Algorithms and Data Structures

40168130
40168130@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

1 Introduction

Implementing a checkers game in C# that allows human vs human, human vs AI and AI vs AI play, whilst also having a replay feature that allows playback of a previous game and the undoing and redoing of moves is a problem that requires careful choice of algorithm for the AI and data structures to allow fast and efficient processing. All code snippets throughout this report will be in C#.

2 Design

This section will break down the design of each key individual feature of the game, explaining why each data structure was chosen and in the case of the AI, which algorithm was chosen and why.

The Board & Pieces A draughts board when broken down is simply a grid of x and y coordinates. A 2D array is accessed in a very similar way using two integers to retrieve the value at that position in the array *array[x,y]*. This is why it is a logical choice to use a 2D array to represent all piece positions on a draughts board. Each move will be from an x and y coordinate to an x and y coordinate and these can be used to call the values at these positions as shown in listing 1.

Listing 1: Getting the piece type from a moves x and y coordinates in C#

```
piece = positions[move.XFrom, move.YFrom];
```

Simplicity was key when choosing the data type for the pieces, there are only 4 possible piece types in a game of checkers and creating a custom type to represent them seemed unnecessary, so instead integers were used. Ranging from 1 to 4 each integer represented a different piece type with 0 representing an empty space on the board. However, doing comparisons using nothing but integers can begin to become hard to read for a developer so opting to create a static *Pieces* class to store the values against a static variable was much more developer friendly.

Listing 2: statement before using static Pieces class

```
switch (move.PieceTaken)
{
case Pieces.1:
case Pieces.3:
whitePieces--;
break;
case Pieces.2:
case Pieces.4:
blackPieces--;
break;
}
```

Listing 3: statement after using static Pieces class

```
switch (move.PieceTaken)
{
case Pieces.White_King:
case Pieces.White_Man:
whitePieces--;
break;
case Pieces.Black_King:
case Pieces.Black_Man:
blackPieces--;
break;
}
```

Moves Each move in draughts requires at least 2 sets of coordinates, an x and y from, and an x and y to. Arguably you could have more coordinates for multiple captures in a move but refraining from doing so allowed for easier undoing and redoing of moves, allowing the player to undo to before their second capture rather than having to undo all their captures at once and requiring less calculations in the code. Again simplicity was key when storing these values, so each move was designed to store 4 integers, with each one representing a part of a set of coordinates. This in itself would be enough to have a working checkers move, but in order to effectively implement both the move logger and AI more information had to be stored about a move so that the board values could be correctly adjusted and checks made purely from information stored about the move. The player, the piece being moved and any piece being captured were also stored as integers, this was due to the player being represented as either 1 or 2 and each piece type being represented as an integer from 1 to 4. These needed to be stored so that when undoing and redoing a move the piece positions could be adjusted to the correct values from before or after the move was made, this also allows for cleaner functions that instead of passing around the player number between them as parameters, and having to calculate the

piece being moved multiple times, the a *move* type variable can simply be passed around instead. Storing only 2 sets of coordinates per turn cause a problem when a player wants to make a successive capture. This could have been solved by having a simple boolean stored in the move that stores whether or not the player can make another capture after the move has been made, but this would involve calculating the moves the player can legally make multiple times per capture and calculating again if the player wanted to redo a move. The more efficient option was to store all possible successive moves that can be made as a list of moves that could be checked against easily.

Move Logger The move logger was initially designed to easily allow the user to undo and redo moves made throughout the course of the game. The stack lends itself perfectly as a data structure for an undo system, this is due to it being a "Last in First Out" *(LIFO)*, structure [1]. This structure restricts access to a data set so that items can only be added to one end of the set which I will refer to as the "top".
C# has it's own stack class already implemented which allows for adding an item to the stack *(*pushing) using *stack.Push()*, and for removing an item from the top of the stack *(*popping) using *stack.Pop()*, this method is particularly useful as it also returns the item that has been removed so it can be processed.
Creating a stack of *Move* types give a data set that can be easily used for storing all moves made as each move made throughout the game can be pushed to the stack. Moves can be undone by popping a move from the undo stack and processing it so that it is represented on the board. Redoing moves can be done in a similar fashion by pushing each move that has been undone onto a separate move stack and then popping the moves from this second stack whenever the user redoes a move and processing them so that it is represented on the board.

Listing 4: Undo and redo move functions in C#

```
public Move UndoMove()
{
    Move move = undoStack.Pop();

    redoStack.Push(move);

    return move;
}

public Move RedoMove()
{
    Move move = redoStack.Pop();

    undoStack.Push(move);

    return move;
}
```

This initial design of simply undoing and redoing moves evolved as it became apparent that the move logger could have other uses as well. Displaying the undo log would allow users to see a full list of all moves made, and loading all moves from a game into the redo log in reverse order would allow the *RedoMove()* function to be used as a replay feature.

Artificial Intelligence Choosing an efficient but effective algorithm for the AI isn't simple, there are several effective approaches for zero-sum games like draughts. However, multiple sources pointed to the minimax algorithm, a recursive algorithm that builds a tree of possible board states and assigns a value to each of them. The values of each board state are determined by a set of conditions such as, pieces taken, piece kinged etc. In order for the board state and all required values to be accessible in the algorithm a new data type would have to be defined. The *MoveNode* type, was designed to store the current board state, the move being applied, the value of the board state, and the nodes parent and children. When designing an AI using this algorithm one of the key variable components is how each move is valued, should capturing a king worth the same as capturing two regular pieces and other similar questions have to be asked. The initial implementation of this algorithm used a system of king captures are worth double of a regular piece but this would result in some situations where a piece would wait for another to be kinged so it could then capture it, as it was now worth double. Small unexpected issues like this result in a lot of trial and refinement of different number combinations. Using just the minimax algorithm can become resource intensive quickly with a performance of $O(b^d)$ where *b* is branching factor and *d* is depth. This is due to every possible move being evaluated, which for a search depth of 5 moves can reach over 20000 nodes to evaluate per turn, due to this it is sensible to apply some kind of pruning to the algorithm. Alpha-beta pruning can work great with the minimax algorithm, if node ordering is optimal alpha-beta pruning can give a best case performance of $O(\sqrt{b^d})$ but isn't perfect as it can still return a worst case performance of $O(b^d)$, the same as the standard minimax algorithm, this is dependant on the ordering of nodes.

3 Enhancements

There are several additions and improvements that could be made to this project, some of which were planned for but unfortunately couldn't be implemented due to time constraints.

GUI This was a large part of the project that was intended to be implemented from the initial design stages. The games framework was built was so that it could be used in a plug-in fashion with both a console application and a full GUI but this couldn't be implemented due to time constraints.

AI Piece Position Value Whilst the AI currently works relatively well there could be some improvements made to the calculations used for calculating the value of a move. The AI currently only takes into account whether a piece is captured, becomes a king and if it can make a successive capture when calculating move values but board position could also be taken into account as is explained by

Kevin Wilson [**?**] each position on the board can be given a value which takes into account how exposed a piece would be in that position. This combined with the calculations currently being used could create a much smarter AI that takes it's surroundings into account more.

Mid-game Saves/Multiple Saves At present the game automatically saves a game as json file once a game has finished, this feature could be greatly improved to allow the user to save the game at any point so that they may return to it at a later time. The game also only saves to the one json file, it would be preferable if multiple games could be saved either to the same file or multiple files so that user could choose which game they would like to replay.

Tuples Move start and end positions are stored as 4 separate integers, these could instead be replace with 2 tuples and with C# 7 these tuples can now be named, which would improve code quality and maintenance.

Listing 5: A move with ints and with tuples and how each are called

```
int xFrom = 1;
int yFrom = 2;
int xTo = 2;
int yTo = 3;
positions[move.xFrom, move.yFrom];


Tuple from = (x: 1, y: 2);
Tuple to = (x: 2, y: 3);
positions[move.from.x, move.from.y];
```

4 Critical Evaluation

The move logger in particular stands out as a surprise success. Not only does it successfully allow tracking and displaying of all moves made, along with the undoing and redoing of moves, it has also managed to double up as a data type for the replay feature, something which was completely unplanned when designing the project but resulted in a smaller code base and cleaner code. Along with that the separation of game functionality into it's own framework without any dependencies to the console application was successful and if time was available to make a GUI it would have made the process much quicker and smoother. In contrast to these the AI, whilst functioning, is slower than expected and can quite frequently make moves that have no apparent pay-off. This is due to more tweaking being needed with how values are assigned to a node, and the lack of ordering of nodes causing the pruning to skip some potentially good move choices.

5 Personal Evaluation

Throughout this project I have constantly been learning, from the very start with using a 2D array, a data type which I have never had to use before, all the way to the last lines of my AI. Building the console application was surprisingly interesting as at first it seemed that it would be difficult to build any sort of UI that would be easy to read any look at. This initially started as a simple grid of numbers with the numbers printed along the sides and bottom being the exact same as their array counterparts.

Listing 6: The original console board layout

```
0|0,2,0,2,0,2,0,2
1|2,0,2,0,2,0,2,0
2|0,2,0,2,0,2,0,2
3|0,0,0,0,0,0,0,0
4|0,0,0,0,0,0,0,0
5|1,0,1,0,1,0,1,0
6|0,1,0,1,0,1,0,1
7|1,0,1,0,1,0,1,0
 ---------------
  0 1 2 3 4 5 6 7
```

It became very apparent that it was very difficult to track what moves were being made especially with AI moves as the move wasn't written by the user. So I decided that the board would have to be completely redesigned, labelling the bottom with letters and reversing the number order at the sides as you would with a normal draughts board and displayed the array values as text equivalents.

Listing 7: The redesigned board

```
8|  -- BP -- BP -- BP -- BP
 |
7| BP -- BP -- BP -- BP --
 |
6|  -- BP -- BP -- BP -- BP
 |
5|  --  --  --  --  --  --  --  --
 |
4|  --  --  --  --  --  --  --  --
 |
3| WP -- WP -- WP -- WP --
 |
2|  -- WP -- WP -- WP -- WP
 |
1| WP -- WP -- WP -- WP --
 ---------------------------
   A  B  C  D  E  F  G  H
```

This greatly improved the appearance of the board and allowed for a more natural input of moves from the user turning commands from *3,4,4,5* to *A3,B4* after applying some number and character manipulation.

Displaying the log was another significant issue encountered, simply printing it to the console would print over the board and so the user would be presented with a list of

moves but no way to visually track it on the board, fortunately with a bit of research I discovered that the position of the console cursor can be moved using *Console.SetCursorPosition(int32,int32)* which meant, since I knew the dimensions of the board on screen, I could print the board and then move the location of the cursor to the right side of the board and print the log on the screen at the same time as the board. The overall result from this is a easy to understand board and log, which while not as nice as a full GUI could be, is still effective.

The AI, as I expected from the start, proved to be the biggest challenge. Not only was it the most time consuming but also the most frustrating and confusing part of the project. Researching AI for board games brought me straight to the minimax algorithm but implementing it was not so easy. Having never worked with recursive functions before getting my head around the concept proved to be difficult, and with such a large number of nodes being generated for moves it was not something could easily step through in the code to see what was happening. So after a few failed attempts at implementation, I fortunately found a website that let you step through the code whilst showing a visualisation of the tree [3]. This made it much easier to understand and after that implementing and understanding the algorithm became far simpler.

Overall I feel like I have achieved more than I anticipated I would in this project even though there were some things that weren't implemented due to time constraints. If I were to do it again I would first make sure I done more research before beginning the project and second plan my time more effectively so that all features could be implemented.

References

[1] S.Wells, Lab 2 - Basic Data Structures, page 3

[2] A L. Aradhya, Minimax Algorithm in Game Theory, "http://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/"

[3] K.Wilkinson, AI Project: Checkers, "http://kwilkins.com/2011/01/ai-project-checkers/"

[4] Minimax Search, "http://will.thimbleby.net/algorithms/doku.php?id=minimax_search"