L4. Search for Decisions in Games

- Decisions in games
- Minimax algorithm
- α-β algorithm
- Tic-Tac-Toe game

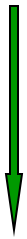
Games

Othello
Chess
TicTacToe
……

Games as search problems

Game playing is one of the oldest areas of endeavor in AI. What makes games really different is that they are usually much too hard to solve within a limited time.

For chess game: there is an average branching factor of about 35,



games often go to 50 moves by each player,

so the search tree has about 35^{100}

(there are only 10^{40} different legal position).

The result is that the complexity of games introduces a completely new kind of uncertainty that arises not because there is missing information, but because one does not have time to calculate the exact consequences of any move.

In this respect, games are much more like the real world than the standard search problems.

But we have to begin with analyzing how to find the theoretically best move in a game problem. Take a Tic-Tac-Toe as an example.

Perfect two players game

The problem is defined with the

following components:

- **initial state**: the board position, indication of whose move,
- a set of **operators**: legal moves.
- a **terminal test**: state where the game has ended.
- an **utility function**, which give a numeric value, like +1, -1, or 0.

So MAX must find a strategy, including the correct move for each possible move by Min, that leads to a terminal state that is winner and the go ahead make the first move in the sequence.

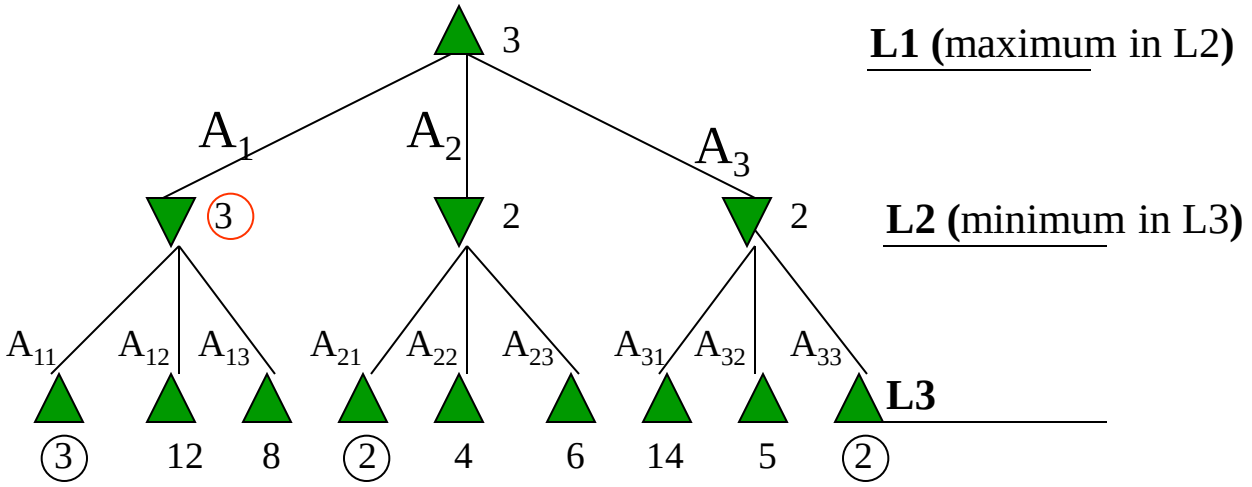
Notes: utility function is a critical component that determines which is the best move.

Minimax

ゲームは、自分にとっては最も有利な手を自分が打ち（max）、次に相手が自分にとって最も不利な手を打ち（min）、それらが交互に繰り返されることによって成り立ちます。

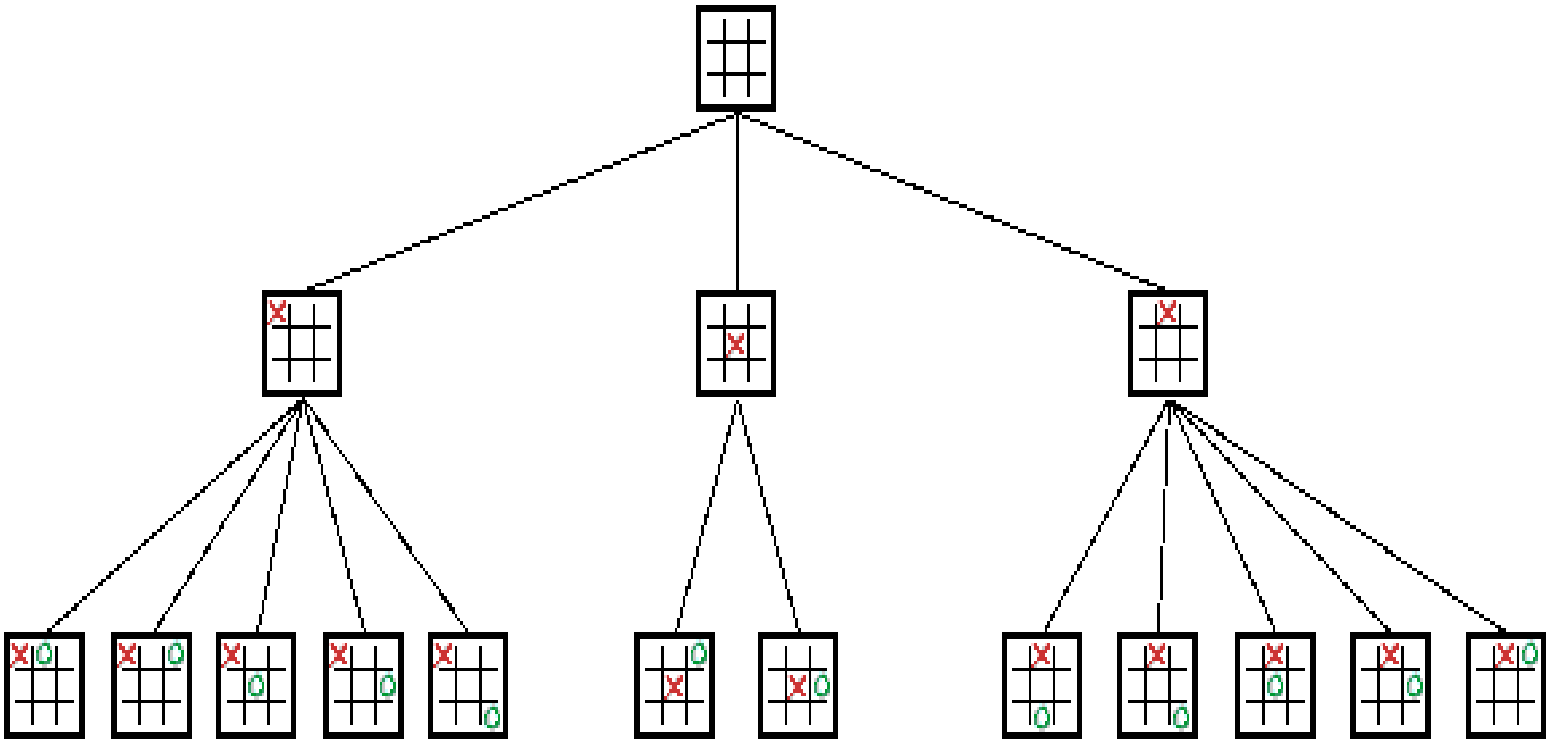
It includes five steps:

- Generate the whole game tree.
- Apply the utility function to each terminal state to get its value.
- Use the utility of the terminal states to determine the utility of the nodes one level higher up in the search tree.
- Continue backing up the values from the leaf nodes toward the root.
- MAX chooses the move that leads to the highest value.

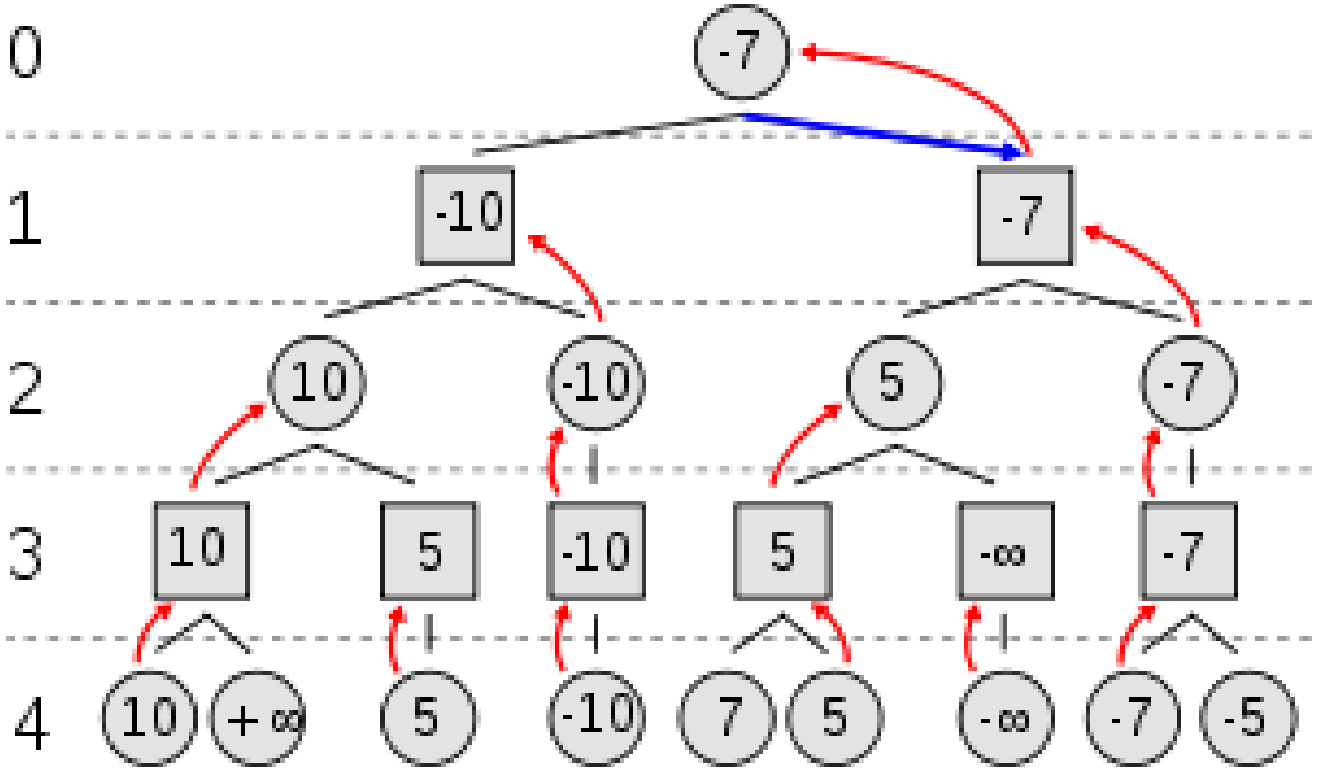


MinMax - Searching tree

- Ex: Tic-Tac-Toe (symmetrical positions removed)

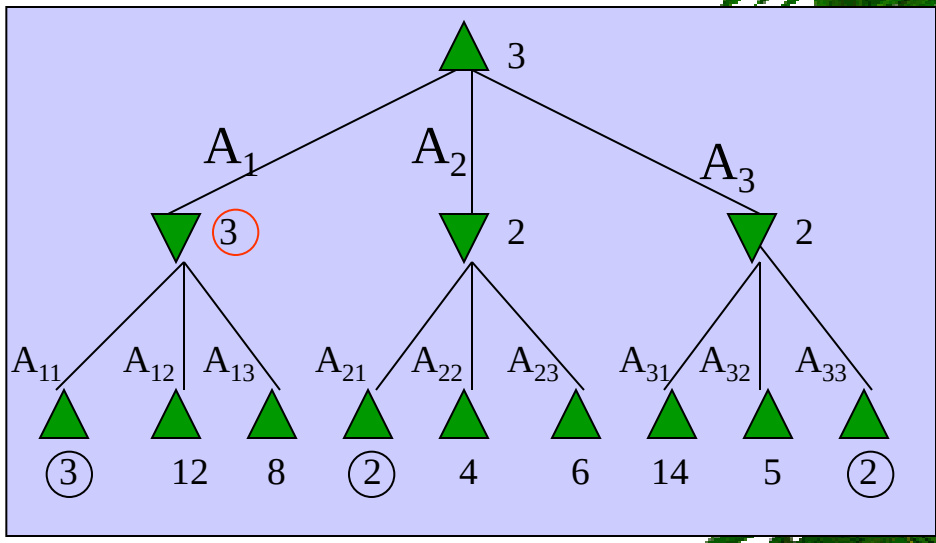


- Min-Max searching tree evaluation an example



```
MinMax (GamePosition game) {
    return MaxMove (game);
}
```

```
MaxMove (GamePosition game) {
    if (GameEnded(game)) {
        return EvalGameState(game);
    } else {
            best_move <- {};
            moves <- GenerateMoves(game);
            ForEach moves {
                move <- MinMove(ApplyMove(game));
                if (Value(move) > Value(best_move)) {
                    best_move <- move;
                }
             }
            return best_move;
            }
    }
```




```
MinMove (GamePosition game) {
    best_move <- {};
    moves <- GenerateMoves(game);
    ForEach moves {
        move <- MaxMove(ApplyMove(game));
        if (Value(move) < Value(best_move)) {
            best_move <- move;
        }
    }
    return best_move;
}
```

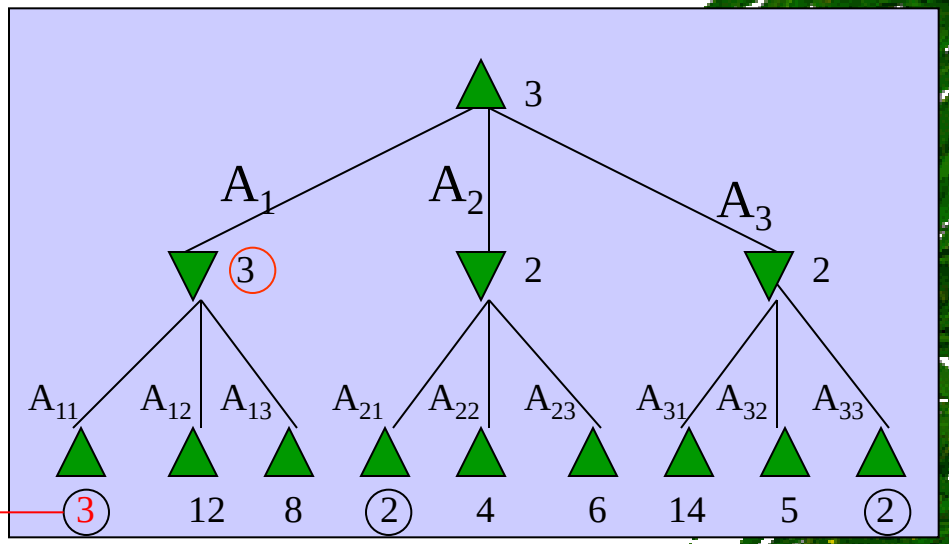
```
MinMax (GamePosition game) {
    return MaxMove (game);
}
```

```
MaxMove (GamePosition game) {
    if (GameEnded(game)) {
        return EvalGameState(game);
    } else {
            best_move <- {};
            moves <- GenerateMoves(game);
            ForEach moves {
                move <- MinMove(ApplyMove(game));
                if (Value(move) > Value(best_move)) {
                    best_move <- move;
                }
            }
            return best_move;
            }
    }
```

In A_{11} case, return 3

A_1, A_2, A_3

A_1,




```
MinMove (GamePosition game) {
    best_move <- {};
    moves <- GenerateMoves(game);
    ForEach moves {
        move <- MaxMove(ApplyMove(game));
        if (Value(move) < Value(best_move)) {
            best_move <- move;
        }
    }
    return best_move;
}
```

In A_1 case A_{11}, A_{12}, A_{13},

A_{11},

In A_{11} case, Value(A_{11}) is 3
Value(best_move) is 3
best_move is A_{11}
In A_{12} case, Value(A_{12}) is 12
Value(best_move) is 3
best_move is still A_{11}
In A_{13} case, Value(A_{13}) is 8
Value(best_move) is 3
best_move is A_{11}

To sum up:

So the MAX player will try to select the move with highest value in the end. But the MIN player also has something to say about it and he will try to select the moves that are better to him, thus minimizing MAX's outcome

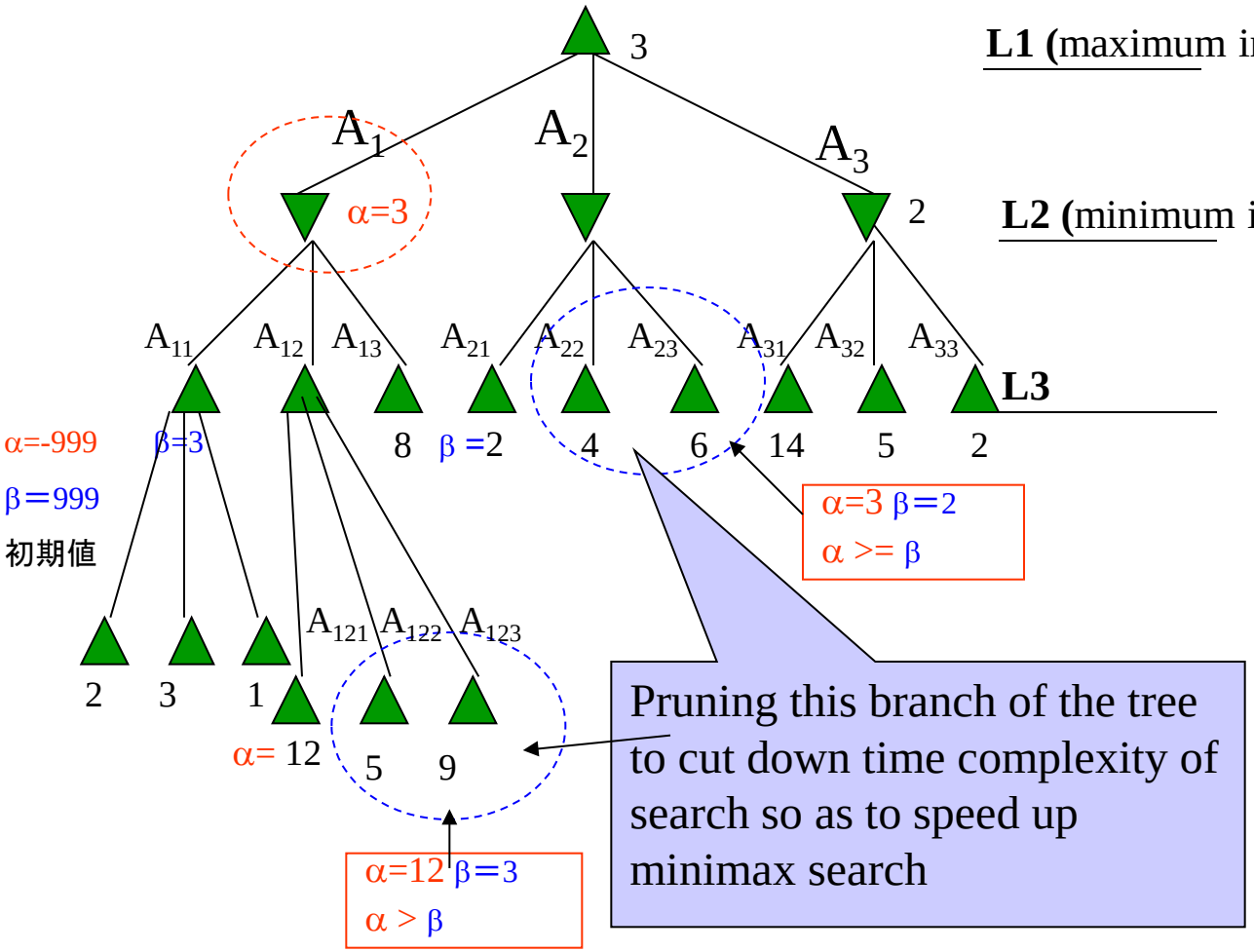
<Minimax 法>

想定される最大の損害が最小になるように決断を行う戦略。将棋やチェスなどコンピュータに思考させるためのアルゴリズムの一つ。

実行例1

α-β pruning (Alpha-Beta法)

実行例2



```
MaxMove (GamePosition game, Integer alpha, Integer beta) {
  if (GameEnded(game) || DepthLimitReached()) {
    return EvalGameState(game, MAX);
  }
  else {
    best_move <- {};
    moves <- GenerateMoves(game);
    ForEach moves {
      move <- MinMove(ApplyMove(game), alpha, beta);
      if (Value(move) > Value(best_move)) {
        best_move <- move;
        alpha <-  Value(move);
      }
      // Ignore remaining moves
      if (alpha >= beta)
        return best_move;
    }
    return best_move;
  }
}
```

α=12 β=3
α >= β

```
MinMove (GamePosition game, Integer alpha, Integer beta) {
  if (GameEnded(game) || DepthLimitReached()) {
    return EvalGameState(game, MIN);
  }
  else {
    best_move <- {};
    moves <- GenerateMoves(game);
    ForEach moves {
      move <- MaxMove(ApplyMove(game), alpha, beta);
      if (Value(move) < Value(best_move)) {
        best_move <- move;
        beta <- Value(move);
      }
      // Ignore remaining moves
      if (beta < = alpha)
        return best_move;
    }
    return best_move;
  }
}
```

α=3 β=2
α >= β

まとめ

ゲームは、自分にとっては最も有利な手を自分が打ち（max）、次に相手が自分にとって最も不利な手を打ち（min）、それらが交互に繰り返されることによって成り立ちます。

<αーβ　法（狩り>

Minimaxを改良したもの。枝刈りを行うことでMinimaxより評価するノードを抑えている

<Minimax　algorithmとαーβ　algorithmの違い>

Minimax法ではすべてを探索し最良の手を選択するのに対して、α-β法は、minimax法で採用されないと判断された手については、そこから先を探索しないことで無駄な探索に費やす時間をカットしている。また、α-β法による結果はminimax法での結果と同じになる。

枝刈りを行うことにより探索がminimax法より早く終わるのでα-β法のほうが効率的である。

Tic Tac Toe game

In SymTic.java

```
// 評価する.
public int evaluate(int depth, int level, int refValue) {
   int e = evaluateMyself();
   if ((depth==0)||(e==99)||(e==-99)||((e==0)&&(Judge.finished(this)))) {
     return e;
   } else if (level == MAX) {
     int maxValue = -999;
     Vector v_child = this.children(usingChar);
     for (int i=0; i<v_child.size(); i++) {
       SymTic st = (SymTic)v_child.elementAt(i); //st is a move
       int value = st.evaluate(depth, MIN, maxValue);
       if (value > maxValue ) {
         maxChild = st;
         maxValue = value; //maxValue = α
       }
       if (value > = refValue) { //refValue =  β
         return value;
       }
     }
     return maxValue;
   }
```

```
else {
      int minValue = 999;
      Vector v_child = this.children('o');
      for (int i=0; i<v_child.size(); i++) {
        SymTic st = (SymTic)v_child.elementAt(i);
        int value = st.evaluate(depth-1, MAX, minValue);
        if (value < minValue) {
          minValue = value; // minValue = β
        }
        if (value <= refValue) { // refValue =   α
          return value;
        }   }
      return minValue;
   }}
```

```
private int evaluateMyself() {
   char c = Judge.winner(this);
   if (c == usingChar) {   //win the game
     return 99;
   } else if (c != ' ') {       //lose the game
     return -99;
   } else if (Judge.finished(this)) {    //draw the game
     return 0;
   }
```

Home work,

Understand TicTacToe Game program and try to run it.

(Optional) Take a look of two web sites and try to make your own Othello game

Othello Game (Min-Max):

http://aidiary.hatenablog.com/entry/20041226/1274148758

http://uguisu.skr.jp/othello/minimax.html

α－β

http://hp.vector.co.jp/authors/VA015468/platina/algo/2_3.html

（**Optional**）You may try to make a chess game

Java Chess Engine

Minimax and Alpha-Beta Pruning
https://www.youtube.com/watch?v=fJ4uQpkn9V0
α－β algorithm
https://www.youtube.com/watch?v=fJ4uQpkn9V0
(part 1)
https://www.youtube.com/watch?v=Wyh-5P5-7U8
(part 2)
https://www.youtube.com/watch?v=8xBjxYHVwxM
Verifying an Alpha-Beta Algorithm works Correctly