Chapter 6
Adversarial Search

CS 461 – Artificial Intelligence

Pinar Duygulu

Bilkent University, Spring 2008

Slides are mostly adapted from AIMA and MIT Open Courseware

Outline

- Games
- Optimal decisions
- Minimax algorithm
- α-β pruning
- Imperfect, real-time decisions

Games

- Multi agent environments : any given agent will need to consider the actions of other agents and how they affect its own welfare.
- The unpredictability of these other agents can introduce many possible contingencies
- There could be competitive or cooperative environments

- Competitive environments, in which the agent's goals are in conflict require adversarial search – these problems are called as games

Games

- In game theory (economics), any multiagent environment (either cooperative or competitive) is a game provided that the impact of each agent on the other is significant
- AI games are a specialized kind - deterministic, turn taking, two-player, zero sum games of perfect information
- In our terminology – deterministic, fully observable environments with two agents whose actions alternate and the utility values at the end of the game are always equal and opposite (+1 and –1)

Games – history of chess playing

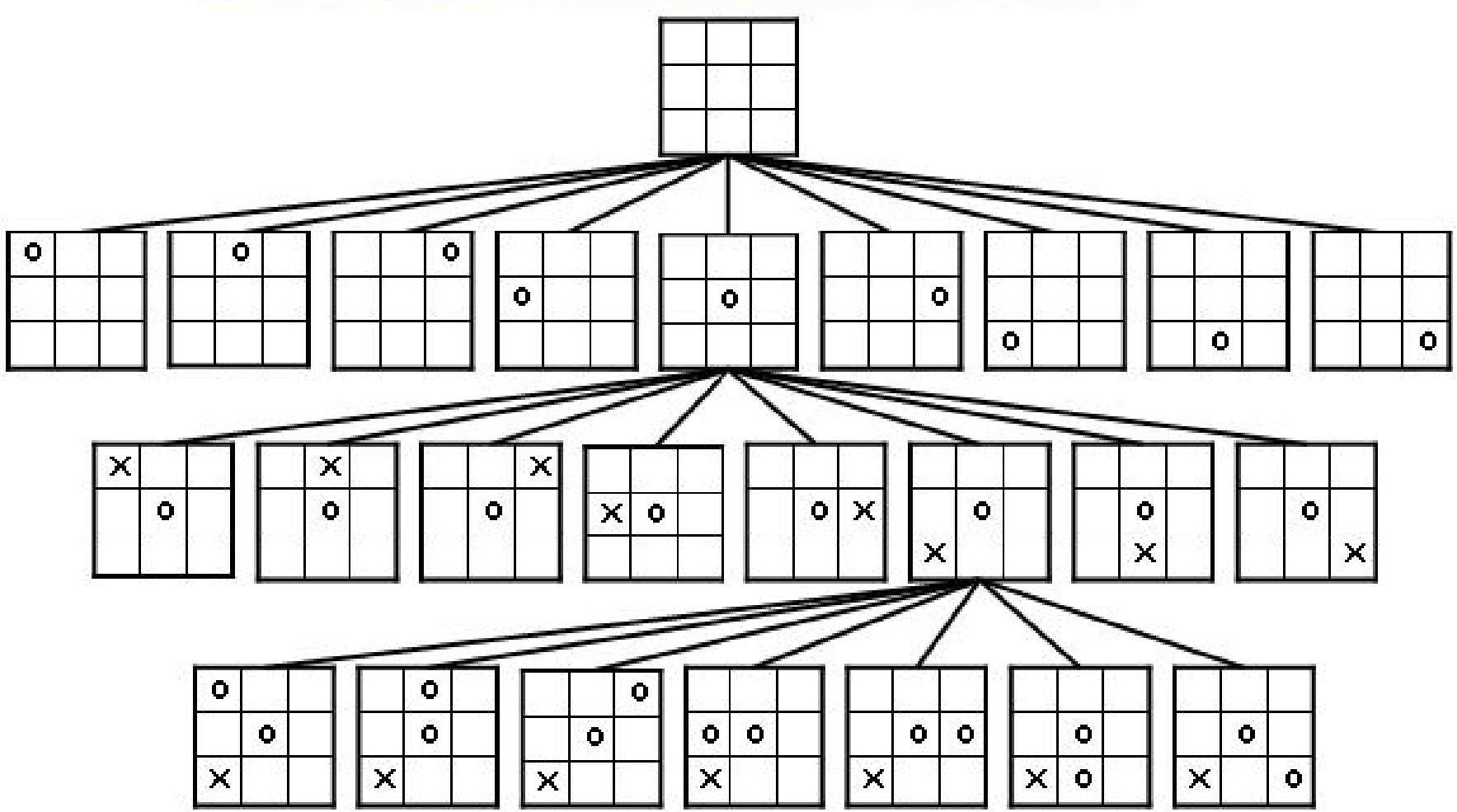
- 1949 – Shannon paper – originated the ideas
- 1951 – Turing paper – hand simulation
- 1958 – Bernstein program
- 1955-1960 – Simon-Newell program
- 1961 – Soviet program
- 1966 – 1967 – MacHack 6 – defeated a good player
- 1970s – NW chess 4.5
- 1980s – Cray Bitz
- 1990s – Belle, Hitech, Deep Thought,
- 1997 - Deep Blue - defeated Garry Kasparov

Game Tree search

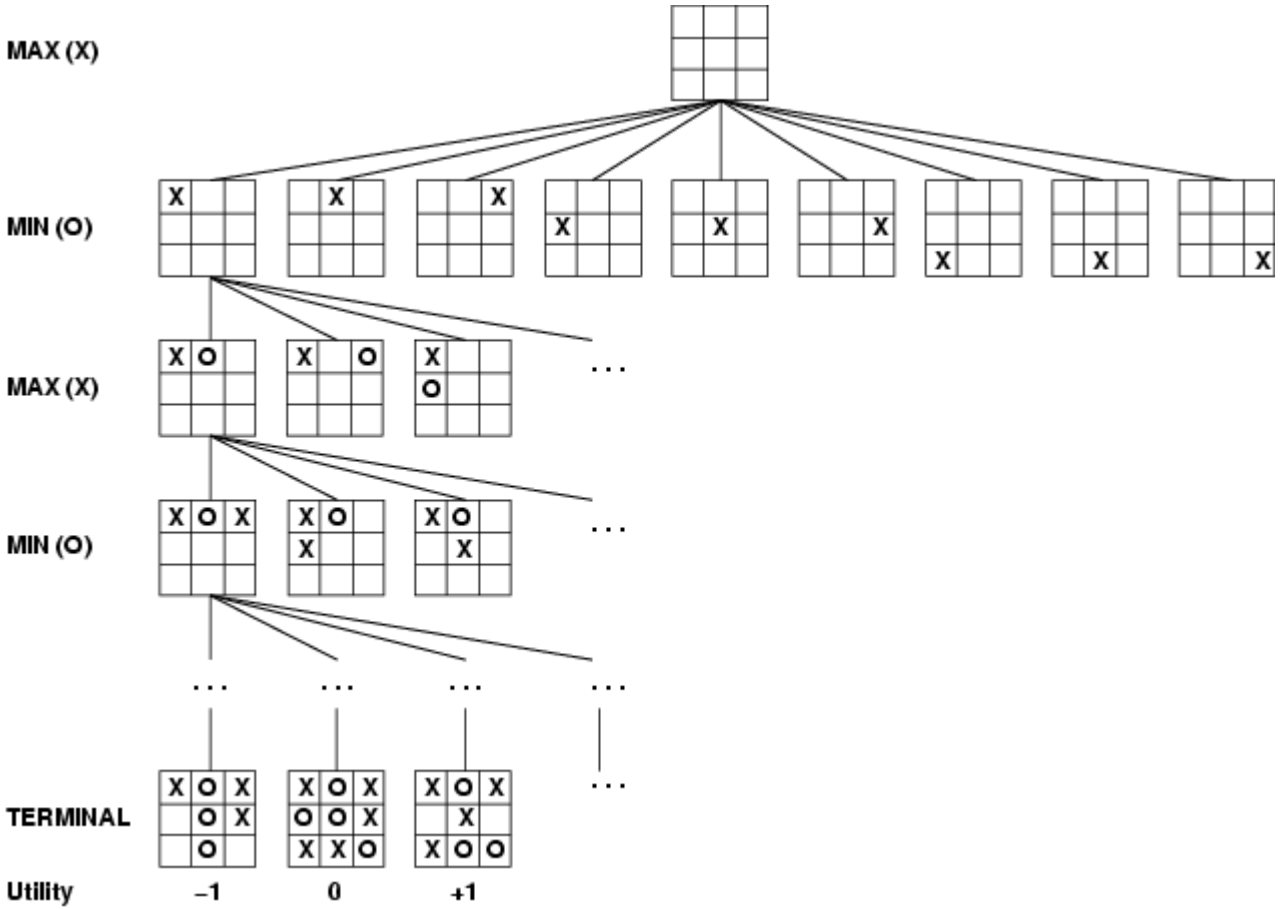
- Initial state: initial board position and player
- Operators: one for each legal move
- Goal states: winning board positions
- Scoring function: assigns numeric value to states
- Game tree: encodes all possible games

- We are not looking for a path, only the next move to make (that hopefully leads to a winning position)
- Our best move depends on what the other player does

Partial Game Tree for Tic-Tac-Toe



Game tree (2-player, deterministic, turns)

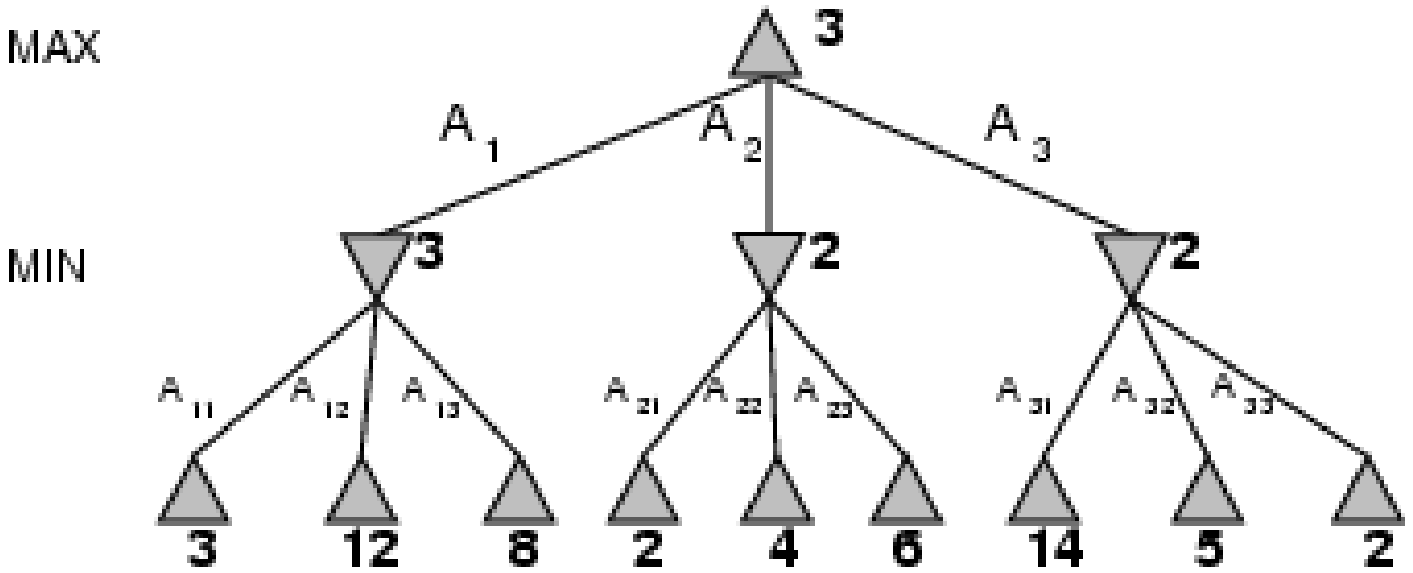


Optimal strategies

- In a normal search problem, the optimal solution would be a sequence of moves leading to a goal state - a terminal state that is a win
- In a game, MIN has something to say about it and therefore MAX must find a contingent strategy, which specifies
  - MAX's move in the initial state,
  - then MAX's moves in the states resulting from every possible response by MIN,
  - then MAX's moves in the states resulting from every possible response by MIN to those moves
  - …
- An optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent

Minimax

- Perfect play for deterministic games
- Idea: choose move to position with highest minimax value
  = best achievable payoff against best play
- E.g., 2-ply game:



Minimax value

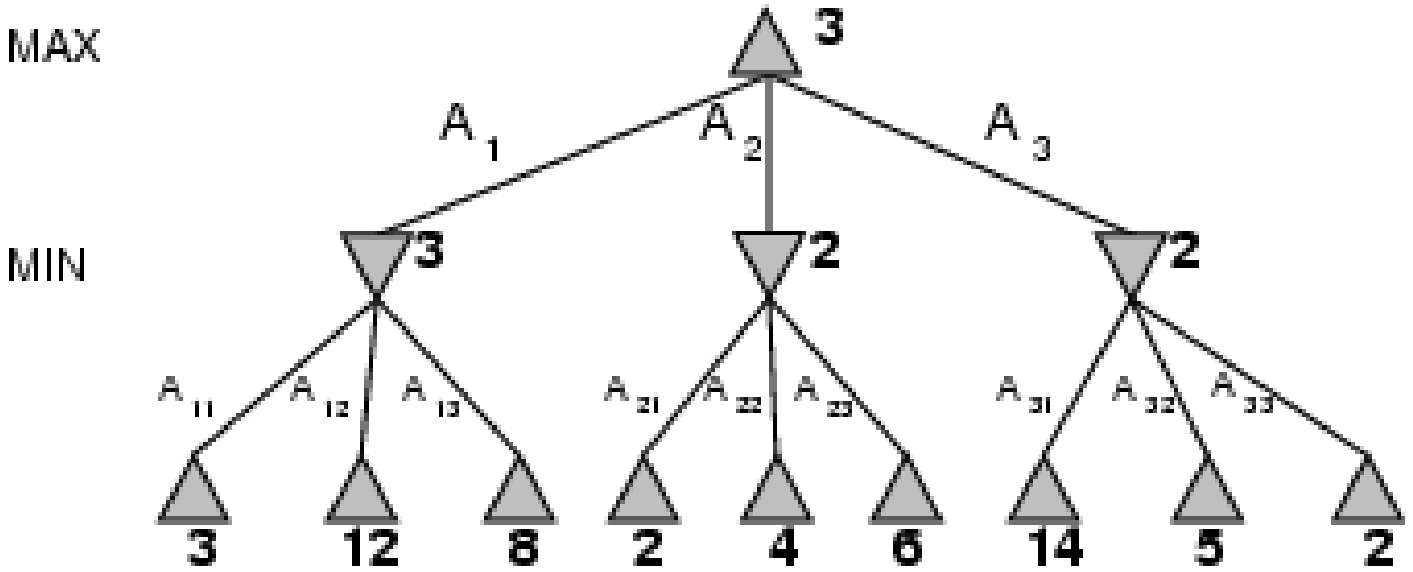
- Given a game tree, the optimal strategy can be determined by examining the minimax value of each node (MINIMAX-VALUE(n))
- The minimax value of a node is the utility of being in the corresponding state, assuming that both players play optimally from there to the end of the game
- Given a choice, MAX prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Minimax

MINIMAX-VALUE(root) = max(min(3,12,8), min(2,4,6), min(14,5,2))
= max(3,2,2)
= 3

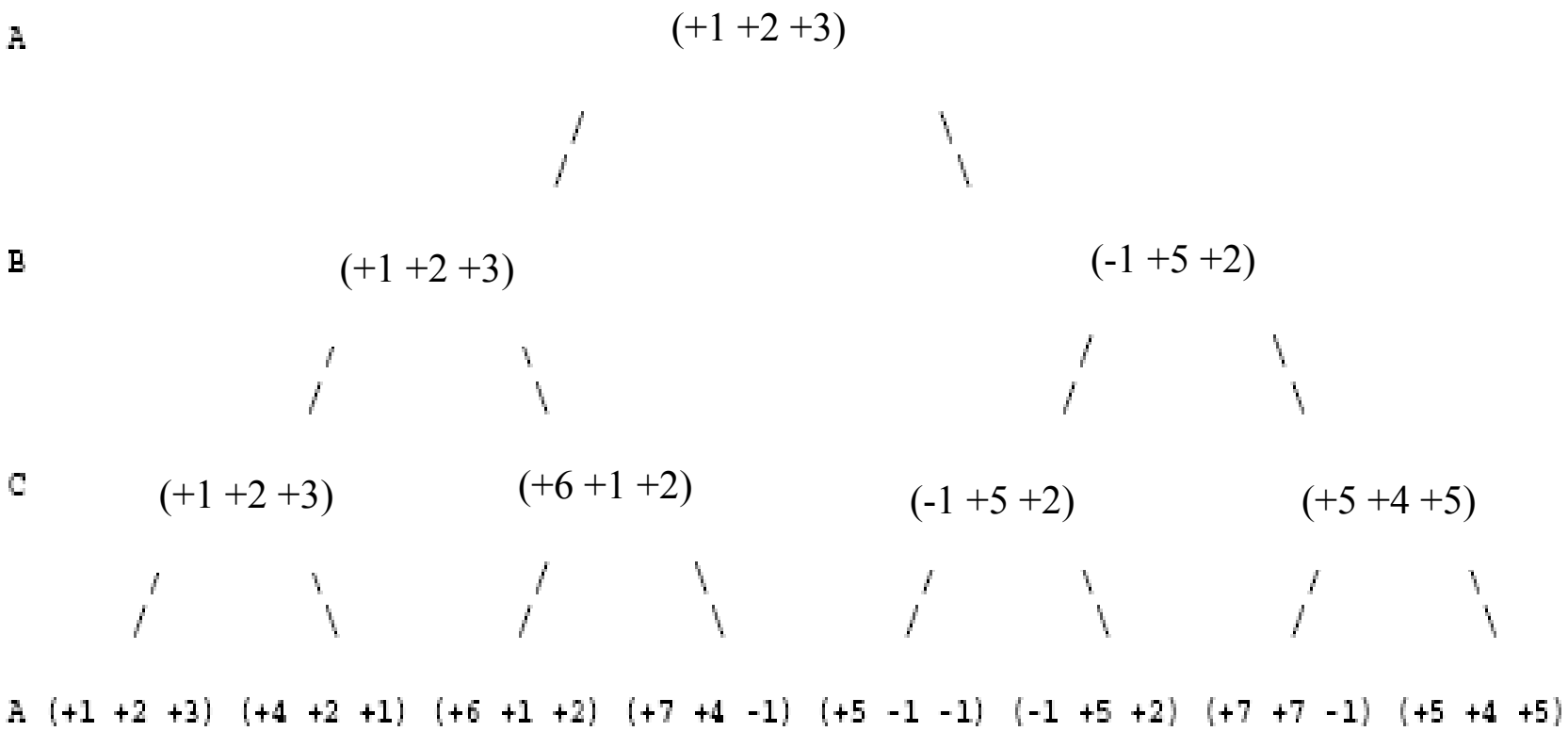


Properties of minimax

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)

- For chess, b ≈ 35, m ≈100 for "reasonable" games
  → exact solution completely infeasible

Tree Player and Non-zero sum games



α-β pruning

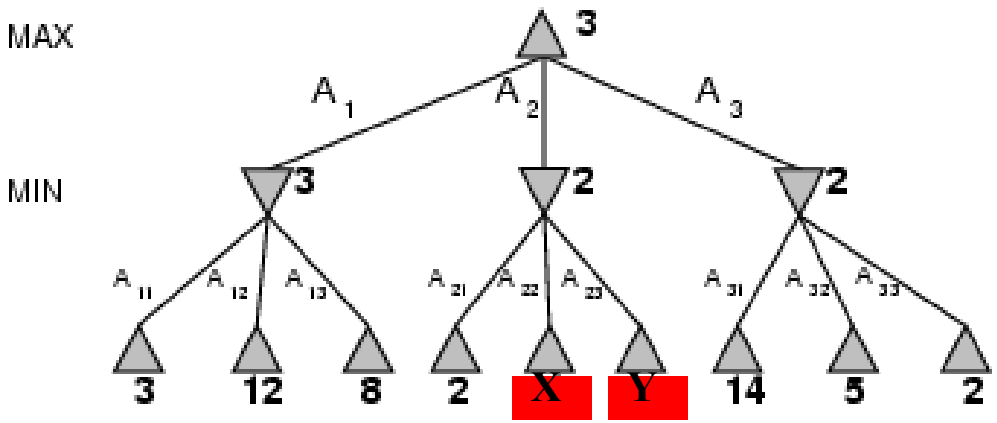
- It is possible to compute the correct minimax decision without looking at every node in the game tree

MINIMAX-VALUE(root) = max(min(3,12,8), min(2,x,y), min(14,5,2))

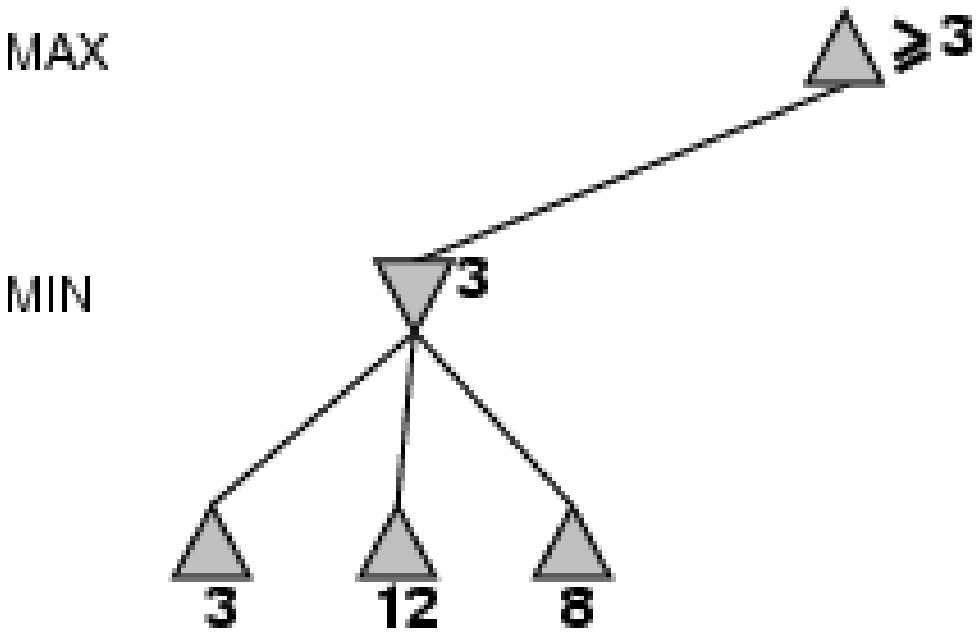
= max(3,min(2,x,y),2)

= max(3,z,2) where z <=2

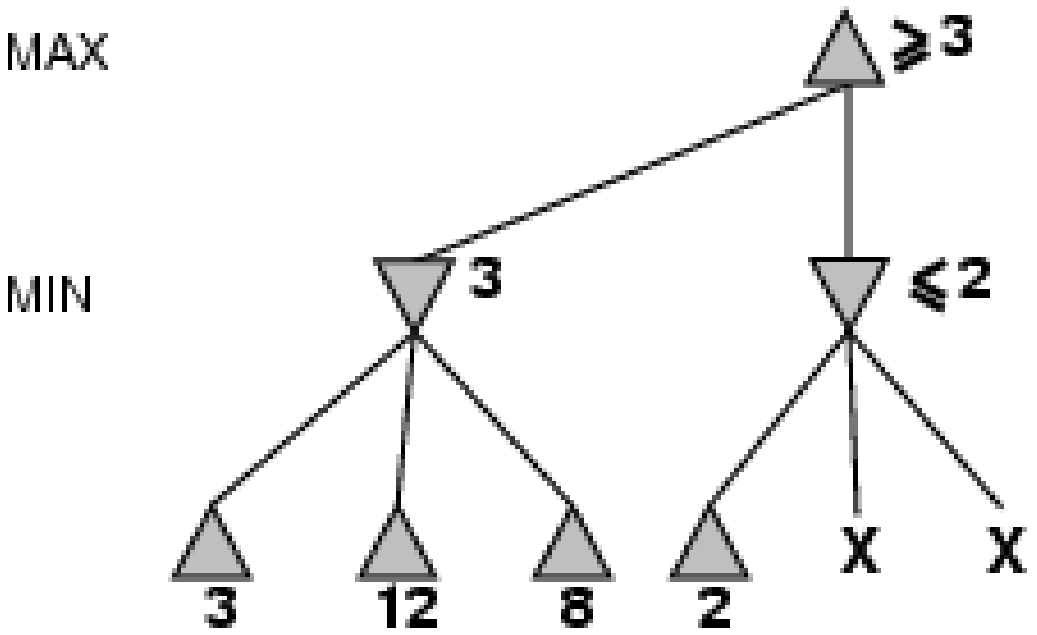
= 3



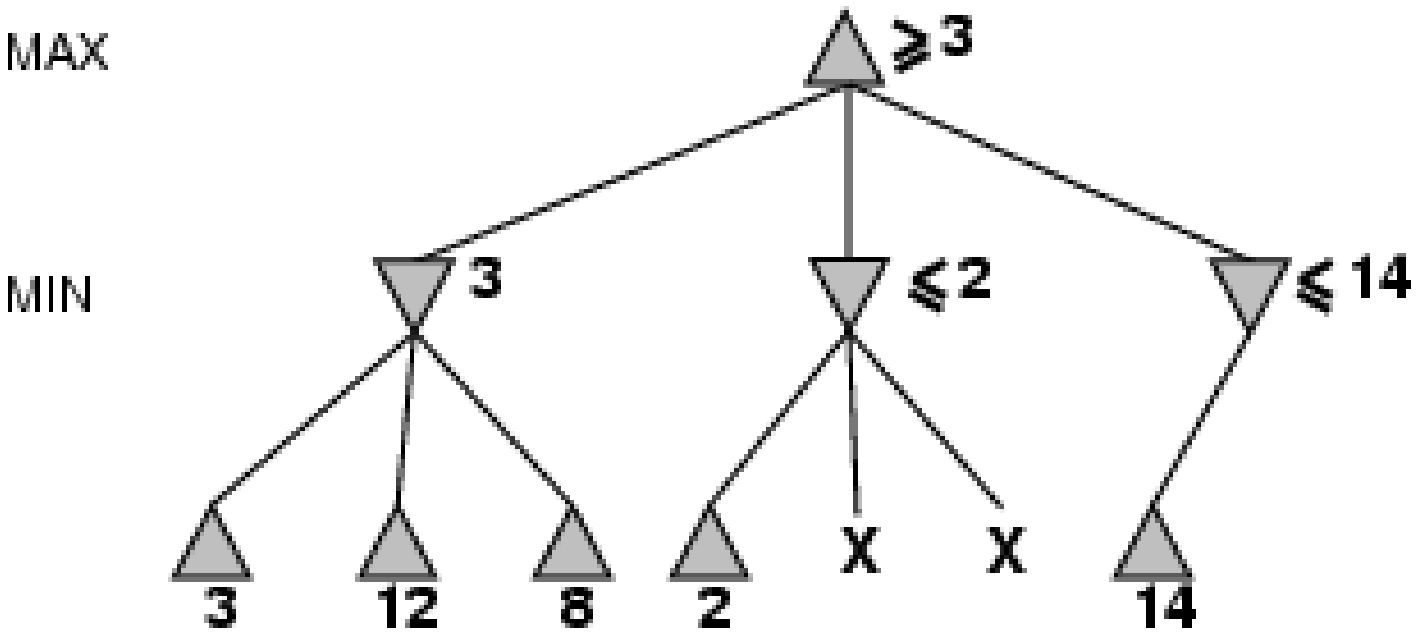
α-β pruning example



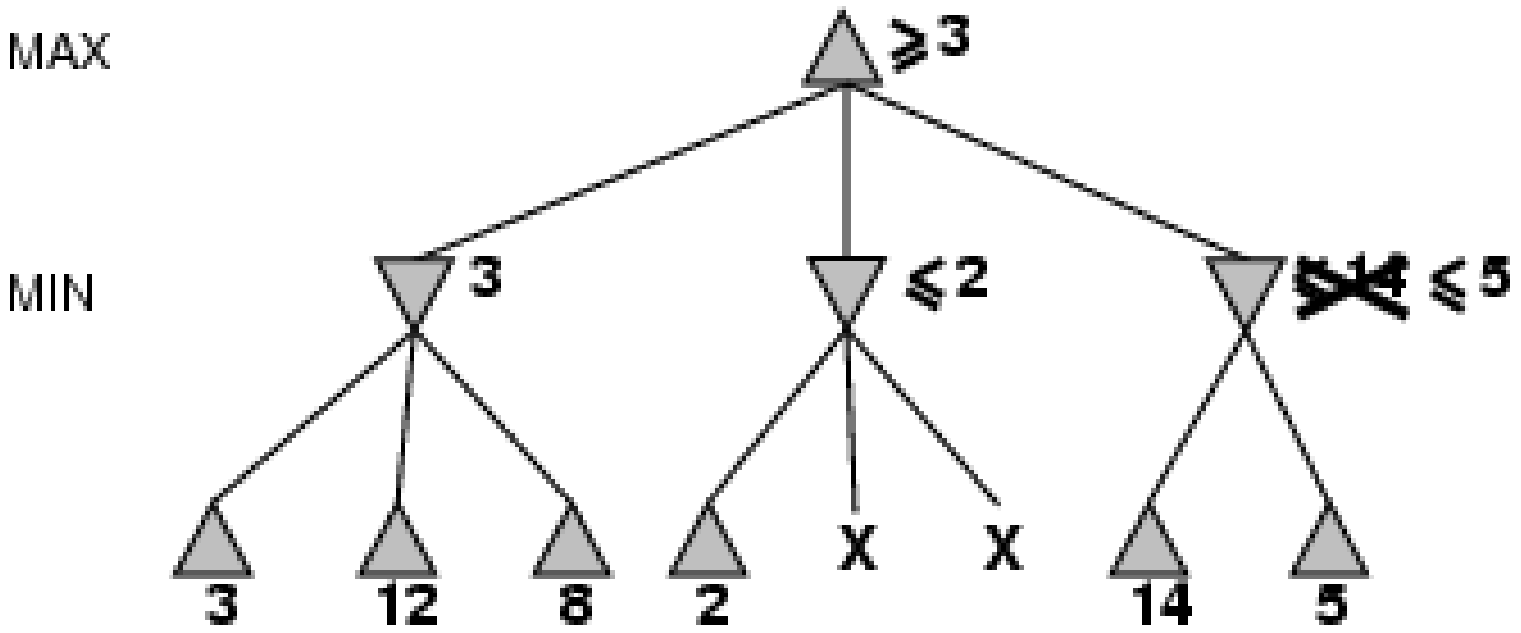
α-β pruning example



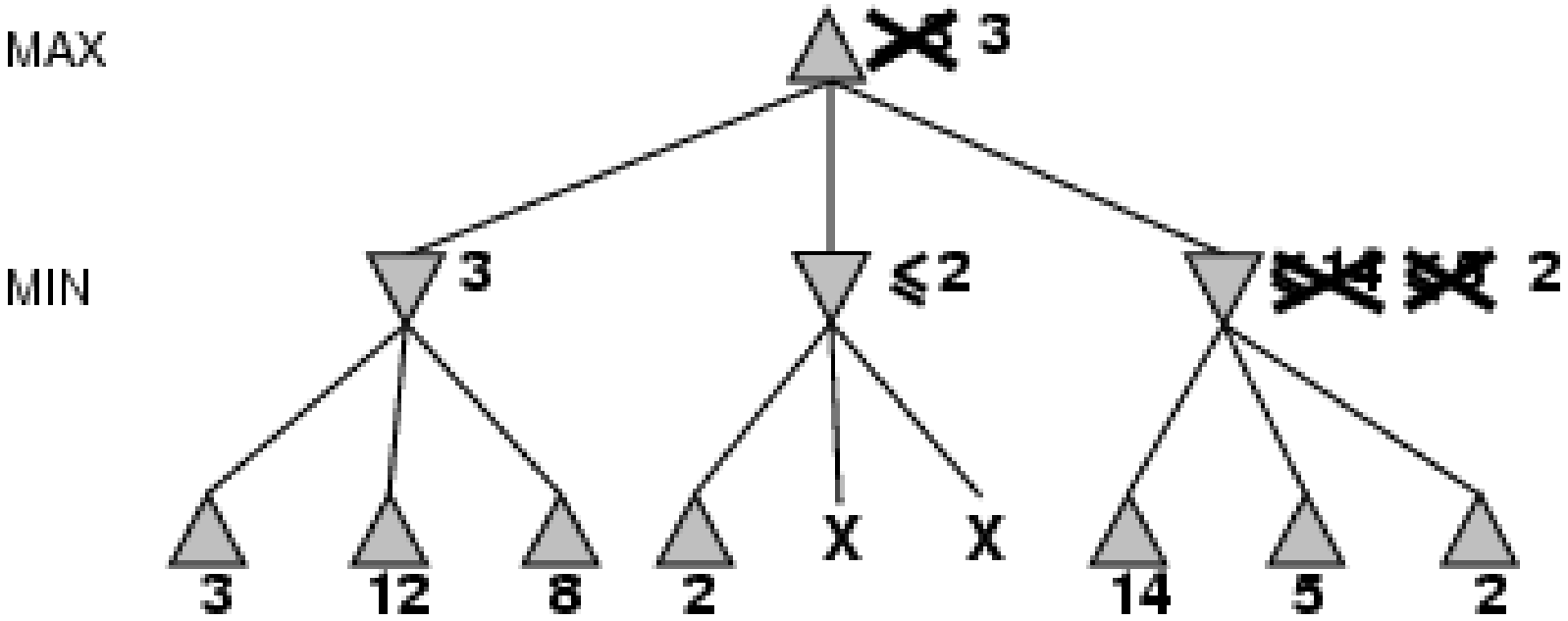
α-β pruning example



α-β pruning example



α-β pruning example

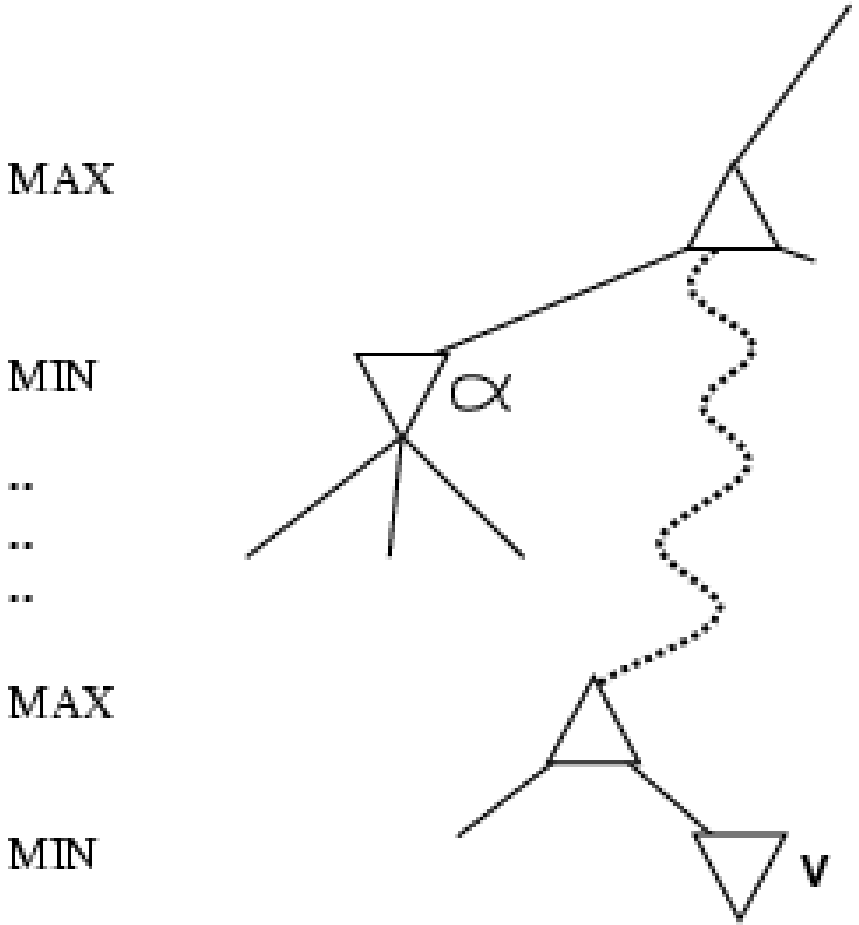


Properties of α-β

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{m/2})$
  → doubles depth of search
- A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Why is it called α-β?

- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for *max*
- If *v* is worse than α, *max* will avoid it
  → prune that branch
- Define β similarly for *min*



The α-β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game

    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v

---

function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

The α-β algorithm

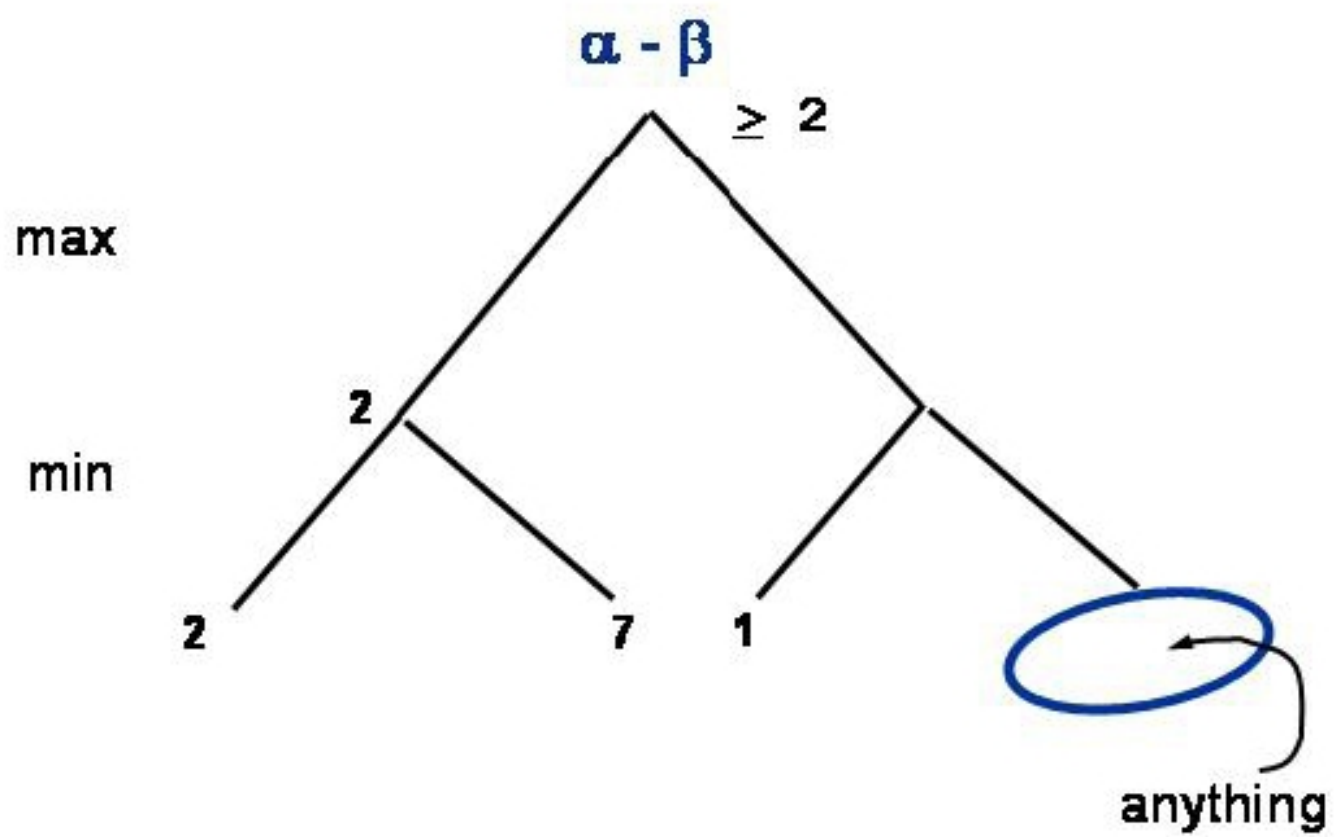
```
function MIN-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

The α-β algorithm

α - β

```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

α-β pruning



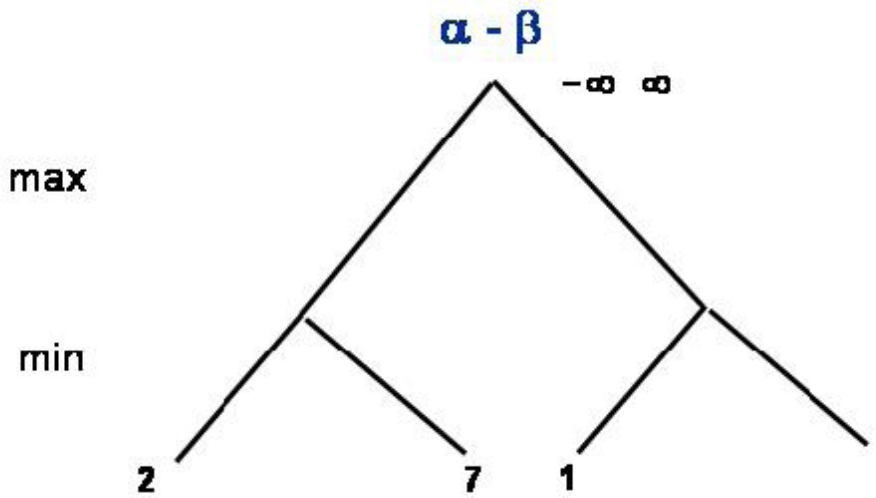
α is lower bound on score

β is upper bound on score

α-β pruning example

α - β

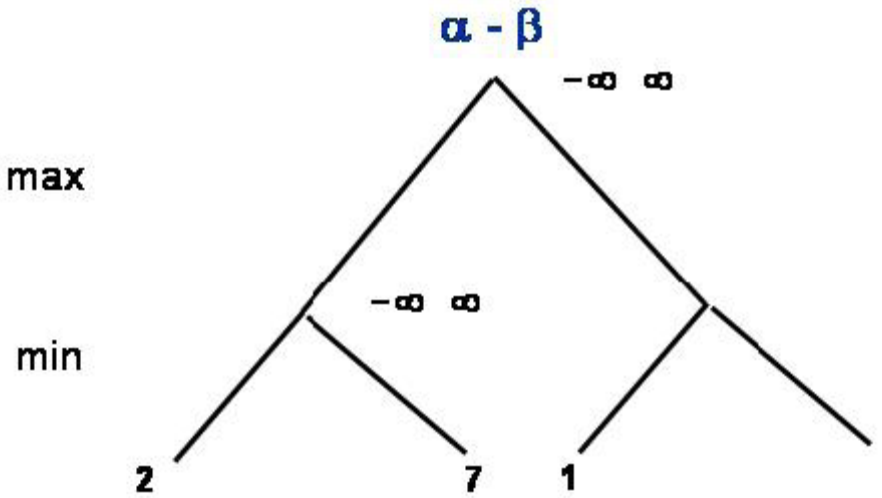
```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```



α-β pruning example

α - β

```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

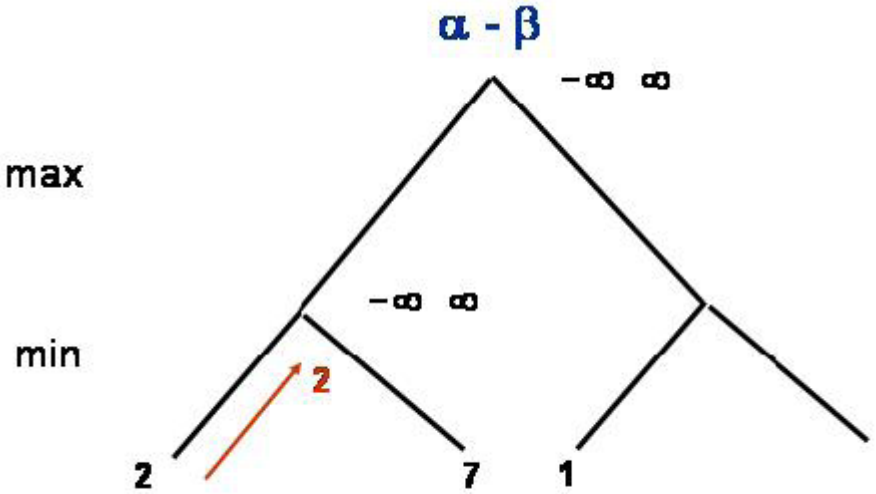


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

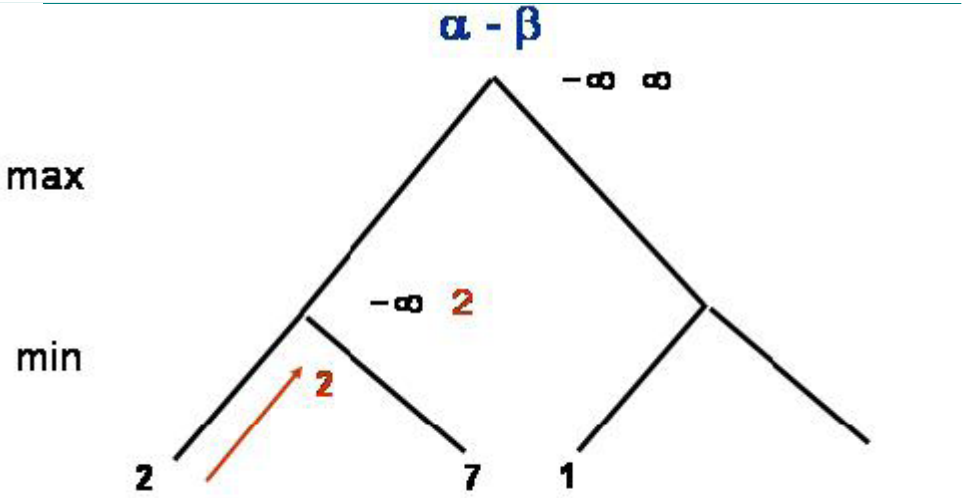
```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```



α-β pruning example

α - β

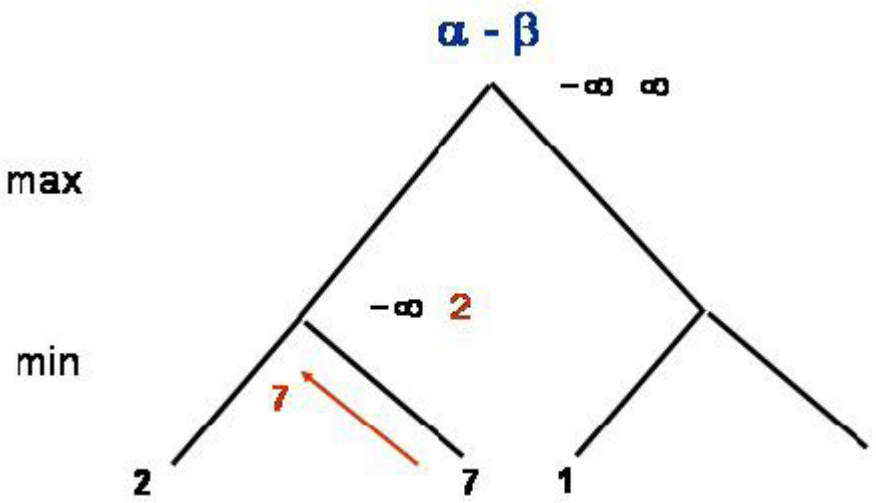
```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```



α-β pruning example

α - β

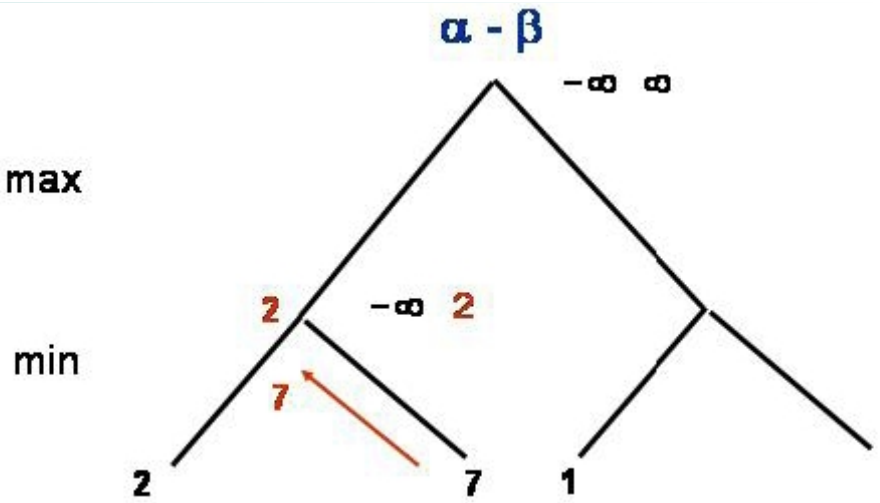
```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```



α-β pruning example

α - β

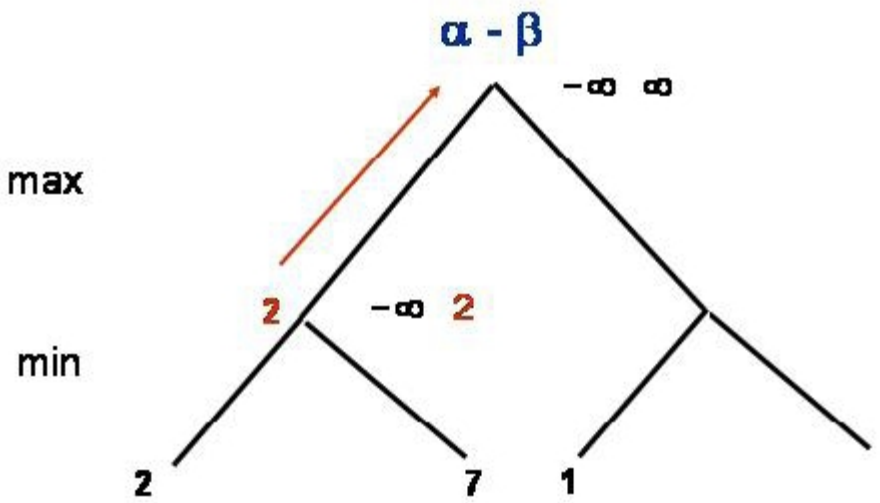
```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```



α-β pruning example

α - β

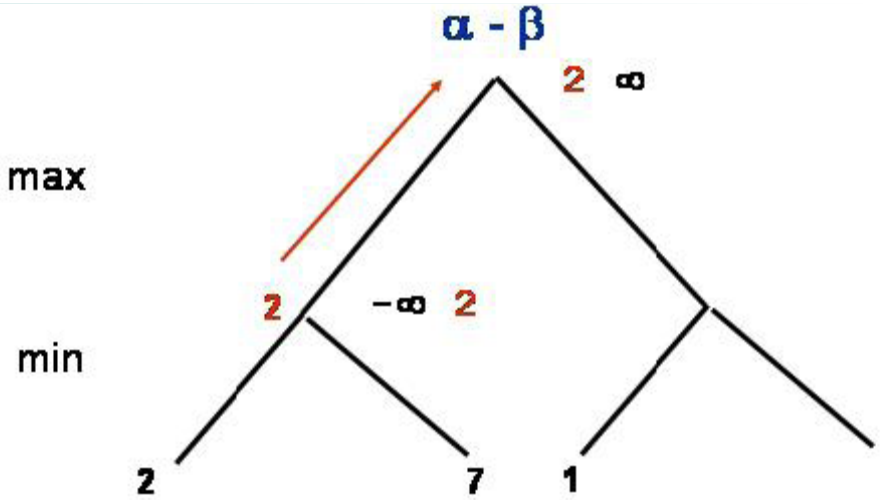
```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```



α-β pruning example

α - β

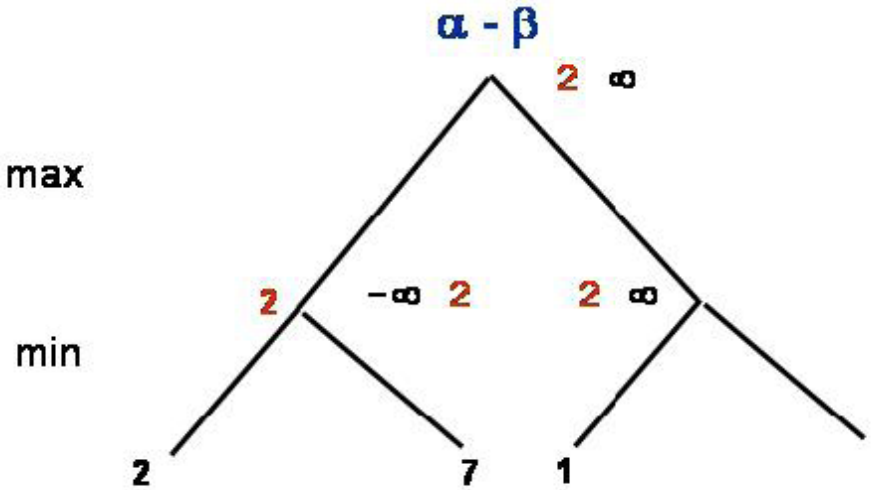
```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```



α-β pruning example

α - β

```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

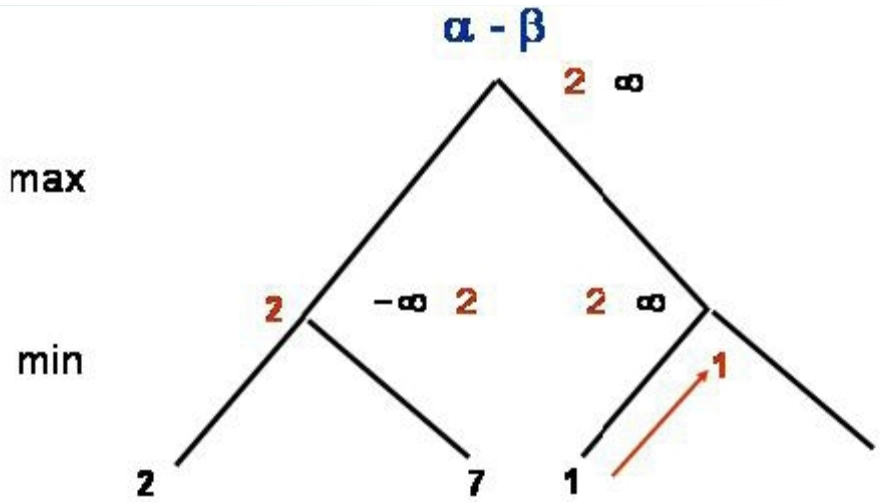


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

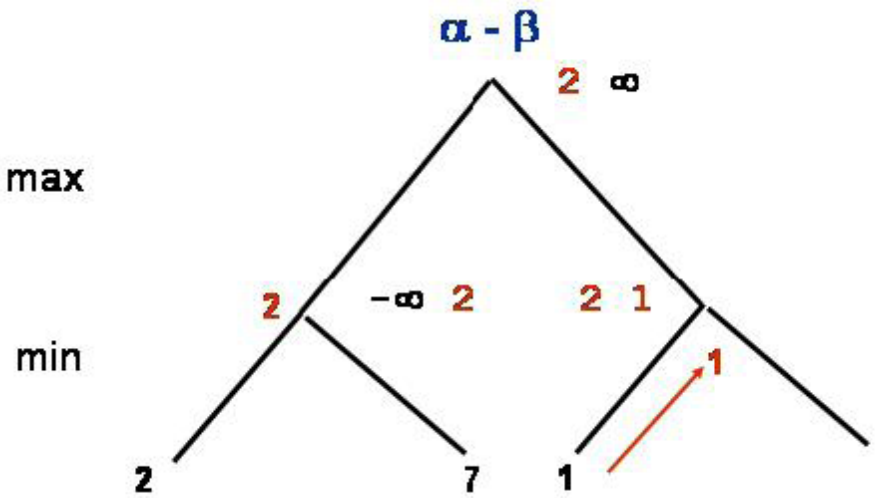


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

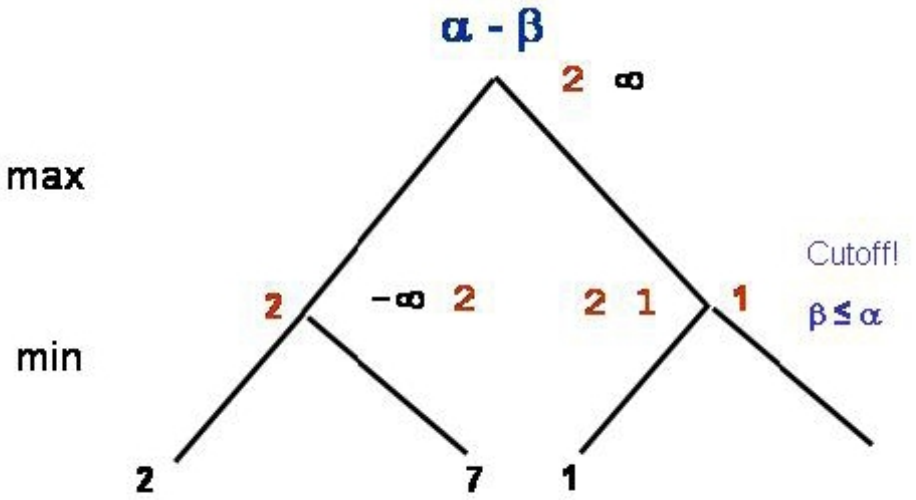


α-β pruning example

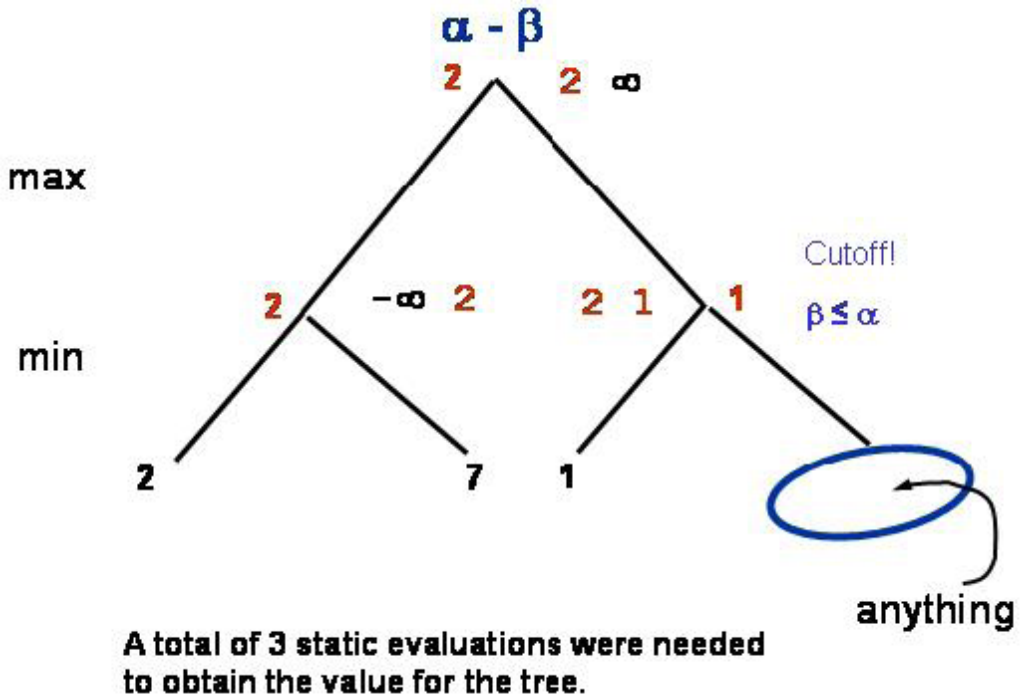
α - β

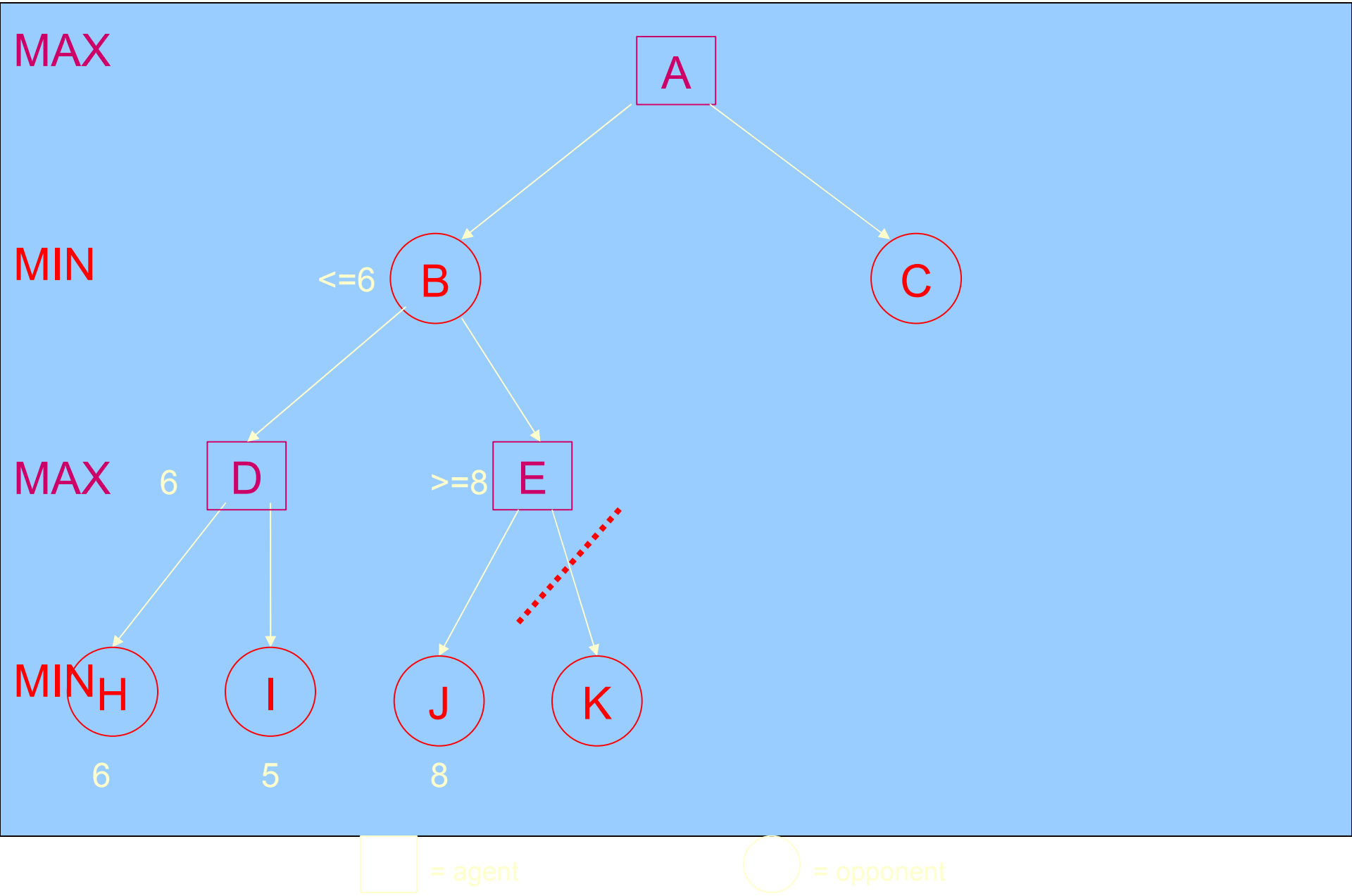
// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

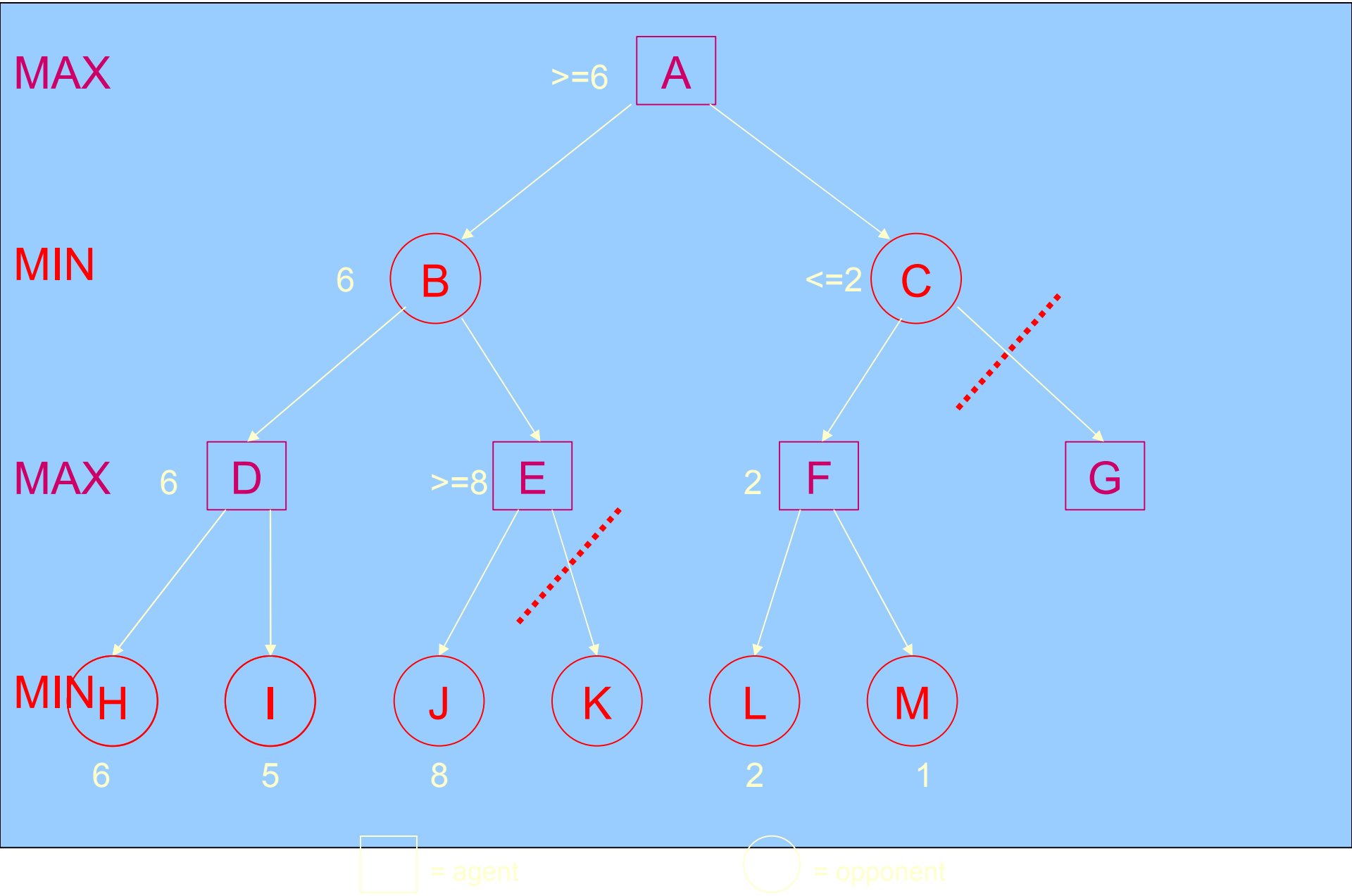
```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

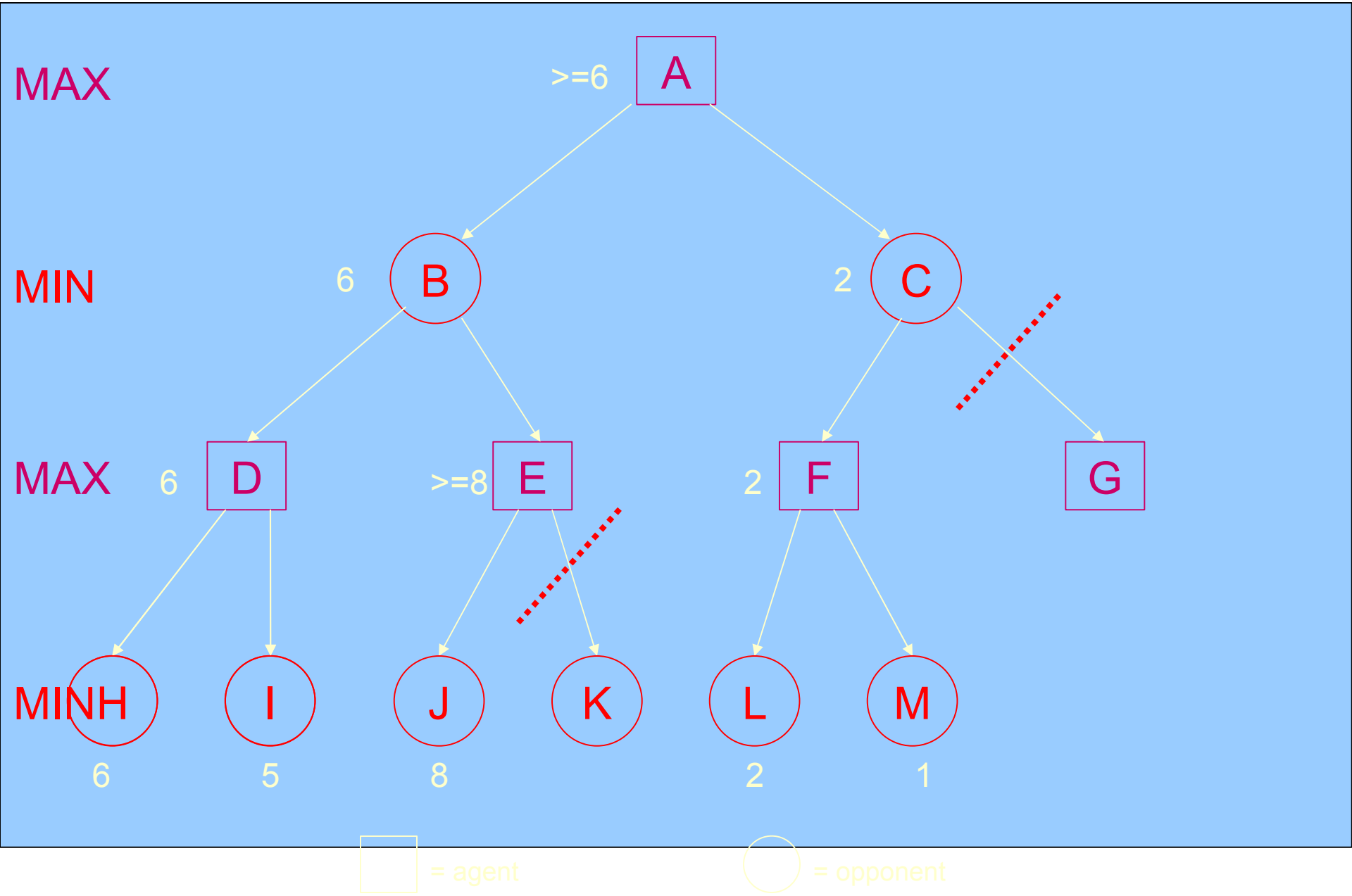


α-β pruning example

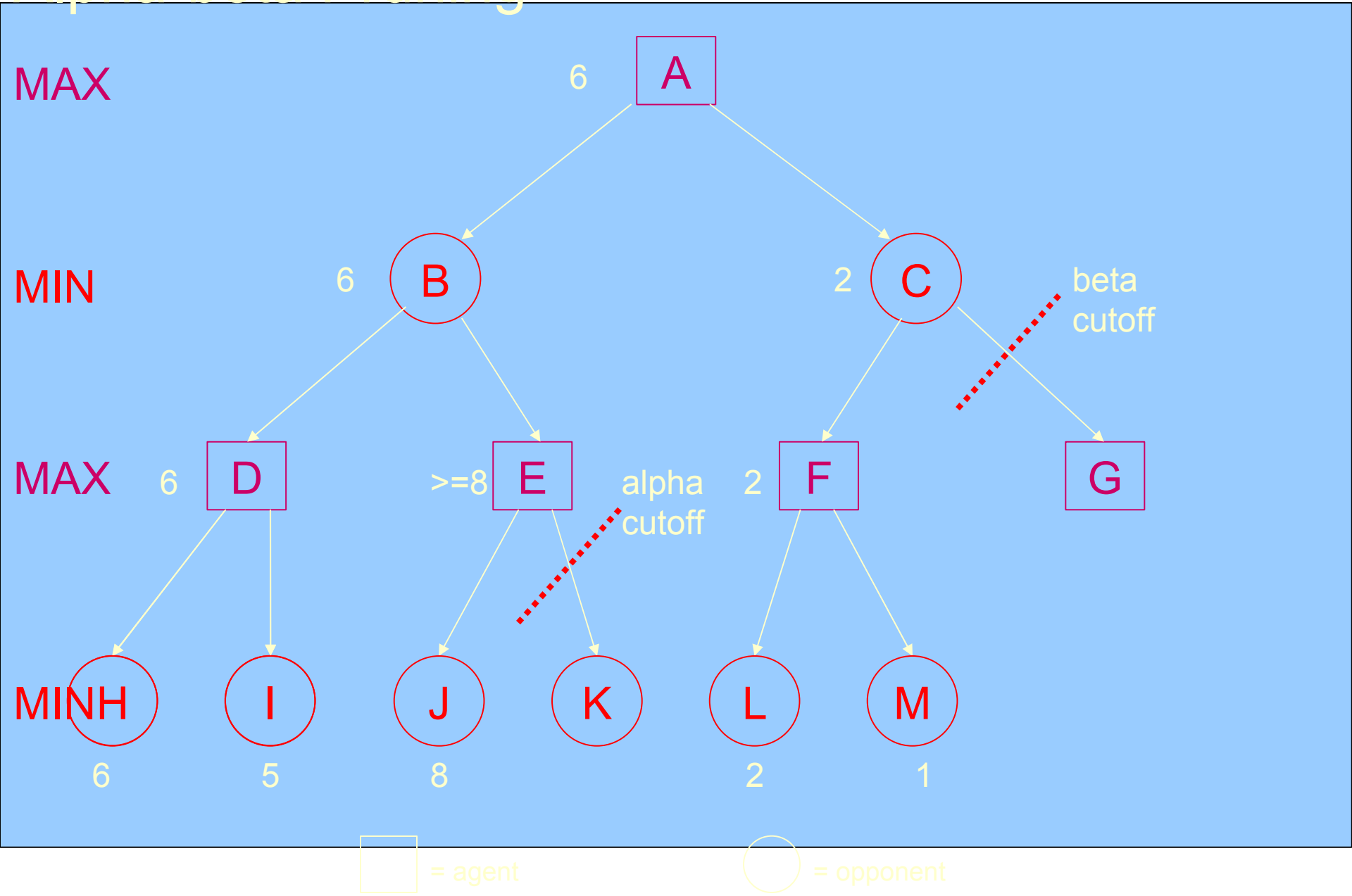



MAX
A
MIN
<=6
B
C
MAX
6
D
>=8
E
MIN
H
I
J
K
6
5
8
= agent
= opponent

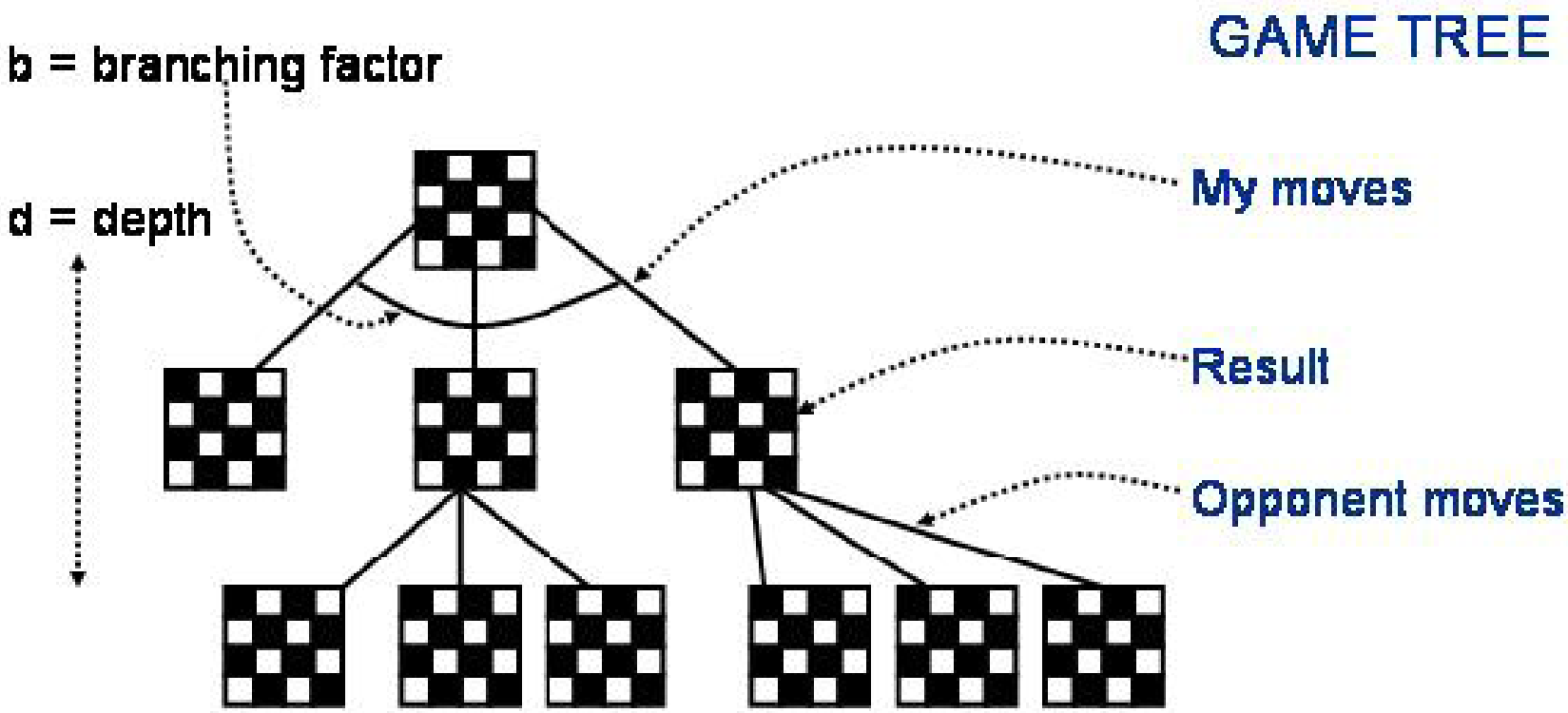

MAX
MIN
MAX
MIN
>=6
A
6
B
<=2
C
6
D
>=8
E
2
F
G
H
I
J
K
L
M
6
5
8
2
1
= agent
= opponent


MAX
MIN
MAX
MINH
>=6
A
6
B
2
C
6
D
>=8
E
2
F
G
I
J
K
L
M
6
5
8
2
1
= agent
= opponent

Alpha-beta Pruning



Move generation



Chess
b = 36
d > 40

36^{40} is big!

Resource limits

Suppose we have 100 secs, explore 10^4 nodes/sec

→ 10^6 nodes per move

Standard approach:

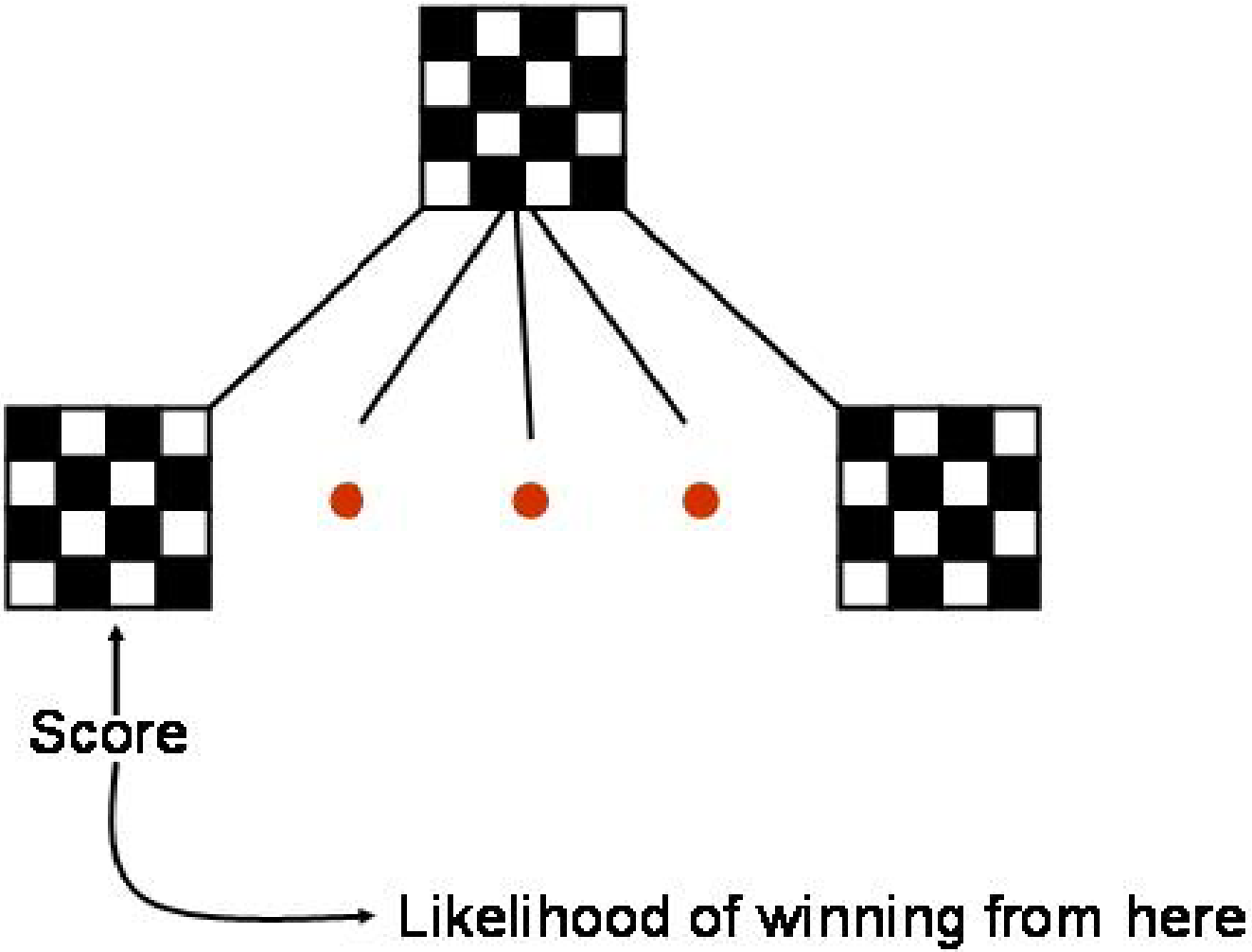
- cutoff test:

  e.g., depth limit (perhaps add quiescence search)

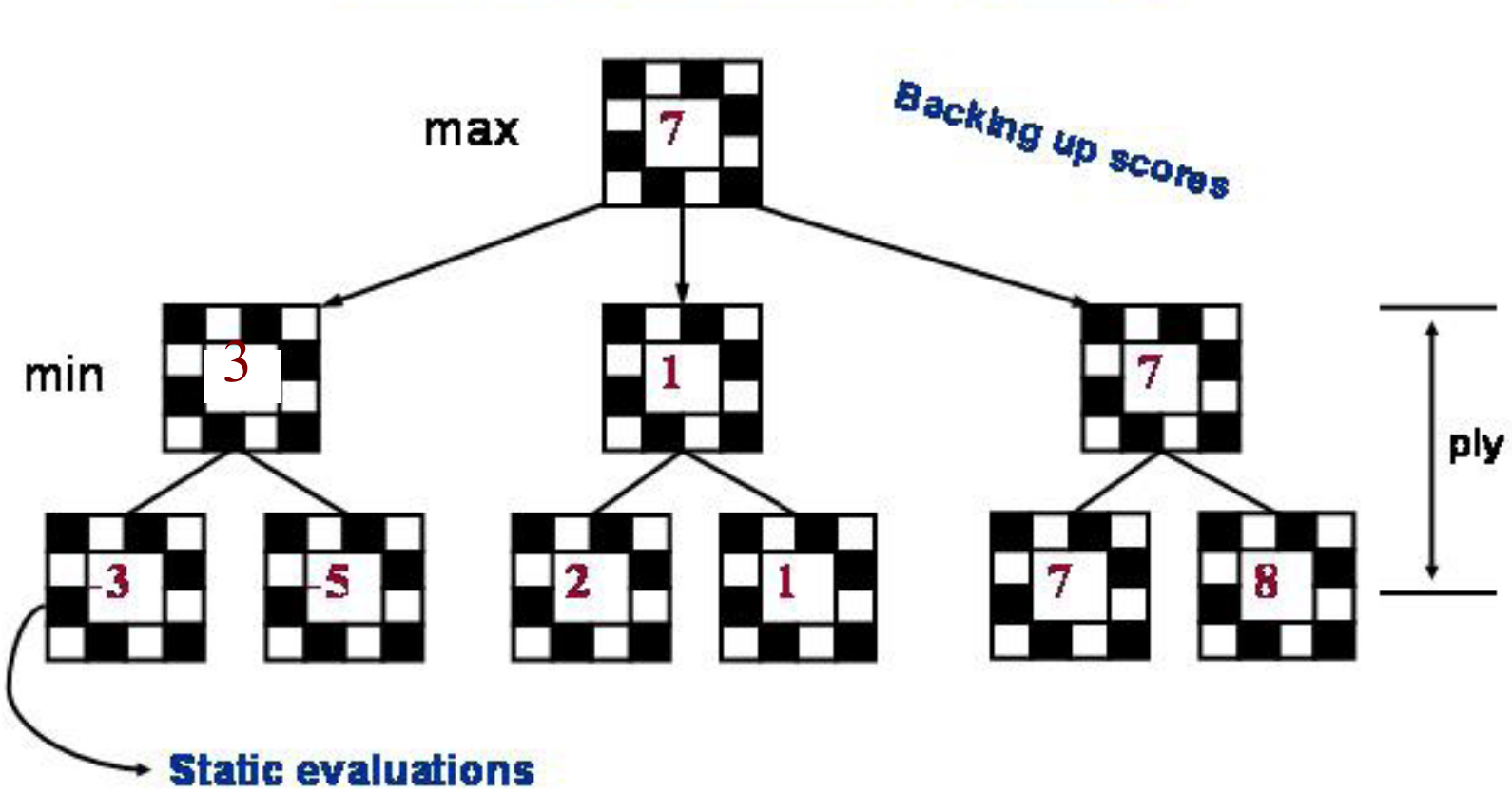
- evaluation function

  = estimated desirability of position

Evaluation function



Min-Max


max
7
Backing up scores
min
3
1
7
ply
3
5
2
1
7
8
Static evaluations

Evaluation functions

- A typical evaluation function is a linear function in which some set of coefficients is used to weight a number of "features" of the board position.

- For chess, typically linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- e.g., $w_1 = 9$ with

  $f_1(s)$ = (number of white queens) – (number of black queens), etc.

Evaluation function

$$S = c_1 \times \text{material} + c_2 \times \text{pawn structure} + c_3 \times \text{mobility} + c_4 \times \text{king safety} + c_5 \times \text{center control} + \ldots$$

P	1
K	3
B	3.5
R	5
Q	9

- "material", : some measure of which pieces one has on the board.
- A typical weighting for each type of chess piece is shown
- Other types of features try to encode something about the distribution of the pieces on the board.

Cutting off search

MinimaxCutoff is identical to *MinimaxValue* except

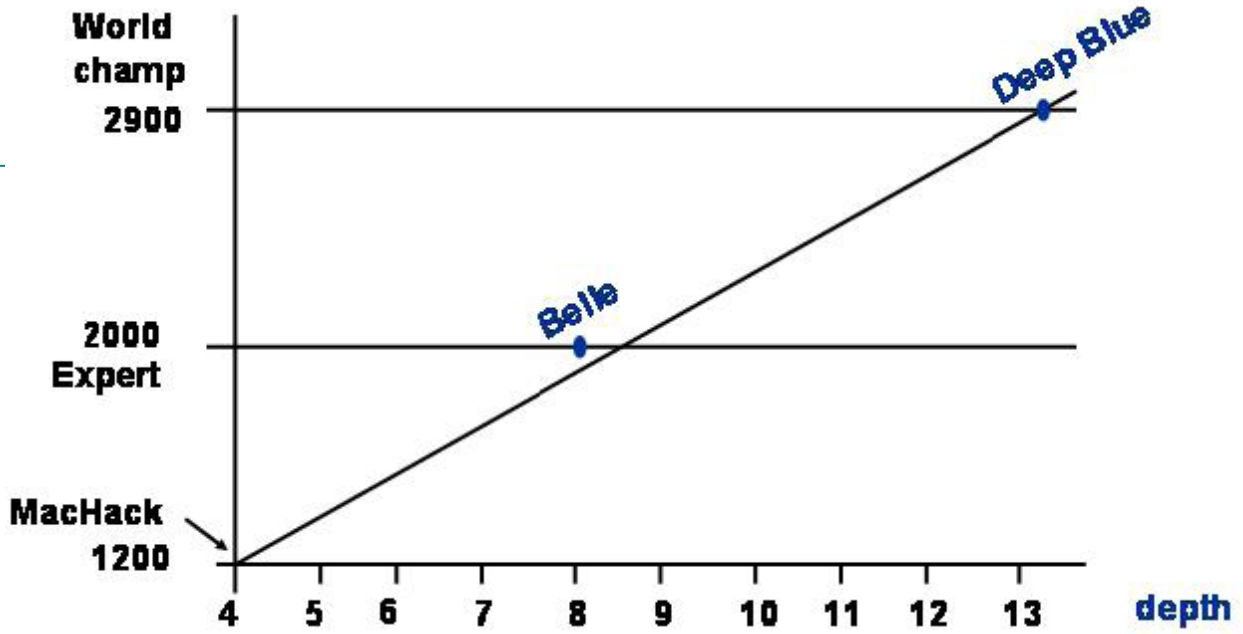
1. *Terminal?* is replaced by *Cutoff?*
2. *Utility* is replaced by *Eval*

Does it work in practice?

$b^m = 10^6$, b=35 → m=4

4-ply lookahead is a hopeless chess player!

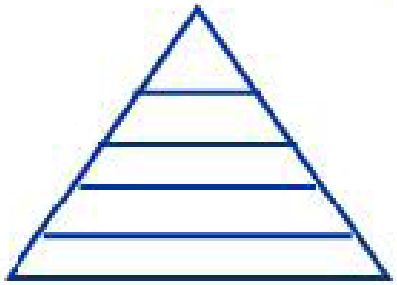
- 4-ply ≈ human novice
- 8-ply ≈ typical PC, human master
- 12-ply ≈ Deep Blue, Kasparov



- The key idea is that the more lookahead we can do, that is, the deeper in the tree we can look, the better our evaluation of a position will be, even with a simple evaluation function. In some sense, if we could look all the way to the end of the game, all we would need is an evaluation function that was 1 when we won and -1 when the opponent won.

- it seems to suggest that brute-force search is all that matters.
- And Deep Blue is brute indeed... It had 256 specialized chess processors coupled into a 32 node supercomputer. It examined around 30 billion moves per minute. The typical search depth was 13ply, but in some dynamic situations it could go as deep as 30.

Practical issues



Variable branching

Iterative deepening

- order best move from last search first
- use previous backed up value to initialize $[\alpha, \beta]$
- keep track of repeated positions (transposition tables)

Horizon effect

- quiescence
  - Pushing the inevitable over search horizon

Parallelization

Other games

- Backgammon
  - Involves randomness – dice rolls
  - Machine-learning based player was able to draw the world champion human player.
- Bridge
  - Involves hidden information – other players' cards – and communication during bidding.
  - Computer players play well but do not bid well
- Go
  - No new elements but huge branching factor
  - No good computer players exist

Deterministic games in practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

- Othello: human champions refuse to compete against computers, who are too good.

- Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

Summary

- Games are fun to work on!
- They illustrate several important points about AI
- perfection is unattainable → must approximate
- good idea to think about what to think about