Leversi

Linus Närvä (c10lna)

December 2, 2014

Course:	Artificial Intelligence - Methods and Applications (Fall 14)
Teachers:	Ola Ringdahl, Juan Carlos Nieves Sanchez
Institution:	Computer Science

1 Introduction

This report describes the implementation of Leversi. A Reversi game implementation with a graphical user interface. The focus of this report lies on the implementation of the AI player.

1.1 Work in progress

At the time of writing, the GUI part of the program is not yet finished.

2 Implementation details

Everything was implemented in c++11 using the Gtkmm package to implement the GUI parts. It is intended to run on Linux, so in the getting started section the operating system is assumed to be Linux.

3 Getting started

The first thing to do is of course to download the file. It can be found in the Umeå University CS department computer system at:

```
~c10lna/edu/5DV122/lab1/leversi
```

3.1 Requirements

g++4.8 or later and gtkmm 3.0 or later.

3.2 Compilation

Just enter the directory 'src' and run *make build*[1]. Then run *make clean* to remove object files (optional). This should create two executables. *leversi* and *leversi-bot*. The former runs the program with the GUI. The second is a command line interface for reading in a state representation and return a move.

[1]Just running *make* will also attempt to compile the unit tests which adds the unit test framework 'Catch 1.0' to the dependencies.

class	**description**
ReversiState	State representation with getters and setters for all values. It is not responsible for keeping itself legal, and does not implement any portion of the transition model.
ReversiAction	Defines the transition model i.e. moving from one state to another, by applying an action and without breaking any rules.
Outcome	Trivial but important class specifying the outcome of an action after it has been applied (sufficient information to undo the action to restore the state). It is implemented as a struct containing the action and the flips it caused.
Game	The full game is all events that happened during the play. It is implemented as a *current state* and a list of outcomes of actions. Actions can be *committed* to the game, modifying the state and adding the outcome to the history. Actions can also be undone, restoring an earlier state of the game.

Table 1: Describes the most significant class components of the Reversi game model.

3.3 Execution

3.3.1 leversi

Make sure that the the directory you run it from includes the *res* directory (containing the graphical objects used). Then just run the executable.

3.3.2 leversi-bot

Run the executable with no arguments and it will print usage information to standard error.

4 Game representation

The class representation of the game model are described in table 1.

The most significant part of of this model implementation is the class *Game*. It stores only one copy of ReversiState in memory, but allows actions executed on this state to be undone. Intuitively this should yield better performance during maximin search, because the state doesn't need to be copied in recursive calls. Only one state representation is ever stored in memory.

5 Decision procedure

The desicion procedure for choosing the move is divided over three logical components. The *move searcher* (MaximinSearcher) searches for a move that is in some sense optimal. The *evaluator* (WashingtonEvaluator.hpp) evaluates a states and moves, thus defining what is considered optimal by the move searcher. Finally the *time manager* ensures that maximum performance is obtained, given a tolerated *latency*.

5.1 Searching for the best move

The *move searcher* uses the method of *maximin adversial search*, with *alpha-beta pruning* and *move ordering*. The original Maximin adversial search algorithm is used (as described in figure 5.7 on page 170 of [1]), with one minor modification. $Max - Value$ and $Min - Value$ are

member function	**description**
$maximinAction()$	A user friendly interface omitting the recursion stuff. Just calls $\_maximinAction()$ with initial parameters.
$\_maximinAction()$	Recursive sub function. Performs termination detection and move ordering. Calls $maxValue()$ or $minValue()$ depending on if the max or minplayer are currently playing in the state.
$maxValue()$	Expands a max node. Calls $\_maximinAction()$.
$minValue()$	Expands a min node. Calls $\_maximinAction()$.

Table 2: Table summarising the *MaximinSearcher* member functions responsible for the search, and how they relate to each other. It is somewhat simplified.

not responsible for termination detection. Instead an intermediate recursive call handles termination detection and also move ordering. The relevant member functions of MaximinSearcher are described in table 2.

5.2 State evaluation

Two evaluators are have been implemented. SimpleEvaluator is the simplest evaluator possible. Its utility function just computes

$$utility := numMaxCoins - numMinCoins.$$

The more sophisticated utility implemented as the WashingtonEvaluator class (it is called WashingtonEvaluator because it was invented at Washington university). Basically it does four things.

1. Tries to maximise the number of coins.
2. Tries to minimize the mobility of the opponent.
3. Tries to grab the corners.
4. Tries to grab 'stable' positions, which are less likely to be taken later in the game.

See the article[2] for details.

The evaluator is also responsible for computing a move utility score used for move ordering. This score does not take into account if the min or max player is playing (The zero-sum rule says that "what the min player gains what the max player looses"). The implementation is trivial and just computes the gain of advantage in number of coins by the player compared to the adversary. The formula is

$$\begin{array}{ll} moveUtility := 1 + 2 * numFlips, & \text{or just} \\ moveUtility := 0 & \text{if the action was } \textit{pass}. \end{array}$$

The WashingtonEvaluator inherits this utility from SimpleEvaluator.

5.3 Handling time limit

Because Maximin search is a form Depth First traversal, it is feasible to use the method of *iterative deepening*, to successively produce better guesses until the time runs out. This will cause only a slight increase in time complexity, however it will still be exponential. Specifically we have time complexity $O(b^d)$, where b is the effective branch factor and d the deepest iteration depth.

The implementation uses a prediction function (described below) to predict how long the next iteration will take. If it is predicted to complete before a given point in time, then another iteration is performed. Otherwise it terminates the deepening.

The con of this method is of course that the prediction may be wrong. The pro is that it will not always use up all the time given, without loosing performance.

5.3.1 Prediction function

The prediction function assumes that each node will use the same amount of time (see the result section for a verification of this). The depth d, the number of nodes n and the time used t by the last iteration is used to predict how long the next iteration (with depth increased by one) will take t'. An approximative branching factor b and a predicted number of nodes are computed as intermediate steps.

1. $b = n^{\frac{1}{d}}$
2. $n' = n + b^{d+1}$
3. $t' = \frac{n'}{n}t$

5.3.2 Algorithm

Below is the algorithm description of the member function maximinAction() of class Timeboxed-MaximinSearcher.

```
FUNCTION timeBoxedMoveFinder(maxTime, minDepth) -> ReversiAction
BEGIN
        t0 := currentTime();
        tEnd := t0 + maxTime;
        d := minDepth;

        DO
                searcher.setMaxDepth(d);

                tStart := currentTime();
                action := searcherfindBestAction();
                tFinish := currentTime();

                n := searcher.getNumNodes();

                nPredict := n + b^(d+1);

                b := searcher.getBranchFactor();

                tPredict := tFinish + (tFinish - tStart)*(nPredict/n);

        WHILE (tPredict < tEnd);

        RETURN action;
END
```

starter	**winner**	**score**
reference	Leversi	20
Leversi	Leversi	59

Table 3: Scores for Leversi versus the reference implementation, taking turns to start. In both cases, the reference implementation was pwnt.

6 Results

Unless otherwise mentioned values below where obtained by letting Leversi go two rounds against the reference implementation provided by the institution[2]. In one Leversi starts, and in the other the the reference implementation starts.

Data was benchmarked on a DELL XPS L322 computer using a 3'd gen i7 processor. The system was build using O4 optimization and the NODEBUG macro enabled. The full list of CFLAGS in the Makefile was:

```
-Wall -std=c++11 -O4 -DNODEBUG
```

The think time was set to 5 seconds.

6.1 Score

Results are show in table 3. Clearly Leversi was superior. Score was calculated using:

$$score := numWinnerCoins - numLooserCoins.$$

6.2 Performance, depth, branching factor

Some performance results for all turns are summarized in table 4. Notably it shows that we can count on the ai to be able to think about 8 moves in the future. That value however should not be mistaken for actual performance. Figure 1, shows how the branching factor changes over turns. It peaks near the middle of the game, probably due to the fact that most moves are available then. It declines rapidly at the end because the number of moves decreases as the board gets filled and also because the recursion reaches game over. In the beginning it is surprisingly high compared to at the end, considering that only a few moves are available. This is probably because, it is hard to prioritize between moves that are equally good.

A more reliable analysis of time usage is deferred to the next section.

6.3 Time limit

The problem with the time limit analysis in the previous section is that the last seven (the effective depth) actions are different to the others. Because game over is reached, there is no point in continuing the recursion which then terminates much faster (sometimes below one millisecond as seen in the previous section). The data in table 5 shows statistical properties for the time when the last seven actions in the both games where discarded (the depth until game over is then 14 or 15).

There is one outlier taking more than 5 seconds. But otherwise every action is inside the time limit (the second longest taking 4346 milliseconds). Otherwise the the time used is about usually about two seconds. Considering that the branching factor is about 3.6, the value seems sound.

[2]Instutution of Computer Science: Umeå University

what	**mean**	**median**	**min**	**max**
branching factor	3.5967	3.8482	-*	5.2364
depth	8	8	7*	16
time (ms)	1847.6	1922	< 1	6212

Table 4: Some performance results for the games played.
* At the end of the game when there are only one (or two) moves left the minimum branching factor will always occur and it will be exceptionally low. I've excluded it from the table because it does not say anything about performance. Consequentially the min depth is the one corresponding to the max branch factor. Of course I have verified that it is the minimum if one skips the last few turns of the game when the search reaches game over nodes.

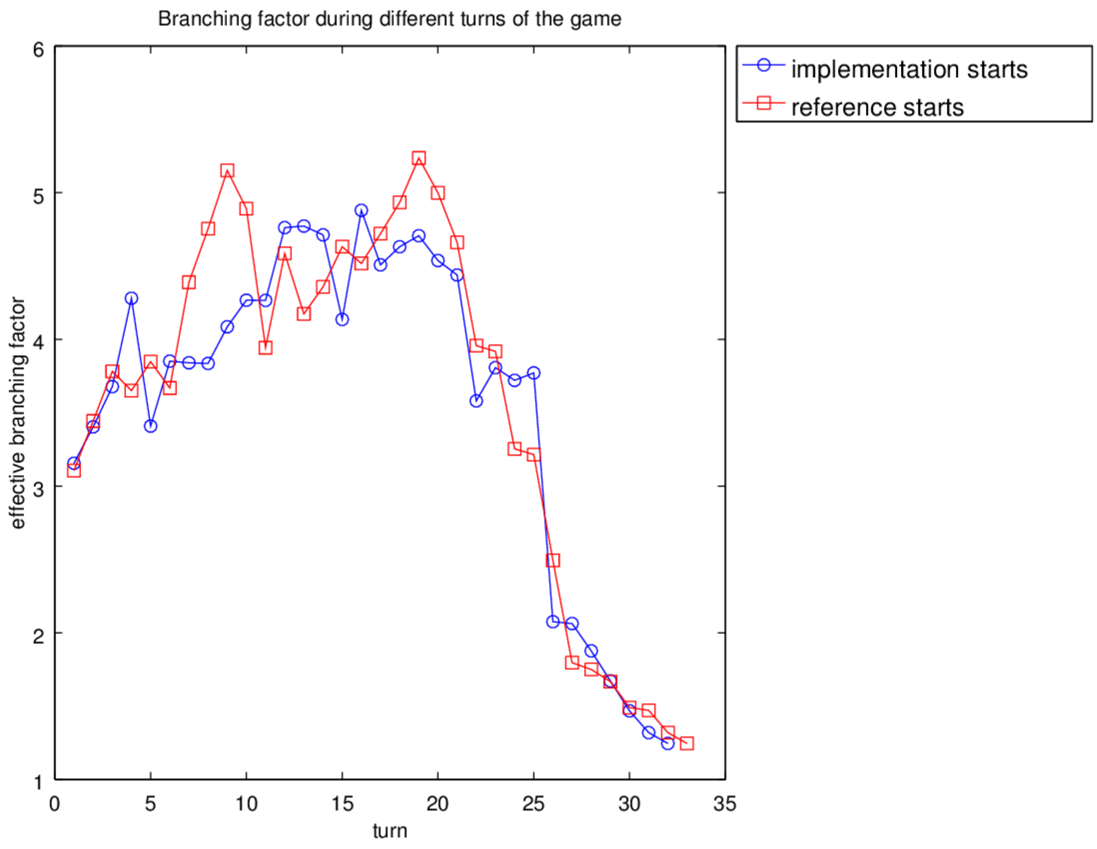

Figure 1: Branching factor for each of Leversis turns. It peaks in the middle of the game.

what	**mean**	**median**	**min**	**max**	**<1000**	**>5000**
trimmed time (ms)	2266	2083	952	6212	3.9%	2.0%

Table 5: Time data when the last 14 turns of the game are skipped.

If outliers taking more than 5 seconds are totally unacceptable (e.g. due to a special rule), then the approach used in Leversi is insufficient. It should be combined with a time-out, terminating the current recursive call if it has not completed in time and returning the action from the previous iteration. For playing against humans however it is ideal, because the exceptions to the time limit are very few, and it does not waste the players time, by continuing, even though it will not finish in time.

7 References

References

[1] Russel, Stuart; Norvig, Peter.
Artificial Intelligence a modern approach
Third edition
Pearson
ISBN-13: 978-0-13-207148-2 ISBN-10: 0-13-207148-7
©2010 by Pearson Education Inc., publishing as Prentice Hall, Upper Saddle River, New Jersey 07458.

[2] Sannidhanam, Vaishnavi; Annamalai, Muthukaruppan
An Analysis of Heuristics in Othello
2004 adjusted version
Department of Computer Science and Engineering,
Paul G. Allen Center,
University of Washington,
Seattle, WA-98195,
vaishu@cs.washington.edu,
muthu@cs.washington.edu.