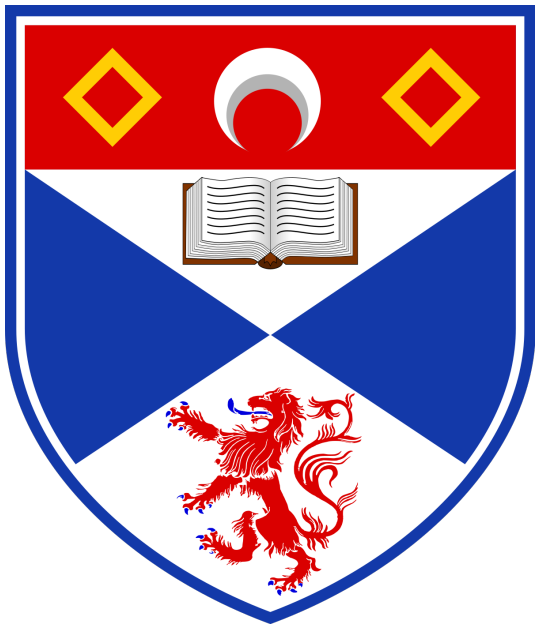
Combinatorial Games: Amazons



University of St Andrews

April 12th 2021

Connor Macfarlane

160002090

Supervisors: Stephen Linton &
Alexander Konovalov

Abstract

This document describes my research into combinatorial game theory, and my implementation of a program that allows users to play Amazons against an AI that utilises a strategy based on the underlying mathematics. It also explains my other Amazons strategy implementations, as well as my experiments designed to compare the effectiveness of each.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 10,630 words long, including project specification and plan.

Contents

1 Introduction

Amazons (or Game of the Amazons) is a combinatorial game that was invented in 1988 by Walter Zamkauskas. It is played on a 10 by 10 board and combines elements of chess (queen moves specifically) with the idea of removing squares from play, until one of the two players is unable to move and must concede.

My intention with this project was to develop a program to allow users to play the game of Amazons, as well as develop different AIs that will play using a variety of strategies. The AI strategies employed include a combinatorial game theory implementation, with an endgame database optimisation, a Monte-Carlo tree search, a heuristic-based implementation, as well as a random choice AI.

The main focus of the project was with the combinatorial game theory strategy, specifically evaluating the game board into the recursive game value structure detailed in "Winning Ways" and "On Numbers and Games" by J.H Conway. The next stage was then to optimise how these game values are produced, first by storing them in a file and then a database, as well as optimising how they are stored by taking advantage of the different symmetries of a board.

After implementing these different strategies, I then wanted to test out their effectiveness via a series of simulation games. This allowed for optimising the Monte-Carlo strategy, while also comparing how each of the strategies performed against each other. Ultimately, I wanted to find out what mix of the implemented strategies was most effective, and compare what I found with other papers on the topic.

Aside from the main aim of this project of applying combiantorial game theory to Amazons, I also wanted to add features to the Amazons program that allowed the user to learn about the game, as well as improve. The tutorial and review game options provide this, by first teaching users how to play the game, as well as review their losses and attempt to learn from their mistakes.

This report explains five main components of the program in sections 5-9, detailing any interesting design and implementation decisions for each, along with how each component was tested.

1.1 Project Objectives

The objectives for this project were outlined during the planning stage and specified in the DOER document. As the project evolved, the focus has been shifted from a couple of objectives to alternative requirements, which will be discussed in more detail in the evaluation section.

Primary Objectives

1. Research combinatorial game theory, specifically the game 'Amazons'
2. Implement a framework to allow the playing of Amazons
3. Develop a program that plays Amazons using strategies that are based on the underlying mathematics
4. Develop other program implementations that play amazons
5. Run experiments to test my implementations effectiveness

Secondary Objectives

1. Develop a nice GUI to show the playing of the game
2. Basic teaching of the rules of the game (like a tutorial)
3. Teaching of the maths behind decisions for the user to see, to develop their skills in-game

Tertiary Objectives

1. Research and develop programs for similar games or variants of Amazons

2 Context Survey

2.1 Combinatorial Games History

Combinatorial game theory is the study of strategies and the mathematics of non-random games of perfect knowledge. To put this into simpler terms, combinatorial games include any games not involving chance (e.g. monopoly) or hidden information (e.g. poker). While the study of combinatorial games dates back centuries with ancient games such as chess, the first attempt at fully analysing a non-trivial combinatorial game only occurred in the early 1900s.

The analysis of Nim by C.L. Bouton in 1902 allowed for a consistent theory of impartial games (a subset of combinatorial games where both players are allowed all the same moves) to be developed in the 1930's. After this, J.H. Conway's theory for partizan (non-impartial) games allowed for many more games to be studied in this framework, including chess, checkers and go (Nowakowski, 1998).

2.2 Amazons Origins

"El Juego de las Amazonas" is a combinatorial game that was invented in 1988 by Walter Zamkauskas of Argentina, which roughly translates to "The Amazon Game", or its abbreviation, Amazons. It was originally created for a puzzle magazine "El Acertijo", however after its introduction to a postal gaming club in 1993, its playerbase grew quickly. The first international match was held between Argentina and the United States between 1994 and 1995, demonstrating that the popularity of Amazons spans continents (Pegg, 1999).

2.3 Combinatorial Game Research

The study of games such as Amazons and chess has been formalised and generalised by J.H. Conway in "On Number and Games", and reiterated in "Winning Ways for Mathematical Plays" by Berlekamp, Conway and Guy. The notation outlined in both books is now the standard, recursively defining a game as 2 sets of games, labelled left and right. The left set of games illustrates all the possible games that can result from a Black move, also known

as the left options. Likewise, the right set indicates all the possible games resulting from a White move (Berlekamp, 2000).

$$G = \{G^L | G^R\}$$

Some games can be expressed as numbers, for example a game in which neither player can move is given the value zero. Similarly, games in which one player can move but their opponent cannot are also given values. In "Winning Ways", positive values indicate a game that leaves Left (Black) with that number of moves, and negative values similarly indicate a game that leaves Right (White) with that number of moves. In some papers this system is inverted, with white being assigned the positive values, however for this project I will be following the "Winning Ways" notation.

$$0 = \{|\},\ \{0|\} = 1,\ \{|0\} = -1$$

Next, in order to simplify the lists of possible moves for both Left and Right down to only the "best" options, we must be able to compare games. The less than or equal to definition provides this functionality, allowing for the removal of dominated games.(Conway, 1976).

$$x \leq y \text{ unless some } y \leq x^L \text{ or some } y^R \leq x$$

For a given game, we can also say who has a winning strategy, either Left, Right, the first player to move or the second player to move. Using the less than or equal to formula, and a zero game, we can work out which category a game falls into. Second player wins occur in a zero position, as neither player wants to move. First player wins are the result of a "fuzzy" position, meaning we can't specify which player has a clear winning strategy. (Berlekamp et al., 1982).

$x = 0$	Second player win
$x < 0$	Right win
$x > 0$	Left win
$x \mid\mid 0$	First player win

Another interesting concept is that of "hot" and "cold" games. Games in which neither player wants to move are cold, whereas games in which either player has a lot to gain from making a move, are known as hot. When playing several games simultaneously (e.g. when the board is split into partitions in Amazons), playing in the hottest game is key to making the best moves (Berlekamp et al., 1982).

$$\{1|-1\} = \pm 1 \text{ is a hot game, } \{5|-5\} = \pm 5 \text{ is a hotter game}$$

It is also possible to find the sum of two games, using the addition formula. This is fairly logical, taking the left side as an example, we either move in game x and have to add this to game y, giving us $x^L + y$, or we move in game y and and have to add this to game x, resulting in the game $x + y^L$. This allows us to find the game value for a sum of games, or to link it back to Amazons, a sum of partitions on the board (Conway, 1976).

$$x + y = \{x^L + y, x + y^L | x^R + y, x + y^R\}$$

2.4 Amazons Strategies

The main difficulty in implementing an Amazons strategy program is the complexity involved, due to the large branching factor present in the early and middle-game stages. The large board size, coupled with the "move and shoot" moving pattern leads to roughly 4 million possible board states after just two moves, making a full tree-search strategy initially impossible.

A variety of strategies have already been employed in different amazons programs, including minimum-distance heuristic evaluation with "Arrow" (Müller and Tegos, 2002) and "Amazong" (Lieberum, 2005), or the Monte-Carlo Tree Search approach described by (Kloetzer et al., 2007).

The heuristic-based approaches tend to be the fastest in the early and middle game, as they use simple arithmetic operations to assign values to each of the boards unburnt squares. Almost all the heuristic programs agree that using min-distance is the optimum strategy, which involves assigning values to squares that indicate how many moves are required to reach that square, from any of each players amazon pieces. Using this information, we know

which player has better access to certain squares, and can therefore shoot the squares which are more accessible to the opponent, limiting their possible moves (Lieberum, 2005).

The Monte-Carlo Tree Search strategy is limited by the large branching factor, and so it must utilise an effective "selection" process to ensure that it spends the majority of its allocated time on promising branches of the tree. To allow the algorithm to pick these promising branches, an upper-confidence bound formula is used, which assigns a score to each node in the tree, based on its average win score and the number of times the node has been visited (Kloetzer et al., 2007).

3 Requirements Specification

The requirements of the project allow for a more detailed description of each of the objectives outlined in the DOER, and include a set of additional requirements that were added during the development process. Each of these requirements for the project have been achieved.

3.1 Primary Requirements

Requirement 1

The first requirement for this project is to research combinatorial game theory, with the aim of using this theory to implement a strategy to play the game Amazons. This will involve reading sections from various books on the topic, including "Winning Ways" and "On numbers and games".

Requirement 2

Implementing a framework to allow the playing of amazons, by both human and AI players. This should allow for 2 users to play each other, a user to play an AI, and 2 AI to face each other.

Requirement 3

Develop an AI implementation that utilises the combinatorial games theory knowledge from requirement 1 to play Amazons in conjunction with the framework program from requirement 2.

Requirement 4

Develop 3 alternative AI implementations to play Amazons, specifically Monte-Carlo Tree Search, a heuristic approach, and a random choice implementation.

Requirement 5

Update the framework to allow for large numbers of simulation Amazon games to be run between types of AI, before running experiments to test the different strategies effectiveness.

3.2 Secondary Requirements

Requirement 6

Implement a tutorial section for the Amazons program, allowing the user to learn both about the rules of Amazons, and how to play moves via the console.

Requirement 7

Implement review games functionality for the Amazons program, allowing the user to go back to game played in the previous program execution and jump between each of the moves.

Requirement 8

Add additional functionality to review games that allows the user to see how a board is evaluated with the combinatorial games strategy, by showing game values for each of the partitions, as well as the game value returned for the whole board.

Requirement 9

Optimise the combinatorial game theory board evaluation with an endgame database, allowing for a more realistic chance of evaluating the partitions to return a move.

Requirement 10

Test different values of exploration parameter for the Monte-Carlo strategy with simulation games, to find the optimal value to use for Amazons.

Requirement 11

Implement some form of partition simulation functionality, allowing the user to specify a partition and simulate it, or generate a random partition to simulate.

Requirement 12

Implement automated unit testing to ensure that the board, game value, endgame database and all AI implementations work correctly.

4 Software Engineering Process

4.1 Development Methodology

The overall objectives of this project were decided upon early, however it was hard to plan out all the smaller tasks that had to be completed, due to the scope of the task. For this reason, an iterative development strategy seemed appropriate, and so an Agile approach was taken. This was enforced with weekly supervisor meetings, allowing for week-long sprints each with small requirements and solutions discussed and then implemented the following week.

This approach worked well, allowing a clear goal for each week, and keeping constant progress through the project timeframe. The weekly supervisor meetings were very helpful, allowing any issues to be discussed and ultimately resolved very quickly, as well as setting a target date for each task.

I also wrote a weekly supervisor meeting preparation document, which I gave to the supervisor at the start of each meeting. This outlined the progress I had made since the previous meeting and contained screenshots of any interesting parts of the development. I would also note any issues or questions I had for the supervisor, as well as any ongoing tasks that were still to be completed, helping us both keep track. I found this strategy to be optimal, as otherwise it was likely that I would forgot to ask an important question, or discuss an important decision to be made.

4.2 Tool Usage

GitHub was used for version control. This allowed for the Agile development strategy, with the iterative improvements being visible to the supervisors as soon as they were completed. The website UI is straightforward, clearly showing the code changes with each update, and keeping a backlog of updates in case of an issue further down the development tree. The access control provided by using GitHub was also helpful, as I was able to show my progress to anyone simply by adding a collaborator via the website.

Java was the programming language used for developing the framework to play Amazons, as well as the different strategies to play the game. The structure of a board game lends itself well to an object-oriented model, with each of

the different aspects of the game each being easily described with a class (e.g. players, pieces, moves, the board). It also provides the required functionality to perform all the tasks of this project (e.g. console I/O, file I/O, unit testing libraries).

Another benefit of OOP is that breaking down the large project into its smaller component parts also makes the tasks easier to understand, as well as helping with the iterative development cycle strategy.

JUnit was used for unit testing. This allowed me to generate automated tests, meaning we can check that implemented solutions work as intended instantly, with each incremental change.

JavaDoc comments are written for every class and every method, to make the code easy to follow, and to allow API documentation for the entire project to be generated in HTML form.

H2 database engine was the database management system used, allowing for the endgame database to be stored on disk. Its a very fast, open-source JDBC (Java Database Connectivity) API. The advantages of using h2 over other SQL Database options are its speed and small footprint (it only requires a 2 MB JAR file to use). This Database solution drastically speeds up the time taken to fetch and receive the HashMap of partitions, while improving reliability and scalability. It allows for databases to be stored either in memory, or on disk, of which the latter is ideal for this scenario.

JDBC is the API used to connect to, query and update the database, as it's the fundamental specification for Java database connectivity.

SQL statements were used to query and update the database.

IntelliJ was the Integrated Development Environment (IDE) used throughout the development, testing and debugging of the project. The breakpoints and evaluating expressions mid-execution features were essentially in fixing issues that were found via the unit tests, and during Amazons games.

5 Amazons Framework

The amazons framework was the first stage of the implementation, however it was constantly evolving throughout development, with additional features being added and updated as they were required. It also fulfills the second primary objective of the project, "implementing a framework to allow the playing of Amazons".

5.1 Introduction

The user is greeted with an introduction message upon running the program. This allows them to select one of the three modes available: playing a game, reviewing the previous game, or going through a quick tutorial to explain how to play. Single integer values were used for the user input to save the user from having to enter a full word, making it as quick and easy as possible.

```
Welcome, to "Game of the Amazons"
Would you like to play a game? (press 1)
Or review the previous game? (press 2)
Or have a quick tutorial on how to play? (press 3)
```

5.2 Playing a game

Once the user has decided to play a game of Amazons, they must setup the game by selecting from a series of options. First, the user selects the number of human players for the game. This allows a human vs human game, AI vs human or AI vs AI, meaning the user can either play with a friend, play against the computer to practice, or watch a competitive game be played between 2 AIs. If the user selects to use AI player(s), then they will be prompted to enter the type of AI they want to use for each AI player.

The user is also given the choice of 2 board sizes, 6 by 6 or 10 by 10. I decided to add a 6 by 6 board option on top of the standard 10 by 10, as this allows for much quicker games, which is good for game simulations, as well simplifying the game for beginner players.

During play, the user enters moves as a comma separated list of algebraic notation squares (e.g. a1, a4, c2). During the early stages of development, co-ordinate values were used instead, with a single value being entered on each line. The algebraic notation was chosen as it is familiar to those who play chess, and it can make the moves easier to understand for the user.

All moves entered by the user are validated, first checking each square entered is valid for the board size, and then checking that the move can be played. This involves checking that the start position given has an amazon, amazon can move to end position with a valid move, and that the burnt square can be shot with a valid move. If any of the moves squares are incorrect, the user will be notified with a message explaining which square they entered is invalid.

5.3 Reviewing Games

This wasn't an objective outlined in the DOER originally, however after discussing the benefits for potential users of being able to review games at a supervisor meeting, this was added as an additional feature. After a game is played, the game file is stored locally, and is then retrieved the next time the program is run, allowing the user to review their last game.

When reviewing the user can navigate through the moves that were played either by moving forwards and backwards, or by entering a specific move number, to jump to that move. The are shown the board state at a specific move, as well as the move that was played. The user can also request a list of all the moves played, to allow them to a specific point of interest during the game. In the case that there is no previous game to review, the user is asked to play a game before trying to review again.

In order to allow the user to quickly jump from any move number to any other, I decided that storing a list of boards, one for each move played, was a good strategy. This allows quick access to the board state at a specified move using the move index requested by the user, rather than having to play every single move from the starting state. An alternative solution is to use a doubly linked list of boards, allowing the user to move forwards and backwards using the next and previous references.

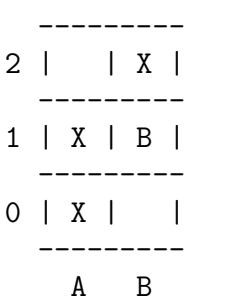
```
The previous game was played on a 6 by 6 board.
To see a list of all the moves played, enter 'm'.
To move onto the next move, enter 'n'.
To go back a move, enter 'b'.
To jump to a specific move, enter the move number (e.g. 12)
Finally, to see this list of options again, enter 'help'
```

5.3.1 Partition Game Values

To fulfill the final secondary objective, the review games section of the program shows the user how a board is evaluated with the combinatorial game theory strategy, once it is split into enough partitions. This allows the user to gain a better understand of how the CGT strategy works to evaluate a board into a game value and then return a move. This doesn't work for every game that we can review, as it may be the case that not all the partitions of a board are stored in the endgame database, making this functionality dependant on database's contents.

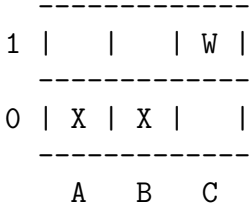
```
  -------------------------
5 |   |   | W | X |   | X |
  -------------------------
4 | X | X |   | X | B | X |
  -------------------------
3 | X | X | X | X | X | X |
  -------------------------
2 |   | X | X | X | X | X |
  -------------------------
1 | X | B | X | X |   |   |
  -------------------------
0 | X |   | X | W |   | X |
  -------------------------
    A   B   C   D   E   F

move 23: b4 -> c5 and arrow shot at b4
```

Partition 1 is shown below



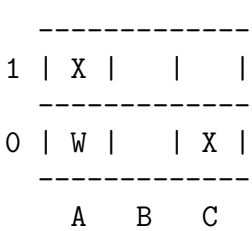
This evaluates to the game value: 1

Partition 2 is shown below



This evaluates to the game value: -3

Partition 3 is shown below

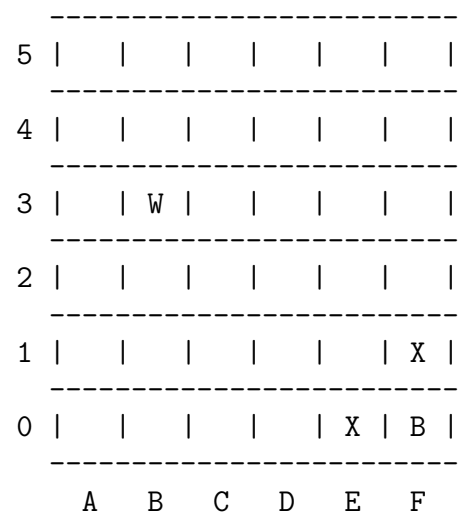


This evaluates to the game value: -3

```
Partition 4 is shown below

  -----
1 |   |
  -----
0 | B |
  -----
    A

This evaluates to the game value: 1

After adding all the partition game values, and simplifying,
the resultant game value for the board is: < 0  |  -2 >
```

5.4 Tutorial

The final option for the user, but probably the best place to start for a beginner to the game, is the tutorial mode. This fulfills the second secondary objective, which was "a basic teaching of the rules of the game". The first aim of this tutorial is to teach a newcomer to Amazons about the rules, giving a brief overview of how the pieces move and how a player wins. Next, the user must also be able to input their intended moves, so the algebraic notation is explained. Finally, to ensure that the user understands both these concepts, a board is shown that has several "winning" moves. The user is required to play one of these "winning" moves to show that they understand both the Amazons rules, and how to input moves, before the tutorial is finished.




```
Enter a winning move now to finish the tutorial.
```

5.5 Testing

The Amazons framework combines use of all the other components, and so testing the framework is essentially integration testing, combined with testing of the different framework functionality. I decided that this component required user testing rather than automated tests, to get feedback on how user-friendly it is, additional features that can be added and any bugs that are found.

Both supervisors provided feedback after testing the program at different stages of the project, requesting features such as algebraic notation being used rather than co-ordinates and removing case sensitivity from the move input. Bugs were also found during these user tests, for example a player not being able to shoot to a square they just moved from, and so this feedback was very helpful for ensuring the Amazons framework worked correctly.

6 Game Board

6.1 Printing

One of the most basic and crucial requirements of the Board class, is its ability to be output to the terminal. This allows the user to see the squares and make a decision as to where they want to move next. In order to meet this requirement, I implemented a method that prints the board in an ariel view, with the pieces shown as the letters "B" and "W", and burnt squares indicated with an "X".

The algebraic notation values for each row and column are shown also, to ensure that selecting squares for the user is as easy as possible. An example of a printed six by six board setup is shown below.

```
  -------------------------
5 |   |   |   | B |   |   |
  -------------------------
4 |   |   |   |   |   |   |
  -------------------------
3 | W |   |   |   |   |   |
 -------------------------
2 |   |   |   |   |   | W |
  -------------------------
1 |   |   |   |   |   |   |
  -------------------------
0 |   |   | B |   |   |   |
  -------------------------
    A   B   C   D   E   F
```

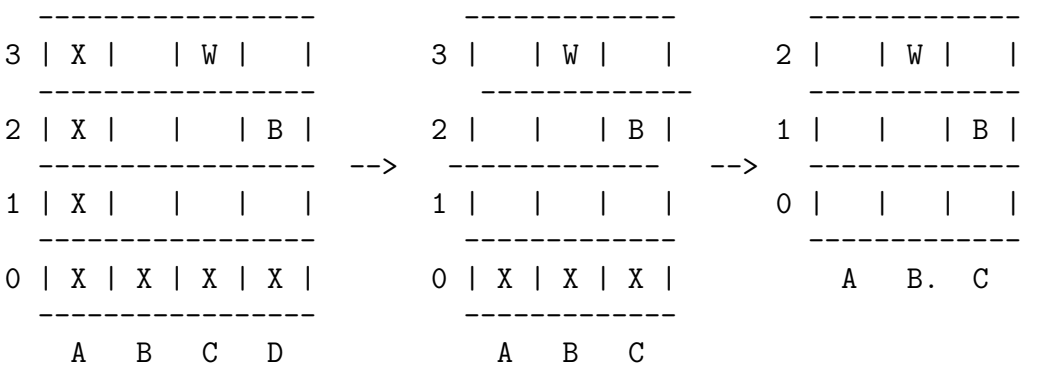
6.2 Simplification

Board objects are stored as a 2-D array of square objects. This allows boards either a rectangular or square shape. As squares are burnt off the board, the board shape can change, with entire rows or columns being burnt, and this can lead to boards which can be simplified, or split into partitions.

For a board to be simplified in this context, it must have either an outermost row or column fully burnt. Once this occurs, we can safely remove the burnt row or column, as this is no longer a playable area for the pieces.

I therefore implemented a simplification method that can be called on any board object, and will remove any outermost burnt rows or columns from the board, returned a new "simplified" board object. This is implemented recursively for cases where multiple burnt rows or columns exist.

This board simplification is applied before checking for equality, which will be discussed next, however it isn't used before evaluating a board, or displaying a board during a game, to preserve the coordinate values.



6.3 Equality

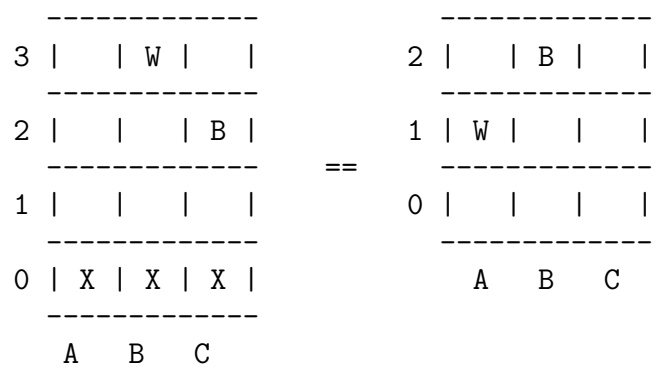
For testing board transformations, simplifications and splitting a board into partitions, it is crucial to implement an equals method, that checks the equality of two board objects. This is more complex than just checking that each of the squares match for both boards, as we must consider all the possible transformations that a board can undergo. By applying all the possible board transformations to the first board, and comparing each of these boards to the second board, we can return if the two boards are equal.

Before starting to apply these transformations to the first board however, we must first check the dimensions of both boards. If the sizes don't match, we don't give up yet, as its still possible that by rotating the board 90 degrees or 270 degrees, the sizes will match. If the sizes do match however, we have 2

possible cases to consider. The first case is when the boards are both squares, which due to the group of symmetries of a square, leave us with eight possible variations.

These variations are our original square, the 3 possible rotations (90, 180 and 270 degrees), reflection in the horizontal and vertical perpendicular bisectors of the square, as well as reflection along the diagonal bisectors (Kazdan, 2013). In the case where our boards are rectangular, we only have 4 possible variations, as two of the rotations and both diagonal reflections result in a different shape of board.

The diagram below shows 2 equal boards, as the left board is simplified (remove the burnt row) and rotated 90 degrees counter-clockwise.



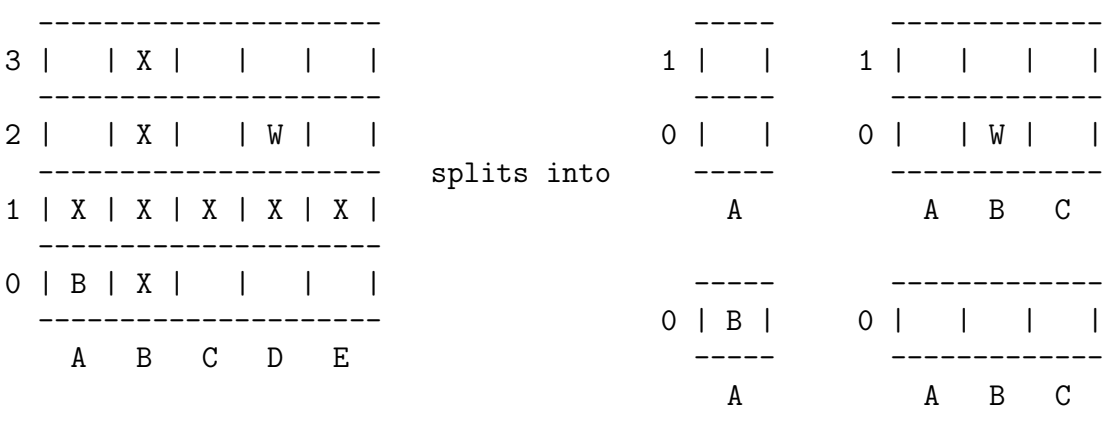
6.4 Splitting into Partitions

Due to the nature of "Game of the Amazons", each game will eventually divulge into a set of subgames, which are played on partitions of the board, and are separated by either burnt columns or rows. These subgames are the main focus of my combinatorial game theory strategy, where we decide which subgame to play in, using the gamevalue associated with each. Therefore, a method that will return a list of these smaller board partitions is necessary for the CGT strategy.

In order to split the board, I decided to use a depth-first traversal implementation of the connected components algorithm, which is an application of graph theory. This algorithm takes a set of nodes and forms a connected component

as the set of nodes that can be reached by any other node, via traversing edges (Baeldung, 2020). In the case of the Amazons board, this means choosing a square and finding all the squares that can be reached via moving either up, down, left or right, or diagonally without using burnt squares (mimicking the way a piece can move).

Essentially, my implementation gives all the squares a starting value of -1, finds an unburnt square and gives it a value of 1, gives all the squares that can be reached from that square a value of 1 also, before repeating for the next unburnt square that has the starting value, giving it a value of 2, and so on.



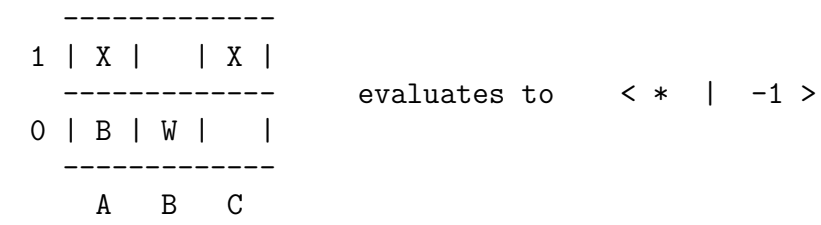
6.5 Evaluation

To use the Combinatorial Game Theory strategy, we must be able to evaluate a game state into a game value, meaning transform a board object into a game value object. To do this, we get all the possible moves for both black and white for a given board, before creating a new board and a new game value for each possible move. Each of blacks moves and game values are stored in the left list (without storing duplicates), and whites in the right list. Next, we recursively evaluate those boards, until we reach a depth where there are no possible moves for either player.

In some cases the board we want to evaluate can be split into partitions. To handle this, we evaluate each of the board partitions separately, and add the resultant game values. In order to add the game values of partitions, we must know where the partition is located on the actual board, in order to adjust the

move co-ordinates correctly. I considered implementing a method that finds where a partition is located on a board, however this doesn't handle the case of duplicate partitions occurring on the same board, so this information must instead be stored at the point of splitting the board.

How the game value objects are then used will be discussed later in the GameValue section, however from the perspective of evaluating a board object, this is all that is required.



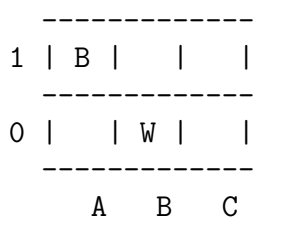
6.6 Hashcode

In order to allow the program to store an endgame database full of GameValue objects associated with Board partition objects, we need a way to generate a unique identification value for each Board object. One way to do this is to create a hashCode() method, that should return a unique value for each different Board object.

Each Board object has a 2-D array of Square objects, with each Square object being in one of four possible states. These are empty, burnt, containing a white piece or containing a black piece. So, in order to generate a unique number for each board possibility, I decided to use this state information in combination with multiplication of prime numbers.

This manual implementation of a hash code function was favoured over the use of standard libraries due to differences with their goals compared to the needs of this system. The Objects.hashCode() function states that "This integer need not remain consistent from one execution of an application to another execution of the same application" (Docs.oracle.com, 2021). As my application is utilising an endgame database that requires these values to remain the same between executions, using this library method won't provide an adequate solution.

I therefore decided to create a hashCode implementation that is dependant on the Board objects variables only, ensuring the returned value is consistent between different executions of the program.




```
generates the hashcode -1638887711
```

6.7 Testing

The board class is by far the largest for this project, with methods for all sorts of functionality, and requiring the most unit tests as a result. These tests cover boards hashcode, inversion, simplification, rotations, flipping, splitting, evaluation when board is full, empty, different shapes and evaluation when the board can be split into partitions.

I also test each of the possible moves transformations for the database smallest hash optimisation within the board tests, as the move transformation doesn't apply to just one move, rather all the moves associated with a particular board. When checking the moves are transformed correctly, I assert that each move is valid, as this confirms that the transformation has been correctly applied.

7 Game Value

The game value class is a recursive data structure, with two lists of game value objects stored, as well as an associate move. The "left" list holds game value objects for each of blacks possible moves for a given board, while the "right" lists stores all of whites possible moves. Each game value also stores its associated move.

7.1 Printing

To show the game value objects in a form that we can understand, I implemented a toString() method that returns a game value in the notation shown in "winning ways". Numerical values indicate a number of moves that either player can gain or lose by choosing a given move. Positive values are used to indicate lefts (blacks) potential gained moves, while negative numbers show rights (whites).

Symbols are used to indicate some special types of game values, including * for a game value of zero for both left and right, and $\pm$ for a "switch" game, where both players are keen to move.

7.2 Comparisons

We often want to compare two game value objects, in order to remove dominated game values, or to see what subgame we next want to move in during a game. The formula for less than or equal to given in "On Numbers and Games" as discussed in the combinatorial research section. Using this formula, I implemented a method that checks if a game value object is less than or equal to another game value object.

This less than or equal to method has many applications, including comparing two game value objects to see which is better for left, as well as returning which outcome class a game value object falls into (left, right, first player or second player win).

7.3 Equality

I initially used my toString() method to check the equality of GameValue objects, which works well for numbers, however it relies on the assumption that more complex GameValue objects always have their game value lists in the same order, which isn't always the case.

My first idea for a solution to this problem was to standardise how these lists are stored, by creating a method that would sort lists of GameValue objects into a standard form. After a bit of consideration however I quickly decided that this would be too complex, due to the large number of variables at play (values, symbols, lists etc.).

Instead, I implemented a method that checks the equality of two GameValue objects, by recursively checking each of the left and right GameValue objects had a corresponding matching object in the other GameValue. This solution seemed a bit more intuitive, taking advantage of the recursive structure of the GameValue class itself. With this definition, two GameValue objects can be equal, but have different moves associated with them.

7.4 Simplification

When simplifying a GameValue object, we want to remove any game values from either side that are dominated by other game values, as well as any duplicate game values. The game value class also contains a boolean flag that lets us know if the object has already been simplified, and will ensure that we don't waste time re-simplifying.

7.4.1 Domination

To remove dominated game values from the left side of a game value object, we first find the max game value for the left sides list. This is done using a standard finding maximum algorithm, combined with a compare game value method, which utilises the less than or equal to comparison discussed in the previous section.

Once we have the "maximum" game value for the left side, we can just iter-

ate through the left side again, removing any game values that the compare methods deems to be dominated by our maximum. The process is the same for the right side, but we are now looking for the opposite value returned from the compare method.

7.4.2 Redundancies

The second part to simplification of a GameValue object is to remove any duplicate game values on the left or right, as even though they belong to different moves, they are both equal in terms of our valuation. Duplicates refer to a game values toString() notation version (what we as humans would physically write to identify a GameValue), not the actual object reference.

In order to remove duplicates, I decided to use a function that returns the indices of any duplicate GameValues, in an ArrayList. This separation of removing the duplicates and iterating over the list exists to avoid Concurrent Modification Errors.

7.5 Testing

Before the game value unit tests are run, a range of different game common game values (0, 1, *, 1/4 etc.) are generated, to be used throughout testing. The unit tests cover deep copying, inversion, simplification, equality (including out-of-order), getting the simplest form, and the different outcome classes a game value can fall under.

8 Endgame Database

The combinational game theory strategy relies on an endgame database to work correctly, ensuring the best possible chance that the board partitions can be evaluated at run-time and therefore allowing a game value to be returned. This implementation was devised through a series of incrementally improving approaches, which will be explained next.

8.1 First approach

When evaluating a Board object, we want to return a GameValue object that represents all the possible moves for both players, in that board. In order to optimise the amount of time required for the AI to play a move, we can store the evaluated boards GameValue objects in a HashMap, which acts as an endgame database.

To store and retrieve these GameValue objects from our HashMap, the hash-code value of a board acts as a key, uniquely identifying each board. This HashMap is stored in memory at run-time, which allows for very fast access and also removes the possibility of having to evaluate the same board multiple times. The downside of storing the whole endgame database in memory is the scalability limitations it brings, with the size of the database being dependant on the size of memory available at run-time.

This HashMap is stored in a file at the end of each game, and read from the same file into memory before a new game begins. This allows it to persist in between different executions of the program, meaning as the user or AI strategies play through games it will grow in size and produce quicker moves more often.

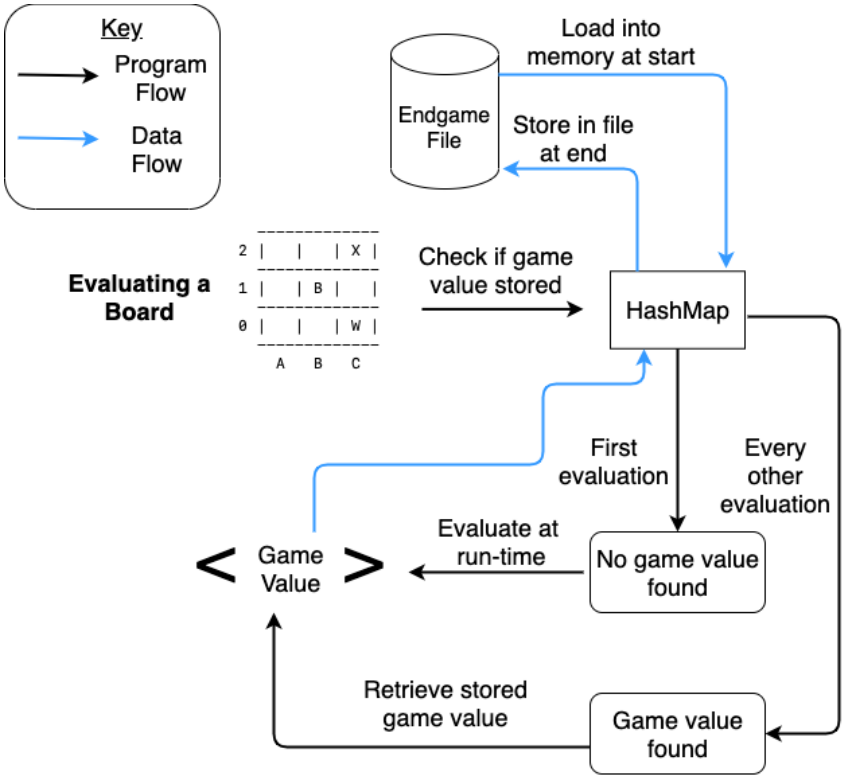


Figure 1: Evaluating a board, with endgame file optimisation

8.2 Second Approach

The first approach optimises the amount of computation required to return an evaluated board by storing the associated game values in a HashMap, and a file in between executions. This does however result in a lot of redundant information being stored if we consider the different transformations a board can go through and still return the same game value.

Each rectangular board has 4 variations which can be attained by rotation as well as horizontal and vertical flipping of the board. This results in the first approach evaluating the same board at least 4 times (8 times in the special case of a square board), as well as storing this redundant information in our endgame database.

A better approach is to instead only store the board variation with the small-

est hash value in our endgame database. This adds additional complexity in retrieving a game value for a given board, as we now have to apply transformations to each of the moves positions from the smallest hash value board, to the board we are trying to evaluate.

The special case is of course when the smallest hash value board is the board we are trying to evaluate, in which case we don't need to apply any transformations. In the general case however, we know the transformation required to get from our current board to the smallest hash value board, and so we can just apply the reverse transformation to each of the moves start, end and shooting coordinates, in order to return the equivalent move on the board we are evaluating.

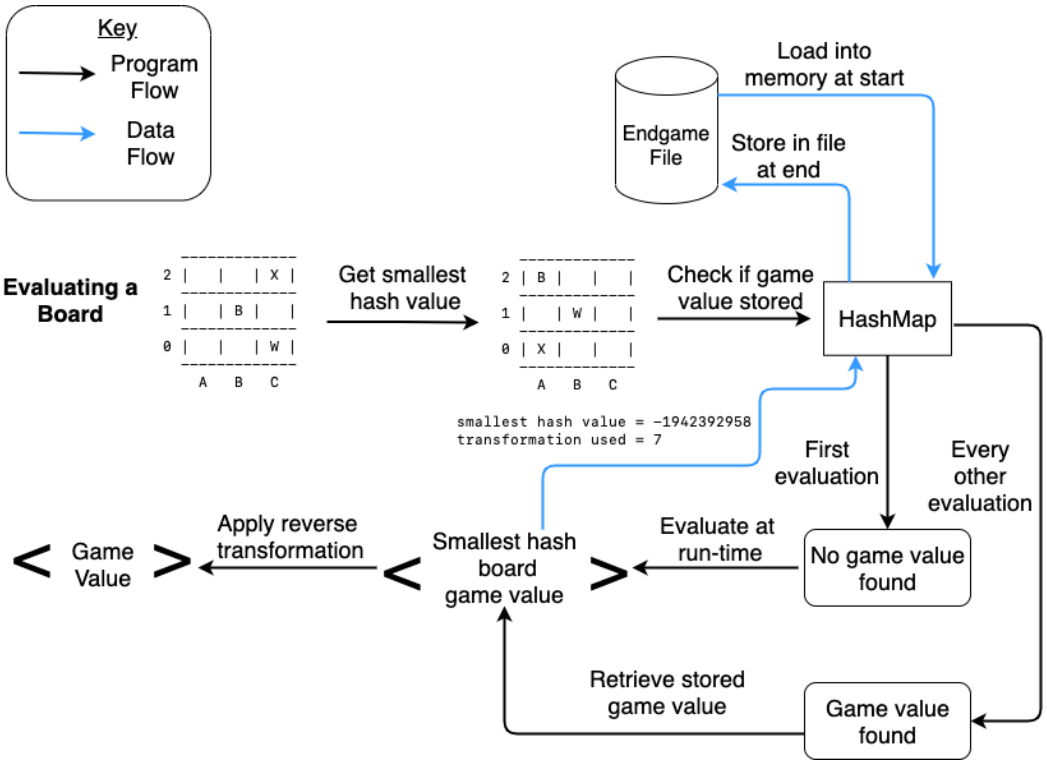


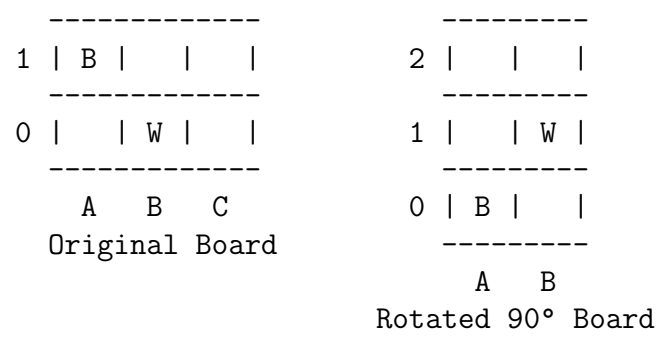
Figure 2: Evaluating a board, storing only lowest hash board variations in file

8.2.1 Move Transformations

Using the formula I derived for each transformation, this is just a simple arithmetic operation for each square in the move, meaning it only requires a small amount of computation at run-time to transform each move within the game value. Only the moves at a depth of 1 are transformed, as they are the only moves a player can make, which reduces the complexity required.

For a rotation at for a given board square (x, y), a 90 degree anti-clockwise rotation can be found using the formula below. This can then be used to rotate the square into any one of the 4 board variations, by multiple applications.

$$x' = y$$
$$y' = maximum\ x\ index - x$$

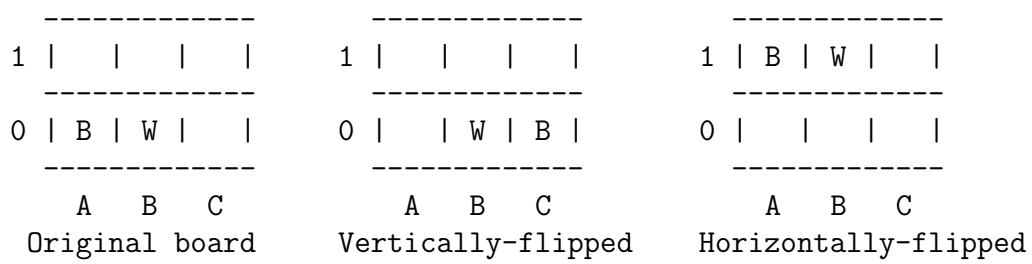


For a vertical flip, we use the formula below, as our squares position on the y-axis is stationary, while we flip the position on the x-axis.

$$x' = maximum\ x\ index - x$$
$$y' = y$$

Similarly, for a horizontal flip, we use the formula below.

$$x' = x$$
$$y' = maximum\ y\ index - y$$



8.3 Final Approach

Both the first and second approaches rely on HashMap being stored in memory at run-time and they utilise a file that the HashMap is read from and written to at the start and end of the program execution. This storing of a Java object in a file works on a small-scale, however it leads to exceptions and crashes as the object gets larger, meaning it isn't a very scalable solution.

Another issue with the first two approaches is their reliance on memory to store the HashMap, limiting the scalability of the solution once again. As we increase the number of game values stored, this causes significant time overheads at the start and end of the program, as the entire HashMap is read from and written to the file.

To solve both of these issues, and to provide a more scalable solution, I decided to use a SQL Database to store each of the key-value pairs that were previously stored in the HashMap. The table structure is simple, with a primary key column for the keys, and a value column for the associated game values, which are stored as a medium blob type. This medium blob type allows objects of up to 16.78MB to be stored, which should be enough for any game value object.

The database approach removes the need to read and write a large java HashMap at the start and end of the programs execution, instead we just query the table for a key value at run-time and only read in one key-value pair at a time. This

makes the program more reliable, and reduces the amount of time overheads at the start and end of execution. It also removes the need to store the entire endgame database in memory, instead storing it on disk and querying / inserting rows when required. This provides a much more scalable solution, removing the bottleneck of the size of memory.

The real reason we want to use an endgame database is to allow for the evaluation of boards that can't be evaluated mid-game, due to their complexity. Therefore we must fill the database with larger and more complex partitions, and we no longer want to evaluate boards at run-time, only look them up in the endgame database. If all the partitions are stored, then we can evaluate the board into a game value, and if not we just use a different strategy to return a move.

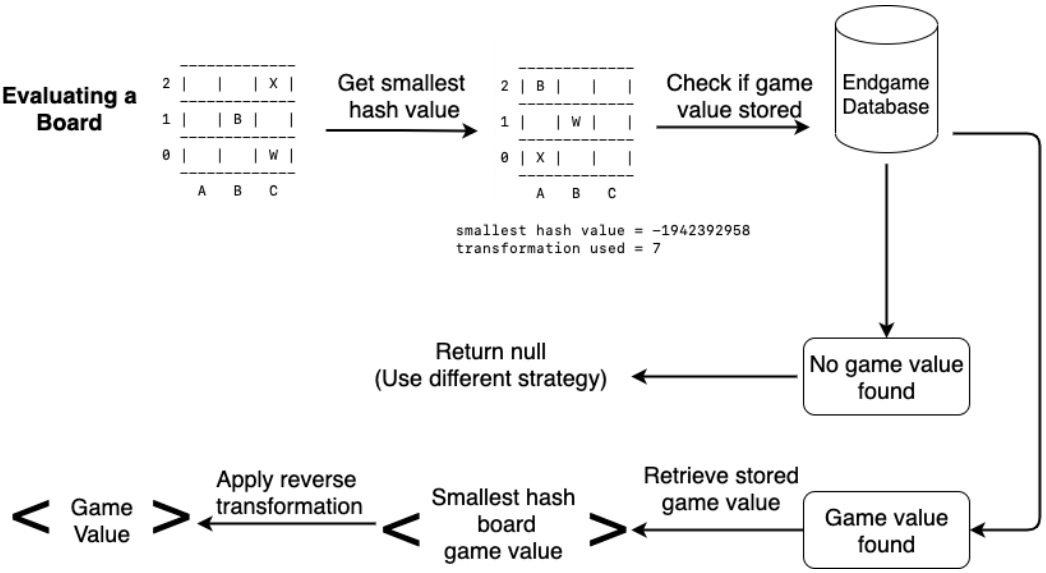


Figure 3: Evaluating a board, utilising h2 database

8.4 Filling the Database

In the first and second approaches, the HashMap was filled with game values as the boards partitions were evaluated during games. This strategy can however take a lot of games and doesn't necessarily ensure that all boards will be seen and evaluated. Instead, to ensure that the database contains game values for all the possible boards up to a given size, I decided to create a method that

systematically fills the endgame database.

This is achieved by generating all possible board shapes up to a specified size, and then iterating through all possible board variations for each possible board shape. To get all possible board variations for a given shape, I generate a list of integers, one for each square, and give each integer a value between 0 and 3. These integer values indicate one of the four possible states a square can be in, empty, black piece, white piece or burnt.

Next, we must find all the possible lists that can exist, for size n, and each value in the range 0 to k. I did this recursively, with each call being given a position in the list, a value to add to the list, and then calling itself k times, passing each recursive call a different value in the range 0 to k.

Once we have a set of all possible lists of size n, with values in the range 0 to k, each list represents a board and each value represents a square on that board. By filling the boards squares with their values associated states, we now have every possible board variation of the particular shape we were given. It should be noted that I remove any boards where all the squares are burnt, as these can never be a partition, as well as any boards with over a specified number of empty squares, as the evaluation of each board can take a very long time above 8-9 empty squares.

For a board of size 3 by 3, there are 4^9 (262,144) possible variations, and once we move up to a 4 by 4 board this value reaches 16,777,216. This is the first variable affecting how long the database filler runs for, with the other being the number of empty squares, which dictates how long a given board will take to evaluate once it it's found to be missing from the database.

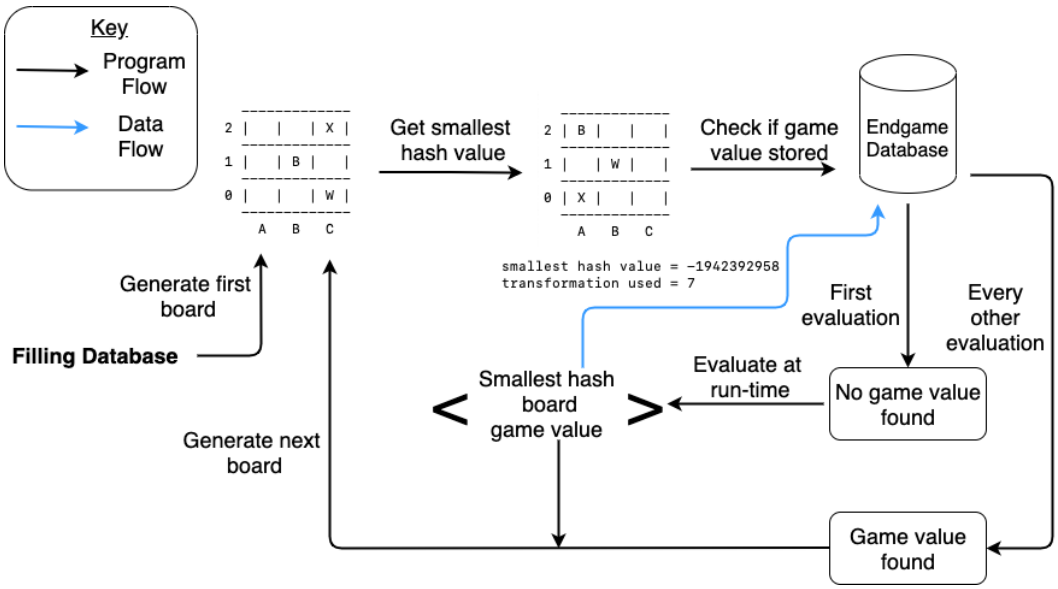


Figure 4: Filling the endgame database

Initially, to generate all board combinations I stored all the possible boards up to the specified size, however at the 4 by 4 size, this causes an OutOfMemory error. The solution is to generate each board and apply the evaluation to it immediately, before moving onto the next board, removing the need to store a list of boards. These kind of memory optimisations aren't required when working with small amounts of data, however they are essential for larger datasets.

8.5 Testing

The database tests use the same database, but with different table, to allow the use of testing data. Every time the database tests are run, the database is first dropped, and before every test, the table is emptied, to allow the tests to be run in any order and give consistent results.

The required functionality of the database is storing a hashcode value and associated game value from the table, and as unit testing doesn't allow for a specific ordering of tests, this is all tested in a single test method. We also want to be able to retrieve the size of the database, so this is also tested.

9 AI Implementations

9.1 Random Choice

The first Amazons AI that I implemented uses a random choice move strategy. One of the benefits of this strategy is the almost instantaneous move selection, as a list of valid moves for the specified colour is generated, and a move from this list is selected at random. This also allowed for quick testing of the Amazons framework during development, as full games can be played out between 2 of this type of AI, in matter of milliseconds.

This AI implementation was kept throughout development and used for the experiments as its a good benchmark for testing other Amazons strategy implementations, any "good" strategy should be able to outperform this.

9.2 Heuristic

During my Amazons strategy research, I found that a lot of the other implementations favoured a heuristic-based approach, especially in the opening and middle-game. The reason for this is the large branching factor present in Amazons, and so a heuristic method that doesn't check all the possibilities at a depth of multiple moves is advantageous.

As previously discussed in the context survey, the consensus seems to be that a min-distance heuristic optimal. For example, "Amazong" assigns values to each square indicating how many queen moves are required to reach it, for both players (Lieberum, 2005). This is useful as a way of finding what squares each player has better access to, allowing the program to shoot the squares that are most accessible to their opponent, limiting their possible moves.

I decided that a simpler method of limiting an opponents possible moves, was to do just that, find the move that left the opponent with the smallest list of possible moves. This only requires checking for each possible move, how many possible moves does this give the opponent, and then utilising a standard finding min algorithm.

The advantage of this strategy is is speed, quickly finding a move that may not be the optimum, but is definitely a contender.

9.3 Combinatiorial Game Theory

The combinatorial game theory strategy attempts to use the board evaluation to return the associated game value. This game value is simplified, removing all the dominated and redundant moves, leaving us with a list of the best moves, from which one is then picked, and returned. This strategy requires the board to be split into partitions that are either stored in the endgame database, or can be evaluated at run-time.

In the opening and middle-game, this strategy can't be employed, as the board isn't separated into small enough sub-games. In order to deal with this issue, we can use a different strategy until we reach the endgame. Using the results discussed in the AI experiments section, I decided that the Monte-Carlo strategy was best suited for this, due to its high winning rate compared to the heuristic strategy.

9.4 Monte-Carlo Tree Search

The final Amazons strategy implemented utilises a Monte-Carlo Tree Search algorithm. This algorithm is split into 4 stages: Selection, Expansion, Simulation and Propagation.

Selection

Due to the large branching factor present in Amazons, the first stage of the algorithm is the most crucial, as it defines what branches of the tree will be visited most frequently. An Upper Confidence Bound (UCB) value is assigned to each node, and this value is then used to select a "promising" node.

$$UCB = \frac{win\ score}{visit\ count} + c\sqrt{\frac{\ln(parents\ visit\ count)}{visit\ count}}$$

The first component of the formula is always a value in the range 0 to 1, with nodes that provide more wins being given a value closer to 1, to ensure that "promising" nodes are visited the most. The second component of the formula relates to exploration, it gives a higher value to nodes that have had fewer visits, in an attempt to reduce the likelihood of node starvation.

The c value is known as the "exploration parameter", and it essentially defines the weighting we want to give to exploration. This is typically given the value $\sqrt{2}$, however we don't want to explore as large a portion of the tree, due to the large branching factor in Amazons. A better strategy is to focus on a smaller portion of the tree, but in more depth, giving us more confidence that we have found a "good" move, instead of trying to search the whole tree for the optimum move.

As discussed in the next section, after experimenting with the exploration parameter value, I found that an exploration parameter value of 0.8 produced "good" moves the most frequently, resulting in a higher win percentage. This lower value producing better results agrees with my hypothesis that Amazons is better suited to a deeper rather than wider search, due to its large branching factor.

Expansion

Once a node has been selected, if it is not a terminal node then we can expand the tree, meaning add a child node for each of the possible moves. We then randomly choose one of the newly created children nodes, to act as our promising node for the simulation stage.

Simulation

The simulation stage is just playing out the promising node with random moves from both players, until the game is finished. An optimisation of this stage is possible, with heuristic moves being played instead of random, which introduces some domain knowledge about Amazons to the algorithm.

Propagation

After a game is simulated and the winner is returned, we can then propagate this result back up to the root of the tree. Each node has a win score associated with it, indicating how many wins have occurred from that node, and so each node from the simulated node to the root node has its win score updated appropriately.

9.5 Testing

The AI implementations unit tests use a 6 by 6 board example to ensure that each AI strategy returns a valid move for both the black and white pieces. As this only ensures that the moves are valid, simulation games were also used to check that better moves were made using the better strategies, and the AI experiments results show this to be the case.

10 AI Experiments

The program also has functionality to run simulation games, allowing us to test the effectiveness of the different Amazons strategies. By passing the command-line argument "experiments", the program will allow the user to select 2 AI types, and enter the number of simulation games to run, before returning the results. For the simulation games, both AI types are given the white pieces half of the time, to ensure a fair test.

10.1 Monte-Carlo

10.1.1 Exploration Parameter

In order to choose a value for the exploration parameter, I decided to run some experiments, testing how a set of values each performed against the heuristic strategy. Each evaluation parameter was tested in 100 simulation games, on a 6 by 6 board, with the Monte-Carlo AI being given 5 seconds per move.

I first tried an evaluation parameter of zero, which essentially removes the exploration component completely from the UCB formula, leaving only the average wins percentage component. This clearly isn't optimal, as it only wins 18% of games against the heuristic.

On the other end of the scale, a value of $\sqrt{2}$ typically given, so this was also tested, winning 62% of its games. As this value was lowered, the win percentage increased, until reaching a value of 0.8, with a win percentage of 88%. These results back up my claims that for Amazons, a lower exploration parameter allows for more accurate moves, due to the large branching factor.

Exploration Parameter	**Monte-Carlo Win %**
0	0.18
0.5	0.76
0.8	0.88
1	0.82
$\sqrt{2}$	0.62

10.1.2 Heuristic Optimisation

The Monte-Carlo algorithm uses random play during the simulation stage, however in some cases using a better strategy than random choice can yield better results. To test this, I decided to use the heuristic strategy to playout the simulation games, rather than random play, and compare the two.

The results show a decrease in the performance of the Monte-Carlo strategy when the heuristic is used to simulate the games, rather than random play. This could be caused by the increased time required to simulate the games due to the extra computation required for each move, leading to a less developed tree and therefore less accurate results. These results led me to continue using random play in the Monte-Carlo strategy.

Simulation Strategy	**Monte-Carlo Win %**
Random play	0.82
Heuristic	0.68

10.2 Strategy Comparisons

Next, I decided to test how all 4 strategies performed against each other. For these comparisons, I am using the Monte-Carlo strategy with 5 seconds per move, and an exploration parameter of 0.8. The combinatorial game strategy was first given 5 hours on the school host servers, to evaluate board partitions up to size 4 by 4 and with a maximum of 5 empty squares, and fill its database with the resultant game values. This resulted in an endgame database of 105,697 entries (roughly 1.5 GB), which with the smallest hash optimisation means at least 500,000 possible boards are stored.

I ran 100 simulation games between each strategy pair, using the "experiments" option of the program. The values indicate the win percentage of the first strategy (left-most column) against the second strategy.

	Heuristic	**Random**	**CGT**	**Monte-Carlo**
Heuristic	0.5	0.94	0.26	0.18
Random	0.06	0.51	0	0.02
CGT	0.74	1	0.49	0.52
Monte-Carlo	0.82	0.98	0.48	0.52

Comments

When the heuristic plays itself, the win percentage is always exactly 50%, as there is no random element, meaning the same moves will be selected by both players.

Interestingly, when the Monte-Carlo strategy played the heuristic, it won 82% of the time, however when the combinatorial game theory strategy faced the heuristic, it only won 74% of its games. The CGT strategy uses the Monte-Carlo strategy until the board is partitioned enough to evaluate, and so I expected a slight improvement rather than a decline in its performance. This can easily be explained by the random nature of the Monte-Carlo strategy, with a bigger sample possibly returning a different result.

When the combinatorial game theory strategy came up against the Monte-Carlo strategy however, it did show a slight edge in performance (winning 52% of the games), which could be the result of better move choice as a result of the board evaluation. The fact that the CGT strategy can only be employed once we reach an end-game state of the board means that the improvement on the standard Monte-Carlo strategy is only slight, as most of the games are won or lost before reaching this state. The moves made once the board is partitioned enough may be optimal due to the CGT strategy, however for some games the battle may already be lost.

When playing against random moves, the combinatorial game theory strategy won all 100 games, giving it a perfect 100% win percentage. This is another slight improvement again when compared to the Monte-Carlo strategy.

11 Evaluation and Critical Appraisal

11.1 Objectives Success

In terms of meeting the expectations set by the initial objectives, this project has been a success. All of the main goals of the project were met, with all but one of the secondary aims completed as well as some additional requirements that were defined during development.

Primary Objectives

1. Research combinatorial game theory, specifically the game 'Amazons'
2. Implement a framework to allow the playing of Amazons
3. Develop a program that plays Amazons using strategies that are based on the underlying mathematics
4. Develop other program implementations that play amazons
5. Run experiments to test my implementations effectiveness

All of the primary objectives were fulfilled, including adding some additional features to the Amazons framework, as well as 3 alternative AI implementations that used a range of different strategies. The combinatorial game theory research spanned several books, and proved crucial to develop the CGT strategy. The Amazons framework was also a success, providing a user-friendly interface to allow users to play or spectate Amazons games.

The Amazons combinatorial game theory strategy was tested and shown to work effectively in the AI experiments section, which proved it to be a good strategy for the end-game of Amazons, when compared to the other implemented strategies. I spent a lot of time further optimising this strategy, first with storing game values in a file and then also removing redundancies using board symmetries. The database further improved my implementation, allowing for a much more reliable and scalable solution, which can potentially be used to store a very large number of game values.

The other AI implementations that were developed provided a good benchmark for testing the CGT strategy, while the Monte-Carlo strategy allows for good

performance in the early and middle-game, before the CGT strategy can be utilised. The experiments were also helpful in terms of optimising both the Monte-Carlo strategy with its exploration parameter, as well as comparing the performance of each strategy.

Secondary Objectives

1. Develop a nice GUI to show the playing of the game
2. Basic teaching of the rules of the game (like a tutorial)
3. Teaching of the maths behind decisions for the user to see, to develop their skills in-game

I decided that rather than developing a GUI, my time would be better spent optimising the CGT strategy and adding additional functionality like the reviewing games feature. This decision was made as the console-based solution works well, with input validation and a board being output to the user in a very clear format, so there wasn't much need for a better UI.

The teaching of the game rules is fulfilled with the tutorial section of the Amazons program, giving the user a detailed explanation and also testing their knowledge with a "winnable" board position to play from.

The teaching of the maths behind decisions is implemented within the review games functionality. Once the game board is split into enough partitions to be evaluated, the user is shown each partition, along with its game value, as well as the resultant game value for the whole board. This gives the user a better understand of how the combinatorial game theory strategy works under the hood.

Tertiary Objectives

1. Research and develop programs for similar games or variants of Amazons

Due to the scope of the rest of the project, and the time allocated, I decided to focus on the rest of the objectives and additional requirements rather than spend time developing programs to play other combinatorial games. With the game value class already implemented, this would however make adding other game variants substantially easier than starting from scratch, and leaves the door open to future projects on other combinatorial games.

11.2 Similar Work

Comparing this project to related work done on the topic, progress has been made in terms of actually implementing the theory in practice, for a full game. Previously, research has been done on the sum of n by 2 Amazons partitions, analysing thermographs of hot games to determine their temperature (Berlekamp, 2000). While this project didn't delve into the thermographs, it instead focused on using the game values of partitions to evaluate a whole board, and this along with an endgame database of previously evaluated partitions hasn't been done before, at least not in the public domain.

The project has provided a new strategy for Amazons, combining the Monte-Carlo strategy which has been previously been implemented for Amazons (Kloetzer et al, 2007), with the combiantorial game theory strategy that has been used in theory (Berlekamp, 2000).

12 Conclusions

In conclusion, this project has been a very enjoyable experience, from the initial planning stage, throughout development and testing, and finally in writing this report. It has taught me about the theory of combiantorial games, as well as some good strategies to play Amazons. Hopefully it also provides a way for beginners to the game to learn and improve in Amazons, as well as teaching how partitions can be evaluated and used to decide on a good move.

The optimisations of the board evaluation which allow for a large database of evaluated partitions would have to be one of the key achievements of the project. Another is implementing a Monte-Carlo Tree Search strategy for the early and middle-game, before optimising its exploration parameter using simulation games.

One drawback of my project could be its reliance on the endgame database to evaluate game boards, which requires time to fill before a game, and as the database grows in size its takes longer to query and fetch game values.

Due to the time allocated, Amazons was the only combinatorial game framework implemented, however the project could be extended in the future to be used with chess, checkers or go. Both the Monte-Carlo and combinatorial game theory strategies could be generalised in this way, to accommodate the playing of these other games.

References

[1] Conway, J., 1976. On numbers and games. Wellesley, Mass.: A.K. Peters.

[2] Nowakowski, R., 1998. Games of no chance. Cambridge: Cambridge University Press.

[3] Berlekamp, E., Conway, J. and Guy, R., 1982. Winning ways for your mathematical plays. Available at: https://annarchive.com/files/Winning%20Ways%20for%20Your%20Mathematical%20Plays%20V1.pdf [Accessed 18 March 2021].

[4] Pegg Jr., E., 1999. Amazons. [online] Chessvariants.com. Available at: https://www.chessvariants.com/other.dir/amazons.html [Accessed 28 March 2021].

[5] Martin Müller, and Theodore Tegos 2002. Experiments in computer amazons. In More Games of No Chance (pp. 243–260). Cambridge University Press. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.171.553 [Accessed 28 Mar 2021].

[6] Berlekamp, E., 2000. Sums of N × 2 Amazons, Available at: https://projecteuclid.org/download/pdf_1/euclid.lnms/1215089741 [Accessed 28 March 2021].

[7] Lieberum, J., 2005. An evaluation function for the game of amazons. Theoretical Computer Science, 349(2), pp.230-244. Available at https://www.sciencedirect.com/science/article/pii/S0304397505005979 [Accessed 28 March 2021].

[8] Kloetzer, J., Iida, H., and Bouzy, B. 2007. The Monte-Carlo approach in Amazons. Available at https://www.researchgate.net/publication/258121759_The_Monte-Carlo_approach_in_Amazons [Accessed 28 March 2021].

[9] Kazdan, J., 2013. [online] Www2.math.upenn.edu. Available at: https://www2.math.upenn.edu/~kazdan/202F13/notes/symmetries-square.pdf [Accessed 17 March 2021].

[10] Baeldung on Computer Science. 2020. Connected Components in a Graph — Baeldung on Computer Science. [online] Available at: https://www.baeldung.com/cs/graph-connected-components [Accessed 17 March 2021].

[11] Docs.oracle.com. 2021. Object (Java Platform SE 7). [online] Available at: https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode() [Accessed 5 February 2021].

Appendices

A Testing Summary

The project was tested throughout development using automated unit testing, across 4 test classes, which are used to test the different components of the project. These tests can be run using a command-line argument, which is discussed in the "Options Usage" sub-section.

1. AITests
2. BoardTests
3. DatabaseTests
4. GameValueTests

In order to pass all the unit tests, an endgame database of every board shape and variation up to size 3 by 3, with at least 5 empty squares is required. This is submitted with the project, however it is reproducible using the command line options "resetDatabase" and "fillDatabase". I recommend giving the program a couple of hours (enter 120 minutes) to run when filling up with the boards to pass unit tests.

The debugging was done using these unit tests, in combination with the IntelliJ debugger, which allows for breakpoints and evaluating of expression at specific points in the programs execution.

B User Manual

The project uses 3 libraries which must be added to the classpath for both the compilation and running of the program. The compilation and running of the program must occur from within the "src" folder.

B.1 Regular Usage

The project must be compiled using the command:

“javac -cp ”.:../lib/junit-4.13.2.jar:../lib/hamcrest-all-1.3.jar:../lib/h2-1.4.200.jar“ *.java”.

To the run the program, use the command:

“java -cp ”.:../lib/junit-4.13.2.jar:../lib/hamcrest-all-1.3.jar:../lib/h2-1.4.200.jar“ GameEngine [Option]”.

B.2 Options Usage

The program has a set of command-line options, allowing the user to access different functionality.

“**experiments**”: allows the user to specify 2 AI Types to run some simulation games with.

“**unitTests**”: runs all the automated unit tests, showing the success of each and any errors that occur.

“**fillDatabase**”: fills the endgame database with partitions up to specified size and a specified number of empty squares, for a specified amount of time

“**resetDatabase**”: drops the current database table, and creates a new one

“**databaseSize**”: checks the number of entries in the endgame database

B.3 Project Layout

The project consists of the following directories.

./src/ contains all of the project code

./lib/ contains the 3 external libraries used for the project.

./JavaDoc/ contains all of the API documentation for the project