TDDC17

Seminar 3 Plus

Search III: Adverserial Search and Games
Ch 6

Patrick Doherty
Dept of Computer and Information Science
Artificial Intelligence and Integrated Computer Systems Division

Why Study Board Games?

Board games are one of the oldest branches of AI (Shannon and Turing 1950).

- Board games present a very abstract and pure form of competition between two opponents and clearly require a form of “intelligence”.
- The states of a game are easy to represent
- The possible actions of the players are well-defined
  - Realization of the game as a search problem
  - It is nonetheless a contingency problem, because the characteristics of the opponent are not known in advance

Challenges

Board games are not only difficult because they are contingency problems, but also because the search trees can become astronomically large.

Examples:

- **Chess**: On average 35 possible actions from every position, 100 possible moves/ply (50 each player): $35^{100} \approx 10^{150}$ nodes in the search tree (with "only" 10^{40} distinct chess positions (states)).
- **Go**: On average 200 possible actions with circa 300 moves: $200^{300} \approx 10^{700}$ nodes.

Good game programs have the properties that they

- delete irrelevant branches of the game tree,
- use good evaluation functions for in-between states, and
- look ahead as many moves as possible.

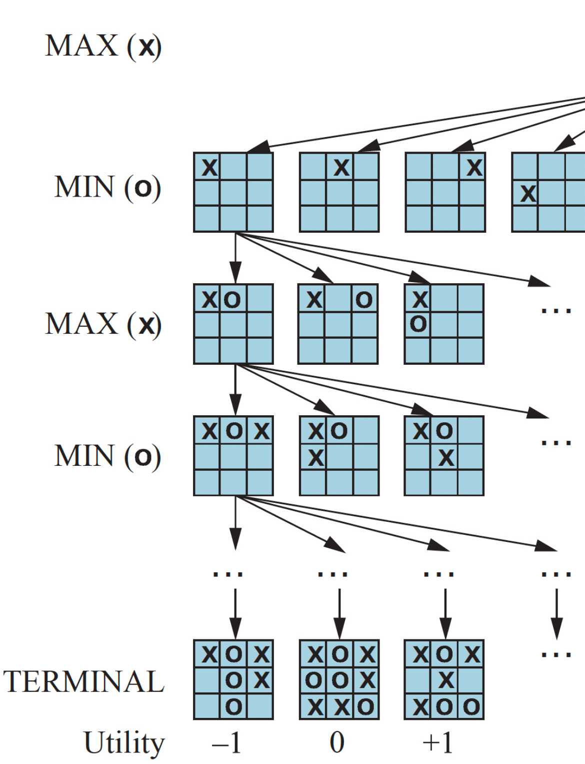
More generally: Adverserial Search

- Multi-Agent Environments
  - agents must consider the actions of other agents and how these agents affect or constrain their own actions.
  - environments can be cooperative or competitive.
  - One can view this interaction as a "game" and if the agents are competitive, their search strategies may be viewed as "adversarial".
- Most often studied: Two-agent, zero-sum games of perfect information
  - Each player has a complete and perfect model of the environment and of its own and other agents actions and effects
  - Each player moves until one wins and the other loses, or there is a draw.
  - The utility values at the end of the game are always equal and opposite, thus the name zero-sum.
  - Chess, checkers, Go, Backgammon (uncertainty)

Games as Search

- The Game
  - Two players: One called **MIN**, the other **MAX**. **MAX** moves first.
  - Each player takes an alternate turn until the game is over.
  - At the end of the game points are awarded to the winner, penalties to the loser.
- Formal Problem Definition:
  - Initial State: S_0 – Initial board position
  - TO-MOVE(s) - The player whose turn it is to move in state s
  - ACTION(s) - The set of legal moves in state s
  - RESULT(s,a) - The transition model: the state resulting from taking action a in state s.
  - IS-TERMINAL(s) - A terminal test. True when game is over.
  - UTILITY(s,p) – A utility function. Gives final numeric value to player p when the game ends in terminal state s.
    - For example, in Chess: win (1), lose (-1), draw (0):

(Partial) Game Tree for Tic-Tac-Toe



- $\approx 9! = 362{,}880$ terminal nodes
- $5{,}478$ distinct states

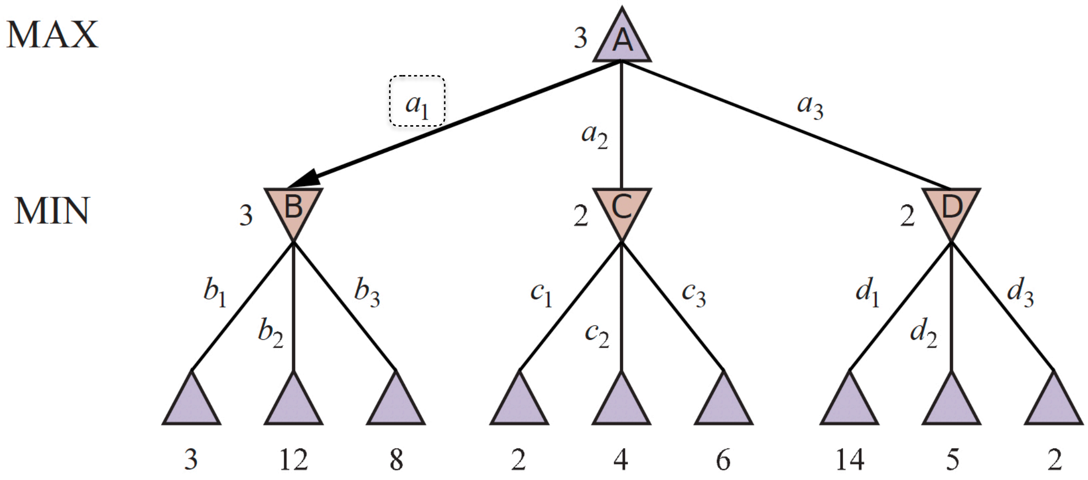
- Game trees can be infinite
- Often large! Chess has:
  - 10^{40} distinct states
  - average of 50 moves
  - average b-factor of 35
  - $35^{100} = 10^{154}$ nodes

Optimal Decisions in Games: Minimax Search

1. Generate the complete game tree using depth-first search.
2. Apply the utility function to each terminal state.
3. Beginning with the terminal states, determine the utility of the predecessor nodes (parent nodes) as follows:
    1. Node is a MIN-node
       Value is the minimum of the successor nodes
    2. Node is a MAX-node
       Value is the maximum of the successor nodes
4. From the initial state (root of the game tree), MAX chooses the move that leads to the highest value (minimax decision).

Note: Minimax assumes that MIN plays perfectly. Every weakness (i.e. every mistake MIN makes) can only improve the result for MAX.

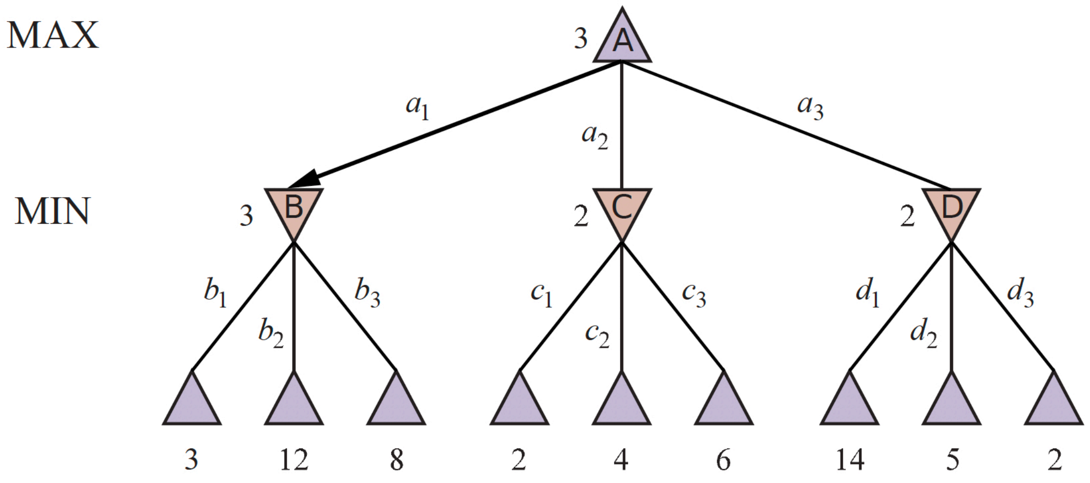
Minimax Tree



- Interpreted from MAX's perspective
- Assumption is that MIN plays optimally
- The minimax value of a node is the utility for MAX
- MAX prefers to move to a state of maximum value and MIN prefers minimum value

What move should MAX make from the Initial state?

MAX utility values



$$MINIMAX(s) =$$

$$\begin{cases} UTILITY(s, MAX) & \text{if } IS\text{-}TERMINAL(s) \\ max_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } TO\text{-}MOVE(s) = MAX \\ min_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } TO\text{-}MOVE(s) = MIN \end{cases}$$

Minimax Algortihm

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  $v \leftarrow -\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if $v2 > v$ then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  $v \leftarrow +\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if $v2 < v$ then
      v, move ← v2, a
  return v, move
```

Assume max depth of the tree is m and b legal moves at each point:

- Time complexity: $\mathbf{O}(b^m)$
- Space complexity:
  - Actions generated at same time: $\mathbf{O}(bm)$
  - Actions generated one at a time: $\mathbf{O}(m)$

Serves as a basis for mathematical analysis of games and development of approximations to the minimax algorithm

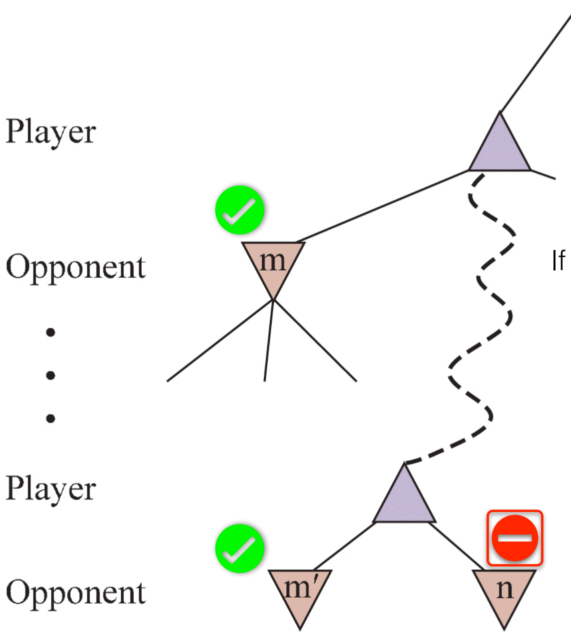
Recursive algorithm that proceeds all the way down to the leaves of the tree and then backs up the minimax values through the tree as the recursion unwinds

Alpha-Beta Pruning

- Minimax search examines a number of game states that is exponential in the number of moves (depth in the tree).
- Can be improved by using Alpha-Beta Pruning.
  - The same move is returned as minmax would
  - Can effectively cut the number of nodes visited in half (still exponential, but a great improvement).
  - Prunes branches that can not possibly influence the final decision.
  - Can be applied to infinite game trees using cutoffs.

The General idea

How do we determine when m, m' is a better choice than n?



If the player has a better choice at the same level m', or a better choice at any point higher up in the tree m, then n (and the subtree below) will never be chosen (searched)

Consider a node n somewhere in the tree Such that the player has a choice of moving to n

Alpha-Beta Values

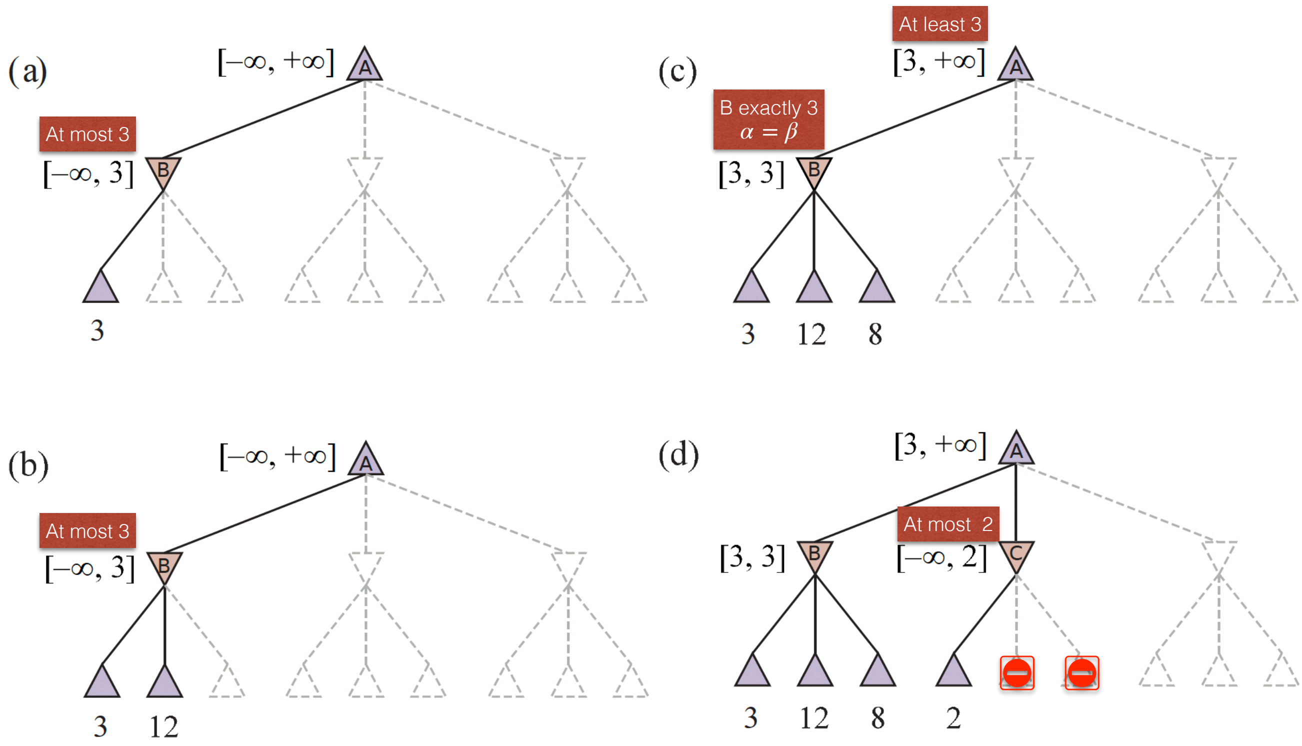
alpha – the value of the best (i.e., highest value) choice we have found so far at any choice point along the path for MAX. (actual value is at least alpha)....lower bound

beta - the value of the best (i.e., lowest value) choice we have found so far at any choice point along the path for MIN. (actual value is at most beta)...upper bound

Lower bound $[\alpha, \beta]$ Upper bound

Associate lower and upper bounds
on values of nodes in the search tree

Alpha-Beta Progress



At most 3

At most 3

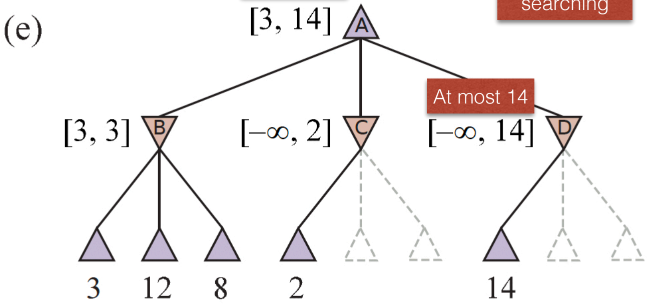
At least 3

B exactly 3
$\alpha = \beta$

At most 2

But B = 3, so MAX would never choose C
Because its value is at most 2 and could be worse
No need to search in the subtrees (terminal nodes)

Alpha-beta progress

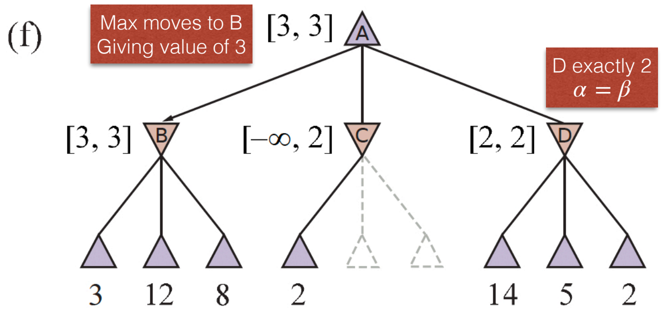


At most 14

14>3 so keep searching

At most 14

Minimax is a depth-first search, so we only need to think of nodes/values along single paths when recursing values upwards.



Max moves to B Giving value of 3

D exactly 2 $\alpha = \beta$

2nd successor is 5 5 > 3, so keep Searching

3rd successor is 2

Alpha-Beta Search

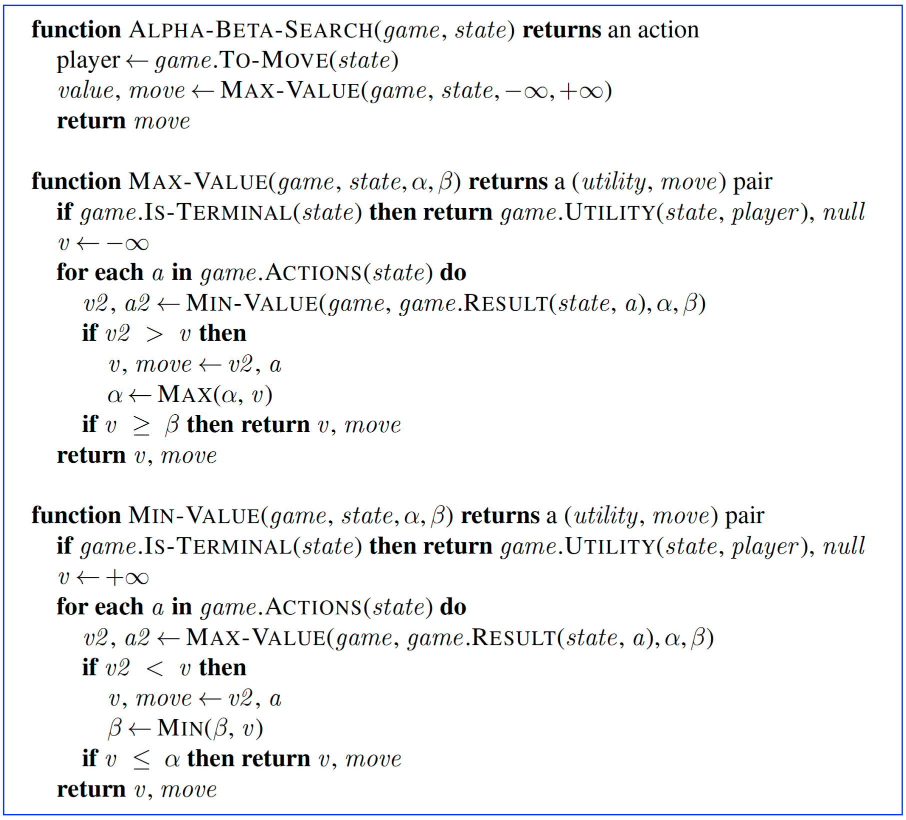
Returns a move for MAX

Similar to Minimax search. Functions are the same except Bounds are maintained on variables α and β

Effectiveness of α-β pruning is sensitive to to order in which states are examined.

With perfect move-ordering scheme, alpha-beta uses $\mathbf{O}(b^{m/2})$ nodes to pick a move rather than Minimax's $\mathbf{O}(b^{m})$ nodes. But perfect move-ordering is not possible. One can get close though.

Minimax with alpha-beta pruning is still not adequate for games like chess and Go due to the huge state spaces involved. Need something better!



```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, $-\infty$, $+\infty$)
  return move

function MAX-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  $v \leftarrow -\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 > v$ then
      v, move ← v2, a
      $\alpha \leftarrow$ MAX($\alpha$, v)
    if $v \geq \beta$ then return v, move
  return v, move

function MIN-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  $v \leftarrow +\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 < v$ then
      v, move ← v2, a
      $\beta \leftarrow$ MIN($\beta$, v)
    if $v \leq \alpha$ then return v, move
  return v, move
```

Heuristic Alpha-Beta Search

Intuition:
Due to limited computation time, cutoff the search early
and apply a heuristic evaluation function to states,
Effectively treating non-terminal nodes as if they were terminal

Recall MINIMAX(s)

$$MINIMAX(s) = \begin{cases} UTILITY(s, MAX) & \text{if } IS\text{-}TERMINAL(s) \\ max_{a \in Actions(s)} MINIMAX(RESULT(s, a)) & \text{if } TO\text{-}MOVE(s) = MAX \\ min_{a \in Actions(s)} MINIMAX(RESULT(s, a)) & \text{if } TO\text{-}MOVE(s) = MIN \end{cases}$$

$$H\text{-}MINIMAX(s, \underline{d}) = \begin{cases} \underline{EVAL(s, MAX)} & \text{if } \underline{IS\text{-}CUTOFF(s, d)} \\ max_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s, a), \underline{d+1}) & \text{if } TO\text{-}MOVE(s) = MAX \\ min_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s, a), \underline{d+1}) & \text{if } TO\text{-}MOVE(s) = MIN \end{cases}$$

Heuristic Alpha-Beta Search

$$H\text{-}MINIMAX(s, \underline{d}) = \begin{cases} \underline{EVAL(s, MAX)} & \text{if } \underline{IS\text{-}CUTOFF(s, d)} \\ max_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s, a), \underline{d+1}) & \text{if } TO\text{-}MOVE(s) = MAX \\ min_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s, a), \underline{d+1}) & \text{if } TO\text{-}MOVE(s) = MIN \end{cases}$$

- Replace the $UTILITY(s, p)$ fn with an $EVAL(s, p)$ fn which estimates the expected utility of state s to player p.
- Replace the $IS\text{-}TERMINAL(s)$ test with an $IS\text{-}CUTOFF(s, d)$ test which must return true for terminal states, but is otherwise free to decide when to cut off the search, possibly using search depth so far or any other state properties deemed useful.

Example (Chess):

$$EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

where each f_i represents the material value of a chess piece (bishop = 3, queen=9) and the weights w_i represent how important a feature is in a state. Weights should be normalised so their sum is between range of: loss(0) to a win(+1)

Modify Alpha-Beta Search

Add bookkeeping so current depth is incremented on each recursive call

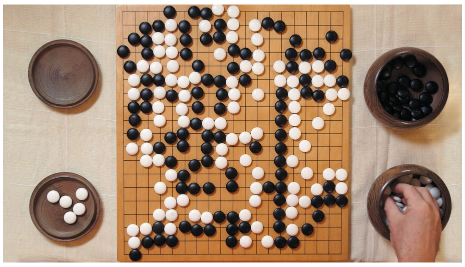
```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, $-\infty$, $+\infty$)
  return move

function MAX-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
  if game . IS-CUTOFF(state, depth) then return game . EVAL(state, player), null
  $v \leftarrow -\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 > v$ then
      v, move ← v2, a
      $\alpha \leftarrow$ MAX($\alpha$, v)
    if $v \geq \beta$ then return v, move
  return v, move

function MIN-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
  if game . IS-CUTOFF(state, depth) then return game . EVAL(state, player), null
  $v \leftarrow +\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 < v$ then
      v, move ← v2, a
      $\beta \leftarrow$ MIN($\beta$, v)
    if $v \leq \alpha$ then return v, move
  return v, move
```

The Game of GO

nature
At last – a computer program that can beat a champion Go player PAGE 484
ALL SYSTEMS GO



- Two major weaknesses of Alpha-Beta Search:
  - GO has a branching factor starting at 361
    - limiting alpha-beta search to 4-5 ply (ply is a half move taken by 1 player)
  - Difficult to figure out a good evaluation function for GO
    - Material value not a strong indicator and most positions in flux until the end of the game

Modern GO programs instead use:

Monte Carlo Search (MCTS)

+ Lots of other techniques!

MCTS Strategy

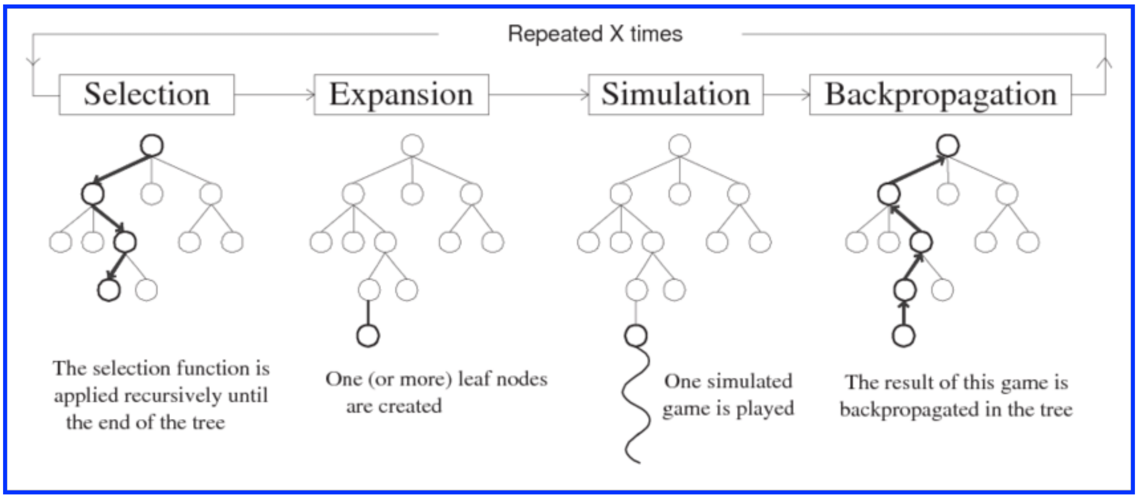
- MCTS does not use a heuristic evaluation function:
  - The value of a state is estimated as the average utility over a number of simulations of complete games starting from the state.
    - Average utility could be win percentage for example
- Simulations (also called playouts or rollouts)
  - Chooses moves first for one player and then the other until a terminal node is reached.
    - Simple policy: choose randomly
- How do we choose moves during playouts??
  - MCTS uses playout policies which are mappings between states and actions
    - Playout policies bias moves toward good ones
    - For GO and other games, playout policies can be learned from self-play using Neural Networks (Deep Learning)

MCTS Strategy

- Given a playout policy:
  - From what positions do we start the playouts?
  - How many playouts do we allocate to each position?
- Pure Monte Carlo search:
  - Do *N* simulations starting from each child in the current state of the game (determine quality of direct children (without a selection policy) and then select a move, repeat, until time runs out)
  - Focus is symmetric
  - For most games this is not adequate.
- Selection Policy selectively focuses computational resources on important parts of the game tree
  - Builds an asymmetric tree (capitalises on rich parts of search area)
  - Balances:
    - Exploitation (states that have done well in past playouts)
    - Exploration (states that have had few playouts)
  - One popular and effective selection policy is UCT (upper confidence bounds applied to trees)

4 Steps in MCTS

MCTS maintains a search tree and grows it on each iteration using the following steps:


Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until the end of the tree
One (or more) leaf nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Starting at the root of the search tree, choose a move using the selection policy, repeating the process until a leaf node is reached

Grow the search tree by generating a new child/children.

Perform a playout from a child using the playout policy. These moves are not recorded in the search tree

Use the simulation result to update the utilities of the nodes going back up to the root.

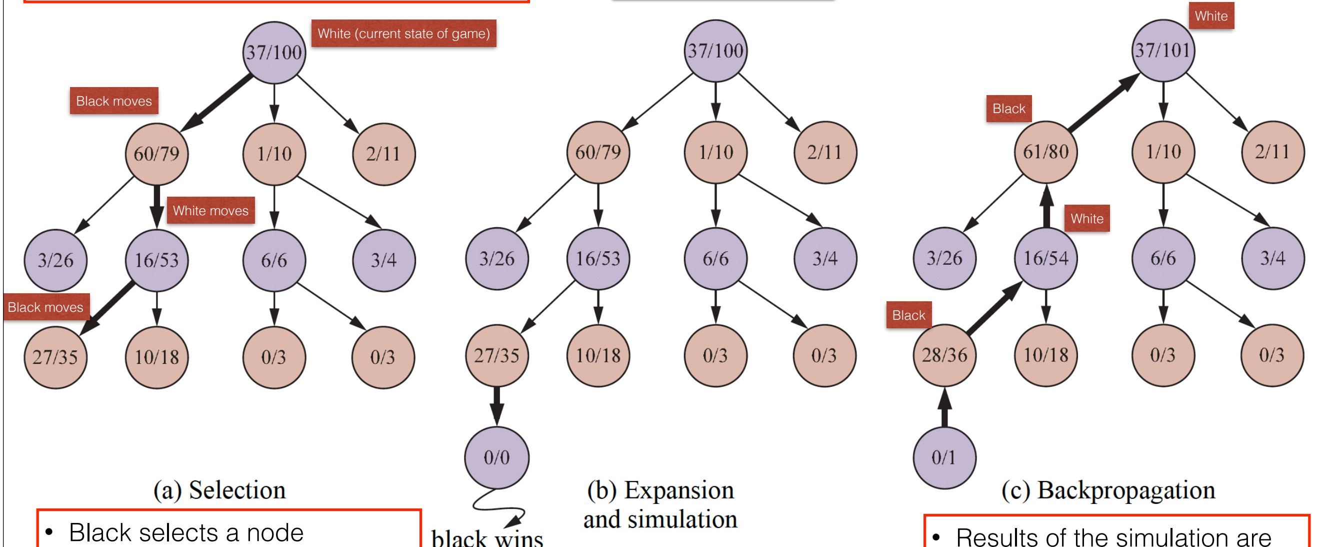
LINKÖPING UNIVERSITY

After X times: Choose the best move from start state

One Iteration of MCTS

- White has previously moved.
- What should blacks move be (2nd level)?
- White has won 37 out of 100 playouts (37/100) done so far
- Suppose we will do 1000 iterations. What does the 101th iteration look like?

# of wins/# of playouts



(a) Selection

(b) Expansion and simulation

(c) Backpropagation

- Black selects a node where it has won 60 of 79 playouts (60/79)
- Uses UCT selection metric
- Selection continues to a leaf node where black has won 27 out of 35 playouts (27/35)

- Generate a new child node labeled 0/0
- Execute a playout
- Black wins this simulation

- Results of the simulation are back propagated up the tree branch.
- Black won, so black nodes are incremented in # of wins/# of playouts
- White loses, white nodes are incremented in number of playouts only.

LINKÖPING UNIVERSITY

UCT: A Selection Policy

UCT: upper confidence bound applied to trees

Ranks each possible move based on an upper confidence bound formula called UCB1:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{ln\ N(Parent(n))}{N(n)}}$$

- $U(n)$: Total utility of all playouts that go through n
- $N(n)$: The number of playouts through node n
- $Parent(n)$: The parent node of node n
- $\frac{U(n)}{N(n)}$-term: is the exploitation term. The average utility of n. For example win percentage.
- $\sqrt{}$ - term : is the exploration term.
  - Numerator: ln of the number of times we have explored the parent
    - If n is selected some non-zero % of the time, the exploration term goes to zero as the counts increase, and eventually the playouts are given to the node with the highest average utility.
  - Denominator: count $N(n)$
    - The exploration term will be high for nodes only explored a few times
- C: Constant that balances exploitation and exploration.
  - Theoretically, $\sqrt{2}$ is best value for C, but this constant is often learned or chosen through trial and error.
  - $C = 1.4$ would choose the $60/79$ (more exploitation) node in the example during *Selection*, while $C = 1.5$ would choose the $2/11$ node (more exploration) during Selection.

LINKÖPING UNIVERSITY

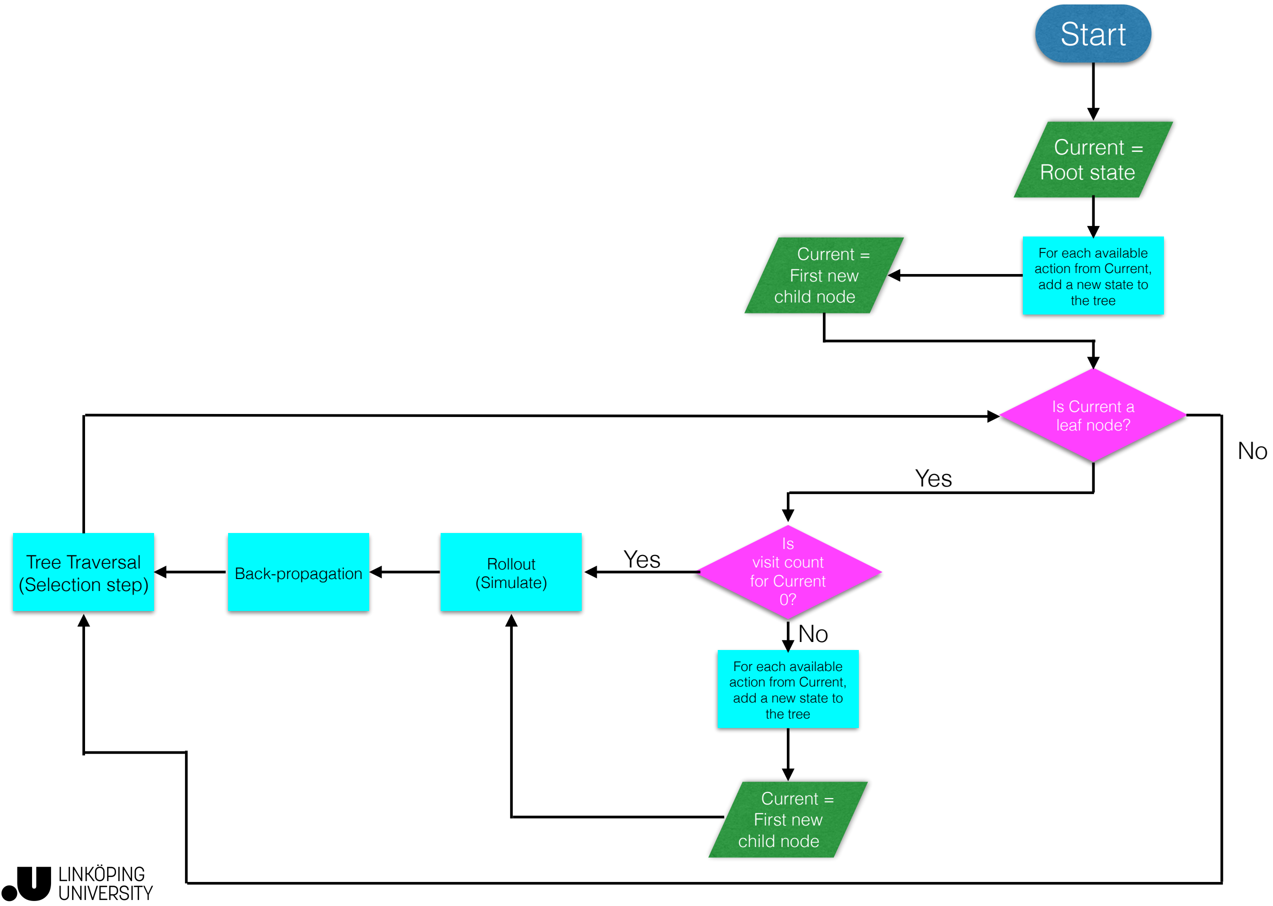
MCTS Algorithm

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
    tree ← NODE(state)
    while IS-TIME-REMAINING() do
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts
```

- When iterations terminate, the node with the highest number of playouts (less uncertainty) is returned rather than highest average utility.
  - The UCT/UCB1 selection strategy ensures that the node with the most playouts is almost always the node with the highest win percentage
- The time to complete a playout is linear in the depth of the game tree, so there is time for multiple playouts
  - Example: game with branching factor of 32, where average game is 100 ply:
    - Suppose we have computational power to consider a billion states before moving
      - Minimax can search 6 ply deep
      - Alpha-Beta Pruning can search 12 ply deep with perfect move ordering
      - Monte Carlo search can do 10 million playouts

LINKÖPING UNIVERSITY

MCTS Algorithm Schematic



Start

Current = Root state

For each available action from Current, add a new state to the tree

Current = First new child node

Is Current a leaf node?

No

Yes

Is visit count for Current 0?

Yes

No

For each available action from Current, add a new state to the tree

Current = First new child node

Rollout (Simulate)

Back-propagation

Tree Traversal (Selection step)

Some Observations

ALPHAGO [2016] put four ideas together:

- Visual pattern recognition
- Reinforcement learning
- Neural networks
- Monte Carlo search

Defeated:

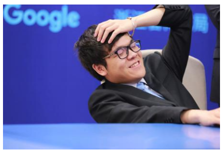
- Lee Sedol (by a score of 4-1 in 2015)
- Kie Jie (by a score of 3-0 in 2016)

Image: Lee Jin-man/AP

Lee Sedol

Lee Sedol retired from Go lamenting:

"Even if I became number 1, there is an entity that can not be defeated"



Kie Jie

"After humanity spent thousands of years improving our tactics, computers tell us that humans are completely wrong. I would go as far as to say not a single human has touched the edge of the truth of Go."

2018: ALPHAZERO surpassed ALPHAGO at Go!!

- Also defeated top programs in chess and Shogi
- Learns through self-play without human expert knowledge and without access to past games
- Uses reinforcement and deep learning

Timeline

- 1952 - Computer masters Tic-Tac-Toe
- 1994 - Computer masters Checkers
- 1997 - IBM’s Deep Blue defeats Garry Kasparov in Chess
- 2011 - IBM’s WATSON defeats human Jeopardy champions
- 2014 - Google’s algorithms learn to play Atari Games
- 2015 - Wikipedia - “ Thus it is very unlikely that it will be possible to program a reasonably fast algorithm for playing the Go endgame flawlessly, let alone the whole Go game”.
- 2015 - Google’s AlphaGo defeats Fan Hui (2-dan player) in Go
- 2016 - Google’s AlphGo defeats Lee Sedol 4-1 (9-dan player) in Go
- 2017 - Google’sAlphaZero defeats STOCKFISH (2017 TCEC computer chess champion)
- 2018 - Google’s AlphaZero surpasses AlphGo at Go (no human expertise, just self play)
- 2019 - Deep Mind’s ALPHASTAR program ranks in top 0.02% of officially ranked human players for StarCraft