# Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

Martin Müller

Department of Computing Science
University of Alberta
mmueller@ualberta.ca

Fall 2022

## Today's Topics

Today's Topics:

- Minimax for win/draw/loss and numeric scores
- Alphabeta

## Coursework

- Work on Assignment 2
- Quiz 5: review minimax search parts 1 and 2.
  Double-length quiz
- Read Schaeffer et al, *Checkers is solved.* Science, 2007
- Activities for Lecture 9

## Minimax and Alphabeta

## Minimax Search: From Two to Many Different Outcomes

- Last time: boolean negamax solver
  for games with win-loss outcomes
- What about win-loss-draw?
- What about general numeric scores?
- Similar principles
  - A little bit more involved
  - Remember our setting:
    two player zero sum games, no chance element, perfect information
- Minimax search:
  - We maximize score
  - Opponent minimizes our score
- Zero-sum: each point we win, the opponent loses

## OR Node = MAX Node

- Our turn, we maximize
- Example, win-draw-loss game:
    - Set win-score $>$ draw-score $>$ loss-score
    - For example, can use
      win = +1, draw = 0, loss = -1
- OR node $n$, children $c_1$, ..., $c_k$
- score($n$) = max( score($c_1$), score($c_2$), ... score($c_k$))

## Example: Boolean OR and Maximum of 0, 1

- Example shows equivalence between
  - Logical OR
  - Taking the maximum of numbers in the set { 0, 1 }
- Booleans
  - True = we win
  - False = we lose
  - win($n$) = win($c_1$) **or** win($c_2$) **or ... or** win($c_k$)
  - win($n$) if win($c_i$) = True for at least one $i$
- Numbers in the set { 0, 1 }
  - 1 = we win
  - 0 = we lose
  - score($n$) = max( score($c_1$), score($c_2$), ... score($c_k$))
  - score($n$) = 1 if score($c_i$) = 1 for at least one $i$

## MAX Node with Numeric Scores

- Example: MAX node *n*
- Five children with scores 2, 5, -3, 6, 10
- score(*n*) = max(2, 5, -3, 6, **10**) = 10
- Question: Do we always have to evaluate all children now?

## MAX Node with Numeric Scores

- Example: MAX node *n*
- Five children with scores 2, 5, -3, 6, 10
- score(*n*) = max(2, 5, -3, 6, **10**) = 10
- Question: Do we always have to evaluate all children now?
- With scores, usually yes
- We can stop early in two scenarios
    - We know the highest possible score,
      and one child achieves it
      (similar to boolean case)
    - We have a *bound*, and only want to know
      if we can reach at least that bound.
      Can stop as soon as one child achieves bound

## Examples - Stopping Early in MAX Nodes

- Scenario 1: maximum possible score is say 1000
- score($c_1$) = 527
  - Keep searching...
- score($c_2$) = 1000
  - Reached maximum
  - No need to look at $c_3$, $c_4$...

- Scenario 2: we want to reach a bound, say at least 500
- score($c_1$) = 527
  - First child is good enough, stop.
  - No need to look at $c_2$, $c_3$...

## Examples - Stopping Early in MAX Nodes

- Scenario 1: maximum possible score is say 1000
- score($c_1$) = 527
  - Keep searching...
- score($c_2$) = 1000
  - Reached maximum
  - No need to look at $c_3$, $c_4$...

- Scenario 2: we want to reach a bound, say at least 500
- score($c_1$) = 527
  - First child is good enough, stop.
  - No need to look at $c_2$, $c_3$...
  - Question: What kind of boundedly rational decision-making solution is this an example of?

## AND Node = MIN Node

- Opponent minimizes among all their moves
- AND node $n$, children $c_1, ..., c_k$:
- score($n$) = min(score($c_1$), score($c_2$), ... score($c_k$))
- Compare win/loss case: $n$ is win iff all children are wins

## Boolean AND vs Computing Minimum

- Boolean AND is equivalent to taking MIN over { 0, 1 } scores
- Booleans
  - win($n$) = win($c_1$) **and** win($c_2$) **and** ... **and** win($c_k$)
  - win($n$) if win($c_i$) = True for all $i$
- Numbers in the set { 0, 1 }
  - score($n$) = min(score($c_1$), score($c_2$), ... score($c_k$))
  - score($n$) = 1 if score($c_i$) = 1 for all $i$

## Naive Minimax Search, General Case

- Similar to boolean case
- Compute max over all children in OR node
- Compute min over all children in AND node
- Two different functions `MinimaxOR`, `MinimaxAND`
- They call each other recursively
- Stop in terminal state, evaluate statically

## Naive Minimax Search - OR node

Changes to boolean minimax in **bold**

```
int MinimaxOR(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    int best = -INFINITY
    foreach legal move m from state
        state.Execute(m)
        int value = MinimaxAND(state)
        if (value > best)
            best = value
        state.Undo()
    return best
```

## Naive Minimax Search - AND node

```
int MinimaxAND(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    int best = +INFINITY
    foreach legal move m from state
        state.Execute(m)
        int value = MinimaxOR(state)
        if (value < best)
            best = value
        state.Undo()
    return best
```

## Negamax Search for Numbers

- Similar to boolean case
- Evaluation from current player's point of view
- Single `Negamax` function, calls itself recursively
- Negate result of children to change to current player's view
    - Result of children always from other player's view
- Compute the max of the negated results

## Naive Negamax Search - No Pruning

```
int Negamax(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluateForToPlay()
    int best = -INFINITY
    foreach legal move m from state
        state.Execute(m)
        int value = -Negamax(state)
        if (value > best)
            best = value
        state.Undo()
    return best
```

## Python Codes

- `minimax_sample_tree.py`, `minimax_sample_tree_data.py`
  artificial game tree to illustrate minimax and alphabeta
- `naive_minimax.py`, `naive_negamax.py`, `naive_minimax_negamax_test.py`
  Minimax and Negamax without any pruning, tests on sample tree

## Inefficiency of Plain Minimax/Negamax

- Inefficient. No pruning
  - In $(b, d)$ tree, searches all $b^d$ paths
  - Compare to efficient pruning in boolean case
- What's wrong? How can we prune moves?
- Revisit our two pruning scenarios above
  - One idea will be of limited use in practice
  - Other idea is very powerful, leads to alphabeta algorithm

## Pruning Idea From Earlier Scenario 1

- If maximum possible value is reached:
- Return directly, prune remaining moves
- Easy to implement
- Powerful with only two values { 0, 1 }
- May not help much if we have many scores
- It is rare to win by the maximum score in real games

```
int Negamax(GameState state)
    ...
        int value = -Negamax(state)
        if (value > best)
            best = value
            if best == MAXVALUE:
                return best
    ...
```

## Pruning Idea From Earlier Scenario 2

- Idea was: prune when reaching “good enough” value
- Reduces search to the boolean case
- What does “good enough” mean?
- Answer: better than a bound

We look at two cases

- First: bound is already given to us
- Second: compute, update bounds during the search
    - One bound for each player (alpha and beta)

## Reduce Minimax Search to the Boolean Case

- Assume we already have a candidate minimax value *m*
    - **Question:** Where might *m* come from?

## Reduce Minimax Search to the Boolean Case

- Assume we already have a candidate minimax value *m*
  - **Question:** Where might *m* come from?
- We can do **two** boolean searches to verify if *m* is the minimax result
- Remember: each terminal state will be evaluated with its score (a number)
- We replace those scores with a boolean win/loss result
  - win: score above a threshold *m*
  - loss: score below a threshold *m*
  - What about score = *m*?
    - It depends, see next slides

## Reduce Minimax Search to Two Boolean Searches

Assume we already have a candidate minimax value *m*

- First search: Can we get *at least m*?
  scores $v \geq m$ are wins,
  scores $v < m$ are losses
- Second search: Can we get *more than m*?
  scores $v > m$ are wins,
  $v \leq m$ are losses

If:

- Search 1 returns a win
- Search 2 returns a loss

Then: *m* **must** be the minimax value

## Understanding the Boolean Search Result(s)

- Given a candidate minimax value *m*
- Game can have three possible results: greater than, smaller than, equal to *m*
- What if Search 1 returns a loss?
    - Minimax value is smaller than *m*
    - Stop, no need for Search 2
- What if Search 1 returns a win?
    - Do Search 2
    - What if Search 2 also returns a win?
        - Minimax value is greater than *m*
    - Search 1 returns win, Search 2 returns loss:
        - Minimax value is equal to *m*

## Boolean Searches and Proof Trees

- Scenario:
  - Win with test ($v \geq m$)
  - Loss with test ($v > m$)
- Proof tree of the first search:
  - Our winning strategy: achieve at least *m*
- Disproof tree of the second search:
  - Opponent's winning strategy:
    prevent us from getting more than *m*
- Together, these two strategies prove that:
  - No player can do better than *m* ...
  - ... against a perfect opponent

## Example - Solve TicTacToe

- Example: solve TicTacToe
- Set win-score = 1, draw-score = 0, loss-score = -1
- Set $m$ = draw-score = 0
- First boolean search: test ($v \geq m$), can X draw-or-win?
  - Search result: **yes**
- Second boolean search: test ($v > m$), can X win?
  - Search result: **no**
- Together, both searches prove:
  - TicTacToe is a draw...
- See Python code
  `boolean_negamax_test_tictactoe.py`

## Discussion

- We learn something useful with both search outcomes
  - Search with boolean test ($v \geq m$) or ($v > m$)
  - Result True: lower bound on true minimax value
  - Result False: upper bound on true minimax value
- Important variants of alpha-beta search are based on this idea
  - SCOUT, NegaScout/PVS, MTD(f),...
- We will discuss the standard alpha-beta algorithm now
- Return to these ideas later
  - How to use boolean searches to speed up alpha-beta

## Alpha-beta Search

- Use if we have more than two outcomes, e.g. numeric score
- Idea: keep lower and upper bounds $(\alpha, \beta)$ on the true minimax value
- prune a position if its value *v* falls outside the $(\alpha, \beta)$ window
  - $v < \alpha$ we will avoid this position, we already found a better alternative
  - $v > \beta$ opponent will avoid this position, they already found a better alternative
  - If $v = \beta$ opponent can also ignore this position, they already found an equally good alternative

## Alpha-beta Search - Negamax Style

Changes from naive negamax in bold

```
int AlphaBeta(GameState state, int alpha, int beta)
    if (state.IsTerminal())
        return state.StaticallyEvaluateForToPlay()
    foreach legal move m from state
        state.Execute(m)
        int value = -AlphaBeta(state, -beta, -alpha)
        if (value > alpha) # alpha was 'best' in negamax
            alpha = value
        state.Undo()
        if (value >= beta)
            return beta
    return alpha
```

Initial call:

```
AlphaBeta(root, -INFINITY, +INFINITY)
```

## Negamax Alphabeta Details

- Negamax - everything is from *current player*'s point of view
- Avoids two separate cases for AND, OR nodes
- Negate scores when changing from player to opponent on each level, going down as well as going up in the tree
- Example: score +5 for player becomes -5 for opponent
- Window $(\alpha, \beta)$ becomes $(-\beta, -\alpha)$ for opponent
- Example:
    - window (+5, +10) for current player
    - window (-10,-5) for opponent
    - These are exactly the same window!
    - Imagine mirroring the window along $x = 0$ axis

## How does Alphabeta work? (1)

- Let $v$ be value of node, $v_1, v_2, ..., v_n$ values of children
- By definition: in OR node, $v = \max(v_1, v_2, ..., v_n)$
- Fully evaluated child establishes lower bound on parent
- Example: if $v_1 = 5$,
  - $v = \max(5, v_2, ..., v_n) \geq 5$
- Other moves of value $\leq 5$ do not help us
  - They can be pruned
- In code:
  - Set alpha to the best value so far
  - From now on, ignore moves of lesser (or equal) value

## How does Alphabeta work? (2)

- By definition: in AND node, $v = \min(v_1, v_2, ..., v_n)$
- Fully evaluated child establishes upper bound
- Example: if $v_1 = 2$,
    - $v = \min(2, v_2, ..., v_n) \leq 2$

## How does Alphabeta work? (2)

Main idea of pruning in alphabeta: the beta cut

- **`if (value >= beta) return beta`**
- The move is too good for the current player - cut.
- How can a move be too good?
- beta corresponds to -alpha for opponent one level up
- `value >= beta`
  is same as `-value >= -alpha` one level up for opponent
- That's the same as `value <= alpha` for opponent
- The opponent can already get alpha elsewhere,
  is not interested in achieving only `value` and will not play
  to here

## Python Codes for Alphabeta

- `tic_tac_toe_integer_eval.py`
  Static evaluation win = +1, draw = 0, loss = -1 instead of boolean evaluation at leaves
- `alphabeta.py`
  Algorithm, negamax style
- `alphabeta_test.py`
  Try on artificial game tree

## From Exact Search to Heuristic Search

- All our algorithms so far search each move sequence until the end of game
- This is needed for exact solver
- Heuristic play:
    - Stop search earlier (e.g. at depth of *d* moves)
    - Evaluate “leaf” node by a heuristic
- Depth-limited searches can be good for move ordering
- Idea (details later):
  iterative deepening, increase depth 1, 2, 3,...
- Next slide: alphabeta with depth limit

## Depth-limited Alpha-beta Search

```
int AlphaBeta(GameState state, int alpha, int beta, int depth)
    if (state.IsTerminal() OR depth == 0)
        return state.StaticallyEvaluateForToPlay()
    foreach legal move m from state
        state.Execute(m)
        int value = -AlphaBeta(state, -beta, -alpha, depth - 1)
        if (value > alpha)
            alpha = value
        state.Undo()
        if (value >= beta)
         return beta // or value - see failsoft
    return alpha
```

Python code: `alphabeta_depth_limited.py`, `alphabeta_depth_limited_tictactoe_test.py`

## Summary

- From boolean case to numeric scores
- Naive Minimax and naive Negamax search
- Boolean searches to prove bounds on numeric scores
- Alphabeta search cuts off useless branches, much more efficient
- Next time:
  - Search improvements for boolean negamax and alphabeta
  - Search on DAGs
  - Reduce search depth in Go endgame solver