Informatik II (D-ITET)
Tutorial 10

TA: Anwar Hithnawi, E-mail: hithnawi@inf.ethz.ch
Distributed Systems Group, ETH Zürich
Outlook

- Exercise 9: Solution discussion
- Exercise 10: Overview (Mergsort, Hanoi Towers, Reversi)
Ex9.Q1 – Game Theory

- **Components of a game tree**
  - Root → Beginning (state before any move)
  - Node → Possible state of the game
  - Edge → Move
  - Leaf → End of the game (final state)
Ex9.Q1a,b,c – Game Tree Min-Max

Height: 4
Search Depth: 3
Best path: left

Max strategy

4

0

1

2

3

4

7

8

MAX

MIN

MAX

MIN
Ex9.Q1d – The α-β 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.

- α
  - The largest known value of all MAX treamehts 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 α)

- β
  - The largest known value of all MIN treamehts 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 β)
Ex9.Q1d – The Tree

\[ \alpha = \text{Best already explored option along path to the root for maximizer} \]

\[ \beta = \text{Best already explored option along path to the root for minimizer} \]
Ex9.Q2a – Min-Max

- Two helping methods:
  - `max(...)`
  - `min(...)`

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

- Until we reach depth `d`
  - `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
}
```
Ex9.Q2a – Min-Max – max(...)

```java
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;
}
```
Ex9.Q2b – timeLimit

- 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;
}
Ex9.Q2b – timeLimit – 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;  
}
Ex9.Q3c – Evaluation function

- What are all of the elements of the game that are interesting and should be evaluated to determine if one board is better than another? What are the relative weights that should be assigned to each such element?

- Propositions for possible, static evaluations:
  - **Agility**
    - How many moves are possible for me / my opponent?
  - **Rows**
    - How many rows of connected counters 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 counters...**
    - Will be flipped by a given move and in how many directions? Are the counters lying inside the board or in the borders?
  - **How many counters...**
    - 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)
Outlook

- Exercise 9: Solution discussion
- Exercise 10: Overview (Megsort, Hanoi Towers, Reversi)
Ex10.Q1 – Mergesort

- Mergesort
  - Is a recursive and stable sort algorithm, which is based on the divide and conquer principle
  - 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
Ex10.Q1a – Manual work

- Mergesort
  - Consider the data to be sorted as a list and decompose it into smaller lists, which will have to be sorted
  - The smaller sorted lists are merged together in a zipper way, until one common list is achieved

http://upload.wikimedia.org/wikipedia/commons/c/cc/Merge-sort-example-300px.gif
public void merge(int[] A, int[] B, int[] C) {
    int i, j, k, m, n;
    i = 0;
    j = 0;
    k = 0;
    m = A.length;
    n = B.length;
    while (i < m && j < n) {
        if (A[i] <= B[j]) {
            C[k] = A[i];
            i++;
        } else {
            C[k] = B[j];
            j++;
        }
        k++;
    }
    if (i < m) {
        for (int p = i; p < m; p++) {
            C[k] = A[p];
            k++;
        }
    } else {
        for (int p = j; p < n; p++) {
            C[k] = B[p];
            k++;
        }
    }
}
Ex10.Q1b – Implementation

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

- **MergeSort.java** (build)
  - Implement the **ISort** interface
  - Tip: recursive helping method
  - Tip: one does not always build a new list, but one can play with beginning-end indices
Ex10.Q1c,d – Measure.java

- 10 "measured points"
  - Make sure that the random arrays are built beyond the time measurements!

- Repeat the measurements
  - Ignore both min and max (extreme values)
  - Take the average of n measurements (overall n+2 measurement runs)

- Build a diagram
  - Your favorite tool (e.g.: GNUplot, Matplotlib, Excel, Matlab, …)
  - Deliver a graphic
  - Interpretation must add up!
Ex10.Q2 – Hanoi Towers

- In the lecture
  - Recursive solution to the problem

- The only possibility is to move the bottommost (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) derives:
  - All other discs are on tower 2!
  - At first, the n-1 other discs must be moved from tower 1 to tower 2
Ex10.Q2 – Hanoi Towers

- 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)
Ex10.Q2.a/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.
Ex10.Q2b,c – Pseudo-code

- Describe all "developed" algorithms in pseudo-code
  - For the starting tower of height 4
  - Are adaptations necessary when starting with a tower of height 5?
Hints for Ex10.Q3 – Reversi (Part 4)

HumanPlayer
nextMove()
Waits for entry from command line

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

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

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

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

Download
Excersie 7
Excersie 8
Excersie 9
Excersie 10
Hints for Ex10.Q3a – Reversi (Part 4)

- 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.
- Beware that in this exercise the algorithm of the lecture is requested, not one of its adaptation.
- Implement a move abort through a throwable timeout.
Reversi Tournament

- Tournament: Wednesday 31.05.2017, 12:30, CABinett (Stuz2).
  - Catering + Prices to who reaches the quarterfinals(!)

- Submission:
  - **Deadline: Wednesday, May 23, 2017, 23:59 (Zürich Time)**
  - Submit your player to the Reversi-Platform.
  - You can work alone or in groups of two

- A few pieces of advice concerning the tournament
  - Start with writing the Idea for the evaluation function in pseudo-code
  - Keep developing the pseudo-code
  - The pseudo-code yields hints about how the information about the next move should be computed
  - Keep implementing the different versions of the pseudo-code for the tournament player
Have Fun!