Overview

- Debriefing Exercise 9
- Briefing Exercise 10
What is the search depth and the height of the tree?

Depth search = 3 and Height = 4 (or 3)
U9.A1b,c MinMax Algorithm

Best path: to the left
U9.A1d Alpha-Beta Algorithm

- The α-β algorithm
  - Reduces the game tree through pruning, but delivers the MinMax value of the root in the same way as the MinMax algorithm
  - The MinMax algorithm evaluates the whole search tree. In this case, nodes that don't influence the outcome (choice of the branch at the root) are also evaluated. The Alpha-Beta search ignores those nodes.

- $\alpha$
  - The largest known value of all MAX values of the MIN nodes
  - Is relevant for the evaluation of Min nodes (Evaluation of the successors can be aborted as soon as the computed return value is below $\alpha$)

- $\beta$
  - The largest known value of all MIN values of the MAX nodes
  - Is relevant for the evaluation of Max nodes (Evaluation of the successors can be aborted as soon as the computed return value is above $\beta$)
U9.A1d Alpha-Beta Algorithm

\[ A(-\infty, +\infty) \]

\[ B(-\infty, +\infty) \]

\[ C(4) \]

\[ D(-\infty, 4) \]

\[ F(7) \]

\[ E(-\infty, 4) \]

\[ G(7) \]

\[ H(4) \]

\[ I(4, +\infty) \]

\[ J(4, +\infty) \]

\[ K(3) \]

\[ L(3) \]

\[ M(4, +\infty) \]

\[ T(2) \]

\[ S(2) \]

\[ Q(4, +\infty) \]

\[ R(1) \]

\[ S(4, +\infty) \]

\[ T(4, 3) \]

\[ U(4, 2) \]

\[ V(4, 2) \]

\[ W(4, 2) \]

\[ X(4, 2) \]

\[ Y(4, 2) \]

\[ Z(4, 2) \]

\[ \alpha \]

\[ \beta \]

\[ 2 \leq \alpha \]

\[ 3 \leq \alpha \]

\[ 7 \geq \beta \]

\[ 7 \leq \alpha \]

\[ 2 \leq \alpha \]
Two helping methods:
- `max(...)`
- `min(...)`

Idea: `max()` and `min()` call each other in turns

Until we reach depth `d`
- `nextMove()`

```java
class BestMove {
    /**
     * The coordinates of the proposed next move
     */
    public Coordinates coord;

    /**
     * The value of the callers game board according to the min-max analysis
     */
    public int value;

    public BestMove(int value, Coordinates coord) {
        this.value = value;
        this.coord = coord;
    }
}

public Coordinates nextMove(GameBoard gb){
    BestMove bestMove = null;
    bestMove = max(d, gb, 0);
    return bestMove coord;
}
```
Find all the possible moves for MAX

Check for the timeout

Check for the depth

If MAX has no move, It’s MIN’s turn
max (2)

For all the moves, make the hypothetical move and then it’s MIN’s turn

```java
BestMove bestMove = new BestMove(Integer.MIN_VALUE, null);
for (Coordinates coord : availableMoves) {
    GameBoard hypothetical = gb.clone();
    hypothetical.checkMove(myColor, coord);
    hypothetical.makeMove(myColor, coord);
    BestMove result = min(maxDepth, timeout, hypothetical, depth + 1);
    if (result.value > bestMove.value) {
        bestMove.coord = coord;
        bestMove.value = result.value;
    }
}
return bestMove;
```

Choose the best one out of all the possible ones
Reversi – Upper and lower bounds for game situations

/**
 * Get the upper bound for the value of a game situation.
 * @param gb a game board
 * @return the maximum value possible for any situation on the given game board
 */
private int maxEval(GameBoard gb) {
    return gb.getSize() * gb.getSize();
}

/**
 * Get the lower bound for the value of a game situation.
 * @param gb a game board
 * @return the maximum value possible for any situation on the given game board
 */
private int minEval(GameBoard gb) {
    return -1 * maxEval(gb);
}
**
* Estimate the value of a game situation.
*
* @param gb
* the situation to consider
* @return the value of the current game board from the perspective of the player
*/

private int eval(GameBoard gb) {
    return gb.countStones(myColor) - gb.countStones(otherColor);
}
Reversi – Evaluate final game state

/**
 * Returns the value of a finished game
 * @param gb the situation
 * @return the value of the finished game from the perspective of the player.
 */
private int finalResult(GameBoard gb) {
    final int myStones = gb.countStones(myColor);
    final int otherStones = gb.countStones(otherColor);
    if (myStones > otherStones) return maxEval(gb);
    if (otherStones > myStones) return minEval(gb);
    return draw(gb);
}

private int draw(GameBoard gb) {
    return 0;
}
U9.A2b Reversi - Time Limit

- Timeout per move:
  - `nextMove()` has to return a valid move before the time-out of `timeLimit` milliseconds
  - `nextMove()`

```java
public Coordinates nextMove(GameBoard gb) {
    long timeout = System.currentTimeMillis() + timeLimit - 10;

    BestMove bestMove = null;
    try {
        bestMove = max(1, timeout, gb, 0);
    } catch (Timeout e) {
        throw new AssertionError("Hmm, not enough time for recursion depth 1");
    }

    try {
        for (int i = 2; ; i++) {
            bestMove = max(i, timeout, gb, 0);
        }
    } catch (Timeout e) {
    }
    return bestMove.coord;
}
```
Propositions for possible, static evaluations:

- **Agility**
  - How many moves are possible for me / my opponent?

- **Rows**
  - How many rows of connected stones are there?
  - How long are they? Their location is also interesting!
  - A fully occupied border is really good, while a long sequence in the opponent's can potentially allow for good moves

- **How many stones...**
  - Will be flipped by a given move and in how many directions? Are the stones lying inside the board or in the borders?

- **How many stones...**
  - Of a specific color are lying on the board? (That might be the evaluation function for the final game, when a thorough analysis of the search tree is possible. In the middle of the game, this might be inappropriate.)

- **Positions**
  - To be evaluated on the field (e.g. corner points)
Overview

- Debriefing Exercise 9
- Briefing Exercise 10
U10.A1 Merge Sort

- **Merge Sort**
  - Is a recursive and stable sort algorithm, which is based on the divide and conquer principle
  - It was developed in 1945 by John von Neumann

- **Divide and conquer principle**
  - Separate the enemies to vanquish them
  - Political and military strategy
  - Was already applied in the Roman empire

---

John von Neumann
1903 Budapest – 1957 Washington
**U10.A1a – Paper exercise**

- **Merge Sort**
  - Consider the data to be sorted as a list
  - Decompose the initial list in smaller lists and sort them
  - How to merge two sorted lists?
    - “Zipper” way

![Diagram showing merge sort process](Wikipedia)
U10.A1b Implementation

- ISort defines an interface
  - ISort.sort takes an ArrayList and return a new sorted ArrayList

- MergeSort.java (create a new class)
  - Implement the ISort interface

  ```java
  public ArrayList<T> sort(ArrayList<T> items);
  ```

- How about a helper method?
  - Two new parameters: start and end index

  ```java
  private ArrayList<T> sortRec(ArrayList<T> items, int begin, int end)
  ```

- You don’t have to always build a new list!
U10.A1c,d Statistics

- Generate N random numbers
  - N = 100, 200, 400, 800 etc.

- Sort the N random numbers using Merge Sort

- Repeat the experiment K times
  - For a statistically relevant result, you must actually do it K+2 times
  - Remove the lowest and highest result
  - Average the rest

- Plot your results & compare to the theoretical runtime
  - O(n log n)
  - Use a tool (e.g. GNUplot, Excel, Matlab, etc.)
  - Add the diagram to your homework
  - Don’t forget to interpret the results!!
U10.A2 Towers of Hanoi

- Puzzle
- Game pieces:
  - 3 rods
  - N different size disks

- Starting position: the disks are neatly ordered
- Goal of the game: move all the disks to another rod

- Constraints:
  - Only one disk can be moved at a time
  - A disk can only be moved if it is the upper most disk in the stack
  - No disk can be placed on top of a smaller disk
U10.A2 Hanoi Towers

- In the lecture
  - Recursive solution to the problem

- The only possibility is to move the bottom (largest) disc from tower 1 to tower 3:
  - (a) There is nothing else on tower 1
  - (b) Tower 3 is empty

- From (a) and (b) we derive that:
  - All other discs are on tower 2!
  - At first, the n-1 other discs must be moved from tower 1 to tower 2
U9.A2 Hanoi Towers – 3 discs

- Solution for the 3-disc case
  - Name the 3 towers from left to right 1, 2, 3 and the discs from the smallest to the largest A, B, C
  - Then use the number-letter pair to indicate where a disc has to be moved
  - C2 means for example that the largest disc has to be moved to the tower in the middle.

- Steps for the solution:
  - A3, B2, A2, C3, A1, B3, A3 (7 steps)
U9.A2a,b

Identify regularities:

- For each step in the execution of the recursive algorithm of the lecture, exactly one tower is not necessary.

- When shifting a tower of height 4 in 15 steps, give the sequence of tower number that is not used.
U9.A2b,c

- Describe all "developed" algorithms in pseudo-code
  - For the starting tower of height 4
  - Are changes necessary when starting with a tower of height 5?

<table>
<thead>
<tr>
<th>HumanPlayer</th>
<th>RandomPlayer</th>
<th>GreedyPlayer</th>
<th>MinMaxPlayer</th>
<th>α-β-Player</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nextMove()</strong>&lt;br&gt;Waits for entry from command line</td>
<td><strong>nextMove()</strong>&lt;br&gt;Chooses a random (but valid!) next move</td>
<td><strong>nextMove()</strong>&lt;br&gt;Chooses the next move by means of an easy and non-recursive evaluation function</td>
<td><strong>nextMove()</strong>&lt;br&gt;Choose the next move by means of a Minimax analysis through a new evaluation function</td>
<td><strong>nextMove()</strong>&lt;br&gt;Chooses the next move by means of the α-β analysis with your own evaluation function</td>
</tr>
</tbody>
</table>
U10.A3a Reversi

- Build an evaluation function, which follows the $\alpha$-$\beta$ process, which produces the same result as the pure MinMax method of the previous exercise sheet.

- $\alpha$-$\beta$ algorithm
  - Use the algorithm from the lecture! (don’t use a different version)
  - Throw a Timeout exception (just like in MinMax)
Have Fun!