Simulate Tic-tac-toe using TM

Pranjal Singh(UG201314013), Kartik Singh(UG201310015)

Abstract

Simulation of Tic-tac-toe using Turing Machines. We are given input strings from 2 players (one representing 0 & the other representing X). We aim to check whether ends in stalemate or if someone emerges as a victor. For this, we are using Tree Data Structure & 2-tape Turing Machine. By using Tree Data structure, we aim to check for all the possible states that may emerge from a particular state.

1 MOTIVATION

The main motivation behind doing this project is to simulate one of our favorite childhood games – tic tac toe. This very game saved us from boredom in our free periods. So, it was quite an easy choice for us to replicate one of the most iconic childhood games. Despite its apparent simplicity, Tic-tac-toe requires detailed analysis to determine even some elementary combinatory facts, the most interesting of which are the number of possible games and the number of possible positions. A position is merely a state of the board, while a game usually refers to the way a terminal position is obtained.

1. Naive counting leads to 19,683 possible board layouts (3^9since each of the nine spaces can be X, O or blank), and 362,880 i.e. 9!) possible games (different sequences for placing the Xs and Os on the board).
2. The game ends when three-in-a-row is obtained.
3. The number of Xs is always either equal to or exactly 1 more than the number of Os (if X starts).

So, to improve the time complexity & to ensure that minimum space is wasted, we are going to simulate this on a multi tape Turing machine.

2 THEORY

A Turing Machine is an abstract machinethat manipulates symbols on a strip of tape according to a set of rules; to be more exact, it is a mathematical model that defines such a device.Despite the model's simplicity, given any **computer algorithm**, a Turing machine can be constructed that is capable of simulating that algorithm's logic.
The machine operates on an infinite memory tape divided into cells. The machine positions its head over a cell and "reads" the symbol there.Then per the symbol and its present place in a *finite table* of user-specified instructions the machine (i) writes a symbol (e.g. a digit or a letter from a finite alphabet) in the cell (some models allowing symbol erasure and/or no writing), then (ii) either moves the tape one cell left or right (some models allow no motion, some models move the head) then (iii) (as determined by the observed symbol and the machine's place in the table) either proceeds to a subsequent instruction or halts the computation. A Turing machine consists of a line of cells known as the "tape", together with a single active cell, known as the "head". The cells on the tape can have a certain set of possible colors, and the head can be in a certain set of possible states. Any particular Turing machine is defined by a rule which specifies what the head should do at each step. The rule looks at the state of the head, and the color of the cell that the head is on. Then it specifies what the new state of the head should be, what color the head should "write" onto the tape, and whether the head should move left or right. The prize Turing machine has two possible states of its head, and three possible colors on its tape. The animation below shows the operation of the machine, with the states of the head represented by the orientation of the arrows. In the example shown, the Turing machine starts from a "blank" tape in which every cell is white. In the analogy with a computer, the "tape" of the Turing machine is the computer memory, idealized to extend infinitely in each direction. The initial arrangement of colors of cells on the tape corresponds to the input given to the computer. This input can contain both a "program" and "data". The steps of the Turing machine correspond to the running of the computer. The rules for the Turing machine are analogous to machine-code instructions for the computer. Given particular input, each part of the rule specifies what "operation" the machine should perform. The remarkable fact is that certain Turing machines are "universal", in the sense that with appropriate input, they can be made to perform any ordinary computation. Not every Turing machine has this property; many can only behave in very simple ways. In effect, they can only do specific computations; they cannot act as "general-purpose computers". This prize is about determining how simple the rules for

a Turing machine can be, while still allowing the Turing machine to be "universal". A universal Turing machine has the property that it can emulate any other Turing machine—or indeed any computer or software system. Given rules for the thing to be emulated, there is a way to create initial conditions for the universal Turing machine that will make it do the emulation.

3 Implementation Details

The Turing Machine that we are going to use is to be used to simulate the classic game of TIC-TAC-TOE.As notified earlier, a Turing machine consists of a read/write head and a tape to perform computations, with Turing machines being the ultimate computation model. For the purpose of simulating TIC-TAC-TOE, we choose a Turing Machine model with 3 tapes i.e. a multi-tape model. The description of the algorithm used and computation is discussed in the sections below. We use a 3-tape Turing Machine model for modelling the TIC-TAC-TOE game. We define the uses of various tapes as follows:

- TAPE1: It consists of the input of player 1.The input is of the form of numbers between 1 and 9 spaced by a zero where the numbers 1 to 9 notify the grids of a 3 by 3 matrix.Example-304020.
- TAPE2: It consists of the input of player 2.The input is of the form of numbers between 1 and 9 spaced by a zero where the numbers 1 to 9 notify the grids of a 3 by 3 matrix.Example-304020.
- TAPE3: This tape is the main working tape with the positions of both the 'X' and 'O' players on it. This tape is the checked for both the X's and the O's if they form a match. If they do, the Turing machine comes to a halt into the final state.

As we can see in the definition above, the Turing machine itself consists of a tape head, some way of holding a state, and a mechanism to impose a set of transition rules. It assumes the presence of a tape on which it can move its head and on which it can read and write its symbols, but the tape is not actually part of the machine. It is neither part of it mathematically (there is no mention of the tape in the mathematical definition of a TM) nor conceptually. Sometimes it is argued that a Turing machine models computers; certainly, with computers, we consider internal memory and disk space part of the machine, so analogously, the tape should be considered part of a Turing machine. But a Turing machine doesn't model computers, it models the activity of computing.When we think of that activity, we do

not consider it to be restricted to a fixed amount of working memory. When we think of a person conducting computations, we do not assume the person is operating with a given, fixed amount of resources such as working memory or scrap paper. What is more, we do not consider scrap paper to be part of the person. Likewise, a Turing machine does not contain the tape it uses. From the definition above you can also see that the computation of a Turing machine will never use more than a finite amount of tape. To be exact, after N steps it can never have strayed further than positions from where it started. It has never looked anywhere else on the tape. So the tape doesn't need to be infinite: in actual practice, it can be extended whenever the tape head reaches an end. However, while we never actually need an infinite amount of tape, it is fundamental to computing with Turing machines that we can supply arbitrary amounts of tape as the need arises. For some Turing machine configurations, it is possible to calculate a maximum stretch of tape in advance such that the machine's operation will never run off of it, no matter what the initial tape contents are; but this is not true in general. As a matter of fact, some computational tasks can only be implemented by Turing machines with the property that, no matter how much tape you provide initially, even if you only decide on the amount after looking at the initial tape contents, the machine may still turn out to run off the end of it when put to work. There is a hard mathematical proof of this fact. So it is fundamental to the computing poer of Turing Machines that they can write on an arbitarily large amount of tape; that we cannot even predict how much tape will be needed. This is what books and articles about Turing machines (such as the writeups below) really mean to say when they (again, incorrectly, in my opinion) write that Turing machines "have" an "infinite" amount of tape.

The machine has a tape head that can read and/or write a symbol on a tape. It can also shift the tape one position to the left and to the right. At any time the machine is in some internal state.The machine's operation is defined by a finite set of transitions. A transition is specified by four items: a state, a symbol, another state, and an action. The transition is applicable whenever the machine is in the first state, and the head is on the specified symbol. It is applied by moving into the second state and performing the action. The action is either a move to the left, or a move to the right, or a symbol (to be written on the tape, overwriting whatever symbol was there). That's it. Anything that behaves according to these specifications is a Turing machine. The transitions completely define the machine's operation. Different Turing machines differ by having different transition tables. The set of transitions must be finite, implicitly restricting the set of internal states and the set of different symbols the machine can read or write to be finite as well. If no

two transitions share the same first state,symbol pair, which ensures that at most one transition is applicable at any point in the computation, the Turing machine is said to be deterministic.
A computation of a Turing machine is a (finite) sequence of valid applications of transitions. Clearly, the set of possible computations depends on the initial tape content. The Turing machine may end up in a state, above a symbol, for which no applicable transitions exist. The computation is finished: the Turing machine halts. It is also possible for a Turing machine NEVER to halt.
For example, consider the Turing machine without any transitions. This machine, regardless of the tape contents, will always halt immediately. Now consider the machine with one transition: in state s, on symbol a, it will go to state s and write symbol a. This Turing machine will always halt whenever it is started with its tape head on a symbol other than a; but when stared on an a, it will never halt.
We input the positions of both the players on the 2 tapes i.e. Tape 1 and Tape 2.The player who gets the first chance gets his tape evaluated first. His positions of the grids (either X or O) get stored onto the 3rd tape followed by either X or O and a Blank. This symbol is checked onto the 3X3 grid to find out whether a match occurs or not. If a match occurs, the Turing machine goes into the final state and comes to a halt. If, however, the Turing machine does not find a match in the 3X3 grid, it continues on to store the input of the 2nd player on the tape. Again, this input is checked to find out if it a match occurs or not. If a match occurs, the Turing Machine goes to the final state and comes to a halt else, it proceeds to store the next input. The 3rd tape which is used here, may be optional. A total of possible eight cases determine whether a case is satisfied or not. These are as follows:

1. 1, 2 and 3rd position.
2. 4, 5 and 6th position.
3. 7, 8 and 9th position.
4. 1, 4 and 7th position.
5. 2, 5 and 8th position.
6. 3, 6 and 9th position.
7. 1, 5 and 9rd position.
8. 3, 5 and 7th position.

4 Algorithm

- Final Game States are ranked according to whether they're a win, draw, or loss.
- Intermediate Game States are ranked according to whose turn it is and the available moves.
- If it's X's turn, set the rank to that of the *maximum* move available. In other words, if a move will result in a win, X should take it.
- If it's O's turn, set the rank to that of the *minimum* move available. In other words, If a move will result in a loss, X should avoid it.

Strategy for winning every game-
A player can play a perfect game of Tic-tac-toe (to win or, at least, draw) if they choose the first available move from the following list, each turn

1. Win: If you have two in a row, play the third to get three in a row.
2. Block: If the opponent has two in a row, play the third to block them.
3. Fork: Create an opportunity where you can win in two ways.
4. Block Opponent's Fork: **Option 1**: Create two in a row to force the opponent into defending, as long as it doesn't result in them creating a fork or winning. For example, if "X" has a corner, "O" has the center, and "X" has the opposite corner as well, "O" must not play a corner in order to win. (Playing a corner in this scenario creates a fork for "X" to win.) **Option 2**: If there is a configuration where the opponent can fork, block that fork.
5. Center: Play the center.
6. Opposite Corner: If the opponent is in the corner, play the opposite corner.
7. Empty Corner: Play an empty corner.
8. Empty Side: Play an empty side.

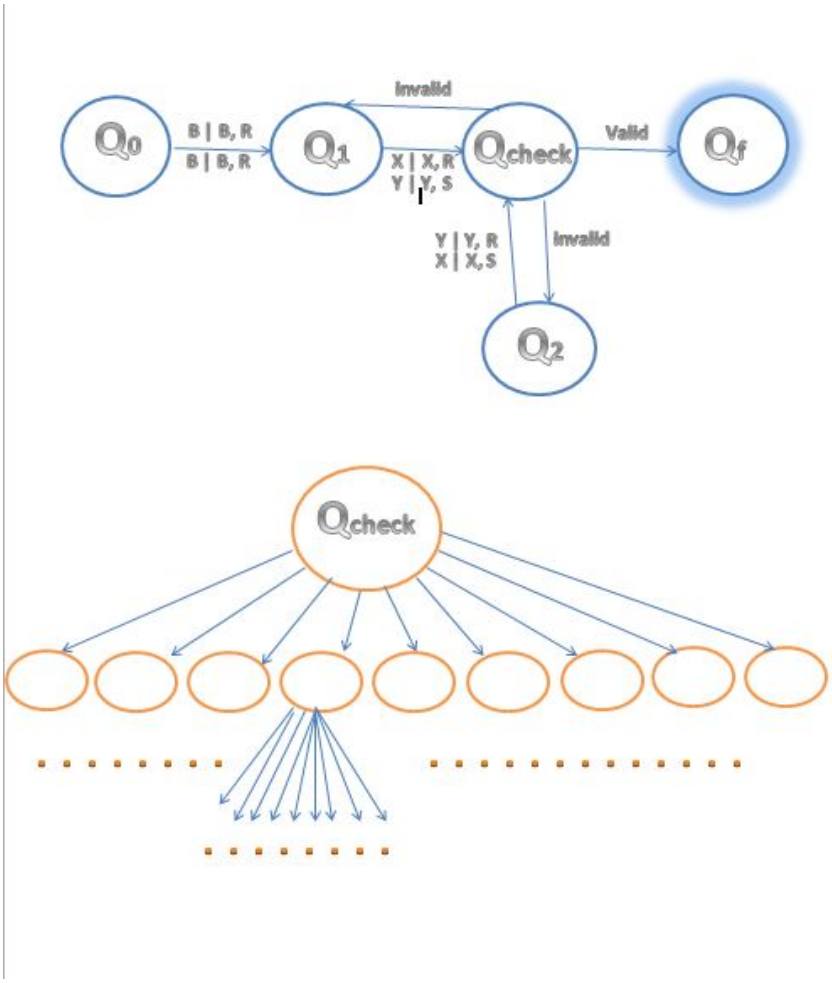

Figure 1: Transition Diagram

5 Transition Diagrams

6 Data Structures on tape

We are going to implement the tree data structures on the tapes. Note that this isn't a full game tree. A full game tree has hundreds of thousands of game states, so that's uh... not really possible to include in an image. Some of the discrepancies between the example above and a fully-drawn game tree include:

- The first Game State would show nine moves descending from it, one for each of the empty spaces on its board

- Similarly, the next level of Game States would show eight moves descending from them, and so on for each Game State

Representing the game as a game tree allows the computer to evaluate each of its current possible moves by determining whether it will ultimately result in a win or a loss. We'll get into how the computer determines this in the next section, Ranking. Before that, though, we need to clearly define the central concepts defining a Game Tree:

- The board state. In this case, where the X's and O's are.
- The current player - the player who will be making the next move.
- The next available moves. For humans, a move involves placing a game token. For the computer, it's a matter of selecting the next game state. As humans, we never say, "I've selected the next game state", but it's useful to think of it that way in order to understand the minimax algorithm.
- The game state - the grouping of the three previous concepts.

So, a Game Tree is a structure for organizing all possible (legal) game states by the moves which allow you to transition from one game state to the next. This structure is ideal for allowing the computer to evaluate which moves to make because, by traversing the game tree, a computer can easily "foresee" the outcome of a move and thus "decide" whether to take it. Next we'll go into detail about how to determine whether a move is good or bad.

6.1 Ranking Game States

The basic approach is to assign a numerical value to a move based on whether it will result in a win, draw, or loss. We'll begin illustrating this concept by showing how it applies to final game states, then show how to apply it to intermediate game states.

6.2 Final Game States

It's X's turn, and X has three possible moves, one of which (the middle one) will lead immediately to victory. It's obvious that an AI should select the winning move. The way we ensure this is to give each move a numerical value based on its board state. Let's use the following rankings:

- Win: 1

- Draw: 0
- Lose: -1

These rankings are arbitrary. What's important is that winning corresponds to the highest ranking, losing to the lowest, and a draw's between the two. Since the lowest-ranked moves correspond with the worst outcomes and highest-ranked moves correspond with the best outcomes, we should choose the move with the highest value. This is the "max" part of "minimax". You might be wondering whether or not we should apply different rankings based on the player whose turn it is. For now, let's ignore the question entirely and only view things from X's perspective. Of course, only the most boring game in the world would start out by presenting you with the options of "win immediately" and "don't win immediately." And an algorithm would be useless if it only worked in such a situation. But guess what! Minimax isn't a useless algorithm. Below I'll describe how to determine the ranks of intermediate Game States.

6.3 Intermediate Game States

As you can see, it's X's turn in the top Game State. There are 3 possible moves, including a winning move. Since this Game State allows X to win, X should try to get it if possible. This Game State is as good as winning, so its rank should be 1. In general, we can say that the rank of an intermediate Game State where X is the current player should be set to the maximum rank of the available moves. It's O's turn, and there are 5 possible moves, three of which are shown. One of the moves results in an immediate win for O. From X's perspective this Game State is equivalent to a loss, since it allows O to select a move that will cause X to lose. Therefore, its rank should be -1. In general, we can say that the rank of an intermediate Game State where O is the current player should be set to the *minimum* rank of the available moves. That's what the "mini" in "minimax" refers to.

By the way - the above game tree probably looks ridiculous to you. You might say, "Well of course you shouldn't make such a dumb move. Why would anyone give up a win and allow O to win?" Minimax is our way of giving the computer the ability to "know" that it's a dumb move, too.

7 Conclusion and Improvements

All those paths which consists any one of the 8 halting conditions will result in reaching to the final state. Moreover, all the paths leading from the root

to the leaves will result in the output being as invalid and hence, the Turing Machine keeps running. If there is no halting state even after all the inputs have been exhausted, the result would be considered as a draw.

7.1 Runtime and space usage

For Turing machines notions are easily. Since considerations of **EFFICIENCY** were soon very important for actually doing stuff, this was a definite advantage of Turing machines.
Thus, Turing machines have been established as **THE** model of computation, which could be seen as a combination of historical "accident" and some of its key properties. Nevertheless, many models have been defined since and are avidly used, in particular in order to overcome the shortcomings of Turing machines; for instance, they are tedious to "program" (i.e. define).

7.2 2. Provides Concurrency

The primary issue with simulation languages is the difficulty in creating models. These languages are introduced to make simulations run faster or for creating standards, but they take much longer to construct relative to the solutions being replaced. So, all the simulation performance benefits are lost to the model construction time. The Smart Machine addresses this by providing a small set of instructions and an integrated regular expression language. Models can be constructed using only these instructions and are as such readable by non-modelers.

References

[1] `http://stackoverflow.com/questions/16303327/advantages-of-a-2-stack-pda-and-a-multitape-turing-machine`

[2] `http://www.embedded.com/design/prototyping-and-development/4006677/Turing-inspired-scripting-language-simplifies-simulation-complexity`

[3] `http://cs.stackexchange.com/questions/9341/practical-importance-of-turing-machines`

[4] `https://www.youtube.com/watch?v=lBNZIO1JxXs`

[5] `http://neverstopbuilding.com/minimax`

[6] https://en.wikipedia.org/wiki/Turing_machine

[7] http://mathworld.wolfram.com/TuringMachine.html

[8] http://www.alanturing.net/turing_archive/pages/reference%20articles/what%20is%20a%20turing%20machine.html

[9] https://en.wikipedia.org/wiki/Tic-tac-toe

[10] http://www.i-programmer.info/babbages-bag/23-turing-machines.html

We would like to thank all the authors who contributed towards the above web pages. Also, we would to extend our gratitude who gave us motivation & most importantly time to complete the project without feeling it as a burden.