned backed-up values. We are assuming it is *MAX*'s turn to move at the start, so *MAX* should choose as his first move the one corresponding to the successor having the largest backed-up value.

The utility of this whole procedure rests on the assumption that the backed-up values of the start node's successors are more reliable measures of the ultimate relative worth of these positions than are the values that would be obtained by directly applying the static evaluation function to these positions. The backed-up values are, after all, based on "looking ahead" in the game tree and therefore depend on features occurring nearer the end of the game.

A simple example using the game of tic-tac-toe illustrates the minimaxing method. Let us suppose that *MAX* marks crosses (×) and *MIN*

marks circles (O) and that it is *MAX*'s turn to play first. We conduct a breadth-first search, until all of the nodes at level 2 are generated, and then we apply a static evaluation function to the positions at these nodes. Let our evaluation function $e(p)$ of a position p be given simply by:

If p is not a winning position for either player,

$$e(p) = (\text{number of complete rows, columns, or diagonals that are still open for } MAX) - (\text{number of complete rows, columns, or diagonals that are still open for } MIN).$$

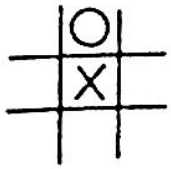
If p is a win for *MAX*,

$$e(p) = \infty \ (\infty \text{ denotes a very large positive number}).$$

If p is a win for *MIN*,

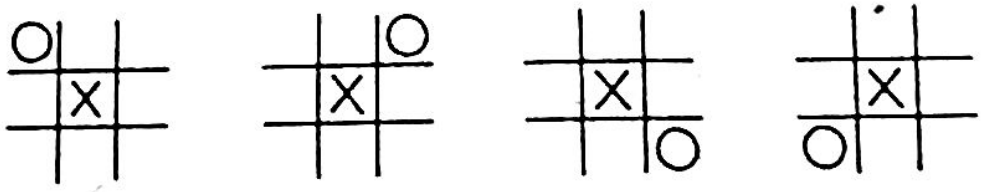
$$e(p) = -\infty.$$

Thus, if p is



we have $e(p) = 6 - 4 = 2$.

We make use of symmetries in generating successor positions; thus the following game states



are all considered identical. (Early in the game, the branching factor of the tic-tac-toe tree is kept small by symmetries; late in the game, it is kept small by the number of open spaces available.)

In Figure 3.8 we show the tree generated by a search to depth 2. Static evaluations are shown below the tip nodes, and backed-up values are circled.

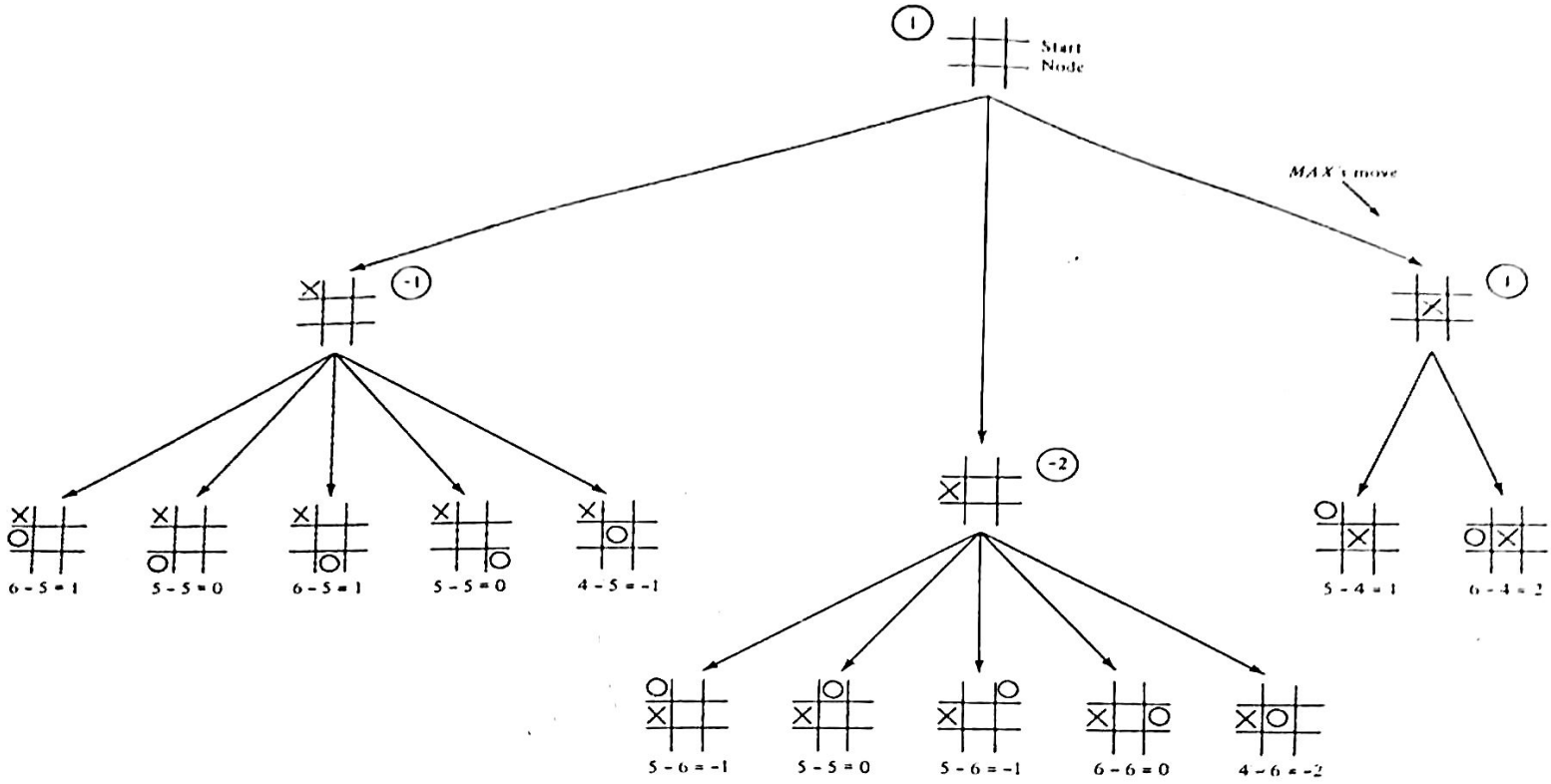


Fig. 3.8 Minimax applied to tic-tac-toe (stage 1).

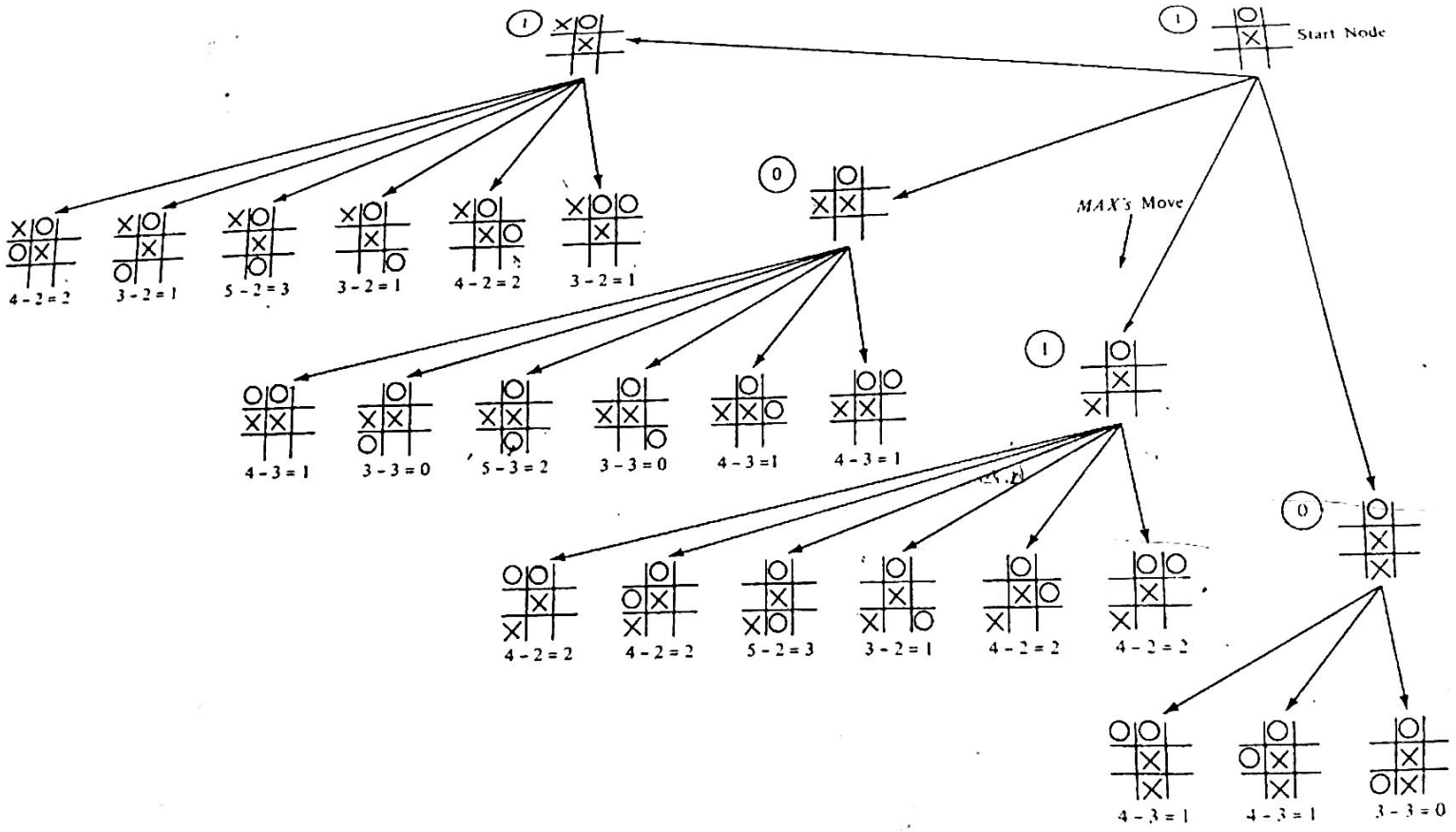


Fig. 3.9 Minimax applied to tic-tac-toe (stage 2).

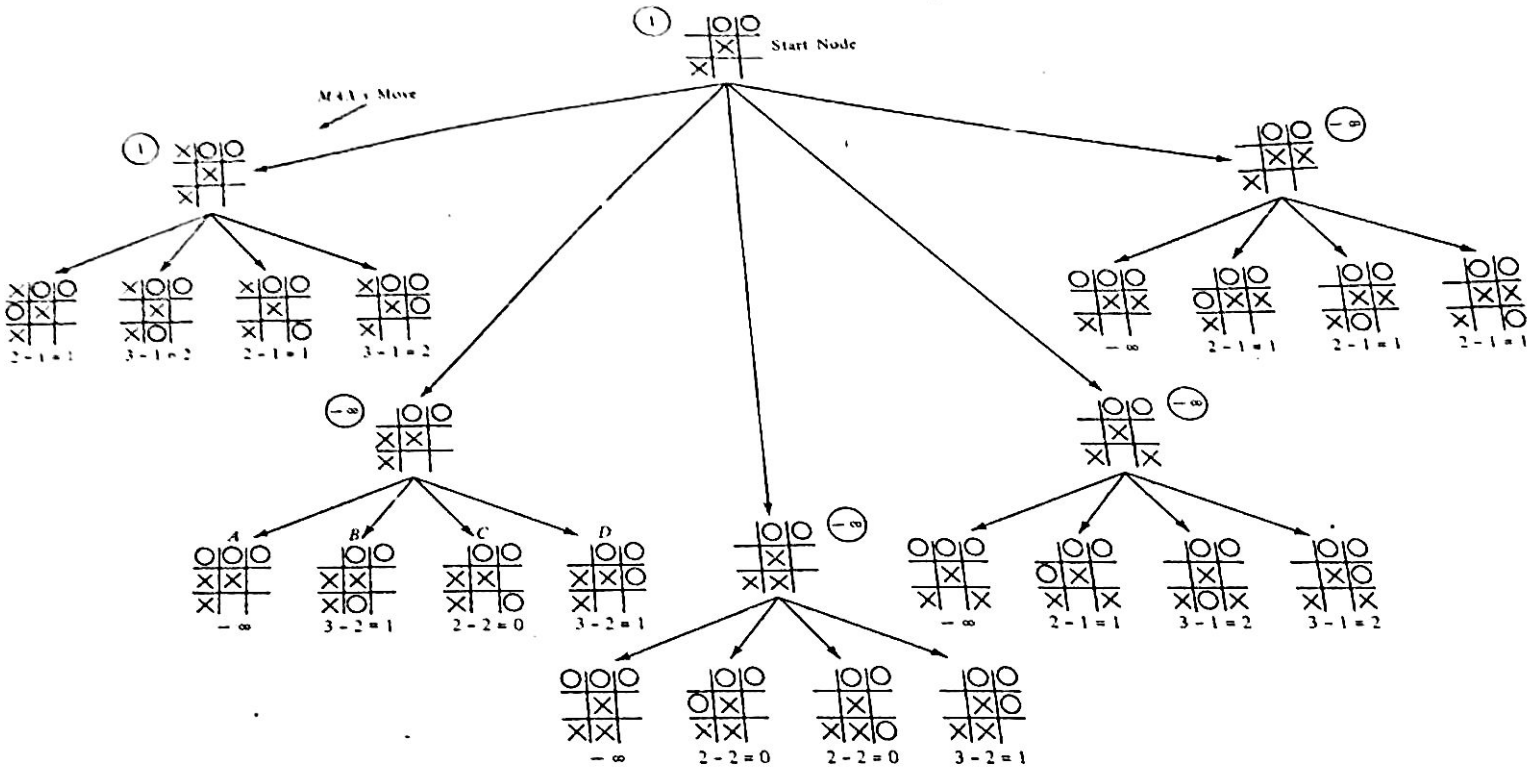
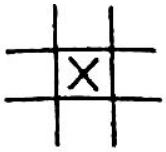
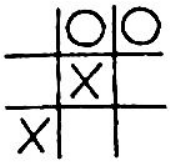


Fig. 3.10 Minimax applied to tic-tac-toe (stage 3).



has the largest backed-up value, it is chosen as the first move. (Coincidentally, this is *MAX*'s best first move.)

Now let us suppose that *MAX* makes this move and *MIN* replies by putting a circle in the square directly above the × (a bad move for *MIN*, who must not be using a good search strategy). Next *MAX* searches to depth 2 below the resulting configuration, yielding the search tree shown in Figure 3.9. There are now two possible "best" moves; suppose *MAX* makes the one indicated. Now *MIN* makes the move that avoids his immediate defeat, yielding



MAX searches again, yielding the tree shown in Figure 3.10. Some of the tip nodes in this tree (for example, the one marked *A*) represent wins for *MIN* and thus have evaluations equal to $-\infty$. When these evaluations are backed up, we see that *MAX*'s best move is also the only one that avoids his immediate defeat. Now *MIN* can see that *MAX* must win on his next move, so *MIN* gracefully resigns.

3.4.2. THE ALPHA-BETA PROCEDURE

The search procedure that we have just described separates completely the processes of search-tree generation and position evaluation. Only after tree generation is completed does position evaluation begin. It happens that this separation results in a grossly inefficient strategy. Remarkable reductions (amounting sometimes to many orders of magnitude) in the amount of search needed (to discover an equally good move) are possible if one performs tip-node evaluations and calculates backed-up values simultaneously with tree generation.

Consider the search tree of Figure 3.10 (the last stage of our tic-tac-toe search). Suppose that a tip node is evaluated as soon as it is generated. Then after the node marked *A* is generated and evaluated, there is no point in generating (and evaluating) nodes *B*, *C*, and *D*; that is, since *MIN* has *A* available and *MIN* could prefer nothing to *A*, we know

immediately that *MIN* will choose *A*. We can then assign *A*'s parent the backed-up value of $-\infty$ and proceed with the search, having saved the search effort of generating and evaluating nodes *B*, *C*, and *D*. (Note that the savings in search effort would have been even greater if we were searching to greater depths; for then none of the descendants of nodes *B*, *C*, and *D* would have to be generated either.) It is important to observe that failing to generate nodes *B*, *C*, and *D* can in no way affect what will turn out to be *MAX*'s best first move.

In this example, the search savings depended on the fact that node *A* represented a win for *MIN*. The same kind of savings can be achieved, however, even when none of the positions in the search tree represents a win for either *MAX* or *MIN*.

Consider the first stage of the tic-tac-toe tree shown in Figure 3.8. We repeat part of this tree in Figure 3.11. Suppose that search had progressed in a depth-first manner and that whenever a tip node is generated, its static evaluation is computed. Also suppose that whenever a position can be given a backed-up value, this value is computed. Now consider the situation occurring at that stage of the depth-first search immediately after node *A* and all of its successors have been generated, but before node *B* is generated. Node *A* is now given the backed-up value of -1. At this point we know that the backed-up value of the start node is bounded from below by -1. Depending on the backed-up values of the other successors of the start node, the final backed-up value of the start node may be greater than -1, but it cannot be less. We call this lower bound an *alpha value* for the start node.

Now let depth-first search proceed until node *B* and its first successor node, *C*, are generated. Node *C* is then given the static value of -1. Now we know that the backed-up value of node *B* is bounded from above by -1. Depending on the static values of the rest of node *B*'s successors, the final backed-up value of node *B* can be less than -1 but it cannot be greater. We call this upper bound on node *B* a *beta value*. We note at this point, therefore, that the final backed-up value of node *B* can never exceed the alpha value of the start node, and therefore we can discontinue search below node *B*. We are guaranteed that node *B* will not turn out to be preferable to node *A*.

This reduction in search effort was achieved by keeping track of bounds on backed-up values. In general, as successors of a node are given

backed-up values, the bounds on backed-up values can be revised. But we note that:

(a) The alpha values of *MAX* nodes (including the start node) can never decrease, and

(b) the beta values of *MIN* nodes can never increase.

Because of these constraints we can state the following rules for discontinuing the search:

(1) Search can be discontinued below any *MIN* node having a beta value less than or equal to the alpha value of any of its *MAX* node ancestors. The final backed-up value of this *MIN* node can then be set to its beta value. This value may not be the same as that obtained by full minimax search, but its use results in selecting the same best move.

(2) Search can be discontinued below any *MAX* node having an alpha value greater than or equal to the beta value of any of its *MIN* node ancestors. The final backed-up value of this *MAX* node can then be set to its alpha value.

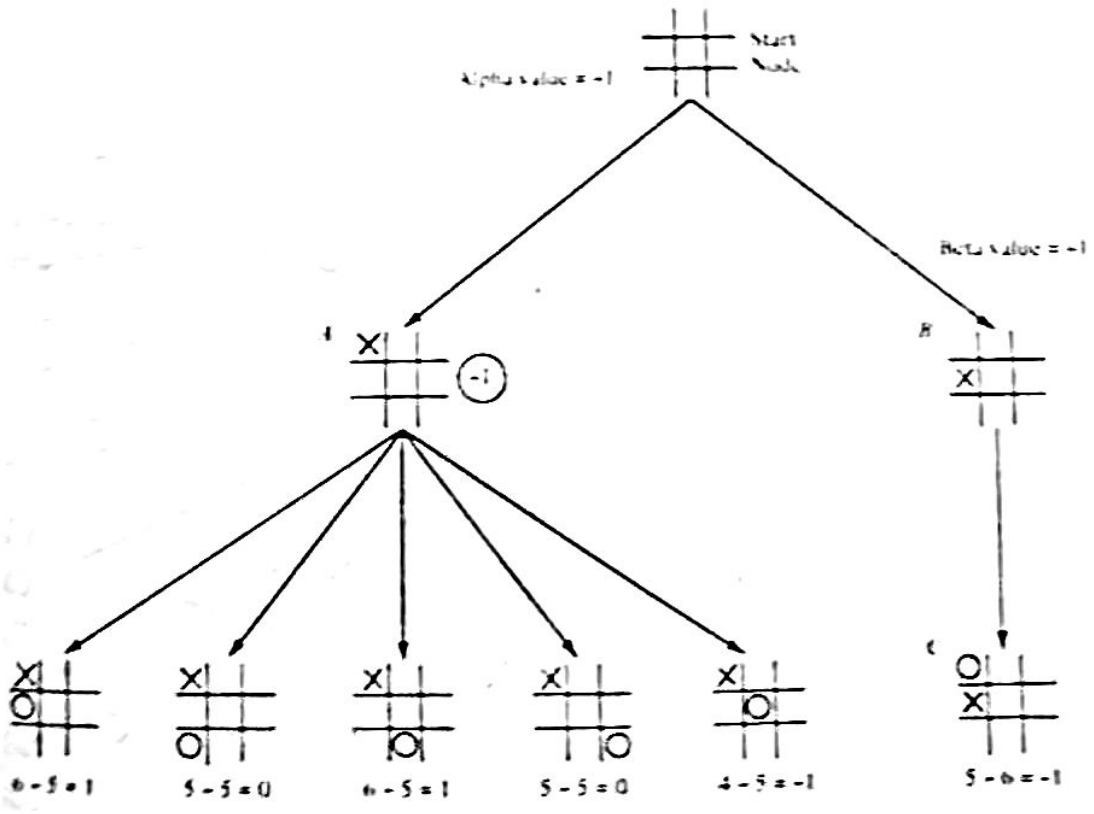


Fig. 3.11 Part of the first stage tic-tac-toe tree.