Artificial Intelligence

CS482, CS682, MW 1 – 2:15, SEM 201, MS 227

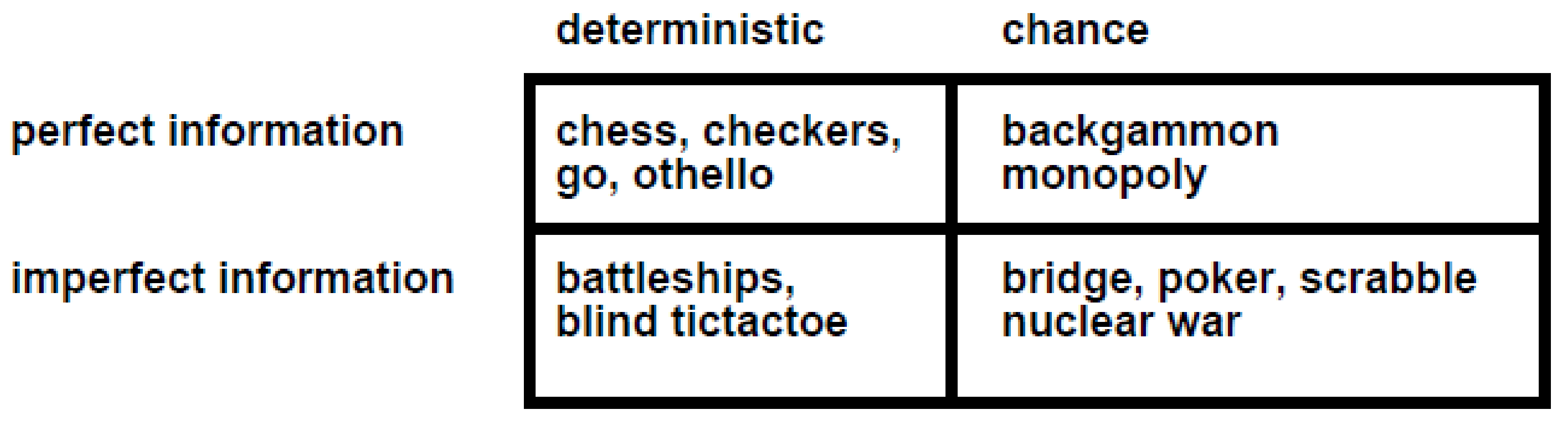
Prerequisites: 302, 365

Instructor: Sushil Louis, sushil@cse.unr.edu, http://www.cse.unr.edu/~sushil

Games and game trees

- Multi-agent systems + competitive environment → games and adversarial search
- In game theory any multiagent environment is a game as long as each agent has “significant” impact on others
- In AI many games were
  - Game theoretically: Deterministic, Turn taking, Two-player, Zero-sum, Perfect information
  - AI: deterministic, fully observable environments in which two agents act alternately and utility values at the end are equal but opposite. One wins the other loses
- Chess, Checkers
- Not Poker, backgammon,

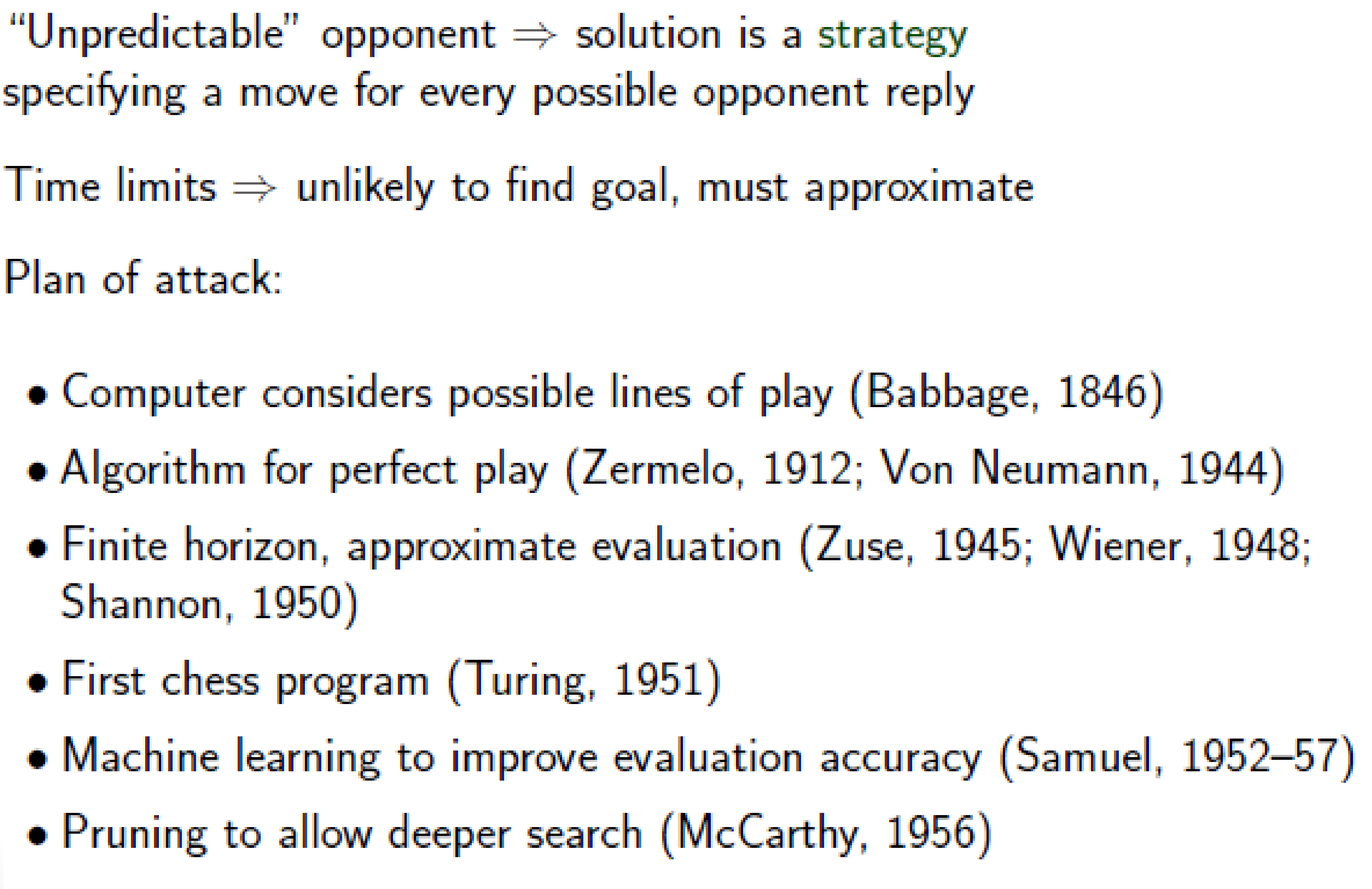
Game types



	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

Starcraft? Counterstrike? Halo? WoW?

Search in Games



"Unpredictable" opponent ⇒ solution is a strategy specifying a move for every possible opponent reply

Time limits ⇒ unlikely to find goal, must approximate

Plan of attack:

- Computer considers possible lines of play (Babbage, 1846)
- Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
- Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- First chess program (Turing, 1951)
- Machine learning to improve evaluation accuracy (Samuel, 1952–57)
- Pruning to allow deeper search (McCarthy, 1956)

Tic-Tac-Toe

- Two player, deterministic, small tree
- Two players: Max versus Min
- Approximately: 9! tree nodes

Tic-Tac-Toe

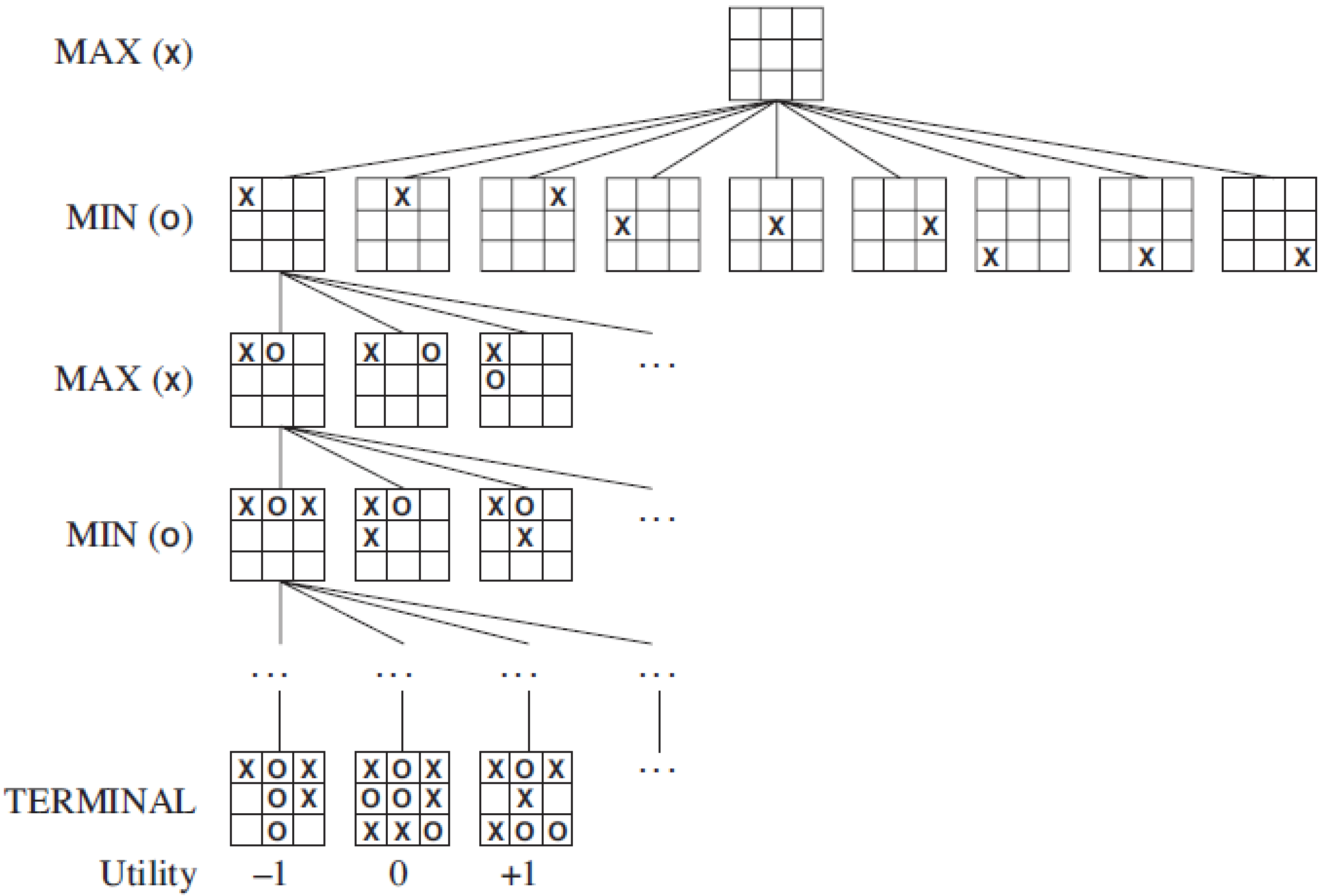


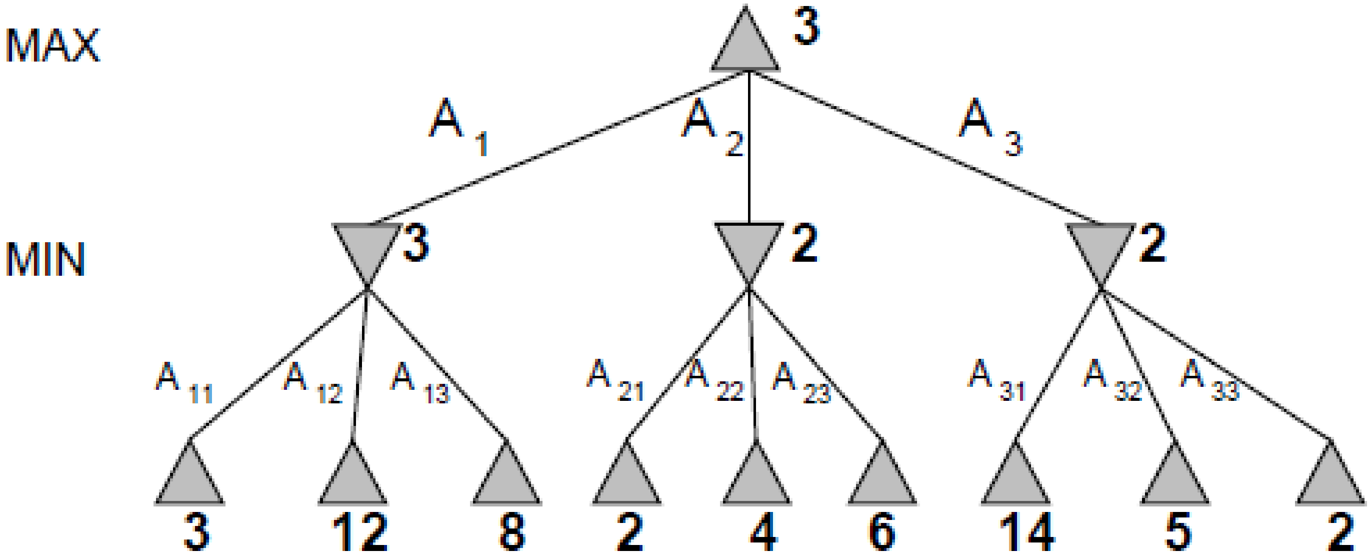
Figure 5.1 **FILES: figures/tictactoe.eps (Tue Nov 3 16:23:55 2009).** A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Minimax search

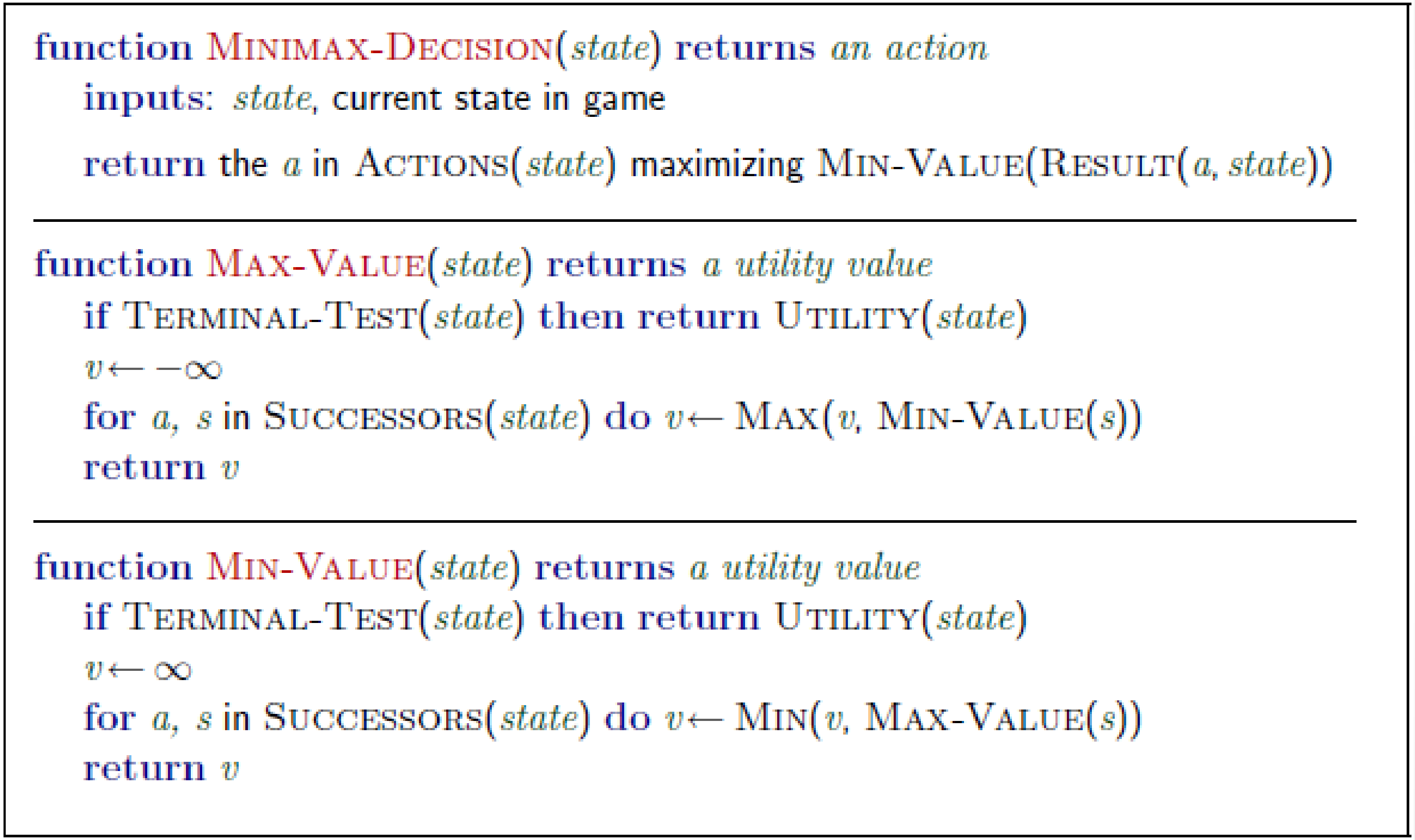
Perfect play for deterministic, perfect-information games

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

E.g., 2-ply game:



Minimax 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)
    $v \leftarrow -\infty$
    for a, s in SUCCESSORS(state) do $v \leftarrow$ MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    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))
    return v
```

3 player Minimax

- Two player minimax reduces to one number because utilities are opposite – knowing one is enough
- But there should actually be a vector of two utilities with player choosing to maximize their utility at their turn
- So with three players → you have a 3 vector
- Alliances?

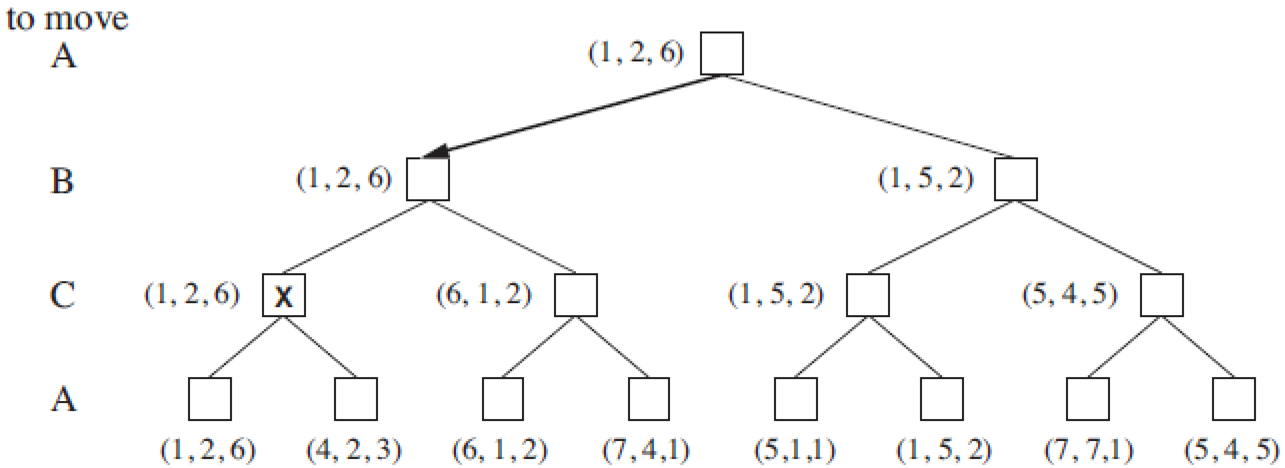
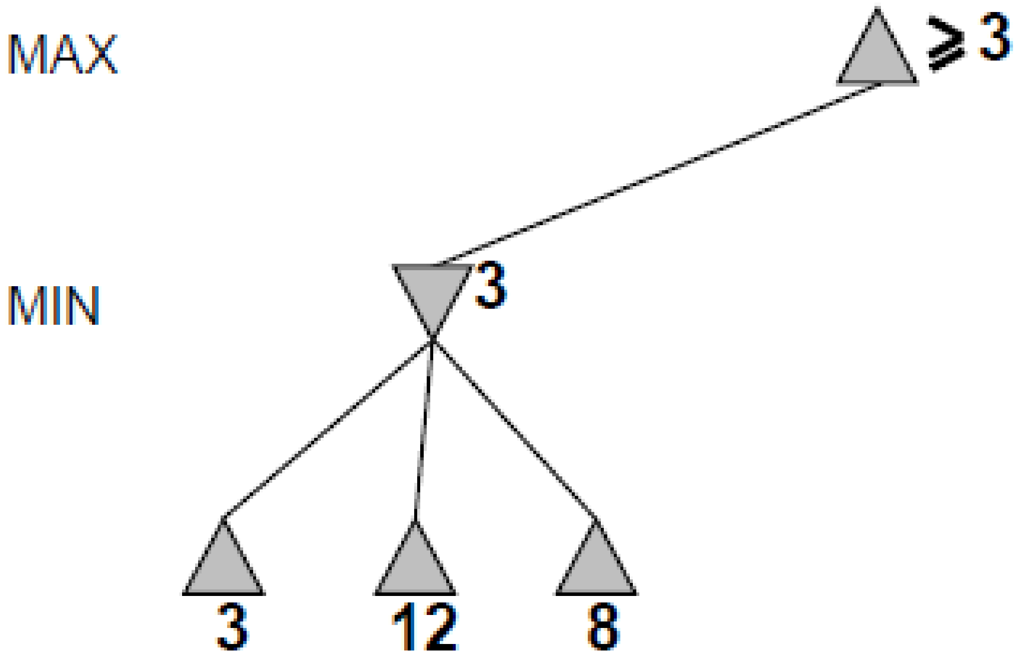


Figure 5.4 **FILES: figures/minimax3.eps (Tue Nov 3 16:23:11 2009).** The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

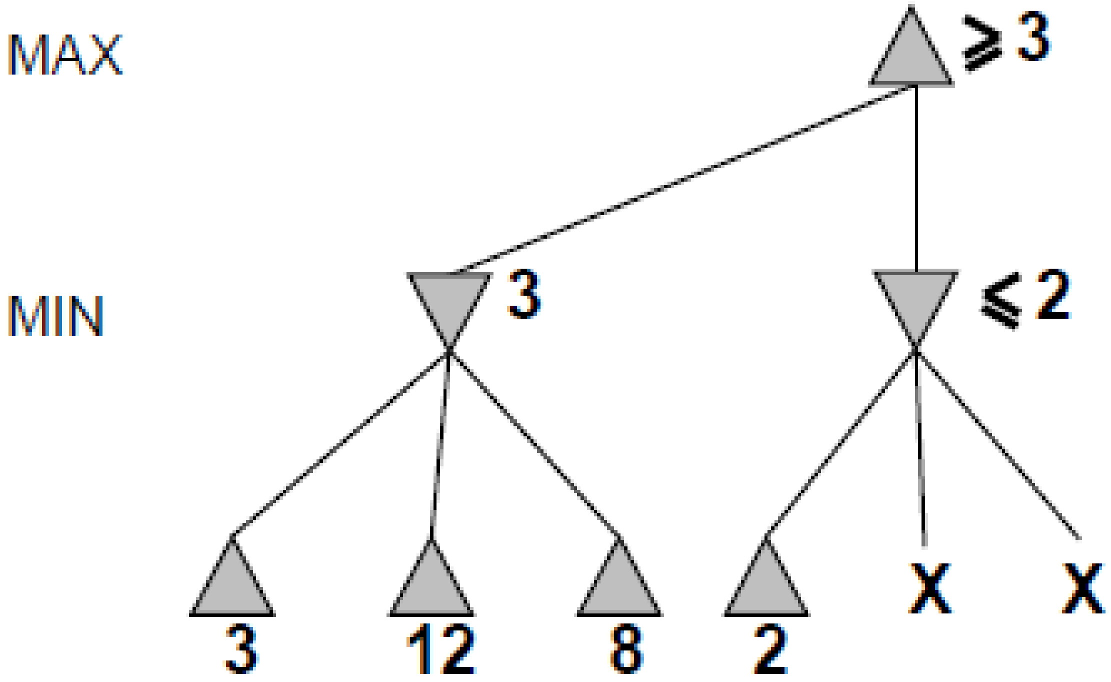
Minimax properties

- Complete?
  - Only if tree is finite
    - Note: A finite strategy can exist for an infinite tree!
- Optimal?
  - Yes, against an optimal opponent! Otherwise, hmmmm
- Time Complexity?
  - O(b^m)
- Space Complexity?
  - O(bm)
- Chess:
  - b ~= 35, m ~= 100 for reasonable games
  - Exact solution still completely infeasible

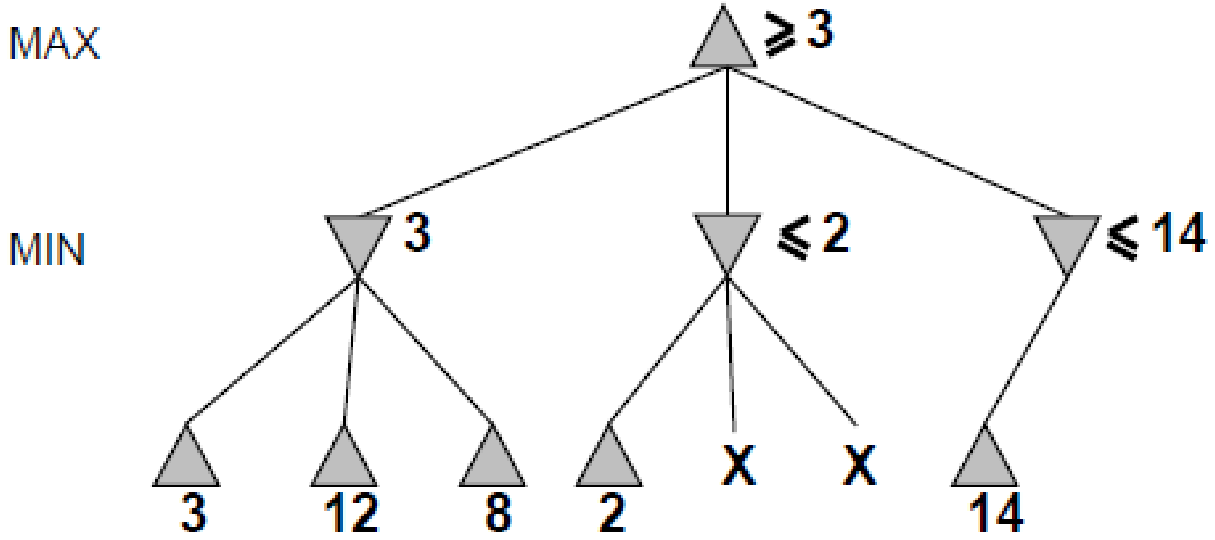
Alpha-beta pruning



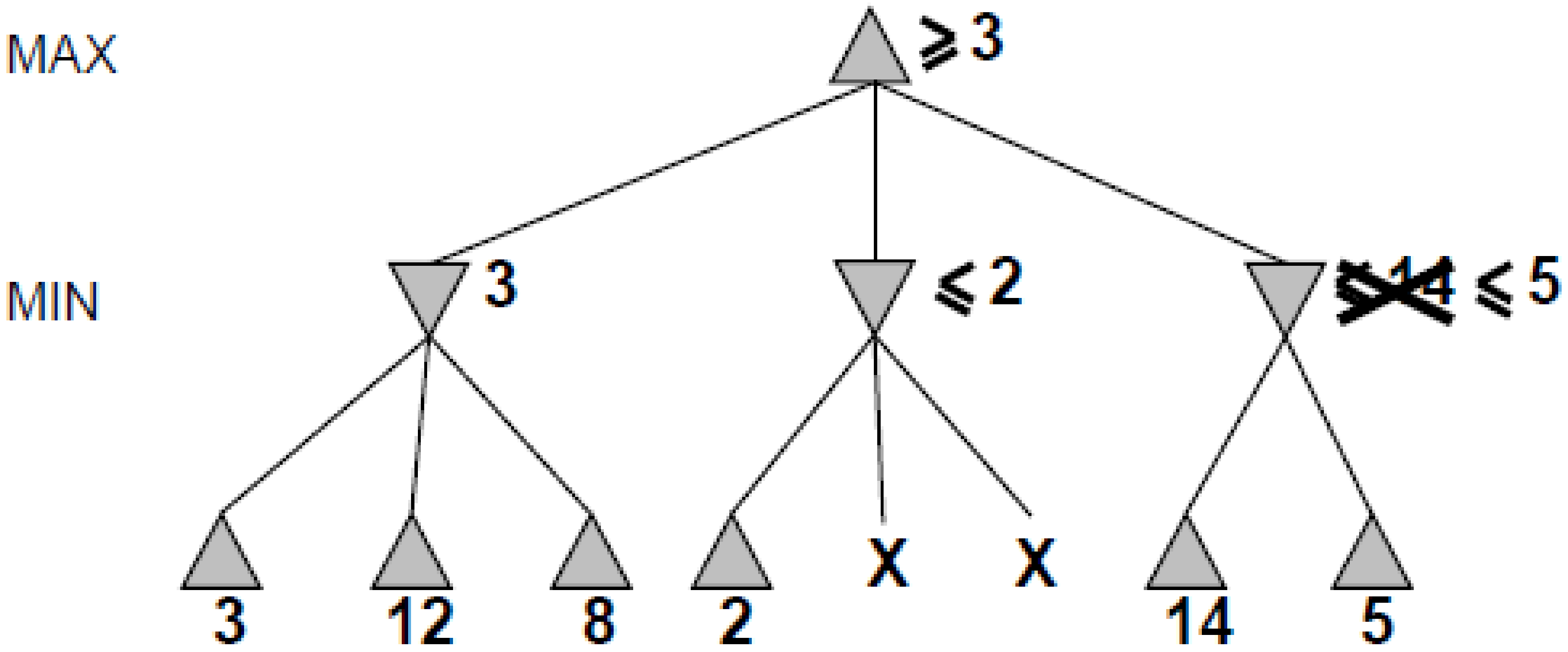
Alpha-beta



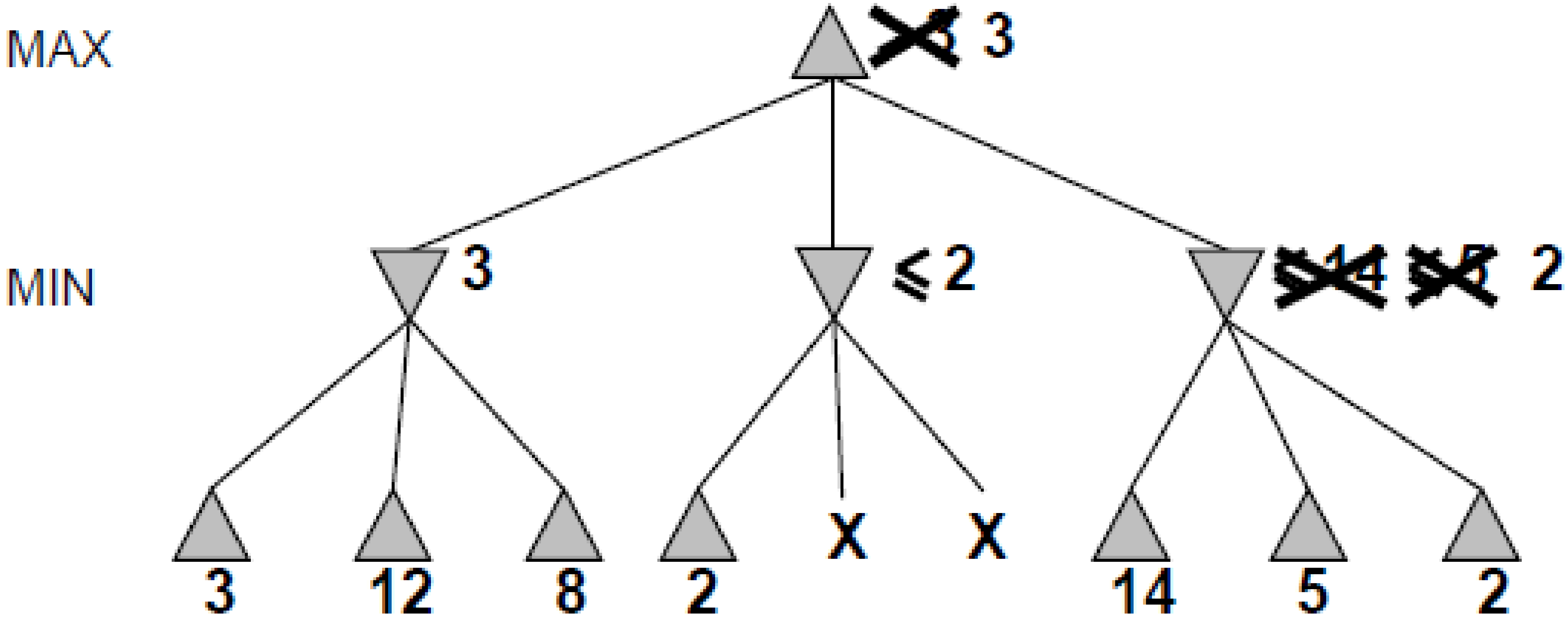
Alpha-beta



Alpha-beta



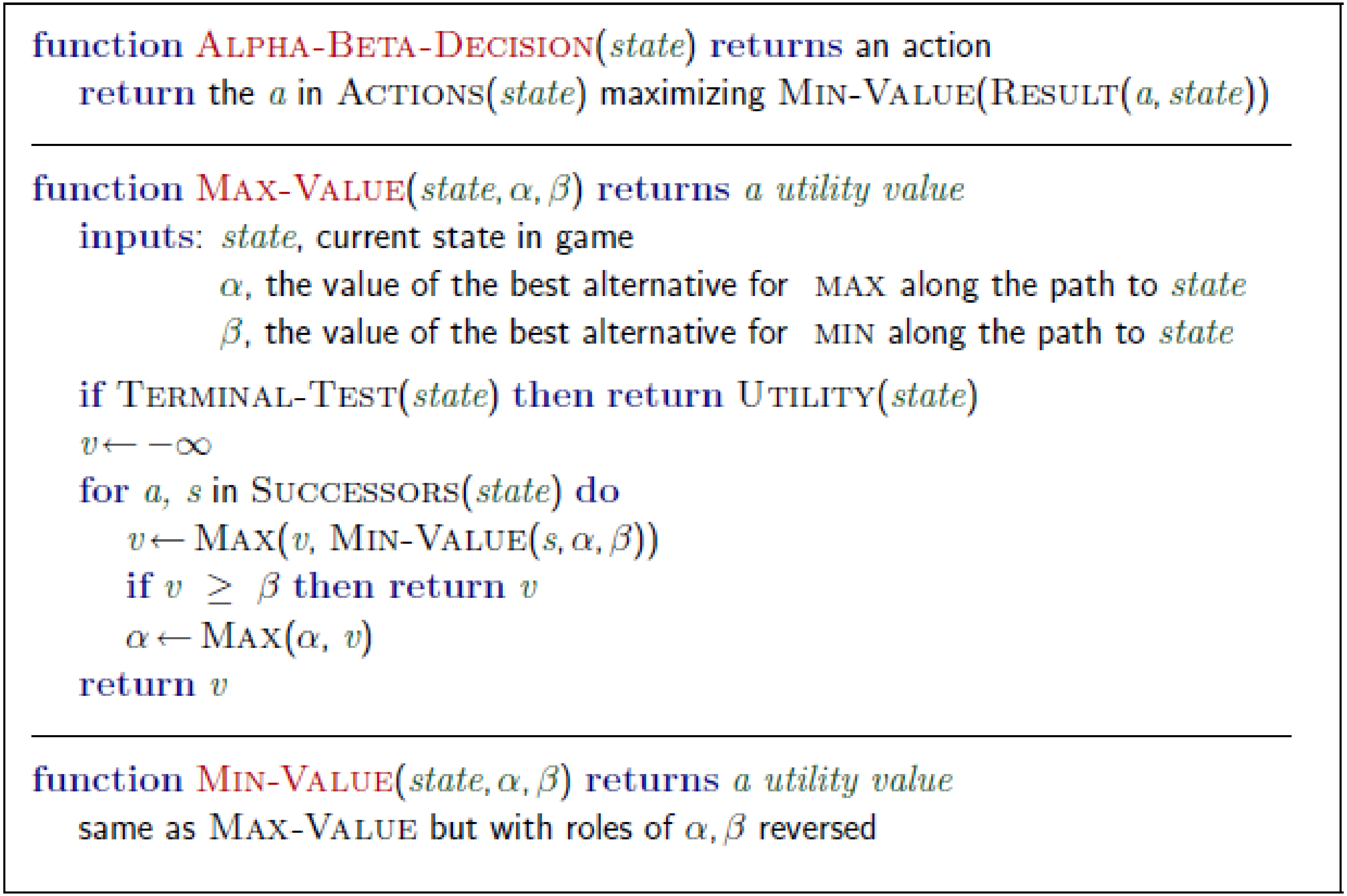
Alpha-beta



Alpha-beta

- Alpha is the best value (for Max) found so far at any choice point along the path for Max
  - Best means highest
  - If utility v is worse than alpha, max will avoid it
- Beta is the best value (for Min) found so far at any choice point along the path for Min
  - Best means lowest
  - If utility v is larger than beta, min will avoid it

Alpha-beta algorithm



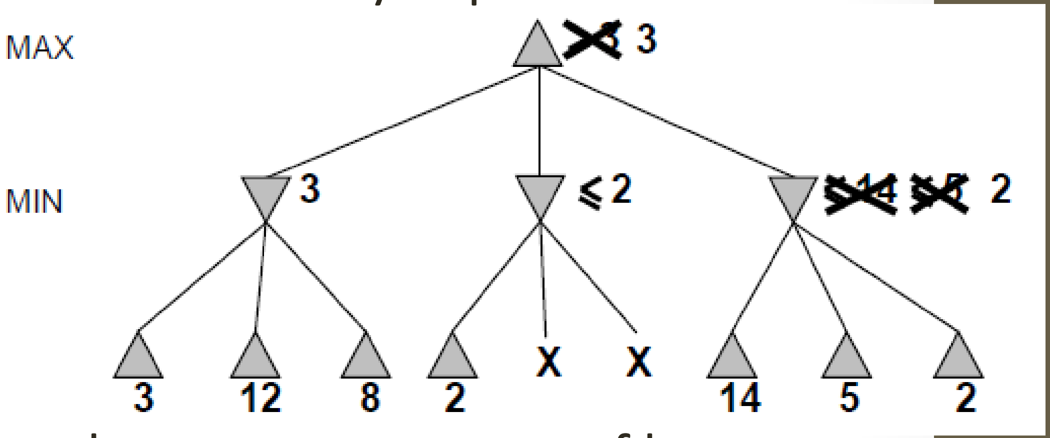
```
function ALPHA-BETA-DECISION($state$) returns an action
    return the $a$ in ACTIONS($state$) maximizing MIN-VALUE(RESULT($a, state$))

function MAX-VALUE($state, \alpha, \beta$) returns a utility value
    inputs: $state$, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to $state$
            $\beta$, 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($s, \alpha, \beta$))
        if $v \geq \beta$ then return $v$
        $\alpha \leftarrow$ MAX($\alpha$, $v$)
    return $v$

function MIN-VALUE($state, \alpha, \beta$) returns a utility value
    same as MAX-VALUE but with roles of $\alpha, \beta$ reversed
```

Alpha beta example

- Minimax(root)
  - = max (min (3, 12, 8), min(2, x, y), min (14, 5, 2))
  - = max(3, min(2, x, y), 2)
  - = max(3, aValue <= 2, 2)
  - = 3

Alpha-beta pruning analysis

- Alpha-beta pruning can reduce the effective branching factor
- Alpha-beta pruning's effectiveness is heavily dependent on **MOVE ORDERING**
- **14, 5, 2 versus 2, 5, 14**
- **If** we can order moves well
- O($b^{\frac{m}{2}}$)
- Which is O(($b^{1/2}$).^{m}
- Effective branching factor then become square root of b
- For chess this is huge → from 35 to 6
- Alpha-beta can solve a tree twice as deep as minimax in the same amount of time!
  - Chess: Try captures first, then threats, then forward moves, then backward moves comes close to b = 12



Imperfect information

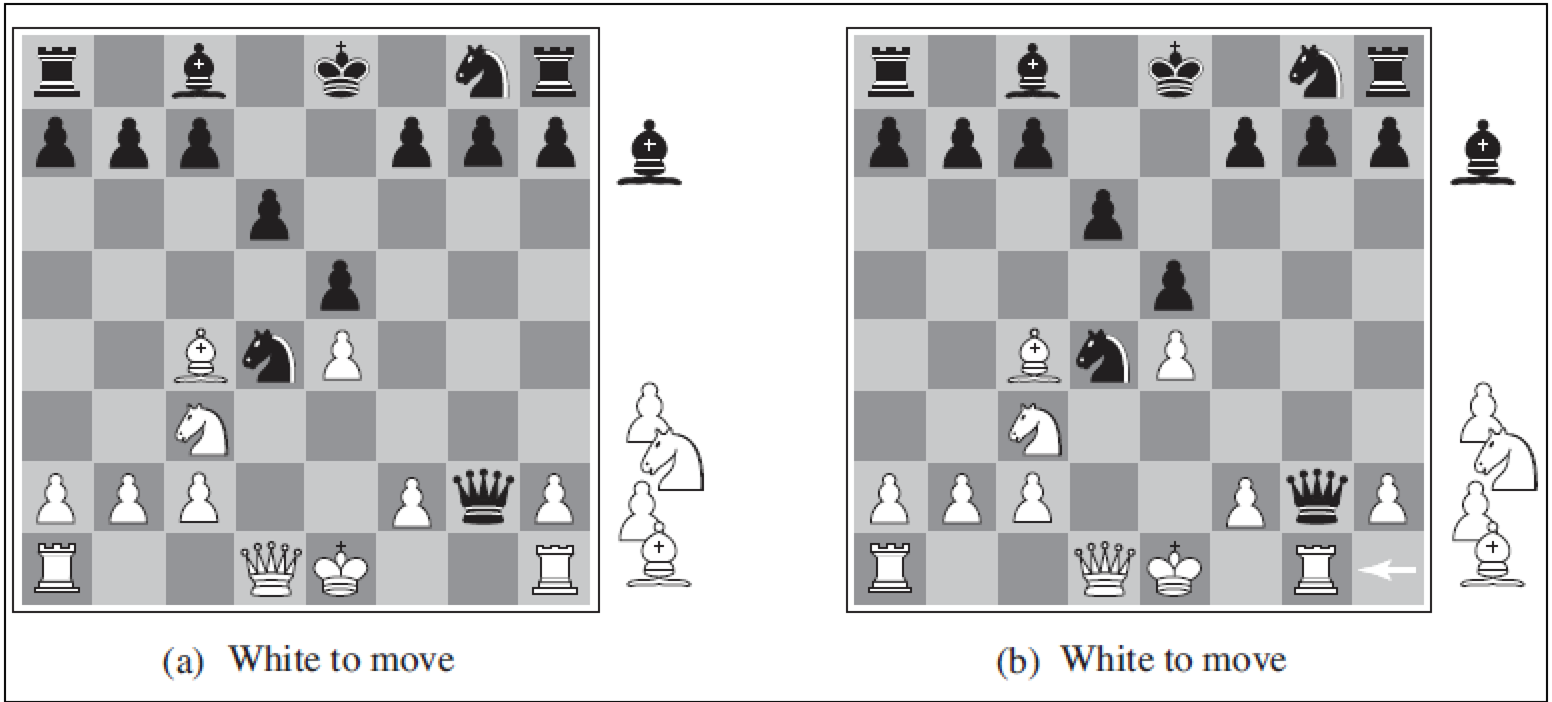
- You still cannot reach all leaves of the chess search tree!
- What can we do?
    - Go as deep as you can, then
    - Utility Value = Evaluate(Current Board)
    - Proposed in 1950 by Claude Shannon

- Apply an **evaluation function** to non-terminal nodes
- Use a **cutoff test** to decide when to stop expanding nodes and apply the evaluation function

Evaluation function

- Must order nodes in the same way as the utility function
  - Wins > Draws > Losses
- Fast
  - Otherwise it is better to search deeper and get more information
- For non-terminal states, high evaluations should mean higher probability of winning
  - Chess is not a chancy game
  - But computational limitations make eval function chancy!

Which is better?



(a) White to move

(b) White to move

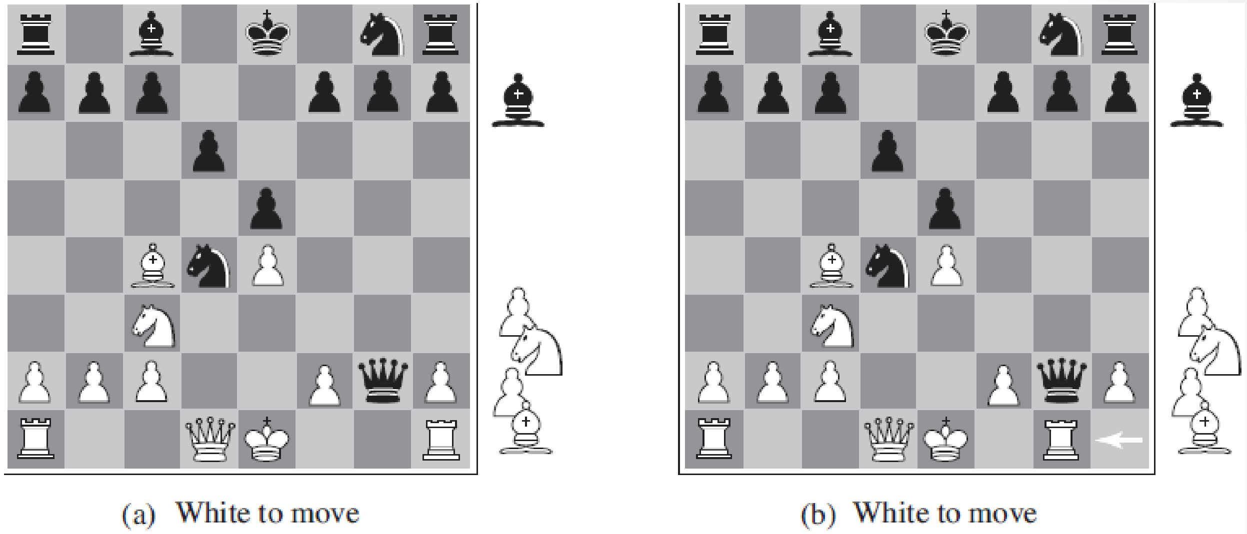
Figure 5.8 **FILES: figures/chess-evaluation3.eps (Tue Nov 3 16:22:33 2009).** Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

Evaluation functions

- A function of board features
  - Use proportions of board-states with winning, losing, and drawing states to compute probabilities.
    - 72% winning (1.0)
    - 20% draws (0.0)
    - 8% losses (0.5)
    - Then: evalFunction(board state) = (0.72 * 1) + (0.2 * 0) + (0.08 * 0.5)
  - Use a **weighted linear sum** of board features (Can also use non-linear f)
    - Chess book: pawn = 1, bishop/knight = 3, rook = 5, queen = 9
    - Good pawn structure = A, king safety = B
    - evalFunction(board state) = w_1* pawns + w_2 * bishops + w_3 * knight + w_4 * rook + … + w_n * good pawn structure + ….
  - All this information for chess comes from centuries of human expertise
  - For new games?

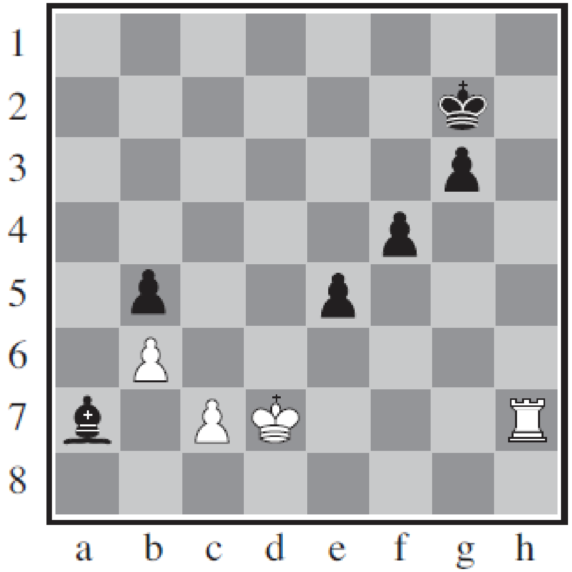
When do we cutoff search

- Quiescence


(a) White to move

(b) White to move

- Horizon effect and singular extension



Forward pruning

- Beam search
- ProbCut – learn from experience to reduce the chance that good moves will be pruned
  - Like alpha-beta but prunes nodes that are probably outside the current alpha-beta window
  - Othello

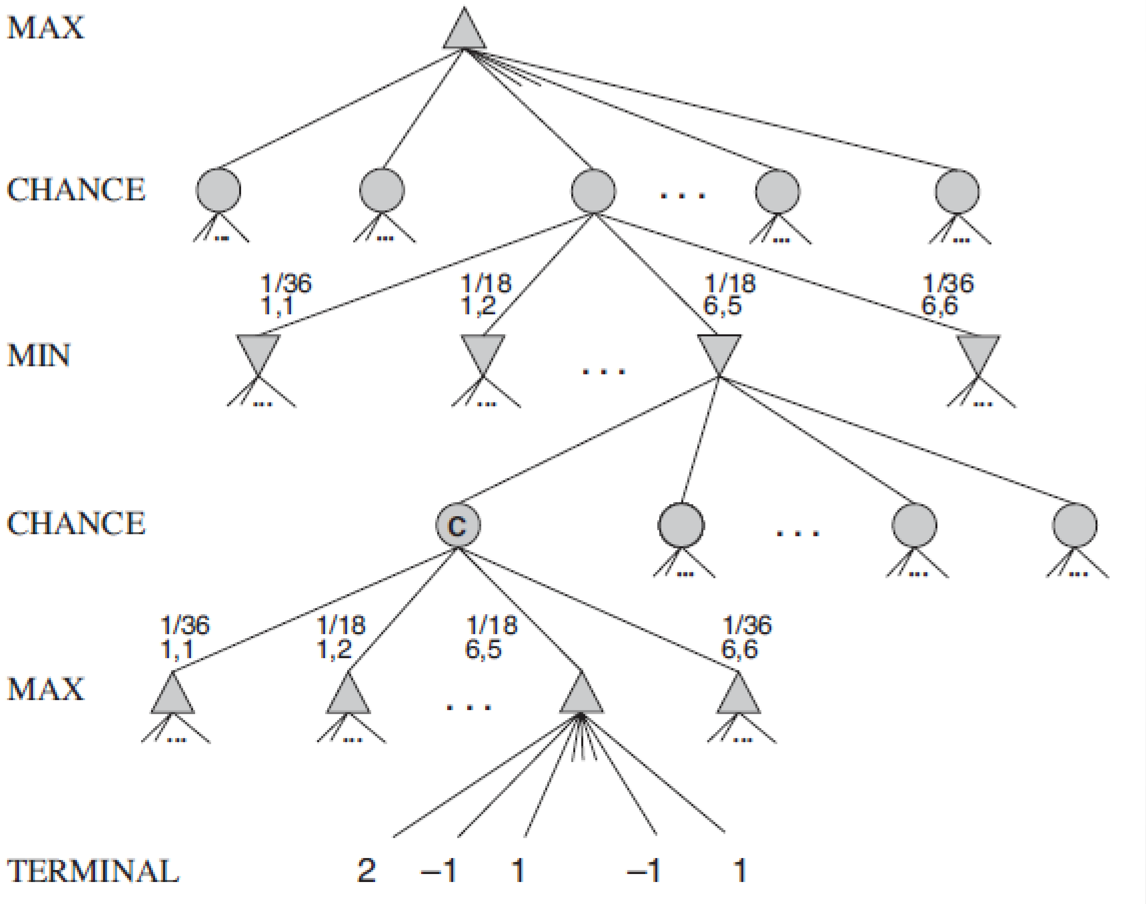
- Combine all these techniques plus

Table lookups

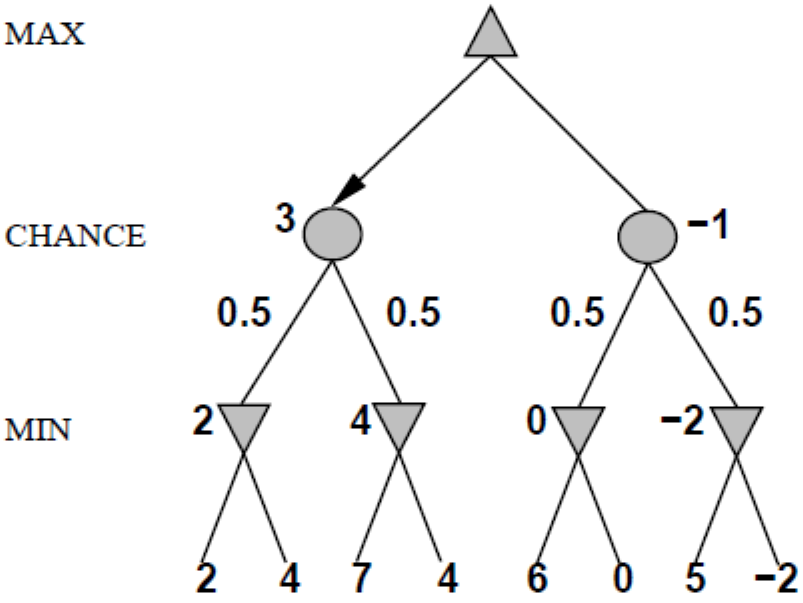
- Chess
  - Openings (perhaps upto 10 moves)
  - Endings (5, 6 pieces left)
    - King-Rook versus King (KRK)
    - King-Bishop-Knight versus King (KBNK)
- Checkers
  - Is solved!

Stochastic Games

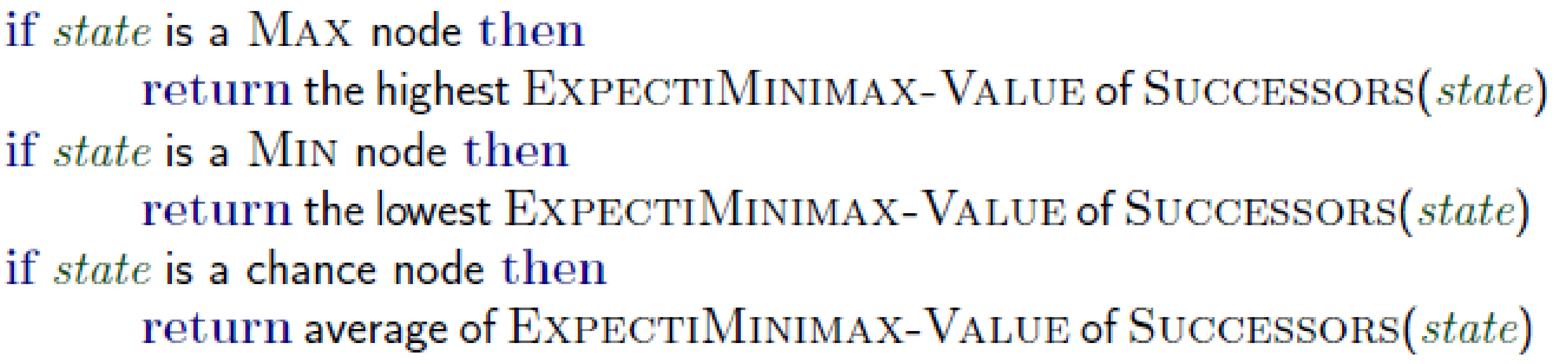
- Chance is involved (Backgammon, Dominoes, …)
- Increases depth if modeled like:



Simple example (coin flipping)

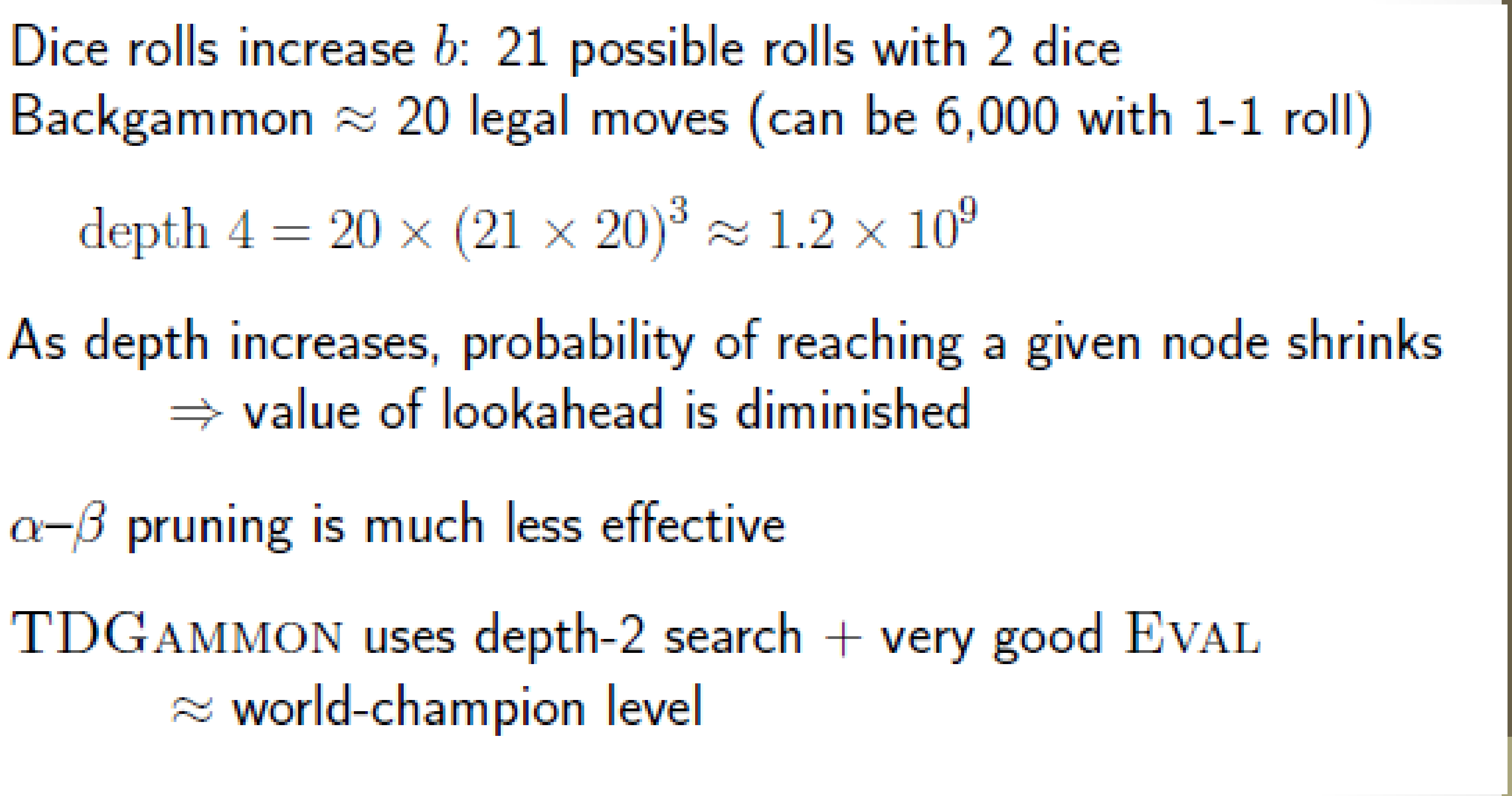


Expected value minimax



```
if state is a Max node then
    return the highest ExpectiMinimax-Value of Successors(state)
if state is a Min node then
    return the lowest ExpectiMinimax-Value of Successors(state)
if state is a chance node then
    return average of ExpectiMinimax-Value of Successors(state)
```

Backgammon



Dice rolls increase b: 21 possible rolls with 2 dice
Backgammon $\approx$ 20 legal moves (can be 6,000 with 1-1 roll)

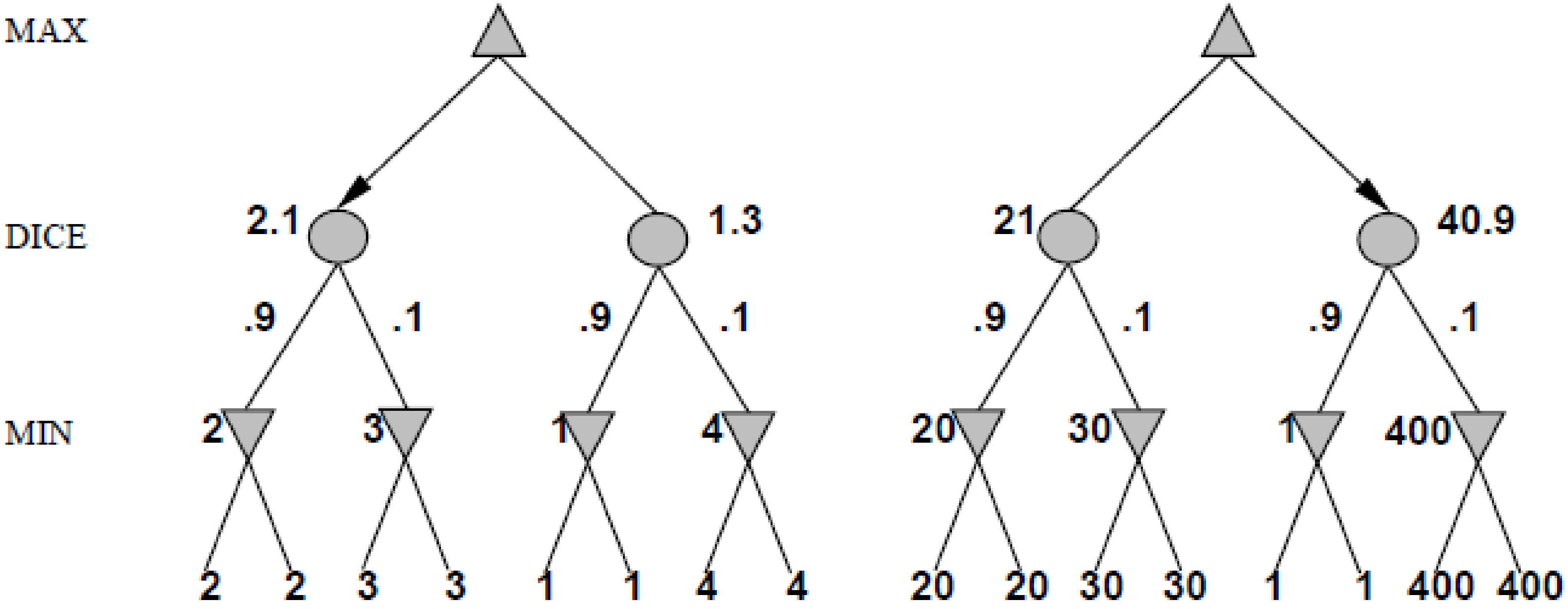
$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

As depth increases, probability of reaching a given node shrinks
$\Rightarrow$ value of lookahead is diminished

α–β pruning is much less effective

TDGammon uses depth-2 search + very good Eval
$\approx$ world-champion level

With chance, exact values matter

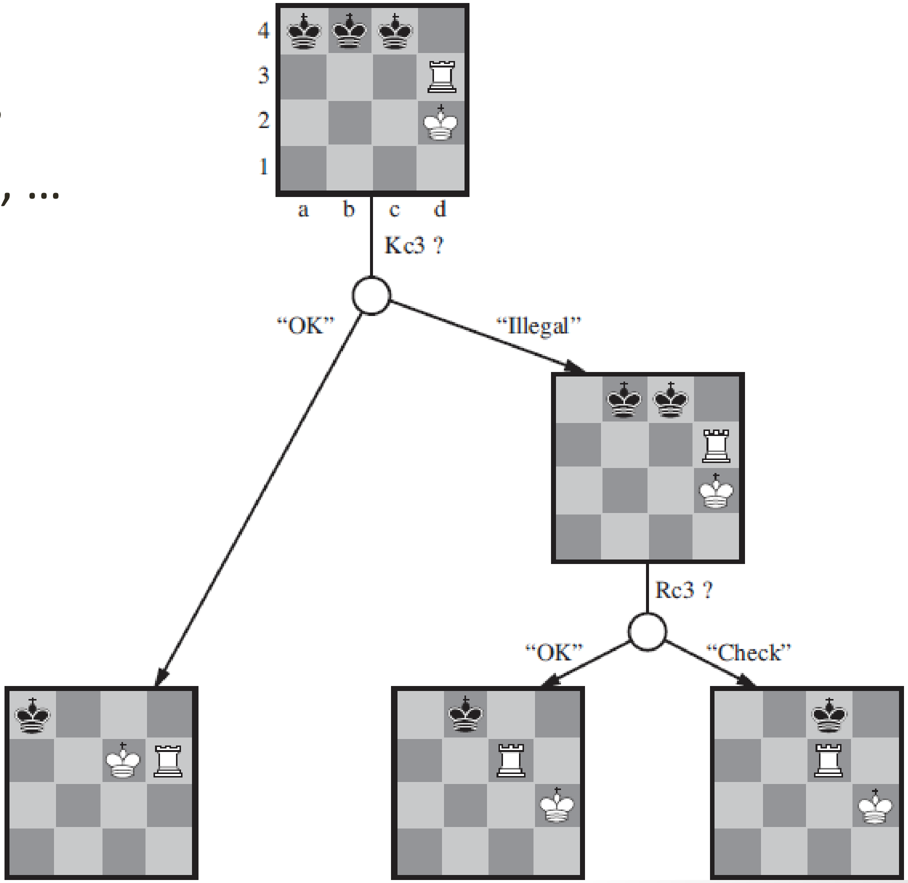


Behaviour is preserved only by positive linear transformation of EVAL

Hence EVAL should be proportional to the expected payoff

Fog of War

- Use belief states to represent the set of states you could be in given all the percepts so far
- Kriegspiel
  - You can only see your pieces
  - Judge says: Ok, illegal, check, …



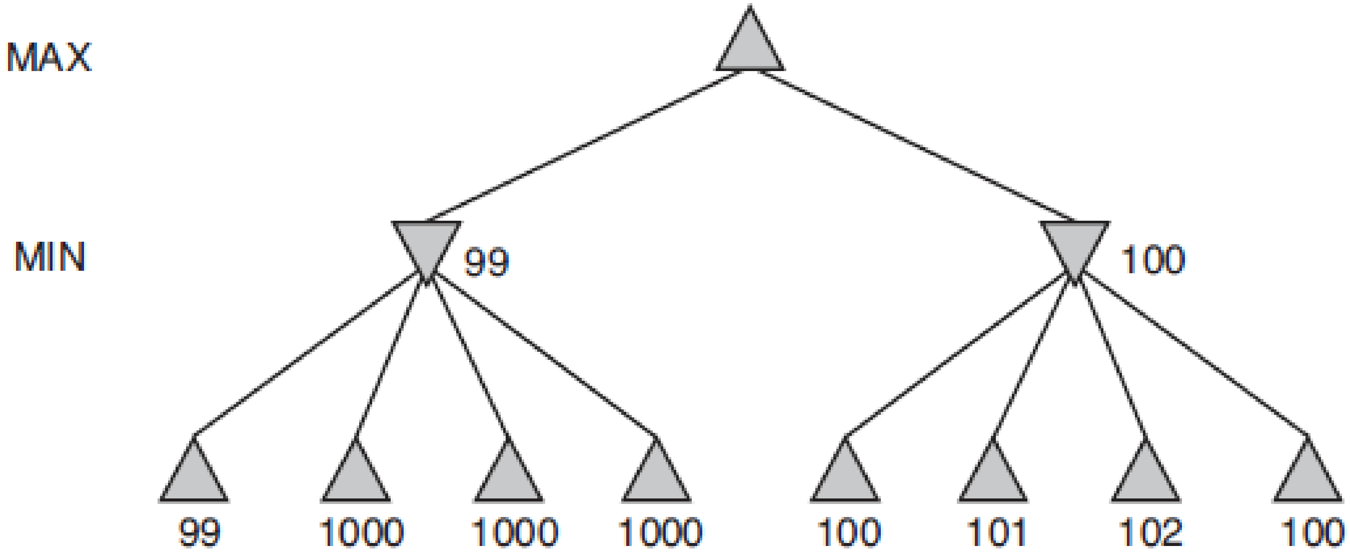
What is a belief state?

Card Games

- Consider all possible deals of a deck of cards, solve each deal as a fully observable game, then choose best move averaged over all deals
- Computationally infeasible but:
  - Let us do Monte Carlo approximation
  - Deal a 100 deals, a 1000 deals, … whatever is computational feasible
  - Choose best outcome move

- Read section 5.7 – state of the art game programs

Errors in evaluation functions!



Summary

- Games are fun to work on
- They give insight on several important issues in AI
    - Perfection is unattainable → approximate
    - Think about what to think about
    - Uncertainty constrains assignment of values to states
    - Optimal decisions depend on information state, not real state

- Games are to AI as grand prix racing is to automobile design

Searching with Nondeterministic actions

- In the past, we knew what state we were in and a solution was a path from root to goal.
- Now, how do you find paths when the environment is partially observable or non-deterministic or both and you don't know what state you are in?

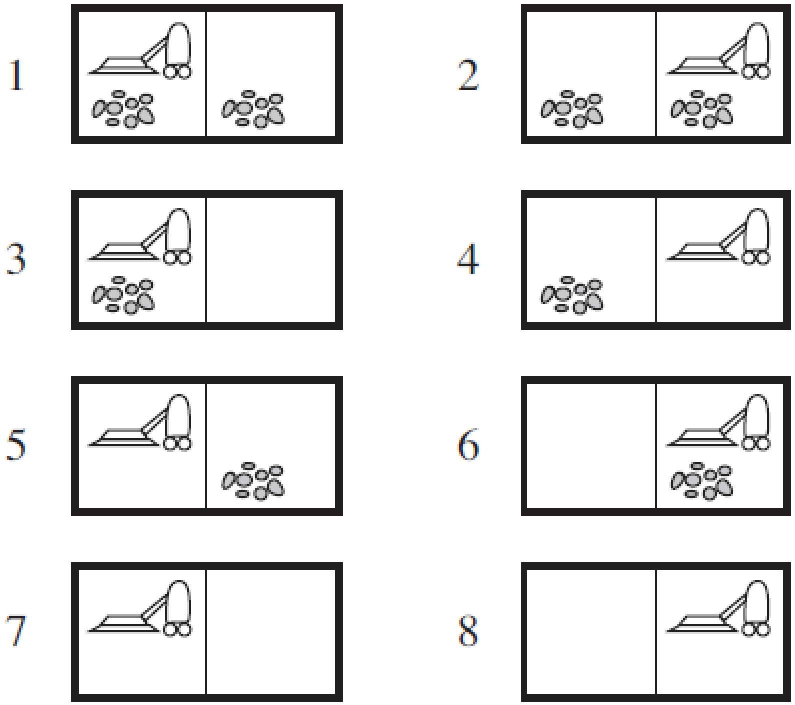
- You make contingency plans
  - If in state x then y
- You use percepts
  - I did an action with a non-deterministic result, percepts can tell me which result actually occurred

Erratic Vacuum cleaners

Suck

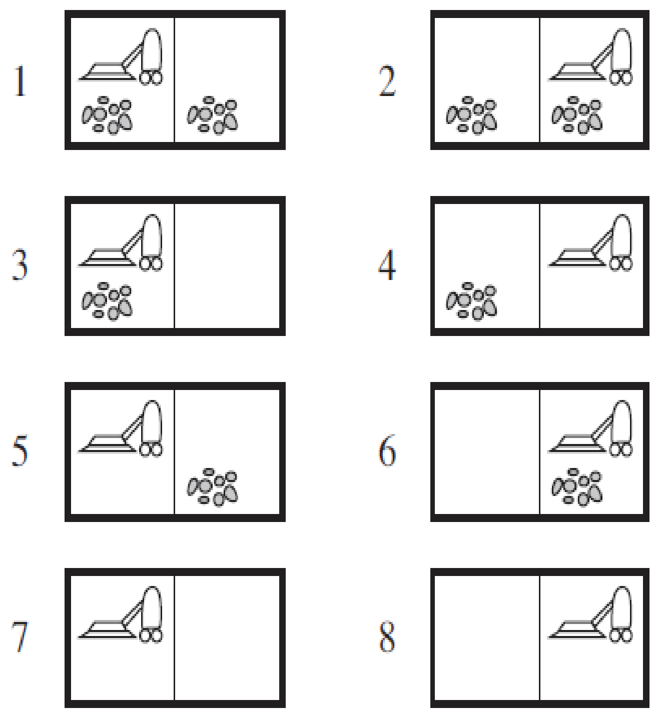
- Sometimes cleans adjacent square ☺
- Sometimes deposits dirt in current square ☹

- Transition Model
  - Result → Results
  - Suck({1}) → {5, 7}

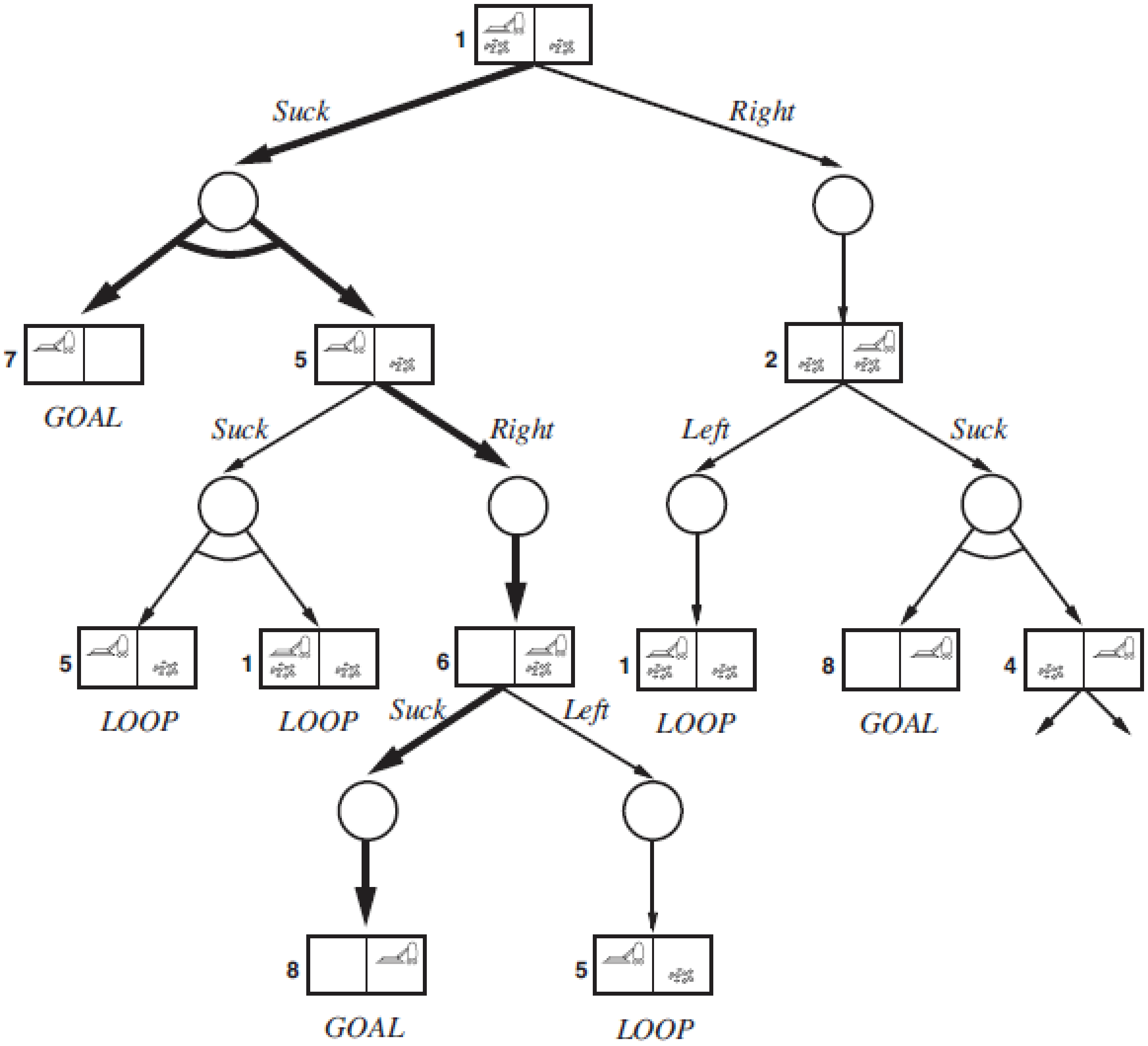


Erratic Vacuum cleaners

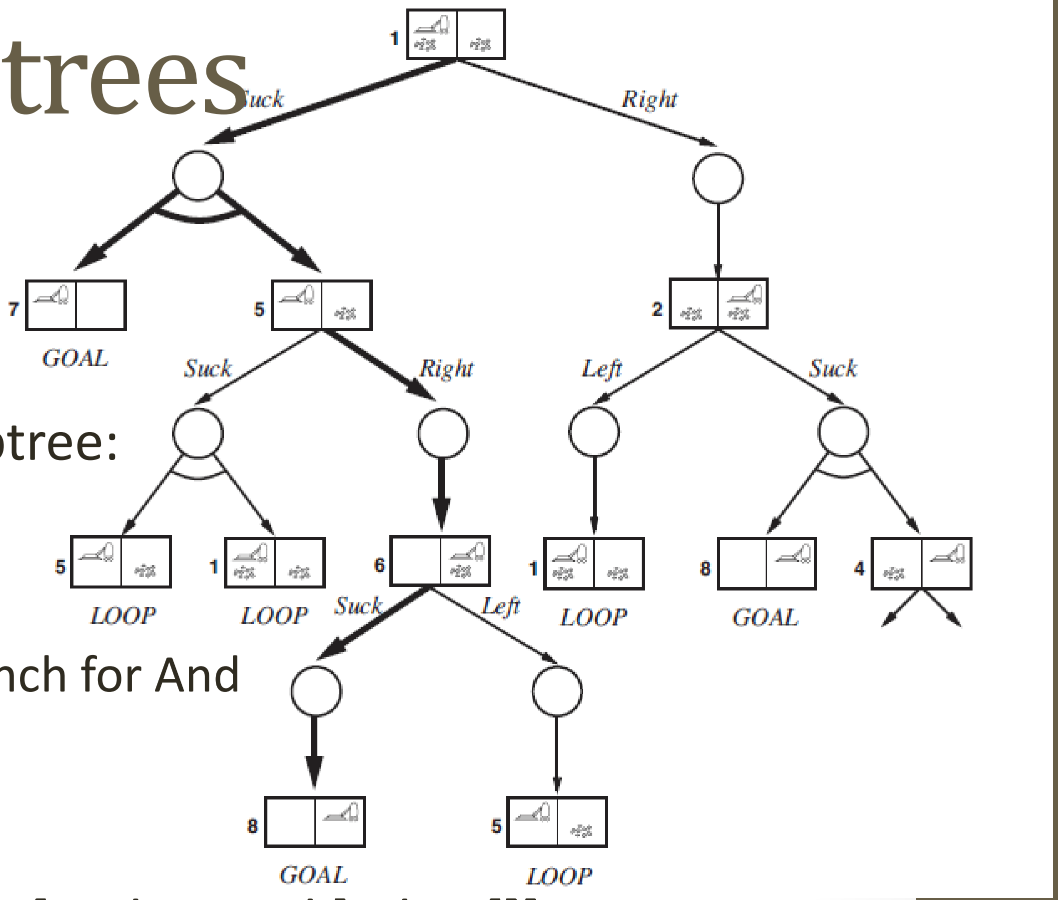
- Sometimes cleans adjacent square ☺
- Sometimes deposits dirt in current square ☹
- Solution
  - [Suck, if State == 5 then [Right, Suck] else []]
- Solutions are trees! Not sequences
- Solutions are nested if-then-else
- Many problems in the real world are of this type because exact prediction is impossible
  - Keep your eyes open when you drive/walk/fly



And-Or search trees

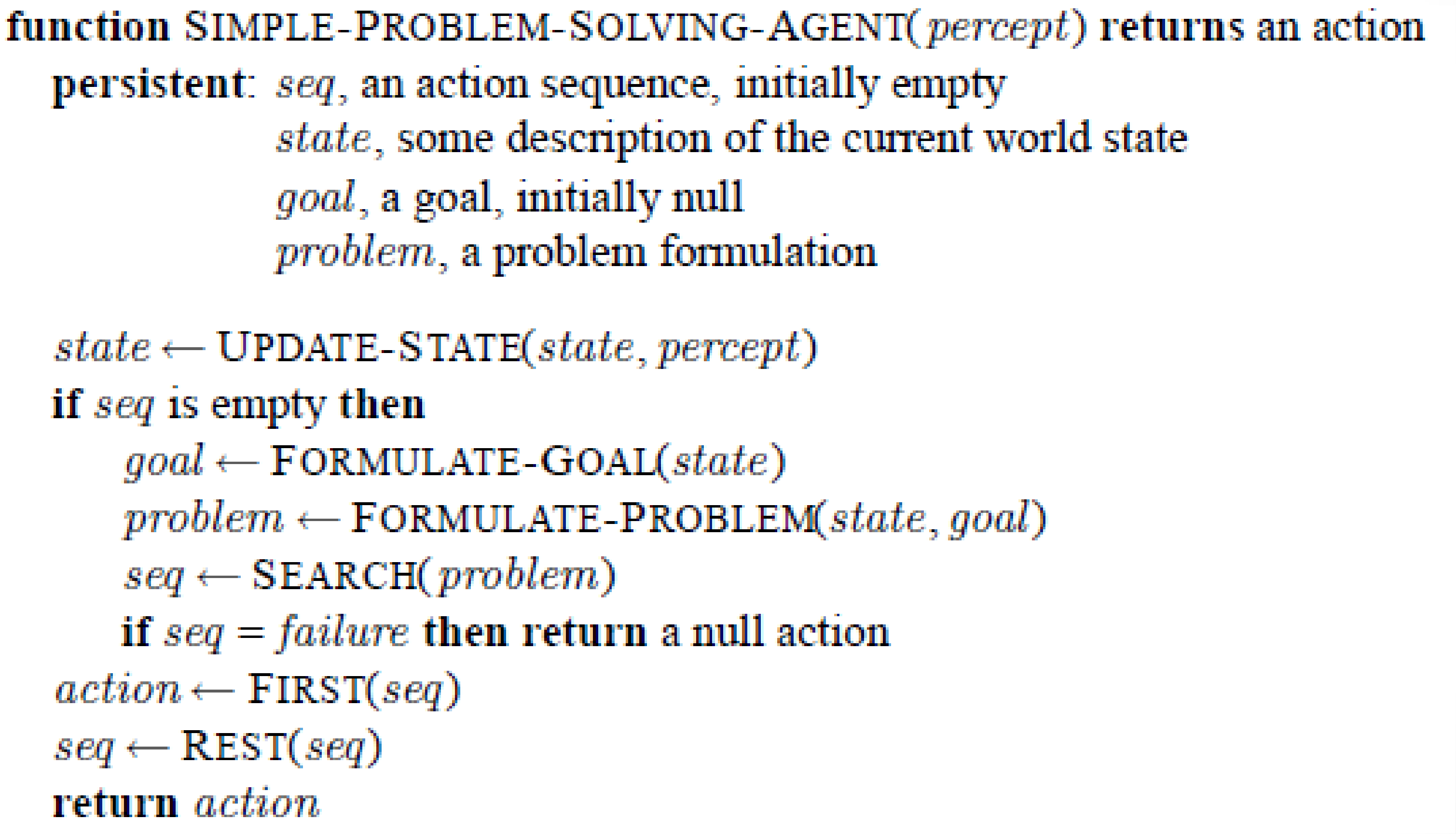


And-Or search trees



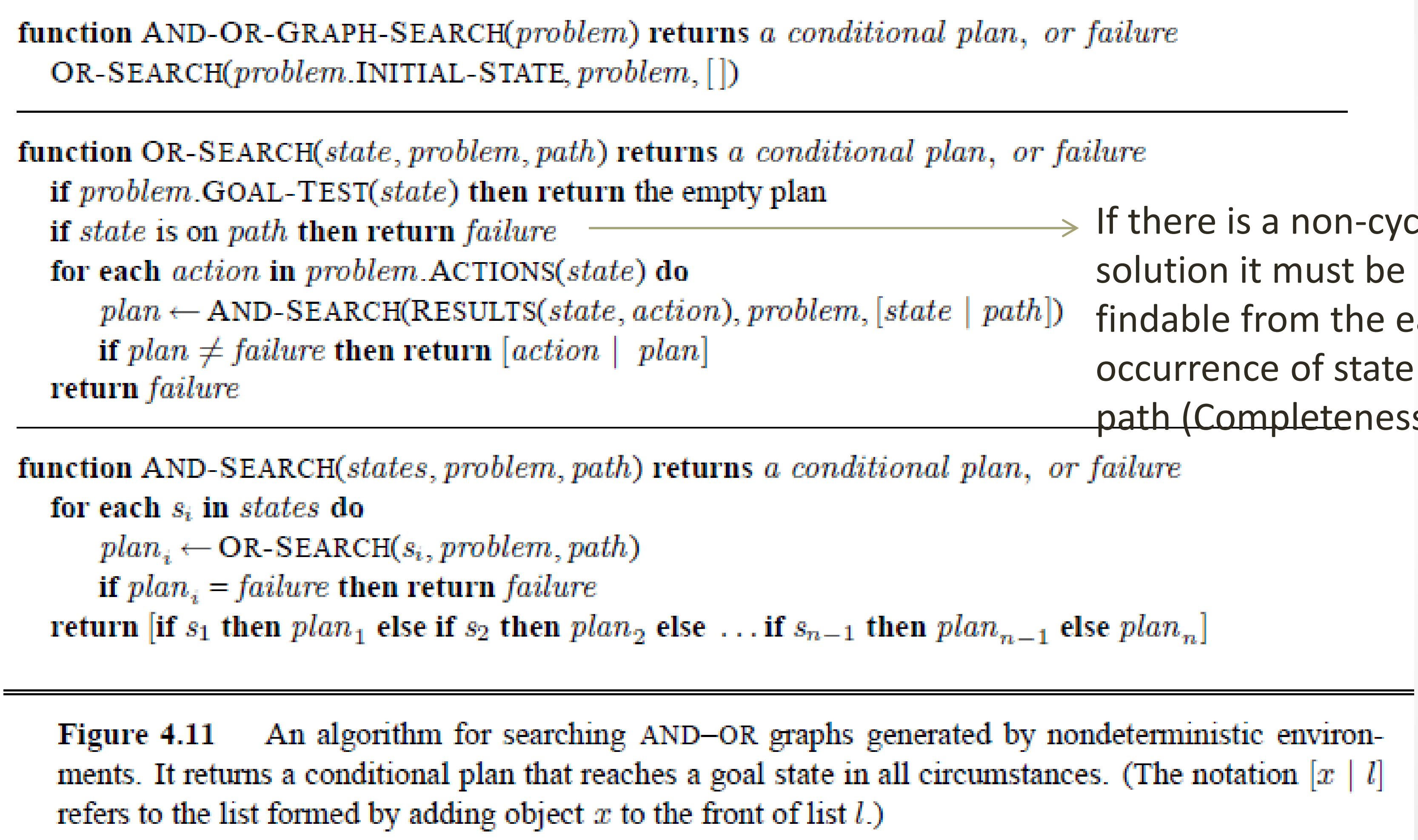
- Or nodes (Suck or Right)
- And node {5, 7} → Results
- And-Or tree solution is a subtree:
  - Goal node at every leaf
  - One action at each Or-node
  - Includes every outcome branch for And
- Same as:
  - [Suck, if State == 5 then [Right, Suck] else []]

Remember the simple problem solving agent?



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    persistent: seq, an action sequence, initially empty
                state, some description of the current world state
                goal, a goal, initially null
                problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
        if seq = failure then return a null action
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

And-Or problem solver



```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
   OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
   if problem.GOAL-TEST(state) then return the empty plan
   if state is on path then return failure
   for each action in problem.ACTIONS(state) do
       plan $\leftarrow$ AND-SEARCH(RESULTS(state, action), problem, [state | path])
       if plan $\neq$ failure then return [action | plan]
   return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
   for each $s_i$ in states do
       $plan_i$ $\leftarrow$ OR-SEARCH($s_i$, problem, path)
       if $plan_i$ = failure then return failure
   return [if $s_1$ then $plan_1$ else if $s_2$ then $plan_2$ else ... if $s_{n-1}$ then $plan_{n-1}$ else $plan_n$]
```

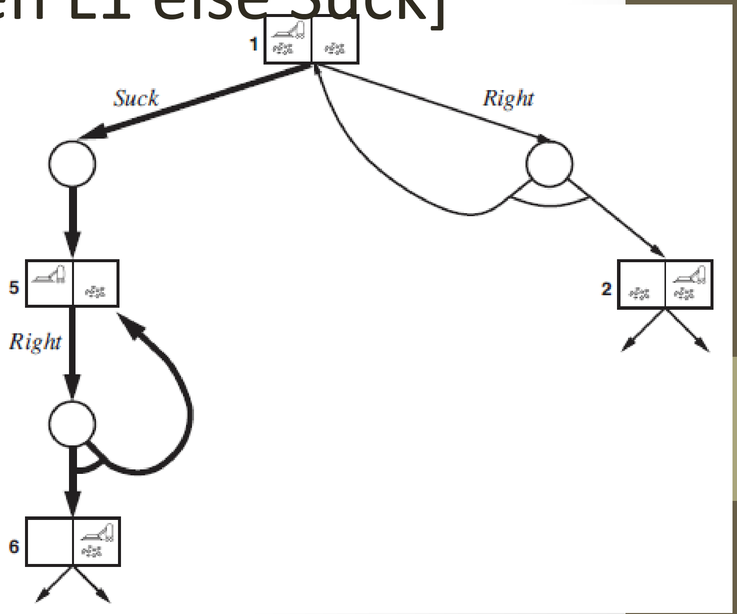
Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

If there is a non-cyclic solution it must be findable from the earlier occurrence of state in path (Completeness)

Recursive, breadth-first. Can use breadth-first, …

Slippery vacuum worlds

- Movement actions sometimes fail and leave you in the same location
- No acyclic solutions!
- Labels enable cycles
- [Suck, L1: Right, if State == 5 then L1 else Suck]



Search

- Problem solving by searching for a solution in a space of possible solutions
- Uninformed versus Informed search
- Atomic representation of state
- Solutions are fixed sequences of actions