CS 331: Artificial Intelligence
Adversarial Search

Games we will consider

- Deterministic
- Discrete states and decisions
- Finite number of states and decisions
- Perfect information i.e. fully observable
- Two agents whose actions alternate
- Their utility values at the end of the game are equal and opposite (we call this zero-sum)

“It’s not enough for me to win, I have to see my opponents lose”
Which of these games fit the description?

Two-player, zero-sum, discrete, finite, deterministic games of perfect information

Which of these games fit the description?

Two-player, zero-sum, discrete, finite, deterministic games of perfect information
What makes games hard?

- Hard to solve e.g. Chess has a search graph with about $10^{40}$ distinct nodes
- Need to make a decision even though you can’t calculate the optimal decision
- Need to make a decision with time limits

Formal Definition of a Game

A quintuplet $(S, I, \text{Succ}(), T, U)$:

<table>
<thead>
<tr>
<th>S</th>
<th>Finite set of states. States include information on which player’s turn it is to move.</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Initial board position and which player is first to move</td>
</tr>
<tr>
<td>Succ()</td>
<td>Takes a current state and returns a list of (move,state) pairs, each indicating a legal move and the resulting state</td>
</tr>
<tr>
<td>T</td>
<td>Terminal test which determines when the game ends. Terminal states: subset of $S$ in where the game has ended</td>
</tr>
<tr>
<td>U</td>
<td>Utility function (aka objective function or payoff function): maps from terminal state to real number</td>
</tr>
</tbody>
</table>
Nim

Many different variations. We’ll do this one.

- Start with 9 beaver logos
- In one player’s turn, that player can remove 1, 2 or 3 beaver logos
- The person who takes the last beaver logo wins
Formal Definition of Nim

A quintuplet (S, I, Succ(), T, U):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Max(IIIII), Max(III), Max(II), Max(I)</th>
<th></th>
<th>Min(IIII), Min(III), Min(II), Min(I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td>Max(IIIII)</td>
<td></td>
<td>Succ(Max(IIIII)) = {Min(IIII),Min(III),Min(II)}</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>Succ(Max(III)) = {Min(II),Min(I)}</td>
<td></td>
<td>Succ(Max(II)) = {Min(I)}</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>Max(I), Max(II), Max(III), Min(I), Min(II), Min(III)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td>Utility(Max(I) or Max(II) or Max(III)) = +1,</td>
<td></td>
<td>Utility(Min(I) or Min(II) or Min(III)) = -1</td>
</tr>
</tbody>
</table>

Notation: Max(IIIII) # matches left

Who’s move

Nim Game Tree

We’ll call the players Max and Min, with Max starting first
How to Use a Game Tree

- Max wants to maximize his utility
- Min wants to minimize Max’s utility
- Max’s strategy must take into account what Min does since they alternate moves
- A move by Max or Min is called a ply

The Minimax Value of a Node

The minimax value of a node is the utility for MAX of being in the corresponding state, assuming that both players play optimally from there to the end of the game

\[
\text{MINIMAX-VALUE}(n) = \begin{cases} 
\text{UTILITY}(n) & \text{If } n \text{ is a terminal state} \\
\max_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s) & \text{If } n \text{ is a MAX node} \\
\min_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s) & \text{If } n \text{ is a MIN node}
\end{cases}
\]

Minimax value maximizes worst-case outcome for MAX
Nim Game Tree

Max

Min

Max

Min

Max

Min

Max

Min

Max

Min

Minimax Values in Nim Game Tree

Max

Min

Max

Min

Max

Min

Max

Min

Max

Min

Max

Min
Minimax Values in Nim Game Tree

Max

Min

Max

Min

Max

Min

Max

Min

Max

Min

Max

Min

Max

Min

Max

Min

Max

Min

15

16
Minimax Values in Nim Game Tree

Minimax decision at the root: taking this action results in the successor with highest minimax value

Another Example

= Maximizing player
= Minimizing player
The 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 ← - Infinity
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 ← Infinity
for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
return v

The MINIMAX algorithm

• Computes minimax decision from the current state
• Depth-first exploration of the game tree
• Time Complexity O(b^m) where b=# of legal moves, m=maximum depth of tree
• Space Complexity:
  – O(bm) if all successors generated at once
  – O(m) if only one successor generated at a time (each partially expanded node remembers which successor to generate next)
Now have a vector of utilities for players (A,B,C). All players maximize their utilities. Note: In two-player, zero-sum games, we have a single value because the values are always opposite.
Minimax With 3 Players

A

B
(1,2,6) (4,2,3) (6,1,2) (7,4,1) (5,1,1) (1,5,2) (7,7,1) (5,4,5)

C (1,2,6) (6,1,2) (1,5,2) (5,4,5)
Subtleties With Multiplayer Games

• Alliances can be made and broken
• For example, if A and B are weaker than C, they can gang up on C
• But A and B can turn on each other once C is weakened
• But society considers the player that breaks the alliance to be dishonorable

Pruning

• Can we improve on the time complexity of $O(b^m)$?
• Yes if we prune away branches that cannot possibly influence the final decision
If we know that the only two outcomes are +1 and -1, what branches do we not need to explore when minimax backtracks?
Pruning in Nim

Max

Min

Max

Min

Max

Min

What happens if we have more than just two outcomes?

Pruning Intuition (General Case)

MAX

MIN

The max player will never choose the right subtree once it knows that it is upper bounded by 1.

Suppose we just went down this branch. We know that the minimax value of its parent will be ≤ 1.
Pruning Example

MINIMAX-VALUE(root)

= max(min(3,12,8),min(2,x,y),min(14,5,2))

= max(3,min(2,x,y),2)

= max(3,z,2) where \( z \leq 2 \)

= 3

Pruning Intuition

Remember that minimax search is DFS.
At any one time, we only have to consider the nodes along a single path in the tree

In general, let:

- \( \alpha \) = highest minimax value of all of the MAX player’s choices expanded on current path
- \( \beta \) = lowest minimax value of all of the MIN player’s choices expanded on current path
- If at a MIN player node, prune if minimax value of node \( \leq \alpha \)
- If at a MAX player node, prune if minimax value of node \( \geq \beta \)
### ALPHA-BETA Pseudocode

**Function: ALPHA-BETA-SEARCH(state)** returns an action

**Inputs:** state, current state in game

\[ v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \]

**Return:** the action in SUCCESSORS(state) with value \( v \)

**Function: MAX-VALUE(state, \( \alpha \), \( \beta \))** returns a utility value

**Inputs:** state, current state in game

- \( \alpha \), the value of the best alternative for MAX along the path to state
- \( \beta \), the value of the best alternative for MIN along the path to state

**If TERMINAL-TEST(state) then return UTILITY(state)**

\[ v \leftarrow -\infty \]

**For a, s in SUCCESSORS(state) do**

\[ v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, \alpha, \beta)) \]

**If** \( v \geq \beta \) **then return** \( v \)

\[ \alpha \leftarrow \text{MAX}(\alpha, v) \]

**Return** \( v \)

---

**Function: MIN-VALUE(state, \( \alpha \), \( \beta \))** returns a utility value

**Inputs:** state, current state in game

- \( \alpha \), the value of the best alternative for MAX along the path to state
- \( \beta \), the value of the best alternative for MIN along the path to state

**If TERMINAL-TEST(state) then return UTILITY(state)**

\[ v \leftarrow +\infty \]

**For a, s in SUCCESSORS(state) do**

\[ v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, \alpha, \beta)) \]

**If** \( v \leq \alpha \) **then return** \( v \)

\[ \beta \leftarrow \text{MIN}(\beta, v) \]

**Return** \( v \)
Illustrating the Pseudocode

- In the example to follow, the notation $(-\infty, +\infty)$ represents the $(\alpha, \beta)$ values for the corresponding node.
- This example is intended to illustrate how the actual implementation of Alpha-Beta pruning works.

\[ \Delta = \text{Maximizing player} \]
\[ \nabla = \text{Minimizing player} \]

Alpha-Beta Pruning Example
### Alpha-Beta Pruning Example

**Alpha-Beta Pruning Example**

<table>
<thead>
<tr>
<th>e)</th>
<th>f)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>((3, +\infty))</td>
<td>((3, +\infty))</td>
</tr>
<tr>
<td>Pruning happens: (2 \leq \alpha (\alpha=3))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>g)</th>
<th>h)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>((3, +\infty))</td>
<td>((3, +\infty))</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Alpha-Beta Pruning Example

<table>
<thead>
<tr>
<th>i)</th>
<th>j)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>((3, +\infty))</td>
<td>((3, +\infty))</td>
</tr>
<tr>
<td>((3, +\infty))</td>
<td>((3, 14))</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>k)</th>
<th>l)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>((3, +\infty))</td>
<td>((3, +\infty))</td>
</tr>
<tr>
<td>((3, 5))</td>
<td>((3, 14))</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pruning happens: \(2 \leq \alpha (\alpha=3)\) but not much is pruned since we're at the bottom.
Effectiveness of Alpha-Beta

• Depends on order of successors
• Best case: Alpha-Beta reduces complexity from $O(b^m)$ for minimax to $O(b^{m/2})$
• This means Alpha-Beta can lookahead about twice as far as minimax in the same amount of time

Implementation Details

• In games we have the problem of transposition
• Transposition means different permutations of the move sequence that end up in the same position
• Results in lots of repeated states
• Use a transposition table to remember the states you’ve seen (similar to closed list)
What you should know

• Be able to draw up a game tree
• Know how the Minimax algorithm works
• Know how the Alpha-Beta algorithm works
• Be able to do both algorithms by hand