Informatik II
Tutorial 10
Mihai Bâce
mihai.bace@inf.ethz.ch
10.05.2017
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
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.

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

- \( \beta \) \( \text{The largest known value of all MIN values of the MAX nodes} \)
  - 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

MAX

MIN

MAX

MIN
U9.A2 Reversi [Part 3]

- Two helping methods:
  - \( \text{max}(...) \)
  - \( \text{min}(...) \)

- Idea: \( \text{max}() \) and \( \text{min}() \) call each other in turns

- Until we reach depth \( d \)
  - \( \text{nextMove}() \)

```java
public Coordinates nextMove(GameBoard gb) {
    BestMove bestMove = null;
    bestMove = max(d, gb, 0);
    return bestMove.coord;
}
```

```java
class BestMove {
    public Coordinates coord;
    public int value;
    public boolean cut;
    // whether it was cut at the maximum recursion depth
}
```
private BestMove max(int maxDepth, GameBoard gb, int depth) {
    if (depth == maxDepth)  return new BestMove(eval(gb), null, true);
    ArrayList<Coordinates> availableMoves = getMovesFor(myColor, gb);
    if (availableMoves.isEmpty()){
        if( gb.isMoveAvailable(otherColor) ){
            BestMove result = min(maxDepth, gb, depth + 1);
            return new BestMove(result.value, null, false);
        } else
            return new BestMove(finalResult(gb), null, false);
    }
    BestMove bestMove = new BestMove(minEval(gb) - 1, null, false);
    for( Coordinates coord : availableMoves ) {
        GameBoard hypothetical = gb.clone();
        hypothetical.checkMove(myColor, coord);
        hypothetical.makeMove(myColor, coord);
        BestMove result = min(maxDepth, hypothetical, depth + 1);
        bestMove.cut = bestMove.cut || result.cut;
        if( result.value > bestMove.value ){
            bestMove.coord = coord;
            bestMove.value = result.value;
        }
    }
    return bestMove;
}
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);
}
/**
 * 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);
}
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("oh oh, not enough time for depth 1");
    }
    return null;
}
try {
    for (int i = 2; bestMove.cut; i++)
        bestMove = max(i, timeout, gb, 0);
} catch (Timeout e) {
}
return bestMove.coord;
}
U9.A2b Reversi - Timeout

class Timeout extends Throwable{
}

private BestMove max(int maxDepth, long timeout, GameBoard gb, int depth) throws Timeout
{
    if( System.currentTimeMillis() > timeout )
        throw new Timeout();

    if( depth == maxDepth ){
        return new BestMove( eval(gb), null, true );
    }

    return bestMove;
}
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
  - Is 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
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
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!
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
- Use a tool (e.g. GNUplot, Excel, Matlab, etc.)
- Add the diagram to your homework
- Don’t forget to interpret the results!!

O(n log n)
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
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)
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?
nextMove()  
Waits for entry from command line

nextMove()  
Chooses a random (but valid!) next move

nextMove()  
Chooses the next move by means of an easy and non-recursive evaluation function

nextMove()  
Choose the next move by means of a Minimax analysis through a new evaluation function

nextMove()  
Chooses the next move by means of the α-β analysis with your own evaluation function
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!