Lecture 10
Adversarial Search

Song Chaoyang

Assistant Professor

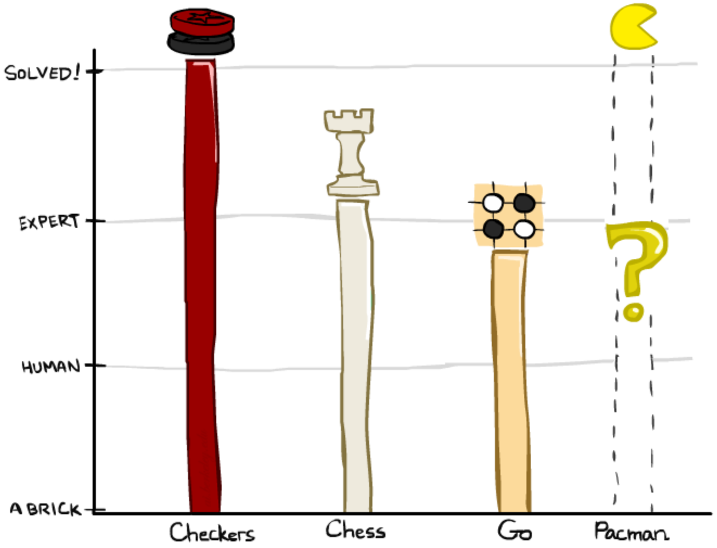
Department of Mechanical and Energy Engineering

songcy@sustech.edu.cn

[Slides adapted from Sergey Levine & Stuart Russell, CS188, UCB]

Game Playing State-of-the-Art

- **Checkers:**
  - 1950: First computer player.
  - 1994: First computer champion.
  - 2007: Checkers solved!
- **Chess:**
  - 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply.
- **Go:**
  - **2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.**

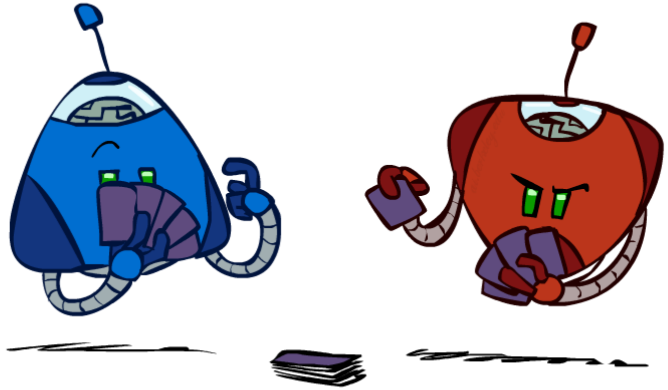


AncoraSIR.com

SUSTech
Southern University of Science and Technology

Types of Games

- Many different kinds of games!

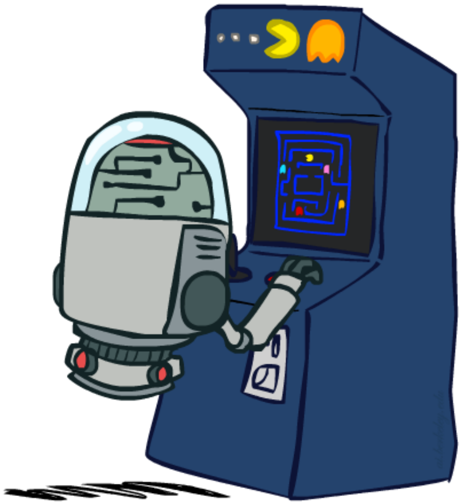


- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?

- Want algorithms for calculating a strategy (policy) which recommends a move from each state

Deterministic Games

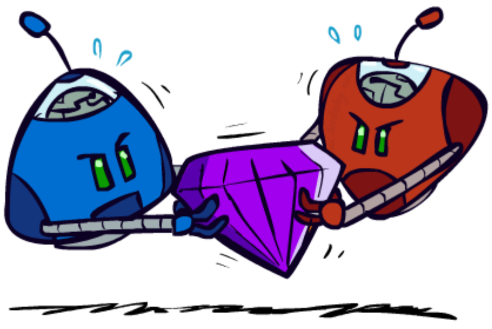
- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R

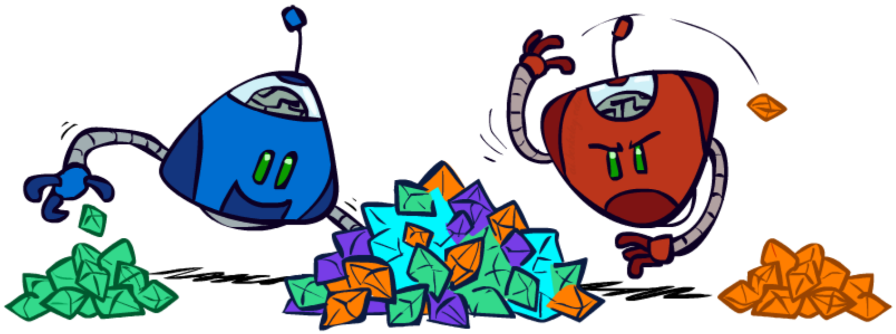


- Solution for a player is a policy: S $\rightarrow$ A

AncoraSIR.com

Zero-Sum Games





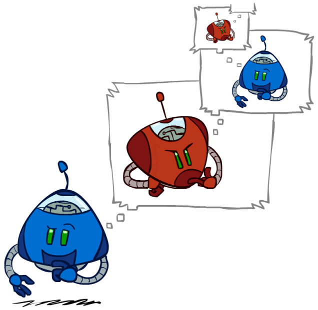
- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - ***Adversarial***, pure competition
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

AncoraSIR.com

SUSTech
Southern University of Science and Technology

Adversarial Search Problems

Elements



- ***Multiagent environments***
  - each agent needs to consider the actions of other agents, which is unpredictable and how they affect its own welfare.

- ***Competitive*** environments
  - the agents' goals are in conflict, giving rise to ***adversarial search*** problems—often known as ***games***.

- In all, deterministic, turn-taking, two-player, zero-sum games

  of perfect information (such as chess).

AncoraSIR.com

Game as search problem

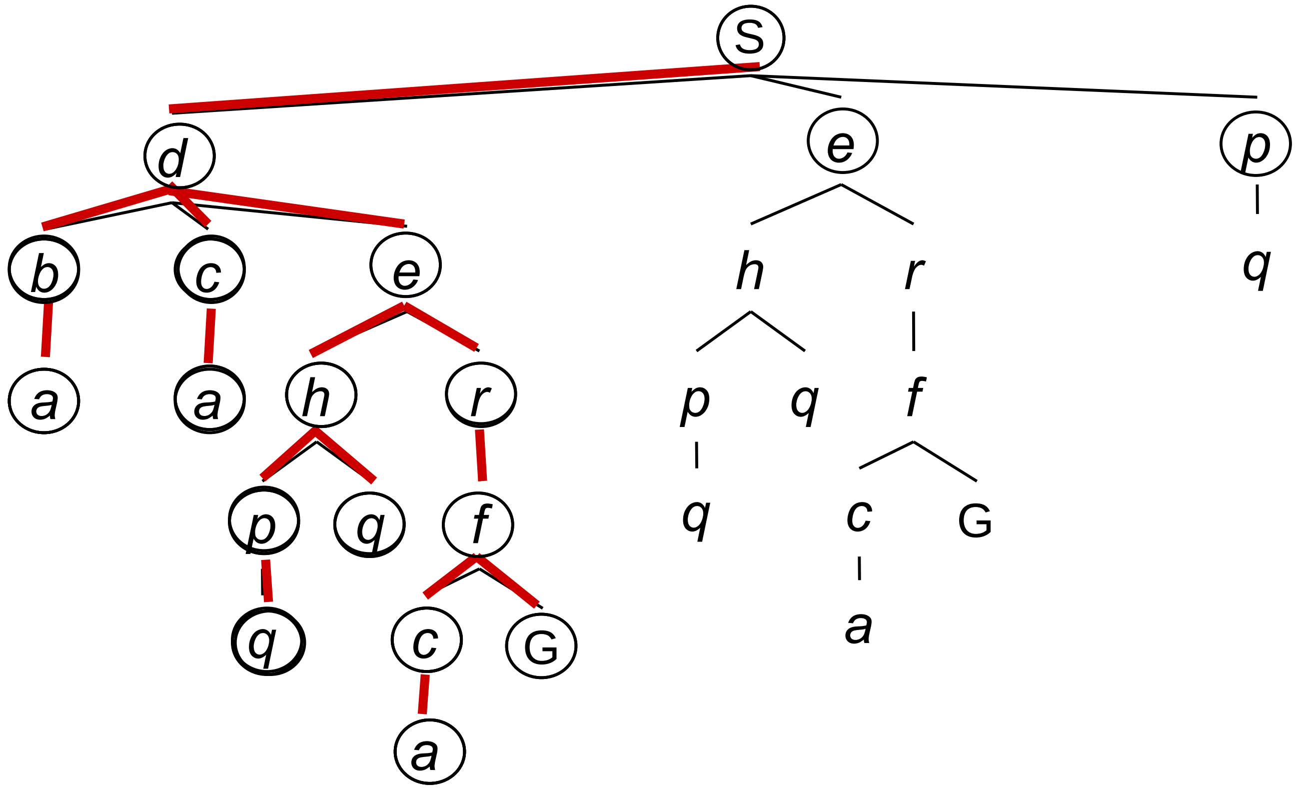
Definition

- Two players MAX and MIN. MAX moves first.

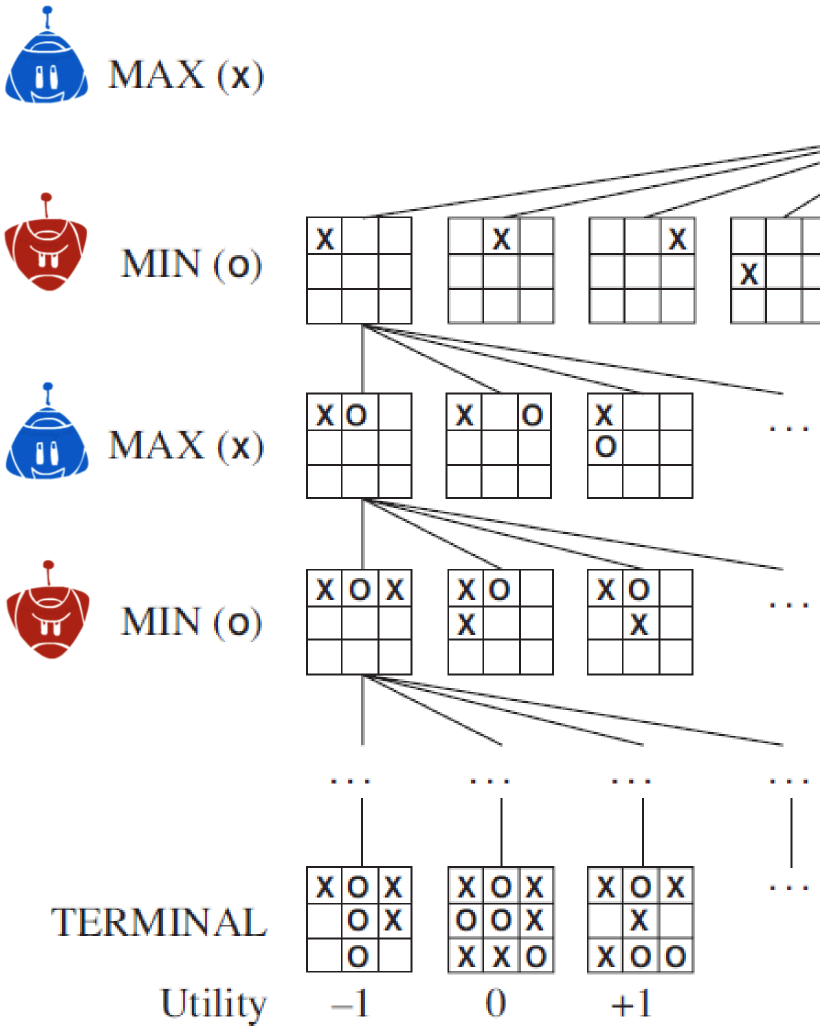
- A game as a kind of search problem has the following elements:
  - S0: ***initial state***
  - PLAYER(s): Defines which player has the move in a state.
  - ACTIONS(s): Returns the set of legal moves in a state.
  - RESULT(s, a): ***transition model***.
  - TERMINAL-TEST(s): ***terminal test***, true when the game is over and false otherwise.
  - UTILITY(s, p): ***utility function*** defines the final numeric value for a game that ends in terminal state s for a player p.

Single-Agent Trees

Recall

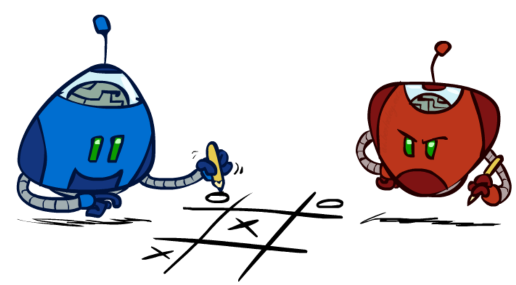


Games tree – Tic-tac-toe



- From the initial state, **MAX(crosses)** has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach terminal states.

- The game tree is large with 9! = 362880 terminal nodes.

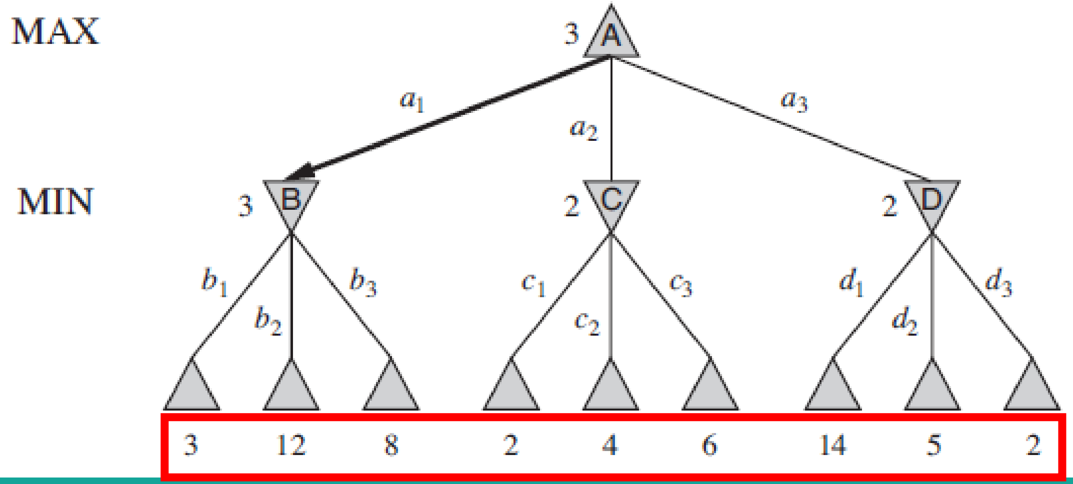


AncoraSIR.com

Adversarial Search (Minimax)

How to find the optimal decision

- A two-ply game tree and players alternate turns.
  - △ nodes are "MAX nodes", ▽ nodes are "MIN nodes"
- Compute each node's ***minimax value***: the best achievable utility against a rational (optimal) adversary
- The terminal nodes show the utility values for MAX



AncoraSIR.com

Minimax Implementation

Defined in recursive function

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

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

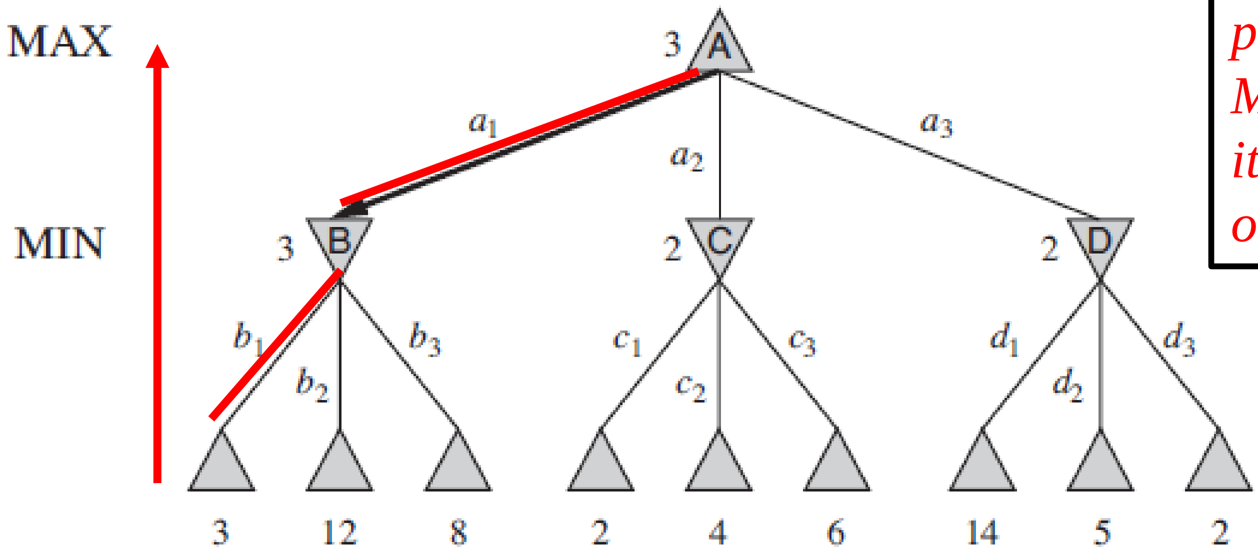
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

AncoraSIF

Optimal Decision in Games

Compute minimax value

- Minimax value is computed bottom up
  1. 3 is the best outcome for MIN in node B, 2 is the best outcome for MIN in node C and D.
  2. 3 is the best outcome for MAX in node A.

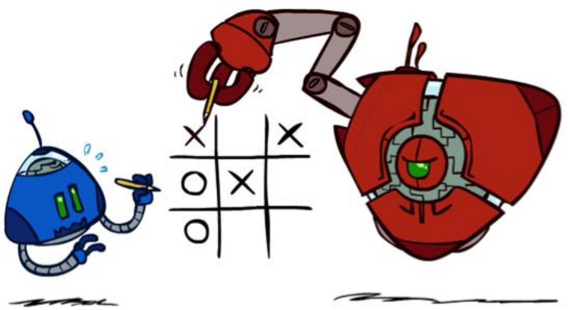


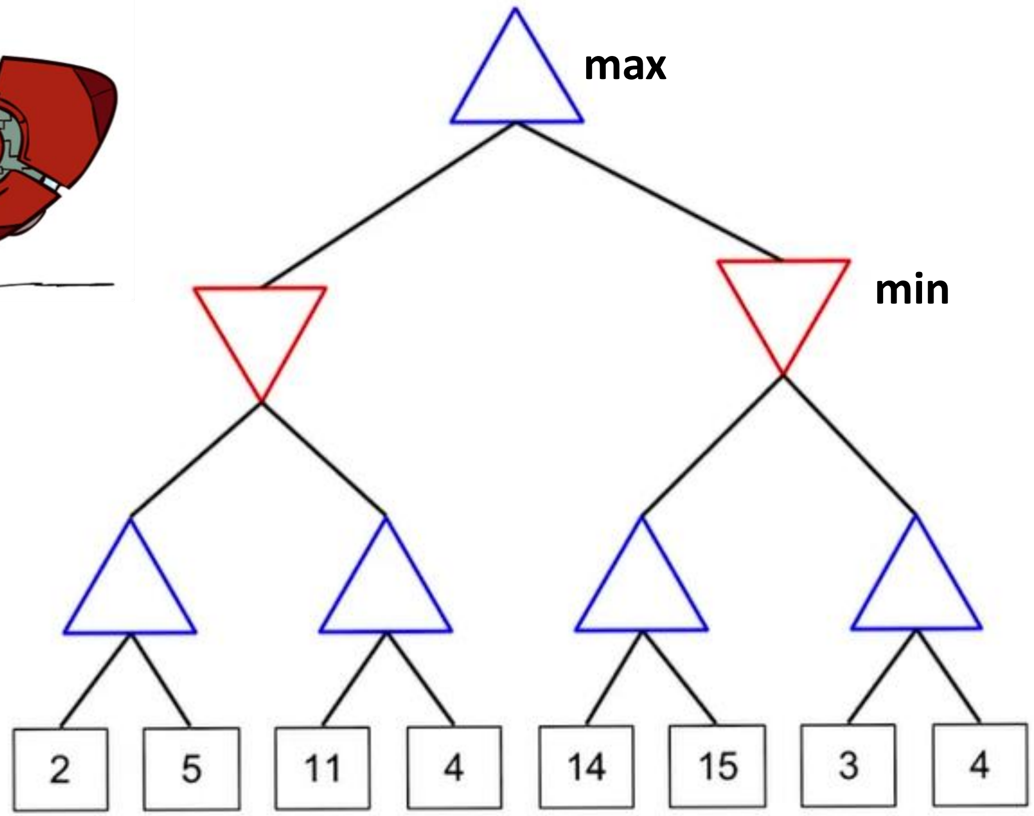
This definition of optimal play for MAX assumes that MIN also plays optimally — it maximizes the worst-case outcome for MAX.

Optimal Decision in Games

Compute minimax value

- Minimax Quiz

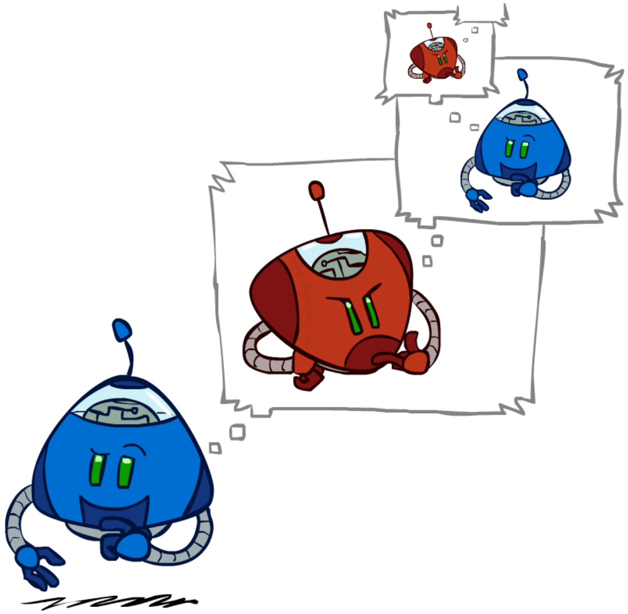




AncoraSIR.com

Minimax Efficiency

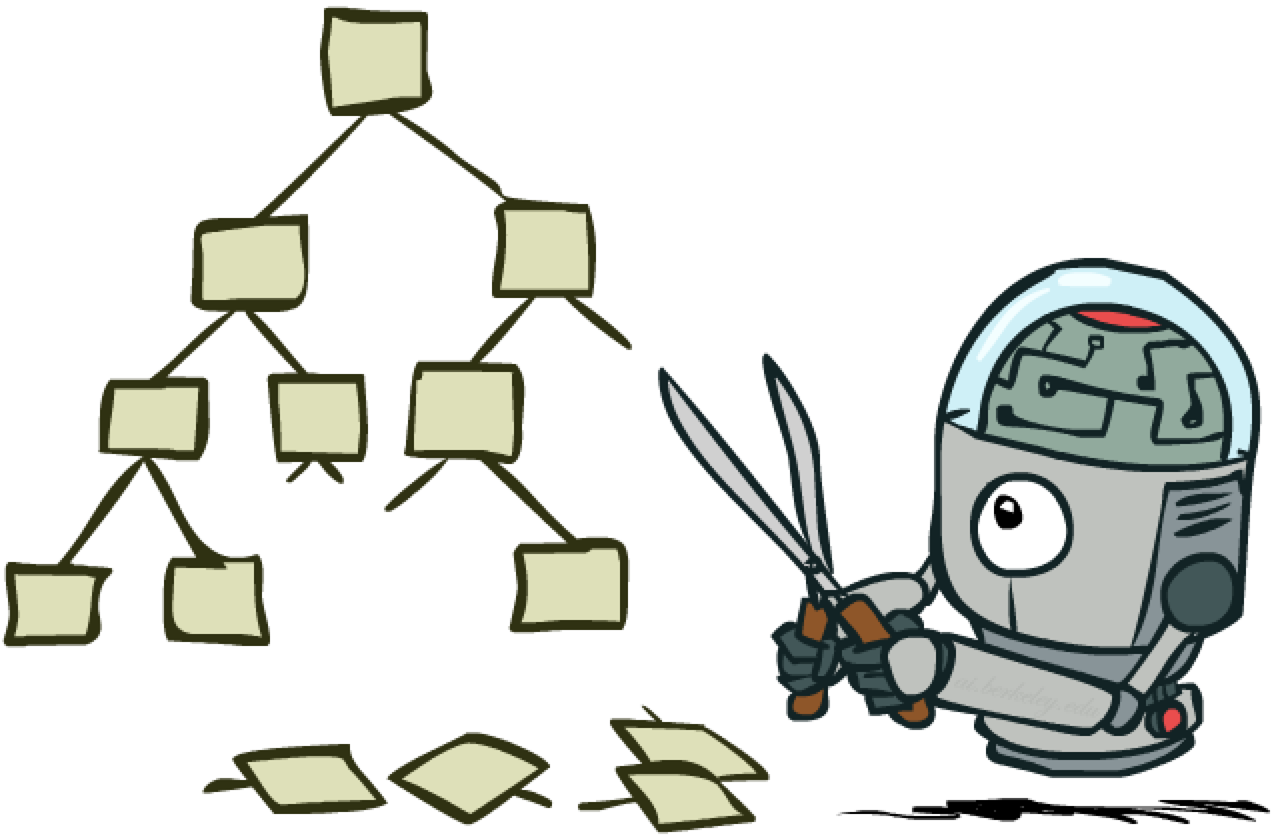
- If the maximum depth of the tree is m and there are b legal moves at each point, then
  - Just like (exhaustive) DFS
  - The time complexity is $O(b^m)$
  - The space complexity is $O(bm)$



- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution completely infeasible.
  - Do we need to expand all nodes?

AncoraSIR.com

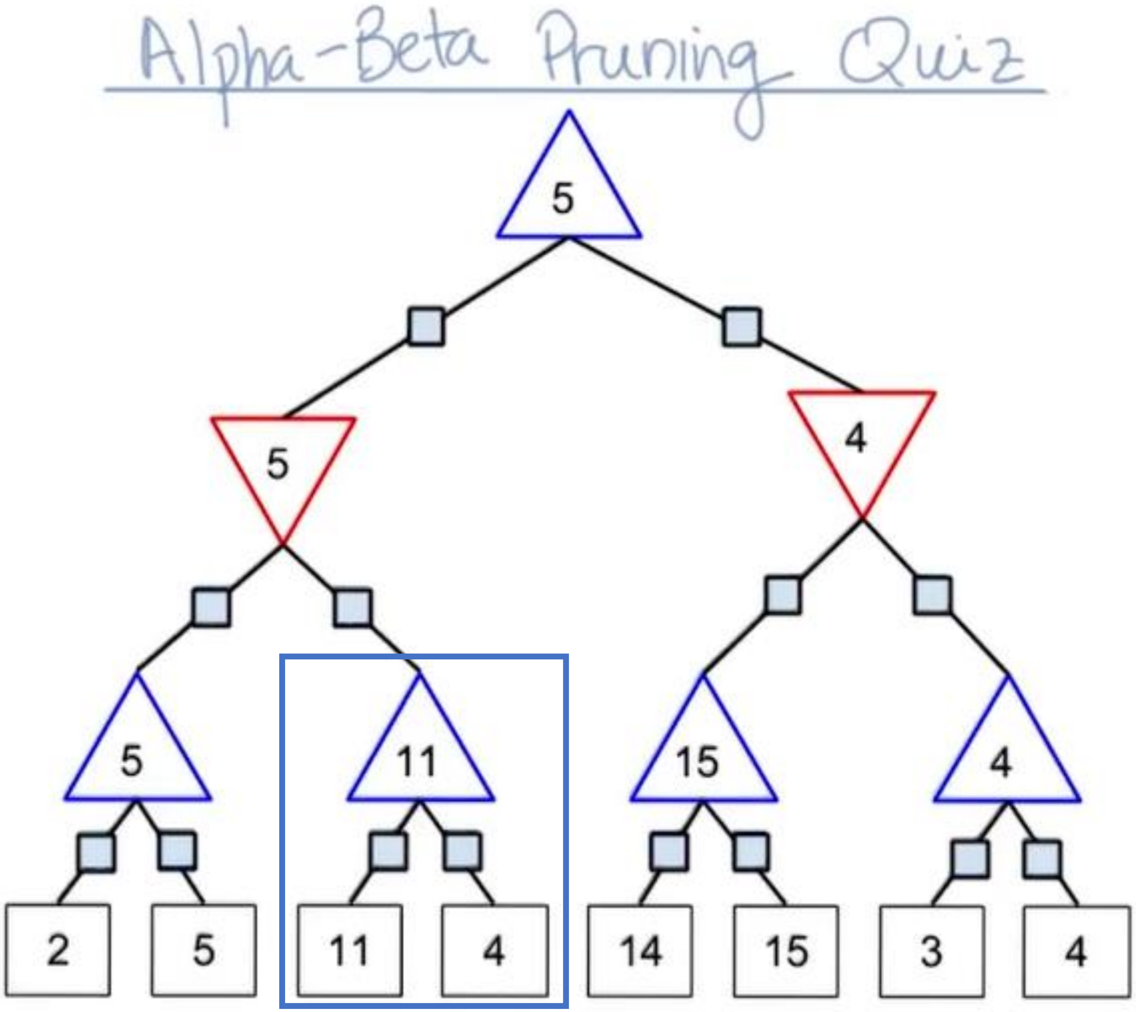
Game Tree Pruning



AncoraSIR.com

Alpha-Beta Pruning

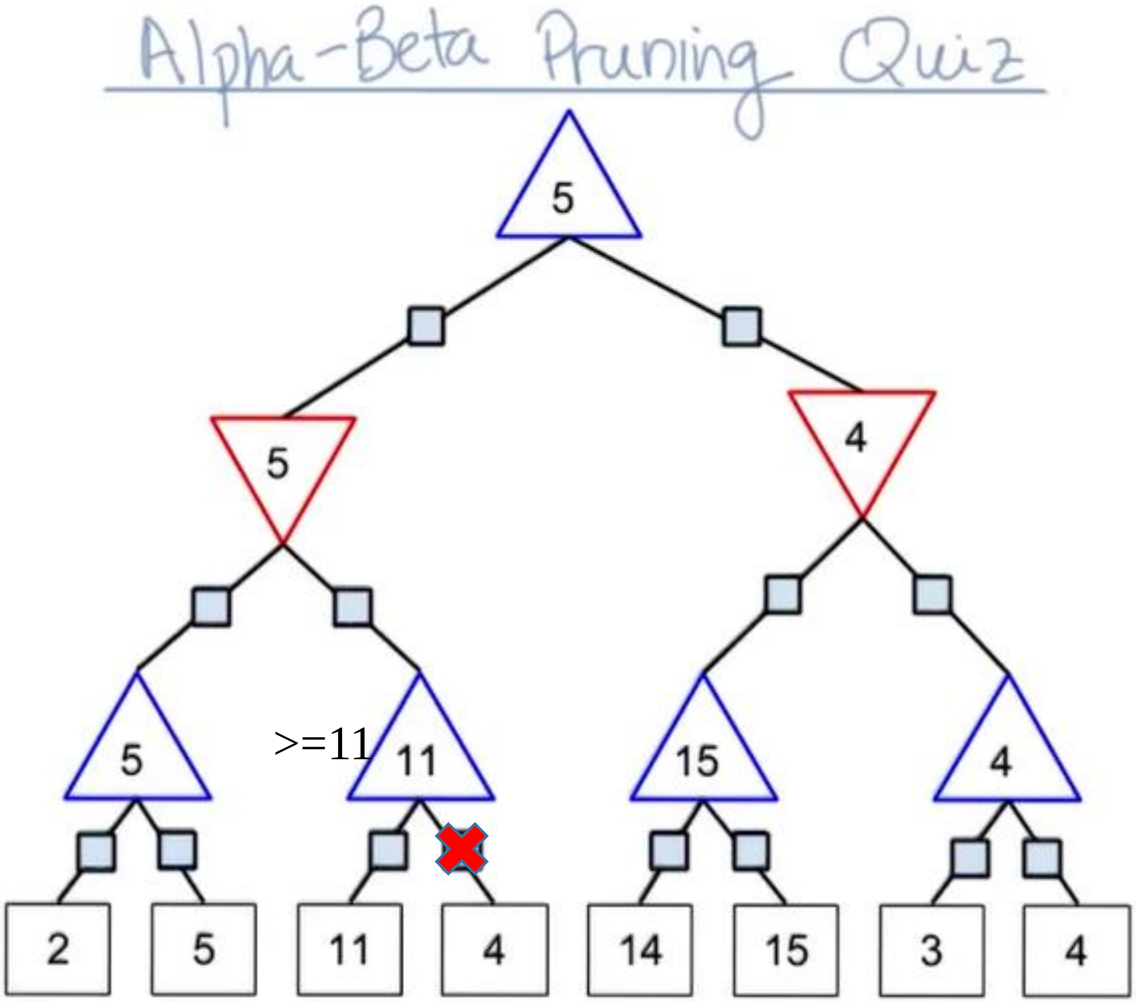
Quiz



AncoraSIR.com

Alpha-Beta Pruning

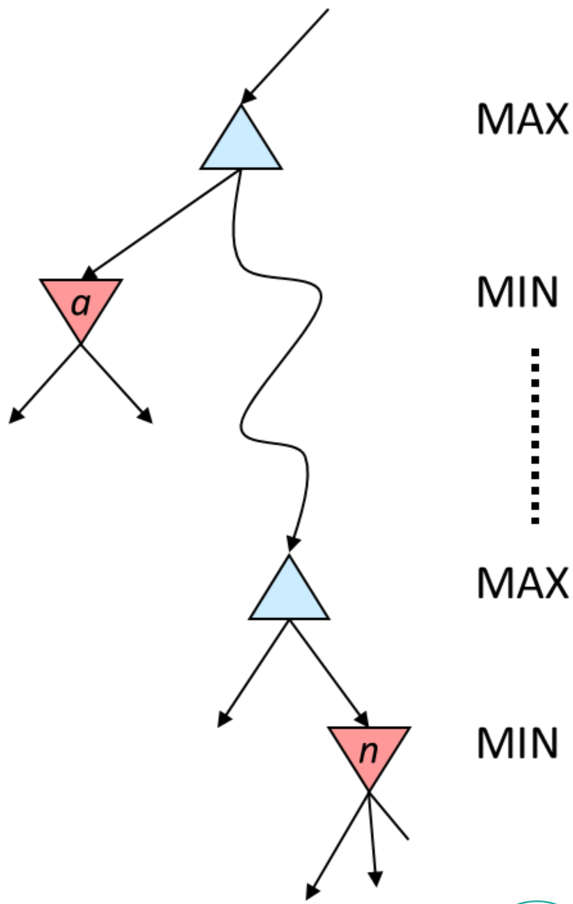
Quiz



AncoraSIR.com

Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)



- MAX version is symmetric

Alpha-Beta Implementation

α: MAX's best option on path to root

β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Pruning

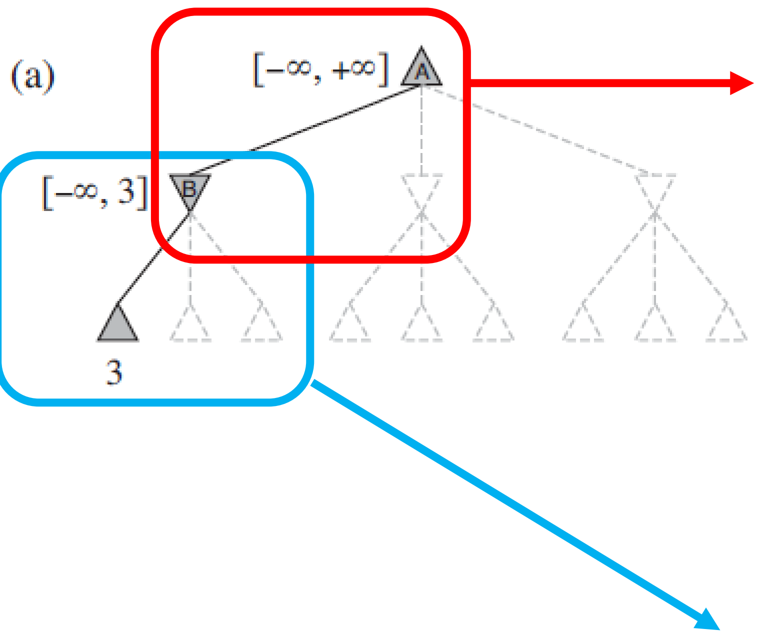
- This pruning has no effect on minimax value computed for the root!
  - Returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
- Time complexity drops to $O(b^{m/2})$
  - Full search of, e.g. chess, is still hopeless…

SUSTech
Southern University of Science and Technology

AncoraSIR.com

Alpha-Beta Pruning

[α, β]



function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

MAX-VALUE(A, -∞, +∞)

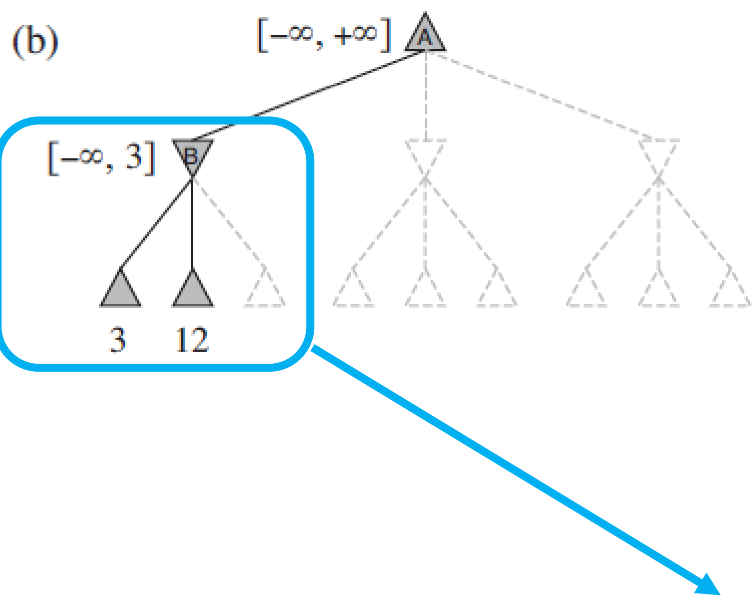
function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  B
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha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

MIN-VALUE(B, -∞, +∞)

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
   3 $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v 3
   3 $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-Beta Pruning

[α, β]



function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

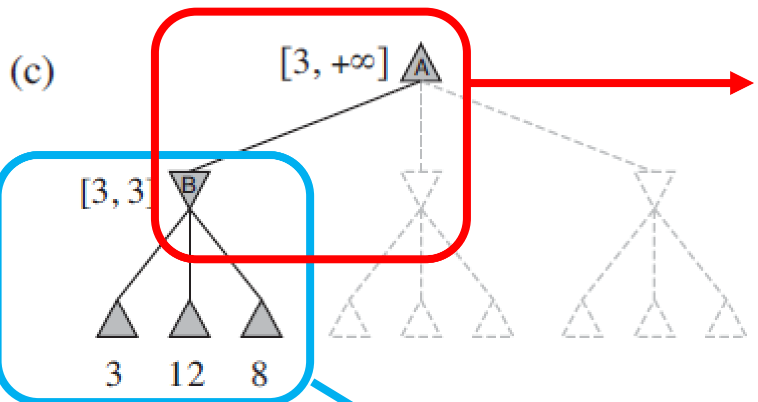
function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
 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

MIN-VALUE(B, -∞, +∞)

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    3 $v \leftarrow$ MIN(3 v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v 12
    3 $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-Beta Pruning

[α, β]



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

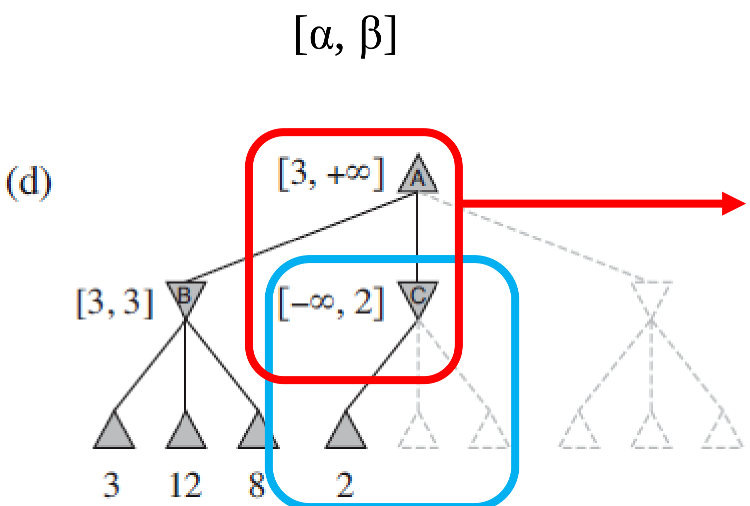
---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  B
  for each a **in** ACTIONS($state$) **do**
   3 $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
   3
   if $v \geq \beta$ **then return** v
   3 $\alpha \leftarrow$ MAX(α, v) MAX(-∞, 3)
  return v

MIN-VALUE(B, -∞, +∞)

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
   3 $v \leftarrow$ MIN(v 3, MAX-VALUE(RESULT(s,a), α, β))
   8
   if $v \leq \alpha$ **then return** v
   3 $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-Beta Pruning



The left two actions from C are skipped!

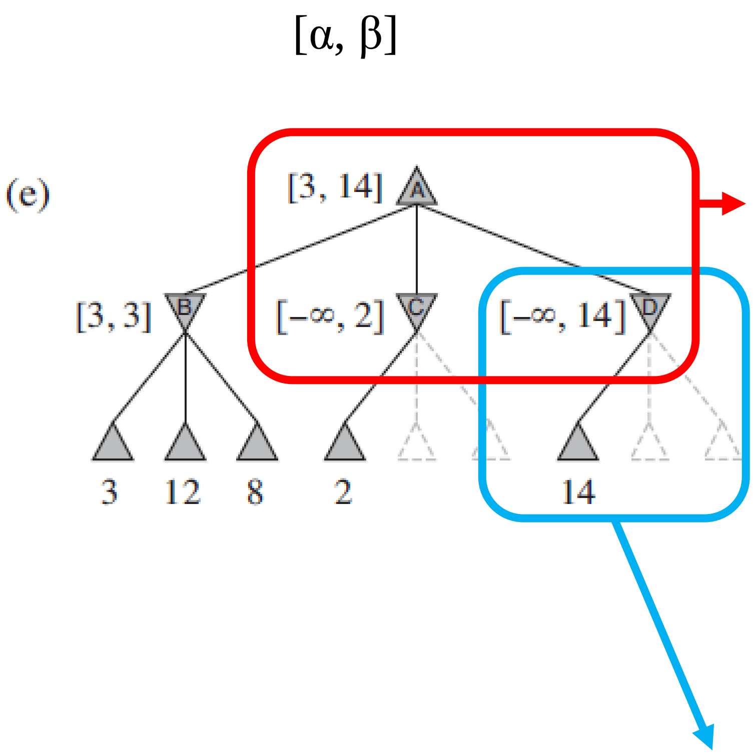
```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in ACTIONS(state) with value v
```

```
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))      C
        if v ≥ β then return v
        α ← MAX(α, v)  MAX(3, 2)
    return v
```

MIN-VALUE(C, 3, +∞)

```
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
      2 v ← MIN(v, MAX-VALUE(RESULT(s,a), α, β))      2
        if v ≤ α then return v        (v = 2, α = 3)
        β ← MIN(β, v)     2
    return v
```

Alpha-Beta Pruning



function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

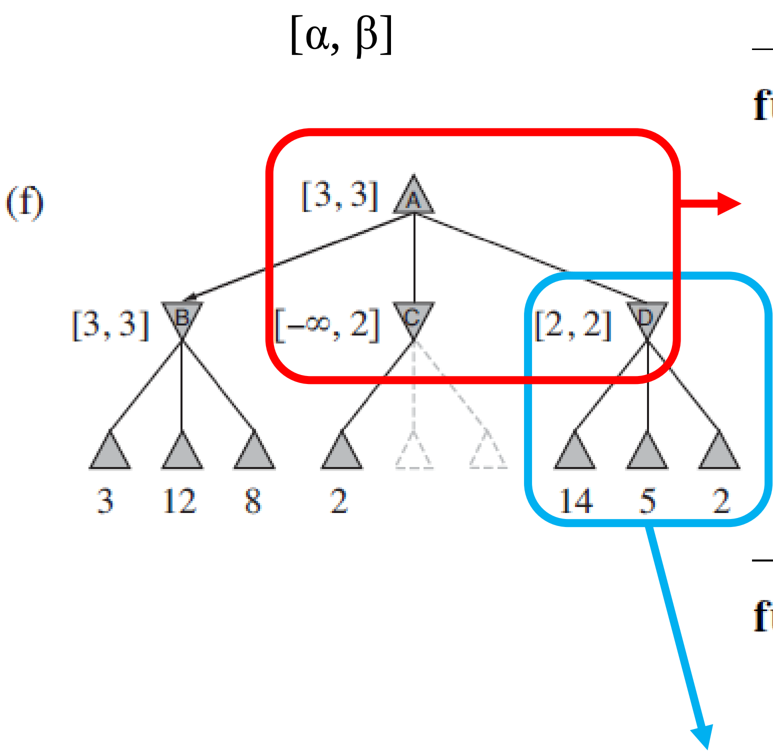
---

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

MIN-VALUE(D, 3, +∞)

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
  14 $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v 14
  14 $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-Beta Pruning

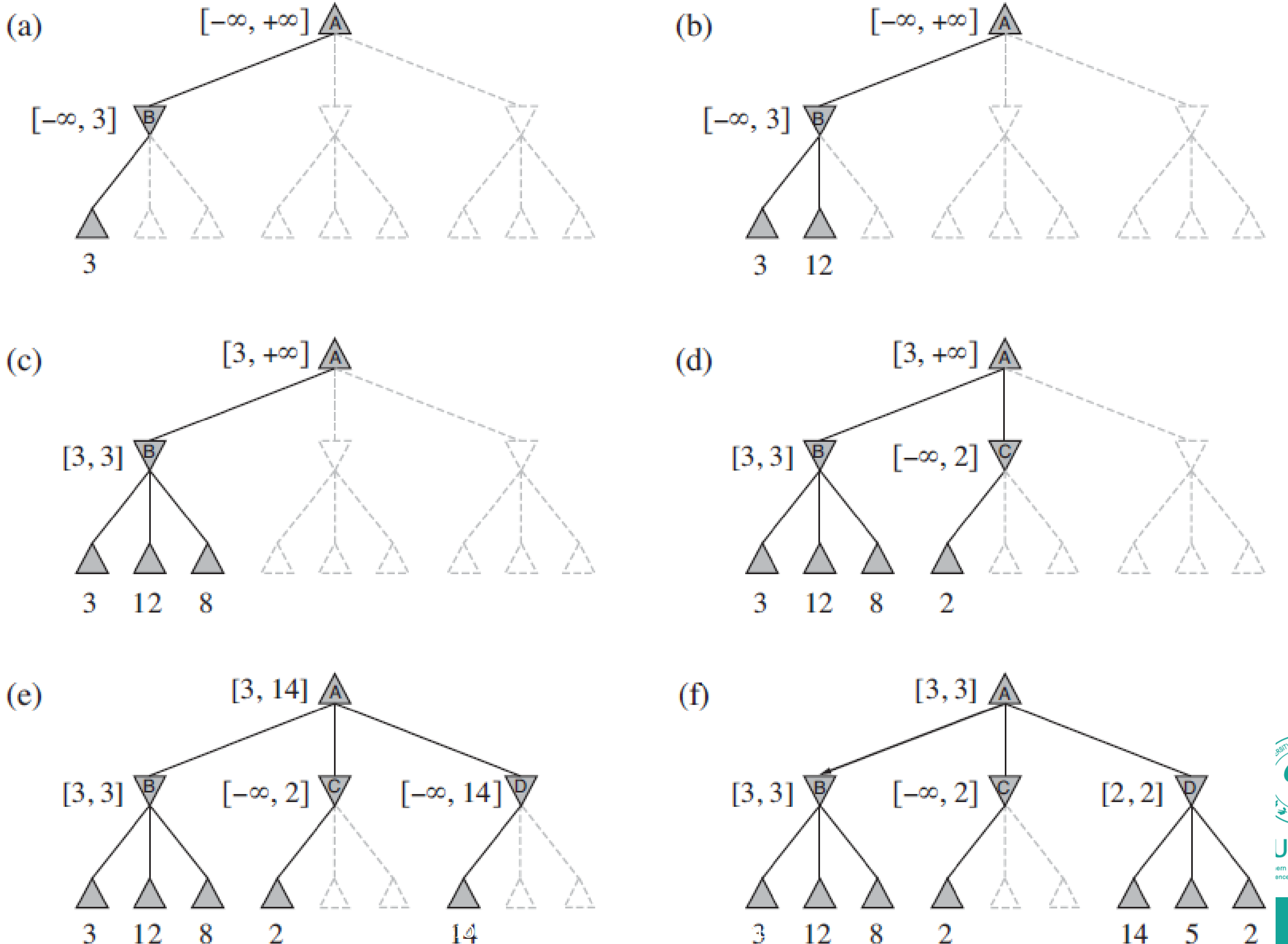



```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v
```

```
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
                                        D
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
                                                2
      if v ≥ β then return v
  3α ← MAX(α, v)
   return v
```

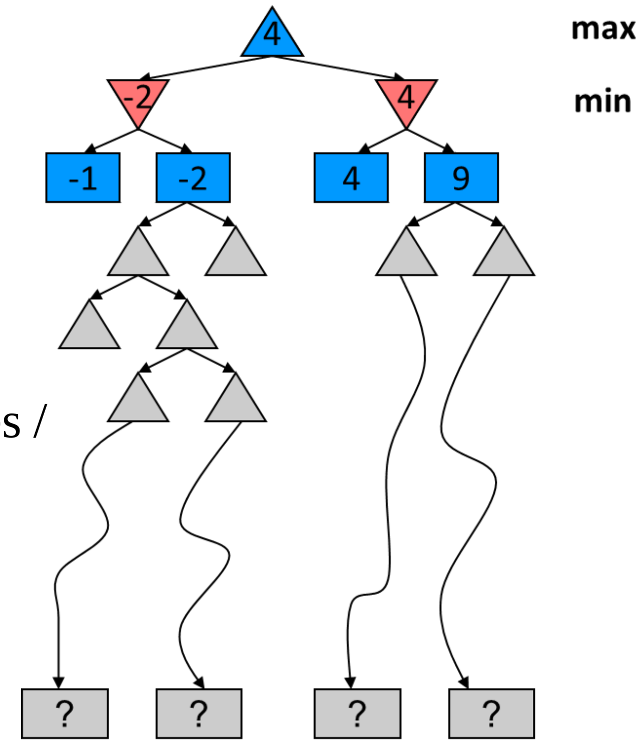
```
             MIN-VALUE(D, 3, +∞)
function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
  2   v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
                                                2
      if v ≤ α then return v
  2   β ← MIN(β, v)
   return v 2
```

Alpha-Beta Pruning



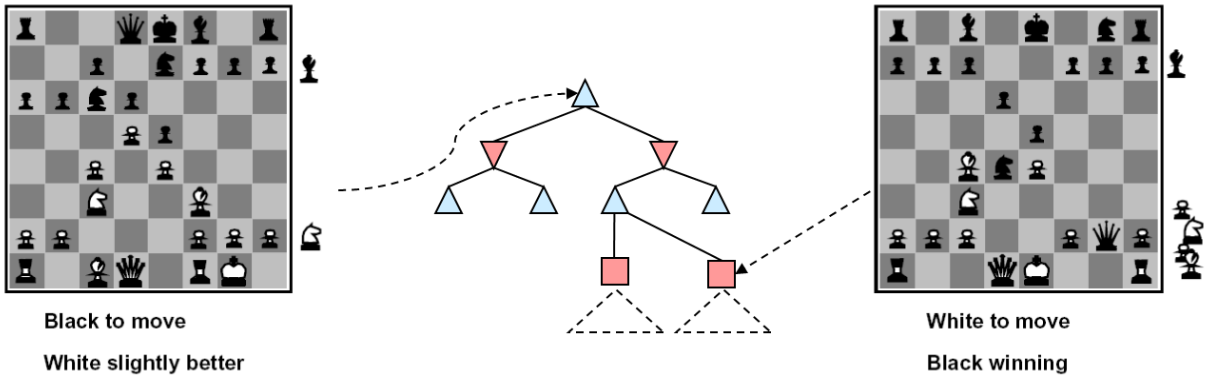
Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- Use iterative deepening for an anytime algorithm



Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search



- Ideal function: returns the actual minimax value of the position

- In practice: typically weighted linear sum of features:

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

- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

AncoraSIR.com

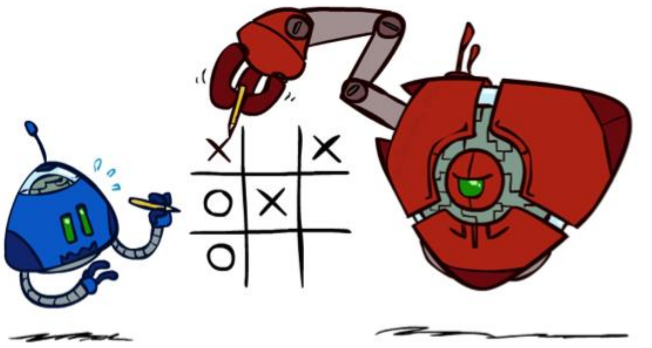
Depth Matters

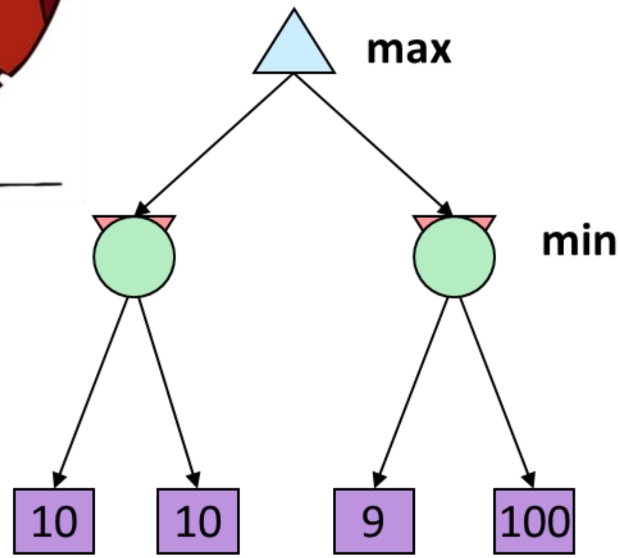
- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

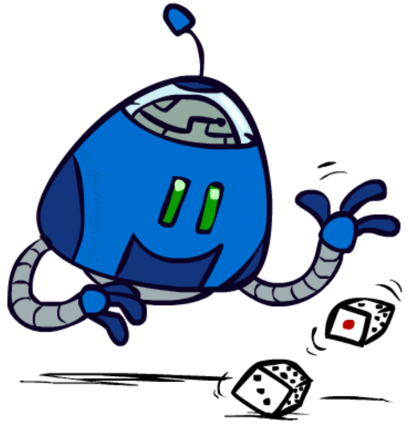
AncoraSIR.com

Uncertain Outcomes

Worst-Case vs. Average Case





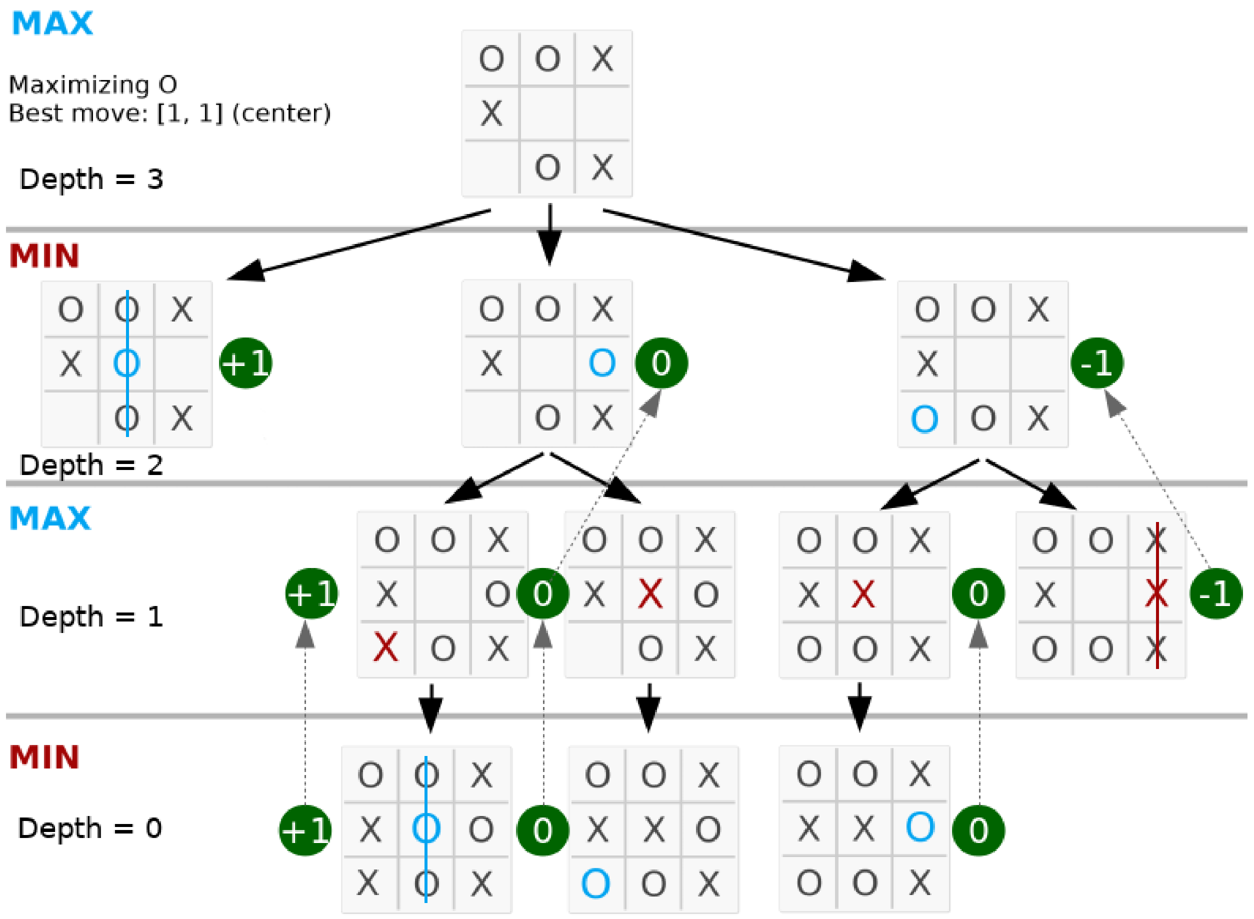


Idea: Uncertain outcomes controlled by chance, not an adversary!

AncoraSIR.com

Project 3

- Build an Tic Tac Toe game with Minimax algorithm.



AncoraSIR.com

Building a Game Class

- Class TicTacToe

  ❑***self.board***: Representing the game state, track the current position each player on the board.

```
self.board = [" ", " ", " ",
              " ", " ", " ",
              " ", " ", " "]
```

  ❑***self. availableMoves*** (move, “O”): Finding all legal moves for a player

  ❑***self. makeMove***: Make a move and update the board

SUSTech Southern University of Science and Technology

AncoraSIR.com

Building a Game Class

- Class TicTacToe
  - ❑***self.checkWin***: return the winner of the game or none if the no winners.

```
Win combos = ([0, 1, 2], [3, 4, 5], [6, 7, 8],
              [0, 3, 6], [1, 4, 7], [2, 5, 8],
              [0, 4, 8], [2, 4, 6])
```

  - ❑***self.gameOver***: return true if X player wins or O player wins or draw (no winners and the board is full), otherwise return false

AncoraSIR.com

Building a Game Class

- Class TicTacToe
  - ❑***self.make_best_move***: evaluates all the available moves using **minimax()** and then returns the best move the maximizer can make

```
function make_best_move(board):
    bestMove = NULL
    for each move in board :
        if current move is better than bestMove
            bestMove = current move
    return bestMove
```

Note: There maybe no optimal or multiple optimal moves

AncoraSIR.com

Building a Game Class

- ***self.minimax***: consider all the possible ways the game can go and returns the best value for that move

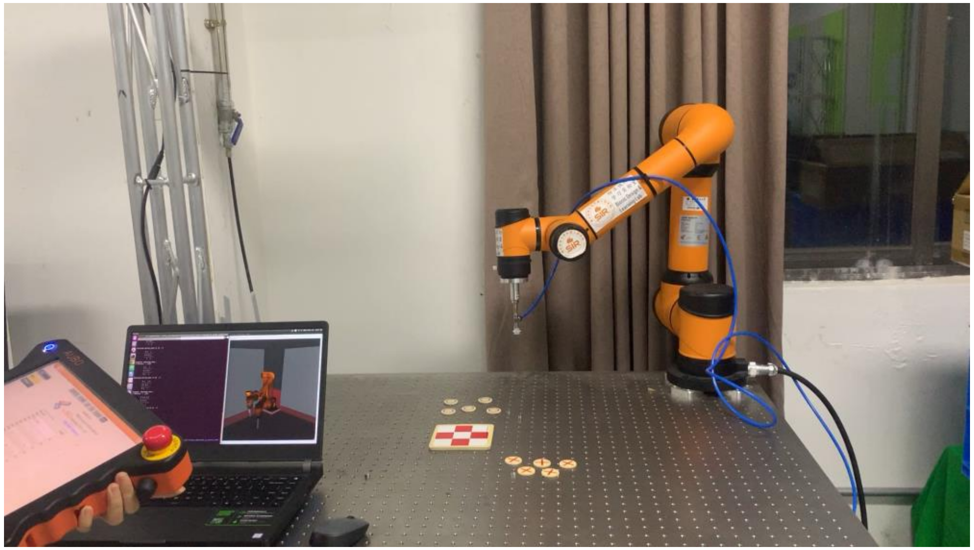
```
function minimax(board, depth, isMaximizingPlayer):

    if current board state is a terminal state:
        return value of the board

    if isMaximizingPlayer:
        bestVal = -INFINITY
        for each move in board :
            value = minimax(board, depth+1, false)
            bestVal = max( bestVal, value)
        return bestVal

    else:
        bestVal = +INFINITY
        for each move in board:
            value = minimax(board, depth+1, true)
            bestVal = min( bestVal, value)
        return bestVal
```

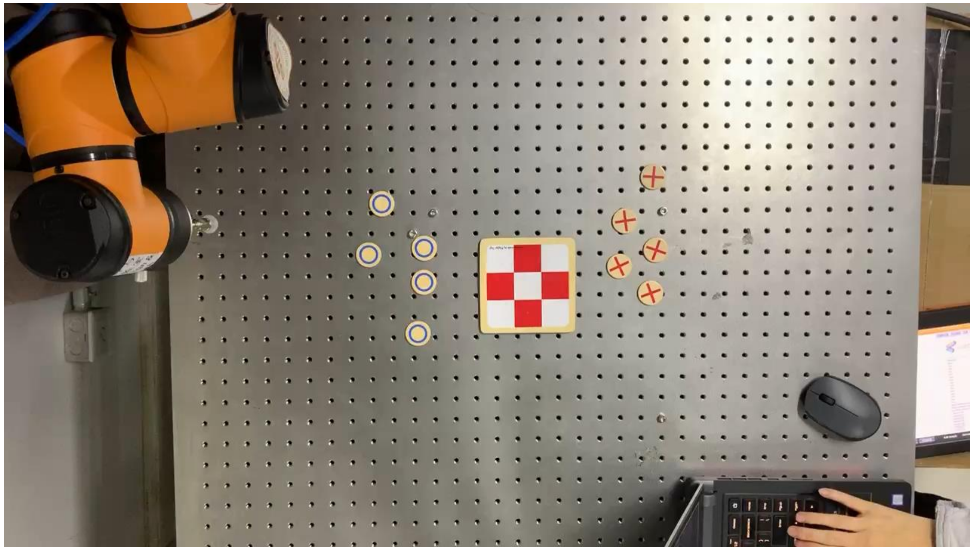
```
The utility of a
terminal state has
three possible value:
1.  X wins: -1
2.  O wins: 1
3.  Draw: 0
```

AncoraSIR.com

SUSTech
Southern University of Science and Technology

Project Demo





AncoraSIR.com

Thank you!

Prof. Song Chaoyang

- Dr. Wan Fang (sophie.fwan@hotmail.com)

AncoraSIR.com