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:

- Quiz 3 review
- Assignment 2 preview
- Solving two-player games, TicTacToe example
- Winning strategy
- Concepts for game trees: OR nodes, AND nodes
- Minimax algorithm for the boolean (win/loss) case

Coursework

- Read Schaeffer et al, Checkers is solved.
- Activities 8
- Start Assignment 2 - NoGo endgame solver
  - You may change teams, but if so you must email **Abbas**

Assignment 2 - NoGo Endgame Solver

- Assignment 2
- Specification published now
- Preview in class today
- Starter code - plays legal random NoGo
- Some functions pre-implemented, some you must implement
- Indicated in the code - read it now to get an idea

Python Sample Codes

- Study TicTacToe solver now to prep for assignment 2
- See python code page

Assignment 2 Preview

- Goal: write a perfect solver for NoGo endgames
- Assignment description:
  `https://webdocs.cs.ualberta.ca/~mmueller/courses/cmput455/assignments/a2.html`
- We will start talking about the concepts and algorithms in class today

Assignment 2 Preview

- You are given a “clean” NoGo random player
- You are also given sample code for solving TicTacToe
  See python sample code for lectures 8, 9 and 10

Assignment 2 Preview

- As before, we will have both public and private test cases
- There will be bonus marks for solvers that are significantly faster than ours
- Same presubmission and early feedback procedure as in assignment 1
- By default: same teams. Let TA know of any changes to teams

Solving Games with Minimax Search

Solving Games

- What does solving a game mean?
- Find the correct outcome of the game
  - With best play...
  - ...by both players.
- Different kinds of solving
  - Ultra-weakly solved: know the outcome, but have no concrete strategies
  - Weakly solved: contains a winning strategy starting from the initial state
  - Strongly solved: provide an algorithm that can win from any winnable position of the game
- **Question:** Why would we want an algorithm for starting from *any* position?

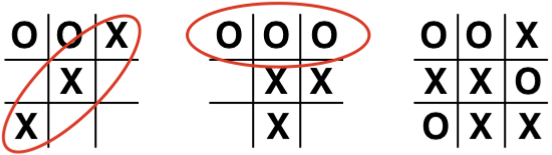
Solving Games

- *How* to play if we have a win?
- Need a *winning strategy*
- Start with TicTacToe example
  - Game rules: 3×3 board, Black places X, White places O
  - Win: 3 in a row horizontally, vertically, or diagonally
  - Draw: board full, no player got three in a row

Wins, Losses and Draws

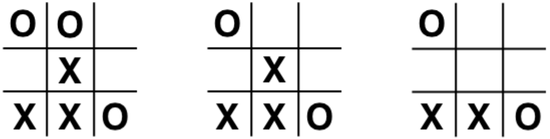
Terminal states

Win (for X) | Loss | Draw



Using search to find Win or Loss

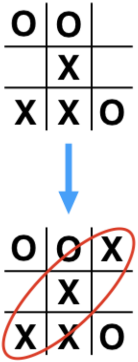
X wins in one move | O loses in two moves | X wins in three moves



Winning Strategy - Depth 1

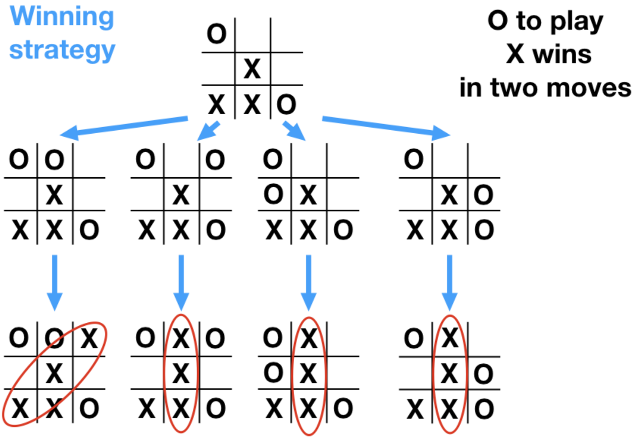
Winning strategy

X wins in one move



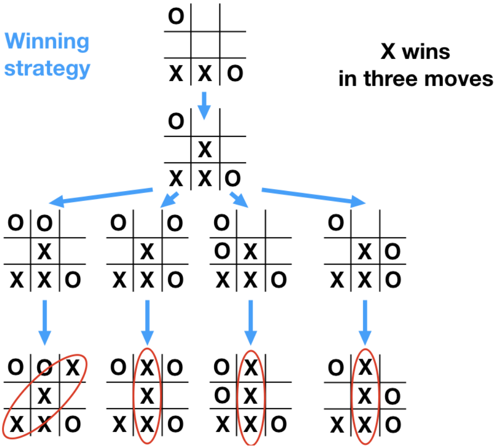
- X can win in one move
- Winning strategy just contains that move

Winning Strategy - Depth 2



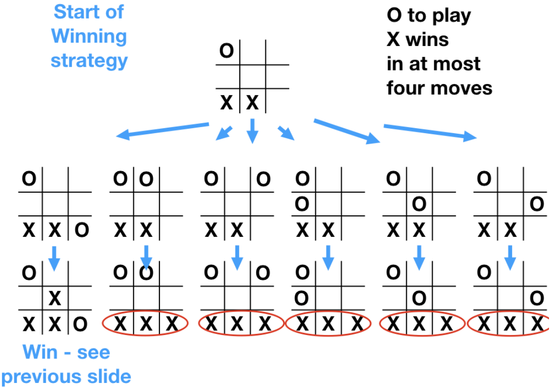
- Winning strategy:
  d=1: include **all opponent moves**
  d=2: one winning move for us in each branch $\rightarrow$ we win

Winning Strategy - Depth 3



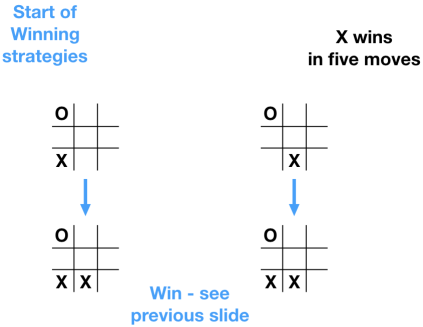
- d=1: One move for us
  d=2: one branch for each possible opponent reply
  d=3: one winning move in each branch $\rightarrow$ we win

Winning Strategy - Depth 4



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

Winning Strategies - Depth 5 Examples



- Two examples
- d=1: One move in each example,
  both lead to the same winning position (from previous slide)

What a Winning Strategy Looks Like

- In a game, if it is our turn, we have a choice
- Play move1 or move2 or...
  - It is enough to know **one** winning move
- If it is the opponent's move,
  we need to win against *all* their moves
- Win against move1 and move2 and ...
  - We need to include *all* their moves in our strategy

Winning Strategy as a Tree (or DAG)

- Consequence: the winning strategy is a tree (or DAG)
- The winning strategy includes:
- One move when it's our turn
- All moves when it's the opponent's turn
- The tree (or DAG) of a winning strategy is much smaller than the whole state space
- **Why?**

Winning Strategy as a Tree (or DAG)

- Consequence: the winning strategy is a tree (or DAG)
- The winning strategy includes:
- One move when it's our turn
- All moves when it's the opponent's turn
- The tree (or DAG) of a winning strategy is much smaller than the whole state space
- **Why?**
  - It branches at every *second* level
    - Branch only when it's the loser's turn
    - The whole strategy can still be very large

Proving a Win

- To prove a win we need to find a winning strategy
- Usually, we do not store it
  - We just use search to prove a win (see later)
- Usually, we build a strategy top-down from the root
- Conceptually, we can also build a strategy bottom-up from the end
- First question:
  - What are winning terminal positions?
  - The rules of the game give the answer

Evaluation of Terminal Positions

Game over, what's the result?

Different Types of evaluation:

- Simplest case: binary (or boolean) evaluation, win-loss
  - Examples: Coin toss, Go with non-integer komi, NoGo
- Popular case: win-draw-loss
  - Examples: TicTacToe, chess, checkers, 5-in-a-row
  - Go with integer komi
- More general case: games with score
  - Examples: win by 5 points
  - Win $10,000,000

Tic Tac Toe Sample Code

- `tic_tac_toe.py` has board representation, rules
- Similar to `Go1`, board stored in 1-d array of size 9
- Status codes
  `EMPTY = 0, BLACK = 1` for 'X', `WHITE = 2` for 'O'
- Useful functions `legalMoves, endOfGame, play, undoMove,...`

Board indexing:

```
0 1 2
3 4 5
6 7 8
```

AND/OR Tree

In a game tree:

- Position where it is our turn:
  OR node
- Position where it is the opponent's turn:
  AND node
- Alternating play
  - Each move **from an OR node** leads **to an AND node**
  - Each move **from an AND node** leads **to an OR node**

Leaf Nodes

- *Leaf nodes* are *terminal states* of the game
- Game is over
- Can determine the result from the rules
- Examples:
  - Count the score in Go $\rightarrow$ winner
  - TicTacToe 3-in-a-row $\rightarrow$ win
  - TicTacToe board full, no 3-in-a-row $\rightarrow$ draw

Winning in an OR Node

- Our turn
- Finding one winning move is enough
- OR node n
- Children $c_1, \ldots, c_k$
- win(n) = win(c_1) **or** win(c_2) **or ... or** win(c_k)
- Shortcut evaluation:
  can stop at the *first* child that is a win
- We can play that move to win from n
- Example: c_1, c_2, c_3 all lose, c_4 wins - can stop here
- Best case for search: the first child c_1 is a win

Winning in an AND Node

- Opponent's turn
- We win only if we win after *all* opponent moves
- AND node n
- Children $c_1, ..., c_k$
- win(n) = win(c_1) **and** win(c_2) **and ... and** win(c_k)
- Shortcut evaluation:
  can stop at the *first* child that is a loss
- The opponent can play that move to make us lose from n
- Best case for search: the first child c_1 is a loss
  - **Question:** Why is a *loss* the best case?

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
- Tic Tac Toe example earlier: two depth five wins by moving to the same win-in-4 position
- Prove win-in-4 position once, use for two different lines

Re-using proofs in a DAG

- Main technique to store states and results:
- *Hash table*
- Also called a *transposition table*
- How to use in search: details later

Boolean Minimax Algorithm

Minimax Algorithm - Boolean Version - OR Node

- 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 lose, I lose.

```
// Basic Minimax with boolean outcomes
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 - AND Node

- AND node: All opponent moves need to win for me
- If any of their moves lose me the game, I lose.

```
// Basic Minimax with boolean outcomes
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 pseudocode showing execute, undo move
- Python3 code `boolean_minimax.py`

```
// Minimax, boolean outcomes, execute/undo
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
```

Boolean Minimax Algorithm - AND Node

- Less abstract version showing execute, undo move

```
// Minimax, boolean outcomes, execute/undo
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
```

Negamax Algorithm - Main Idea

- All evaluation in `StaticallyEvaluate()`, `MinimaxBooleanOR(s)` and `MinimaxBooleanAND(s)` is from a *fixed* player's point of view
- We can also 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
- My win is your loss, my loss is your win

Negamax Algorithm - Boolean Version

```
// Negamax, boolean outcomes
bool NegamaxBoolean(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluateForToPlay()
    foreach legal move m from state
        state.Execute(m)
        bool isWin = NOT NegamaxBoolean(state)
        state.Undo()
        if (isWin)
            return true
    return false
```

Python Implementation and Solve TicTacToe

- Boolean negamax solver `boolean_negamax.py`
- Use to solve TicTacToe:
  `boolean_negamax_test_tictactoe.py`
- Main question: how to handle draws?
- Boolean solver only deals with two outcomes
- We can choose whether draws should count for Black or White
  - In TicTacToe code: function `setDrawWinner`
- More on this topic next class

Boolean Minimax - Discussion

- Basic recursive algorithm
- Runtime depends on:
  - depth of search
  - width (branching factor)
  - **move ordering** - stops when first winning move found
- Easy modification to compute *all* winning moves
  - Add a top-level loop which does not stop at the first win
- Questions: 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 more advanced algorithms later
- What is the runtime? Depends on *move ordering*
- Simple model: uniform tree, *depth d*, *branching factor b*
- What is best case, worst case?

Boolean Minimax - Efficiency

- Best case: about $b^{d/2}$, first move always causes cutoff for winning side
- *Cutoff* = early return from function because we found a move that works
  - Exact calculation for best case - a little later
- Worst case: about b^d, no move causes cutoff
- Exact number: Visits all nodes in the tree, count as before:

$$1 + b + b^2 + ... + b^d = (b^{d+1} - 1)/(b - 1)$$

Proof Tree

- A winning strategy for a player
- Dual concept: disproof tree
  - proves that we lose, cannot win
- A subset of a game tree
- Gives us a winning move in each position we may encounter (as long as we follow the strategy...)
- Covers all possible opponent replies at each point when it's their turn

Definition of Proof Tree

Definition (Proof tree)

A subtree P of game tree G is a *proof tree* iff all of the following are true:

- P contains the 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

- Exactly the same definitions work on DAG, even on arbitrary graph
- Another name for proof tree: solution tree
- Efficiency:
  want to find a *minimal* or at least a small proof tree

Size of Proof Tree

- Scenario: uniform (b, d) tree, OR node at root, we win
- How many nodes at each level?
- Level 0: 1 node (root)
- Level 1: $\geq$ 1 nodes (at least one child...), best case 1
- Level 2: $\geq b$ nodes (all children of level 1 nodes), best case b
- General pattern for best case: $1, 1, b, b, b^2, b^2, b^3, b^3, ...$
- Activities: Find formulas for size of proof trees in the best 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
  - Compare with heuristic in treasure hunt example, Lecture 7
- We can use good *move ordering heuristics*, or techniques based on successively deeper searches
- More later

Summary of Solving Games

- Concepts: winning strategy, AND/OR trees
- Solving OR nodes, AND nodes
- Boolean Minimax and Negamax
- Efficient pruning of tree - stop at first winning move
- Good move ordering finds that first move faster