Kyle Seifert

Comp Sci 2200 – 101

Gosnell

12/8/2023

Solving Tic-Tac-Toe Using Deterministic Finite Automata

Tic-Tac-Toe is a relatively simple game that most people can consistently beat without having to study and think about their moves all that much. However, it is more of a challenge to “solve” Tic-Tac-Toe in an efficient and elegant way. This means accounting for every possibility and finding the move that is most likely to win (or tie at best) from that position. While it would be possible to simply find the most optimal move given every set of previous moves, that would be wildly inefficient and require much more data than needed. Thus, the challenge is in finding a way to break down the mechanics of the game and solve it using the simplest structure.

# Simplifying Tic-Tac-Toe

Tic-Tac-Toe has Since there are 9 squares and each one fills up after each move, the naïve answer is that there are 9! = 362,880 different games of Tic-Tac-Toe. When accounting for moves that result in a win and end the game early, the best estimate is 26,830 possible games (Schaefer). Creating a state machine for this many possible games would be tedious and slow, so there must be some way to reduce the possible outcomes. One way to reduce the number of outcomes is to group moves that are similar symmetrically or rotationally. Below are examples of moves that would be rotationally or symmetrically similar:

A game with red and blue xs and arrows

Description automatically generatedBy considering states like these as one state, we can reduce the number of possible games from 26,830 to 765. This number is far more manageable and does not account for games where the player makes obviously wrong moves. Using these strategies of simplification, it is possible to solve Tic-Tac-Toe in a relatively readable and concise way.

# Solving Tic-Tac-Toe

Moving first should be simpler to figure out given that we can force only one starting move. There are three types of first moves: edge, corner, or center. Starting on the edge is the worst possible move, since if the opponent plays properly they can always force your loss or draw from the starting position. Corner and center are both strong choices that can lead to a win or, at worst, draw when played properly. Between these, center is the more efficient choice because, when considering the properties of similarity, there are only two continuations (edge or corner) versus the five continuations of a corner start (middle, adjacent side, adjacent corner, opposite corner, opposite side). From each of these two possibilities, it is relatively easy to map out the rest of the game. If the opponent moves on an edge tile, ideal play results in a win every time. Below is a diagram of the possible continuations when the opponent responds by moving in an edge tile:A screenshot of a game

Description automatically generated

The case where the opponent moves in an edge tile requires only 1 start board state (with the initial center move), 3 winning board states, and 7 interim board states (the 5 featured along with 2 for whether each path is blocked in the second continuation). When the opponent moves in a corner spot, a win is not guaranteed. However, it is possible to at least draw every time when played correctly. Below is a diagram of the possible continuations when the opponent responds by moving in a corner tile:

A screenshot of a game

Description automatically generatedThe case where the opponent moves in a corner tile requires the same 1 start board state, 5 winning board states, and 15 interim board states. Altogether, starting in the center tile results in 31 board states, 8 of which are winning boards states and 1 of which is the start state. With all this work done, it is possible to make the most efficient possible machine to solve Tic-Tac-Toe.

# Choosing a Method

There is no perfect structure to fit the Tic-Tac-Toe solution in. The method that makes the most sense seems to be, based on my research, a multi-tape Turing Machine. When looking towards examples on the subject, I found a paper on the topic which went far beyond the scope of what we’ve learned in class and featured transitions I couldn’t make sense of. A citation for the paper is in my references for those curious (Garg). Based on what we’ve learned in class, I believe that the best choice would be a nondeterministic finite automaton. While possibly not as elegant as the multi-tape Turing Machine, the limited scope provided by forcing the machine to start allows for a relatively efficient NFA. An NFA is a stronger choice than a DFA due to its ability to use the empty character. NFAs also remove the need to add transitions for invalid game input such as spaces that are already taken.

# Nondeterministic Finite Automaton Representation

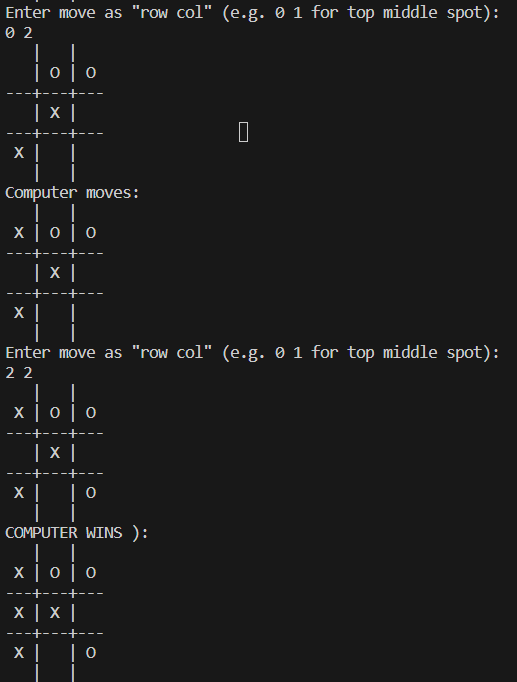
The NFA required for mapping Tic-Tac-Toe is possible to create in the same way that we have been practicing in class, but only if the naïve approach ignoring similarity is used. Software like JFLAP isn’t capable of distinguishing qualities like “opposite corner” and “adjacent edge”. Although it is not testable with any software, a portion of what the NFA would look like, with descriptors for player moves as transitions and a single possible board layout to represent each state, is featured below:

A screenshot of a game

Description automatically generated

To use the principles of NFAs to solve Tic-Tac-Toe, I must make my own program to display and play Tic-Tac-Toe using C++. The research I’ve done to reduce the number of distinct states necessary will allow me to code a relatively efficient Tic-Tac-Toe playing program. The code for this program is attached at the end of this report, but below are screenshots of accurate example output:

A screenshot of a computer program

Description automatically generated

Understanding the different ways to group moves allowed me to make my code more concise by finding patterns in input and output for each move. This allowed me to check multiple qualities or assign new moves in statements more succinct than what simply mapping out every move alone would yield. For example, the first check for whether a corner or edge move was made can be made by adding the row and col values and figuring out if the sum is even or odd, respectively. Since this is just a proof of concept, it assumes perfect input, but it would not be hard to add features to make sure that the user inputs valid moves. Using programming techniques such as including more function calls could make this code more efficient by removing repetitions of routines that get repeated, like requesting user input or searching for a diagonal/edge win. However, with only 40 conditional statements, 15 of which could be reduced by function calls, this method is still far more effective than brute force mapping all possible games of Tic-Tac-Toe.

# Conclusions

While to most people Tic Tac Toe is a simple children’s game, it has many hidden intricacies that can make its solution daunting. Using the principles of state-based automata it is possible to simplify and, relatively efficiently, create a machine capable of playing the best move possible given any set of moves. While there is still room for improvement, thinking about the game in terms of a limited number of states allowed me to group games together and make an NFA which I was able to turn into a working C++ program to win or tie Tic-Tac-Toe every time. Doing so has strengthened my skills and given me insight regarding attacking problems like a true computer scientist.

References

Schaefer, Steve (2002). "MathRec Solutions (Tic-Tac-Toe)". Mathematical Recreations. Archived from the original on 28 June 2013. Retrieved 18 September 2015.

Garg, S.L., Songara, D., & Maheshwari, S. (2017). The winning strategy of Tic Tac Toe Game model by using Theoretical Computer Science. *2017 International Conference on Computer, Communications and Electronics (Comptelix)*, 89-95.

<https://github.com/kasn33/TicTacToe>

#include <iostream>

using namespace std;

//struct for TicTacToe board

struct TicTacToe{

    char board[3][3] = {{' ',' ',' '},{' ',' ',' '},{' ',' ',' '}};

};

//print out properly formatted board

void printBoard(TicTacToe tic){

    cout<< "   |   |" << endl;

    cout<< " " << tic.board[0][0] << " | " << tic.board[0][1] << " | " << tic.board[0][2] << endl;

    cout<< "---+---+---" << endl;

    cout<< " " << tic.board[1][0] << " | " << tic.board[1][1] << " | " << tic.board[1][2] << endl;

    cout<< "---+---+---" << endl;

    cout<< " " << tic.board[2][0] << " | " << tic.board[2][1] << " | " << tic.board[2][2] << endl;

    cout<< "   |   |" << endl;

}

int main(){

    //variables used

    int row, col; //user input variables

    bool adjacentEdge = false;

    //start and display game

    TicTacToe game;

    cout<< "Game Starting!" << endl;

    printBoard(game);

    //play first computer move (always same)

    cout<<"Computer moves first:" << endl;

    game.board[1][1] = 'X';

    printBoard(game);

    //initial user input

    cout<< "Enter move as \"row col\" (e.g. 0 1 for top middle spot):" << endl;

    cin>>row>>col;

    game.board[row][col] = 'O';

    printBoard(game);

    //nested if statements for every continuation (using requirements that group similar statements)

    if((row+col)%2 == 0){//check if corner

        game.board[(row+2)%4][(col+2)%4] = 'X';//play opposite corner

        cout<<"Computer moves:" << endl;

        printBoard(game);

        cout<< "Enter move as \"row col\" (e.g. 0 1 for top middle spot):" << endl; //input

        cin>>row>>col;

        game.board[row][col] = 'O';

        printBoard(game);

        if((row+col)%2 == 0){//check if corner

            if(game.board[(row+2)%4][col] == 'O')//find blocking move

                game.board[1][col] = 'X';

            else

                game.board[row][1] = 'X';

            cout<<"Computer moves:" << endl;

            printBoard(game);

            cout<< "Enter move as \"row col\" (e.g. 0 1 for top middle spot):" << endl; //input

            cin>>row>>col;

            game.board[row][col] = 'O';

            printBoard(game);

            if(col==1 && game.board[(row+2)%4][col] == 'X'){//check if blocking column win (no need to check middle space)

                if(game.board[row][col+1] == 'X')

                    game.board[row][col-1] = 'X';

                else

                    game.board[row][col+1] = 'X';

            }

            else if(row==1 && game.board[row][(col+2)%4] == 'X'){//check if blocking row win (no need to check middle space)

                if(game.board[row+1][col] == 'X')

                    game.board[row-1][col] = 'X';

                else

                    game.board[row+1][col] = 'X';

            }

            else{//if unblocked, find and use winning move

                for(int i = 0; i<3 ; i+=2){

                    if(game.board[i][1] == 'X')

                        game.board[(i+2)%4][1] = 'X';

                    if(game.board[1][i] == 'X')

                        game.board[1][(i+2)%4] = 'X';

                }

                cout<< "COMPUTER WINS ):" << endl;//announce win and exit so don't do remaining operations

                printBoard(game);

                exit(0);

            }

            cout<<"Computer moves:" << endl;

            printBoard(game);

            cout<< "Enter move as \"row col\" (e.g. 0 1 for top middle spot):" << endl; //input

            cin>>row>>col;

            game.board[row][col] = 'O';

            printBoard(game);

            for(int i = 0; i<3; i++){//check for and fill last empty space

                for(int j = 0; j<3; j++){

                    if(game.board[i][j] == ' ')

                        game.board[i][j] = 'X';

                }

            }

            cout<< "TIE GAME /:" << endl;//announce tie and exit to not interfere with other continuances

            printBoard(game);

            exit(0);

        }

        for(int i=0; i<2; i++){//check if adjacent edge and make next move to block within loop if so

            if(game.board[i][col] == 'O' && game.board[i+1][col] == 'O'){

                game.board[(i+2)%3][col] = 'X';

                adjacentEdge = true;

            }

            if(game.board[row][i] == 'O' && game.board[row][i+1] == 'O'){

                game.board[row][(i+2)%3] = 'X';

                adjacentEdge = true;

            }

        }

        if(adjacentEdge){

            cout<<"Computer moves:" << endl;

            printBoard(game);

            cout<< "Enter move as \"row col\" (e.g. 0 1 for top middle spot):" << endl; //input

            cin>>row>>col;

            game.board[row][col] = 'O';

            printBoard(game);

            for(int i=0; i<3; i+=2){

                if(game.board[i][0] == 'X' && game.board[i][1] == ' ' && game.board[i][2] == 'X'){//check for horizontal win

                    game.board[i][1] = 'X';

                    cout<< "COMPUTER WINS ):" <<endl;

                    printBoard(game);

                    exit(0);

                }

                if(game.board[0][i] == 'X' && game.board[1][i] == ' ' && game.board[2][i] == 'X'){//check for vertical win

                    game.board[1][i] = 'X';

                    cout<< "COMPUTER WINS ):" <<endl;

                    printBoard(game);

                    exit(0);

                }

                for(int j=0; j<3; j+=2){

                    if(game.board[i][j] == 'X' && game.board[(i+2)%4][(j+2)%4] == ' '){//check if diagonal win possible

                        game.board[(i+2)%4][(j+2)%4] = 'X';

                        cout<< "COMPUTER WINS ):" <<endl;

                        printBoard(game);

                        exit(0);

                    }

                }

            }

        }

        else{//opposite edge case

            for(int i=0; i<3; i+=2){

                for(int j=0; j<3; j+=2){

                    if(game.board[i][j] == ' ' && i != row && j != col)//find opposite corner and play there

                        game.board[i][j] = 'X';

                }

            }

            cout<<"Computer moves:" << endl;

            printBoard(game);

            cout<< "Enter move as \"row col\" (e.g. 0 1 for top middle spot):" << endl; //input

            cin>>row>>col;

            game.board[row][col] = 'O';

            printBoard(game);

            for(int i=0; i<3; i+=2){

                if(game.board[i][0] == 'X' && game.board[i][1] == ' ' && game.board[i][2] == 'X'){//check for horizontal win

                    game.board[i][1] = 'X';

                    cout<< "COMPUTER WINS ):" <<endl;

                    printBoard(game);

                    exit(0);

                }

                if(game.board[0][i] == 'X' && game.board[1][i] == ' ' && game.board[2][i] == 'X'){//check for vertical win

                    game.board[1][i] = 'X';

                    cout<< "COMPUTER WINS ):" <<endl;

                    printBoard(game);

                    exit(0);

                }

                for(int j=0; j<3; j+=2){

                    if(game.board[i][j] == 'X' && game.board[(i+2)%4][(j+2)%4] == ' '){//check if diagonal win possible

                        game.board[(i+2)%4][(j+2)%4] = 'X';

                        cout<< "COMPUTER WINS ):" <<endl;

                        printBoard(game);

                        exit(0);

                    }

                }

            }

        }

    }

    else{//if player plays on edge

        if(row==1)

            game.board[0][(col+2)%4] = 'X';

        else

            game.board[(row+2)%4][0] = 'X';

        cout<<"Computer moves:" << endl;

        printBoard(game);

        cout<< "Enter move as \"row col\" (e.g. 0 1 for top middle spot):" << endl; //input

        cin>>row>>col;

        game.board[row][col] = 'O';

        printBoard(game);

        if(game.board[(row+2)%4][(col+2)%4] == 'X'){//if diagonal win blocked

            for(int i=0; i<2; i++){//find spot to block

                if(game.board[i][col] == 'O' && game.board[i+1][col] == 'O'){

                    game.board[(i+2)%3][col] = 'X';

                }

                if(game.board[row][i] == 'O' && game.board[row][i+1] == 'O'){

                     game.board[row][(i+2)%3] = 'X';

                }

            }

            cout<<"Computer moves:" << endl;

            printBoard(game);

            cout<< "Enter move as \"row col\" (e.g. 0 1 for top middle spot):" << endl; //input

            cin>>row>>col;

            game.board[row][col] = 'O';

            printBoard(game);

            for(int i=0; i<3; i+=2){

                if(game.board[i][0] == 'X' && game.board[i][1] == ' ' && game.board[i][2] == 'X'){//check for horizontal win

                    game.board[i][1] = 'X';

                    cout<< "COMPUTER WINS ):" <<endl;

                    printBoard(game);

                    exit(0);

                }

                if(game.board[0][i] == 'X' && game.board[1][i] == ' ' && game.board[2][i] == 'X'){//check for vertical win

                    game.board[1][i] = 'X';

                    cout<< "COMPUTER WINS ):" <<endl;

                    printBoard(game);

                    exit(0);

                }

                for(int j=0; j<3; j+=2){

                    if(game.board[i][j] == 'X' && game.board[(i+2)%4][(j+2)%4] == ' '){//check if diagonal win possible

                        game.board[(i+2)%4][(j+2)%4] = 'X';

                        cout<< "COMPUTER WINS ):" <<endl;

                        printBoard(game);

                        exit(0);

                    }

                }

            }

        }

        else{//if diagonal win not blocked

            for(int i=0; i<3; i+=2){

                for(int j=0; j<3; j+=2){

                    if(game.board[i][j] == ' ' && game.board[(i+2)%4][(j+2)%4] == 'X')

                        game.board[i][j] = 'X';

                }

            }

            cout<< "COMPUTER WINS ):" <<endl;

            printBoard(game);

        }

    }

    return 0;

}