Intelligent Game Playing

Instructor: **B. John Oommen**

Chancellor's Professor

Fellow : IEEE ; Fellow : IAPR

School of Computer Science, Carleton University, Canada

The primary source of these notes are the slides of Professor Hwee Tou Ng from Singapore. The Multi-Player Game section was due to Mr. Spencer Polk. **I sincerely thank them for this**.

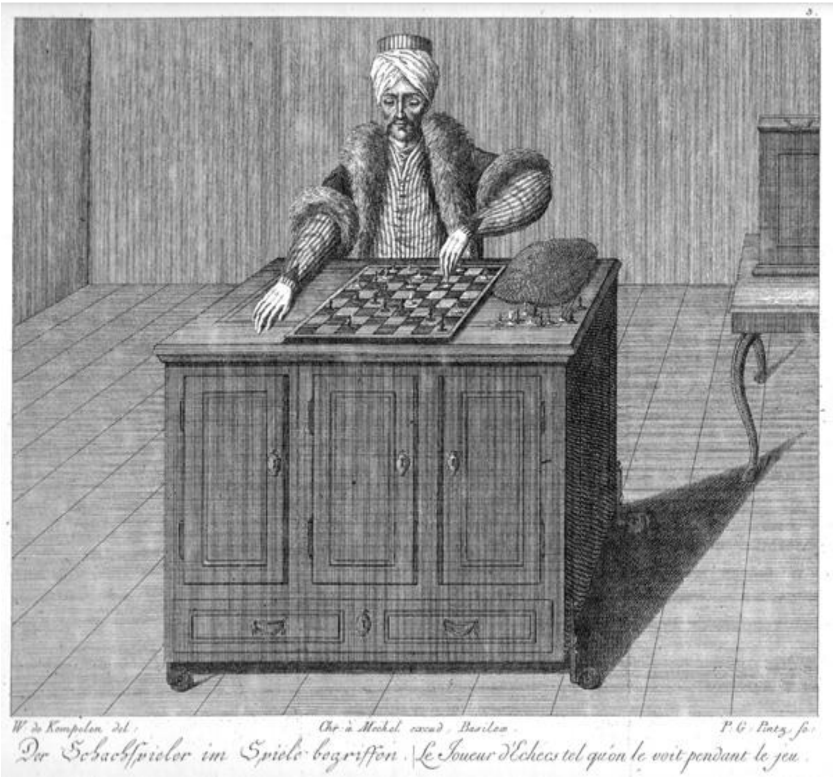
Games - Introduction

- Question: Can machines outplay humans?

- Captured imaginations for centuries
  - Appearance in myth and legend
  - Popular topic in fiction

- Thanks to AI and search techniques, the dream has come true!

History



The Turk (1770)

- “The Turk”: In 1770
- A chess playing machine
- Toured Europe
- Facing well-known opponents
  - e.g. Napoleon, Ben Franklin
- Of course: Revealed - fraud

History

- “The Turk” shows how fascinating this idea is

- 1914: King vs Rook strategies by automaton

- True AI game playing – Claude Shannon: 1950
  - Based on earlier work by Nash and Neumann

- Shannon's algorithm still used
  - **Mini-Max Search** (we will return to it shortly)

History

- Shannon's 1950 paper focused on Chess
  - Chess remains very important to game playing research
- At the time, seen as purely theoretical exercise
- 1970s: First commercial Chess programs
- 1980s: Chess programs playing at Expert level
  - Still some time until Grandmaster level...

History

Kasparov *vs* Deep Blue

- 1997: IBM's Deep Blue
  - Defeats Garry Kasparov
- First defeat of Grandmaster
- Field: Branched out since
  - Poker, Go: Now important games
  - IBM Watson on Jeopardy

Games vs. Search Problems

- "Unpredictable" opponent
  - Specifying a move for every possible opponent reply
- Time limits
  - Unlikely to find goal, must approximate

Mini-Max Search

- Search to find the correct move in a two player game
- The optimal solution:
  - Exponential algorithm
  - Generate all possible paths
  - Only play those that lead to a winning final position
- Realistic alternative to the Optimal
- Use finite depth look-ahead with a heuristic function
- Evaluate how good a given game state is

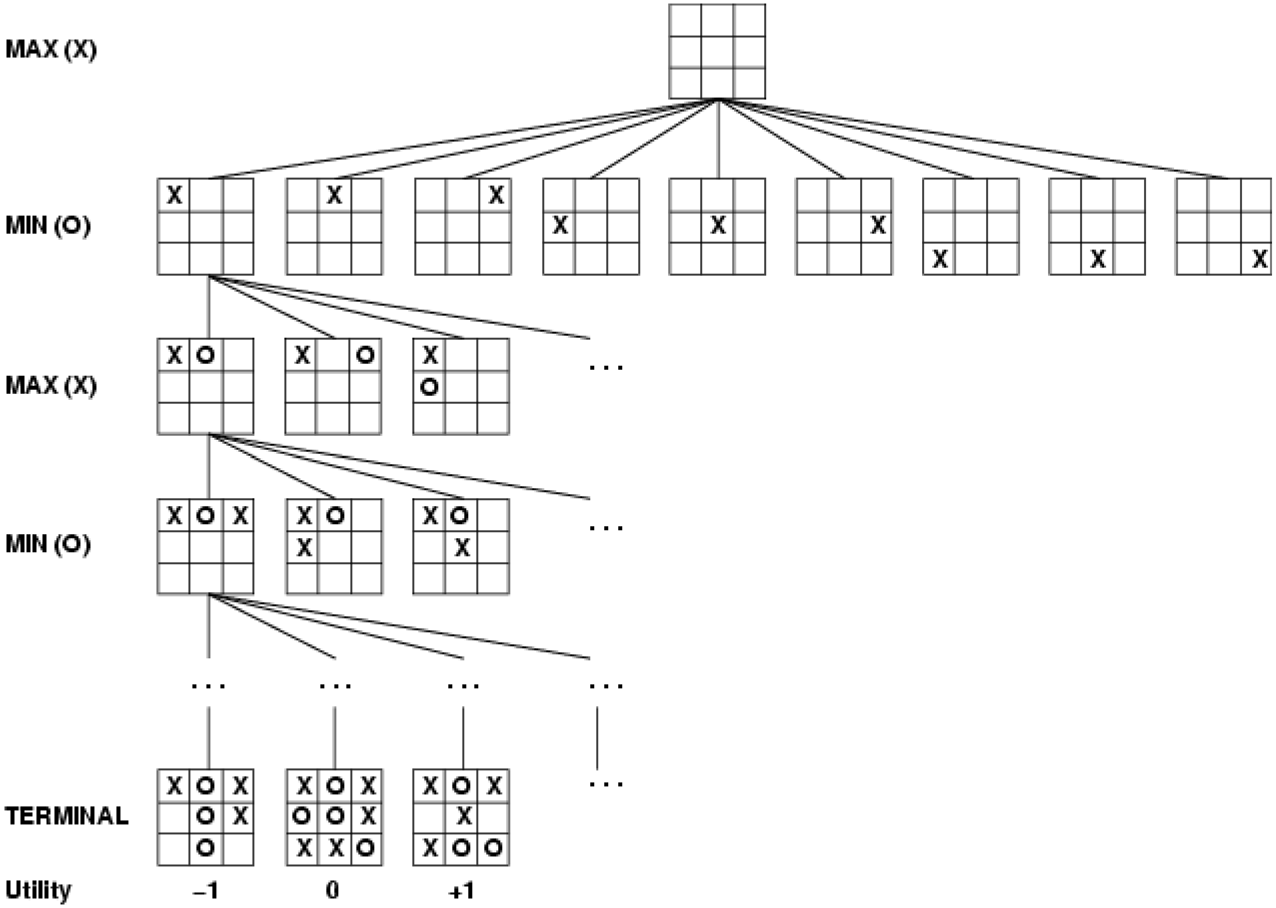
Mini-Max

- Extend Tree down to a given search depth

- Top of tree is the Computer’s move
  - Wants move to ultimately be one step closer to a winning position
  - Wants move that maximizes own chance of winning

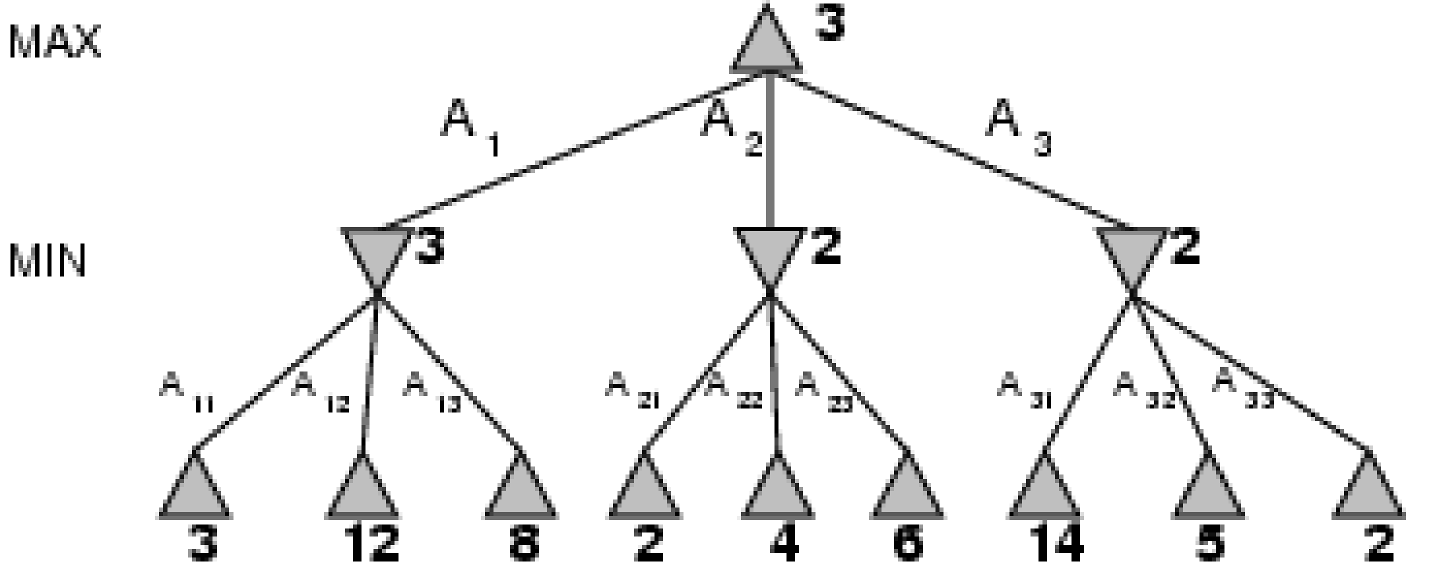
- Next move is Opponent’s
  - Opponent assumed to perform a move that his best
  - Wants move that minimizes Computer’s chance of winning

Game tree
2-player, Deterministic, Turns



Mini-Max

- **Perfect** play for deterministic games
- **Idea**: Choose move to position with highest Mini-Max value
  = Best achievable payoff against best play
- Example: 2-ply game**:**



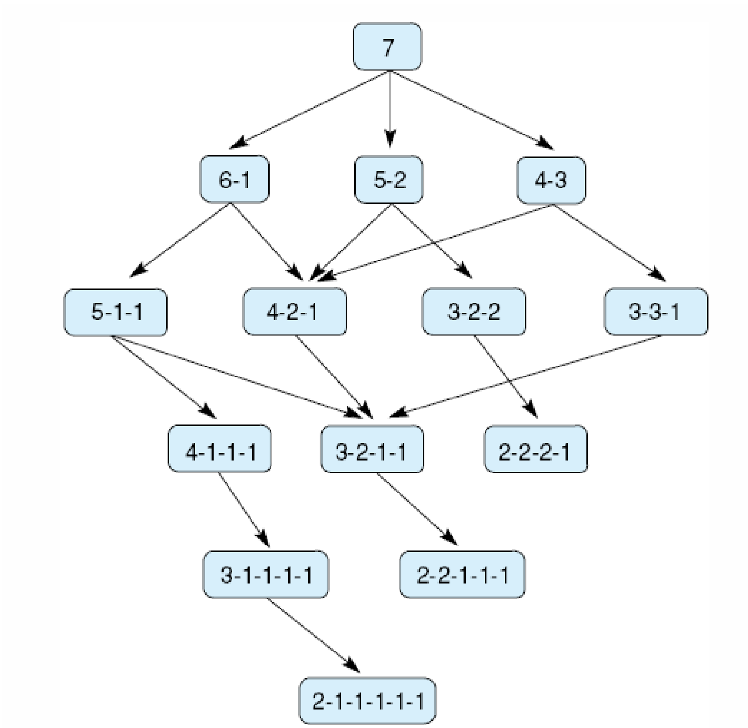
Mini-Max for Nim

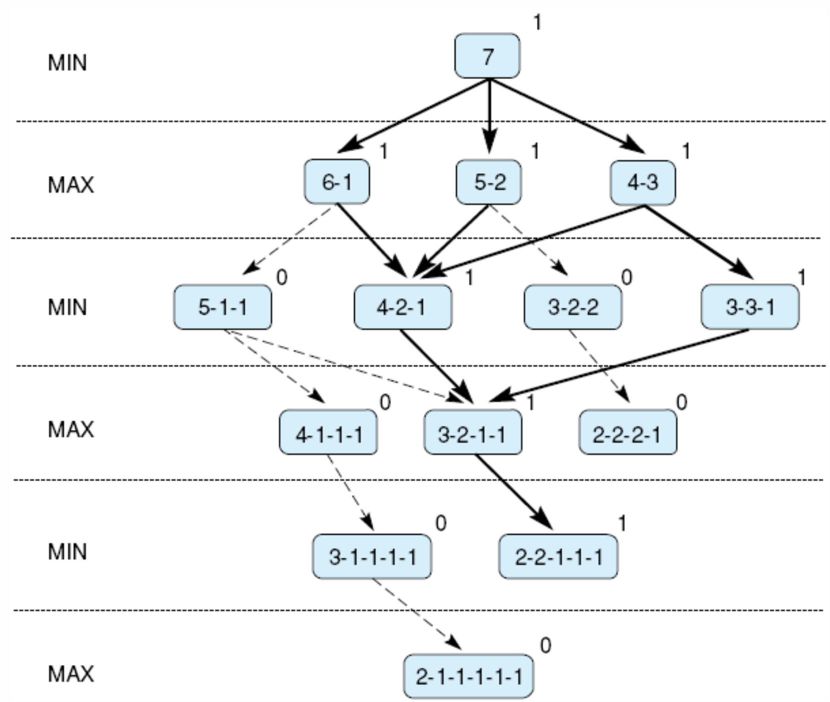
- Nim Game
  - Two players start with a pile of tokens
  - Legal move: Split (any) existing pile into two non-empty differently sized piles
  - Game ends when no pile can be unevenly split
  - Player who cannot make his move loses the game
- Search strategy
  - Existing heuristic search methods do not work

Mini-Max for Nim

- Label nodes as MIN or MAX, alternating for each level
- Define utility function (payoff function).
- Do full search on tree
  - Expand all nodes until game is over for each branch
- Label leaves according to outcome
- Propagate result up the tree with:
  - M(n) = max(child nodes) for a MAX node
  - m(n) = min(child nodes) for a MIN node
- Best next move for MAX is the one leading to the child with the highest value (and vice versa for MIN)

Mini-Max for Nim





Mini-Max Algorithm

function MINIMAX-DECISION(game) **returns** *an operator*
for each *op* **in** OPERATORS[game] **do**
    VALUE[op] := MIN-VALUE(APPLY(*op*, *game*), *game*)
end
return the *op* with the highest VALUE[op]

function MAX-VALUE(*state*, *game*) **returns** *a utility value*
if CUTOFF-TEST(*state*,) **then return** EVAL(*state)*
value := - ∞
for each *s* **in** SUCCESSORS(*state*) **do**
    value := MAX(value, MIN-VALUE(*s*, *game*))
end
return value

function MIN-VALUE(*state*, *game*) **returns** *a utility value*
if CUTOFF-TEST(*state*,) **then return** EVAL(*state)*
value := ∞
for each *s* **in** SUCCESSORS(*state*) **do**
    value := MIN(value, MAX-VALUE(*s*, *game*))
end
return value

Problems with Mini-Max

- Horizon effect: Can’t see beyond depth
  - Due to exponential increase in tree size, only very limited depth feasible
  - Solution: Quiescence search. Start at the leaf nodes of the main search, and try to solve this problem.
  - In Chess, quiescence searches usually include all capture moves, so that tactical exchanges don't mess up the evaluation. In principle, quiescence searches should include any move which may destabilize the evaluation function--if there is such a move, the position is not quiescent.
- May want to use look up tables
  - For end games
  - Opening moves (called Book Moves)

Properties of Mini-Max

- **Complete?**
  - Yes (if tree is finite)
- **Optimal?**
  - Yes (against an optimal opponent)
- **Time complexity?**
  - $O(b^m)$
- **Space complexity?**
  - O(bm) (depth-first exploration)
- Chess: b ≈ 35, m ≈100 for "reasonable" games
  - Exact solution completely infeasible

Branch and Bound: The α-β Algorithm

- Branch and Bound: If current path (branch) is already worse then some other known path:
  - Stop expanding it (bound).
- Alpha-Beta is a branch and bound technique for Mini-Max search
- If you know that the level above won't choose your branch because you have already found a value along one of your sub-branches that is too good, stop looking at other sub-branches that haven't been looked at yet

The α-β Algorithm

- Instead of maintaining a single mini-max value , the α-β pruning algorithm, maintains two: α, β
- Together provide a bound on the possible values of the mini-max tree at any given point.
- At any given point, α: minimum the player can expect to receive
- At any given point, β: maximum value the player can expect

The α-β Algorithm

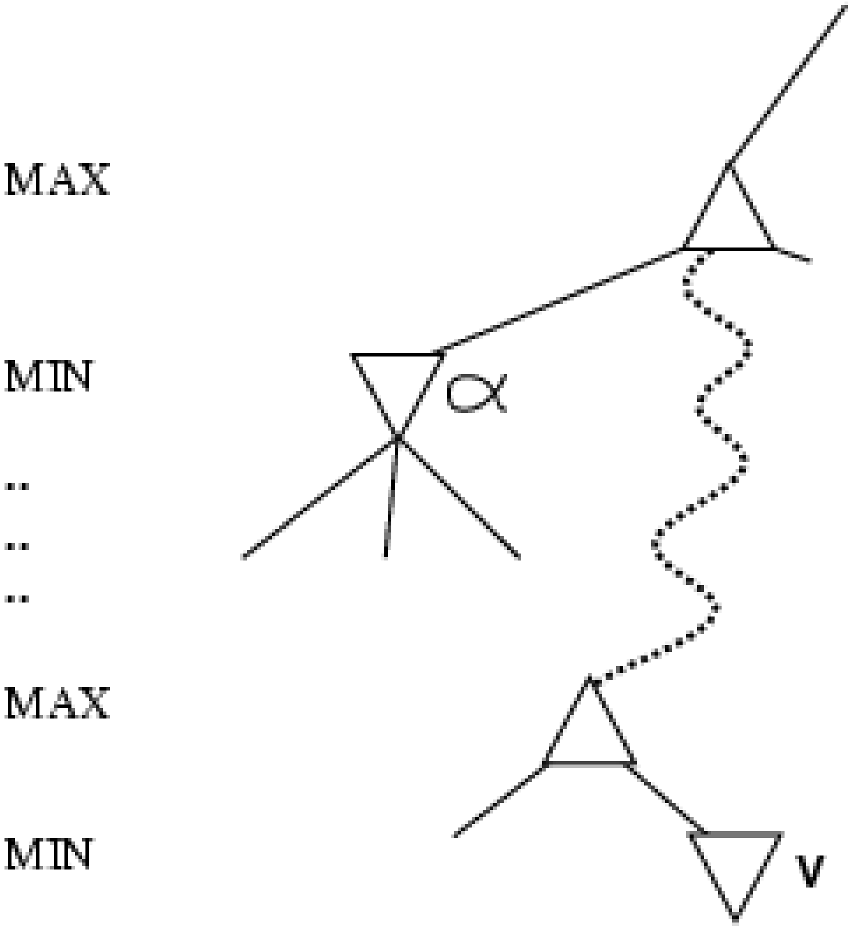
- If it is ever the case that this bound is reversed or has range of 0 ($\beta <= \alpha$), then better options exist for the player at other pre-explored nodes
- As α is the minimum value we know we can get
- Thus this node cannot be the mini-max value of the tree.
- There is no point in exploring any more of this node's children
- Potentially saving considerable computation time in a game with a large branching factor/depth

Properties of α-β

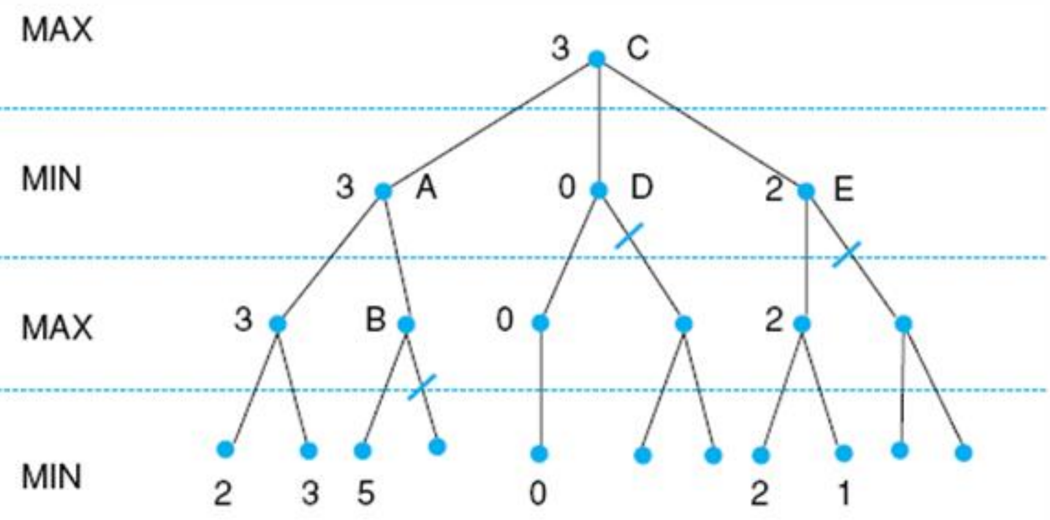
- Pruning does not affect final result
- Good move ordering improves pruning effectiveness
- With "perfect ordering" time complexity = $O(b^{m/2})$
  - Doubles depth of search
- α-β is a simple example of the value of reasoning about which computations are really relevant

Why it is called α-β

- α: Value of the best 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*

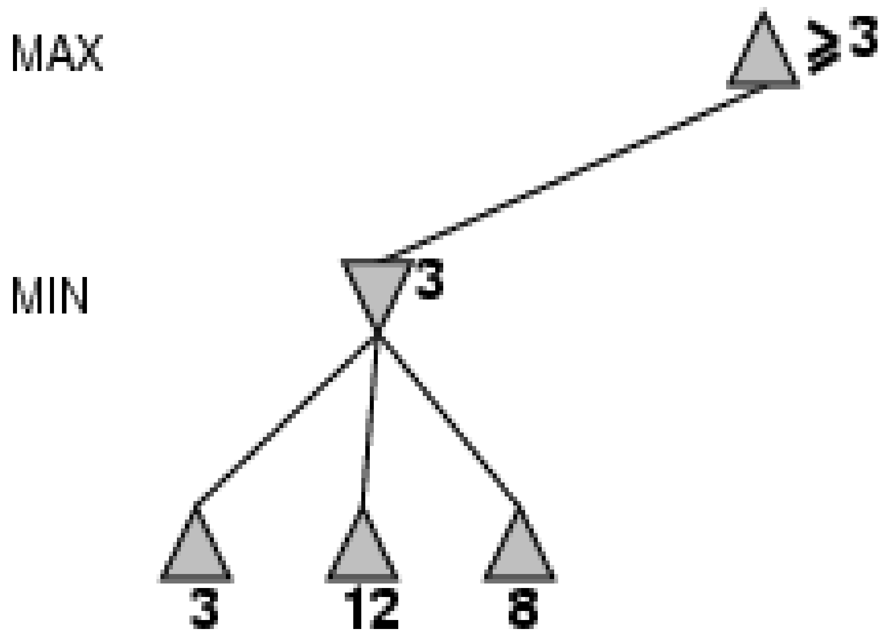


Effects of α-β

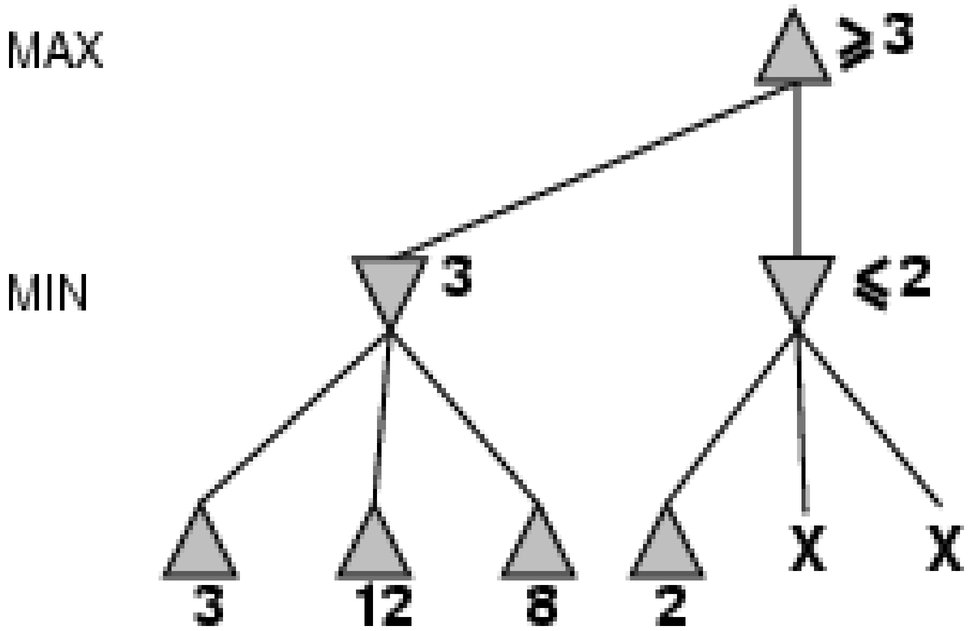


A has $\beta = 3$ (A will be no larger than 3)
B is β pruned, since $5 > 3$
C has $\alpha = 3$ (C will be no smaller than 3)
D is α pruned, since $0 < 3$
E is α pruned, since $2 < 3$
C is 3

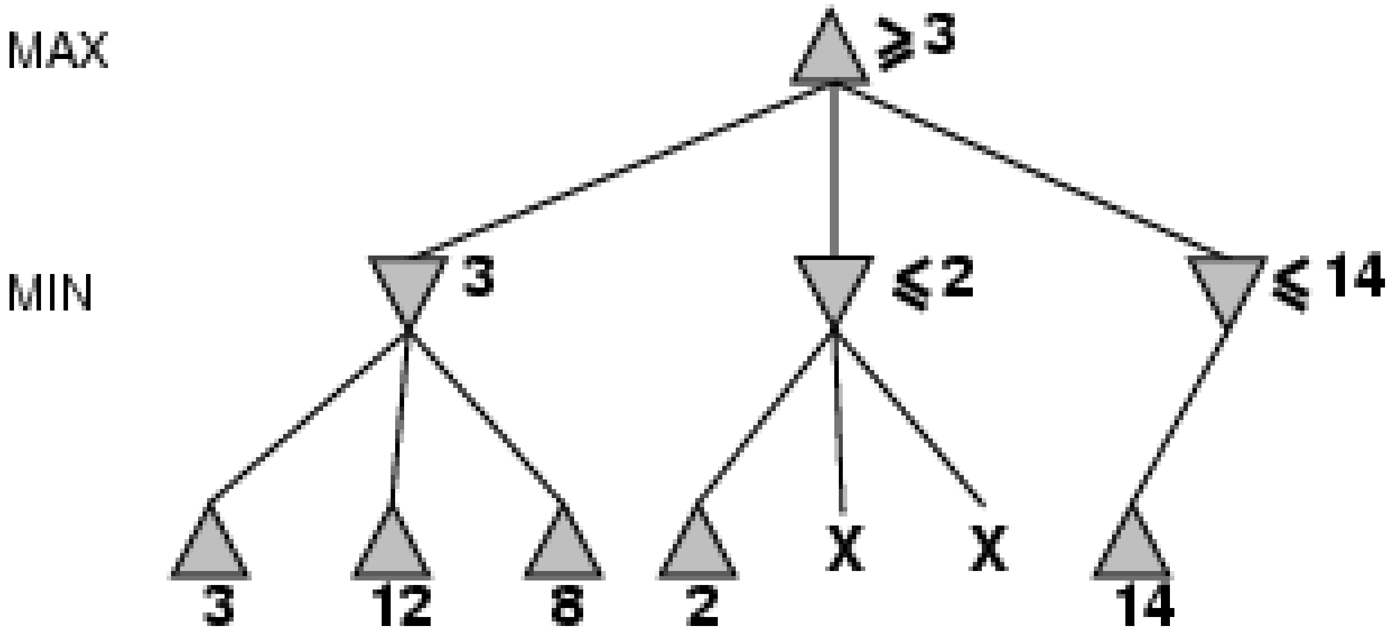
Example: α-β Pruning



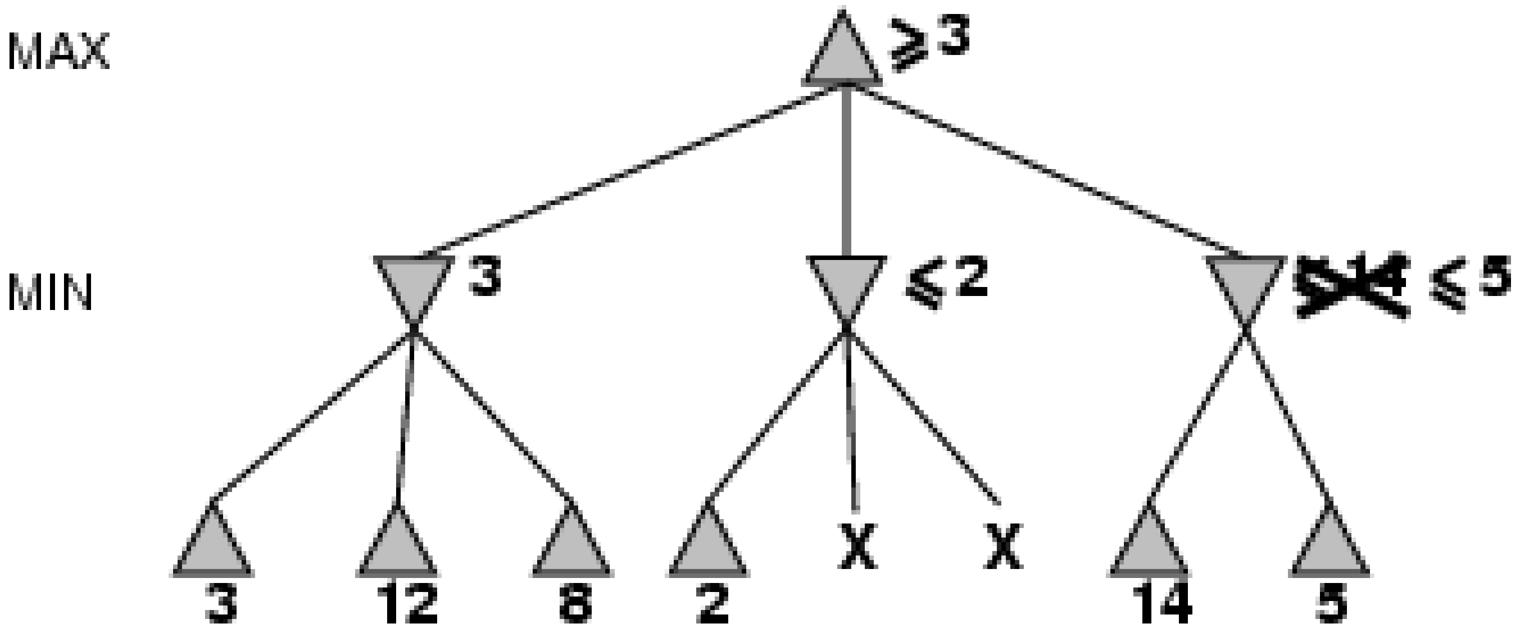
Example: α-β Pruning



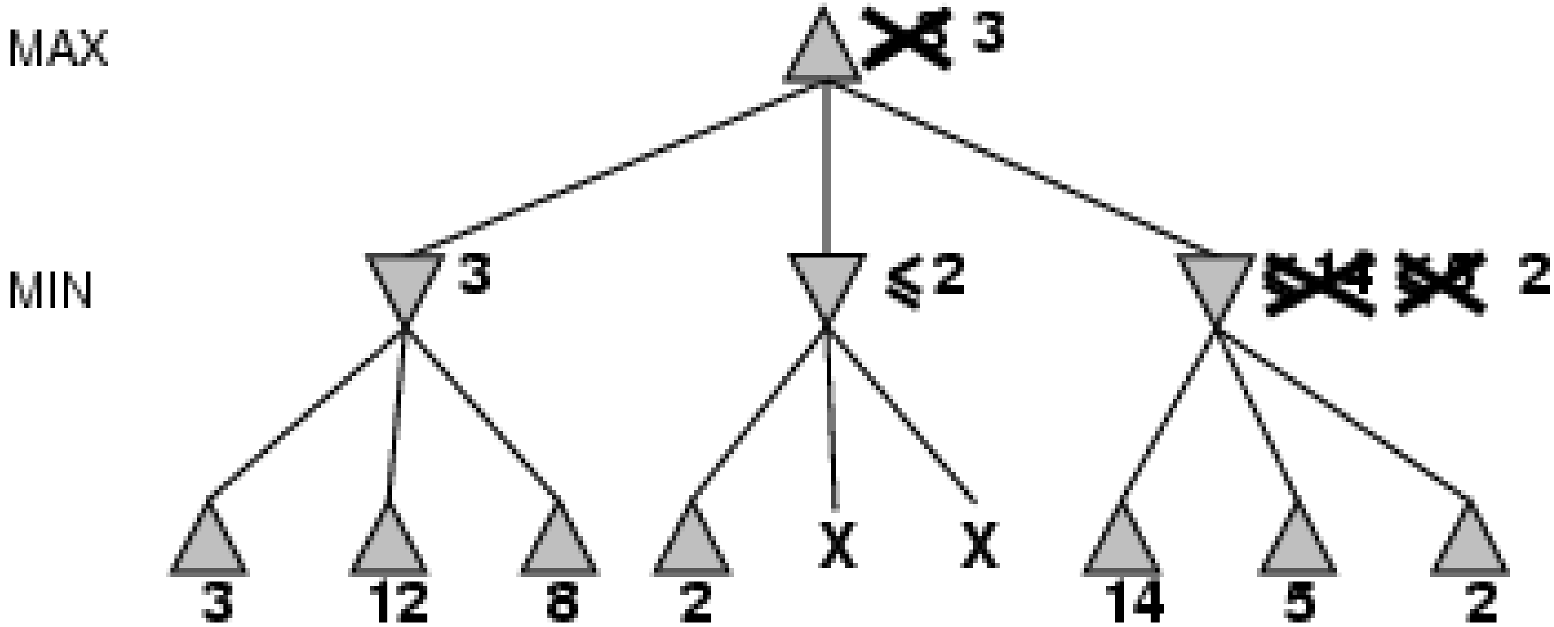
Example: α-β Pruning



Example: α-β Pruning



Example: α-β Pruning



The α-β Algorithm

- From Russell and Norvig

α = best score for MAX so far
β = best score for MIN so far

game = game description
state = current state in game

function MAX-VALUE(*state*, *game*, *α*, *β*) **returns** *a utility value*

if CUTOFF-TEST(*state*,) **then return** EVAL(*state*)
for each *s* **in** SUCCESSORS(*state*) **do**
    α := MAX(*α*, MIN-VALUE(*s*, *game*, *α*, *β*))
    if $\alpha \geq \beta$ **then return** α
end
return α

function MIN-VALUE(*state*, *game*, *α*, *β*) **returns** *a utility value*

if CUTOFF-TEST(*state*,) **then return** EVAL(*state*)
for each *s* **in** SUCCESSORS(*state*) **do**
    β := MIN(*β*, MAX-VALUE(*s*, *game*, *α*, *β*))
    if $\beta \leq \alpha$ **then return** β
end
return β

The α-β Algorithm

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

  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  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*)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ 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*)
  $v \leftarrow +\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(s, α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Improving Game Playing

- Increase Depth of Search
- Have better heuristic for game state evaluation

Changing Levels of Difficulty

- Increase Depth of Search

Resource Limits

- Suppose we have 100 secs, explore 10^4 nodes/sec
  - 10^6 nodes per move

- Standard approach:
  - Cutoff test: Depth limit (perhaps add quiescence search)
- Evaluation function:
  - Estimated desirability of position

Evaluation Functions

- Chess, typically linear weighted sum of features

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

- Example: $w_1 = 9$ with

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

  etc.

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?

$b^m = 10^6$, $b=35 \rightarrow m=4$

4-ply lookahead is a hopeless chess player!

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

Quiescence search

- Quiescence search: Study moves that are noisy
- They appear good, but moves around them - bad
- Investigate them with a localized leaf search
- Attempt to identify delaying tactics and change the seemingly-good value of the node
- A very natural extension of mini-max
- Simply run search again at a leaf node until that leaf node becomes quiet
- As with iterative deepening, running time of the algorithm won’t increase by more than a constant

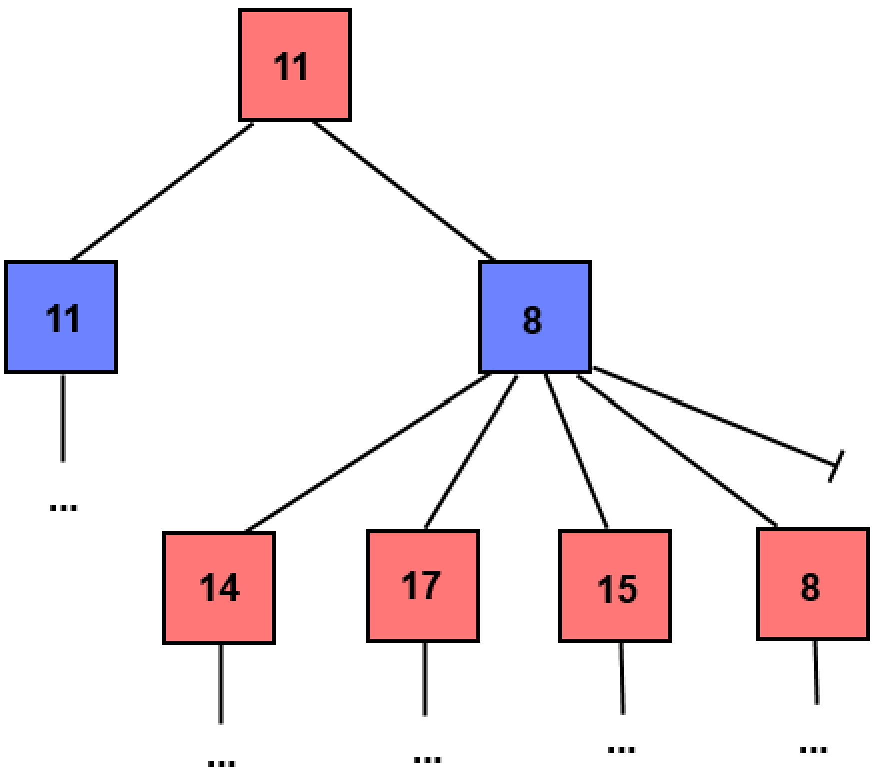
Real Deterministic Games

- 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 Kasparov in a six-game match in 1997.
  - Deep Blue searches 200 million positions per second
  - Uses very sophisticated evaluation
  - Undisclosed methods for extending some lines of search up to 40 ply.

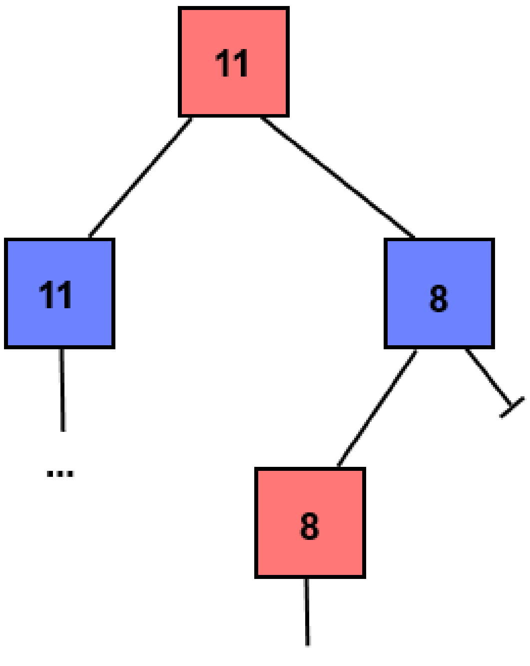
Move Ordering

- Best possible pruning is achieved if the best move is searched first at each level of the tree
- Problem: If we knew the best move, we would not need to search!
- Thus, we employ move ordering *heuristics*, which search the best move first
- **Example:** In Chess, search capturing moves before non-capturing moves
- **What we want:** domain *independent* techniques

Example: Poor Move Ordering



Example: Good Move Ordering



Principal Variation Move

- As it is a search algorithm, can apply Iterative Deepening to Mini-Max
- At each level, we thus find a move path we expect us and the opponent to take
- At the next stage, search it first!
  - Called **Principal Variation** move
- Even though Iterative Deepening takes some time, PV-move can greatly improve overall performance!

Other Heuristics

- **Killer Moves**: Remember move that produced a cut on this level of the tree
  - If we encounter it again, search it first!
  - Normally remember two moves per level

- **History Heuristic**: Same as Killer Moves, want to remember moves that produce cuts
  - Want to use info on all levels of tree
  - Hold array of counters, increment based on level cut occurred at
  - Details outside scope of this talk

Real Deterministic Games

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

Things to Remember: Games

- Games are fun to work on!
- They illustrate several important points about AI
- Perfection is unattainable
- Must approximate paths and solutions
- Good idea to think about what to think about

Two Player to Multi-Player Games

- Mini-Max: Originally envisioned for Chess
  - Two player, deterministic, perfect information game
- What if we want to play a *multi*-player game?
  - Instead of two players, we have *N* players, where *N* > 2
  - Examples: Chinese Checkers, Poker
- New challenges, requiring new techniques!

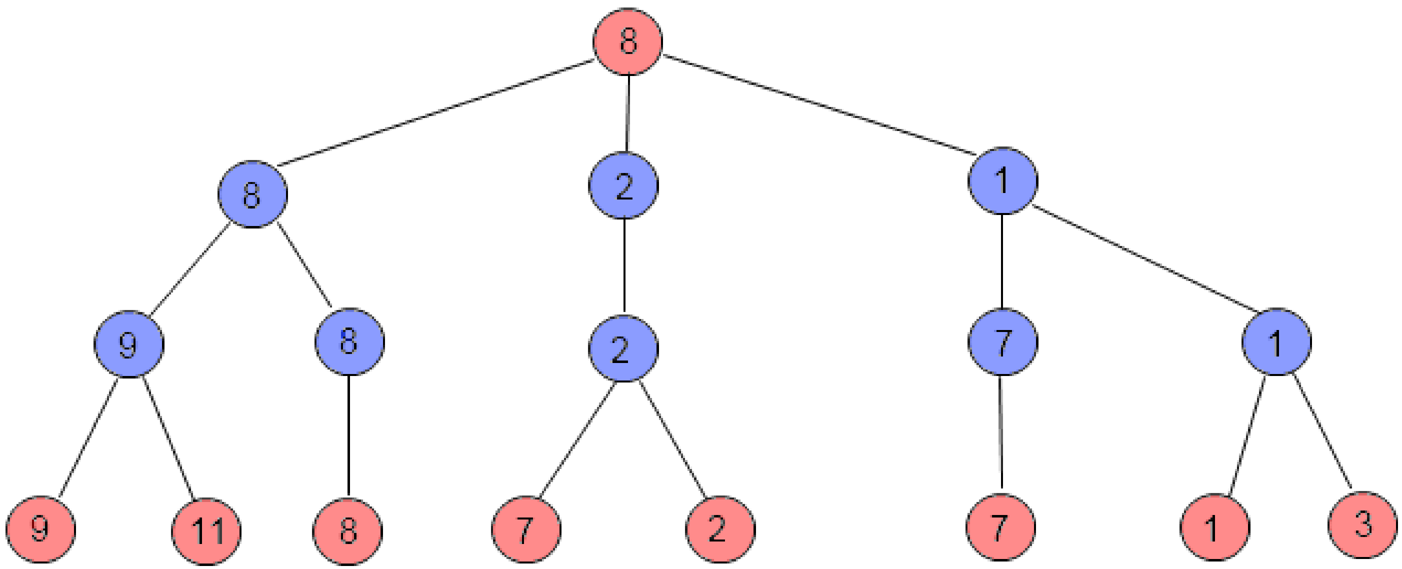
Qualities of Multi-Player Games

- In two player zero sum games, your gain is reflected in equal loss for opponent
  - No longer true for multi-player game
  - Loss spread between multiple opponents
- *Coalitions* may arise during play
- More opponent turns occur between perspectives

Extending Mini-Max to Multi-Player Games

- Problem: Mini-Max operates using a single value
  - Worked for two player games, as opponent's gain is our loss
- Single score very valuable – Allows **pruning**
  - Would like to keep pruning to speed up the search
- Simple solution: *All* opponents minimize our score
  - So, MAX-MIN-MIN, MAX-MIN-MIN-MIN, etc
- Called the *Paranoid Algorithm*

Paranoid Algorithm



Sample Paranoid Tree (Red MAX, Blue MIN)

Paranoid Algorithm

```
function integer paranoid(node, depth):
    if node is terminal or depth <= 0 then
        return heuristic value of node
    else
        if node is max then
            val = −∞
            for all child of node do
                val = max(val, paranoid(child, depth − 1)
            end for
        else
            val = ∞
            for all child of node do
                val = min(val, paranoid(child, depth − 1)
            end for
        end if
        return val
    end if
```

Paranoid Algorithm

- Algorithm *exact same* as Mini-Max in many implementations
- Pros
  - Easy to implement and understand
  - Subject to α-β pruning on MAX/MIN borders
  - Not for phases between MIN nodes
- Cons
  - Views all opponents as a coalition – leads to bad play
  - Limited look-ahead for perspective player
  - Need to have multiple MIN phases in a row

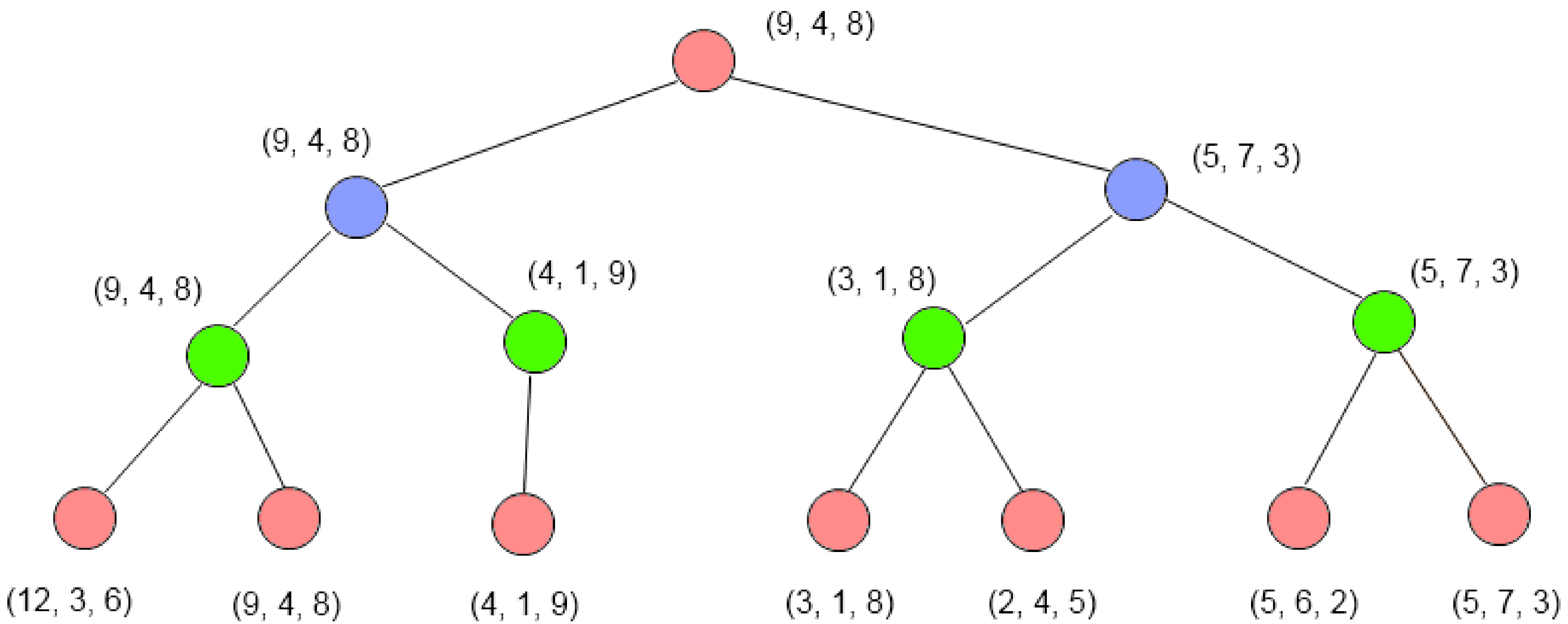
Max-N Algorithm

- 1986: Luckhardt and Irani
- Addresses coalition problem of Paranoid
- Keeps *tuple* of scores, not one value
- Assumption: Players maximize their *own* score
  - No consideration for other players

- Heuristic returns value for each player
  - i.e. [6, 3, 8] for three-player game

- *N*th player maximizes *N*th value

Max-N Algorithm



Sample Max-N Tree

Max-N Algorithm

```
function integer[] max-n(node, depth):
    if node is terminal or depth <= 0 then
            return heuristic value of node
    else
            val = −∞
            tuple = []
            for all child of node do
                    val = max(val, max-n(child; depth − 1)[node.player])
                    if val changed
                            tuple = max-n(child; depth-1)
                    end if
            end for
            return tuple
    end if
```

Max-N Algorithm

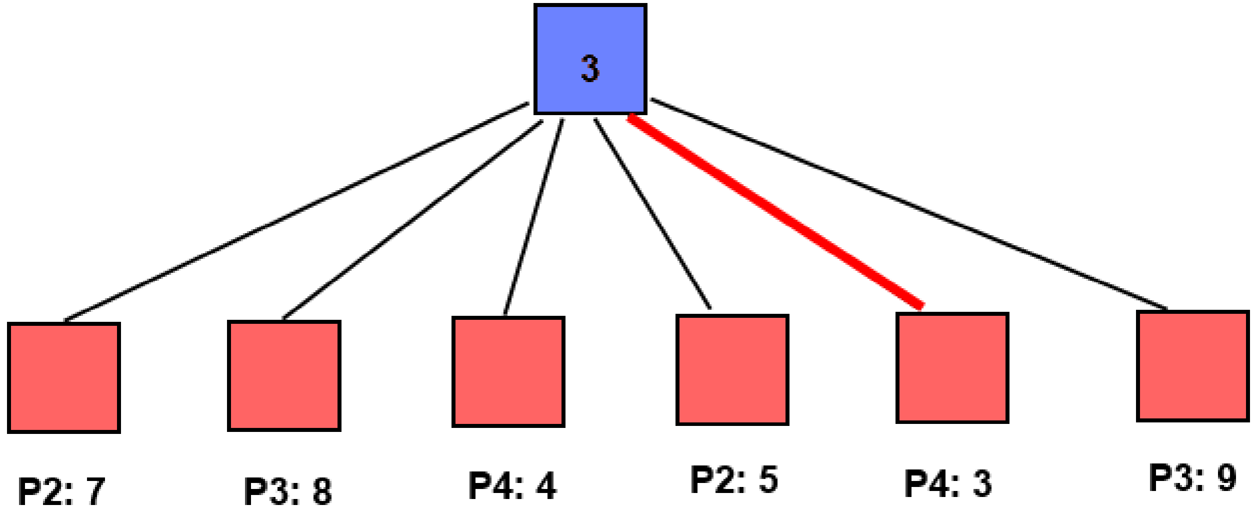
- In terms of raw Mini-Max, very simple extension
- Pros
  - Players “look out for number one”
  - More realistic play
  - Perspective player can see more opportunities
  - Reason: Possibilities are not excluded as readily
- Cons
  - Pruning is very complicated, and not as good
  - Can be worse than Paranoid due to decreased search depth

Best-Reply Search

- Relatively new: 2011 (Schadd and Winands)
- *All* opponents considered to be *one* player
  - They only get ONE turn between them

- Only opponent with best move is thought to act
- Return to MAX-MIN-MAX-MIN...
- Essentially a return to Mini-Max algorithm
  - With a *very powerful* opponent!!

Best-Reply Search



Sample BRS Tree (one level)

Best-Reply Search

```
function integer best-reply(node, depth):
    if node is terminal or depth <= 0 then
            return heuristic value of node
    else
            if node is max then
                    val = −∞
                    for all child of node do
                            val = max(val, best-reply(child; depth − 1)
                    end for
            else
                    val = ∞
                    for all opponents do
                            for all opponent’s child at node do
                                    val = min(val; best-reply(child;
    depth − 1)
                            end for
                    end for
            end if
    end if
```

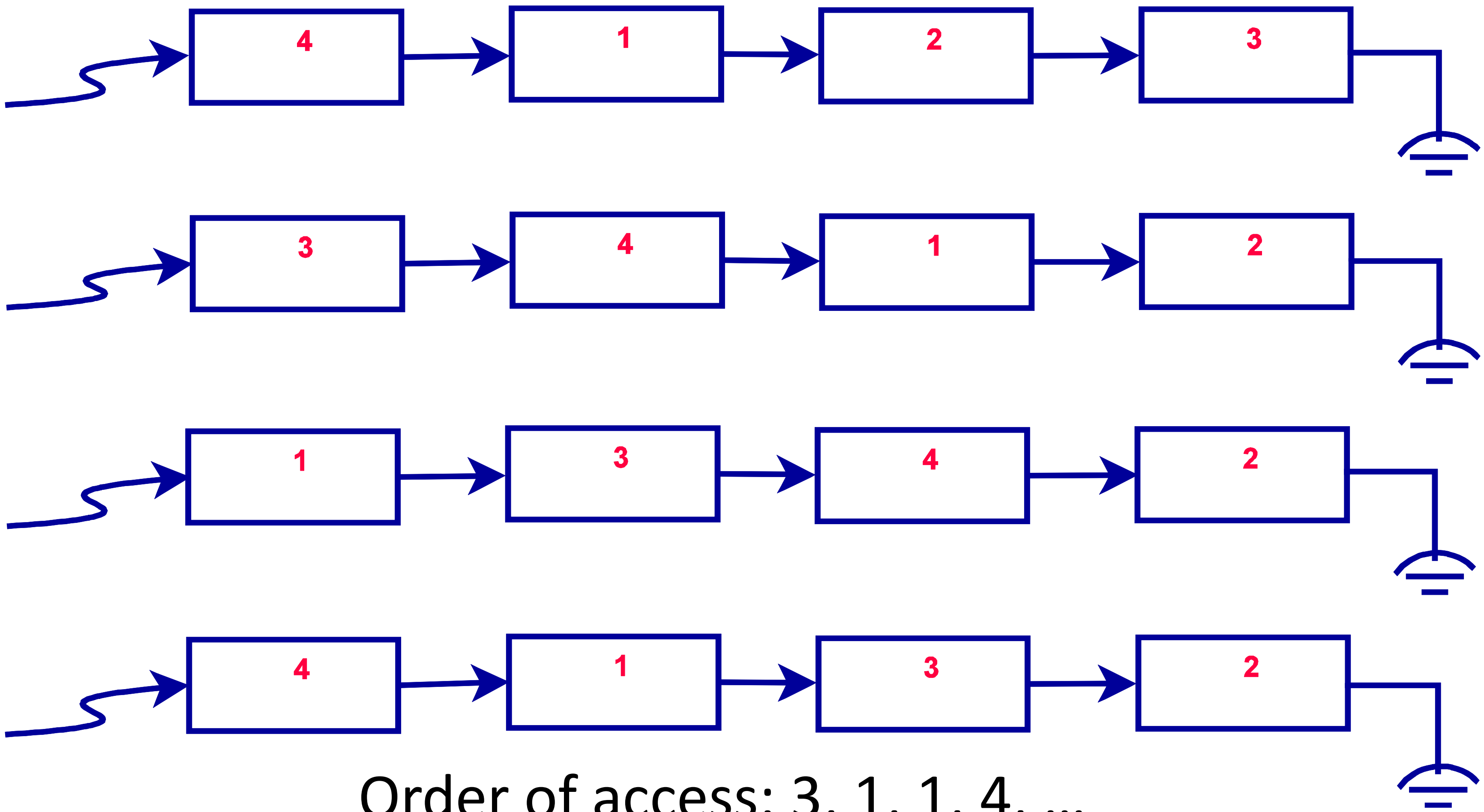
Best-Reply Search

- Attempt to get "best of both worlds"
- Pros
  - Balance between coalition and free-for-all
  - Allows α-β pruning
  - Significant lookahead for perspective player
- Cons
  - Illegal game states analyzed
  - Not applicable to some games
  - This is the domain of some current research (2015)

Adaptive Data Structures

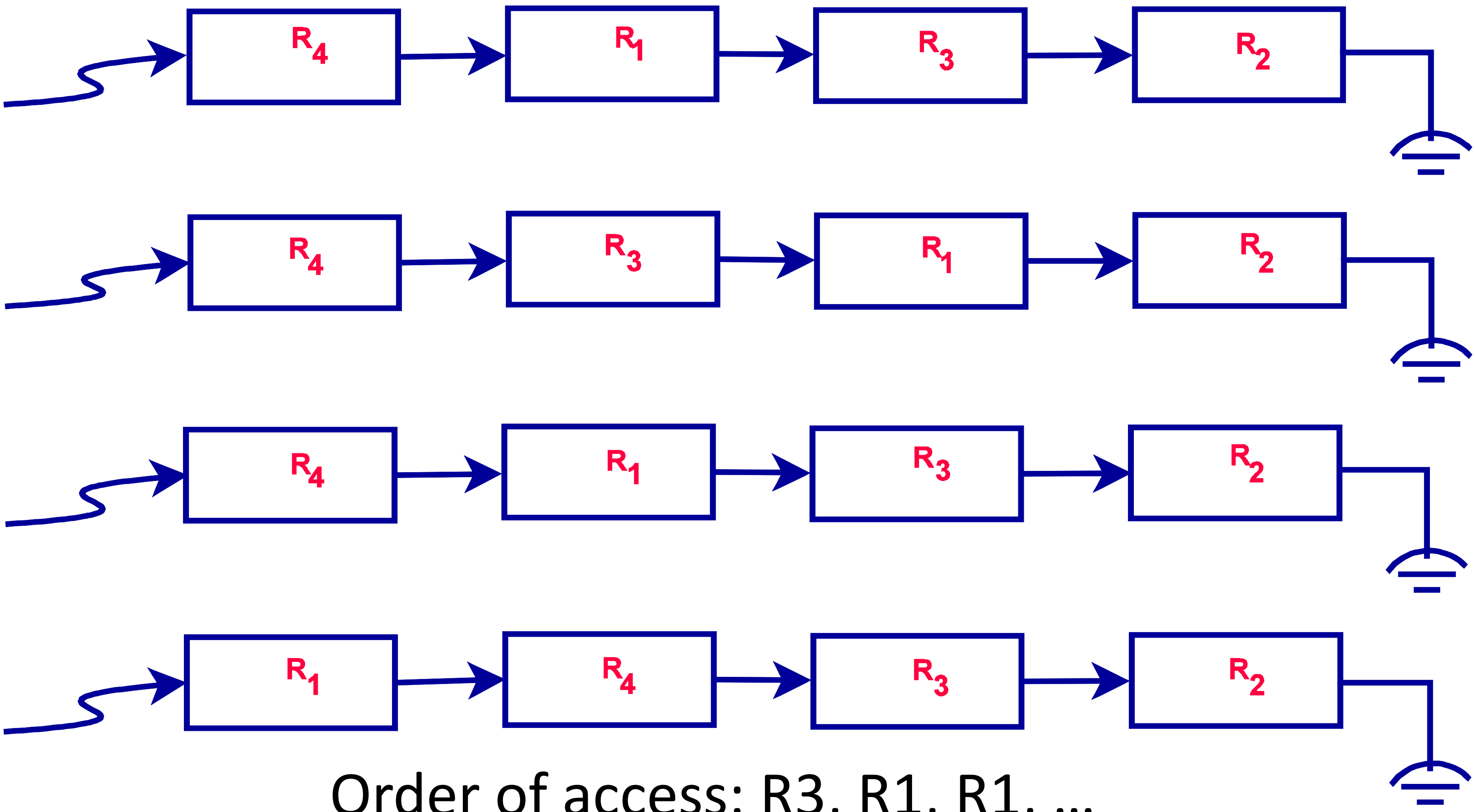
- Other, completely unrelated field
- Concerned with record access frequency
- Problem:
  - Elements in data structure accessed with different frequency
- Solution:
  - Change the structure of the data structure as elements queried
- Can use list, tree or others

ADS – Move to Front Rule



Order of access: 3, 1, 1, 4, …

ADS – Transposition Rule

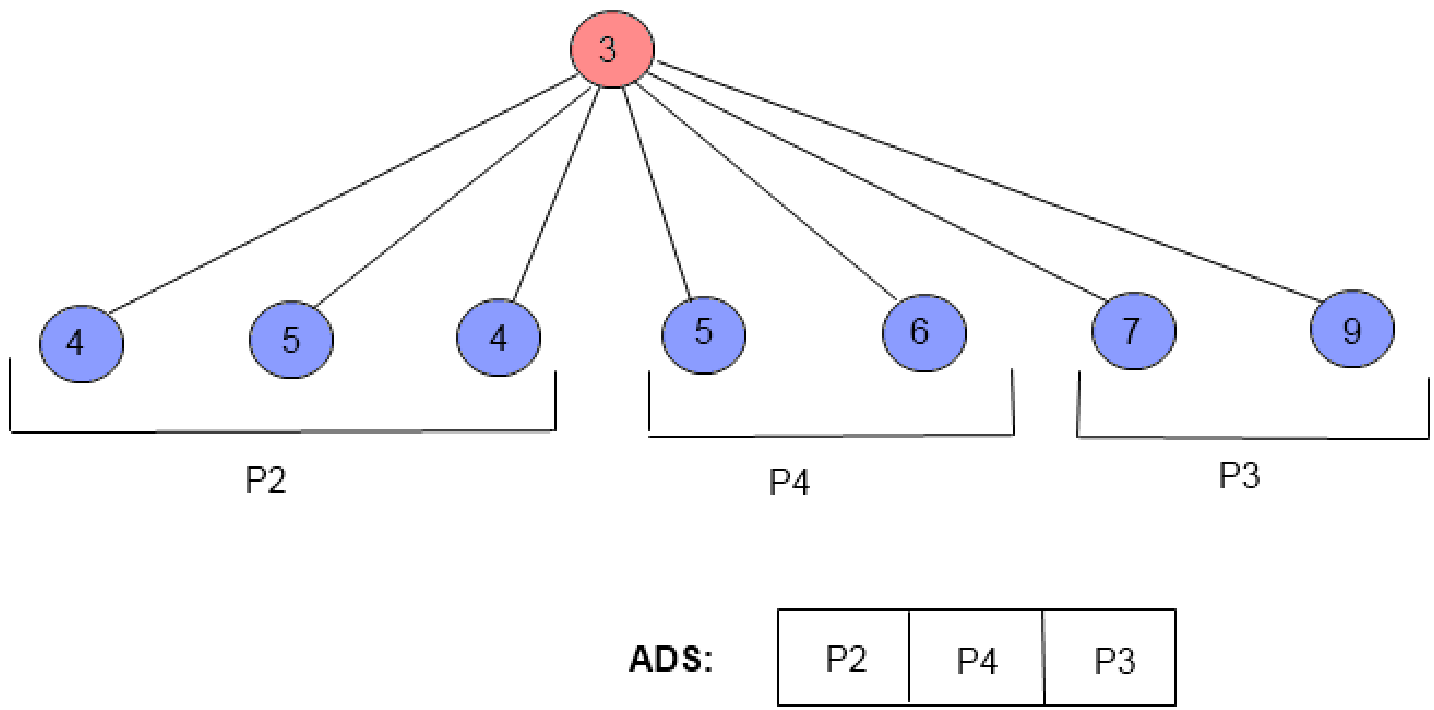


Order of access: R3, R1, R1, …

The Threat-ADS Heuristic

- Our contribution, usable with the BRS
- ADS operations are constant, and small
- We use an ADS that contains **opponents**
- When an opponent is found to have the most *minimizing move*, we query the ADS
- ADS moves over time to **relative opponent threats**
- When grouping moves, do it in the order of the ADS
- Improves **move ordering**, leading to better pruning!

The Threat-ADS Heuristic



BRS with Threat-ADS (one level)

BRS with Threat-ADS

```
function integer brs_threat_ads(node, depth):
    if node is terminal or depth <= 0 then
        return heuristic value of node
    else
        if node is max then
            val = −∞
            for all child of node do
                val = max(val, best-reply(child; depth − 1)
            end for
        else
            val = ∞
            for all opponents in ADS do
                for all opponent's child at node do
                    val = min(val; best-reply(child; depth − 1)
                end for
            end for
            ADS.update(val.opponent)
        end if
    end if
```

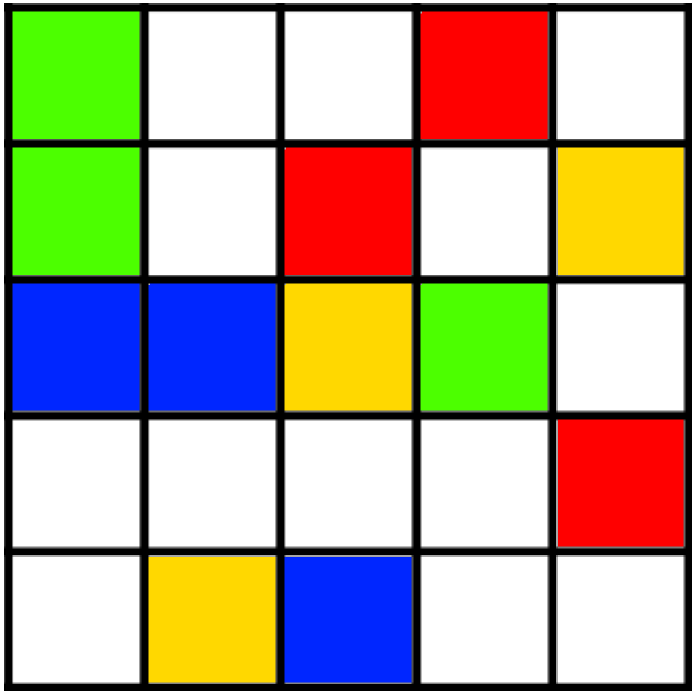
Experimental Framework

- Game needed to test Threat-ADS heuristic
- Needs:
  - BRS must be applicable
  - Game should be simple to implement
- Use established games Focus and Chinese Checkers
- Also develop the Virus Game

Virus Game

- Turn based game with N players
- Played on 2D board
- Goal is to eliminate all other players
- Turn: Player “infects” a square they are adjacent to
- All nearby squares, according to a configured pattern, are given to that player

Virus Game



Experimental Configuration

- One player: BRS with Threat-ADS
- Others: Random (Interested in *tree pruning*)
- Take *Node Count* over first few turns of the game
  - Count each node expanded, but not those pruned
- Average over 50 games
- Run for each of three games mentioned
- Run over a variety of configurations
  - Varying number of players
  - Varying starting state

Results (Initial Board State)

Game	Threat-ADS?	Avg. Node Count
Virus Game	No	264,000
Virus Game	Yes	237,000
Focus	No	6,859,000
Focus	Yes	6,443,000
Chinese Checkers	No	3,485,000
Chinese Checkers	Yes	3,070,000

Results (Midgame Board State)

Game	Threat-ADS?	Avg. Node Count
Virus Game	No	307,000
Virus Game	Yes	275,000
Focus	No	14,460,000
Focus	Yes	13,050,000
Chinese Checkers	No	8,170,000
Chinese Checkers	Yes	7,680,000

Monte-Carlo Methods

- *Entirely* different way of looking at game playing
  - Applicable to two player and multi-player games

- No game heuristics required!
- Driven by *random game playing*
  - Strong when no good heuristic is available
  - Big example in research is Go

- Very simple example:
  - Play 50 random games for each move
  - Pick one with highest winrate

Monte-Carlo Tree Search

- Simple example above
  - Works for easy games
  - Look-ahead is useful
- Apply random game playing to game tree search
- Navigate:
  - From root to unvisited node
  - Then play random game(s)
- Path guided by exploration/exploitation balance
- At end of time, pick most promising move
- Very powerful: Relatively new compared to Mini-Max

UCT Algorithm

- Dominant Monte-Carlo Tree Search technique (2015)
- Starting from root:
  - If there is an unvisited child, pick it
  - Otherwise, pick child that maximizes **UCTValue**

    UCTVal = winrate + sqrt(ln(parent.visits)/visits)
  - Repeat until an unvisited child is found
- Propagate winrate back up to root
- Repeat until time is up
- Pick move that has *highest winrate*

UCT Algorithm

```
function integer uct(node, depth):
    for time-steps do
            position = root
            while position is explored
                    val = −∞
                    for child of position
                            !– Unexplored node check here--!
                            val = max(val, UCTValue(child))
                            position = val.node
                    end for
            end while

            Play random game(s) at child
            while position is not root
                    update win-rate for player at node
                    position = position.parent
            end while

    end for
```

Multi-Player UCT Algorithm

- *Very easy* to extend
  - We do not have to maintain heuristic values
  - UCT handles N-player games in its base form

- For the player making the move
  - Simply record winrate at each node
  - Assume player will pick move most likely to lead to win

- No change from previous algorithm

More on UCT Value

- UCTValue has two parts
- Winrate is self-explanatory
  - Value between 0.0 and 1.0 indicating proportion of wins
- Second part: **sqrt(ln(parent.visits)/visits)**
- Specifics not important, but also between 0.0 and 1.0
- Goes *up* the less this child has been explored in relation to its parent
- Achieves exploration/exploitation balance!
- Sometimes constants usually added to tweak this

Applications of UCT

- Best performance available for Go
  – Top player is currently Zen
  – Defeated 9-dan player with three stone handicap

- Applied to wide range of games
  – Poker
  – Settlers of Catan
  – Magic: The Gathering