**Game**

MinMax

Alpha-Beta

---

**Definition**

- It is a Search depth Tree
- Defined by:
  - Initial state (how the board is set up)
  - Operators (legal moves)
  - Terminal state (game is over)
  - Utility or payoff function (who won, by how much)
- Two strategies have been defined:
  - MinMax algorithm
  - Alpha-Beta algorithm

---

**MinMax Algorithm**

- Two-player games with perfect information, the minmax can determine the best move for a player by evaluating (evaluating) the entire game tree.
- Player 1 is called Max
  - Maximizes result
- Player 2 is called Min
  - Minimizes opponent’s result

---

**MinMax Example**

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minmax value

- best achievable payoff against best play

**E.g., 2-ply game:**

```
MIN
  / 1
 / 3
A1 A2 A3

MAX
  |
  |
  2

```

---

**MinMax Analysis**

- Time Complexity: \( b^d \)
- Space Complexity: \( b^d \)
- Problem → **Resources Limited!**
- Time to make an action (it is a game)
- Can we do better? Yes!
- How? Cutting useless branches!

---

**MinMax steps**

```
def MinMax(state, depth, type):
    if terminate(state):
        return Eval(state)
    if type == max:
        best_score = float('-inf')
        for child in state.successors:
            value = MinMax(child, depth+1, min)
            if value > best_score:
                best_score = value
        return best_score
    if type == min:
        best_score = float('inf')
        for child in state.successors:
            value = MinMax(child, depth+1, max)
            if value < best_score:
                best_score = value
        return best_score
```
Partial Game Tree for Tic-Tac-Toe

Utility values:
f(n)=+1 if position is a win for X;
f(n)=-1 if position is a win for O;
f(n)=0 if position is a draw

Minimax Algorithm

function MINIMAX-DECISION(state) returns an action
inputs: state, current state in game
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

Two-Ply Game Tree

Two-Ply Game Tree

Two-Ply Game Tree

Two-Ply Game Tree
Properties of Minimax

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Minimax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
</tr>
</tbody>
</table>

How do we deal with resource limits?
- Evaluation function: return an estimate of the expected utility of the game from a given position
- Alpha-beta pruning: return appropriate minimax decision without exploring entire tree

Heuristic Evaluation Function: Tic Tac Toe

Heuristic Evaluation Functions

Eval($s$) = $w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s)$

Alpha-Beta Algorithm
- It is based on process of eliminating a branch of the search tree “pruning” the search tree.
- It is applied as standard minimax tree, but it returns the same move as minimax would but prunes away branches that cannot possibly influence the final decision.

Alpha-beta Pruning

Replace TERMINAL-TEST and UTILITY in Minimax:

if TERMINAL-TEST(state) then return UTILITY(state)

if CUTOFF-TEST(state) then return EVAL(state)
Example

Alpha-Beta Example
(continued)

Alpha-Beta Example
(continued)

Alpha-Beta Example
(continued)

Alpha-Beta Example
(continued)
**α-β Algorithm**

```c
int AlphaBeta (state s, int depth, int type, int Alpha, int Beta)
{
  if (terminate(s)) return Eval(s);
  if (type == max)
  {
    for (child = 1, child <= NmbSuccessor(s); child++)
    {
      value = AlphaBeta(Successor(s, child), depth+1, min, Alpha, Beta)
      if (value > Alpha)  Alpha = Value;
      if (value >= Beta) return Beta;
    }
    return Alpha;
  }
  if (type == min)
  {
    for (child = 1, child <= NmbSuccessor(s); child++)
    {
      value = AlphaBeta(Successor(s, child), depth+1, max, Alpha, Beta)
      if (value < Beta)  Beta = Value;
      if (value <= Alpha) return Alpha;
    }
    return Beta;
  }
}
```

**Alpha-Beta Example (continued)**

**Alpha-Beta Example**

**Alpha-Beta Example (continued)**

**Alpha-Beta Analysis**

- In perfect case (perfect ordering) the depth is decreased twice in time complexity:
  -  \( O \left( b^{d/2} \right) \)
  - which means that the branching factor \( b \) is decreased to \( \sqrt{b} \).
Conclusion

- …Performance Performance !!!
- Improve Alpha-Beta to guarantee best-case results
- Improve heuristic evaluation function to improve the predictive capabilities of the search
- Use parallelism to increase the search depth