Informatik II (D-ITET)

Tutorial 9

TA: Anwar Hithnawi, E-mail: hithnawi@inf.ethz.ch

Distributed Systems Group, ETH Zürich
Outlook

- Exercise 8: Solution discussion
- Exercise 9: Overview (Evaluation of game tree, Reversi)
Solution U8.A1a,b – Binary Search

\[ mi = (re - li) / 2 + li; \]
Solution U8.A1c – 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 U8.A1d – 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(ArrayList<Unit<Key, Value>> haystack, Key needle) {
    return findRecursive(haystack, needle, 0, haystack.size());
}

private Value findRecursive( ArrayList<Unit<Key, Value>> haystack, 
                              Key needle, int begin, int end) {

    ...
}
```
Solution U8.A1d – BinarySearch

```java
private Value findRecursive( ArrayList<Unit<Key, Value>> haystack, Key needle, int begin, int end)
{
    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);
    }
}
```
L8.A4a – 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 <Weight, 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) {
                bestSelection = selection;
                maxValue = value;
            }
        }
    }

    return bestSelection;
}
public class Backtracking implements IRucksack {

    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;
    }
}

the actual Backtracking
L8.A4d – 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 U8.A3a – checkMove: How To

Invalid!!

Valid?
Solution U8.A3a – checkMove

- boolean checkMove(GameBoard ..., Coordinates c)
  - Field must be free!
  - Test all directions
    - As long as we haven’t stumbled upon at least one direction that is "valid"
      ```java
      for( int i = -1; i <= 1; ++i )
        for( int j = -1; j <= 1; ++j )
          if( i != 0 || j != 0 )
            if( checkDirection(gb, player, c, i, j )
              return true; //possible move
      return false; //no possible move
      ```
  
- Check with respect to a given direction the given move is valid
  - checkDirection(GameBoard gb, int player, Coordinates coord, int x, int y)

- Follow a continues line of same-colored stones
  - follow(GameBoard gb, int player, Coordinates coord, int x, int y)
Solution U8.A3a – 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 U8.A3b - 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 evaluation
  - Select the (or randomly) a move that maximizes your counters
  - Evaluation function (until now)
    - Proportion of own counters vs. opponent's
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;
}

private int eval(GameBoard gb) {
    return gb.countStones(myColor) - gb.countStones(Utils.other(myColor));
}
Outlook

- Exercise 8: Solution discussion
- Exercise 9: Overview (Evaluation of game tree, Reversi)
Hints U9.A1

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
U9.A1 – 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)
U9.A1b – Minimax algorithm

- Algorithm to determine the optimal game strategy for zero-sum
- Secures the highest possible gain when the opponent plays to win
U9.A1b – Minimax algorithm
U9.A1c – 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
U9.A1d – 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 Alpha-Beta search ignores these nodes.
- \( \alpha \rightarrow \) 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 \( \alpha \))
- \( \beta \rightarrow \) 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 \( \beta \))
U9.A1d – The $\alpha$-$\beta$ 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
U9.A1d – The $\alpha$-$\beta$ algorithm
U9.A1d – The tree

- Compute the value of the root in the following game tree using the $\alpha$-$\beta$ method

$$\beta$$-cut: MIN has already nodes with $\beta < 7 \rightarrow$ Abort before the end!
U9.A1d – The $\alpha$-$\beta$ algorithm

Online example:
http://www.vs.inf.ethz.ch/edu/I2/slides/Info2-ITET-AlphaBeta.pdf
(user: i2 password: i22012)

Online JAVA Applet:
http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html
### Hints U9.A2 – Reversi (Part 3)

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HumanPlayer</td>
<td>nextMove()</td>
<td>Waits for entry from command line</td>
</tr>
<tr>
<td>RandomPlayer</td>
<td>nextMove()</td>
<td>Chooses a random (but valid!) next move</td>
</tr>
<tr>
<td>GreedyPlayer</td>
<td>nextMove()</td>
<td>Chooses the next move by means of an easy and non-recursive evaluation function</td>
</tr>
<tr>
<td>MinMaxPlayer</td>
<td>nextMove()</td>
<td>Choose the next move by means of a Minimax analysis through a new evaluation function</td>
</tr>
</tbody>
</table>

**Download**

**Übung 7**

**Übung 8**

**Übung 9**
Hints U9.A2 – 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 time-out yet
U9.A2b – timeLimit

- 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
U9.A2c – 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
U9.A2c – Evaluation function (II)

- Possible evaluation functions
  - How many counters are flipped?
  - Where are the flipped counters 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!