Edinburgh Napier UNIVERSITY

Algorithm and Data Structure - Tic Tac Toe Game Report

Louis Boursier
40404293@napier.ac.uk
Edinburgh Napier University - SET08122

Abstract

This document is a report for the first coursework in the algorithm and data structure module [**?**]. For this coursework, we had to build a console user interface tic tac toe game using the C language and standard library. The objective is to demonstrate our understanding of the basics data structure, and evaluate the performance of an algorithm based on the size of a problem input, but also to demonstrate a working knowledge of a relevant data structures API and design and develop and evaluate data structures and algorithms.

Keywords – Louis, Boursier, Game, Tic Tac Toe, Algorithm, Data Structure, AI

1 Introduction

I had to design the algorithm of the game and define what structure I will need to implement it. I had to prioritized the most appropriate data structure and algorithm for the game, based on a complexity analysis and game making and artificial intelligence knowledge. I also enhance the basic game with a expandable grid game in terms of height, width, and pawns in sequence to win the game. It is possible to play against another player or the computer. The player can undo and redo the moves. The player can also save the game and load it later to replay it and then continue it.

2 Design

2.1 Game loop design

The game follows a traditional game loop that can be described the following way:

Listing 1: Game loop

```
1 game = askInfo()
2 do
3     displayInfo(game)
4     currentPlayer = setPlayer(game)
5     do
6         move = askInfo()
7     while isValidMove(move) = False
8     play(game, move)
9 while (winner = getWinner()) != NULL
10 displayInfo(winner)
```

2.2 Data structures

Listing 2: Player

```
1 struct Player {
2     int id;
3 };
```

I chose a Struct for the player structure, even if it only contains an integer for identifying the player. This is for two reasons: first, have a more readable code in terms of syntax, and second, plan ahead for an extension of the player information (a name for example).

Listing 3: Position

```
1 struct Position {
2     int x;
3     int y;
4 };
```

For the position of the pawn, I have simply created a Struct with two integers for the X and Y coordinates. That way, I have got all the information in one variable instead of having to deal with two separate integers.

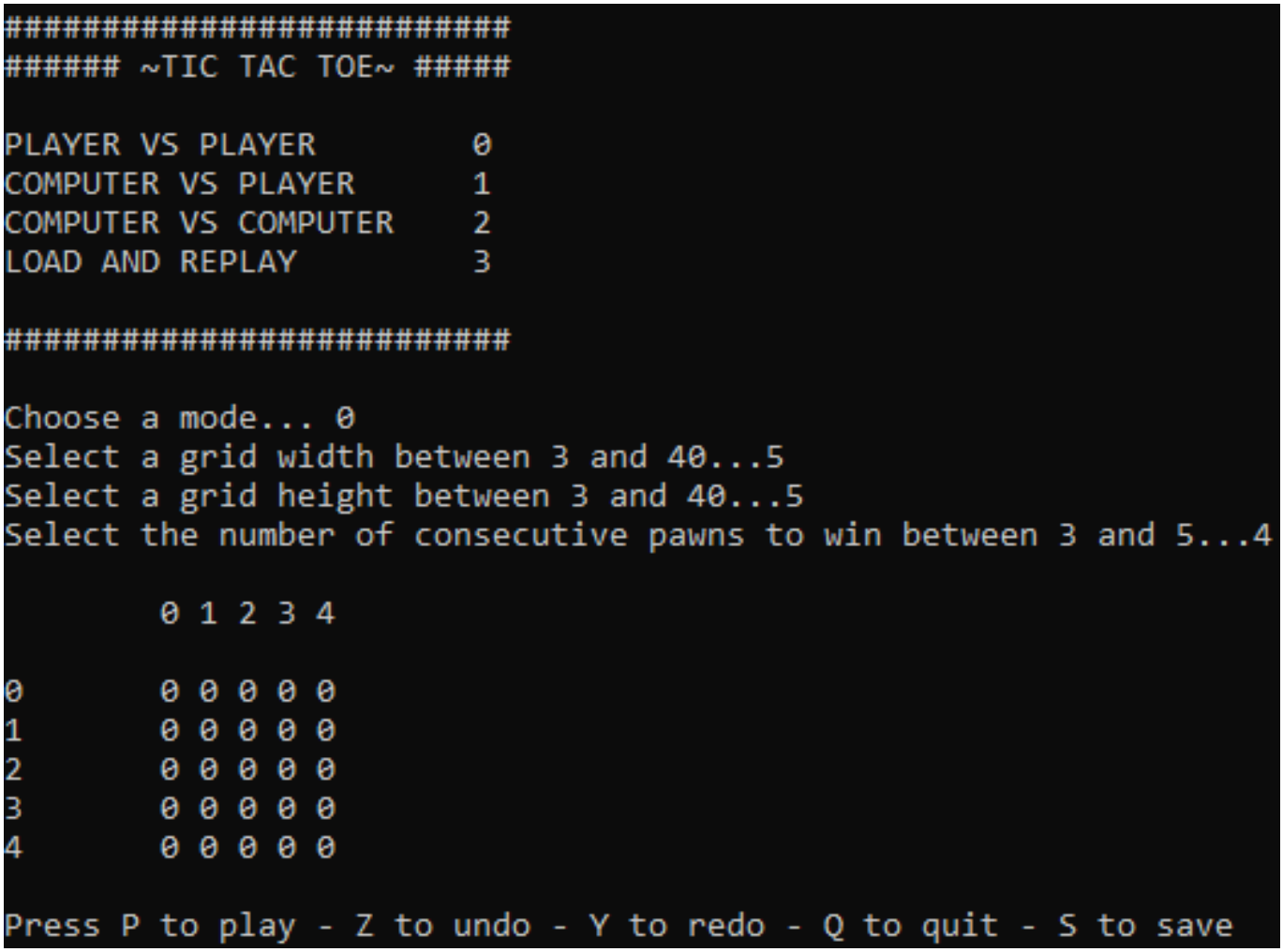

Figure 1: **Game** - A player versus player game with a 5 by 5 grid and 4 pawns for winning

Listing 4: Content

```
1 struct Content {
2     struct Position* position;
3     struct Player* player;
4 };
```

This Struct represents a move played by a player. Even though this Struct is not absolutely necessary, I have created it because it allowed me to write the code more easily, and it is also more readable. Moreover, having a Struct containing the position and the player can allow to create a game with more than two players in an unordered sequence of playing. For the content, I have created a Struct that contains two other Struct: the Player Struct that we saw before and the Position Struct.

Listing 5: Node

```
1 struct DoubleLinkedNode {
2     struct DoubleLinkedNode* right;
3     struct DoubleLinkedNode* left;
4     struct Content* content;
5 };
```

This Struct is a node used for the doubly linked list. Because it is a doubly linked list, it contains a pointer to its previous and next elements. It also contains a pointer to the move that has been played, thanks to the Content Struct. I have used a doubly linked list because I already knew that I will have to implement an undo an redo functionality. And this data structure is adequate for this usage because undo and redo can correspond to previous and next pointers.

Listing 6: Piece

```
1 char pawn;
```

I chose to use a char to represent the pieces. Because the char is one of the lightest variable in terms of memory (1 byte). I need a light variable because the game board will be a 2 dimensional grid of char. And because the artificial intelligence algorithm will instantiate a lot of those arrays to predict the future moves and chose the best one, I need those arrays to be as light as possible. Nevertheless, the char allows me to have 256 different pawns, hence 255 players (one pawn for the empty state) if I need it, and it should be enough.

Listing 7: Game Board

```
1 char grid[height][width];
```

I chose an array to store the game pieces. Indeed, it is perfectly suited for this need. The access of the elements is fast and easy. And because the size of the game board is fixed at the beginning of the game, I will have no problem dealing with its constant size.

2.3 Algorithms

There are only two algorithms to explain in detail. The winning or draw detection and the artificial intelligence. Let us start by the winning or draw detection algorithm.

Listing 8: Win detection

```
1 int getWinner(int height, int width, int winNumber, char grid[height][width]){
2
3   int lastPawnPlayer = -1; // Stores the id of the current player pawns in sequence
4   int consecutivePawns = 0; // Stores the number of pawns in sequence
5   int totalPawnsToFinish = height * width; // The max number of pawns that can be played to detect a draw
6   int pawnsCount = 0; // The counter of pawns played to detect a draw
7
8   for(int i=0 ; i<height ; i++){ // Check the horizontal wins
9     consecutivePawns = 0; // reset to 0 on each new lines
10    lastPawnPlayer = -1;
11    for(int j=0 ; j<width ; j++){
12      int currentPlayer = grid[i][j];
13      if(currentPlayer != lastPawnPlayer){
14        lastPawnPlayer = currentPlayer;
15        consecutivePawns = 0;
16      }
17      if(currentPlayer != CELL_EMPTY && ++pawnsCount == totalPawnsToFinish){
18        return 0; // Draw
19      }
20      if(currentPlayer != CELL_EMPTY && ++consecutivePawns == winNumber){
21        return currentPlayer;
22      }
23    }
24  }
25
26  for(int i=0 ; i<width ; i++){ // Check the vertical wins
27    consecutivePawns = 0;
28    lastPawnPlayer = -1;
29    for(int j=0 ; j<height ; j++){
30      int currentPlayer = grid[j][i];
31      if(currentPlayer != lastPawnPlayer){
32        lastPawnPlayer = currentPlayer;
33        consecutivePawns = 0;
34      }
35      if(currentPlayer != CELL_EMPTY && ++consecutivePawns == winNumber){
36        return currentPlayer;
37      }
38    }
39  }
40
41  for(int i=0 ; i<height ; i++){ // Check the crossed wins bottom left to top right, upper part
42    consecutivePawns = 0;
43    lastPawnPlayer = -1;
44    for(int j=i ; j>=0 ; j--){
45      int currentPlayer = grid[j][i-j];
46      if(currentPlayer != lastPawnPlayer){
47        lastPawnPlayer = currentPlayer;
48        consecutivePawns = 0;
49      }
50      if(currentPlayer != CELL_EMPTY && ++consecutivePawns == winNumber){
51        return currentPlayer;
52      }
53    }
54  }
55
56  for(int i=width-1 ; i>=0 ; i--){ // Check the crossed wins bottom left top right, lower part
57    consecutivePawns = 0;
58    lastPawnPlayer = -1;
59    for(int j=0 ; j<height && j+width-1-i<width; j++){
60      int currentPlayer = grid[height-1-j][j+width-1-i];
61      if(currentPlayer != lastPawnPlayer){
62        lastPawnPlayer = currentPlayer;
```

```
63              consecutivePawns = 0;
64          }
65          if(currentPlayer != CELL_EMPTY && ++consecutivePawns == winNumber){
66              return currentPlayer;
67          }
68      }
69  }
70
71  for(int i=0 ; i<width ; i++){ // Check the crossed wins top left to bottom right, upper part
72      consecutivePawns = 0;
73      lastPawnPlayer = -1;
74      for(int j=0 ; j<height && j<=i ; j++){
75          int currentPlayer = grid[j][width-i+j-1];
76          if(currentPlayer != lastPawnPlayer){
77              lastPawnPlayer = currentPlayer;
78              consecutivePawns = 0;
79          }
80          if(currentPlayer != CELL_EMPTY && ++consecutivePawns == winNumber){
81              return currentPlayer;
82          }
83      }
84  }
85
86  for(int i=0 ; i<height-1  ; i++){ // Check the crossed wins top left to bottom right, lower part
87      consecutivePawns = 0;
88      lastPawnPlayer = -1;
89      for(int j=0 ; j<width && j<=i ; j++){
90          int currentPlayer = grid[height-i+j-1][j];
91          if(currentPlayer != lastPawnPlayer){
92              lastPawnPlayer = currentPlayer;
93              consecutivePawns = 0;
94          }
95          if(currentPlayer != CELL_EMPTY && ++consecutivePawns == winNumber){
96              return currentPlayer;
97          }
98      }
99  }
100 return -1; // No winner and no draw
101 }
```

This function returns -1 if there is no winner, 0 if there is a draw, or the id of the winner otherwise. It first checks the horizontal wins, then the verticals, then crossed wins from bottom left to top right (upper and lower part) and finally the crossed wins from top left to bottom right (upper and lower part). It browses the cells on the way defined previously, and when it encounters the same pawn that it encountered before, it increments the counter of the player's pawn. Otherwise, the counter is reset to 0. If, after incrementing the counter, the counter is equal to the number of pawns in sequence to win, then the player won and we return its id. We know if there is a draw if all the cells have been played. The way this function is written allow the game to detect wins or draws on every rectangular shaped boards.

Listing 9: Min Max

```
1  int minMax(int consecutivePawnsForWin, int height, int width, char grid[height][width], int depth, _Bool isMax){
2      int winnerId = getWinner(height, width, consecutivePawnsForWin, grid);
3      int moveValue = 0;
4      if(depth == 0 || winnerId != -1){ // If we reach the max depth search or a final game state
5          if(winnerId == 0){
6              return - depth; // Draw
7          }else if(winnerId == CELL_PAWN_B){
8              return (-100 - depth); // Opponent's win, we prefer losing later than sooner
9
10         }else{
11             return (100 + depth); // Current player's win (AI), we prefer wining sooner than later
12
13         }
14     }
15     char newGrid[height][width]; // Clones the existing grid to a new one
16     for(int line=0 ; line<height ; line++){
17         for(int col=0 ; col<width ; col++){
18             newGrid[line][col] = grid[line][col];
19         }
20     }
21     if(isMax){ // Max
22         moveValue = -200; // Ensures that a move will be selected
23         for(int line=0 ; line<height ; line++){
24             for(int col=0 ; col<width ; col++){
25                 if(isMovePossible(height, width, newGrid, line, col)){ // Tests all possible moves
26                     newGrid[line][col] = CELL_PAWN_A; // Player's turn
27                     int tmp = minMax(consecutivePawnsForWin, height, width, newGrid, depth-1, 0);
28                     if(tmp>moveValue){ // Selects the max value because we want to play the best move
29                         moveValue = tmp;
30                     }
31                     newGrid[line][col] = CELL_EMPTY; // Undo the move
32                 }
33             }
```

```
34        }
35        return moveValue;
36    }else{ // Min
37        moveValue = 200; // Ensures that a move will be selected
38        for(int line=0 ; line<height ; line++){
39            for(int col=0 ; col<width ; col++){
40                if(isMovePossible(height, width, newGrid, line, col)){ // Tests all possible moves
41                    newGrid[line][col] = CELL_PAWN_B; // Opponent's turn
42                    int tmp = minMax(consecutivePawnsForWin, height, width, newGrid, depth-1, 1);
43                    if(tmp<moveValue){ // Selects the min value because the opponent will play the worst move for us
44                        moveValue = tmp;
45                    }
46                    newGrid[line][col] = CELL_EMPTY; // Undo the move
47                }
48            }
49        }
50        return moveValue;
51    }
52    return 0; // Never reached but here for the int return of the function
53 }
```

The Min Max algorithm is a simple search algorithm for a game played in sequence with a define set of actions and winning conditions. We just assume that the opponent will play the best move, hence the worst for us: this is for the Min. And we want to play the best move: this is for the Max. This algorithm takes the current state of the game, and for each possible move, select the one with the bigger score. This is the best move we have calculated. Actually, we call the max function for each of the possible moves. And max will recursively call the min function and vice versa. This is because it represents a simulation of the game where it is our turn to play and then the opponents turn etc... The recursive calls end when we reach a final state of the game. A this stage, the score is then recursively returned (for example 1 for a win, 0 for a draw and -1 for a lose). But when the grid size gets bigger than 3 by 3 or when the set of action is too large, we can set a maximum depth of recursive call, where we evaluate the board. Even if there is no winner, we can still compute a score by looking for good positions and advantages. For example, if we have two aligned pawns that are not blocked, this should returns a better score than having only single scattered pawns.

3 Enhancements

With more time, I would have implemented a way of saving multiple games and not only one. I would have allow the computer to play against itself. And I would also have added an alpha beta pruning to the min max algorithm. I would like to create a grid with different shapes and create a game with more than two players, with an on-line mode. Finally, I would have look for a smarter AI, because this one is weak on big grid. I may have look into heuristics and focus on the promising cells only and not check all the cells.

4 Critical Evaluation

In this section I will analyse the main function of the game and their complexity.

4.1 Display grid

Because I display the grid cell by cell, the complexity is directly proportional to the size of the grid. So the complexity is pretty bad because it is O(n) with n the number of cells, but because this function is only called once per loop, and the grid will never be that big, it is not problem in practice.

4.2 isWinner function

As for the display grid function, this function is proportional to the size of the grid, but because we need to browse the grid 6 times (see the upper explanations) it is longer in practice even though its complexity is still O(n).

4.3 Play a move

The doubly linked list works on a LIFO stack way. It means that the function pushNodeTop() works in a constant O(1) time because we always have the pointer on the last element and we don't need to browse all the list but just push to the top.

```
1 void pushNodeBottom(struct DoubleLinkedNode **node, struct Content *newValue){ // Inserts a new node in a FIFO stack way
2     struct DoubleLinkedNode *newNode = malloc(sizeof(struct DoubleLinkedNode));
3     newNode->content = newValue;
4     newNode->right = NULL;
5     newNode->left = NULL;
6     if(*node != NULL){
7         newNode->left = *node; // We connect the new element to the old one
8         (*node)->right = newNode; // And we connect the old one to the new one
9     }
10    *node = newNode; // The pointer now points to the last element
11 }
```

4.4 Save game

The saving of the game works in a O(n) time with n the number of nodes, because we need to browse all the moves from last to first and write them to a file. The complexity is bad but as before it is not a problem in practice because the list is never that big, and the save is only used occasionally when the player wants it.

4.5 Load game

The loading of the game loads the doubly linked saved in game. So pushNodeBottom() works the same way as the pushNodeTop() function for adding a new node, except that it works in a FIFO stack way because the save file is written from the last node to the first, and we need to create the list in the good sequence. Anyway, the complexity is the same O(n).

4.6 Undo and Redo with the doubly linked list

Because we only move the head pointer to the previous or next move, the complexity is an excellent O(1).

```
1 // Moves the pointer of the head of the stack to the previous element without removing any node
2 void previousNode(struct DoubleLinkedNode **node){
3     if(*node != NULL && (*node)->right != NULL){
4         *node = (*node)->right;
5     }
6 }
7 // Moves the pointer of the head of the stack to the next element without removing any node
8 void nextNode(struct DoubleLinkedNode **node){
9     if(*node != NULL && (*node)->left != NULL){
10        *node = (*node)->left;
11    }
12 }
```

4.7 Min Max

Here is the weak point of the game in terms of complexity and computing resources. Because of the nature of the algorithm, the resources needed for computing it quickly explodes with the size of the possible moves. Indeed, it creates a tree of all possible moves from a game state until a final state or depth.

With n the number of possible moves and considering that if a move is played only one move is withdrawn:

$$possibleMoves = \sum_{n=0}^{depth} n - i * (n - i - 1) \tag{1}$$

$$possibleMoves = n! * (n + 1)!/2 \tag{2}$$

$$O(n) = n! \tag{3}$$

5 Personal Evaluation

Having already implemented a chess game [**?**] with Min Max and Alpha Beta pruning in C with a doubly linked list for undo and redo, I mostly have to recall how I did it three years ago. But I also learnt to analyse the complexity of the algorithm. This allowed me to formally explain the complexity and weakness of the Min Max algorithm (even with the pruning).