Algorithms and Data structures Coursework Report

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

Abstract

hello.

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 are also beneficial.

Adding the ability to record the results of games and even record games themselves is another possibility. Finally Creative freedom is encourage 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

2.1 Architecture

The following classes were created:

- Information
- Board
- Movement
- PlayerA
- PlayerB
- PlayerKing
- Error
- Redo/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:

• Stacks: Used for the undo & redo 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. Movement algorithm is split up into several section for human players:

• Checking the player piece position start and chosen end destination exist on the board. This 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 the end coords of both start and end position to ensure they are not the same. For both Players.

• 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 endcoord int < startcoord int, for PlayerA, this returns false as the end position is one row behind start position. Returns false, else true. PlayerB checks if endcoord int is > startcoord 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 endcoord int is only +1 of startcoord int. This indicates the end position for the piece is the next row down for PlayerA. PlayerB is endcoord int -1 of startcoord int.

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 string[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 go to player A or B.

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

• 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 endcoord[1] is less/greater than startcoord[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 endcoord[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:

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 a piece, of either player, then the capture is successful. And the opponents piece count drops by one. And the other player has their turn.

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

• Checking if there is a second opponent marker can be taken. 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 into 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. opponent piece dropped by one and ends player turn.

Else Second capture has failed. And player turn ends with only one piece taken.

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 layer piece. The second piece capture is dependant on which of the above bools was successful. if Fwd bool is true PlayerA second capture method is enacted. If Bck bool is true PlayerB second capture method is enacted.

The checks for for this si 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.

2.4.1 Computer

The computer Player AI does work, to an extent: The ÄI" 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.

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 exits, 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.

Turn then moves to the Human player.

2.5 Undo/Redo

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 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. Either by number/lettering each one individually.

The user would then enter the letter/number of the piece to move, the background code getting the array index of the chosen piece and then checking if it is a legal piece and if it is possible to move. Giving the user an option of which direction to move the piece; If there is such an option. The same would be for capturing opponent pieces/ turning into a king etc. This is the only option to change at the moment.

Third. For loops. Using the above amendment to the array for the board, use of for loops could be reduced. At the very least they can be used with proper practice: the int variables created inside, and solely used by, each for loop. Instead of the current where they are declared global for used throughout the development. This would be the biggest weakness of this game of draughts. Another option would be to implement foreach loops. As seen in the computer class.

Fourth. Checking each player(s) positions on the board and performing a quick algorithm to check which piece can be moved/take an opponent's piece successfully that turn then adding these pieces to a list. This will increase the speed to search the tiles array for a piece to move. As only a select number of pieces would be able to move. Only an input coord of a piece in this list would be accepted. Then just perform the regular legal move checks for its destination coord. The legal move list can also be implemented for the AI. With a random number generator to pick the first piece. Instead of in the order they are in the list.

Fifth. Adding an algorithm that creates a scoring system for each piece in the legal move list (see above) 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 adapting PlayerB class code to allow this.

3 Evaluation

3.1 Critical Evaluation

Some fancy stuff written here

3.2 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 manageable. I have learned a lot from this coursework: my understanding of how to use data structures in a real world setting.

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 end coord 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 coord 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.

The algorithm:

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;
}
```

References

[1] S. Keshav, “How to read a paper,” *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 83–84, July 2007.