Game Playing

Search the action space of 2 players

Russell & Norvig Chapter 6

Bratko Chapter 22

University of Amsterdam

Game Playing

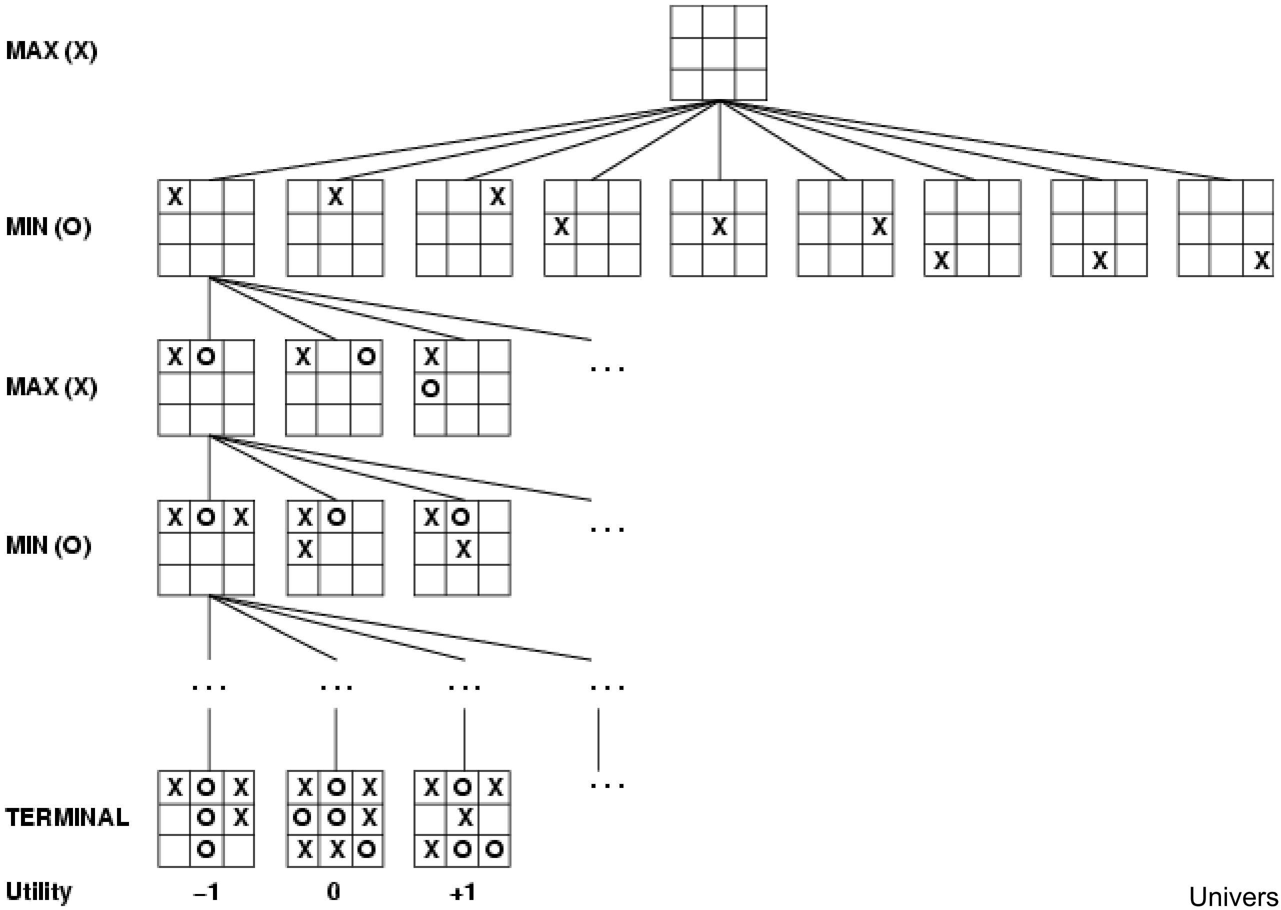
- ‘Games contribute to AI like Formula 1 racing contributes to automobile design.’
- ‘Games, like the real world, require the ability to make *some* decision, even when the *optimal* decision is infeasible.’
- ‘Games penalize inefficiency severely’.

Games vs. search problems

- "Unpredictable" opponent → specifying a move for every possible opponent reply
- Time limits → unlikely to find *the* solution, must approximate *a* solution

University of Amsterdam

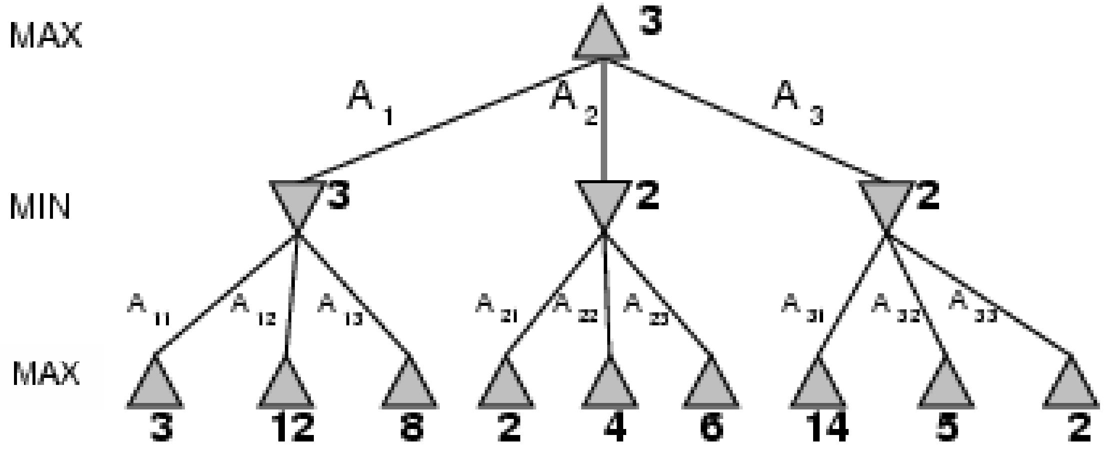
Game tree of tic-tac-toe
(2-player, deterministic, turn-taking, zero sum)



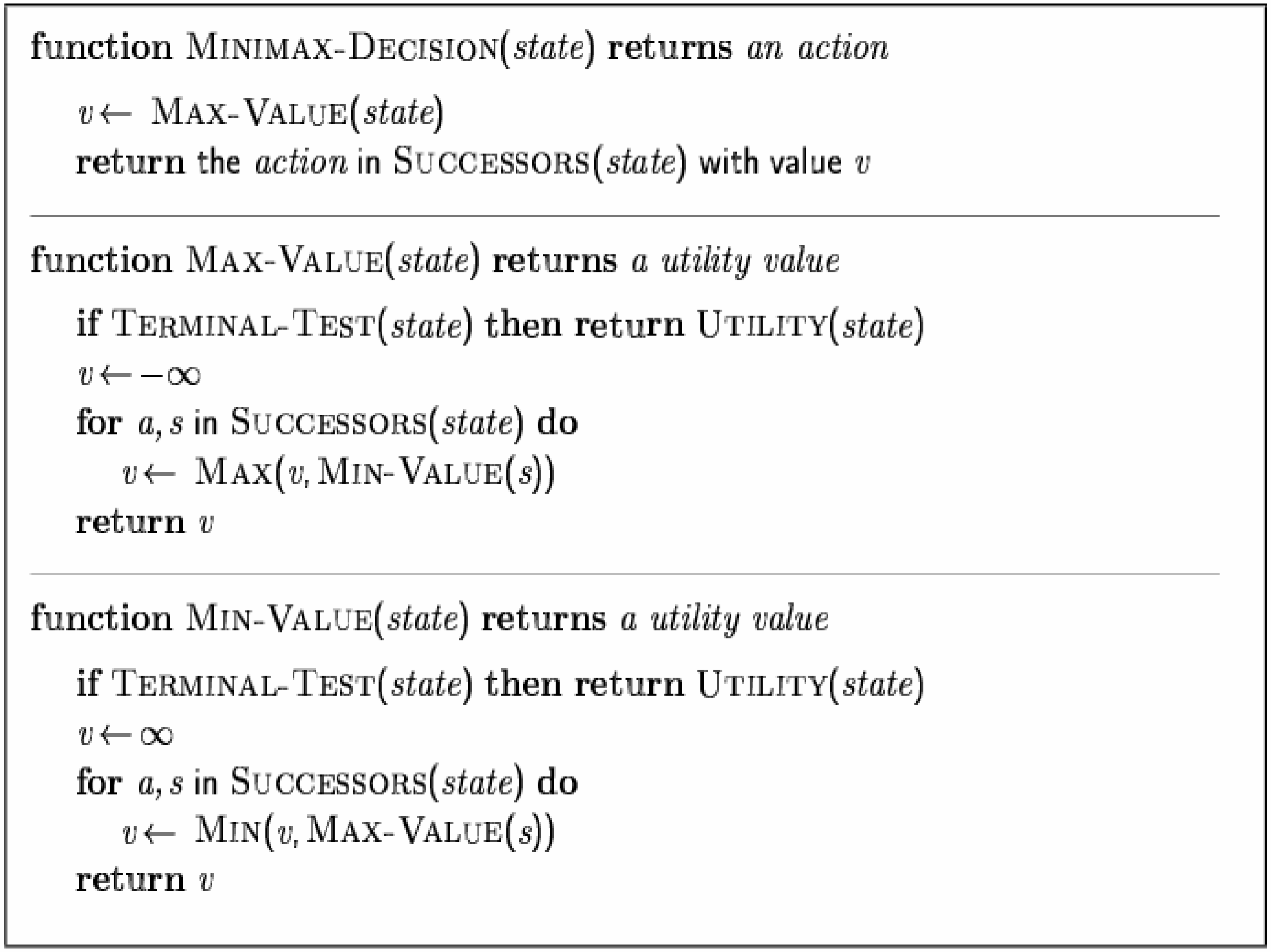
University of Amsterdam

Minimax

- Perfect play for deterministic games
- Idea: choose move to position with highest minimax value = best achievable payoff against perfect playing opponent
- E.g., 2-ply game:



Minimax algorithm



```
function MINIMAX-DECISION(state) returns an action

   v ← MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value

   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s))
   return v

function MIN-VALUE(state) returns a utility value

   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s))
   return v
```

Minimax prolog implementation

```
minimax( Pos, BestSucc, Val)  :-
  moves( Pos, PosList), !,                 % Legal moves in Pos
  best( PosList, BestSucc, Val)
   ;
  staticval( Pos, Val).                    % Terminal Pos has no successors

best( [ Pos], Pos, Val)  :-
  minimax( Pos, _, Val), !.

best( [Pos1 | PosList], BestPos, BestVal)  :-
  minimax( Pos1, _, Val1),
  best( PosList, Pos2, Val2),
  betterof( Pos1, Val1, Pos2, Val2, BestPos, BestVal).

betterof( Pos0, Val0, Pos1, Val1, Pos0, Val0)  :-
  min_to_move( Pos0), Val0 > Val1, !       % MAX prefers the greater value
  ;
  max_to_move( Pos0), Val0 < Val1, !.      % MIN prefers the lesser value

betterof( Pos0, Val0, Pos1, Val1, Pos1, Val1).
% Otherwise Pos1 better than Pos0
```

Game interface

- Maarten van Soomeren’s implementation is based on Bratko’s implementation: fig22_3.txt
- The tic-tac-toe game interface is based on 4 relations:

```
moves( Pos, PosList)     % Legal moves in Pos, fails when Pos is terminal
staticval( Pos, Val).    % value of a Terminal node (utility function)
min_to_move( Pos )       % the opponents turn
max_to_move( Pos )       % our turn
```

- Bratko’s terminal position are win (+1) or loose (-1), Maarten’s terminal positions are heuristic values

Properties of minimax

- 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

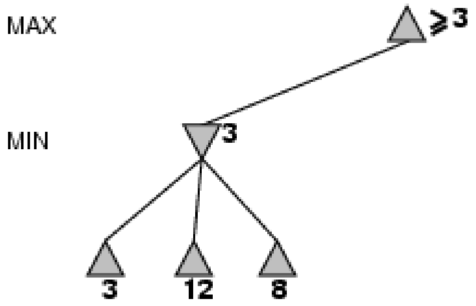
University of Amsterdam

α-β pruning

- Efficient minimaxing
- Idea: once a move is clearly inferior to a previous move, it is not necessary to know *exactly* how much inferior.
- Introduce two bounds:

  Alpha = minimal value the MAX is guaranteed to achieve

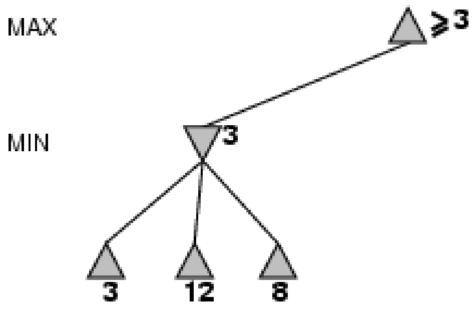
  Beta = maximal value the MAX can hope to achieve
- Example:



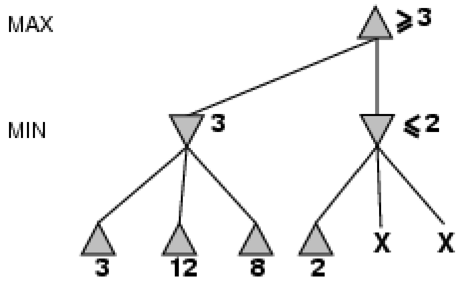
University of Amsterdam

α-β pruning

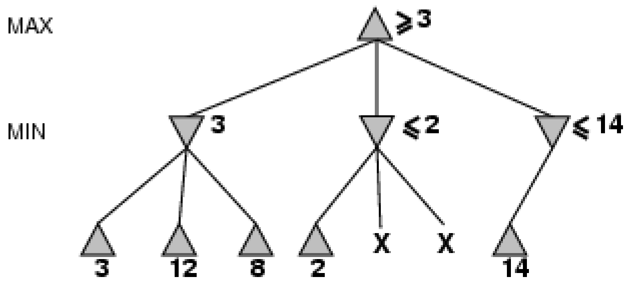
- Example:



Alpha = 3



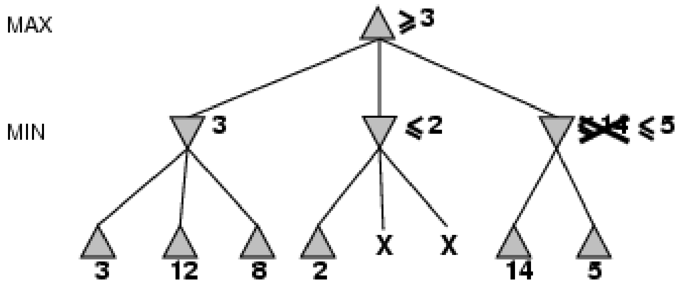
Val < Alpha,
!



Val > Alpha
Newbound(β)

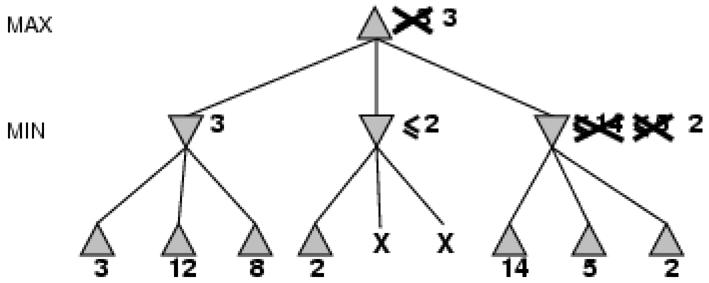
α-β pruning

- Example:



Val > α

Newbound(β)



Val < α

!

Properties of α-β

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

University of Amsterdam

AlphaBeta prolog implementation

```
alphabeta( Pos, Alpha, Beta, GoodPos, Val) :-
  moves( Pos, PosList), !,                  % Legal moves in Pos
  boundedbest( PosList, Alpha, Beta, GoodPos, Val)
   ;
  staticval( Pos, Val).                     % Terminal Pos has no successors

boundedbest( [Pos | PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta( Pos, Alpha, Beta, _, Val),
  goodenough( PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).
…
goodenough( _, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move( Pos), Val > Beta, !        % MAX prefers the greater value
  ;
  max_to_move( Pos), Val < Alpha, !.      % MIN prefers the lesser value

goodenough( PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal)  :-
  newbounds( Alpha, Beta, Pos, Val, NewAlpha, NewBeta),    % Refine bounds
  boundedbest( PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof( Pos, Val, Pos1, Val1, GoodPos, GoodVal).
```

Properties of α-β implementation

- \+ straightforward implementation
- \- It doesn’t answer the solution tree
- \- With the depth-first strategy, it is difficult to control

University of Amsterdam

Prolog assignment

- Download AlphaBeta implementation from Bratko: fig22_5.txt
- Replace in your solution minimax for AlphaBeta. Create test-routines to inspect the performance difference

```
alphabeta( Pos, Alpha, Beta, GoodPos, Val, MaxDepth)
```

Resource usages in chess

Suppose we have 100 secs, explore 10^4 nodes/sec

→ 10^6 nodes per move $\approx 35^{8/2}$

→ α-β reaches depth 8 → human chess player

Needed additional modifications:

- cutoff test:

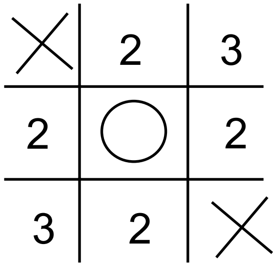
  e.g., depth limit (perhaps add quiescence search)□

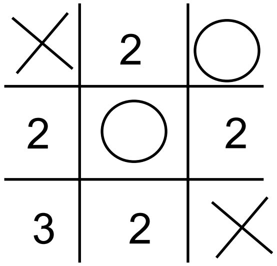
- evaluation function

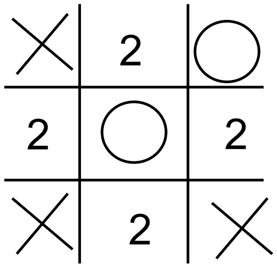
  = estimated desirability of position

University of Amsterdam

Evaluation-functions are quite static







- We need domain knowledge (heuristics)
- At many equivalent quiescence positions, we need long term plans, and we have to stick to them
- An expert system is needed with long term plans

Advantages of separating production rules from inference engine

- *Modularity:* each rule an concise piece of knowledge
- *Incrementability:* new rules can be added independently of other rules
- *Modifiability:* old rules can be changed
- *Transparent*

University of Amsterdam

Production rules

- *If* precondition P *then* Conclusion C
- *If* situation S *then* action A
- *If* conditions C1 and C2 hold *then* Condition C does not hold

University of Amsterdam

Advice Language

Central in Advice Language is an advice table.

Each table is ordered collection of production rules.

When the precondition is forfilled, a list of advices can be tried, in the order specified.

A ‘piece-of-advice’ is the central building block in AL0.

Piece-of-Advice

Extending Situation Calculus:

- Us-move-constraints:
  selects a subset of all legal us-moves
- Them-move-constraints:
  selects a subset of all legal them-moves

Combination of precondition and actions.

University of Amsterdam

Advice Language

Stop criteria:

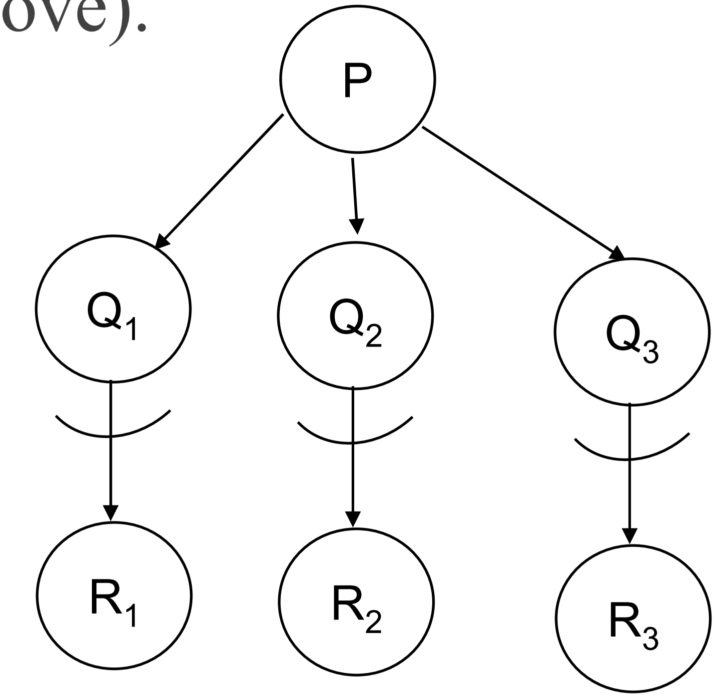
- Better-goal:
  a goal to be achieved
- Holding-goal:
  a goal to be maintained while playing toward the better-goal

University of Amsterdam

The result

Solution trees are implemented with forcing trees:

AND/OR trees where AND-nodes have only one arc (selected us-move).



University of Amsterdam

Prolog assignment

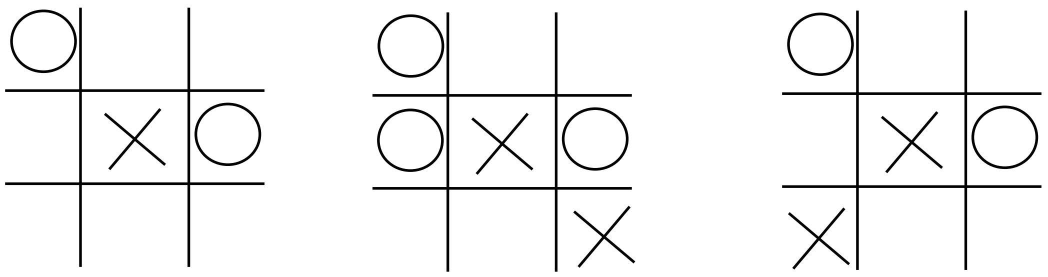
- Select subset of legal moves with Advice Language:

- Download:

http://www.science.uva.nl/~arnoud/education/ZSB/follow_strategy.pl

http://www.science.uva.nl/~arnoud/education/ZSB/advice.pl

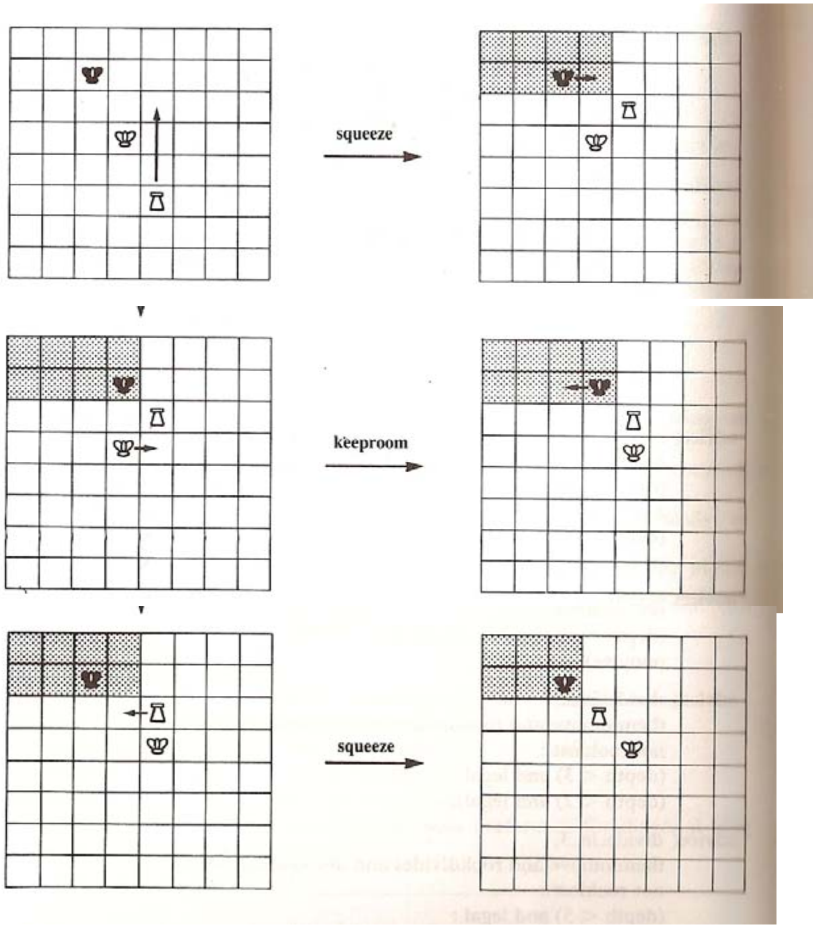
- Test:



Advice Language applied to chess

- Bratko gives a solution for the King and Rook vs King problem
- Advice table consist of two rules:
  - edge_rule (trying mate_in_2)
  - else_rule
- Both rules the following advices in this order:
  - squeeze, approach, keeproom, divide_in_2, divide_in_3

Illustration of game-play



University of Amsterdam

Assignment of this week

- Generate an expert system for the chess problem Rook and Rook versus King

University of Amsterdam