Overview:

In this practical we had to make a backgammon game in Java. We also had to develop networking and AI for the game. We extended the initial specification by adding the full rule set for the game, as well as implementing a GUI which easily allows the user to play versus AI, or through the network. We implemented a set of three different AI, that use different methods to evaluate their next move.

Design:

Main game

We first started to approach making the game by thinking what classes are needed to be represented. From our previous experience working on Fox and Geese in CS1002, we decided we should make a Move class and use that to represent moves rather than passing ints around the board and pieces. For the board, we needed more than just a board as there were the 24 columns as well as the bar and the end columns. This is why we also made a class for the columns and the board simply contains an array of columns. The board is also the class that contains the dice. We thought about have the dice be it's own class as well, but it doesn't contain much information and we need to set the dice from the board. We used two int arrays to represent the dice, one for the normal dice rolls and another if we rolled doubles. A value of 0 on the dice would mean it cannot be used. We initially set the colour of the pieces on the board to be represented by a boolean (for example black is true, white is false) but when it came to calculations and gameplay elements, we found it was easier to have it be an int so we can do maths on the pieces. Now we have the colour be represented by 1 and -1. This makes it easy to tell what is white and black since one would be positive and the other negative and aids us in calculations like moving the pieces across the board.

Many of the gameplay checks are done in the Column class because we want to be able to call rules on specific columns to test if those moves are valid. This way we can select the column we want to make a move from and test that move. Most of the methods for checking valid moves are booleans with various conditions. When we move a piece, we just remove a piece from the selectedColumn and add a piece to the column that you want to move to. We use a select method and a selectedColumn in the Board to make our moves. We did this instead of just inputting two numbers and those being the column numbers because it could be used in the future for the GUI and the AI. The GUI will need this because it doesn't take in text input and it also helps the AI as it can iterate over the selectedColumn to check for moves from that column.

From the beginning, we were not sure how to deal with pieces captured by the opponent. The lecture notes talked about it being on the 0^{th} or 25^{th} column but we were not sure, so we added the bar (which we call the wood column) where the pieces go to when they are captured. We read up on the full rules of the game and implemented those since we did the bar already anyway. The full rules that were different from the simplified rules which we added were rules on must playing pieces in the bar, and advanced bearing off. Instead of needing to roll the exact number of that column to bear off, if there are no pieces on the 6^{th} column and a 6 is rolled, then we have to bear off a piece on the next highest numbered column (eg, the 5^{th} column if there are pieces there).

AI

Homura

Homura is a brute force AI. We programmed her initially to just make random moves and record all the moves made in the game. If she wins all the moves she made in the game get a +1 to their value and are stored in a serialised data file. We chose to serialize the Timeline class, as it provided an easy interface to read and write from and to. One concern is that our data structure is relatively large, since we work with int arrays. This means that the Timeline file grows large very quickly, and becomes slow to work with.
The next time she plays, she will take moves with high value if possible and if not generate new random moves. This way she will learn after millions of games which moves are statistically better regardless of board position. The moves are stored as a Java object called TimelineMoves which holds the int “to” and “from” as the column numbers, “wins” as number of games won with that move played.

We realised that just keeping track of turn number doesn't help her improve much because there could be so many different board positions and the turn doesn't accurately take into accoutn the board position. So our first improvement to her was instead of saving turn number to save an integer array that represents the board. The pieces are multiplied by the colour of the column, which in our design is represented by 1 or -1. So for example -5 would mean 5 black pieces and 3 would mean 3 white pieces. This would make her better because she would have a better idea of whether a move is good as she checks the board state to see if they are identical. This idea, although great in theory was hard to put into practice as it became very rare for her to ever use a move she has learned because of so many different and possible board states the game could be in. She is only likely to use known moves during the start or end of the game because at those points the board state is quite similar. Otherwise she is almost always just learning new moves. We think that given enough time and a large enough set of data, she could eventually learn and be good at backgammon. It would take a lot of hard drive space to store all the data and time to play thousands or millions of games, so we are unable to make her good at the game at this stage. She serves as more of a proof of concept of a brute force AI that learns all possible board states and moves and draws from data whether those moves are good or not. As such we have not included her in testing the AI against each other, since she is basically a random AI who saves every move made in the game.

```
Game [Java Application] /usr/local/jdk1.8.0_31/bin/java (27 Feb 2015 10:53:11)
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
Homura chan learned a new move
using known move
using known move
Homura chan learned a new move
Homura chan learned a new move
using known move
using known move
Homura chan learned a new move
Homura chan learned a new move
using known move
using known move
Homura chan learned a new move
White wins
```

Aoi

This is our AI which evaluates the value of the board before and after she makes her move. The

values had to be manually put in based on our own knowledge of the game and so it not a very accurate representation of what is supposed to be a good move.

```
public class Aoi implements AI{

    public static final int SINGLE_SAME_PIECE_VALUE = -10;
    public static final int SINGLE_DIFF_PIECE_VALUE = -100;
    public static final int PAIR_VALUE = 25;
    public static final int GREATER_THAN_PAIR_VALUE = 10;
    public static final int END_VALUE = 75;
    public static final int WOOD_VALUE = -50;
```

For example we gave the value for pairs to be very high because being in a pair is safe and means our checkers can't be hit off by the opponent. By the same logic, a piece which is by itself has a negative value. Every time Aoi evaluates the board to make a move, she will choose the possible board state that returns the highest value. So she would try to create pairs or get rid of singular pieces on the board. Opponent pieces that are by themselves have a very large negative value, which means Aoi will prioritise trying to capture opponent pieces in order to improve the board value.

Miki

This AI is meant to be an improved version of Aoi. Instead of just evaluting the next board state and using the one which returns a high value. Miki makes a tree of boards and goes down the trees finding the average returned values of all the trees.

Networking

To implement networking, we extended the sample code given to us, with some modifications. We created classes for both the client and the server, which allows an instance of the game to run as both, with minimal additional configuration. To synchronize between client and server, we implemented logic in our Game class, which checks for a data packet from the socket connection every few milliseconds.
When a new string is pushed in the input stream, we check whether it is a special case (like “you-win; bye” as instructed from the protocol), and then processes the turn, using regular expressions to make the text easier to work with. We then check and execute each move, and passing the turn if the move received is (-1,-1).
On the server side, we wait for players to connect, while at the client side, the server must already be open for the connection to be established. If the connection fails, the game defaults to a local game.

GUI

We developed the GUI to be as extensible as possible. Our GUI implements a grid bag layout, since we believe this is the most appropriate layout to model a game like backgammon, where most objects are stuck together. We separated each graphical element into its own class, extending Jlabel, which allowed us to produce a very intricate GUI.

We used a big spritesheet to contain all the sprites in our game, including tiles backgrounds such as the wood panels and the background of the board. We implemented this as we believed it is more efficient at drawing, since all sprites can be loaded with only one file stored in memory. In addition, this allowed us to easily edit the style of our game, and can allow for external modifications, simply by changing which spritesheet is used.

We used layers to draw the GUI by drawing our components in sequence. For example, the board

and wood are drawn first, then the columns, and finally the individual pieces. To improve performance, we took care not to overload the AWT thread with drawing requests. We implemented Mouse listeners and mouse motion listeners to repaint the window at every mouse interaction. In addition, we repaint the window at each game loop and at each move execution. This allowed us to have a very responsive GUI while still preserving a high frame rate.

Testing

AI

Our first test to see if our AI is good at the game is to play it against the random AI. In our initial test, Aoi was winning more against random AI, around 65-35. But this wasn't a good result as she should be winning almost 100% of the time, or at least 90-10 in 100 games. We concluded that it was because we had yet to implement the full set of rules. At this point we didn't add rules for having to play a checker off the bar and not being able to play any other piece should there be a piece on the bar and also the advanced bearing off rules. After adding those rules, Aoi performed very consistently against the random AI, having a very high win rate (99-1) all the time. This means she does perform the way we want her to.

```
<terminated> Game [Java Application] /usr/local/jdk1.8.0_31/bin/java (27 Feb 2015 10:44:26)
White wins
starting game...
White wins
starting game...
White wins
starting game...
White wins
Black won: 7
White won: 993
```

```
<terminated> Game [Java Application] /usr/local/jdk1.8.0_31/bin/java (27 Feb 2015 10:44:02)
White wins
starting game...
White wins
starting game...
White wins
starting game...
White wins
Black won: 1
White won: 99
```

The disadvantage to her evaluation system is that she uses values that we manually decide, so she ability is only as good as our skill in backgammon, which is not very high. For example we decided that having single pieces on the board was bad as it could be captured and capturing opponent pieces were a high priority. However, there is more strategy than just saying single pieces are a bad board position because they are necessary to advance the board as well. Also there could be cases where a single piece cannot be captured by an opponent as there is no possible dice roll they can have to move their pieces onto ours. This kind of higher evaluation would make Aoi better than what she is currently. She can easily beat the random AI, but may have more difficultly against a skilled player.

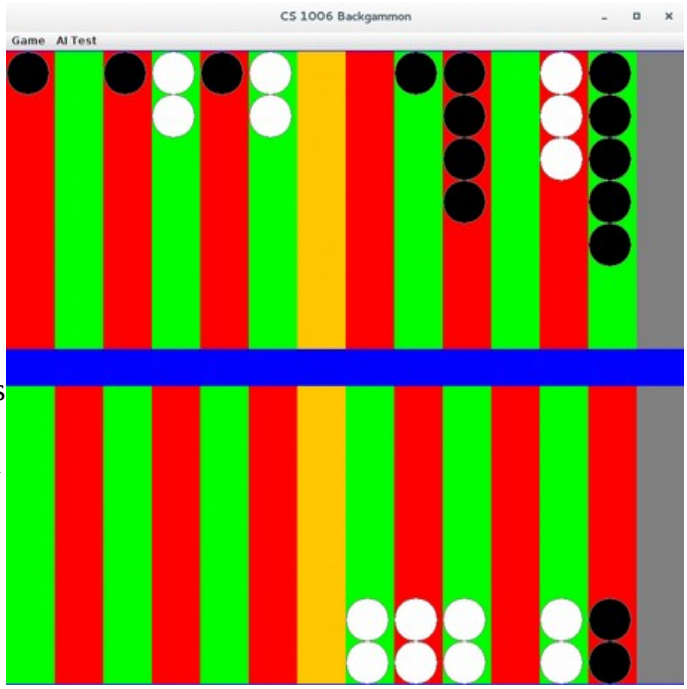
```
<terminated> Game [Java Application] /usr/local/jdk1.8.0_31/bin/java (27 Feb 2015 10:43:40)
White wins
starting game...
White wins
starting game...
White wins
starting game...
White wins
Black won: 1
White won: 99
```

Miki

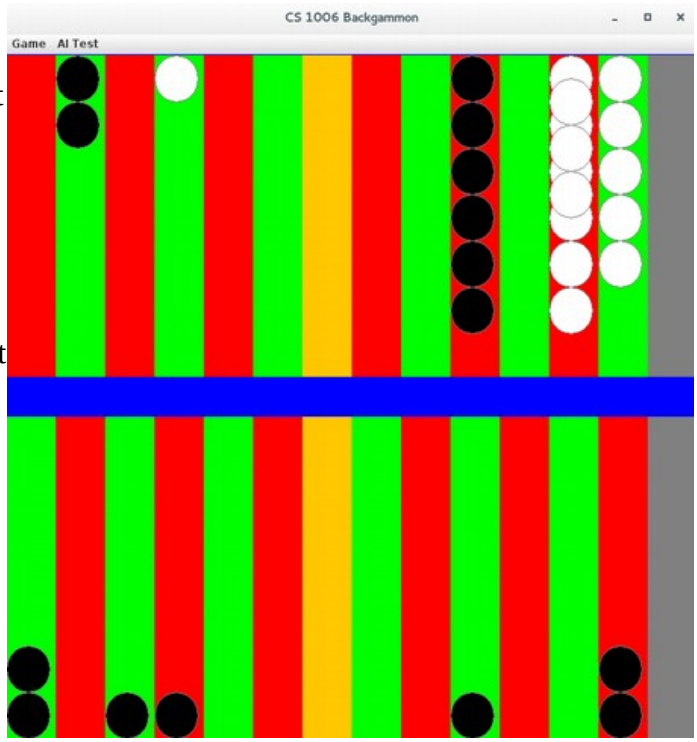
Since Miki uses a tree data structure to evalute her move, this made it difficult to visually identify how exactly to improve her. We used a similar cost function to Aoi to evalute boards in a min-max way, but this proved more complicated for future boards. We tried to use various ways of displaying her data, but ultimately this proved difficult to understand. For some reason, Aoi consistently outperformed Miki in all tests, possibly because of our evaluation algorithm.

```
Game [Java Application] /usr/local/jdk1.8.0_31/bin/java (1 Mar 2015 15:57:46)
        Evaluating children of ai.miki.FutureBoard@449b2d27
from 3804 boards
        Evaluating children of ai.miki.FutureBoard@449b2d27
from 3804 boards
        Evaluating children of ai.miki.FutureBoard@5479e3f
from 5136 boards
        Evaluating children of ai.miki.FutureBoard@27082746
from 3804 boards
        Evaluating children of ai.miki.FutureBoard@27082746
from 3804 boards
        Evaluating children of ai.miki.FutureBoard@66133adc
from 2472 boards
        Evaluating children of ai.miki.FutureBoard@7bfcd12c
from 3804 boards
        Evaluating children of ai.miki.FutureBoard@42f30e0a
from 3804 boards
        Evaluating children of ai.miki.FutureBoard@24273305
from 3804 boards
        Evaluating children of ai.miki.FutureBoard@24273305
from 3804 boards
        Evaluating children of ai.miki.FutureBoard@5b1d2887
from 5136 boards
        Evaluating children of ai.miki.FutureBoard@46f5f779
from 2472 boards
        Evaluating children of ai.miki.FutureBoard@46f5f779
from 2472 boards
        Evaluating children of ai.miki.FutureBoard@1c2c22f3
from 3804 boards
        Evaluating children of ai.miki.FutureBoard@18e8568
from 5136 boards
        Evaluating children of ai.miki.FutureBoard@18e8568
from 5136 boards
        Evaluating children of ai.miki.FutureBoard@33e5ccce
from 3033 boards
```

Aoi (White) vs Miki (Black)



We played Aoi vs Miki many times and most of the time Aoi was able to beat Miki. The problem we had with Miki was that when her pieces were captured, after playing them back onto the board, she would move the pieces on the other quadrants first, leaving a large stack of pieces where she got them off the stack. She would then only move them one by one towards the home board, which makes her slower compared to Aoi.
We can also see that Aoi will always stack her pieces up to two where possible. Our static values for stacks greater than two are much smaller than the value with stacks of two. Even though they should both prioritise capturing opponent pieces, Miki moves her pieces more slowly and without protecting them the way Aoi does. This leads to Aoi gaining an advantage and as we can see in the second screenshot of the game, Aoi already has most of her pieces in the home board when Miki still has a large stack at the beginning. We think it may be something wrong with her logic in evaluating the board state as it doesn't make sense that the AI that doesn't look ahead performs better than the AI that checks a few turns ahead before making a move. It may be that because backgammon is a game of chance with dice, looking into the future doesn't provide as meaningful a result.



Evaluation:

One possible weakness of our implementation is the AI. Although we tried, we couldn't successfully make Miki consistently beat Aoi, which is supposed to be a simpler AI. In addition, although our network implementation allows us to play using our client, we haven't tested our game against other AI and human players. There might be conflicts between our rule set and other people's, especially since we use additional features, such as the wood bars and the end columns. Even though our program strictly follows the protocol, we haven't made provisions in case other players try to send unsupported messages.

Conclusion:

In conclusion, we believe we believe we have completely satisfied the initial specification, and added a comprehensive rule set of the game. This allowed us to produce sophisticated AI that perform decently well against players, while completely destroying AI that are based on random chance.