FIT3080 – Artificial Intelligence

Adversarial Search Algorithms

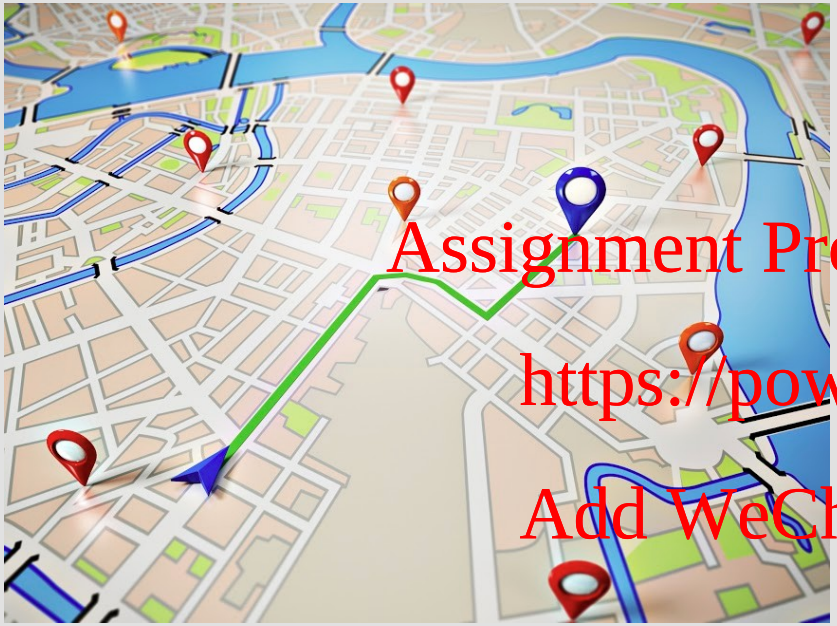
Announcements

- Assignment 1 is out!
  - Modelling and solving problems using search
  - 4 parts (mix of conceptual and practical tasks)
  - Check Moodle for the specification
  - Due date: **September 7, 11:55pm (Melbourne)**

Last week

- Informed tree/graph search
  - Expand ‘most promising’ nodes first,
  - Using evaluation function: f(n) = g(n) + h(n)
- Heuristics
  - What are they?
  - Properties & guarantees
  - Effectiveness
- Bounded suboptimal search

Last week

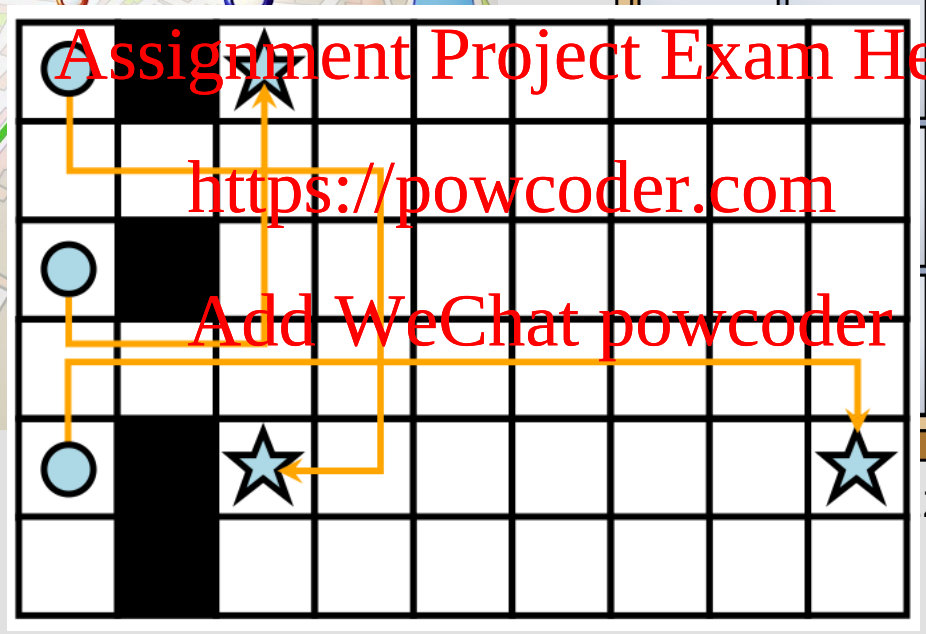


Route planning

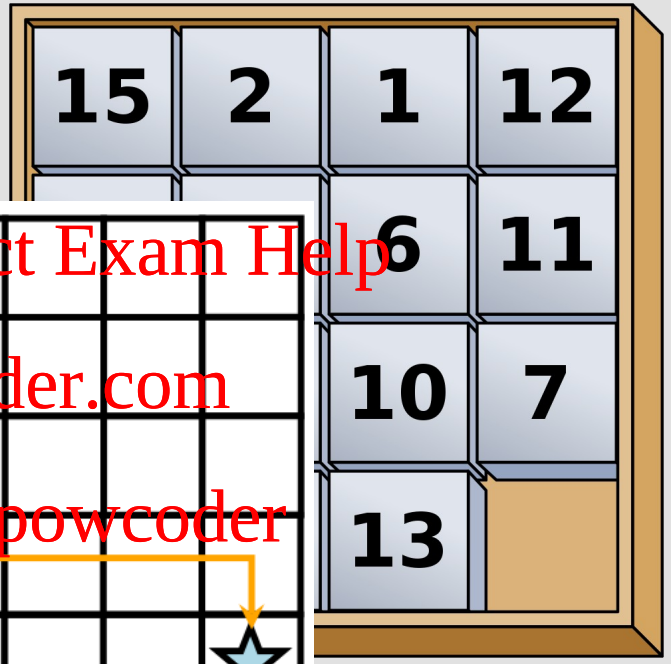
Puzzles

Last week

Route



Coordination problems



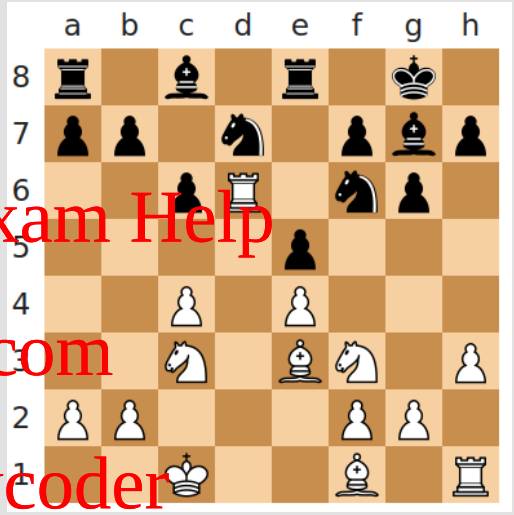
zles

This week: games!

Tic-Tac-Toe

This week: games!

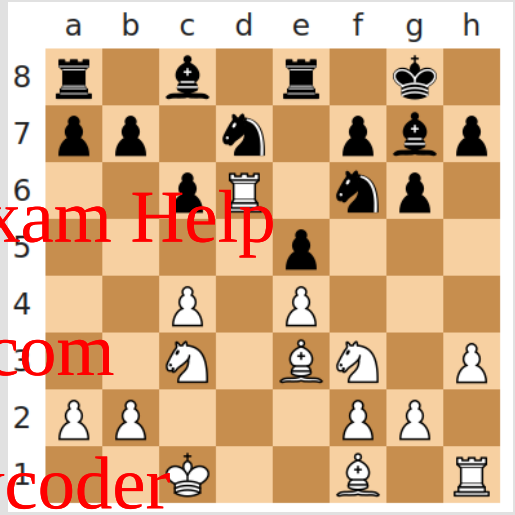
Tic-Tac-Toe



Chess

This week: games!

Tic-Tac-Toe

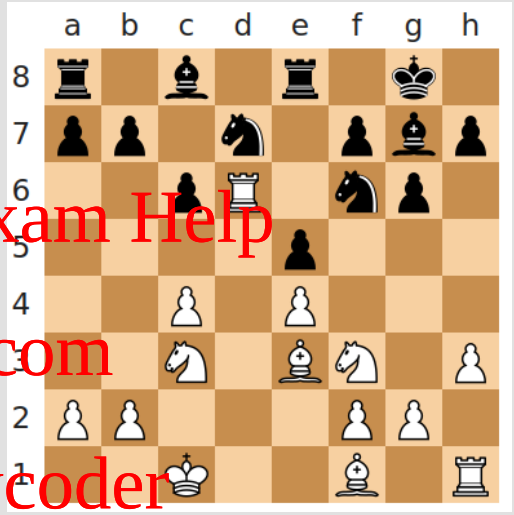


Chess

Games with an adversary

This week: games!

Tic-Tac-Toe



Chess

Games with an adversary

Man vs. Machine!

This week: games!

Chinook vs. Tinsley (1993/4)

Watch Jonathan Schaffer wonderful account:
https://www.youtube.com/watch?v=VWqtNS9pmOI

This week: games!

Deep Blue vs. Gary Kasparov (1996/7)

A short recap from the BBC:
https://www.youtube.com/watch?v=KF6sLCeBj0s

This week: games!

Deep Blue vs. Gary Kasparov (1996/7)

A short recap from the BBC:
https://www.youtube.com/watch?v=KF6sLCeBj0s

This week: games!

AlphaGo vs. Lee Sedol (2016)

There's a wonderful documentary about this match:
https://www.youtube.com/watch?v=WXuK6gekU1Y

Types of games

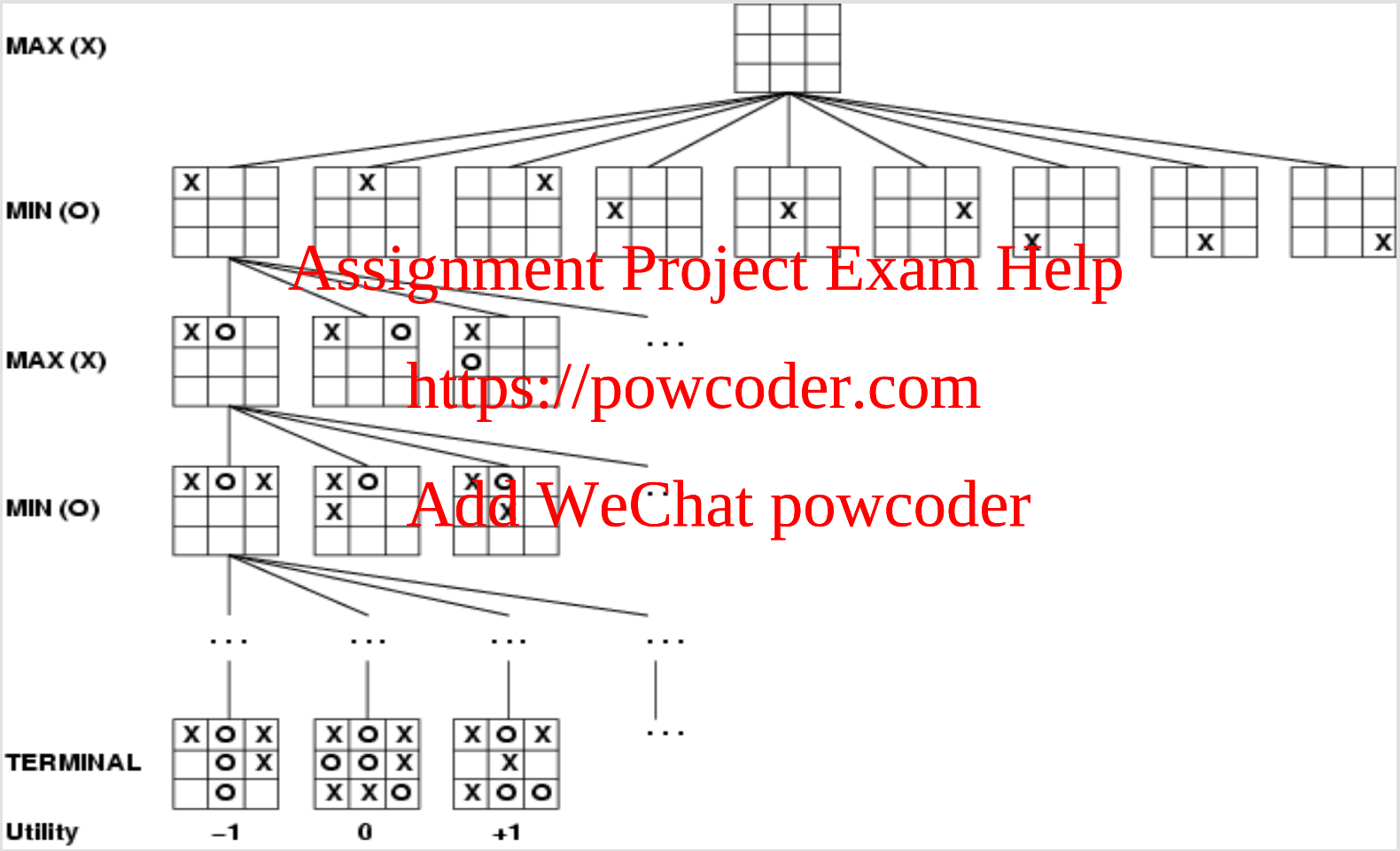
- Perfect information (deterministic, full obsv.)
  - Chess, Go, noughts-and-crosses (tic-tac-toe), draughts (checkers), etc.
- Imperfect information (stochastic, partially obsv.)
  - Poker, Texas hold'em (variant of poker), bridge, backgammon, etc
- n-Players (e.g. n = 2 for chess)
- Sequential (vs. simultaneous)
- Zero-sum (vs non-zero sum games)

Our focus: 2p, sequential, perfect information, zero sum

Game Trees

- Start state: initial configuration of the board
- Players are MAX and MIN
  - A position favourable to MAX has a value > 0 (winning often ∞)
  - A position favourable to MIN has a value < 0 (winning often −∞)
- Each player chooses the most promising move **for them**
- Each move adds new nodes to the frontier
- The set of nodes generated for one player is called a **ply**
- Each node has an associated **utility** for each player
- Leaves of the tree are called **terminal nodes**

Game Tree (2-player, Deterministic, Turns)



Solving Game Trees

A solution is a game-tree where we can label the root node with an outcome (win, loss, draw, or a utility value)

- Each move available to the opponent is a subproblem (another game)
- All subproblems need to be solved before the parent node can be considered solved

A strategy for MAX (resp. MIN) is a solution-tree which contains exactly one successor for each MAX (resp. MIN) node.

A **winning strategy** for MAX (resp. MIN) is a solution-tree where every terminal is a WIN (resp. LOSS, resp. max utility value)

Solving Game Trees

Goal: find a winning strategy for MAX

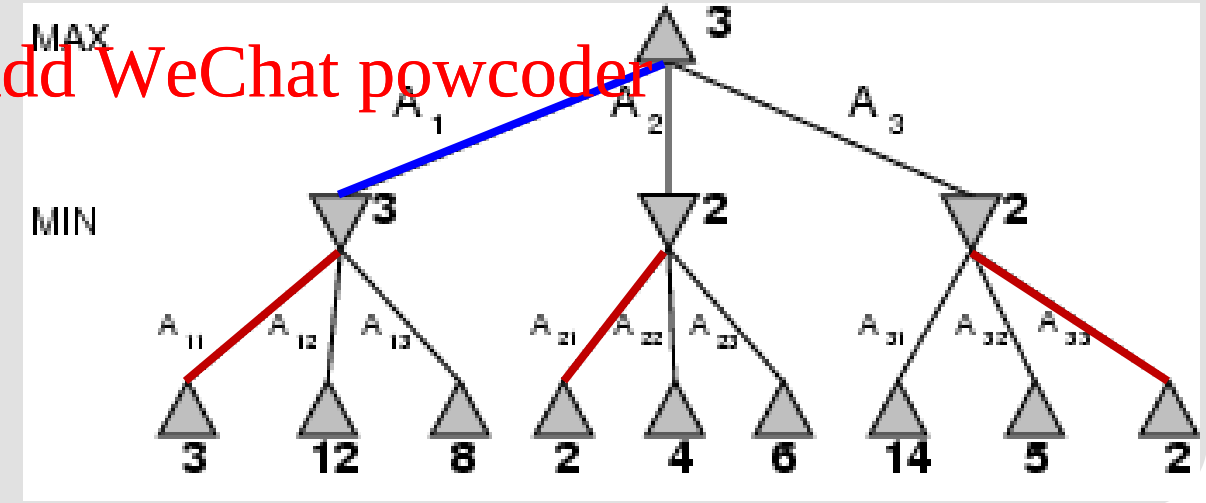
- For all nodes representing a game situation where it is MIN's move next, show that MAX can win from **every** position to which MIN might move
- For all nodes representing a game situation where it is MAX's move next, show that MAX can win from **just one** position to which MAX might move

Games versus Search Problems

- **Unpredictable opponent:**
  - **Must specify a move for every possible opponent reply**
- **Time limits:**
  - **not all games can be searched to the end**
  - **find a good first move**

Search ideas

- If MAX were to choose among tip nodes (or available nodes), she would take the node with the largest value
- If MIN were to choose among tip nodes (or available nodes), she would take the node with the smallest value
- Choose a move to the position with the highest ***minimax value***: best achievable payoff against best play
- E.g., 2-ply game:
-



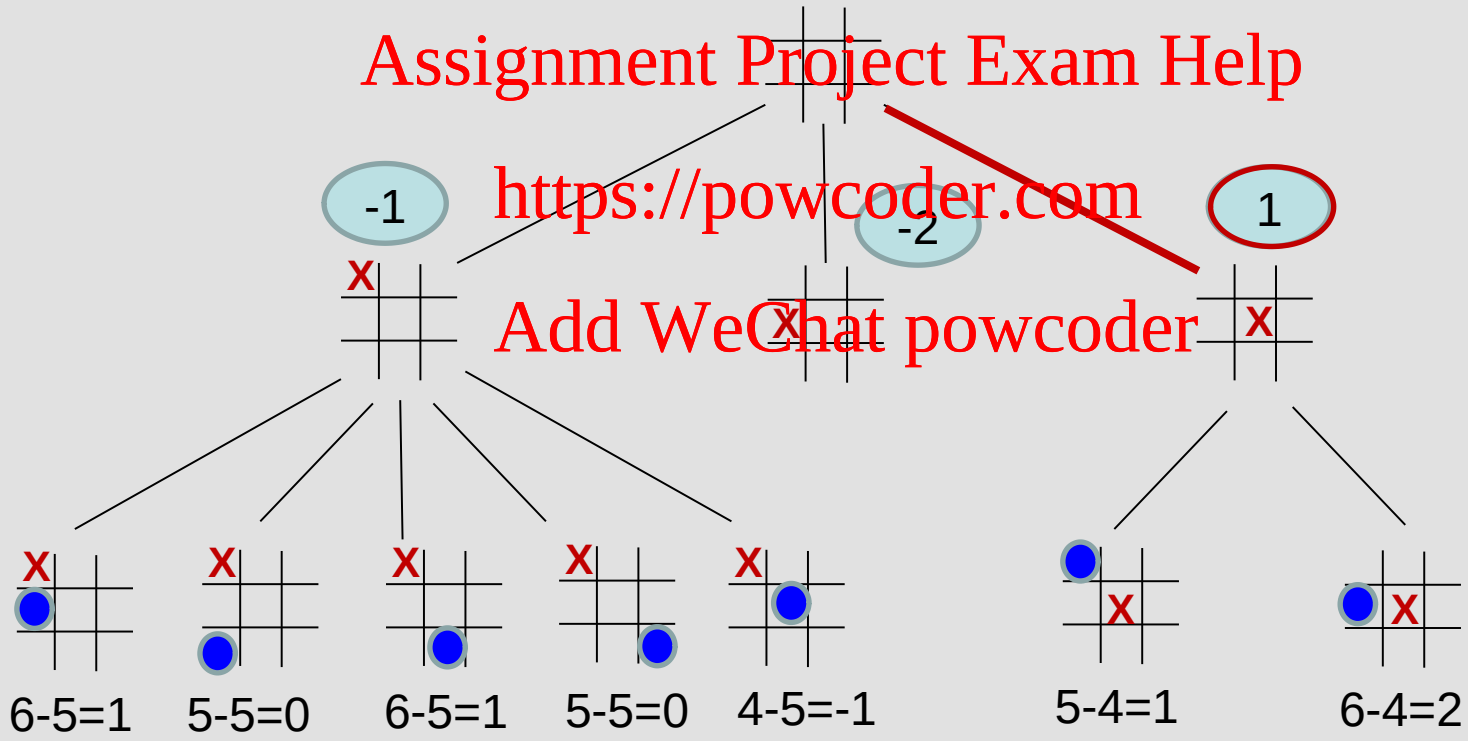
Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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

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

Minimax Example: Tic-Tac-Toe

Evaluation function:

{ # rows, columns, diagonals available to MAX –
  # of rows, columns, diagonals available to MIN }



Properties of Minimax

Complete Depth First Exploration:

all paths to depth *m*

Complete? **Yes (if tree is finite)**

Optimal? **Yes (against an optimal opponent)**

Time complexity? $O(b^m)$

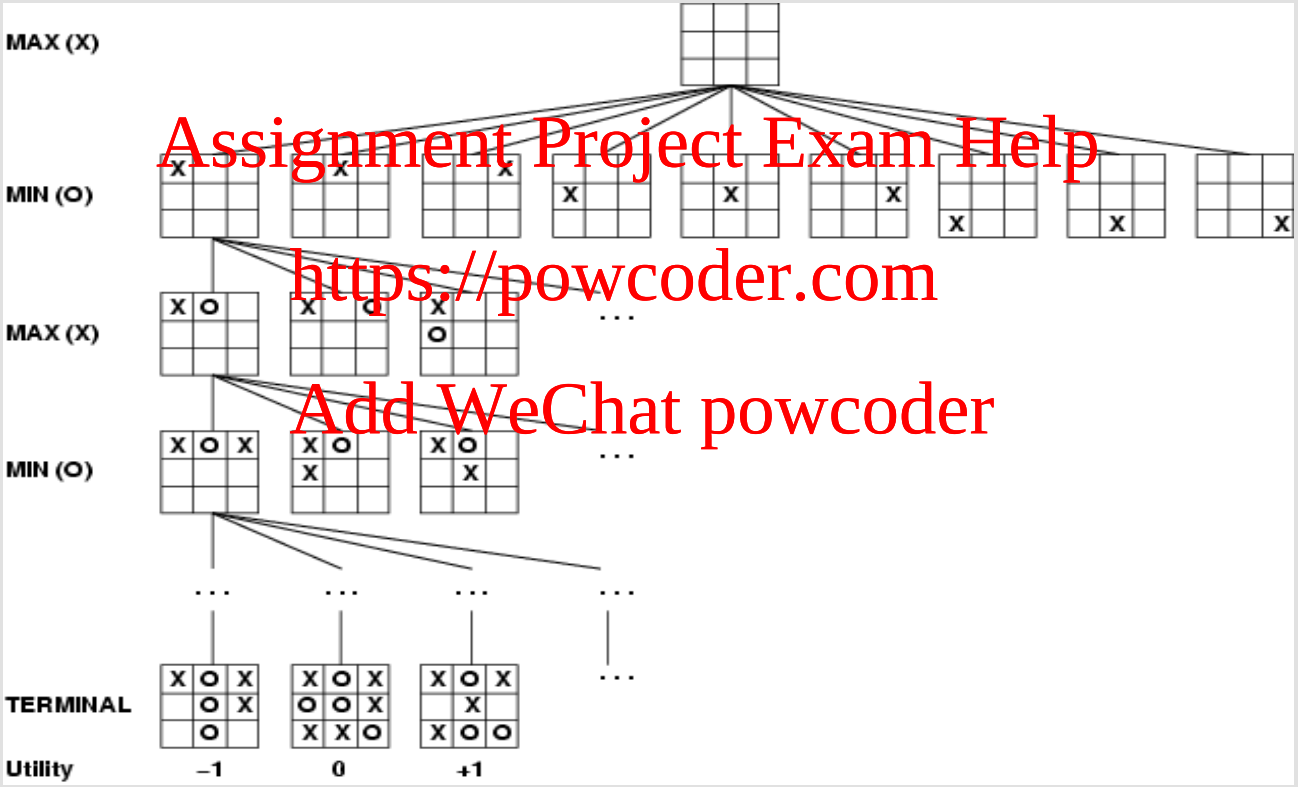
Space complexity? $O(bm)$ **(depth-first exploration)**

For chess, b ≈ 35, m ≈100 for "reasonable" games

→ exact solution completely infeasible

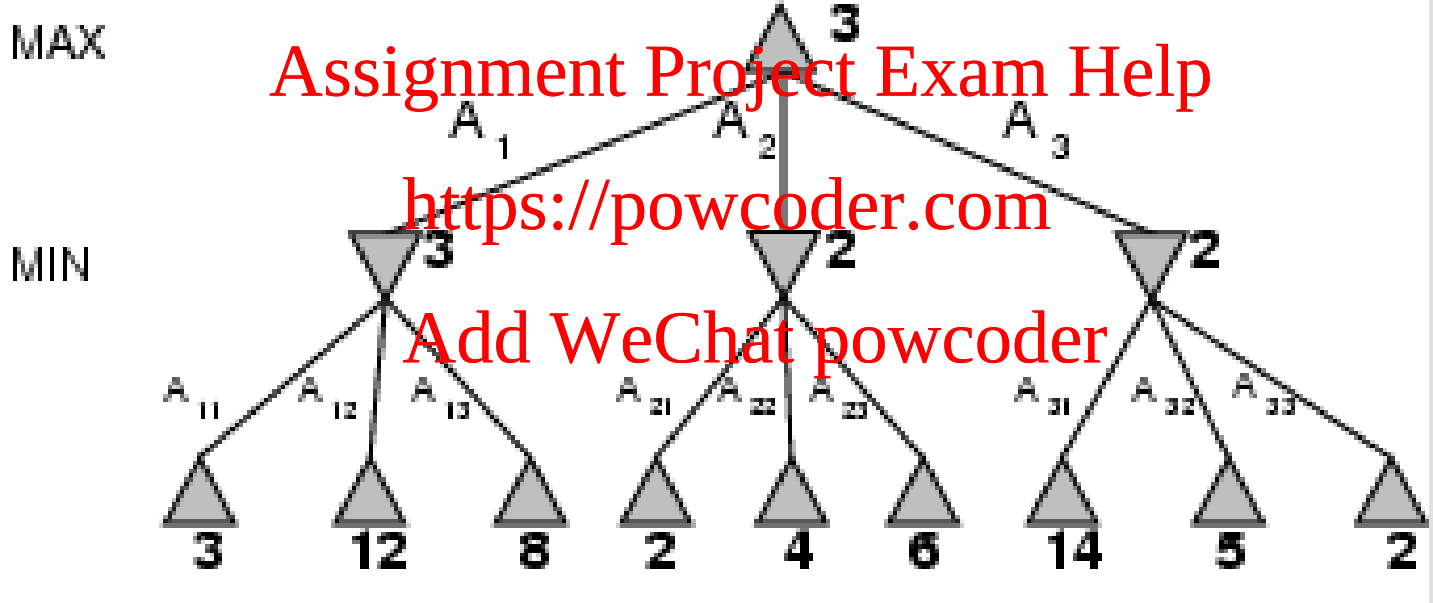
Pruning the Search Space

We can improve performance by eliminating symmetric moves



Pruning the Search Space

We can improve performance by eliminating redundant nodes



Definitions: α and β Values

- **α-value of a MAX node – current largest final backed-up value of its successors**
  - α-value is the lower bound for a MAX backed-up value
  -
- **β-value of a MIN node – current smallest final backed-up value of its successors**
  - β-value is the upper bound for a MIN backed-up value

α-β Procedure

- **Rules for discontinuing the search:**
  - **α cut-off**: search can be discontinued below any **MIN** node having a β-value ≤ **α-value** of **any** of its MAX node ancestors
    - The final backed-up value of this MIN node is set to its β-value
  - **β cut-off**: search can be discontinued below any **MAX** node having an α-value ≥ **β-value** of **any** of its MIN node ancestors
    - The final backed-up value of this MAX node is set to its α-value

Termination Condition

- All the successors of the start node are given final backed-up values
- The best first move is that which creates the successor with the highest backed-up value

Minimax Algorithm (again)

function MINIMAX-DECISION($state$) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT($state, a$))

---

function MAX-VALUE($state$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE($state$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow \infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

The α-β Algorithm (I)

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

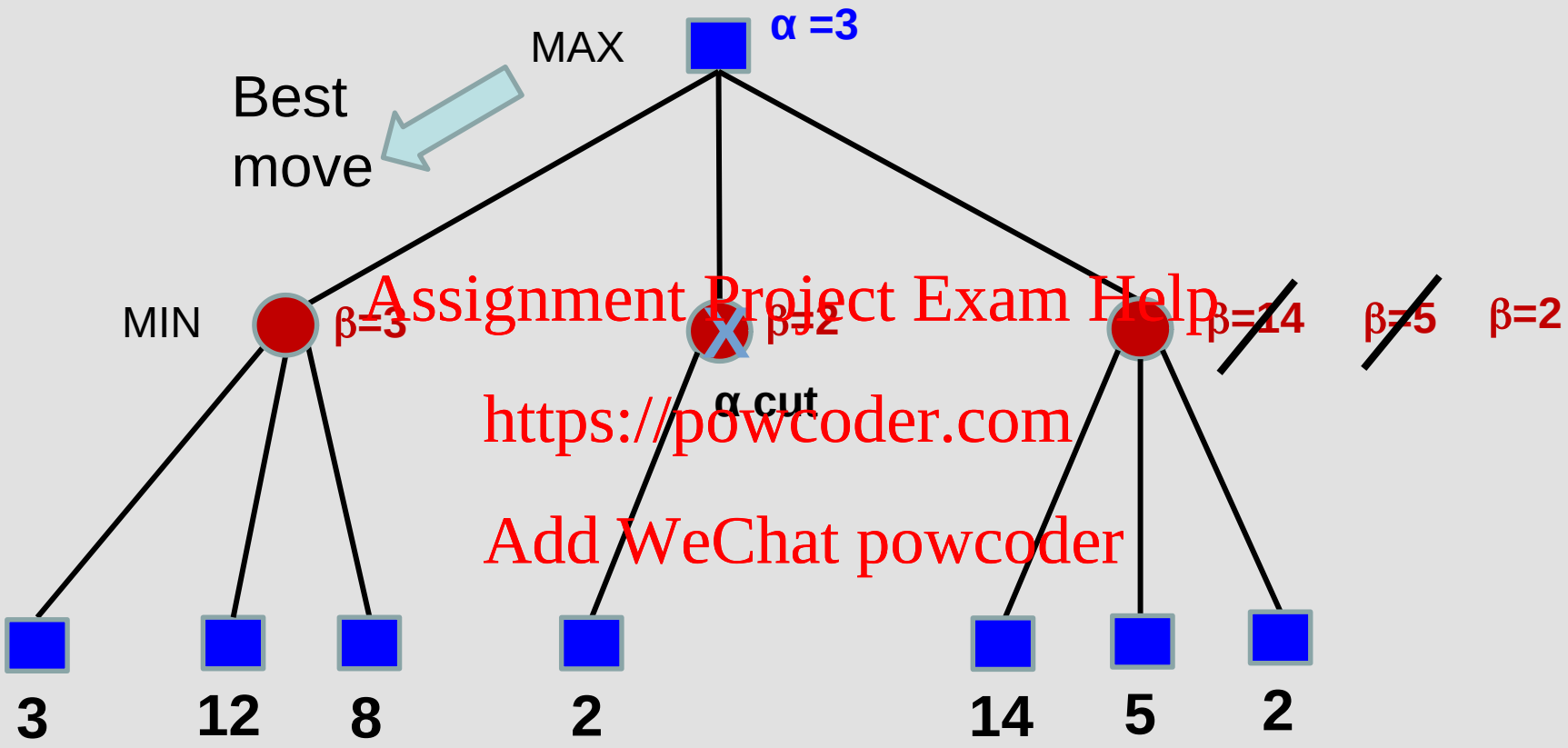
β cut-off

α cut-off

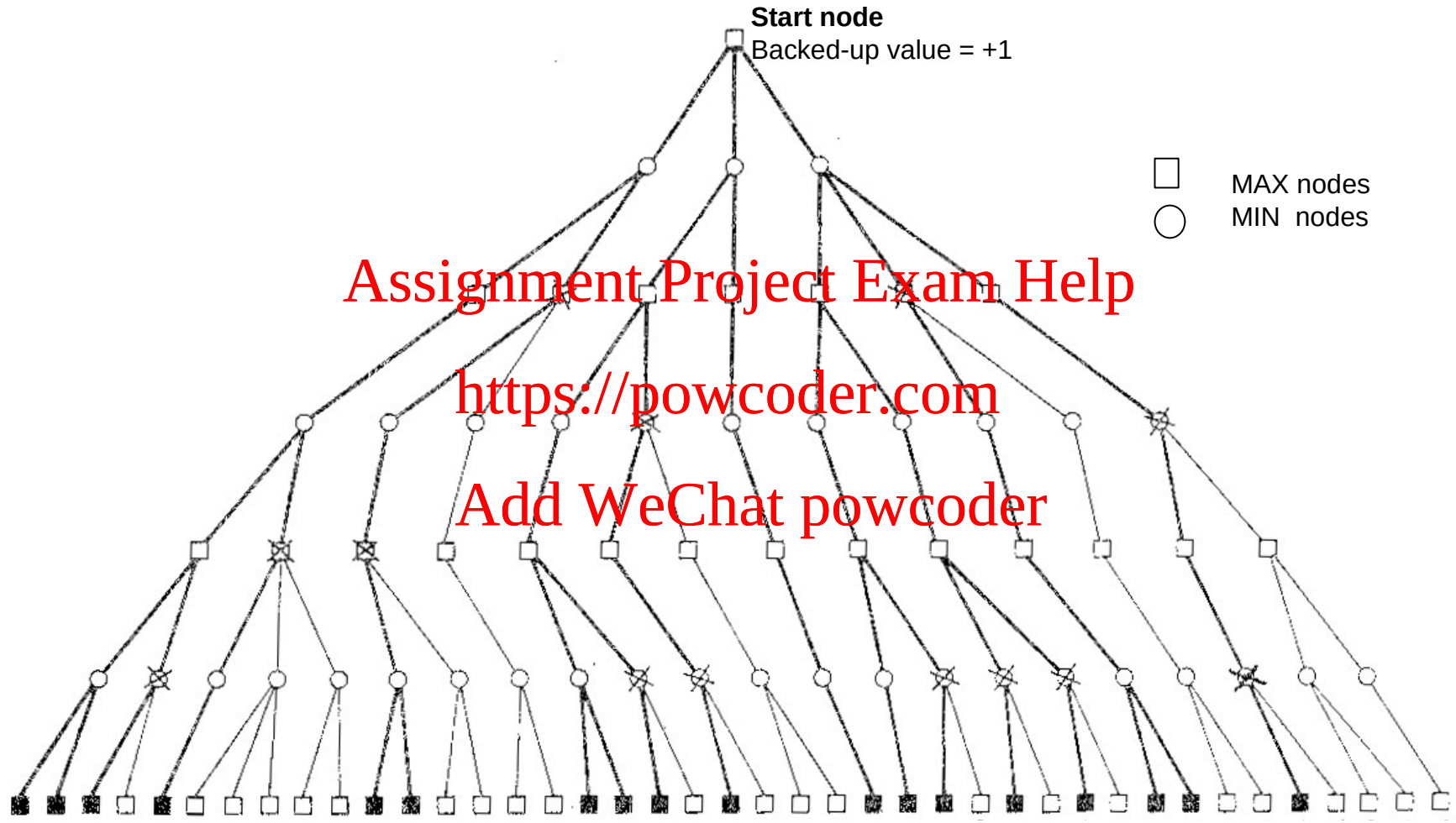
α = the value of the best (i.e., **highest-value**) choice we have found so far at any choice point along the path for MAX.

β = the value of the best (i.e., **lowest-value**) choice we have found so far at any choice point along the path for MIN.

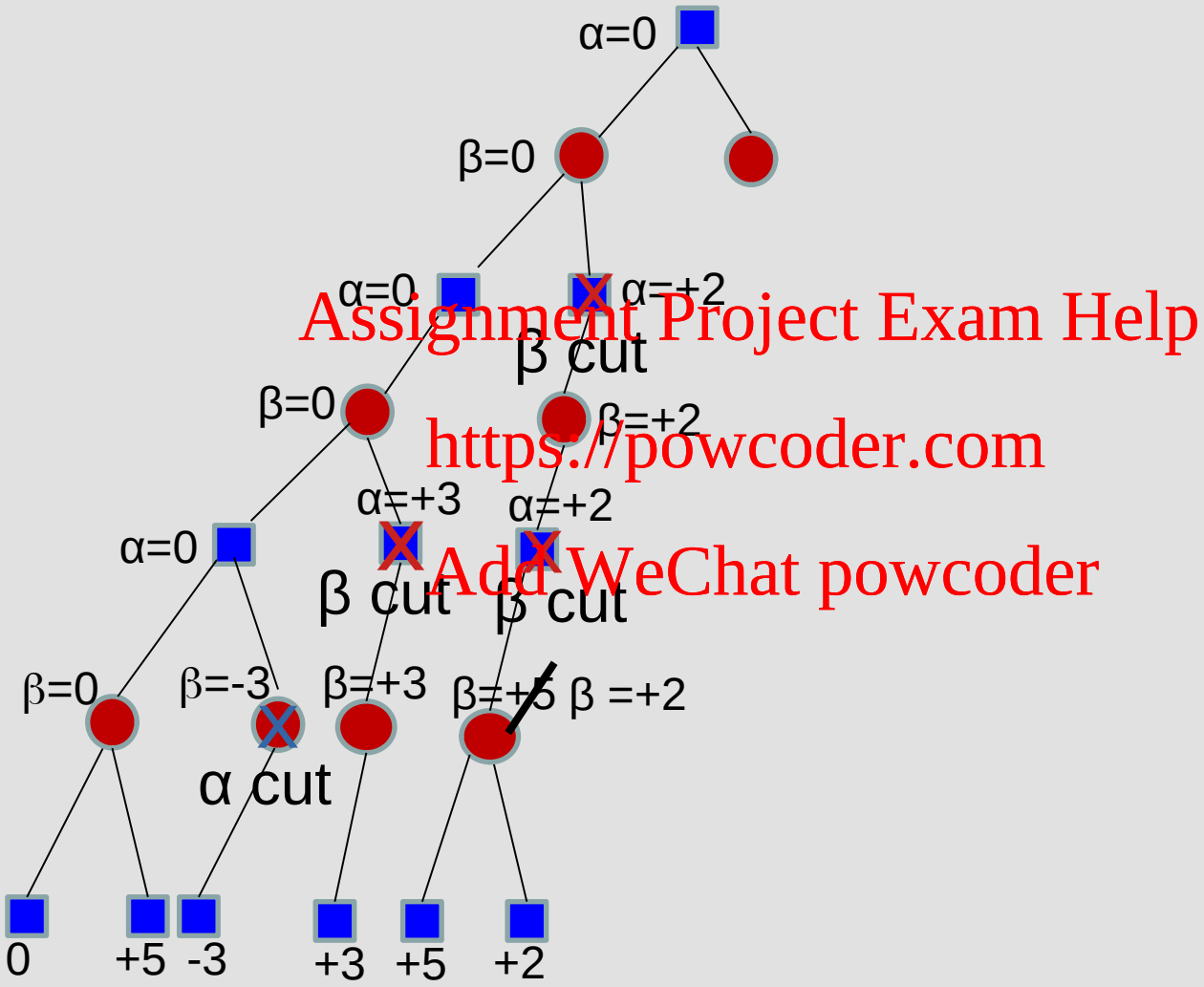
α-β Pruning – Example



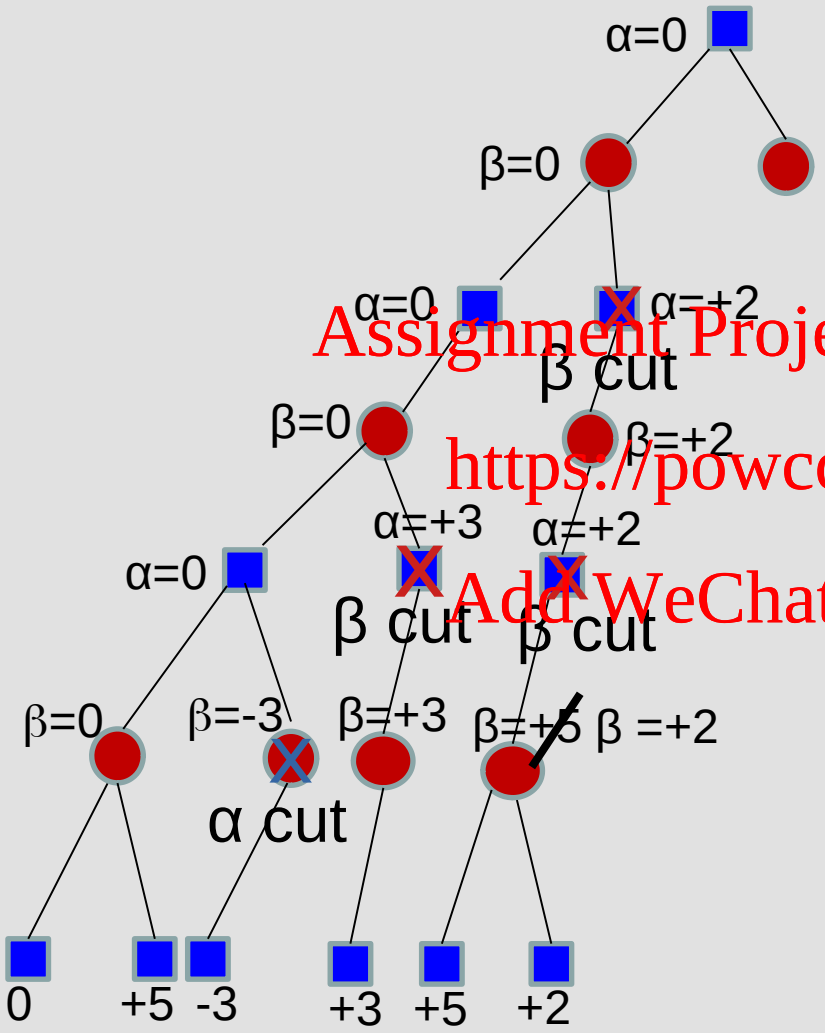
α-β Pruning – Large Example



α-β Pruning – Part of Large Example



α-β Pruning – Part of Large Example



Does the order of the moves matter?

Move Ordering

The effectiveness of the αβ algorithm depends on the order in which states are examined

- **With perfect ordering:**
  - **time complexity = $O(\text{sqrt}(b)^d) = O(b^{d/2})$**
- i.e., depth of search can be doubled

Adding dynamic ordering schemes brings us close to the theoretical limit

Heuristics for Games

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

- Still might not reach a terminal node!

Idea: treat the frontier nodes as terminal states

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

Evaluation Functions

Some ideas for generating these.

From experience:

- **Material values** in chess (1 = pawn, 3 = bishop, … 9 = Queen)
- **State features**:
- $f_1(s) = (\#\text{ of white queens}) - (\#\text{ of black queens})$
- **Combinations of features** (e.g., linear weighted sum)

$$\text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s),$$

From preprocessing:

- End-game databases

How do we search?

Depth limited search (fixed cost)

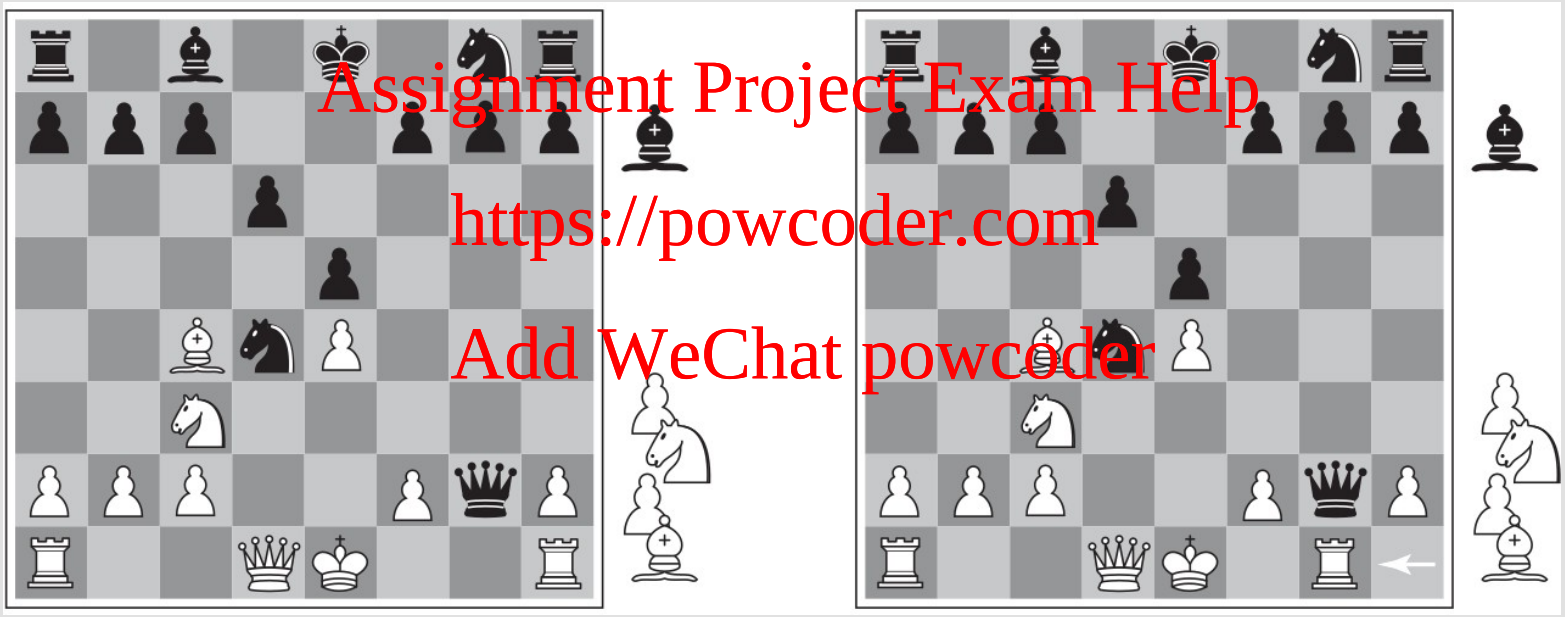
Depth-First Iterative Deepening (anytime)

Beam search (forward pruning)

Don't search at all! (use table lookups)

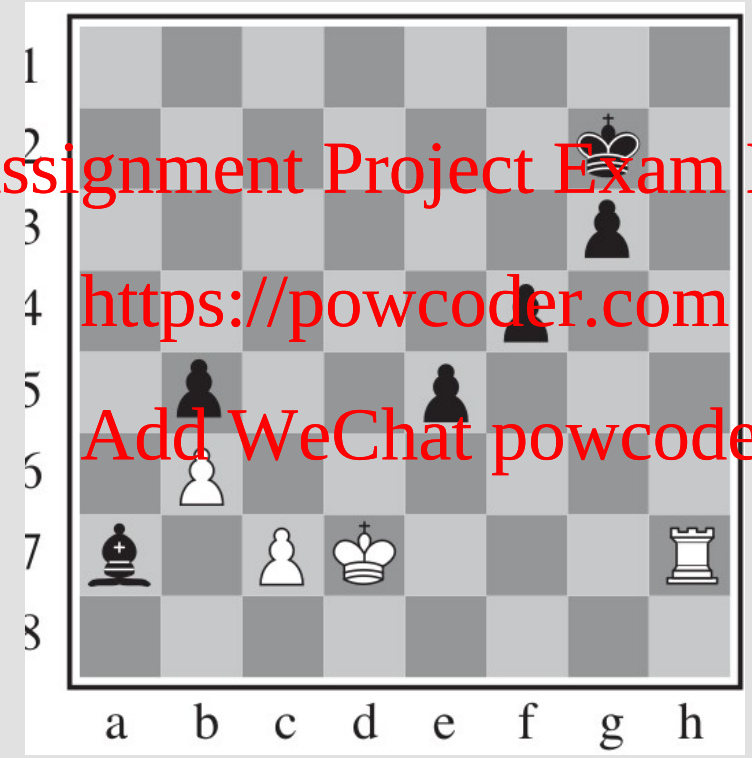
Search considerations

Potentially misleading evaluations



Search considerations

Potentially misleading evaluations



Heuristic AB Tree Search

Starting from a fixed board position: (state, player)

1)Run AB up to some limit (DFS, or DFID)
2)Treat the frontier nodes s' as terminal states
3)Compute a score using EVAL(s', p)
4)Back up the values
5)Pick the best move
6)Repeat from the new board position

Types of Search

- Type 1:
  wide, but shallow

- Type 2:
  narrow, but deep

- **Historically practitioners preferred Type 1 Searches.**
- **More recently, Type 2 approaches (and Type1/2 combinations) have emerged.**

Monte Carlo Simulation

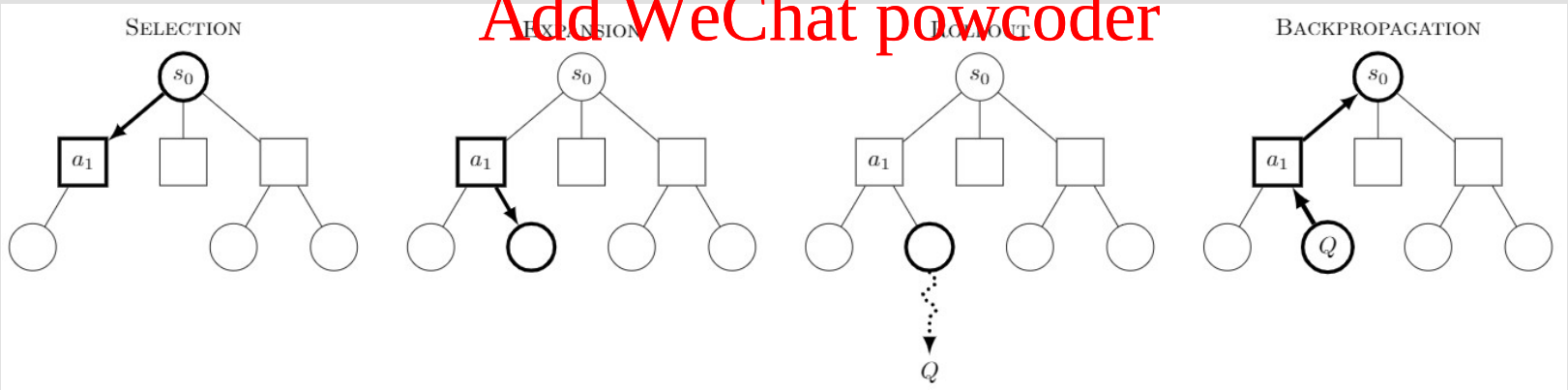
Use sampling to obtain estimates of utility at a given state

- Starting from a given position (s, p)
- "Roll out" the game until a terminal state
- Fixed policy determines the moves at each ply (fast!)
- Repeat many times to obtain an "average" utility

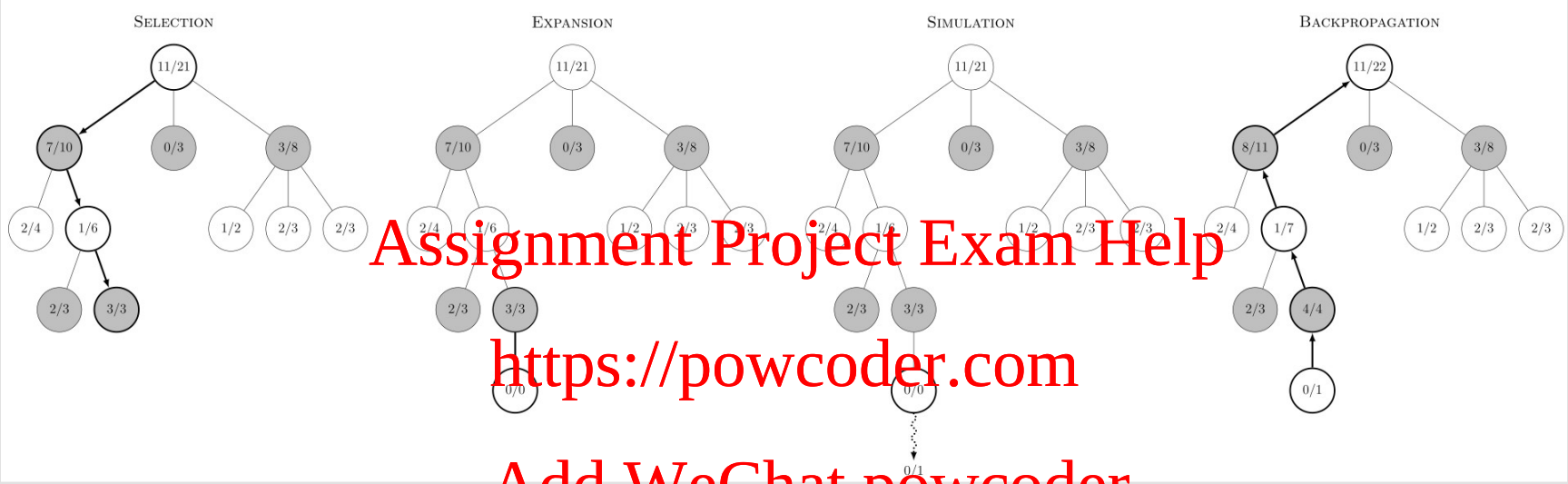
Monte Carlo Tree Search

Four main ingredients:

1) Selection: pick a node from the frontier
2) Expand: generate successors to extend the frontier
3) Simulate games from the newly added node
4) Propagate the results



Monte Carlo Tree Search



Instead of always maximising average utility, we can bias the choices: **exploration vs exploitation**

Deterministic Games in Practice

- **Checkers**: Chinook defeated the world champion in an abbreviated game in 1990. It uses αβ search combined with a pre-computed database defining perfect play for 39 trillion endgame positions.
- **Chess**: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 30 billion positions per move (200 million per second), normally searching to depth 14, and extending the search up to depth 40 for promising options. Heuristics reduce the EBF to about 3.
- **Othello**: In 1997, a computer defeated the world champion 6-0. Humans are no match for computers.
- **Go**: *b ~ 19 x 19 = 361*, which is too large for αβ. In 2016, AlphaGo, which uses Deep Learning, beat the world champion 4-1.
- AlphaGo also used Monte Carlo Tree Search (MTCS)

Summary: Adversarial Search

Games illustrate important points about AI

- **Perfection is unattainable → must approximate**
- **It causes one to think about which problems are worth thinking about**

The basic ideas discussed today generalise:

- **>2 players → more ply and more costs**
- **Stochastic or hidden information**
  → probabilistic reasoning (expectimax)

Next week – Logical Agents

- **Propositional logic**
  - **Inference**
  - **Resolution**
  - **Satisfiability**
- **First Order Logic**
- **Logic and agents**