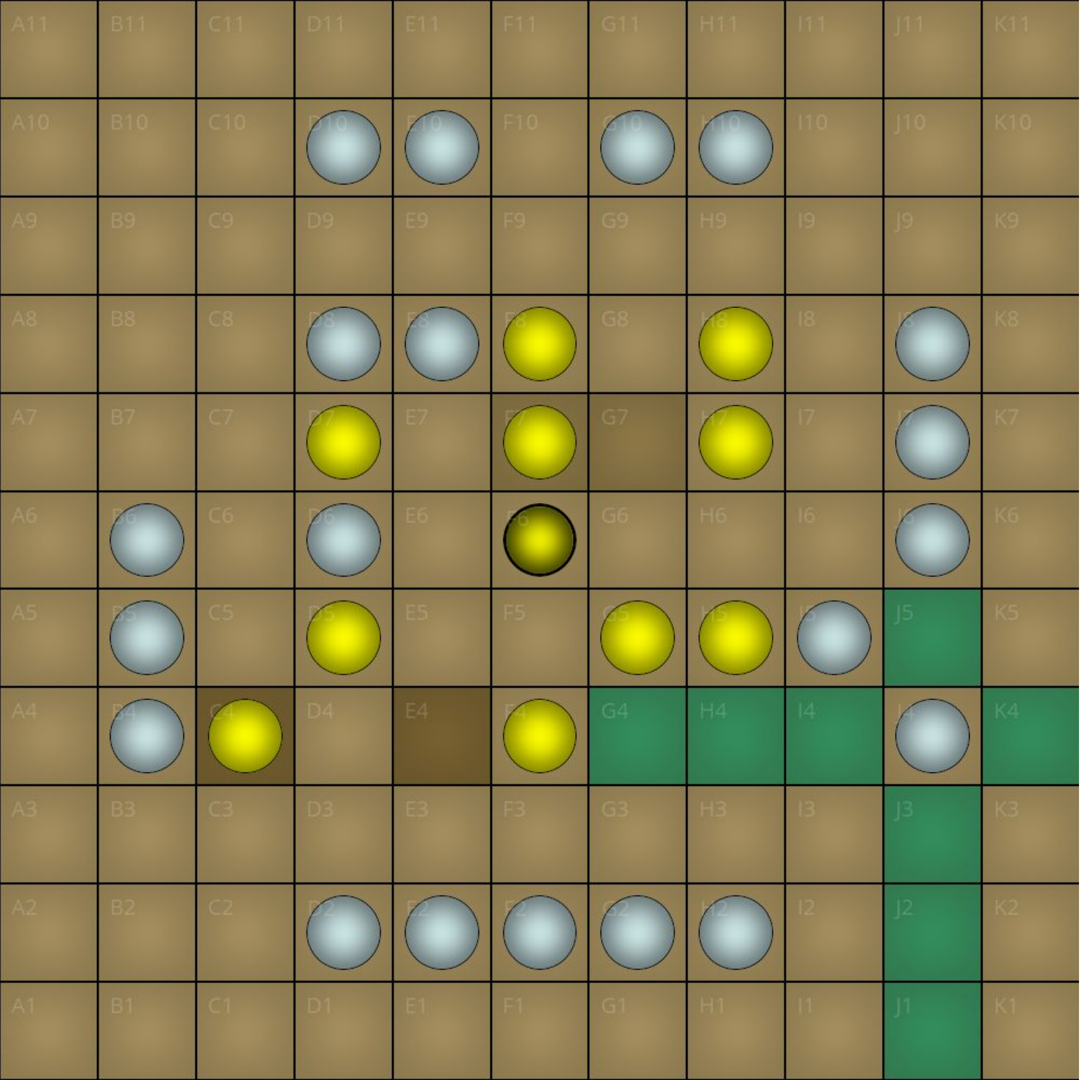

A11 B11 C11 D11 E11 F11 G11 H11 I11 J11 K11
A10 B10 C10 D10 E10 F10 G10 H10 I10 J10 K10
A9 B9 C9 D9 E9 F9 G9 H9 I9 J9 K9
A8 B8 C8 D8 E8 F8 G8 H8 I8 J8 K8
A7 B7 C7 D7 E7 F7 G7 H7 I7 J7 K7
A6 B6 C6 D6 E6 F6 G6 H6 I6 J6 K6
A5 B5 C5 D5 E5 F5 G5 H5 I5 J5 K5
A4 B4 C4 D4 E4 F4 G4 H4 I4 J4 K4
A3 B3 C3 D3 E3 F3 G3 H3 I3 J3 K3
A2 B2 C2 D2 E2 F2 G2 H2 I2 J2 K2
A1 B1 C1 D1 E1 F1 G1 H1 I1 J1 K1

Breakthru

Intelligent Search & Games
Lecturer Mark Winands
Project Submission
David Pomerenke

Structure

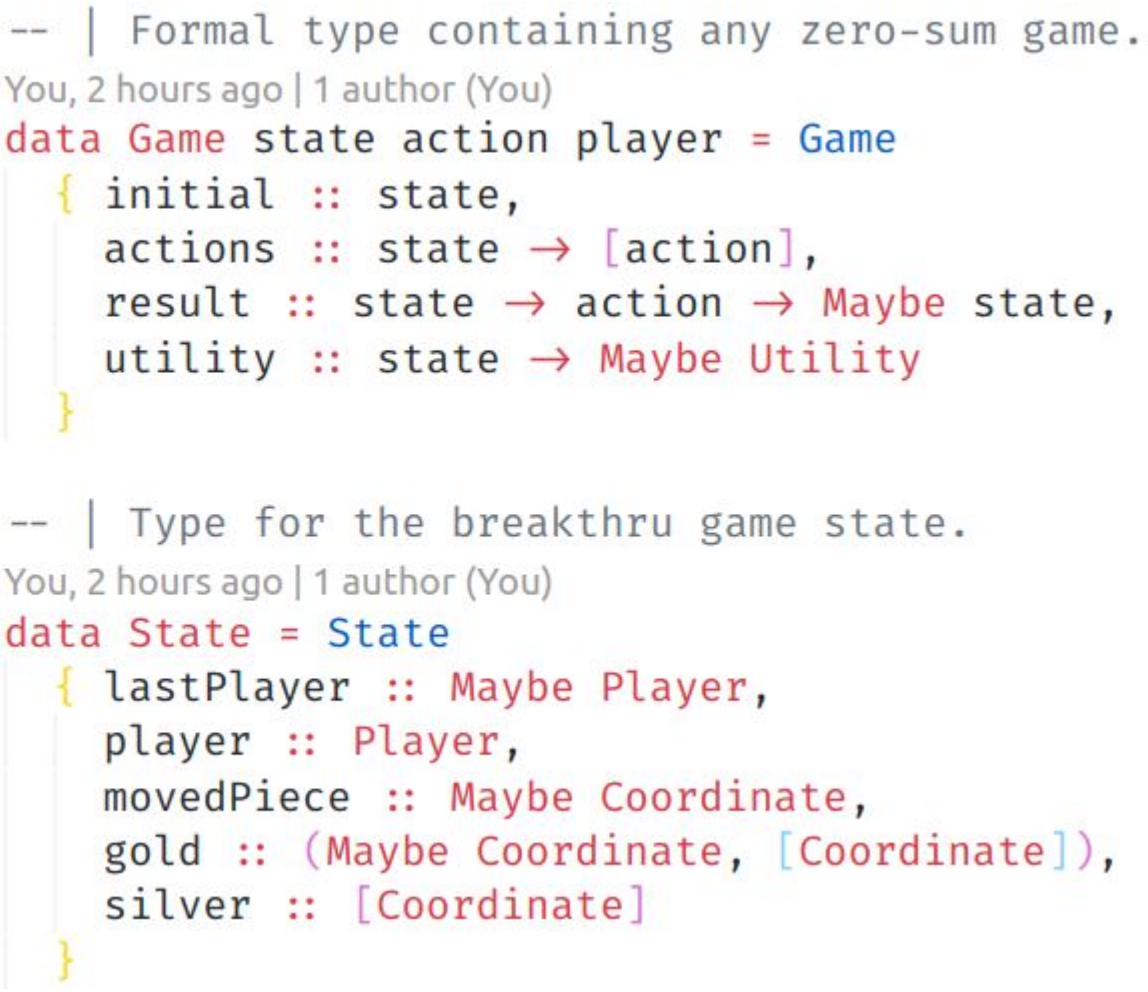
- Architecture
- Rules
- Agents
    - Random
    - Minimax
    - Alpha-Beta
- User Interface
- Evaluation
- Gameplay Demonstration

Architecture

- Elm Frontend /static
  - Display board, available actions
  - Animate movements & keep history
  - Query API for actions, results, AI moves (manual JSON)
  - Advantage: Correctness, rapid development
- Haskell Server /src
  - Provide API for actions, results, AI moves (automatic JSON)
  - Evaluation of agents
  - Advantage: High level, speed, parallelism
  - Disadvantage: Algorithms need to be rewritten
- Python Evaluation /evaluation
  - Visualize results

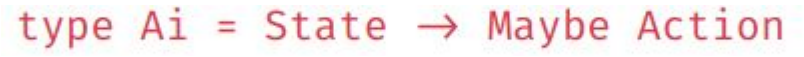
Rules

Game.hs



```
-- | Formal type containing any zero-sum game.
You, 2 hours ago | 1 author (You)
data Game state action player = Game
  { initial :: state,
    actions :: state → [action],
    result :: state → action → Maybe state,
    utility :: state → Maybe Utility
  }

-- | Type for the breakthru game state.
You, 2 hours ago | 1 author (You)
data State = State
  { lastPlayer :: Maybe Player,
    player :: Player,
    movedPiece :: Maybe Coordinate,
    gold :: (Maybe Coordinate, [Coordinate]),
    silver :: [Coordinate]
  }
```

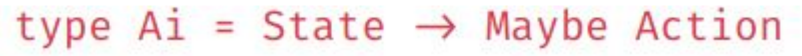
Agents



```haskell
type Ai = State → Maybe Action
```

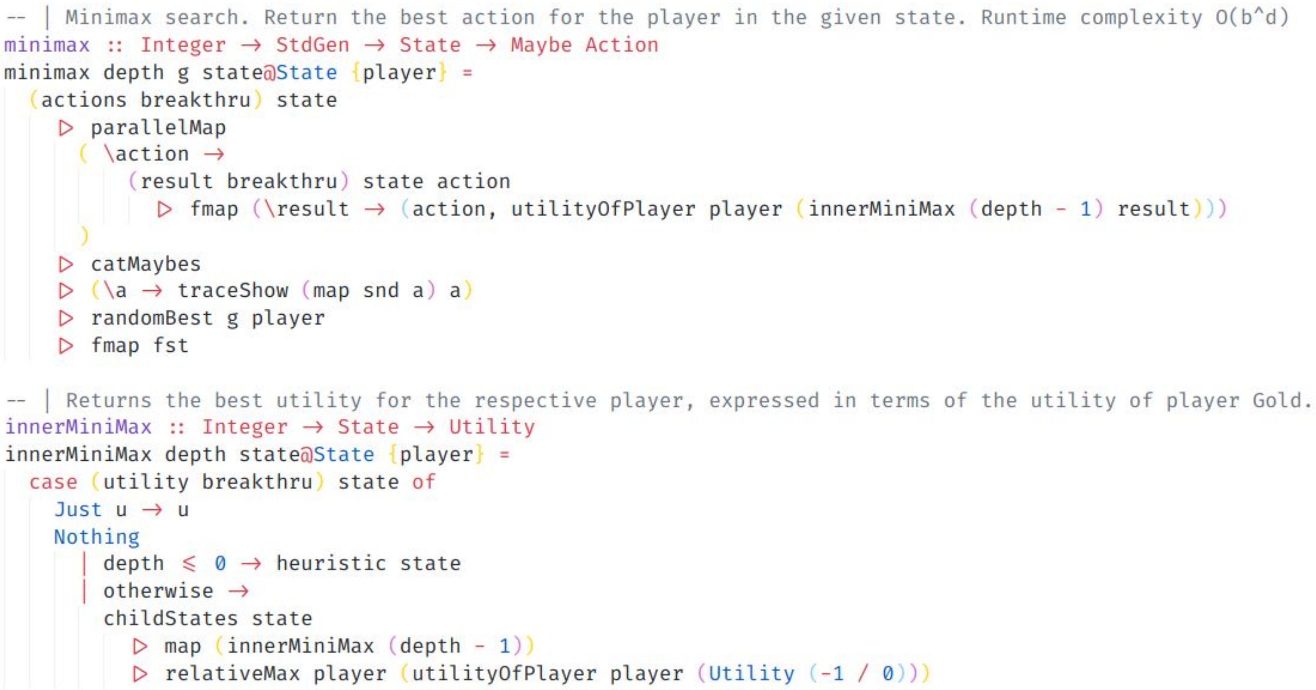
- Random (Helpers.hs)
    - Retrieve actions, select random element
    - Deterministic randomness
- Non-random
    - Evaluation function: #Gold / #max Gold - #Silver / max Silver

Agents



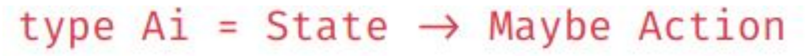
```
type Ai = State → Maybe Action
```

- Minimax (Minimax.hs)



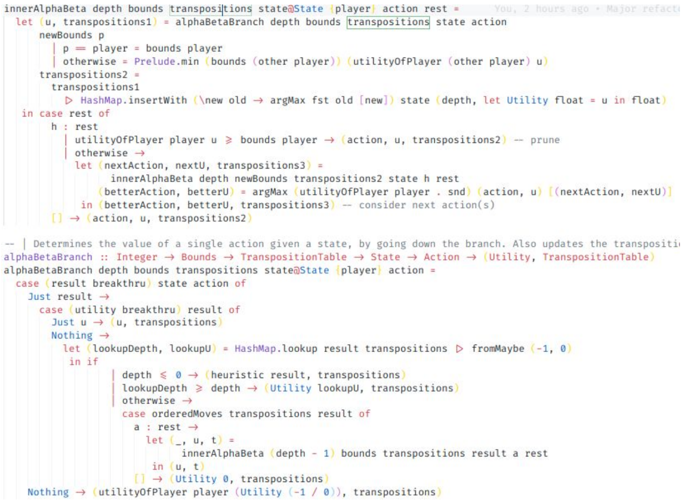
```
-- | Minimax search. Return the best action for the player in the given state. Runtime complexity O(b^d)
minimax :: Integer → StdGen → State → Maybe Action
minimax depth g state@State {player} =
  (actions breakthru) state
    ▷ parallelMap
      ( \action →
          (result breakthru) state action
            ▷ fmap (\result → (action, utilityOfPlayer player (innerMiniMax (depth - 1) result)))
      )
    ▷ catMaybes
    ▷ (\a → traceShow (map snd a) a)
    ▷ randomBest g player
    ▷ fmap fst

-- | Returns the best utility for the respective player, expressed in terms of the utility of player Gold.
innerMiniMax :: Integer → State → Utility
innerMiniMax depth state@State {player} =
  case (utility breakthru) state of
    Just u → u
    Nothing
      | depth ≤ 0 → heuristic state
      | otherwise →
        childStates state
          ▷ map (innerMiniMax (depth - 1))
          ▷ relativeMax player (utilityOfPlayer player (Utility (-1 / 0)))
```

Agents



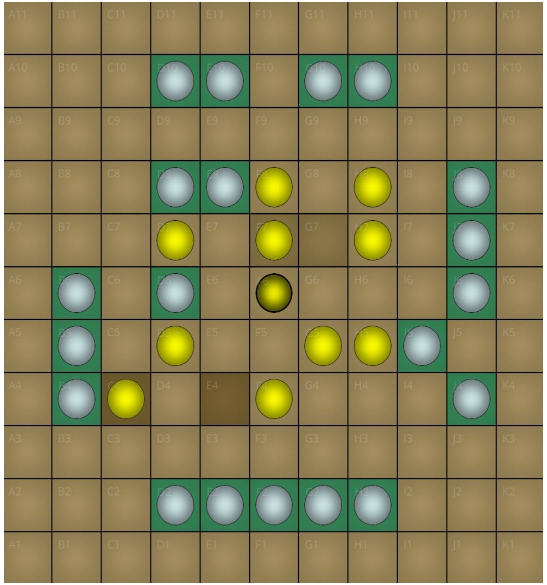
```
type Ai = State → Maybe Action
```

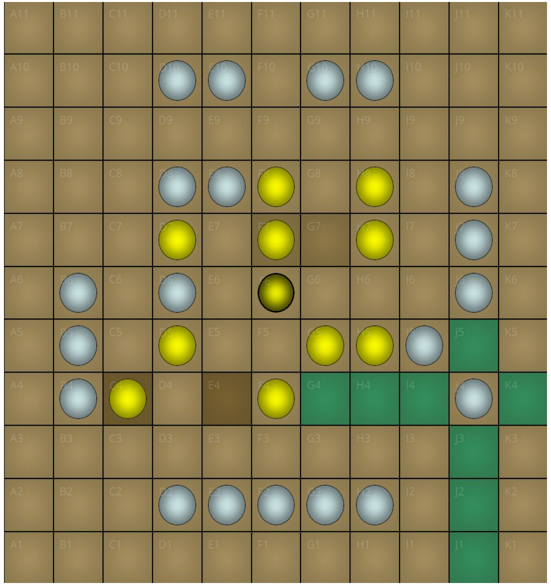
- Alphabeta (AlphaBeta.hs)
  - (John Hughes 1990: Functional implementation of AlphaBeta with laziness)
  - 2 parts
    - alphaBetaBranch: goes down one action
    - innerAlphaBeta: compares siblings and **prunes!**
  - Transposition Table
    - Generic hashing, no Zobrist hashing
  - Iterative deepening
  - Move ordering
  - ~O(b^d/2) instead of O(b^d)

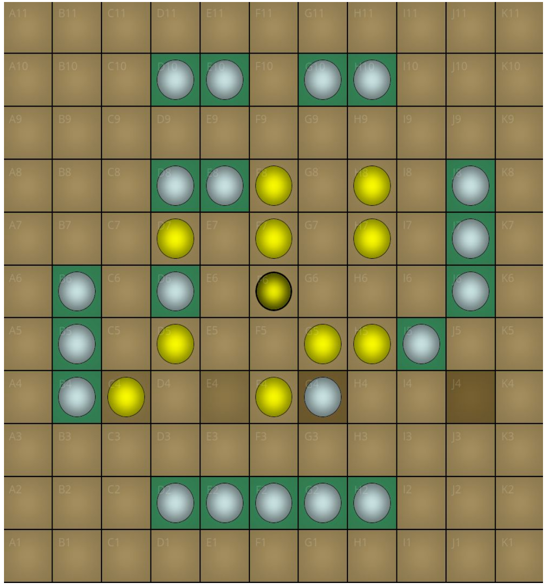


```
innerAlphaBeta depth bounds transpositions state@State {player} action rest =      You, 2 hours ago • Major refacto
  let (u, transpositions1) = alphaBetaBranch depth bounds transpositions state action
      newBounds p
        | p == player = bounds player
        | otherwise = Prelude.min (bounds (other player)) (utilityOfPlayer (other player) u)
      transpositions2 =
        transpositions1
          ▷ HashMap.insertWith (\new old → argMax fst old [new]) state (depth, let Utility float = u in float)
   in case rest of
        h : rest
          | utilityOfPlayer player u ≥ bounds player → (action, u, transpositions2) -- prune
          | otherwise →
            let (nextAction, nextU, transpositions3) =
                   innerAlphaBeta depth newBounds transpositions2 state h rest
                 (betterAction, betterU) = argMax (utilityOfPlayer player . snd) (action, u) [(nextAction, nextU)]
              in (betterAction, betterU, transpositions3) -- consider next action(s)
        [] → (action, u, transpositions2)

-- | Determines the value of a single action given a state, by going down the branch. Also updates the transpositi
alphaBetaBranch :: Integer → Bounds → TranspositionTable → State → Action → (Utility, TranspositionTable)
alphaBetaBranch depth bounds transpositions state@State {player} action =
  case (result breakthru) state action of
    Just result →
      case (utility breakthru) result of
        Just u → (u, transpositions)
        Nothing →
          let (lookupDepth, lookupU) = HashMap.lookup result transpositions ▷ fromMaybe (-1, 0)
           in if
                  | depth ≤ 0 → (heuristic result, transpositions)
                  | lookupDepth ≥ depth → (Utility lookupU, transpositions)
                  | otherwise →
                    case orderedMoves transpositions result of
                      a : rest →
                        let (_, u, t) =
                               innerAlphaBeta (depth - 1) bounds transpositions result a rest
                         in (u, t)
                      [] → (Utility 0, transpositions)
    Nothing → (utilityOfPlayer player (Utility (-1 / 0)), transpositions)
```

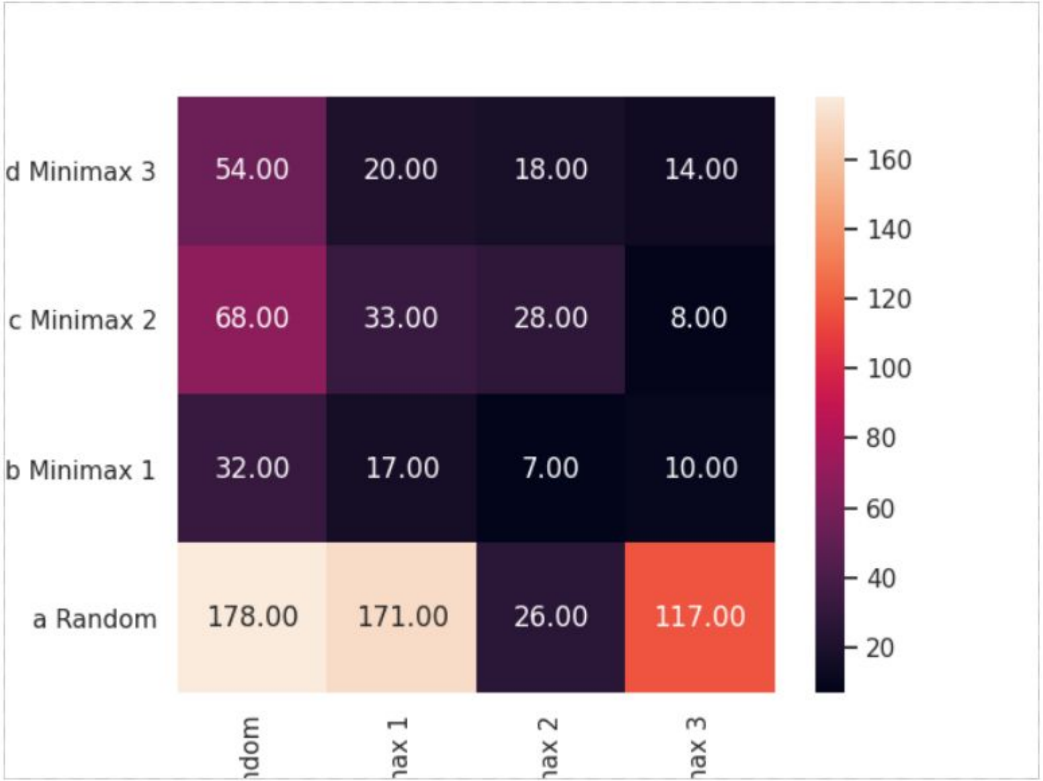
User Interface



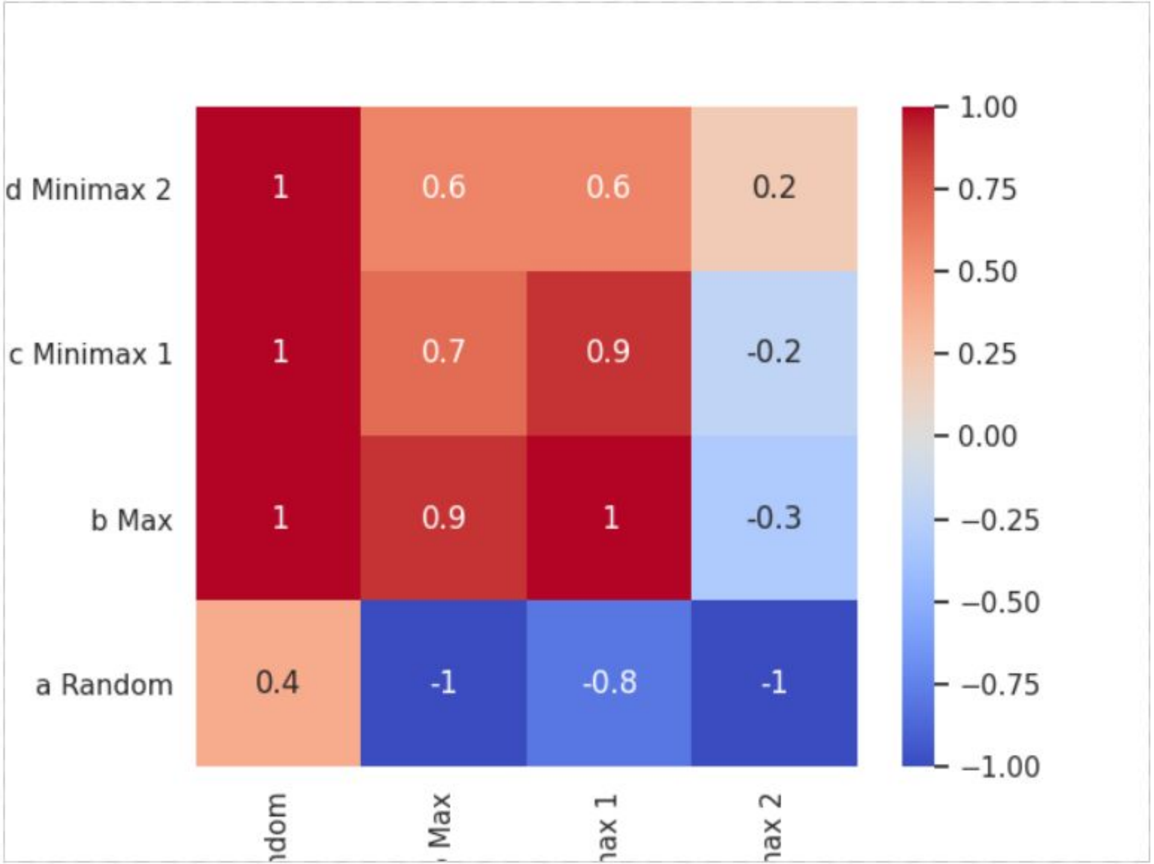




Evaluation (/evaluation/depth.png)



Evaluation (/evaluation/utility.png)



commit 5cd39830033f23ed15997e1630184add1670f419