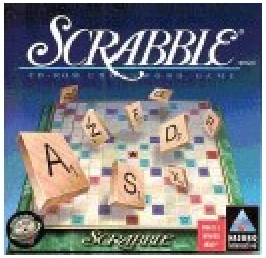
# CSE 401

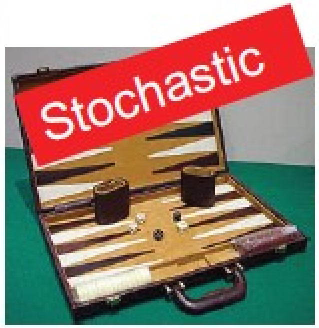
*Game Playing*
*Alpha-Beta Search*
*and*
*General Issues*

# *Which of these are:* 2-player zero-sum discrete finite deterministic games of perfect information

- Two player: Duh!
- Zero-sum: In any outcome of any game, Player A's gains equal player B's losses.
- Discrete: All game states and decisions are discrete values.
- Finite: Only a finite number of states and decisions.
- Deterministic: No chance (no die rolls).
- Games: See next page
- Perfect information: Both players can see the state, and each decision is made sequentially (no simultaneous moves).




# *Which of these are:* 2-player zero-sum discrete finite deterministic games of perfect information


- Two player: Duh!
- Zero-sum: In any outcome of any game, Player A's gains equal player B's losses.
- Discrete: All game states and decisions are discrete values.
- Finite: Only a finite number of states and decisions.
- Deterministic: No chance (no die rolls).
- Games: See next page
- Perfect information: Both players can see the state, and each decision is made sequentially (no simultaneous moves).




# Definition

A Two-player zero-sum discrete finite deterministic game of perfect information is a quintuplet: *( S , I , Succs , T , V )* where

| | | |
|---|---|---|
| S | = | a finite set of states (note: state includes information sufficient to deduce who is due to move) |
| I | = | the initial state |
| *Succs* | = | a function which takes a state as input and returns a set of possible next states available to whoever is due to move |
| T | = | a subset of S. It is the terminal states: the set of states at which the game is over |
| V | = | a mapping from terminal states to real numbers. It is the amount that A wins from B. (If it's negative A loses money to B). |

Convention: assume Player A moves first.

For convenience: assume turns alternate.

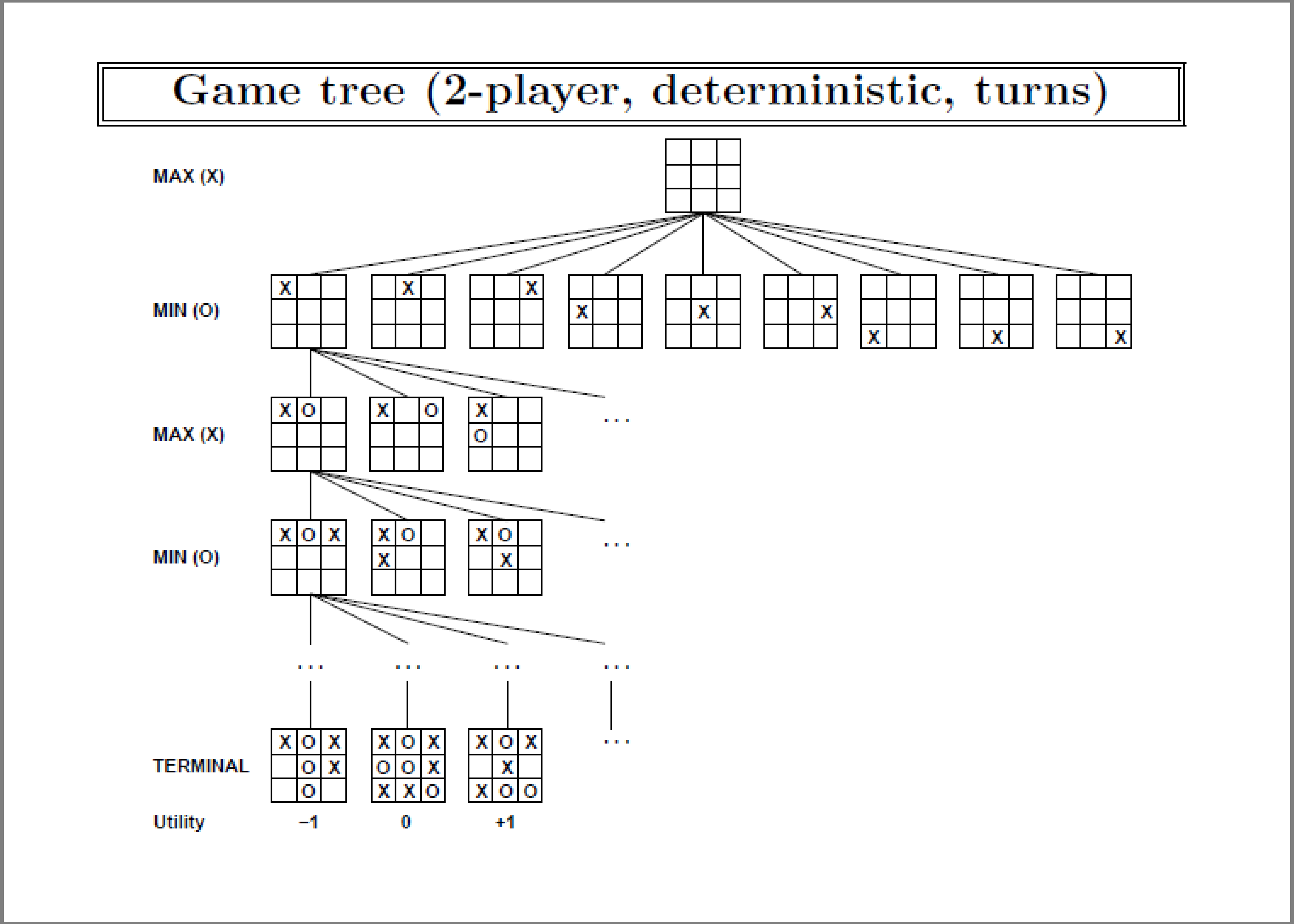
## *Utility Function*

- Gives the utility of a game state
  - utility(State)
- Examples
  - -1, 0, and +1, for Player 1 loses, draw, Player 1 wins, respectively
  - Difference between the point totals for the two players
  - Weighted sum of factors (e.g. Chess)
    - $utility(S) = w_1 f_1(S) + w_2 f_2(S) + \ldots + w_n f_n(S)$
      - $f_1(S)$ = (Number of white queens) – (Number of black queens), $w_1 = 9$
      - $f_2(S)$ = (Number of white rooks) – (Number of black rooks), $w_2 = 5$

## *Game Playing*

- **Game tree**
  - **describes the possible sequences of play**
  - **is a graph if we merge together identical states**
- **Minimax:**
  - **utility values assigned to the leaves**
- **Values “backed up” the tree by**
  - **MAX node takes max value of children**
  - **MIN node takes min value of children**
  - **Can read off best lines of play**
- **Depth Bounded Minimax**
  - **utility of terminal states estimated using an “evaluation function”**

# Game Tree for Tic-tac-toe



# *Minimax*

$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if n is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node.} \end{cases}$$

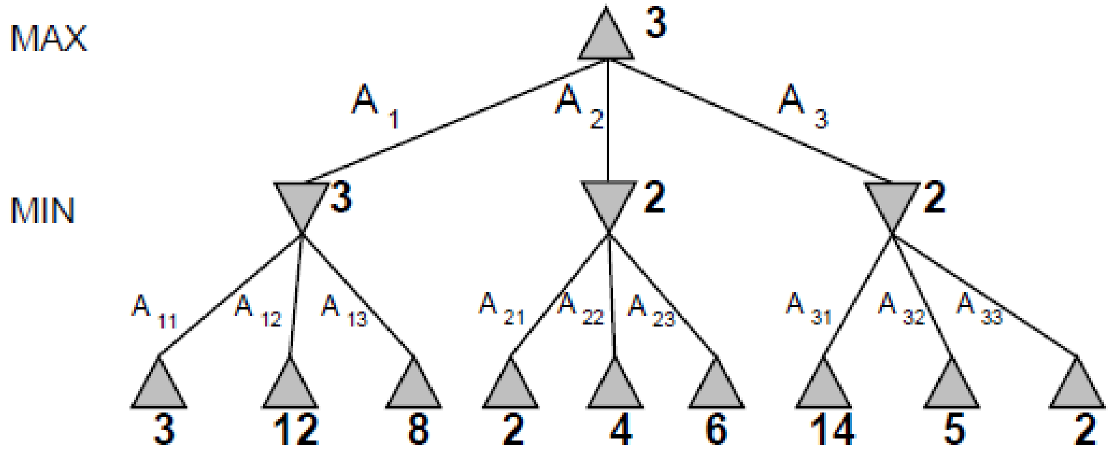
# MiniMax

## Minimax

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:

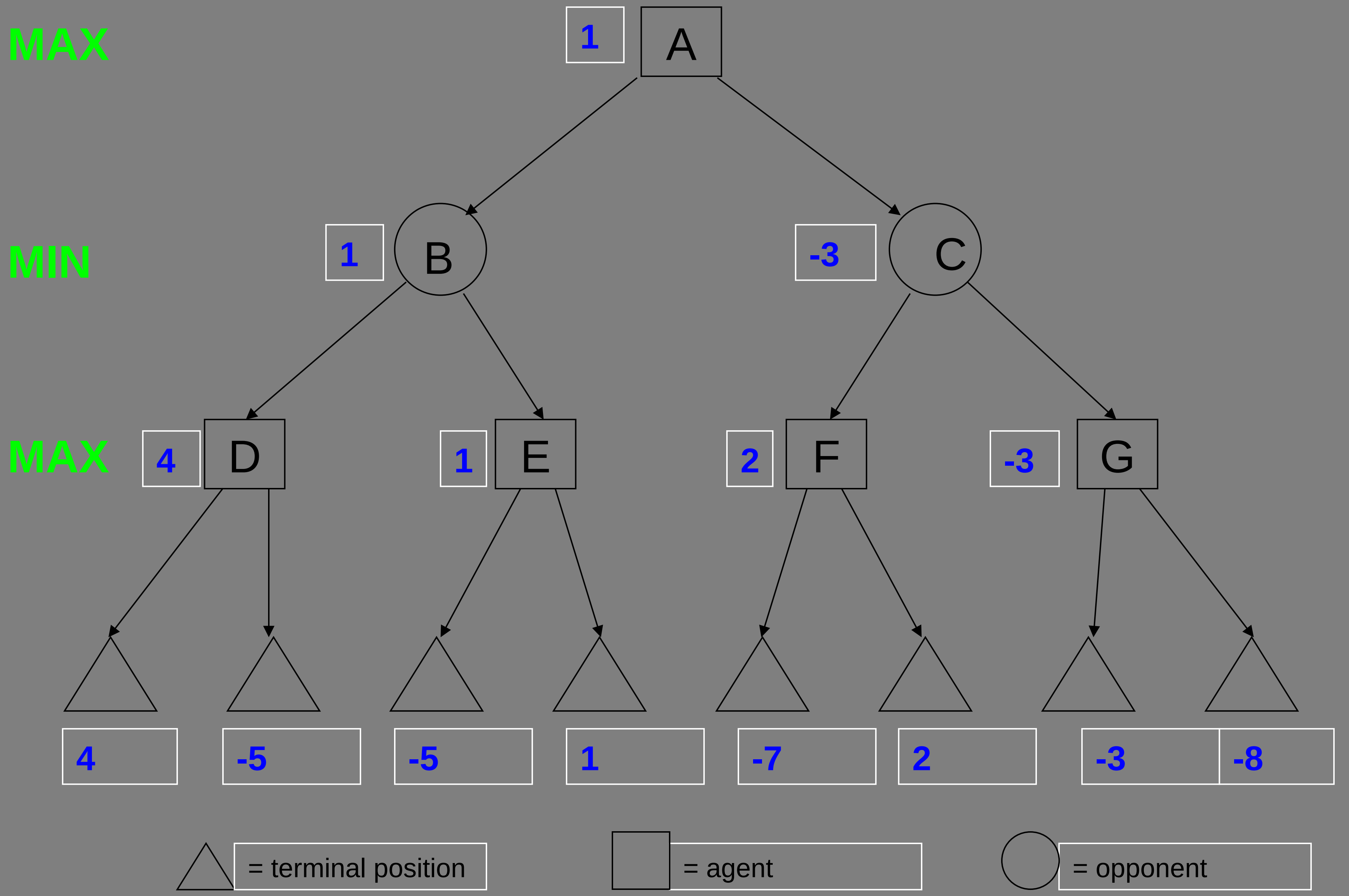


# Basic Algorithm

```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game
   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

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
```


MAX
MIN
MAX
1
A
1
B
-3
C
4
D
1
E
2
F
-3
G
4
-5
-5
1
-7
2
-3
-8
= terminal position
= agent
= opponent

# *Game Playing – Beyond Minimax*

- **Efficiency of the search**
  - **Game trees are very big**
  - **Evaluation of positions is time-consuming**
- **How can we reduce the number of nodes to be evaluated?**
  - **“alpha-beta search”**
- **Bounding the depth of minimax has deficiencies**
  - **Why?**
  - **How can we mitigate these deficiencies?**

## *Game Playing – Improving Efficiency*

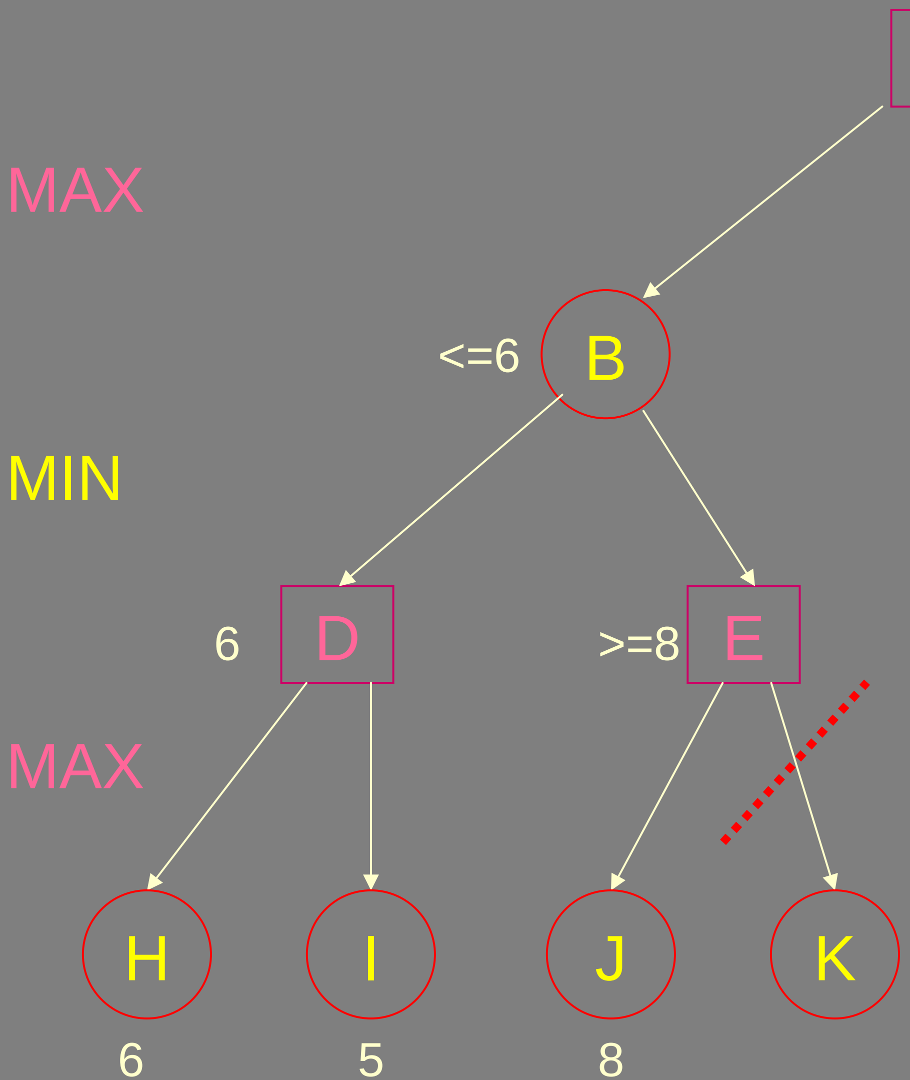
- **Suppose that we are doing depth-bounded minimax**

- **We have a game tree to create and then insert the minimax values in order to find the values for our possible moves from the current position**

# *Game Playing – Minimax using DFS*

- **The presentation of minimax was done by “backing up from the leaves” – a “bottom-up” breadth-first search.**
- **This has the disadvantage of taking a lot of space**
  - **Compare this with the space usage issues for DFS vs. BFS in earlier lectures**
- **If we can do minimax using DFS then it is likely to take a lot less space**
- **Minimax can be implemented using DFS**
- **But reduced space is not the only advantage:**

**STOP! What else can you deduce now!?**




On discovering util( D ) = 6

we know that util( B ) <= 6

On discovering util( J ) = 8

we know that util( E ) >= 8

Can stop expansion of E as best play will not go via E

Value of K is irrelevant – prune it!

= agent

= opponent

## *Game Playing – Pruning nodes*

- **If we are scanning the tree using DFS then there was no point in evaluating node K**
- **Whatever the value of K there cannot be any rational sequence of play that would go through it**
  - **Node K can be pruned from the search: i.e. just not selected for further expansion**
- **“At node B then MIN will never select E; because J is better than D for MAX and so MIN must not allow MAX to have that opportunity”**

- **Q. So what! It’s just one node?**
- **A. Suppose that the depth limit were such that K was far from the depth bound. Then evaluating K corresponds to a large sub-tree. Such prunings can save an exponential amount of work**

## *Game Playing – Improving Efficiency*

- **Suppose that we were doing Breadth-First Search, would you still be able to prune nodes in this fashion?**

- **NO! Because the pruning relied on the fact that we had already evaluated node D by evaluating the tree underneath D**

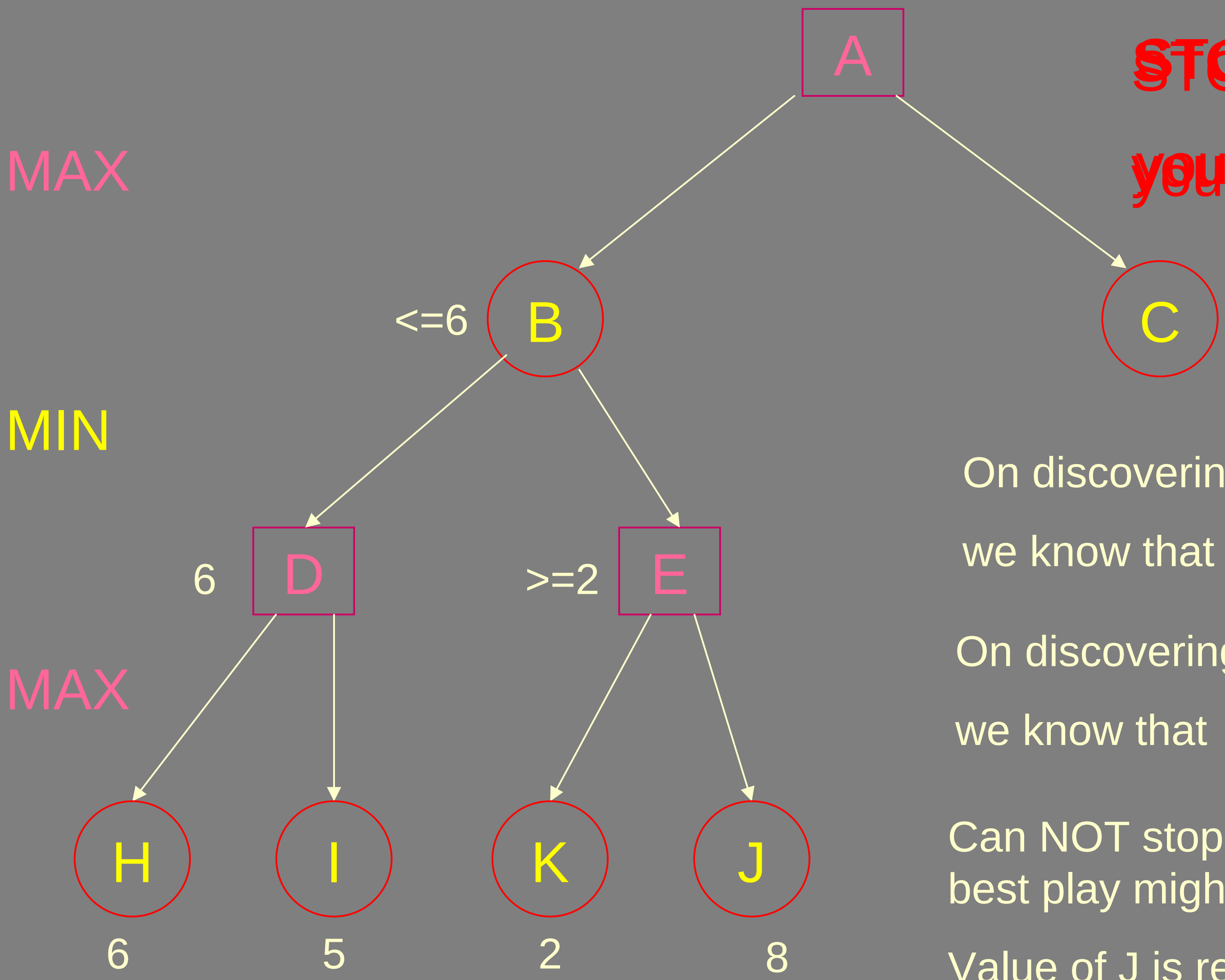
- **This form of pruning is an example of "alpha-beta pruning" and relies on doing a DEPTH-FIRST search of the depth bounded tree**

## *Game Playing – Node-ordering*

- **Suppose that**
  - **nodes K and J were evaluated in the opposite order**
  - **can we expect that we would be able to do a similar pruning?**

- **The answer depends on the value of K**
- **Suppose that K had a value of 2 and is expanded first:**

**STOP! What else can you deduce now!?**




On discovering util( D ) = 6

we know that util( B ) <= 6

On discovering util( K ) = 2

we know that util( E ) >= 2

Can NOT stop expansion of E as best play might still go via E

Value of J is relevant – no pruning

 = agent

 = opponent

# *Game Playing – Node-ordering*

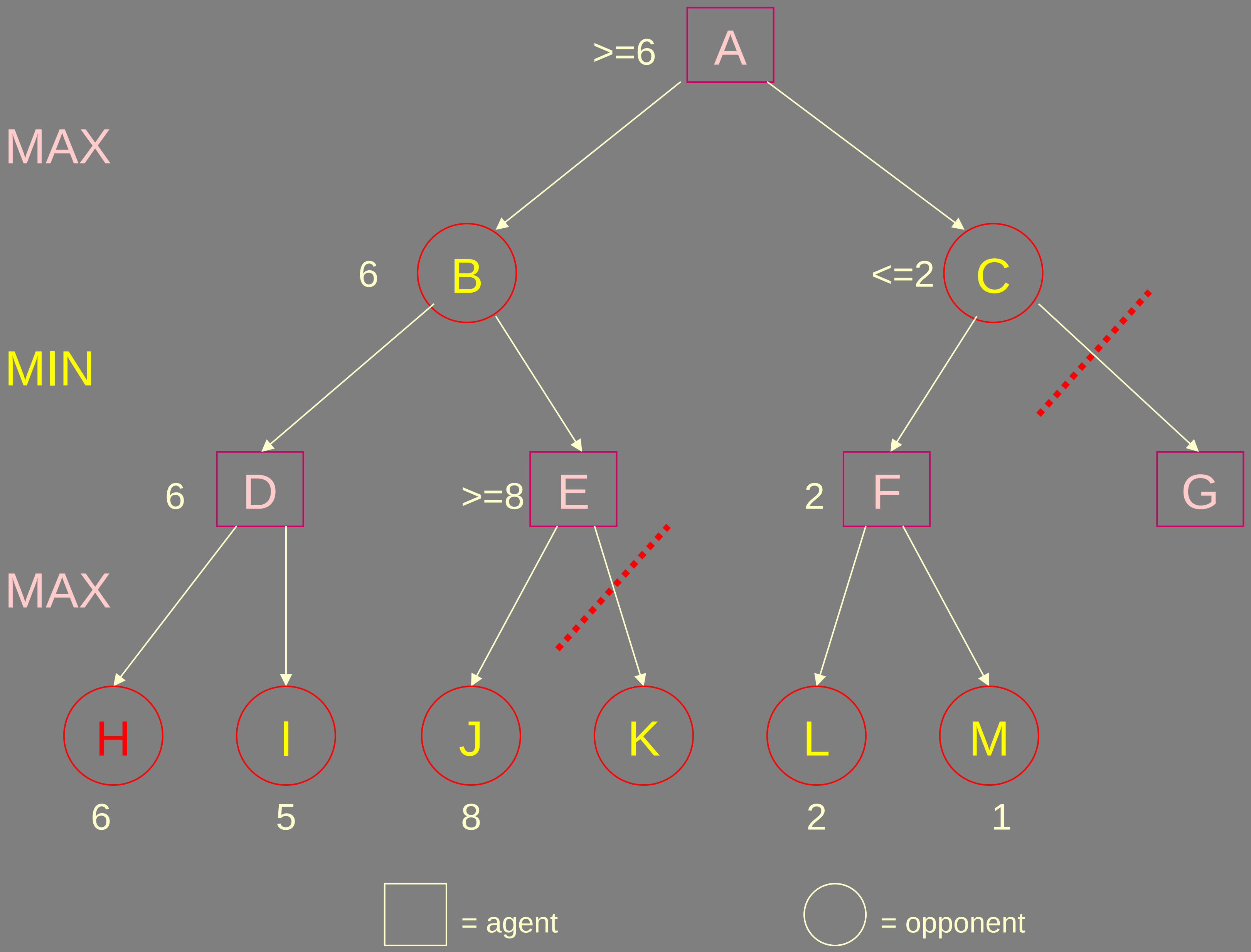
- **When K had a value of 2 and was expanded first then we did not get to prune a child of E**
- **To maximise pruning we want to first expand those children that are best for the parent**
  - **cannot know which ones are really best**
  - **use heuristics for the “best-first” ordering**
- **If this is done well then alpha-beta search can effectively double the depth of search tree that is searchable in a given time**
  - **Effectively reduces the branching factor in chess from about 30 to about 8**
  - **This is an enormous improvement!**

## *Game Playing – Improving Efficiency*

- **The games are symmetric so is natural that we can also do a similar pruning with the MIN and MAX roles reversed**

- **The reasoning is identical other than for the reversal of roles**

- **Can deduce that some other nodes can not be involved in the line of best play**


MAX
MIN
MAX
>=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

# *Game Playing – Alpha-Beta Implementation*

- **The pruning was based on using the results of the “DFS so far” to deduce upper and lower bounds on the values of nodes**
- **Conventionally these bounds are stored in terms of two parameters**
  - **alpha α**
  - **beta β**

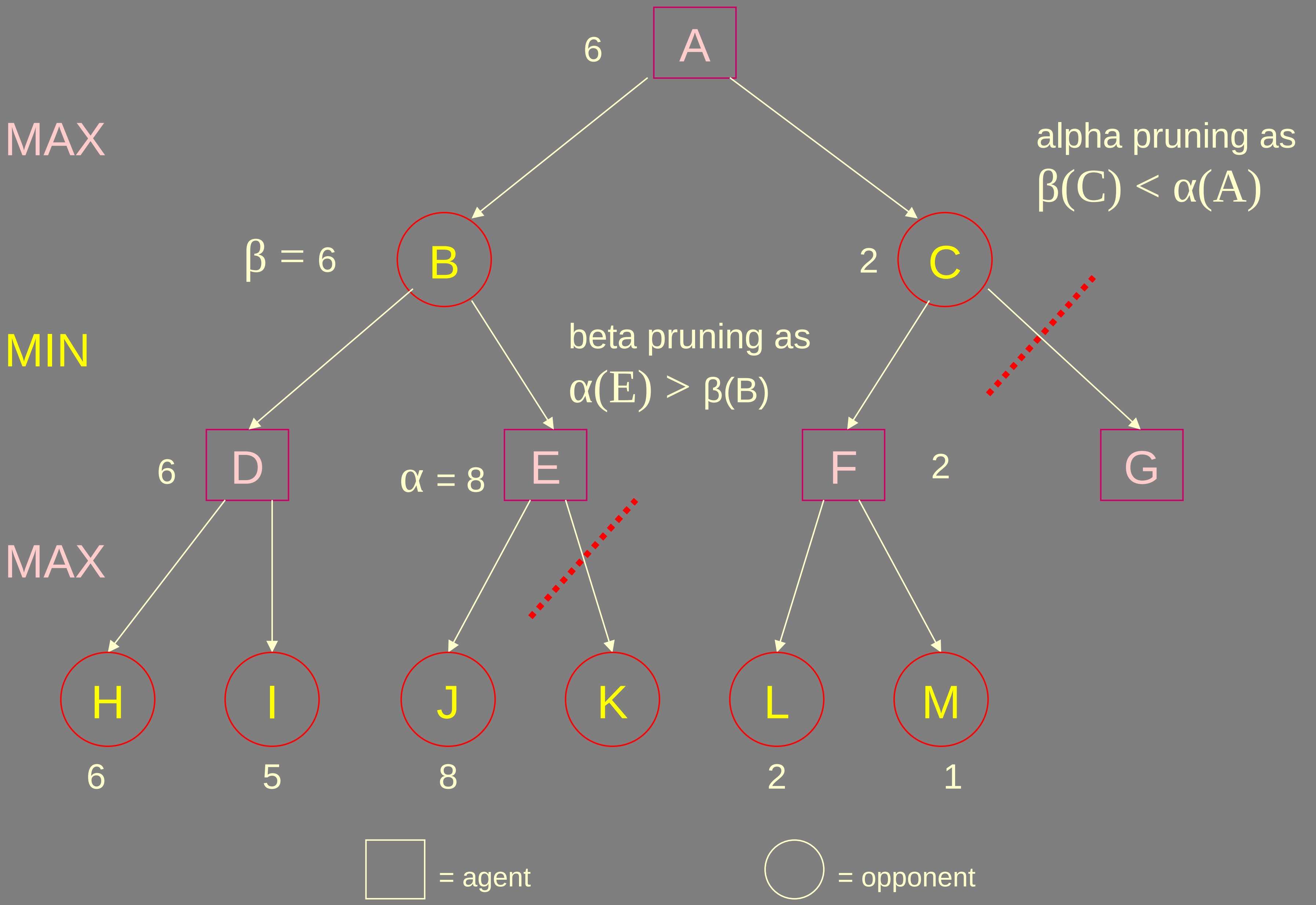
## *Game Playing – Alpha-Beta Implementation*

- **α values are stored with each MAX node**
- **each MAX node is given a value of alpha that is the current best lower-bound on its final value**
  - **initially is - ∞ to represent that nothing is known**
  - **as we do the search then α at a node can increase, but it can never decrease – it always gets better for MAX**

## *Game Playing – Alpha-Beta Implementation*

- **β values are stored with each MIN node**
- **each MIN node is given a value of beta that is the current best upper-bound on its final value**
  - **initially is** $+\infty$ **to represent that nothing is known**
  - **as we do the search then β at a node can decrease, but it can never increase – it always gets better for MIN**

# Alpha-beta Pruning



# Properties of $\alpha$–$\beta$

Pruning **does not** affect final result

Good move ordering improves effectiveness of pruning

With "perfect ordering," time complexity $= O(b^{m/2})$
$\Rightarrow$ **doubles** solvable depth

A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Unfortunately, $35^{50}$ is still impossible!

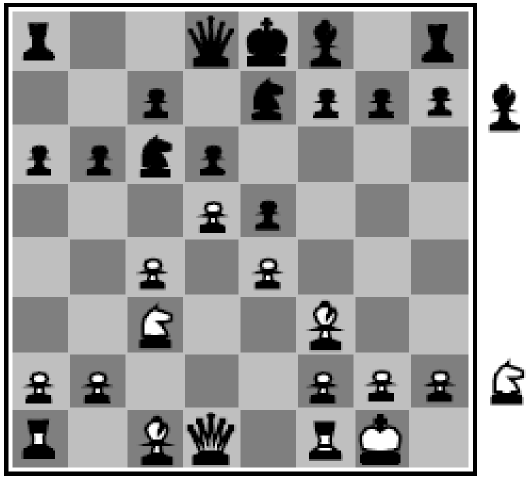
# Resource limits

Standard approach:

- Use CUTOFF-TEST instead of TERMINAL-TEST
  e.g., depth limit (perhaps add quiescence search)
- Use EVAL instead of UTILITY
  i.e., evaluation function that estimates desirability of position

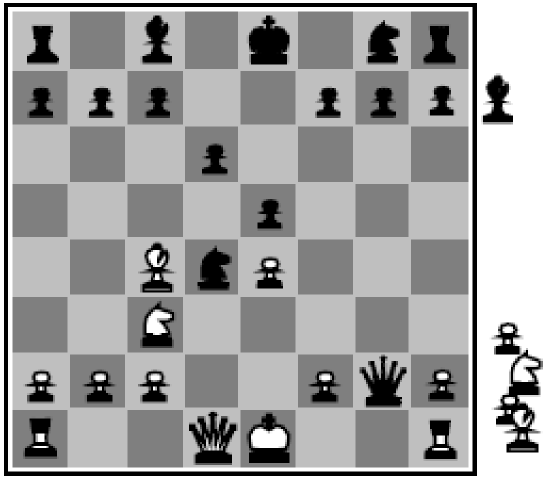
Suppose we have $100$ seconds, explore $10^4$ nodes/second
$\Rightarrow$ $10^6$ nodes per move $\approx 35^{8/2}$
$\Rightarrow$ $\alpha$–$\beta$ reaches depth 8 $\Rightarrow$ pretty good chess program

# Evaluation functions



Black to move

White slightly better



White to move

Black winning

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.

## *Game Playing – Deficiencies of Minimax*

- **The bound on the depth of search is artificial and can lead to many anomalies.**
- **We only consider two:**

1. **Non-quiescence**
   **"quiescent" = inactive, quiet, calm, …**
2. **Horizon Effect**

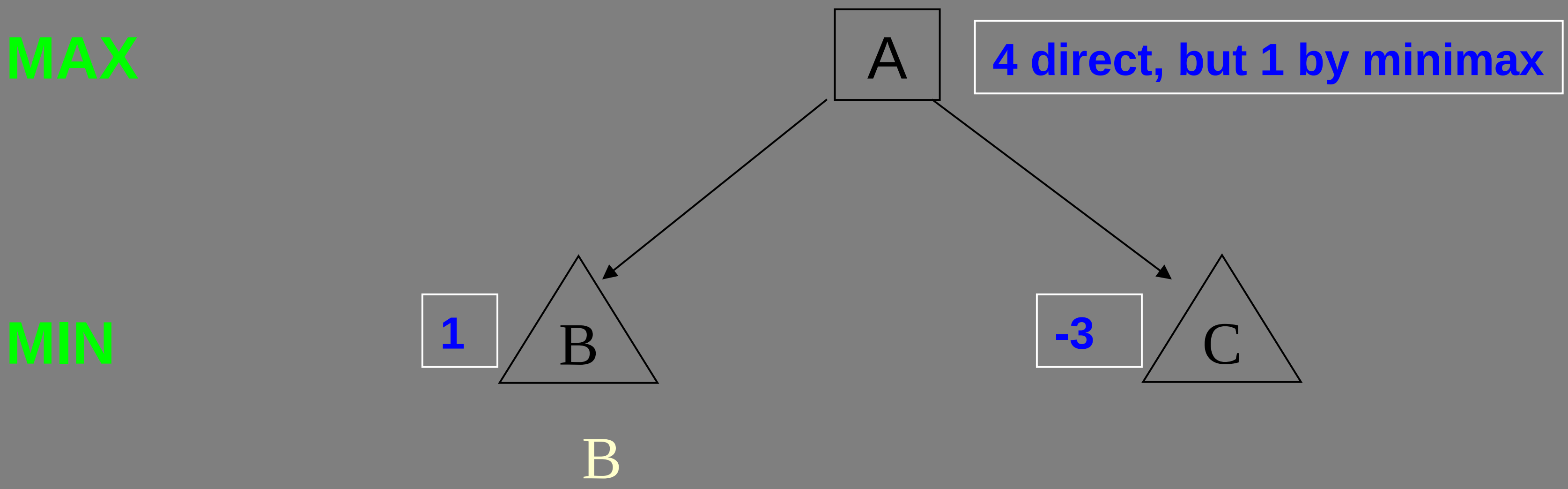
- **(These deficiencies also apply to alpha-beta as it is just a more efficient way to do the same calculation as minimax)**

## *Game Playing – Non-Quiescence*

- **Suppose that change depth bound from k to k+1 – i.e. expand one more move**

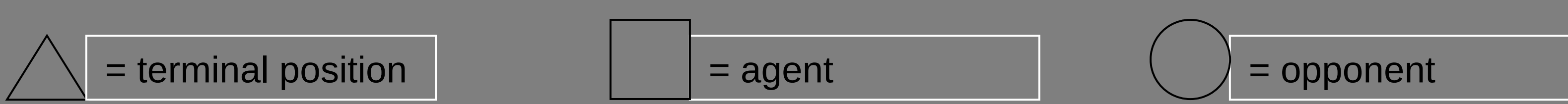
- **The values given to a node might change wildly**

# Example of non-quiescence




Utility values of “terminal” positions obtained by an evaluation function

Direct evaluation does not agree with one more expansion and then using of minimax



## *Game Playing – Quiescence Search*

- **Suppose that change depth bound from k to k+1 – i.e. expand one more move**
- **The values given to a node might change wildly**

- **Keep on increasing the depth bound in that region of the game tree until the values become "quiescent" ("quiet", i.e. stop being "noisy")**

## *Game Playing – Quiescence Search*

- **In quiescence search the depth bound is not applied uniformly but adapted to the local situation**
  - **in this case so that the values are not wildly changing**
- **Many other improvements to minimax also work by adapting to depth-bound to get better results and/or do less work**

## *Game Playing – Horizon Effect*

- **Sometimes there is a bad thing, “X”, such that**
  1. **X cannot be avoided**
  2. **X can be delayed by some pointless moves**
  3. **X is not detected by the evaluation function**
- **In this case depth-limited minimax can be fooled**
- **It will use the pointless moves to push X beyond the depth limit, “horizon”, in which case it will “not see X”, and ignore it.**
- **This can lead the search to take bad moves because it ignores the inevitability of X**

# *Game Playing – Beyond alpha-beta*

- **We looked briefly at two problems**
  - **“non-quiescence”, and the “horizon effect”**
- **and one solution “quiescence search”**

- **To seriously implement a game**
  - **Deep-blue, chinook, etc**
- **it is necessary to solve many such problems!**

- **Good programs implement many techniques and get them to work together effectively**

## *Game Playing – Game Classification*

- **So far have only considered games such as chess, checkers, and nim. These games are:**

1. **Fully observable**
   - **Both players have full and perfect information about the current state of the game**
2. **Deterministic**
   - **There is no element of chance**
   - **The outcome of making a sequence of moves is entirely determined by the sequence itself**

## *Game Playing – Game Classification*

- **Fully vs. Partially Observable**

- **Some games are only partially observable**
- **Players do not have access to the full "state of the game"**
- **E.g. card games – you typically cannot see all of your opponents cards**

## *Game Playing – Game Classification*

- **Deterministic vs. Stochastic**
- **In many games there is some element of chance**
- **E.g. Backgammon – throw dice in order to move**
- **(You are expected to be aware of these simple classifications)**

# *Game Playing – Summary*

- **Game Trees**
- **Minimax**
  - **utility values propagate back to the root**
- **Bounded Depth Minimax**
- **Alpha-Beta Search**
  - **uses DFS**
  - **with depth bound**
  - **ordering of nodes is important in order to maximise pruning**
- **Deficiencies of Bounded Depth Search**
  - **Non-quiescence**
    - **Combat using quiescence search**
  - **Horizon Problem**
    - **Combat with ?? (look it up!)**

*End of Game Playing*

**Garry Kasparov and Deep Blue. © 1997, GM Gabriel Schwartzman's Chess Camera, courtesy IBM.**