Computing Science (CMPUT) 657
Algorithms for Combinatorial Games

Martin Müller

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

Winter 2022

Minimax search

- Why minimax?
- AND/OR game tree model, proof tree
- Solving games - minimax principle
- Boolean minimax solver
- Alphabeta solver
- Heuristic game player with alphabeta

Why Minimax?

- Setting: Two player adversarial zero-sum games
- Possible results of game are ordered
- Example 1: win $>$ draw $>$ loss
- Example 2: result is numeric score, larger is better
- Adversarial games: the opponent wants the *opposite* result
- Example: for them it is best if *we* lose

Why Minimax? (2)

- Notation for result: o(player)
- Classical setting: *zero sum* game
- o(player1) + o(player2) = 0
- o(player2) = - o(player1)
- Assume we are player1
- Example 1:
  - win (for us): o(player1) = +1, o(player2) = -1
  - draw: o(player1) = o(player2) = 0
  - loss (for us) o(player1) = -1, o(player2) = +1
- Example 2: result is money won
  - Example: o(player1) = $12000
  - Zero sum: o(player2) = $-12000

Why Minimax? (3)

- How to play such games?
- Assume we have different moves to choose from
- We should choose the one with **max**imum result (for us)
- The opponent will also choose a move with **max**imum result (for them)
- Zero sum:
  their maximum result is our **min**imum result
- Examples soon

Concept: AND/OR Tree

- Formalizes concept of game tree with alternating players
- Example: logic for “can we win?”
- OR node: player’s turn - can win if:
  - move 1 wins OR ...
  - move 2 wins OR ...
  - move *n* wins
- AND node: opponent’s turn - player wins only if opponent’s move 1 AND move 2 AND... all win (for player)
- Applications outside of games: Goals expressed recursively as conjunction or disjunction of subgoals.

AND/OR Tree (2)

- Normal form: alternating layers of AND, OR nodes
- Any AND/OR tree has an equivalent normal form (merge children of node with same type, or introduce dummy nodes with only one child)
- Generalization: AND/OR on a DAG (directed acyclic graph) - later in this class
- How about general (cyclic) graphs? Has problems with loops, cyclic definition of results
- Optional, detailed discussion: watch Dasgupta lecture (see resources)

Non-Game AND/OR Example

- Assembling a car
- OR node: choice between different alternatives, e.g. different suppliers of components
- AND node: all the ingredients of a car
- AND example: need engine AND 4 tires AND windscreen AND ...
- OR example: get tires from Michelin OR Bridgestone OR Pirelli OR ...

Proof Tree

- A winning strategy for a player
- Dual concept: disproof tree - proves that player loses
- Subset of game tree
- Covers all possible opponent replies

Definition of Proof Tree

- Subtree P of game tree G is proof tree iff:
- P contains root of G
- All leaf nodes of P are wins
- If interior AND node is in P, then *all its children* are in P
- If interior OR node is on P, then *at least one child* is in P

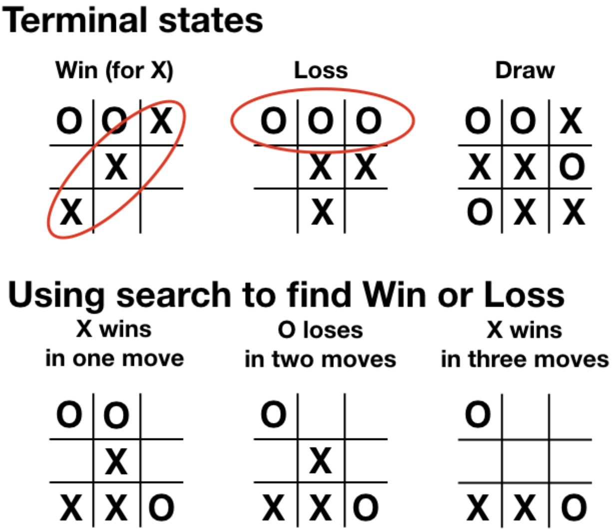
Comments on Proof Tree

- Same definitions work on DAG, even on arbitrary graph
- Terminology: solution tree (in optional read - Pijls/de Bruin paper)
- Efficiency: want to find *minimal* or at least a small proof tree

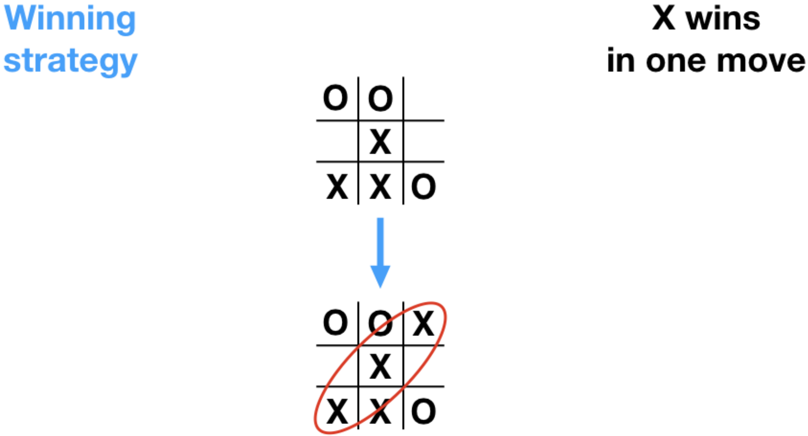
Comments on Proof Tree (2)

- In uniform (b, d) tree, with OR node at root, number of nodes in best case at each level is $1, 1, b, b, b^2, \ldots$
- Search is most efficient if it looks only at the proof tree
- In practice, that's impossible...
- Good move ordering is crucial to get close to optimal
- Ideal: Small number of moves (close to 1) expanded before a winning one is found

Wins, Losses and Draws

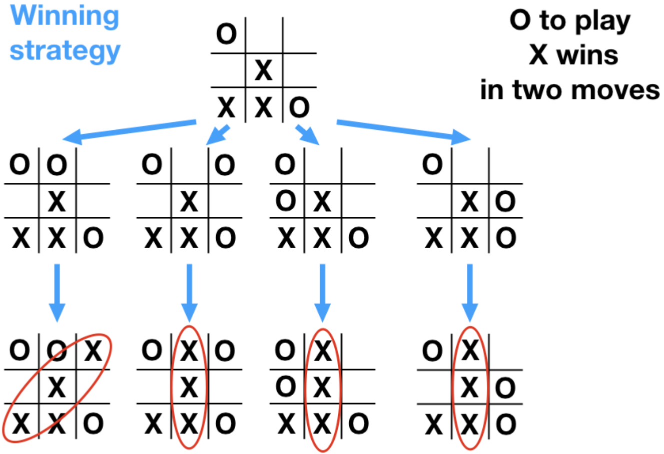


Winning Strategy - Depth 1



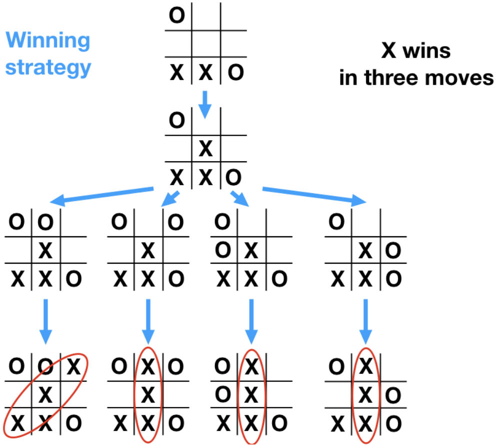
- X can win in one move
- Winning strategy just contains that move

Winning Strategy - Depth 2



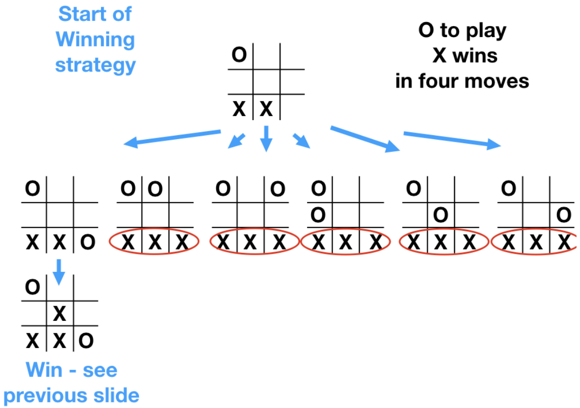
- Winning strategy: d=1: **all opponent moves**, d=2: one reply for each → win

Winning Strategy - Depth 3



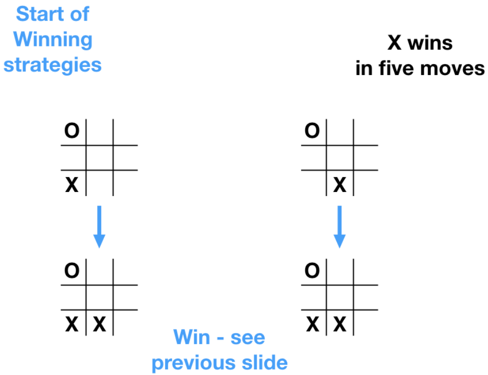
- d=1: One move, d=2: one branch for each opponent reply,
  d=3: one move in each branch $\rightarrow$ win

Winning Strategy - Depth 4



- d=1: all opponent moves, d=2: one move, leads to a known winning position

Winning Strategies - Depth 5



- d=1: One move in each case,
  both lead to a known winning position

What if the State Space is a DAG?

- Exactly the same concepts work in DAG
- Difference in practice:
- We can store and share wins and losses computed earlier
- Different paths to reach the same node
- Only prove a win (or loss) for a node once, then remember
- Main technique:
  hash table
  also called *transpositon table*
- Details later

Minimax Algorithm - Boolean Version

- Each player tries to win. Zero-sum - opponent's win is my loss
- OR node: If I have at least one winning move, I can win (by playing that move)
- If all my moves are losses, I lose.

```
// Basic Minimax with boolean results
bool MinimaxBooleanOR(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    foreach successor s of state
        if (MinimaxBooleanAND(s))
            return true
    return false
```

Minimax Algorithm - Boolean Version

- Each player tries to win. Zero-sum - opponent's win is my loss
- AND node: All my moves need to be winning
- If any of my moves are losses, I lose.

```
// Basic Minimax with boolean results
bool MinimaxBooleanAND(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    foreach successor s of state
        if (NOT MinimaxBooleanOR(s))
            return false
    return true
```

Minimax Algorithm - Boolean Version (2)

- Less abstract version showing execute, undo move

```
// Minimax, boolean results, execute/undo moves
bool MinimaxBooleanOR(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    foreach legal move m from state
        state.Execute(m)
        bool isWin = MinimaxBooleanAND(state)
        state.Undo()
        if (isWin)
            return true
    return false
```

Minimax Algorithm - Boolean Version (2b)

- Less abstract version showing execute, undo move

```
// Minimax, boolean results, execute/undo moves
bool MinimaxBooleanAND(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    foreach legal move m from state
        state.Execute(m)
        bool isWin = MinimaxBooleanOR(state)
        state.Undo()
        if (NOT isWin)
            return false
    return true
```

From Minimax to Negamax

- Comment: all evaluation in `StaticallyEvaluate()`, `MinimaxBooleanOR(s)` and `MinimaxBooleanAND(s)` is from the *fixed* root player's point of view
- Sometimes it is more natural to evaluate from the point of view of the current player
- $\Rightarrow$ Negamax formulation of minimax search
- current player changes with each move - negate result of recursive call

Negamax Algorithm - Boolean Version


```
// Negamax, boolean results, execute/undo moves
bool NegamaxBoolean(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
        // evaluate from toPlay's point of view
    foreach legal move m from state
        state.Execute(m)
        bool isWin = NOT NegamaxBoolean(state)
        state.Undo()
        if (isWin)
            return true
    return false
```

Boolean Minimax - Discussion

- Basic recursive algorithm
- Runtime depends on:
  - depth of search
  - width (branching factor)
  - **move ordering** - stops when first winning move found
- Easy to compute all winning moves instead - add top-level loop
- Questions (for later): best-case, worst-case performance?

Boolean Minimax - Discussion (2)

- Boolean case is simpler special case of minimax search
- Efficient pruning - stops as soon as win is found
- Important tool used in other algorithms

Boolean Minimax - Discussion (3)

- What is the runtime? Depends on *move ordering*
- Simple model: uniform tree, *depth d*, *branching factor b*
- What is best case, worst case?

Best Case For Boolean Minimax Search

- Search is most efficient if it looks only at the proof tree
- This means, at OR nodes we only look at a winning move
  - We never look at a non-winning move first
- In practice, that's usually impossible - too hard.
- Good *move ordering* is crucial for efficient search
- We can use good *move ordering heuristics*, including *iterative deepening* techniques based on successively deeper searches

Boolean Minimax - Efficiency

- Best case: about $b^{d/2}$, first move causes cutoff
- Worst case: about b^d, no move causes cutoff
- We can do exact calculation in class

Minimax search

- General case - score of terminal position can be any finite number
- Frequent special case: small set of values, e.g. win-draw-loss
- Minimax: We try to **max**imize our score, opponent tries to **min**imize it
- Zero-sum: each extra 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 scores
      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

Minimax Search - OR node


```
int MinimaxOR(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
        \\ evaluate from root player's view
    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
```

Minimax Search - AND node


```
int MinimaxAND(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
        \\ evaluate from root player's view
    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 Formulation of Minimax Search

```
int Negamax(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
        \\ evaluate from toPlay's view
    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
```

Comments on Plain Minimax/Negamax

- VERY inefficient
  - No pruning, as opposed to boolean case above
  - In (b, d) tree, searches all b^d paths, all $\sum b^i$ nodes
- How can we add pruning?
- Simple idea: prune if max. value reached (usually does not help much)

Pruning for Minimax/Negamax

- Two main ideas:
- Reduce to boolean case
- Alpha-beta: update upper and lower bounds on value during search, use for pruning

Reduce to Boolean Case

- Assume we already have a candidate minimax value m (to discuss: where might m come from?)
- We can do two boolean searches to verify if m is the minimax result
  1. Search if ($v \geq m$) is a win
  2. Search if ($v > m$) is a loss
  3. if search with $v \geq m$ succeeds, but $v > m$ fails, then m must be the minimax value (discuss: why?)
- even if a search fails, we learn something (upper or lower bound on true value)
- Important alpha-beta refinements are based on this idea: SCOUT, NegaScout/PVS
- Discuss alphabeta now, return to those ideas then.

Alpha-beta Search

- Idea: keep upper and lower bounds (α, β) on the true minimax value
- prune a position if its value *v* falls outside the window
  1. $v < \alpha$ we will avoid it, we have a better alternative
  2. $v > \beta$ opponent will avoid it, they have a better alternative
  3. If $v = \beta$ we can also ignore this line (Think about why)

Alpha-beta Search - Code

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

- This is a negamax formulation.
- Initial call: `AlphaBeta(root, -INFINITY, +INFINITY)`

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, $v = \max(v_1, v_2, ..., v_n)$
- By definition: in AND node, $v = \min(v_1, v_2, ..., v_n)$
- Fully evaluated moves establish lower bound
- E.g. if $v_1 = 5$, $v = \max(5, v_2, ..., v_n) \geq 5$
- Other moves of value ≤ 5 do not help us, can be pruned

How does Alphabeta Work? (2)

- Similar reasoning at AND node - moves establish upper bound
- E.g. $v_1 = 2$, $v = \min(2, v_2, ..., v_n) \leq 2$
- If a move leads to position that is too bad for one of the players, then cut.

Alphabeta Trace Example

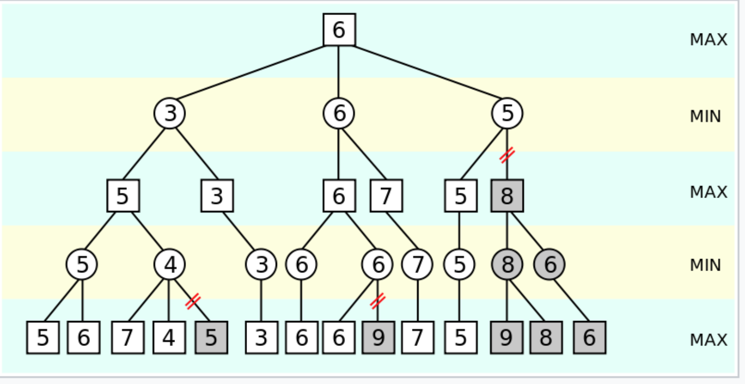


Image source: `https://en.wikipedia.org/wiki/Alpha-beta_pruning`

Let's trace on Whiteboard

Full Search vs Heuristic Search

- So far: search till end of game
- Needed for exact solver
- For heuristic play, can stop search earlier (e.g. after *d* moves)
- Depth limited searches need an evaluation function
- They can also give good move ordering - iterative deepening idea
- Here is alphabeta code with depth limit

Depth-limited Alpha-beta Search - Code


```
int AlphaBeta(GameState state, int alpha, int beta, int depth)
    if (state.IsTerminal() OR depth == 0)
        return state.StaticallyEvaluate()
    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
```