Game playing - Adverserial Search

AIMA.II - Chapter 6
Games

◊ Game theory: a branch of economics which views any multiagent environment (competitive or cooperative) as a game (provided that the impact of the agents on each other is significant).

◊ One of the oldest areas of AI.

Games are difficult. Chess programs were especially chosen because success would be a proof of a machine doing something intelligent.
Games vs. search problems

The presence of an opponent introduces uncertainty which in turn makes the decision problem more complicated than search problems.

“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)
Game Description

A game can be formally defined as a kind of search problem with:

◊ initial state (of the board and whose turn it is)
◊ set of operators (which define the legal moves)
◊ terminal test (goal test)
◊ utility function: (numeric value for the outcome of a game)

Ex. backgammon (+1, -1, +2); Chess (win, lose, draw)
Game Description

♦ Zero-sum games: if one opponent gains, the other loses an equal amount (i.e. they are using opposite utility functions). More general concept is constant-sum games.

♦ Non-zerosum games: opponents may join forces to increase their gains together.
Games vs. Search problems: Time constraints

Real problem is that games are usually much too hard to solve:

In chess:

◊ Average branching factor: 35

◊ Games go to about 50 moves by each player

\[ 35^{100} \text{ nodes!} \hspace{0.5cm} 10^{40} \text{ different legal positions} \]

Time limits ⇒ unlikely to find goal, must approximate

On the other hand tic-tac-toe is boring because it is too simple to determine the best move.
Two-Person Games

Players: MAX and MIN taking turns until game is over

We can view MAX as the agent: in other words, MAX is constructing the search tree at each move and plays so as to maximize its gains assuming a perfect opponent.
Search Tree for the Game Tic-Tac-Toe

MAX (X)
MIN (O)
TERMINAL

Utility

-1  0 +1
Games vs. Search problems

The complexity of games introduces a new kind of uncertainty:

not due to lack of information but because one does not have time to calculate the exact consequences of any move.
## Types of games

<table>
<thead>
<tr>
<th>Perfect Information</th>
<th>Deterministic</th>
<th>Chance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>chess, checkers, go, othello</td>
<td>backgammon, monopoly</td>
</tr>
<tr>
<td>Imperfect Information</td>
<td></td>
<td>bridge, poker, scrabble nuclear war</td>
</tr>
</tbody>
</table>
Minimax

Minimax algorithm is designed to determine the optimal strategy for MAX: 
*Perfect play for deterministic, perfect-information games*

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

E.g., 2-ply game:

\[
\begin{array}{cccccccc}
3 & 12 & 8 & 6 & 4 & 2 & 14 & 5 & 2 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
3 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
A_1 & A_2 & A_3 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
A_{11} & A_{12} & A_{13} \\
3 & 12 & 8 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
A_{21} & A_{22} & A_{23} \\
2 & 4 & 6 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
A_{31} & A_{32} & A_{33} \\
14 & 5 & 2 \\
\end{array}
\]
Minimax algorithm

**function Minimax-Decision(game) returns an operator**

for each op in OPERATORS[game] do
    Value[op] ← Minimax-Value(Apply(op, game), game)
end
return the op with the highest Value[op]

**function Minimax-Value(state, game) returns a utility value**

if Terminal-Test[game](state) then
    return Utility[game](state)
else if max is to move in state then
    return the highest Minimax-Value of Successors(state)
else
    return the lowest Minimax-Value of Successors(state)
Minimax algorithm

◊ The optimal strategy can be determined by examining the minimax value of each node.

◊ Max maximizes its worst-case outcome!

◊ Recursive search.
Properties of minimax

**Minimax**: Maximizes the utility under the assumption that the opponent will play perfectly to minimize it.

- **Complete??**
- **Optimal??**
- **Time complexity??**
- **Space complexity??**
Properties of minimax

**Complete??** Yes, if tree is finite (chess has specific rules for this)

**Optimal??** Yes, against an optimal opponent. Otherwise??

**Time complexity??** $O(b^m)$

**Space complexity??** $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for “reasonable” games

$\Rightarrow$ exact solution completely infeasible
The minimax algorithm assumes that the program has time to search all the way down to terminal states, which is usually not practical.

Shannon proposed that instead of going all the way down to terminal states and using the utility function, the program should cut-off the search earlier, and apply a heuristic evaluation function to the leaves of the tree.
Resource limits

Suppose we have 100 seconds, and you can explore $10^4$ nodes/second
⇒ $10^6$ nodes per move

Standard approach:

• *cutoff test*
  e.g., depth limit

• *evaluation function*
  = estimated desirability of position
Resource limits

• cutoff test
  e.g., depth limit
  Replace terminal test by cutoff-test in MINIMAX-VALUE

• evaluation function
  = estimated desirability of position
  Replace Utility by EVAL in MINIMAX-VALUE
function Minimax-Decision(game) returns an operator

    for each op in OPERATORS[game] do
        Value[op] ← Minimax-Value(Apply(op, game), game)
    end

    return the op with the highest Value[op]

function Minimax-Value(state, game) returns a utility value

    if CutOff-Test[game](state) then
        return Eval[game](state)
    else if MAX is to move in state then
        return the highest Minimax-Value of Successors(state)
    else
        return the lowest Minimax-Value of Successors(state)
Evaluation functions

Estimate of the expected utility of the game from a given position.

E.g. material value for each piece: 1 for pawn, 3 for knight or bishop,...

Performance of a game-playing program is extremely dependent on the quality of its evaluation function.
For chess, typically linear 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
\[ f_1(s) = (\text{number of white queens}) - (\text{number of black queens}) \]
Evaluation functions:

◊ should agree with the utility function on terminal states.

◊ must not take too long to calculate!

◊ should accurately reflect the *chances* of winning (if we have to cut-off, we do not know what will happen in subsequent moves)
Cutting off search

\textsc{MinimaxCutoff} is identical to \textsc{MinimaxValue} except

1. \textsc{Terminal} is replaced by \textsc{Cutoff}?
2. \textsc{Utility} is replaced by \textsc{Eval}

Does it work in practice?
Suppose we have 100 seconds to decide on a move and we can explore $10^4$ nodes/second

\[ b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4 \]

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov
Exact values don’t matter

Behaviour is preserved under any monotonic transformation of \texttt{Eval}

Only the order matters:

- payoff in deterministic games acts as an ordinal utility function
Cutting off search

Should look further:

♦ in positions where favorable captures can be made (non-quiescent positions)

♦ in positions where unavoidable (but beyond the horizon moves) will affect the situation drastically

e.g. pawn turning into queen
\( \alpha-\beta \text{ pruning} \)

With minimax 4-ply possible, but even average human players can make plans 6-8 ply ahead!

How can we improve minimax search?
$\alpha - \beta$ pruning example

\[
\begin{array}{c}
\text{MAX} \\
\text{MIN}
\end{array}
\begin{array}{c}
3 \\
3 \\
12 \\
8
\end{array}
\begin{array}{c}
\geq 3
\end{array}
\]
$\alpha-\beta$ pruning example
\(\alpha-\beta\) pruning example
$\alpha-\beta$ pruning example

MAX

MIN

3 12 8
3
2
14
5

X

$\geq 3$

$< 2$

$< 5$
$\alpha-\beta$ pruning example
Properties of $\alpha-\beta$

Pruning *does not* affect final result

Good move ordering improves effectiveness of pruning (e.g. pick move with $b=2$ rather than $b=14$)

With “perfect ordering,” time complexity = $O(b^{m/2})$

$\Rightarrow$ *doubles* depth of search

$\Rightarrow$ can easily reach depth 8 and play good chess
Why is it called $\alpha-\beta$?

◊ $\alpha$ is the best value (to MAX) found so far off the current path

If $V$ is worse than $\alpha$, MAX will avoid it $\Rightarrow$ prune the rest of that branch

◊ Define $\beta$ for MIN and prune similarly those branches that MIN would avoid.
The $\alpha$–$\beta$ algorithm

Basically Minimax + keep track of $\alpha$, $\beta$ + prune
function **Max-Value** *(state, game, α, β)* returns the minimax value of *state*

**inputs:** *state*, current state in game

*game*, game description

α, the best score for MAX along the path to *state*

β, the best score for MIN along the path to *state*

**if** Cutoff-Test(*state*) **then return** Eval(*state*)

**for each** s in Successors(*state*) **do**

α ← Max(α, Min-Value(s, game, α, β))

**if** α ≥ β **then return** β

end

return α

function **Min-Value** *(state, game, α, β)* returns the minimax value of *state*

**if** Cutoff-Test(*state*) **then return** Eval(*state*)

**for each** s in Successors(*state*) **do**

β ← Min(β, Max-Value(s, game, α, β))

**if** β ≤ α **then return** α

end

return β
Minimax is an optimal method for selecting a move from a given search tree provided that the leaf node evaluations are exactly correct.

In the situations shown above, if the value of 99 is a mistake (since all neighbors have much higher utilities), the algorithm will invariably make the wrong choice. Hence, MINIMAX is only as good as the heuristic evaluation is.
Digression: Exact values don’t matter

Behaviour is preserved under any monotonic transformation of \textsc{Eval}

Only the order matters:
- payoff in deterministic games acts as an ordinal utility function
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.

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 (note: Baron von Kempelen’s “Turk” in 1769!).
Deterministic games in practice

Backgammon: First program to make a serious impact, BKG, used only a one-ply search but a very complicated evaluation function (1980). It plays a strong amateur level. In 1992, neural network techniques to learn evaluation function.

Othello/Reversi: Smaller search space \((b = 5 - 15)\). 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.
Nondeterministic games
Nondeterministic games

E.g., in backgammon, the dice rolls determine the legal moves

Simplified example with coin-flipping instead of dice-rolling:

```
    MAX
   /   \
CHANCE    \\
  /  \      \   \
MIN  2  4  7  4  0  5  -2

2  4  7  4  6  0  5  -2
```
Algorithm for nondeterministic games

**Expectiminimax** gives perfect play

Just like **Minimax**, except we must also handle chance nodes:

\[
\begin{align*}
\text{if } state \text{ is a Max node then} & \quad \text{return the highest } \text{Expectiminimax-Value} \text{ of Successors}(state) \\
\text{if } state \text{ is a Min node then} & \quad \text{return the lowest } \text{Expectiminimax-Value} \text{ of Successors}(state) \\
\text{if } state \text{ is a chance node then} & \quad \text{return average of } \text{Expectiminimax-Value} \text{ of Successors}(state) \\
\end{align*}
\]
Nondeterministic games in practice

Dice rolls increase $b$: 21 possible rolls with 2 dice

Backgammon $\approx 20$ legal moves

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

As depth increases, probability of reaching a given node shrinks

$\Rightarrow$ value of lookahead is diminished

TDGammon uses depth-2 search + very good Eval

$\approx$ world-champion level
Nondeterministic games in practice

\(\alpha-\beta\) pruning cannot be used in this form. The advantage of \(\alpha-\beta\) pruning is that it ignores future developments that just are NOT going to happen (given best play).

In games with dice, there are no likely sequences of moves, because for those moves to take place, the dice would have to come out the right way to make them legal.

However, a version of \(\alpha-\beta\) pruning is possible but only if the leaf values are bounded. (Why?).
Digression: Exact values DO matter

Behaviour is preserved only by positive linear transformation of Eval

Hence Eval should be proportional to the expected payoff
Games of imperfect information

E.g., card games, where opponent’s initial cards are unknown

Typically we can calculate a probability for each possible deal

Seems just like having one big dice roll at the beginning of the game

Idea: compute the minimax value of each action in each deal,
then choose the action with highest expected value over all deals

Special case: if an action is optimal for all deals, it’s optimal.

GIB, current best bridge program, approximates this idea by
1) generating 100 deals consistent with bidding information
2) picking the action that wins most tricks on average
Summary

Games are fun to work on!

They illustrate several important points about AI

◊ perfection is unattainable ⇒ must approximate

◊ good idea to think about what to think about

◊ uncertainty constrains the assignment of values to states