Game Trees: The Minimax Method
Overview

• Game Tree
• Minimax
  – Theorem
  – Method
  – Algorithm
  – Example
• Optimization
  – Glossary
  – alpha-beta Pruning
  – SCOUT
'n Spieler, S1, S2,..., Sn, spielen ein gegebenes Gesellschaftsspiel. Wie muß einer dieser Spieler, Sm, spielen, um dabei ein möglichst günstiges Resultat zu erzielen?‘ ~ Neumann

Zur Theorie der Gesellschaftsspiele by J. v. Neumann
Game Tree

- directed, acyclic graph
- nodes := positions
- edges := moves
- number of Leafs := number of possible game plays
• complete game tree of Tic Tac Toe without cutting off any nodes:
  – number of plies = 9
  – number of possible game plays = 9!
  – number of nodes of ply n = 9! – (9-(n-1))!
  – number of positions = $9^2 \times 8^2 \times 7^2 \times 6^2 \times 5^2 \times 4^2 \times 3^2 \times 2^2 + 1 = \mathbf{131\ 681\ 894\ 401}$
Minimax – Theorem

• every finite, zero-sum, two-person game has optimal mixed strategies

• zero-sum: player A’s gain = player B’ loss  
  player A’s loss = player B’s gain

• mixed strategies: a collection of moves together with a corresponding set of weights which are followed probabilistically in the playing of a game
  – or: a collection of pure strategies with assigned probabilities
  – pure strategy: a complete set of moves for one player
• one-ply game: player A chooses best move
• two-ply game: player B chooses his best move, after A chose his move
  – > best move for ply 1 isn’t necessarily best move over 2 plies
• minimax algorithm plays every possible game returning the best move for player A (Max) and worst move for player B (Mini) per ply
• set the probability of Mini choosing the worst move for Max to 100% and other moves to 0%
Minimax – Method (2)

- MINImizing the cost for a MAXimum cost scenario
- or: maximizing the gain of a minimum gain scenario
Minimax – Algorithm

**Pseudocode:**

```plaintext
int maxi( int depth ) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for ( all moves ) {
        score = mini( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}

int mini( int depth ) {
    if ( depth == 0 ) return -evaluate();
    int min = +oo;
    for ( all moves ) {
        score = maxi( depth - 1 );
        if( score < min )
            min = score;
    }
    return min;
}
```

chessprogramming.wikispaces.com
Minimax Example (1)
Minimax Example (2)
• searching whole game trees of more advanced games would consume too much time

• Shannon Number: $10^{120}$

(number of atoms in observable universe: estimated between $4 \times 10^{79}$ and $10^{81}$)

Shannon: “A machine operating at the rate of one variation per micro-second would require over $10^{90}$ years to calculate the first move!”

C.E. Shannon, “Programming a computer for playing chess”, 1950
Optimization (2)

• looking ahead to up to $x$ plies, not the whole tree $\Rightarrow$ heuristic evaluation function
  – Deep Blue looked ahead $\sim 17/18$ plies in 2006

• cutting off decision trees, which solution wouldn’t change the outcome of the algorithm
Optimization – Glossary (1)

• Null-Window Search:
  - reduction of search space by a boolean test related to a passed value

• Negamax:
  - implementation of minimax and derived algorithms with max(a, b) == -min(-a, -b) instead of Max and Mini

```c
int negaMax(int depth) {
    if (depth == 0) return evaluate();
    int max = -oo;
    for (all moves) {
        score = -negaMax(depth - 1);
        if (score > max)
            max = score;
    }
    return max;
}
```

chessprogramming.wikispaces.com
• Transposition Tables:
  – previous searches storing database
  – transpositions don’t have to be reevaluated

• Aspiration Windows:
  – guesses expected value and calculates an alpha/beta-window around the guess
  – window is narrower => more beta cutoffs
  – if alpha < value < beta: costly research
• “It is possible to improve on the brute-force search by using a "branch-and-bound" technique, ignoring moves which are incapable of being better than moves which are already known.”

• alpha / beta is the highest favorability Max / the lowest favorability Mini is assured of
Optimization – alpha-beta Pruning (2)

**Pseudocode:**

```python
function alphabeta(node, depth, α, β, Player)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if Player = MaxPlayer
        for each child of node
            α := max(α, alphabeta(child, depth-1, α, β, not(Player) ))
            if β ≤ α
                break
            return α
    else
        for each child of node
            β := min(β, alphabeta(child, depth-1, α, β, not(Player) ))
            if β ≤ α
                break
        return β
(* Initial call *)
alphabeta(origin, depth, -infinity, +infinity, MaxPlayer)
```

Wikipedia
Optimization – alpha-beta Pruning (4)
Optimization – alpha-beta Pruning (5)
• winner of the 2011 ACM A.M. Turing Award for innovations that enabled remarkable advances in the partnership between humans and machines that is the foundation of Artificial Intelligence (AI)

• developed SCOUT, a null-window search expecting the value of the left leaf as the ‘initial solution’
SCOUT, the algorithm described in this paper, has evolved as a purely theoretical tool for analyzing the mean complexity of game-searching tasks where the terminal nodes are assigned random and independent values [1]. With the aid of SCOUT we were able to show that such games can be evaluated with a branching factor of $P^*/(1-P^*)$, where $P^*$ is the root of $x^d + x - 1 = 0$, and that no directional algorithm (e.g., ALPHA-BETA) can do better. We have recently tested the performance of SCOUT on a 'real' game (i.e., the game of Kalah) and were somewhat surprised to find that, even for low values of $h$, the efficiency of SCOUT surpasses that of the α-β procedure [2]. The purpose of this paper is to call attention of game-playing practitioners to the potentials of SCOUT as a practical game-searching tool.
Procedure: TEST(S, v, >)

To test whether S satisfies the inequality \( V(S) > v \), start applying the same test (calling itself) to its successors from left to right:

If S is MAX, return TRUE as soon as one successor is found to be larger than v; return FALSE if all successors are smaller than or equal to v.

If S is MIN, return FALSE as soon as one successor is found to be smaller than or equal to v; return TRUE if all successors are larger than v.

An identical procedure, called TEST(S, v, \( \geq \)), can be used to verify the inequality \( V(S) \geq v \), with the obvious revisions induced by the equality sign.

Pseudocode:

function TEST(S, v, >)
    if terminal node
        if \( V(S) > v \)
            return TRUE
        else return FALSE
    else
        if MAX
            for each child of node
                if TEST(child, v, >)
                    return TRUE
                return FALSE
        else
            for each child of node
                if not TEST(child, v, >)
                    return FALSE
                return TRUE
Procedure: EVAL(S)

EVAL evaluates a MAX position S by first evaluating (calling itself) its left most successor $S_1$, then 'scouting' the remaining successors, from left to right, to determine (calling TEST) if any meets the condition $V(S_k) > V(S_1)$. If the inequality is found to hold for $S_k$, this node is then evaluated exactly (calling EVAL($S_k$)) and its value $V(S_k)$ is used for subsequent 'scoutings' tests. Otherwise $S_k$ is exempted from evaluation and $S_{k+1}$ selected for a test. When all successors have been either evaluated or tested and found unworthy of evaluation, the last value obtained is issued as $V(S)$.

An identical procedure is used for evaluating a MIN position $S$, save for the fact that the event $V(S_k) \geq V(S_1)$ now constitutes grounds for exempting $S_k$ from evaluation. Flow-charts describing both SCOUT and TEST in algorithmic details can be found in [1].
Optimization – SCOUT (5)
Optimization – SCOUT (6)
### Optimization – SCOUT (7)

<table>
<thead>
<tr>
<th>Search Depth</th>
<th>Random Ordering</th>
<th>Dynamic Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SCOUT</td>
<td>α-β</td>
</tr>
<tr>
<td>2</td>
<td>82</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>394</td>
<td>380</td>
</tr>
<tr>
<td>4</td>
<td>1173</td>
<td>1322</td>
</tr>
<tr>
<td>5</td>
<td>2514</td>
<td>4198</td>
</tr>
<tr>
<td>6</td>
<td>5111</td>
<td>6944</td>
</tr>
</tbody>
</table>

SCOUT: a simple game searching algorithm with proven optimal properties by Judea Pearl, April 1980
Further Algorithms

- NegaScout
- PVS (essentially the same as Negascout)
- MTD(f)
- SSS* and its variations
- C* and its variations
Sources (1)

Sources (2)