Invers

Bachelor Thesis in Computer Science

by

Thomas Gruber

Matr.Nr.: 1030286, t.gruber@student.tugraz.at

Supervisor: Associate Prof. DI Dr. Oswin Aichholzer

Institute for Software Technology (7160)

Faculty of Computer Science and Biomedical Technlogy

Graz University of Technology

Abstract

This paper describes the basic rules of the abstract two player board game *Invers*, as well as the implementation for a program that can play this game. It uses the Minimax-Algorithm with Alpha-Beta-Pruning and a simple, yet effective heuristic function.

Course number(s): 716.404
Document version: 1.0
Date: November 19, 2015

1 The game and its rules

This thesis deals with the board game "Invers" by Kris Burm that was published by Spiele-Verlag PERI GmbH in 1991. It is not very widely known (e.g. [1] or [2]) and therefore there isn't much literature about this game. It is the only game by Kris Burm that was nominated for the Game of the Year-Award that does not have an own entry in the German Wikipedia [5]. It does not appear on the Boardspace internet main page [3] and is ranked in the higher thousands of places there. It is rather safe to say that this game is very little known, standing in the shadow of the much more popular games of the "Gipf"-Project by Kris Burm [4].

The game is out of production in 2015, and it's unclear if it will be produced ever again. The only option to get a copy of the game is to buy a used one.

There was no academic paper to be found that deals with this game, as well as no implementation.

1.1 Rules

The game consists of a game board and 38 stones, 19 of each color. There are two colours of stones: yellow and red ones. A stone has a top side and a reversed side. The top side is red (or yellow) all over (from now on referred to as the *normal* side), whereas on the inverted side there is only a small dot in the corresponding colour on neutral black background.

Before the start of the game, the stones are laid out on the board with their normal sides up, alternating a red and a yellow one into each direction. Thus, the board looks like a 6x6 chess board in the beginning. For each player, there is one stone left. This stone is the players *hand*. These two stones are already *inverted*.

Now each player puts a stone into the game board in a way, that the whole column or row moves one position further. On the other side of the board, one stone is pushed out of the board. The player turns the stone around (*inverts* it), if it was not inverted yet, and takes it as his new hand. This means that a stone is always put into the game field inverted. Both player do this in turn.

Winner is, who has inverted all stones of his colour first.

There is only one rule restricting the possible moves: No player is allowed to push an *opponent inverted* stone out of the board.

Although the official rulebook deals with the case where a player is not able to do a move at all (*"Ist kein Zug mehr möglich und ein Spieler hat noch nicht alle Steine umgedreht, so ist jener Spieler Sieger, der die meisten eigenen Steine umgedreht hat."*), this rule can not be applied, because there is no possible position on the board where a player can not make a valid move. To prevent a player from making a move, an inverted opponent stone has to be on the other side of the board. There are only 19 stones of each color. Stones in the edge positions count double, because they can *block* two input positions. Thus, at most 23 input positions can be blocked theoretically, however, there are 24 possible input positions.

2 Complexity of the game

The board consists of 36 stones, every one can be either red or yellow and either normal or inverted. This gives a (very rough) first upper estimation of the number of possible positions: 4^{36}.

However, not all of these positions are valid, because there are only 19 stones of one color, so there are either 17 yellow and 19 red stones or 16 yellow and 16 red stones or 19 yellow and 17 red stones on the board.

But even then not all positions are reachable, because due to the fact that inverted stones can enter the board only in the outer rows and columns, there is no possible position with inverted stones in the middle, but not in the outer areas.

A tighter estimation is somewhat harder to come up with.

The size of the set of possible games is much bigger. The minimal game length is 36 moves, and that happens if every player can invert a stone in every turn. Considering 24 possible moves (in the later game it becomes less), there are 24^{36} possible games. However, in a theoretical game, where both players try to minimize the possible moves of the opponent, there are still about $\frac{24!}{12!} * 12^{16}$ games left.

3 Algorithms used

Due to the complexity of the game, it is not possible to calculate the whole game tree. Therefore, the Minimax Algorithm with Alpha-Beta-Pruning, move-presorting and on-the-fly calculation depth changing was chosen to

calculate the game tree.

The Minimax Algorithm can be used for Invers natively, without any changes to the algorithm itself. This also holds for Alpha-Beta-Pruning.

Presorting of the possible moves makes the calculation time considerably smaller. The (very simple) approach to this presorting is that making a move that results into an inverted own stone is normally the best move, inverting an opponent stone is normally the worst one, and pushing out an own inverted stone is somewhere in between. The moves can be sorted to be calculated in the corresponding order, taking only about three times as much calculation time for calculating all possible moves than without presorting.

Due to the fact that the game tree becomes much *narrower* as the game goes on, calculation times drop drastically in the mid- and end-game. Therefore, about halfway into the game, the calculation depth is increased by two. However, another time increasing it later in the game, making it four more half turns, has proven to be too much, with calculation times increasing strongly.

There are settings to change the behavior of the algorithm. Presorting of the moves can be disabled, as well as Alpha-Beta-Pruning and on-the-fly calculation depth change.

4 Heuristic evaluation function

4.1 Overview

The heuristic function that is used in the implementation is rather simple and was only intended for testing purposes, however, it has proven to give a really good performance and due to the fact that it is simple its calculation time is rather short.

There are two main ideas that describe the heuristic evaluation function best.

The first idea is that it is *good* to invert more stones than the opponent. Therefore, it is best to invert a stone every round and preventing the opponent of doing so. This is in fact the one and only key to winning the game. However, this rule alone is not strong enough, because it would make it necessary to calculate rather deep into the game tree. On the other hand, this rule is really strong, so no matter how good the position on the board is, it can't outweigh a stone advantage.

The weaker rule, that selects the best position between all those positions that have the best value, regarding to rule one, takes the individual positions into account. Assuming that an own not inverted stone on an outside position is better than on an inside position, heuristic points are given for such positions. For the opponent it is the other way round: Opponent's normal stones on the outside are bad, opponent's normal stones on the inside area are good.

This heuristic function shifts the own stones to the edge and shifts the opponent's stones to the middle.

4.2 Detailed description

There are only 6 different positions on the board that are actually different and cannot be transformed into each other by rotations and/or mirroring.

```
|1|2|3|3|2|1|
|2|4|5|5|4|2|
|3|5|6|6|5|3|
|3|5|6|6|5|3|
|2|4|5|5|4|2|
|1|2|3|3|2|1|
```

Only these six positions are evaluated.

4.2.1 The Standard Invers Heuristic Evaluation Function

This function is a set of values that is very intuitive and therefore used for most benchmark tests. The other very nice property of the SIHEF is that it is rather unlikely to run into infinite loops when the random value is chosen correctly.

It gives a player points for *good* positions on the board and subtracts points for *bad* positions. The addition and subtraction of points is symmetric. Therefore, a positive value usally is *good*, a negative one is *bad*.

First, a winning (or losing) position is always evaluated with extreme values like maximal or minimal integer values. The other values have to be chosen in a way that it is impossible for the evaluation value to reach these extreme values while in a non-winning (or non-losing) position.

Then, if the board is neither a winning nor a losing position, 1000 points are given for each own inverted stone, and 1000 points are subtracted for

every opponent inverted stone. This value is chosen to be high, so that no position advantage can be big enough to outweigh a stone advantage, which seems legit at first. However, some experimental values indicate that this may not be true in every case.

Then, for every own not inverted stone on a specific position points are rewarded, and for every opponent stone points are subtracted.
Position 1: 10 points
Position 2: 20 points
Position 3: 30 points
Position 4: 40 points
Position 5: 50 points
Position 6: 60 points

4.2.2 The Additional Invers Heuristic Evaluation Function

This function has to be proven to be much better in playing the game than the SIHEF. However, it is much more prone to running into infinite loops. The rules are the same as in the SIHEF, only the values differ.

Number of inverted stones: 10000
Position 1: 10 points
Position 2: 30 points
Position 3: 50 points
Position 4: 90 points
Position 5: 150 points
Position 6: 300 points

This function penalizes stones in the middle of the board much more severely, which seems to be the reason for its better performance.

The Changed Additional Invers Heuristic Evaluation Function has the same values except for the value for the number of inverted stones where it has the value 300.

Interestingly, this function performs about the same as the AIHEF itself, although it treats a stone in the middle of the game similarly harsh as having more or less stones inverted yet.

5 Playing against itself

The program is capable of playing against itself with two different heuristic variable vectors and different calculation depths (however, not two different algorithms), and it is capable of playing whole sets of games against itself to determine the winner of a set of games. To make this approach work, there is an option for shuffling the possible moves, adding a bit of randomness to the game. One problem occurs, however. If two AI opponents play against each other, they often run into an endless loop, always making the same moves over and over again. Although a board setup where this may happen seems quite unlikely at first glance, this happens extremely often, making the program play against itself very unpractical. There is no game rule to prevent such behavior, though, so infinite loops are allowed by the game rules.

To prevent them in the program, shorter loops (4, 6 and 8 moves) are prevented automatically. However, due to the fact that loop detection is very time consuming, it seems to be better to turn this function off completely. Instead, there is a random value that changes the heuristic evaluation values a bit, making the selection of similarly good moves a bit random.

6 Experimental values

6.1 Calculation depth

The estimations in this chapter are very rough estimations for the efficiency of the algorithm, tested on a standard computer.

Plain Minimax without Alpha-Beta-Pruning is only capable of calculating the next *four* half turns, making it a rather weak approach. Six half turns are possible, though very time consuming and not applicable for real time playing.

Minimax with Alpha-Beta-Pruning is able to calculate the next *six* half turns in a timespan that is small enough to allow it for playing. As the game progresses and the game tree becomes narrower, eight half turns are possible, although the change of calculation depth is clearly recognizable in reaction time.

Minimax with move presorting proves to be the best approach, making *eight* half turns possible in a rather comfortable reaction time. Although the

presorting algorithm is very simple, it proves to be strong enough for making two more half turns possible with ease when using Alpha-Beta-Pruning.

7 How good is the program as a Player?

This question is not so easy to answer, because - as mentioned earlier, the game is relatively unknown, so there is no official league. It is even very hard to find people who have heard of the game before, or even played it before.

However, nobody has ever won against the program, and not even close. When playing with the best settings, it normally wins with at least two, more often with three stones in advance.

7.1 Playing Strength

The values in the following chapter were determined experimentally by letting the program play against itself rather often. The game sets were not huge, but large enough to give a rough overview over some qualities of the game and the program.

7.1.1 Same heuristic evaluation variables

All the values in the following table were determined using the same heuristic evaluation function for both players. The used function is the Standard Invers Heuristic Evaluation Function (SIHEF) that will be explained in the corresponding chapter. It is not the strongest function, but the most intuitive with a few nice properties.

P1 Calc. Depth	P2 Calc. Depth	P1 Wins	P2 Wins	Games
4	4	245	255	500
6	6	27	23	50

These numbers suggest that both players have roughly the same chance of winning. This is rather interesting, because on a 2x2-board Player One always has a winning strategy (rather trivially), and on a 6x6-board one would intuitivelly see Player One in advantage, because he is always either one stone ahead or has the same amount of inverted stones if both players

play good and are always able to invert a stone. However, Player Two seems to win about the same number of games.

P1 Calc. Depth	P2 Calc. Depth	P1 Wins	P2 Wins	Games
4	6	6	44	50
6	4	42	8	50

These numbers state that an increase in calculation depth is a big advantage, however, the player with a smaller depth still has a chance of winning.

Player one had calculation depth 4, Player Two had calculation depth 8, the played 10 games. Player One won 0 times, Player Two won 10 times.

P1 Calc. Depth	P2 Calc. Depth	P1 Wins	P2 Wins	Games
4	8	0	10	10
8	4	10	0	10

If the difference in calculation depth rises, the player with lower depth has almost no chance of winning anymore.

7.2 Different heuristic evaluation variables

All the values in the following tables are derived comparing the Standard Invers Heuristic Evaluation Function with another function that has proven to be better by far. The values used here are descussed extensively in the chapter about the Heuristic Evaluation Function. The game tree depth is 6 for both players.

However, letting the program play against itself often leads to infinite loops of the same moves. It is very hard to prevent this behavior. Infinite four-, six- and eight-move loops are handled in the code, but there is no real theoretical upper boundary for the size of an infinite loop. These loops can be prevented by adding some random values, but they also strongly depend on the heuristic evaluation variables. Choosing two variable vectors in a wrong way can lead to infinite loops in almost every game, despite using random values.

P1 H. Function	P2 H. Function	P1 Wins	P2 Wins	Games
SIHEF	AIHEF	15	85	100
AIHEF	CAIHEF	25	25	50

This is a very interesting fact, because the only difference between the AIHEF and the CAIHEF is the penalty for not moving a stone out of the board. In the AIHEF, this penalty is severe, whereas it is rather weak in the CAIHEF. Although one would expect the AIHEF to be superior, the experiment shows that the difference is not very big.

8 Instructions for using the Program

The program consists of two different parts, a back-end written in C++ that is responsible for all the calculations, and a front-end that is a graphical user interface written in JAVA using the swing library.

8.1 C++ back-end

The C++ program is a makefile project with two main targets: the target `release` and the target `test`. For playing and everyday use only the release target is necessary. The test target contains a unit test suite that is dependent on the CPPUNIT-library. To compile and run the tests, CPPUNIT has to be installed and the path to this installation has to be specified in the makefile. Compilation of the test project with `make test`, running of the tests with `./test_main`.

There is the file `constants.h` that should be presented seperately. In it all the important variables that change the application behavior are collected to be easily adjustable. Collecting them in a header file gives the compiler the possiblity to make some crucial optimizations, which is important considering the depth of the recursion tree. A few of the variables that can be adjusted in this file: calculation tree depth of both players, move presorting toggle, algorithm used, random value span etc.

The release target contains the runnable program. It only depends on the STL and can be built with `make release`. Under Windows there are a few dependencies from cygwin in the current version. So if the program has to run under Windows, there has to be a cygwin installation and the system path has to be set to its directory. This dependency only exists if the precompiled version is used. There is no problem with compiling the version under Windows, because its only dependency in the code is from the STL.

The program takes a few parameters:

1. parameter either `true` or `false`. It states if Player One is human or not.

2. parameter either `true` or `false`. It states if PlayerTwo is human or not.
3. parameter is the name of a file containing the heuristic evaluation variables for Player One.
4. parameter is the name of a file containing the heuristic evaluation variables for Player Two.
Both these files have to be in the same directory as the executable.
5. parameter either `true` or `false`. It states if the program is called by a GUI. Thus, it should always be `false` when calling the program on a command line.
6. parameter is an `int` value. It gives the amount of games that should be played in direct succession. This parameter is useful, if both players are AI players and play a series of games against each other for statistical questions.
7. parameter is the name of a file containing a saved game.

The program can be started without parameters, with the first two parameters, the first six parameters or all parameters. A few examples:

```
./invers true false
```
Play one game as Player One against the computer who is Player Two.

```
./invers
```
Two human players can play against each other.

```
./invers false false std_heuristic_values.txt \\
add_heuristic_values.txt false 50
```
The computer will play 50 times against itself, using different heuristic vectors for the two players. This can be used to determine if the first or the second heuristic vector wins more often.

```
./invers false true std_heuristic_values.txt \\
std_heuristic_values.txt false 1 savegame.txt
```
Load a game that is saved in the file savegame.txt and play as Player Two against the computer.

8.2 Java front-end

The Java program is a graphical user interface application using the swing library for making the graphical parts. It is written using netbeans, so the

code is present as a netbeans project, however, it should easily be importable to eclipse as well. If the directoray structure remains unchanged, everything should work out-of-the-box. If anything needs to be changed, a few things have to be taken into consideration.

If the Java program cannot find the C++-back end, probably the path to the C++ executable has to be set in the run arguments. IMPORTANT: The following file has to be present in the Java working directory: `std_heuristic-values.txt`.

Then the program can be started by pressing the Run button of the IDE. Everything else should be self-explanatory.

If the java file get called from the *command line*, you have to start the program with `java -jar Invers.jar ../../back_end/` from the directory `invers_bakk/front_end/dist/`. Make sure the file `std_heuristic_values.txt` is present in that directory as well, altough it should be by default.

9 The interface between the Java program and the C++ program

In the communication of the two applications, the C++ part makes all the heavy and interesting things. The Java part is very simple and can not do much on its own.

The Java program starts the C++ application via the runtime environment and sets up pipes for communication.

From the point of view of the Java program the communication is easy:

- Read the new playboard from the input pipe.
- Draw the board to the screen.
- Send `-1` to the output pipe if the OK button is pressed, or send the value of the move position (corresponding to the butten the player has pressed).
- Start again with reading.

From the point of view of the C++ program the communication looks like this:

- Read from the input pipe. Should receive either `-1` or a special command like `SAVE`.
- Send the current board to the output stream.
- Read the move the player has made from the input stream.
- Make that move internally.
- Send the new board to the output pipe.
- Now the AI move gets calculated (which can be time consuming). When finished, the application starts again with reading from the input pipe.

Thus, two cycles in the Java application correspond to just one cycle in the C++ program.

A board gets sent in the following way:
A Player One normal stone is encoded by `Y`, a Player One inverted stone by `y`, a Player Two normal stone by `R` and a Player Two inverted stone by `r`. These values are sent to the pipe, delimited by a space character. The first character that gets sent is the Player One Hand, then the Player Two Hand and then the board, stone by stone, starting from position 0 (top left) to 35 (bottom right).

After that, if the user is allowed to make a move (because it is their turn), a list of all allowed moves gets sent in the same way, encoded as their turn position, again delimited by a space character. This information is used by the Java program to enable all the buttons that can be used by the user because they encode allowed moves.
A possible string sent from the C++ to the Java application could look like this:

```
r y R Y R r Y r r Y r r Y R Y R Y y r R r Y Y y y \\
R R y r r Y r Y r r Y y r 0 4 6 13 23 12 5 14 22 21 1
```

Although this board does not make sense in a perspective of the game, it is perfectly valid for the Java program and it would draw the board without any complaint.

Overall, the interface between the two programs is very simple. Thus, the Java application is very simplistic, too. It has no internal state that knows anything about the game itself, it can only draw a board to the screen and send information back to the C++ program, according to the button the user has pressed.

References

[1] *Amazon Page of the Game Invers*. Sept. 21, 2015. URL: http://www.amazon.de/INVERS-Das-Schiebe-und-Umdrehspiel/dp/B002RIA490/ref=sr_1_1?ie=UTF8&qid=1442831357&sr=8-1&keywords=invers.

[2] *Boardspace Invers Game Page*. Sept. 21, 2015. URL: http://boardgamegeek.com/boardgame/1082/invers.

[3] *Boardspace Main Page*. Sept. 21, 2015. URL: http://boardspace.net/english/index.shtml.

[4] *Gipf Project Homepage*. Sept. 21, 2015. URL: http://www.gipf.com/.

[5] *Wikipedia-Entry of Kris Burm (German)*. Sept. 21, 2015. URL: https://de.wikipedia.org/w/index.php?title=Kris_Burm&oldid=145854335.