Functional Implementation of Minimax

Rick Mccasker | Semester 1, 2019 | CAB402

Insights and Discussion

The F# version initially felt hostile due to the use of strictly no mutable state and values represented as functions. This immutability resulted in higher computation times when dealing with changing variables, the main case being a new state being created for each node, however it made debugging the program significantly easier as a chain of shallow copies of mutable states couldn't be generated resulting in not having to follow a potentially convoluted chain of copies. The concept that many lines of 'functions' replacing variables to achieve some singular result was foreign at first but allowed for an easier time segmenting localised functions and combined with the higher order functions, it allowed for some powerful concise transformations to inputs. However this benefit is offset by the fact that this could easily become a mess of chained functions, even with good coding practices, resulting in failure to scale as well as hybrid or object oriented approaches discussed further in this report.

F# impure felt better to worker with as it was far less restrictive, permitting mutable states. Mutable state allowed for more simpler and more readable code such as the insertions of tokens into the board rather than the convoluted process of rebuilding the list done in the pure version, showing improvements of roughly about 10% for various tests, comparison shown in Figure 1. Admittedly these improvements were significantly lower than predicted and is likely lacking potential improvements. Being able to use for loops was also an advantage as the disadvantage of them being unable to return anything is solved using alterations to exterior mutable variables through each iteration. Mutable state was also tested for local variables in GameOutcome and AB pruning but resulted in worse times.

TEST N.O	F# Pure	F# Impure	C#
748	0.0604651	0.0535026	0.0132368
7485	0.0551433	0.0475349	0.0119002
14236958	0.0552279	0.0483719	0.0111715
159	0.0553619	0.0499365	0.0131069

Figure 1: Snippet of test times averaged over 20 runs

C# by far felt the best to implement and had the best times overall. Of course, it had the benefit of its programming style being far more like the pseudo code so reimplementation of A/B pruning was easier, but the code was far more readable and had helpful functions such as break that reduced useless computation time. Inline functions could be defined as lambda expressions if desired, so this object-oriented style could be seamlessly combined with functional programming for a cleaner split hybrid style than the F# impure version. 0.3263797

Conclusion and Final Comparison

Overall the C# version proved to be the easiest to implement while providing the most efficient and concise code for this project, as can be easily demonstrated in Figures 2 and 3, which shows the same implementation for the pure F# and C# versions. The F# versions both suffered from poor times in comparison and appeared to have degrading code quality the more functions that were introduced, which could result in poor scaling. However, despite this its stricter programming style allows for easier to debug and overall more reasonable code, as mentioned above, and the use of tools such as the pipeline operator allows for seamless transformations in an easy to understand format.

```
let ApplyMove (oldState:GameState) (move: Move) : GameState =
    let rec InsertEle oldArr ele i =
        match i, oldArr with
        | i, [] -> []
        | 0, head::tail -> ele::tail
        | i, head::tail -> head::InsertEle tail ele (i-1)
    let GetCurrentPlayerToken player =
        match player with
        | Nought -> "O"
        | Cross -> "X"
    let CalcCoord (move: Move) size =
        move.row*size+move.col
    let ChangePlayer player =
        match player with
        | Nought -> Cross
        | Cross -> Nought

    let newBoard = InsertEle oldState.board (GetCurrentPlayerToken oldState.currentTurn) (CalcCoord move oldState.boardSize)
    let newState = {currentTurn= ChangePlayer oldState.currentTurn; boardSize = oldState.boardSize; board = newBoard}
    newState
```

Figure 2: Pure F# implementation of ‘ApplyMove’

```
public Game ApplyMove(Game game, Move move)
{
    Player newPlayer;
    string token;
    string[,] newBoard = game.Board.Clone() as string[,];

    if (game.Turn == Player.Nought)
    {
        newPlayer = Player.Cross;
        token = "O";
    }
    else
    {
        newPlayer = Player.Nought;
        token = "X";
    }
    newBoard[move.Row, move.Col] = token;
    Game newGame = new Game(newPlayer, game.Size, newBoard);
    return newGame;
}
```

Figure 3: C# implementation of ‘ApplyMove’

Statement of Completeness

All functions implemented as requested.

F# Impure has minimal speed increase and could not be solved effectively.

TicTacTioeViewModelTests.Test fails for as will not set human turn appropriately. All other tests pass as expected.