Informatik II (D-ITET)
Tutorial 9

TA: Hồng-Ân Cao, E-mail: hong-an.cao@inf.ethz.ch
Distributed Systems Group, ETH Zürich
Outlook

- **Exercise 8: Solution discussion**
- **Exercise 9: Overview (Game Tree, Reversi)**
Advanced Java Generics

Reading recommendation:

http://docs.oracle.com/javase/tutorial/java/generics/index.html
Solution Ex8.Q1a,b – Binary Search

\[ \text{mi} = (\text{re} - \text{li})/2 + \text{li}; \]
Solution Ex8.Q1c – Binary Search (with factor 3)

mi = (re - li) / 3 + li;

Fact: when looking for smaller figures, which are present in the array, this method is faster. But in general, it performs less well (deeper tree)
Solution Ex8.Q1d – BinarySearch

- Copying subarray is not good practice → You should rather build a method that takes two integers as arguments, such as `begin` and `end`:

```java
public Value find(List<Unit<Key, Value>> haystack, Key needle) {
    return findRec(haystack, needle, 0, haystack.size());
}
private Value findRec(List<Unit<Key, Value>> haystack, Key needle, int begin, int end) {
    numberofCalls++;
    if (begin == end) { return null; }
    int middle = begin + (end - begin) / factor;
    Unit<Key, Value> middleThing = haystack.get(middle);
    int match = needle.compareTo(middleThing.key);
    if (match == 0) { return middleThing.value; }
    else if (match < 0) { return findRec(haystack, needle, begin, middle); }
    else { return findRec(haystack, needle, middle + 1, end); }
}
private int factor;
private int numberofCalls;
BinarySearch() {
    factor = 2;
    numberofCalls = 0;
}
public void setFactor(int factor){ this.factor = factor; }
public int getNumberOfCalls() { return numberofCalls; }
```
static final int[] keys = { 3, 7, 17, 25, 33, 47, 56, 62, 65, 66, 68, 70, 78, 89, 92 }; 
static ArrayList<Unit<Integer, Integer>> numbers;  
private static void createKeys() {
    numbers = new ArrayList<Unit<Integer, Integer>>();
    for (int key : keys) {
        numbers.add(new Unit<Integer, Integer>(key, key));
    }
}
public static void main(String[] args) {
    createKeys();
    measure1(2);
    measure1(3);
    measure2(2);
    measure2(3);
    measure3(2);
    measure3(3);
}

private static void measure1(int factor) {
    BinarySearch<Integer, Integer> search = new BinarySearch<Integer, Integer>();
    search.setFactor(factor);
    for (int key : keys) {
        search.find(numbers, key);
    }
    double result = ((double) search.getNumberofCalls()) / keys.length;
    System.out.println(String.format("Measurements Question 1, Factor: %d, middle depth: %.2f", factor, result));
}

private static void measure2(int factor) {
    BinarySearch<Integer, Integer> search = new BinarySearch<Integer, Integer>();
    search.setFactor(factor);
    for (int i = 0; i < 100; i++) {
        search.find(numbers, i);
    }
    double result = ((double) search.getNumberofCalls()) / 100;
    System.out.println(String.format("Measurements Question 2, Factor: %d, middle depth: %.2f", factor, result));
}
Solution Ex8.Q2a – Simple Thief Strategy

- Does the simple thief strategy always deliver the optimal solution?
  - Yes, because it is going through all configurations

- Is there always one single optimal solution?
  - No, there can be multiple optimal solutions
  - Proof by counter-example:
    - Item \(<\text{Weight}, \text{Value}>\)
      \[
      [ \langle 1,1 \rangle, \langle 2,1 \rangle, \langle 3,2 \rangle ]
      \]
    - \(W_{\text{max}} = 3\)
    - Solution 1 = [true, true, false]
    - Solution 2 = [false, false, true]
public Selection findBest(ArrayList<Integer> values, ArrayList<Integer> weights, int maxWeight) {
    if (values.size() != weights.size()) {
        throw new IllegalArgumentException("sizes of values and weights vectors are not equal");
    }
    final int max = (int) Math.pow(2, values.size());

    Selection bestSelection = null;
    int maxValue = -1;
    for (int i = 0; i < max; i++) {
        Selection selection = new Selection(values.size(), i);
        if (selection.sum(weights) <= maxWeight) {
            int value = selection.sum(values);
            if (value >= maxValue) { // Multiple possible optimal solutions
                bestSelection = selection;
                bestSelection = selection;
                maxValue = value;
            }
        }
    }
    return bestSelection;
}
public Selection findBest(ArrayList<Integer> values, ArrayList<Integer> weights, int maxWeight) {
    if (values.size() != weights.size()) {
        throw new IllegalArgumentException("sizes of values and weights vectors are not equal");
    }
    Selection result = find(new Selection(0), 0, values, weights, maxWeight);
    return result;
}
private Selection find(Selection selection, int weight, ArrayList<Integer> values, ArrayList<Integer> weights, int maxWeight) {
    final int depth = selection.size();
    if (depth == values.size()) {
        return selection;
    }
    Selection without = new Selection(depth + 1, selection.bits());
    without.set(depth, false);
    Selection resultWithout = find(without, weight, values, weights, maxWeight);
    if (weight + weights.get(depth) <= maxWeight) {
        Selection with = new Selection(depth + 1, selection.bits());
        with.set(depth, true);

        Selection resultWith = find(with, weight + weights.get(depth), values, weights, maxWeight);
        if (resultWith.sum(values) > resultWithout.sum(values)) {
            return resultWith;
        }
    }
    return resultWithout;
}
Solution Ex8.Q2d – Runtime

- Measurement (2GHz Intel)
  - Brute Force: ~4.0s
  - Backtracking: ~0.08s
- Backtracking requires about 2% of the time spent on the Brute Force approach
  - \( W \) accounts for about 5x the average weight
  - On average only 5 items out of 20 are chosen (20%)
  - A lot of premature aborts
Solution Ex8.Q3a – checkMove: How To

Invalid!!

Valid?
Solution Ex8.Q3a – checkMove

- boolean checkMove(GameBoard gb, int player, Coordinates coord)
  - Field must be free!
  - Test all directions
    - As long as we haven’t stumbled upon at least one direction that is "valid"
- checkDirection(GameBoard gb, int player, Coordinates coord, int x, int y)
  - Checks if with respect to a given direction the given move is valid
  - Keep going until zero or more stones of the opponent and then a stone of a different color follows.
public boolean checkMove(GameBoard gb, int player, Coordinates coord) {
    try {
        if (gb.getOccupation(coord) != GameBoard.EMPTY) { return false; }
    } catch (OutOfBoundsException e) { return false; }
    for (int x = -1; x <= 1; x++) {
        for (int y = -1; y <= 1; y++) {
            if (x == 0 && y == 0) continue;
            if (checkDirection(gb, player, coord, x, y)) return true;
        }
    }
    return false;
}

private boolean checkDirection(GameBoard gb, int player, Coordinates coord, int x, int y) {
    Coordinates c = new Coordinates(coord.getRow() + x, coord.getCol() + y);
    try {
        if (gb.getOccupation(c) != Utils.other(player)) {
            return false;
        }
    } catch (OutOfBoundsException e) { return false; }
    return follow(gb, Utils.other(player), c, x, y);
}

private boolean follow(GameBoard gb, int player, Coordinates coord, int x, int y) {
    Coordinates c = new Coordinates(coord.getRow() + x, coord.getCol() + y);
    int occupation;
    try { occupation = gb.getOccupation(c); } catch (OutOfBoundsException e) { return false; }
    if (occupation == player) {
        return follow(gb, player, c, x, y);
    } else if (occupation == Utils.other(player)) { return true; }
    return false;
}
Solution Ex8.Q3a – Reversi Hints & Tricks

- **checkMove** is declared in `reversi.GameBoard`
  - `boolean reversi.GameBoard.checkMove(int, Coordinates)`

- More useful methods
  - `boolean reversi.GameBoard.isMoveAvailable(int)`
  - `boolean reversi.GameBoard.validCoordinates(Coordinates)`
  - `int reversi.GameBoard.countStones(int)`
  - `int reversi.Utils.other(int)`
  - ...

- Fact: JavaDoc is cool…
  - If the others made the effort of writing it…
Solution Ex8.Q3b - GreedyPlayer

- **Player-AI**
  - For all possible moves
    - Simulate move on copy of current board
    - Evaluate the resulting situation
    - Save move and evaluation in a list
  - Sort the list / Look for the Maximum/Maxima
  - Select the (or randomly) a move that maximizes your counters
  - Evaluation function (until now)
    - Proportion of own counters vs. opponent's

- **Data structures**
  - `MoveInfo`: save the evaluated move information
    - Coordinates and evaluation
  - `List<MoveInfo>` (e.g. an `ArrayList`)
    - More efficient than a `Vector`
    - More practical that an `Array`
Solution Ex8.Q3b - GreedyPlayer

```java
public Coordinates nextMove(GameBoard gb) {
    Coordinates bestMove = null;
    int bestValue = Integer.MIN_VALUE; // minEval(gb);

    for (int x = 1; x<=gb.getSize(); x++) {
        for (int y = 1; y <= gb.getSize(); y++) {
            Coordinates c = new Coordinates(x, y);
            if (gb.checkMove(myColor, c)) {
                GameBoard hypotheticalBoard = gb.clone();
                hypotheticalBoard.checkMove(myColor, c);
                hypotheticalBoard.makeMove(myColor, c);
                int value = eval(hypotheticalBoard);
                if (value > bestValue) {
                    bestValue = value;
                    bestMove = c;
                }
            }
        }
    }
    return bestMove;
}

/**
 * Evaluate a situation on a board
 * @param gb the situation in question
 * @return the value of the situation. Larger means better.
 */
private int eval(GameBoard gb){
    return gb.countStones(myColor) - gb.countStones(Utils.other(myColor));
}

/**
 * Returns less than the worst possible evaluation
 */
private int minEval(GameBoard gb){
    return -1 * (gb.getSize() * gb.getSize() + 1);
}
```
Outlook

- Exercise 8: Solution discussion
- Exercise 9: Overview (Game Tree, Reversi)
Hints Hints Ex9.Q1

Game theory / Evaluation of game tree

a) A little bit of theory

b) Minimax algorithm

c) Optimal strategy for MAX player

d) Alpha/Beta algorithm
Hints 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)
Hints Ex9.Q1b – Minimax algorithm

- Algorithm to determine the optimal game strategy for zero-sum

- Secures the highest possible gain when the opponent plays to win

- For zero-sum games better algorithms exist
Hints Ex9.Q1b – Minimax algorithm
Hints Ex9.Q1c – Strategy for Max

- Strategy
  - A strategy (for Max) is a graph, which derives from a game tree, where all edges are stricken out and only edges coming out of Max nodes are kept.

By proxy a set of nodes / edges, not just a path
Hints Ex9.Q1d – The α-β algorithm

- The α-β algorithm
  - Prunes the game tree step by step, but delivers the same minimax value of the root as the Minimax algorithm
  - The Minimax algorithm evaluates the whole search tree. The nodes that don't influence the outcome of the algorithm, i.e. the choice of the path at the root, are also considered. The α-β search ignores these nodes.
- α \rightarrow \text{lower bound for Max (what Max can achieve through the moves that have been investigated until now)}
  - 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 \rightarrow \text{upper bound for Min}
  - 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)
Hints Ex9.Q1d – The α-β algorithm

- Remember the operation tree from lecture 9
  - MAX nodes are OR nodes
  - MIN nodes are AND nodes
  - Leaves can only take 0 and 1 as values
Hints Ex9.Q1d – The tree

- Compute the value of the root in the following game tree using the α-β method

\[ \alpha – \beta \text{ cut: MIN has already nodes with } \beta < 7 \rightarrow \text{Abort before the end!} \]
Hints Ex9.Q1d – The α-β algorithm

Online example:
(user: i2 password: i22012)

Online Tutorial:
http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

Online Java Applet:
http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html
Hints Ex9.Q2 – Reversi (Part 3)

**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()`
  - Chooses the next move by means of a Minimax analysis through a new evaluation function

**Download**
**Exercise 7**
**Exercise 8**
**Exercise 9**
Hints Ex9.Q2 – Reversi (Part 3)

- Evaluation of the game trees
  - Implement a method that evaluates the game tree through Minimax (or NegaMax) until the depth d is reached (alternating Max and Min)
  - Depth of research can be configured
  - Recursive approach
    - Build the game tree recursively
    - Evaluate the state at depth d
    - Minimax on the obtained evaluation yields the strategy
    - Incorporate all special cases (e.g. pass)!

- No timeout yet
Hints Ex9.Q2b – Time limit

- Time limitation per move:
  - Your `nextMove()` method should return a valid move before the time-out in milliseconds
  - Plan a time buffer (in the order of 10 ms) so that cancellation and delivery of result don't happen straight away!
  - Possible approach: throw an out-of-time exception
Hints Ex9.Q2c – Evaluation function (I)

- You can find a "source of inspiration" in the following article:
  - "The Development of a World Class Othello Program"
  - Kai-Fu Lee and Sanjoy Mahajan, 1990

- To download from the Reversi website
  - username: i2bib
  - password: reversi

- Artificial Intelligence: A Modern Approach
  - Stuart Russell and Peter Norvig (3rd Edition, 2009)
  - http://aima.cs.berkeley.edu/
Hints Ex9.Q2c – Evaluation function (II)

- Possible evaluation functions
  - How many stones are flipped?
  - Where are the flipped stones located (center/border)?
  - ....

- 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!

Source: https://xkcd.com/1002/

No longer the case since 2015

No longer the case in 2016