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

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. (**N**eXT **I**nterface **B**uilder)

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 therefor 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.

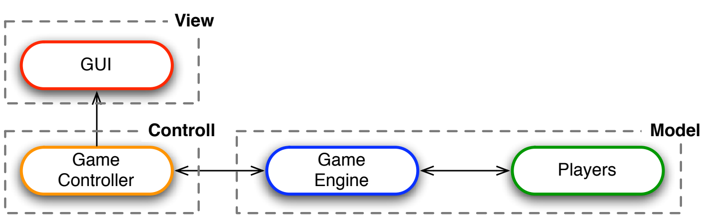


Figure 2.1: Concept Design

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 change 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 necessary 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 is 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.

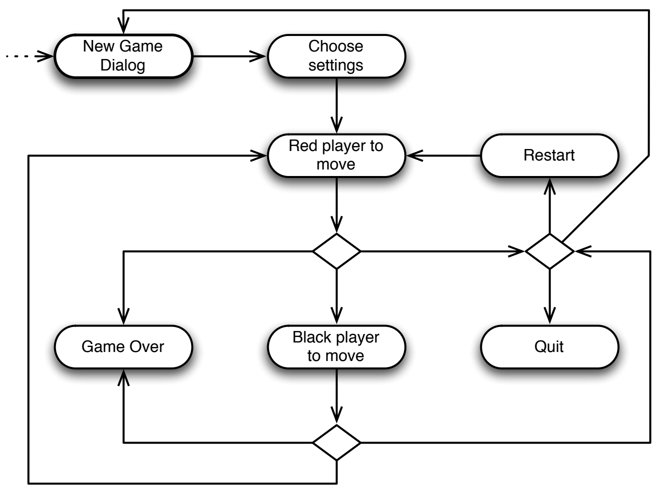


Figure 2.2: UML Flow Control Diagram.

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 fugue 2.3.

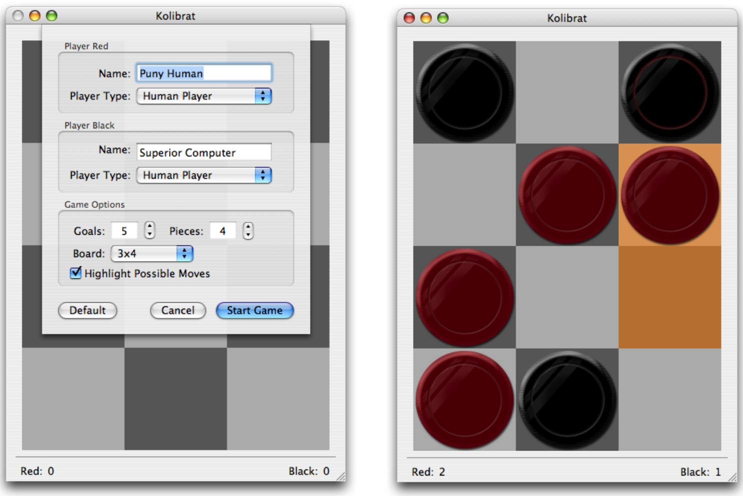


Figure 2.3: Kolibrat settings and game board.

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.

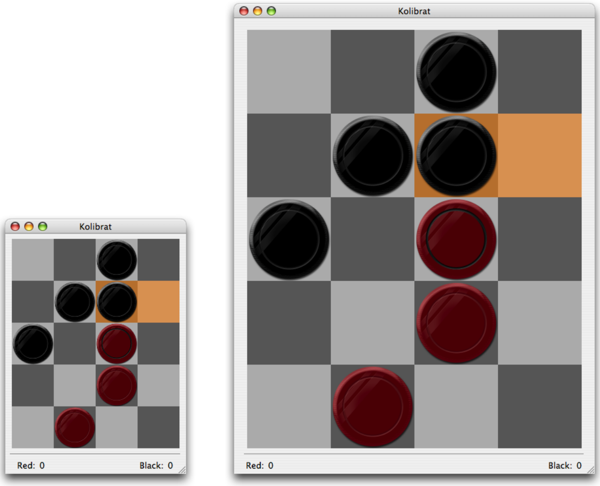


Figure 2.4: Big and small game window.

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 prat of the game has been spilt 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

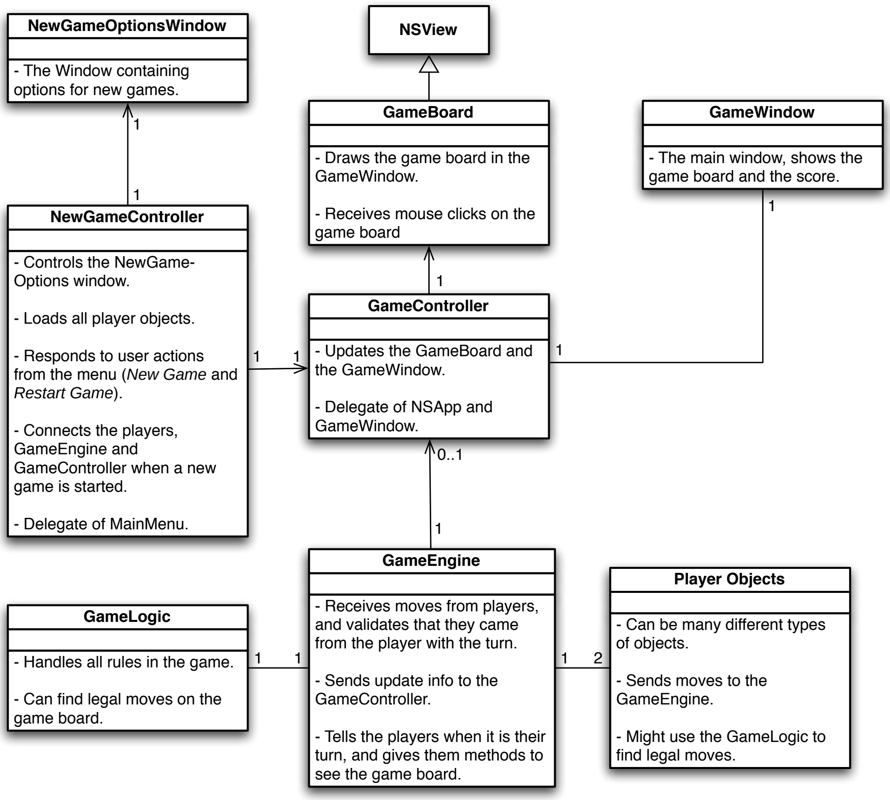


Diagram 1: Concept UML Diagram

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 is 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 the 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 a 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.

```
GameLogic
- int maxGoals
- int maxPicesOnBoard
- BoardSize boardSize

- (void)changePlayerOn:(GameState *)gs
+ (id)initWithMaxPices:(int)max goalsToWin:(int)goals boardSize:(BoardSize)board
+ (GameState)CreateNewGameState
+ (void)resetGameState:(GameState *)gs
+ (BOOL)currentPlayerCanInsertPieceOnState:(GameState *)gs
+ (NSSet *)legalMovesForPiceInField:(BoardField)field withState:(GameState *)gs
+ (NSSet *)allLegalMoves:(GameState *)gs
+ (BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs
+ (BOOL)playerMovingCanInsertPieceOnState:(GameState *)gs
+ SimpleList allLegalMoves(GameState *gs)
+ void legalMovesForPiceInField(BoardField *field, GameState *gs, SimpleList
  *superList, SimpleList *goodList,  SimpleList *badList)
+ BOOL makeMoveOnState(BoardMove *playerMove, GameState *gs)
+ void freeGameState( GameState *state )
+ GameState copyGameState( GameState *state )
+ BOOL blackPlayerAdheadOf(int x, int y, GameState *gs)
+ BOOL redPlayerAdheadOf(int x, int y, GameState *gs)
```

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.

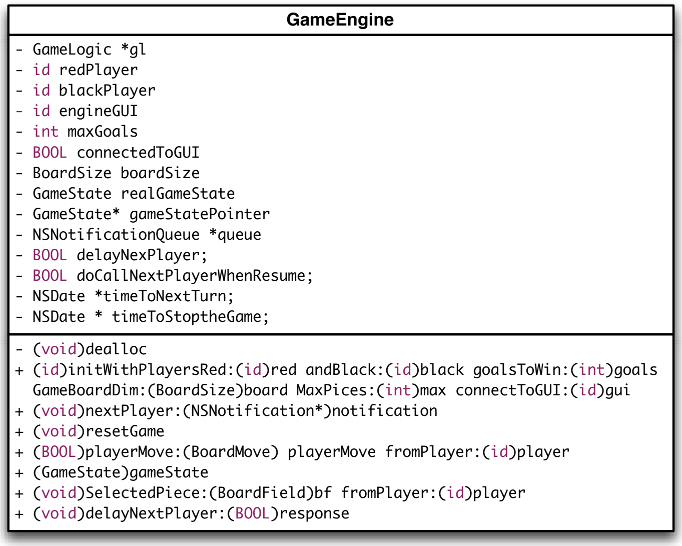


Figure 2.6: The GameEngine class

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 form this, the object only implement the methods required by the `PlayerProtocol`.

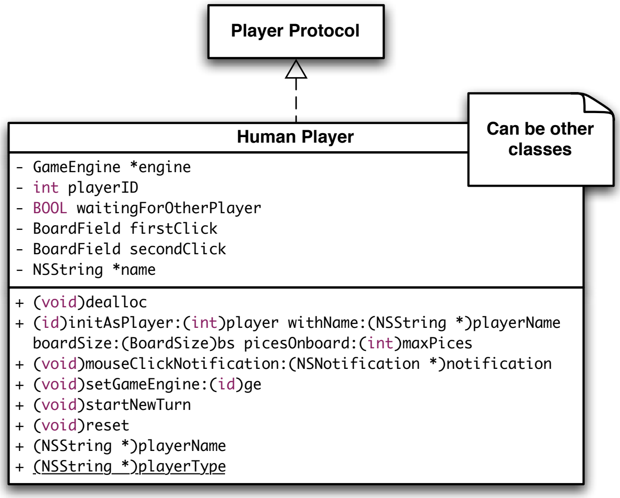


Figure 2.7: The HumanPlayer class

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.

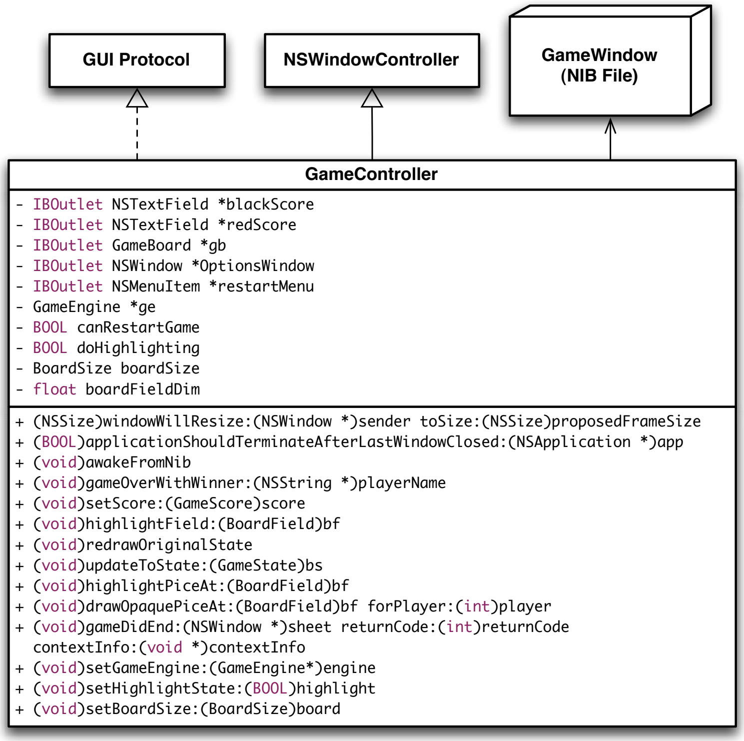


Figure 2.8: The GameController class

GameBoard

The `GameBoard` class handles drawing the game board in the main window. It to is loaded by `GameWindow` at launch. Most of its methods is 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 want 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.

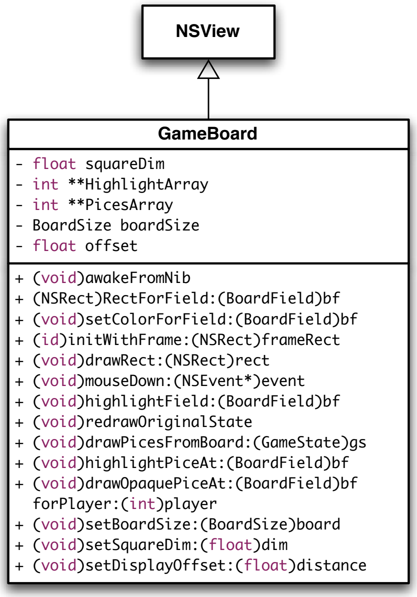


Figure 2.9: The GameBoard class

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`.

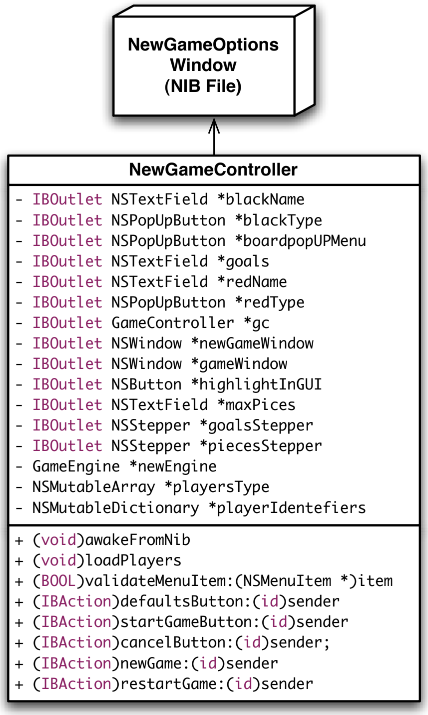


Figure 2.10: The NewGameController class

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 `gamestate` 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 3x4 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 2x16 = 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 that 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).

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

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).

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

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).

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

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 that 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 that 100 MB file.

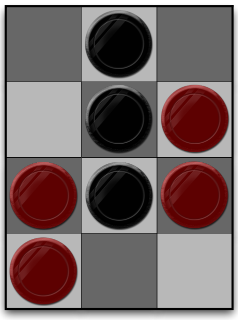
While it certainly is possible to solve the smaller Kolibrat boards completely, solving the larger is still not possible today. The game draughts have $5 \cdot 10^{20}$ unique states and have recently been solved proving that both player have a draw strategy [1]. It took 16 years from the project started 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

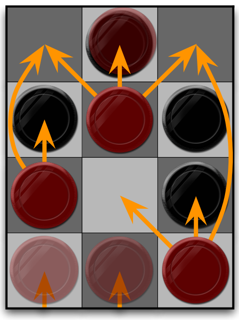
y-axis A games **branching** factor is the factor by witch its **game tree** branches, in short the number of moves the player has to chose from when he makes a move. The branching factor is important since is determines the number of states an artificial player must look through to find the best move 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).

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

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).



(a) No Moves for red.



(b) Ten moves for red.

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.

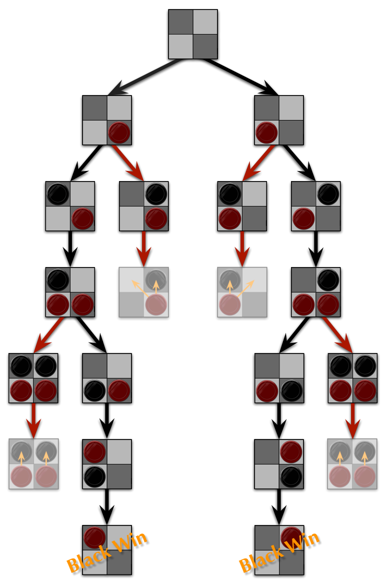


Figure 3.2: Victory strategy for black on a 2x2 board.

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.

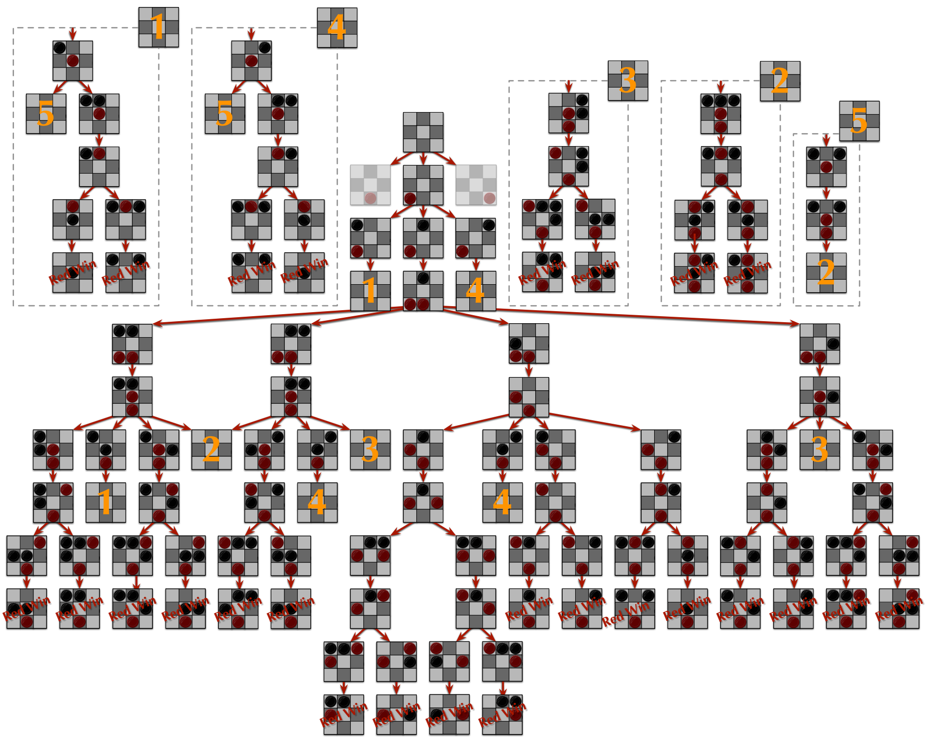


Figure 3.3: Victory strategy for red on a 3x3 board.

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 is 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

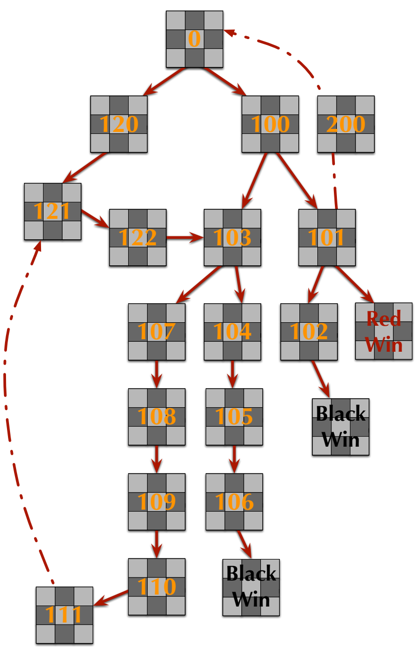


Figure 3.4: Artificial game tree from FakeLogic.

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 in infinite loop, which is the correct answer. This do 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 represents 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 form state to state.

When agents compete against each other in such an environment the agent usually uses knowledge of the games rules to look ahead in the game, trying to find a state that ensures it 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 and 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 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.

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

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. On 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.

$$\textsc{Eval} = \sum_{i=1}^{n} w_i f_i(s) \tag{3.6}$$

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 wining, 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 is 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 de 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

```
rand ← 1.0
for i = 0 to 59 do
    heuristicsArray[i] ← RandomHeuristicValues()
end for
winners ← 0
count ← 59
while rand ≥ 0 do
    i ← Rand(0, count)
    player_1 ← heuristicsArray[i]
    remove player_1 from heuristicsArray
    count−−
    i ← Rand(0, count)
    player_2 ← heuristicsArray[i]
    remove player_2 from heuristicsArray
    victoryArray ← FindWinner(player_1, player_2)
    winners++
    if winners = 30 then
        for all h heuristics in victoryArray do
            new ← NewHeuristicFrom(h, rand)
            insert new somewhere in heuristicsArray[ ]
            insert h somewhere in heuristicsArray[ ]
        end for
        remove all entries in victoryArray
        rand ← rand − 0.02
        winners ← 0
        count ← 59
    end if
end while
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

```
input: h
input: rand
new ← copy of h
for all weights w in new do
    w = w+RAND(−500, 500) · rand
end for
minval ← 0
for all weights w in new do
    minval ← MIN(w, minval)
end for
if minval < 0 then
    for all weights w in new do
        w = w + minval
    end for
end if
maxval ← 0
for all weights w in new do
    maxval ← MAX(w, maxval)
end for
if maxval > 1000 then
    for all weights w in new do
        w = w · 1000/maxval
    end for
end if
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 pice 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 by 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 Simulated Annealing AI. It uses almost all the same parameters, but the weight function is simply chosen by trail 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* i a pleasant language to work with.

In the development process the UML diagrams have been a great help. Im 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 thing 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 beet 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.

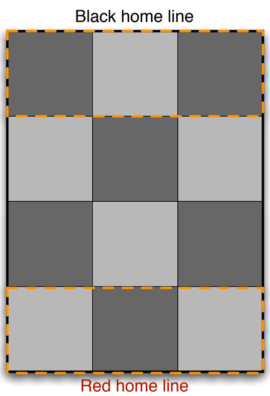


Figure A.1: Empty game board.

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 form 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 cane 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 insert into the board it can it can move forward at an angle, but not strait 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.

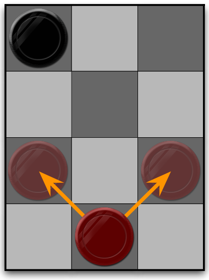


Figure A.2: Pieces can't move strait forward.

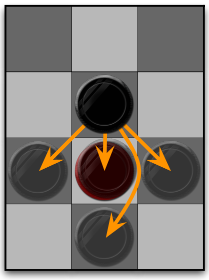


Figure A.3: Pieces can attack or jump over other pieces.

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.

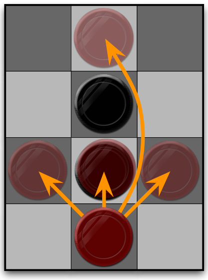


Figure A.4: Pieces can jump over multiple other pieces.

Multiple turns

In certain board positions one of the players cane 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 cant move he gains a third turn i 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 is 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 check 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

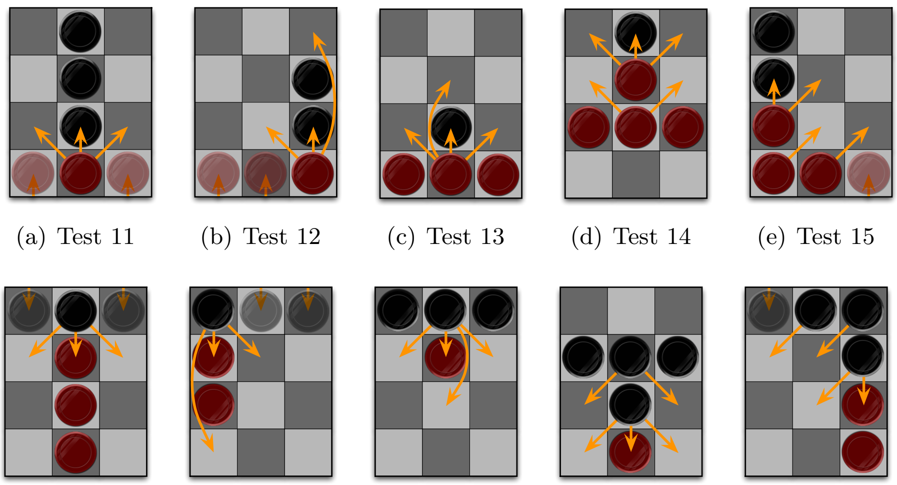
This test validates that no player can move when the game status is set to game over. The test is performed on a gameboard where red has a piece in (0,0) and black one in (2,3) red got 5 points and is the winner. Red is the next player to move. The test ensures that the move from (0,0) to (1,1) is rejected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



(a) Test 11 (b) Test 12 (c) Test 13 (d) Test 14 (e) Test 15

(f) Test 16 (g) Test 17 (h) Test 18 (i) Test 19 (j) Test 20

Figure B.1: Boards for test 11 to 20

Appendix C

Source Code

C.1 Kolibrat Source Code

C.1.1 HumanPlayer.h

```
//  Kolibrat
//  HumanPlayer.h
//
//  Created by Aron Lindberg.

#import <Cocoa/Cocoa.h>
#import "Datastructures.h"
#import "PlayerProtocol.h"
#import "GameEngine.h"

// Confirms to the Player Protocol.
@interface HumanPlayer : NSObject  < Player_Protocol >
{
// Private instance variabels.
@private
    GameEngine *engine;
    int playerID;
    BOOL waitingForOtherPlayer;
    BoardField firstClick;
    BoardField secondClick;
    NSString *name;
}
// Class Methods.
+ (NSString *)playerType;

// Public Instance Methods.
- (void)mouseClickNotification:(NSNotification *)notification;
- (id)initAsPlayer:(int)player withName:(NSString *)playerName
    boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(
    int)maxGoals;
```

```
- (void)setGameEngine:(id)ge;
- (void)reset;
- (NSString *)playerName;
- (void)startNewTurn;
@end
```

C.1.2 HumanPlayer.m

```
//  Kolibrat
//  HumanPlayer.m
//
//  Created by Aron Lindberg.

#import "HumanPlayer.h"

@implementation HumanPlayer

- (void)dealloc
{
    [name release];
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    [super dealloc];
}

- (id)initAsPlayer:(int)player withName:(NSString *)playerName boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(int)maxGoals
{
    self = [super init];
    if (self != nil)
    {
        NSNotificationCenter *mainCenter = [NSNotificationCenter defaultCenter];

        [mainCenter addObserver:self
                       selector:@selector(changeOfTurnNotification:)
                           name:@"ChangeOfTurn"
                         object:nil];

        [mainCenter addObserver:self
                       selector:@selector(mouseClickNotification:)
                           name:@"MouseClick"
                         object:nil];
        playerID = player;
        waitingForOtherPlayer = TRUE;
        firstClick = NIL_FIELD;
        secondClick = NIL_FIELD;

        name = [NSString stringWithString:playerName];
        [name retain];
    }
```

```
    return self;
}

// This method is called everytime the GUI sends a MouseClick
    notifikation.
- (void)mouseClickNotification:(NSNotification *)notification
{
    if( waitingForOtherPlayer == TRUE )
        return;

    if( BOARDFIELD_EQUALS_NIL( firstClick ))
    {
        firstClick = [[notification object] retriveField];
        [engine SelectedPiece:firstClick fromPlayer:self];
        return;
    }

    secondClick = [[notification object] retriveField];

    if([engine playerMove:makeMove( firstClick, secondClick )
        fromPlayer:self])
    {   // The move was legal.
        firstClick = NIL_FIELD;
        secondClick = NIL_FIELD;
        waitingForOtherPlayer = TRUE;
    }
    else
    {   // The move was not legal.
        firstClick = secondClick;
        [engine SelectedPiece:firstClick fromPlayer:self];
    }

}

// sets the game engine to retrive data from.
- (void)setGameEngine:(id)ge
{
    engine = ge;
}

// This method is called when a changeOfTurn Notification is sent.
- (void)startNewTurn
{
    waitingForOtherPlayer = FALSE;
}

// Resets the player, called when the human chooses "reset game"
    from the menu.
- (void)reset
{
    firstClick = NIL_FIELD;
    secondClick = NIL_FIELD;
}

// Returns the name of the player.
```

```
- (NSString *)playerName
{
    return name;
}

// Returns the name of the playerType.
+ (NSString *)playerType
{
    return @"Human Player";
}
@end
```

C.1.3 Datastructures.h

```
//  Kolibrat
//  Datastructures.h
//
//  Created by Aron Lindberg.

#import <Cocoa/Cocoa.h>

#define NIL_MOVE makeMove( makeBoardField( -1, -1 ), makeBoardField( -1, -1 ))
#define NIL_FIELD (makeBoardField( -1, -1 ))

#define BOARDFIELD_NOT_NIL( bf ) ( bf.x != -1 || bf.y != -1 )
#define BOARDFIELD_EQUALS_NIL( bf ) ( bf.x == -1 && bf.y == -1 )
#define BOARDFIELDS_IS_EQUAL( bf1, bf2 ) ( bf1.x == bf2.x && bf1.y == bf2.y )

#define GAME_RUNNING( gameStatus ) (gameStatus.gameOver == FALSE)
#define GAME_NOT_RUNNING( gameStatus ) (gameStatus.gameOver == TRUE )

#define RUNNING makeGameStatus( FALSE, FALSE )
#define RED_WON makeGameStatus( TRUE, PLAYER_RED )
#define BLACK_WON makeGameStatus( TRUE, PLAYER_BLACK )

#define BOARD_CONVERTER( bfc ) (bfc).occupiedByRed == TRUE && (bfc).occupiedByBlack == FALSE  ? PLAYER_RED : ((bfc).occupiedByRed == FALSE && (bfc).occupiedByBlack == TRUE ? PLAYER_BLACK : EMPTY)
#define EMPTY_FIELD( bfc ) (bfc).occupiedByRed == FALSE && (bfc).occupiedByBlack == FALSE
#define RED_FIELD(bfc) (bfc).occupiedByRed == TRUE && (bfc).occupiedByBlack == FALSE
#define BLACK_FIELD( bfc ) (bfc).occupiedByRed == FALSE && (bfc).occupiedByBlack == TRUE
#define NOT_BLACK_FIELD( bfc ) (bfc).occupiedByBlack == FALSE
#define NOT_RED_FIELD( bfc ) (bfc).occupiedByRed == FALSE

#define PLAYER_RED TRUE
```

```
#define PLAYER_BLACK FALSE

enum { // Constants related to drawing the game board.
    BLACK = PLAYER_BLACK,
    RED = PLAYER_RED,
    EMPTY = 3 ,
    HIGHLIGHTFIELD = 4 ,
    BLACK_HIGHLIGHT = 5 ,
    RED_HIGHLIGHT = 6 ,
    BLACK_OPAQUE = 7 ,
    RED_OPAQUE = 8 ,
};

// Some handy Typedefinitions.
typedef struct gameStatusStruct {
    BOOL gameOver;
    BOOL winner;
} GameStatus;

typedef struct boardFieldStruct {
    short int x;
    short int y;
} BoardField;

typedef struct boardMoveStruct {
    BoardField from;
    BoardField to;
} BoardMove;

typedef struct gameScoreStruct {
    short int red;
    short int black;
} GameScore;

typedef struct boardSizeStruct {
    int height;
    int width;
} BoardSize;

typedef struct boardFieldContentStruct {
    bool occupiedByRed;
    bool occupiedByBlack;
} BoardFieldContent;

typedef struct gameStateStruct {
    BoardFieldContent **board;
    GameScore score;
    BoardMove lastMove;
    GameStatus gameStatus;
    BOOL playerMoving;
    unsigned short int redPicesOnBoard;
    unsigned short int blackPicesOnBoard;
    BoardSize boardSize;
} GameState;
```

```
typedef struct simpleList {
    int ellementsInList;
    struct simpleListEllement* head;
    struct simpleListEllement* tail;
} SimpleList;

typedef struct simpleListEllement {
    struct boardMoveStruct moveData;
    struct simpleListEllement* next;
} simpleListEllement;

// Some plain C methods to make instances of the custom typedefinitions.
BoardField makeBoardField( int x, int y );
GameScore makeGameScore( int red, int black );
BoardSize makeBoardSize( int height, int width );
BoardMove makeMove( BoardField from, BoardField to );
BoardMove makeMoveFromInt( int fromx, int fromy, int tox, int toy );
GameState makeGameState( BoardSize boardSize );
GameStatus makeGameStatus( BOOL gameOver, BOOL winner );
SimpleList makeSimpleList();

BoardFieldContent makeBoardFieldWithContent( bool takenByRed, bool takenByBlack );
BoardFieldContent makeBlackField();
BoardFieldContent makeRedField();
BoardFieldContent makeEmptykField();

// Some plain C methods to work with SimpleLists.
void concatSimpleLists(SimpleList *a, SimpleList *b);
void addEllementToSimpleList(SimpleList *list, BoardMove *data);
void removeHeadFromSimpleList(SimpleList *list );
void freeSimpleList(SimpleList *list );

// Header for BoardFieldObject, this is bacically an object wrapper for the BoardField structure.
@interface BoardFieldObject : NSObject
{
@private
    BoardField board;
}
+ (id)boardfieldObjectWithField:(BoardField)bf;
- (id)initWithField:(BoardField)bf;
- (BoardField)retriveField;
@end

// Header for MoveObject, this is bacically an object wrapper for the move structure.
@interface MoveObject : NSObject
{
@private
    BoardMove m;
}
+ (id)moveObjectWithMove:(BoardMove)theMove;
```

```
- (id)initWithMove:(BoardMove)theMove;
- (BoardMove)retriveMove;
- (BOOL)isEqual:(id)anObject;
- (unsigned)hash;
@end
```

C.1.4 Datastructures.m

```
//  Kolibrat
//  Datastructures.m
//
//  Created by Aron Lindberg.

#import "Datastructures.h"

BoardFieldContent makeBoardFieldWithContent( bool takenByRed, bool
    takenByBlack )
{
    struct boardFieldContentStruct temp;
    temp.occupiedByRed = takenByRed;
    temp.occupiedByBlack = takenByBlack;
    return temp;
}

BoardFieldContent makeBlackField()
{
    struct boardFieldContentStruct temp;
    temp.occupiedByRed = FALSE;
    temp.occupiedByBlack = TRUE;
    return temp;
}

BoardFieldContent makeRedField()
{
    struct boardFieldContentStruct temp;
    temp.occupiedByRed = TRUE;
    temp.occupiedByBlack = FALSE;
    return temp;
}

BoardFieldContent makeEmptykField()
{
    struct boardFieldContentStruct temp;
    temp.occupiedByRed = FALSE;
    temp.occupiedByBlack = FALSE;
    return temp;
}

GameStatus makeGameStatus( BOOL gameOver, BOOL winner )
{
    struct gameStatusStruct temp;
    temp.gameOver = gameOver;
```

```
    temp.winner = winner;
    return temp;
}

BoardField makeBoardField( int x, int y )
{
    struct boardFieldStruct temp;
    temp.x = x;
    temp.y = y;
    return temp;
}

GameScore makeGameScore( int red, int black )
{
    struct gameScoreStruct temp;
    temp.red = red;
    temp.black = black;
    return temp;
}

BoardMove makeMove( BoardField from, BoardField to )
{
    struct boardMoveStruct temp;
    temp.from.x = from.x;
    temp.from.y = from.y;
    temp.to.x = to.x;
    temp.to.y = to.y;
    return temp;
}

BoardMove makeMoveFromInt( int fromx, int fromy, int tox, int toy )
{
    struct boardMoveStruct temp;
    temp.from.x = fromx;
    temp.from.y = fromy;
    temp.to.x = tox;
    temp.to.y = toy;
    return temp;
}

BoardSize makeBoardSize( int height, int width )
{
    struct boardSizeStruct temp;
    temp.height = height;
    temp.width = width;
    return temp;
}

SimpleList makeSimpleList()
{
    struct simpleList temp;

    temp.ellementsInList = 0 ;
    temp.head = NULL;
    temp.tail = NULL;
```

```c
    return temp;
}

void concatSimpleLists(SimpleList *a, SimpleList *b)
{
    if( b->ellementsInList == 0 )
        return;

    if( a->ellementsInList > 0 )
    {
        a->tail->next = b->head;
        a->tail = b->tail;
        a->ellementsInList += b->ellementsInList;
    }

    else
    {
        a->head = b->head;
        a->tail = b->tail;
        a->ellementsInList = b->ellementsInList;
    }
}

void addEllementToSimpleList(SimpleList *list, BoardMove *data)
{
    if( list->ellementsInList != 0 )
    {
        list->tail->next = malloc(sizeof(simpleListEllement));
        list->tail = list->tail->next;
        list->tail->moveData =  *data;
        list->ellementsInList ++;
    }
    else if( list->ellementsInList == 0 )
    {
        list->tail = malloc(sizeof(simpleListEllement));
        list->tail->moveData =  *data;
        list->head = list->tail;
        list->ellementsInList ++;
    }
}

void removeHeadFromSimpleList(SimpleList *list )
{
    if( list->ellementsInList > 1 )
    {
        simpleListEllement *nextHead = list->head->next;

        free( list->head );
        list->head = nextHead;
        list->ellementsInList --;
    }
    else if( list->ellementsInList == 1 )
    {
        free( list->head );

```

```
        list->head = NULL;
        list->tail = NULL;

        list->ellementsInList --;
    }
}

void freeSimpleList(SimpleList *list )
{

    while ( list->ellementsInList > 0 )
    {
        removeHeadFromSimpleList(list);
    }
}

GameState makeGameState( BoardSize boardSize )
{
    struct gameStateStruct temp;

    temp.score.red = 0 ;
    temp.score.black = 0 ;

    temp.blackPicesOnBoard = 0 ;
    temp.redPicesOnBoard = 0 ;

    temp.boardSize = boardSize;

    temp.playerMoving = PLAYER_RED;
    temp.gameStatus = RUNNING;

    temp.board = malloc(boardSize.width * sizeof(BoardFieldContent
        *));

    int i;
    for(i = 0 ; i < boardSize.width; i++)
    {
        temp.board[i] = malloc(boardSize.height * sizeof(
            BoardFieldContent));
    }

    int x, y;
    for ( x = 0 ; x < boardSize.width ; x++ ) {
        for ( y = 0 ; y < boardSize.height ; y++ ) {
            temp.board[x][y] = makeBoardFieldWithContent( FALSE,
                FALSE );
        }
    }
    return temp;
}

// Implamentation for the BoardField object wrapper.
@implementation BoardFieldObject

- (BoardField)retriveField
```

```
{
    return board;
}

- (id)initWithField:(BoardField)bf
{
    self = [super init];
    if (self != nil) {
        board.x = bf.x;
        board.y = bf.y;
    }
    return self;
}

+ (id)boardfieldObjectWithField:(BoardField)bf
{
    return [[BoardFieldObject alloc] initWithField:bf];
}

@end

// Implamentation for the MoveObject object wrapper.
@implementation MoveObject

- (BoardMove)retriveMove
{
    return m;
}
- (id)initWithMove:(BoardMove)theMove
{
    self = [super init];
    if (self != nil)
    {
        m.from.x = theMove.from.x;
        m.from.y = theMove.from.y;
        m.to.x = theMove.to.x;
        m.to.y = theMove.to.y;
    }
    return self;
}

// This method is used by NSSet (and others) to determine if 2
    objects is identical.
- (BOOL)isEqual:(id)anObject
{
    if( ![self isKindOfClass: [anObject class]] )
        return FALSE;

    BoardMove otherMove = [anObject retriveMove];

    if( otherMove.from.x == m.from.x &&
        otherMove.from.y == m.from.y &&
        otherMove.to.x == m.to.x &&
        otherMove.to.y == m.to.y)
        {
```

```
        return TRUE;
    }
    return false;
}

// This method is used by NSSet (and others) to determine if 2
    objects is identical.
- (unsigned)hash
{
    unsigned hashVal;
    hashVal = m.from.x + m.from.y * 10 + m.to.x * 100 + m.to.y *
        1000 ;
    return hashVal;
}

// Method to initalise object.
+ (id)moveObjectWithMove:(BoardMove)theMove
{
    return [[MoveObject alloc] initWithMove:theMove];
}

@end
```

C.1.5 GameLogic.h

```
//  Kolibrat
//  GameLogic.h
//
//  Created by Aron Lindberg.

#import "Datastructures.h"



@interface GameLogic : NSObject
{
// Private instance variabels.
@private
    int maxGoals;
    int maxPicesOnBoard;
    BoardSize boardSize;
}

// Public instance methods.
- (id)initWithMaxPices:(int)max goalsToWin:(int)goals boardSize:(
    BoardSize)board;

- (GameState)CreateNewGameState;
- (void)resetGameState:(GameState *)gs;

- (BOOL)playerMovingCanInsertPieceOnState:(GameState *)gs;
```

```
- (NSSet *)legalMovesForPiceInField:(BoardField)field withState:(
    GameState *)gs;
- (NSSet *)allLegalMoves:(GameState *)gs;

- (BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs;

// Public C instance methods.
SimpleList allLegalMoves(GameState *gs);
void legalMovesForPiceInField(BoardField *field, GameState *gs,
    SimpleList *superList, SimpleList *goodList,  SimpleList *
    badList);
BOOL makeMoveOnState(BoardMove *playerMove, GameState *gs);
void freeGameState( GameState *state );
GameState copyGameState( GameState *state);
BOOL blackPlayerAdheadOf(int x, int y, GameState *gs);
BOOL redPlayerAdheadOf(int x, int y, GameState *gs);
@end
```

C.1.6 GameLogic.m

```
//  Kolibrat
//  GameLogic.m
//
//  Created by Aron Lindberg.

#import "GameLogic.h"

@implementation GameLogic

// Global variabels, shared in al instances of GameLogic.
static int *maxGoalsPointer;
static int *maxPicesOnBoardPointer;
static BoardSize *boardSizePointer;

// Public methods.
- (id)initWithMaxPices:(int)max goalsToWin:(int)goals boardSize:(
    BoardSize)board
{
    self = [super init];
    if (self != nil)
    {
        maxPicesOnBoard = max;
        boardSize = board;
        maxGoals = goals;

        maxGoalsPointer = &maxGoals;
        maxPicesOnBoardPointer = &maxPicesOnBoard;
        boardSizePointer = &boardSize;
    }
    return self;
}

```

```
- (GameState)CreateNewGameState
{
    GameState gs = makeGameState(boardSize);
    [self resetGameState: &gs];
    return gs;
}

- (void)resetGameState:(GameState *)gs
{
    int x, y;
    for ( x = 0 ; x < boardSize.width ; x++ ) {
        for ( y = 0 ; y < boardSize.height ; y++ ) {
            gs->board[x][y] = makeEmptykField();
        }
    }
    gs->score = makeGameScore( 0 , 0 );
    gs->playerMoving = PLAYER_RED;
    gs->redPicesOnBoard = 0 ;
    gs->blackPicesOnBoard = 0 ;
    gs->gameStatus = RUNNING;
}

- (BOOL)playerMovingCanInsertPieceOnState:(GameState *)gs;
{
    if( gs->playerMoving == PLAYER_RED && gs->redPicesOnBoard <
        maxPicesOnBoard )
        return TRUE;
    if( gs->playerMoving == PLAYER_BLACK && gs->blackPicesOnBoard <
         maxPicesOnBoard )
        return TRUE;

    return FALSE;
}

- (NSSet *)legalMovesForPiceInField:(BoardField)field withState:(
    GameState *)gs
{
    if(field.x < 0 || field.y < 0 || field.x > boardSize.width - 1
        || field.y > boardSize.height - 1 )
    {
        NSException* e = [NSException exceptionWithName:@"Move
            error." reason:[NSString stringWithFormat:@"The board
            field (%i,%i) is outside of the board.", field.x,
            field.y] userInfo:nil];
        @throw e;
    }

    NSMutableSet *setOfLegalMoves = [NSMutableSet setWithCapacity:4
         ];

    SimpleList superList = makeSimpleList();
    SimpleList goodList = makeSimpleList();
    SimpleList badList = makeSimpleList();

```

```
    legalMovesForPiceInField( &field, gs, &superList, &goodList, &badList);

    concatSimpleLists( &superList, &goodList );
    concatSimpleLists( &superList, &badList );

    while( superList.ellementsInList > 0 )
    {
        [setOfLegalMoves addObject: [MoveObject moveObjectWithMove: superList.head->moveData]];
        removeHeadFromSimpleList( &superList);
    }

    return setOfLegalMoves;
}

- (NSSet *)allLegalMoves:(GameState *)gs
{
    NSMutableSet *setOfLegalMoves = [NSMutableSet setWithCapacity: 10 ];

    SimpleList list = allLegalMoves(gs);

    while( list.ellementsInList > 0 )
    {
        MoveObject *mo = [MoveObject moveObjectWithMove: list.head->moveData];

        [setOfLegalMoves addObject: mo];
        [mo release];
        removeHeadFromSimpleList( &list);
    }

    freeSimpleList( &list );

    return setOfLegalMoves;
}

SimpleList allLegalMoves(GameState *gs)
{
    SimpleList superList = makeSimpleList();
    SimpleList goodList = makeSimpleList();
    SimpleList badList = makeSimpleList();

    int x, y;
    for ( x = 0 ; x < boardSizePointer->width ; x++ )
    {
        for ( y = 0 ; y < boardSizePointer->height ; y++ )
        {
            BoardField boardField = makeBoardField( x, y );
            legalMovesForPiceInField( &boardField , gs, &superList, &goodList, &badList);
        }
    }

```

```
    concatSimpleLists( &superList, &goodList );
    concatSimpleLists( &superList, &badList );

    return superList;
}

- (BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs
{
    BoardField from = playerMove.from;
    BoardField to = playerMove.to;

    if(from.x < 0 || from.y < 0 || from.x > boardSize.width - 1 ||
        from.y > boardSize.height - 1 )
    {
        NSException* e = [NSException exceptionWithName:@"Move
            error." reason:[NSString stringWithFormat:@"The board
            field (%i,%i) is outside of the board.", from.x, from.y
            ] userInfo:nil];
        @throw e;
    }

    if(to.x < 0 || to.y < 0 || to.x > boardSize.width - 1 || to.y >
         boardSize.height - 1 )
    {
        NSException* e = [NSException exceptionWithName:@"Move
            error." reason:[NSString stringWithFormat:@"The board
            field (%i,%i) is outside of the board.", to.x, to.y]
            userInfo:nil];
        @throw e;
    }

    NSSet *setOfLegalMoves = [self legalMovesForPiceInField:from
        withState:gs];

    if( [setOfLegalMoves containsObject:[MoveObject
        moveObjectWithMove:playerMove]] )
    {   // The move is legal.
        return makeMoveOnState( &playerMove, gs);
    }
    return FALSE;
}

BOOL makeMoveOnState(BoardMove *playerMove, GameState *gs)
{

    if( gs->gameStatus.gameOver == TRUE )
        return FALSE;

    BoardField from = playerMove->from;
    BoardField to = playerMove->to;

    gs->lastMove.from = playerMove->from;
    gs->lastMove.to = playerMove->to;

```

```
    if( BOARDFIELDS_IS_EQUAL( from, to ) && gs->playerMoving ==
        PLAYER_RED && to.y == 0 )
    {   // Insert peice for red.
        gs->board[to.x][to.y] = makeRedField();
        gs->redPicesOnBoard ++;
    }

    else if( BOARDFIELDS_IS_EQUAL( from, to ) && gs->playerMoving
        == PLAYER_RED && to.y == ( boardSizePointer->height - 1 ))
    {   // Score point for red.
        gs->board[to.x][to.y] = makeEmptykField();
        gs->score.red ++;
        gs->redPicesOnBoard --;
        if( gs->score.red == *maxGoalsPointer )
        {
            gs->gameStatus = RED_WON;
        }
    }

    else if( BOARDFIELDS_IS_EQUAL( from, to ) && gs->playerMoving
        == PLAYER_BLACK && to.y == ( boardSizePointer->height - 1 )
        )
    {   // Insert peice for black.
        gs->board[to.x][to.y] = makeBlackField();
        gs->blackPicesOnBoard ++;
    }

    else if( BOARDFIELDS_IS_EQUAL( from, to ) && gs->playerMoving
        == PLAYER_BLACK && to.y == 0 )
    {   // Score point for black.
        gs->board[to.x][to.y] = makeEmptykField();
        gs->score.black ++;
        gs->blackPicesOnBoard --;
        if( gs->score.black == *maxGoalsPointer )
        {
            gs->gameStatus = BLACK_WON;
        }
    }

    else
    {   // If the move is an attack.
        if( RED_FIELD( gs->board[to.x][to.y] ))
            gs->redPicesOnBoard-- ;
        if( BLACK_FIELD( gs->board[to.x][to.y] ))
            gs->blackPicesOnBoard-- ;

        // Move piece on the board.
        gs->board[from.x][from.y] = makeEmptykField();
        if( gs->playerMoving == PLAYER_RED )
            gs->board[to.x][to.y] = makeRedField();
        else if( gs->playerMoving == PLAYER_BLACK )
            gs->board[to.x][to.y] = makeBlackField();
    }

    if( gs->gameStatus.gameOver == TRUE )
```

```
    {
        return TRUE;
    }

    // finds the next player.
    BOOL nextPlayer;
    BOOL otherPlayer;

    if( gs->playerMoving == PLAYER_RED )
    {
        nextPlayer = PLAYER_BLACK;
        otherPlayer = PLAYER_RED;
    }
    else if( gs->playerMoving == PLAYER_BLACK )
    {
        nextPlayer = PLAYER_RED;
        otherPlayer = PLAYER_BLACK;
    }

    gs->playerMoving = nextPlayer;

    SimpleList allMoves = makeSimpleList();
    allMoves = allLegalMoves( gs );

    if ( allMoves.ellementsInList == 0 )
    {
        gs->playerMoving = otherPlayer;
        allMoves = allLegalMoves( gs );

        if( allMoves.ellementsInList == 0 )
        {   // No player can move, playerMoving has lost.
            if( gs->playerMoving == PLAYER_RED)
                gs->gameStatus = BLACK_WON;
            else
                gs->gameStatus = RED_WON;
        }
    }
    freeSimpleList(&allMoves);

    return TRUE;
}

GameState copyGameState( GameState *state)
{
    struct gameStateStruct temp;

    temp.blackPicesOnBoard = state->blackPicesOnBoard;
    temp.redPicesOnBoard = state->redPicesOnBoard;
    temp.playerMoving = state->playerMoving;
    temp.gameStatus = state->gameStatus;
    temp.score = state->score;
    temp.lastMove = state->lastMove;

    temp.boardSize = state->boardSize;

```

```
    temp.board = malloc(boardSizePointer->width * sizeof(
        BoardFieldContent *));

    int i;
    for(i = 0 ; i < boardSizePointer->width; i++)
    {
        temp.board[i] = malloc(boardSizePointer->height * sizeof(
            BoardFieldContent));
    }

    int x, y;
    for ( x = 0 ; x < boardSizePointer->width ; x++ ) {
        for ( y = 0 ; y < boardSizePointer->height ; y++ ) {
            temp.board[x][y] = state->board[x][y];
        }
    }
    return temp;
}

void freeGameState( GameState *state )
{
    int i;
    for(i = 0 ; i < boardSizePointer->width; i++)
    {
        free(state->board[i]);
    }

    free(state->board);
}

void legalMovesForPiceInField(BoardField *field, GameState *gs,
    SimpleList *superList, SimpleList *goodList,  SimpleList *
    badList)
{

    if( gs->gameStatus.gameOver == TRUE )
        return;

    register int x = field->x;
    register int y = field->y;

    // All moving rules for the red player.
    if( gs->playerMoving == PLAYER_RED )
    {
        // Red wants to insert a piece on the board.
        if( EMPTY_FIELD( gs -> board[x][y] ) && y == 0 && gs->
            redPicesOnBoard < *maxPicesOnBoardPointer )
        {
            BoardMove move = makeMoveFromInt( x, y, x , y);

            if( blackPlayerAdheadOf( x, y, gs ))
                addEllementToSimpleList( badList, &move );
            else
                addEllementToSimpleList( goodList, &move );
        }
```

```

        // Red wants to move left and forward.
        if( RED_FIELD( gs->board[x][y] ) && x > 0 && y < ( boardSizePointer->height - 1 )
            && EMPTY_FIELD( gs->board[x - 1 ][y + 1 ] ))
        {
            BoardMove move = makeMoveFromInt( x, y, x - 1, y + 1 );

            if( blackPlayerAdheadOf( x, y, gs ))
                addEllementToSimpleList( badList, &move );
            else
                addEllementToSimpleList( goodList, &move );
        }

        // Red wants to move right and forward.
        if( RED_FIELD( gs->board[x][y] ) && x < ( boardSizePointer->width - 1 ) &&
            y < ( boardSizePointer->height - 1 ) && EMPTY_FIELD( gs->board[x + 1 ][y + 1 ] ))
        {
            BoardMove move =makeMoveFromInt( x, y, x + 1, y + 1 );

            if( blackPlayerAdheadOf( x, y, gs ))
                addEllementToSimpleList( badList, &move );
            else
                addEllementToSimpleList( goodList, &move );
        }

        // Red wants to attack black.
        if( RED_FIELD( gs->board[x][y] ) && y < ( boardSizePointer->height - 1 ) &&
            BLACK_FIELD( gs->board[x][y + 1 ] ))
        {
            BoardMove move = makeMoveFromInt( x, y, x, y + 1 );
            addEllementToSimpleList( superList,     &move );
        }

        // Red wants to jump over black.
        if( RED_FIELD( gs->board[x][y] ) && y < ( boardSizePointer->height - 2 ) &&
            BLACK_FIELD( gs->board[x][y + 1 ] ))
        {
            int jumpDistance = 1 ;
            while( y + jumpDistance < (boardSizePointer->height - 2 ) &&
                   BLACK_FIELD( gs->board[x][y + 1 + jumpDistance] ))
            {
                jumpDistance ++;
            }
            if( EMPTY_FIELD(gs->board[x][y + 1 + jumpDistance]))
            {
                BoardMove move = makeMoveFromInt( x, y, x, y + 1 + jumpDistance );
                addEllementToSimpleList( superList, &move );
```

```
            }
        }

        // Red wants to gain a point.
        if( RED_FIELD( gs->board[x][y] ) && y == (
            boardSizePointer->height - 1 ))
        {
            BoardMove move =makeMoveFromInt( x, y, x , y);
            addEllementToSimpleList( superList, &move );
        }
    }

    // All moving rules for the black player.
    if( gs->playerMoving == PLAYER_BLACK )
    {
        // Black wants to insert a piece on the board.
        if( EMPTY_FIELD( gs->board[x][y] ) && y == (
            boardSizePointer->height - 1 )
            && gs->blackPicesOnBoard < *maxPicesOnBoardPointer )
        {
            BoardMove move = makeMoveFromInt( x, y, x , y);

            if( redPlayerAdheadOf( x, y, gs ))
                addEllementToSimpleList( badList, &move );
            else
                addEllementToSimpleList( goodList, &move );
        }

        // Black wants to move left and forward.
        if( BLACK_FIELD( gs->board[x][y] ) && x > 0 && y > 0 &&
            EMPTY_FIELD( gs->board[x - 1 ][y - 1 ] ))
        {
            BoardMove move = makeMoveFromInt( x, y, x - 1, y - 1 );

            if( redPlayerAdheadOf( x, y, gs ))
                addEllementToSimpleList( badList, &move );
            else
                addEllementToSimpleList( goodList, &move );
        }

        // Black wants to move right and forward.
        if( BLACK_FIELD( gs->board[x][y] ) && x < (
            boardSizePointer->width - 1 )
            && y > 0 && EMPTY_FIELD( gs->board[x + 1 ][y - 1 ] ))
        {
            BoardMove move = makeMoveFromInt( x, y, x + 1, y - 1 );

            if( redPlayerAdheadOf( x, y, gs ))
                addEllementToSimpleList( badList, &move );
            else
                addEllementToSimpleList( goodList, &move );
        }

        // Black wants to attack red.
        if( BLACK_FIELD( gs->board[x][y] ) && y > 0 &&
```

```
                RED_FIELD( gs->board[x][y - 1 ] ))
            {
                BoardMove move = makeMoveFromInt( x, y, x, y - 1 );
                addEllementToSimpleList( superList, &move );
            }

            // Black wants to jump over red.
            if( BLACK_FIELD( gs->board[x][y] ) && y > 1 &&
                RED_FIELD( gs->board[x][y - 1 ] ))
            {
                int jumpDistance = 1 ;
                while( (y - 1 ) - jumpDistance > 0 &&
                       RED_FIELD( gs->board[x][( y - 1 ) - jumpDistance
                           ] ))
                {
                    jumpDistance ++;
                }
                if( EMPTY_FIELD( gs->board[x][( y - 1 ) - jumpDistance]
                     ))
                {
                    BoardMove move = makeMoveFromInt( x, y, x, ( y - 1
                        ) - jumpDistance );
                    addEllementToSimpleList( superList, &move );
                }
            }

            // Black wants to gain a point.
            if( BLACK_FIELD( gs->board[x][y] ) && y == 0 )
            {
                BoardMove move = makeMoveFromInt( x, y, x, y );
                addEllementToSimpleList( superList, &move );
            }
        }
}

// Helper methods.
BOOL blackPlayerAdheadOf(int x, int y, GameState *gs)
{
    if( BLACK_FIELD( gs->board[x][y + 1 ] ))
        return TRUE;
    else
        return FALSE;
}

BOOL redPlayerAdheadOf(int x, int y, GameState *gs)
{
    if( BLACK_FIELD( gs->board[x][y - 1 ] ))
        return TRUE;
    else
        return FALSE;
}

@end
```

C.1.7 GameEngine.h

```
//   Kolibrat
//   GameEngine.h
//
//   Created by Aron Lindberg.

#import "Datastructures.h"
#import "GUIProtocol.h"
#import "PlayerProtocol.h"
#import "GameLogic.h"

//#define TIMEOUT [NSDate dateWithTimeIntervalSinceNow:400 ]
#define TIMEOUT [NSDate distantFuture]

@interface GameEngine : NSObject
{
// Private instance variabels.
@private
    id redPlayer;
    id blackPlayer;
    id engineGUI;
    int maxGoals;
    BOOL delayNexPlayer;
    BOOL doCallNextPlayerWhenResume;

    BoardSize boardSize;
    GameState realGameState;
    GameState *gameStatePointer;

    NSNotificationQueue *queue;
    GameLogic *gl;

    NSDate *timeToNextTurn;
    NSDate * timeToStoptheGame;
}

// Public methods.
- (id)initWithPlayersRed:(id)red andBlack:(id)black goalsToWin:(int
    )goals GameBoardDim:(BoardSize)board MaxPices:(int)max
    connectToGUI:(id)gui;
- (void)resetGame;

// Methods used by the Player Objects.
- (BOOL)playerMove:(BoardMove)playerMove fromPlayer:(id)player;
- (GameState)gameState;
- (void)SelectedPiece:(BoardField)bf fromPlayer:(id)player;

// Methods used by the GUI.
- (void)delayNextPlayer:(BOOL)response;

@end
```

C.1.8 GameEngine.m

```
//  Kolibrat
//  GameEngine.m
//
//  Created by Aron Lindberg.

#import "GameEngine.h"

@implementation GameEngine

// Called when the object instance is destroyed.
- (void)dealloc
{
    // Release all child objects used by the GameEngine.
    [gl release];
    [redPlayer release];
    [blackPlayer release];

    [[NSNotificationCenter defaultCenter] removeObserver:self];

    [super dealloc];
}

// invoked when the GameEngine is initialized.
- (id)initWithPlayersRed:(id)red andBlack:(id)black goalsToWin:(int
    )goals GameBoardDim:(BoardSize)board MaxPices:(int)max
    connectToGUI:(id)gui
{
    if ((self = [super init] ) != nil) {

        doCallNextPlayerWhenResume = NO;
        delayNexPlayer = NO;

        gl = [[GameLogic alloc] initWithMaxPices:max goalsToWin:
            goals boardSize:board];
        realGameState = [gl CreateNewGameState];
        gameStatePointer = &realGameState;

        queue = [NSNotificationQueue defaultQueue];
        NSNotificationCenter *mainCenter = [NSNotificationCenter
            defaultCenter];

        [mainCenter addObserver:self selector:@selector(nextPlayer
            :)name:@"NextPlayer" object:nil];

        boardSize.width = board.width;
        boardSize.height = board.height;

        redPlayer = red;
        blackPlayer = black;
        maxGoals = goals;

        engineGUI = gui;
```

```

        timeToNextTurn = [NSDate date];
        [timeToNextTurn retain];

        [engineGUI updateToState:  copyGameState( gameStatePointer ) ];

        if( ![engineGUI conformsToProtocol:@protocol(GUI_Protocol)] && engineGUI != nil )
        {
            NSException* e = [NSException exceptionWithName:@"GUI Error." reason:@"The GUI does not responds to needed method calls." userInfo:nil];
            @throw e;
        }

        // Ensure that red player responds to nesseary method cals.
        if( ![redPlayer conformsToProtocol:@protocol( Player_Protocol )])
        {
            NSException* e = [NSException exceptionWithName:@"Player Error" reason:@"The Player does not responds to needed method calls." userInfo:nil];
            @throw e;
        }

        // Ensure that black player responds to nesseary method cals.
        if( ![blackPlayer conformsToProtocol:@protocol( Player_Protocol )])
        {
            NSException* e = [NSException exceptionWithName:@"Player Error" reason:@"The Player does not responds to needed method calls." userInfo:nil];
            @throw e;
        }
    }

    // Send message that next player should be given the turn.
    NSNotification *message = [NSNotification notificationWithName:@"NextPlayer" object:nil];
    [queue enqueueNotification:message postingStyle:NSPostWhenIdle ];

    timeToStoptheGame = TIMEOUT;
    [timeToStoptheGame retain];

    return self;
}

// Givs turn to next player.
- (void)nextPlayer:(NSNotification *)notification
{
    [NSThread sleepUntilDate: timeToNextTurn];
    [timeToNextTurn release];
```

```
    // Delay to ensure that the game is watcheble when to AI's play
         each other.
    timeToNextTurn = [NSDate dateWithTimeIntervalSinceNow:0.5 ];
    [timeToNextTurn retain];

    if( delayNexPlayer == NO )
    {
        if( realGameState.playerMoving == PLAYER_RED )
            [redPlayer startNewTurn];
        else if( realGameState.playerMoving == PLAYER_BLACK )
            [blackPlayer startNewTurn];
    }
    else
    {
        doCallNextPlayerWhenResume = YES;
    }
}

// Resets the games state.
- (void)resetGame
{
    [gl resetGameState: gameStatePointer ];

    [redPlayer reset];
    [blackPlayer reset];

    if( engineGUI != nil )
    {
        [engineGUI updateToState:  copyGameState( gameStatePointer
            ) ];
    }

    // Send message that next player should be given the turn.
    NSNotification *message = [NSNotification notificationWithName:
        @"NextPlayer" object:nil];
    [queue enqueueNotification:message postingStyle:NSPostWhenIdle
        ];

    timeToStoptheGame = TIMEOUT;
    [timeToStoptheGame retain];
}

// Called by the player objects when they wish to make a move.
- (BOOL)playerMove:(BoardMove)playerMove fromPlayer:(id)player
{
    if( GAME_NOT_RUNNING( realGameState.gameStatus ))// The game is
         not in play.
        return FALSE;

    BOOL returnValue;

    if( realGameState.playerMoving == PLAYER_RED && redPlayer ==
        player )
```

```
    {   // This method call came from red player, and red has the turn.
        returnValue = [gl makeMove:playerMove withState: &realGameState];
    }

    else if( realGameState.playerMoving == PLAYER_BLACK && blackPlayer == player )
    {   // This method call came from black player, and black has the turn.
        returnValue = [gl makeMove:playerMove withState: &realGameState];
    }

    else
    {   // The call is not legal, the player don't have the turn.
        return FALSE;
    }

    if( engineGUI != nil ) // Update the GUI.
        [engineGUI updateToState:  copyGameState( gameStatePointer ) ];

    // Check if the game is over.
    if( realGameState.gameStatus.gameOver == TRUE )
    {
        // Check if red won.
        if( engineGUI != nil && realGameState.gameStatus.winner == PLAYER_RED )
            [engineGUI gameOverWithWinner: [redPlayer playerName]];

        // Check if black won
        if( engineGUI != nil && realGameState.gameStatus.winner == PLAYER_BLACK )
            [engineGUI gameOverWithWinner: [blackPlayer playerName]];

        NSNotification *message;
        if( realGameState.gameStatus.winner == PLAYER_RED )
            message = [NSNotification notificationWithName:@"GameOver" object:redPlayer];
        else
            message = [NSNotification notificationWithName:@"GameOver" object:blackPlayer];

        [queue enqueueNotification:message postingStyle:NSPostWhenIdle];

        // Return now do not start a new turn.
        return returnValue;
    }

    // This code stops the game if timeToStoptheGame is set. Used by the Simulated Annealing program.
    if( [timeToStoptheGame timeIntervalSinceNow] <= 0 )
```

```
    {
        [timeToStoptheGame release];
        NSNotification *message;
        if( realGameState.score.red > realGameState.score.black )
        {
            message = [NSNotification notificationWithName:@"GameOver" object:redPlayer];
            [queue enqueueNotification:message postingStyle:NSPostWhenIdle];
        }
        else if( realGameState.score.red < realGameState.score.black )
        {
            message = [NSNotification notificationWithName:@"GameOver" object:blackPlayer];
            [queue enqueueNotification:message postingStyle:NSPostWhenIdle];
        }
        else
        {
            NSNotification *message;

            srandom([[NSDate date] timeIntervalSince1970 ]);
            if( random() % 100 > 49 )
            {
                message = [NSNotification notificationWithName:@"GameOver" object:redPlayer];
                [queue enqueueNotification:message postingStyle:NSPostWhenIdle];
            }
            else
            {
                message = [NSNotification notificationWithName:@"GameOver" object:blackPlayer];
                [queue enqueueNotification:message postingStyle:NSPostWhenIdle];
            }
        }
        return returnValue;
    }
    NSNotification *message = [NSNotification notificationWithName:@"NextPlayer" object:nil];
    [queue enqueueNotification:message postingStyle:NSPostWhenIdle];

    return returnValue;
}

// Returns the game state.
- (GameState)gameState
{
    return copyGameState( gameStatePointer );
}

```

```
// This method is called by the players. This tells the engine that the player
// wants to highlight this piece in the GUI.
- (void)SelectedPiece:(BoardField)bf fromPlayer:(id)player
{
    if( engineGUI == nil || GAME_NOT_RUNNING( realGameState.gameStatus ))
        return;

    if(bf.x < 0 || bf.y < 0 || bf.x > boardSize.width - 1 || bf.y > boardSize.height - 1 )
    {
        NSException* e = [NSException exceptionWithName:@"Move error." reason:[NSString stringWithFormat:@"The board field (%i,%i) is outside of the board.", bf.x, bf.y] userInfo:nil];
        @throw e;
    }

    GameState* gsp = gameStatePointer;

    int x = bf.x;
    int y = bf.y;

    // Rules for Red Player.
    if( realGameState.playerMoving == PLAYER_RED && redPlayer == player)
    {
        // The user is about to insert a peice.
        if( y == 0 && EMPTY_FIELD(gsp->board[x][y]) && [gl playerMovingCanInsertPieceOnState: gsp] )
            [engineGUI drawOpaquePiceAt:bf forPlayer:PLAYER_RED];

        // The user is about to gain a point.
        if( y == (boardSize.height - 1 ) && RED_FIELD( gsp->board[x][y] ))
            [engineGUI drawOpaquePiceAt:bf forPlayer:PLAYER_RED];

        // The user wants to highlight a piece.
        if( y != (boardSize.height - 1 ) && RED_FIELD( gsp->board[x][y]))
        {
            [engineGUI highlightPiceAt:bf];

            // Now highlighting possible moves for that piece.
            NSSet *setOfMoves = [gl legalMovesForPiceInField:bf withState: gameStatePointer];
            NSEnumerator *e = [setOfMoves objectEnumerator];
            MoveObject *thisMoveObject;

            while (thisMoveObject = [e nextObject])
            {
                BoardMove thisMove = [thisMoveObject retriveMove];
                [engineGUI highlightField: thisMove.to ];
            }
```

```
        }
    }

    // Rules for Black Player.
    if( realGameState.playerMoving == PLAYER_BLACK && blackPlayer
        == player)
    {
        // The user is about to insert a peice.
        if( y == (boardSize.height - 1 ) && EMPTY_FIELD( gsp->board
            [x][y] ) && [gl playerMovingCanInsertPieceOnState: gsp]
             )
            [engineGUI drawOpaquePiceAt:bf forPlayer:PLAYER_BLACK];

        // The user is about to gain a point.
        if( y == 0 && BLACK_FIELD( gsp->board[x][y] ))
            [engineGUI drawOpaquePiceAt:bf forPlayer:PLAYER_BLACK];

        // The user wants to highlight a piece.
        if( y != 0 && BLACK_FIELD( gsp->board[x][y] ))
        {
            [engineGUI highlightPiceAt:bf];

            // Now highlighting possible moves for that piece.
            NSSet *setOfMoves = [gl legalMovesForPiceInField:bf
                withState: gsp];
            NSEnumerator *e = [setOfMoves objectEnumerator];
            MoveObject *thisMoveObject;

            while (thisMoveObject = [e nextObject])
            {
                BoardMove thisMove = [thisMoveObject retriveMove];
                [engineGUI highlightField: thisMove.to ];
            }
        }
    }
}

// Called by the GUI when the new game sheet is shown.
- (void)delayNextPlayer:(BOOL)response
{
    if( response == YES )
    {
        delayNexPlayer = YES;
    }

    else if( response == NO )
    {
        delayNexPlayer = NO;
        if( doCallNextPlayerWhenResume == YES )
        {
            doCallNextPlayerWhenResume = NO;
            NSNotification *message = [NSNotification
                notificationWithName:@"NextPlayer" object:nil];
            [queue enqueueNotification:message postingStyle:
                NSPostWhenIdle];
```

```
        }
    }
    return;
}

@end
```

C.1.9 GameController.h

```
//  Kolibrat
//  GameController.h
//
//  Created by Aron Lindberg.

#import "Datastructures.h"
#import "GUIProtocol.h"
#import "GameBoard.h"
#import "GameEngine.h"

// Confirms to the GUI Protocol.
@interface GameController : NSWindowController < GUI_Protocol >
{
// Private instance variabels.
@private
    IBOutlet NSTextField *blackScore;
    IBOutlet NSTextField *redScore;
    IBOutlet GameBoard *gb;
    IBOutlet NSWindow *OptionsWindow;
    IBOutlet NSMenuItem *restartMenu;
    GameEngine *ge;
    BOOL canRestartGame;
    BOOL doHighlighting;
    BoardSize boardSize;
    float boardFieldDim;
}

// Public methods.
- (NSSize)windowWillResize:(NSWindow *)sender toSize:(NSSize)
    proposedFrameSize;
- (void)gameDidEnd:(NSWindow *)sheet returnCode:(int)returnCode
    contextInfo:(void  *)contextInfo;
- (void)gameOverWithWinner:(NSString *)playerName;
- (void)highlightField:(BoardField)bf;
- (void)redrawOriginalState;
- (void)highlightPiceAt:(BoardField)bf;
- (void)drawOpaquePiceAt:(BoardField)bf forPlayer:(int)player;
- (void)setGameEngine:(GameEngine*)ge;
- (void)setHighlightState:(BOOL)highlight;
- (void)setBoardSize:(BoardSize)board;
- (void)updateToState:(GameState)bs;

@end
```

C.1.10 GameController.m

```
 //  Kolibrat
//   GameController.m
//
//   Created by Aron Lindberg.

#import "GameController.h"

@implementation GameController

// this method is called because GameControler is the main game windows delegate. It ensures that the window resizes propperly.
- (NSSize)windowWillResize:(NSWindow *)sender toSize:(NSSize)proposedFrameSize
{
    // Calculates the maximun size of the window.
    float MaxWindowSize = 128 * boardSize.width + 2 * 20 ;

    // Calculates the new size of the border arraound the game board.
    float border = proposedFrameSize.width /  MaxWindowSize * 20 ;

    // Calculates the new game board dimentions.
    float gameboardWidth = proposedFrameSize.width - 2 * border;

    //Sets the distance from the left window margain and the game board.
    [gb setDisplayOffset:border];

    //Calculates that sets the boardField dimention of the new game board.
    boardFieldDim = gameboardWidth / boardSize.width;
    [gb setSquareDim: boardFieldDim ];

    float newWindowWidth = 2 * border + boardFieldDim * boardSize.width;
    float newWindowHeight = border + boardFieldDim * boardSize.height + 59 ;

    // Returns the updated windowsize that the window will resize to.
    return NSMakeSize( newWindowWidth, newWindowHeight );
}

// This causes Kolibrat to terminate after last window is closed. This works becouse GameController is the NSApp delegate.
- (BOOL)applicationShouldTerminateAfterLastWindowClosed:(NSApplication *)app
{
    return YES;
}

// This method is called once, when the game window is loaded.
```

```
- (void)awakeFromNib
{
    canRestartGame = NO;
    [redScore setIntValue: 0 ];
    [blackScore setIntValue: 0 ];

    [[self window] useOptimizedDrawing:YES];
}

// This method is called by the GameEninge when the game is over.
- (void)gameOverWithWinner:(NSString *)playerName
{
    NSString *title = @"Game Over";
    NSString *defaultButton = @"OK";
    NSString *alternateButton = @"Quit";
    NSString *otherButton = @"Restart Game";
    NSString *message;

    if( playerName != nil )
        message = [NSString stringWithFormat:@"The game is over. %@ won.", playerName];
    else
        message = @"The game is over, and ends in a draw since no player can move.";


// Wait 0.8 secondt to dispay the GameOver dialog.
[NSThread sleepUntilDate: [NSDate dateWithTimeIntervalSinceNow: 0.8 ]];

    NSBeginAlertSheet( title,
                       defaultButton,
                       alternateButton,
                       otherButton,
                       [self window],
                       self,
                       @selector(gameDidEnd:returnCode:contextInfo:),
                       nil,
                       nil,
                       message );
}

// Sets the score in the game window.
- (void)setScore:(GameScore)score
{
    [redScore setIntValue:score.red];
    [blackScore setIntValue:score.black];
}

// All these messages are sent to the game board, that handles the aktual drawing.
- (void)highlightField:(BoardField)bf
{
    if( doHighlighting == YES )
```

```
        [gb highlightField:bf];
}

- (void)redrawOriginalState
{
    [gb redrawOriginalState];
}

- (void)updateToState:(GameState)bs
{
    [self setScore:bs.score];
    [gb redrawOriginalState];
    [gb drawPicesFromBoard:bs];
}

- (void)highlightPiceAt:(BoardField)bf
{
    if(bf.x < 0 || bf.y < 0 || bf.x > boardSize.width - 1 || bf.y >
         boardSize.height - 1 )
    {
        NSException* e = [NSException exceptionWithName:@"Move
            error." reason:[NSString stringWithFormat:@"The board
            field (%i,%i) is outside of the board.", bf.x, bf.y]
            userInfo:nil];
            @throw e;
    }
    [gb highlightPiceAt:bf];
}

- (void)drawOpaquePiceAt:(BoardField)bf forPlayer:(int)player
{
    if(bf.x < 0 || bf.y < 0 || bf.x > boardSize.width - 1 || bf.y >
         boardSize.height - 1 )
    {
        NSException* e = [NSException exceptionWithName:@"Move
            error." reason:[NSString stringWithFormat:@"The board
            field (%i,%i) is outside of the board.", bf.x, bf.y]
            userInfo:nil];
            @throw e;
    }
    [gb drawOpaquePiceAt:bf forPlayer:player];
}

- (void)gameDidEnd:(NSWindow *)sheet returnCode:(int)returnCode
    contextInfo:(void *)contextInfo // change name to
    UserResponseToGameOverDialog
{
    if( returnCode == NSAlertAlternateReturn )
        [NSApp terminate:self];
    if( returnCode == NSAlertOtherReturn )
        [ge resetGame];
    if( returnCode == NSAlertDefaultReturn )
        // All is fine (do nothing), the Game Engine is disabled.
    if( returnCode == NSAlertErrorReturn )
    {
```

```
        NSException* e = [NSException exceptionWithName:@"Unknown Error!" reason:@"Invalid return data from Game Did End sheet." userInfo:nil];
        @throw e;
    }
}

- (void)setGameEngine:(GameEngine*)engine
{
    ge = engine;
}

- (void)setHighlightState:(BOOL)highlight
{
    doHighlighting = highlight;
}

- (void)setBoardSize:(BoardSize)board
{
    boardSize.height = board.height;
    boardSize.width = board.width;
    [gb setBoardSize:board];

    float WindowMaxWidth = 2 * 20 + 128 * boardSize.width;
    float WindowMaxHeight = 20 + 128 * boardSize.height + 59 ;

    float WindowMinWidth = 2 * 10 + 64 * boardSize.width;
    float WindowMinHeight = 10 + 64 * boardSize.height + 59 ;

    [[self window] setMaxSize: NSMakeSize( WindowMaxWidth, WindowMaxHeight )];
    [[self window] setMinSize: NSMakeSize( WindowMinWidth, WindowMinHeight )];

    NSRect gameWindowFrame = [[self window] frame];

    [gb setDisplayOffset: 20 ];
    [gb setSquareDim: 128 ];
    [gb setNeedsDisplay: YES ];

    [[self window] setFrame: NSMakeRect( gameWindowFrame.origin.x,gameWindowFrame.origin.y, WindowMaxWidth, WindowMaxHeight) display:YES];
}

@end
```

C.1.11 GameBoard.h

```
//  Kolibrat
//  GameBoard.h
//
```

```
//  Created by Aron Lindberg.

#import <Cocoa/Cocoa.h>
#import "Datastructures.h"

@interface GameBoard : NSView
{

// Private instance variabels.
@private
    float squareDim;
    int **HighlightArray;
    int **PicesArray;
    BoardSize boardSize;
    float offset;
}

// Public instance methods.
- (id)initWithFrame:(NSRect)frameRect;
- (void)drawRect:(NSRect)rect;
- (void)mouseDown:(NSEvent*)event;
- (void)highlightField:(BoardField)bf;
- (void)redrawOriginalState;
- (void)drawPicesFromBoard:(GameState)gs;
- (void)highlightPiceAt:(BoardField)bf;
- (void)drawOpaquePiceAt:(BoardField)bf forPlayer:(int)player;
- (void)setBoardSize:(BoardSize)board;
- (void)setSquareDim:(float)dim;
- (void)setDisplayOffset:(float)distance;

@end
```

C.1.12 GameBoard.m

```
//  Kolibrat
//  GameBoard.m
//
//  Created by Aron Lindberg.

#import "GameBoard.h"

// Class variabels
NSRect emptyRect;
NSImage *blackPice;
NSImage *redPice;
NSImage *blackPiceHL;
NSImage *redPiceHL;

@implementation GameBoard

// Internal method used to change the size of the game board.
- (void)changeSizeOfBoardArray
```

```
{
    int i;

    HighlightArray = malloc(boardSize.width * sizeof(int *));
    PicesArray     = malloc(boardSize.width * sizeof(int *));

    for(i = 0 ; i < boardSize.width; i++)
    {
        HighlightArray[i] = malloc(boardSize.height * sizeof(int));
        PicesArray[i]     = malloc(boardSize.height * sizeof(int));
    }

    int x, y;
    for ( x = 0 ; x < boardSize.width ; x++ ) {
        for ( y = 0 ; y < boardSize.height ; y++ ) {
            HighlightArray[x][y] = EMPTY;
            PicesArray[x][y] = EMPTY;
        }
    }
}

// Called when the GUI loads.
- (void)awakeFromNib
{
    // temp boarder size until the player starts a new game.
    boardSize.width = 3 ;
    boardSize.height = 4 ;

    [self changeSizeOfBoardArray];

    squareDim = 128 ;
    offset = 20 ;
}

// interneal method to get rectangels for BoardFields.
- (NSRect)RectForField:(BoardField)bf
{
    NSRect field;
    field.origin = NSMakePoint( bf.x * squareDim + offset, bf.y * squareDim );
    field.size = NSMakeSize( squareDim, squareDim );
    return field;
}

// internal method to get color of a BoardField.
- (void)setColorForField:(BoardField)bf
{
    if ( HighlightArray[bf.x][bf.y] != HIGHLIGHTFIELD )
        if( bf.x % 2 == bf.y % 2 )
            [[NSColor lightGrayColor] set];
        else
            [[NSColor darkGrayColor] set];

    else if( HighlightArray[bf.x][bf.y] == HIGHLIGHTFIELD )
        if( bf.x % 2 == bf.y % 2 )
```

```
            [[NSColor colorWithCalibratedRed: 0.851 green: 0.569
                blue: 0.310 alpha: 1.0 ] set];
        else
            [[NSColor colorWithCalibratedRed: 0.718 green: 0.435
                blue: 0.173 alpha: 1.0 ] set];
}


- (id)initWithFrame:(NSRect)frameRect
{
    if ((self = [super initWithFrame:frameRect]) != nil)
    {
        squareDim = ([self bounds].size.width / boardSize.width) ;
        redPice = [NSImage imageNamed:@"red"];
        redPiceHL = [NSImage imageNamed:@"redHL"];
        blackPice = [NSImage imageNamed:@"black"];
        blackPiceHL = [NSImage imageNamed:@"blackHL"];
    }
    return self;
}

// This method is called everythime Cocoa wants to redraw the GUI.
- (void)drawRect:(NSRect)rect
{
    int x; int y;
    for ( x = 0 ; x < boardSize.width ; x++ ) {
        for ( y = 0 ; y < boardSize.height ; y++ ) {
            BoardField bf = makeBoardField( x, y );
            [self setColorForField: bf];
            [NSBezierPath fillRect: [self RectForField: bf]];

            if( PicesArray[x][y] == RED )
                [redPice drawInRect:[self RectForField:
                    makeBoardField( x, y )]
                           fromRect:emptyRect
                          operation:NSCompositeSourceAtop fraction:
                               1 ];
            if( PicesArray[x][y] == RED_HIGHLIGHT )
                [redPiceHL drawInRect:[self RectForField:
                    makeBoardField( x, y )]
                             fromRect:emptyRect
                            operation:NSCompositeSourceAtop
                                fraction: 1 ];
            if( PicesArray[x][y] == RED_OPAQUE )
                [redPice drawInRect:[self RectForField:
                    makeBoardField( x, y )]
                           fromRect:emptyRect
                          operation:NSCompositeSourceAtop fraction:
                               0.35 ];

            if(  PicesArray[x][y] == BLACK )
                [blackPice drawInRect:[self RectForField:
                    makeBoardField( x, y )]
                             fromRect:emptyRect
```

```
                                operation:NSCompositeSourceAtop
                                    fraction: 1 ];
            if(  PicesArray[x][y] == BLACK_HIGHLIGHT )
                [blackPiceHL drawInRect:[self RectForField:
                    makeBoardField( x, y )]
                                 fromRect:emptyRect
                                operation:NSCompositeSourceAtop
                                    fraction: 1 ];
            if(  PicesArray[x][y] == BLACK_OPAQUE )
                [blackPice drawInRect:[self RectForField:
                    makeBoardField( x, y )]
                                 fromRect:emptyRect
                                operation:NSCompositeSourceAtop
                                    fraction: 0.35 ];
        }
    }
}

// This method is called by NSApplication everytime the mouse is
    clicked.
- (void)mouseDown:(NSEvent*)event
{
    NSPoint mouse = [self convertPoint: [event locationInWindow]
        fromView: nil];
    int x = -1 ; int y = -1 ;
    while ( mouse.x - offset > 0 ) {
        mouse.x -= squareDim;
        x++;
    }
    while ( mouse.y > 0 ) {
        mouse.y -= squareDim;
        y++;
    }

    if( x < 0 || y < 0 || x > boardSize.width -1 || y >
        boardSize.height -1 )
        return; // The click was not on the board.

    // Saves the click coordinates as an object.
    BoardFieldObject *clickCordinate =
                [BoardFieldObject boardfieldObjectWithField:
                    makeBoardField(x,y)];

    // Sends a Notification informing other classes of the click.
    [[NSNotificationCenter defaultCenter] postNotificationName:@"
        MouseClick" object:clickCordinate];
}

// Method to highlight a field on the board.
- (void)highlightField:(BoardField)bf
{
    HighlightArray[bf.x][bf.y] = HIGHLIGHTFIELD;
    [self setNeedsDisplay:YES];
}

```

```
// Method to restore (unhighlight) the board.
- (void)redrawOriginalState
{
    int x; int y;
    for ( x = 0 ; x < boardSize.width ; x++ ) {
        for ( y = 0 ; y < boardSize.height ; y++ ) {
            HighlightArray[x][y] = EMPTY;
            if( PicesArray[x][y] == RED_HIGHLIGHT )
                PicesArray[x][y] = RED;
            if( PicesArray[x][y] == RED_OPAQUE )
                PicesArray[x][y] = EMPTY;
            if( PicesArray[x][y] == BLACK_HIGHLIGHT )
                PicesArray[x][y] = BLACK;
            if( PicesArray[x][y] == BLACK_OPAQUE )
                PicesArray[x][y] = EMPTY;
        }
    }
    [self setNeedsDisplay:YES];
}

- (void)drawPicesFromBoard:(GameState)gs
{
    int x; int y;
    for ( x = 0 ; x < boardSize.width; x++ ) {
        for ( y = 0 ; y < boardSize.height ; y++ )
        {
            PicesArray[x][y] =  BOARD_CONVERTER( gs.board[x][y] );
        }
    }
    [self setNeedsDisplay:YES];
}

- (void)highlightPiceAt:(BoardField)bf
{
    if( PicesArray[bf.x][bf.y] == RED )
    {
        PicesArray[bf.x][bf.y] = RED_HIGHLIGHT;
        [self setNeedsDisplay:YES];
    }
    else if( PicesArray[bf.x][bf.y] == BLACK )
    {
        PicesArray[bf.x][bf.y] = BLACK_HIGHLIGHT;
        [self setNeedsDisplay:YES];
    }

}

- (void)drawOpaquePiceAt:(BoardField)bf forPlayer:(int)player
{
    if( player == RED )
        PicesArray[bf.x][bf.y] = RED_OPAQUE;
    else if( player == BLACK )
        PicesArray[bf.x][bf.y] = BLACK_OPAQUE;
    [self setNeedsDisplay:YES];
}
```

```
- (void)setBoardSize:(BoardSize)board
{
    int i;
    for(i = 0 ; i < boardSize.width ; i++)
    {
        free(HighlightArray[i]);
        free(PicesArray[i]);
    }
    free(HighlightArray);
    free(PicesArray);

    boardSize.height = board.height;
    boardSize.width = board.width;

    [self changeSizeOfBoardArray];
    [self setNeedsDisplay:YES];
}

- (void)setSquareDim:(float)dim
{
    squareDim = dim;
}

- (void)setDisplayOffset:(float)distance
{
    offset = distance;
    [self setNeedsDisplay:YES];
}

@end
```

C.1.13 NewGameSheetController.h

```
//  Kolibrat
//  NewGameSheetController.h
//
//  Created by Aron Lindberg.

#import "GameController.h"
#import "Datastructures.h"
#import "HumanPlayer.h"
#import "GameEngine.h"
#import "BasicAI.h"
#import "AdvancedAI.h"

@interface NewGameSheetController : NSObject
{
@public
    IBOutlet NSTextField *blackName;
    IBOutlet NSPopUpButton *blackType;
    IBOutlet NSPopUpButton *boardpopUPMenu;
```

```
    IBOutlet NSTextField *goals;
    IBOutlet NSTextField *redName;
    IBOutlet NSPopUpButton *redType;
    IBOutlet GameController *gc;
    IBOutlet NSWindow *newGameWindow;
    IBOutlet NSWindow *gameWindow;
    IBOutlet NSButton *highlightInGUI;
    IBOutlet NSTextField *maxPices;
    IBOutlet NSStepper *goalsStepper;
    IBOutlet NSStepper *piecesStepper;

@private
    GameEngine *newEngine;
    NSMutableArray *playersType;
    NSMutableDictionary *playerIdentefiers;
}
- (IBAction)defaultsButton:(id)sender;
- (IBAction)startGameButton:(id)sender;
- (IBAction)newGame:(id)sender;
- (IBAction)restartGame:(id)sender;
- (IBAction)cancelButton:(id)sender;
- (void)awakeFromNib;
- (void)loadPlayers;
- (BOOL)validateMenuItem:(NSMenuItem *)item;

@end
```

C.1.14 NewGameSheetController.m

```
//  Kolibrat
//  NewGameSheetController.m
//
//  Created by Aron Lindberg.

#import "NewGameSheetController.h"

@implementation NewGameSheetController

// This method is called when the NIB file is initialised. And is
    used to load data into the NSPopUpButtons.
- (void)awakeFromNib
{
    [self loadPlayers];

    [boardpopUPMenu removeAllItems];

    [boardpopUPMenu addItemWithTitle:@"2x2"];
    [boardpopUPMenu addItemWithTitle:@"2x4"];
    [boardpopUPMenu addItemWithTitle:@"3x3"];
    [boardpopUPMenu addItemWithTitle:@"3x4"];
    [boardpopUPMenu addItemWithTitle:@"4x5"];
    [boardpopUPMenu addItemWithTitle:@"5x6"];
```

```
    [boardpopUPMenu addItemWithTitle:@"9x9"];

    [boardpopUPMenu setTitle:@"3x4"];

    [blackType removeAllItems];
    [blackType addItemsWithTitles: playersType];

    [redType removeAllItems];
    [redType addItemsWithTitles: playersType];
}

// This method is called when the button "Default Options" is
    clicked.
- (IBAction)defaultsButton:(id)sender
{
    [boardpopUPMenu setTitle:@"3x4"];
    [goals setIntValue: 5 ];
    [highlightInGUI setState: 1 ];
    [maxPices setIntValue: 4 ];
    [goalsStepper setIntValue: 5 ];
    [piecesStepper setIntValue: 4 ];
}

// This method is called when the "Start Game" button is clicked.
- (IBAction)startGameButton:(id)sender
{
    // Initialises the red and black player.
    NSDictionary *playerClassTypes = playerIdentefiers;

    // Gets the types of both players as a string.
    NSString *redTypeTitel = [[redType selectedItem]title];
    NSString *blackTypeTitel = [[blackType selectedItem]title];

    // Gets the class og both players.
    Class redClass = [playerClassTypes objectForKey:redTypeTitel];
    Class blackClass = [playerClassTypes objectForKey:
        blackTypeTitel];

    // Gets the name og both players.
    NSString *nameOfRed = [redName stringValue];
    NSString *nameOfBlack = [blackName stringValue];

    // Array with the size of the game board.
    NSArray *boardSizeArray = [[[boardpopUPMenu selectedItem] title
        ]
                                    componentsSeparatedByString:@"x"];

    // BoardSize instance that contains the size of the choosen
        board.
    BoardSize board = makeBoardSize( [[boardSizeArray objectAtIndex
        : 1 ] intValue], [[boardSizeArray objectAtIndex: 0 ]
        intValue] );

    // Initialises both players.
```

```
    id red = [[redClass alloc] initAsPlayer:PLAYER_RED withName: nameOfRed boardSize:board picesOnboard:[maxPices intValue] goalsToWin:[goals intValue]];
    id black = [[blackClass alloc] initAsPlayer:PLAYER_BLACK withName: nameOfBlack boardSize:board picesOnboard:[maxPices intValue] goalsToWin:[goals intValue]];

    if( newEngine != nil )
    {// The engine is not nil, so there is a old engine we need to release.
        [newEngine release];
    }

    // Sends the Highlight Option and board size to the GUI.
    [gc setHighlightState: [highlightInGUI state]];
    [gc setBoardSize: board ];

    // Initialises the GameEngine with the red and black player.
    newEngine = [[GameEngine alloc] initWithPlayersRed:red
                                                andBlack:black
                                              goalsToWin:[goals intValue]
                                            GameBoardDim:board
                                                MaxPices:[maxPices intValue]
                                            connectToGUI:gc];

    // Connects the players to the GameEngine.
    [red setGameEngine:newEngine];
    [black setGameEngine:newEngine];

    // Connects the GUI to the GameEngine.
    [gc setGameEngine:newEngine];

    // Removes the newGame sheet so the game can begin.
    [newGameWindow orderOut:self];
    [NSApp endSheet:newGameWindow];
}

// Loads all players wirtten al plugins, and the human player.
- (void)loadPlayers
{
    playersType = [[NSMutableArray alloc] init];
    playerIdentefiers = [NSMutableDictionary dictionaryWithCapacity: 20 ];

    [playerIdentefiers retain];

    [playersType retain];

    if( [[HumanPlayer class] conformsToProtocol:@protocol( Player_Protocol )])
    {
        [playersType addObject: [ HumanPlayer playerType ]];
```

```
        [playerIdentefiers setObject: [ HumanPlayer class ] forKey
            :[ HumanPlayer playerType ]];
    }

    if( [[BasicAI class] conformsToProtocol:@protocol(
        Player_Protocol )])
    {
        [playersType addObject: [ BasicAI playerType ]];
        [playerIdentefiers setObject: [ BasicAI class ] forKey:[
            BasicAI playerType ]];
    }

    if( [[AdvancedAI class] conformsToProtocol:@protocol(
        Player_Protocol )])
    {
        [playersType addObject: [ AdvancedAI playerType ]];
        [playerIdentefiers setObject: [ AdvancedAI class ] forKey:[
             AdvancedAI playerType ]];
    }

    // Temp varibales to store data for loading the players.
    NSString *currPath;
    NSBundle *currBundle;
    Class currPrincipalClass;
    NSMutableArray *bundlePaths = [ NSMutableArray array ];
    NSEnumerator *searchPathEnum;
    NSMutableArray *bundleSearchPaths = [ NSMutableArray array ];
    NSMutableArray *allBundles = [ NSMutableArray array ];
    NSArray *librarySearchPaths;

    librarySearchPaths = NSSearchPathForDirectoriesInDomains(
        NSLibraryDirectory, NSAllDomainsMask - NSSystemDomainMask,
        YES );

    searchPathEnum = [librarySearchPaths objectEnumerator];

    while(currPath = [searchPathEnum nextObject])
    {
        [bundleSearchPaths addObject: [currPath
            stringByAppendingPathComponent:@"Application Support/
            Kolibrat/PlugIns"]];
    }

    [bundleSearchPaths addObject:[[NSBundle mainBundle]
        builtInPlugInsPath]];

    searchPathEnum = [bundleSearchPaths objectEnumerator];
    while(currPath = [searchPathEnum nextObject])
    {
        NSDirectoryEnumerator *bundleEnum;
        NSString *currBundlePath;
        bundleEnum = [[NSFileManager defaultManager]
            enumeratorAtPath:currPath];
        if(bundleEnum)
        {
```

```
            while(currBundlePath = [bundleEnum nextObject])
            {
                if([[currBundlePath pathExtension] isEqualToString:
                    @"bundle"])
                {
                    [allBundles addObject:[currPath
                        stringByAppendingPathComponent:
                        currBundlePath]];
                }
            }
        }
    }

    [bundlePaths addObjectsFromArray: allBundles];
    NSEnumerator *pathEnum = [bundlePaths objectEnumerator];

    while(currPath = [pathEnum nextObject])
    {
        currBundle = [NSBundle bundleWithPath:currPath];
        if( currBundle )
        {
            currPrincipalClass = [currBundle principalClass];
            if(currPrincipalClass)
            {

                if( [currPrincipalClass conformsToProtocol:
                    @protocol( Player_Protocol )])
                {
                    [playersType addObject: [currPrincipalClass
                        playerType]];
                    [playerIdentefiers setObject:
                        currPrincipalClass forKey:[
                        currPrincipalClass playerType ]];
                }
            }
        }
    }
}

// This method controlls wheter or not the menu items are avaible
    or not. Used to disable "Restart Game" until a game has been
    started.
- (BOOL)validateMenuItem:(NSMenuItem *)item
{
    NSString *name = @"Restart Game";

    if( [name isEqualToString: [item title]] && newEngine != nil )
        return TRUE;
    else if( [name isEqualToString: [item title]] )
        return FALSE;

    return TRUE;
}

```

```
// This method is called when the user chooses "New Game" from the menu.
- (IBAction)newGame:(id)sender;
{
    [gameWindow setIsVisible:YES]; // Shows the game window.

    [newEngine delayNextPlayer:TRUE];

    [NSApp beginSheet:newGameWindow // shows the new game options sheet.
       modalForWindow:gameWindow
        modalDelegate:nil
       didEndSelector:nil
          contextInfo:nil];
}

// This method is called when the user chooses "Restart Game" from the menu.
- (IBAction)restartGame:(id)sender
{
    [newEngine resetGame];
}

- (IBAction)cancelButton:(id)sender
{
    [newGameWindow orderOut:self];
    [NSApp endSheet:newGameWindow];
    [newEngine delayNextPlayer:FALSE];
}

@end
```

C.1.15 GUIProtocol.h

```
//  Kolibrat
//  GUI_Protocol.h
//
//  Created by Aron Lindberg.

@protocol GUI_Protocol
- (void)gameOverWithWinner:(NSString *)playerName;
- (void)highlightField:(BoardField)boardField;
- (void)redrawOriginalState;
- (void)updateToState:(GameState)gameState;
- (void)highlightPiceAt:(BoardField)boardField;
- (void)drawOpaquePiceAt:(BoardField)boardField forPlayer:(int)player;
@end
```

C.1.16 PlayerProtocol.h

```
//   Kolibrat
//   Player_Protocol.h
//
//   Created by Aron Lindberg.

@protocol Player_Protocol
- (id)initAsPlayer:(int)player withName:(NSString *)playerName
     boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(
     int)maxGoals;
- (void)startNewTurn;
- (void)setGameEngine:(id)ge;
- (void)reset;
- (NSString *)playerName;
+ (NSString *)playerType;
@end
```

C.1.17 main.m

```
//   Kolibrat
//   main.m
//
//   Created by Aron Lindberg.

#import <Cocoa/Cocoa.h>

int main(int argc, char *argv[])
{
     return NSApplicationMain(argc,  (const char **) argv);
}
```

C.2 Kolibrat Test Source Code

C.2.1 Kolibrat Test.m

```
//   Kolibrat Test
//   Kolibrat Test.m
//
//   Created by Aron Lindberg.

#import <Foundation/Foundation.h>
#import "Datastructures.h"
#import "GameLogic.h"

int main (int argc, const char * argv[])
{
     NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

```

```
    int maxPices = 4 ;
    int goalsToWin = 5 ;
    BoardSize size = makeBoardSize( 4 , 3 );

    BOOL testPassed = TRUE;

    GameLogic *logic = [[GameLogic alloc] initWithMaxPices:maxPices goalsToWin:goalsToWin boardSize:size];

    NSLog(@"GameLogic instance created.");
    NSLog(@"Max pixes on baord is: %i.", maxPices);
    NSLog(@"Number of goals to win is: %i.", goalsToWin);
    NSLog(@"Board dimentions (%i,%i).",size.height, size.width );

    NSLog(@"Starting test 1: Insert pice for red in (1,0) on an empty board.");

    GameState state1 = [logic CreateNewGameState];

    if( [logic makeMove:makeMoveFromInt( 1 , 0 , 1 , 0 ) withState:&state1] != TRUE )
    {
        NSLog(@"Error the GameLogic rejected the move.");
        testPassed = FALSE;
    }
    if( state1.redPicesOnBoard != 1 || state1.blackPicesOnBoard != 0 )
    {
        NSLog(@"Error in number of pices on the board.");
        testPassed = FALSE;
    }
    if( state1.playerMoving != PLAYER_BLACK )
    {
        NSLog(@"Error black player don't is not the new moving player.");
        testPassed = FALSE;
    }
    if( state1.board[ 1 ][ 0 ].occupiedByRed != TRUE || state1.board[ 1 ][ 0 ].occupiedByBlack != FALSE )
    {
        NSLog(@"Error red don't have a piece in (1,0).");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 1 passed.");

    NSLog(@"Starting test 2: Insert pice for black in (1,3) on an empty board.");

    GameState state2 = [logic CreateNewGameState];
    state2.playerMoving = PLAYER_BLACK;
    testPassed = TRUE;

```

```
    if( [logic makeMove:makeMoveFromInt( 1 , 3 , 1 , 3 ) withState
        :&state2] != TRUE )
    {
        NSLog(@"Error the GameLogic rejected the move.");
        testPassed = FALSE;
    }

    if( state2.redPicesOnBoard != 0 || state2.blackPicesOnBoard !=
        1 )
    {
        NSLog(@"Error in number of pices on the board.");
        testPassed = FALSE;
    }
    if( state2.playerMoving != PLAYER_RED )
    {
        NSLog(@"Error red player don't is not the new moving
            player.");
        testPassed = FALSE;
    }
    if( state2.board[ 1 ][ 3 ].occupiedByRed != FALSE ||
        state2.board[ 1 ][ 3 ].occupiedByBlack != TRUE )
    {
        NSLog(@"Error black don't have a piece in (1,3).");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 2 passed.");

    NSLog(@"Starting test 3: try to insert pice for red in (1,3) on
         an board with 4 red pices.");

    GameState state3 = [logic CreateNewGameState];
    state3.redPicesOnBoard = 4 ;
    testPassed = TRUE;

    if( [logic makeMove:makeMoveFromInt( 1 , 0 , 1 , 0 ) withState
        :&state3] != FALSE)
    {
        NSLog(@"Error the illegal move was not rejected by
            GameLogic.");
        testPassed = FALSE;
    }
    if( state3.redPicesOnBoard != 4 || state3.blackPicesOnBoard !=
        0 )
    {
        NSLog(@"Error in number of pices on the board.");
        testPassed = FALSE;
    }
    if( state3.playerMoving != PLAYER_RED )
    {
        NSLog(@"Error black player is the new moving player, it
            shuld be red.");
        testPassed = FALSE;
    }
```

```
    if( state3.board[ 1 ][ 0 ].occupiedByRed != FALSE ||
        state3.board[ 1 ][ 0 ].occupiedByBlack != FALSE )
    {
        NSLog(@"Error red have a piece in (1,0).");
        testPassed = FALSE;
    }
    if( testPassed == TRUE )
        NSLog(@"Test 3 passed, another piece was not added to the
            board.");

    NSLog(@"Starting test 4: try to insert pice for red in (2,2),
        an illegal move. The board is empty.");

    GameState state4 = [logic CreateNewGameState];

    testPassed = TRUE;

    if( [logic makeMove:makeMoveFromInt( 2 , 2 , 2 , 2 ) withState
        :&state4] )
    {
        NSLog(@"Error the illegal move was not rejected by
            GameLogic.");
        testPassed = FALSE;
    }
    if( state4.redPicesOnBoard != 0 || state4.blackPicesOnBoard !=
        0 )
    {
        NSLog(@"Error in number of pices on the board.");
        testPassed = FALSE;
    }
    if( state4.playerMoving != PLAYER_RED )
    {
        NSLog(@"Error black player is the new moving player, it
            shuld be red.");
        testPassed = FALSE;
    }
    if( state4.board[ 2 ][ 2 ].occupiedByRed != FALSE ||
        state4.board[ 2 ][ 2 ].occupiedByBlack != FALSE )
    {
        NSLog(@"Error red have a piece in (2,2).");
        testPassed = FALSE;
    }
    if( testPassed == TRUE )
        NSLog(@"Test 4 passed, the piece was not added to the
            board.");

    NSLog(@"Starting test 5: Score point for red.");

    GameState state5 = [logic CreateNewGameState];

    testPassed = TRUE;

    state5.redPicesOnBoard = 1;
    state5.board[ 1 ][ 3 ].occupiedByRed = TRUE;

```

```

    if( [logic makeMove:makeMoveFromInt( 1 , 3 , 1 , 3 ) withState:&state5] != TRUE )
    {
        NSLog(@"Error the move was rejected by GameLogic.");
        testPassed = FALSE;
    }
    if( state5.redPicesOnBoard != 0 || state5.blackPicesOnBoard != 0 )
    {
        NSLog(@"Error in number of pices on the board.");
        testPassed = FALSE;
    }
    if( state5.playerMoving != PLAYER_BLACK )
    {
        NSLog(@"Error black player not is the new moving player.");
        testPassed = FALSE;
    }
    if( state5.board[ 1 ][ 3 ].occupiedByRed != FALSE || state5.board[ 1 ][ 3 ].occupiedByBlack != FALSE )
    {
        NSLog(@"Error red have a piece in (1,3), it shuld be gone.");
        testPassed = FALSE;
    }

    if( state5.score.red != 1 || state5.score.black != 0 )
    {
        NSLog(@"Error in game score.");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 5 passed.");

    NSLog(@"Starting test 6: Score point for black.");

    GameState state6 = [logic CreateNewGameState];

    testPassed = TRUE;

    state6.blackPicesOnBoard = 1 ;
    state6.board[ 1 ][ 0 ].occupiedByBlack = TRUE;
    state6.playerMoving = PLAYER_BLACK;

    if( [logic makeMove:makeMoveFromInt( 1 , 0 , 1 , 0 ) withState:&state6] != TRUE )
    {
        NSLog(@"Error the move was rejected by GameLogic.");
        testPassed = FALSE;
    }
    if( state6.redPicesOnBoard != 0 || state6.blackPicesOnBoard != 0 )
    {
        NSLog(@"Error in number of pices on the board.");
```

```
            testPassed = FALSE;
        }
        if( state6.playerMoving != PLAYER_RED )
        {
            NSLog(@"Error red player not is the new moving player.");
            testPassed = FALSE;
        }
        if( state6.board[ 1 ][ 0 ].occupiedByRed != FALSE ||
            state6.board[ 1 ][ 0 ].occupiedByBlack != FALSE )
        {
            NSLog(@"Error black have a piece in (1,0), it shuld be
                gone.");
            testPassed = FALSE;
        }

        if( state6.score.red != 0 || state6.score.black != 1 )
        {
            NSLog(@"Error in game score.");
            testPassed = FALSE;
        }

        if( testPassed == TRUE )
            NSLog(@"Test 6 passed.");

        NSLog(@"Starting test 7: Try to insert piece in ocupied field."
            );

        GameState state7 = [logic CreateNewGameState];

        testPassed = TRUE;

        state7.blackPicesOnBoard = 1 ;
        state7.board[ 1 ][ 0 ].occupiedByBlack = TRUE;

        if( [logic makeMove:makeMoveFromInt( 1 , 0 , 1 , 0 ) withState
            :&state7] != FALSE )
        {
            NSLog(@"Error the illegal move was not rejected by
                GameLogic.");
            testPassed = FALSE;
        }
        if( state7.redPicesOnBoard != 0 || state7.blackPicesOnBoard !=
            1 )
        {
            NSLog(@"Error in number of pices on the board.");
            testPassed = FALSE;
        }
        if( state7.playerMoving != PLAYER_RED )
        {
            NSLog(@"Error in the next player to move.");
            testPassed = FALSE;
        }
        if( state7.board[ 1 ][ 0 ].occupiedByRed != FALSE ||
            state7.board[ 1 ][ 0 ].occupiedByBlack != TRUE )
        {
```

```
        NSLog(@"Error don't black have a piece in (1,0).");
        testPassed = FALSE;
    }

    if( state7.score.red != 0 || state7.score.black != 0 )
    {
        NSLog(@"Error in game score.");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 7 passed.");

    NSLog(@"Starting test 8: Gamestatus is changed when red wins.");

    GameState state8 = [logic CreateNewGameState];

    testPassed = TRUE;

    state8.redPicesOnBoard = 1 ;
    state8.board[ 1 ][ 3 ].occupiedByRed = TRUE;
    state8.score.red = 4 ;

    if( [logic makeMove:makeMoveFromInt( 1 , 3 , 1 , 3 ) withState:&state8] != TRUE )
    {
        NSLog(@"Error the move was rejected by GameLogic.");
        testPassed = FALSE;
    }
    if( state8.redPicesOnBoard != 0 || state8.blackPicesOnBoard != 0 )
    {
        NSLog(@"Error in number of pices on the board.");
        testPassed = FALSE;
    }

    if( state8.board[ 1 ][ 3 ].occupiedByRed != FALSE || state8.board[ 1 ][ 3 ].occupiedByBlack != FALSE )
    {
        NSLog(@"Error the field (1,3) is not empty.");
        testPassed = FALSE;
    }

    if( state8.score.red != 5 || state8.score.black != 0 )
    {
        NSLog(@"Error in game score.");
        testPassed = FALSE;
    }

    if( state8.gameStatus.gameOver != TRUE || state8.gameStatus.winner != PLAYER_RED )
    {
        NSLog(@"Error in game status, red is not the winner.");
        testPassed = FALSE;
```

```
    }
    if( testPassed == TRUE )
        NSLog(@"Test 8 passed.");

    NSLog(@"Starting test 9: Ensure that no players can move when
        the game is over.");

    testPassed = TRUE;

    state8.redPicesOnBoard = 1 ;
    state8.board[ 0 ][ 0 ].occupiedByRed = TRUE;

    state8.blackPicesOnBoard = 1 ;
    state8.board[ 2 ][ 3 ].occupiedByBlack = TRUE;

    state8.playerMoving = PLAYER_RED;

    if( [logic makeMove:makeMoveFromInt( 0 , 0 , 1 , 1 ) withState
        :&state8] != FALSE )
    {
        NSLog(@"Error the illegal move by red was not rejected by
            GameLogic.");
        testPassed = FALSE;
    }

    state8.playerMoving = PLAYER_BLACK;

    if( [logic makeMove:makeMoveFromInt( 2 , 3 , 1 , 2 ) withState
        :&state8] != FALSE )
    {
        NSLog(@"Error the illegal move by black was not rejected by
             GameLogic.");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 9 passed.");

    NSLog(@"Starting test 10: Ensure that no players can move
        outside of the board.");

    testPassed = TRUE;
    GameState state10 = [logic CreateNewGameState];

    state10.redPicesOnBoard = 1 ;
    state10.board[ 0 ][ 0 ].occupiedByRed = TRUE;

    state10.blackPicesOnBoard = 1 ;
    state10.board[ 2 ][ 3 ].occupiedByBlack = TRUE;

    @try {


        if( [logic makeMove:makeMoveFromInt( 0 , 0 , -1 , 1 )
            withState:&state10] != FALSE )
```

```
        {
            NSLog(@"Error the illegal move by red was not rejected
                by GameLogic.");
            testPassed = FALSE;
        }
    }
    @catch ( NSException *e ) {

        NSLog(@"GameLogic threw an exception, this is ok.");
    }

    state10.playerMoving = PLAYER_BLACK;

    @try {
        if( [logic makeMove:makeMoveFromInt( 2 , 3 , 1 , 4 )
            withState:&state10] != FALSE )
        {
            NSLog(@"Error the illegal move by black was not
                rejected by GameLogic.");
            testPassed = FALSE;
        }
    }
    @catch ( NSException *e ) {

        NSLog(@"GameLogic threw an exception, this is ok.");
    }

    if( testPassed == TRUE )
        NSLog(@"Test 10 passed.");

    NSLog(@"Starting test 11: Test of moves on a non empty board 1.
        ");

    testPassed = TRUE;
    GameState state11 = [logic CreateNewGameState];

    state11.redPicesOnBoard = 1 ;
    state11.blackPicesOnBoard = 3 ;

    state11.board[ 1 ][ 0 ].occupiedByRed = TRUE;

    state11.board[ 1 ][ 1 ].occupiedByBlack = TRUE;

    state11.board[ 1 ][ 2 ].occupiedByBlack = TRUE;
    state11.board[ 1 ][ 3 ].occupiedByBlack = TRUE;

    NSSet *allmoves = [logic allLegalMoves:&state11];

    if( [allmoves count] != 5 )
    {
        NSLog(@"Error in the number of allowed moves.");
        testPassed = FALSE;
    }

```

```
    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt( 0 , 0 , 0 , 0 )]] != TRUE )
    {
        NSLog(@"Error the move to insert a piece in (0,0) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt( 2 , 0 , 2 , 0 )]] != TRUE )
    {
        NSLog(@"Error the move to insert a piece in (2,0) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt( 1 , 0 , 0 , 1 )]] != TRUE )
    {
        NSLog(@"Error the move from (1,0) to (0,1) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt( 1 , 0 , 2 , 1 )]] != TRUE )
    {
        NSLog(@"Error the move from (1,0) to (2,1) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt( 1 , 0 , 1 , 1 )]] != TRUE )
    {
        NSLog(@"Error the move from (1,0) to (1,1) was not allowed.");
        testPassed = FALSE;
    }
    if( testPassed == TRUE )
        NSLog(@"Test 11 passed.");

    NSLog(@"Starting test 12: Test of moves on a non empty board 2.");

    testPassed = TRUE;
    GameState state12 = [logic CreateNewGameState];

    state12.redPicesOnBoard = 1 ;
    state12.blackPicesOnBoard = 2 ;

    state12.board[ 2 ][ 0 ].occupiedByRed = TRUE;

    state12.board[ 2 ][ 1 ].occupiedByBlack = TRUE;

```

```
    state12.board[ 2 ][ 2 ].occupiedByBlack = TRUE;

    allmoves = [logic allLegalMoves:&state12];

    if( [allmoves count] != 5 )
    {
        NSLog(@"Error in the number of allowed moves.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt( 0 , 0 , 0 , 0 )]] != TRUE )
    {
        NSLog(@"Error the move to insert a piece in (0,0) was not
            allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt( 1 , 0 , 1 , 0 )]] != TRUE )
    {
        NSLog(@"Error the move to insert a piece in (1,0) was not
            allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt( 2 , 0 , 1 , 1 )]] != TRUE )
    {
        NSLog(@"Error the move from (2,0) to (1,1) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt( 2 , 0 , 2 , 1 )]] != TRUE )
    {
        NSLog(@"Error the move from (2,0) to (2,1) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt( 2 , 0 , 2 , 3 )]] != TRUE )
    {
        NSLog(@"Error the move from (2,0) to (2,3) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 12 passed.");

    NSLog(@"Starting test 13: Test of moves on a non empty board 3.
        ");
```

```

    testPassed = TRUE;
    GameState state13 = [logic CreateNewGameState];

    state13.redPicesOnBoard = 3 ;
    state13.blackPicesOnBoard = 1 ;

    state13.board[ 0 ][ 0 ].occupiedByRed = TRUE;

    state13.board[ 1 ][ 0 ].occupiedByRed = TRUE;

    state13.board[ 2 ][ 0 ].occupiedByRed = TRUE;

    state13.board[ 1 ][ 1 ].occupiedByBlack = TRUE;

    allmoves = [logic allLegalMoves:&state13];

    if( [allmoves count] != 4 )
    {
        NSLog(@"Error in the number of allowed moves.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt( 1 , 0 , 2 , 1 )]] != TRUE )
    {
        NSLog(@"Error the move from (1,0) to (2,1) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt( 1 , 0 , 0 , 1)]] != TRUE )
    {
        NSLog(@"Error the move from (1,0) to (0,1) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt( 1 , 0 , 1 , 1 )]] != TRUE )
    {
        NSLog(@"Error the move from (1,0) to (1,1) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt( 1 , 0 , 1 , 2 )]] != TRUE )
    {
        NSLog(@"Error the move from (1,0) to (1,2) was not allowed.");
        testPassed = FALSE;
    }

```

```
    if( testPassed == TRUE )
        NSLog(@"Test 13 passed.");

    NSLog(@"Starting test 14: Test of moves on a non empty board 4.
        ");

    testPassed = TRUE;
    GameState state14 = [logic CreateNewGameState];

    state14.redPicesOnBoard = 4;
    state14.blackPicesOnBoard = 1;

    state14.board[0][1].occupiedByRed = TRUE;

    state14.board[1][1].occupiedByRed = TRUE;

    state14.board[2][1].occupiedByRed = TRUE;

    state14.board[1][2].occupiedByRed = TRUE;

    state14.board[1][3].occupiedByBlack = TRUE;

    allmoves = [logic allLegalMoves:&state14];

    if( [allmoves count] != 5 )
    {
        NSLog(@"Error in the number of allowed moves.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,1,2,2)]] != TRUE )
    {
        NSLog(@"Error the move from (1,1) to (2,2) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,1,0,2)]] != TRUE )
    {
        NSLog(@"Error the move from (1,1) to (0,2) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,2,0,3)]] != TRUE )
    {
        NSLog(@"Error the move from (1,2) to (0,3) was not allowed.
            ");
        testPassed = FALSE;
    }

```

```
    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,2,2,3)]] != TRUE )
    {
        NSLog(@"Error the move from (1,2) to (2,3) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,2,1,3)]] != TRUE )
    {
        NSLog(@"Error the move from (1,2) to (1,3) was not allowed.
            ");
        testPassed = FALSE;
    }
    if( testPassed == TRUE )
        NSLog(@"Test 14 passed.");

    NSLog(@"Starting test 15: Test of moves on a non empty board 5.
        ");

    testPassed = TRUE;
    GameState state15 = [logic CreateNewGameState];

    state15.redPicesOnBoard = 3;
    state15.blackPicesOnBoard = 2;

    state15.board[0][0].occupiedByRed = TRUE;

    state15.board[1][0].occupiedByRed = TRUE;

    state15.board[0][1].occupiedByRed = TRUE;

    state15.board[0][2].occupiedByBlack = TRUE;
    state15.board[0][3].occupiedByBlack = TRUE;

    allmoves = [logic allLegalMoves:&state15];

    if( [allmoves count] != 5 )
    {
        NSLog(@"Error in the number of allowed moves.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(2,0,2,0)]] != TRUE )
    {
        NSLog(@"Error the move to insert a piece in (2,0) was not
            allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,0,2,1)]] != TRUE )
    {
```

```
        NSLog(@"Error the move from (1,0) to (2,1) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(0,0,1,1)]] != TRUE )
    {
        NSLog(@"Error the move from (0,0) to (1,1) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(0,1,1,2)]] != TRUE )
    {
        NSLog(@"Error the move from (0,1) to (1,2) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(0,1,0,2)]] != TRUE )
    {
        NSLog(@"Error the move from (0,1) to (0,2) was not allowed.");
        testPassed = FALSE;
    }
    if( testPassed == TRUE )
        NSLog(@"Test 15 passed.");

    NSLog(@"Starting test 16: Test of moves on a non empty board 6.");

    testPassed = TRUE;
    GameState state16 = [logic CreateNewGameState];

    state16.redPicesOnBoard = 3;
    state16.blackPicesOnBoard = 1;

    state16.board[1][0].occupiedByRed = TRUE;

    state16.board[1][1].occupiedByRed = TRUE;

    state16.board[1][2].occupiedByRed = TRUE;

    state16.board[1][3].occupiedByBlack = TRUE;

    state16.playerMoving = PLAYER_BLACK;

    allmoves = [logic allLegalMoves:&state16];

    if( [allmoves count] != 5 )
    {
        NSLog(@"Error in the number of allowed moves.");
```

```
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(0,3,0,3)]] != TRUE )
    {
        NSLog(@"Error the move to insert a piece in (0,3) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(2,3,2,3)]] != TRUE )
    {
        NSLog(@"Error the move to insert a piece in (2,3) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(1,3,0,2)]] != TRUE )
    {
        NSLog(@"Error the move from (1,3) to (0,2) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(1,3,2,2)]] != TRUE )
    {
        NSLog(@"Error the move from (1,3) to (2,2) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(1,3,1,2)]] != TRUE )
    {
        NSLog(@"Error the move from (1,3) to (1,2) was not allowed.");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 16 passed.");

    NSLog(@"Starting test 17: Test of moves on a non empty board 7.");

    testPassed = TRUE;
    GameState state17 = [logic CreateNewGameState];

    state17.redPicesOnBoard = 2;
    state17.blackPicesOnBoard = 1;
```

```
    state17.board[0][1].occupiedByRed = TRUE;

    state17.board[0][2].occupiedByRed = TRUE;


    state17.board[0][3].occupiedByBlack = TRUE;

    state17.playerMoving = PLAYER_BLACK;

    allmoves = [logic allLegalMoves:&state17];

    if( [allmoves count] != 5 )
    {
        NSLog(@"Error in the number of allowed moves.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,3,1,3)]] != TRUE )
    {
        NSLog(@"Error the move to insert a piece in (1,3) was not
            allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(2,3,2,3)]] != TRUE )
    {
        NSLog(@"Error the move to insert a piece in (2,3) was not
            allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(0,3,0,2)]] != TRUE )
    {
        NSLog(@"Error the move from (0,3) to (0,2) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(0,3,1,2)]] != TRUE )
    {
        NSLog(@"Error the move from (0,3) to (1,2) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(0,3,0,0)]] != TRUE )
    {
        NSLog(@"Error the move from (0,3) to (0,0) was not allowed.
            ");
        testPassed = FALSE;
```

```
    }

    if( testPassed == TRUE )
        NSLog(@"Test 17 passed.");

    NSLog(@"Starting test 18: Test of moves on a non empty board 8.
        ");

    testPassed = TRUE;
    GameState state18 = [logic CreateNewGameState];

    state18.redPicesOnBoard = 1;
    state18.blackPicesOnBoard = 3;

    state18.board[1][2].occupiedByRed = TRUE;

    state18.board[0][3].occupiedByBlack = TRUE;
    state18.board[1][3].occupiedByBlack = TRUE;
    state18.board[2][3].occupiedByBlack = TRUE;

    state18.playerMoving = PLAYER_BLACK;

    allmoves = [logic allLegalMoves:&state18];

    if( [allmoves count] != 4 )
    {
        NSLog(@"Error in the number of allowed moves.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,3,0,2)]] != TRUE )
    {
        NSLog(@"Error the move from (1,3) to (0,2) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,3,2,2)]] != TRUE )
    {
        NSLog(@"Error the move from (1,3) to (2,2) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,3,1,2)]] != TRUE )
    {
        NSLog(@"Error the move from (1,3) to (1,2) was not allowed.
            ");
        testPassed = FALSE;
    }

```

```
    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(1,3,1,1)]] != TRUE )
    {
        NSLog(@"Error the move from (1,3) to (1,1) was not allowed.");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 18 passed.");

    NSLog(@"Starting test 19: Test of moves on a non empty board 9.");

    testPassed = TRUE;
    GameState state19 = [logic CreateNewGameState];

    state19.redPicesOnBoard = 1;
    state19.blackPicesOnBoard = 4;

    state19.board[1][0].occupiedByRed = TRUE;

    state19.board[0][2].occupiedByBlack = TRUE;
    state19.board[1][2].occupiedByBlack = TRUE;
    state19.board[2][2].occupiedByBlack = TRUE;

    state19.board[1][1].occupiedByBlack = TRUE;

    state19.playerMoving = PLAYER_BLACK;

    allmoves = [logic allLegalMoves:&state19];

    if( [allmoves count] != 5 )
    {
        NSLog(@"Error in the number of allowed moves.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(1,1,0,0)]] != TRUE )
    {
        NSLog(@"Error the move from (1,1) to (0,0) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(1,1,1,0)]] != TRUE )
    {
        NSLog(@"Error the move from (1,1) to (1,0) was not allowed.");
        testPassed = FALSE;
    }

```

```
    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,1,2,0)]] != TRUE )
    {
        NSLog(@"Error the move from (1,1) to (2,0) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,2,0,1)]] != TRUE )
    {
        NSLog(@"Error the move from (1,2) to (0,1) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(1,2,2,1)]] != TRUE )
    {
        NSLog(@"Error the move from (1,2) to (2,1) was not allowed.
            ");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 19 passed.");

    NSLog(@"Starting test 20: Test of moves on a non empty board
        10.");

    testPassed = TRUE;
    GameState state20 = [logic CreateNewGameState];

    state20.redPicesOnBoard = 2;
    state20.blackPicesOnBoard = 3;

    state20.board[0][0].occupiedByRed = TRUE;
    state20.board[0][1].occupiedByRed = TRUE;

    state20.board[0][2].occupiedByBlack = TRUE;
    state20.board[0][3].occupiedByBlack = TRUE;
    state20.board[1][3].occupiedByBlack = TRUE;

    state20.playerMoving = PLAYER_BLACK;

    allmoves = [logic allLegalMoves:&state20];

    if( [allmoves count] != 5 )
    {
        NSLog(@"Error in the number of allowed moves.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:
        makeMoveFromInt(2,3,2,3)]] != TRUE )
```

```
    {
        NSLog(@"Error the move to insert a piece in (2,3) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(1,3,2,2)]] != TRUE )
    {
        NSLog(@"Error the move from (1,3) to (2,2) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(0,3,1,2)]] != TRUE )
    {
        NSLog(@"Error the move from (0,3) to (1,2) was not allowed.");
        testPassed = FALSE;
    }

    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(0,2,1,1)]] != TRUE )
    {
        NSLog(@"Error the move from (0,2) to (1,1) was not allowed.");
        testPassed = FALSE;
    }


    if( [allmoves containsObject:[MoveObject moveObjectWithMove:makeMoveFromInt(0,2,0,1)]] != TRUE )
    {
        NSLog(@"Error the move from (0,2) to (0,1) was not allowed.");
        testPassed = FALSE;
    }

    if( testPassed == TRUE )
        NSLog(@"Test 20 passed.");

    [pool release];
    return 0;
}
```

C.3 MiniMax Source Code

C.3.1 AIDefinitions.h

```
//  Kolibrat AI
```

```
//  AIDefinitions.h
//
//  Created by Aron Lindberg.

#import "Datastructures.h"
#import "GameLogic.h"

#define HASHSIZE 20000

static struct hashList *hashtable[HASHSIZE];

typedef struct hashList {
    GameState state;
    BOOL foundNow;
    struct hashList *next;
    short int level;
    int score;
} HASHList;



typedef struct treeState {
    BoardMove lastMove;
    unsigned short int numberOfChildren;
    unsigned short int level;
    int boardScore;
    GameState gs;
    int a;
    int b;
} TreeState;

typedef struct treeListElement {
    struct treeState *state;
    void *nextEllement;
} TreeListElement;


typedef struct treeStateList {

    struct treeListElement *firstElement;
    struct treeListElement *lastElement;
    double numberElements;

} TreeStateList;

// Some plain C methods to make instances of the custom
    typedefinitions.
void addTreeStateTo( TreeState* state, TreeStateList* list );
void removeFirstElement( TreeStateList *list );
TreeStateList makeTreeStateList( TreeState* state);
TreeState makeTreeState( GameState* gs, TreeState* anceter);
TreeState makeTreeStateChild(TreeState* anceter);
void freeTreeState( TreeState* ts);

void addTreeStateTo( TreeState* state, TreeStateList* list );
```

```
void removeFirstElement( TreeStateList *list );
TreeStateList makeTreeStateList( TreeState* state);

unsigned hashValue( GameState *gs );
BOOL equalGameStates( GameState *gs1, GameState *gs2 );
HASHList* findInHashTable( GameState *gs );
BOOL insertIntoHashTable( GameState *gs, int level , int badscore);
void freeHashTable();
void NewCalcHashTable();
```

C.3.2 AIDefinitions.m

```
//  Kolibrat AI
//  AIDefinitions.m
//
//  Created by Aron Lindberg.

#import "AIDefinitions.h"

TreeStateList makeTreeStateList( TreeState* state)
{
    struct treeStateList temp;
    temp.numberElements = 1;
    return temp;
}

void addTreeStateTo( TreeState* state, TreeStateList* list )
{
    (*(*list).lastElement).nextEllement = state;
    (*list).numberElements ++;
}

void removeFirstElement( TreeStateList *list )
{
    TreeListElement *seconodElement = (*(*list).firstElement)
        .nextEllement;
    (*list).firstElement = seconodElement;
    (*list).numberElements --;
}


TreeState makeTreeState( GameState *gs, TreeState* anceter)
{
    struct treeState temp;
    temp.level = 0;
    temp.boardScore = 0;
    temp.a = INT_MIN;
    temp.b = INT_MAX;
    temp.gs = copyGameState( gs );

    return temp;
}
```

```c

TreeState makeTreeStateChild(TreeState* anceter)
{
    struct treeState temp;

    temp.level = anceter->level;
    temp.boardScore = anceter->boardScore;
    temp.a = anceter->a;
    temp.b = anceter->b;

    temp.gs = copyGameState( &anceter->gs );

    return temp;
}

void freeTreeState( TreeState* ts)
{
    freeGameState( &ts->gs);
}

unsigned hashValue( GameState *gs )
{
    unsigned value;
    value = gs->playerMoving * 1325879543;
    value += gs->blackPicesOnBoard *59448612;
    value += gs->redPicesOnBoard *65939875;
    value += gs->redPicesOnBoard *65934321;
    value += gs->score.red * 765423104;
    value += gs->score.black * 126423265;
    value +=  gs->gameStatus.gameOver * 99842321;

    int x, y;
    for ( x = 0 ; x < gs->boardSize.width ; x++ ) {
        for ( y = 0 ; y < gs->boardSize.height ; y++ ) {
            value += gs->board[x][y].occupiedByRed * 1236549875 * ( x + 1);
            value += gs->board[x][y].occupiedByBlack * 978642321 * (y + 1);
        }
    }
    return value % HASHSIZE;
}


BOOL equalGameStates( GameState *gs1, GameState *gs2 )
{
    if( gs1->blackPicesOnBoard != gs2->blackPicesOnBoard)
        return FALSE;

    if( gs1->redPicesOnBoard != gs2->redPicesOnBoard )
        return FALSE;

    if( gs1->playerMoving != gs2->playerMoving )
        return FALSE;

```

```
    if( gs1->gameStatus.gameOver != gs2->gameStatus.gameOver )
        return FALSE;

    if( gs1->gameStatus.winner != gs2->gameStatus.winner )
        return FALSE;

    if( gs1->score.red != gs2->score.red )
        return FALSE;

    if( gs1->score.black != gs2->score.black )
        return FALSE;

    int x, y;
    for ( x = 0 ; x < gs1->boardSize.width ; x++ ) {
        for ( y = 0 ; y < gs1->boardSize.height ; y++ ) {
            if( gs1->board[x][y].occupiedByBlack != gs2->board[x][y].occupiedByBlack )
            {
                return FALSE;
            }
            if( gs1->board[x][y].occupiedByRed != gs2->board[x][y].occupiedByRed )
            {
                return FALSE;
            }
        }
    }
    return TRUE;
}

HASHList* findInHashTable( GameState *gs )//, HASHList hashtable[] )
{
    HASHList *test;

    if( &(hashtable[hashValue( gs )]) == NULL )
        return NULL;

    for( test = hashtable[hashValue( gs )] ; test != NULL ; test = test->next )
    {
        if ( equalGameStates( gs, &(test->state) ) )
            return test;
    }
    return NULL;
}

BOOL insertIntoHashTable( GameState *gs, int level , int badscore)
{
    struct hashList *test;
    unsigned hashvalue;

    if( (findInHashTable( gs )) == NULL )
    {
        test = malloc( sizeof( HASHList ));
```

```

        hashvalue = hashValue( gs );

        test->foundNow = TRUE;
        test->level = level;

        test->score = badscore;

        test->state.blackPicesOnBoard = gs->blackPicesOnBoard;
        test->state.redPicesOnBoard = gs->redPicesOnBoard;

        test->state.playerMoving = gs->playerMoving;
        test->state.gameStatus = gs->gameStatus;

        test->state.score = gs->score;

        test->state.board = malloc(gs->boardSize.width * sizeof(
            BoardFieldContent *));

        int i;
        for(i = 0; i < gs->boardSize.width; i++)
        {
            test->state.board[i] = malloc(gs->boardSize.height *
                sizeof(BoardFieldContent));
        }

        int x, y;
        for ( x = 0 ; x < gs->boardSize.width ; x++ ) {
            for ( y = 0 ; y < gs->boardSize.height ; y++ ) {
                test->state.board[x][y] = gs->board[x][y];
            }
        }

        test->next = hashtable[hashvalue];
        hashtable[hashvalue] = test;

        return TRUE;
    }
    return FALSE;

}

void freeHashTable()
{
    int i;
    for( i = 0 ; i < HASHSIZE ; i ++ )
    {
        struct hashList *nextstate = hashtable[i];

        while(  nextstate != NULL )
        {
            struct hashList *ok = nextstate->next;
            free( nextstate );
            nextstate = ok;
        }
```

```
        hashtable[i] = NULL ;
    }
}

void NewCalcHashTable()
{
    int i;
    for( i = 0 ; i < HASHSIZE ; i ++ )
    {
        struct hashList *nextstate = hashtable[i];

        while(  nextstate != NULL )
        {
            struct hashList *ok = nextstate->next;
            nextstate->foundNow = FALSE;
            nextstate = ok;
        }

        if( hashtable[i] != NULL )
            hashtable[i]->foundNow = FALSE;
    }
}
```

C.3.3 AdvancedAI.h

```
//  Kolibrat AI
//  AdvancedAI.h
//
//  Created by Aron Lindberg.

#import "Datastructures.h"
#import "PlayerProtocol.h"
#import "AIDefinitions.h"
#import "GameEngine.h"
#import "GameLogic.h"

@interface AdvancedAI : NSObject < Player_Protocol >
{
// Private instance variabels.
@private
    NSString *name;
    id engine;
    id gl;

    BOOL playerID;
    BOOL enemyID;

    int otherPlayer;
    BOOL waitingForOtherPlayer;

    NSDate *timeToReturnMove;

```

```
    struct hashList *hashtable[HASHSIZE];

    int val1, val2, val3, val4, val5, val6, val7, val8, val9,
        val10, val11;

}
// Instane methods.
-(int)minValue:(TreeState*)treeState scorePointer:(int*)bestScore
    boardMovePointer:(BoardMove*)bestMove searchDepth:(int)maxLevel
    ;
-(int)maxValue:(TreeState*)treeState scorePointer:(int*)bestScore
    boardMovePointer:(BoardMove*)bestMove searchDepth:(int)maxLevel
    ;
- (BoardMove)searchForMove:(GameState)gs;
- (id)initAsPlayer:(int)player withName:(NSString *)playerName
    boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(
    int)maxGoals;
- (void)reset;
- (void)setGameEngine:(id)ge;
- (int)eval:(TreeState *)ts;
- (NSString *)playerName;

// Class methods.
+ (NSString *)playerType;

@end
```

C.3.4 AdvancedAI.m

```
//  Kolibrat AI
//  AdvancedAI.m
//
//  Created by Aron Lindberg.

#import "AdvancedAI.h"

#define TIME_TO_THINK 2.0

@implementation AdvancedAI

- (void) dealloc
{
    [name release];
    [gl release];
    [super dealloc];
}

// Called when the player gets the turn.
- (void)startNewTurn
{
    if( [engine playerMove: [self searchForMove: [engine gameState
        ]] fromPlayer:self] == FALSE )
```

```
        NSLog(@"SOS the engine denied movement.");
}

// The method that begins the MiniMax Search.
- (BoardMove)searchForMove:(GameState)gs
{
    int bestScore = INT_MIN;
    BoardMove bestMove = allLegalMoves( &gs ).head->moveData;

    timeToReturnMove = [NSDate dateWithTimeIntervalSinceNow: TIME_TO_THINK];

    int depth = 1 ;
    BoardMove lastbestMove;

    while ( [timeToReturnMove timeIntervalSinceNow] > 0 )
    {
        NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init ];

        TreeState searchTree = makeTreeState(&gs , NULL);
        bestScore = INT_MIN;
        lastbestMove = bestMove;

        [self maxValue:&searchTree scorePointer:&bestScore boardMovePointer:&bestMove searchDepth:depth ];
        depth ++;
        freeTreeState(&searchTree);

        NewCalcHashTable();

        [pool release];
    }

    freeHashTable();
    return lastbestMove;
}

-(int)maxValue:(TreeState*)treeState scorePointer:(int*)bestScore boardMovePointer:(BoardMove*)bestMove searchDepth:(int)maxLevel
{

    BOOL updateScore = FALSE;
    insertIntoHashTable( &treeState->gs , treeState->level, INT_MIN );

    if( treeState->level == maxLevel || treeState->gs.gameStatus.gameOver == TRUE || [timeToReturnMove timeIntervalSinceNow] < 0 )
    {

        if( treeState->level == 1 && treeState->boardScore > *bestScore)
        {
            *bestMove = treeState->gs.lastMove;
```

```
            *bestScore = treeState->boardScore;
        }

        if ( treeState->level == 1 && treeState->
            gs.gameStatus.winner == playerID &&  treeState->
            gs.gameStatus.gameOver == TRUE )
        {
            *bestMove = treeState->gs.lastMove;
            *bestScore = INT_MAX;
            if( updateScore == TRUE )
                findInHashTable(&treeState->gs)->score = treeState-
                    >boardScore;
            return INT_MAX;
        }
        if( updateScore == TRUE )
            findInHashTable(&treeState->gs)->score = treeState->
                boardScore;
        return [self eval:treeState];
    }

    treeState->boardScore = INT_MIN;

    SimpleList allMoves = makeSimpleList();
    allMoves = allLegalMoves(&treeState->gs);

    while ( allMoves.head != NULL)
    {
        TreeState new = makeTreeStateChild( treeState );
        new.level = treeState->level + 1 ;
        makeMoveOnState( &allMoves.head->moveData  ,&new.gs);

        if(  findInHashTable( &new.gs ) == NULL || (findInHashTable
            ( &new.gs )->level  == new.level && findInHashTable( &
            new.gs )->foundNow == FALSE   ) )
        {
            if( findInHashTable( &new.gs ) != NULL )
            {
                findInHashTable( &new.gs )->foundNow = TRUE;
            }
            if( new.gs.playerMoving == enemyID )
                treeState->boardScore  = MAX( treeState->
                    boardScore,   [self minValue:&new  scorePointer
                    :bestScore boardMovePointer:bestMove
                    searchDepth:maxLevel] );
            else
            {
                treeState->boardScore  = MAX( treeState->
                    boardScore, [self maxValue:&new  scorePointer:
                    bestScore boardMovePointer:bestMove searchDepth
                    :maxLevel] );
            }
        }

        if( treeState->boardScore >= treeState->b )
        {
```

```
            freeTreeState(&new);
            freeSimpleList(&allMoves);

            if( treeState->level == 1 && treeState->boardScore > *bestScore)
            {
                *bestMove = treeState->gs.lastMove;
                *bestScore = treeState->boardScore;
            }
            findInHashTable(&(treeState->gs))->score = treeState->boardScore;
            return treeState->boardScore;
        }

        if( treeState->boardScore > treeState->a )
        {
            treeState->a = MAX( treeState->a, treeState->boardScore );
        }

        removeHeadFromSimpleList(&allMoves);
        freeTreeState(&new);
    }

    freeSimpleList(&allMoves);

    if( treeState->level == 1 && treeState->boardScore > *bestScore )
    {
        *bestMove = treeState->gs.lastMove;
        *bestScore = treeState->boardScore;
    }
    if( updateScore == TRUE )
        findInHashTable(&treeState->gs)->score = treeState->boardScore;
    return treeState->boardScore;
}

-(int)minValue:(TreeState*)treeState scorePointer:(int*)bestScore boardMovePointer:(BoardMove*)bestMove searchDepth:(int)maxLevel
{
    BOOL updateScore = FALSE;
    insertIntoHashTable( &treeState->gs , treeState->level, INT_MAX );

    if( treeState->level == maxLevel || treeState->gs.gameStatus.gameOver == TRUE || [timeToReturnMove timeIntervalSinceNow] < 0 )
    {
        if( treeState->level == 1 && treeState->boardScore > *bestScore)
        {
            *bestMove = treeState->gs.lastMove;
            *bestScore = treeState->boardScore;
        }
```

```

        if ( treeState->level == 1 && treeState->gs.gameStatus.winner == playerID &&  treeState->gs.gameStatus.gameOver == TRUE )
        {
            *bestMove = treeState->gs.lastMove;
            *bestScore = INT_MAX;
            if( updateScore == TRUE )
                findInHashTable(&treeState->gs)->score = treeState->boardScore;
            return INT_MAX;
        }

        if( updateScore == TRUE )
            findInHashTable(&treeState->gs)->score = treeState->boardScore;
        return [self eval:treeState];
    }

    treeState->boardScore = INT_MAX;

    SimpleList allMoves = makeSimpleList();
    allMoves = allLegalMoves(&treeState->gs);

    while ( allMoves.head != NULL)
    {
        TreeState new = makeTreeStateChild( treeState );
        new.level = treeState->level + 1;
        makeMoveOnState( &allMoves.head->moveData  ,&new.gs);

        if( findInHashTable( &new.gs ) == NULL || (findInHashTable( &new.gs )->level  == new.level && findInHashTable( &new.gs )->foundNow == FALSE   ) )
        {
            if( findInHashTable( &new.gs ) != NULL )
                findInHashTable( &new.gs )->foundNow = TRUE;

            if( new.gs.playerMoving == playerID )
                treeState->boardScore  = MIN( treeState->boardScore, [self maxValue:&new  scorePointer:bestScore boardMovePointer:bestMove searchDepth:maxLevel]);
            else
                treeState->boardScore  = MIN( treeState->boardScore, [self minValue:&new  scorePointer:bestScore boardMovePointer:bestMove searchDepth:maxLevel]);
        }

        if( treeState->boardScore <= treeState->a )
        {
            freeTreeState(&new);
            freeSimpleList(&allMoves);

```

```
                if( treeState->level == 1 && treeState->boardScore > *bestScore)
                {
                    *bestMove = treeState->gs.lastMove;
                    *bestScore = treeState->boardScore;
                }
                if( updateScore == TRUE )
                    findInHashTable(&treeState->gs)->score = treeState->boardScore;
                return treeState->boardScore;
            }

            if( treeState->boardScore < treeState->b )
                treeState->b = MIN( treeState->b , treeState->boardScore);

            removeHeadFromSimpleList( &allMoves);
            freeTreeState(&new);
        }

        freeSimpleList(&allMoves);

        if( treeState->level == 1 && treeState->boardScore > *bestScore )
        {
            *bestMove = treeState->gs.lastMove;
            *bestScore = treeState->boardScore;
        }

        if( updateScore == TRUE )
            findInHashTable(&treeState->gs)->score = treeState->boardScore;
        return treeState->boardScore;
}

- (int)eval:(TreeState *)ts
{
// val1: Value of a piece on the buttom line.
// val2: Value increce for a pice per leve.
// val3: Bonus for standing in the middel of the board.
// val4: Penalty for standing in front of the enemy, when they have the turn.
// val5: Bonus for having 2 pieces on a row.
// val6: The value of a legal move.
// val7: The value of a goal.
// val8: the value of having the turn.
// val9: the value of beeing able to insert pieces on the board.
// val10: The value of having a piece standing on the opponents home line.
// val11: The value of being the player having most pieces on the board.

    int redScore = 0 ;
    int blackScore = 0 ;

```

```
    if( GAME_NOT_RUNNING( ts->gs.gameStatus ))
    {
        if( ts->gs.gameStatus.winner == PLAYER_RED )
        {
            redScore = (INT_MAX - 100 ) - ts->level ;
            blackScore = 0 ;
        }
        if( ts->gs.gameStatus.winner == PLAYER_BLACK )
        {
            blackScore = (INT_MAX - 100 ) - ts->level ;
            redScore = 0 ;
        }
        return playerID == PLAYER_RED ? redScore - blackScore :
            blackScore - redScore;
    }

    BOOL realPlayerMovin = ts->gs.playerMoving;

    // Calculate score for red:
    int x, y;
    for ( x = 0 ; x < ts->gs.boardSize.width ; x++ ) {
        for ( y = 0 ; y < ts->gs.boardSize.height ; y++ ) {
            if( RED_FIELD( ts->gs.board[x][y] ))
            {
                redScore += val1 + val2 * y;

                if( x > 0 && x < ts->gs.boardSize.width - 1 )
                    redScore += val3;

                if( y < ( ts->gs.boardSize.height - 1 )  &&
                    BLACK_FIELD( ts->gs.board[ x ][ y + 1 ] ) &&
                    ts->gs.playerMoving == PLAYER_BLACK )
                {
                    redScore -= val4;
                }

                if( y < ( ts->gs.boardSize.height - 1 )  &&
                    RED_FIELD( ts->gs.board[ x ][ y + 1 ] ))
                {
                    redScore += val5;
                }
            }
        }
    }

    for ( x = 0 ; x < ts->gs.boardSize.width ; x++ )
    {
        if( RED_FIELD( ts->gs.board[x][ts->gs.boardSize.height] ))
        {
            redScore += val10;
        }
    }

    if( ts->gs.redPicesOnBoard > ts->gs.blackPicesOnBoard  )
    {
```

```
            redScore += val11;
        }

        if( realPlayerMovin == PLAYER_RED )
        {
            redScore += val8;
        }

        ts->gs.playerMoving = PLAYER_RED;

        if( [gl playerMovingCanInsertPieceOnState: &ts->gs] )
        {
            redScore += val9;
        }

        NSSet *redMoves = [gl allLegalMoves:  &ts->gs ];
        redScore += [redMoves count] * val6;
        redScore += ts->gs.score.red * val7;

        // Calculate score for black:
        for ( x = 0 ; x < ts->gs.boardSize.width ; x++ ) {
            for ( y = 0 ; y < ts->gs.boardSize.height ; y++ ) {
                if( BLACK_FIELD( ts->gs.board[x][y] ))
                {
                    blackScore += val1 + val2 * y;

                    if( x > 0 && x < ts->gs.boardSize.width - 1 )
                        blackScore += val3;

                    if( y > 0 && RED_FIELD( ts->gs.board[ x ][ y - 1 ] ) &&  ts->gs.playerMoving == PLAYER_RED )
                    {
                        blackScore -= val4;
                    }

                    if(  y > 0 && RED_FIELD( ts->gs.board[ x ][ y - 1 ] ))
                    {
                        blackScore += val5;
                    }
                }
            }
        }

        for ( x = 0 ; x < ts->gs.boardSize.width ; x++ )
        {
            if( BLACK_FIELD( ts->gs.board[x][ts->gs.boardSize.height] ) )
            {
                blackScore += val10;
            }
        }

        if( realPlayerMovin == PLAYER_BLACK )
        {
```

```
            blackScore += val8;
        }

        ts->gs.playerMoving = PLAYER_BLACK;

        if( [gl playerMovingCanInsertPieceOnState: &ts->gs] )
        {
            blackScore += val9;
        }

        NSSet *blackMoves = [gl allLegalMoves:  &ts->gs ];
        blackScore += [blackMoves count] * val6;

        if( ts->gs.blackPicesOnBoard > ts->gs.redPicesOnBoard  )
        {
            blackScore += val11;
        }

        blackScore += ts->gs.score.black * val7;

        if( GAME_NOT_RUNNING( ts->gs.gameStatus ))
        {
            if( ts->gs.gameStatus.winner == PLAYER_RED )
            {
                redScore = (INT_MAX - 100 ) - ts->level ;
                blackScore = 0 ;
            }
            if( ts->gs.gameStatus.winner == PLAYER_BLACK )
            {
                blackScore = (INT_MAX - 100 ) - ts->level ;
                redScore = 0 ;
            }
        }

        ts->gs.playerMoving = realPlayerMovin;
        return playerID == PLAYER_RED ? redScore - blackScore :
            blackScore - redScore;
}

- (id)initAsPlayer:(int)player withName:(NSString *)playerName
        boardSize:(BoardSize)bs picesOnboard:(int)maxPices goalsToWin:(
        int)maxGoals
{
        self = [super init];
        if (self != nil)
        {
            playerID = player;
            enemyID = !playerID;

            if( playerID == PLAYER_RED )
                otherPlayer = PLAYER_BLACK;
            else
                otherPlayer = PLAYER_RED;

```

```
        gl = [[GameLogic alloc] initWithMaxPices:maxPices
            goalsToWin:maxGoals boardSize:bs];

        name = [NSString stringWithString:playerName];
        [name retain];

        val1 = 50;
        val2 = 26;
        val3 = 52;
        val4 = 57;
        val5 = 52;
        val6 = 33;
        val7 = 100;
        val8 = 9;
        val9 = 53;
        val10 = 3;
        val11 = 17;
    }
    return self;
}

- (void)setGameEngine:(id)ge
{
    engine = ge;
}

- (void)reset
{
    // Nothing to reset.
}

// The name of this instance of player.
- (NSString *)playerName
{
    return name;
}

// Defines the name of this type of player.
+ (NSString *)playerType
{
    return [NSString stringWithFormat:@"Advanced AI@%1.1f",
        TIME_TO_THINK];
}

@end
```

C.4 Simulated Annealing Source Code

C.4.1 simAneling.h

```
//  simAnnealing
```

```
//   simAnnealing.h
//
//   Created by Aron Lindberg.

#import <Cocoa/Cocoa.h>
#import <Foundation/Foundation.h>
#import "Datastructures.h"
#import "GameLogic.h"
#import "AIPlayer2.h"
#import "PlayerProtocol.h"
#import "AIDefinitions.h"
#import "GameEngine.h"

typedef struct eval {
    int val1;
    int val2;
    int val3;
    int val4;
    int val5;
    int val6;
    int val7;
    int val8;
    int val9;
    int val10;
    int val11;
} EVAL_VARS;

@interface simAneling : NSObject {

@private
    GameEngine *newEngine;
    AIPlayer *red;
    AIPlayer *black;

    EVAL_VARS currentRed, currentBlack;

    int nextAi;
    int aiInWinner;
    int aiInLoser;

    EVAL_VARS AI_stuff[60];
    EVAL_VARS AI_winners[30];
    EVAL_VARS AI_losers[30];

    int generation;
    float randfactor;

    NSFileHandle *fh;
    EVAL_VARS nextplayer1, nextplayer2;
}

- (void)awakeFromNib;
- (EVAL_VARS) randomise:(EVAL_VARS)this;

@end
```

C.4.2 simAneling.m

```
//  simAnnealing
//  simAnnealing.h
//
//  Created by Aron Lindberg.

#import "simAneling.h"

#define GOALS_TO_WIN 1
#define PICES_ON_BOARD 6
#define BOARD makeBoardSize( 4, 6)

@implementation simAneling

- (void)awakeFromNib
{
    randfactor = 1.0;

    fh = [NSFileHandle fileHandleForWritingAtPath:@"/Users/Output.txt"];
    [fh retain];
    [fh writeData:[@"Start of new Calculation.\n" dataUsingEncoding:NSASCIIStringEncoding]];

    srandom([[NSDate date] timeIntervalSince1970]);

    int i;
    for ( i = 0 ; i < 60 ; i++ )
    {
        AI_stuff[i].val1 = 600 + random() % 200 - 100;
        AI_stuff[i].val2 = 50 + random() % 200 - 100;
        AI_stuff[i].val3 = 730 + random() % 200 - 100;
        AI_stuff[i].val4 = 760 + random() % 200 - 100;
        AI_stuff[i].val5 =  500 + random() % 200 - 100;
        AI_stuff[i].val6 = 70 + random() % 200 - 100;
        AI_stuff[i].val7 = 1000 + random() % 200 - 100;
        AI_stuff[i].val8 =  50 + random() % 200 - 100;
        AI_stuff[i].val9 = 50 + random() % 200 - 100;
        AI_stuff[i].val10 = 50 + random() % 200 - 100;
        AI_stuff[i].val11 = 50 + random() % 200 - 100;

        [fh writeData:[[NSString stringWithFormat:@"AI_stuff[%i]: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i, %i\n", i , AI_stuff[i].val1,AI_stuff[i].val2 , AI_stuff[i].val3, AI_stuff[i].val4, AI_stuff[i].val5, AI_stuff[i].val6, AI_stuff[i].val7, AI_stuff[i].val8, AI_stuff[i].val9, AI_stuff[i].val10, AI_stuff[i].val11 ] dataUsingEncoding:NSASCIIStringEncoding]];
        NSLog(@"AI_stuff[%i]: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i, %i\n", i , AI_stuff[i].val1,AI_stuff[i].val2 , AI_stuff[i].val3, AI_stuff[i].val4, AI_stuff[i].val5, AI_stuff[i].val6, AI_stuff[i].val7, AI_stuff[i].val8, AI_stuff[i].val9, AI_stuff[i].val10, AI_stuff[i].val11)
```

```
            ;
    }

    generation = 0;
    nextAi = 0;

    NSNotificationCenter *mainCenter = [NSNotificationCenter defaultCenter];

    [mainCenter addObserver:self selector:@selector(RestartGame:) name:@"GameOver" object:nil];

    red = [[AIPlayer alloc] initAsPlayer:PLAYER_RED withName: @"RED" boardSize:BOARD picesOnboard:PICES_ON_BOARD goalsToWin: INT_MAX];
    black = [[AIPlayer alloc] initAsPlayer:PLAYER_BLACK withName: @"BLACK" boardSize:BOARD picesOnboard:PICES_ON_BOARD goalsToWin: INT_MAX];

    currentRed = AI_stuff[nextAi];
    nextAi++;
    currentBlack = AI_stuff[nextAi];

    nextAi++;

    [red setEvaluationValues:AI_stuff[0]];
    [black setEvaluationValues:AI_stuff[1]];

    newEngine = [[GameEngine alloc] initWithPlayersRed:red
                                              andBlack:black
                                            goalsToWin:GOALS_TO_WIN
                                          GameBoardDim:BOARD
                                              MaxPices: PICES_ON_BOARD
                                          connectToGUI:nil];
    [newEngine retain];

    [red retain];
    [black retain];

    [red setGameEngine:newEngine];
    [black setGameEngine:newEngine];

    aiInWinner = 0;
    aiInLoser = 0;
}


- (void)RestartGame:(NSNotification *)notification
{
    [fh writeData:[[NSString stringWithFormat:@"\n----> Game %i is over <----\n", aiInWinner + 1 ] dataUsingEncoding:NSASCIIStringEncoding]];

    if( [notification object] == red )
```

```
    {
        AI_winners[aiInWinner] = currentRed;
        aiInWinner++;

        [fh writeData:  [[NSString stringWithFormat:@"the red
            winner had: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i, %i
            \n" , currentRed.val1 ,
            currentRed.val2 , currentRed.val3 , currentRed.val4
                ,currentRed.val5 , currentRed.val6 ,
                currentRed.val7,  currentRed.val8 ,
                currentRed.val9, currentRed.val10, currentRed.val11
                  ] dataUsingEncoding:NSASCIIStringEncoding]]  ;

        [fh writeData: [[NSString stringWithFormat:   @"the black
            loser had: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i, %i \
            n",currentBlack.val1,currentBlack.val2 ,
            currentBlack.val3, currentBlack.val4,
            currentBlack.val5, currentBlack.val6,
                currentBlack.val7, currentBlack.val8,
                currentBlack.val9, currentBlack.val10,
                currentBlack.val11] dataUsingEncoding:
                NSASCIIStringEncoding]];

    }

    else if( [notification object] == black )
    {
        AI_winners[aiInWinner] = currentBlack;
        aiInWinner++;
        [fh writeData: [[NSString stringWithFormat: @"the black
            winner had: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i, %i
            \n",currentBlack.val1,currentBlack.val2 ,
            currentBlack.val3, currentBlack.val4,
            currentBlack.val5, currentBlack.val6,
            currentBlack.val7, currentBlack.val8,
            currentBlack.val9, currentBlack.val10,
            currentBlack.val11] dataUsingEncoding:
            NSASCIIStringEncoding]];
        [fh writeData:[[NSString stringWithFormat: @"the red loser
            had: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i, %i \n"
            ,currentRed.val1,currentRed.val2 , currentRed.val3,
            currentRed.val4, currentRed.val5, currentRed.val6,
            currentRed.val7, currentRed.val8, currentRed.val9,
            currentRed.val10, currentRed.val11] dataUsingEncoding:
            NSASCIIStringEncoding]];
    }

    if( nextAi != 60 )
    {
        nextplayer1 = AI_stuff[nextAi];
        nextAi++;
        nextplayer2 = AI_stuff[nextAi];
        nextAi++;

        [red setEvaluationValues:nextplayer1];
```

```

        [black setEvaluationValues:nextplayer2];

        currentRed = nextplayer1;
        currentBlack = nextplayer2;
        [newEngine resetGame];
        return;
    }

    if( nextAi == 60  )
    {
        generation++;

        [fh writeData:[[NSString stringWithFormat:@"\n\n\n ----->
            Generating generation %i. Randomfactor is: %1.3f <-----
            \n", generation, randfactor ] dataUsingEncoding:
            NSASCIIStringEncoding]];

        BOOL spotfree[60];

        int i;
        for(i = 0 ; i < 60 ; i++ )
        {
            spotfree[i] = TRUE;
        }

        nextAi = 0;
        aiInWinner = 0;
        aiInLoser = 0;

        int somevar1 = 0;

        for( i=0; i < 30 ; i++ )
        {
            int temp = random() % 60;
            while( spotfree[ temp ] != TRUE )
            {
                temp = random() % 60;
            }

            spotfree[temp] = false;

            AI_stuff[ temp ] = [self randomise: AI_winners[ i ] ] ;
            somevar1 ++;

            temp = random() % 60;
            while( spotfree[ temp ] != TRUE )
            {
                temp = random() % 60;
            }

            spotfree[temp] = false;

            AI_stuff[ temp ] =  AI_winners[ i ];
            somevar1 ++;
```

```
        }

        for ( i= 0 ; i < 60 ; i++ )
        {
            [fh writeData:[[NSString stringWithFormat:@"AI_stuff[%i
                ]: %i, %i, %i, %i, %i, %i, %i, %i, %i, %i, %i\n", i
                 , AI_stuff[i].val1,AI_stuff[i].val2 , AI_stuff[i]
                .val3, AI_stuff[i].val4, AI_stuff[i].val5, AI_stuff
                [i].val6, AI_stuff[i].val7, AI_stuff[i].val8,
                AI_stuff[i].val9, AI_stuff[i].val10, AI_stuff[i]
                .val11 ] dataUsingEncoding:NSASCIIStringEncoding]];
        }

        if( randfactor > 0.05 )
            randfactor = randfactor - 0.02 ;
        else
            randfactor = randfactor - 0.002 ;

        nextplayer1 = AI_stuff[nextAi];
        nextAi++;
        nextplayer2 = AI_stuff[nextAi];
        nextAi++;

        currentRed = nextplayer1;
        currentBlack = nextplayer2;

        if( randfactor > 0 )
            [newEngine resetGame];
    }
}

-(EVAL_VARS) randomise:(EVAL_VARS)this
{
    int v1 =  (random() % 1001 ) - 500 ;
    int v2 =  (random() % 1001 ) - 500 ;
    int v3 =  (random() % 1001 ) - 500 ;
    int v4 =  (random() % 1001 ) - 500 ;
    int v5 =  (random() % 1001 ) - 500 ;
    int v6 =  (random() % 1001 ) - 500 ;
    int v7 =  (random() % 1001 ) - 500 ;
    int v8 =  (random() % 1001 ) - 500 ;
    int v9 =  (random() % 1001 ) - 500 ;
    int v10 =  (random() % 1001 ) - 500 ;
    int v11 =  (random() % 1001 ) - 500 ;

    v1 = (int) v1 * randfactor;
    v2 = (int) v2 * randfactor;
    v3 = (int) v3 * randfactor;
    v4 = (int) v4 * randfactor;
    v5 = (int) v5 * randfactor;
    v6 = (int) v6 * randfactor;
    v7 = (int) v7 * randfactor;
    v8 = (int) v7 * randfactor;
    v9 = (int) v7 * randfactor;
    v10 = (int) v7 * randfactor;
```

```
    v11 = (int) v7 * randfactor;

    this.val1 += v1;
    this.val2 += v2;
    this.val3 += v3;
    this.val4 += v4;
    this.val5 += v5;
    this.val6 += v6;
    this.val7 += v7;
    this.val8 += v8;
    this.val9 += v9;
    this.val10 += v10;
    this.val11 += v11;

    int t1 = MIN( this.val1 , this.val2 );
    int t2 = MIN( this.val3 , this.val4 );
    int t3 = MIN( this.val5 , this.val6 );
    int t4 = MIN( this.val7 , this.val8 );
    int t5 = MIN( this.val9 , this.val10 );
    int t6 = MIN( this.val11 , t1 );
    int t7 = MIN (t2, t3 );
    int t8 = MIN (t4, t5 );
    int t9 = MIN (t6, t7 );
    int totalmin = MIN( t8,t9);

    if ( totalmin < 0 )
    {
        this.val1 += abs(totalmin);
        this.val2 += abs(totalmin);
        this.val3 += abs(totalmin);
        this.val4 += abs(totalmin);
        this.val5 += abs(totalmin);
        this.val6 += abs(totalmin);
        this.val7 += abs(totalmin);
        this.val8 += abs(totalmin);
        this.val9 += abs(totalmin);
        this.val10 += abs(totalmin);
        this.val11 += abs(totalmin);
    }

    t1 = MAX( this.val1 , this.val2 );
    t2 = MAX( this.val3 , this.val4 );
    t3 = MAX( this.val5 , this.val6 );
    t4 = MAX( this.val7 , this.val8 );
    t5 = MAX( this.val9 , this.val10 );
    t6 = MAX( this.val11 , t1 );
    t7 = MAX (t2, t3 );
    t8 = MAX (t4, t5 );
    t9 = MAX (t6, t7 );
    int totalmax = MAX( t8 , t9 );

    if ( totalmax > 1000 )
    {
        this.val1 =  (int)(((float)this.val1 * 1000) / (totalmax)
            ) ;
```

```
        this.val2 =  (int)(((float)this.val2 * 1000) / (totalmax)) ;
        this.val3 =  (int)(((float)this.val3 * 1000) / (totalmax)) ;
        this.val4 =  (int)(((float)this.val4 * 1000) / (totalmax)) ;
        this.val5 =  (int)(((float)this.val5 * 1000) / (totalmax)) ;
        this.val6 =  (int)(((float)this.val6 * 1000) / (totalmax)) ;
        this.val7 =  (int)(((float)this.val7 * 1000) / (totalmax)) ;
        this.val8 =  (int)(((float)this.val8 * 1000) / (totalmax)) ;
        this.val9 =  (int)(((float)this.val9 * 1000) / (totalmax)) ;
        this.val10 =  (int)(((float)this.val10 * 1000) / (totalmax)) ;
        this.val11 =  (int)(((float)this.val11 * 1000) / (totalmax)) ;
    }
    return this;
}

@end
```

C.5 Forced Loops Source Code

C.5.1 FakeLogic.h

```
//  Forced Loops
//  Fake Logic.h
//
//  Created by Aron Lindberg.

#import <Cocoa/Cocoa.h>

@interface FakeLogic : NSObject {

// Private instance variabels.
@private
    int maxGoals;
    int maxPicesOnBoard;
    BoardSize boardSize;
    int lastState;
}

// Public instance methods.
- (id)initWithMaxPices:(int)max goalsToWin:(int)goals boardSize:(BoardSize)board;
- (GameState)CreateNewGameState;
```

```
- (void)resetGameState:(GameState *)gs;
- (NSSet *)allLegalMoves:(GameState *)gs;
- (BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs;

@end
```

C.5.2 FakeLogic.m

```
//  Forced Loops
//  Fake Logic.m
//
//  Created by Aron Lindberg.

#import "FakeLogic.h"

@implementation FakeLogic

static int *maxGoalsPointer;
static int *maxPicesOnBoardPointer;
static BoardSize *boardSizePointer;

- (id)initWithMaxPices:(int)max goalsToWin:(int)goals boardSize:(
    BoardSize)board
{
    self = [super init];
    if (self != nil)
    {
        maxPicesOnBoard = max;
        boardSize = board;
        maxGoals = goals;
        maxGoalsPointer = &maxGoals;
        maxPicesOnBoardPointer = &maxPicesOnBoard;
        boardSizePointer = &boardSize;
    }
    return self;
}

- (GameState)CreateNewGameState
{
    GameState gs = makeGameState(boardSize);
    [self resetGameState: &gs];
    return gs;
}

- (void)resetGameState:(GameState *)gs
{
    int x, y;
    for ( x = 0 ; x < boardSize.width ; x++ ) {
        for ( y = 0 ; y < boardSize.height ; y++ ) {
            gs->board[x][y] = makeEmptykField();
        }
    }
```

```
    gs->score = makeGameScore(0,0);
    gs->lastMove = NIL_MOVE;
    gs->playerMoving = PLAYER_RED;
    gs->redPicesOnBoard = 0;
    gs->blackPicesOnBoard = 0;
    gs->boardSize = boardSize; // TODO
    gs->gameStatus = RUNNING;
}

- (NSSet *)allLegalMoves:(GameState *)gs
{
    NSMutableSet *setOfLegalMoves = [NSMutableSet setWithCapacity:10];

    if( gs->score.red == 0 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc] initWithMove:makeMove(makeBoardField(-2,100),makeBoardField(0,0))]];
        [setOfLegalMoves addObject: [[MoveObject alloc] initWithMove:makeMove(makeBoardField(-2,120),makeBoardField(0,0))]];
    }
    else if( gs->score.red == 100 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc] initWithMove:makeMove(makeBoardField(-2,101),makeBoardField(0,0))]];
        [setOfLegalMoves addObject: [[MoveObject alloc] initWithMove:makeMove(makeBoardField(-2,103),makeBoardField(0,0))]];
    }
    else if( gs->score.red == 101 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc] initWithMove:makeMove(makeBoardField(-1,-10),makeBoardField(0,0))]];
        [setOfLegalMoves addObject: [[MoveObject alloc] initWithMove:makeMove(makeBoardField(-2,102),makeBoardField(0,0))]];
        [setOfLegalMoves addObject: [[MoveObject alloc] initWithMove:makeMove(makeBoardField(-2,200),makeBoardField(0,0))]];
    }
    else if( gs->score.red == 102 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc] initWithMove:makeMove(makeBoardField(-1,-11),makeBoardField(0,0))]];
    }
    else if( gs->score.red == 103 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc] initWithMove:makeMove(makeBoardField(-2,107),makeBoardField(0,0))]];
```

```
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,104)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 104 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,105)
            ,makeBoardField(0,0))]];

    }
    else if( gs->score.red == 105 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,106)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 106 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-1,-11)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 107 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,108)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 108 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,109)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 109 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,110)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 110 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,111)
            ,makeBoardField(0,0))]];
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-1,10)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 111 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,121)
            ,makeBoardField(0,0))]];  //121
```

```
    }
    else if( gs->score.red == 120 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,121)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 121 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,122)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 122 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,103)
            ,makeBoardField(0,0))]];
    }
    else if( gs->score.red == 200 )
    {
        [setOfLegalMoves addObject: [[MoveObject alloc]
            initWithMove:makeMove(makeBoardField(-2,0)
            ,makeBoardField(0,0))]];
    }
    else
    {
        NSLog(@"Warning, states out of scroop.");
    }
    [setOfLegalMoves retain];
    return setOfLegalMoves;
}

- (BOOL)makeMove:(BoardMove)playerMove withState:(GameState *)gs
{
lastState = gs->score.red;
    if( playerMove.from.x == -1 && playerMove.from.y == -10 )
    {
        gs->gameStatus.gameOver = TRUE;
        gs->gameStatus.winner = PLAYER_RED;
        gs->score.red = -10;
    }
    else if(  playerMove.from.x == -1 && playerMove.from.y == -11 )
    {
        gs->gameStatus.gameOver = TRUE;
        gs->gameStatus.winner = PLAYER_BLACK;
        gs->score.red = -11;
    }
    else if(  playerMove.from.x == -2 )
    {
        gs->score.red = playerMove.from.y;
        gs->playerMoving = !gs->playerMoving;
    }
    return TRUE;
}
```

```

@end
```

C.5.3 Forced Loops.m

```
//   Kolibrat
//   GameLogic.h
//
//   Created by Aron Lindberg.

#import <Foundation/Foundation.h>
#import "Datastructures.h"
#import "GameLogic.h"

@implementation gameStateObject

- (BOOL)hasLoop
{
    return thisPathHasLoop;
}

- (NSMutableSet *)parrents
{
    return parrents;
}

- (int) numberOfChildren
{
    return [childs count];
}

- (GameState) returnState
{
    return copyGameState( &state );
}

- (void) addParrent:(gameStateObject *)newParrent
{
    [parrents addObject:newParrent];
}

-(void) releaseAllChildren
{
    [childs removeAllObjects];
    [childs release];
}

- (id) initWithGameState:(GameState)newState parrent:(
    gameStateObject*)motherObject makeChildren:(BOOL)makeChildren
{
    self = [super init];
    if (self != nil) {
```

```
        childs = [[NSMutableSet alloc] initWithCapacity:10];
        state = copyGameState( &newState );

        parrents = [[NSMutableSet alloc] initWithCapacity:3];
        if( motherObject != nil )
            [self addParrent:motherObject];

        thisPathHasLoop = NO;
        ChildrenIsFound = NO;
    }
    return self;
}

- (void) dealloc
{
    if( [childs count] != 0)
    {
        [childs removeAllObjects];
        [childs release];
    }
    [super dealloc];
}

- (NSArray*) findMyChildren
{
    [childs removeAllObjects];

    BoardSize board = makeBoardSize( BOARDHEIGHT, BOARDWIDTH);
    GameLogic *logic = [[GameLogic alloc] initWithMaxPices:MAX_PIECES goalsToWin:GOALS_TO_WIN boardSize:board];

    NSSet * moves = [NSSet setWithSet: [logic allLegalMoves: &state]];
    activechildren = [moves count];
    NSEnumerator * enumerator = [moves objectEnumerator];
    MoveObject *currentMove;
    gameStateObject * childObject;

    while (currentMove = [enumerator nextObject])
    {
        GameState child = copyGameState( &state );
        [logic makeMove:[currentMove retriveMove] withState: &child];

        childObject = [[gameStateObject alloc] initWithGameState:child parrent:self makeChildren:NO ];

        [childs addObject:childObject];
    }
    ChildrenIsFound = TRUE;
    return [childs allObjects];
}
```

```
- (void) childHasLoop:(gameStateObject *)loopChild withStates:(
    NSMutableArray *)states andKnownStates:(NSMutableSet *)
    knownStates testPlayer:(BOOL)plr
{
    if( thisPathHasLoop == TRUE)
        return;

    if( state.playerMoving == plr )
    {
        NSEnumerator * enumerator = [childs objectEnumerator];  //
            for all children
        gameStateObject *currentChildState;

        while (currentChildState = [enumerator nextObject])
        {
            [currentChildState isSubpathOfLoop:states
                andKnownStates:knownStates];
        }

        thisPathHasLoop = TRUE;

        NSEnumerator *e = [parrents objectEnumerator];
        id obj;
        while( obj = [e nextObject])
        {
            [obj childHasLoop:self withStates:states andKnownStates
                :knownStates testPlayer: plr ];
        }
    }

    else
    {
        [loopChild isSubpathOfLoop:states andKnownStates:
            knownStates];

        activechildren--;

        if( activechildren == 0 )
        {
            thisPathHasLoop = TRUE;

            NSEnumerator *e = [parrents objectEnumerator];
            id obj;
            while( obj = [e nextObject])
            {
                [obj childHasLoop:self withStates:states
                    andKnownStates:knownStates testPlayer: plr ];
            }
        }
    }
}

- (void) isSubpathOfLoop:(NSMutableArray *)states andKnownStates:(
    NSMutableSet *)knownStates
{
```

```
    if( thisPathHasLoop == TRUE )
        return;

    NSEnumerator * enumerator = [childs objectEnumerator];

    gameStateObject *currentChildState;

    while (currentChildState = [enumerator nextObject])
    {
        [currentChildState isSubpathOfLoop:states andKnownStates: knownStates ];
    }

    [states removeObject:self];
    [knownStates removeObject:self];

    thisPathHasLoop = TRUE;
}

- (BOOL)isEqual:(id)anObject
{
    if( ![self isKindOfClass: [anObject class]] )
        return FALSE;

    BOOL returnVal = FALSE;

    GameState newState = [anObject returnState];

    if( newState.blackPicesOnBoard == state.blackPicesOnBoard &&
        newState.redPicesOnBoard == state.redPicesOnBoard &&
        newState.playerMoving == state.playerMoving &&
        newState.gameStatus.gameOver == state.gameStatus.gameOver &&
        newState.gameStatus.winner == state.gameStatus.winner &&
        newState.score.red == state.score.red &&
        newState.score.black == state.score.black )

    {
        returnVal = TRUE;

        int x, y;
        for ( x = 0 ; x < BOARDWIDTH ; x++ )
        {
            for ( y = 0 ; y < BOARDHEIGHT ; y++ )
            {
                if( newState.board[x][y].occupiedByRed != state.board[x][y].occupiedByRed ||
                    newState.board[x][y].occupiedByBlack != state.board[x][y].occupiedByBlack )
                {
```

```
                    returnVal = FALSE;
                }
            }
        }
    }

    return returnVal;
}

- (unsigned)hash
{
    unsigned hashVal;

    int x, y;
    for ( x = 0 ; x < BOARDWIDTH ; x++ ) {
        for ( y = 0 ; y < BOARDHEIGHT ; y++ ) {
            if( RED_FIELD( state.board[x][y] ))
            {
                hashVal += y;
            }
        }
    }

    hashVal += state.redPicesOnBoard;

    if( state.playerMoving == PLAYER_RED )
    {
        hashVal =  hashVal * 7;
    }

    hashVal += state.score.red * 3;
    hashVal += state.score.black * 11;

    return hashVal;
}

@end

GameState copyGameState( GameState *state )
{
    struct gameStateStruct temp;

    temp.blackPicesOnBoard = state->blackPicesOnBoard;
    temp.redPicesOnBoard = state->redPicesOnBoard;

    temp.boardSize = state->boardSize;

    temp.playerMoving = state->playerMoving;
    temp.gameStatus = state->gameStatus;

    temp.lastMove = state->lastMove;
    temp.score = state->score;

    temp.board = malloc(BOARDWIDTH * sizeof(BoardFieldContent *));

```

```
    int i;
    for(i = 0; i < BOARDWIDTH; i++)
    {
        temp.board[i] = malloc(BOARDHEIGHT * sizeof(
            BoardFieldContent));
    }

    int x, y;
    for ( x = 0 ; x < BOARDWIDTH ; x++ ) {
        for ( y = 0 ; y < BOARDHEIGHT ; y++ ) {
            temp.board[x][y] = state->board[x][y];
        }
    }
    return temp;
}

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int player;
    for( player = 1 ; player >= 0 ; player--)
    { // Do the test both for the red and black player.

        gameStateObject * rootgameState;

        BoardSize board = makeBoardSize( BOARDHEIGHT, BOARDWIDTH);
        GameLogic *logic = [[GameLogic alloc] initWithMaxPices:
            MAX_PIECES goalsToWin:GOALS_TO_WIN boardSize:board];
        [logic retain];

        NSMutableArray * states = [[NSMutableArray alloc]
            initWithCapacity:50000];
        GameState emptyBoard = [logic CreateNewGameState];

        gameStateObject * emptyBoardObject = [[gameStateObject
            alloc] initWithGameState:emptyBoard parrent: nil
            makeChildren: YES ];

        rootgameState = emptyBoardObject;

        [states addObject:emptyBoardObject];

        gameStateObject * nextGameObject;
        GameState next;
        NSMutableSet * childStates = [[NSMutableSet alloc]
            initWithCapacity:50000];
        NSMutableSet * knownStates = [[NSMutableSet alloc]
            initWithCapacity:50000];

        [knownStates retain];

        int lastActive = 0;
        int val;
        while ( [states count] > 0 )
```

```
        {
            NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
            val++;
            if( val % 10000 == 999 )
                if( lastActive > [states count]  )

                    lastActive = [states count];

            nextGameObject = [states objectAtIndex:0];
            next = [nextGameObject returnState];

            [knownStates addObject:nextGameObject];
            [nextGameObject retain];
            [states removeObjectAtIndex:0];

            if( next.gameStatus.gameOver == TRUE && player == PLAYER_RED )
            {
                //NSLog(@"red win somwhere.");
            }
            else if( next.gameStatus.gameOver == TRUE && player == PLAYER_BLACK )
            {
                //NSLog(@"black win somwhere.");
            }
            else if( next.playerMoving == player )
            {
                NSMutableSet * tempset2 = [[NSMutableSet alloc] initWithArray:[nextGameObject findMyChildren]];

                [tempset2 retain];
                [tempset2 minusSet:knownStates];
                if(  [tempset2 count] ==    [[nextGameObject findMyChildren] count] )
                {
                    [states addObjectsFromArray:[nextGameObject findMyChildren]];
                }
                else
                {
                    NSEnumerator *e = [[nextGameObject parrents] objectEnumerator];
                    id obj;
                    while( obj = [e nextObject])
                    {
                        [obj childHasLoop: nextGameObject withStates:states andKnownStates: knownStates testPlayer:player];
                    }
                }


            }
```

```
            else
            {
                NSMutableSet * tempset = [[NSMutableSet alloc] initWithArray:[nextGameObject findMyChildren]];
                [tempset minusSet:knownStates];

                if( [tempset count] == [[nextGameObject findMyChildren] count]  )
                    [states addObjectsFromArray: [nextGameObject findMyChildren ]];
                else
                {
                    NSMutableSet * tempset3 = [[NSMutableSet alloc] initWithArray:[nextGameObject findMyChildren]]   ;
                    [tempset3 intersectSet:knownStates];

                    NSEnumerator * enumerator = [tempset3 objectEnumerator];
                    gameStateObject *obj;

                    while (obj = [enumerator nextObject])
                    {
                        NSEnumerator * knownE = [knownStates objectEnumerator];
                        id obj2;

                        while (obj2 = [knownE nextObject])
                        {
                            if( [obj2 isEqual:obj] )
                                [obj2 addParrent:nextGameObject];
                        }
                    }

                    NSMutableSet * tempset4 = [[NSMutableSet alloc] initWithArray:[nextGameObject findMyChildren]]   ;

                    [tempset4 minusSet:knownStates];
                    [states addObjectsFromArray: [tempset4 allObjects] ];
                }

            }

            [nextGameObject release];

            [pool release];
        }

        if( [rootgameState hasLoop] && player == PLAYER_RED )
            NSLog(@"Red can enforce a loop.");
        else if( [rootgameState hasLoop] && player == PLAYER_BLACK )
```

```
            NSLog(@"Black can enforce a loop.");
        else if( player == PLAYER_BLACK )
            NSLog(@"Black can not enforce a loop.");
        else if( player == PLAYER_RED )
            NSLog(@"Red can not enforce a loop.");

        [states removeAllObjects];
        [childStates removeAllObjects];
        [knownStates removeAllObjects];
    }
    [pool release];
    return 0;
}
```

Flow Diagrams

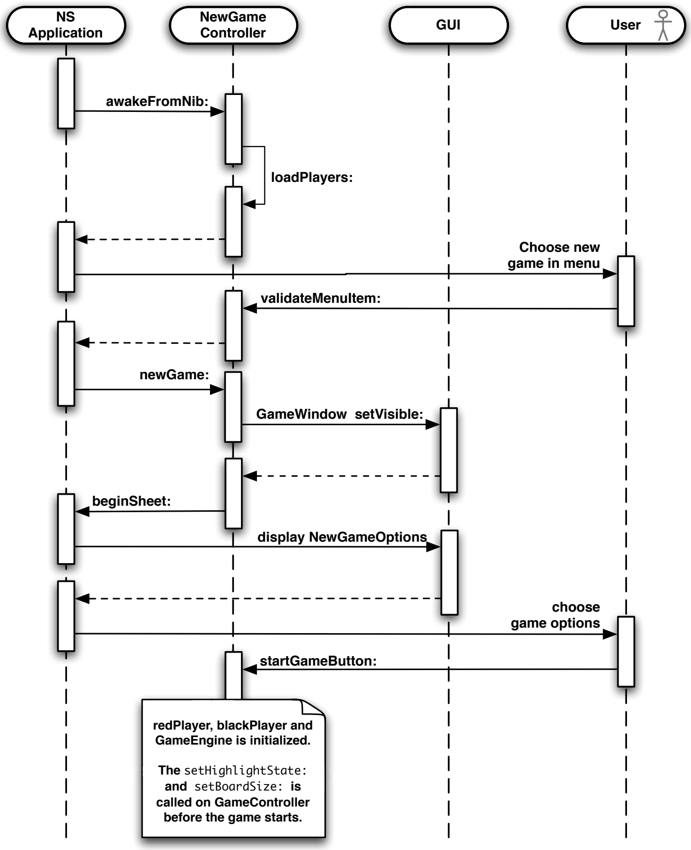


Diagram 2: UML flow diagram of game launch.

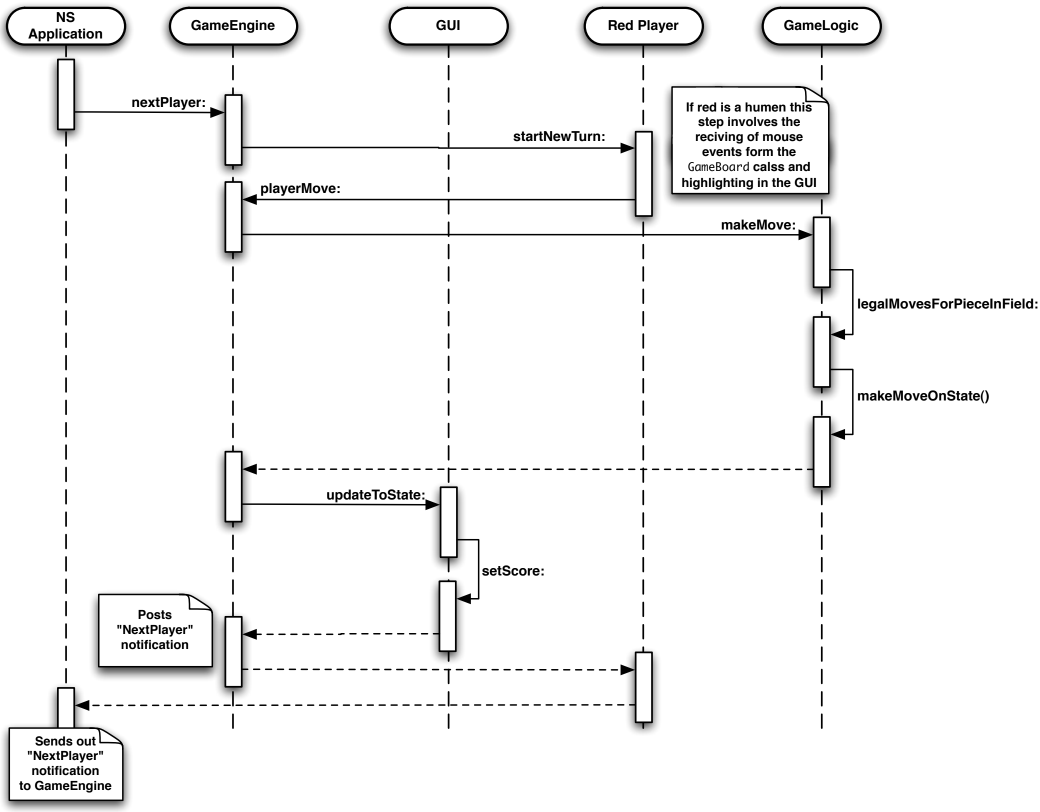


Diagram 3: UML flow diagram while playing through one turn.

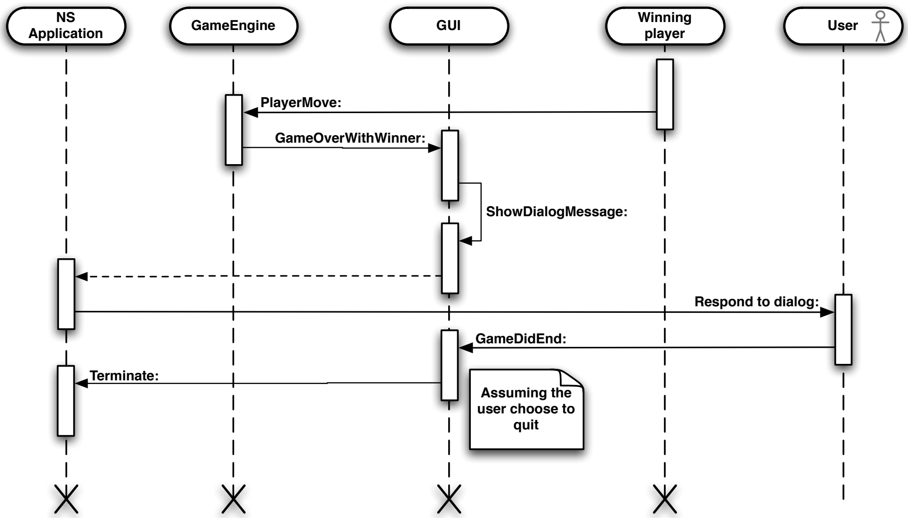


Diagram 4: UML flow diagram when the game is over.

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.