Algorithms and Data structures Coursework Report

Jonathan Mitchell
40311730@live.napier.ac.uk
Edinburgh Napier University - Algorithms & Data Structures (SET09117)

Keywords – Draughts, Algorithms, Data Structures, Console

1 Introduction

Create the game Draughts The objective of this report is to create the game draughts in a chosen language and a chosen format - console/form/WPF or similar. Particular attention is to be placed on the Data Structures and Algorithms used.

The game, at the very least, should allow for 2 human players to play against each other. Adding "AI" algorithms to allow for Player vs Computer, or even Computer vs Computer are optional additions. Based on development capabilities within the Project time-scale. Additional features such as undo/redo a move, recording and then replaying a previous game are also beneficial.

Finally Creative freedom is encouraged during the development. An individual uniqueness is preferred when developing the game of Draughts.

The application associated with this report was created using a console application. Written in c#

2 Design

The game was based loosely, in the beginning on a noughts & crosses game found on-line [1]. This is where the basis for movement originated from.

2.1 Architecture

The following classes were created:

- Information
- Board
- Movement
- PlayerA
- PlayerB
- PlayerKing
- Error
- Undo
- Skynet

2.2 Data Structures

The data structures used are the following:

• Array: Arrays were used for the static grouping of similar values. As the values in each index is not being added nor removed, just searched through and/or replaced with another value. They are Used for:

Creating the board. Each index of the array is either blank or contains a coordinate with/without a player marker. Each index is then inserted into the "box" created in the console writelines.

An empty array was created, the same size of the board array, to copy the board into this for pushing to the undo stack. This is because pushing an array onto the stack only copies a reference to the array, which can change the stacked array if any index is changed AFTER it being pushed. Copying the board array to the empty array after each move/capture ensures only that move is stored on the stack.

Storing the numbers & letters for new destination coord creation. These are used in algorithms to create new destination coordinates if an opponent piece is to be captured or to detect second/third/fourth capture piece. The computer also uses these to be able to move one of it pieces around the board, as well as capturing opponent pieces.

• Stacks: Used for the undo feature. This is because Stacks provide easily implementation to add (pop) and remove(push) items from them. As stacks can store any type of variable/data structures. They are ideal for pushing an entire copy of the board and then popping it off the stack when needed.

• Lists: Used for computer AI. Lists were used with the AI's available moves. This is due to the lists being dynamic. Items are added/removed from the AI move list as needed. Lists make this process incredibly easy by implementing the add & remove methods.

2.3 Algorithms

2.3.1 Movement

This movement only deals with moving a player piece ONLY. Capturing an opponent piece is described after. Movement algorithm is split up into several sections for human players:

• Checking the player piece start position and chosen end position exist on the board. This is done by using for loops. Cycling through each index of the tiles array comparing if the start position is found and contains a player piece. it then checks destination. If this exists and does not contain a piece belonging to either player:

• preventing sideways movement - done by checking the index[0] of both start and end positions to ensure they are not the same. For both Players. Then:

• preventing backwards movement - the position of string choice & destination [0] are compared to the letter array. When these are the same the letter index position is stored in int variables. When both have been found they are compared. If the destination int < start int, for PlayerA, this returns false as the end position is one or more rows behind start position. Returns false, else true. PlayerB checks if destination int is > start int this will return false as it needs to be less than.

• Limiting forward movement to only available diagonal positions. This algorithm is split into two. The first section, similar to preventing backwards movement, checks that the letter for destination coords is only +1 of the letter for the start coords. This indicates the end position for the piece is the next row down for PlayerA. PlayerB is -1 of these checks. This means it is the next row up. If not, returns false.

If the above is true. Next to be checked is the destination number is +/- 1 of start position. By comparing the destination[1] with choice[1]. If the above is correct; indicates the destination is the forward diagonal, by one row, of starting position. And the move is complete. Player turn will then go to the next player.

Any bool that returns false results in the relevant error message being displayed. And no move is made. The current Player has to choose another move.

2.3.2 Capturing Piece

It was decided that only a maximum of 3 opponent pieces can be captured. this is due to the fact it would allow a player piece to move across the length of the board, most likely resulting in a king piece being created. It is also believed this is an acceptable number of pieces to capture in one turn to result in victory/dominance of the game in play. • Checking for opponent marker occurs when a legal move has been started and the destination contains an opponents piece. Then:

• Capture opponent piece. This is started by getting the New end destination(NewDest) of the player piece. By checking if the destination[1] is less/greater than start[1]. IF less than newDest will be left fwd. IF greater than newDest will be right fwd.

NewDest left: using a for loop with IF statements. the letter of NewDest is gained when destination[0] is the same index position as +1 of letter[i]. NewDest letter = i. Getting the number is when destination[1] - 1 of number[i].NewDest number = i. These are then combined and returned.

NewDest right: This is calculated in the same as if it was left, only difference is the +/- 1 are switched.

Regardless of NewDest being left or right Fwd of opponent piece. this new coord is searched for in the tiles arrays. If it exists and does not contain either player's piece, then the capture is successful. Player piece moves to the NewDest and the opponents piece is removed from the board. And the opponents piece count drops by one. The algorithm next checks if a second piece can be taken (see below). If no more pieces can be taken the other player then has their turn.

Else error appears. And the move does not happen. The current Player has to choose another move.

• Capturing multiple opponent player pieces. This happens when the current player has taken an opponent piece and another opponent piece is detected left/right FwdDiag of the new position of the piece moved to capture an opponent piece. The FwdDiag left/right is found exactly the same as getting the NewDest for first capture. Only there is 2 positions to check. So the method was split in two for this. They are checked if either contains an opponent piece. If either/or both do, a NewDest is got for after the new opponent piece to take. This is then checked to see if it is empty of any player piece. If yes. Second capture successful. This cycle continues until either 5 opponent pieces have been taken or a piece cannot be taken. The other player then takes their turn.

2.3.3 King Pieces

This movement is a partial amalgamation of Player A & B movement and opponent capture pieces: only checking for sideways movement. Backwards/forward movement is different. As a king piece can move both forwards & backwards.

Changes to movement:

Checking if movement is legal 'forward'(going down to the next row of the board) or 'backwards'(going up to the next row of the board) by two separate bools. Depending on the direction of movement, only one of the bools will return true. If there is no opponent piece in the end position. A move is successful as long as it is a legal move.

Capturing opponent piece(s) depends on a few changes. The first piece captured is the same as a normal piece capture. The second piece capture is dependant on which variable (fwdLeft, fwdRight, bckRight & bckLeft) contains a coord that exists on the board AND contains an opponent piece. If the coord behind that exists and is empty. Another piece capture is done. This is repeated up to 5 times. This allows a maximum of 6 opponent pieces to be captured a captured a turn. A sufficient amount to result in a win.

The checks for this is the same as a normal player piece. If all passes the capture(s) are successful else fails.

2.4 Player vs Computer

As this part of the programme begins, the position of the computer's pieces are automatically loaded into the list for the class. This is what the class uses to move its pieces across the board, albeit only in the order they are in the list. As a piece is removed around the board. Its original position is removed from the list and the new position is added. Not replaced, as this could lead to the computer only moving the one piece across the board, until it is either captured/stuck in a corner

As the player captures the computers pieces, they are removed from the list. Similar to what the class does when it moves a piece.

Just as piece capture is automatic for players, it is for the computer. Similar process by replacing the relevant tiles on the board.

The "Skynet" AI inherits from the PlayerB class as its movement is the same as the playerb. This is the same for computer king movement.

2.4.1 Computer

The computer Player AI does work, to an extent: The "AI" does move pieces correctly across the board. There is no algorithm for decision making in the sense of the class counting up each potential move and performing the highest scoring. It just moves each piece as they are listed in the list.

The only addition to the computer is finding the beginning destination for a piece:

Capturing an opponents piece involved finding the forward Diagonal (FwdDiag) for both left & right. The AI first checks the FwdDiag left if there is an opponent piece, if yes attempts to take by checking if the new destination, BEHIND the opponent piece exists and is empty. If these two conditions are met, the opponent piece is taken. Else it checks the right FwdDiag for an opponent piece. Check/capture is the same manner. If neither of these are successful/not possible. The AI attempts to move its piece to the left FwdDiag, if it exists, else attempts right. If both are blocked - opponent pieces that can't be captured, player pieces present, or position doesn't exist. The AI then moves onto the next piece in the list. Process is repeated until a successful capture/-move is completed. Capture piece for AI is also capped at 3 per turn.

Turn then moves to the Human player.

2.5 Undo

This section of the game consisted of a class, implementing a stack, that pushes the full array of the board before a move is made by a player. Then pops it back if the player chose to undo a move. Redo of a move has not been implemented as the design would allow the player to redo a move on their own accord, albeit it would take longer to type in the coords again than just a "Y" for redo move.

A Stack was used for this as it is very simple to implement and can store the full array and return just as easily. Being an ADT, it is dynamic therefore no need to manage the memory needs of this Data structure, it is done automatically.

2.6 Visual enhancements

The colorful.console library was used for the visual enhancements, text colour change for front page & rules section. [2]

3 Enhancements

Firstly. The game development would be done in WPF. Due to the graphic capabilities of WPF, a much better visual representation of the game would be possible. The board, itself, could be created using the Grid & Grid definition in XAML. Each grid spot, 64 in total, could be assigned an index of the player board array. Placing the player pieces, as separate objects, and assigning them to the current array index they are placed upon. Using click event for the mouse, the player can then click each piece and click the destination.

Background code would check they are clicking the correct player piece, based on player turn, and validating whether the move is legal or not. This code would be very similar to that developed. The only real difference would be the position would be based on the array index and not a typed coordinate.

Also, buttons can be implemented for Undo/Redo functions etc. Without the need to ask the user, removing a slight annoyance, they can then just click on the button to perform this action.

WPF also displays the game in a GUI style. Which is more desirable, in the "Modern Age", than a console application.

Second. If sticking to the console application. The array design would be changed. Instead of displaying the coords on the board. Only the pieces would be. BY creating a board with 8 rows & columns. Rows lettered(A-H) & columns numbered(1-8). The player would still input the coords, but they would not appear on the physical board. The computer would search for the corresponding letter/number combination and perform the necessary checks. Ideally there would be an algorithm to detect the current legal move(s) for the current player. Displaying what these are and only accepting, as input, one of these pieces. The destination would then be limited to left or right of this piece. removing need to input destination. If there is the ability to capturing a piece, the first is done automatically, with a choice made available for any future captures.

The above can also be implemented for the AI. Whereby a list is created with all pieces available and their position(s) on the board. Using the above algorithm, a second list can be created that contains only the current legally moveable pieces. Finally adding an algorithm that creates a scoring system for each piece in the legal move list point scored for safe move, capture piece, safe capture etc. the highest score being moved each turn.

Adding the ability for the player to chose which AI to play against (Assuming a second is implemented that is more difficult to beat) would be another potential enhancement. This would mean creating another AI that is different form the first. An easy/normal AI combination?

Adding the ability to record a game would be relatively simple. After each move/undo. The board array would copied to another array and stored in a queue. The queue ADT design of first in first out would mean on a replay it would play each move as it happened. Either using a time delay or pressing a key to play through each move as it happened. After each game is completed, storing the queue onto a list would make it possible to store multiple games. An option in the menu would need to available to watch the repays and select which game you want to see.

With more time, checking over the coding to see if there is a way to break up large section of code into separate methods, may help to make it easier to read as some sections are a little long and more complex than they may need to be.

4 Evaluation

4.1 Critical Evaluation

4.1.1 Features that work well

The AI movement feature is a strong point for this implementation. Being simple and easy to code. The human player can play a full game against it. And even though it uses each piece linearly in the list, which does make it somewhat predictable, it can be a slight challenge as the game progresses and its remaining pieces are more "randomised" from movement.

Capture Detection is well implemented. Split up into functions, this algorithm allows for multiple piece captures, if possible. It has been limited to five captures: logic being any more than this could mean your opponent is pretty much beaten. Also it would probably result in the king piece being created.

Player King implementation is another positive for this build. If a little long, it allows for capturing an opponents piece in any legal direction. By having each next capture piece movement either non-existing coord or a legal move. The need for validation of directions etc. is not needed. Only validation is that the newDest coord created does exist and a legal capture is possible.

The Undo feature works well. As the whole Array is copied before a move is completed, then popped back. This makes for an easy "Reset" for the board, with no complexity.

4.1.2 Features that don't work well

The creation of the board could be done better, in terms of creation. See above enhancements.

The for/foreach loops in use could be shortened. By using above enhancement of legal move list(s) etc. they would only need to be searched through less times than the whole board array each time.

By implementing legal list moves, the code for checking backwards/sideways & forward movement would not be needed as this would be done through an automatic algorithm at the start of each turn. This could potentially increase player time as the player would be presented with the possible pieces to move. Instead of having to figure what is possible each turn.

4.2 Personal Evaluation

From a personal point of view. I enjoyed the challenge of creating this game. Originally overwhelmed, I began to break the problem into the above mentioned steps. And found it to be be manageable. I have learned a lot from this coursework: my understanding of how to use data structures in a real world setting.

The real big issue I had with this is myself. I had to rewrite several classes due to duplicate/redundant variables, poor coding and general failure to conform to good programming practice: keeping for loop variables inclusive. This was due to my failure to realise, at the time, I could split up the piece capture/movement into if statements within the for loops.

I am most proud of my algorithms for creating the coordinate for the new destination during capturing an opponents piece. These were something I had not done before. My problem solving skills flourished here. As I figured out an algorithm to get the new letter & number coord, by creating 2 arrays: one for letters present on the board and one for numbers present on the board.

By checking if the destination second index in comparison to the start coord second index value; the algorithm determines if the new destination coord is going to be left or right of current destination coord. Next part of the algorithm is to compare the first value of endcoord char array with the letter char array using a for loop. When the end coord = letter index - 1, the index value is stored in a string. This will comprise the first element, letter, of the new destination coord. The second element, number, of the new destination is found in a similar fashion: by comparing the second index of destination coords with the number char array and storing the value of number char array when this = destination index 2 - 1. The two values are then concatenated to form the coords of the new destination of the player piece. This is returned for validating an opponents piece can be taken. See Appendix A.

After implementing the AI algorithm, using the foreach loops. It made me realise this is much better than using for loops. Which, I think, was due to using the noughts & crosses game [1] as reference. This uses for loops. I admit, for second piece capture, it was mostly guess & check when it came to which variable for the for-loops in which to use to make the second capture successful and appear on the board correctly.

The implementation of the AI proved to be a bit more difficult than I thought. Initially I didn't fully check the possible FwdDiag movement for each piece the AI was going through. Instead I treated some of the algorithm as though it was a human player and assumed it would know where it wanted to move/capture a piece. This resulted in a few errors/irregularity in movement, until I realised this.

The code also went through several implementations. This was mostly due to the fact there was duplication of variables and unnecessary use of variables. Examples of this include converting a string var into char array var. This is unnecessary as a string is basically a char array. Variable duplication occurred when I created the computer class. I created its own variables to use, this was changed to use the same variables as the player classes, when they all inherited from the Movement base class.

Overall, I learned a lot from this coursework, and if I were to implement it again. I would take the approaches mentioned above in regards to implementing the movement/-capturing pieces.

5 Appendix

5.1 Appendix A: finding new destination Coord

Listing 1: finding new destination coord for capturing opponent piece

```
public string checkEnemyMoveToCapture()
{
if (board.Endcoord[1] < board.Startcoord[1])
{
for (int i = 0; i < board.Letter.Length; i++)
{
if (board.Endcoord[0] == board.Letter[i] − 1)
{
coordL = board.Letter[i];
newL = coordL.ToString();
}
if (board.Destination[1] − 1 == board.Number[i])
{
coordn = board.Number[i];
newN = coordn.ToString();
}
newDest = newL + newN.Trim().ToUpper();
}
}
else if (board.Endcoord[1] > board.Startcoord[1])
{
for (int c = 0; c < board.Letter.Length; c++)
{
if (board.Endcoord[0] == board.Letter[c] − 1)
{
coordL = board.Letter[c];
newL = coordL.ToString();
}
if (board.Destination[1] + 1 == board.Number[c])
{
coordn = board.Number[c];
newN = coordn.ToString();
}
newDest = newL + newN.Trim().ToUpper();
}
}
else
{
newDest = "";
}
return newDest;
}
```

6 Bibliography

References

[1] c sharpcorner., “Tictactoe,” Apr. 2014.

[2] tomakita., “Colorful.console,” 2016.