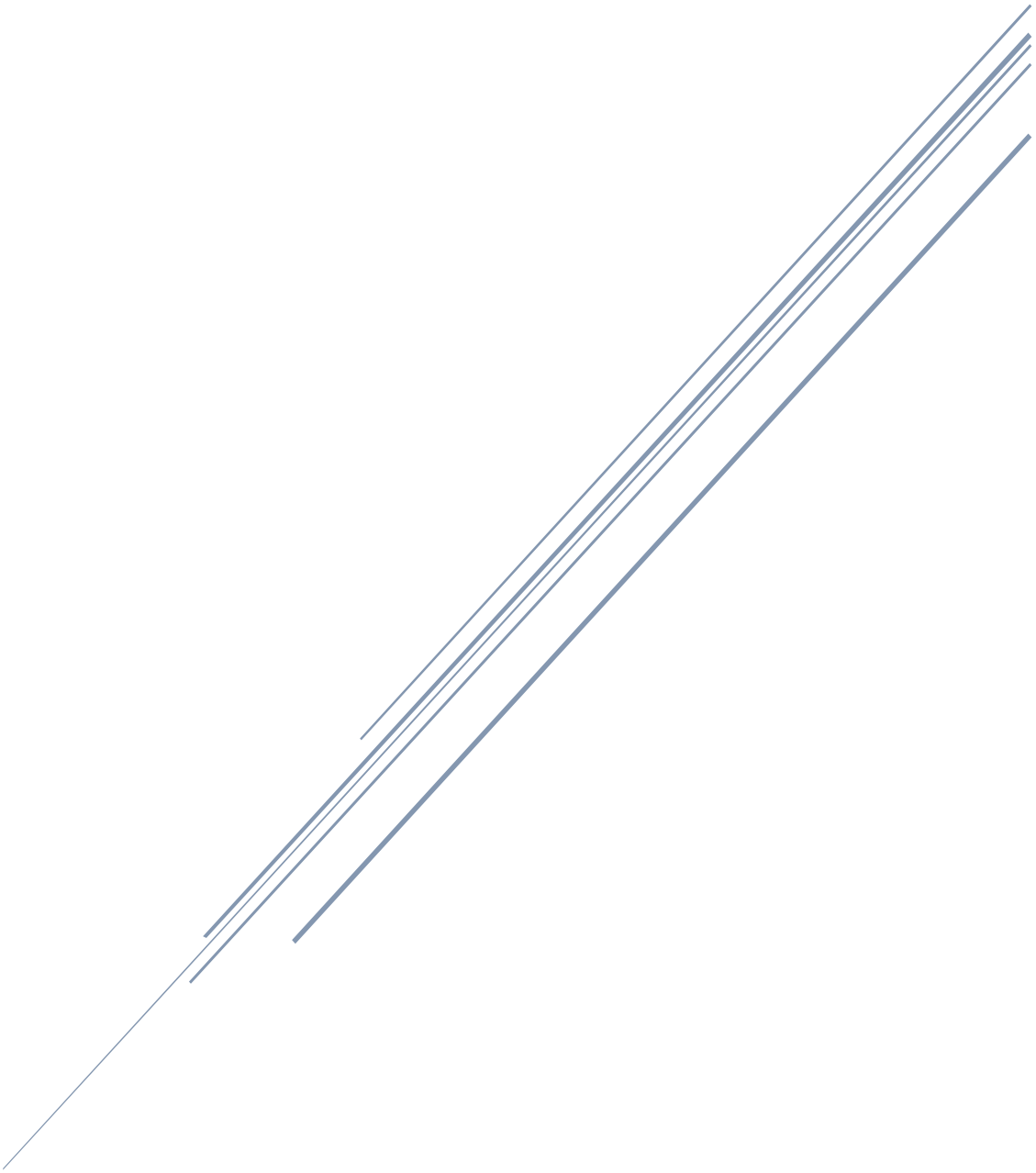
CHAPTER 4

Adversarial Search



4.1. Introduction

4.1.1 Games vs. search problems

In the previous chapter, we discussed search algorithms applicable to single agents aiming to find a solution, often expressed as a sequence of actions. However, there are situations where multiple agents search for a solution within the same search space, a scenario commonly encountered in game playing.

This environment involving more than one agent is referred to as a multi-agent environment, where each agent acts as an opponent to others, engaging in competitive play. In such settings, each agent must consider not only their own actions but also the effects of their rivals' actions on their performance.

A search in which two or more players with conflicting goals attempt to explore the same search space for a solution, is called **adversarial search**, commonly recognized as a **game**.

The objective of adversarial search is to develop intelligent algorithms capable of selecting the most advantageous course of action under conflicting circumstances. To determine the optimal move or action, adversarial search algorithms frequently navigate through a game tree, representing all possible game states and their transitions.

4.1.2 Types of games in AI

In the realm of AI, various games are commonly used as a means of conducting research, developing algorithms, and performing game-playing tests. Here are some game categories within the field of AI:

- ✧ **Perfect information games:** In games with perfect information, all agents have complete access to the game board, and they can fully observe each other's moves. Classic examples of such games include Chess, Checkers, Tic-tac-toe, Go, Otello, and more.
- ✧ **Imperfect information games:** Games involving imperfect information are those where the agents do not possess full knowledge of the game's state and remain unaware of the ongoing developments. This kind of games includes games like Battleship, Bridge, Poker, Scrabble, and similar examples.
- ✧ **Deterministic games:** these games adhere to strict patterns and rules, devoid of any element of chance. Examples of such games include Chess, Checkers, Go, Battleship.
- ✧ **Non-deterministic games:** Characterized by their reliance on an unpredictable element, often involving dice or cards to introduce chance or luck. In these games, the outcomes of actions are uniquely generated and not predetermined. Such games are also known as stochastic games, and examples include Backgammon, Monopoly, Poker, and various others.

4.1.3 Zero-sum games

In game theory, a zero-sum game includes a scenario where the gain or loss of one player directly counterbalances the loss or gain of another player. In simple terms, the net sum of winnings and losses always equals zero within the game. This concept characterizes adversarial searches with a single victor.

4.2 Optimal decisions in games

In this chapter, we are mostly interested in games with the following characteristics:

- ✧ Two-player: The games involve only two players.
- ✧ Turn-taking: Players take turns to make moves.

- ✧ Zero-sum: The games have a structure where one player's gain directly offsets the other player's loss.
- ✧ Perfect information, deterministic, and fully observable: Both players have a complete and certain knowledge of the game state, with no hidden elements or elements of chance.
- ✧ A small number of possible actions.
- ✧ Precise, formal rules.

4.2.1 Formulation of the problem

In Adversarial search, the formulation of the problem involves specifying the rules of the game, identifying the players, defining the available actions, and determining the outcomes of the game. Since we are considering games with two players, they are typically referred to as MAX and MIN. The game begins with one of them making the initial move, after which they take turns until the game reaches its conclusion. At the end of the game, points are awarded to the winning player, and penalties are assessed to the loser.

The following elements are often included in the formulation:

- ✧ **Initial state:** denoted as S_0, it represents the starting configuration of the game before any moves have been taken by the players. For example, in Chess, the initial state is the starting locations of the Chess pieces on the board.
- ✧ **Successor function:** denoted as *successorsFn(s, p)*, it returns a list of (*action*, *successor*) pairs, each indicating a legal move by player *p* from state *s* and its resulting state.
- ✧ **Terminal test:** denoted as *terminalTest(s)*, it is used to determine when the game is over. States where the game has ended are called terminal states.
- ✧ **Utility function:** also referred to as the objective function or payoff function and denoted as *utility(s)*, it defines the final numeric value for a game that ends in terminal state, *s*, for a player, *p*. Higher values are considered to be good for the MAX player and bad for the MIN player, which is the basis of their respective player names.

Game tree: A game tree is a type of tree structure where the nodes represent game states, and the edges of the tree correspond to the moves made by the players. The game tree starts with the initial state and branches out to show the possible successor states resulting from player moves.

Example: Tic-tac-toe game tree

Consider the Tic-tac-toe game (noughts(O) and crosses(X) game), shown in Figure 4.1.

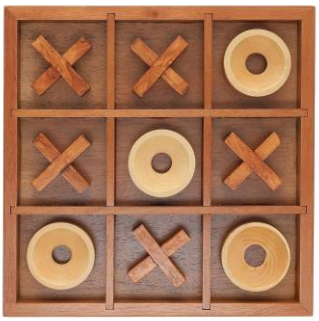


Figure 4.1 Illustration of the Tic-tac-toe game

Problem formulation:

- ***States:*** Represent the different configurations of the game board.
- ***Initial state:*** Depicts the starting state with an empty grid before any player's marks (X or O).
- ***Successor function*:** In Tic-tac-toe, there are typically nine possible actions, each corresponding to placing a player's symbol (either X or O) in one of the nine available cells of the 3x3 game board.
- ***Terminal test:*** In Tic-tac-toe, a terminal state occurs when one player has won by forming a line of three of their symbols (X or O), or when the entire board is filled, resulting in a draw.
- ***Utility function:*** In Tic-tac-toe, the outcomes are usually represented as a win (+1) for the player who won, a loss (-1) for the player who lost, and a draw (0) when the game ends in a tie with no winner.

Game tree:

Figure 4.2 shows a portion of the game tree for the Tic-tac-toe game. The following are some of the game tree's most important features:

- There are two players: MAX(X) and MIN(O).
- The game tree is constructed by the MAX player, who seeks to win the game.
- Players take turns one by one, starting with the MAX player.
- The MAX player makes a move by placing the symbol ´X´, and the MIN player makes a move by placing the symbol ´O´.

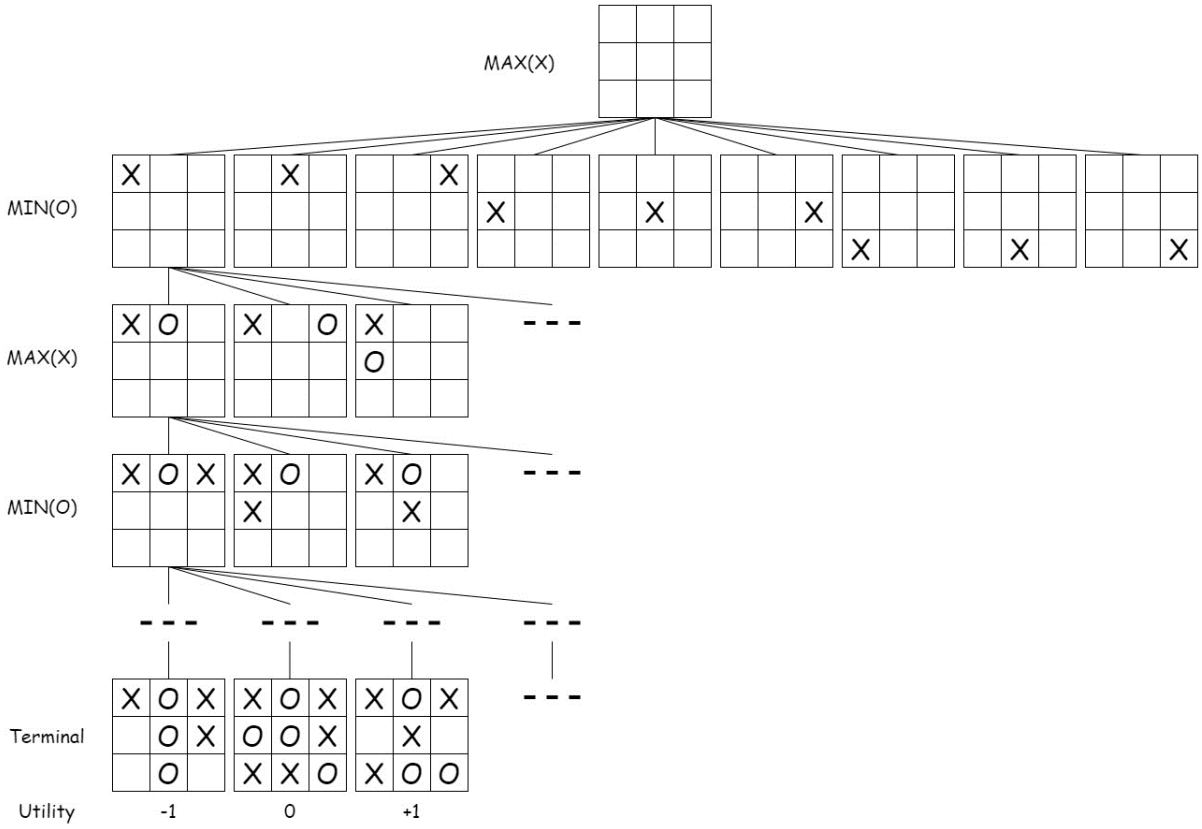


Figure 4.2 Partial tic-tac-toe game tree

- From the initial state, the MAX player has nine possible moves.
- In the game tree, each level is referred to as a *ply*, with alternating Max and MIN layers. The game

progresses through turns until it reaches a leaf node, which happens when one player achieves a sequence of three in a row or when all squares on the board are occupied.

- ✧ The game can result in one of the three outcomes: MAX wins, MIN wins, or the game ends in a tie. The number displayed on every leaf node represents the utility value of the terminal state from the perspective of the MAX player, who is constructing the game tree to make the best move for winning the game.

4.2.2 MINIMAX algorithm

In a single-agent search problem, the optimal solution is a sequence of actions to reach a goal state, typically a winning terminal state. In adversarial search problems, MIN's moves are crucial in determining the outcome. So, MAX must plan a strategy that covers MAX's moves from the initial state, responses to potential MIN actions, and subsequent reactions to MIN's responses.

Within a game tree, the optimal strategy can be determined from the minimax value of each node, denoted as ***MINIMAX_VALUE (s, p)***. This value indicates how favorable the node represented by state s is for player p, under the assumption that both players make optimal moves from that point to the end of the game. Notably, the minimax value of a terminal node is directly equivalent to its utility.

Moreover, when making choices, MAX seeks to move to states with the maximum value, aiming to maximize the game tree's outcome, while MIN prefers states with the minimum value, aiming to minimize the outcome. In simple terms (Eq 4.1):

$$MINIMAX\_VALUE(s,p) = \begin{cases} UTILITY(s) & if\ s\ is\ a\ terminal\ state \\ max_{s' \in successors(s)}\ MINIMAX\_VALUE(s', MIN) & if\ p\ is\ MAX \\ min_{s' \in successors(s)}\ MINIMAX\_VALUE(s', MAX) & if\ p\ is\ MIN \end{cases} \quad (4.1)$$

- ✧ The MINIMAX algorithm determines the optimal decision from the current state.
- ✧ This algorithm uses a recursive process to calculate minimax values for each successor state, following the equations defining the minimax strategy.
- ✧ The recursion proceeds until it reaches terminal nodes, and then the minimax values propagate upward through the tree as the recursion unfolds.

The pseudocode for the MINIMAX algorithm is provided in Figure 4.3.

Input:
- *state*: The current state in the game
- *player*: The current player

Output: an action

Function MINIMAX (state, player)
Begin
  v <- MINIMAX_VALUE (*state, player*)

```
   return the action related to the value v
End
```

Input:
- *state*: A state in the game
- *player*: MAX or MIN

Output: a value

```
Function  MINIMAX_VALUE (state, player)
Begin

   if (terminalTest(state)) then
      return utility(state)

   if (player is MAX) then
      v <-  −∞
      for each (action, successor) in successorsFn(state, player) do
         v <- maximum (v, MINIMAX_VALUE (successor, -player))      //Here, -player=MIN
   else
      v <-  +∞
      for each (action, successor) in successorsFn(state, player) do
         v <- minimum (v, MINIMAX_VALUE (successor, -player))      //Here, -player=MAX

   return v
End
```

Figure 4.3 MINIMAX algorithm

Example 1: Let's apply the MINIMAX algorithm to the game tree depicted in Figure 4.4. The execution of the MINIMAX algorithm is illustrated in Figure 4.5.

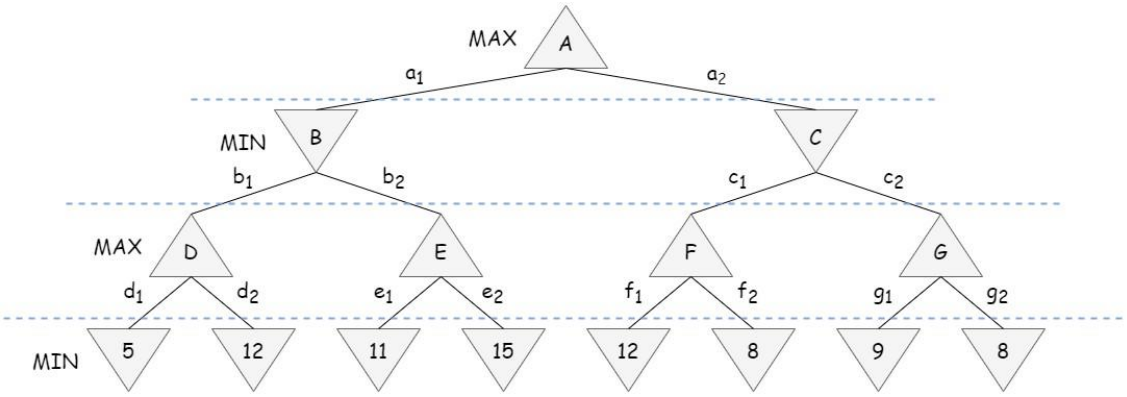


Figure 4.4 Game tree of Example 1

- ✧ The algorithm begins by calling the function MINIMAX(A, MAX), followed by the function MINIMAX_VALUE(A, MAX).
- ✧ Initially, the algorithm descends into the nodes located at the bottom level through the recursive

call of the MINIMAX_VALUE function until it reaches the terminal nodes. At these terminal nodes, it applies the Utility function to determine their values.

- ✧ In the provided example, the MAX node labeled as D has two successor states with values of 5 and 12, resulting in a minimax value of 12. Similarly, the remaining three MAX nodes E, F, and G, yield minimax values of 15, 12, and 9, respectively, as shown in Figure 4.5.
- ✧ The MIN node marked as B has two successor states with values of 12 and 15, resulting in a minimax value of 12. Likewise, the other MIN node, C, obtains a minimax value of 9, as illustrated in Figure 4.5.
- ✧ The root node, which is a MAX node, has successor states with minimax values of 12 and 9. Consequently, the root node itself has a minimax value of 12. At the root, we can determine the optimal minimax decision: selecting action $\mathbf{a_1}$ is the best choice for MAX.

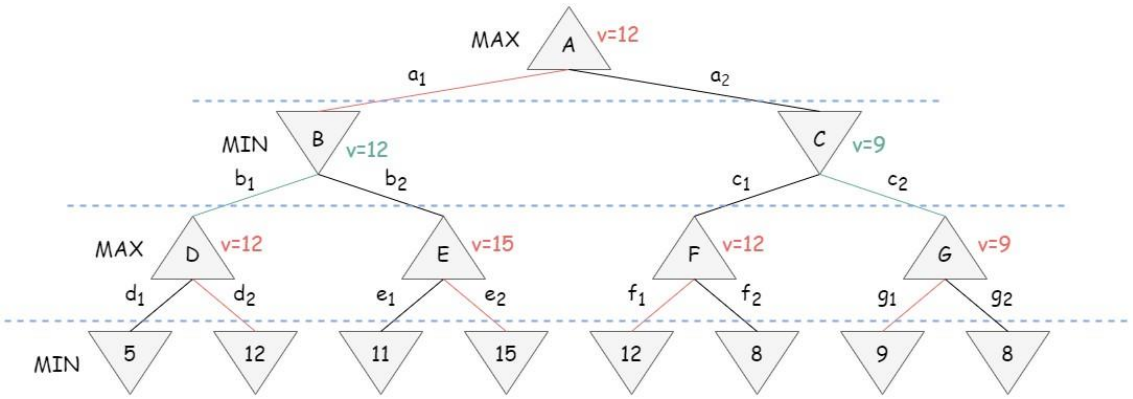


Figure 4.5 Execution of the MINIMAX algorithm in Example 1

Example 2: Tic-tac-toe game

In the context of the Tic-tac-toe game.

- ✧ The initial game board is as follows:

O	O	X
	X	
O	X	

- ✧ Suppose the MAX player is playing with the cross (X).
- ✧ The starting player is the MAX player, aiming to win the game.
- ✧ Assign labels of +1, -1, or 0 to terminal nodes, indicating victories for MAX (+1), MIN (-1) or a draw (0).

Solution: The MINIMAX game tree is given in Figure 4.6.

Conclusion: The MAX player will choose the last action since it has the best value, but it will not result in a win; instead, it will result in a draw.

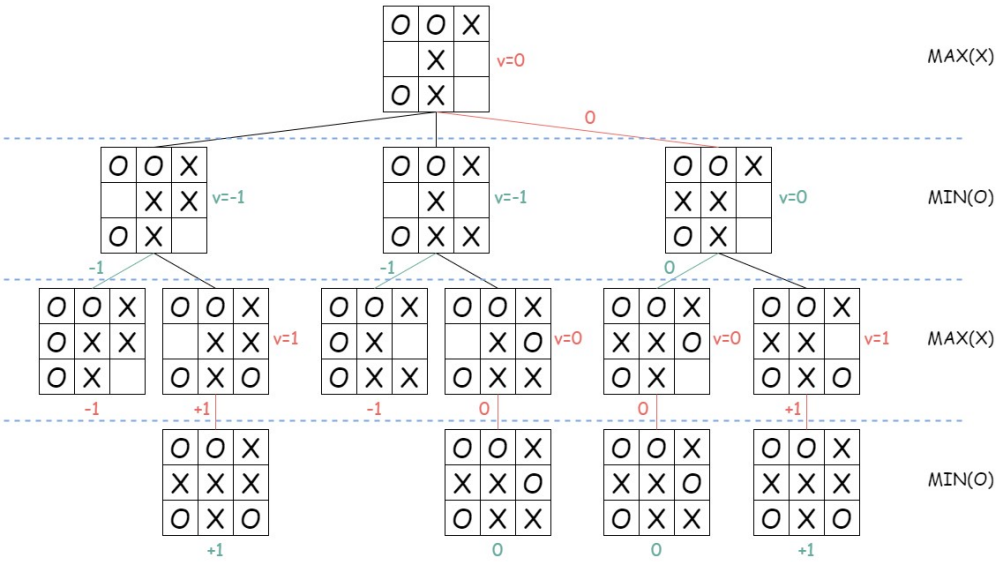


Figure 4.6 MINIMAX game tree for the Tic-tac-toe game

Properties:

The MINIMAX algorithm performs a complete depth-first exploration of the game tree. If the game has a maximum depth of *m*, and there are *b* legal moves available at each node, the time complexity of the minimax algorithm can be represented as $O(b^m)$. The space complexity is $O(bm)$ for an algorithm that generates actions all actions simultaneously, or $O(m)$ for an algorithm that generates actions one at a time.

4.2.3 Depth-limited MINIMAX algorithm

For practical real-world games, the time cost is often excessively high. Therefore, in the majority of cases, fully expanding the game tree to its terminal nodes is not feasible. Instead, a look-ahead strategy involving *d* moves in employed. The state space is explored up to a depth of *d*, and each terminal node within this subset is assigned a value based on a **heuristic evaluation function**.

In terms of implementation, the utility function is replaced by the heuristic evaluation function, and the cut-off test is determined by whether the state is a terminal state or if the search depth reaches the specified limit. This results in the following MINIMAX_VALUE function (Eq 4.2):

$$MINIMAX\_VALUE(s,d,p) = \begin{cases} HEURISTIC\_EVAL(s) & if\ s\ is\ a\ terminal\ state\ or\ d\ is\ 0 \\ max_{s' \in successors(s)}\ MINIMAX\_VALUE(s', d-1, MIN) & if\ p\ is\ MAX \\ min_{s' \in successors(s)}\ MINIMAX\_VALUE(s', d-1, MAX) & if\ p\ is\ MIN \end{cases} \quad (4.2)$$

This function enables us to find the best possible value within the constraints of the specified search depth *d*. Subsequently, these values are propagated back towards the root node. The value propagated back represents the heuristic value of the best state that can be reached from that node. The depth-limited MINIMAX algorithm is given in Figure 4.8.

Input:

- *state*: The current state in the game

- *depthLimit*: The maximum depth of the game tree
- *player*: The current player

Output: an action

```
Function DEPTH_LIMITED_MINIMAX (state, depthLimit, player)
Begin
  v <- MINIMAX_VALUE(state, depthLimit, player)
  return the action related to the value v
End
```

Input:
- *state*: A state in the game
- *depth*: Initially equal to depthLimit and will decrease until it reaches the value of 0
- *player*: MAX or MIN

Output: a value

```
Function MINIMAX_VALUE (state, depth, player)
Begin

  if (depth = 0 or terminalTest(state)) then      // Cut-off test
    return heuristicEval(state)

  if (player is MAX) then
    v <- −∞
    for each (action, successor) in successorsFn(state, player) do
      v <- maximum (v, MINIMAX_VALUE (successor, depth-1, -player))      //Here, -player=MIN
  else
    v <- +∞
    for each (action, successor) in successorsFn(state, player) do
      v <- minimum (v, MINIMAX_VALUE (successor, depth-1, -player))      //Here, -player=MAX

  return v
End
```

Figure 4.7 Depth-limited MINIMAX algorithm

Example 3:

In the context of the Tic-tac-toe game, we can employ the following heuristic evaluation function (see Figure 4.9):

- ✧ Suppose the MAX player is playing with the cross (X).
- ✧ The starting player is the MAX player, striving to win the game.
- ✧ HEURISTIC-EVAL(s) = X(s) – O(s)

  X(s) represents the total of the MAX player's possible winning lines.

  O(s) represents the total of the opponent's possible winning lines.

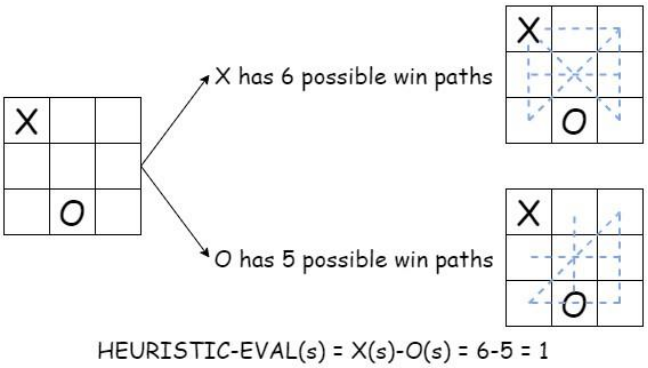


Figure 4.8 Heuristic evaluation function for the Tic-tac-toe game

Using this heuristic evaluation function and a depth limit of 2, we generate the game tree shown in Figure 4.10.

Conclusion: The MAX player will choose the first action since it has the best value.

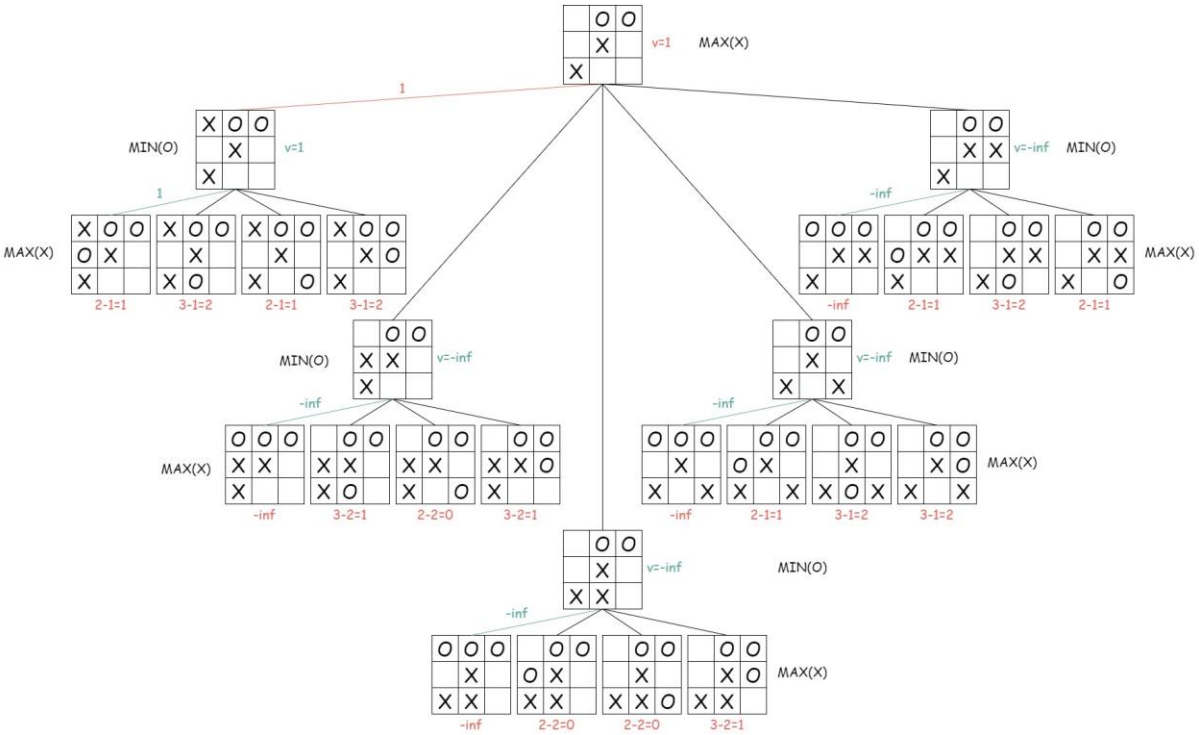


Figure 4.9 Depth-limited MINIMAX game tree for the Tic-tac-toe game

4.2.4 MINIMAX Alpha-Beta Pruning algorithm

To enhance the optimization of the MINIMAX algorithm, the Alpha-Beta Pruning technique is employed. It significantly reduces computational time, enabling a faster search and more in-depth exploration of the game tree. This technique prunes branches of the game tree that do not need to be searched when a better move is already known to exist.

Now, let's delve into the intuition behind this technique. Consider the two-ply game tree from Figure 4.11 as an example. Suppose the two unevaluated successors of node *D* have values x and y. In this case, the value of the root node *A* can be determined as follows:

$$MINIMAX(A) = \max(\min(15,6,1), \min(3,11,7), \min(2,x,y))$$
$$= \max(1,3,\min(2,x,y))$$
$$= \max(1,3,z) \qquad where\ z = \min(2,x,y) \leq 2$$
$$= 3$$

In other words, the value of the root node A and the MINIMAX decision are not influenced by the values of the pruned leaves x and y.

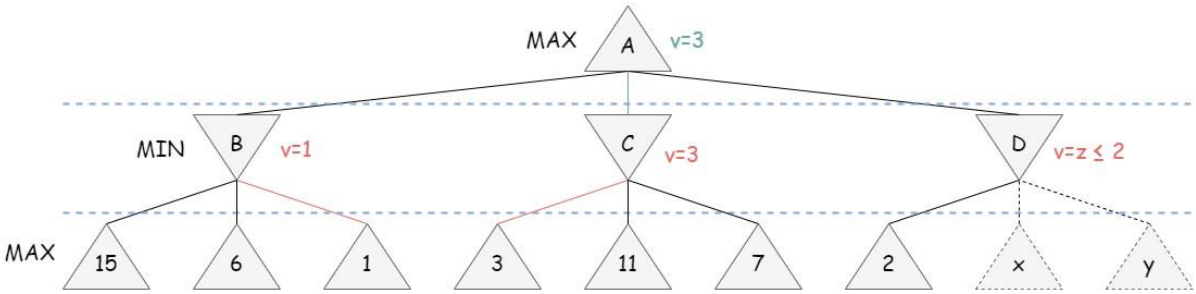


Figure 4.10 Intuition behind the alpha-beta pruning technique

Let's define the parameters α and β:

- ✧ α represents the best value (i.e., the highest value) that the maximizing player (MAX) can guarantee at the current level in the game tree. In other words, α is the lower bound on the possible values for MAX. The initial value of α is $-\infty$.
- ✧ β represents the best value (i.e., the lowest value) that the minimizing player (MIN) can guarantee at the current level in the game tree. In other words, β is the upper bound on the possible values for MIN. The initial value of β is $+\infty$.
- ✧ Each node in the game tree keeps track of its α and β values.
- ✧ The value of α can be updated when it's the turn of the MAX player
- ✧ Similarly, the value of β can be updated when it's the turn of the MIN player.
- ✧ The Alpha–Beta search updates the values of α and β as it progresses and prunes the remaining branches at a node (i.e., stops the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.

The explanation of alpha pruning and beta pruning are shown in Figure 4.12 and Figure 4.13, respectively. The complete algorithm is given in Figure 4.14.

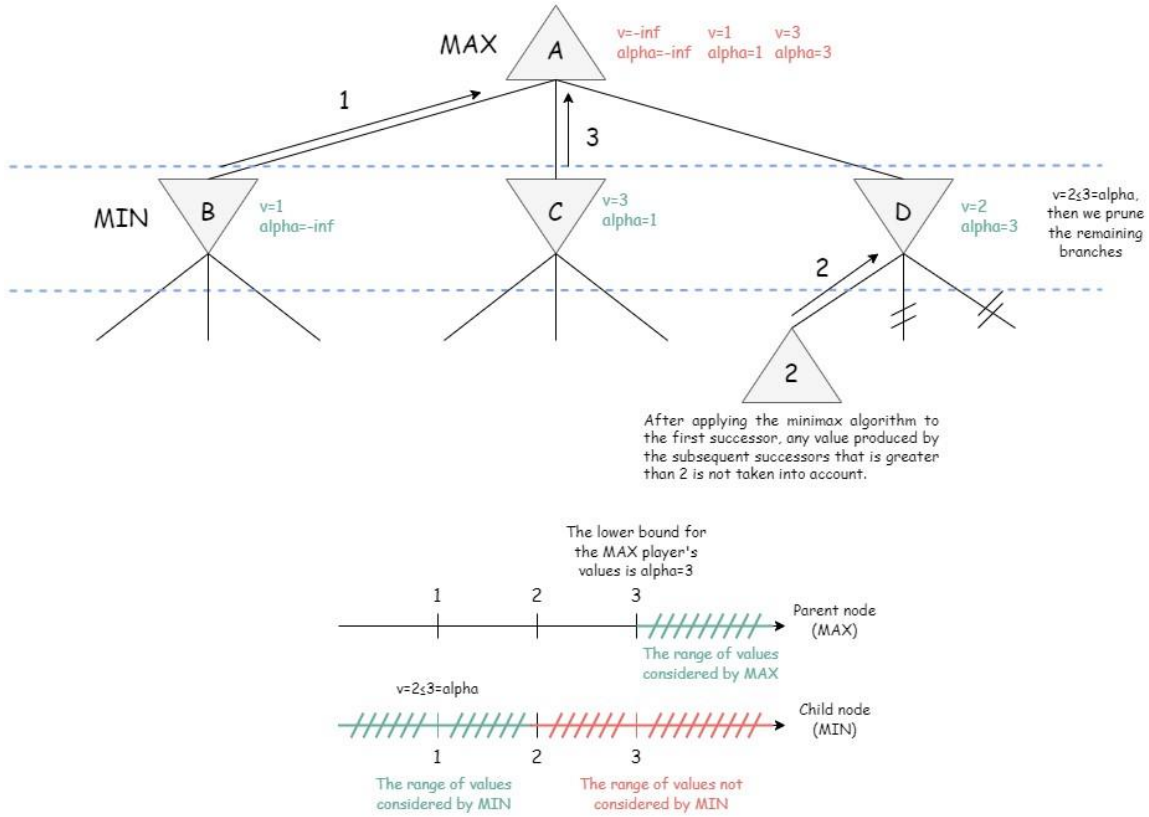


Figure 4.11 Explanation of Alpha pruning

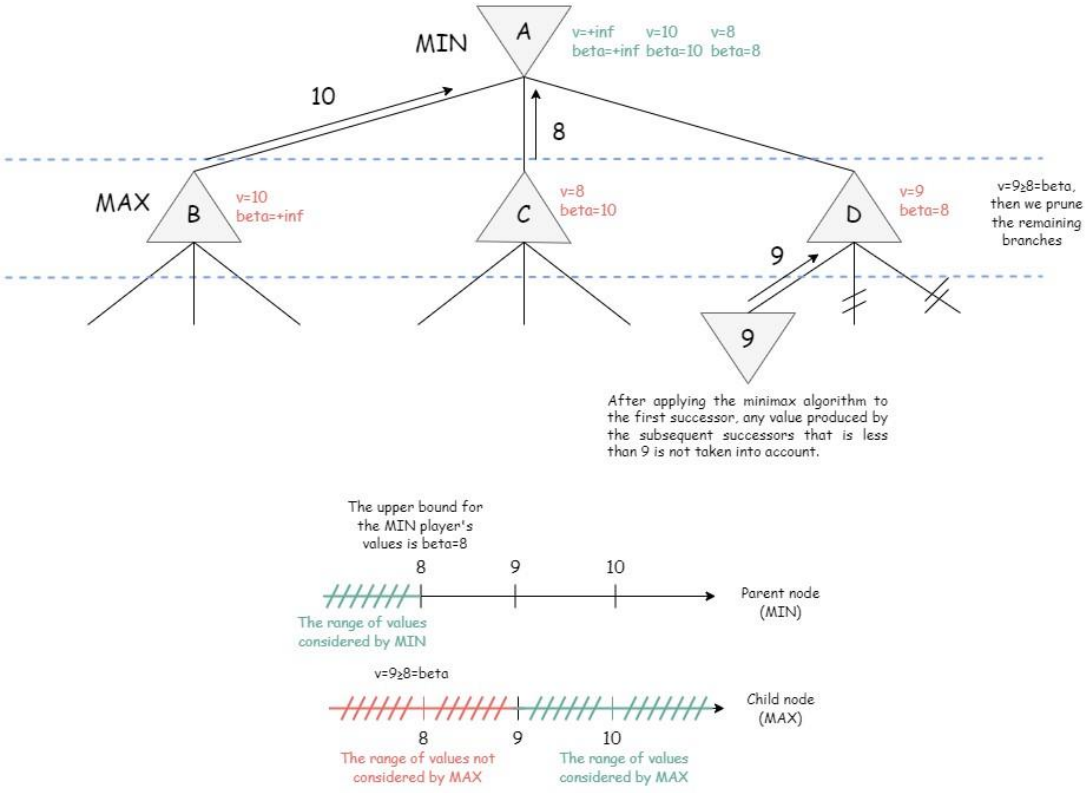


Figure 4.12 Explanation of Beta pruning

Input:
- *state*: The current state in the game
- *depthLimit*: The maximum depth of the game tree
- *player*: The current player

Output: an action

```
Function MINIMAX_ALPHA_BETA_PRUNING (state, depthLimit, player)
Begin
  v <- MINIMAX_VALUE(state, depthLimit, −∞, +∞, player)
  return the action related to the value v
End
```

Input:
- *state*: A state in the game
- *depth*: Initially equal to depthLimit and will decrease until it reaches the value of 0
- *player*: MAX or MIN

Output: a value

```
Function MINIMAX_VALUE (state, depth, α, β, player)
Begin

  if (depth = 0 or terminalTest(state)) then
    return heuristicEval(state)

  if (player is MAX) then
    v <- −∞
    for each (action, successor) in successorsFn(state, player) do
      v <- maximum (v, MINIMAX_VALUE (successor, depth-1, α, β, MIN))
      if (v ≥ β) then
        return v     // β cut-off
      α <- maximum (α, v)
  else
    v <- +∞
    for each (action, successor) in successorsFn(state, player) do
      v <- minimum (v, MINIMAX_VALUE (successor, depth-1, α, β, MAX))
      if (v ≤ α) then
        return v     // α cut-off
      β <- minimum (β, v);

  return v
End
```

Figure 4.13 MINIMAX alpha-beta pruning algorithm

Example 4: Now, Let's apply the MINIMAX alpha-beta pruning algorithm to the game tree depicted in Figure 4.15. The execution of the MINIMAX alpha-beta pruning algorithm is shown in Figure 4.16.

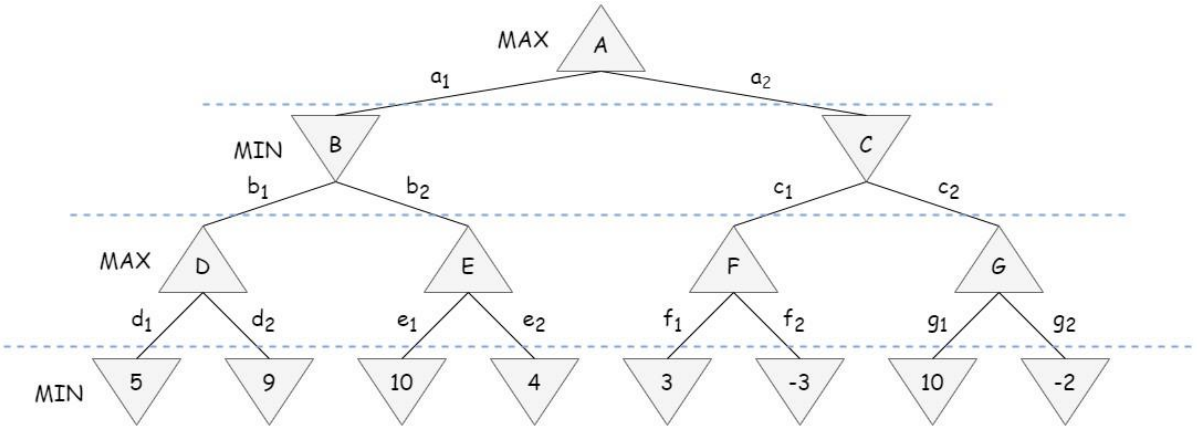


Figure 4.14 Game tree of Example 4

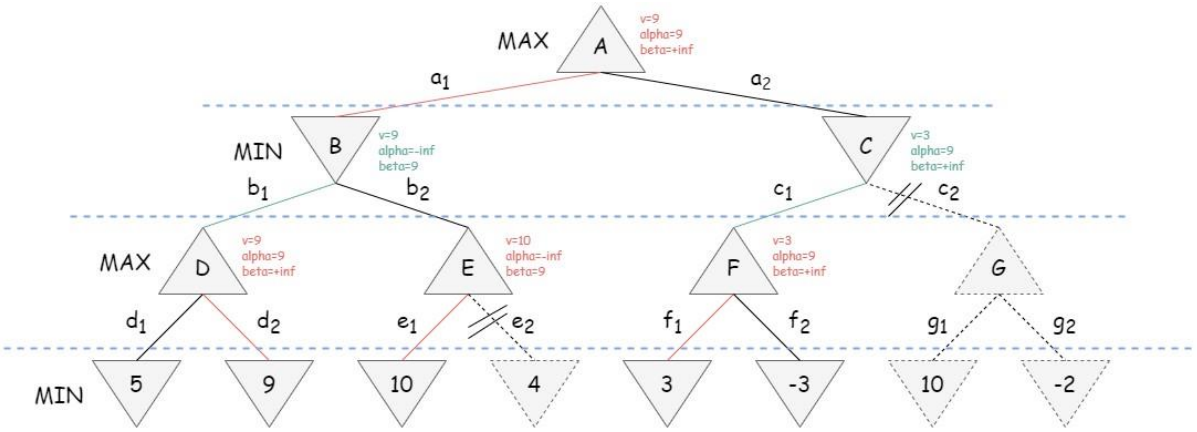


Figure 4.15 Execution of the MINIMAX algorithm in Example 4

Example 5: Tic-tac-toe game

In the context of the Tic-tac-toe game.

- The initial game board is as follows:

O	O	X
	X	
O	X	

- Suppose the MAX player is playing with the cross (X).
- The starting player is the MAX player, aiming to win the game.
- Assign labels of +1, -1, or 0 to terminal nodes, indicating victories for MAX (+1), MIN (-1) or a draw (0).

Solution: The MINIMAX Alpha-Beta Pruning game tree is given in Figure 4.16.

Conclusion: The MAX player will choose the first action since it has the best value, but it will not result in a win; instead, it will result in a draw.

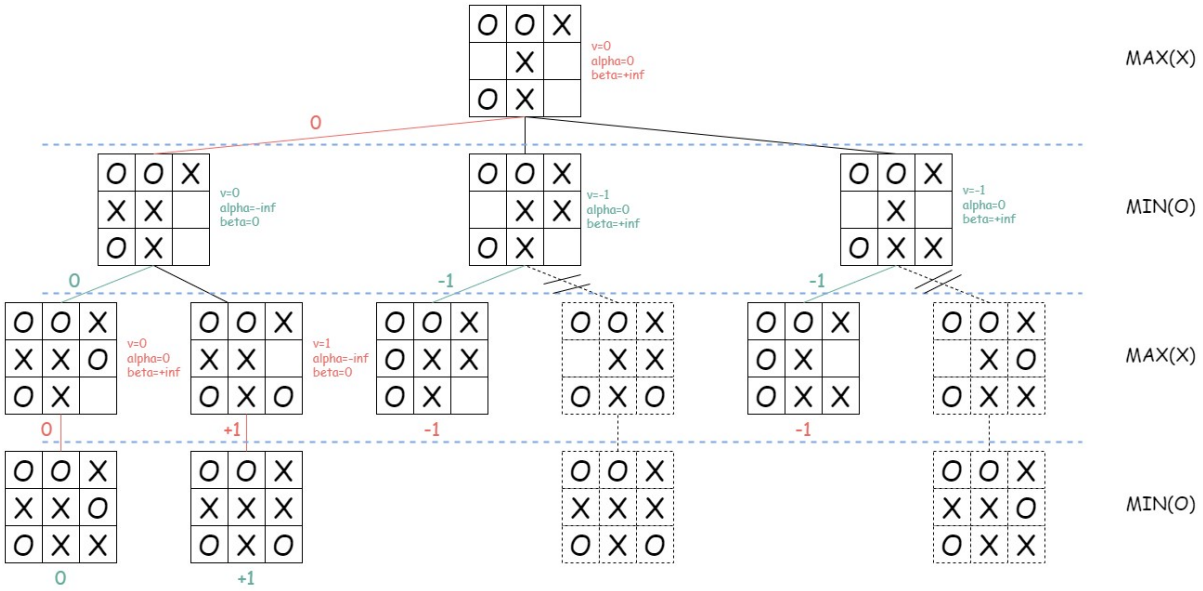


Figure 4.16 MINIMAX Alpha-Beta Pruning game tree for the Tic-tac-toe game

4.2.5 NEGAMAX algorithm

To simplify the implementation of the MINIMAX algorithm, we employ the NEGAMAX search, a variant of the MINIMAX algorithm. This algorithm leverages the zero-sum nature of the two-player games. The core concept behind this algorithm is to exploit the mathematical property that *min(a, b)* is equivalent to *-max(-b, -a)*. In essence, the value of a game position for player A is the negation of its value for player B. Therefore, the player whose turn it is looks for a move that maximizes the negation of the resulting value. This principle remains consistent whether it's player A or B taking their turn. This approach allows us the use of a single procedure to evaluate both positions, streamlining the coding process compared to the traditional MINIMAX algorithm, where player A maximizes the value of their move, and player B minimizes it. This leads to the following NEGAMAX_VALUE function (Eq 4.3):

$$NEGAMAX\_VALUE(s,d,p)$$

$$= \begin{cases} HEURISTIC\_EVAL(s) & if\ (s\ is\ terminal\ or\ d\ is\ 0)\ and\ p\ is\ MAX \\ -HEURISTIC\_EVAL(s) & if\ (s\ is\ terminal\ or\ d\ is\ 0)\ and\ p\ is\ MIN \\ max_{s' \in successors(s)} -NEGAMAX\_VALUE(s', d-1, -p) & otherwise \end{cases} \quad (4.3)$$

The depth-limited NEGAMAX algorithm is given in Figure 4.17.

Input:
- *state*: The current state in the game
- *depthLimit*: The maximum depth of the game tree
- *player*: The current player

Output: an action

Function NEGAMAX_DEPTH_LIMITED (state, depthLimit, player)

```
Begin
   v <- NEGAMAX_VALUE(state, depthLimit, player)
   return the action related to the value v
End
```

Input:
- *state*: A state in the game
- *depth*: Initially equal to depthLimit and will decrease until it reaches the value of 0
- *player*: MAX or MIN

Output: a value

```
Function NEGAMAX_VALUE(state, depth, player)
Begin

   if (depth = 0 or terminalTest(state)) then
      if (player = MIN) then
         return -heuristicEval(state)
      else
         return heuristicEval(state)

   v <- −∞
   for each (action, successor) in successorsFn(state, player) do
      v <- maximum (v, -NEGAMAX_VALUE(successor, depth-1, -player))

   return v
End
```

Figure 4.17 Depth-limited NEGAMAX algorithm

Example 5: Let's apply the NEGAMAX algorithm to the game tree depicted in Figure 4.18. The execution of the NEGAMAX algorithm is illustrated in Figure 4.19.

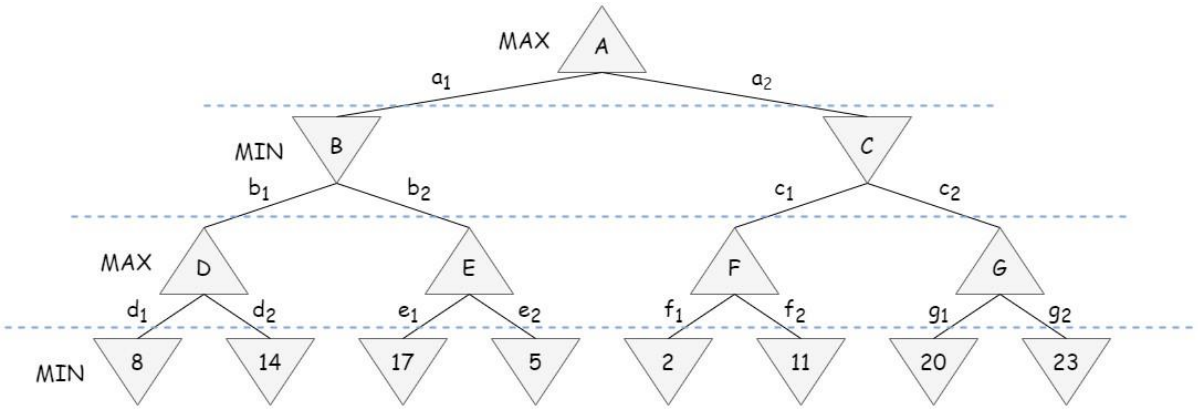


Figure 4.18 Game tree of Example 5.

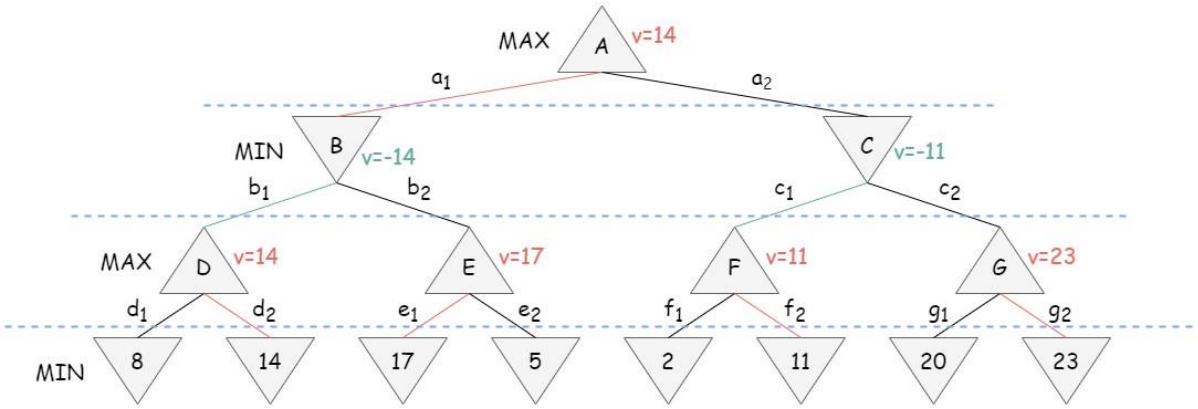


Figure 4.19 Execution of the NEGAMAX algorithm in Example 5.

4.2.6 NEGAMAX Alpha-Beta Pruning algorithm

The Alpha-Beta pruning technique can also be applied to the NEGAMAX algorithm. The enhanced version of the NEGAMAX algorithm with Alpha-Beta property is shown in Figure 4.20.

Input:
- *state*: The current state in the game
- *depthLimit*: The maximum depth of the game tree
- *player*: The current player

Output: an action

Function NEGAMAX(state, depthLimit, player)
Begin
  v <- NEGAMAX_VALUE(state, depthLimit, $-\infty$, $+\infty$, player)
  return the action related to the value *v*
End

Input:
- *state*: A state in the game
- *depth*: Initially equal to depthLimit and will decrease until it reaches the value of 0
- *player*: MAX or MIN

Output: a value
Function NEGAMAX_VALUE(state, depth, α, β, player)
Begin

  if (depth = 0 or *terminalTest(state)*) **then**
    if (player = MIN) **then**
      return *-heuristicEval(state)*
    else
      return *heuristicEval(state)*

  v <- $-\infty$

```
  for each (action, successor) in successorsFn(state, player) do
    v <- maximum (v, -NEGAMAX_VALUE(successor, depth-1, −β, −α, -player))
    α <- maximum(α, v)
    if (α ≥ β) then
      return v

  return v
End
```

Figure 4.20 NEGAMAX alpha-beta pruning algorithm

Example 6: Let's apply the NEGAMAX alpha-beta pruning algorithm to the game tree depicted in Figure 4.21. The execution of the NEGAMAX alpha-beta pruning algorithm is illustrated in Figure 4.22.

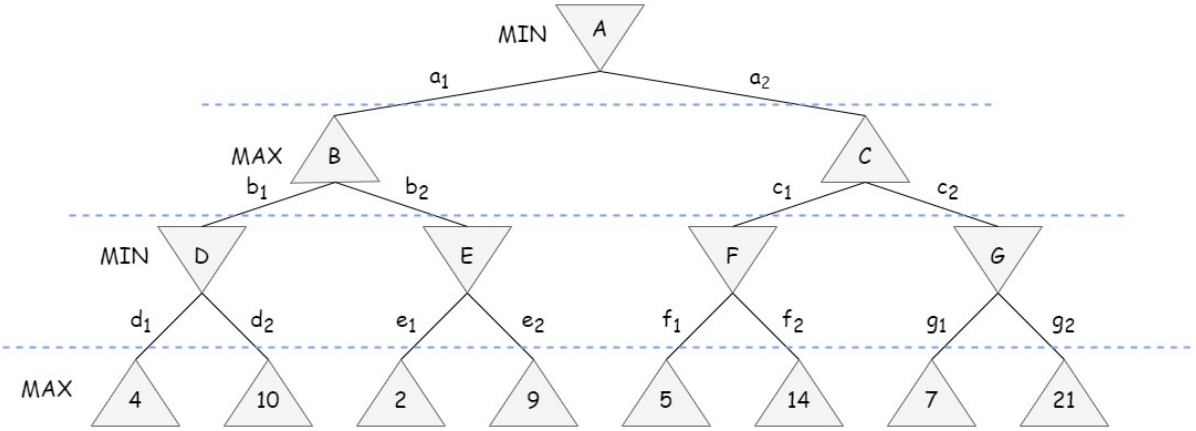


Figure 4.21 Game tree of Example 6.

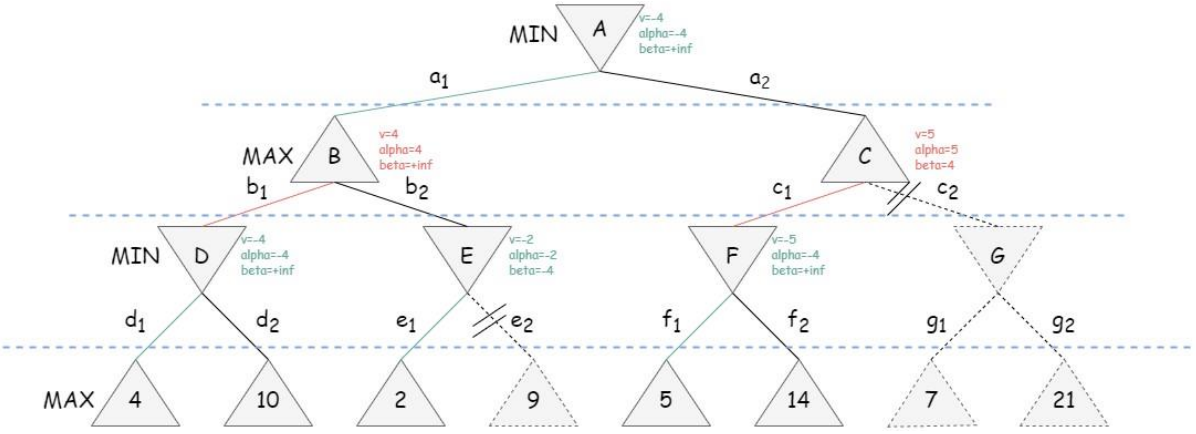


Figure 4.22 Execution of the NEGAMAX Alpha-Beta pruning algorithm in Example 6.

Exercise 1: Fan-Tan Game

Let's consider the two-player game of Fan-Tan. We have two matchbox compartments, one containing a match, and the other containing 3 matches. Each player can take as many matches as they want from one compartment at a time. The player who takes the last match wins. Give the MINIMAX tree for this game, assuming that the starting player is the MAX player and the initial state is shown in Figure 4.23:

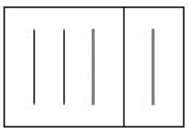


Figure 4.23 Initial state for the Fan-Tan game

Solution: The MINIMAX game tree is given in Figure 4.24.
Conclusion: The MAX player will choose the last action since it's the winning action (the one with the best value).

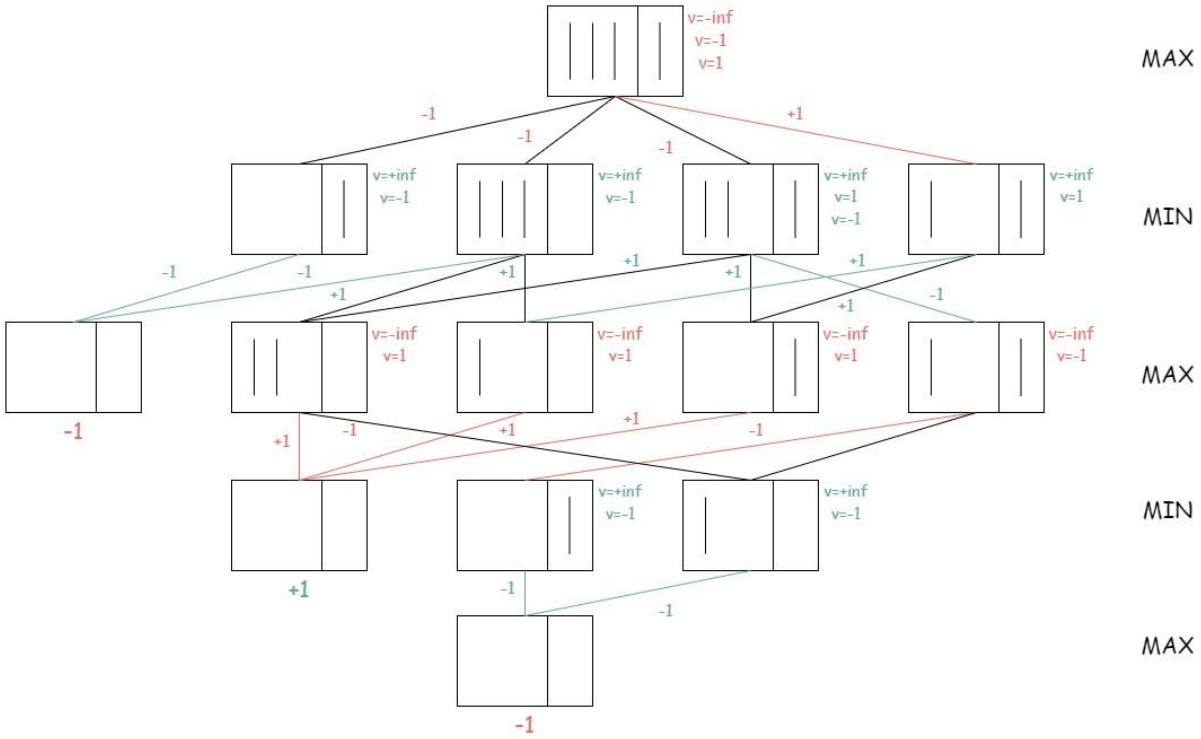


Figure 4.24 Execution of the MINIMAX algorithm for Fan-Tan game

Exercise 2: Grundy's Game

A pile of tokens is placed on a table, with two opponents taking turns. In each turn, a player must split the pile of tokens into two separate piles, ensuring that both piles have different sizes. For example, when there are 6 tokens, they can be divided into pairs like 5 and 1 or 4 and 2, but not equally into 3 and 3. The game continues until one of the players is unable to make a valid move, at which point that player loses.
Give the NEGAMAX tree for this game starting with a pile of 7 tokens, assuming that the initial player is the MAX player. Can the MAX player win the game? If so, which action should be taken by this player?
Solution: The NEGAMAX game tree is given in Figure 4.25.
Conclusion: Regardless of the action played by the MAX player, he will lose the game, as all actions have a value of -1.

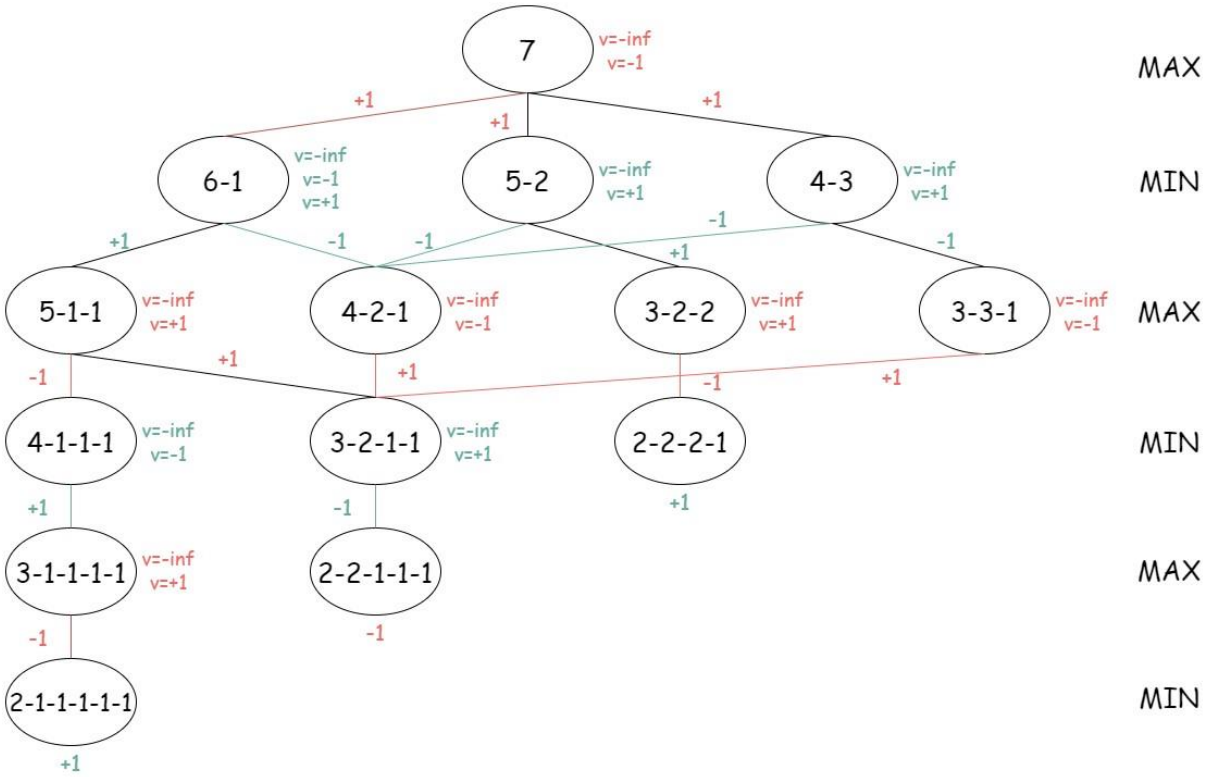


Figure 4.25 Execution of the NEGAMAX algorithm for Grundy's game

Exercise 3:

Apply the MINIMAX algorithm with alpha-beta pruning to the game tree shown in Figure 4.26. Which action should be chosen by the MAX player?

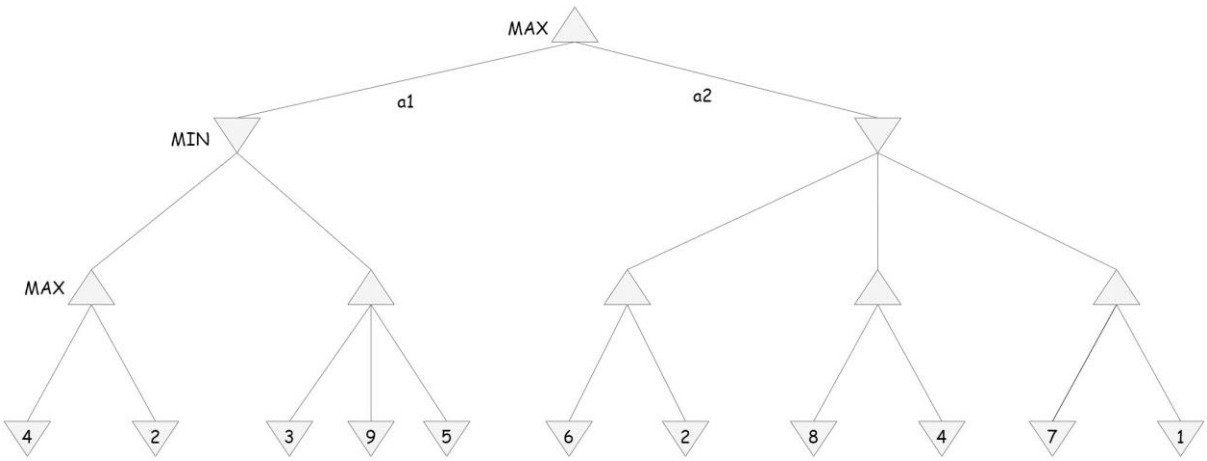


Figure 4.26

Solution: The MINIMAX game tree with alpha-beta pruning is given in Figure 4.27.

Conclusion: Action chosen by the MAX player is: a2

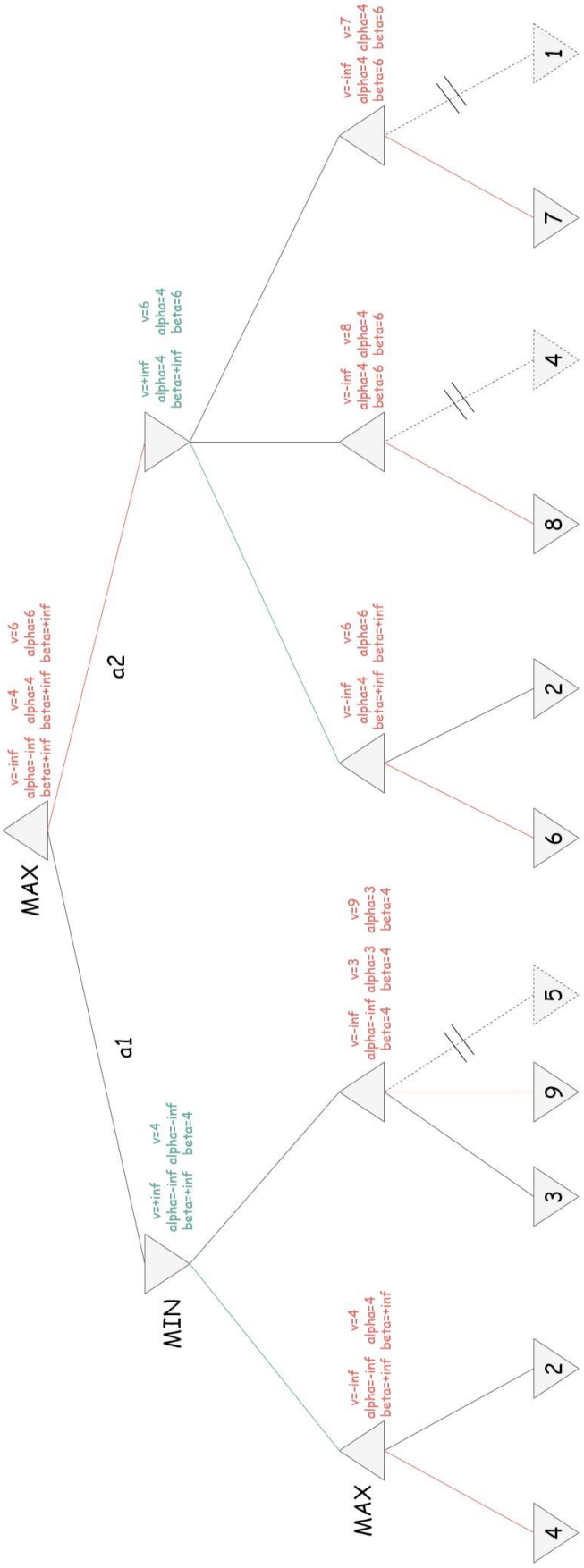

MAX
v=-inf alpha=-inf beta=+inf
v=4 alpha=-inf beta=+inf
v=6 alpha=6 beta=+inf
a1
a2
MIN
v=+inf alpha=-inf beta=+inf
v=4 alpha=-inf beta=4
v=+inf alpha=4 beta=+inf
v=6 alpha=4 beta=6
MAX
v=-inf alpha=-inf beta=+inf
v=4 alpha=4 beta=+inf
v=-inf alpha=-inf beta=4
v=3 alpha=3 beta=4
v=9 alpha=3 beta=4
v=-inf alpha=4 beta=+inf
v=6 alpha=4 beta=+inf
v=-inf alpha=4 beta=6
v=8 alpha=4 beta=6
v=-inf alpha=4 beta=6
v=7 alpha=4 beta=6
4
2
5
9
3
6
2
8
4
7
1

Figure 4.27

Exercise 4: Domineering Game

In the game of Domineering, two players take turns placing their dominoes on a grid. One player can only place tiles vertically (the green tiles), and the other can only place them horizontally (the red tiles). The game continues until one player is unable to make a move, at which point they lose. Give the MINIMAX Alpha-Beta Pruning tree for this game, assuming that the starting player is the MIN player, playing with the green tiles and the initial state is shown in Figure 4.28:

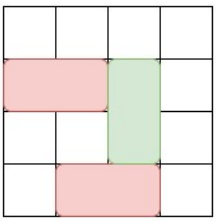


Figure 4.28 Initial state for the Domineering game

Solution: The MINIMAX Alpha-Beta Pruning game tree is given in Figure 4.29.

Conclusion: Regardless of the action played by the MIN player, he will lose the game, as all actions have a value of 1.

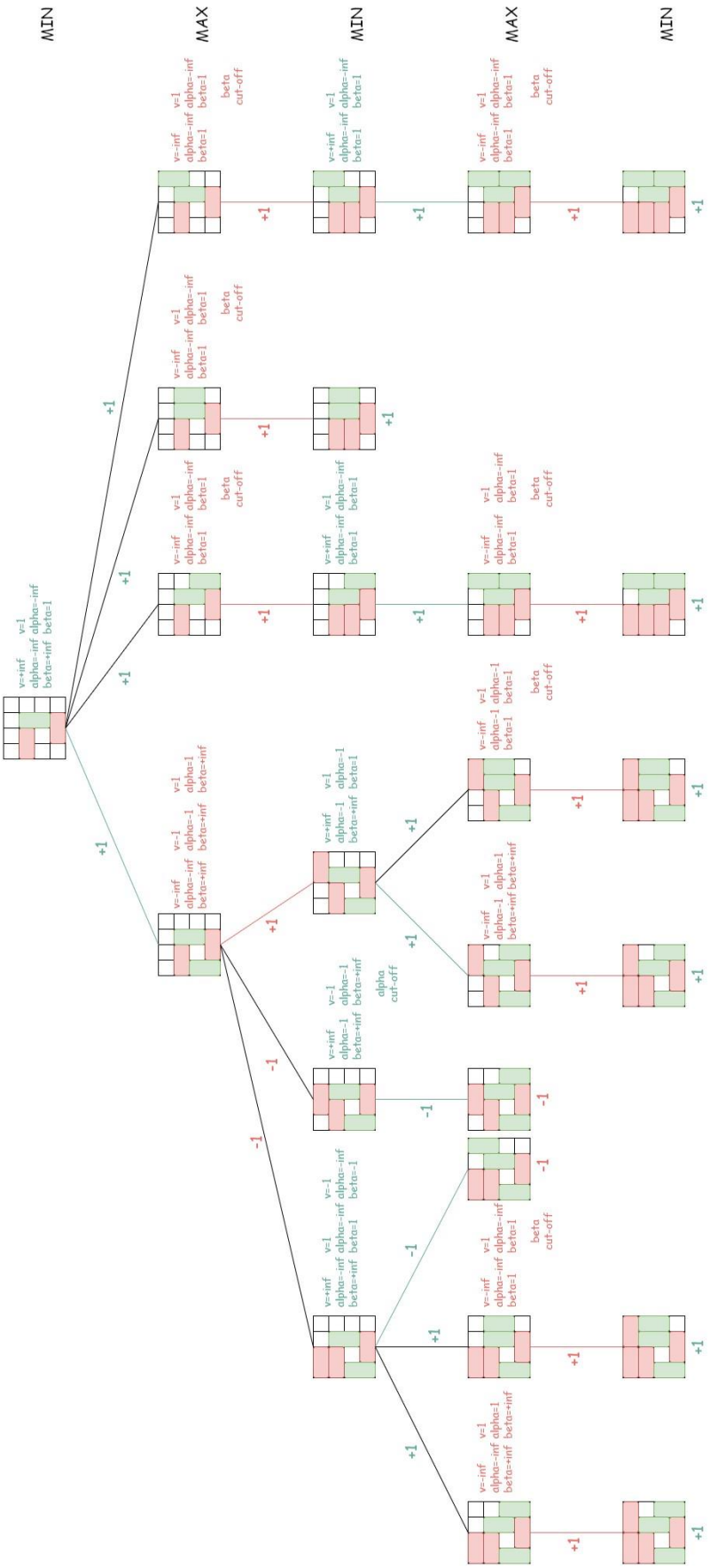


Figure 4.29 Execution of the NEGAMAX Alpha-Beta pruning algorithm for Domineering