
Bachelor Thesis

A.I. in board games

Aron Lindberg, s042422

Technical University of Denmark
Informatics and Mathematical Modelling
Thesis tutored by Thomas Bolander

August 19, 2007

Abstract

In this thesis, a board game named **Kolibrat** is implemented in the *Objective-C* programming language and Apple Computers *Cocoa* API. Besides from documenting this work, the thesis focus on the development of artificial players.

Resumé

I denne afhandling bliver et brætspil **Kolibrat** implementeret i programmerings sproget *Objective-C* og Apple Computers *Cocoa API*. Udover at dokumenterer dette arbejde fokuserer denne afhandling på arbejdet med udviklingen af kunstigt intelligente spillere.

Contents

1	Preface	1
1.1	Preconditions	2
1.2	Aims and limitations	3
1.2.1	Limitations	3
1.3	Structure of thesis	4
2	Game Development	5
2.1	Concept design	5
2.1.1	Graphical user interface	6
2.1.2	Game Controller	6
2.1.3	Game Engine	6
2.1.4	Players	6
2.2	User Interface	6
2.2.1	Flow control	7
2.2.2	GUI implementation	8
2.3	Game Implementation	9
2.3.1	Class details	11
2.3.2	Kolibrat flow diagrams	16
2.3.3	C data structures	16
2.4	Testing	18
2.4.1	Test summary	19
3	Artificial intelligence	20
3.1	Game Analysis	20
3.1.1	Game state space	20
3.1.2	Branching factor	23
3.1.3	Complete analysis of Kolibrat	25
3.1.4	Analyzing forced loops	27

3.2	AI in Kolibrat games	29
3.2.1	Mini-Max agent	29
3.2.2	Optimizing the Mini-Max agent	30
3.3	Implementing the Mini-Max agent	32
3.3.1	Additional possible Mini-Max enhancements	33
3.4	Optimizing the Heuristic function	35
3.4.1	Neural network utility function	35
3.4.2	Weighted linear evaluation function	36
3.4.3	Choosing the heuristic parameters	37
3.4.4	Determining the parameters weight	38
3.5	Simulated annealing Implementation	39
3.5.1	Simulated annealing results	42
3.6	Heuristics Comparison	44
4	Conclusions	48
4.1	Future prospects	49
A	Kolibrat Rulebook	50
A.1	Game Objectives	51
A.2	Rules for movement	51
B	Tests Details	54
C	Source Code	58
C.1	Kolibrat Source Code	58
C.1.1	HumanPlayer.h	58
C.1.2	HumanPlayer.m	59
C.1.3	Datastructures.h	61
C.1.4	Datastructures.m	64
C.1.5	GameLogic.h	69
C.1.6	GameLogic.m	70
C.1.7	GameEngine.h	80
C.1.8	GameEngine.m	81
C.1.9	GameController.h	88
C.1.10	GameController.m	89
C.1.11	GameBoard.h	92
C.1.12	GameBoard.m	93
C.1.13	NewGameSheetController.h	98
C.1.14	NewGameSheetController.m	99

C.1.15	GUIProtocol.h	104
C.1.16	PlayerProtocol.h	104
C.1.17	main.m	105
C.2	Kolibrat Test Source Code	105
C.2.1	Kolibrat Test.m	105
C.3	MiniMax Source Code	125
C.3.1	AIDefinitions.h	125
C.3.2	AIDefinitions.m	127
C.3.3	AdvancedAI.h	131
C.3.4	AdvancedAI.m	132
C.4	Simulated Annealing Source Code	141
C.4.1	simAneling.h	141
C.4.2	simAneling.m	143
C.5	Forced Loops Source Code	149
C.5.1	FakeLogic.h	149
C.5.2	FakeLogic.m	150
C.5.3	Forced Loops.m	154
Flow Diagrams		163
Bibliography		166

Chapter 1

Preface

The first thing to do when one is about to write a thesis about developing artificial intelligence for a board game, it to choose the actual board game. The choice is not critical, but the game should have certain properties. The properties that I have looked for in the games, that I have considered to use in this thesis, is basically two things. First the game must have no clear strategy for winning, meaning that there should be no set of obvious moves that ensure one of the players victory no matter what the other player does.

The second property is that I want a game with simple rules and is easy to learn. Still the game must be difficult to predict and master. Hopefully the game also satisfy other properties like being entertaining, but these properties is less important than the first two. After discussing the choice of possible games, my tutor Thommas Bolander and I have decided to use the rather unknown board game Kolibart as it seems to satisfy all the properties given above. In addition it can be played on multiple board sizes.

With the game chosen the only thing left to decide is the means of implementation. The most logic choice here would be *Java*, as it has been the programming language of choice in most classes. But since I believe in variety, and that learning something new is always good, I have made the choice to implement Kolibrat in the less known language *Objective-C*. I choose this language because it seems to be a powerful and structured language. The language is also the language of choice for programming

on an *Apple Macintosh* as Apple supplies a powerful **IDE** and **API** that uses *Objective-C* as its foundation.

1.1 Preconditions

In order to get most out of this thesis the reader is expected to be familiar with the rules of Kolibrat, if not they can be found on page 50 in appendix A. Understanding *Objective-C* would be a great help in order to understand the source code, but since *Objective-C* is a superset of *ANSI C* most of the source code is readable to people with a good understanding of *C*. Likewise the reader is also expected to know the **UML** notation.

Objective-C is however a unique programming language, and it uses some terms that is unique to the language. To avoid misunderstandings words that have a special meaning in *Objective-C* or could be misleading if shortly explained below.

Protocol Is the same to *Objective-C* as interfaces is to *Java*. It specifies a list of methods that the class must implement.

Delegate Certain *Objective-C* classes have delegate methods. If a delegate for some object has been set, all method calls to its delegate methods are forwarded to the delegate object, but only if the delegate object implements a method by that name.

Notification server *Cocoa* implements a notification system that allows all objects to send and receive event messages, even if they have no idea about who the sender or receiver might be. This is used for program wide signaling of events.

Sheet Is a special window that is attached to another window, and glides in above that window. This is often used to display save dialogues, warnings, or simple options, as it locks the window below from the user.

NIB files A file format used to store the GUI interface in *Cocoa*. It consist of an XML list and interface objects. ([NeXT Interface Builder](#))

1.2 Aims and limitations

Since time is always a limit, not everything of interest can be tried out and tested to its full extend. Therefor some topics, interesting though they are, will not be touched in this thesis. To limit the thesis a set of objectives and limitations has been made, they are specified below.

Implementing a working game

Implementing the game has the first priority, as a working game is necessary in order to run and test **artificial players**. The game itself should be simple, but look and feel elegant, while giving the user a way to set the options he needs. The game should also be constructed with flexibility in mind, meaning that it should not rely on constants, but variables that can change at runtime. This ensures that almost all aspects of the game can be easily altered either by the user or the programmer.

Implementing AI

Implementation of the artificial intelligence in Kolibrat is the main area of focus in this thesis. As with the implementation of Kolibrat I want the artificial players to be flexible. Hopefully the game can be implemented in a way that allows it to load different AI's as plug-ins at runtime. This will allow the user to add or remove AI's as he pleases. Writing the artificial players as plug-ins also has the advantage, that other programmers can develop their own AI's without having the source code for Kolibrat.

The plan is to spend as much time on the development of artificial intelligence as possible. As a minimum requirement, at least one AI using the **MINIMAX** algorithm, must be implemented and optimized as much as possible.

1.2.1 Limitations

On the other hand technologies like **Sound** and **3D Graphics** will not be touched in this thesis. Even though they are interesting add-ons, they are not essential for the game experience. In a small game like Kolibrat, one could even argue that this is an advantage since the game can be played while the user performs other activities.

Adding **Network** game-play would be interesting, but is not essential and therefore not a topic that will be touched further, in this thesis. The same goes for the ability to undo moves and saving or loading games.

1.3 Structure of thesis

The main content of this thesis is structured into two chapters. Chapter [2](#) deals with the development, design and implementation of Kolibrat. Afterwards chapter [3](#) deals with a mathematic analyze of the game Kolibrat, along with the development and implementation choices made while developing the artificial players.

Section [2.1](#) goes through the details of the basic design choices made while developing Kolibrat. Section [2.2](#) deals with the development of the games graphical user interface. Section [2.3](#) deals with the more detailed design of Kolibrats internal objects and goes into details about the implementation of Kolibrat. Finally section [2.4](#) concludes the first chapter by going into details about the *White-Box* testing done on Kolibrat.

Chapter [3](#) starts off with a longer mathematical analyze of Kolibart in section [3.1](#). Section [3.2](#) gives an introduction to the field of artificial intelligence in two player games, it also gives a brief overview of the available algorithms used in these situations. Section [3.3](#) deals with the implementation of the MINIMAX algorithm, and is followed by section [3.4](#) that deals with optimizing the **heuristic function** used by the MINIMAX algorithm. Finally section [3.6](#) compares the different artificial players that has been developed for Kolibrat.

The thesis is finished with a chapter of conclusions, giving a summary of the achievements in the thesis.

Appendix [A](#) contains the KOLIBRAT rule-book and appendix [B](#) gives details on the the *White Box* tests performed on Kolibrat. The following appendixes lists the source code for the entire Kolibrat game, and its artificial player.

Chapter 2

Game Development

This chapter deals with all aspects of the development of Kolibrat. First a conceptual design of Kolibrat is devised. After that there is a section on user interface development. These sections are followed by a section on game design and then by one detailing the game implementation.

2.1 Concept design

From the very beginning Kolibrat was developed with flexibility in mind, thus making it easy to extend and change later on, it also follows the model, view, control design pattern. This allow the game to run without a GUI, or to run with different GUI's without the need to change anything in Kolibrats data structure. The initial concept of the game design is shown on figure 2.1.

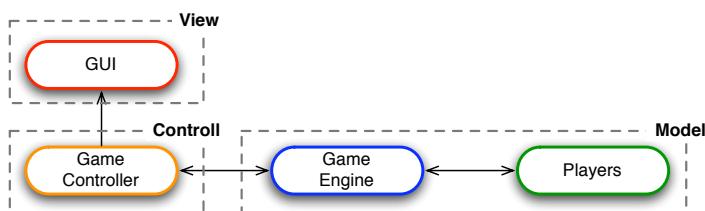


Figure 2.1: Concept Design

2.1.1 Graphical user interface

The GUI must allow the player to make certain choices at the beginning of a game, the most important being the board size, the players type and player names. In a game the GUI must show the score and the game board, while at the same time give the user a chance to quit or restart the game. Another important part of the GUI is to redirect all mouse clicks to other parts of the program, like the player objects.

2.1.2 Game Controller

The game controller must be able to control the game window and the new game options. In addition to that it should send the options chosen in the beginning of a game to the Game Engine. It must also tell the GUI to display game over information when one of the players wins.

2.1.3 Game Engine

The Game Engine will handle the game logic, like the game rules. It must also send game information to the players, giving them a chance to move, and send these moves on to the GUI.

2.1.4 Players

The player classes must respond to move requests made by the Game Engine and return moves to the Game Engine. In order to allow different types of player classes to work with the Game Engine, the game engine should be able to communicate with player objects that does not necessarily inherit from or subclass each other.

2.2 User Interface

The user interface is in many ways the most important part of a program since it is the link between the user and the program. The goal with the development of Kolibrats interface has been to create a simple, complete and elegant GUI. One that gives the user the options he needs and nothing else.

The best way to make the GUI simple, would be to make the game a single windowed application, that shows the game board and use a sheet to change game settings. This approach has a number of advantages

compared to having more than one window. First of all window control becomes simpler and the final design takes up less space. Besides from that it makes sense to use a sheet because it ensures that no information on game settings is shown when it is not needed, and the sheet locks the game window ensuring that the player can not move pieces around while setting up a new game.

2.2.1 Flow control

In order to ensure a logical flow of events through the game a **flow diagram** has been made, and can be seen on figure 2.2. The only option available to the user at launch is the new game option, which sends him on to the dialogue for choosing game settings for a new game. When the user is done, the game will start with a red as the first player to move. The states now alternate between the states where either black or red has the turn until the game is over. When the game is over the user will be able to start a new game, restart the game or quit, but this is not shown in figure 2.2 in order to simplify it. As seen in figure 2.2 the user also have the choice to begin a new game, restart or quit, at any time he wants.

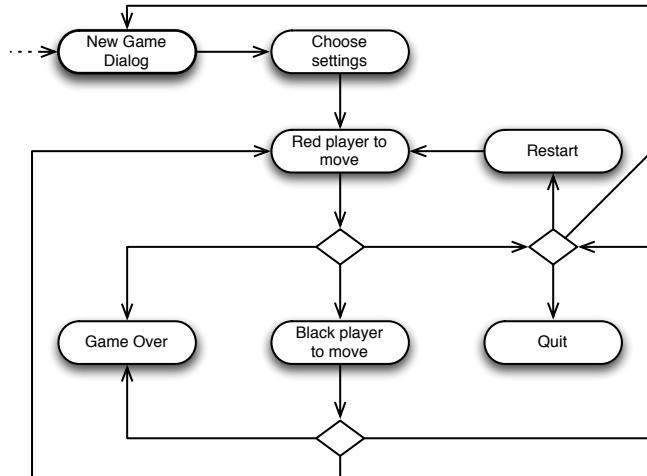


Figure 2.2: UML Flow Control Diagram.

2.2.2 GUI implementation

The user interface has been made in the program **Interface Builder** and is stored in an **NIB** file. Thus no actual code has been written to produce the GUI. The GUI is linked to the source code by using identical variable and class names in both the GUI and the source code.

The GUI itself is designed to follow *Apples Human Interface Guidelines*, to produce a game that looks familiar to a mac user. A picture of the user interface is shown in figure 2.3.

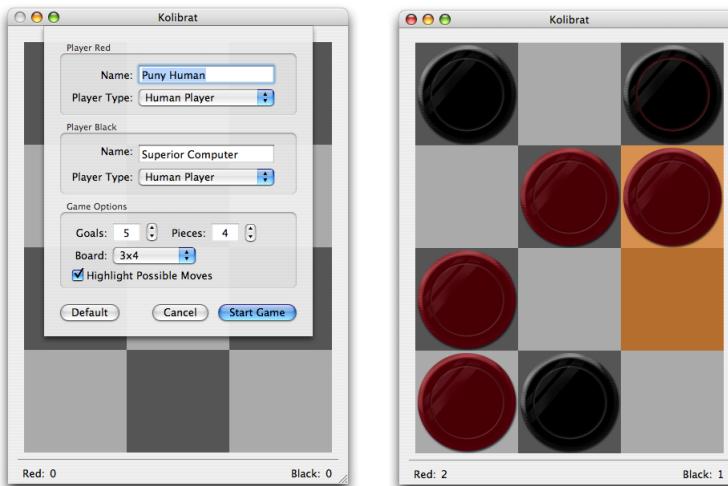


Figure 2.3: Kolibrat settings and game board.

The new game window allow the player to choose the size of the game board, the maximum amount of pieces each player can have on the board and the number of goals a player must gain to win.

In this layout the users can only choose from predetermined board sizes. This is not a technical limit, the game can at runtime begin a game on any board size, but in order to save the user from choosing stupid board sizes like 1x1 or 50x99 the choice of boards has been limited to some predetermined board sizes.

The game window can also be resized by the user. The maximum size

of the window is when each square on the game board has a dimension of 128x128 pixels. From that the window can be scaled down until each square has a dimension of 64x64 pixels. The algorithms that handles the resizing, takes care to ensure that the window keeps its proportions while resizing. An example of this can be seen on figure 2.4.

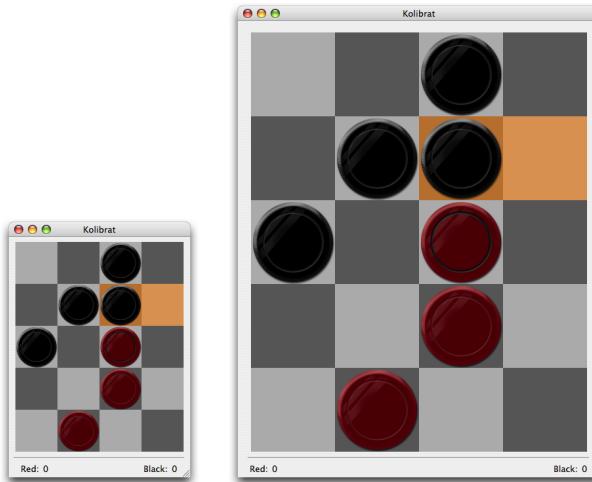


Figure 2.4: Big and small game window.

2.3 Game Implementation

In this section, the design from figure 2.1 is refined into a more exact model representing actual objects in the final game. The result is shown in diagram 1.

The most notable differences in the new layout is that the control part of the game has been split into two controller objects. `NewGameController` handles the window used to start new games. The `GameController` is used to control the main game window. The `GameController` also controls a customised `NSView` that draws the actual game board in the `GameWindow`.

Another change is the addition of the `GameLogic` object. This functionality has been moved from `GameEngine` into its own object. This will allow

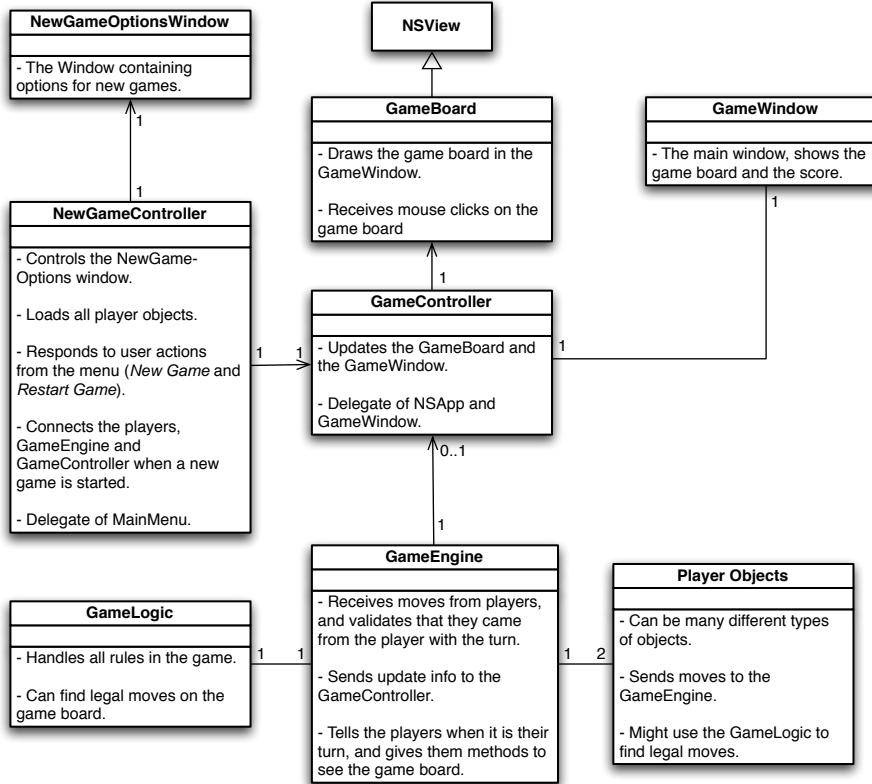


Diagram 1: Concept UML Diagram

other objects like artificial players to use the same object, for finding moves and move around on the game board, as the **GameEngine** does.

A little less noticeable is it that **GameBoard** is in charge of all mouse clicks that are received on the game board. This has the advantage that since **GameBoard** draws the game board, it can easily transform mouse coordinates into game board coordinates. The game board coordinates can then be passed on to the rest of Kolibrat by using the *Notification* system.

The games menu bar is controlled by the **NewGameController**. This is the only logic choice since the menu bars main features are restarting or starting a new game. It is also the class responsible for loading the plug-in player objects. Also a player object is now defined as an object

that implements and responds to a `PlayerProtocol`. The exact interface in this protocol can be seen on page 104. A `GUIProtocol` has also been defined and specifies the methods a GUI must implement, if one wants to make a new GUI. This could be a terminal interface, or a GUI for the web.

2.3.1 Class details

This section goes into details about the individual classes in Kolibrat, and explain the workings of some selected methods. The individual methods are not discussed sine most methods is self-explanatory and the source code is well documented and have a small description of almost all methods in the game.

GameLogic

The `GameLogic` class is the object that implements all the rules in Kolibrat. When initialized the object receives information on the maximal number of pieces that players can have on the board, the amount of goals needed to win and the board size. With this information the class can return legal moves and make moves on a `GameState`. Some of the methods have been implemented in C to improve performance, since that became an issue with the development of artificial players. In figure 2.5 the methods whose return type is not enclosed in brackets is implemented in C.

<code>GameLogic</code>	
-	<code>int maxGoals</code>
-	<code>int maxPicesOnBoard</code>
-	<code>BoardSize boardSize</code>
-	<code>(void)changePlayerOn:(GameState *)gs</code>
+	<code>(id)initWithMaxPices:(int)max goalsToWin:(int)goals boardSize:(BoardSize)board</code>
+	<code>(GameState*)CreateNewGameState</code>
+	<code>(void)resetGameState:(GameState *)gs</code>
+	<code>(BOOL)currentPlayerCanInsertPieceOnState:(GameState *)gs</code>
+	<code>(NSSet *)legalMovesForPieceInField:(BoardField)field withState:(GameState *)gs</code>
+	<code>(NSSet *)allIllegalMoves:(GameState *)gs</code>
+	<code>(BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs</code>
+	<code>(BOOL)playerMovingCanInsertPieceOnState:(GameState *)gs</code>
+	<code>SimpleList *allLegalMoves(Gamestate *gs)</code>
+	<code>void legalMovesForPieceInField(BoardField *field, GameState *gs, SimpleList *superList, SimpleList *goodList, SimpleList *badList)</code>
+	<code>BOOL makeMoveOnState(BoardMove *playerMove, GameState *gs)</code>
+	<code>void freeGameState(GameState *state)</code>
+	<code>GameState copyGameState(GameState *state)</code>
+	<code>BOOL blackPlayerAdheadOf(int x, int y, GameState *gs)</code>
+	<code>BOOL redPlayerAdheadOf(int x, int y, GameState *gs)</code>

Figure 2.5: The GameLogic class

GameEngine

The `GameEngine` class stores the games state. Is parses game information on to the GUI, if one is connected. It also handles all communication between the players and the rest of Kolibrat. In order to avoid having to maintain two set of game rules the engine uses `GameLogic` to validate the players moves, by searching for the players move in the set of legal moves returned by `GameLogic`. The class also ensure that there is at least 0.5 second between two moves to make games between two fast artificial players watchable. The methods in `GameEngine` can be seen on figure 2.6.

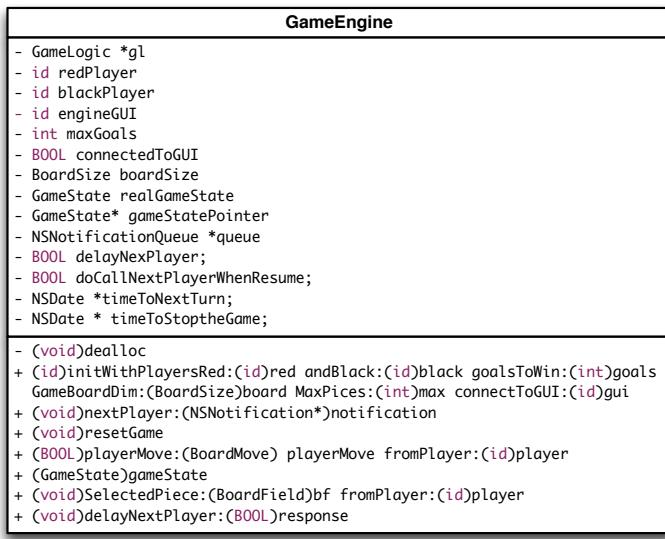


Figure 2.6: The GameEngine class

The method `SelectedPiece:` can be called by the players on pieces they want highlighted in the GUI. And the `delayNextPlayer:` method is called by `NewGameController` to stop a game between two artificial players when the user brings up the sheet with `NewGameOptions`.

Human Player

This class implements the human player. It works by receiving mouse click notifications from the `GameBoard`. When appropriate the human player

object returns possible moves to the `GameEngine`. Besides from this, the object only implement the methods required by the `PlayerProtocol`.

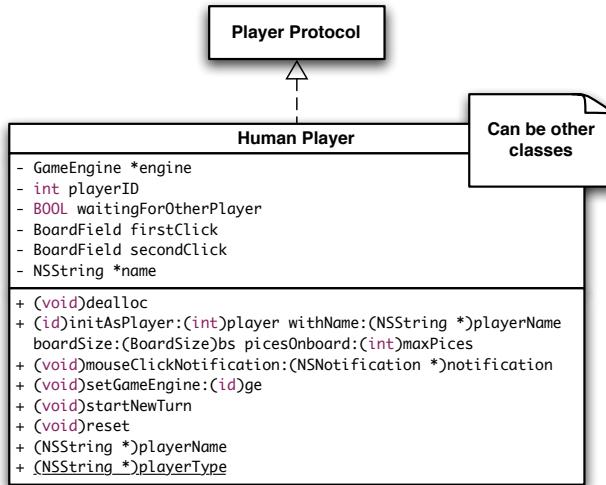


Figure 2.7: The `HumanPlayer` class

GameController

This class is the link between the interface and the data model. It receives information from the `GameEngine` and sends the appropriate information on to the `GameWindow`. This class is initialized by the `awakeFromNib:` method that is called by the `GameWindow` when its NIB file is loaded at launch. The classes variables and methods can be seen on figure 2.8. At compile time `IBOutlet` is converted to a pointer, and `IBAction` to `void`. `IBOutlet` and `IBAction` us only used to inform the programmer and the compiler that this is a variable or method that is linked to the GUI through a NIB file.

The `GameController` is a delegate of `NSApplication` and implements the delegate method `applicationShouldTerminate...:` that terminates the application when the last window is closed. The `gameOverWithWinner:` method is called by the `GameEngine` when a game is over, and when the user responds to the game over dialog the `gameDidEnd:` method is called to cope with the users response.

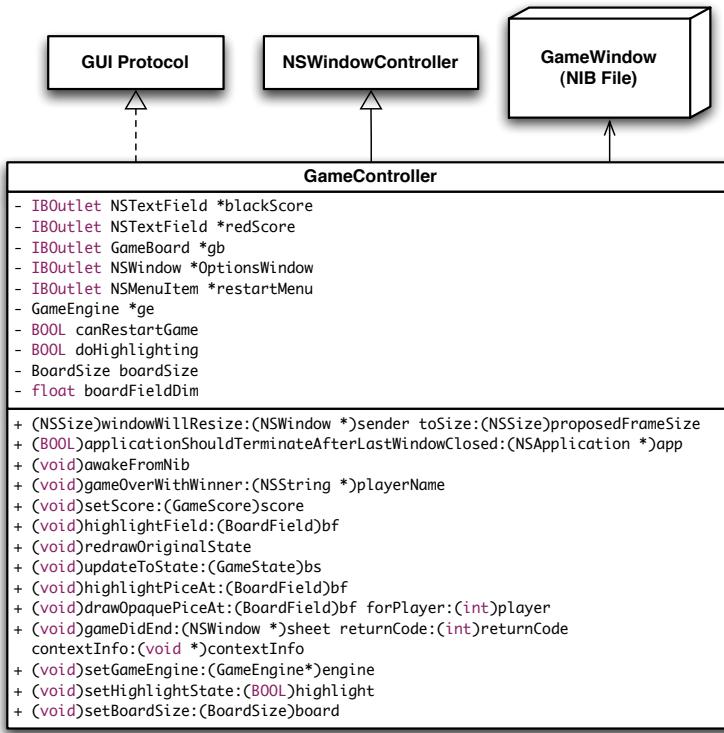


Figure 2.8: The GameController class

GameBoard

The **GameBoard** class handles drawing the game board in the main window. It is loaded by **GameWindow** at launch. Most of its methods are related to drawing the board and its pieces. The **mouseDown:** method is the method that receives mouse clicks, and transforms them into board coordinates. All actual drawing is done in the **drawRect:** method, as it is automatically called when the system wants to redraw the window.

The **GameWindow** uses some quite advanced calculations to resize the game window. The **setSquareDim:** and **setDisplayOffset:** is part of this and ensures that the game board have the right size and is placed at the correct distance from the edge of the window.

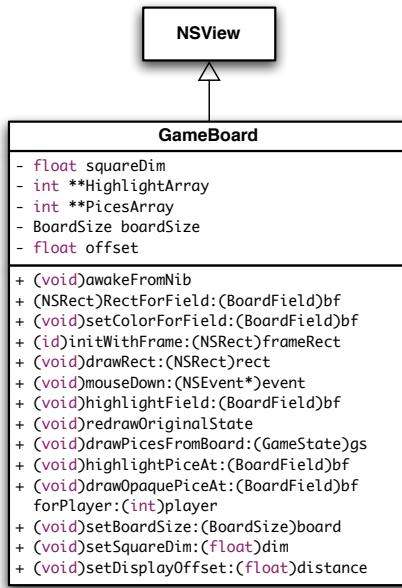


Figure 2.9: The GameBoard class

NewGameController

The `NewGameController` class handles all aspects of new game creation. At launch it checks the games plug-in folder for additional players and loads these into an array of players. When the user choose to start a new game the `NewGameController` displays the `NewGameOptionsWindow` above the game window as a sheet. In the `NewGameOptionsWindow` the user can choose the game settings, along with other options. The class can be seen on figure 2.10.

The `validateMenuItem:` is used to enable or disable items in the menu bar. This method is used to disable the Restart Game menu item until a game has been started. The method is called on the `NewGameController` because it is the delegate of the `NSMenu` class. The methods `newGame:` and `restartGame:` is the classes that is executed when the user chooses these options in the menu bar. The `defaultsButton:, cancelButton:` and `startGameButton:` is the methods that is executed when the user pushes these options in the `NewGameOptionsWindow`.

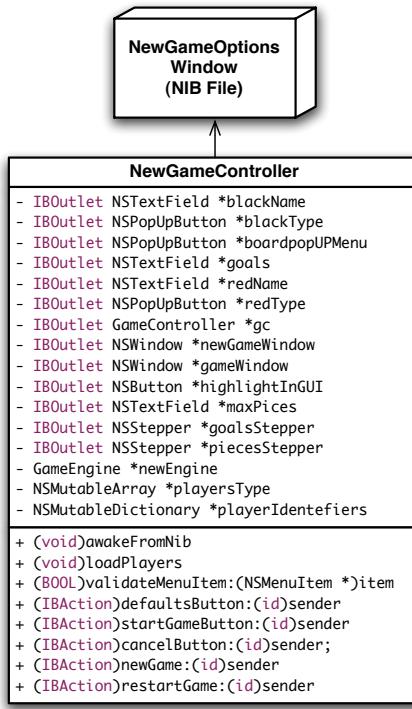


Figure 2.10: The NewGameController class

2.3.2 Kolibrat flow diagrams

In order to better understand the flow the events through Kolibrat, when the game is running, some UML flow diagrams has been made to illustrate this. Due to their size the actual flow diagrams is shown in the back of the report, behind the appendix. Diagram 2 is shown on page 163 and displays the program flow at game launch. Diagram 3 is shown on page 164 and details the flow while playing through one turn. Diagram 4 is shown on page 165 and shows the flow of events when one of the players win a game.

2.3.3 C data structures

This section describes the data structures used in Kolibrat. It gives a superficial explanation on how the data structures is implemented and focus on what the structures is used for. To see the actual implementation

and how the structures are made from primitive C variables see the implementation on page [64](#).

GameStatus This data structure is used to store the game status. That is whether or not the game is over, and in that case who the winner is. This data structure is mostly used in the larger data structure **GameState** that stores all data on a state in the game.

BoardField This structure is used to parse coordinates for pieces around in the game. It is constructed of two short integers that respectively gives the x and y coordinate of the piece in question.

BoardMove This structure is made up of two **BoardField** structures, and denotes the field a piece moves from and the field it moves to. This structure is used widely in the game, **GameLogic** uses it to store legal moves and the player objects uses this structure to pass the moves they want to make on to the **GameEngine**.

GameScore This structure stores the score for both players, in two short integer variables. This structure is mostly used in the larger **gamerate** structure.

BoardSize This structure stores the size of the game board. All objects that works with pieces on the game board needs to know the size of the game board in order to avoid array out of bounce errors, they uses this structure to store that knowledge.

BoardFieldContent This is a small structure defining two boolean variables. One to indicate that this **BoardField** is occupied by red and one to indicate it is occupied by black. This is used by the **GameStatus** in a two-dimensional array to store the location of the pieces of the board.

GameState This structure is used by the **GameEngine** to store all information on the game, and by the AI's when they construct a game tree. Aside form containing a two-dimensional array of **BoardFieldContent** structures, a **GameScore** and **GameStatus** structure it also stores info on the player moving, and the amount of pieces each player have on the board.

The `GameState` structure has been designed to save as much space as possible. The score is stored in an unsigned short integer for both players, as each unsigned short integer is 16 bits the total size of this is 32 bits. The placement of pieces is stored in a two-dimensional array with the size of the game board of booleans for both players, assuming the game board have 3×4 fields this gives $3 \cdot 4 \cdot 2 = 24$ bits of data to store the location of both players pieces. The player who have the turn is stored in a boolean variable and only takes up one bit. The number of pieces that both player has on the board is also stored in an unsigned short integer and thus takes up $2 \times 16 = 32$ bits, the same as the game scores. The game status is stored in two Boolean variables one to tell whether the game is over and one to tell the winner. This adds up to a total of 91 bits or a little less than 12 bytes. This value might vary, especially on 64 bit systems where some of the primitive C variables has changed size, compared to their 32 bit equals.

2.4 Testing

This section contains a description of the testing done on the kolibrat souse code. All classes has been severely black box tested, both by crash testing the compiled application, and through heavy use of the debuggers line by line execution provided by the IDE.

In addition to this the GameLogic part of the game has been put through a white box test, to ensure that it contains no errors and that the rules of the games is implemented as intended. All in all 20 test cases has been carried out testing different aspects of the GameLogic.

A short description of the tests is shown below. Fore a more in-depth description of the tests see appendix [B](#) for a detailed description of the tests. Or take a look at the source code of the test at in appendix [C.2.1](#).

Test 1 Insert pice for red in (1,0) on an empty board.

Test 2 Insert pice for black in (1,3) on an empty board.

Test 3 Trying to insert pice for red in (1,3) on an board with 4 red pieces.

Test 4 Trying to insert pice for red in (2,2) while the board is empty.

Test 5 Score point for red.

Test 6 Score point for black.

Test 7 Try to insert piece in occupied field.

Test 8 Gamestatus is changed when red wins.

Test 9 Ensure that no players can move when the game is over.

Test 10 Ensure that no players can move outside of the board.

Test 11-15 Tests of moves on a non empty board for red player.

Test 16-20 Tests of moves on a non empty board for black player.

2.4.1 Test summery

These tests combined tests all functions and all lines of code in `GameLogic` to ensure that it responds as expected in all game situations. This means that besides from testing all lines of code in `GameLogic` is also attempts to find any errors there could be in the implementation of the Kolibrat rule-book found in appendix A.

During the tests two errors, that in some situations allowed both players to remove pieces from the game board, where found in test 11 and 16. The problem has been fixed, so that `GameLogic` now passes all the tests.

Chapter 3

Artificial intelligence

This chapter focus on the development and implementation of artificial intelligent players. The first section makes a longer analyses of the mathematical properties that Kolibrat possess. While the flowing sections describe AI in general and the implementation and optimizations done on Kolibrats search techniques.

3.1 Game Analysis

To know what to expect from an artificial player, it is good to have an idea of what one can expect from Kolibrat AI. Therefor some mathematical studies of Kolibrat has been made prior to the AI development to determine properties like the **branching factor** and the size of the search space, meaning the number of the unique **GameStates**. These properties can tell if it is possible to solve the game completely, or how many moves an agent can be expected to search before a move must be returned.

3.1.1 Game state space

The total number of different states in a Kolibrat game important, since if this number is small enough the game can be solved in an attempt to find a certain **victory strategy** for one of the players. To do this we first of all need a formula that describes the number of ways, a piece can be placed on the game board.

If we define that b is the number of fields on the game board and that

p is highest number of pieces, a player can have on the board. The first piece can be placed in b different places, and the second in $b - 1$ ways. In other words if we have x pieces on the board they can be placed on $f(x)$ different ways.

$$f(x) = \frac{b!}{(b - x)!}$$

This formula works fine, but have the one flaw since it considers all pieces to be different. So if red has placed his first piece in (1, 1) and his second piece in (1, 2) this board state is considered different from a state where red placed his first piece in (1, 2) and the second in (1, 1). To solve this the result of the formula must be divided by the factorial number of pieces on the board, and this must be done for each player individually. If p_1 is the number of pieces that the red player has on the board and p_2 is the number of pieces that black has on the board the formula becomes (3.1).

$$f(p_1, p_2) = \frac{b!}{p_1! \cdot p_2! \cdot (b - x)!} \quad (3.1)$$

Formula (3.1) gives the total number of different ways to place pieces on the board. To get the size of the total amount of states the result from formula (3.1) must be multiplied by two, since for each board position both players could have the turn, and all these states could each have any possible combination of scores. If the number of goals that is required to win is s then formula (3.1) must therefore be multiplied by $(s + 1)^2 - 1$. This gives the formula shown on equation (3.2).

$$f(p_1, p_2) = \frac{b!}{p_1! \cdot p_2! \cdot (b - x)!} \cdot 2 \cdot (s + 1)^2 - 1 \quad (3.2)$$

To get the total number of possible states the sum of all possible combinations of pieces is taken. This is done in equation (3.3).

$$\sum_{p_1=0}^p \sum_{p_2=0}^p \left(\frac{b!}{p_1! \cdot p_2! \cdot (b - x)!} \cdot 2 \cdot (s + 1)^2 - 1 \right) \quad (3.3)$$

With formula (3.3) it is now possible to calculate the total number of states that a Kolibrat game has, based on the size of the board and the

highest number of pieces each player can insert. In table 3.1 calculations for the total state space can be seen for some common board sizes. The values is based on games that is won at 4 points.

Breadth	Height	Max Pieces	Board positions	Total States
2	2	2	63	3149
3	4	4	170019	8500949
3	4	5	343467	17173349
3	4	6	460815	23040749
4	5	5	123479901	6173995049
4	5	6	509103141	25455157049
5	6	6	$1.58 \cdot 10^{11}$	$7.91 \cdot 10^{12}$
9	9	15	$4.20 \cdot 10^{30}$	$2.10 \cdot 10^{32}$
9	9	30	$2.66 \cdot 10^{38}$	$1.33 \cdot 10^{40}$

Table 3.1: States based on board size, score and pieces on the board.

As seen in the table a standard Kolibrat game with a 3x4 game board and a maximum of 4 pieces on the board for each player only has 8500949 states. If each state takes up 12 bytes, this gives that all states will take up 102 MB of memory, plus some memory needed for bookkeeping. While this is still a considerable amount of memory it is well within the limits of a modern computer to work with. Given the time a computer could calculate and solve the complete game to find the best possible strategy for winning.

The memory needed for the bookkeeping is actually also quite a bit, assuming the computer system solving the game is a 64 bit computer like most modern computers today. A pointer takes up 64 bits of data or 8 bytes, if every state must have a pointer to all states accessible from itself this means quite a bit of extra data. Assuming that every state has about 4 possible moves, this means that every state must have four 64 bit pointers to other states, with 8500949 this gives an additional 272 MB data to be stored in order to connect the states to each other. This adds up to a total of a little less than 400 MB for a complete solution to a kolibrat game on a 3x4 board.

This number can be reduced be a factor of two by implementing an algorithm that can invert the board so red becomes black, and black red.

By also implementing an algorithm that can mirror the board along the y-aksis the total amount of unique states can be divided by a factor close to 2. The factor is only close to 2 because a state that is symmetric only appears once in the complete set of states, where as all non symmetric states appears twice. With these improvements to an algorithm it will be possible to store the complete solution to the kolibrat game in a little less than 100 MB file.

While it certainly is possible to solve the smaller Kolibrat boards completely, solving the larger is still not possible today. The game draughts have $5 \cdot 10^{20}$ unique states and have recently been solved proving that both player have a draw strategy [1]. It took 16 years from the project started to the proof were complete, demonstrating that solving the larger Kolibrat games is impossible unless massive computer mainframes work on the problem for yeas. For compairson chess have about 10^{50} unique states [2], and have never been solved.

3.1.2 Branching factor

y-axis A games **branching** factor is the factor by witch its **game tree** branches, in short the number of moves the player has to chose from when he makes a move. The branching factor is important since is determines the number of states an artificial player must look through to find the best move by looking ahead in the game. If an agent has to look d moves ahead in a game with a branching factor of b to find the best move, the amount of states s the agent must look through is determined by formula (3.4).

$$s = \sum_{i=0}^d b^i = \frac{b^{d+1} - 1}{b - 1} \quad (3.4)$$

In Kolibrat on a 3x4 board and with a maximum of four pieces, on the board the average branching factor has been determined to be about 3.12 on average. This number also seems fairly constant, and different playing styles has little to no effect on the factor. Even even if the average branching factor seems quite constant based on several measurements with different playing styles, the individual branching factors varies quite a lot from turn to turn. The minimum branching factor is zero and although this is a pretty rare situation it do appear, one example can be seen in

figure 3.1(a). The maximum branching factor has been determined to be ten and is equally unlikely to appear in a game, as only a few board positions gives a player this many choices in moving. One example of such a position can be seen on figure 3.1(b).

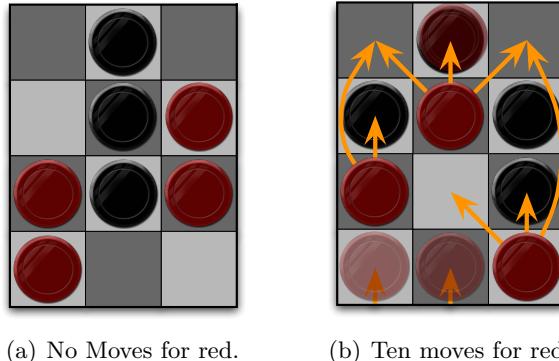


Figure 3.1: Board positions.

On other board sizes the average branching factor also seems pretty constant and independent of variations in the playing style. Table 3.2 displays measured branching factors for different board sizes and the highest amount pieces.

Breadth	Height	Max Pieces	Branching factor
2	2	2	1.6
3	4	4	3.2
3	4	5	3.2
3	4	6	3.2
4	5	5	4.5
4	5	6	5.0
5	6	6	6.0
9	9	15	12.3
9	9	30	12.3

Table 3.2: Average branching factor.

Effective branching factor

While it is not possible to change the branching factor, as it is game specific, it is possible to make changes to the algorithm that examines the game tree. These changes could allow the algorithm to discard some states before they have been examined. When this is done the **effective branching factor** that identifies that branching factor of the states that the algorithm has to expand to find the best move, becomes different from the average branching factor. By sorting out states that can not possible lead to the best possible move, the algorithm can use more time to look at states that might turn out to be the best move, and in that way decrease the efficient branching factor. A more detailed discussion on how to decrease the efficient branching factor is described in section 3.2.2.

3.1.3 Complete analysis of Kolibrat

While games on larger boards will takes years to solve it is easy to solve some of the games on smaller boards. In a game on a 2x2 board played to 1 point and with a maximum of 2 pieces on the board for each player black has a victory strategy, the proof is shown in figure 3.2.

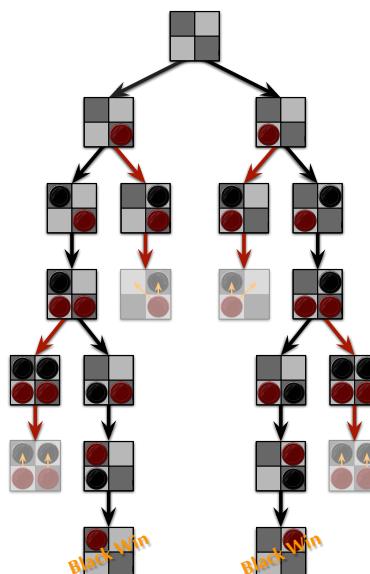


Figure 3.2: Victory strategy for black on a 2x2 board.

Even though figure 3.2 is not complete with the moves that lead to a red victory all possible moves that lead to a black victory is shown.

Victory strategy on a 3x3 board

As with the 2x2 board, it can also be proved that red has a victory strategy on a 3x3 board, if the game is won after the first goal. The incomplete game tree on figure 3.3 show only the moves that bring victory to red, but all possible black moves are shown.

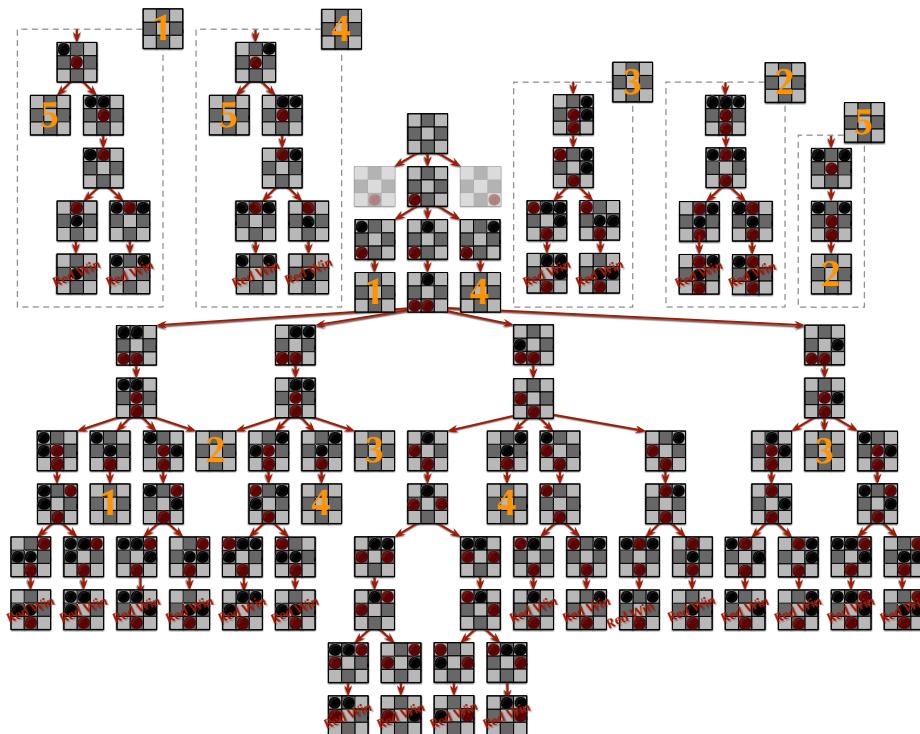


Figure 3.3: Victory strategy for red on a 3x3 board.

On this board size red has the advantage. Because red it is the first to move, he is also the first payer that can reach the centre of the board, which equals victory on a 3x3 board.

Victory strategy on other boards

As there is no rules in Kolibrat that allow games to end in a draw all kolibrat games must have a victory strategy for either red or black, unless the game is played in a way that makes the game go on forever. It might be that on some boards the players have the choice between playing forever or breaking the cycle and loose, but there is no proof of that. On a 3x4 game board played to one point red always wins. I have not proved that black can not win, but I have not been able to find a game that ends with a black victory when both players look more than a few moves ahead in the game, but played to four points black seem to have the advantage.

3.1.4 Analyzing forced loops

The question is now whether or not one of the players can force the game to go on forever. In order to do this there has to be a sequence of moves that the player can choose and no matter what moves the opponent chose the game must end in a state it has previously been in. A mathematical proof of whether or not this is possible, is out of scope for this thesis. But the solution can be found by a computer using brute force calculation. The pseudo code for testing this property is not included since the pseudo code for solving this problem exceeds 50 lines of code. The complete source code for the FORCEDLOOP program is listed in appendix C.5.1.

The program begins with constructing a game graph from the empty game board. All states that is found is added to a set of `knownStates`. Lets assume that the program tests whether or not red player can enforce a loop. When red is moving and one of the states that red can move to is in `knownStates` then all the child states are discarded and red's parent(s) is told that one of their children has a loop. Else the children is added to an `activeStates` array for further calculations.

When parent p , where red has the turn, is told one of its children has a loop, all children of that parent(s) are told they are part of a loop and they are discarded. Now p 's parent is told it has a child with a loop and p marks itself as part of a loop. When a parent where black has the turn is informed that a child has a loop it marks that child as dead. When all its children are dead it calls its own parent to tell that one of its children it is part of a loop, and marks itself as part of a loop.

When black have the turn and one of its children is in `knownStates` then the states that are in `knownStates` is told they have another parent (blacks state). All other child states that is not in `knownStates` are added to `activeStates` for further calculations.

In order to rule out the possibility of infinite loops the program must run the test for both red and black player. An interesting side effect of the `FORCEDLOOP` program is that with only a few modifications the program could be modified to search for victory strategies for one of the players. Instead of calling the parent when a child had a loop the states must call the parent when it knows that a child state is a victory node for red or black. Unfortunately the programs data structures is not the

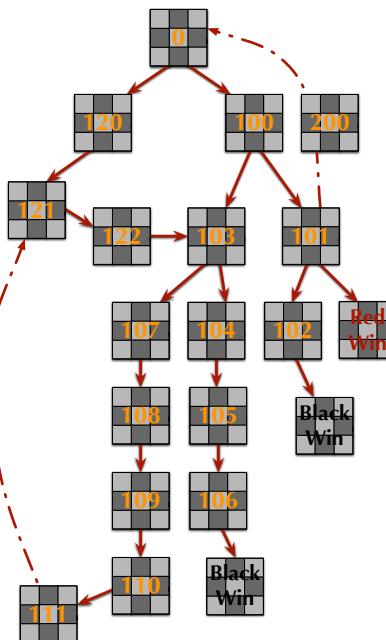


Figure 3.4: Artificial game tree from FAKELOGIC.

most memory efficient. The amount of ram the application uses quickly exceed several gigabytes. Although only smaller boards up to 3x3 has been tested with this program all tested boards have returned no forced loops.

To ensure that the FORCEDLOOP algorithm and its implementation works as intended, a class FAKELOGIC has been implemented and generates a game tree as the one on figure 3.4. When testing FORCEDLOOP on that game tree the program returns that red, but not black can enforce an infinite loop, which is the correct answer. This does not prove that there are no errors, but the game tree in figure 3.4 has many, if not all of the pitfalls, a real game tree will have. The numbers on figure 3.4 represent the internal ID numbers used in the FAKELOGIC implementation to recognize states, and the semi-dotted lines represent a backwards loop.

3.2 AI in Kolibrat games

If you define an **agent** as an artificial intelligent player that acts on behalf of a user, that isn't there. Then this agent must attempt to solve a given problem for that user. In this case the problem of winning in a game of Kolibrat. This problem is not exactly easy to solve, since the environment the agent operates in has multiple agents (one more) that works against it.

The environment the agents is operating in is fully observable, the agents have full access to all information in the current game state. The environment is also sequential and static since the players move one after another and the kind of information that is stored in a state is consistent from state to state.

When agents compete against each other in such an environment the agent usually uses knowledge of the game's rules to look ahead in the game, trying to find a state that ensures its victory. The following sections discuss the MINIMAX agent which works that way.

3.2.1 Mini-Max agent

The max-min agent is a highly specified agent developed especially for fully observable, static, sequential and multi agents environments. It is an old and well tested approach to solving two-player game problems.

It works by constructing a full game tree down to some level. From that level the game tree is evaluated from the bottom up. The moving

player is defined as the ***max*** player, since it is his move we want to maximize. The other player is the ***min*** player since he wants to minimize the **utility** of the moving players final move.

When the game tree is evaluated all states on the bottom level of the tree is given values by evaluation in an heuristic function, sometimes also called a utility function. This returns the utility value of each state on the bottom level. If the player one level above the bottom level is ***min*** he will choose the lowest utility value among all his children and take that value as his value. If it is ***max*** he will choose the highest utility value among all his children and take that value as his value. This continues level for level, until the top of the tree is reached, at that point the child that contains the highest utility value is selected as the best move.

Assuming that the heuristic function is perfect the MINIMAX agent will play perfectly, making no mistakes at all. Unfortunately the function is usually only a crude estimate of a states real utility value. Having a good utility function is essential for the MINIMAX agent, if the function is bad or even wrong the agent will perform badly compared to other agents with better utility functions, even if the agent with the bad utility function is given more time to look further ahead in the game.

The heuristic function

The definition of a **heuristic** function is a function that estimates the cost of the cheapest path from the current state to the a goal state [3, page 95]. Since guessing the distance to a goal state is highly dependent of the opponents playing style a utility function is used instead in multi-agent environments. The utility function basically does the same thing, but it do not return the expected length from the current state to the goal, but the states utility value. If the utility function is correct a state that is close to winning will have i higher value, than states further away from winning. However there is no guarantee of this, since most utility values is only estimates, based on certain properties of the current state.

3.2.2 Optimizing the Mini-Max agent

Because of the popularity of the MINIMAX agent, a lot of work has gone into optimizing the original algorithm. Most of these improvements are

trade-offs between memory usage and calculation time. Some possible optimizations is listed below.

Alpha-beta pruning

A simple yet efficient way to optimize the performance of MINIMAX is to implement ALPHA-BETA pruning. ALPHA-BETA pruning works by adding two variables to the each state, and only works if the MINIMAX agent is implemented to use **deep-first search**. The first represents the utility value of the best state the red player could have reached by taking another path in the game tree. The second the best utility value (the lowest) that black player could have reached by taking another path in the game tree.

If at any point one of the players reaches a state s that is evaluated to have a better utility value for that player, but is below a state where the other player can force the game into another part of the game tree that is preferable for him. If this happens the ALPHA-BETA enhancement realizes that the opponent will never allow the play to reach this part of the game tree and that entire part of the tree is abandoned.

In order to get the optimum effect of ALPHA-BETA pruning all moves must be sorted. The list of moves must be sorted so that states generated from moves expected to be good, is explored first. This ensures a high probability for finding the best move in the first try, and thus a bigger chance of reaching a state where ALPHA-BETA realizes that this part of the game tree can be cut-off.

An implementation of ALPHA-BETA pruning where the moves are sorted, will on average decrease the efficient branching factor to the square-root of the average branching factor [3, page 169].

Implementing a hash table

Another way to decrease the efficient branching factor is to ensure that two identical states is never both explored. This puts some demands on the MINIMAX implementation. First of all if two identical game states is discovered it must always be the one closest to the top of the tree, that is explored. This is not necessary the case since the MINIMAX agent uses **deep-first search**. Also if two equal states is found on the same level

only one of them should be explored.

Implementing a hash table to avoid exploring equal states can lead to a significant decrease of the efficient branching factor, but the trade-off is highly increased memory usage. In MINIMAX a state can be removed from memory when it has been evaluated, now all states must be preserved in memory until the best move has been found.

Multithreaded Programming

A completely different way of improving the performance of MINIMAX would be to give it more processing power. Since most new computers today ship with multiple processors, a serious boost in performance could be gained by taking advantage of all the processors. Implementing MINIMAX in a thread safe manner will complicate the implementation, but this is also the only significant disadvantage.

3.3 Implementing the Mini-Max agent

The MINIMAX agent is usually implemented as a deep first search, to decrease the memory requirements. To ensure the agent returns a move within reasonable time the search is normally cut-off at some predetermined level.

In this implementation, the agent is given a specific amount of time to find the best possible move. Since a best move can only be determined after a search to some level has been completed, the algorithm must complete at least one search in this time interval. To do this the agent uses **iterative deepening search**. This search continually performs deep first searches, at first the search is cut-off at level 1, then the cut-off level is increased by one until the time runs out. At that time the best move from the last completed search is returned as the best move.

It may seem like a waste of calculation time to recalculate the entire game tree for each level, but the advantages really surpass the disadvantages. The calculation overhead is determined by equation (3.5), where b is the

average branching factor, and d the search depth.

$$\frac{\sum_{i=0}^{d-1} b^i}{\sum_{i=0}^d b^i} = \frac{b^d - 1}{b^{d+1} - 1} \approx \frac{1}{b} \quad (3.5)$$

As seen the overhead is on the whole equal to a fraction of the branching factor. In a game on a 3x4 board this gives a overhead of 33%, which is still a considerable overhead, but considering the advantages it is still the best option if the calculation is time limited.

Since there is a rule in kolibrat that allows a player to move twice, some minor modifications had to be made to the original pseudo code for the Mini-Max agent work with Kolibrat, but essentially the implementation is equal to the original pseudo code taken from [3, page 170], but with an enhanced cut-off test and **iterative deepening search**. The HASH table enhancement has also been included.

The running time of the algorithm is determined by the time it takes to compute one state multiplied by the amount of states the algorithm searches. The time it takes to compute one state is *constant*, and when the algorithm has made a search down to level d it will have processed $\frac{b^{d+1}}{b-1}$ states. This gives a total running time of $O(b^d)$. Without any enhancements b is the average branching factor, but if the agent uses ALPHA-BETA pruning or other enhancements b represents the effective branching factor.

3.3.1 Additional possible Mini-Max enhancements

Even though the MINIMAX agent has been implemented with the enhancements specified above, there is still aspects of the algorithm that could be improved. For that reason some suggestions for additional enhancements are discussed in the sections below.

Randomness

In some situations, especially when the MINIMAX agent uses a simple heuristic function to evaluate the board, some or all of the possible moves end up with the same utility value. In this situation the agent always

chooses the first move among the moves with equal utility value. This makes the agent deterministic and might lead to board situations where the agent always makes the wrong move.

This behavior could be improved by choosing the moves at random, among the moves with the highest utility value. To make the agent completely non deterministic the agent could also use a probability function to choose its move. The utility value could be used as an indicator of how likely a move is to be chosen, meaning that the agent chooses a random move among all the possible moves, but moves with a higher utility value has a higher chance of being chosen.

Board specific agents

The heuristic function MINIMAX uses now is independent of the board. The heuristic will however not perform as well, on some boards since the optimal heuristic function varies from board to board.

One way to improve the heuristic function would be to make it board specific. This could for instance be telling the agent that standing in that board field is really good, or if the agent can force the game into this state it will win. The only problem is to find fields on the board that is good to occupy or states that leads to victory, but this could be calculated in the same way as the heuristic function is optimized in section [3.4](#).

Continuous search

Optimizing the search by improving the search algorithm in the MINIMAX agent is another way to increase the performance of the agent. This improvement would allow the agent to save the game tree in between moves, and save calculation time by not having to recalculate the entire tree the at every move, this could also be combined with **iterative deepening search** to decrease the 30% overhead.

A search algorithm like this will require more memory, and some performance will be lost to keeping track of the new advanced data structures required to implement this. If implemented correctly this enhancement will save time, but the decrease in calculation time is limited.

Assuming the game is played on a 3x4 board with an average branching factor b of 3.2, and the agent have constructed a game tree. When the agent has found his move $\frac{1}{b}$ of the tree can be neglected, and when the opponent has made his move another $\frac{1}{b}$ of the tree can be neglected. this gives that only $\frac{1}{b^2}$ of the entire tree is still useable when the agent regains the turn. With $b = 3.2$ this gives that only 9.8% of the tree don't have to be recalculated at the start of next turn, and on bigger boards this number is even smaller.

The real improvement is achieved by combining this search algorithm with the **iterative deepening search**, which will reduce the calculation overhead from a fraction of the branching factor to zero.

3.4 Optimizing the Heuristic function

The heuristic function is in many ways the most important component in the MINIMAX agent. If the heuristic is bad, or makes plain out wrong assumptions about the game states utility the agent will perform bad compared to agents with better heuristics, even if they have more time to analyze the game board.

There is different approaches to optimizing the heuristic function. One approach would be to use a **neural network** as the evaluation function, another to use a **weighted linear evaluation** function. The advantages and disadvantages of both approaches is described in the next section.

3.4.1 Neural network utility function

A neural network is a network of connected mathematical functions. The network takes a set of input parameters and returns a set of output parameters. The input could be a game state and the output could be the states utility value.

The disadvantage of using neural networks as a utility function is that after the neural network is implemented it must be trained to return a correct utility value. There is more than one algorithm that can train a neural network but they all have one thing in common. They take a neural

network, a set of states and their correct utility value as arguments. The algorithm now manipulates the network to provide the correct output to the input parameters. One problem with this approach is that generating a set of input states and finding the correct output (the utility value) can be difficult, since there are no simple way to determine the correct utility value for a state.

The advantages of using neural networks to evaluate a state is that the evaluation is based completely on computer generated information, about how good and less good states look like. Almost all other approaches to generating a heuristic function, involves some form of human reasoning about what is important and what is not.

3.4.2 Weighted linear evaluation function

A weighted linear evaluation function takes a set of functions $f_i(s)$, that each represent a property the evaluation function takes into account. If s is the state and w_i is the weight for $f_i(s)$ then a weighted linear evaluation function can be defined as in (3.6), where n is the number of properties from the state that the function takes into account.

$$\text{EVAL} = \sum_{i=1}^n w_i f_i(s) \quad (3.6)$$

This approach is often used to construct evaluation functions since it is simple and flexible. The only problem is to determine the functions and their weight, but there are methods for that.

Kolibrat is a zero-sum game, meaning that the sum of the players scores are always zero. When one player makes a move that increases his chance of winning, his opponents chances of winning is decreased. This increases the first players utility value, and decreases the opponents. The player closest to winning will always have a positive score, and his opponent will have a negative score, of the same value. This property makes it easy for the agents using the function to analyze how they are doing compared to their opponent.

3.4.3 Choosing the heuristic parameters

Choosing the best possible combination of parameters for the heuristic function is not possible for humans, as they can only make educated guesses about whether or not a parameter is important, and they might overlook important parameters. The only way of generating parameters for a heuristic function is by using algorithms that can transform problems specified in logic languages into a set of heuristic parameters.

The best known program to do this is *Armand E. Prieditis* program ABSOVLER. It takes a problem formulated in a logic language like **1. order logic** and returns a heuristic function for this problem.

Using the ABSOVLER to determine the best parameters for a evaluations function for Kolibrat would be the optimal solution, but it have some major drawbacks. In order to use the ABSOVLER it must first be implemented and this seems like a task worthy of its own thesis, judging by the available information about the ABSOVLER. Also according to [4] ABSOVLER has only been used to solve simple problems in static, single agent environments like the *Eight Puzzle*, *Rubik's cube* or the *eight queens problem* so hoping it can discover a heuristics for a game like Kolibrat somehow seen optimistic.

Chosen parameters

given the circumstances the ABSOVLER seems like an unrealistic approach to finding the parameters for the heuristic evaluation function. The alternative approach is to lets humans choose the parameters for the heuristics and then determine their importance by other means. The most important property of the parameter for the heuristic is that they somehow increases when the player is getting closer to winning, and decreases when the player is getting closer to loosing. After analyzing the game, I have come up with the following possible parameters to include in a heuristic function.

1. Value of a piece on the line of insertion.
2. The value increment when a piece gets one field closer to the opponents home line.
3. The added value when a piece is in the middle of the board.

4. The penalty for having a piece standing in front of the opponent in his turn.
5. The added value to have two pieces standing in a row.
6. An added value for the number of legal moves the player can make.
7. Value of a scored point.
8. The value of having the turn.
9. The value of being able to insert pieces on the board.
10. The value of having a piece standing on the opponents home line.
11. The value of being the player having most pieces on the board.

3.4.4 Determining the parameters weight

To use these parameters in an weighted linear evaluation function, the weight of each parameter must first be determined. There are a few different methods that is suitable for determining the value of the different parameters. The most known of these are **genetic algorithms** and **simulated annealing**, both of these will be described below.

Genetic algorithms

Genetic algorithms works by generating an initial population of heuristic evaluation functions. The functions is now ranked from the most fit to the least fit individual, by using the functions in actual game play. The best heuristic functions are then chosen as parents for the next generation of heuristic evaluation functions. This continues until there hasn't been considerable improvements over the last few generations.

To generate a new generation of heuristic evaluation functions the parents are put through a mutation process, where some of its weight values are changed at random. After that the parents are mated with each other to produce a new generation. This works by randomly swapping weight values from the parents onto a new heuristic evaluation functions that is part of the next generation. The swapping are done in a way that ensures that strongly related weight values values almost always come from the

same parent.

While the genetic algorithm certainly will be able to find optimal values for the different parameters weights, is might not be the best approach. The genetic algorithms biggest strength comes from its intelligent crossings between related and unrelated properties in the heuristic evaluation function. But in Kolibrat there is no clear way to determine which properties is strongly related, and which is not.

Simulated annealing

In many ways simulated annealing works in the same way as the genetic algorithm, but when a new generation is made only the mutation process is carried out. Also all the parameters is mutated at once, not just a selected few.

This algorithm has the advantage compared to genetic algorithms that it requires no knowledge about the internal relations between the evaluation functions parameters and their weights. Aside from this both algorithms should be able to reach the same result, but maybe not in the same amount of time.

3.5 Simulated annealing Implementation

Due to its lack of pre-required knowledge of the internal relations between the evaluation functions parameters and their weight, the algorithm for simulated annealing will be used to determine the weights of the heuristic functions parameters.

In the original pseudo code [3, page 116] the simulated annealing algorithm only kept one heuristic function in its population until a new function defeated it and took its place. In this implementation the algorithm has been changed to keep 60 heuristic functions, then choose the 30 best, mutate them and add both the mutants and the non-mutated 30 functions to the next population. The altered pseudo code is shown in algorithm 1, and the source code can be seen in appendix C.4.1.

The function RANDOMHEURISTICVALUES generates random weights between 0 and 1000 for all the parameters the algorithm it is trying to

Algorithm 1 SIMANNEALING

```

1: rand  $\leftarrow$  1.0
2: for i = 0 to 59 do
3:   heuristicsArray[i]  $\leftarrow$  RANDOMHEURISTICVALUES()
4: end for
5: winners  $\leftarrow$  0
6: count  $\leftarrow$  59
7: while rand  $\geq$  0 do
8:   i  $\leftarrow$  RAND(0, count)
9:   player1  $\leftarrow$  heuristicsArray[i]
10:  remove player1 from heuristicsArray
11:  count  $-$ 
12:  i  $\leftarrow$  RAND(0, count)
13:  player2  $\leftarrow$  heuristicsArray[i]
14:  remove player2 from heuristicsArray
15:  victoryArray  $\leftarrow$  FINDWINNER(player1, player2)
16:  winners ++
17:  if winners = 30 then
18:    for all h heuristics in victoryArray do
19:      new  $\leftarrow$  NEWHEURISTICFROM(h, rand)
20:      insert new somewhere in heuristicsArray[]
21:      insert h somewhere in heuristicsArray[]
22:    end for
23:    remove all entries in victoryArray
24:    rand  $\leftarrow$  rand - 0.02
25:    winners  $\leftarrow$  0
26:    count  $\leftarrow$  59
27:  end if
28: end while
29: return heuristicsArray

```

optimize. The RAND function takes two numbers as arguments and generates a random number in between these. FINDWINNER takes two heuristics and loads these into two players. The two players then battle each other to find the best heuristic. If the game is not over in four minutes the FINDWINNER terminates the game, and chooses a random winner. The NEWHEURISTICFROM function takes a heuristic h and the current random factor $rand$ as inputs and generates a new heuristic. The pseudo code for the NEWHEURISTICFROM is showed in algorithm 2.

Algorithm 2 NEWHEURISTICFROM

```

1: input:  $h$ 
2: input:  $rand$ 
3:  $new \leftarrow$  copy of  $h$ 
4: for all weights  $w$  in  $new$  do
5:    $w = w + \text{RAND}(-500, 500) \cdot rand$ 
6: end for
7:  $minval \leftarrow 0$ 
8: for all weights  $w$  in  $new$  do
9:    $minval \leftarrow \text{MIN}(w, minval)$ 
10: end for
11: if  $minval < 0$  then
12:   for all weights  $w$  in  $new$  do
13:      $w = w + minval$ 
14:   end for
15: end if
16:  $maxval \leftarrow 0$ 
17: for all weights  $w$  in  $new$  do
18:    $maxval \leftarrow \text{MAX}(w, maxval)$ 
19: end for
20: if  $maxval > 1000$  then
21:   for all weights  $w$  in  $new$  do
22:      $w = w \cdot 1000/maxval$ 
23:   end for
24: end if
25: return  $new$ 
```

Running the SIMANNEALING algorithm on a fast computer takes about 24 hours, when the FINDWINNER function terminates after the first point is scored on a 3x4 game board with a maximum of 3 pieces for each player.

3.5.1 Simulated annealing results

After running the SIMANNEALING algorithm it turns out that optimizing on many parameters at the same time, requires that some problematic factors must be taken into account. The problem is that with many parameters where some of them is more insignificant than others, the chance of generating a great heuristic are smaller compared to running the algorithm on only the 3 most significant parameters. Because of this the values never stabilize and the algorithm returns random results. This can be solved by lowering the rate by which the *rand* value is decreased, since this gives the values more time to stabilize close to the optimal weight values.

Another problem is that when two bad heuristics play each other, it can happen that they both play so bad that none of them can win in the given timeframe, and the winner is chosen at random. If this happens to often it adds noise to the results. If the noise level is to high the good heuristics is never discovered before *rand* reaches zero. This problem can also be rescued by lowering the rate by which the *rand* value is decreased.

Yet another pitfall is that if the players is given to much time to think the better heuristics never finishes the game, but play on until FINDWINNER terminates the game. To avoid this the time each player have to move, should lie in the interval from about 0.8 seconds to 1.3 seconds.

It also turns out that at some point the heuristics reaches a point where they get so good that other factors like who is the starting player, becomes more important. When this point is reached the winner is always the same player. At this point it is no longer the best heuristic that that wins, but the player that have the advantage or disadvantage of starting. From then on the SIMANNEALING algorithm can no longer improve the heuristic functions since the FINDWINNER function now returns random results. So there is no best heuristic only a set of heuristics that lie in the best interval of weight values.

When the SIMANNEALING algorithm is done all data is saved to a file. After running the algorithm with the following properties, the properties weights has been found and can be seen in table 3.3.

prop-1 Value of a piece on the bottom line.

prop-2 Value increase for a piece per level.

prop-3 Bonus for standing in the middle of the board.

prop-4 Penalty for standing in front of the enemy, when he has the turn

prop-5 Bonus for having 2 pieces on a row.

prop-6 The value of a legal move.

prop-7 The value of a goal.

prop-8 The value of having the turn.

prop-9 The value of being able to insert pieces on the board.

prop-10 The value of having a piece standing on the opponents home line.

prop-11 The value of being the player having most pieces on the board.

As expected table 3.3 shows that the most important property is having points. But the resulting distribution is interesting. Standing in front of the enemy in his turn is almost as bad as loosing a point, and the value of being able to insert a piece is the third most important property. The value of having a piece on the opponents home line seems surprisingly small, but this could be explained with the fact that the value of a piece standing there already is worth $prop1 + 3 \cdot prop2 = 1370$, so the total value of a piece standing at the opponents home line is 1400.

As noted earlier the weight of the heuristics parameters is board specific. This is confirmed in table 3.3 where the results of the 3x3 board with 4 pieces, and the 4x6 with 6 pieces is compared. Not surprisingly the value of property 2 has decreased and so has property 3. the same goes for property 9 that also seems to be less important on bigger boards.

Calculation deviations

The output from the SIMANNEALING algorithm variates depending on the time the players were given to calculate their move and depending on the board size, but even when the algorithm is executed with the same

property	weight on 3x4	weight on 4x6
1	500	580
2	260	170
3	520	730
4	750	960
5	520	410
6	330	160
7	1000	1000
8	90	240
9	530	220
10	30	220
11	170	0

Table 3.3: Heuristic weight values

settings there is still some variation.

Provided that the output of SIMANNEALING is not random due to decreasing the *rand* variable to fast, or by some other means, the final output of the algorithm is usually within ± 150 from the average output with the same parameters.

3.6 Heuristics Comparison

After the development and implementation of the MINIMAX algorithm, and several heuristics to evaluate the game board, the time has come to compare the efficiency of these heuristic functions. The following four heuristics will be tested against each other.

Basic AI the first AI developed, it only looks on the location of the pieces and the number of scored points to evaluate the board.

Simple AI A little more advanced than the **Basic AI**. It is improved by also trying to avoid standing in front of the enemy, and add a bonus for having pieces in a row.

Advanced AI This is the most advanced AI developed before the Simu-

lated Annealing AI. It uses almost all the same parameters, but the weight function is simply chosen by trial and error, by humans.

SimAnnealing AI The AI using the weights returned by the SIMANNEALING algorithm.

The exact heuristics for each AI can be seen in table 3.4.

Property	Basic AI	Simple AI	Advanced AI	SimAnnealing AI
1	0	1	1	50
2	1	2	2	26
3	0	1	1	52
4	0	2	2	57
5	0	1	1	52
6	0	0	0	33
7	4	10	10	100
8	0	0	2	9
9	0	0	2	53
10	0	0	1	3
11	0	0	1	17

Table 3.4: AI heuristic weight values

The results in table 3.5 shows that on equal themes the SIMANNEALING AI the the best, followed by the ADVANCED AI, SIMPLE AI and finely the BASIC AI which lost all its matches.

The ADVANCED AI, SIMPLE AI and BASIC AI is close to each other in strength, compared to the SIMANNEALING AI as seen in table 3.5. The BASIC AI only needed 5 times as much time as its opponent to beat the SIMPLE AI, and 10 times as much to beat the ADVANCED AI. But even with 60 times as much time as the SIMANNEALING AI it still could not win. None of the other AI's have defeated the SIMANNEALING AI, only BASIC AI and ADVANCED AI managed to avoid loosing by going into an infinite loop.

While all of the AI's play intelligent, and well beyond human beginner level, there is a clear difference in their playing style. The BASIC AI and SIMPLE AI both play well, but most of the time their moves are predictable. ADVANCED AI is superior to the first two and is less predictable

Red player	Black player	Time [s]	Winner (Score)
Basic AI	Basic AI	Red: 2, Black: 2	Black (5, 3)
Basic AI	Simple AI	Red: 2, Black: 2	Black (5, 3)
Basic AI	Advanced AI	Red: 2, Black: 2	Black (5, 3)
Basic AI	SimAnnealing AI	Red: 2, Black: 2	Black (5, 2)
Basic AI	Simple AI	Red: 5, Black: 2	∞
Basic AI	Advanced AI	Red: 5, Black: 2	∞
Basic AI	SimAnnealing AI	Red: 5, Black: 2	Black (5, 4)
Basic AI	Simple AI	Red: 10, Black: 2	Red (5, 1)
Basic AI	Advanced AI	Red: 10, Black: 2	∞
Basic AI	Advanced AI	Red: 10, Black: 1	Red (5, 2)
Basic AI	SimAnnealing AI	Red: 10, Black: 1	Black (5, 4)
Basic AI	SimAnnealing AI	Red: 20, Black: 1	Black (5, 3)
Basic AI	SimAnnealing AI	Red: 30, Black: 1	Black (5, 2)
Basic AI	SimAnnealing AI	Red: 30, Black: 0.5	∞
Simple AI	Advanced AI	Red: 2, Black: 2	Black (5, 4)
Simple AI	SimAnnealing AI	Red: 2, Black: 2	Black (5, 1)
Simple AI	Advanced AI	Red: 5, Black: 2	Red (5, 4)
Simple AI	SimAnnealing AI	Red: 5, Black: 2	Black (5, 3)
Simple AI	SimAnnealing AI	Red: 20, Black: 2	Black (5, 2)
Simple AI	SimAnnealing AI	Red: 20, Black: 1	Black (5, 3)
Simple AI	SimAnnealing AI	Red: 20, Black: 0.5	Black (5, 2)
Advanced AI	SimAnnealing AI	Red: 2, Black: 2	Black (5, 1)
Advanced AI	SimAnnealing AI	Red: 5, Black: 2	Black (5, 2)
Advanced AI	SimAnnealing AI	Red: 20, Black: 2	Black (5, 2)
Advanced AI	SimAnnealing AI	Red: 20, Black: 1	Black (5, 3)
Advanced AI	SimAnnealing AI	Red: 20, Black: 0.5	∞
SimAnnealing AI	SimAnnealing AI	Red: 2, Black: 20	Black (5, 2)
SimAnnealing AI	SimAnnealing AI	Red: 20, Black: 2	Red (5, 4)

Table 3.5: Victory table for different heuristics.

in its moves, it seems to have a deeper understanding of the game and is capable of making moves, that at first glance is bad but then turns out to be a trap for the other player. The SIMANNEALING AI however plays like it has thought the entire game through, from the beginning. Without knowing it, it is capable of laying traps and sacrificing pieces, only to gain an advantage later in the game.

Chapter 4

Conclusions

After having implemented Kolibrat in *Objective-C* the first thought that springs into mind is that memory management in Java is a bliss. Besides from this major drawback *Objective-C* is a pleasant language to work with.

In the development process the UML diagrams have been a great help. I am especially satisfied with the final structure of Kolibrat itself, its structure is stable and even though I have made some last minute changes to its core components, it only took short amount of time before the code could compile again.

Kolibrat itself satisfies all the requirements mentioned in section 1.2 about the aims of this thesis, and although there were a few difficulties getting the SIMANNEALING algorithm to work properly, the results are impressive in my opinion.

If there had been any more time available it would have been interesting to develop an agent that used another algorithm than MINIMAX, but the additional work on optimizing the heuristic function kind of makes up for this. It would have been interesting though to solve Kolibrat on the 3x4 board and see how some of the other AI's performs against it. Also matching the SIMANNEALING algorithm against a trained neural network would have been interesting, but time didn't allow it.

If I had a chance to redo some of the decisions made in the development, there is a few things that I would properly reconsider. First of all when

running the SIMANNEALING algorithm I would match the heuristics up against another heuristic like the ADVANCED AI and then choose the 30 best heuristics, by looking at the number of moves that they needed to beat their opponent, since I believe this would lead to better results than the current implementation. Also after realizing the time it takes to run the SIMANNEALING algorithm, version 2.0 will definitely have a resume calculation option.

Also the development of agents is difficult and error-prone, since it is hard to determine if its algorithm actually returns the best move, or if there is an implementation error. If any more agents were to be developed I would therefore also implement another FAKELOGIC object to test it with.

4.1 Future prospects

One of the hardest things to do in this project has been to limit the focus areas of this project. There have been many things that would have been interesting to attempt or look into. Some of the things that could be added to the Game in the future includes the following:

- Checks to ensure that the game can will not allow the user to start a game on a board that will make the game window bigger than the computers screen.
- Adding sound effects to the game.
- Adding the ability for the player to play against others over a network. Eventually implemented as i special player class.
- Some mechanism to download new AI's over the internet.
- Give players the possibility of saving the game.
- Add the ability to undo or redo an unlimited number of moves.
- Making a Unix or Linux version either with a text interface or a new GUI made for linux.
- Adding a fullscreen 3D interface using openGL.

Appendix A

Kolibrat Rulebook

Kolibrat is a board game usually played on a 3x4 game board. The game involves two players, a red and a black player. Each player have a home line on the game board, red the button line and black the top line. The board and the home lines is showed in figure A.1.

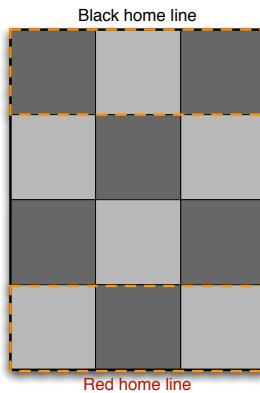


Figure A.1: Empty game board.

Each player can insert pieces on his home line and move them forward on the board.

A.1 Game Objectives

A player wins the game by moving pieces from his home line forward and onto the opponents home line. When a piece reaches the opponents home line the piece is safe and cannot be taken. A piece on the opponents home line can be removed from the board and exchanged with a point. When a player reaches five points he has won the game.

The game has one other losing condition, if one of the players make a move that bring the game into a state, where none of the players can move, the player making the last move has lost.

A.2 Rules for movement

The rules for movement is described below, if a player can make a move he is forced to do so, he has no way to skip his turn.

Insetting a piece

When one of the players have the turn, and they got less than four pieces on the game board, they can choose to insert a piece on the gameboard anywhere on the row they owe, if the field is empty. This counts as a move and the other player gets the turn.

Moving forward

When a piece has been inserted into the board it can move forward at an angle, but not straight forward. as seen in figure [A.2](#).

Attacking the opponent

When two pieces stand in front of each other the player moving can move forward at an angle as always, but now he can also choose to take the other piece, to take its place and remove it from the board, or he can jump over the piece and land behind it as seen in figure [A.3](#).

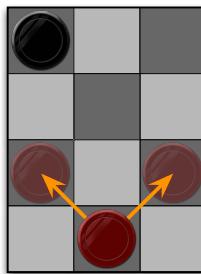


Figure A.2: Pieces can't move strait forward.

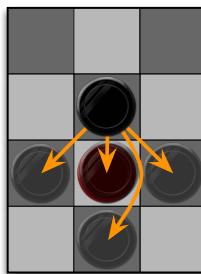


Figure A.3: Pieces can attack or jump over other pieces.

Jumping above multiple pieces

It is possible to jump above more than one of the opponents pieces at the same time, meaning that if you stand in front of two of your opponents pieces you can choose to attack to first or you can jump above both pieces. It is not possible to attack the last piece. On larger boards it is possible to jump over more than two of the opponents pieces, it is not possible to jump off the board however, there must be an empty space behind the opponents pieces for your piece to land on. See figure A.4 for details.

Gaining Points

When a piece reaches the row owned by the opponent if can not be taken by the opponent, and the opponent can not insert a piece at that coordinate. A piece standing at the opponents row can be removed and exchanged for a point by the player owning the piece, this counts as a move and the opponent is given the turn.

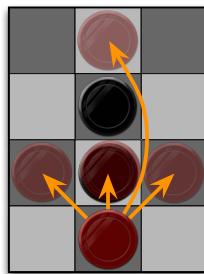


Figure A.4: Pieces can jump over multiple other pieces.

Multiple turns

In certain board positions one of the players can come in a situation where he can not make a move, in that case the player forfeits his turn and the other player thus gains two turns in a row, if the opponent still can't move he gains a third turn in a row and so on until the other player can move again.

Appendix B

Tests Details

This section contains the details of the tests performed on the gameLogic object to ensure it confirms to the Kolibrat rule-book. Unless anything else is specified in the test the game is played on a 3x4 game board and, the maximum of pieces is 4 and the game ends when the fifth point is gained by one of the players.

Test 1

This test tries to insert a red piece in (1, 0) on an empty game board, when red has the turn. In the test it checks that the move is considered legal, that red now has one piece on the board, that black is the next player to move and finally that red actually now have a piece in (1, 0).

Test 2

This test tries to insert a black piece in (1, 3) on an empty game board, when black has the turn. In this test it checks that the move is considered legal, that black now has one piece on the board, that red is the next player to move and finally that black actually now have a piece in (1, 3).

Test 3

This test tries to do an illegal move by attempting to inset a red piece on the board in the empty field (1,3) on a board with 4 red pieces already on it. The test checks that move is rejected by the game engine, that there is still 4 red pieces and 0 black pieces on the board. It also checks if red is still the moving player and checks that the field (1, 3) is still empty.

Test 4

This test tries to do an illegal move by attempting to inset a red piece in (2, 2) on an empty board. The test ensures that the move is rejected, that the number of pieces on the board has not changed and that red player still is the player with the turn. Finely the board is checked to ensure that (2, 2) is still empty.

Test 5

This test attempts to score a point for red. The board is empty except for one red piece in (1, 3) and red has the turn. The piece in (1, 3) is attempted removed from the board in order to score a point. The test validates that the move is not rejected by the GameLogic, that red now have zero pieces on the game board, that black player has the turn and that red player now have one point.

Test 6

This test attempts to score a point for black. The board is empty except for one black piece in (0, 1) and black has the turn. The piece in (0,1) is attempted removed from the board in order to score a point. The test validates that the move is not rejected by the GameLogic, that black now have zero pieces on the game board, that red player has the turn and that black player now have one point.

Test 7

This test attempts to insert a red piece on (1, 0), where (1, 0) is already occupied by black, aside from that the board is empty, and red have the turn. the test validates if the move is declared illegal, that black have 1 piece on the board and red 0. It also checks that black still got a piece in (1,0) and that both players still got 0 points.

Test 8

This test validates that the game status is updated when red gains a fifth point and thereby wins the game. The game-board is empty except for one red piece in (1, 3). Red have the turn, and 4 points. The tests validates that the removal of the piece is considered legal, that red and

black now both have zero pieces on the board, that (1,3) is now empty that red now have five points, that the gamestatus is over and that red is declared the winner.

Test 9

This test validates that no player can move when the game status is set to game over. The test is performed on a gameboard where red has a piece in (0,0) and black one in (2,3) red got 5 points and is the winner. Red is the next player to move. The test ensures that the move from (0,0) to (1,1) is rejected by GameLogic then is changes next player to black and validates that black is rejected to move from (2,3) to (1,2).

Test 10

This test performs two moves that would have been legal if it would have been possible to move outside of the board. The test validates that red cant move from (0,0) to (-1,1) and that black cant move from (2,3) to (1,4).

Test 11 to 20

This set of moves makes tests no non empty game boards. Test 11 to 15 tests for errors in the moving rules for red, and test 16 to 20 tests the same rules for black. Each test ensures that GameLogic returns the current amount of legal moves, and that all the moves returned match the moves specified al legal in the Kolibrat rule book. The actual tests can be seen on figure [B.1](#).

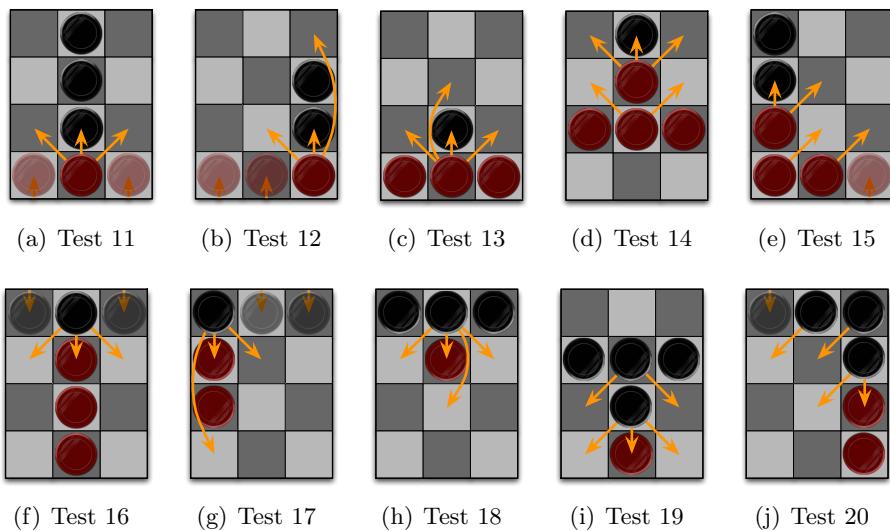


Figure B.1: Boards for test 11 to 20

Appendix C

Source Code

C.1 Kolibrat Source Code

C.1.1 HumanPlayer.h

```
1 //  Kolibrat
2 //  HumanPlayer.h
3 //
4 //  Created by Aron Lindberg.
5
6 #import <Cocoa/Cocoa.h>
7 #import "Datastructures.h"
8 #import "PlayerProtocol.h"
9 #import "GameEngine.h"
10
11 // Confirms to the Player Protocol.
12 @interface HumanPlayer : NSObject < Player_Protocol >
13 {
14 // Private instance variabels.
15 @private
16     GameEngine *engine;
17     int playerID;
18     BOOL waitingForOtherPlayer;
19     BoardField firstClick;
20     BoardField secondClick;
21     NSString *name;
22 }
23 // Class Methods.
24 + (NSString *)playerType;
25
26 // Public Instance Methods.
27 - (void)mouseClickNotification:(NSNotification *)notification;
28 - (id)initAsPlayer:(int)player withName:(NSString *)playerName
    boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(int)maxGoals;
```

```
29 - (void)setGameEngine:(id)ge;
30 - (void)reset;
31 - (NSString *)playerName;
32 - (void)startNewTurn;
33 @end
```

C.1.2 HumanPlayer.m

```
1 // Kolibrat
2 // HumanPlayer.m
3 //
4 // Created by Aron Lindberg.
5
6 #import "HumanPlayer.h"
7
8 @implementation HumanPlayer
9
10 - (void)dealloc
11 {
12     [name release];
13     [[NSNotificationCenter defaultCenter] removeObserver:self];
14     [super dealloc];
15 }
16
17 - (id)initAsPlayer:(int)player withName:(NSString *)playerName
18     boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(int)maxGoals
19 {
20     self = [super init];
21     if (self != nil)
22     {
23         NSNotificationCenter *mainCenter = [NSNotificationCenter
24                                         defaultCenter];
25
26         [mainCenter addObserver:self
27                         selector:@selector(changeOfTurnNotification
28                                         :)
29                         name:@"ChangeOfTurn"
30                         object:nil];
31
32         [mainCenter addObserver:self
33                         selector:@selector(mouseClickNotification:)
34                         name:@"MouseClicked"
35                         object:nil];
36
37         playerID = player;
38         waitingForOtherPlayer = TRUE;
39         firstClick = NIL_FIELD;
40         secondClick = NIL_FIELD;
41
42         name = [NSString stringWithFormat:playerName];
43         [name retain];
44     }
45 }
```

```
41     return self;
42 }
43
44 // This method is called everytime the GUI sends a MouseClick
notifikation.
45 - (void)mouseClickNotification:(NSNotification *)notification
46 {
47     if( waitingForOtherPlayer == TRUE )
48         return;
49
50     if( BOARDFIELD_EQUALS_NIL( firstClick ) )
51     {
52         firstClick = [[notification object] retriveField];
53         [engine SelectedPiece:firstClick fromPlayer:self];
54         return;
55     }
56
57     secondClick = [[notification object] retriveField];
58
59     if([engine playerMove:makeMove( firstClick, secondClick )
60         fromPlayer:self])
61     {
62         // The move was legal.
63         firstClick = NIL_FIELD;
64         secondClick = NIL_FIELD;
65         waitingForOtherPlayer = TRUE;
66     }
67     else
68     {
69         // The move was not legal.
70         firstClick = secondClick;
71         [engine SelectedPiece:firstClick fromPlayer:self];
72     }
73 }
74
75 // sets the game engine to retrive data from.
76 - (void)setGameEngine:(id)ge
77 {
78     engine = ge;
79 }
80
81 // This method is called when a changeOfTurn Notification is sent.
82 - (void)startNewTurn
83 {
84     waitingForOtherPlayer = FALSE;
85 }
86
87 // Resets the player, called when the human chooses "reset game"
from the menu.
88 - (void)reset
89 {
90     firstClick = NIL_FIELD;
91     secondClick = NIL_FIELD;
92 }
93
94 // Returns the name of the player.
```

```

93 - (NSString *)playerName
94 {
95     return name;
96 }
97
98 // Returns the name of the playerType.
99 + (NSString *)playerType
100 {
101     return @"Human Player";
102 }
103 @end

```

C.1.3 Datastructures.h

```

1 // Kolibrat
2 // Datastructures.h
3 //
4 // Created by Aron Lindberg.
5
6 #import <Cocoa/Cocoa.h>
7
8 #define NIL_MOVE makeMove( makeBoardField( -1, -1 ), makeBoardField
    ( -1, -1 ) )
9 #define NIL_FIELD (makeBoardField( -1, -1 ))
10
11 #define BOARDFIELD_NOT_NIL( bf ) ( bf.x != -1 || bf.y != -1 )
12 #define BOARDFIELD_EQUALS_NIL( bf ) ( bf.x == -1 && bf.y == -1 )
13 #define BOARDFIELDS_IS_EQUAL( bf1, bf2 ) ( bf1.x == bf2.x && bf1.y
    == bf2.y )
14
15 #define GAME_RUNNING( gameStatus ) (gameStatus.gameOver == FALSE)
16 #define GAME_NOT_RUNNING( gameStatus ) (gameStatus.gameOver == TRUE
    )
17
18 #define RUNNING makeGameStatus( FALSE, FALSE )
19 #define RED_WON makeGameStatus( TRUE, PLAYER_RED )
20 #define BLACK_WON makeGameStatus( TRUE, PLAYER_BLACK )
21
22 #define BOARD_CONVERTER( bfc ) (bfc).occupiedByRed == TRUE && (bfc)
    .occupiedByBlack == FALSE ? PLAYER_RED : ((bfc).occupiedByRed
    == FALSE && (bfc).occupiedByBlack == TRUE ? PLAYER_BLACK :
    EMPTY)
23 #define EMPTY_FIELD( bfc ) (bfc).occupiedByRed == FALSE && (bfc)
    .occupiedByBlack == FALSE
24 #define RED_FIELD(bfc) (bfc).occupiedByRed == TRUE && (bfc)
    .occupiedByBlack == FALSE
25 #define BLACK_FIELD( bfc ) (bfc).occupiedByRed == FALSE && (bfc)
    .occupiedByBlack == TRUE
26 #define NOT_BLACK_FIELD( bfc ) (bfc).occupiedByBlack == FALSE
27 #define NOT_RED_FIELD( bfc ) (bfc).occupiedByRed == FALSE
28
29 #define PLAYER_RED TRUE

```

```
30 #define PLAYER_BLACK FALSE
31
32 enum { // Constants related to drawing the game board.
33     BLACK = PLAYER_BLACK,
34     RED = PLAYER_RED,
35     EMPTY = 3 ,
36     HIGHLIGHTFIELD = 4 ,
37     BLACK_HIGHLIGHT = 5 ,
38     RED_HIGHLIGHT = 6 ,
39     BLACK_OPAQUE = 7 ,
40     RED_OPAQUE = 8 ,
41 };
42
43 // Some handy Typedefinitions.
44 typedef struct gameStatusStruct {
45     BOOL gameOver;
46     BOOL winner;
47 } GameStatus;
48
49 typedef struct boardFieldStruct {
50     short int x;
51     short int y;
52 } BoardField;
53
54 typedef struct boardMoveStruct {
55     BoardField from;
56     BoardField to;
57 } BoardMove;
58
59 typedef struct gameScoreStruct {
60     short int red;
61     short int black;
62 } GameScore;
63
64 typedef struct boardSizeStruct {
65     int height;
66     int width;
67 } BoardSize;
68
69 typedef struct boardFieldContentStruct {
70     bool occupiedByRed;
71     bool occupiedByBlack;
72 } BoardFieldContent;
73
74 typedef struct gameStateStruct {
75     BoardFieldContent **board;
76     GameScore score;
77     BoardMove lastMove;
78     GameStatus gameStatus;
79     BOOL playerMoving;
80     unsigned short int redPicesOnBoard;
81     unsigned short int blackPicesOnBoard;
82     BoardSize boardSize;
83 } GameState;
84
```

```
85 typedef struct simpleList {
86     int ellementsInList;
87     struct simpleListEllement* head;
88     struct simpleListEllement* tail;
89 } SimpleList;
90
91 typedef struct simpleListEllement {
92     struct boardMoveStruct moveData;
93     struct simpleListEllement* next;
94 } simpleListEllement;
95
96 // Some plain C methods to make instances of the custom
97 // type definitions.
97 BoardField makeBoardField( int x, int y );
98 GameScore makeGameScore( int red, int black );
99 BoardSize makeBoardSize( int height, int width );
100 BoardMove makeMove( BoardField from, BoardField to );
101 BoardMove makeMoveFromInt( int fromx, int fromy, int tox, int toy )
102 ;
102 GameState makeGameState( BoardSize boardSize );
103 GameStatus makeGameStatus( BOOL gameOver, BOOL winner );
104 SimpleList makeSimpleList();
105
106 BoardFieldContent makeBoardFieldWithContent( bool takenByRed, bool
107     takenByBlack );
107 BoardFieldContent makeBlackField();
108 BoardFieldContent makeRedField();
109 BoardFieldContent makeEmptyField();
110
111 // Some plain C methods to work with SimpleLists.
112 void concatSimpleLists(SimpleList *a, SimpleList *b);
113 void addEllementToSimpleList(SimpleList *list, BoardMove *data);
114 void removeHeadFromSimpleList(SimpleList *list );
115 void freeSimpleList(SimpleList *list );
116
117 // Header for BoardFieldObject, this is basically an object wrapper
118 // for the BoardField structure.
118 @interface BoardFieldObject : NSObject
119 {
120     @private
121     BoardField board;
122 }
123 + (id)boardfieldObjectWithField:(BoardField)bf;
124 - (id)initWithField:(BoardField)bf;
125 - (BoardField)retriveField;
126 @end
127
128 // Header for MoveObject, this is basically an object wrapper for
128 // the move structure.
129 @interface MoveObject : NSObject
130 {
131     @private
132     BoardMove m;
133 }
134 + (id)moveObjectWithMove:(BoardMove)theMove;
```

```

135 - (id)initWithMove:(BoardMove)theMove;
136 - (BoardMove)retriveMove;
137 - (BOOL)isEqual:(id)anObject;
138 - (unsigned)hash;
139 @end

```

C.1.4 Datastructures.m

```

1 // Kolibrat
2 // Datastructures.m
3 //
4 // Created by Aron Lindberg.
5
6 #import "Datastructures.h"
7
8 BoardFieldContent makeBoardFieldWithContent( bool takenByRed, bool
9     takenByBlack )
10 {
11     struct boardFieldContentStruct temp;
12     temp.occupiedByRed = takenByRed;
13     temp.occupiedByBlack = takenByBlack;
14     return temp;
15 }
16
17 BoardFieldContent makeBlackField()
18 {
19     struct boardFieldContentStruct temp;
20     temp.occupiedByRed = FALSE;
21     temp.occupiedByBlack = TRUE;
22     return temp;
23 }
24
25 BoardFieldContent makeRedField()
26 {
27     struct boardFieldContentStruct temp;
28     temp.occupiedByRed = TRUE;
29     temp.occupiedByBlack = FALSE;
30     return temp;
31 }
32
33 BoardFieldContent makeEmptykField()
34 {
35     struct boardFieldContentStruct temp;
36     temp.occupiedByRed = FALSE;
37     temp.occupiedByBlack = FALSE;
38     return temp;
39 }
40
41 GameStatus makeGameStatus( BOOL gameOver, BOOL winner )
42 {
43     struct gameStatusStruct temp;
44     temp.gameOver = gameOver;

```

```
44     temp.winner = winner;
45     return temp;
46 }
47
48 BoardField makeBoardField( int x, int y )
49 {
50     struct boardFieldStruct temp;
51     temp.x = x;
52     temp.y = y;
53     return temp;
54 }
55
56 GameScore makeGameScore( int red, int black )
57 {
58     struct gameScoreStruct temp;
59     temp.red = red;
60     temp.black = black;
61     return temp;
62 }
63
64 BoardMove makeMove( BoardField from, BoardField to )
65 {
66     struct boardMoveStruct temp;
67     temp.from.x = from.x;
68     temp.from.y = from.y;
69     temp.to.x = to.x;
70     temp.to.y = to.y;
71     return temp;
72 }
73
74 BoardMove makeMoveFromInt( int fromx, int fromy, int tox, int toy )
75 {
76     struct boardMoveStruct temp;
77     temp.from.x = fromx;
78     temp.from.y = fromy;
79     temp.to.x = tox;
80     temp.to.y = toy;
81     return temp;
82 }
83
84 BoardSize makeBoardSize( int height, int width )
85 {
86     struct boardSizeStruct temp;
87     temp.height = height;
88     temp.width = width;
89     return temp;
90 }
91
92 SimpleList makeSimpleList()
93 {
94     struct simpleList temp;
95
96     temp.elementsInList = 0 ;
97     temp.head = NULL;
98     temp.tail = NULL;
```

```
99     return temp;
100 }
101
102 void concatSimpleLists(SimpleList *a, SimpleList *b)
103 {
104     if( b->ellementsInList == 0 )
105         return;
106
107     if( a->ellementsInList > 0 )
108     {
109         a->tail->next = b->head;
110         a->tail = b->tail;
111         a->ellementsInList += b->ellementsInList;
112     }
113
114     else
115     {
116         a->head = b->head;
117         a->tail = b->tail;
118         a->ellementsInList = b->ellementsInList;
119     }
120 }
121
122 void addEllementToSimpleList(SimpleList *list, BoardMove *data)
123 {
124     if( list->ellementsInList != 0 )
125     {
126         list->tail->next = malloc(sizeof(simpleListEllement));
127         list->tail = list->tail->next;
128         list->tail->moveData = *data;
129         list->ellementsInList++;
130     }
131     else if( list->ellementsInList == 0 )
132     {
133         list->tail = malloc(sizeof(simpleListEllement));
134         list->tail->moveData = *data;
135         list->head = list->tail;
136         list->ellementsInList++;
137     }
138 }
139
140 void removeHeadFromSimpleList(SimpleList *list )
141 {
142     if( list->ellementsInList > 1 )
143     {
144         simpleListEllement *nextHead = list->head->next;
145
146         free( list->head );
147         list->head = nextHead;
148         list->ellementsInList--;
149     }
150     else if( list->ellementsInList == 1 )
151     {
152         free( list->head );
153     }
```

```
154         list->head = NULL;
155         list->tail = NULL;
156
157         list->ellementsInList --;
158     }
159 }
160
161 void freeSimpleList(SimpleList *list )
162 {
163
164     while ( list->ellementsInList > 0 )
165     {
166         removeHeadFromSimpleList(list);
167     }
168 }
169
170 GameState makeGameState( BoardSize boardSize )
171 {
172     struct gameStateStruct temp;
173
174     temp.score.red = 0 ;
175     temp.score.black = 0 ;
176
177     temp.blackPicesOnBoard = 0 ;
178     temp.redPicesOnBoard = 0 ;
179
180     temp.boardSize = boardSize;
181
182     temp.playerMoving = PLAYER_RED;
183     temp.gameStatus = RUNNING;
184
185     temp.board = malloc(boardSize.width * sizeof(BoardFieldContent
186                         *));
187
188     int i;
189     for(i = 0 ; i < boardSize.width; i++)
190     {
191         temp.board[i] = malloc(boardSize.height * sizeof(
192                             BoardFieldContent));
193
194         int x, y;
195         for ( x = 0 ; x < boardSize.width ; x++ ) {
196             for ( y = 0 ; y < boardSize.height ; y++ ) {
197                 temp.board[x][y] = makeBoardFieldWithContent( FALSE,
198                                                 FALSE );
199             }
200         }
201
202 // Implementation for the BoardField object wrapper.
203 @implementation BoardFieldObject
204
205 - (BoardField)retriveField
```

```

206 {
207     return board;
208 }
209
210 - (id)initWithField:(BoardField)bf
211 {
212     self = [super init];
213     if (self != nil) {
214         board.x = bf.x;
215         board.y = bf.y;
216     }
217     return self;
218 }
219
220 + (id)boardfieldObjectWithField:(BoardField)bf
221 {
222     return [[BoardFieldObject alloc] initWithField:bf];
223 }
224
225 @end
226
227 // Implementation for the MoveObject object wrapper.
228 @implementation MoveObject
229
230 - (BoardMove)retriveMove
231 {
232     return m;
233 }
234 - (id)initWithMove:(BoardMove)theMove
235 {
236     self = [super init];
237     if (self != nil)
238     {
239         m.from.x = theMove.from.x;
240         m.from.y = theMove.from.y;
241         m.to.x = theMove.to.x;
242         m.to.y = theMove.to.y;
243     }
244     return self;
245 }
246
247 // This method is used by NSSet (and others) to determine if 2
248 // objects is identical.
249 - (BOOL)isEqual:(id)anObject
250 {
251     if( ![self isKindOfClass: [anObject class]] )
252         return FALSE;
253
254     BoardMove otherMove = [anObject retriveMove];
255
256     if( otherMove.from.x == m.from.x &&
257         otherMove.from.y == m.from.y &&
258         otherMove.to.x == m.to.x &&
259         otherMove.to.y == m.to.y)
260     {

```

```

260         return TRUE;
261     }
262     return false;
263 }
264
265 // This method is used by NSSet (and others) to determine if 2
266 // objects is identical.
267 - (unsigned)hash
268 {
269     unsigned hashVal;
270     hashVal = m.from.x + m.from.y * 10 + m.to.x * 100 + m.to.y *
271             1000 ;
272     return hashVal;
273 }
274
275 // Method to initalise object.
276 + (id)moveObjectWithMove:(BoardMove)theMove
277 {
278     return [[MoveObject alloc] initWithMove:theMove];
279 }
279 @end

```

C.1.5 GameLogic.h

```

1 // Kolibrat
2 // GameLogic.h
3 //
4 // Created by Aron Lindberg.
5
6 #import "Datastructures.h"
7
8
9
10 @interface GameLogic : NSObject
11 {
12 // Private instance variabels.
13 @private
14     int maxGoals;
15     int maxPicesOnBoard;
16     BoardSize boardSize;
17 }
18
19 // Public instance methods.
20 - (id)initWithMaxPices:(int)max_goalsToWin:(int)goals boardSize:(
21     BoardSize)board;
22
23 - (GameState)CreateNewGameState;
24 - (void)resetGameState:(GameState *)gs;
25 - (BOOL)playerMovingCanInsertPieceOnState:(GameState *)gs;

```

```

26 - (NSSet *)legalMovesForPiceInField:(BoardField)field withState:(
27   GameState *)gs;
28 - (NSSet *)allLegalMoves:(GameState *)gs;
29 - (BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs;
30
31 // Public C instance methods.
32 SimpleList allLegalMoves(GameState *gs);
33 void legalMovesForPiceInField(BoardField *field, GameState *gs,
34   SimpleList *superList, SimpleList *goodList, SimpleList *
35   badList);
36 BOOL makeMoveOnState(BoardMove *playerMove, GameState *gs);
37 void freeGameState( GameState *state );
38 GameState copyGameState( GameState *state);
39 BOOL blackPlayerAdheadOf(int x, int y, GameState *gs);
40 BOOL redPlayerAdheadOf(int x, int y, GameState *gs);
41 @end

```

C.1.6 GameLogic.m

```

1 // Kolibrat
2 // GameLogic.m
3 //
4 // Created by Aron Lindberg.
5
6 #import "GameLogic.h"
7
8 @implementation GameLogic
9
10 // Global variabels, shared in al instances of GameLogic.
11 static int *maxGoalsPointer;
12 static int *maxPicesOnBoardPointer;
13 static BoardSize *boardSizePointer;
14
15 // Public methods.
16 - (id)initWithMaxPices:(int)max goalsToWin:(int)goals boardSize:(
17   BoardSize)board
18 {
19   self = [super init];
20   if (self != nil)
21   {
22     maxPicesOnBoard = max;
23     boardSize = board;
24     maxGoals = goals;
25     maxGoalsPointer = &maxGoals;
26     maxPicesOnBoardPointer = &maxPicesOnBoard;
27     boardSizePointer = &boardSize;
28   }
29   return self;
30 }
31

```

```

32 - (GameState)CreateNewGameState
33 {
34     GameState gs = makeGameState(boardSize);
35     [self resetGameState: &gs];
36     return gs;
37 }
38
39 - (void)resetGameState:(GameState *)gs
40 {
41     int x, y;
42     for (x = 0 ; x < boardSize.width ; x++) {
43         for (y = 0 ; y < boardSize.height ; y++) {
44             gs->board[x][y] = makeEmptykField();
45         }
46     }
47     gs->score = makeGameScore(0, 0);
48     gs->playerMoving = PLAYER_RED;
49     gs->redPicesOnBoard = 0;
50     gs->blackPicesOnBoard = 0;
51     gs->gameStatus = RUNNING;
52 }
53
54 - (BOOL)playerMovingCanInsertPieceOnState:(GameState *)gs;
55 {
56     if( gs->playerMoving == PLAYER_RED && gs->redPicesOnBoard <
57         maxPicesOnBoard )
58         return TRUE;
59     if( gs->playerMoving == PLAYER_BLACK && gs->blackPicesOnBoard <
60         maxPicesOnBoard )
61         return TRUE;
62 }
63
64 - (NSSet *)legalMovesForPiceInField:(BoardField)field withState:(
65     GameState *)gs
66 {
67     if(field.x < 0 || field.y < 0 || field.x > boardSize.width - 1
68         || field.y > boardSize.height - 1 )
69     {
70         NSEException* e = [NSEException exceptionWithName:@"Move
71             error." reason:[NSString stringWithFormat:@"The board
72                 field (%i,%i) is outside of the board.", field.x,
73                 field.y] userInfo:nil];
74         @throw e;
75     }
76     NSMutableSet *setOfLegalMoves = [NSMutableSet setWithCapacity:4
77     ];
78
79     SimpleList superList = makeSimpleList();
80     SimpleList goodList = makeSimpleList();
81     SimpleList badList = makeSimpleList();
82 }
```

```

78     legalMovesForPiceInField( &field, gs, &superList, &goodList, &
79         badList );
80
81     concatSimpleLists( &superList, &goodList );
82     concatSimpleLists( &superList, &badList );
83
84     while( superList.ellementsInList > 0 )
85     {
86         [setOfLegalMoves addObject: [MoveObject moveObjectWithMove:
87             superList.head->moveData]];
88         removeHeadFromSimpleList( &superList );
89     }
90
91
92 - (NSSet *)allLegalMoves:(GameState *)gs
93 {
94     NSMutableSet *setOfLegalMoves = [NSMutableSet setWithCapacity:
95         10 ];
96
97     SimpleList list = allLegalMoves(gs);
98
99     while( list.ellementsInList > 0 )
100    {
101        MoveObject *mo = [MoveObject moveObjectWithMove: list.head-
102            >moveData];
103
104        [setOfLegalMoves addObject: mo];
105        [mo release];
106        removeHeadFromSimpleList( &list );
107    }
108
109    return setOfLegalMoves;
110 }
111
112 SimpleList allLegalMoves(GameState *gs)
113 {
114     SimpleList superList = makeSimpleList();
115     SimpleList goodList = makeSimpleList();
116     SimpleList badList = makeSimpleList();
117
118     int x, y;
119     for ( x = 0 ; x < boardSizePointer->width ; x++ )
120     {
121         for ( y = 0 ; y < boardSizePointer->height ; y++ )
122         {
123             BoardField boardField = makeBoardField( x, y );
124             legalMovesForPiceInField( &boardField , gs, &
125                 superList, &goodList, &badList );
126         }
127     }

```

```

128     concatSimpleLists( &superList, &goodList );
129     concatSimpleLists( &superList, &badList );
130
131     return superList;
132 }
133
134 - (BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs
135 {
136     BoardField from = playerMove.from;
137     BoardField to = playerMove.to;
138
139     if(from.x < 0 || from.y < 0 || from.x > boardSize.width - 1 ||
140        from.y > boardSize.height - 1 )
141     {
142         NSError* e = [NSError exceptionWithName:@"Move
143             error." reason:[NSString stringWithFormat:@"The board
144                 field (%i,%i) is outside of the board.", from.x, from.y
145                 ] userInfo:nil];
146         @throw e;
147     }
148
149     if(to.x < 0 || to.y < 0 || to.x > boardSize.width - 1 || to.y >
150        boardSize.height - 1 )
151     {
152         NSError* e = [NSError exceptionWithName:@"Move
153             error." reason:[NSString stringWithFormat:@"The board
154                 field (%i,%i) is outside of the board.", to.x, to.y
155                 ] userInfo:nil];
156         @throw e;
157     }
158
159     NSSet *setOfLegalMoves = [self legalMovesForPiceInField:from
160                               withState:gs];
161
162     if( [setOfLegalMoves containsObject:[MoveObject
163                                         moveObjectWithMove:playerMove]] )
164     { // The move is legal.
165         return makeMoveOnState( &playerMove, gs);
166     }
167     return FALSE;
168 }
169
170 BOOL makeMoveOnState(BoardMove *playerMove, GameState *gs)
171 {
172
173     if( gs->gameStatus.gameOver == TRUE )
174         return FALSE;
175
176     BoardField from = playerMove->from;
177     BoardField to = playerMove->to;
178
179     gs->lastMove.from = playerMove->from;
180     gs->lastMove.to = playerMove->to;
181
182 }
```

```

172     if( BOARDFIELDS_IS_EQUAL( from, to ) && gs->playerMoving ==
173         PLAYER_RED && to.y == 0 )
174     { // Insert peice for red.
175         gs->board[to.x][to.y] = makeRedField();
176         gs->redPicesOnBoard++;
177     }
178
179     else if( BOARDFIELDS_IS_EQUAL( from, to ) && gs->playerMoving ==
180             PLAYER_RED && to.y == ( boardSizePointer->height - 1 ) )
181     { // Score point for red.
182         gs->board[to.x][to.y] = makeEmptykField();
183         gs->score.red++;
184         gs->redPicesOnBoard--;
185         if( gs->score.red == *maxGoalsPointer )
186         {
187             gs->gameStatus = RED_WON;
188         }
189     }
190
191     else if( BOARDFIELDS_IS_EQUAL( from, to ) && gs->playerMoving ==
192             PLAYER_BLACK && to.y == ( boardSizePointer->height - 1 ) )
193     { // Insert peice for black.
194         gs->board[to.x][to.y] = makeBlackField();
195         gs->blackPicesOnBoard++;
196     }
197
198     else if( BOARDFIELDS_IS_EQUAL( from, to ) && gs->playerMoving ==
199             PLAYER_BLACK && to.y == 0 )
200     { // Score point for black.
201         gs->board[to.x][to.y] = makeEmptykField();
202         gs->score.black++;
203         gs->blackPicesOnBoard--;
204         if( gs->score.black == *maxGoalsPointer )
205         {
206             gs->gameStatus = BLACK_WON;
207         }
208     }
209
210     else
211     { // If the move is an attack.
212         if( RED_FIELD( gs->board[to.x][to.y] ) )
213             gs->redPicesOnBoard--;
214         if( BLACK_FIELD( gs->board[to.x][to.y] ) )
215             gs->blackPicesOnBoard--;
216
217         // Move piece on the board.
218         gs->board[from.x][from.y] = makeEmptykField();
219         if( gs->playerMoving == PLAYER_RED )
220             gs->board[to.x][to.y] = makeRedField();
221         else if( gs->playerMoving == PLAYER_BLACK )
222             gs->board[to.x][to.y] = makeBlackField();
223     }
224
225     if( gs->gameStatus.gameOver == TRUE )

```

```
222     {
223         return TRUE;
224     }
225
226     // finds the next player.
227     BOOL nextPlayer;
228     BOOL otherPlayer;
229
230     if( gs->playerMoving == PLAYER_RED )
231     {
232         nextPlayer = PLAYER_BLACK;
233         otherPlayer = PLAYER_RED;
234     }
235     else if( gs->playerMoving == PLAYER_BLACK )
236     {
237         nextPlayer = PLAYER_RED;
238         otherPlayer = PLAYER_BLACK;
239     }
240
241     gs->playerMoving = nextPlayer;
242
243     SimpleList allMoves = makeSimpleList();
244     allMoves = allLegalMoves( gs );
245
246     if ( allMoves.ellementsInList == 0 )
247     {
248         gs->playerMoving = otherPlayer;
249         allMoves = allLegalMoves( gs );
250
251         if( allMoves.ellementsInList == 0 )
252         { // No player can move, playerMoving has lost.
253             if( gs->playerMoving == PLAYER_RED)
254                 gs->gameStatus = BLACK_WON;
255             else
256                 gs->gameStatus = RED_WON;
257         }
258     }
259     freeSimpleList(&allMoves);
260
261     return TRUE;
262 }
263
264 GameState copyGameState( GameState *state)
265 {
266     struct gameStateStruct temp;
267
268     temp.blackPicesOnBoard = state->blackPicesOnBoard;
269     temp.redPicesOnBoard = state->redPicesOnBoard;
270     temp.playerMoving = state->playerMoving;
271     temp.gameStatus = state->gameStatus;
272     temp.score = state->score;
273     temp.lastMove = state->lastMove;
274
275     temp.boardSize = state->boardSize;
276 }
```

```

277     temp.board = malloc(boardSizePointer->width * sizeof(
278         BoardFieldContent *));
279
280     int i;
281     for(i = 0 ; i < boardSizePointer->width; i++)
282     {
283         temp.board[i] = malloc(boardSizePointer->height * sizeof(
284             BoardFieldContent));
285     }
286
287     int x, y;
288     for ( x = 0 ; x < boardSizePointer->width ; x++ ) {
289         for ( y = 0 ; y < boardSizePointer->height ; y++ ) {
290             temp.board[x][y] = state->board[x][y];
291         }
292     }
293
294     return temp;
295 }
296
297 void freeGameState( GameState *state )
298 {
299     int i;
300     for(i = 0 ; i < boardSizePointer->width; i++)
301     {
302         free(state->board[i]);
303     }
304
305     free(state->board);
306 }
307
308 if( gs->gameStatus.gameOver == TRUE )
309     return;
310
311 register int x = field->x;
312 register int y = field->y;
313
314 // All moving rules for the red player.
315 if( gs->playerMoving == PLAYER_RED )
316 {
317     // Red wants to insert a piece on the board.
318     if( EMPTY_FIELD( gs -> board[x][y] ) && y == 0 && gs->
319         redPicesOnBoard < *maxPicesOnBoardPointer )
320     {
321         BoardMove move = makeMoveFromInt( x, y, x , y );
322
323         if( blackPlayerAdheadOf( x, y, gs ))
324             addEllementToSimpleList( badList, &move );
325         else
326             addEllementToSimpleList( goodList, &move );
327     }
328 }
```

```

327
328     // Red wants to move left and forward.
329     if( RED_FIELD( gs->board[x][y] ) && x > 0 && y < (
330         boardSizePointer->height - 1 )
331         && EMPTY_FIELD( gs->board[x - 1 ][y + 1 ] ))
332     {
333         BoardMove move = makeMoveFromInt( x, y, x - 1, y + 1 );
334
335         if( blackPlayerAdheadOf( x, y, gs ) )
336             addEllementToSimpleList( badList, &move );
337         else
338             addEllementToSimpleList( goodList, &move );
339     }
340
341     // Red wants to move right and forward.
342     if( RED_FIELD( gs->board[x][y] ) && x < ( boardSizePointer-
343         >width - 1 ) &&
344         y < ( boardSizePointer->height - 1 ) && EMPTY_FIELD(
345             gs->board[x + 1 ][y + 1 ] ))
346     {
347         BoardMove move = makeMoveFromInt( x, y, x + 1, y + 1 );
348
349         if( blackPlayerAdheadOf( x, y, gs ) )
350             addEllementToSimpleList( badList, &move );
351         else
352             addEllementToSimpleList( goodList, &move );
353
354     // Red wants to attack black.
355     if( RED_FIELD( gs->board[x][y] ) && y < ( boardSizePointer-
356         >height - 1 ) &&
357         BLACK_FIELD( gs->board[x][y + 1 ] ))
358     {
359         BoardMove move = makeMoveFromInt( x, y, x, y + 1 );
360         addEllementToSimpleList( superList, &move );
361     }
362
363     // Red wants to jump over black.
364     if( RED_FIELD( gs->board[x][y] ) && y < ( boardSizePointer-
365         >height - 2 ) &&
366         BLACK_FIELD( gs->board[x][y + 1 ] ))
367     {
368         int jumpDistance = 1 ;
369         while( y + jumpDistance < (boardSizePointer->height - 2
370             ) &&
371             BLACK_FIELD( gs->board[x][y + 1 + jumpDistance]
372                 ))
373         {
374             jumpDistance++;
375         }
376         if( EMPTY_FIELD(gs->board[x][y + 1 + jumpDistance]))
377         {
378             BoardMove move = makeMoveFromInt( x, y, x, y + 1 +
379                 jumpDistance );
380             addEllementToSimpleList( superList, &move );

```

```

374         }
375     }
376
377     // Red wants to gain a point.
378     if( RED_FIELD( gs->board[x][y] ) && y == (
379         boardSizePointer->height - 1 ))
380     {
381         BoardMove move = makeMoveFromInt( x, y, x, y );
382         addEllementToSimpleList( superList, &move );
383     }
384
385     // All moving rules for the black player.
386     if( gs->playerMoving == PLAYER_BLACK )
387     {
388         // Black wants to insert a piece on the board.
389         if( EMPTY_FIELD( gs->board[x][y] ) && y == (
390             boardSizePointer->height - 1 )
391             && gs->blackPicesOnBoard < *maxPicesOnBoardPointer )
392         {
393             BoardMove move = makeMoveFromInt( x, y, x, y );
394
395             if( redPlayerAdheadOf( x, y, gs ) )
396                 addEllementToSimpleList( badList, &move );
397             else
398                 addEllementToSimpleList( goodList, &move );
399         }
400
401         // Black wants to move left and forward.
402         if( BLACK_FIELD( gs->board[x][y] ) && x > 0 && y > 0 &&
403             EMPTY_FIELD( gs->board[x - 1 ][y - 1 ] ) )
404         {
405             BoardMove move = makeMoveFromInt( x, y, x - 1, y - 1 );
406
407             if( redPlayerAdheadOf( x, y, gs ) )
408                 addEllementToSimpleList( badList, &move );
409             else
410                 addEllementToSimpleList( goodList, &move );
411         }
412
413         // Black wants to move right and forward.
414         if( BLACK_FIELD( gs->board[x][y] ) && x < (
415             boardSizePointer->width - 1 )
416             && y > 0 && EMPTY_FIELD( gs->board[x + 1 ][y - 1 ] ) )
417         {
418             BoardMove move = makeMoveFromInt( x, y, x + 1, y - 1 );
419
420             if( redPlayerAdheadOf( x, y, gs ) )
421                 addEllementToSimpleList( badList, &move );
422             else
423                 addEllementToSimpleList( goodList, &move );
424
425         // Black wants to attack red.
426         if( BLACK_FIELD( gs->board[x][y] ) && y > 0 &&

```

```

426             RED_FIELD( gs->board[x][y - 1] ) )
427         {
428             BoardMove move = makeMoveFromInt( x, y, x, y - 1 );
429             addEllementToSimpleList( superList, &move );
430         }
431
432         // Black wants to jump over red.
433         if( BLACK_FIELD( gs->board[x][y] ) && y > 1 &&
434             RED_FIELD( gs->board[x][y - 1] ) )
435         {
436             int jumpDistance = 1 ;
437             while( (y - 1) - jumpDistance > 0 &&
438                 RED_FIELD( gs->board[x][(y - 1) - jumpDistance]
439                         ] ) )
440             {
441                 jumpDistance++;
442             }
443             if( EMPTY_FIELD( gs->board[x][(y - 1) - jumpDistance]
444                         ) )
445             {
446                 BoardMove move = makeMoveFromInt( x, y, x, (y - 1)
447                     - jumpDistance );
448                 addEllementToSimpleList( superList, &move );
449             }
450
451             // Black wants to gain a point.
452             if( BLACK_FIELD( gs->board[x][y] ) && y == 0 )
453             {
454                 BoardMove move = makeMoveFromInt( x, y, x, y );
455                 addEllementToSimpleList( superList, &move );
456             }
457
458     // Helper methods.
459     BOOL blackPlayerAdheadOf( int x, int y, GameState *gs )
460     {
461         if( BLACK_FIELD( gs->board[x][y + 1] ) )
462             return TRUE;
463         else
464             return FALSE;
465     }
466
467     BOOL redPlayerAdheadOf( int x, int y, GameState *gs )
468     {
469         if( BLACK_FIELD( gs->board[x][y - 1] ) )
470             return TRUE;
471         else
472             return FALSE;
473     }
474
475 @end

```

C.1.7 GameEngine.h

```

1 //  Kolibrat
2 //  GameEngine.h
3 //
4 //  Created by Aron Lindberg.
5
6 #import "Datastructures.h"
7 #import "GUIProtocol.h"
8 #import "PlayerProtocol.h"
9 #import "GameLogic.h"
10
11 // #define TIMEOUT [NSDate dateWithTimeIntervalSinceNow:400 ]
12 #define TIMEOUT [NSDate distantFuture]
13
14 @interface GameEngine : NSObject
15 {
16 //  Private instance variabels.
17 @private
18     id redPlayer;
19     id blackPlayer;
20     id engineGUI;
21     int maxGoals;
22     BOOL delayNexPlayer;
23     BOOL doCallNextPlayerWhenResume;
24
25     BoardSize boardSize;
26     GameState realGameState;
27     GameState *gameStatePointer;
28
29     NSNotificationQueue *queue;
30     GameLogic *gl;
31
32     NSDate *timeToNextTurn;
33     NSDate * timeToStoptheGame;
34 }
35
36 //  Public methods.
37 - (id)initWithPlayersRed:(id)red andBlack:(id)black goalsToWin:(int)
38     )goals GameBoardDim:(BoardSize)board MaxPices:(int)max
39     connectToGUI:(id)gui;
40 - (void)resetGame;
41
42 //  Methods used by the Player Objects.
43 - (BOOL)playerMove:(BoardMove)playerMove fromPlayer:(id)player;
44 - (GameState)gameState;
45 - (void)SelectedPiece:(BoardField)bf fromPlayer:(id)player;
46
47 //  Methods used by the GUI.
48 - (void)delayNextPlayer:(BOOL)response;
49
50 @end

```

C.1.8 GameEngine.m

```

1 //  Kolibrat
2 //  GameEngine.m
3 //
4 //  Created by Aron Lindberg.
5
6 #import "GameEngine.h"
7
8 @implementation GameEngine
9
10 // Called when the object instance is destroyed.
11 - (void)dealloc
12 {
13     // Release all child objects used by the GameEngine.
14     [gl release];
15     [redPlayer release];
16     [blackPlayer release];
17
18     [[NSNotificationCenter defaultCenter] removeObserver:self];
19
20     [super dealloc];
21 }
22
23 // invoked when the GameEngine is initialized.
24 - (id)initWithPlayersRed:(id)red andBlack:(id)black goalsToWin:(int)
25     goals GameBoardDim:(BoardSize)board MaxPices:(int)max
26     connectToGUI:(id)gui
27 {
28     if ((self = [super init]) != nil) {
29
30         doCallNextPlayerWhenResume = NO;
31         delayNexPlayer = NO;
32
33         gl = [[GameLogic alloc] initWithMaxPices:max goalsToWin:
34             goals boardSize:board];
35         realGameState = [gl CreateNewGameState];
36         gameStatePointer = &realGameState;
37
38         queue = [NSNotificationQueue defaultQueue];
39         NSNotificationCenter *mainCenter = [NSNotificationCenter
40             defaultCenter];
41
42         [mainCenter addObserver:self selector:@selector(nextPlayer
43             :) name:@"NextPlayer" object:nil];
44
45         boardSize.width = board.width;
46         boardSize.height = board.height;
47
48         redPlayer = red;
49         blackPlayer = black;
50         maxGoals = goals;
51
52         engineGUI = gui;

```

```
48      timeToNextTurn = [NSDate date];
49      [timeToNextTurn retain];
50
51      [engineGUI updateToState: copyGameState( gameStatePointer
52          ) ];
53
54      if( ![engineGUI conformsToProtocol:@protocol(GUI_Protocol)]
55          && engineGUI != nil )
56      {
57          NSError* e = [NSError exceptionWithName:@"GUI
58              Error." reason:@"The GUI does not responds to
59              needed method calls." userInfo:nil];
60          @throw e;
61      }
62
63      // Ensure that red player responds to nesseary method cals.
64      if( ![redPlayer conformsToProtocol:@protocol(
65          Player_Protocol )])
66      {
67          NSError* e = [NSError exceptionWithName:@""
68              Player Error" reason:@"The Player does not responds
69              to needed method calls." userInfo:nil];
70          @throw e;
71      }
72
73  }
74
75  // Send message that next player should be given the turn.
76  NSNotification *message = [NSNotification notificationWithName:
77      @"NextPlayer" object:nil];
78  [queue enqueueNotification:message postingStyle:NSPostWhenIdle
79      ];
80
81  timeToStoptheGame = TIMEOUT;
82  [timeToStoptheGame retain];
83
84  return self;
85 }
86
87 // Givs turn to next player.
88 - (void)nextPlayer:(NSNotification *)notification
89 {
90     [NSThread sleepUntilDate: timeToNextTurn];
91     [timeToNextTurn release];
92 }
```

```

90
91     // Delay to ensure that the game is watchable when to AI's play
92     // each other.
93     timeToNextTurn = [NSDate dateWithTimeIntervalSinceNow:0.5 ];
94     [timeToNextTurn retain];
95
96     if( delayNextPlayer == NO )
97     {
98         if( realGameState.playerMoving == PLAYER_RED )
99             [redPlayer startNewTurn];
100        else if( realGameState.playerMoving == PLAYER_BLACK )
101            [blackPlayer startNewTurn];
102    }
103    else
104    {
105        doCallNextPlayerWhenResume = YES;
106    }
107
108 // Resets the games state.
109 - (void)resetGame
110 {
111     [gl resetGameState: gameStatePointer ];
112
113     [redPlayer reset];
114     [blackPlayer reset];
115
116     if( engineGUI != nil )
117     {
118         [engineGUI updateToState: copyGameState( gameStatePointer
119             ) ];
120     }
121
122     // Send message that next player should be given the turn.
123     NSNotification *message = [NSNotification notificationWithName:
124         @"NextPlayer" object:nil];
125     [queue enqueueNotification:message postingStyle:NSPostWhenIdle
126         ];
127 }
128
129 // Called by the player objects when they wish to make a move.
130 - (BOOL)playerMove:(BoardMove)playerMove fromPlayer:(id)player
131 {
132     if( GAME_NOT_RUNNING( realGameState.gameStatus ))// The game is
133         not in play.
134         return FALSE;
135
136     BOOL returnValue;
137
138     if( realGameState.playerMoving == PLAYER_RED && redPlayer ==
139         player )

```

```

138     { // This method call came from red player, and red has the
139         returnValue = [gl makeMove:playerMove withState: &
140                         realGameState];
141     }
142
143     else if( realGameState.playerMoving == PLAYER_BLACK &&
144               blackPlayer == player )
145     { // This method call came from black player, and black has
146       the turn.
147       returnValue = [gl makeMove:playerMove withState: &
148                         realGameState];
149     }
150
151     else
152     { // The call is not legal, the player don't have the turn.
153       return FALSE;
154     }
155
156     if( engineGUI != nil ) // Update the GUI.
157     [engineGUI updateToState: copyGameState( gameStatePointer
158                               ) ];
159
160     // Check if the game is over.
161     if( realGameState.gameStatus.gameOver == TRUE )
162     {
163         // Check if red won.
164         if( engineGUI != nil && realGameState.gameStatus.winner ==
165             PLAYER_RED )
166             [engineGUI gameOverWithWinner: [redPlayer playerName]];
167
168         // Check if black won
169         if( engineGUI != nil && realGameState.gameStatus.winner ==
170             PLAYER_BLACK )
171             [engineGUI gameOverWithWinner: [blackPlayer playerName
172                               ]];
173
174         NSNotification *message;
175         if( realGameState.gameStatus.winner == PLAYER_RED )
176             message = [NSNotification notificationWithName:@"GameOver"
177                         object:redPlayer];
178         else
179             message = [NSNotification notificationWithName:@"GameOver"
180                         object:blackPlayer];
181
182         [queue enqueueNotification:message postingStyle:
183                         NSPostWhenIdle];
184
185         // Return now do not start a new turn.
186         return returnValue;
187     }
188
189     // This code stops the game if timeToStoptheGame is set. Used
190     // by the Simulated Annealing program.
191     if( [timeToStoptheGame timeIntervalSinceNow] <= 0 )

```

```
180     {
181         [timeToStoptheGame release];
182         NSNotification *message;
183         if( realGameState.score.red > realGameState.score.black )
184         {
185             message = [NSNotification notificationWithName:@"
186                 GameOver" object:redPlayer];
187             [queue enqueueNotification:message postingStyle:
188                 NSPostWhenIdle];
189         }
190         else if( realGameState.score.red <
191                 realGameState.score.black )
192         {
193             message = [NSNotification notificationWithName:@"
194                 GameOver" object:blackPlayer];
195             [queue enqueueNotification:message postingStyle:
196                 NSPostWhenIdle];
197         }
198         else
199         {
200             NSNotification *message;
201             srand([NSDate date] timeIntervalSince1970 ]);
202             if( random() % 100 > 49 )
203             {
204                 message = [NSNotification notificationWithName:@"
205                     GameOver" object:redPlayer];
206                 [queue enqueueNotification:message postingStyle:
207                     NSPostWhenIdle];
208             }
209             else
210             {
211                 message = [NSNotification notificationWithName:@"
212                     GameOver" object:blackPlayer];
213                 [queue enqueueNotification:message postingStyle:
214                     NSPostWhenIdle];
215             }
216         }
217         return returnValue;
218     }
219     // Returns the game state.
220     - (GameState)gameState
221     {
222         return copyGameState( gameStatePointer );
223     }
224 }
```

```

223 // This method is called by the players. This tells the engine that
224 // the player
225 // wants to highlight this piece in the GUI.
226 - (void)SelectedPiece:(BoardField)bf fromPlayer:(id)player
227 {
228     if( engineGUI == nil || GAME_NOT_RUNNING(
229         realGameState.gameStatus ))
230     {
231         return;
232
233         NSError* e = [NSError exceptionWithName:@"Move
234             error." reason:[NSString stringWithFormat:@"The board
235                 field (%i,%i) is outside of the board.", bf.x, bf.y]
236                 userInfo:nil];
237
238         @throw e;
239     }
240
241     GameState* gsp = gameStatePointer;
242
243     int x = bf.x;
244     int y = bf.y;
245
246     // Rules for Red Player.
247     if( realGameState.playerMoving == PLAYER_RED && redPlayer ==
248         player)
249     {
250         // The user is about to insert a peice.
251         if( y == 0 && EMPTY_FIELD(gsp->board[x][y]) && [gl
252             playerMovingCanInsertPieceOnState: gsp] )
253             [engineGUI drawOpaquePiceAt:bf forPlayer:PLAYER_RED];
254
255         // The user is about to gain a point.
256         if( y == (boardSize.height - 1) && RED_FIELD( gsp->board[x
257             ][y] ) )
258             [engineGUI drawOpaquePiceAt:bf forPlayer:PLAYER_RED];
259
260         // The user wants to highlight a piece.
261         if( y != (boardSize.height - 1) && RED_FIELD( gsp->board[x
262             ][y] ) )
263         {
264             [engineGUI highlightPiceAt:bf];
265
266             // Now highlighting possible moves for that piece.
267             NSSet *setOfMoves = [gl legalMovesForPiceInField:bf
268                 withState: gameStatePointer];
269             NSEnumerator *e = [setOfMoves objectEnumerator];
270             MoveObject *thisMoveObject;
271
272             while (thisMoveObject = [e nextObject])
273             {
274                 BoardMove thisMove = [thisMoveObject retriveMove];
275                 [engineGUI highlightField: thisMove.to ];
276             }

```

```

267         }
268     }
269
270     // Rules for Black Player.
271     if( realGameState.playerMoving == PLAYER_BLACK && blackPlayer
272         == player)
273     {
274         // The user is about to insert a peice.
275         if( y == (boardSize.height - 1) && EMPTY_FIELD( gsp->board
276             [x][y] ) && [gl playerMovingCanInsertPieceOnState: gsp]
277                 )
278             [engineGUI drawOpaquePiceAt:bf forPlayer:PLAYER_BLACK];
279
280         // The user is about to gain a point.
281         if( y == 0 && BLACK_FIELD( gsp->board[x][y] ) )
282             [engineGUI drawOpaquePiceAt:bf forPlayer:PLAYER_BLACK];
283
284         // The user wants to highlight a piece.
285         if( y != 0 && BLACK_FIELD( gsp->board[x][y] ) )
286         {
287             [engineGUI highlightPiceAt:bf];
288
289             // Now highlighting possible moves for that piece.
290             NSSet *setOfMoves = [gl legalMovesForPiceInField:bf
291                 withState: gsp];
292             NSEnumerator *e = [setOfMoves objectEnumerator];
293             MoveObject *thisMoveObject;
294
295             while (thisMoveObject = [e nextObject])
296             {
297                 BoardMove thisMove = [thisMoveObject retriveMove];
298                 [engineGUI highlightField: thisMove.to ];
299             }
300         }
301     }
302
303 // Called by the GUI when the new game sheet is shown.
304 - (void)delayNextPlayer:(BOOL)response
305 {
306     if( response == YES )
307     {
308         delayNexPlayer = YES;
309     }
310
311     else if( response == NO )
312     {
313         delayNexPlayer = NO;
314         if( doCallNextPlayerWhenResume == YES )
315         {
316             doCallNextPlayerWhenResume = NO;
317             NSNotification *message = [NSNotification
318                 notificationWithName:@"NextPlayer" object:nil];
319             [queue enqueueNotification:message postingStyle:
320                 NSPostWhenIdle];

```

```

316         }
317     }
318     return;
319 }
320
321 @end

```

C.1.9 GameController.h

```

1 //  Kolibrat
2 //  GameController.h
3 //
4 //  Created by Aron Lindberg.
5
6 #import "Datastructures.h"
7 #import "GUIProtocol.h"
8 #import "GameBoard.h"
9 #import "GameEngine.h"
10
11 // Confirms to the GUI Protocol.
12 @interface GameController : NSWindowController < GUI_Protocol >
13 {
14 // Private instance variabels.
15 @private
16     IBOutlet NSTextField *blackScore;
17     IBOutlet NSTextField *redScore;
18     IBOutlet GameBoard *gb;
19     IBOutlet NSWindow *OptionsWindow;
20     IBOutlet NSMenuItem *restartMenu;
21     GameEngine *ge;
22     BOOL canRestartGame;
23     BOOL doHighlighting;
24     BoardSize boardSize;
25     float boardFieldDim;
26 }
27
28 // Public methods.
29 - (NSSize)windowWillResize:(NSWindow *)sender toSize:(NSSize)
   proposedFrameSize;
30 - (void)gameDidEnd:(NSWindow *)sheet returnCode:(int)returnCode
   contextInfo:(void *)contextInfo;
31 - (void)gameOverWithWinner:(NSString *)playerName;
32 - (void)highlightField:(BoardField)bf;
33 - (void)redrawOriginalState;
34 - (void)highlightPiceAt:(BoardField)bf;
35 - (void)drawOpaquePiceAt:(BoardField)bf forPlayer:(int)player;
36 - (void)setGameEngine:(GameEngine*)ge;
37 - (void)setHighlightState:(BOOL)highlight;
38 - (void)setBoardSize:(BoardSize)board;
39 - (void)updateToState:(GameState)bs;
40
41 @end

```

C.1.10 GameController.m

```
1 // Kolibrat
2 // GameController.m
3 //
4 // Created by Aron Lindberg.
5
6 #import "GameController.h"
7
8 @implementation GameController
9
10 // This method is called because GameController is the main game
11 // windows delegate. It ensures that the window resizes propperly.
12 - (NSSize)windowWillResize:(NSWindow *)sender toSize:(NSSize)
13     proposedFrameSize
14 {
15     // Calculates the maximun size of the window.
16     float MaxWindowSize = 128 * boardSize.width + 2 * 20 ;
17
18     // Calculates the new size of the border arraound the game
19     // board.
20     float border = proposedFrameSize.width / MaxWindowSize * 20 ;
21
22     // Calculates the new game board dimentions.
23     float gameboardWidth = proposedFrameSize.width - 2 * border;
24
25     // Sets the distance from the left window margin and the game
26     // board.
27     [gb setDisplayOffset:border];
28
29     // Calculates that sets the boardField dimention of the new game
30     // board.
31     boardFieldDim = gameboardWidth / boardSize.width;
32     [gb setSquareDim: boardFieldDim ];
33
34     float newWindowWidth = 2 * border + boardFieldDim *
35         boardSize.width;
36     float newWindowHeight = border + boardFieldDim *
37         boardSize.height + 59 ;
38
39     // Returns the updated windowsize that the window will resize
40     // to.
41     return NSMakeSize( newWindowWidth, newWindowHeight );
42 }
43
44 // This causes Kolibrat to terminate after last window is closed.
45 // This works because GameController is the NSApp delegate.
46 - (BOOL)applicationShouldTerminateAfterLastWindowClosed:(
47     NSApplication *)app
48 {
49     return YES;
50 }
51
52 // This method is called once, when the game window is loaded.
```

```
43 - (void)awakeFromNib
44 {
45     canRestartGame = NO;
46     [redScore intValue: 0 ];
47     [blackScore intValue: 0 ];
48
49     [[self window] useOptimizedDrawing:YES];
50 }
51
52 // This method is called by the GameEninge when the game is over.
53 - (void)gameOverWithWinner:(NSString *)playerName
54 {
55     NSString *title = @"Game Over";
56     NSString *defaultButton = @"OK";
57     NSString *alternateButton = @"Quit";
58     NSString *otherButton = @"Restart Game";
59     NSString *message;
60
61     if( playerName != nil )
62         message = [NSString stringWithFormat:@"The game is over. %@
63             won.", playerName];
64     else
65         message = @"The game is over, and ends in a draw since no
66             player can move.";
67
68 // Wait 0.8 secondt to dispay the GameOver dialog.
69 [NSThread sleepUntilDate: [NSDate dateWithTimeIntervalSinceNow: 0.8
70     ]];
71
72     NSBeginAlertSheet( title,
73                         defaultButton,
74                         alternateButton,
75                         otherButton,
76                         [self window],
77                         self,
78                         @selector(gameDidEnd:returnCode:contextInfo
79                         :),
80                         nil,
81                         nil,
82                         message );
83 }
84
85     [redScore intValue:score.red];
86     [blackScore intValue:score.black];
87 }
88
89 // All these messages are sent to the game board, that handles the
90 // aktual drawing.
91 - (void)highlightField:(BoardField)bf
92 {
93     if( doHighlighting == YES )
```

```
93         [gb highlightField:bf];
94     }
95
96 - (void)redrawOriginalState
97 {
98     [gb redrawOriginalState];
99 }
100
101 - (void)updateToState:(GameState)bs
102 {
103     [self setScore:bs.score];
104     [gb redrawOriginalState];
105     [gb drawPicesFromBoard:bs];
106 }
107
108 - (void)highlightPiceAt:(BoardField)bf
109 {
110     if(bf.x < 0 || bf.y < 0 || bf.x > boardSize.width - 1 || bf.y >
111         boardSize.height - 1 )
112     {
113         NSError* e = [NSError exceptionWithName:@"Move
114             error." reason:[NSString stringWithFormat:@"The board
115             field (%i,%i) is outside of the board.", bf.x, bf.y]
116             userInfo:nil];
117             @throw e;
118     }
119     [gb highlightPiceAt:bf];
120 }
121
122 - (void)drawOpaquePiceAt:(BoardField)bf forPlayer:(int)player
123 {
124     if(bf.x < 0 || bf.y < 0 || bf.x > boardSize.width - 1 || bf.y >
125         boardSize.height - 1 )
126     {
127         NSError* e = [NSError exceptionWithName:@"Move
128             error." reason:[NSString stringWithFormat:@"The board
129             field (%i,%i) is outside of the board.", bf.x, bf.y]
130             userInfo:nil];
131             @throw e;
132     }
133     [gb drawOpaquePiceAt:bf forPlayer:player];
134 }
135
136 - (void)gameDidEnd:(NSWindow *)sheet returnCode:(int)returnCode
137     contextInfo:(void *)contextInfo // change name to
138     UserResponseToGameOverDialog
139 {
140     if( returnCode == NSAlertAlternateReturn )
141         [NSApp terminate:self];
142     if( returnCode == NSAlertOtherReturn )
143         [ge resetGame];
144     if( returnCode == NSAlertDefaultReturn )
145         // All is fine (do nothing), the Game Engine is disabled.
146     if( returnCode == NSAlertErrorReturn )
147     {
```

```

138     NSException* e = [NSEException exceptionWithName:@"Unknown
139         Error!" reason:@"Invalid return data from Game Did End
140             sheet." userInfo:nil];
141 }
142
143 - (void)setGameEngine:(GameEngine*)engine
144 {
145     ge = engine;
146 }
147
148 - (void)setHighlightState:(BOOL)highlight
149 {
150     doHighlighting = highlight;
151 }
152
153 - (void)setBoardSize:(BoardSize)board
154 {
155     boardSize.height = board.height;
156     boardSize.width = board.width;
157     [gb setBoardSize:board];
158
159     float WindowMaxWidth = 2 * 20 + 128 * boardSize.width;
160     float WindowMaxHeight = 20 + 128 * boardSize.height + 59 ;
161
162     float WindowMinWidth = 2 * 10 + 64 * boardSize.width;
163     float WindowMinHeight = 10 + 64 * boardSize.height + 59 ;
164
165     [[self window] setMaxSize: NSMakeSize( WindowMaxWidth,
166                                         WindowMaxHeight )];
166     [[self window] setMinSize: NSMakeSize( WindowMinWidth,
167                                         WindowMinHeight )];
167
168     NSRect gameWindowFrame = [[self window] frame];
169
170     [gb setDisplayOffset: 20 ];
171     [gb setSquareDim: 128 ];
172     [gb setNeedsDisplay: YES ];
173
174     [[self window] setFrame: NSMakeRect(
175         gameWindowFrame.origin.x,gameWindowFrame.origin.y,
176         WindowMaxWidth, WindowMaxHeight) display:YES];
175 }
176
177 @end

```

C.1.11 GameBoard.h

```

1 // Kolibrat
2 // GameBoard.h
3 //

```

```

4 // Created by Aron Lindberg.
5
6 #import <Cocoa/Cocoa.h>
7 #import "Datastructures.h"
8
9 @interface GameBoard : NSView
10 {
11
12 // Private instance variabels.
13 @private
14     float squareDim;
15     int **HighlightArray;
16     int **PicesArray;
17     BoardSize boardSize;
18     float offset;
19 }
20
21 // Public instance methods.
22 - (id)initWithFrame:(NSRect)frameRect;
23 - (void)drawRect:(NSRect)rect;
24 - (void)mouseDown:(NSEvent*)event;
25 - (void)highlightField:(BoardField)bf;
26 - (void)redrawOriginalState;
27 - (void)drawPicesFromBoard:(GameState)gs;
28 - (void)highlightPiceAt:(BoardField)bf;
29 - (void)drawOpaquePiceAt:(BoardField)bf forPlayer:(int)player;
30 - (void)setBoardSize:(BoardSize)board;
31 - (void)setSquareDim:(float)dim;
32 - (void)setDisplayOffset:(float)distance;
33
34 @end

```

C.1.12 GameBoard.m

```

1 // Kolibrat
2 // GameBoard.m
3 //
4 // Created by Aron Lindberg.
5
6 #import "GameBoard.h"
7
8 // Class variabels
9 NSRect emptyRect;
10 NSImage *blackPice;
11 NSImage *redPice;
12 NSImage *blackPiceHL;
13 NSImage *redPiceHL;
14
15 @implementation GameBoard
16
17 // Internal method used to change the size of the game board.
18 - (void)changeSizeOfBoardArray

```

```
19 {
20     int i;
21
22     HighlightArray = malloc(boardSize.width * sizeof(int *));
23     PicesArray      = malloc(boardSize.width * sizeof(int *));
24
25     for(i = 0 ; i < boardSize.width; i++)
26     {
27         HighlightArray[i] = malloc(boardSize.height * sizeof(int));
28         PicesArray[i]     = malloc(boardSize.height * sizeof(int));
29     }
30
31     int x, y;
32     for ( x = 0 ; x < boardSize.width ; x++ ) {
33         for ( y = 0 ; y < boardSize.height ; y++ ) {
34             HighlightArray[x][y] = EMPTY;
35             PicesArray[x][y] = EMPTY;
36         }
37     }
38 }
39
40 // Called when the GUI loads.
41 - (void)awakeFromNib
42 {
43     // temp boarder size until the player starts a new game.
44     boardSize.width = 3 ;
45     boardSize.height = 4 ;
46
47     [self changeSizeOfBoardArray];
48
49     squareDim = 128 ;
50     offset = 20 ;
51 }
52
53 // interneal method to get rectangels for BoardFields.
54 - (NSRect)RectForField:(BoardField)bf
55 {
56     NSRect field;
57     field.origin = NSMakePoint( bf.x * squareDim + offset, bf.y *
58                                 squareDim );
59     field.size = NSMakeSize( squareDim, squareDim );
60     return field;
61 }
62
63 // internal method to get color of a BoardField.
64 - (void)setColorForField:(BoardField)bf
65 {
66     if ( HighlightArray[bf.x][bf.y] != HIGHLIGHTFIELD )
67         if( bf.x % 2 == bf.y % 2 )
68             [[NSColor lightGrayColor] set];
69         else
70             [[NSColor darkGrayColor] set];
71
72     else if( HighlightArray[bf.x][bf.y] == HIGHLIGHTFIELD )
73         if( bf.x % 2 == bf.y % 2 )
```

```
73             [[NSColor colorWithCalibratedRed: 0.851 green: 0.569
74                         blue: 0.310 alpha: 1.0 ] set];
75         else
76             [[NSColor colorWithCalibratedRed: 0.718 green: 0.435
77                         blue: 0.173 alpha: 1.0 ] set];
78     }
79
80 - (id)initWithFrame:(NSRect)frameRect
81 {
82     if ((self = [super initWithFrame:frameRect]) != nil)
83     {
84         squareDim = ([self bounds].size.width / boardSize.width) ;
85         redPice = [NSImage imageNamed:@"red"];
86         redPiceHL = [NSImage imageNamed:@"redHL"];
87         blackPice = [NSImage imageNamed:@"black"];
88         blackPiceHL = [NSImage imageNamed:@"blackHL"];
89     }
90     return self;
91 }
92 // This method is called everythime Cocoa wants to redraw the GUI.
93 - (void)drawRect:(NSRect)rect
94 {
95     int x; int y;
96     for ( x = 0 ; x < boardSize.width ; x++ ) {
97         for ( y = 0 ; y < boardSize.height ; y++ ) {
98             BoardField bf = makeBoardField( x, y );
99             [self setColorForField: bf];
100            [NSBezierPath fillRect: [self RectForField: bf]];
101
102            if( PicesArray[x][y] == RED )
103                [redPice drawInRect:[self RectForField:
104                                makeBoardField( x, y )]
105                                fromRect:emptyRect
106                                operation:NSCompositeSourceAtop fraction:
107                                1 ];
108            if( PicesArray[x][y] == RED_HIGHLIGHT )
109                [redPiceHL drawInRect:[self RectForField:
110                                makeBoardField( x, y )]
111                                fromRect:emptyRect
112                                operation:NSCompositeSourceAtop
113                                fraction: 1 ];
114            if( PicesArray[x][y] == RED_OPAQUE )
115                [redPice drawInRect:[self RectForField:
116                                makeBoardField( x, y )]
117                                fromRect:emptyRect
118                                operation:NSCompositeSourceAtop fraction:
119                                0.35 ];
120
121            if( PicesArray[x][y] == BLACK )
122                [blackPice drawInRect:[self RectForField:
123                                makeBoardField( x, y )]
124                                fromRect:emptyRect
125                                operation:NSCompositeSourceAtop
126                                fraction: 1 ];
127        }
128    }
129 }
```

```

118                     operation:NSCompositeSourceAtop
119                     fraction: 1 ];
120             if(  PicesArray[x][y] == BLACK_HIGHLIGHT )
121                 [blackPiceHL drawInRect:[self RectForField:
122                     makeBoardField( x, y )]
123                     fromRect:emptyRect
124                     operation:NSCompositeSourceAtop
125                     fraction: 1 ];
126             if(  PicesArray[x][y] == BLACK_OPAQUE )
127                 [blackPice drawInRect:[self RectForField:
128                     makeBoardField( x, y )]
129                     fromRect:emptyRect
130                     operation:NSCompositeSourceAtop
131                     fraction: 0.35 ];
132         }
133     }
134 }
135 // This method is called by NSApplication everytime the mouse is
136 // clicked.
137 - (void)mouseDown:(NSEvent*)event
138 {
139     NSPoint mouse = [self convertPoint: [event locationInWindow]
140                     fromView: nil];
141     int x = -1 ; int y = -1 ;
142     while ( mouse.x - offset > 0 ) {
143         mouse.x -= squareDim;
144         x++;
145     }
146     while ( mouse.y > 0 ) {
147         mouse.y -= squareDim;
148         y++;
149     }
150
151     if( x < 0 || y < 0 || x > boardSize.width -1 || y >
152         boardSize.height -1 )
153         return; // The click was not on the board.
154
155 // Saves the click coordinates as an object.
156 BoardFieldObject *clickCoordinate =
157     [BoardFieldObject boardfieldObjectWithField:
158         makeBoardField(x,y)];
159
160 // Sends a Notification informing other classes of the click.
161 [[NSNotificationCenter defaultCenter] postNotificationName:@"
162     MouseClick" object:clickCoordinate];
163 }
164
165 // Method to highlight a field on the board.
166 - (void)highlightField:(BoardField)bf
167 {
168     HighlightArray[bf.x][bf.y] = HIGHLIGHTFIELD;
169     [self setNeedsDisplay:YES];
170 }
171
172

```

```
163 // Method to restore (unhighlight) the board.
164 - (void)redrawOriginalState
165 {
166     int x; int y;
167     for ( x = 0 ; x < boardSize.width ; x++ ) {
168         for ( y = 0 ; y < boardSize.height ; y++ ) {
169             HighlightArray[x][y] = EMPTY;
170             if( PicesArray[x][y] == RED_HIGHLIGHT )
171                 PicesArray[x][y] = RED;
172             if( PicesArray[x][y] == RED_OPAQUE )
173                 PicesArray[x][y] = EMPTY;
174             if( PicesArray[x][y] == BLACK_HIGHLIGHT )
175                 PicesArray[x][y] = BLACK;
176             if( PicesArray[x][y] == BLACK_OPAQUE )
177                 PicesArray[x][y] = EMPTY;
178         }
179     }
180     [self setNeedsDisplay:YES];
181 }
182
183 - (void)drawPicesFromBoard:(GameState)gs
184 {
185     int x; int y;
186     for ( x = 0 ; x < boardSize.width; x++ ) {
187         for ( y = 0 ; y < boardSize.height ; y++ )
188         {
189             PicesArray[x][y] = BOARD_CONVERTER( gs.board[x][y] );
190         }
191     }
192     [self setNeedsDisplay:YES];
193 }
194
195 - (void)highlightPiceAt:(BoardField)bf
196 {
197     if( PicesArray[bf.x][bf.y] == RED )
198     {
199         PicesArray[bf.x][bf.y] = RED_HIGHLIGHT;
200         [self setNeedsDisplay:YES];
201     }
202     else if( PicesArray[bf.x][bf.y] == BLACK )
203     {
204         PicesArray[bf.x][bf.y] = BLACK_HIGHLIGHT;
205         [self setNeedsDisplay:YES];
206     }
207
208 }
209
210 - (void)drawOpaquePiceAt:(BoardField)bf forPlayer:(int)player
211 {
212     if( player == RED )
213         PicesArray[bf.x][bf.y] = RED_OPAQUE;
214     else if( player == BLACK )
215         PicesArray[bf.x][bf.y] = BLACK_OPAQUE;
216     [self setNeedsDisplay:YES];
217 }
```

```

218
219 - (void)setBoardSize:(BoardSize)board
220 {
221     int i;
222     for(i = 0 ; i < boardSize.width ; i++)
223     {
224         free(HighlightArray[i]);
225         free(PicesArray[i]);
226     }
227     free(HighlightArray);
228     free(PicesArray);
229
230     boardSize.height = board.height;
231     boardSize.width = board.width;
232
233     [self changeSizeOfBoardArray];
234     [self setNeedsDisplay:YES];
235 }
236
237 - (void)setSquareDim:(float)dim
238 {
239     squareDim = dim;
240 }
241
242 - (void)setDisplayOffset:(float)distance
243 {
244     offset = distance;
245     [self setNeedsDisplay:YES];
246 }
247
248 @end

```

C.1.13 NewGameSheetController.h

```

1 // Kolibrat
2 // NewGameSheetController.h
3 //
4 // Created by Aron Lindberg.
5
6 #import "GameController.h"
7 #import "Datastructures.h"
8 #import "HumanPlayer.h"
9 #import "GameEngine.h"
10 #import "BasicAI.h"
11 #import "AdvancedAI.h"
12
13 @interface NewGameSheetController : NSObject
14 {
15 @public
16     IBOutlet NSTextField *blackName;
17     IBOutlet NSPopUpButton *blackType;
18     IBOutlet NSPopUpButton *boardpopUPMenu;

```

```

19     IBOutlet NSTextField *goals;
20     IBOutlet NSTextField *redName;
21     IBOutlet NSPopUpButton *redType;
22     IBOutlet GameController *gc;
23     IBOutlet NSWindow *newGameWindow;
24     IBOutlet NSWindow *gameWindow;
25     IBOutlet NSButton *highlightInGUI;
26     IBOutlet NSTextField *maxPices;
27     IBOutlet NSS stepper *goalsStepper;
28     IBOutlet NSS stepper *piecesStepper;
29
30 @private
31     GameEngine *newEngine;
32     NSMutableArray *playersType;
33     NSMutableDictionary *playerIdentifiers;
34 }
35 - (IBAction)defaultsButton:(id)sender;
36 - (IBAction)startGameButton:(id)sender;
37 - (IBAction)newGame:(id)sender;
38 - (IBAction)restartGame:(id)sender;
39 - (IBAction)cancelButton:(id)sender;
40 - (void)awakeFromNib;
41 - (void)loadPlayers;
42 - (BOOL)validateMenuItem:(NSMenuItem *)item;
43
44 @end

```

C.1.14 NewGameSheetController.m

```

1 // Kolibrat
2 // NewGameSheetController.m
3 //
4 // Created by Aron Lindberg.
5
6 #import "NewGameSheetController.h"
7
8 @implementation NewGameSheetController
9
10 // This method is called when the NIB file is initialised. And is
11 // used to load data into the NSPopUpButtons.
12 -
13 - (void)awakeFromNib
14 {
15     [self loadPlayers];
16
17     [boardpopUPMenu removeAllItems];
18     [boardpopUPMenu addItemWithTitle:@"2x2"];
19     [boardpopUPMenu addItemWithTitle:@"2x4"];
20     [boardpopUPMenu addItemWithTitle:@"3x3"];
21     [boardpopUPMenu addItemWithTitle:@"3x4"];
22     [boardpopUPMenu addItemWithTitle:@"4x5"];
23     [boardpopUPMenu addItemWithTitle:@"5x6"];

```

```
23 [boardpopUPMenu addItemWithTitle:@"9x9"];
24
25 [boardpopUPMenu setTitle:@"3x4"];
26
27 [blackType removeAllItems];
28 [blackType addItemsWithTitles: playersType];
29
30 [redType removeAllItems];
31 [redType addItemsWithTitles: playersType];
32 }
33
34 // This method is called when the button "Default Options" is
35 // clicked.
36 - (IBAction)defaultsButton:(id)sender
37 {
38     [boardpopUPMenu setTitle:@"3x4"];
39     [goals setIntValue: 5 ];
40     [highlightInGUI setState: 1 ];
41     [maxPices setIntValue: 4 ];
42     [goalsStepper setIntValue: 5 ];
43     [piecesStepper setIntValue: 4 ];
44 }
45
46 // This method is called when the "Start Game" button is clicked.
47 - (IBAction)startGameButton:(id)sender
48 {
49     // Initialises the red and black player.
50     NSDictionary *playerClassTypes = playerIdentefiers;
51
52     // Gets the types of both players as a string.
53     NSString *redTypeTitel = [[redType selectedItem]title];
54     NSString *blackTypeTitel = [[blackType selectedItem]title];
55
56     // Gets the class og both players.
57     Class redClass = [playerClassTypes objectForKey:redTypeTitel];
58     Class blackClass = [playerClassTypes objectForKey:
59                         blackTypeTitel];
60
61     // Gets the name og both players.
62     NSString *nameOfRed = [redName stringValue];
63     NSString *nameOfBlack = [blackName stringValue];
64
65     // Array with the size of the game board.
66     NSArray *boardSizeArray = [[[boardpopUPMenu selectedItem] title
67                                 ]
68                               componentsSeparatedByString:@"x"];
69
70     // BoardSize instance that contains the size of the choosen
71     // board.
72     BoardSize board = makeBoardSize( [[boardSizeArray objectAtIndex: 0 ]
73                                     intValue], [[boardSizeArray objectAtIndex: 1 ]
74                                     intValue] );
75
76     // Initialises both players.
```

```
71     id red = [[redClass alloc] initAsPlayer:PLAYER_RED withName:
72                 nameOfRed boardSize:board picesOnboard:[maxPices intValue]
73                 goalsToWin:[goals intValue]];
74     id black = [[blackClass alloc] initAsPlayer:PLAYER_BLACK
75                  withName: nameOfBlack boardSize:board picesOnboard:[
76                  maxPices intValue] goalsToWin:[goals intValue]];
77
78
79 // Sends the Highlight Option and board size to the GUI.
80 [gc setHighlightState: [highlightInGUI state]];
81 [gc setBoardSize: board ];
82
83 // Initialises the GameEngine with the red and black player.
84 newEngine = [[GameEngine alloc] initWithPlayersRed:red
85                                         andBlack:black
86                                         goalsToWin:[goals
87                                         intValue]
88                                         GameBoardDim:board
89                                         MaxPices:[maxPices
90                                         intValue]
91                                         connectToGUI:gc];
92
93 // Connects the players to the GameEngine.
94 [red setGameEngine:newEngine];
95 [black setGameEngine:newEngine];
96
97 // Connects the GUI to the GameEngine.
98 [gc setGameEngine:newEngine];
99
100 // Removes the newGame sheet so the game can begin.
101 [newGameWindow orderOut:self];
102 [NSApp endSheet:newGameWindow];
103 }
104
105 // Loads all players written al plugins, and the human player.
106 - (void)loadPlayers
107 {
108     playersType = [[NSMutableArray alloc] init];
109     playerIdentifiers = [NSMutableDictionary dictionaryWithCapacity
110                           : 20 ];
111
112     [playerIdentifiers retain];
113
114     if( [[HumanPlayer class] conformsToProtocol:@protocol(
115                                         Player_Protocol )])
116     {
117         [playersType addObject: [ HumanPlayer playerType ]];
```

```

116      [playerIdentefiers setObject: [ HumanPlayer class ] forKey
117          :[ HumanPlayer playerType ]];
118    }
119
120    if( [[BasicAI class] conformsToProtocol:@protocol(
121        Player_Protocol )])
122    {
123        [playersType addObject: [ BasicAI playerType ]];
124        [playerIdentefiers setObject: [ BasicAI class ] forKey:[
125            BasicAI playerType ]];
126    }
127
128    if( [[AdvancedAI class] conformsToProtocol:@protocol(
129        Player_Protocol )])
130    {
131        [playersType addObject: [ AdvancedAI playerType ]];
132        [playerIdentefiers setObject: [ AdvancedAI class ] forKey:[
133            AdvancedAI playerType ]];
134    }
135
136    // Temp varibales to store data for loading the players.
137    NSString *currPath;
138    NSBundle *currBundle;
139    Class currPrincipalClass;
140    NSMutableArray *bundlePaths = [ NSMutableArray array ];
141   NSEnumerator *searchPathEnum;
142    NSMutableArray *bundleSearchPaths = [ NSMutableArray array ];
143    NSMutableArray *allBundles = [ NSMutableArray array ];
144    NSArray *librarySearchPaths;
145
146    librarySearchPaths = NSSearchPathForDirectoriesInDomains(
147        NSSearchPathDirectory, NSAllDomainsMask - NSSystemDomainMask,
148        YES );
149
150    searchPathEnum = [librarySearchPaths objectEnumerator];
151
152    while(currPath = [searchPathEnum nextObject])
153    {
154        [bundleSearchPaths addObject: [currPath
155            stringByAppendingPathComponent:@"Application Support/
156            Kolibrat/PlugIns"]];
157
158        [bundleSearchPaths addObject:[[NSBundle mainBundle]
159            builtInPlugInsPath]];
160
161        searchPathEnum = [bundleSearchPaths objectEnumerator];
162        while(currPath = [searchPathEnum nextObject])
163        {
164            NSDirectoryEnumerator *bundleEnum;
165            NSString *currBundlePath;
166            bundleEnum = [[NSFileManager defaultManager]
167                enumeratorAtPath:currPath];
168            if(bundleEnum)
169            {

```

```
160         while(currBundlePath = [bundleEnum nextObject])
161     {
162         if([[currBundlePath pathExtension] isEqualToString:
163             @"bundle"])
164         {
165             [allBundles addObject:[currPath
166                 stringByAppendingString:
167                     currBundlePath]];
168         }
169     }
170     [bundlePaths addObject: allBundles];
171     NSEnumerator *pathEnum = [bundlePaths objectEnumerator];
172
173     while(currPath = [pathEnum nextObject])
174     {
175         currBundle = [NSBundle bundleWithPath:currPath];
176         if( currBundle )
177         {
178             currPrincipalClass = [currBundle principalClass];
179             if(currPrincipalClass)
180             {
181                 if( [currPrincipalClass conformsToProtocol:
182                     @protocol( Player_Protocol )])
183                 {
184                     [playersType addObject: [currPrincipalClass
185                         playerType]];
186                     [playerIdentifiers setObject:
187                         currPrincipalClass forKey:[
188                             currPrincipalClass playerType ]];
189                 }
190             }
191         }
192 // This method controls wheter or not the menu items are avaible
193 // or not. Used to disable "Restart Game" until a game has been
194 // started.
195 - (BOOL)validateMenuItem:(NSMenuItem *)item
196 {
197     NSString *name = @"Restart Game";
198
199     if( [name isEqualToString: [item title]] && newEngine != nil )
200         return TRUE;
201     else if( [name isEqualToString: [item title]] )
202         return FALSE;
203 }
204 }
```

```

205 // This method is called when the user chooses "New Game" from the
206 // menu.
207 - (IBAction)newGame:(id)sender;
208 {
209     [gameWindow setIsVisible:YES]; // Shows the game window.
210
211     [newEngine delayNextPlayer:TRUE];
212
213     [NSApp beginSheet:newGameWindow // shows the new game options
214      sheet.
215      modalForWindow:gameWindow
216      modalDelegate:nil
217      didEndSelector:nil
218      contextInfo:nil];
219 }
220
221 // This method is called when the user chooses "Restart Game" from
222 // the menu.
223 - (IBAction)restartGame:(id)sender
224 {
225     [newEngine resetGame];
226 }
227
228 - (IBAction)cancelButton:(id)sender
229 {
230     [newGameWindow orderOut:self];
231     [NSApp endSheet:newGameWindow];
232     [newEngine delayNextPlayer:FALSE];
233 }
234
235 @end

```

C.1.15 GUIProtocol.h

```

1 // Kolibrat
2 // GUI_Protocol.h
3 //
4 // Created by Aron Lindberg.
5
6 @protocol GUI_Protocol
7 - (void)gameOverWithWinner:(NSString *)playerName;
8 - (void)highlightField:(BoardField)boardField;
9 - (void)redrawOriginalState;
10 - (void)updateToState:(GameState)gameState;
11 - (void)highlightPiceAt:(BoardField)boardField;
12 - (void)drawOpaquePiceAt:(BoardField)boardField forPlayer:(int)
13     player;
14
15 @end

```

C.1.16 PlayerProtocol.h

```

1 //  Kolibrat
2 //  Player_Protocol.h
3 //
4 //  Created by Aron Lindberg.
5
6 @protocol Player_Protocol
7 - (id)initAsPlayer:(int)player withName:(NSString *)playerName
     boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(int)maxGoals;
8 - (void)startNewTurn;
9 - (void)setGameEngine:(id)ge;
10 - (void)reset;
11 - (NSString *)playerName;
12 + (NSString *)playerType;
13 @end

```

C.1.17 main.m

```

1 //  Kolibrat
2 //  main.m
3 //
4 //  Created by Aron Lindberg.
5
6 #import <Cocoa/Cocoa.h>
7
8 int main(int argc, char *argv[])
9 {
10     return NSApplicationMain(argc, (const char **) argv);
11 }

```

C.2 Kolibrat Test Source Code

C.2.1 Kolibrat Test.m

```

1 //  Kolibrat Test
2 //  Kolibrat Test.m
3 //
4 //  Created by Aron Lindberg.
5
6 #import <Foundation/Foundation.h>
7 #import "Datastructures.h"
8 #import "GameLogic.h"
9
10 int main (int argc, const char * argv[])
11 {
12     NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
13

```

```
14 int maxPices = 4 ;
15 int goalsToWin = 5 ;
16 BoardSize size = makeBoardSize( 4 , 3 );
17
18 BOOL testPassed = TRUE;
19
20 GameLogic *logic = [[GameLogic alloc] initWithMaxPices:maxPices
21 goalsToWin:goalsToWin boardSize:size];
22
23 NSLog(@"GameLogic instance created.");
24 NSLog(@"Max pixels on board is: %i.", maxPices);
25 NSLog(@"Number of goals to win is: %i.", goalsToWin);
26 NSLog(@"Board dimensions (%i,%i).",size.height, size.width );
27
28 NSLog(@"Starting test 1: Insert pice for red in (1,0) on an
29 empty board.");
30
31 GameState state1 = [logic CreateNewGameState];
32
33 if( [logic makeMove:makeMoveFromInt( 1 , 0 , 1 , 0 ) withState
34 :&state1] != TRUE )
35 {
36     NSLog(@"Error the GameLogic rejected the move.");
37     testPassed = FALSE;
38 }
39 if( state1.redPicesOnBoard != 1 || state1.blackPicesOnBoard !=
40 0 )
41 {
42     NSLog(@"Error in number of pices on the board.");
43     testPassed = FALSE;
44 }
45 if( state1.playerMoving != PLAYER_BLACK )
46 {
47     NSLog(@"Error black player don't is not the new moving
48          player.");
49     testPassed = FALSE;
50 }
51 if( state1.board[ 1 ][ 0 ].occupiedByRed != TRUE ||
52     state1.board[ 1 ][ 0 ].occupiedByBlack != FALSE )
53 {
54     NSLog(@"Error red don't have a piece in (1,0).");
55     testPassed = FALSE;
56 }
57 if( testPassed == TRUE )
58     NSLog(@"Test 1 passed.");
59
60 NSLog(@"Starting test 2: Insert pice for black in (1,3) on an
61 empty board.");
62
63 GameState state2 = [logic CreateNewGameState];
64 state2.playerMoving = PLAYER_BLACK;
65 testPassed = TRUE;
```

```
61     if( [logic makeMove:makeMoveFromInt( 1 , 3 , 1 , 3 ) withState
62           :&state2] != TRUE )
63     {
64         NSLog(@"Error the GameLogic rejected the move.");
65         testPassed = FALSE;
66     }
67
68     if( state2.redPicesOnBoard != 0 || state2.blackPicesOnBoard !=
69         1 )
70     {
71         NSLog(@"Error in number of pices on the board.");
72         testPassed = FALSE;
73     }
74     if( state2.playerMoving != PLAYER_RED )
75     {
76         NSLog(@"Error red player don't is not the new moving
77               player.");
78         testPassed = FALSE;
79     }
80     if( state2.board[ 1 ][ 3 ].occupiedByRed != FALSE ||
81         state2.board[ 1 ][ 3 ].occupiedByBlack != TRUE )
82     {
83         NSLog(@"Error black don't have a piece in (1,3).");
84         testPassed = FALSE;
85     }
86
87     if( testPassed == TRUE )
88         NSLog(@"Test 2 passed.");
89
90     GameState state3 = [logic CreateNewGameState];
91     state3.redPicesOnBoard = 4 ;
92     testPassed = TRUE;
93
94     if( [logic makeMove:makeMoveFromInt( 1 , 0 , 1 , 0 ) withState
95           :&state3] != FALSE)
96     {
97         NSLog(@"Error the illegal move was not rejected by
98               GameLogic.");
99         testPassed = FALSE;
100    }
101
102    if( state3.redPicesOnBoard != 4 || state3.blackPicesOnBoard !=
103        0 )
104    {
105        NSLog(@"Error in number of pices on the board.");
106        testPassed = FALSE;
107    }
108
109    if( state3.playerMoving != PLAYER_RED )
110    {
111        NSLog(@"Error black player is the new moving player, it
112              shuld be red.");
113        testPassed = FALSE;
114    }
115
116    if( testPassed == TRUE )
117        NSLog(@"Test 3 passed.");
118
119    if( testPassed == TRUE )
120        NSLog(@"All tests passed successfully.");
121
122    return testPassed;
123}
```

```
107     if( state3.board[ 1 ][ 0 ].occupiedByRed != FALSE ||
108         state3.board[ 1 ][ 0 ].occupiedByBlack != FALSE )
109     {
110         NSLog(@"Error red have a piece in (1,0).");
111         testPassed = FALSE;
112     }
113     if( testPassed == TRUE )
114         NSLog(@"Test 3 passed, another piece was not added to the
115             board.");
116
117     GameState state4 = [logic CreateNewGameState];
118
119     testPassed = TRUE;
120
121     if( [logic makeMove:makeMoveFromInt( 2 , 2 , 2 , 2 ) withState
122          :&state4] )
123     {
124         NSLog(@"Error the illegal move was not rejected by
125             GameLogic.");
126         testPassed = FALSE;
127     }
128     if( state4.redPicesOnBoard != 0 || state4.blackPicesOnBoard !=
129         0 )
130     {
131         NSLog(@"Error in number of pices on the board.");
132         testPassed = FALSE;
133     }
134     if( state4.playerMoving != PLAYER_RED )
135     {
136         NSLog(@"Error black player is the new moving player, it
137             shuld be red.");
138         testPassed = FALSE;
139     }
140     if( state4.board[ 2 ][ 2 ].occupiedByRed != FALSE ||
141         state4.board[ 2 ][ 2 ].occupiedByBlack != FALSE )
142     {
143         NSLog(@"Error red have a piece in (2,2).");
144         testPassed = FALSE;
145     }
146     if( testPassed == TRUE )
147         NSLog(@"Test 4 passed, the piece was not added to the
148             board.");
149
150     NSLog(@"Starting test 5: Score point for red.");
151
152     GameState state5 = [logic CreateNewGameState];
153
154     testPassed = TRUE;
155
156     state5.redPicesOnBoard = 1;
157     state5.board[ 1 ][ 3 ].occupiedByRed = TRUE;
```

```
153
154     if( [logic makeMove:makeMoveFromInt( 1 , 3 , 1 , 3 ) withState
155         :&state5] != TRUE )
156     {
157         NSLog(@"Error the move was rejected by GameLogic.");
158         testPassed = FALSE;
159     }
160     if( state5.redPicesOnBoard != 0 || state5.blackPicesOnBoard !=
161         0 )
162     {
163         NSLog(@"Error in number of pices on the board.");
164         testPassed = FALSE;
165     }
166     if( state5.playerMoving != PLAYER_BLACK )
167     {
168         NSLog(@"Error black player not is the new moving player.");
169         testPassed = FALSE;
170     }
171     if( state5.board[ 1 ][ 3 ].occupiedByRed != FALSE ||
172         state5.board[ 1 ][ 3 ].occupiedByBlack != FALSE )
173     {
174         NSLog(@"Error red have a piece in (1,3), it shuld be gone."
175             );
176         testPassed = FALSE;
177     }
178
179     if( state5.score.red != 1 || state5.score.black != 0 )
180     {
181         NSLog(@"Error in game score.");
182         testPassed = FALSE;
183     }
184
185     if( testPassed == TRUE )
186         NSLog(@"Test 5 passed.");
187
188     NSLog(@"Starting test 6: Score point for black.");
189
190     GameState state6 = [logic CreateNewGameState];
191
192     testPassed = TRUE;
193
194     state6.blackPicesOnBoard = 1 ;
195     state6.board[ 1 ][ 0 ].occupiedByBlack = TRUE;
196     state6.playerMoving = PLAYER_BLACK;
197
198     if( [logic makeMove:makeMoveFromInt( 1 , 0 , 1 , 0 ) withState
199         :&state6] != TRUE )
200     {
201         NSLog(@"Error the move was rejected by GameLogic.");
202         testPassed = FALSE;
203     }
204     if( state6.redPicesOnBoard != 0 || state6.blackPicesOnBoard !=
205         0 )
206     {
207         NSLog(@"Error in number of pices on the board.");
```

```

202         testPassed = FALSE;
203     }
204     if( state6.playerMoving != PLAYER_RED )
205     {
206         NSLog(@"Error red player not is the new moving player.");
207         testPassed = FALSE;
208     }
209     if( state6.board[ 1 ][ 0 ].occupiedByRed != FALSE ||
210         state6.board[ 1 ][ 0 ].occupiedByBlack != FALSE )
211     {
212         NSLog(@"Error black have a piece in (1,0), it shuld be
213             gone.");
214         testPassed = FALSE;
215     }
216     if( state6.score.red != 0 || state6.score.black != 1 )
217     {
218         NSLog(@"Error in game score.");
219         testPassed = FALSE;
220     }
221     if( testPassed == TRUE )
222         NSLog(@"Test 6 passed.");
223
224     NSLog(@"Starting test 7: Try to insert piece in occupied field.");
225
226     GameState state7 = [logic CreateNewGameState];
227
228     testPassed = TRUE;
229
230     state7.blackPicesOnBoard = 1 ;
231     state7.board[ 1 ][ 0 ].occupiedByBlack = TRUE;
232
233     if( [logic makeMove:makeMoveFromInt( 1 , 0 , 1 , 0 ) withState
234         :&state7] != FALSE )
235     {
236         NSLog(@"Error the illegal move was not rejected by
237             GameLogic.");
238         testPassed = FALSE;
239     }
240     if( state7.redPicesOnBoard != 0 || state7.blackPicesOnBoard !=
241         1 )
242     {
243         NSLog(@"Error in number of pices on the board.");
244         testPassed = FALSE;
245     }
246     if( state7.playerMoving != PLAYER_RED )
247     {
248         NSLog(@"Error in the next player to move.");
249         testPassed = FALSE;
250     }
251     if( state7.board[ 1 ][ 0 ].occupiedByRed != FALSE ||
252         state7.board[ 1 ][ 0 ].occupiedByBlack != TRUE )
253     {

```

```
250      NSLog(@"Error don't black have a piece in (1,0).");
251      testPassed = FALSE;
252  }
253
254  if( state7.score.red != 0 || state7.score.black != 0 )
255  {
256      NSLog(@"Error in game score.");
257      testPassed = FALSE;
258  }
259
260  if( testPassed == TRUE )
261      NSLog(@"Test 7 passed.");
262
263  NSLog(@"Starting test 8: Gamestatus is changed when red wins.");
264  ;
265
266  GameState state8 = [logic CreateNewGameState];
267
268  testPassed = TRUE;
269
270  state8.redPicesOnBoard = 1 ;
271  state8.board[ 1 ][ 3 ].occupiedByRed = TRUE;
272  state8.score.red = 4 ;
273
274  if( [logic makeMove:makeMoveFromInt( 1 , 3 , 1 , 3 ) withState
275      :&state8] != TRUE )
276  {
277      NSLog(@"Error the move was rejected by GameLogic.");
278      testPassed = FALSE;
279  }
280  if( state8.redPicesOnBoard != 0 || state8.blackPicesOnBoard !=
281      0 )
282  {
283      NSLog(@"Error in number of pices on the board.");
284      testPassed = FALSE;
285  }
286
287  if( state8.board[ 1 ][ 3 ].occupiedByRed != FALSE ||
288      state8.board[ 1 ][ 3 ].occupiedByBlack != FALSE )
289  {
290      NSLog(@"Error the field (1,3) is not empty.");
291      testPassed = FALSE;
292  }
293
294  if( state8.score.red != 5 || state8.score.black != 0 )
295  {
296      NSLog(@"Error in game score.");
297      testPassed = FALSE;
298  }
299
300  if( state8.gameStatus.gameOver != TRUE ||
301      state8.gameStatus.winner != PLAYER_RED )
302  {
303      NSLog(@"Error in game status, red is not the winner.");
304      testPassed = FALSE;
305  }
```

```
300    }
301    if( testPassed == TRUE )
302        NSLog(@"Test 8 passed.");
303
304    NSLog(@"Starting test 9: Ensure that no players can move when
305          the game is over.");
306    testPassed = TRUE;
307
308    state8.redPicesOnBoard = 1 ;
309    state8.board[ 0 ][ 0 ].occupiedByRed = TRUE;
310
311    state8.blackPicesOnBoard = 1 ;
312    state8.board[ 2 ][ 3 ].occupiedByBlack = TRUE;
313
314    state8.playerMoving = PLAYER_RED;
315
316    if( [logic makeMove:makeMoveFromInt( 0 , 0 , 1 , 1 ) withState
317         :&state8] != FALSE )
318    {
319        NSLog(@"Error the illegal move by red was not rejected by
320              GameLogic.");
321        testPassed = FALSE;
322    }
323
324    state8.playerMoving = PLAYER_BLACK;
325
326    if( [logic makeMove:makeMoveFromInt( 2 , 3 , 1 , 2 ) withState
327         :&state8] != FALSE )
328    {
329        NSLog(@"Error the illegal move by black was not rejected by
330              GameLogic.");
331        testPassed = FALSE;
332    }
333
334    if( testPassed == TRUE )
335        NSLog(@"Test 9 passed.");
336
337    NSLog(@"Starting test 10: Ensure that no players can move
338          outside of the board.");
339
340    testPassed = TRUE;
341    GameState state10 = [logic CreateNewGameState];
342
343    state10.redPicesOnBoard = 1 ;
344    state10.board[ 0 ][ 0 ].occupiedByRed = TRUE;
345
346    state10.blackPicesOnBoard = 1 ;
347    state10.board[ 2 ][ 3 ].occupiedByBlack = TRUE;
348
349    @try {
350
351
352        if( [logic makeMove:makeMoveFromInt( 0 , 0 , -1 , 1 )
353             withState:&state10] != FALSE )
```

```
348         {
349             NSLog(@"Error the illegal move by red was not rejected
350                   by GameLogic.");
350             testPassed = FALSE;
351         }
352     }
353     @catch ( NSError *e ) {
354
355         NSLog(@"GameLogic threw an exception, this is ok.");
356     }
357
358     state10.playerMoving = PLAYER_BLACK;
359
360     @try {
361         if( [logic makeMove:makeMoveFromInt( 2 , 3 , 1 , 4 )
361           withState:&state10] != FALSE )
362         {
363             NSLog(@"Error the illegal move by black was not
364                   rejected by GameLogic.");
364             testPassed = FALSE;
365         }
366     }
367     @catch ( NSError *e ) {
368
369         NSLog(@"GameLogic threw an exception, this is ok.");
370     }
371
372     if( testPassed == TRUE )
373         NSLog(@"Test 10 passed.");
374
375     NSLog(@"Starting test 11: Test of moves on a non empty board 1.
376           ");
377
378     GameState state11 = [logic CreateNewGameState];
379
380     state11.redPicesOnBoard = 1 ;
381     state11.blackPicesOnBoard = 3 ;
382
383     state11.board[ 1 ][ 0 ].occupiedByRed = TRUE;
384
385     state11.board[ 1 ][ 1 ].occupiedByBlack = TRUE;
386
387     state11.board[ 1 ][ 2 ].occupiedByBlack = TRUE;
388     state11.board[ 1 ][ 3 ].occupiedByBlack = TRUE;
389
390     NSSet *allmoves = [logic allLegalMoves:&state11];
391
392     if( [allmoves count] != 5 )
393     {
394         NSLog(@"Error in the number of allowed moves.");
395         testPassed = FALSE;
396     }
397
```

```
398 if( [allmoves containsObject:[MoveObject moveObjectWithMove:
399     makeMoveFromInt( 0 , 0 , 0 , 0 )]] != TRUE )
400 {
401     NSLog(@"Error the move to insert a piece in (0,0) was not
402         allowed.");
403     testPassed = FALSE;
404 }
405 if( [allmoves containsObject:[MoveObject moveObjectWithMove:
406     makeMoveFromInt( 2 , 0 , 2 , 0 )]] != TRUE )
407 {
408     NSLog(@"Error the move to insert a piece in (2,0) was not
409         allowed.");
410     testPassed = FALSE;
411 }
412 if( [allmoves containsObject:[MoveObject moveObjectWithMove:
413     makeMoveFromInt( 1 , 0 , 0 , 1 )]] != TRUE )
414 {
415     NSLog(@"Error the move from (1,0) to (0,1) was not allowed.
416         ");
417     testPassed = FALSE;
418 }
419 if( [allmoves containsObject:[MoveObject moveObjectWithMove:
420     makeMoveFromInt( 1 , 0 , 2 , 1 )]] != TRUE )
421 {
422     NSLog(@"Error the move from (1,0) to (2,1) was not allowed.
423         ");
424     testPassed = FALSE;
425 }
426 if( testPassed == TRUE )
427 {
428     NSLog(@"Test 11 passed.");
429 }
430 NSLog(@"Starting test 12: Test of moves on a non empty board 2.
431         ");
432 testPassed = TRUE;
433 GameState state12 = [logic CreateNewGameState];
434
435 state12.redPicesOnBoard = 1 ;
436 state12.blackPicesOnBoard = 2 ;
437
438 state12.board[ 2 ][ 0 ].occupiedByRed = TRUE;
439
440 state12.board[ 2 ][ 1 ].occupiedByBlack = TRUE;
441
```

```
442     state12.board[ 2 ][ 2 ].occupiedByBlack = TRUE;
443
444     allmoves = [logic allLegalMoves:&state12];
445
446     if( [allmoves count] != 5 )
447     {
448         NSLog(@"Error in the number of allowed moves.");
449         testPassed = FALSE;
450     }
451
452     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
453         makeMoveFromInt( 0 , 0 , 0 , 0 )]] != TRUE )
454     {
455         NSLog(@"Error the move to insert a piece in (0,0) was not
456             allowed.");
457         testPassed = FALSE;
458     }
459
460     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
461         makeMoveFromInt( 1 , 0 , 1 , 0 )]] != TRUE )
462     {
463         NSLog(@"Error the move to insert a piece in (1,0) was not
464             allowed.");
465         testPassed = FALSE;
466     }
467
468     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
469         makeMoveFromInt( 2 , 0 , 1 , 1 )]] != TRUE )
470     {
471         NSLog(@"Error the move from (2,0) to (1,1) was not allowed.
472             ");
473         testPassed = FALSE;
474     }
475
476     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
477         makeMoveFromInt( 2 , 0 , 2 , 1 )]] != TRUE )
478     {
479         NSLog(@"Error the move from (2,0) to (2,1) was not allowed.
480             ");
481         testPassed = FALSE;
482     }
483
484     if( testPassed == TRUE )
485         NSLog(@"Test 12 passed.");
486
487     NSLog(@"Starting test 13: Test of moves on a non empty board 3.
488             ");
```

```
486
487     testPassed = TRUE;
488     GameState state13 = [logic CreateNewGameState];
489
490     state13.redPicesOnBoard = 3 ;
491     state13.blackPicesOnBoard = 1 ;
492
493     state13.board[ 0 ][ 0 ].occupiedByRed = TRUE;
494
495     state13.board[ 1 ][ 0 ].occupiedByRed = TRUE;
496
497     state13.board[ 2 ][ 0 ].occupiedByRed = TRUE;
498
499     state13.board[ 1 ][ 1 ].occupiedByBlack = TRUE;
500
501     allmoves = [logic allLegalMoves:&state13];
502
503     if( [allmoves count] != 4 )
504     {
505         NSLog(@"Error in the number of allowed moves.");
506         testPassed = FALSE;
507     }
508
509     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
510         makeMoveFromInt( 1 , 0 , 2 , 1 )]] != TRUE )
511     {
512         NSLog(@"Error the move from (1,0) to (2,1) was not allowed.
513             ");
514         testPassed = FALSE;
515     }
516
517     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
518         makeMoveFromInt( 1 , 0 , 0 , 1 )]] != TRUE )
519     {
520         NSLog(@"Error the move from (1,0) to (0,1) was not allowed.
521             ");
522         testPassed = FALSE;
523     }
524
525     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
526         makeMoveFromInt( 1 , 0 , 1 , 1 )]] != TRUE )
527     {
528         NSLog(@"Error the move from (1,0) to (1,1) was not allowed.
529             ");
530         testPassed = FALSE;
531     }
532 }
```

```

533     if( testPassed == TRUE )
534         NSLog(@"Test 13 passed.");
535
536     NSLog(@"Starting test 14: Test of moves on a non empty board 4.
537             ");
538
539     testPassed = TRUE;
540     GameState state14 = [logic CreateNewGameState];
541
542     state14.redPicesOnBoard = 4;
543     state14.blackPicesOnBoard = 1;
544
545     state14.board[0][1].occupiedByRed = TRUE;
546
547     state14.board[1][1].occupiedByRed = TRUE;
548
549     state14.board[2][1].occupiedByRed = TRUE;
550
551     state14.board[1][2].occupiedByRed = TRUE;
552
553     state14.board[1][3].occupiedByBlack = TRUE;
554
555     allmoves = [logic allLegalMoves:&state14];
556
557     if( [allmoves count] != 5 )
558     {
559         NSLog(@"Error in the number of allowed moves.");
560         testPassed = FALSE;
561     }
562
563     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
564         makeMoveFromInt(1,1,2,2)]] != TRUE )
565     {
566         NSLog(@"Error the move from (1,1) to (2,2) was not allowed.
567                 ");
568         testPassed = FALSE;
569     }
570
571     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
572         makeMoveFromInt(1,1,0,2)]] != TRUE )
573     {
574         NSLog(@"Error the move from (1,1) to (0,2) was not allowed.
575                 ");
576         testPassed = FALSE;
577     }
578
579     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
580         makeMoveFromInt(1,2,0,3)]] != TRUE )
581     {
582         NSLog(@"Error the move from (1,2) to (0,3) was not allowed.
583                 ");
584         testPassed = FALSE;
585     }

```

```
580     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
581         makeMoveFromInt(1,2,2,3)]] != TRUE )
582     {
583         NSLog(@"Error the move from (1,2) to (2,3) was not allowed.
584             ");
585         testPassed = FALSE;
586     }
587     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
588         makeMoveFromInt(1,2,1,3)]] != TRUE )
589     {
590         NSLog(@"Error the move from (1,2) to (1,3) was not allowed.
591             ");
592         testPassed = FALSE;
593     }
594     if( testPassed == TRUE )
595         NSLog(@"Test 14 passed.");
596
597     NSLog(@"Starting test 15: Test of moves on a non empty board 5.
598             ");
599     testPassed = TRUE;
600     GameState state15 = [logic CreateNewGameState];
601
602     state15.redPicesOnBoard = 3;
603     state15.blackPicesOnBoard = 2;
604
605     state15.board[0][0].occupiedByRed = TRUE;
606     state15.board[1][0].occupiedByRed = TRUE;
607     state15.board[0][1].occupiedByRed = TRUE;
608     state15.board[0][2].occupiedByBlack = TRUE;
609     state15.board[0][3].occupiedByBlack = TRUE;
610
611     allmoves = [logic allLegalMoves:&state15];
612
613     if( [allmoves count] != 5 )
614     {
615         NSLog(@"Error in the number of allowed moves.");
616         testPassed = FALSE;
617     }
618
619     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
620         makeMoveFromInt(2,0,2,0)]] != TRUE )
621     {
622         NSLog(@"Error the move to insert a piece in (2,0) was not
623             allowed.");
624         testPassed = FALSE;
625     }
626     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
627         makeMoveFromInt(1,0,2,1)]] != TRUE )
628     {
```

```
627         NSLog(@"Error the move from (1,0) to (2,1) was not allowed.  
628             ");  
629         testPassed = FALSE;  
630     }  
631     if( [allmoves containsObject:[MoveObject moveObjectWithMove:  
632         makeMoveFromInt(0,0,1,1)]] != TRUE )  
633     {  
634         NSLog(@"Error the move from (0,0) to (1,1) was not allowed.  
635             ");  
636         testPassed = FALSE;  
637     }  
638     if( [allmoves containsObject:[MoveObject moveObjectWithMove:  
639         makeMoveFromInt(0,1,1,2)]] != TRUE )  
640     {  
641         NSLog(@"Error the move from (0,1) to (1,2) was not allowed.  
642             ");  
643         testPassed = FALSE;  
644     }  
645     if( [allmoves containsObject:[MoveObject moveObjectWithMove:  
646         makeMoveFromInt(0,1,0,2)]] != TRUE )  
647     {  
648         NSLog(@"Error the move from (0,1) to (0,2) was not allowed.  
649             ");  
650         testPassed = FALSE;  
651     }  
652     if( testPassed == TRUE )  
653     {  
654         NSLog(@"Test 15 passed.");  
655  
656     NSLog(@"Starting test 16: Test of moves on a non empty board 6.  
657             ");  
658     testPassed = TRUE;  
659     GameState state16 = [logic CreateNewGameState];  
660  
661     state16.redPicesOnBoard = 3;  
662     state16.blackPicesOnBoard = 1;  
663  
664     state16.board[1][0].occupiedByRed = TRUE;  
665     state16.board[1][1].occupiedByRed = TRUE;  
666     state16.board[1][2].occupiedByRed = TRUE;  
667     state16.board[1][3].occupiedByBlack = TRUE;  
668  
669     state16.playerMoving = PLAYER_BLACK;  
670  
671     allmoves = [logic allLegalMoves:&state16];  
672     if( [allmoves count] != 5 )  
673     {  
674         NSLog(@"Error in the number of allowed moves.");
```

```
674     testPassed = FALSE;
675 }
676
677 if( [allmoves containsObject:[MoveObject moveObjectWithMove:
678     makeMoveFromInt(0,3,0,3)]] != TRUE )
679 {
680     NSLog(@"Error the move to insert a piece in (0,3) was not
681         allowed.");
682     testPassed = FALSE;
683 }
684
685 if( [allmoves containsObject:[MoveObject moveObjectWithMove:
686     makeMoveFromInt(2,3,2,3)]] != TRUE )
687 {
688     NSLog(@"Error the move to insert a piece in (2,3) was not
689         allowed.");
690     testPassed = FALSE;
691 }
692
693 if( [allmoves containsObject:[MoveObject moveObjectWithMove:
694     makeMoveFromInt(1,3,0,2)]] != TRUE )
695 {
696     NSLog(@"Error the move from (1,3) to (0,2) was not allowed.
697             ");
698     testPassed = FALSE;
699 }
700
701 if( [allmoves containsObject:[MoveObject moveObjectWithMove:
702     makeMoveFromInt(1,3,2,2)]] != TRUE )
703 {
704     NSLog(@"Error the move from (1,3) to (2,2) was not allowed.
705             ");
706     testPassed = FALSE;
707 }
708
709 if( testPassed == TRUE )
710     NSLog(@"Test 16 passed.");
711
712 testPassed = TRUE;
713 GameState state17 = [logic CreateNewGameState];
714
715 state17.redPicesOnBoard = 2;
716 state17.blackPicesOnBoard = 1;
717
```

```
718     state17.board[0][1].occupiedByRed = TRUE;
719
720     state17.board[0][2].occupiedByRed = TRUE;
721
722     state17.board[0][3].occupiedByBlack = TRUE;
723
724     state17.playerMoving = PLAYER_BLACK;
725
726     allmoves = [logic allLegalMoves:&state17];
727
728     if( [allmoves count] != 5 )
729     {
730         NSLog(@"Error in the number of allowed moves.");
731         testPassed = FALSE;
732     }
733
734
735     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
736                               makeMoveFromInt(1,3,1,3)]] != TRUE )
737     {
738         NSLog(@"Error the move to insert a piece in (1,3) was not
739               allowed.");
740         testPassed = FALSE;
741     }
742
743     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
744                               makeMoveFromInt(2,3,2,3)]] != TRUE )
745     {
746         NSLog(@"Error the move to insert a piece in (2,3) was not
747               allowed.");
748         testPassed = FALSE;
749     }
750
751     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
752                               makeMoveFromInt(0,3,0,2)]] != TRUE )
753     {
754         NSLog(@"Error the move from (0,3) to (0,2) was not allowed.
755               ");
756         testPassed = FALSE;
757     }
758
759     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
760                               makeMoveFromInt(0,3,1,2)]] != TRUE )
761     {
762         NSLog(@"Error the move from (0,3) to (1,2) was not allowed.
763               ");
764         testPassed = FALSE;
765     }
766
767     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
768                               makeMoveFromInt(0,3,0,0)]] != TRUE )
769     {
770         NSLog(@"Error the move from (0,3) to (0,0) was not allowed.
771               ");
772         testPassed = FALSE;
773     }
774 }
```

```
763     }
764
765     if( testPassed == TRUE )
766         NSLog(@"Test 17 passed.");
767
768     NSLog(@"Starting test 18: Test of moves on a non empty board 8.
769     ");
770
771     testPassed = TRUE;
772     GameState state18 = [logic CreateNewGameState];
773
774     state18.redPicesOnBoard = 1;
775     state18.blackPicesOnBoard = 3;
776
777     state18.board[1][2].occupiedByRed = TRUE;
778
779     state18.board[0][3].occupiedByBlack = TRUE;
780     state18.board[1][3].occupiedByBlack = TRUE;
781     state18.board[2][3].occupiedByBlack = TRUE;
782
783     state18.playerMoving = PLAYER_BLACK;
784
785     allmoves = [logic allLegalMoves:&state18];
786
787     if( [allmoves count] != 4 )
788     {
789         NSLog(@"Error in the number of allowed moves.");
790         testPassed = FALSE;
791     }
792
793     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
794         makeMoveFromInt(1,3,0,2)]] != TRUE )
795     {
796         NSLog(@"Error the move from (1,3) to (0,2) was not allowed.
797                 ");
798         testPassed = FALSE;
799     }
800
801     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
802         makeMoveFromInt(1,3,2,2)]] != TRUE )
803     {
804         NSLog(@"Error the move from (1,3) to (2,2) was not allowed.
805                 ");
806         testPassed = FALSE;
807     }
808
809     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
810         makeMoveFromInt(1,3,1,2)]] != TRUE )
811     {
812         NSLog(@"Error the move from (1,3) to (1,2) was not allowed.
813                 ");
814         testPassed = FALSE;
815     }
```

```
810     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
811             makeMoveFromInt(1,3,1,1)]] != TRUE )
812     {
813         NSLog(@"Error the move from (1,3) to (1,1) was not allowed.
814             ");
815         testPassed = FALSE;
816     }
817     if( testPassed == TRUE )
818         NSLog(@"Test 18 passed.");
819     NSLog(@"Starting test 19: Test of moves on a non empty board 9.
820             ");
821     testPassed = TRUE;
822     GameState state19 = [logic CreateNewGameState];
823
824     state19.redPicesOnBoard = 1;
825     state19.blackPicesOnBoard = 4;
826
827     state19.board[1][0].occupiedByRed = TRUE;
828
829     state19.board[0][2].occupiedByBlack = TRUE;
830     state19.board[1][2].occupiedByBlack = TRUE;
831     state19.board[2][2].occupiedByBlack = TRUE;
832
833     state19.board[1][1].occupiedByBlack = TRUE;
834
835     state19.playerMoving = PLAYER_BLACK;
836
837     allmoves = [logic allLegalMoves:&state19];
838
839     if( [allmoves count] != 5 )
840     {
841         NSLog(@"Error in the number of allowed moves.");
842         testPassed = FALSE;
843     }
844
845     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
846             makeMoveFromInt(1,1,0,0)]] != TRUE )
847     {
848         NSLog(@"Error the move from (1,1) to (0,0) was not allowed.
849             ");
850         testPassed = FALSE;
851     }
852
853     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
854             makeMoveFromInt(1,1,1,0)]] != TRUE )
855     {
856         NSLog(@"Error the move from (1,1) to (1,0) was not allowed.
857             ");
858         testPassed = FALSE;
859     }
860 }
```

```
857     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
858             makeMoveFromInt(1,1,2,0)]] != TRUE )
859     {
860         NSLog(@"Error the move from (1,1) to (2,0) was not allowed.
861             ");
862         testPassed = FALSE;
863     }
864
865     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
866             makeMoveFromInt(1,2,0,1)]] != TRUE )
867     {
868         NSLog(@"Error the move from (1,2) to (0,1) was not allowed.
869             ");
870         testPassed = FALSE;
871     }
872
873     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
874             makeMoveFromInt(1,2,2,1)]] != TRUE )
875     {
876         NSLog(@"Error the move from (1,2) to (2,1) was not allowed.
877             ");
878         testPassed = FALSE;
879     }
880
881     if( testPassed == TRUE )
882         NSLog(@"Test 19 passed.");
883
884     NSLog(@"Starting test 20: Test of moves on a non empty board
885             10.");
886
887     testPassed = TRUE;
888     GameState state20 = [logic CreateNewGameState];
889
890     state20.redPicesOnBoard = 2;
891     state20.blackPicesOnBoard = 3;
892
893     state20.board[0][0].occupiedByRed = TRUE;
894     state20.board[0][1].occupiedByRed = TRUE;
895
896     state20.board[0][2].occupiedByBlack = TRUE;
897     state20.board[0][3].occupiedByBlack = TRUE;
898     state20.board[1][3].occupiedByBlack = TRUE;
899
900     state20.playerMoving = PLAYER_BLACK;
901
902     allmoves = [logic allLegalMoves:&state20];
903
904     if( [allmoves count] != 5 )
905     {
906         NSLog(@"Error in the number of allowed moves.");
907         testPassed = FALSE;
908     }
909
910     if( [allmoves containsObject:[MoveObject moveObjectWithMove:
911             makeMoveFromInt(2,3,2,3)]] != TRUE )
```

```

904      {
905          NSLog(@"Error the move to insert a piece in (2,3) was not
906                  allowed.");
907          testPassed = FALSE;
908      }
909
910      if( [allmoves containsObject:[MoveObject moveObjectWithMove:
911                      makeMoveFromInt(1,3,2,2)]] != TRUE )
912      {
913          NSLog(@"Error the move from (1,3) to (2,2) was not allowed.
914                  ");
915          testPassed = FALSE;
916      }
917
918      if( [allmoves containsObject:[MoveObject moveObjectWithMove:
919                      makeMoveFromInt(0,3,1,2)]] != TRUE )
920      {
921          NSLog(@"Error the move from (0,3) to (1,2) was not allowed.
922                  ");
923          testPassed = FALSE;
924      }
925
926
927
928      if( [allmoves containsObject:[MoveObject moveObjectWithMove:
929                      makeMoveFromInt(0,2,1,1)]] != TRUE )
930      {
931          NSLog(@"Error the move from (0,2) to (1,1) was not allowed.
932                  ");
933          testPassed = FALSE;
934
935      if( testPassed == TRUE )
936          NSLog(@"Test 20 passed.");
937
938      [pool release];
939  }

```

C.3 MiniMax Source Code

C.3.1 AIDefinitions.h

```
1 // Kolibrat AI
```

```
2 //  AIDefinitions.h
3 //
4 //  Created by Aron Lindberg.
5
6 #import "Datastructures.h"
7 #import "GameLogic.h"
8
9 #define HASHSIZE 20000
10
11 static struct hashList *hashtable[HASHSIZE];
12
13 typedef struct hashList {
14     GameState state;
15     BOOL foundNow;
16     struct hashList *next;
17     short int level;
18     int score;
19 } HASHList;
20
21
22
23 typedef struct treeState {
24     BoardMove lastMove;
25     unsigned short int numberChildren;
26     unsigned short int level;
27     int boardScore;
28     GameState gs;
29     int a;
30     int b;
31 } TreeState;
32
33 typedef struct treeListElement {
34     struct treeState *state;
35     void *nextEllement;
36 } TreeListElement;
37
38
39 typedef struct treeStateList {
40
41     struct treeListElement *firstElement;
42     struct treeListElement *lastElement;
43     double numberElements;
44
45 } TreeStateList;
46
47 // Some plain C methods to make instances of the custom
48 // typedefinitions.
49 void addTreeStateTo( TreeState* state, TreeStateList* list );
50 void removeFirstElement( TreeStateList *list );
51 TreeStateList makeTreeStateList( TreeState* state);
52 TreeState makeTreeState( GameState* gs, TreeState* anceter);
53 TreeState makeTreeStateChild(TreeState* anceter);
54 void freeTreeState( TreeState* ts);
55 void addTreeStateTo( TreeState* state, TreeStateList* list );
```

```

56 void removeFirstElement( TreeStateList *list );
57 TreeStateList makeTreeStateList( TreeState* state);
58
59 unsigned hashValue( GameState *gs );
60 BOOL equalGameStates( GameState *gs1, GameState *gs2 );
61 HASHList* findInHashTable( GameState *gs );
62 BOOL insertIntoHashTable( GameState *gs, int level , int badscore);
63 void freeHashTable();
64 void NewCalcHashTable();

```

C.3.2 AIDefinitions.m

```

1 // Kolibrat AI
2 // AIDefinitions.m
3 //
4 // Created by Aron Lindberg.
5
6 #import "AIDefinitions.h"
7
8 TreeStateList makeTreeStateList( TreeState* state)
9 {
10     struct treeStateList temp;
11     temp.numberElements = 1;
12     return temp;
13 }
14
15 void addTreeStateTo( TreeState* state, TreeStateList* list )
16 {
17     (*(*list).lastElement).nextEllement = state;
18     (*list).numberElements++;
19 }
20
21 void removeFirstElement( TreeStateList *list )
22 {
23     TreeListElement *seconodElement = (*(*list).firstElement)
24         .nextEllement;
25     (*list).firstElement = seconodElement;
26     (*list).numberElements--;
27 }
28
29 TreeState makeTreeState( GameState *gs, TreeState* anceter)
30 {
31     struct treeState temp;
32     temp.level = 0;
33     temp.boardScore = 0;
34     temp.a = INT_MIN;
35     temp.b = INT_MAX;
36     temp.gs = copyGameState( gs );
37
38     return temp;
39 }

```

```
40
41 TreeState makeTreeStateChild(TreeState* anceter)
42 {
43     struct treeState temp;
44
45     temp.level = anceter->level;
46     temp.boardScore = anceter->boardScore;
47     temp.a = anceter->a;
48     temp.b = anceter->b;
49
50     temp.gs = copyGameState( &anceter->gs );
51
52     return temp;
53 }
54
55 void freeTreeState( TreeState* ts)
56 {
57     freeGameState( &ts->gs);
58 }
59
60 unsigned hashValue( GameState *gs )
61 {
62     unsigned value;
63     value = gs->playerMoving * 1325879543;
64     value += gs->blackPicesOnBoard *59448612;
65     value += gs->redPicesOnBoard *65939875;
66     value += gs->redPicesOnBoard *65934321;
67     value += gs->score.red * 765423104;
68     value += gs->score.black * 126423265;
69     value += gs->gameStatus.gameOver * 99842321;
70
71     int x, y;
72     for ( x = 0 ; x < gs->boardSize.width ; x++ ) {
73         for ( y = 0 ; y < gs->boardSize.height ; y++ ) {
74             value += gs->board[x][y].occupiedByRed * 1236549875 * (
75                 x + 1);
76             value += gs->board[x][y].occupiedByBlack * 978642321 *
77                 (y + 1);
78         }
79     }
80
81     return value % HASHSIZE;
82 }
83
84 BOOL equalGameStates( GameState *gs1, GameState *gs2 )
85 {
86     if( gs1->blackPicesOnBoard != gs2->blackPicesOnBoard)
87         return FALSE;
88
89     if( gs1->redPicesOnBoard != gs2->redPicesOnBoard )
90         return FALSE;
91
92     if( gs1->playerMoving != gs2->playerMoving )
93         return FALSE;
94 }
```

```

93     if( gs1->gameStatus.gameOver != gs2->gameStatus.gameOver )
94         return FALSE;
95
96     if( gs1->gameStatus.winner != gs2->gameStatus.winner )
97         return FALSE;
98
99     if( gs1->score.red != gs2->score.red )
100        return FALSE;
101
102    if( gs1->score.black != gs2->score.black )
103        return FALSE;
104
105    int x, y;
106    for ( x = 0 ; x < gs1->boardSize.width ; x++ ) {
107        for ( y = 0 ; y < gs1->boardSize.height ; y++ ) {
108            if( gs1->board[x][y].occupiedByBlack != gs2->board[x][y]
109                .occupiedByBlack )
110            {
111                return FALSE;
112            }
113            if( gs1->board[x][y].occupiedByRed != gs2->board[x][y]
114                .occupiedByRed )
115            {
116                return FALSE;
117            }
118        }
119    }
120
121 HASHList* findInHashTable( GameState *gs )//, HASHList hashtable[]
122 {
123     HASHList *test;
124
125     if( &(hashtable[hashValue( gs )]) == NULL )
126         return NULL;
127
128     for( test = hashtable[hashValue( gs )] ; test != NULL ; test =
129         test->next )
130     {
131         if ( equalGameStates( gs, &(test->state) ) )
132             return test;
133     }
134
135
136 BOOL insertIntoHashTable( GameState *gs, int level , int badscore)
137 {
138     struct hashList *test;
139     unsigned hashvalue;
140
141     if( (findInHashTable( gs )) == NULL )
142     {
143         test = malloc( sizeof( HASHList ) );

```

```
144     hashvalue = hashValue( gs );
145
146     test->foundNow = TRUE;
147     test->level = level;
148
149     test->score = badscore;
150
151
152     test->state.blackPicesOnBoard = gs->blackPicesOnBoard;
153     test->state.redPicesOnBoard = gs->redPicesOnBoard;
154
155     test->state.playerMoving = gs->playerMoving;
156     test->state.gameStatus = gs->gameStatus;
157
158     test->state.score = gs->score;
159
160     test->state.board = malloc(gs->boardSize.width * sizeof(
161                               BoardFieldContent *));
162
163     int i;
164     for(i = 0; i < gs->boardSize.width; i++)
165     {
166         test->state.board[i] = malloc(gs->boardSize.height *
167                                       sizeof(BoardFieldContent));
168     }
169
170     int x, y;
171     for ( x = 0 ; x < gs->boardSize.width ; x++ ) {
172         for ( y = 0 ; y < gs->boardSize.height ; y++ ) {
173             test->state.board[x][y] = gs->board[x][y];
174         }
175     }
176
177     test->next = hashtable[hashvalue];
178     hashtable[hashvalue] = test;
179
180     return TRUE;
181 }
182 }
183
184 void freeHashTable()
185 {
186     int i;
187     for( i = 0 ; i < HASHSIZE ; i ++ )
188     {
189         struct hashList *nextstate = hashtable[i];
190
191         while( nextstate != NULL )
192         {
193             struct hashList *ok = nextstate->next;
194             free( nextstate );
195             nextstate = ok;
196         }
197 }
```

```

197         hashtable[i] = NULL ;
198     }
199 }
200
201 void NewCalcHashTable()
202 {
203     int i;
204     for( i = 0 ; i < HASHSIZE ; i ++ )
205     {
206         struct hashList *nextstate = hashtable[i];
207
208         while( nextstate != NULL )
209         {
210             struct hashList *ok = nextstate->next;
211             nextstate->foundNow = FALSE;
212             nextstate = ok;
213         }
214
215         if( hashtable[i] != NULL )
216             hashtable[i]->foundNow = FALSE;
217     }
218 }
```

C.3.3 AdvancedAI.h

```

1 // Kolibrat AI
2 // AdvancedAI.h
3 //
4 // Created by Aron Lindberg.
5
6 #import "Datastructures.h"
7 #import "PlayerProtocol.h"
8 #import "AIDefinitions.h"
9 #import "GameEngine.h"
10 #import "GameLogic.h"
11
12 @interface AdvancedAI : NSObject < Player_Protocol >
13 {
14 // Private instance variabels.
15 @private
16     NSString *name;
17     id engine;
18     id gl;
19
20     BOOL playerID;
21     BOOL enemyID;
22
23     int otherPlayer;
24     BOOL waitingForOtherPlayer;
25
26     NSDate *timeToReturnMove;
27 }
```

```

28     struct hashList *hashtable[HASHSIZE];
29
30     int val1, val2, val3, val4, val5, val6, val7, val8, val9,
31     val10, val11;
32 }
33 // Instance methods.
34 -(int)minValue:(TreeState*)treeState scorePointer:(int*)bestScore
35     boardMovePointer:(BoardMove*)bestMove searchDepth:(int)maxLevel
36 ;
35 -(int)maxValue:(TreeState*)treeState scorePointer:(int*)bestScore
36     boardMovePointer:(BoardMove*)bestMove searchDepth:(int)maxLevel
37 ;
36 -(BoardMove)searchForMove:(GameState)gs;
37 -(id)initAsPlayer:(int)player withName:(NSString *)playerName
38     boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(int)maxGoals;
39 -(void)reset;
40 -(void)setGameEngine:(id)ge;
41 -(int)eval:(TreeState *)ts;
42 -(NSString *)playerName;
43 // Class methods.
44 +(NSString *)playerType;
45
46 @end

```

C.3.4 AdvancedAI.m

```

1 // Kolibrat AI
2 // AdvancedAI.m
3 //
4 // Created by Aron Lindberg.
5
6 #import "AdvancedAI.h"
7
8 #define TIME_TO_THINK 2.0
9
10 @implementation AdvancedAI
11
12 -(void) dealloc
13 {
14     [name release];
15     [gl release];
16     [super dealloc];
17 }
18
19 // Called when the player gets the turn.
20 -(void)startNewTurn
21 {
22     if( [engine playerMove: [self searchForMove: [engine gameState
23         ]] fromPlayer:self] == FALSE )

```

```

23         NSLog(@"SOS the engine denied movement.");
24     }
25
26 // The method that begins the MiniMax Search.
27 - (BoardMove)searchForMove:(GameState)gs
28 {
29     int bestScore = INT_MIN;
30     BoardMove bestMove = allLegalMoves( &gs ).head->moveData;
31
32     timeToReturnMove = [NSDate dateWithTimeIntervalSinceNow:
33                         TIME_TO_THINK];
34
35     int depth = 1 ;
36     BoardMove lastbestMove;
37
38     while ( [timeToReturnMove timeIntervalSinceNow] > 0 )
39     {
40         NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init
41                                     ];
42
43         TreeState searchTree = makeTreeState(&gs , NULL);
44         bestScore = INT_MIN;
45         lastbestMove = bestMove;
46
47         [self maxValue:&searchTree scorePointer:&bestScore
48          boardMovePointer:&bestMove searchDepth:depth ];
49         depth++;
50         freeTreeState(&searchTree);
51
52         NewCalcHashTable();
53
54         [pool release];
55     }
56
57     freeHashTable();
58     return lastbestMove;
59 }
60
61 -(int)maxValue:(TreeState*)treeState scorePointer:(int*)bestScore
62           boardMovePointer:(BoardMove*)bestMove searchDepth:(int)maxLevel
63 {
64     BOOL updateScore = FALSE;
65     insertIntoHashTable( &treeState->gs , treeState->level, INT_MIN
66                         );
67
68     if( treeState->level == maxLevel || treeState->
69         gs.gameStatus.gameOver == TRUE || [timeToReturnMove
70         timeIntervalSinceNow] < 0 )
71     {
72
73         if( treeState->level == 1 && treeState->boardScore > *
74             bestScore)
75         {
76             *bestMove = treeState->gs.lastMove;

```

```

70         *bestScore = treeState->boardScore;
71     }
72
73     if ( treeState->level == 1 && treeState->
74         gs.gameStatus.winner == playerID && treeState->
75         gs.gameStatus.gameOver == TRUE )
76     {
77         *bestMove = treeState->gs.lastMove;
78         *bestScore = INT_MAX;
79         if( updateScore == TRUE )
80             findInHashTable(&treeState->gs)->score = treeState-
81                 >boardScore;
82         return INT_MAX;
83     }
84
85     if( updateScore == TRUE )
86         findInHashTable(&treeState->gs)->score = treeState->
87             boardScore;
88     return [self eval:treeState];
89 }
90
91 treeState->boardScore = INT_MIN;
92
93 SimpleList allMoves = makeSimpleList();
94 allMoves = allLegalMoves(&treeState->gs);
95
96 while ( allMoves.head != NULL)
97 {
98     TreeState new = makeTreeStateChild( treeState );
99     new.level = treeState->level + 1 ;
100    makeMoveOnState( &allMoves.head->moveData ,&new.gs );
101
102    if( findInHashTable( &new.gs ) == NULL || (findInHashTable(
103        ( &new.gs )->level == new.level && findInHashTable( (
104            new.gs )->foundNow == FALSE ) ) )
105    {
106        if( findInHashTable( &new.gs ) != NULL )
107        {
108            findInHashTable( &new.gs )->foundNow = TRUE;
109        }
110        if( new.gs.playerMoving == enemyID )
111            treeState->boardScore = MAX( treeState->
112                boardScore, [self minValue:&new scorePointer:
113                    bestScore boardMovePointer:bestMove
114                    searchDepth:maxLevel] );
115        else
116        {
117            treeState->boardScore = MAX( treeState->
118                boardScore, [self maxValue:&new scorePointer:
119                    bestScore boardMovePointer:bestMove searchDepth
120                    :maxLevel] );
121        }
122    }
123
124    if( treeState->boardScore >= treeState->b )
125    {

```

```

113         freeTreeState(&new);
114         freeSimpleList(&allMoves);
115
116         if( treeState->level == 1 && treeState->boardScore > *bestScore)
117         {
118             *bestMove = treeState->gs.lastMove;
119             *bestScore = treeState->boardScore;
120         }
121         findInHashTable(&(treeState->gs))->score = treeState->boardScore;
122         return treeState->boardScore;
123     }
124
125     if( treeState->boardScore > treeState->a )
126     {
127         treeState->a = MAX( treeState->a, treeState->boardScore );
128     }
129
130     removeHeadFromSimpleList(&allMoves);
131     freeTreeState(&new);
132 }
133
134 freeSimpleList(&allMoves);
135
136 if( treeState->level == 1 && treeState->boardScore > *bestScore )
137 {
138     *bestMove = treeState->gs.lastMove;
139     *bestScore = treeState->boardScore;
140 }
141 if( updateScore == TRUE )
142     findInHashTable(&treeState->gs)->score = treeState->boardScore;
143 return treeState->boardScore;
144 }
145
146 -(int)minValue:(TreeState*)treeState scorePointer:(int*)bestScore
147 boardMovePointer:(BoardMove*)bestMove searchDepth:(int)maxLevel
148 {
149     BOOL updateScore = FALSE;
150     insertIntoHashTable( &treeState->gs , treeState->level, INT_MAX );
151
152     if( treeState->level == maxLevel || treeState->
153         gs.gameStatus.gameOver == TRUE || [timeToReturnMove
154         timeIntervalSinceNow] < 0 )
155     {
156         if( treeState->level == 1 && treeState->boardScore > *bestScore)
157         {
158             *bestMove = treeState->gs.lastMove;
159             *bestScore = treeState->boardScore;
160         }

```

```

158
159     if ( treeState->level == 1 && treeState->
160         gs.gameStatus.winner == playerID && treeState->
161         gs.gameStatus.gameOver == TRUE )
162     {
163         *bestMove = treeState->gs.lastMove;
164         *bestScore = INT_MAX;
165         if( updateScore == TRUE )
166             findInHashTable(&treeState->gs)->score = treeState-
167                 >boardScore;
168         return INT_MAX;
169     }
170
171     if( updateScore == TRUE )
172         findInHashTable(&treeState->gs)->score = treeState->
173             boardScore;
174     return [self eval:treeState];
175 }
176
177 treeState->boardScore = INT_MAX;
178
179 SimpleList allMoves = makeSimpleList();
180 allMoves = allLegalMoves(&treeState->gs);
181
182 while ( allMoves.head != NULL)
183 {
184     TreeState new = makeTreeStateChild( treeState );
185     new.level = treeState->level + 1;
186     makeMoveOnState( &allMoves.head->moveData ,&new.gs );
187
188     if( findInHashTable( &new.gs ) == NULL || (findInHashTable(
189         &new.gs )->level == new.level && findInHashTable( &
190         new.gs )->foundNow == FALSE ) )
191     {
192         if( findInHashTable( &new.gs ) != NULL )
193             findInHashTable( &new.gs )->foundNow = TRUE;
194
195         if( new.gs.playerMoving == playerID )
196             treeState->boardScore = MIN( treeState->
197                 boardScore, [self maxValue:&new scorePointer:
198                     bestScore boardMovePointer:bestMove searchDepth
199                     :maxLevel]);
200         else
201             treeState->boardScore = MIN( treeState->
202                 boardScore, [self minValue:&new scorePointer:
203                     bestScore boardMovePointer:bestMove searchDepth
204                     :maxLevel]);
205     }
206
207     if( treeState->boardScore <= treeState->a )
208     {
209         freeTreeState(&new);
210         freeSimpleList(&allMoves);
211     }
212 }
```

```

200         if( treeState->level == 1 && treeState->boardScore > *bestScore)
201     {
202         *bestMove = treeState->gs.lastMove;
203         *bestScore = treeState->boardScore;
204     }
205     if( updateScore == TRUE )
206         findInHashTable(&treeState->gs)->score = treeState->boardScore;
207     return treeState->boardScore;
208 }
209
210     if( treeState->boardScore < treeState->b )
211         treeState->b = MIN( treeState->b , treeState->boardScore );
212
213     removeHeadFromSimpleList( &allMoves );
214     freeTreeState(&new);
215 }
216
217 freeSimpleList(&allMoves);
218
219 if( treeState->level == 1 && treeState->boardScore > *bestScore )
220 {
221     *bestMove = treeState->gs.lastMove;
222     *bestScore = treeState->boardScore;
223 }
224
225 if( updateScore == TRUE )
226     findInHashTable(&treeState->gs)->score = treeState->boardScore;
227     return treeState->boardScore;
228 }
229
230 - (int)eval:(TreeState *)ts
231 {
232 // val1: Value of a piece on the bottom line.
233 // val2: Value increase for a piece per level.
234 // val3: Bonus for standing in the middle of the board.
235 // val4: Penalty for standing in front of the enemy, when they have the turn.
236 // val5: Bonus for having 2 pieces on a row.
237 // val6: The value of a legal move.
238 // val7: The value of a goal.
239 // val8: the value of having the turn.
240 // val9: the value of being able to insert pieces on the board.
241 // val10: The value of having a piece standing on the opponents home line.
242 // val11: The value of being the player having most pieces on the board.
243
244     int redScore = 0 ;
245     int blackScore = 0 ;
246

```

```

247     if( GAME_NOT_RUNNING( ts->gs.gameStatus ) )
248     {
249         if( ts->gs.gameStatus.winner == PLAYER_RED )
250         {
251             redScore = (INT_MAX - 100) - ts->level ;
252             blackScore = 0 ;
253         }
254         if( ts->gs.gameStatus.winner == PLAYER_BLACK )
255         {
256             blackScore = (INT_MAX - 100) - ts->level ;
257             redScore = 0 ;
258         }
259         return playerID == PLAYER_RED ? redScore - blackScore :
260             blackScore - redScore;
261     }
262
263     BOOL realPlayerMovin = ts->gs.playerMoving;
264
// Calculate score for red:
265     int x, y;
266     for ( x = 0 ; x < ts->gs.boardSize.width ; x++ ) {
267         for ( y = 0 ; y < ts->gs.boardSize.height ; y++ ) {
268             if( RED_FIELD( ts->gs.board[x][y] ) )
269             {
270                 redScore += val1 + val2 * y;
271
272                 if( x > 0 && x < ts->gs.boardSize.width - 1 )
273                     redScore += val3;
274
275                 if( y < ( ts->gs.boardSize.height - 1 ) &&
276                     BLACK_FIELD( ts->gs.board[ x ][ y + 1 ] ) &&
277                     ts->gs.playerMoving == PLAYER_BLACK )
278                 {
279                     redScore -= val4;
280                 }
281
282                 if( y < ( ts->gs.boardSize.height - 1 ) &&
283                     RED_FIELD( ts->gs.board[ x ][ y + 1 ] ) )
284                 {
285                     redScore += val5;
286                 }
287             }
288         }
289     }
290     if( RED_FIELD( ts->gs.board[x][ts->gs.boardSize.height] ) )
291     {
292         redScore += val10;
293     }
294
295     if( ts->gs.redPicesOnBoard > ts->gs.blackPicesOnBoard )
296     {

```

```
298         redScore += val11;
299     }
300
301     if( realPlayerMovin == PLAYER_RED )
302     {
303         redScore += val8;
304     }
305
306     ts->gs.playerMoving = PLAYER_RED;
307
308     if( [gl playerMovingCanInsertPieceOnState: &ts->gs] )
309     {
310         redScore += val9;
311     }
312
313     NSSet *redMoves = [gl allLegalMoves: &ts->gs ];
314     redScore += [redMoves count] * val6;
315     redScore += ts->gs.score.red * val7;
316
317     // Calculate score for black:
318     for ( x = 0 ; x < ts->gs.boardSize.width ; x++ ) {
319         for ( y = 0 ; y < ts->gs.boardSize.height ; y++ ) {
320             if( BLACK_FIELD( ts->gs.board[x][y] ) )
321             {
322                 blackScore += val1 + val2 * y;
323
324                 if( x > 0 && x < ts->gs.boardSize.width - 1 )
325                     blackScore += val3;
326
327                 if( y > 0 && RED_FIELD( ts->gs.board[ x ][ y - 1 ]
328                     ) && ts->gs.playerMoving == PLAYER_RED )
329                 {
330                     blackScore -= val4;
331                 }
332
333                 if( y > 0 && RED_FIELD( ts->gs.board[ x ][ y - 1 ]
334                     ) )
335                 {
336                     blackScore += val5;
337                 }
338             }
339
340             for ( x = 0 ; x < ts->gs.boardSize.width ; x++ )
341             {
342                 if( BLACK_FIELD( ts->gs.board[x][ts->gs.boardSize.height] )
343                     )
344                 {
345                     blackScore += val10;
346                 }
347
348             if( realPlayerMovin == PLAYER_BLACK )
349             {
```

```
350         blackScore += val8;
351     }
352
353     ts->gs.playerMoving = PLAYER_BLACK;
354
355     if( [gl playerMovingCanInsertPieceOnState: &ts->gs] )
356     {
357         blackScore += val9;
358     }
359
360     NSSet *blackMoves = [gl allLegalMoves: &ts->gs ];
361     blackScore += [blackMoves count] * val6;
362
363     if( ts->gs.blackPicesOnBoard > ts->gs.redPicesOnBoard )
364     {
365         blackScore += val11;
366     }
367
368     blackScore += ts->gs.score.black * val7;
369
370     if( GAME_NOT_RUNNING( ts->gs.gameStatus ) )
371     {
372         if( ts->gs.gameStatus.winner == PLAYER_RED )
373         {
374             redScore = (INT_MAX - 100 ) - ts->level ;
375             blackScore = 0 ;
376         }
377         if( ts->gs.gameStatus.winner == PLAYER_BLACK )
378         {
379             blackScore = (INT_MAX - 100 ) - ts->level ;
380             redScore = 0 ;
381         }
382     }
383
384     ts->gs.playerMoving = realPlayerMovin;
385     return playerID == PLAYER_RED ? redScore - blackScore :
386         blackScore - redScore;
387 }
388 - (id)initAsPlayer:(int)player withName:(NSString *)playerName
389   boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(int)maxGoals
390 {
391     self = [super init];
392     if (self != nil)
393     {
394         playerID = player;
395         enemyID = !playerID;
396
397         if( playerID == PLAYER_RED )
398             otherPlayer = PLAYER_BLACK;
399         else
400             otherPlayer = PLAYER_RED;
```

```
401     gl = [[GameLogic alloc] initWithMaxPices:maxPices
402             goalsToWin:maxGoals boardSize:bs];
403 
404     name = [NSString stringWithFormat:playerName];
405     [name retain];
406 
407     val1 = 50;
408     val2 = 26;
409     val3 = 52;
410     val4 = 57;
411     val5 = 52;
412     val6 = 33;
413     val7 = 100;
414     val8 = 9;
415     val9 = 53;
416     val10 = 3;
417     val11 = 17;
418 }
419 }
420 
421 - (void)setGameEngine:(id)ge
422 {
423     engine = ge;
424 }
425 
426 - (void)reset
427 {
428     // Nothing to reset.
429 }
430 
431 // The name of this instance of player.
432 - (NSString *)playerName
433 {
434     return name;
435 }
436 
437 // Defines the name of this type of player.
438 + (NSString *)playerType
439 {
440     return [NSString stringWithFormat:@"Advanced AI@%1.1f",
441             TIME_TO_THINK];
442 }
443 @end
```

C.4 Simulated Annealing Source Code

C.4.1 simAnnealing.h

```
1 // simAnnealing
```

```
2 // simAnnealing.h
3 //
4 // Created by Aron Lindberg.
5
6 #import <Cocoa/Cocoa.h>
7 #import <Foundation/Foundation.h>
8 #import "Datastructures.h"
9 #import "GameLogic.h"
10 #import "AIPlayer2.h"
11 #import "PlayerProtocol.h"
12 #import "AIDefinitions.h"
13 #import "GameEngine.h"
14
15 typedef struct eval {
16     int val1;
17     int val2;
18     int val3;
19     int val4;
20     int val5;
21     int val6;
22     int val7;
23     int val8;
24     int val9;
25     int val10;
26     int val11;
27 } EVAL_VARS;
28
29 @interface simAneling : NSObject {
30
31 @private
32     GameEngine *newEngine;
33     AIPlayer *red;
34     AIPlayer *black;
35
36     EVAL_VARS currentRed, currentBlack;
37
38     int nextAi;
39     int aiInWinner;
40     int aiInLoser;
41
42     EVAL_VARS AI_stuff[60];
43     EVAL_VARS AI_winners[30];
44     EVAL_VARS AI_losers[30];
45
46     int generation;
47     float randfactor;
48
49     NSFileHandle *fh;
50     EVAL_VARS nextplayer1, nextplayer2;
51 }
52
53 - (void)awakeFromNib;
54 - (EVAL_VARS) randomise:(EVAL_VARS)this;
55
56 @end
```

C.4.2 simAneling.m

```

1 //  simAnnealing
2 //  simAnnealing.h
3 //
4 //  Created by Aron Lindberg.
5
6 #import "simAneling.h"
7
8 #define GOALS_TO_WIN 1
9 #define PICES_ON_BOARD 6
10 #define BOARD makeBoardSize( 4, 6)
11
12 @implementation simAneling
13
14 - (void)awakeFromNib
15 {
16     randfactor = 1.0;
17
18     fh = [NSFileHandle fileHandleForWritingAtPath:@"/Users/
19         Output.txt"];
20     [fh retain];
21     [fh writeData:[@"Start of new Calculation.\n" dataUsingEncoding
22         :NSASCIIStringEncoding]];
23
24     int i;
25     for ( i = 0 ; i < 60 ; i++ )
26     {
27         AI_stuff[i].val1 = 600 + random() % 200 - 100;
28         AI_stuff[i].val2 = 50 + random() % 200 - 100;
29         AI_stuff[i].val3 = 730 + random() % 200 - 100;
30         AI_stuff[i].val4 = 760 + random() % 200 - 100;
31         AI_stuff[i].val5 = 500 + random() % 200 - 100;
32         AI_stuff[i].val6 = 70 + random() % 200 - 100;
33         AI_stuff[i].val7 = 1000 + random() % 200 - 100;
34         AI_stuff[i].val8 = 50 + random() % 200 - 100;
35         AI_stuff[i].val9 = 50 + random() % 200 - 100;
36         AI_stuff[i].val10 = 50 + random() % 200 - 100;
37         AI_stuff[i].val11 = 50 + random() % 200 - 100;
38
39         [fh writeData:[[NSString stringWithFormat:@"AI_stuff[%i]: %
40             i, %i, %
41             i, %i\n", i,
42             AI_stuff[i].val1, AI_stuff[i].val2, AI_stuff[i].val3,
43             AI_stuff[i].val4, AI_stuff[i].val5, AI_stuff[i].val6,
44             AI_stuff[i].val7, AI_stuff[i].val8, AI_stuff[i].val9,
45             AI_stuff[i].val10, AI_stuff[i].val11 ]
46             dataUsingEncoding:NSUTFStringEncoding]];
47         NSLog(@"%@", i, AI_stuff[i].val1, AI_stuff[i].val2,
48             AI_stuff[i].val3, AI_stuff[i].val4, AI_stuff[i].val5,
49             AI_stuff[i].val6, AI_stuff[i].val7, AI_stuff[i].val8,
50             AI_stuff[i].val9, AI_stuff[i].val10, AI_stuff[i].val11)

```

```
        ;
41    }
42
43    generation = 0;
44    nextAi = 0;
45
46    NSNotificationCenter *mainCenter = [NSNotificationCenter
47                                         defaultCenter];
48
49    [mainCenter addObserver:self selector:@selector(RestartGame:)
50     name:@"GameOver" object:nil];
51
52    red = [[AIPlayer alloc] initAsPlayer:PLAYER_RED withName: @"RED"
53          " boardSize:BOARD picesOnboard:PICES_ON_BOARD goalsToWin:
54          INT_MAX];
55    black = [[AIPlayer alloc] initAsPlayer:PLAYER_BLACK withName: @"BLACK"
56          " boardSize:BOARD picesOnboard:PICES_ON_BOARD
57          goalsToWin: INT_MAX];
58
59    currentRed = AI_stuff[nextAi];
60    nextAi++;
61    currentBlack = AI_stuff[nextAi];
62
63    nextAi++;
64
65    [red setEvaluationValues:AI_stuff[0]];
66    [black setEvaluationValues:AI_stuff[1]];
67
68    newEngine = [[GameEngine alloc] initWithPlayersRed:red
69                           andBlack:black
70                           goalsToWin:GOALS_TO_WIN
71                           GameBoardDim:BOARD
72                           MaxPices:
73                           PICES_ON_BOARD
74                           connectToGUI:nil];
75
76    [newEngine retain];
77
78    [red retain];
79    [black retain];
80
81    [red setGameEngine:newEngine];
82    [black setGameEngine:newEngine];
83
84    aiInWinner = 0;
85    aiInLoser = 0;
86}
87
88
89
90
91 - (void)RestartGame:(NSNotification *)notification
92 {
93    [fh writeData:[[NSString stringWithFormat:@"\n----> Game %i is
94      over <----\n", aiInWinner + 1 ] dataUsingEncoding:
95      NSASCIIStringEncoding]];
96
97    if( [notification object] == red )
98
```

```

86      {
87          AI_winners[aiInWinner] = currentRed;
88          aiInWinner++;
89
90          [fh writeData: [[NSString stringWithFormat:@"the red
91              winner had: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i,
92              \n", currentRed.val1,
93              currentRed.val2, currentRed.val3, currentRed.val4
94              ,currentRed.val5, currentRed.val6,
95              currentRed.val7, currentRed.val8,
96              currentRed.val9, currentRed.val10, currentRed.val11
97              ] dataUsingEncoding:NSUTFStringEncoding]] ;
98
98      else if( [notification object] == black )
99      {
100          AI_winners[aiInWinner] = currentBlack;
101          aiInWinner++;
102          [fh writeData: [[NSString stringWithFormat: @"the black
103              winner had: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i,
104              \n",currentBlack.val1,currentBlack.val2 ,
105              currentBlack.val3, currentBlack.val4,
106              currentBlack.val5, currentBlack.val6,
107              currentBlack.val7, currentBlack.val8,
108              currentBlack.val9, currentBlack.val10,
109              currentBlack.val11] dataUsingEncoding:
110              NSASCIIStringEncoding]];
111          [fh writeData:[[[NSString stringWithFormat: @"the red loser
112              had: %i, %i, %i, %i, %i, %i, %i, %i \n"
113              ,currentRed.val1,currentRed.val2 , currentRed.val3,
114              currentRed.val4, currentRed.val5, currentRed.val6,
115              currentRed.val7, currentRed.val8, currentRed.val9,
116              currentRed.val10, currentRed.val11] dataUsingEncoding:
117              NSASCIIStringEncoding]];
118      }
119
120      if( nextAi != 60 )
121      {
122          nextplayer1 = AI_stuff[nextAi];
123          nextAi++;
124          nextplayer2 = AI_stuff[nextAi];
125          nextAi++;
126
127          [red setEvaluationValues:nextplayer1];

```

```
114
115     [black setEvaluationValues:nextplayer2];
116
117     currentRed = nextplayer1;
118     currentBlack = nextplayer2;
119     [newEngine resetGame];
120     return;
121 }
122
123 if( nextAi == 60 )
124 {
125     generation++;
126
127     [fh writeData:[[NSString stringWithFormat:@"\n\n\n ----->
128         Generating generation %i. Randomfactor is: %1.3f <-----
129         \n", generation, randfactor ] dataUsingEncoding:
130         NSASCIIStringEncoding]];
131
132     BOOL spotfree[60];
133
134     int i;
135     for(i = 0 ; i < 60 ; i++ )
136     {
137         spotfree[i] = TRUE;
138
139         nextAi = 0;
140         aiInWinner = 0;
141         aiInLoser = 0;
142
143         int somevar1 = 0;
144
145         for( i=0; i < 30 ; i++ )
146         {
147             int temp = random() % 60;
148             while( spotfree[ temp ] != TRUE )
149             {
150                 temp = random() % 60;
151
152                 spotfree[temp] = false;
153
154                 AI_stuff[ temp ] = [self randomise: AI_winners[ i ] ];
155                 somevar1++;
156
157                 temp = random() % 60;
158                 while( spotfree[ temp ] != TRUE )
159                 {
160                     temp = random() % 60;
161
162                     spotfree[temp] = false;
163
164                     AI_stuff[ temp ] = AI_winners[ i ];
165                     somevar1++;
```

```

166         }
167
168     for ( i = 0 ; i < 60 ; i++ )
169     {
170         [fh writeData:[[NSString stringWithFormat:@"AI_stuff[%i
171             ]: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i\n", i
172             , AI_stuff[i].val1,AI_stuff[i].val2 , AI_stuff[i]
173             .val3, AI_stuff[i].val4, AI_stuff[i].val5, AI_stuff
174             [i].val6, AI_stuff[i].val7, AI_stuff[i].val8,
175             AI_stuff[i].val9, AI_stuff[i].val10, AI_stuff[i]
176             .val11 ] dataUsingEncoding:NSUTFStringEncoding]];
177     }
178
179     if( randfactor > 0.05 )
180         randfactor = randfactor - 0.02 ;
181     else
182         randfactor = randfactor - 0.002 ;
183
184     nextplayer1 = AI_stuff[nextAi];
185     nextAi++;
186     nextplayer2 = AI_stuff[nextAi];
187     nextAi++;
188
189 }
190
191 -(EVAL_VARS) randomise:(EVAL_VARS)this
192 {
193     int v1 = (random() % 1001 ) - 500 ;
194     int v2 = (random() % 1001 ) - 500 ;
195     int v3 = (random() % 1001 ) - 500 ;
196     int v4 = (random() % 1001 ) - 500 ;
197     int v5 = (random() % 1001 ) - 500 ;
198     int v6 = (random() % 1001 ) - 500 ;
199     int v7 = (random() % 1001 ) - 500 ;
200     int v8 = (random() % 1001 ) - 500 ;
201     int v9 = (random() % 1001 ) - 500 ;
202     int v10 = (random() % 1001 ) - 500 ;
203     int v11 = (random() % 1001 ) - 500 ;
204
205     v1 = (int) v1 * randfactor;
206     v2 = (int) v2 * randfactor;
207     v3 = (int) v3 * randfactor;
208     v4 = (int) v4 * randfactor;
209     v5 = (int) v5 * randfactor;
210     v6 = (int) v6 * randfactor;
211     v7 = (int) v7 * randfactor;
212     v8 = (int) v7 * randfactor;
213     v9 = (int) v7 * randfactor;
214     v10 = (int) v7 * randfactor;

```

```
215     v11 = (int) v7 * randfactor;
216
217     this.val1 += v1;
218     this.val2 += v2;
219     this.val3 += v3;
220     this.val4 += v4;
221     this.val5 += v5;
222     this.val6 += v6;
223     this.val7 += v7;
224     this.val8 += v8;
225     this.val9 += v9;
226     this.val10 += v10;
227     this.val11 += v11;
228
229     int t1 = MIN( this.val1 , this.val2 );
230     int t2 = MIN( this.val3 , this.val4 );
231     int t3 = MIN( this.val5 , this.val6 );
232     int t4 = MIN( this.val7 , this.val8 );
233     int t5 = MIN( this.val9 , this.val10 );
234     int t6 = MIN( this.val11 , t1 );
235     int t7 = MIN (t2, t3 );
236     int t8 = MIN (t4, t5 );
237     int t9 = MIN (t6, t7 );
238     int totalmin = MIN( t8,t9);
239
240     if ( totalmin < 0 )
241     {
242         this.val1 += abs(totalmin);
243         this.val2 += abs(totalmin);
244         this.val3 += abs(totalmin);
245         this.val4 += abs(totalmin);
246         this.val5 += abs(totalmin);
247         this.val6 += abs(totalmin);
248         this.val7 += abs(totalmin);
249         this.val8 += abs(totalmin);
250         this.val9 += abs(totalmin);
251         this.val10 += abs(totalmin);
252         this.val11 += abs(totalmin);
253     }
254
255     t1 = MAX( this.val1 , this.val2 );
256     t2 = MAX( this.val3 , this.val4 );
257     t3 = MAX( this.val5 , this.val6 );
258     t4 = MAX( this.val7 , this.val8 );
259     t5 = MAX( this.val9 , this.val10 );
260     t6 = MAX( this.val11 , t1 );
261     t7 = MAX (t2, t3 );
262     t8 = MAX (t4, t5 );
263     t9 = MAX (t6, t7 );
264     int totalmax = MAX( t8 , t9 );
265
266     if ( totalmax > 1000 )
267     {
268         this.val1 = (int)((float)this.val1 * 1000) / (totalmax)
269             ;
```

```

269         this.val2 = (int)((float)this.val2 * 1000) / (totalmax)
270             );
271         this.val3 = (int)((float)this.val3 * 1000) / (totalmax)
272             );
273         this.val4 = (int)((float)this.val4 * 1000) / (totalmax)
274             );
275         this.val5 = (int)((float)this.val5 * 1000) / (totalmax)
276             );
277         this.val6 = (int)((float)this.val6 * 1000) / (totalmax)
278             );
279     }
280     return this;
281 }
282
283 @end

```

C.5 Forced Loops Source Code

C.5.1 FakeLogic.h

```

1 //  Forced Loops
2 //  Fake Logic.h
3 //
4 //  Created by Aron Lindberg.
5
6 #import <Cocoa/Cocoa.h>
7
8 @interface FakeLogic : NSObject {
9
10 //  Private instance variabels.
11 @private
12     int maxGoals;
13     int maxPicesOnBoard;
14     BoardSize boardSize;
15     int lastState;
16 }
17
18 //  Public instance methods.
19 - (id)initWithMaxPices:(int)maxGoalsToWin:(int)goals boardSize:(
20     BoardSize)board;
21 - (GameState)CreateNewGameState;

```

```

21 - (void)resetGameState:(GameState *)gs;
22 - (NSSet *)allLegalMoves:(GameState *)gs;
23 - (BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs;
24
25 @end

```

C.5.2 FakeLogic.m

```

1 //  Forced Loops
2 //  Fake Logic.m
3 //
4 //  Created by Aron Lindberg.
5
6 #import "FakeLogic.h"
7
8 @implementation FakeLogic
9
10 static int *maxGoalsPointer;
11 static int *maxPicesOnBoardPointer;
12 static BoardSize *boardSizePointer;
13
14 - (id)initWithMaxPices:(int)max goalsToWin:(int)goals boardSize:(BoardSize)board
15 {
16     self = [super init];
17     if (self != nil)
18     {
19         maxPicesOnBoard = max;
20         boardSize = board;
21         maxGoals = goals;
22         maxGoalsPointer = &maxGoals;
23         maxPicesOnBoardPointer = &maxPicesOnBoard;
24         boardSizePointer = &boardSize;
25     }
26     return self;
27 }
28
29 - (GameState)CreateNewGameState
30 {
31     GameState gs = makeGameState(boardSize);
32     [self resetGameState: &gs];
33     return gs;
34 }
35
36 - (void)resetGameState:(GameState *)gs
37 {
38     int x, y;
39     for (x = 0 ; x < boardSize.width ; x++) {
40         for (y = 0 ; y < boardSize.height ; y++) {
41             gs->board[x][y] = makeEmptyField();
42         }
43     }

```

```
44     gs->score = makeGameScore(0,0);
45     gs->lastMove = NIL_MOVE;
46     gs->playerMoving = PLAYER_RED;
47     gs->redPicesOnBoard = 0;
48     gs->blackPicesOnBoard = 0;
49     gs->boardSize = boardSize; // TODO
50     gs->gameStatus = RUNNING;
51 }
52
53 - (NSSet *)allLegalMoves:(GameState *)gs
54 {
55     NSMutableSet *setOfLegalMoves = [NSMutableSet setWithCapacity:
56                                     10];
57
58     if( gs->score.red == 0 )
59     {
60         [setOfLegalMoves addObject: [[MoveObject alloc]
61                               initWithMove:makeMove(makeBoardField(-2,100)
62                                         ,makeBoardField(0,0))]];
63         [setOfLegalMoves addObject: [[MoveObject alloc]
64                               initWithMove:makeMove(makeBoardField(-2,120)
65                                         ,makeBoardField(0,0))]];
66     }
67     else if( gs->score.red == 100 )
68     {
69         [setOfLegalMoves addObject: [[MoveObject alloc]
70                               initWithMove:makeMove(makeBoardField(-2,101)
71                                         ,makeBoardField(0,0))]];
72         [setOfLegalMoves addObject: [[MoveObject alloc]
73                               initWithMove:makeMove(makeBoardField(-2,103)
74                                         ,makeBoardField(0,0))]];
75     }
76     else if( gs->score.red == 101 )
77     {
78         [setOfLegalMoves addObject: [[MoveObject alloc]
79                               initWithMove:makeMove(makeBoardField(-1,-10)
80                                         ,makeBoardField(0,0))]];
81         [setOfLegalMoves addObject: [[MoveObject alloc]
82                               initWithMove:makeMove(makeBoardField(-2,102)
83                                         ,makeBoardField(0,0))]];
84         [setOfLegalMoves addObject: [[MoveObject alloc]
85                               initWithMove:makeMove(makeBoardField(-2,200)
86                                         ,makeBoardField(0,0))]];
87     }
88     else if( gs->score.red == 102 )
89     {
90         [setOfLegalMoves addObject: [[MoveObject alloc]
91                               initWithMove:makeMove(makeBoardField(-1,-11)
92                                         ,makeBoardField(0,0))]];
93     }
94     else if( gs->score.red == 103 )
95     {
96         [setOfLegalMoves addObject: [[MoveObject alloc]
97                               initWithMove:makeMove(makeBoardField(-2,107)
98                                         ,makeBoardField(0,0))]];
99     }
100 }
```

```

80      [setOfLegalMoves addObject: [[MoveObject alloc]
81          initWithMove:makeMove(makeBoardField(-2,104)
82          ,makeBoardField(0,0))]];
83    }
84    else if( gs->score.red == 104 )
85    {
86        [setOfLegalMoves addObject: [[MoveObject alloc]
87          initWithMove:makeMove(makeBoardField(-2,105)
88          ,makeBoardField(0,0))]];
89    }
90    else if( gs->score.red == 105 )
91    {
92        [setOfLegalMoves addObject: [[MoveObject alloc]
93          initWithMove:makeMove(makeBoardField(-2,106)
94          ,makeBoardField(0,0))]];
95    }
96    else if( gs->score.red == 106 )
97    {
98        [setOfLegalMoves addObject: [[MoveObject alloc]
99          initWithMove:makeMove(makeBoardField(-1,-11)
100         ,makeBoardField(0,0))]];
101    }
102    else if( gs->score.red == 107 )
103    {
104        [setOfLegalMoves addObject: [[MoveObject alloc]
105          initWithMove:makeMove(makeBoardField(-2,108)
106          ,makeBoardField(0,0))]];
107    }
108    else if( gs->score.red == 109 )
109    {
110        [setOfLegalMoves addObject: [[MoveObject alloc]
111          initWithMove:makeMove(makeBoardField(-2,110)
112          ,makeBoardField(0,0))]];
113    }
114    else if( gs->score.red == 111 )
115    {
116        [setOfLegalMoves addObject: [[MoveObject alloc]
117          initWithMove:makeMove(makeBoardField(-2,121)
118          ,makeBoardField(0,0))]]; //121

```

```
115     }
116     else if( gs->score.red == 120 )
117     {
118         [setOfLegalMoves addObject: [[MoveObject alloc]
119             initWithMove:makeMove(makeBoardField(-2,121)
120             ,makeBoardField(0,0))]];
121     }
122     else if( gs->score.red == 121 )
123     {
124         [setOfLegalMoves addObject: [[MoveObject alloc]
125             initWithMove:makeMove(makeBoardField(-2,122)
126             ,makeBoardField(0,0))]];
127     }
128     else if( gs->score.red == 122 )
129     {
130         [setOfLegalMoves addObject: [[MoveObject alloc]
131             initWithMove:makeMove(makeBoardField(-2,103)
132             ,makeBoardField(0,0))]];
133     }
134     else
135     {
136         NSLog(@"Warning, states out of scroop.");
137     }
138 }
139
140 -(BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs
141 {
142     lastState = gs->score.red;
143     if( playerMove.from.x == -1 && playerMove.from.y == -10 )
144     {
145         gs->gameStatus.gameOver = TRUE;
146         gs->gameStatus.winner = PLAYER_RED;
147         gs->score.red = -10;
148     }
149     else if( playerMove.from.x == -1 && playerMove.from.y == -11 )
150     {
151         gs->gameStatus.gameOver = TRUE;
152         gs->gameStatus.winner = PLAYER_BLACK;
153         gs->score.red = -11;
154     }
155     else if( playerMove.from.x == -2 )
156     {
157         gs->score.red = playerMove.from.y;
158         gs->playerMoving = !gs->playerMoving;
159     }
160 }
161 }
```

```
162
163 @end
```

C.5.3 Forced Loops.m

```
1 //  Kolibrat
2 //  GameLogic.h
3 //
4 //  Created by Aron Lindberg.
5
6 #import <Foundation/Foundation.h>
7 #import "Datastructures.h"
8 #import "GameLogic.h"
9
10 @implementation gameStateObject
11
12 - (BOOL)hasLoop
13 {
14     return thisPathHasLoop;
15 }
16
17 - (NSMutableSet *)parrents
18 {
19     return parrents;
20 }
21
22 - (int) numberOfChildren
23 {
24     return [childs count];
25 }
26
27 - (GameState) returnState
28 {
29     return copyGameState( &state );
30 }
31
32 - (void) addParrent:(gameStateObject *)newParrent
33 {
34     [parrents addObject:newParrent];
35 }
36
37 -(void) releaseAllChildren
38 {
39     [childs removeAllObjects];
40     [childs release];
41 }
42
43 - (id) initWithGameState:(GameState)(newState parrent:(
44         gameStateObject*)motherObject makeChildren:(BOOL)makeChildren
45 {
46     self = [super init];
47     if (self != nil) {
```

```
47      child = [[NSMutableSet alloc] initWithCapacity:10];
48      state = copyGameState( &newState );
49
50      parrents = [[NSMutableSet alloc] initWithCapacity:3];
51      if( motherObject != nil )
52          [self addParrent:motherObject];
53
54      thisPathHasLoop = NO;
55      ChildrenIsFound = NO;
56
57  }
58  return self;
59 }
60
61 - (void) dealloc
62 {
63     if( [child count] != 0 )
64     {
65         [child removeAllObjects];
66         [child release];
67     }
68     [super dealloc];
69 }
70
71 - (NSArray*) findMyChildren
72 {
73     [child removeAllObjects];
74
75     BoardSize board = makeBoardSize( BOARDHEIGHT, BOARDWIDTH );
76     GameLogic *logic = [[GameLogic alloc] initWithMaxPices:
77                         MAX_PIECES goalsToWin:GOALS_TO_WIN boardSize:board];
78
79     NSSet * moves = [NSSet setWithSet: [logic allLegalMoves: &state
80                                     ]];
81     activechildren = [moves count];
82     NSEnumerator * enumerator = [moves objectEnumerator];
83     MoveObject *currentMove;
84     gameStateObject * childObject;
85
86     while (currentMove = [enumerator nextObject])
87     {
88         GameState child = copyGameState( &state );
89         [logic makeMove:[currentMove retrieveMove] withState: &child
90             ];
91
92         childObject = [[gameStateObject alloc] initWithGameState:
93                         child parrent:self makeChildren:NO ];
94
95         [child addObject:childObject];
96     }
97     ChildrenIsFound = TRUE;
98     return [child allObjects];
99 }
```

```
97 - (void) childHasLoop:(gameStateObject *)loopChild withStates:(  
    NSMutableArray *)states andKnownStates:(NSMutableSet *)  
    knownStates testPlayer:(BOOL)plr  
98 {  
99     if( thisPathHasLoop == TRUE)  
100        return;  
101  
102     if( state.playerMoving == plr )  
103     {  
104        NSEnumerator * enumerator = [childs objectEnumerator]; //  
            for all children  
105         gameStateObject *currentChildState;  
106  
107         while (currentChildState = [enumerator nextObject])  
108         {  
109             [currentChildState isSubpathOfLoop:states  
                andKnownStates:knownStates];  
110         }  
111  
112         thisPathHasLoop = TRUE;  
113  
114        NSEnumerator *e = [parrents objectEnumerator];  
115         id obj;  
116         while( obj = [e nextObject])  
117         {  
118             [obj childHasLoop:self withStates:states andKnownStates  
                :knownStates testPlayer: plr ];  
119         }  
120     }  
121  
122     else  
123     {  
124         [loopChild isSubpathOfLoop:states andKnownStates:  
            knownStates];  
125  
126         activechildren--;  
127  
128         if( activechildren == 0 )  
129         {  
130             thisPathHasLoop = TRUE;  
131  
132            NSEnumerator *e = [parrents objectEnumerator];  
133             id obj;  
134             while( obj = [e nextObject])  
135             {  
136                 [obj childHasLoop:self withStates:states  
                    andKnownStates:knownStates testPlayer: plr ];  
137             }  
138         }  
139     }  
140 }  
141  
142 - (void) isSubpathOfLoop:(NSMutableArray *)states andKnownStates:(  
    NSMutableSet *)knownStates  
143 {
```

```
144     if( thisPathHasLoop == TRUE )
145         return;
146
147    NSEnumerator * enumerator = [childs objectEnumerator];
148
149     gameStateObject *currentChildState;
150
151     while (currentChildState = [enumerator nextObject])
152     {
153         [currentChildState isSubpathOfLoop:states andKnownStates:
154          knownStates ];
155     }
156
157     [states removeObject:self];
158     [knownStates removeObject:self];
159
160     thisPathHasLoop = TRUE;
161 }
162 - (BOOL)isEqual:(id)anObject
163 {
164     if( ![self isKindOfClass: [anObject class]] )
165         return FALSE;
166
167     BOOL returnVal = FALSE;
168
169     GameState newState = [anObject returnState];
170
171     if( newState.blackPicesOnBoard == state.blackPicesOnBoard
172         &&
173         newState.redPicesOnBoard == state.redPicesOnBoard
174         &&
175         newState.playerMoving == state.playerMoving
176         &&
177         newState.gameStatus.gameOver == state.gameStatus.gameOver
178         &&
179         newState.gameStatus.winner == state.gameStatus.winner
180         &&
181         newState.score.red == state.score.red
182         &&
183         newState.score.black == state.score.black )
184
185     {
186         int x, y;
187         for (x = 0 ; x < BOARDWIDTH ; x++)
188     {
189         for (y = 0 ; y < BOARDHEIGHT ; y++)
190         {
191             if( newState.board[x][y].occupiedByRed != state.board[x][y].occupiedByRed ||
192                 newState.board[x][y].occupiedByBlack != state.board[x][y].occupiedByBlack )
```

```
190             returnVal = FALSE;
191         }
192     }
193 }
194 }
195
196 return returnVal;
197 }
198
199 - (unsigned)hash
200 {
201     unsigned hashVal;
202
203     int x, y;
204     for ( x = 0 ; x < BOARDWIDTH ; x++ ) {
205         for ( y = 0 ; y < BOARDHEIGHT ; y++ ) {
206             if( RED_FIELD( state.board[x][y] ) )
207             {
208                 hashVal += y;
209             }
210         }
211     }
212
213     hashVal += state.redPicesOnBoard;
214
215     if( state.playerMoving == PLAYER_RED )
216     {
217         hashVal = hashVal * 7;
218     }
219
220     hashVal += state.score.red * 3;
221     hashVal += state.score.black * 11;
222
223     return hashVal;
224 }
225
226 @end
227
228 GameState copyGameState( GameState *state )
229 {
230     struct gameStateStruct temp;
231
232     temp.blackPicesOnBoard = state->blackPicesOnBoard;
233     temp.redPicesOnBoard = state->redPicesOnBoard;
234
235     temp.boardSize = state->boardSize;
236
237     temp.playerMoving = state->playerMoving;
238     temp.gameStatus = state->gameStatus;
239
240     temp.lastMove = state->lastMove;
241     temp.score = state->score;
242
243     temp.board = malloc(BOARDWIDTH * sizeof(BoardFieldContent *));
244 }
```

```
245     int i;
246     for(i = 0; i < BOARDWIDTH; i++)
247     {
248         temp.board[i] = malloc(BOARDHEIGHT * sizeof(
249             BoardFieldContent));
250     }
251     int x, y;
252     for (x = 0 ; x < BOARDWIDTH ; x++ ) {
253         for (y = 0 ; y < BOARDHEIGHT ; y++ ) {
254             temp.board[x][y] = state->board[x][y];
255         }
256     }
257     return temp;
258 }
259
260 int main (int argc, const char * argv[])
261 {
262     NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
263
264     int player;
265     for( player = 1 ; player >= 0 ; player--)
266     { // Do the test both for the red and black player.
267
268         gameStateObject * rootgameState;
269
270         BoardSize board = makeBoardSize( BOARDHEIGHT, BOARDWIDTH);
271         GameLogic *logic = [[GameLogic alloc] initWithMaxPices:
272             MAX_PIECES goalsToWin:GOALS_TO_WIN boardSize:board];
273         [logic retain];
274
275         NSMutableArray * states = [[NSMutableArray alloc]
276             initWithCapacity:50000];
277         GameState emptyBoard = [logic CreateNewGameState];
278
279         gameStateObject * emptyBoardObject = [[gameStateObject
280             alloc] initWithGameState:emptyBoard parrent: nil
281             makeChildren: YES ];
282
283         rootgameState = emptyBoardObject;
284
285         [states addObject:emptyBoardObject];
286
287         gameStateObject * nextGameObject;
288         GameState next;
289         NSMutableSet * childStates = [[NSMutableSet alloc]
290             initWithCapacity:50000];
291         NSMutableSet * knownStates = [[NSMutableSet alloc]
292             initWithCapacity:50000];
293
294         [knownStates retain];
295
296         int lastActive = 0;
297         int val;
298         while ( [states count] > 0 )
```

```

293     {
294         NSAutoreleasePool * pool = [[NSAutoreleasePool alloc]
295             init];
296         val++;
297         if( val % 10000 == 999 )
298             if( lastActive > [states count] )
299                 lastActive = [states count];
300
301         nextGameObject = [states objectAtIndex:0];
302         next = [nextGameObject returnState];
303
304         [knownStates addObject:nextGameObject];
305         [nextGameObject retain];
306         [states removeObjectAtIndex:0];
307
308         if( next.gameStatus.gameOver == TRUE && player ==
309             PLAYER_RED )
310         {
311             //NSLog(@"red win somewhere.");
312         }
313         else if( next.gameStatus.gameOver == TRUE && player ==
314             PLAYER_BLACK )
315         {
316             //NSLog(@"black win somewhere.");
317         }
318         else if( next.playerMoving == player )
319         {
320             NSMutableSet * tempset2 = [[NSMutableSet alloc]
321                 initWithArray:[nextGameObject findMyChildren]]
322                 ;
323
324             [tempset2 retain];
325             [tempset2 minusSet:knownStates];
326             if( [tempset2 count] == [[nextGameObject
327                 findMyChildren] count] )
328             {
329                 [states addObjectsFromArray:[nextGameObject
330                     findMyChildren]];
331             }
332             else
333             {
334                 NSEnumerator *e = [[nextGameObject parrents]
335                     objectEnumerator];
336                 id obj;
337                 while( obj = [e nextObject])
338                 {
339                     [obj childHasLoop: nextGameObject
340                         withStates:states andKnownStates:
341                             knownStates testPlayer:player];
342                 }
343             }
344         }
345
346     }
347
348 }
```

```
338     else
339     {
340         NSMutableSet * tempset = [[NSMutableSet alloc]
341             initWithArray:[nextGameObject findMyChildren]]
342             ;
343         [tempset minusSet:knownStates];
344
345         if( [tempset count] == [[nextGameObject
346             findMyChildren] count] )
347             [states addObjectFromArray: [nextGameObject
348                 findMyChildren ]];
349         else
350         {
351             NSMutableSet * tempset3 = [[NSMutableSet alloc]
352                 initWithArray:[nextGameObject
353                     findMyChildren]] ;
354             [tempset3 intersectSet:knownStates];
355
356             NSEnumerator * enumerator = [tempset3
357                 objectEnumerator];
358             gameStateObject *obj;
359
360             while (obj = [enumerator nextObject])
361             {
362                 NSEnumerator * knownE = [knownStates
363                     objectEnumerator];
364                 id obj2;
365
366                 while (obj2 = [knownE nextObject])
367                 {
368                     if( [obj2 isEqual:obj] )
369                         [obj2 addParent:nextGameObject];
370                 }
371
372                 NSMutableSet * tempset4 = [[NSMutableSet alloc]
373                     initWithArray:[nextGameObject
374                         findMyChildren]] ;
375
376                 [tempset4 minusSet:knownStates];
377                 [states addObjectFromArray: [tempset4
378                     allObjects] ];
379             }
380         }
381     }
382
383     [nextGameObject release];
384
385     [pool release];
386 }
387
388 if( [rootgameState hasLoop] && player == PLAYER_RED )
389     NSLog(@"Red can enforce a loop.");
390 else if( [rootgameState hasLoop] && player == PLAYER_BLACK
391 )
```

```
381         NSLog(@"Black can enforce a loop.");
382     else if( player == PLAYER_BLACK )
383         NSLog(@"Black can not enforce a loop.");
384     else if( player == PLAYER_RED )
385         NSLog(@"Red can not enforce a loop.");
386
387     [states removeAllObjects];
388     [childStates removeAllObjects];
389     [knownStates removeAllObjects];
390 }
391 [pool release];
392 return 0;
393 }
```

Flow Diagrams

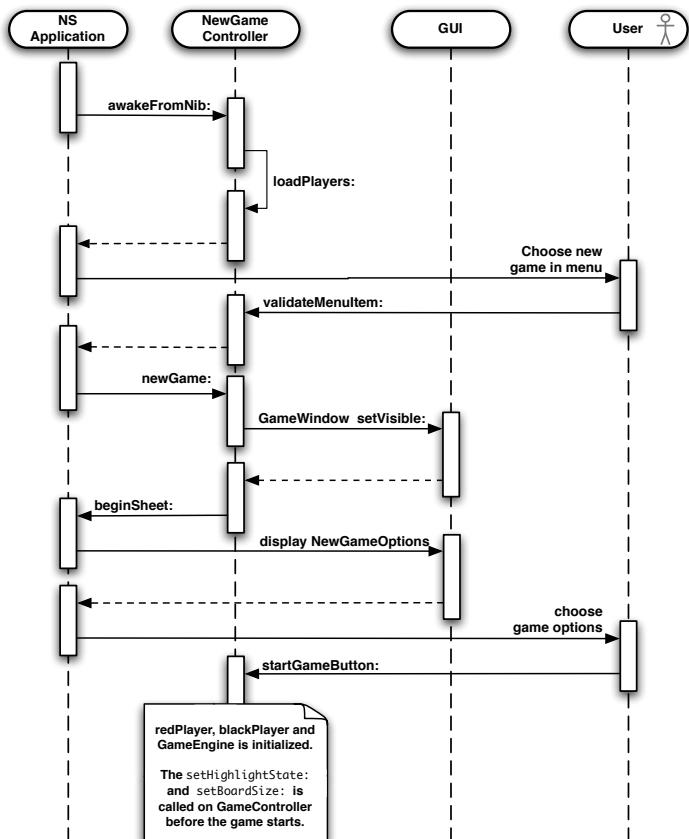


Diagram 2: UML flow diagram of game launch.

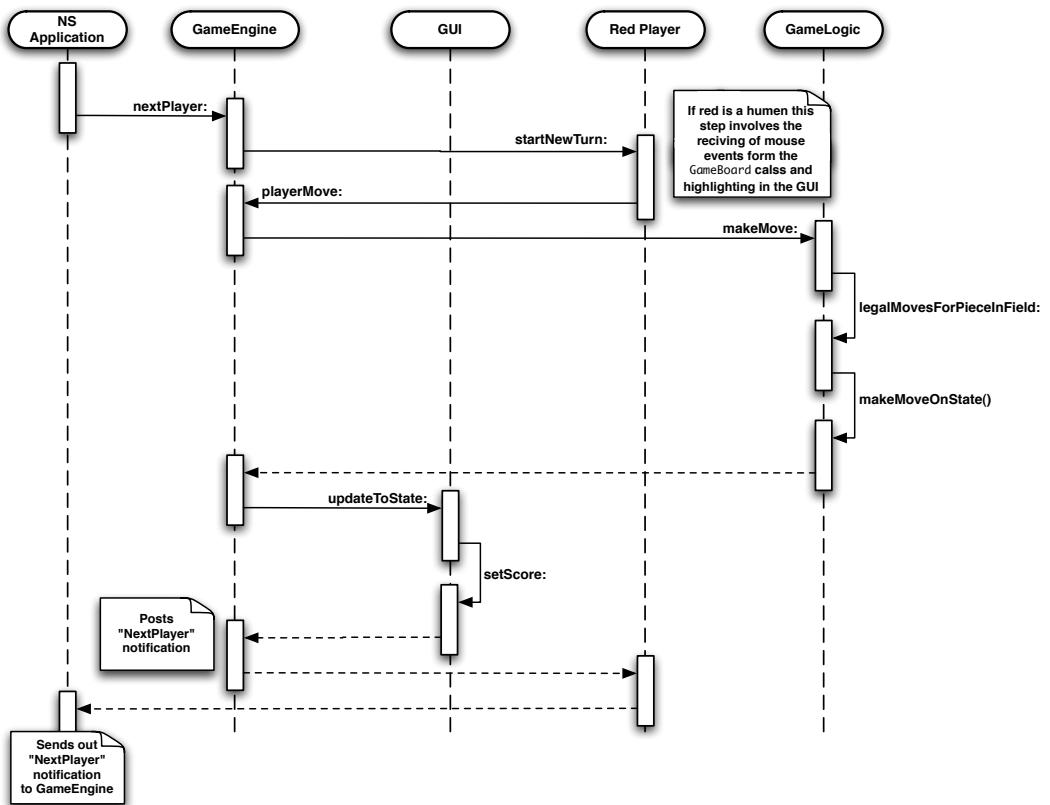


Diagram 3: UML flow diagram while playing through one turn.

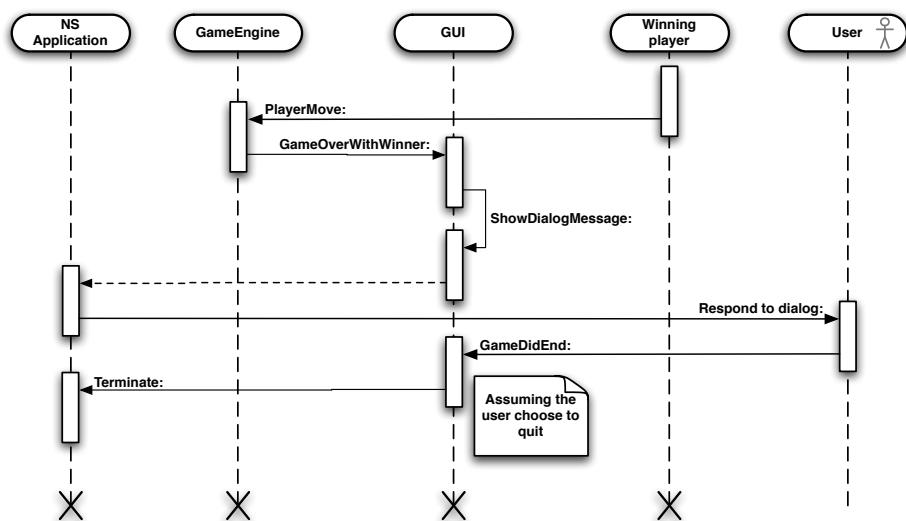


Diagram 4: UML flow diagram when the game is over.

Bibliography

- [1] Suhas Sreedhar. Checkers, solved!, 2007. URL <http://spectrum.ieee.org/print/5379>.
- [2] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314), March 1950. URL http://archive.computerhistory.org/projects/chess/related_materials/text/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon.062303002.pdf.
- [3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2. edition, 2003. ISBN 0137903952.
- [4] Armand E. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12(1-3), August 1993. URL <http://dli.iiit.ac.in/ijcai/IJCAI-91-VOL2/PDF/017.pdf>.