Fist Tablets

James Bruska, David Joesephs, Jacob Melite, Benjamin Lannon

December 11, 2015

Contents

1 Introduction

Our task for the semester was to create an Android Application of the viking game of Hnefatafl. We have a Hnefatafl set in COSI and a few members of the team thought it would be a narrow enough, yet complex application to build. Some of the team have developed for Android in the past, so we decided that it would be our desired platform. The team's job was to implement a UI front-end that a user could move pieces with, and a back-end that would handle the game logic.

2 Problem Description: What is Hnefatafl

The version of Hnefatafl we based our app on uses a 11x11 Grid, where there are 24 black pieces, 12 white pieces, and the king piece. One player takes control of the black pieces and one controls the white and king pieces. The objective of the game is for either the white player to move their king piece to one of the 4 corners or for the black player to stop the king by capturing it. Each piece can move like a rook in chess, where they can move horizontally or vertically. A piece can be captured and removed from the board when two of the opposite color surround it, or in the case of the king, it is surrounded by black on all sides.

3 Methodology

3.1 Design Patterns & Principles

This project uses two main design patterns in order to follow two Java coding principles. The two principles are the *Liskov Substitution Principle* and the *Open Closed Principle.*The Liskov Substitution Principle, in terms of Java, says that if an object has a property and another object extends it, then the child object should have all of the properties of the parent object. The Open Closed Principle can be summarized as all implementations should be open to extension and closed to modification. By following these two principles, a well established program that is safe and still open to change can be created.

The two design patterns implemented that allow this to occur are the Strategy Pattern and the Publisher Subscriber Pattern. The Strategy Pattern is a way to adapt for the Liskov Substitution Principle. With this design pattern, the programmer identifies the things that are changing within a class and then extracts them into a family of algorithms. Then composition can be used to inject this behavior back into the original object. This allows a parent class to have functions that can be modifiable in the child classes and even later at run-time. The Publisher Subscriber Pattern is implemented to help with the Open Closed Principle. This pattern loosens the connection between two objects so that they do not need to know about each other as much. One object acts as publisher and informs the program when something happens. The subscriber objects are constantly listening for a publisher to inform the program of something. When that thing happens, the listeners use the data and then make their own actions. By using these two design patterns, the two coding principles can be followed.

3.2 AI Algorithm

Before any code for the AI could be written, we had to research the optimal algorithm to use, in order to minimize the time it takes for the AI to make a move and to maximize difficulty. Initial research turned up a single paper on Hnefatafl, which stated that Hnefatafl has a branching factor of about 100, about 10 times more than chess, which has a branching factor of around 10-20. The researchers wrote that they used a neural network to create an AI, which we immediately discarded as being too complicated for this project [1]. This led us to asking Professors Sattar, Tamon, and Black for help in creating an AI. Professor Tamon pointed us to a set of algorithms called Branch-and-Bound algorithms, and lent us two books on the topic. Professor Black expanded on Professor Tamon's answer, telling us that the specific Branch-and-Bound algorithm we should use is a Minimax Search with Alpha-Beta Pruning.

The way the algorithm works, in a broad sense, is a Depth-First-Search that remembers the best move so far, and removes sub-optimal moves while building the tree. Moves are determined to be good or bad by assigning a value to each state of the game using an evaluation function, in order to indicate how good the current board state is for either player. The evaluation function we used was based off of the one used in the paper we found. The value the function returns is as follows:

(InitialWhitePieces - RemainingWhitePieces) - (InitialBlackPieces - RemainingBlackPieces) + .1*(KingDistance)

The number of white captures is subtracted from the number of white captures, and the Manhattan Distance of the king to the closest corner is then weighted and added to that. The Manhattan Distance of the king to the closest corner is the number of squares the king has to move in order to reach the nearest corner of the board. The evaluation function is set up so that high values are good for black and low values are good for white. Thus, black is called the "Maximizing Player" and white is called the "Minimizing Player". The algorithm also stores two values, alpha, which represents the maximum value that the maximizing player is guaranteed to achieve, and beta, which represents the minimum value that the minimizing player is guaranteed to achieve. The program initializes alpha to negative infinity and beta to positive infinity. If beta ever drops below or becomes equal to alpha, the node that this happened on can be immediately discarded without exploring the rest of its branches, since the parent node's value will always be better than any value the current node could produce.

The algorithm starts by generating a list of all valid moves for the current player. A random move is selected and executed. The algorithm then recurses until either a player wins or it has looked too far ahead.

We set a limit on the AI to look only 3 moves ahead, since looking 4 moves ahead increases the time required to find the best move from about half a second to ten seconds. Once the algorithm reaches either base case, the board state is evaluated, and the value that is returned is evaluated by the parent node to determine if the move is superior, equal , or inferior to other moves the parent node has made. If the move is superior, the parent node's list of children is cleared and the move is added to the list. If the move is just as good as the other moves, the move is added to the list of children. Otherwise, the move is ignored and not added to the list, to be collected by the garbage collector at a later date. Additionally, this is when alpha or beta is updated, depending on which player's turn it is. The maximizing player would update alpha, and the minimizing player would update beta. In either case, the updated value is then used to compare if beta is less than or equal to alpha. If it is, exploration is stopped and the node is immediately returned. The algorithm then undoes the move it just made, and executes another move from its list of possible moves. The function will return the current node once the list of children is empty. Once the algorithm finishes, the AI chooses a move from the root node's list of children. The list is checked to determine if a move will immediately move the king into a corner or take a piece. Any moves that satisfies either condition is returned to the player as the best move to make. If no move satisfies either condition, a move is selected at random from the list of children, since all the moves are equally good.

4 Implementation

4.1 Task Allocation

James: Game Logic

- Has the most experience with android development and wants to learn other parts of Java
- Due to the android experience, knows how to structure the code and create the design

Jacob: Artificial Intelligence

- Has taken CS344, so has experience with many different algorithms and data structures
- Is interested in learning how an AI is created

David: Graphical User Interface

- Wanted to increase knowledge on programming a GUI
- Interested in what makes a good user interface

Benjamin: Networking, Documentation, and Report management

- Is interested in helping out with other tasks that were remaining.

4.2 Tools

Since we planned on creating an Android app, Android Studio was immediately decided to be our development environment, since Google reccomends it as the best IDE to use when creating Android applications. Android Studio also allows everyone to have the ability to immediately plug in an Android device, deploy the game, and test the application.

We also decided to use Git as our version control system. Originally we were using Clarkson's Gitlab server, which allowed for a clean interface to see the history and progress of individuals, but due to it being connected to Clarkson's VPN, we moved the repository off of Gitlab and onto Bitbucket. This allowed the team to host the repository on more reliable servers and to conveniently access it off-campus.

4.3 Code Layout

4.3.1 GUI

The GUI was drastically modified several times. The first technique that was tried was using image buttons. The reasoning was that a move would move the piece from one button to another by setting the image of the image button. This was quickly thrown out the window and an image view was then used for the board. At first the board was going to be a 13 by 13 board, meaning that there were 169 spots on the board. The problem with this was that the space available to click on one spot on the board was very small. To combat this we decided that a 11 by 11 layout would be a better. This necessitated changing constants in the game logic code but was seen as a good idea by all members of the group. This change was also valid by the rules of Hnefatafl, because while 13 by 13 is the more common layout, there are layouts that are 11 by 11 that are still valid versions of the game.

Now that we have settled upon a 11 by 11 board we turn our attention to the pieces themselves. There are, in total, 37 different pieces: 24 black pieces, 12 white pieces, and one King piece. The beginning layout of pieces are in an easy to remember order. Besides the King piece, which starts in the middle and is called kingpiece, all of the pieces are named with their color first followed by their number, for example black1. The counting starts with the first square to have a piece, in the upper left corner. The numbering then increases as you go across the board. When you have finished counting all of the pieces in one row, you move down one row, go all the way to the left and continue to increment the number according to the color. This ends up with a configuration that looks like this. The numbers in the middle are all white and the border numbers are the black pieces.

```
        01 02 03 04 05
              06

07              01              08
09           02 03 04           10
11 12     05 06 Kp 07 08     13 14
15           09 10 11           16
17              12              18

                19
        20 21 22 23 24
```

When you want to move a piece you have to first click on it to select it. You are then allowed to select anywhere on the board to move that piece. Wherever you touch, if it is a valid place to move the piece, will cause the piece to move. This is accomplished by having an on touch command on every piece, which are all essentially imageviews. When a valid movement place is selected, the image view is then moved to that position on the board. This was implemented using the Publisher Subscriber Pattern, where each piece was a subscriber with its own listener. The GUI has four classes that control the game. The first class is *MainMenu*. This gives the player three options. They can start a local game between two people. play against the AI, or watch a demo of the game being played by two computer players. If the player chooses to play against the AI, there is a splash screen that is controlled by *PlayerChoice*. All this class does is set the player to use white or black pieces, according to their choice. *GameBoard* is the class that controls the game board. It contains everything that is needed to play the game. It also has an exit button in case the player wants to exit the game before it is finished. The final class is *MoveAttempt*. This class was going to store moves when they happened, but we ended up not using it at this time. At a later point in development it will be used.

4.3.2 Game Logic

The game logic is split into many parts similar to the way the real world functions. There is the board and its pieces. The player, who is in a game, makes moves following some set of rules. This layout is what the implementation tried to follow.

First, there is a *Token* class. This holds the information of each board piece, such as its color and position. There is also a *Board* class. This holds the board state, including what tokens are on the board and their positions, and has the ability change these things. Next is the *Move* class, which holds the information about a move (old position, new position, and the deleted tokens). There is also a *TokenMovement* class. This describes all of the legal moves that a token can make. It records the previous moves that were made and deletes the tokens that are captured. It also has the responsibility of doing an undo action. It essentially holds all of the game rules.

The next layer of the design is the players. There is a *Player* class, which establishes the basics of a player (things that all players should have). There are then two types of players, the *HumanPlayer* and the *ComputerPlayer*. The human player is used for human based interaction within the game. The computer player is used for the artificial intelligence to interact with the game. There is also the *PlayerInformation* class, which holds all of the information about a player. This is implemented using the Strategy Pattern. It does not have any subclasses because of the way the code was developed, but it could have other information for different types of players. It was then composed within the *Player* class and established within each type of player. There is also a *PlayerMovement* class, which can be used in the future if the project gets larger or changes. It would implemented similar to the *PlayerInformation* class and would be used if different players or objects could have different allowable moves. such as the knight and queen in chess.

At the highest layer of the design is the game. The players are a part of the game. There is first a *Game* base class. There is then the three game types that extend it: *DemoGame*, *LocalGame*, and *SinglePlayerGame*. The different games contain their own networks and their own players. The demo game contains two computer players that fight each other over a local network. The local game has two human players that fight each other over a local connection. Finally, the single player game has a human player and a computer player that both fight over a local connection. This structure of games, players, boards, and tokens allows for a complete architecture that is easy to understand and can be expanded upon easily.

4.3.3 AI

The classes used for the AI can be separated into three categories. First, there are classes which are used to hold data. Second, there are classes used to manipulate data. Lastly, there is the *ComputerPlayer*, which executes the move suggested by the AI.

The *PositionPair*, *MovementData*, *Node*, and *NodeData* classes are used to store data. To pass around x,y coordinates as a pair, instead of as two separate integers, the *PositionPair* class is used. To store data about where a particular *Token* will move to, a *Token* and a corresponding *PositionPair* are stored in the *MovementData* class. The *Node* class uses a generic type parameter to hold any object as data, and maintains a list of children. The *NodeData* class stores information about which move was made and the value of the move as evaluated by the AI's algorithm, using a *MovementData* object and a *double*.

Classes used to manipulate data include the *ComputerPlayerOptions* and *SimpleAI* classes. The *ComputerPlayerOptions* class uses information from a *Board* object and a *TokenMovement* object to produce a list of all valid moves a particular player can make, storing the information using *MovementData* objects. The *SimpleAI* class ties everything together and runs the Minimax Search with Alpha-Beta Pruning algorithm. It implements the *ArtificialIntelligence* interface, utilizes the *ComputerPlayerOptions* class through object composition, and passes around data using the *PositionPair*, *MovementData*, *Node*, and *NodeData* classes. The *ArtificialIntelligence* interface was created to allow for easy extensibility in case a different algorithm is used for the AI.

Finally, an *ArtificialIntelligence* object is used by the *ComputerPlayer* class to determine what its next move should be.

4.3.4 Networking

The networking hierarchy was based around the Strategy design pattern. Depenging on the game mode, the way moves were sent over a network or locally would all look the same to the Game Logic, allowing the back-end of each mode to implement their own way of passing data. We were able to implement this for a local game on a single device between either players or computers, but we did not have time to implement a multi-device game. The classes consist of a *GameConnection* and a *ChatConnection* class. These hold abstracted functions. There are also the *LocalGameConnection*, *LocalChatConnection*, *SinglePlayerGameConnection*,

and *SinglePlayerChatConneciton* classes. The game connections link the two games and send the moves. The chat connection links a chat dialog between the two players. Lastly, the difference between a local connection and a single player connection is where the data is sent. A local connection trades information on the same device, while the single player connection trades information over a network. The chat connections and the single player connections were not implemented for this project, but could be implemented in a future version. There is also a *NetworkManager* class, which could be implemented to make the system dynamic between the connection types (following the Open Closed Principle). This was not needed for this version due to only one connection type being implemented.

4.4 Testing

The game was incredibly difficult to debug. This is due to the fact that the game logic needed an outside input in order to test. The game's artificial intelligence also needed to the game logic to be working to run at all. Thus, the game logic and artificial intelligence were intertwined.

The testing was done with the Android Studio debugging tool. More specifically, due to the human interface of the program having errors, the game logic and artificial intelligence were tested and debugged at the same time. Two computer players were put into a demo game where they battled each other. There were log statements that occurred when certain actions, such as a move, took place. This allowed the debuggers to watch the game state as it changed. There were also log statements within if statements to inform the debuggers if an error occurred. The combination of all the log statements allowed for a quick assessment of what error occurred and what led to this error.

Later in the process, when the application was having more niche errors, a text version of the board was displayed. This was used because it was so simple that there was very little risk of having an error in the display. It allowed the overall board state to be more easily seen and analyzed.

5 Discussion

One of the major bugs we spent several hours debugging was causing odd behavior by the game logic, such as disappearing pieces and moves being disregarded as invalid when they should be valid. The bug arose when we had to change the x,y positions of tokens to use an 11x11 instead of a 13x13 board, but the person who modified the array positions forgot to modify the coordinates the tokens were initialized with. This had a relatively simple fix, as a few numbers in the token initializations needed to be changed, but this took hours to find, because the initialization of the board array and tokens is done in an object that is separated from the game logic.

There were also other difficult portions of the game logic, such as the separation of tasks and the linking of classes. The separation of tasks as described in the Game Logic section had some difficulties during implementation. This was due to many indecision on which class had what responsibility. There were also more classes that we created during the process, which made the delegation of responsibilities more complex. After a lot of talking and drawing small diagrams, the responsibilities became more clear.

A few problems arose when writing and testing the AI. First, the tree used to store all the information on what moves are being made grows big in a short amount of time. This problem was subverted by clearing a node's list of children when the children stored information on suboptimal moves. This solution not only cut down on storage space, but also made deciding on which move to pick easier, as the list of children did not need to be searched to find the children with information on the optimal moves. Another problem that arose when testing the AI was that many moves have the same evaluation value, due to the evaluation function we chose. This meant that more than one move was being returned as the best move to make. We solved this by randomly picking a move from the list of children. However, this caused a couple other problems. A move that would kill a piece in 2 moves has the same value to the player as a move that kills a piece in 1 move, since in either case one token is captured. Similarly, moving the king into a corner after 2 moves has the same value on moving the king into a corner after 1 move. The first problem was solved by checking the list of returned moves, and picking any move that captured a token on the first turn, if there was such a move. The second problem was fixed in a similar manner, by picking a move that would move the king into a corner on the first turn, if such a move existed. The last major problem that arose with the AI was that moving the king into a corner to win was weighted less than capturing a token. This was solved by returning

a big negative number for a move that moves a king into the corner, indicating that the move is really good for white and really bad for black. A very minor, but serious, bug with the AI was that the black captures and white captures were being weighted wrong. Black capturing a white piece was considered bad for black, and white capturing a black piece was considered bad for white. The fix was as simple as switching a couple negative signs around, but the bug was only caught after the game logic was fixed and we were watching games run via print line statements, perplexed at how the AI was never capturing any pieces.

The most puzzling part of the GUI was that the movement of pieces was taking a very long time to happen for no apparent reason. This was discovered when one member of the team was working on a virtual machine. Each piece was using an image that was 960 by 960 pixels large for a game piece that was approximately 40 by 40 pixels large. When the image for the pieces was scaled down to 400 by 400, the animation became snappy and responsive, making the game much more playable than before.

This was the first time one of our members used git. He didn't even know what it was before he started using it, but when he did he liked it immediately. It solved the problem of "who has the most updated code" before it became a problem.

6 References

[1]Hingston, Phillip. (2007). Evolving Players for an Ancient Game: Hnefatafl. Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on. IEEE, 2007.