DM842

Computer Game Programming II: AI

Lecture 10

Board Games

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

Combinatorial Game Theory

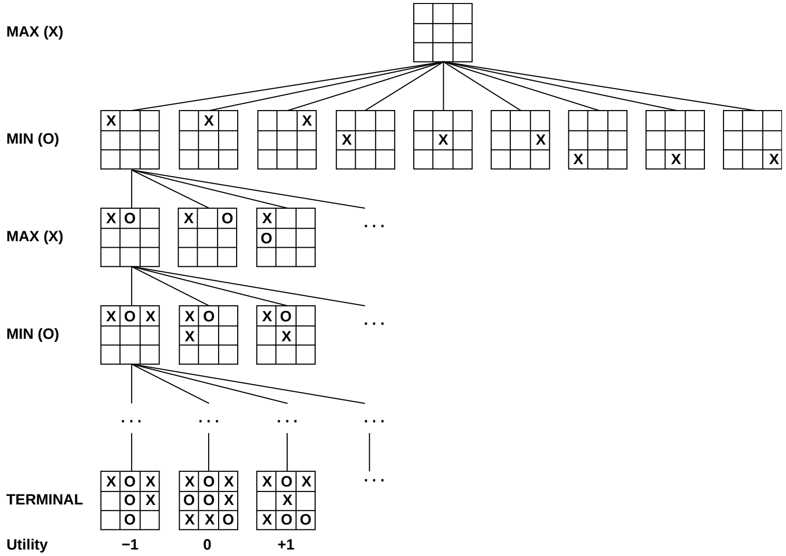
- Combinatorial game theory studies deterministic, sequential two-players games with perfect information
- Is there some move I can make, such that for all moves my opponent might make, there will then be some move I can make to win?
- Perfect play is the behavior or strategy of a player that leads to outcomes at least as good as any other strategy for that player regardless of the response by the opponent, hence even if opponent is infallible. (aka Optimal strategy)
- A solved game is a game whose outcome (win, lose, or draw) can be correctly predicted from any position, given that both players play perfectly.
- See `http://en.wikipedia.org/wiki/Solved_game` for a list of solved games.

Games vs. search problems

Search problem in a game tree (search tree overlapped on the game tree)

- initial state: root of game tree
- successor function: game rules/moves
- terminal test (is the game over?)
- utility function, gives a value for terminal nodes (eg, +1, -1, 0)

Game tree (2-player, deterministic, turns)



Solved Games

A two-player game can be "solved" on several levels:

- Ultra weak: Prove whether the first player will win, lose, or draw from the initial position, given perfect play on both sides.
  It can be non-constructive.

- Weak: Provide an algorithm that secures a win for one player, or a draw for either, against any possible moves by the opponent, from the beginning of the game. Produce at least one complete ideal game (all moves start to end are optimal). It does not analyse suboptimal moves, hence a winning position can turn into a draw if a non-optimal move is played.

- Strong: Exhaustively search a game tree to figure out what would happen if perfect play were realized. Provides optimal strategy from any position.

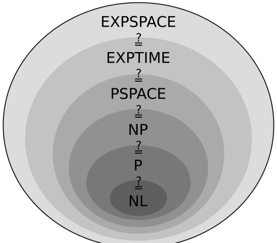
A minimax algorithm that exhaustively traverse the game tree would provide a weak or strong proof.

Measures of game complexity

- State-space complexity: number of legal positions reachable from the initial state. Most often, an upper bound that includes illegal positions.
- Game tree size: total number of possible games that can be played, ie, number of leaves of the game tree.
- Computational complexity of the generalized game (eg, played on a $n \times n$ board): often PSPACE-complete (set of all decision problems that can be solved by a Turing machine using a polynomial amount of space and for which every other problem that can be solved in polynomial space can be transformed to one of these problems in polynomial time.)

Eg: Quantified Boolean formulas

$$\exists x_1, \forall x_2 \exists x_3 : (x_1 \vee x_2) \wedge (x_2 \vee x_3)$$



Outline

MiniMaxing

Starting from the bottom of the tree, scores are bubbled up according to the minimax rule:

- on our moves, we are trying to maximize our score
- on opponent moves, the opponent is trying to minimize our score

(Perfect play for deterministic, perfect-information games)

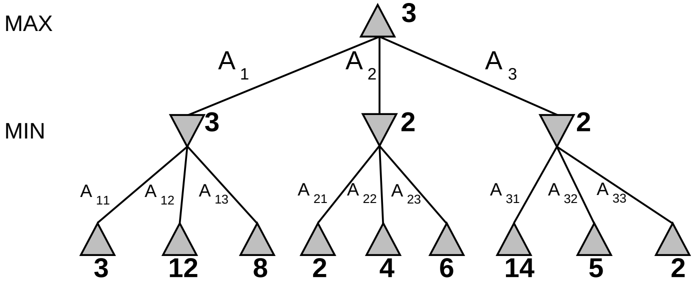
Implementation
recursion + at maximum search depth call the utility (static evaluation) function

Class representing one position in the game:

```
class Board:
   def getMoves()
   def makeMove(move)
   def evaluate(player)
   def currentPlayer()
   def isGameOver()
```

Example

2-ply game:



What if three players?

Minimax algorithm

Recursive Depth First Search:

```
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 $v$
```

Properties of minimax

Complete: Yes, if tree is finite (chess has specific rules for this)
Time complexity: $O(b^m)$
Space complexity: $O(bm)$ (depth-first exploration)

But do we need to explore every path?

Measures of Game Complexity

game-tree complexity: number of leaf nodes in the smallest full-width decision tree that establishes the value of the initial position.
A full-width tree includes all nodes at each depth.
estimates the number of positions to evaluate in a minimax search to determine the value of the initial position.

approximation: game's average branching factor to the power of the number of plies in an average game.
Eg.: chess For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
$\Rightarrow$ exact solution completely infeasible

Historical view

Time limits $\Rightarrow$ unlikely to find goal, must approximate

Plan of attack:

- Computer considers possible lines of play (Babbage, 1846)
- Algorithm for perfect play - MINIMAX - (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 - $\alpha - \beta$ alg. - (McCarthy, 1956)

Resource limits

Standard approaches:

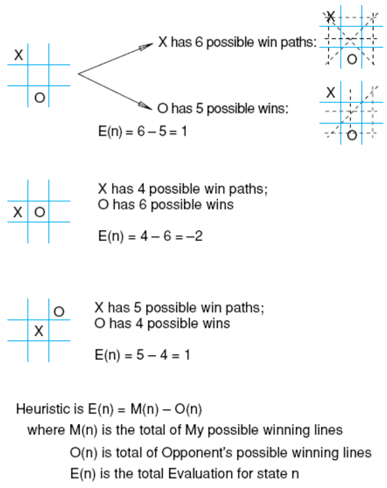
- n-ply lookahead: depth-limited search
- heuristic descent
- heuristic cutoff

  1. Use Cutoff-Test instead of Terminal-Test
     e.g., depth limit (perhaps add quiescence search)
  2. Use Eval instead of Utility
     i.e., evaluation function that estimates desirability of position

Suppose we have 100 seconds, explore 10^4 nodes/second
$\Rightarrow 10^6$ nodes per move $\approx 35^{8/2}$

Heuristic Descent

Heuristic measuring conflict applied to states of tic-tac-toe



X has 6 possible win paths:

O has 5 possible wins:

E(n) = 6 – 5 = 1

X has 4 possible win paths;
O has 6 possible wins

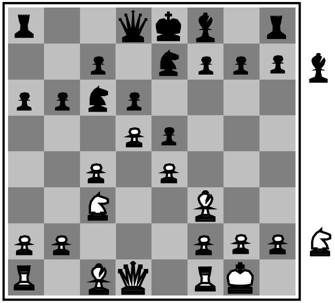
E(n) = 4 – 6 = –2

X has 5 possible win paths;
O has 4 possible wins

E(n) = 5 – 4 = 1

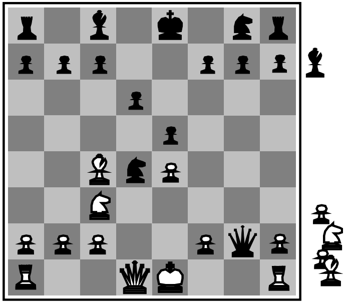
Heuristic is E(n) = M(n) – O(n)
where M(n) is the total of My possible winning lines
O(n) is total of Opponent's possible winning lines
E(n) is the total Evaluation for state n

Evaluation functions



Black to move

White slightly better



White to move

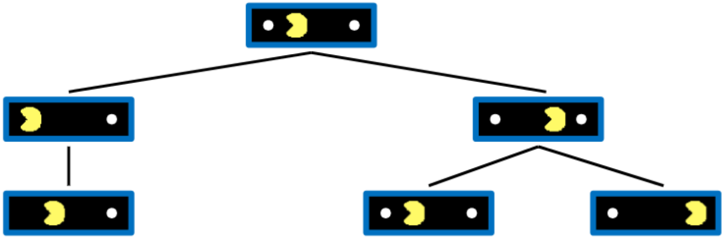
Black winning

For chess, typically weighted sum of features

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

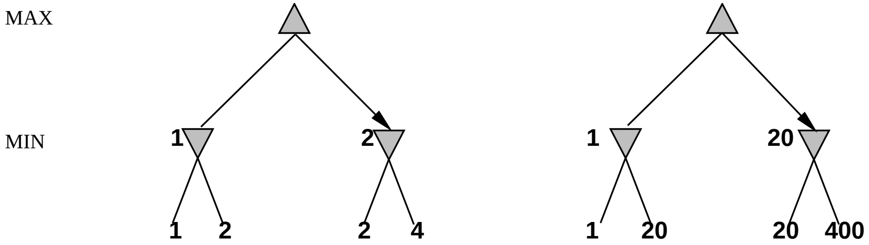
e.g., $w_1 = 9$ with
$f_1(s)$ = (number of white queens) – (number of black queens), etc.

Thrashing



- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

Digression: Exact values don't matter



Behaviour is preserved under any *monotonic* transformation of Eval

Only the order matters:

payoff in deterministic games acts as an ordinal utility function

MiniMaxing

static evaluation function: heuristic to score a state of the game for one player

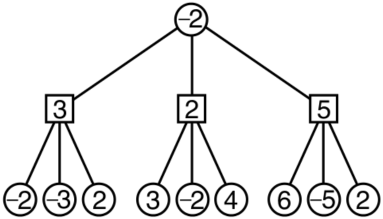
- it reflects how likely a player is to win the game from that board position
- knowledge of how to play the game (ie, strategic positions) enters here. Eg: Reversi, higher score for fewer counters in the middle of the game
- the domain is the natural numbers ($-100; +100$)
- Eg. in Chess: ± 1000 for a win or loss, 10 for the value of a pawn
- there may be several scoring functions which are then combined in a single value (eg, by weighted sum, weigths can depend on the state of the game)
- since heuristic is not perfect, one can enhance them by lookahead to decide which move to take

Negamaxing

For two player and zero sum games:
If one player scores a board at -1, then the opponent should score it at $+1$

$\rightsquigarrow$ simplify the minimax algorithm.

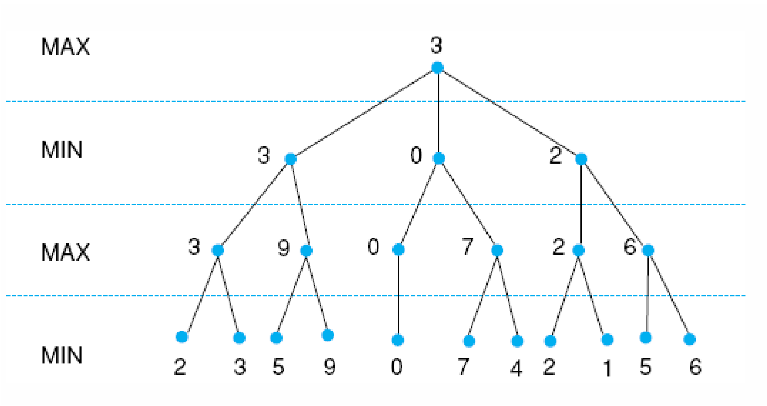
- adopt the perspective of the player that has to move
- at each stage of bubbling up, all the scores from the previous level have their signs changed
- largest of these values is chosen at each time



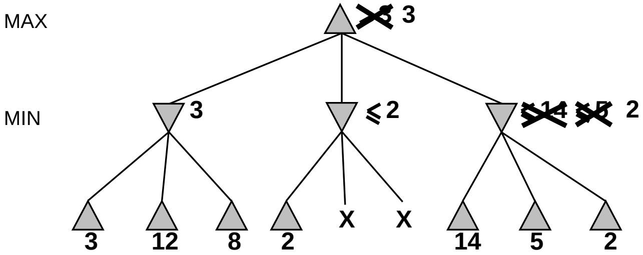
Simpler implementation but same complexity

Outline

Example

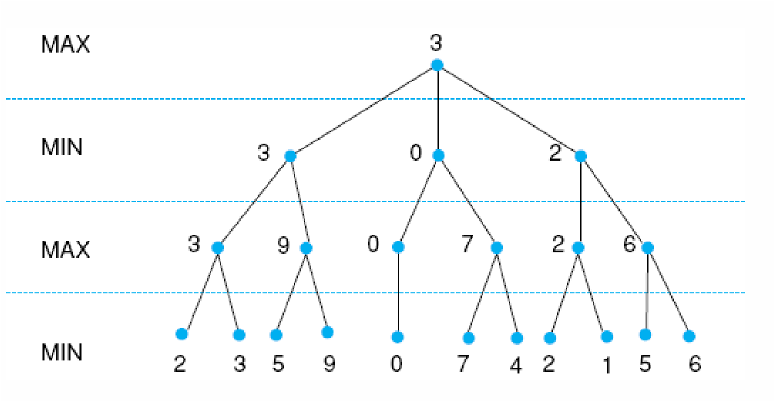


α–β pruning example

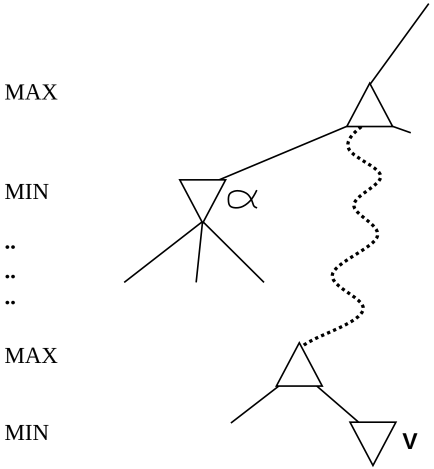


$$\text{Minimax}(root) = \max\{3, \min\{2, x, y\}, \min\{\ldots\}\}$$

Example



Why is it called α–β?



α is the best value (to MAX) found so far along the current path
If V is worse ($<$) than α, MAX will avoid it $\Rightarrow$ prune that branch
Define β similarly for MIN

The α–β algorithm

α is the best value to MAX up to now for everything that comes above in the game tree. Similar for β and MIN.

```
**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($s$,$a$), $\alpha$, $\beta$))
    **if** $v \geq \beta$ **then return** $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  **return** $v$
```

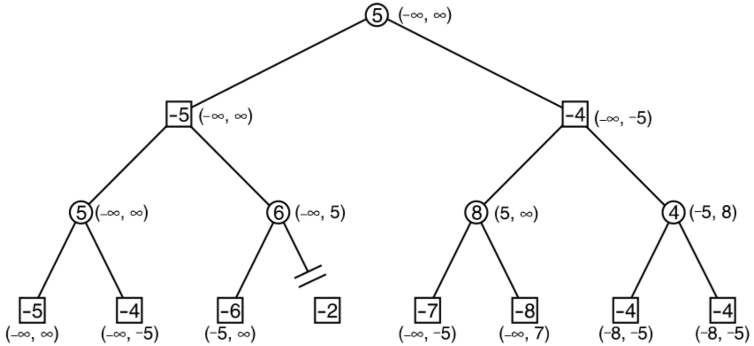
```
**function** MIN-VALUE($state, \alpha, \beta$) **returns** $a\ utility\ value$
  **if** TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  **for each** $a$ **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s$,$a$) , $\alpha$, $\beta$))
 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

Properties of α–β

- (α, β) search window: we will never choose to make moves that score less than alpha, and our opponent will never let us make moves scoring more than beta.
- Pruning *does not* affect final result
- Good move ordering improves effectiveness of pruning (shrinks window) consider first most promising moves:
  - use heuristics
  - use results of previous minimax searches (from iterative deepening or previous turns)
- With "perfect ordering," time complexity = $O(b^{m/2}) \Rightarrow$ *doubles* solvable depth
- if b is relatively small, random orders leads to $O(b^{3m/4})$

Alpha-beta Negamax

It swaps and inverts the alpha and beta values and checks and prunes against just the beta value



NegaMax

```
function negamax(node, depth, color)
    if depth = 0 or node is a terminal node
        return color * the heuristic value of node
    bestValue := -infty
    foreach child of node
        val := -negamax(child, depth - 1, -color)
        bestValue := max( bestValue, val )
    return bestValue

# Initial call for Player A's root node
rootNegamaxValue := negamax( rootNode, depth, 1)
rootMinimaxValue := rootNegamaxValue

# Initial call for Player B's root node
rootNegamaxValue := negamax( rootNode, depth, -1)
rootMinimaxValue := -rootNegamaxValue
```

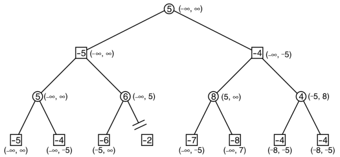
Alpha-Beta NegaMax

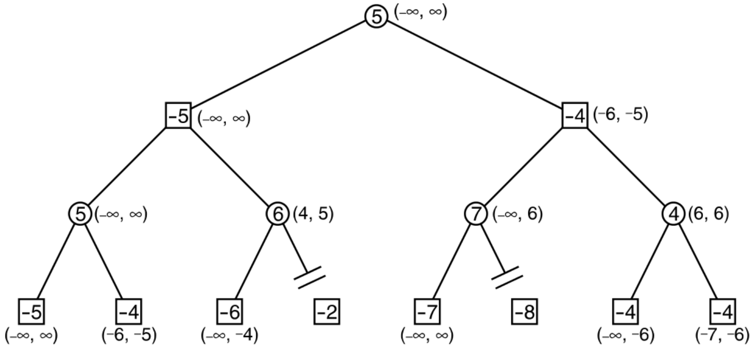
```
function negamax(node, depth, alpha, beta, color)
    if depth = 0 or node is a terminal node
        return color * the heuristic value of node
    bestValue := -infty
    childNodes := GenerateMoves(node)
    childNodes := OrderMoves(childNodes)
    foreach child in childNodes
        val := -negamax(child, depth - 1, -beta, -alpha, -color)
        bestValue := max( bestValue, val )
        alpha := max( alpha, val )
        if alpha >= beta
            break
    return bestValue

# Initial call for Player A's root node
rootNegamaxValue := negamax( rootNode, depth, -infty, +infty, 1)
```

Negascout

- aspiration search restrict the window range artifically maybe using results from previous search (eg. (5 - window size, 5 + window size))
- extreme cases window size = 0
  fail soft: the search returns a more sensible window to guide the guess
- full examination of the first move from each board position (wide search window)
- successive moves are examined using a scout pass with a window based on the score from the first move
- If the pass fails, then it is repeated with a full-width window
- In general, negascout dominates $\alpha\beta$ negamax; it always examines the same or fewer boards.

Alpha-beta Negascout





Outline

Transposition Tables and Memory

- algorithms can make use of a transposition table to avoid doing extra work searching the same board position several times
- working memory of board positions that have been considered
- use specialized hash functions
  desiderata: spread the likely positions as widely as possible through the range of the hash value.
  hash values that change widely when from move to move the board changes very little

Zobrist key

- Zobrist key is a set of fixed-length random bit patterns stored for each possible state of each possible location on the board.
  Example: Chess has 64 squares, and each square can be empty or have 1 of 6 different pieces on it, each of two possible colors. Zobrist key needs to be $64 \times (2 \times 6 + 1) = 832$ different bit-strings.

- the Zobrist keys need to be initialized with random bit-strings of the appropriate size.

- for each non-empty square, the Zobrist key is looked up and XORed with a running hash total.

- they can be incrementally updated

Eg: for a tic-tac-toe game

```
zobristKey[9*2]
def initZobristKey():
  for i in 0..9*2:
    zobristKey[i] = rand32()


def hash(ticTacToeBoard):
  result = 0
  for i in 0..9:
    piece = board.
          getPieceAtLocation(i)
    # piece is 0 or 1
    if piece != UNOCCUPIED:
      result = result xor
            zobristKey[i*2+
            piece]
  return result
```

What to store?

- hash table stores the value associated with a board position
- the best move from each board position
- depth used to calculate that value
- accurate value, or we may be storing “fail-soft” values that result from a branch being pruned.
- accurate value or fail-low value (alpha pruned), or fail-high value (beta pruned)

Implementation:
hash table is an array of lists `buckets[hashValue % MAX_BUCKETS]`

There is no point in storing positions in the hash table that are unlikely to ever be visited again. $\rightsquigarrow$ hash array implementation, where each bucket has a size of one.

how and when to replace a stored value when a clash occurs?

- always overwrite
- replace whenever the clashing node is for a later move
- keep multiple transposition tables with different replacement strategies

Space: linear in branching factor and maximum search depth used

Debug

Measure:

- number of buckets used at any point in time,
- number of times something is overwritten,
- number of misses when getting an entry that has previously been added

If you rarely find a useful entry in the table, then the number of buckets may be too small, or the replacement strategy may be unsuitable, etc.

Deterministic games in practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
- Kalaha (6,6) solved at IMADA in 2011
- Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- Othello: human champions refuse to compete against computers, who are too good.
- Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

Summary: AI4GP

1. Movement
2. Pathfinding
3. Decision making
4. Tactical and strategic AI
5. Board game AI