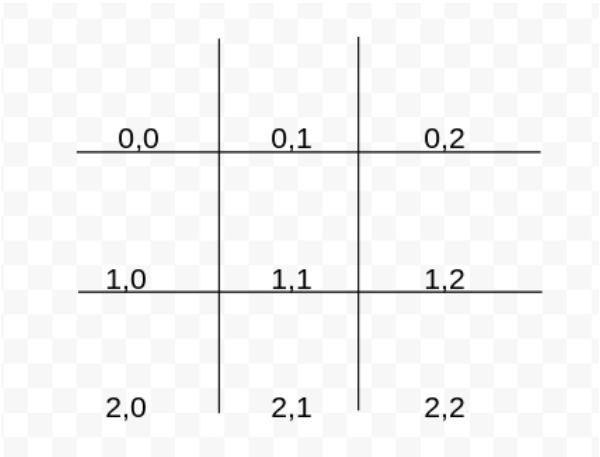
# Tic-Tac-Toe AI

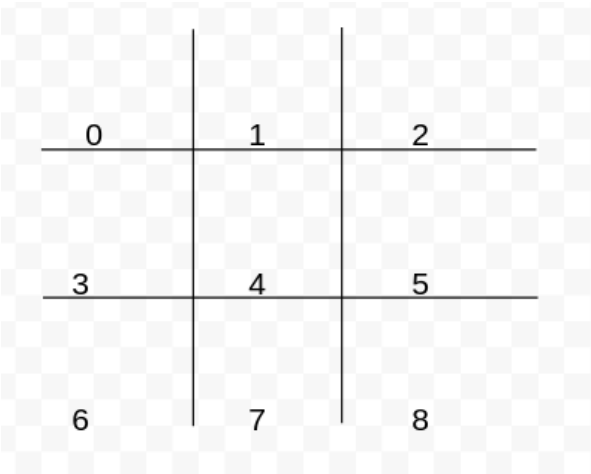
Priyanshu Tirkey | 001710501005

an **out-the-box approach**(at least for me) for solving Tic-Tac-Toe using **simple observation**, ,some smart **mapping technique**, **trie** and **minimax algorithm**.

---

## Grid Representation :




**Note**: the above two configurations.

The key thing to remember is how each box is labelled .

Eg:

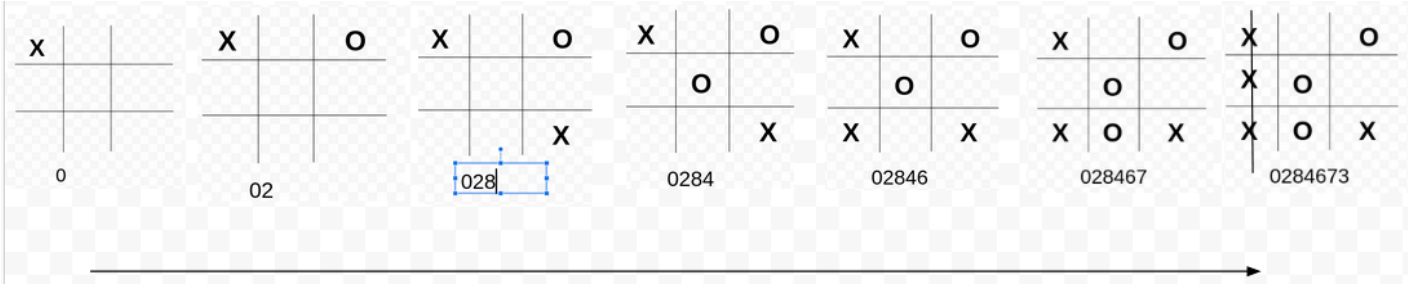
- The 1st box can be denoted by (0,0) and 0
- Box in the middle of the grid can be denoted by (1,1) or 4.

---

## How a sequence of number relates to the game that has taken place:

Consider the sequence “0284673”:

The above sequence means that:
The game proceeds in the following way;




Note :

1. **<u>the sequence of numbers tells how the game proceeds.</u>**
2. **odd indexes** represents position maked by ‘**X**’,
   **even indexes** represent positions marked by ‘**O**’.

(Assume ‘X’ to be a first player who marks ‘X’
and assume ‘O’ to be a second person who marks ‘O’)

**“SO A SEQUENCE OF NUMBER CAN REPRESENT THE WHOLE GAME!”**

also they represent a state

---

## **Deducing an upper bound** on the number of final positions where the **game ends.**

Ways to fill up the grid =

9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1 = 9! = 362880 ways.

9! is also the number of ways string “0123456789” can be arranged.

Used the following snippet to find all possible ways:

```
string w = "012345678";
do {
    cout << w << endl;
} while (next_permutation(w.begin(), w.end()));
```

- Now obviously we are unnecessarily filling all 9 boxes when the game is already over
- Also the game cannot get over in less than 5 moves.

Therefore starting from the prefix of length 5, we check if the game ends before 9 moves.

```
def get_optimal_action_list(action_list):
    '''
        each permutation is reduced in such a way that if a game concludes before 9 moves.
        the extra elements are removed.
    '''
    grid = [[None, None, None],  [None, None, None],  [None, None, None]]

    c = 'O'
    for v in action_list[:5]:
        x = int(v)%3
        y = int(v)//3
        grid[y][x] = c

        c = alter(c)

    if did_some1_win(grid):
        return action_list[:5]

    for i in range(5,9):
        v = int(action_list[i])

        x = v%3
        y = v//3

        grid[y][x] = c
        c = alter(c)

        if did_some1_win(grid):
            return action_list[:i+1]

    return action_list
```

allpossiblegames.txt

```
allpossiblegames.txt
420871365
754082631
875132064
074853621
36078124
3657024
068517342
32687450
485302716
846270531
31507864
320174856
165387240
34258701
```

It is seen that **279576 states exist** where the game ends.

But wait **279576 ways** represent games where the game has concluded.
What about states like “0284” where the game has not concluded.

| | | |
|---|---|---|
| X | | O |
| | O | |
| | | X |

0284

So considering all intermediate states:

```python
#enlist game from start state to end state and save it in allgamestates
for eachgame in allconludedgames:
    for i in range(len(eachgame)+1):
        allgamestates.append(eachgame[:i])
```

allstates.txt

```
81254076
0425387
087123546
823460
6058237
74805612
3514782
53248706
520768413
518426037
```

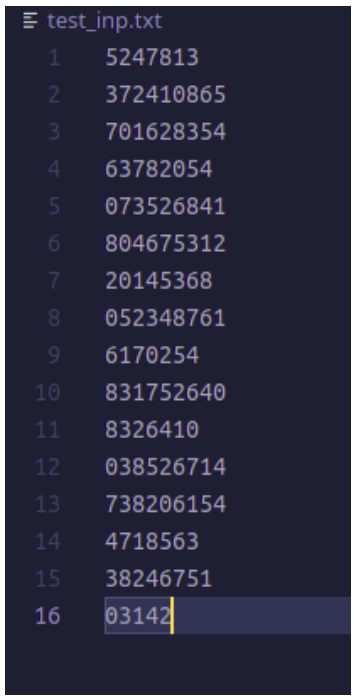
There are **636922 nodes in the decision tree.**
Reminder note: the node not only represents the grid representation but also how we got to that representation.

## <u>All Good but how can i get a path for the best options to take and what algorithm can I use?</u>

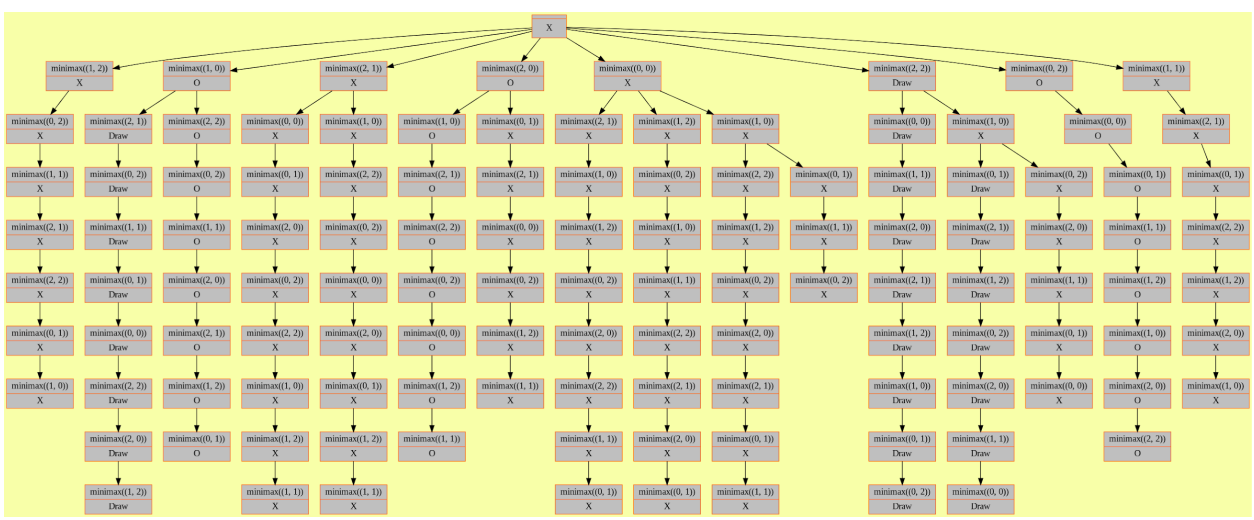
- Each concluded game can be stored in a **Trie.**
- We can apply **minimax** on that trie.

TESTING:

Let for some of the us take games for printing the decision parth



```
test_inp.txt
5247813
372410865
701628354
63782054
073526841
804675312
20145368
052348761
6170254
831752640
8326410
038526714
738206154
4718563
38246751
03142
```

- 

IMG LINK: [decision_tree_for_testinput]

For all **279576** games my system was failing to render the decision tree. Which was obvious **636922 nodes** in data structure seems fine but buy rendering that is madness.
I still tried…

min_max.py

```python
graph = {}
@vs(ignore_args=["root", "action_list", "maximizingPlayer"], node_properties_kwargs={"shape":"record", "color":"#f57542", "style":"filled", "fillcolor":"grey"})
def minimax(pos, root, action_list , maximizingPlayer = True):
    '''
        Return the path taken by the player.
    '''
    if(root.data != ""):
        graph.setdefault(uni_id(action_list), set()).add(action_list+root.data)

    #if at the end of the game
    if len(root.link) == 0:
        # print(action_list+root.data)
        if did_some1_win(gen_grid(map(int,action_list+root.data))):
            if maximizingPlayer:
                return "O"
            else:
                return "X"

        return "Draw"

    if maximizingPlayer:
        maxEval = -1

        for child_data, child_link in root.link.items():
            pos = (int(child_data)//3, int(child_data)%3)
            winner = minimax(pos, root=child_link, action_list=action_list+root.data, maximizingPlayer=False)
            v = mapped_val[winner]
            maxEval = max(maxEval, v)
        return inv_map[maxEval]

    else:
        minEval = +1

        for child_data, child_link in root.link.items():
            pos = (int(child_data)//3, int(child_data)%3)
            winner = minimax(pos, root=child_link, action_list=action_list+root.data, maximizingPlayer=True)
            v = mapped_val[winner]
            minEval = min(minEval, v)
        return inv_map[minEval]

    raise Exception #should not reach this point program should enter the else and return
```

IMG link: [minimax_algo.png]

## Hybrid Adjacency Link:

All good but can the decision tree be compressed to form a graph,
A graph where “02846” and “04826” denotes the same node and we drop the game progress data.

Soln: node is represented by
(<position marked ‘X’(sorted)>,<position marked ‘O’(sorted)>)
I.e both state map to (“068”, “24”)
note information how game proceeds has been dropped.

We make a mapping such as

```
4048   ('068', '24') {'028463', '826407', '648205', '628405', '628407', '046285', '048265', '84620
```

In the line no. 4048

- State (‘068’, ‘24’) on next input of ‘O’ will transit to one of the game given in {}.
- Running the same transformation of (<position marked ‘X’(sorted)>,<position marked ‘O’(sorted)>) for the next state will give us the node.
- Using above two point we can generate the whole graph.

log_main.txt

```
4812   ('1', '') {'14', '17', '16', '10', '18', '12', '13', '15'}
4813   ('1', '7') {'170', '173', '172', '175', '174', '178', '176'}
4814   ('1', '0') {'106', '107', '102', '108', '103', '104', '105'}
4815   ('1', '5') {'157', '153', '158', '152', '150', '156', '154'}
4816   ('1', '4') {'145', '143', '142', '148', '146', '147', '140'}
4817   ('1', '6') {'165', '164', '167', '163', '160', '168', '162'}
4818   ('1', '8') {'185', '182', '184', '186', '183', '187', '180'}
4819   ('1', '3') {'130', '134', '138', '137', '136', '135', '132'}
4820   ('1', '2') {'124', '127', '120', '123', '125', '128', '126'}
4821
```

Also there will be **4820 unique positions** that can be reached while playing TicTacToe.

## Source Code:

Zipped folder : link