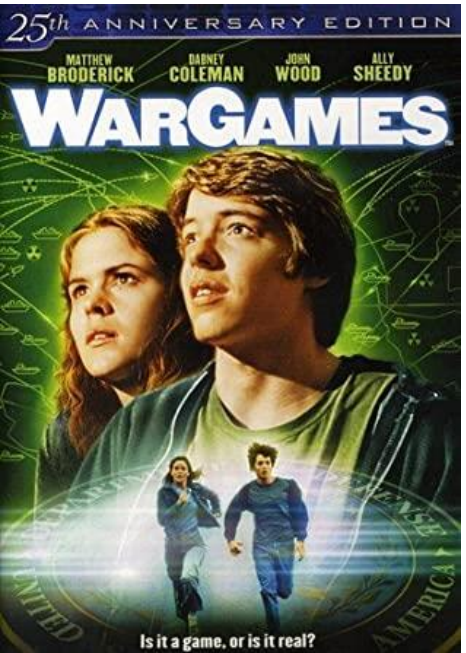
Παίγνια 2 Ατόμων

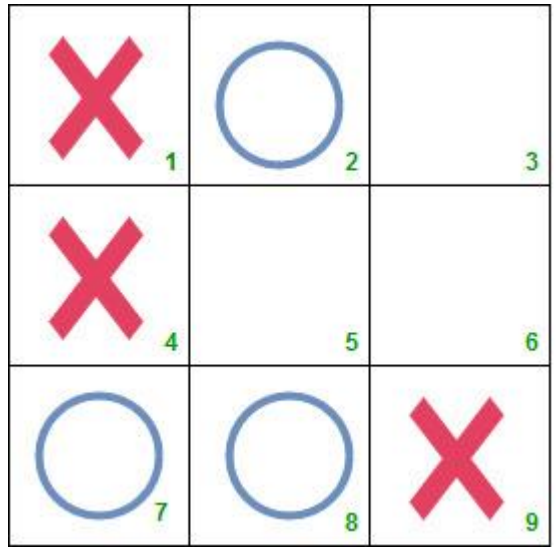
Αλγόριθμοι MiniMax & Alphabeta



Παίγνια 2 Ατόμων

- Είναι προβλήματα αναζήτησης στα οποία οι κινήσεις του πράκτορα εναλλάσσονται με αυτές του αντίπαλου
- Οι αλγόριθμοι θα πρέπει να λαμβάνουν υπόψη και τις πιθανές κινήσεις του αντιπάλου και να βρίσκουν εκείνες τις κινήσεις που θα βελτιστοποιήσουν τη πιθανότητα να κερδηθεί το παιχνίδι, ακόμη και αν ο αντίπαλος επιλέγει πάντα τις βέλτιστες κινήσεις για εκείνον.

Tic Tac Toe



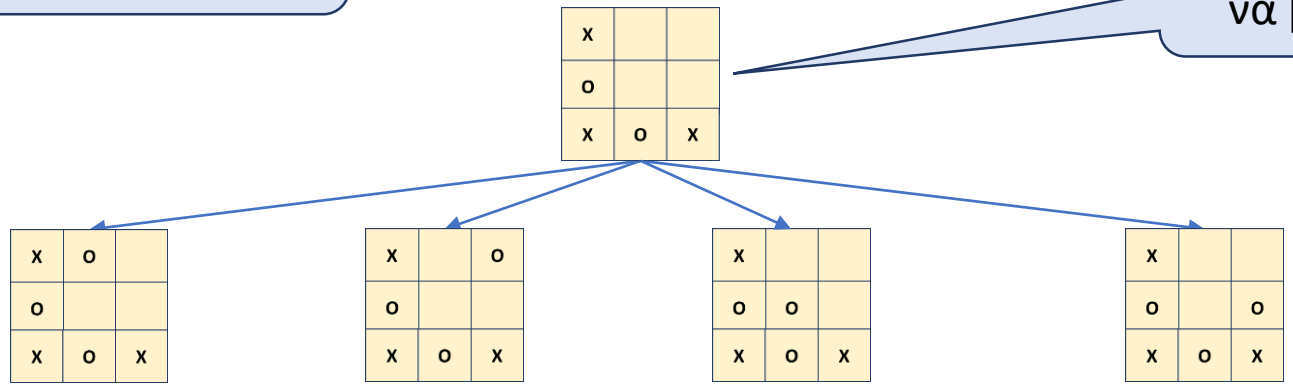
- Ταμπλό 3 * 3
- Ο κάθε παίκτης τοποθετεί το σύμβολο του (Χ ή Ο) σε ένα από τα κενά κελιά
- Οι δύο παίκτες εναλάσσονται
- Το παιχνίδι τελειώνει όταν
  - Ένας παίκτης τοποθετεί τρία σύμβολα του στην ίδια γραμμή ή στήλη ή διαγώνιο – οπότε κερδίζει το παιχνίδι
  - Σε όλα τα κελιά έχει τοποθετηθεί σύμβολο και δεν υπάρχουν τρία ίδια σύμβολο σε καμία γραμμή ή στήλη ή διαγώνιο – οπότε το παιχνίδι λήγει με ισοπαλία

Minimax

Ο υπολογιστής προσπαθεί να μεγιστοποιήσει το όφελος του (max)

MAX

Έστω ότι είναι η σειρά του Η/Υ να βάλει το σύμβολο του (Ο)



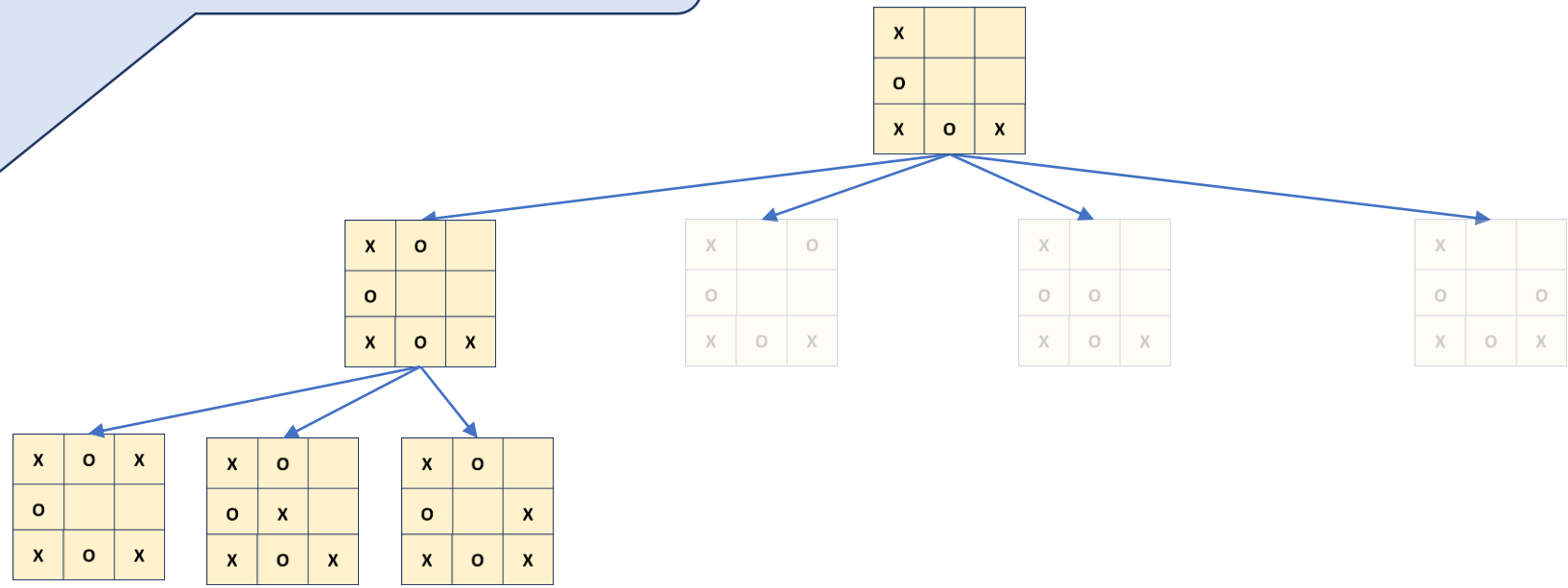
Υπάρχουν 4 κινήσεις και πρέπει να διαλέξει τη καλύτερη

Minimax

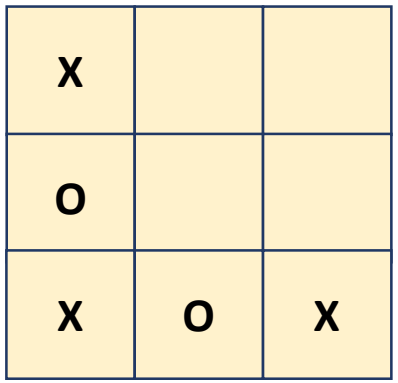
Ο αντίπαλος προσπαθεί να ελαχιστοποιήσει το όφελος του Η/Υ (min)

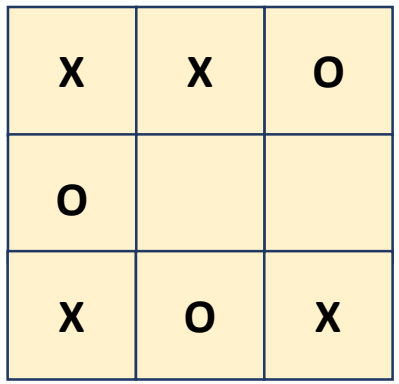
MAX

MIN



Για τη πρώτη κίνηση του βρίσκω όλες τις πιθανές απαντήσεις του αντιπάλου



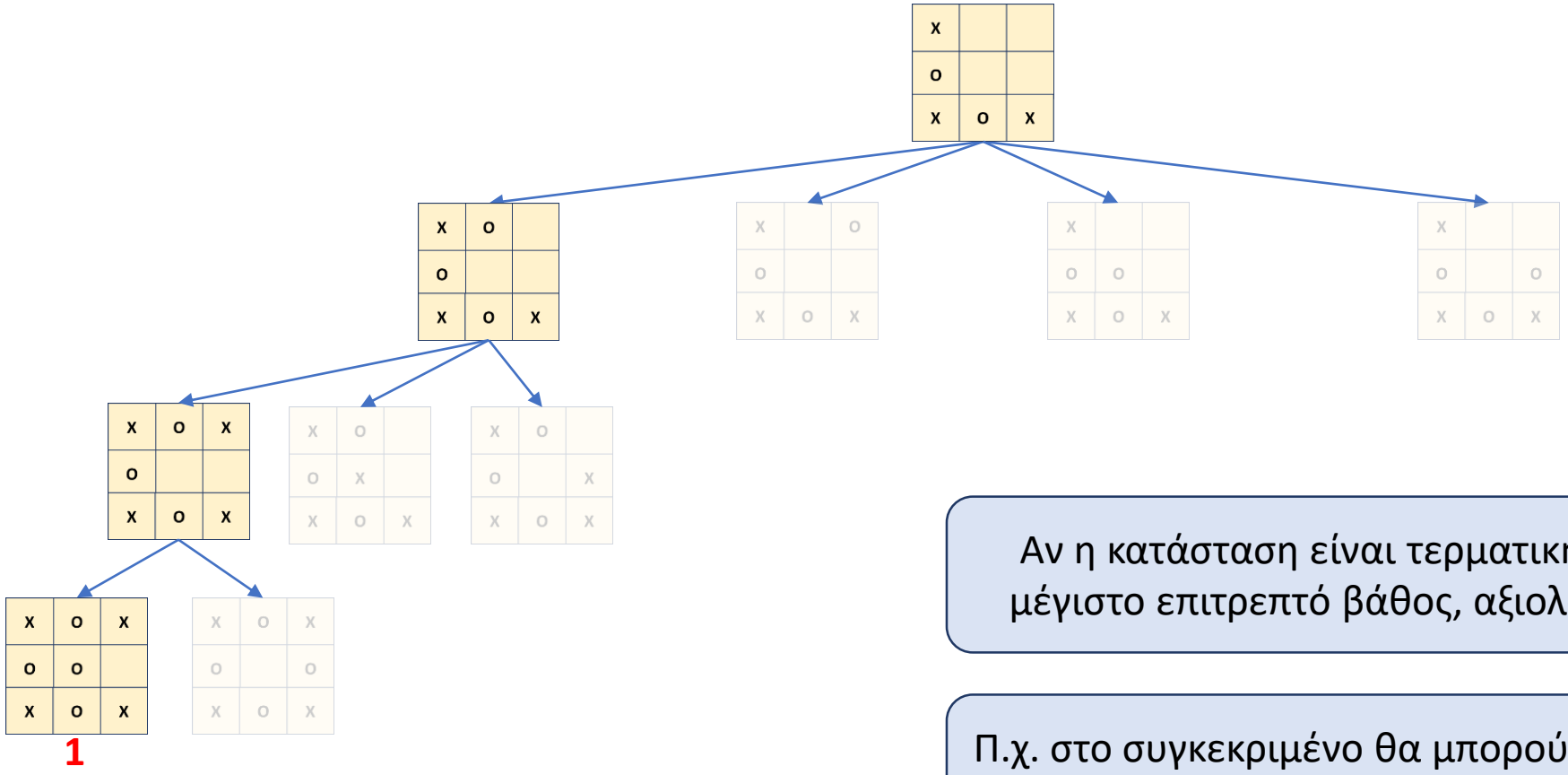


Minimax

MAX

MIN

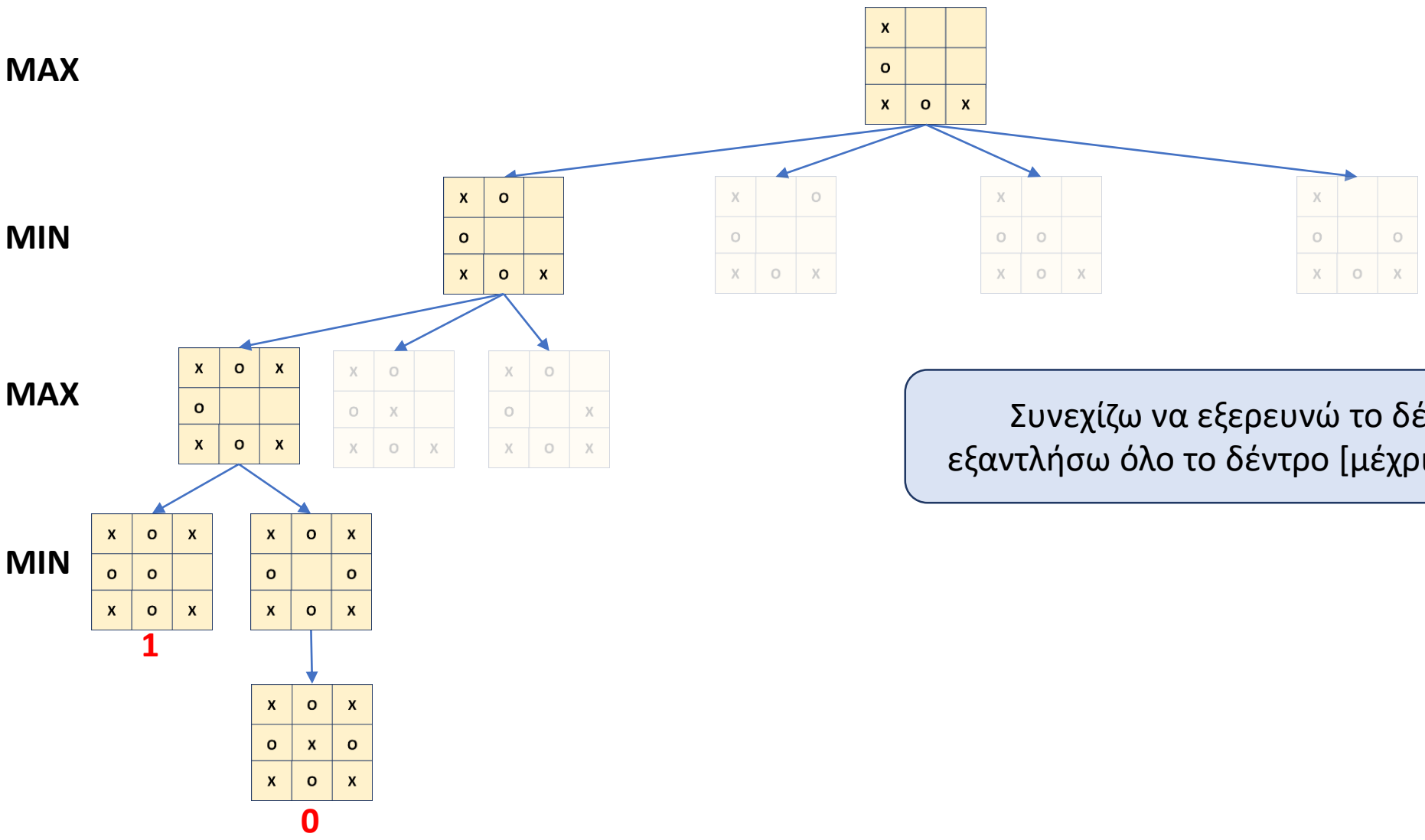
MAX



Αν η κατάσταση είναι τερματική ή έχω φτάσει στο μέγιστο επιτρεπτό βάθος, αξιολογώ την κατάσταση

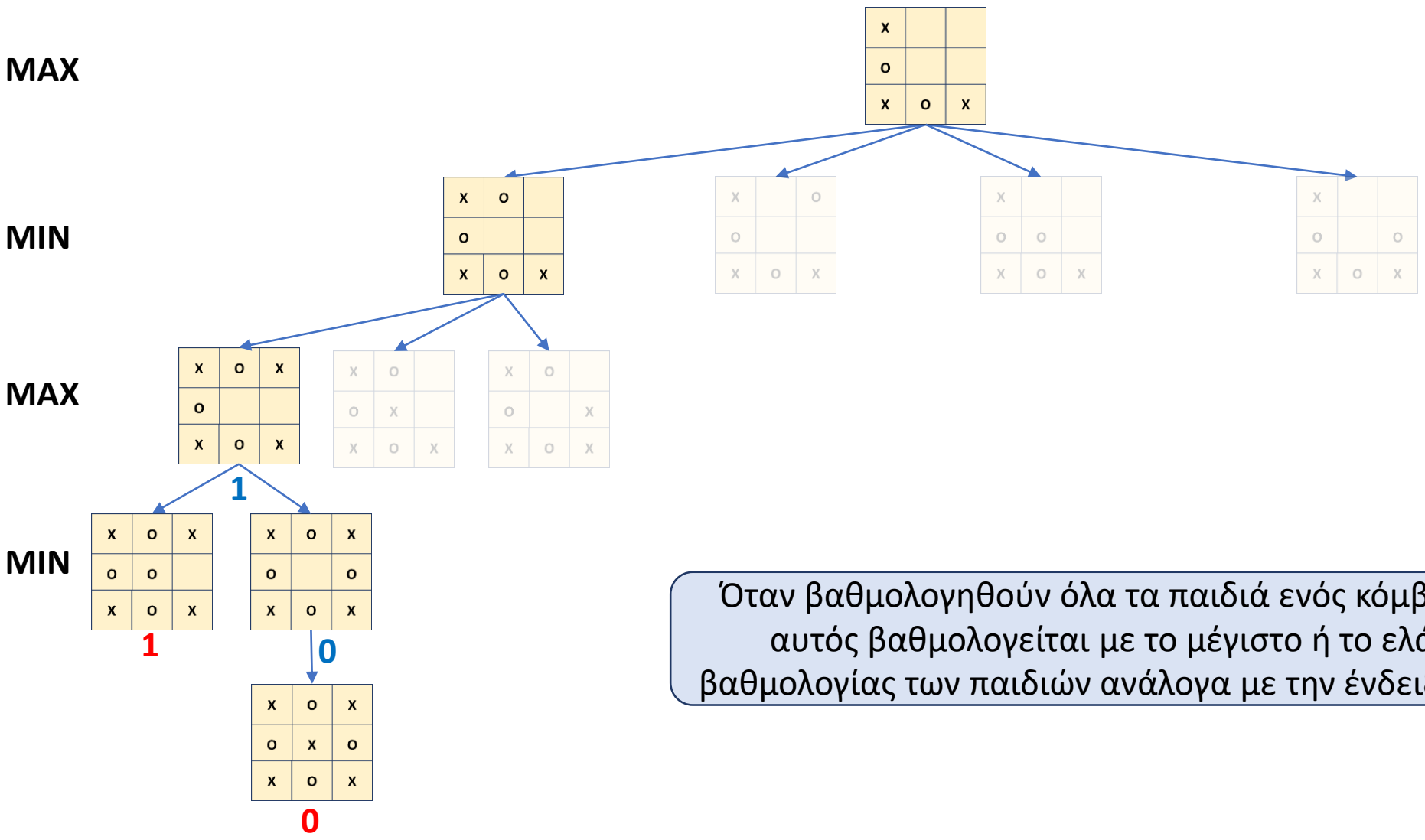
Π.χ. στο συγκεκριμένο θα μπορούσα να αξιολογήσω με 1 τη νίκη, -1 την ήττα και 0 την ισοπαλία

Minimax



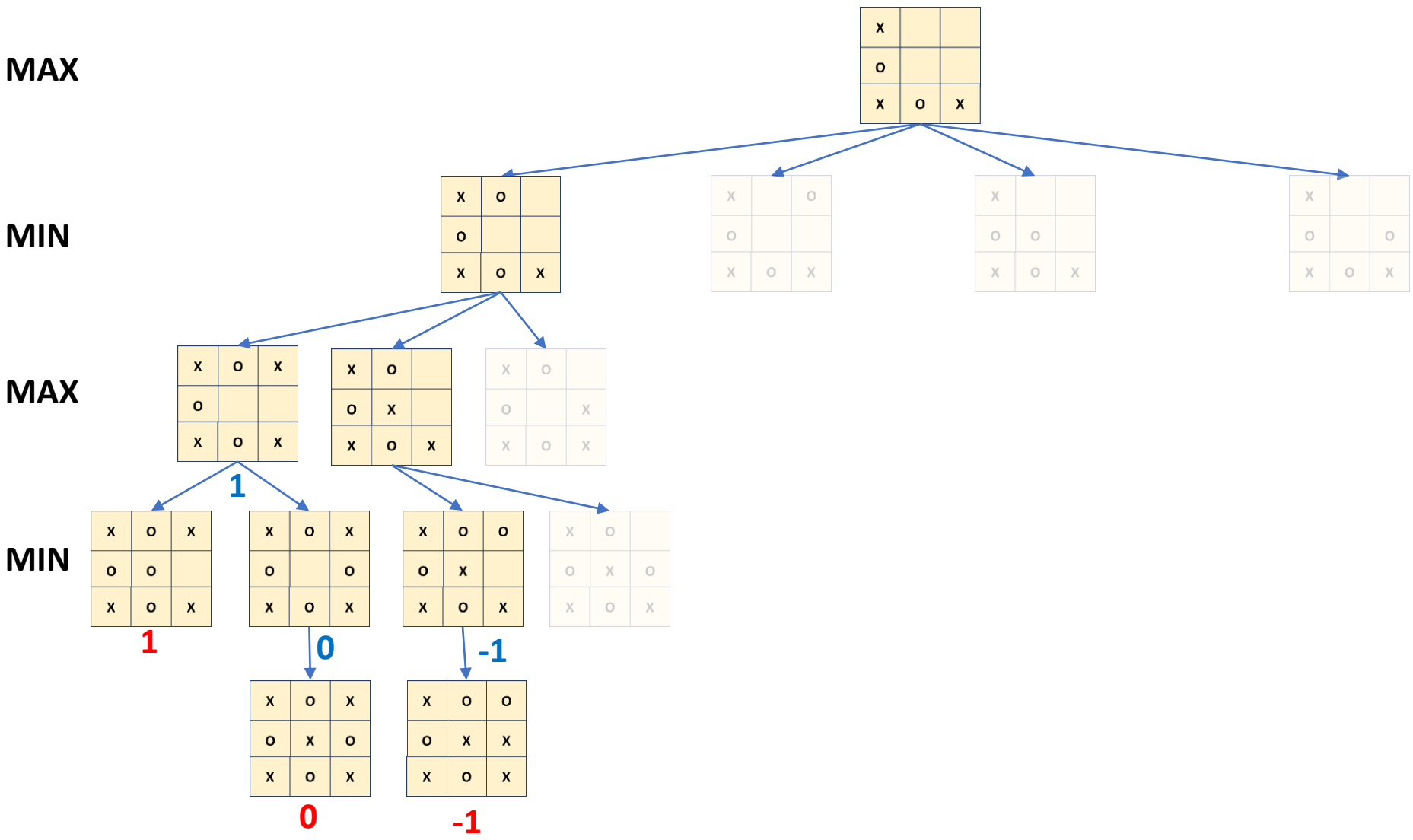
Συνεχίζω να εξερευνώ το δέντρο με DFS μέχρι να εξαντλήσω όλο το δέντρο [μέχρι το επιτρεπόμενο βάθος]

Minimax

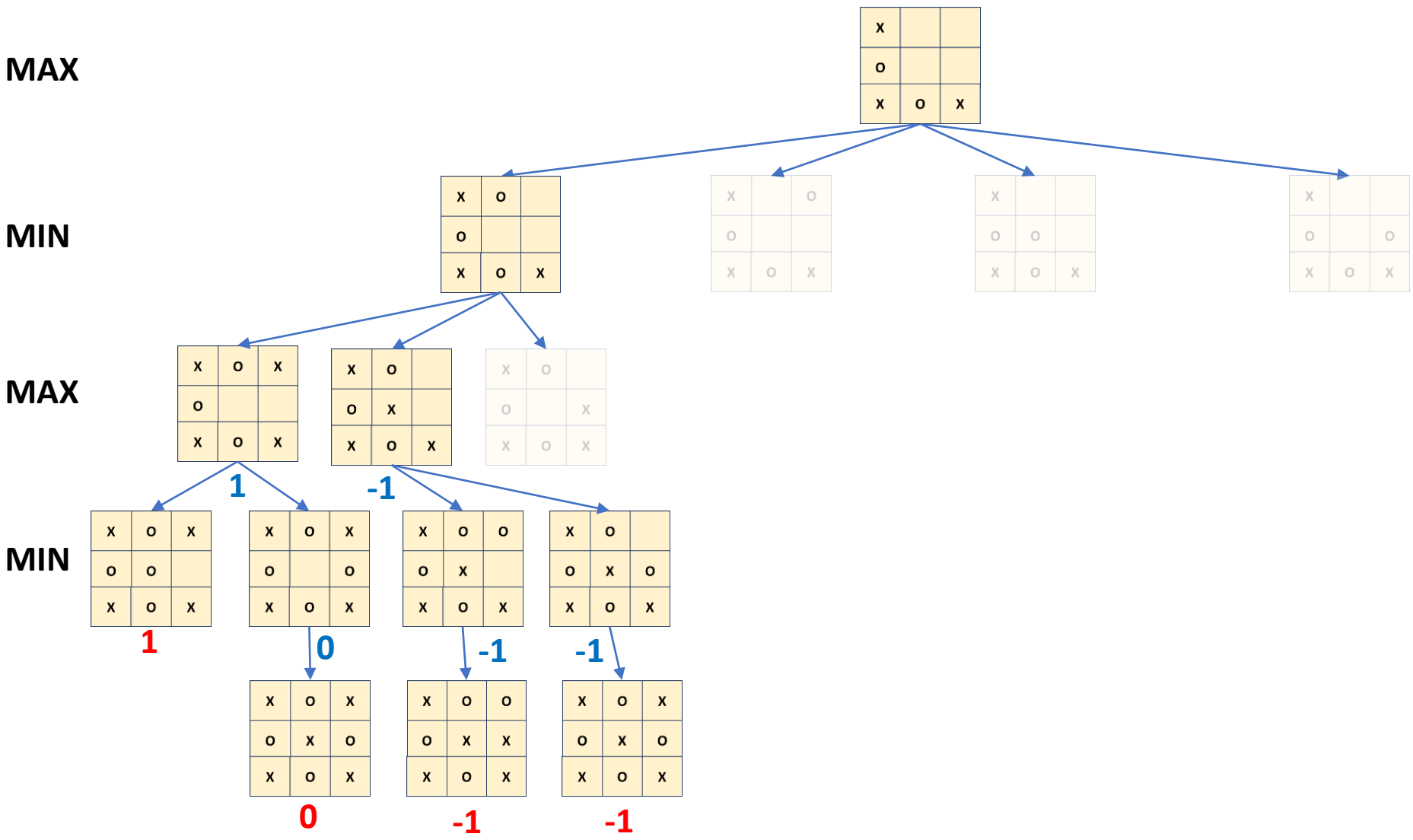


Όταν βαθμολογηθούν όλα τα παιδιά ενός κόμβου, ο κόμβος αυτός βαθμολογείται με το μέγιστο ή το ελάχιστο της βαθμολογίας των παιδιών ανάλογα με την ένδειξη (max ή min)

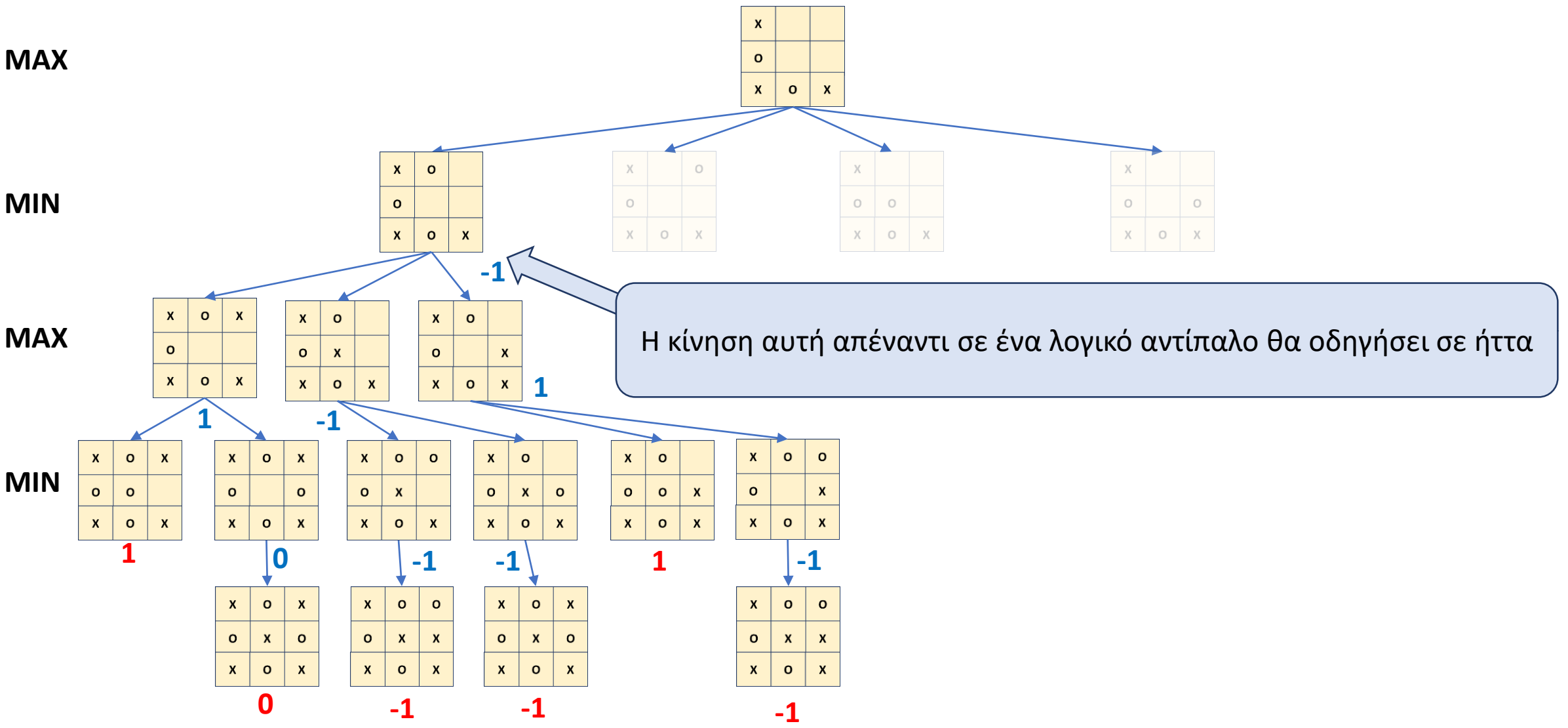
Minimax



Minimax



Minimax


MAX
MIN
MAX
MIN
-1
1
-1
1
1
0
-1
-1
1
-1
0
-1
-1
-1
Η κίνηση αυτή απέναντι σε ένα λογικό αντίπαλο θα οδηγήσει σε ήττα

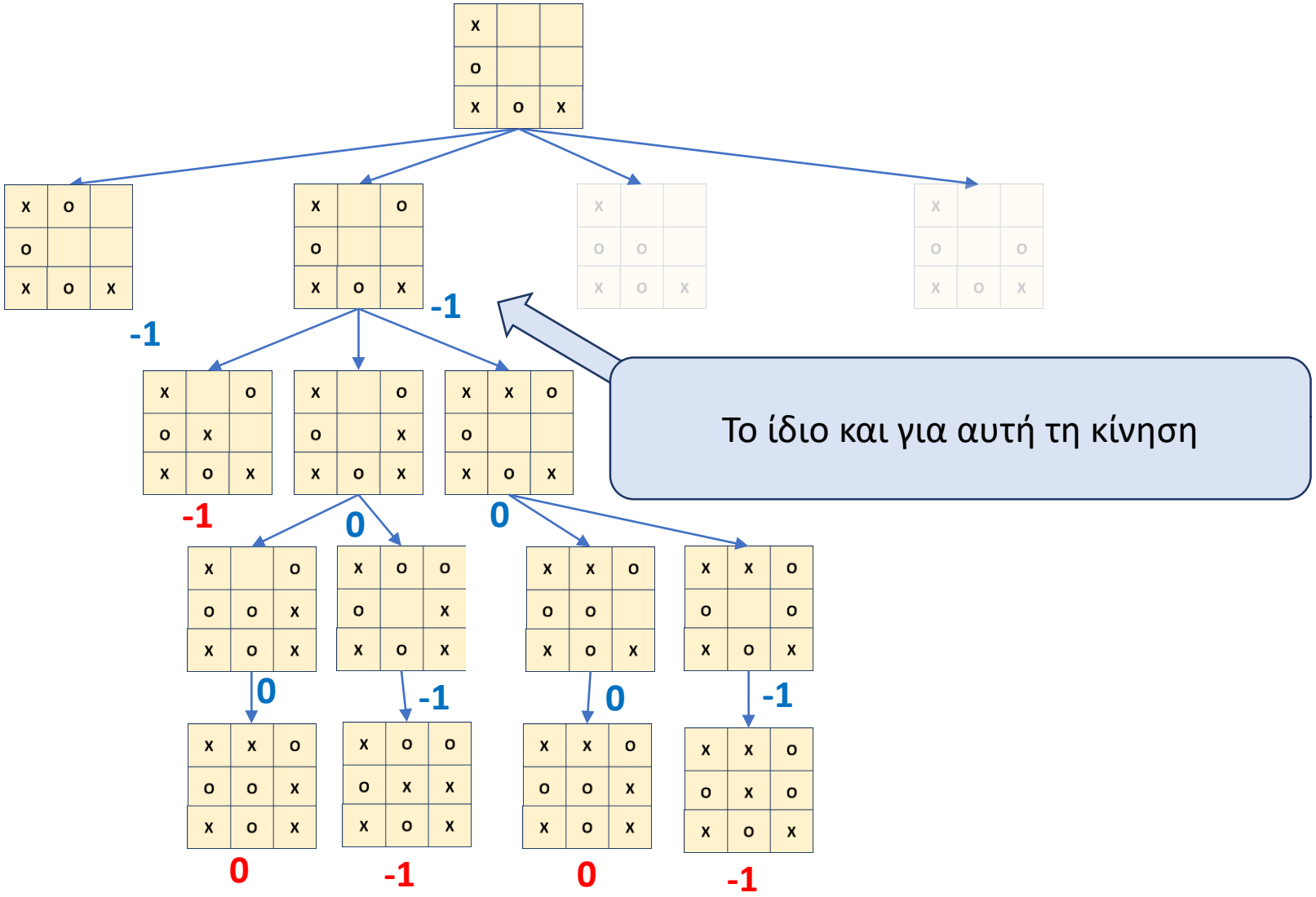
Minimax

MAX

MIN

MAX

MIN



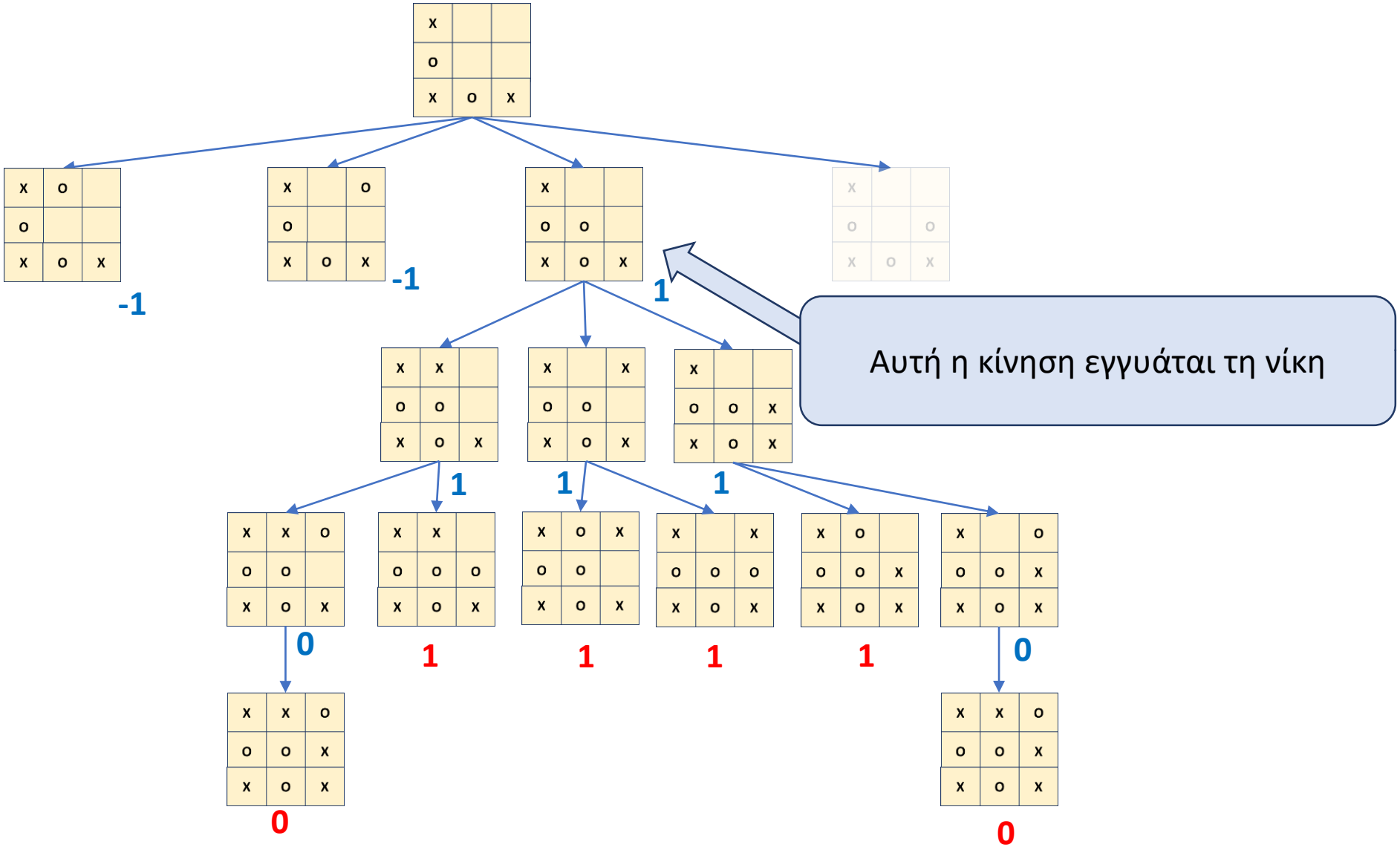
Minimax

MAX

MIN

MAX

MIN



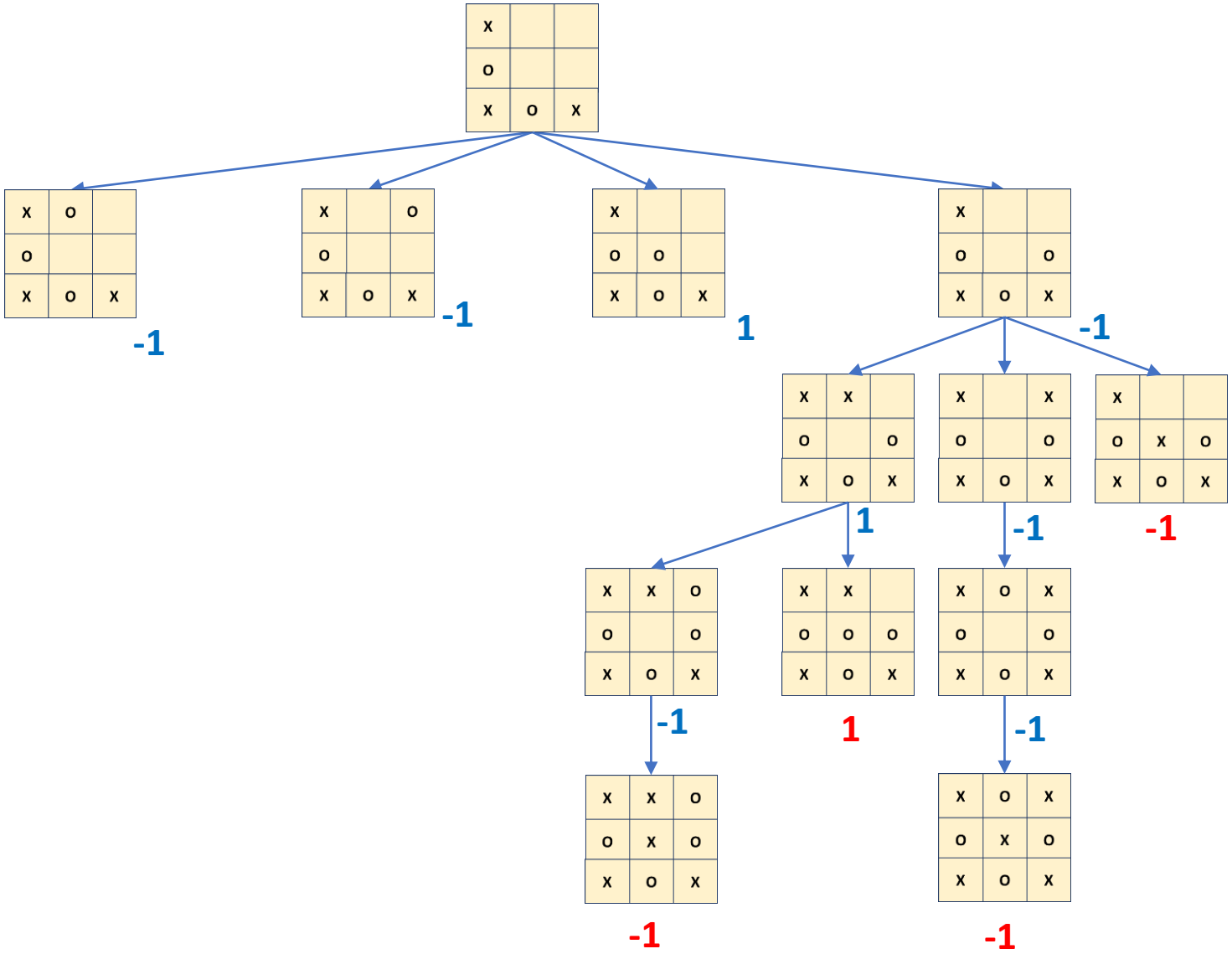
Minimax

MAX

MIN

MAX

MIN



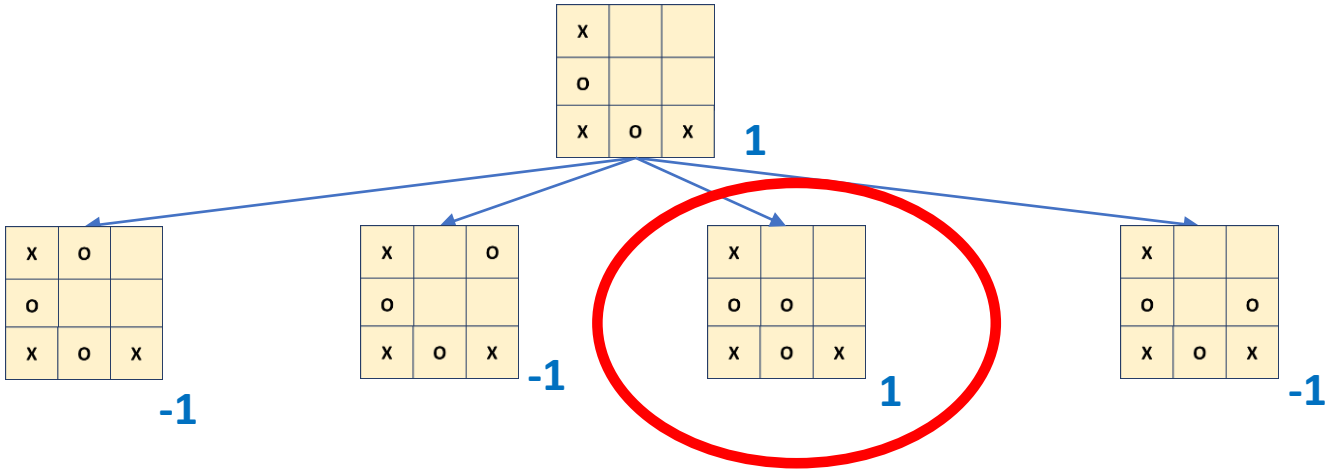
Minimax

MAX

MIN

MAX

MIN



Η ρίζα βαθμολογείται με βάση το μέγιστο των παιδιών

Ο αλγόριθμος προτείνει το παιδί από το οποίο προήλθε η βαθμολογία

MiniMax

Μέγιστο βάθος αναζήτησης

Τρέχουσα κατάσταση

Ο παίκτης που είναι η σειρά του να παίξει (true = Max, false = Min)

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

MiniMax

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

Αναπαράσταση Κατάστασης

Έλεγχος Τερματικής Κατάστασης

Αξιολόγηση Κατάστασης

Εύρεση Διάδοχων Καταστάσεων

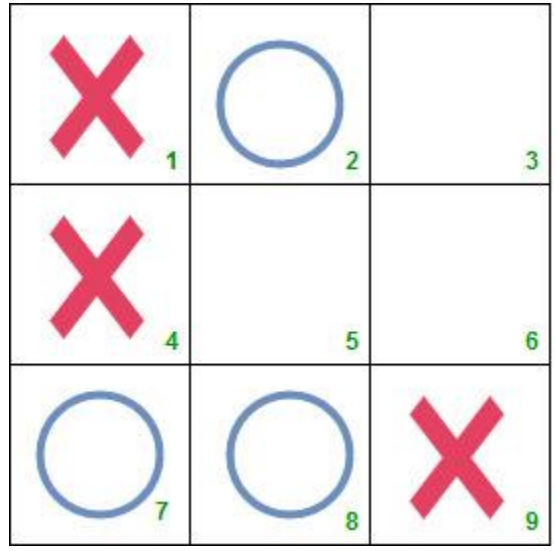
```
class TicTacToe
{
private:
        unsigned char board[SIZE][SIZE];
    public:
        TicTacToe();
        TicTacToe(unsigned char a[][SIZE]);
        bool setCell(int i, int j, char value);
        char getCell(int i, int j);
        vector <TicTacToe> expand(char k);
        string toString();
        int evaluate();
};
```

Αναπαράσταση Κατάστασης

Εύρεση Διάδοχων Καταστάσεων

Αξιολόγηση Κατάστασης

Έλεγχος Τερματικής Κατάστασης



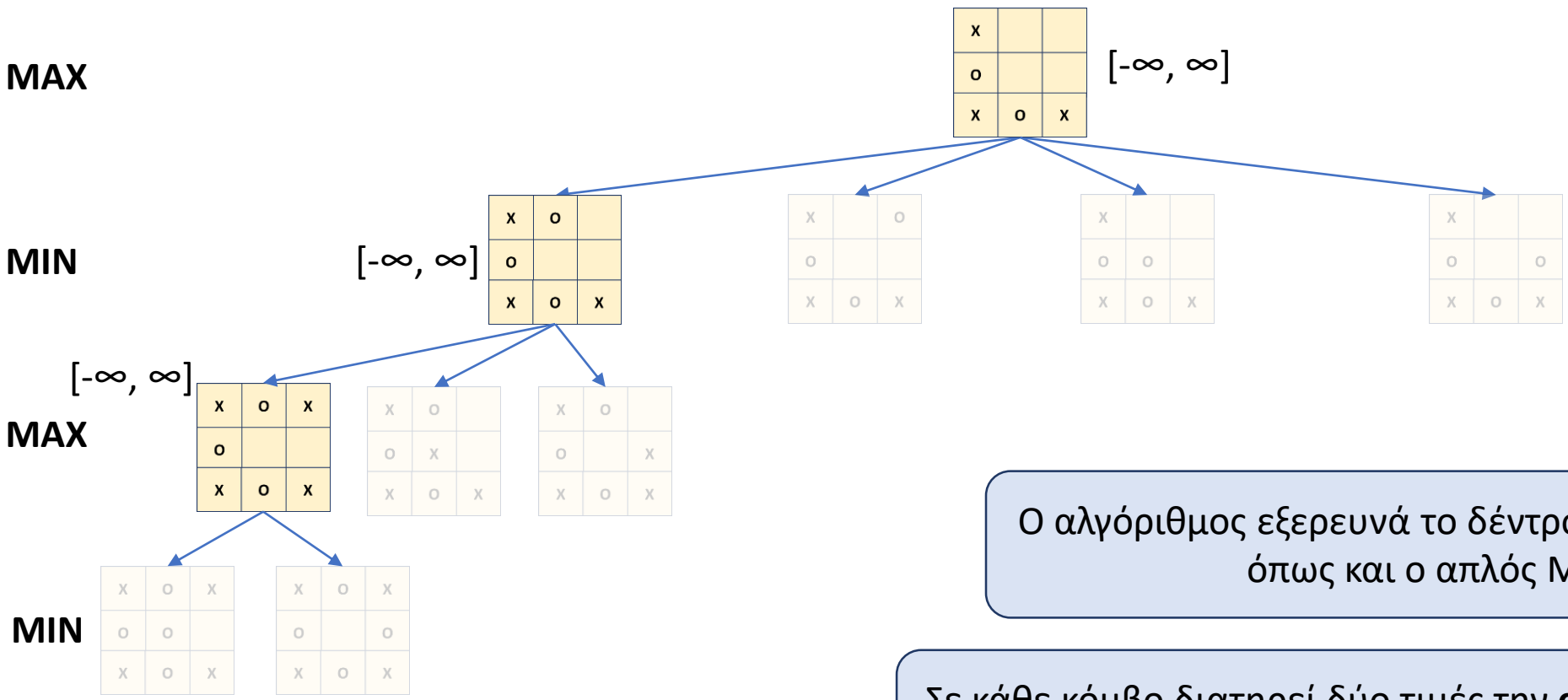
```
vector <TicTacToe> expand(char k)
        {
            vector<TicTacToe> children;

            for (int i=0;i<SIZE;i++)
                for (int j=0;j<SIZE;j++)
                    if (board[i][j]==' ')
                    {
                        TicTacToe child = *this;
                        child.board[i][j]=k;
                        children.push_back(child);
                    }
            return children;
        }
```

```
int evaluate()
{
   for (int i=0;i<SIZE;i++) //check for rows
   {
      bool same=true;
      for (int j=1;j<SIZE;j++)
         if (board[i][j]==' ' ||
                        board[i][j]!=board[i][j-1])
         {
            same=false;
            break;
         }
      if (same)
         return board[i][0]=='X'?1:-1;
   }
   for (int j=0;j<SIZE;j++) //check for columns
   {
      bool same=true;
      for (int i=1;i<SIZE;i++)
         if (board[i][j]==' '||
                        board[i][j]!=board[i-1][j])
         {
            same=false;
            break;
         }
      if (same)
         return board[0][j]=='X'?1:-1;
   }
   bool same=true;
   for (int i=1;i<SIZE;i++) //check for main diagonal
      if (board[i][i]==' '||
                     board[i][i]!=board[i-1][i-1])
      {
         same=false;
         break;
      }
   if (same)
      return board[0][0]=='X'?1:-1;
   same=true;
   for (int i=1;i<SIZE;i++) //check for 2ND diagonal
   if (board[i][SIZE-1-i]==' '
      || board[i][SIZE-1-i]!=board[i-1][SIZE-1-(i-1)])
   {
      same=false;
      break;
   }
   if (same)
      return board[1][1]=='X'?1:-1;
   for (int i=0;i<SIZE;i++)
      for (int j=0;j<SIZE;j++)
         if (board[i][j]==' ')
            return -100; // ONGOING GAME
   return 0;
}
```

```
int minimax (TicTacToe s, int depth, bool isMax, TicTacToe &best, int &count)
{
    int k=s.evaluate();
    count++;
    if (depth==0 || k!=-100)
    {
        best = s;
        return k;
    }
    int max,temp;
    TicTacToe maxState,tempState;
    vector<TicTacToe> children = s.expand(isMax?'X':'O');
    max=minimax(children[0],depth-1,!isMax,tempState,count);
    maxState=children[0];
    for (int i=1;i<children.size();i++)
    {
        temp=minimax(children[i],depth-1,!isMax,tempState,count);
        if ((temp>max) == isMax)
        {
            max=temp;
            maxState=children[i];
        }
    }
    best = maxState;
    return max;
}
```

Minimax με Alpha Beta



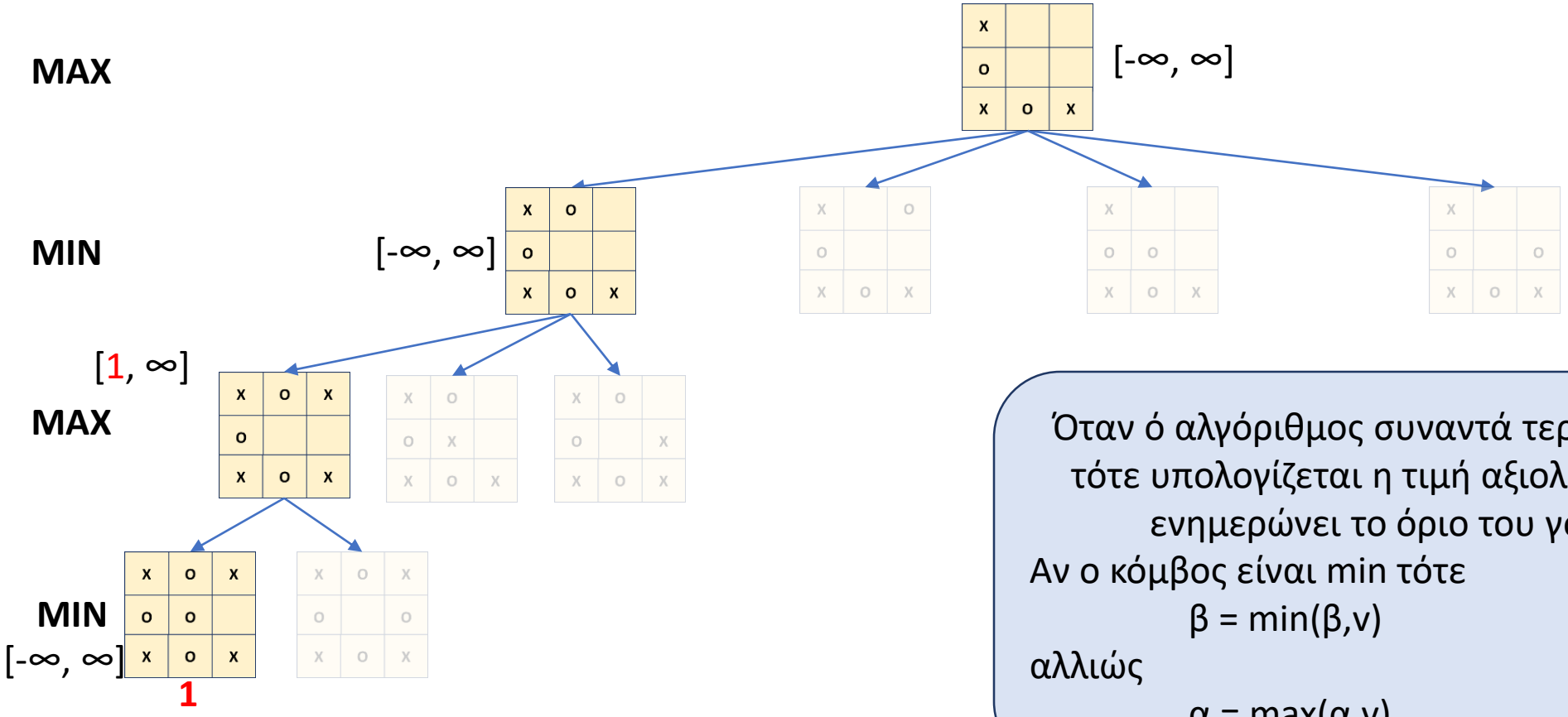
Ο αλγόριθμος εξερευνά το δέντρο σύμφωνα με το DFS όπως και ο απλός Minimax

Σε κάθε κόμβο διατηρεί δύο τιμές την **α** (κάτω όριο) και τη **β** (άνω όριο) αναφορικά με τη τιμή που μπορεί να πάρει ο κόμβος

Η ρίζα αρχικά έχει ως τιμές α=- ∞ και β=∞

Οι τιμές αυτές διαδίδονται προς τα κάτω

Minimax με Alpha Beta



Όταν ό αλγόριθμος συναντά τερματική κατάσταση, τότε υπολογίζεται η τιμή αξιολόγησης ***v*** και αυτή ενημερώνει το όριο του γονέα ως εξής:

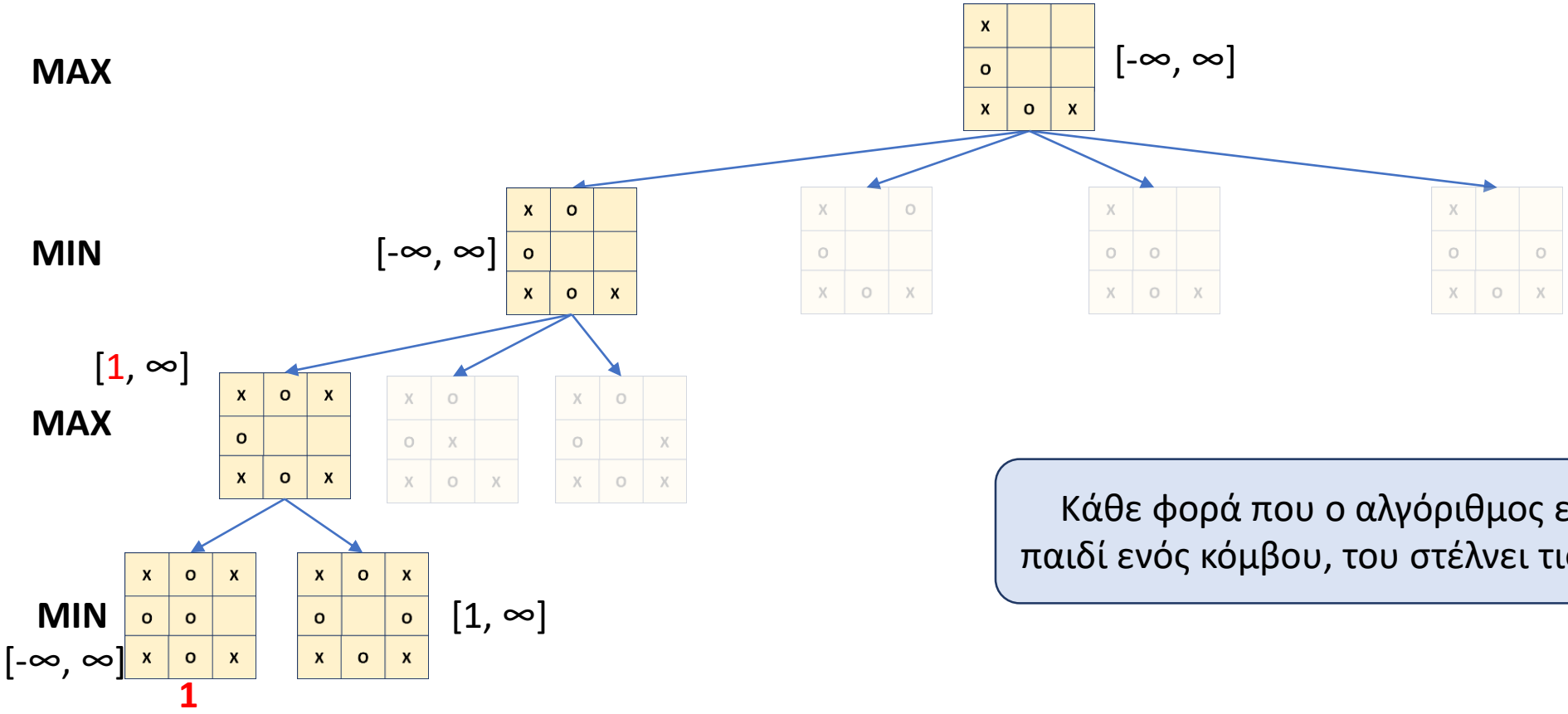
Αν ο κόμβος είναι min τότε

$\beta = \min(\beta, v)$

αλλιώς

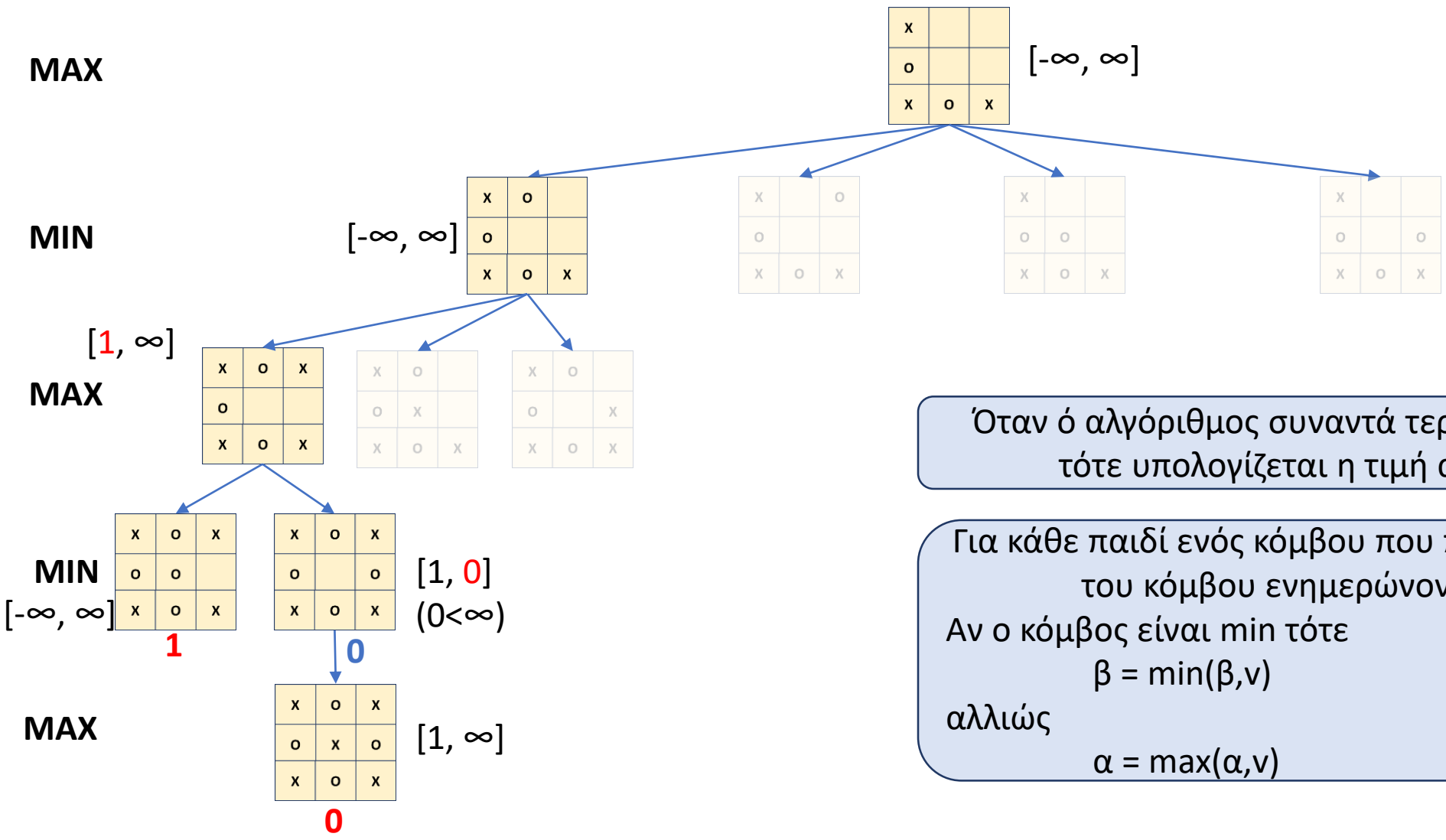
$\alpha = \max(\alpha, v)$

Minimax με Alpha Beta



Κάθε φορά που ο αλγόριθμος εξετάζει το επόμενο παιδί ενός κόμβου, του στέλνει τις ενημερωμένες τιμές

Minimax με Alpha Beta



Όταν ό αλγόριθμος συναντά τερματική κατάσταση, τότε υπολογίζεται η τιμή αξιολόγησης ***v***

Για κάθε παιδί ενός κόμβου που παίρνει τιμή, τα όρια του κόμβου ενημερώνονται ως εξής:

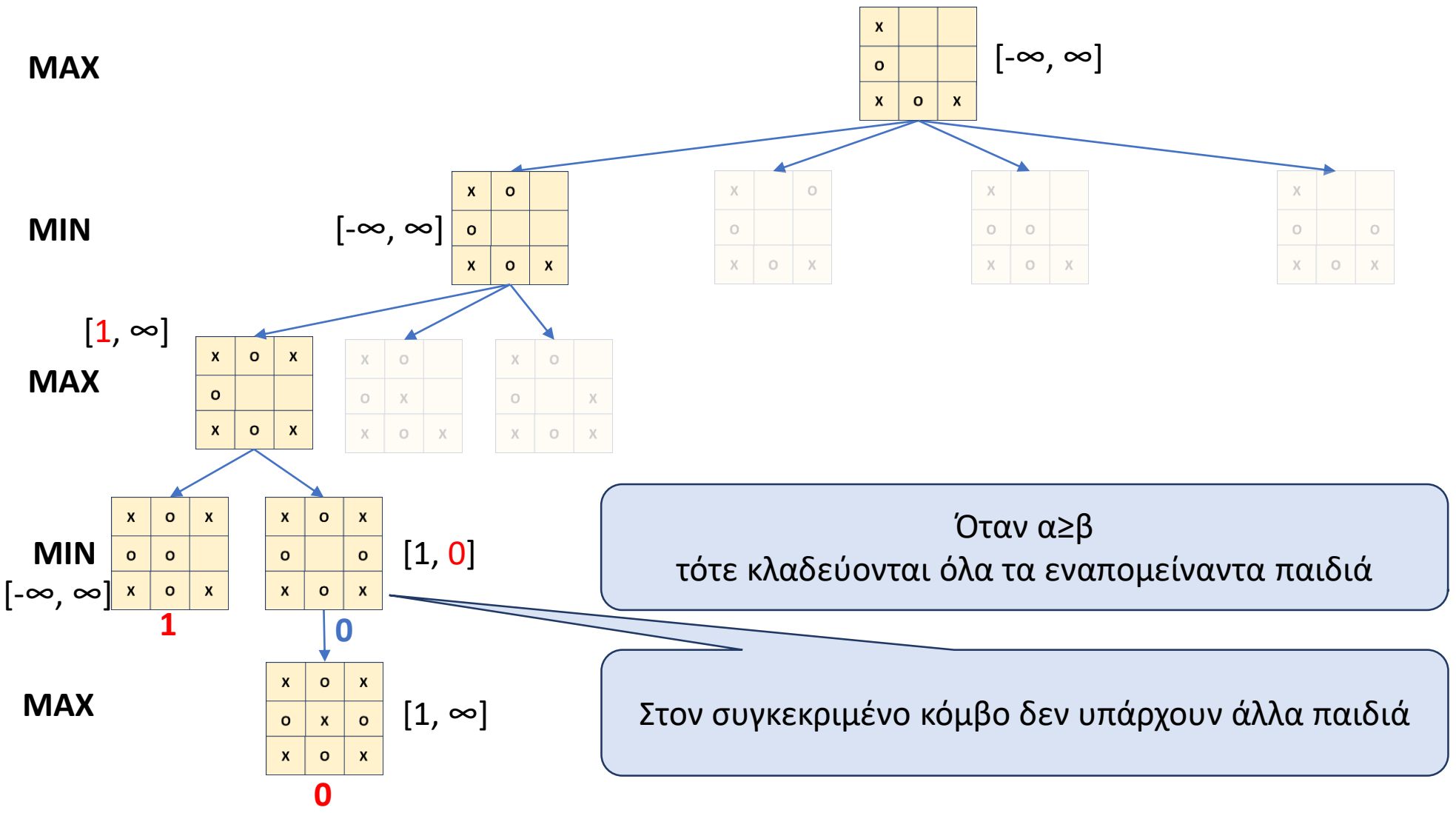
Αν ο κόμβος είναι min τότε

$\beta = \min(\beta, v)$

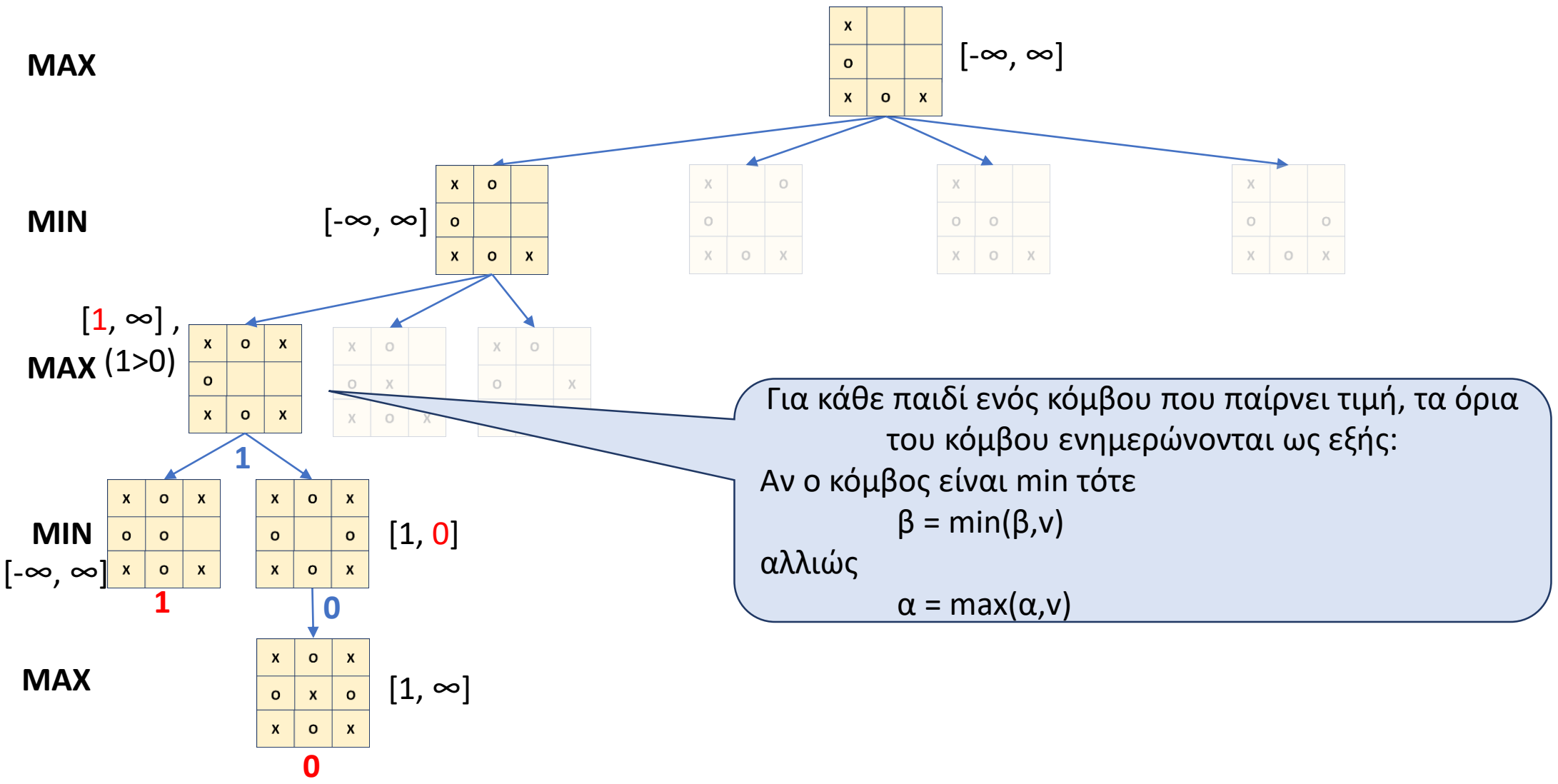
αλλιώς

$\alpha = \max(\alpha, v)$

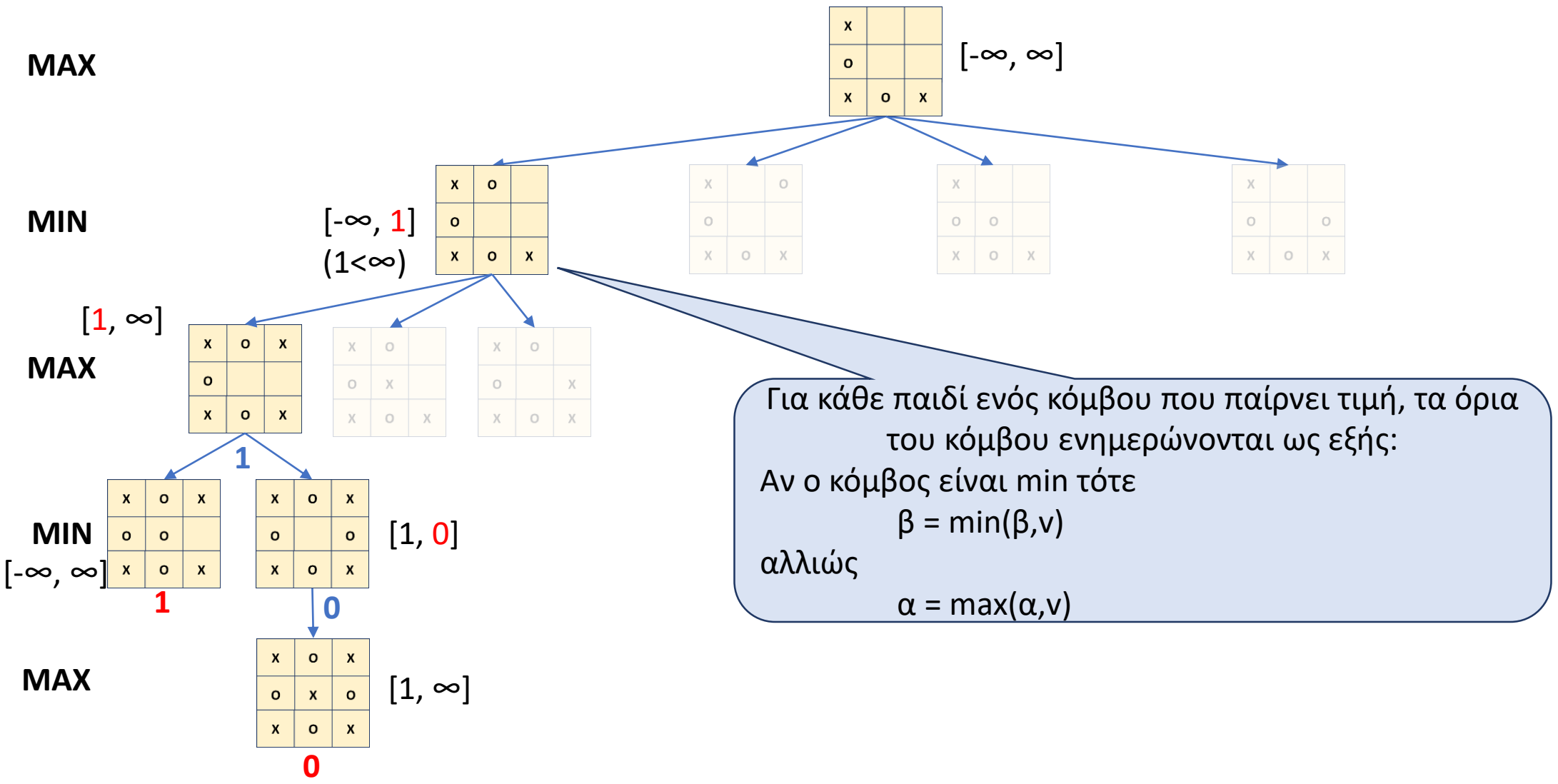
Minimax με Alpha Beta



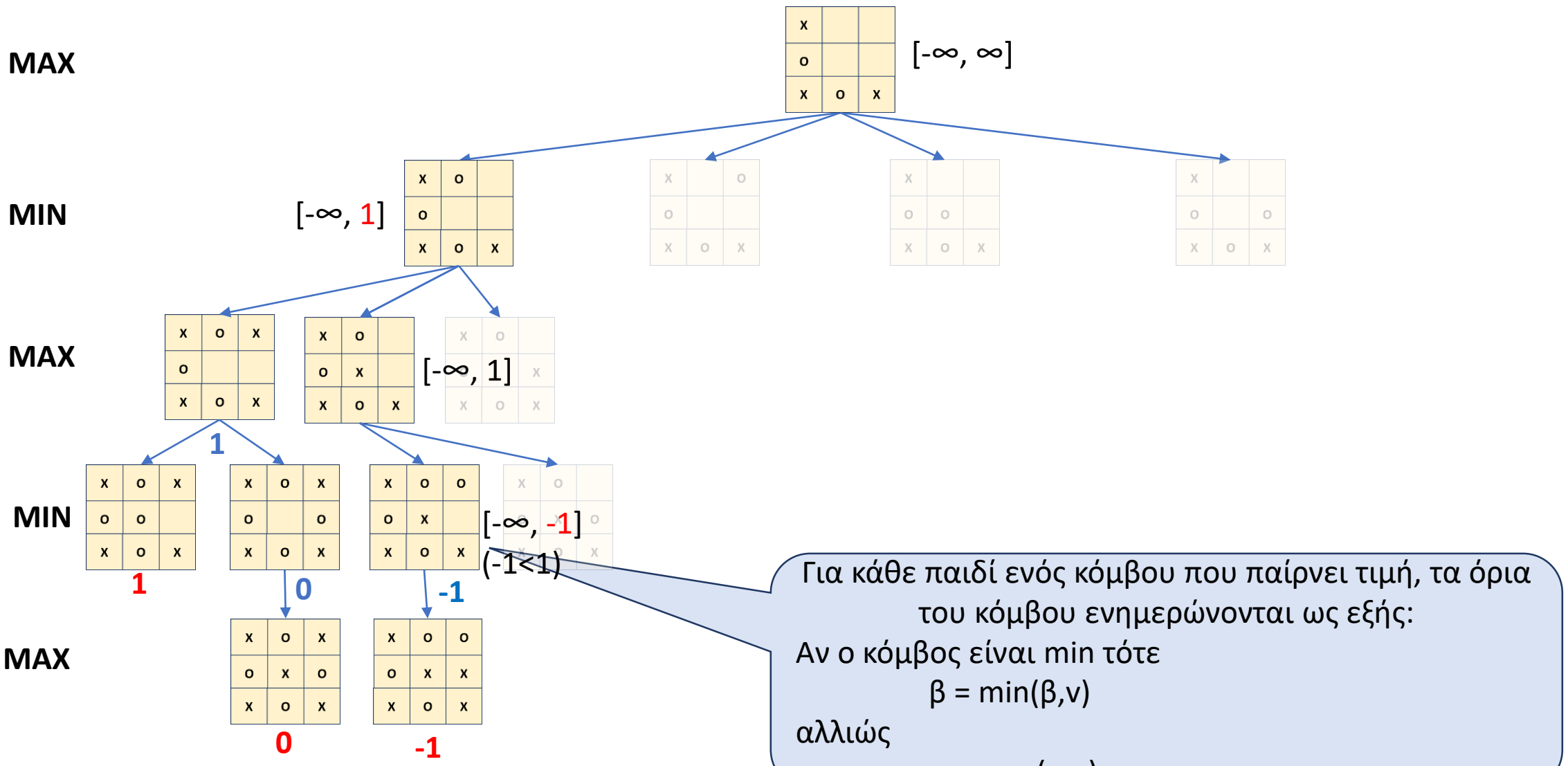
Minimax με Alpha Beta



Minimax με Alpha Beta



Minimax με Alpha Beta



Για κάθε παιδί ενός κόμβου που παίρνει τιμή, τα όρια του κόμβου ενημερώνονται ως εξής:

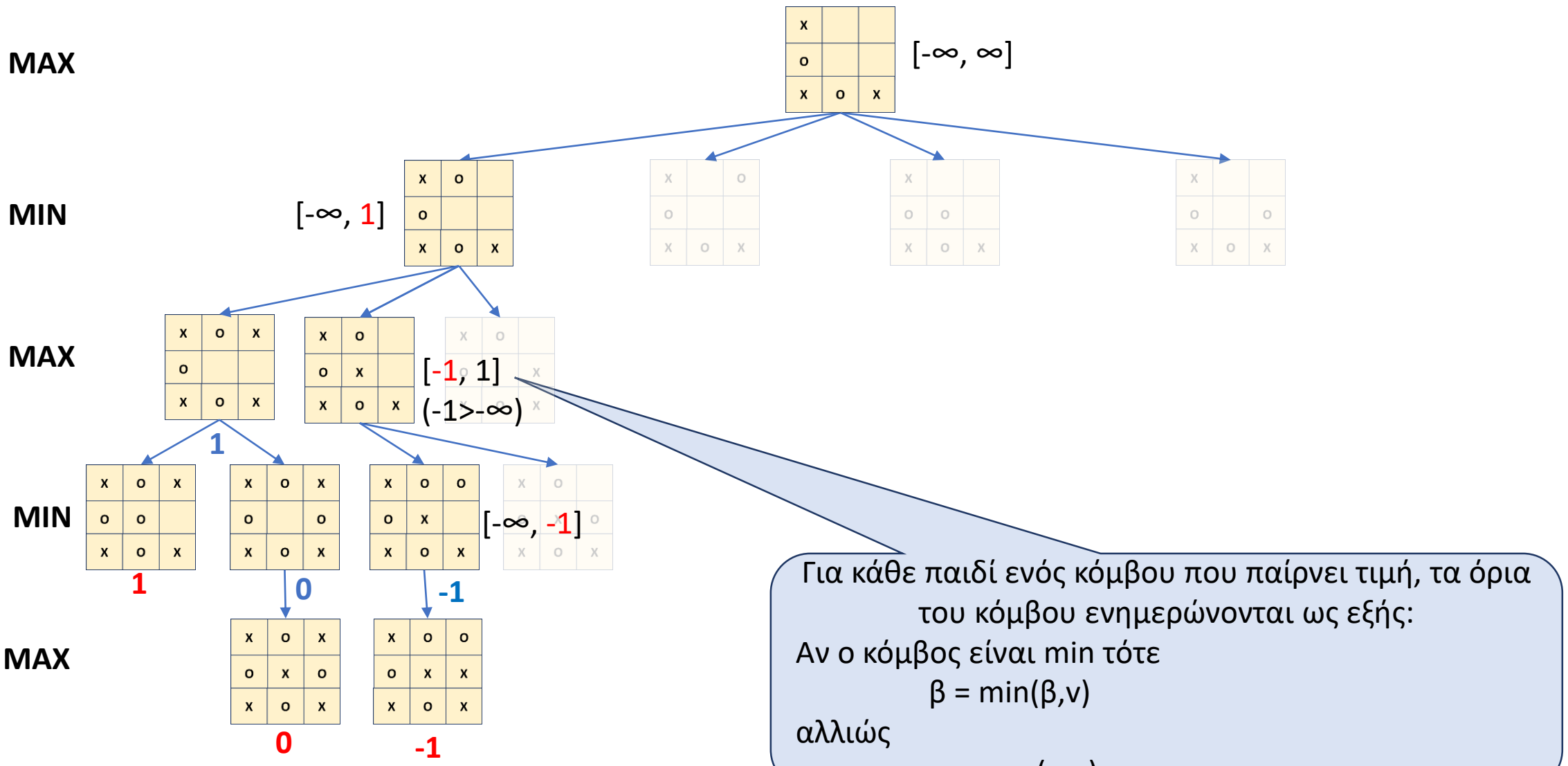
Αν ο κόμβος είναι min τότε

$\beta = \min(\beta, v)$

αλλιώς

$\alpha = \max(\alpha, v)$

Minimax με Alpha Beta



Για κάθε παιδί ενός κόμβου που παίρνει τιμή, τα όρια του κόμβου ενημερώνονται ως εξής:

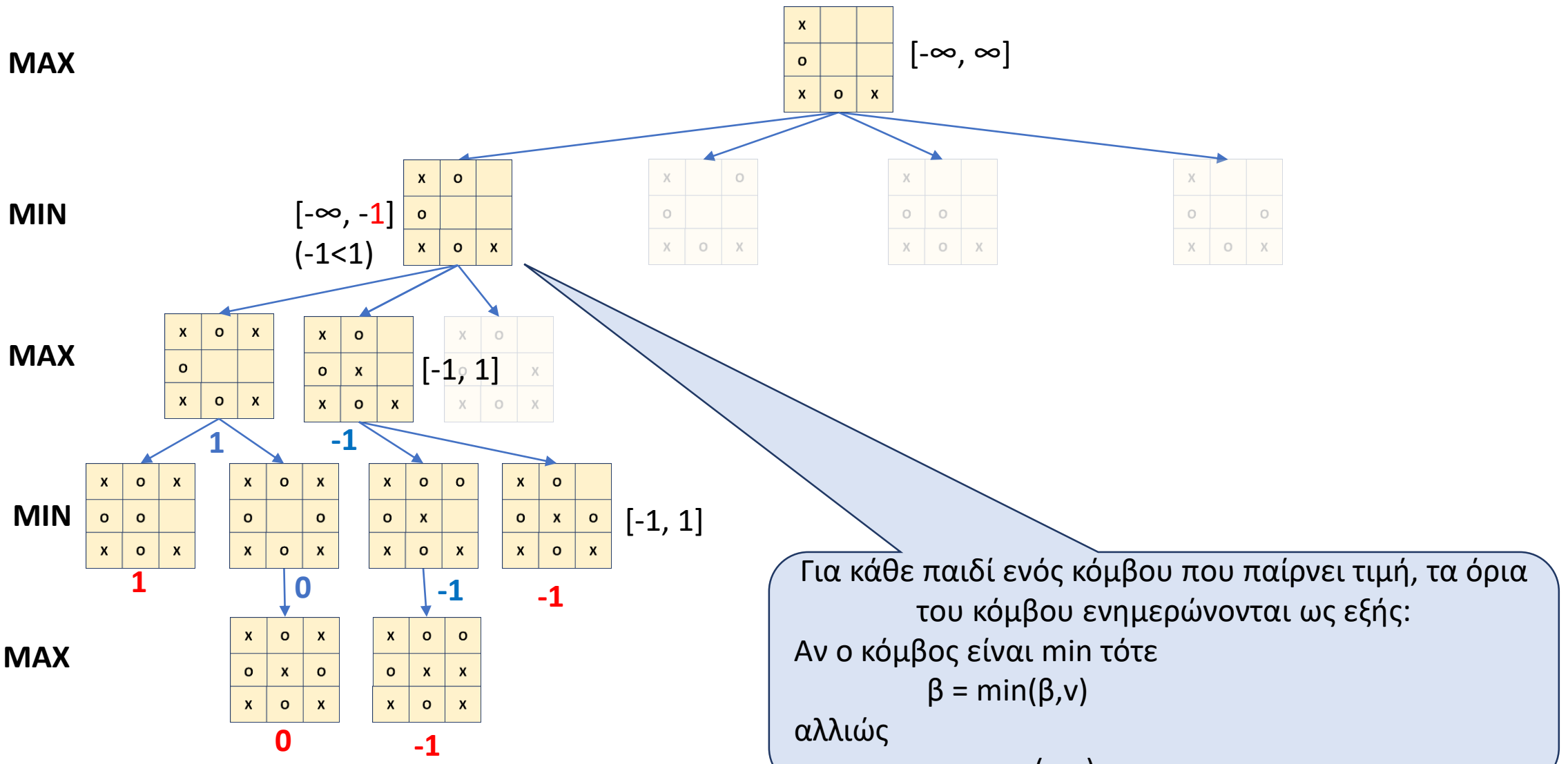
Αν ο κόμβος είναι min τότε

$\beta = \min(\beta, v)$

αλλιώς

$\alpha = \max(\alpha, v)$

Minimax με Alpha Beta



Για κάθε παιδί ενός κόμβου που παίρνει τιμή, τα όρια του κόμβου ενημερώνονται ως εξής:

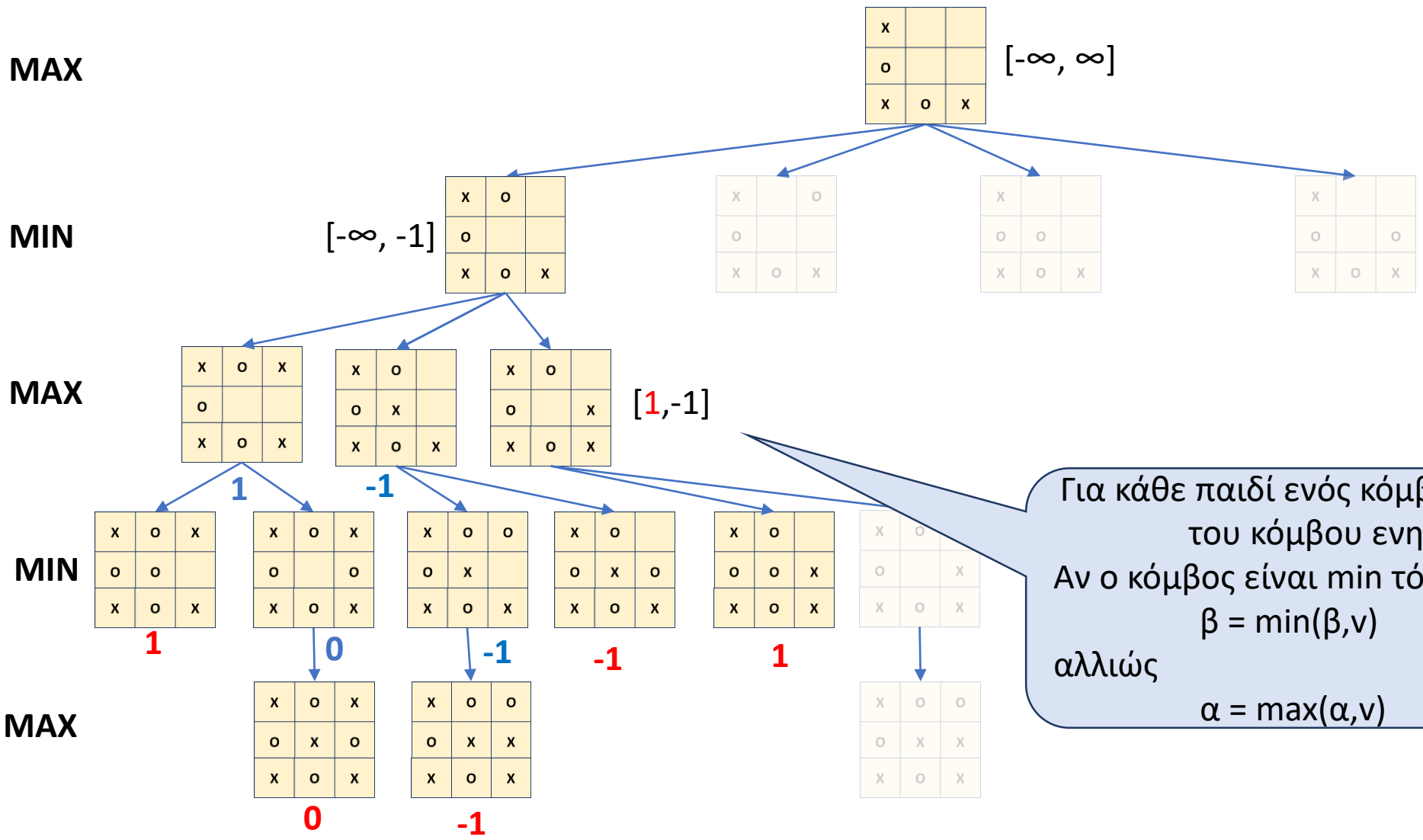
Αν ο κόμβος είναι min τότε

$\beta = \min(\beta, v)$

αλλιώς

$\alpha = \max(\alpha, v)$

Minimax με Alpha Beta



Για κάθε παιδί ενός κόμβου που παίρνει τιμή, τα όρια του κόμβου ενημερώνονται ως εξής:

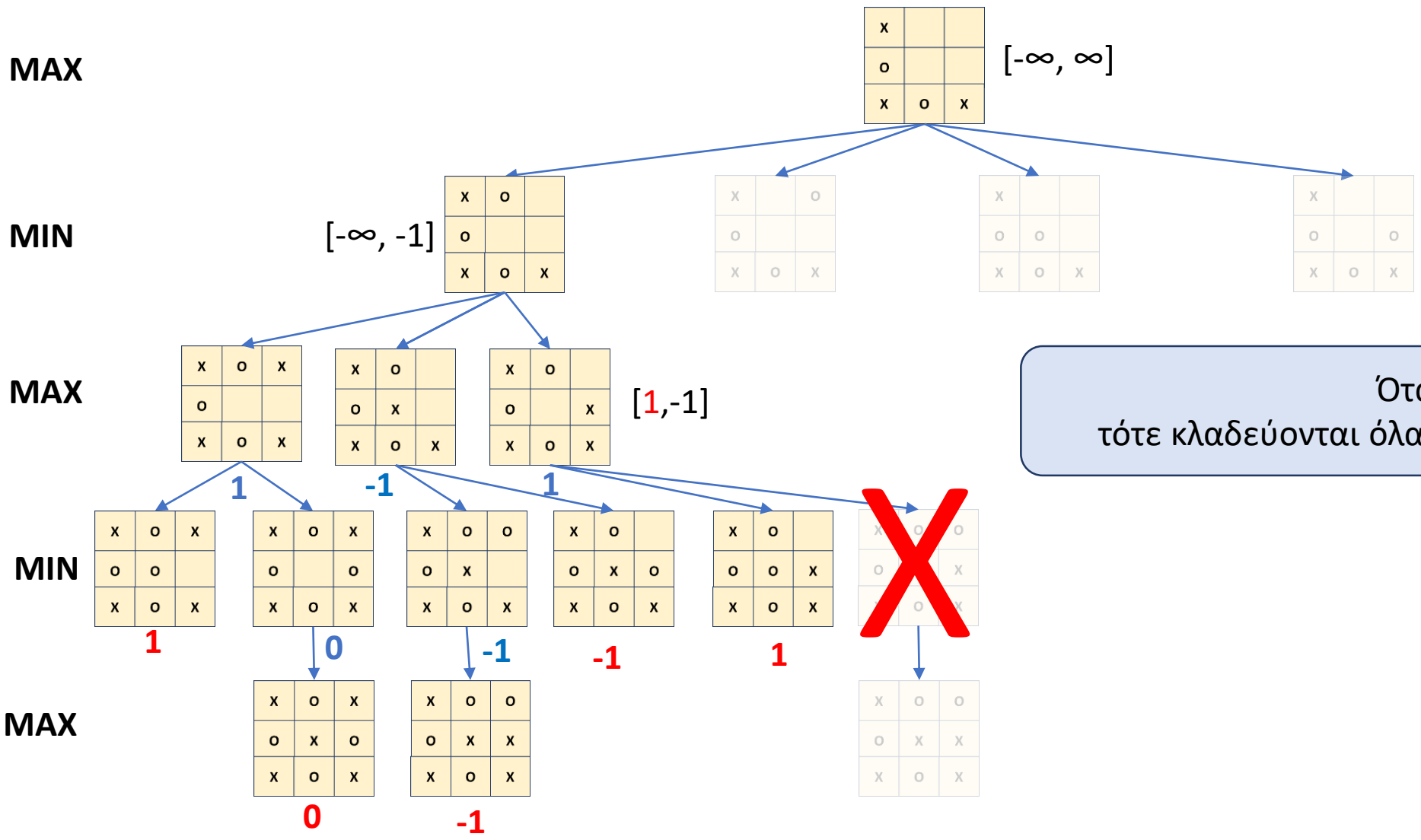
Αν ο κόμβος είναι min τότε

$\beta = \min(\beta, v)$

αλλιώς

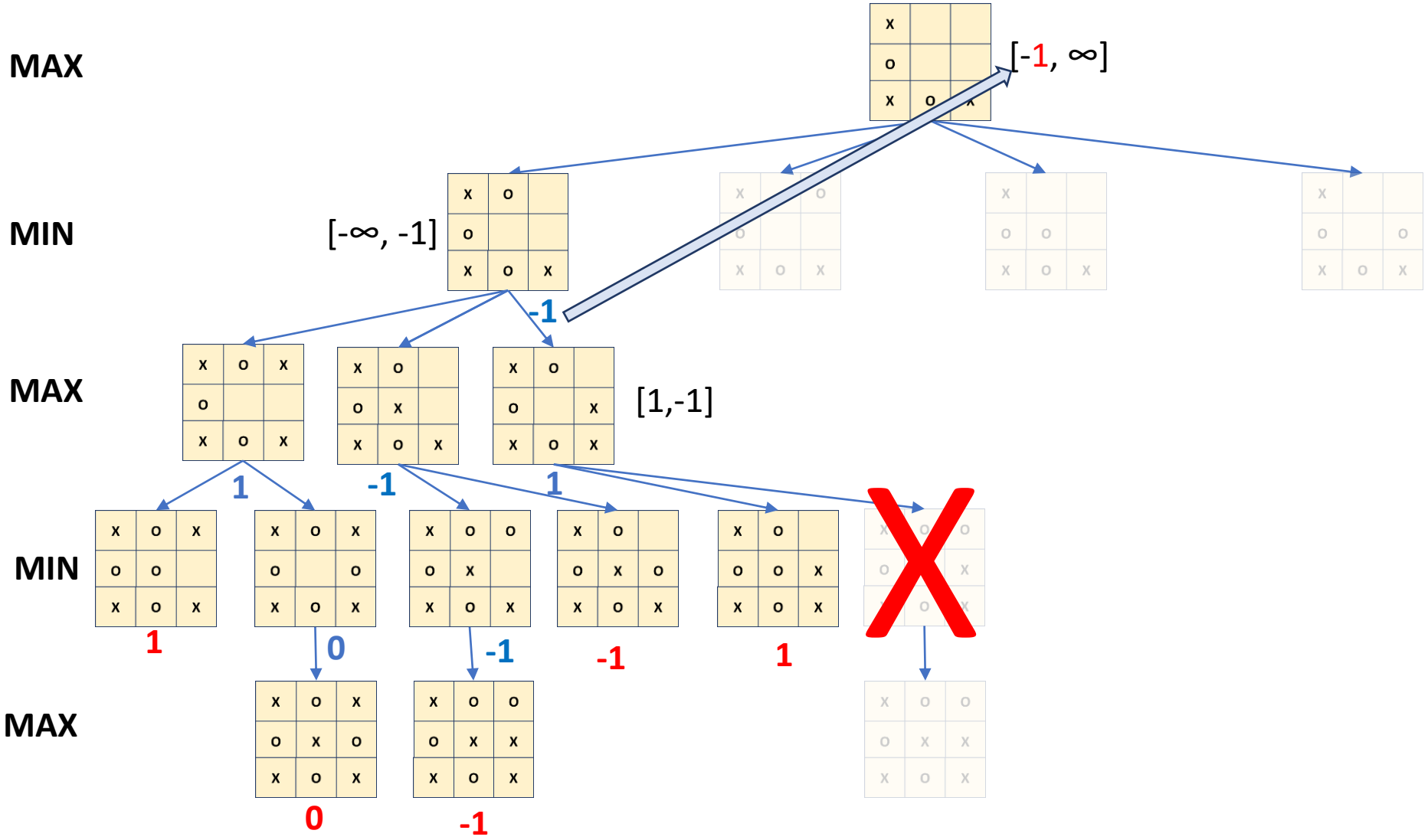
$\alpha = \max(\alpha, v)$

Minimax με Alpha Beta



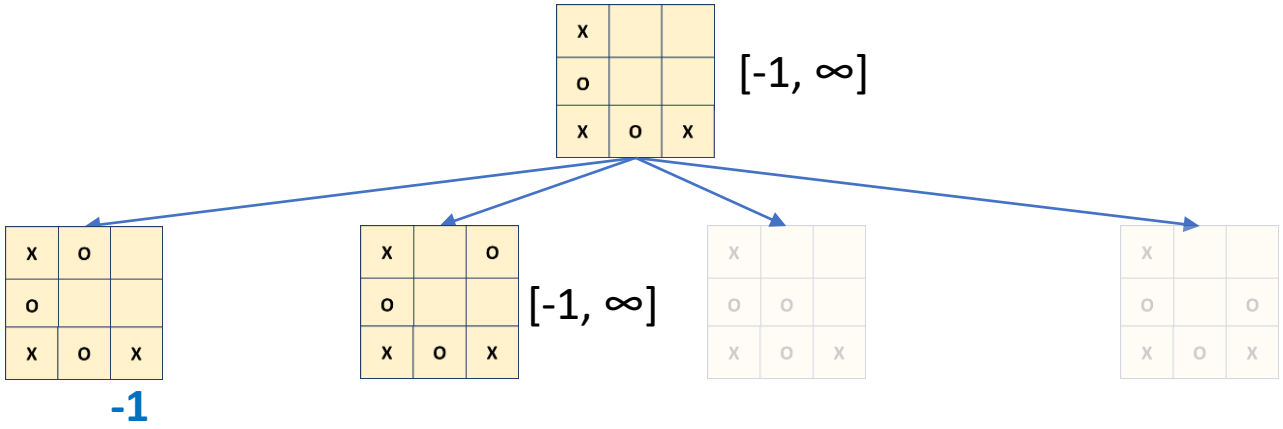
Όταν α ≥ β
τότε κλαδεύονται όλα τα εναπομείναντα παιδιά

Minimax με Alpha Beta



Minimax με Alpha Beta

MAX



MIN

MAX

MIN

MAX

Minimax με Alpha Beta

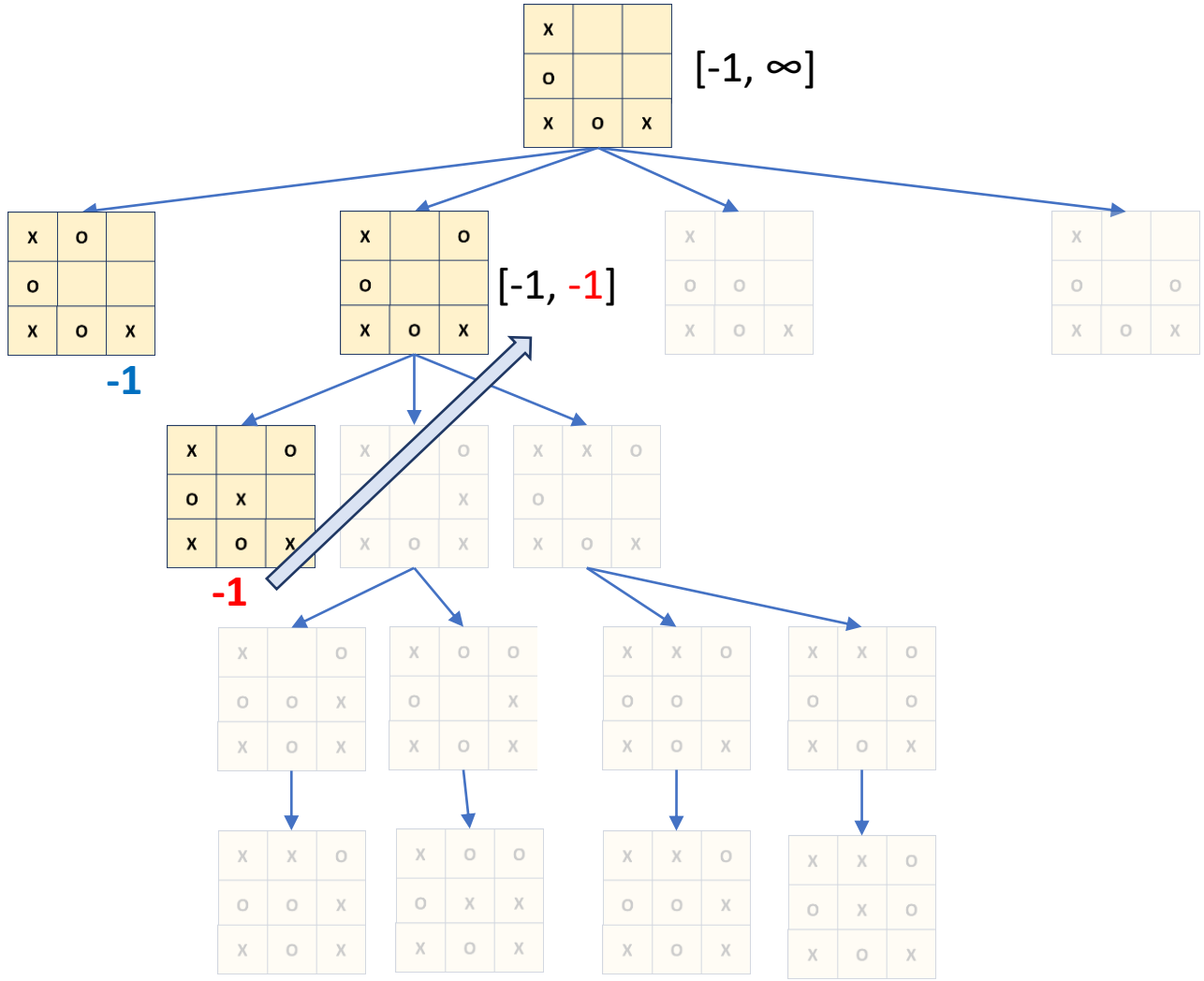
MAX

MIN

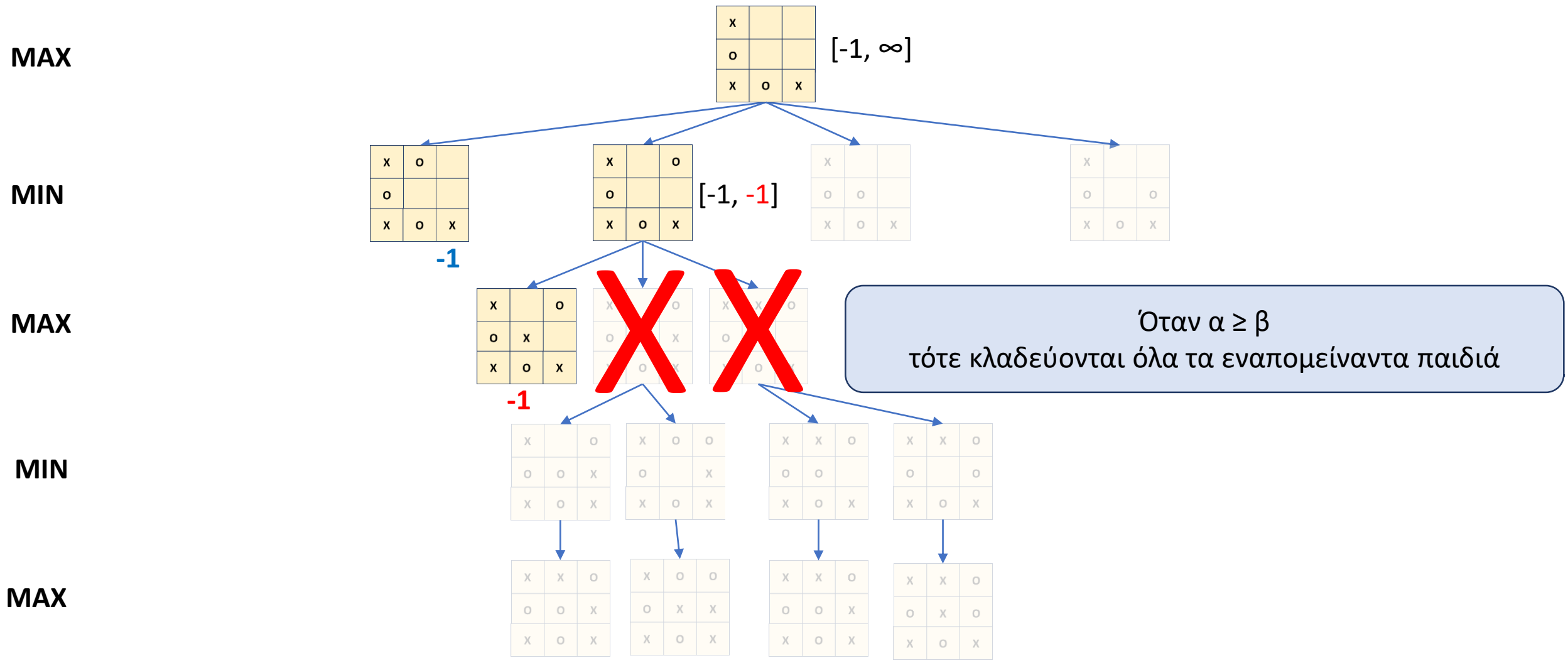
MAX

MIN

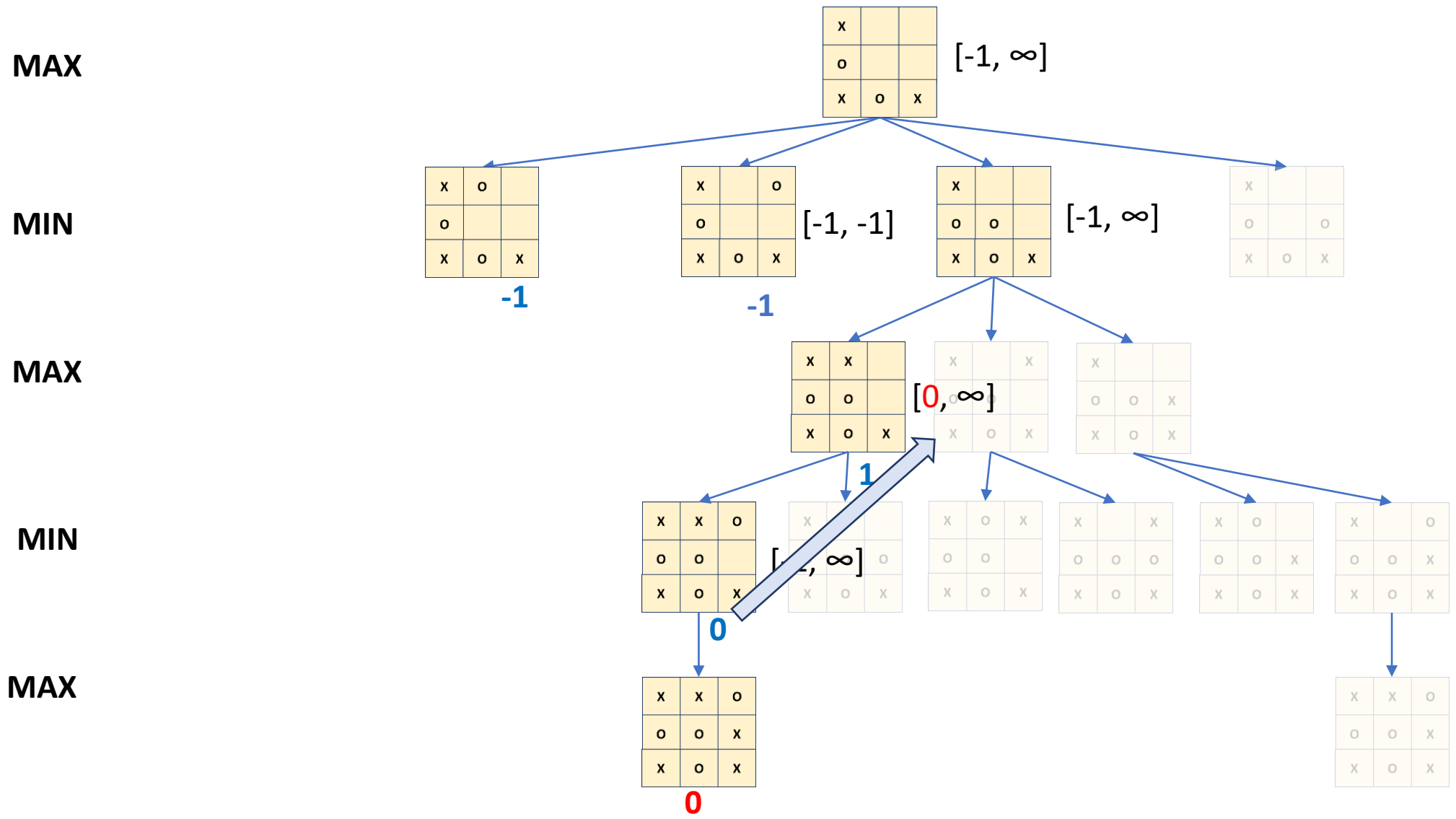
MAX



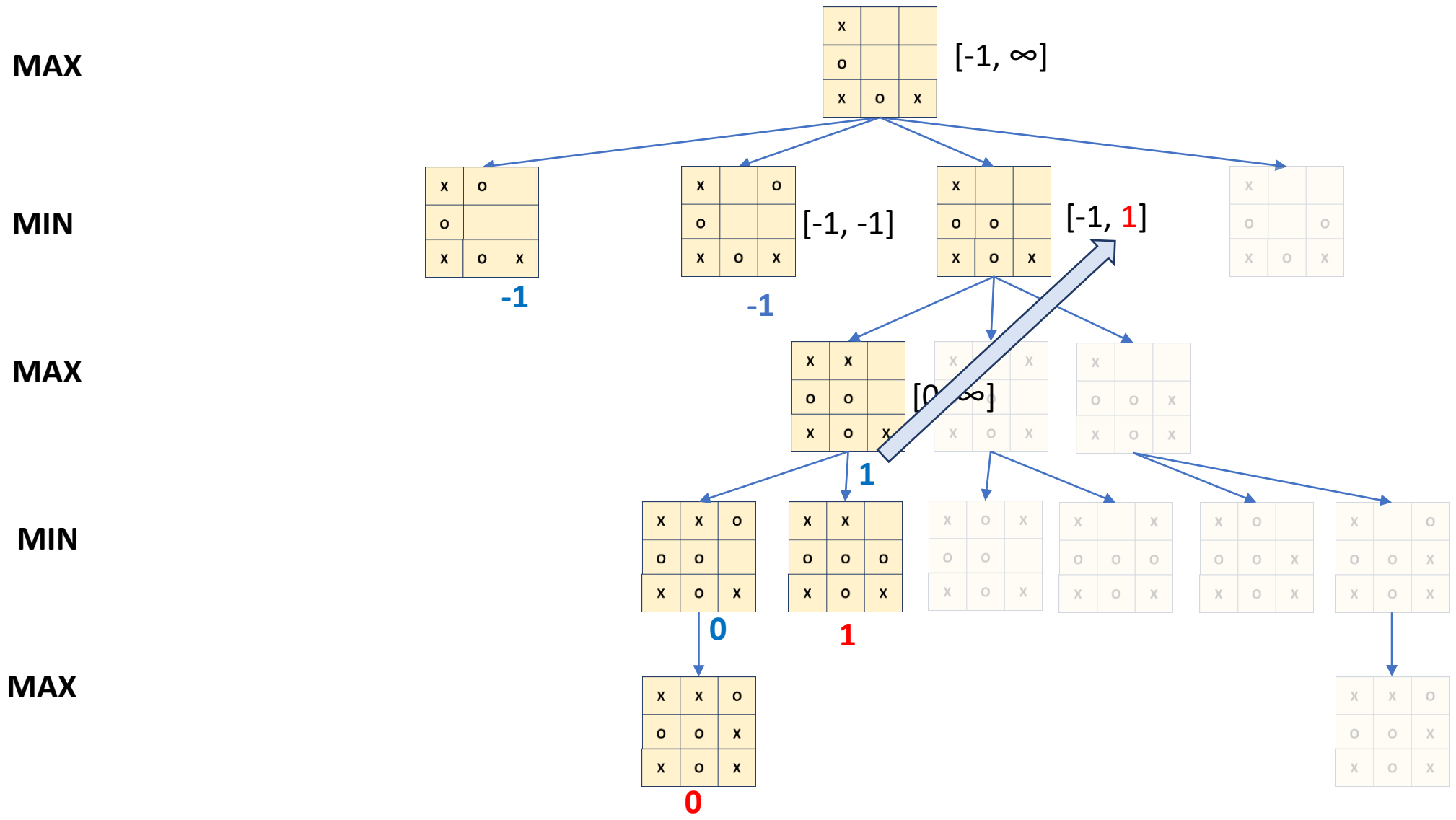
Minimax με Alpha Beta



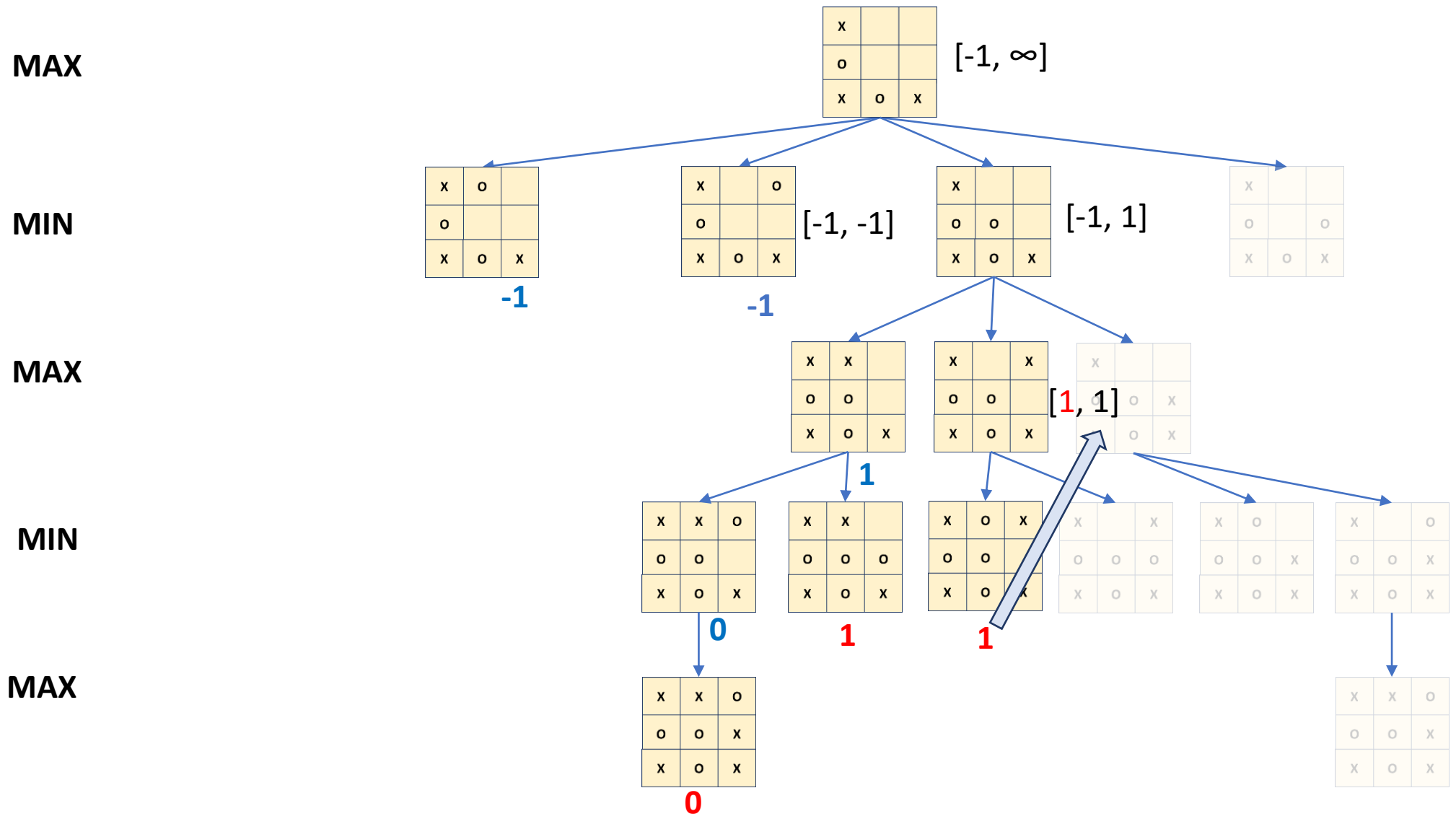
Minimax με Alpha Beta



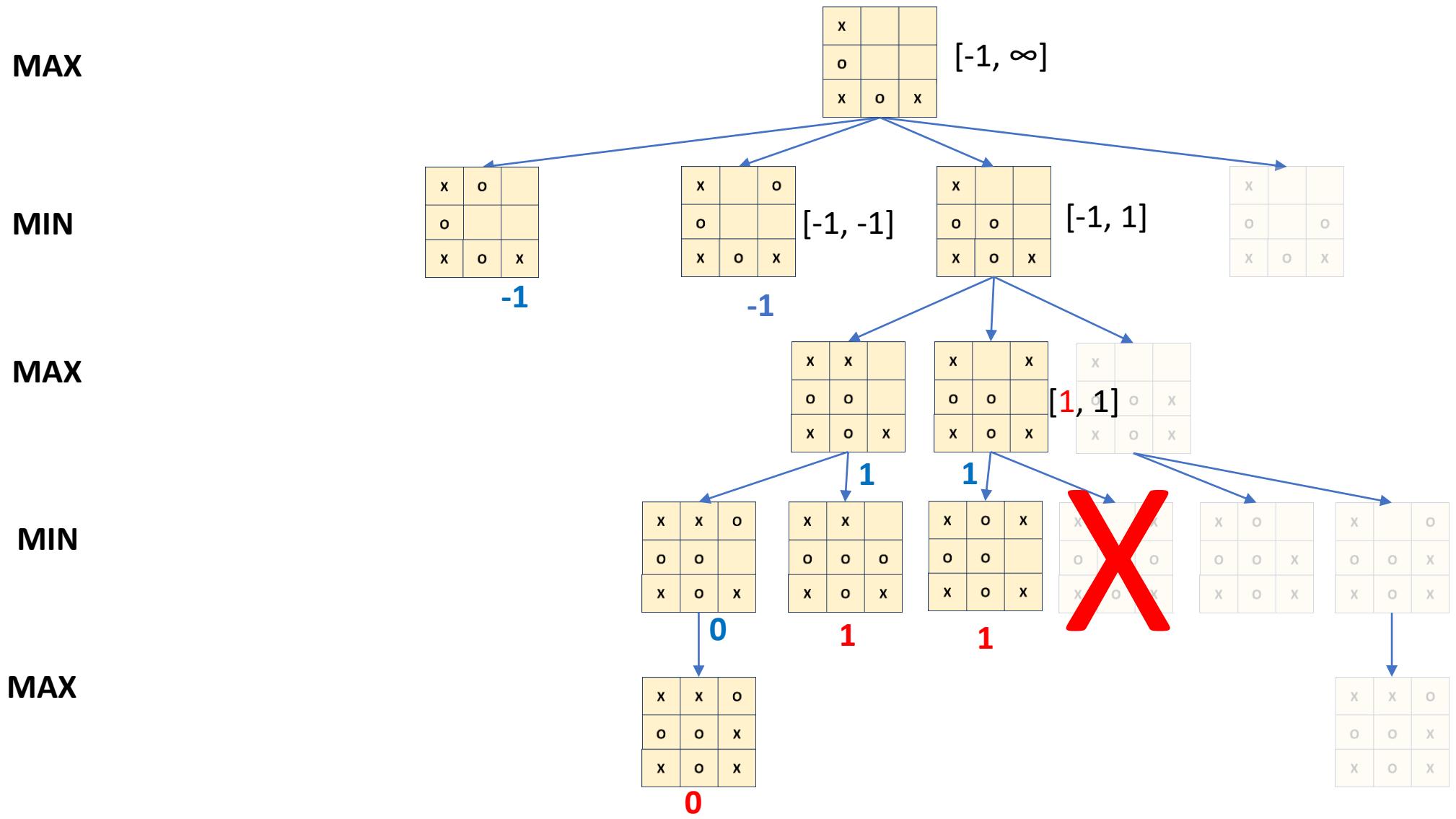
Minimax με Alpha Beta



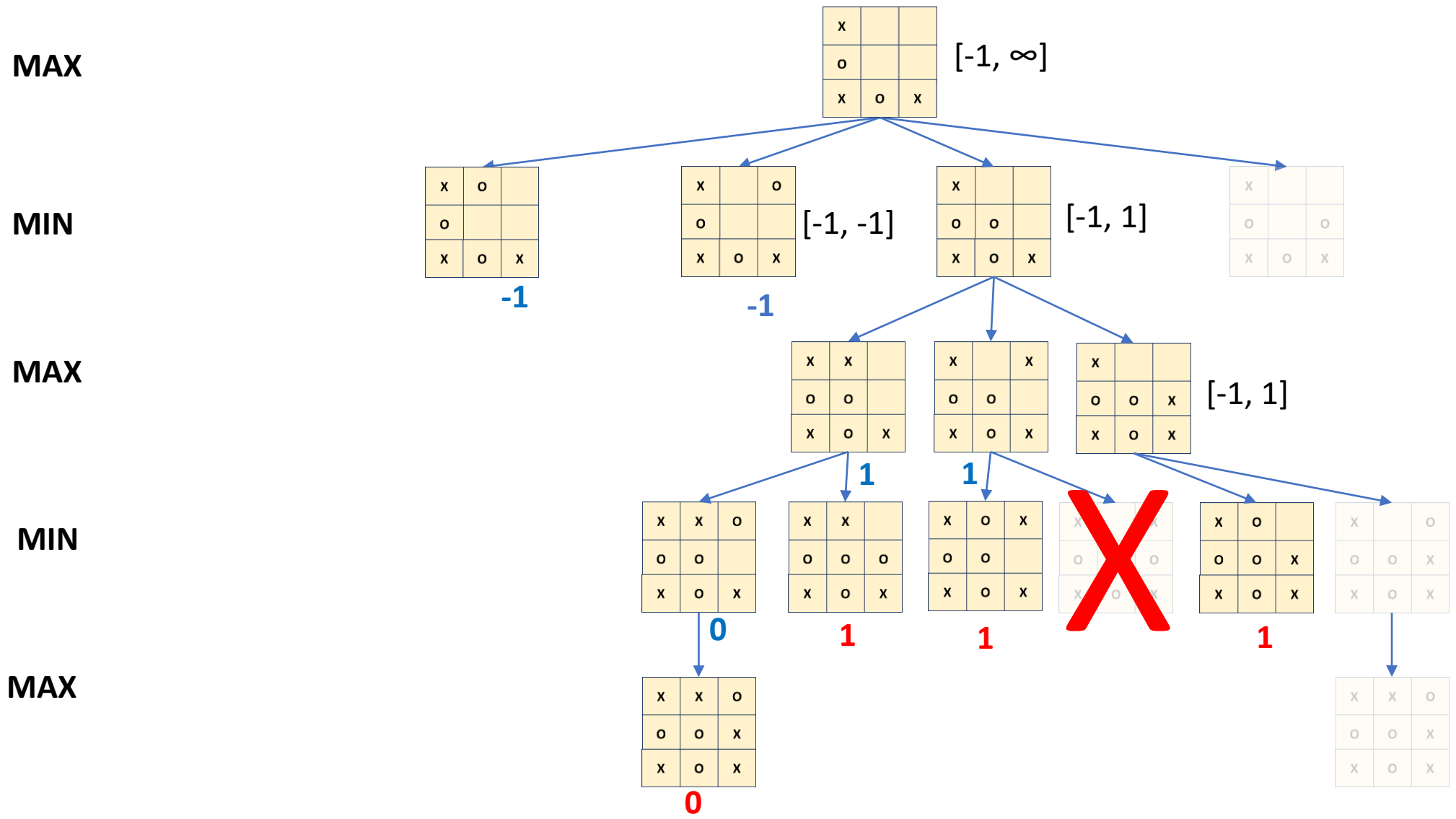
Minimax με Alpha Beta



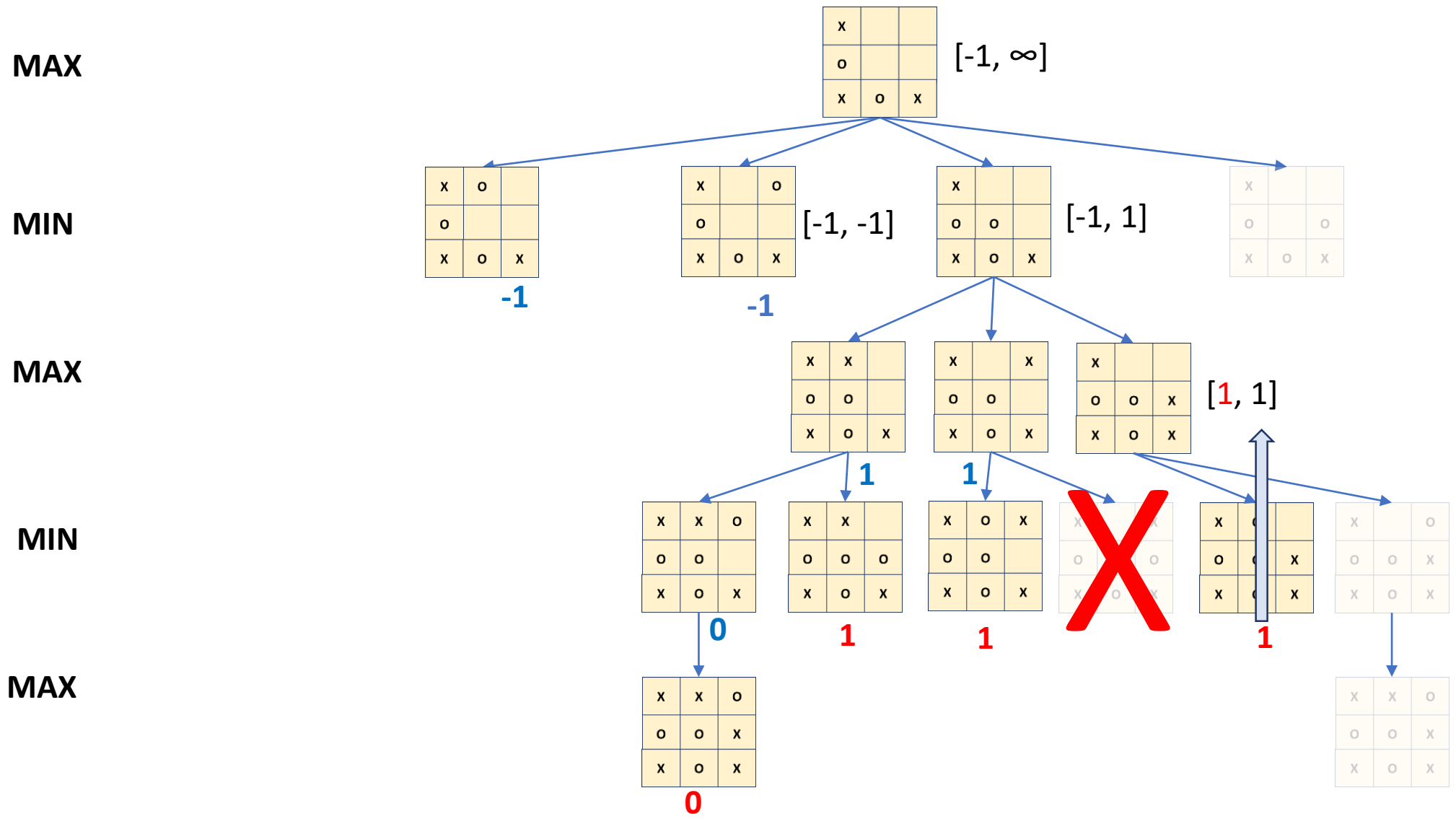
Minimax με Alpha Beta



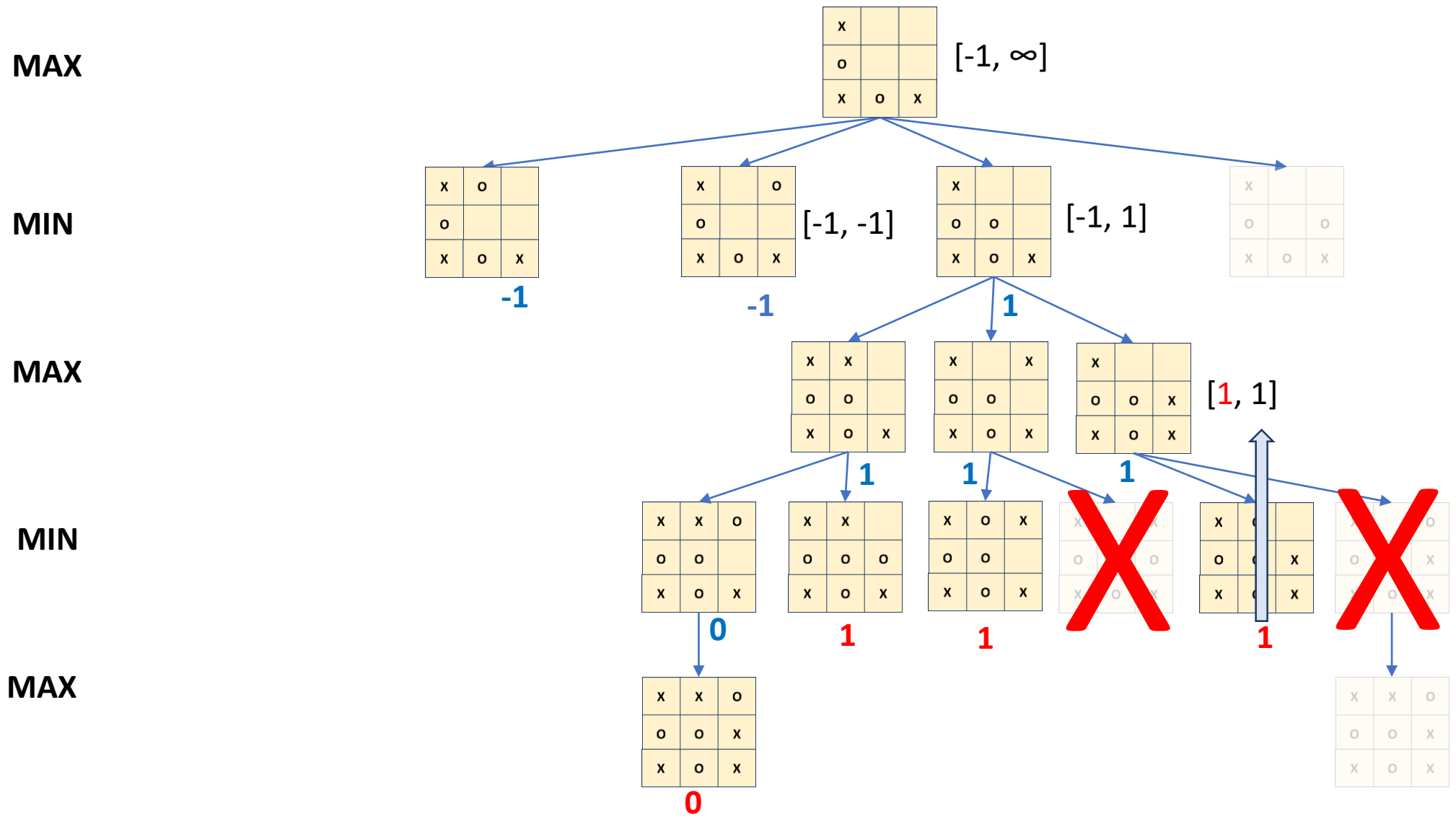
Minimax με Alpha Beta



Minimax με Alpha Beta



Minimax με Alpha Beta



Minimax με Alpha Beta

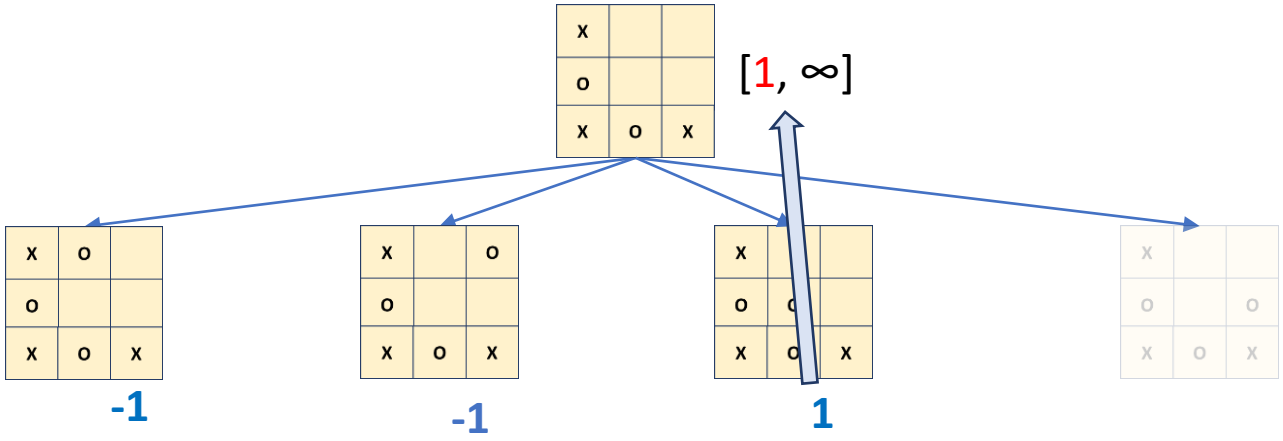
MAX

MIN

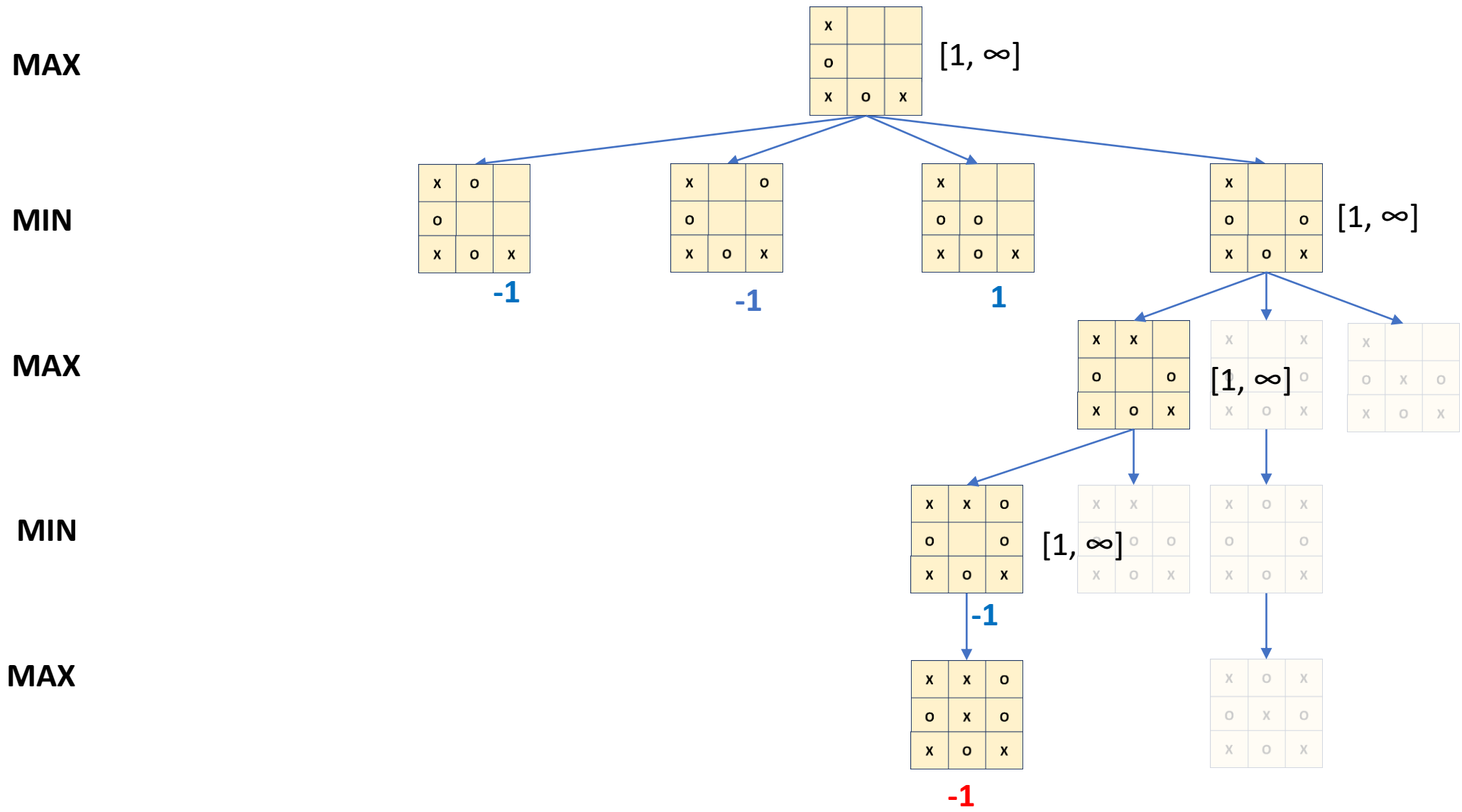
MAX

MIN

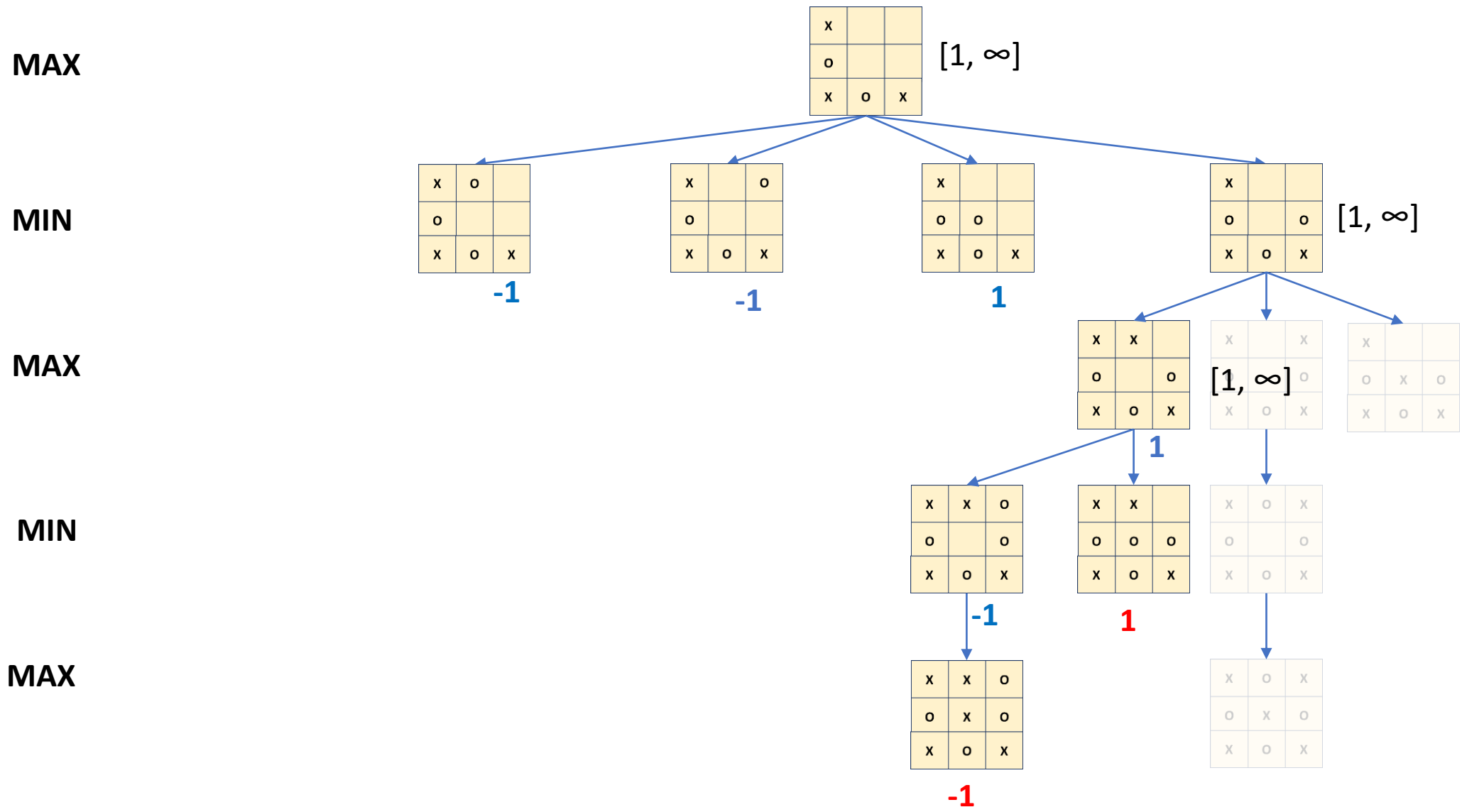
MAX



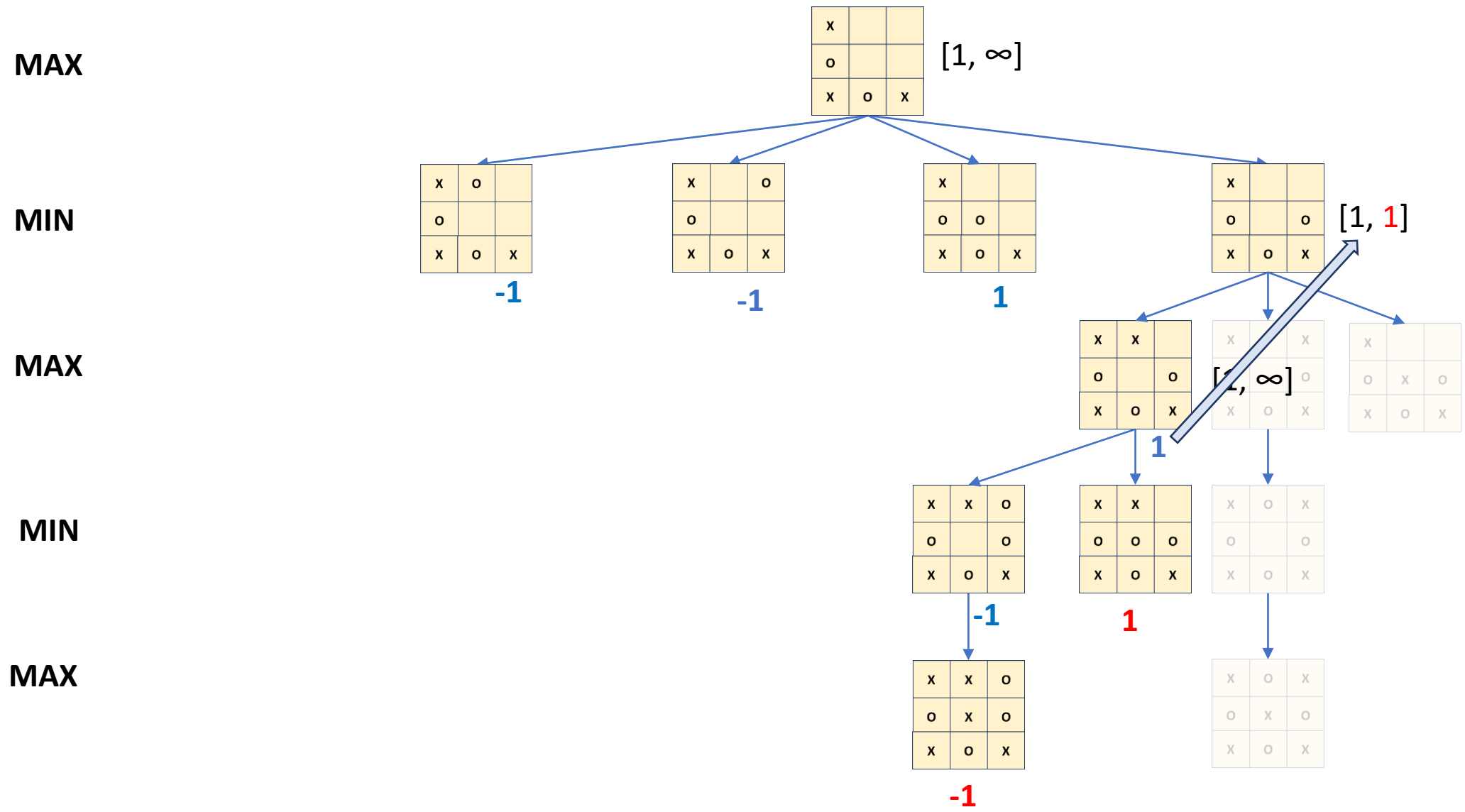
Minimax με Alpha Beta



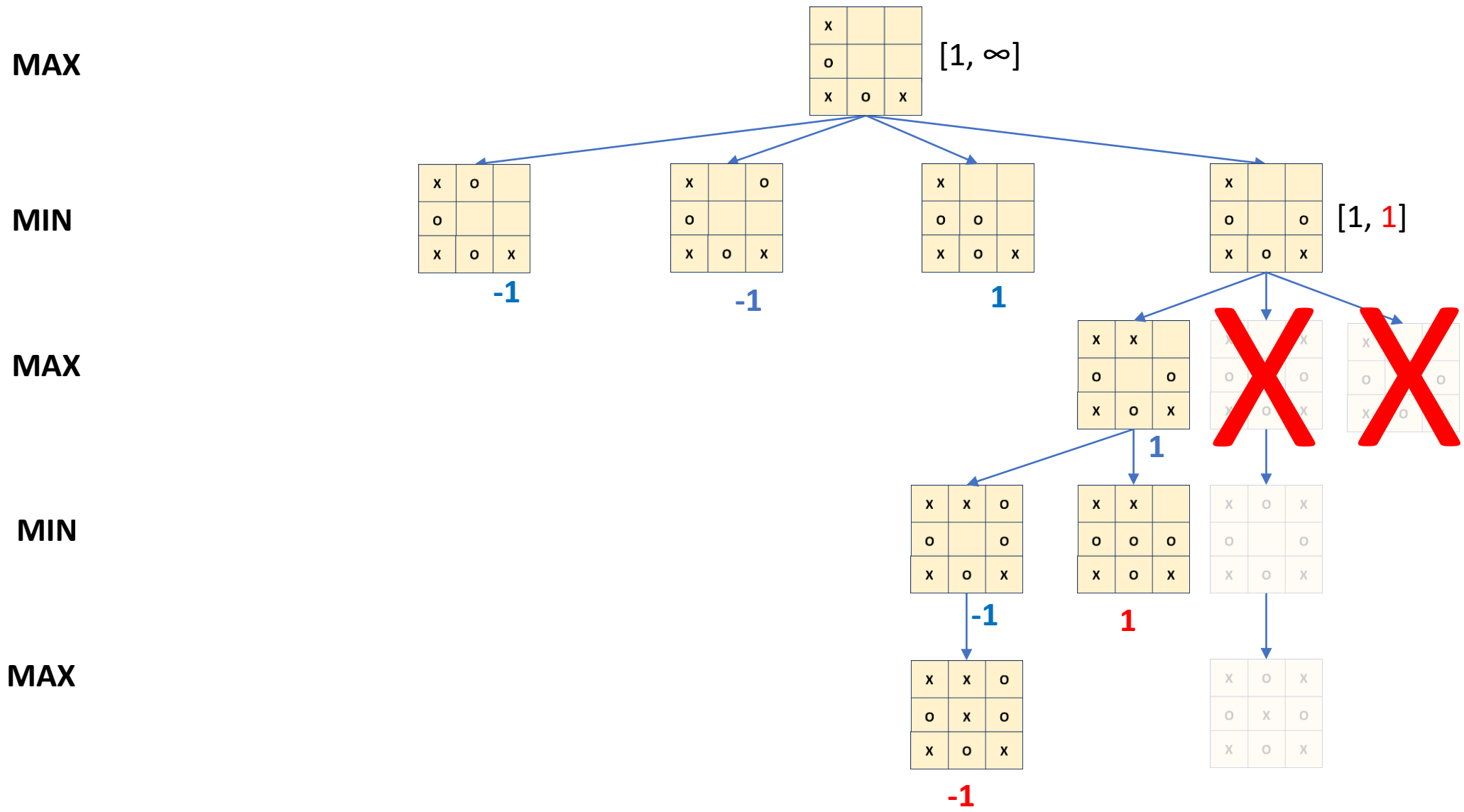
Minimax με Alpha Beta



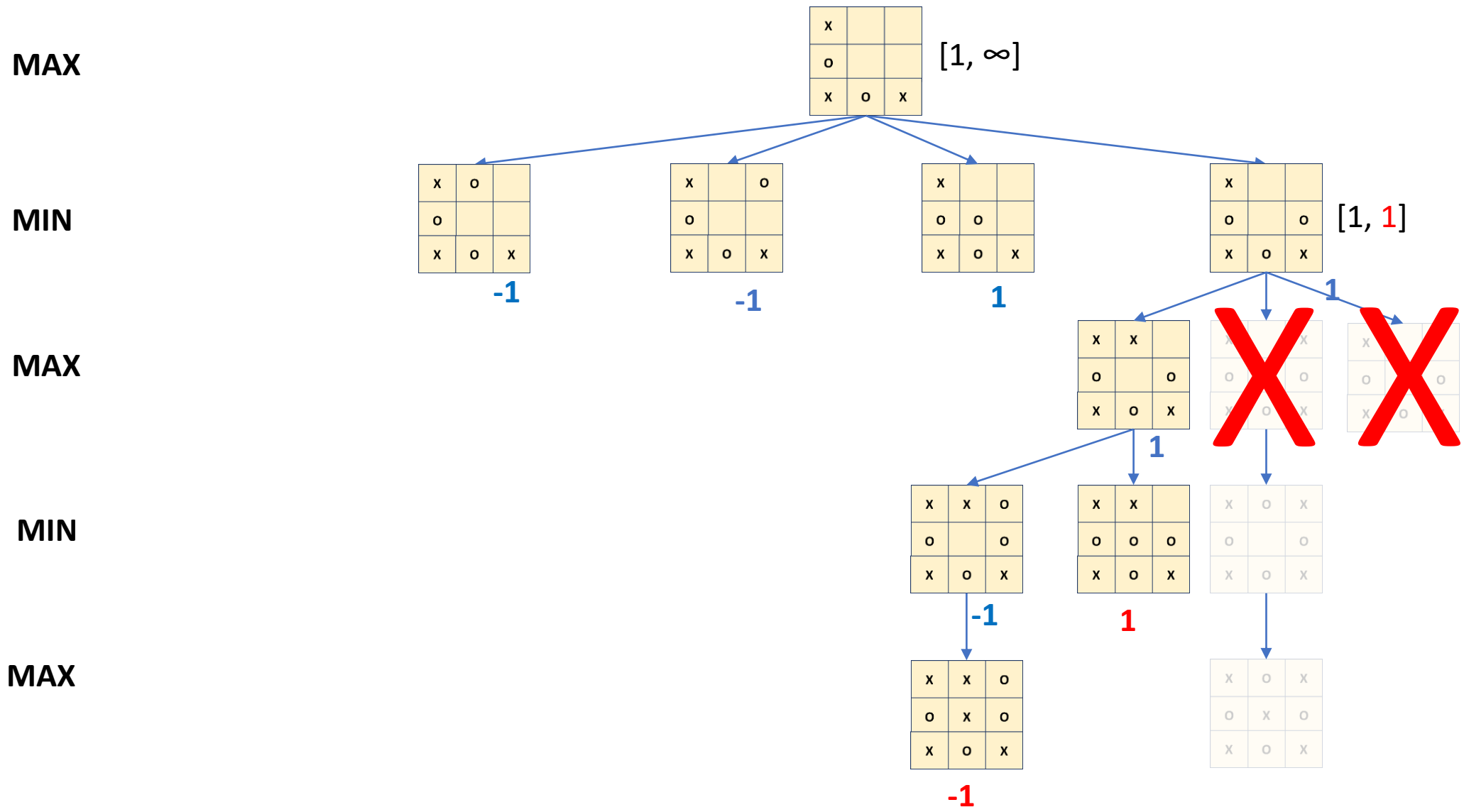
Minimax με Alpha Beta



Minimax με Alpha Beta



Minimax με Alpha Beta



Minimax με Alpha Beta

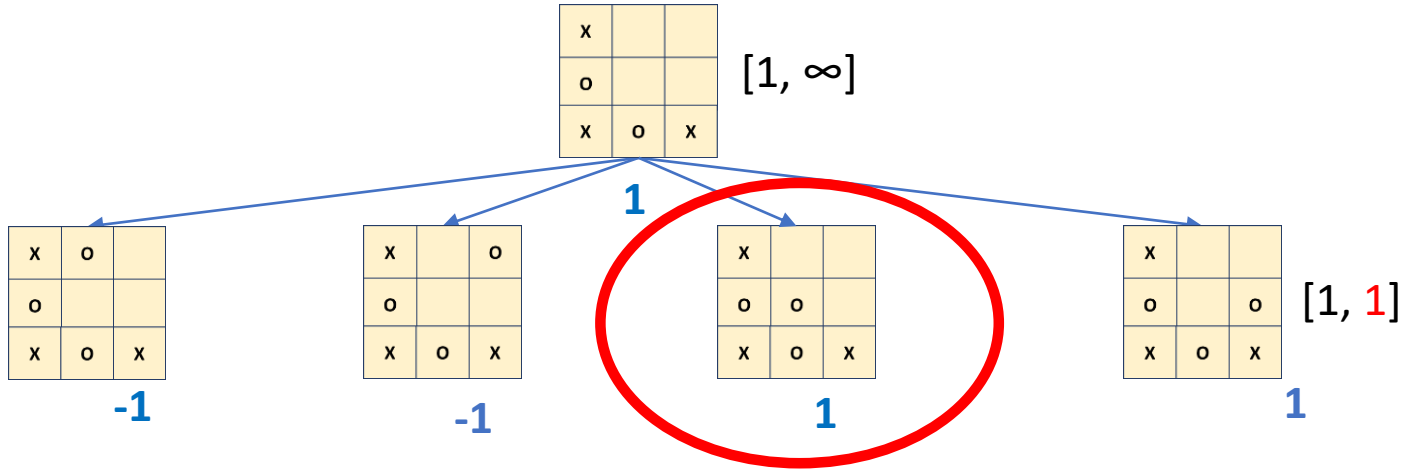
MAX

MIN

MAX

MIN

MAX



MiniMax με Alpha Beta

```
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
```

MiniMax με Alpha Beta

```
int alphabeta (TicTacToe s, int depth, bool isMax, TicTacToe &best, int alpha, int beta)
{
    int max,temp,k=s.evaluate();
    if (depth==0 || k!=-100)
    {
        best = s;
        return k;
    }
    TicTacToe maxState,tempState;
    vector<TicTacToe> children = s.expand(isMax?'X':'O');
    max=alphabeta(children[0],depth-1,!isMax,maxState,alpha,beta,count);
    if (isMax)
        alpha = max;
    else
        beta = max;
    maxState=children[0];
    for (int i=1;i<children.size();i++)
    {
        temp=alphabeta(children[i],depth-1,!isMax,tempState,alpha,beta,count);
        if ((temp>max) == isMax)
        {
            max=temp;
            maxState=children[i];
            if (isMax)
                alpha = max;
            else
                beta = max;
            if (beta<=alpha)
                break;
        }
    }
    best = maxState;
    return max;
}
```