CS2212
Programming Challenge II

N. H. N. D. de Silva
Min Max Trees for a Deterministic game
Introduction

• Two-Person Perfect Information Deterministic Game
• Tree representation
• Utility function
Two-Person Perfect Information
Deterministic Game

- Two players take turns making moves
- Board state fully known, deterministic evaluation of moves -> Deterministic?
- One player wins by defeating the other (or else there is a tie)
- Want a strategy to win, assuming the other person plays as well as possible
Tree representation

- Search Trees are the underlying techniques of round based games with a (very) limited number of moves per round, e.g. BOARDGAMES
- A search tree contains a certain game state (position) in a single node, the children contain the possible moves of the single active player.
Utility function

- Evaluation of board/game state to determine how strong the position of player 1 is.
- Player 1 wants to maximize the utility function -> We’ll call him as **MAX** from now on
- Player 2 wants to minimize the utility function -> We’ll call him as **MIN** from now on
Minimax tree

• Basics
• Evaluation
• Optimizing
Basics

- Generate a new level for each move
- Levels alternate between “max” (player 1 moves) and “min” (player 2 moves)
The basic idea: compute (all) possible moves and evaluate the result (the leaves)
Basics: Notation

Max

Min

Max
Minimax Tree Evaluation

- Assign utility values to leaves
  - Sometimes called “board evaluation function”
  - If leaf is a “final” state, assign the maximum or minimum possible utility value (depending on who would win)
Minimax Tree Evaluation

- Assign utility values to leaves
  - If leaf is not a “final” state, must use some other heuristic, specific to the game, to evaluate how good/bad the state is at that point
  - The number shows how favourable is the board to player one. Larger the positive value greater is the winning edge. Larger the negative value, greater is the losing edge.
Minimax tree

Max

Min

Max

Min

X

O

X

O

X

23 28 21 -3 12 4 70 -4 -12 -70 -5 -100 -73 14 8 -24

100
Minimax Tree Evaluation

- At each min node, assign the minimum of all utility values at children
  - Player 2 chooses the best available move for him
- At each max node, assign the maximum of all utility values at children
  - Player 1 chooses best available move for him
- Push values from leaves to top of tree
Minimax tree

A low value is good for MIN, so MAX would choose the maximum-move!
A high value is good for MAX, so MIN would choose the minimum-move!
Minimax tree

Max

Min

Max

Min

Nodes with values: 28, 12, 70, 4, 100, 12, 70, 4, 100, 100, 73, 14, -8, -24

Values at leaf nodes: 23, 28, 21, -3, 12, 4, 70, -4, -12, -70, -5, -100, 73, 14, 8, 24

Max values: 28, 12, 70

Min values: 23, 28, 21, -3, 12, 4, 70, -4, -12, -70, -5, -100, 73, 14, 8, 24

Top node value: -3
Minimax Evaluation

- Given average branching factor $b$, and depth $m$:
  - A complete evaluation takes time $b^m$
  - A complete evaluation takes space $bm$
- Usually, we cannot evaluate the complete state, since it’s too big
  - Example: CHESS has an average branching factor of 35, so ...
- Instead, we limit the search based on various factors, including time available.
Optimizing

Idea 1: Limit Depth

The easiest idea, the worst playing skills

This is too obvious! So I am not going to explain!
Optimizing

Idea 2: alpha-beta pruning

A safe idea and a pure win!

Not so obvious. Let’s discuss.
α-β Pruning
Pruning the Minimax Tree

- Since we have limited time available, we want to avoid unnecessary computation in the minimax tree.
- Pruning: ways of determining that certain branches will not be useful
α Cuts

- If the current max value is greater than the successor’s min value, don’t explore that min subtree any more
\[ \alpha \text{ Cut example} \]

\[ \begin{align*}
\text{Max} & \quad 21 & -3 \\
\text{Min} & \quad -3 & -4 & -73 & -14 \\
\text{Max} & \quad 12 & 70 & 100 & -73 & -14
\end{align*} \]
α Cut example

- Depth first search along path 1
\( \alpha \) Cut example

- 21 is minimum so far (second level)
- Can’t evaluate yet at top level
-3 is minimum so far (second level)
-3 is maximum so far (top level)
- 12 is minimum so far (second level)
- -3 is still maximum (can’t use second node yet)
\( \alpha \) Cut example

- 12 is minimum so far (second level)
- -3 is still maximum (can’t use second node yet)
\( \alpha \) Cut example

- Since second level node will never be > -70, it will never be chosen by the previous level
- We can stop exploring that node
\( \alpha \) Cut example

- Evaluation at second level is -73
\( \alpha \) Cut example

- Again, can apply \( \alpha \) cut since the second level node will never be \( > -73 \), and thus will never be chosen by the previous level.
\( \alpha \) Cut example

- As a result, we evaluated the Max node without evaluating several of the possible paths
β cuts

- Similar idea to α cuts, but the other way around
- If the current minimum is less than the successor’s max value, don’t look down that max tree any more
Some subtrees at second level already have values $> \min$ from previous, so we can stop evaluating them.
\(\alpha-\beta\) Pruning

- Pruning by these cuts does not affect final result
  - May allow you to go much deeper in tree
- “Good” ordering of moves can make this pruning much more efficient
  - Evaluating “best” branch first yields better likelihood of pruning later branches
  - Perfect ordering reduces time to \(b^{m/2}\)
  - i.e. doubles the depth you can search to!
**α-β Pruning**

- Can store information along an entire *path*, not just at most recent levels!
- Keep along the path:
  - **α**: The best value achievable for MAX value found on this path, hence the maximum value so far. (initialize to most negative utility value)
  - **β**: The best value achievable for MIN value found on this path, hence the minimum value so far. (initialize to most positive utility value)
Pruning at MAX node

- $\alpha$ is possibly updated by the MAX of successors evaluated so far
- If the value that would be returned is ever $> \beta$, then stop work on this branch
- If all children are evaluated without pruning, return the MAX of their values
Pruning at MIN node

- $\beta$ is possibly updated by the MIN of successors evaluated so far
- If the value that would be returned is ever $< \alpha$, then stop work on this branch
- If all children are evaluated without pruning, return the MIN of their values
Idea of $\alpha$-$\beta$ Pruning

- We know $\beta$ on this path is 21
- So, when we get max=70, we know this will never be used, so we can stop here
Alpha Beta pruning is a pure win, but it’s highly dependent on the move ordering!
Utility Evaluation
Function
Utility Evaluation Function

- Very game-specific
- Take into account knowledge about game
- “Stupid” utility
  - 1 if player 1 wins
  - -1 if player 0 wins
  - 0 if tie (or unknown)
- Only works if we can evaluate complete tree
- But, should form a basis for other evaluations
Utility Evaluation

- Need to assign a numerical value to the state
  - Could assign a more complex utility value, but then the min/max determination becomes trickier
- Typically assign numerical values to lots of individual factors
  - \( a = \# \text{player 1’s pieces} - \# \text{player 2’s pieces} \)
  - \( b = 1 \) if player 1 has queen and player 2 does not, \(-1\) if the opposite, or \(0\) if the same
  - \( c = 2 \) if player 1 has 2-rook advantage, \(1\) if a 1-rook advantage, etc.
Evaluation functions

- If you had a *perfect* utility evaluation function, what would it mean about the minimax tree?
  
  You would never have to evaluate more than one level deep!

- Typically, you can’t create such perfect utility evaluations, though.
Evaluation Functions for Ordering

- As mentioned earlier, order of branch evaluation can make a big difference in how well you can prune.
- A good evaluation function might help you order your available moves:
  - Perform one move only
  - Evaluate board at that level
  - Recursively evaluate branches in order from best first move to worst first move (or vice-versa if at a MIN node)
Further Improvements: Quiescent search

- ‘Don’t leave a mess strategy’
- For evaluating the leaves at depth 0, instead of the evaluation function a special function is called that evaluates special moves (e.g. captures) only down to infinite depth
- Guarantees e.g. that the Queen will not be captured at move in depth 0
Further Improvements: Iterative deepening

- First try depth $n=1$
- If time left, try depth $n+1$
- Order moves of depth $n$ when trying depth $n+1$!
- Since alpha beta is order sensitive, this can speed up the process
- Fills time and doesn’t need predefined depth parameter
- Drawback: creates same positions over and over, but…
Further Improvements: Iterative deepening

- Example for multiply generated moves:

- Assumption: worst case: no alpha beta pruning.

  Branching factor 10

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Steps</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10+100</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>10+100+1000</td>
<td>1110</td>
</tr>
<tr>
<td>4</td>
<td>10+100+1000+10000</td>
<td>11110</td>
</tr>
<tr>
<td>5</td>
<td>10+...+100000</td>
<td>111110</td>
</tr>
<tr>
<td></td>
<td>=========</td>
<td>=========</td>
</tr>
<tr>
<td></td>
<td>111110 position</td>
<td>123,450 positions</td>
</tr>
</tbody>
</table>

123,450 / 111,110 = 1.11 => only 11% additional pos. (worst case)
Further Improvements: Aspiration Windows

- Extension of iterative deepening
- Basic Idea: feed alpha beta values of previous search into current search
- Assumption: new values won’t differ too much
- Extend alpha beta by +/- window value
References

- CIS 350 – I; Game Programming by Rolf Lakaemper
- Minimax Trees: Utility Evaluation, Tree Evaluation, Pruning
  originally by Yoonsuck Choe
Q & A