Artificial Computing and Intelligence

Game playing: Three player game simulation : Tic Tac Toe

| Assignment - 2 | Group - 36 | 22-Aug-2022 |
| --- | --- | --- |

**Team**

| Names | BITS Id | Contribution |
| --- | --- | --- |
| loushingkhomba puyam | 2021FC04111 | 100% |
| V RAMESH | 2021FC04480 | 100% |
| VENU GOPAL SHARMA | 2021FC04649 | 100% |
| Vineet Kumar Maheshwari | 2021FC04604 | 100% |

[Overview](#_3dq25a8oqe0o)

[Problem Formulation](#_ybqlnihwecpg)

[Environment](#_hbmgbjqt7cw5)

[Actuators](#_8oo8aohuz4c6)

[Sensors](#_pku7ksccfxwj)

[Design](#_qi07nfr74bsd)

[Variables, Domains and State](#_yy1lbmkp3v0z)

[Constraints](#_7eix7246hm2b)

[Goal](#_6mv70f7aojfw)

[Algorithm](#_fm46ucosmwp4)

[Core Logic - Player turns and automation](#_qgz3pc4ob6c8)

[Core Logic - Minimizing and Maximizing utility](#_9jzbxyalest4)

[Complexity](#_cist0r9wjtau)

# Overview

Simulating the working of TicTacToe game that extends to three players with coins ‘X’, ‘O’ and ‘T’ with a board of dimension 4X4. The first player who matches any three consecutive coins in the same row, same column or same diagonal wins the game.

# Problem Formulation (PEAS)

Fastest to obtain 3 consecutive coins in the same row or same column or same diagonal wins. Number of squares traversed to reach to the main entrance from the starting point.

## Environment

* 4X4 board
* Movements in the same row or same column or same diagonal.

## Actuators

* Can update move anywhere in all directions
* One block at a time
* Use algorithm to decide which move is best one.(MiniMax Algorithm)

## Sensors

* Since we have the board known, we can capture the grid in 4X4
* Check if the grid is filled already.
* Checks if a player won the game before moving the next move.

# Design

Problem is being modeled as a constraint satisfaction problem, where constraints are being organized as a graph and edges are relaxed based on the constraints. We are using (depth first search) DFS, using recursion for assignment of values to the variables as players take turns to win the game.

Game tree for Tic-Tac-Toe (4x4)

## 

## 

## Variables, Domains and State

* 4 x 4 = 16 cells with value being taken as one from the domain [‘X’, ‘O’, ‘T’, ‘-’]
  + ‘-’ being the default
* Assignment of value as we navigate through algorithm defines a particular state

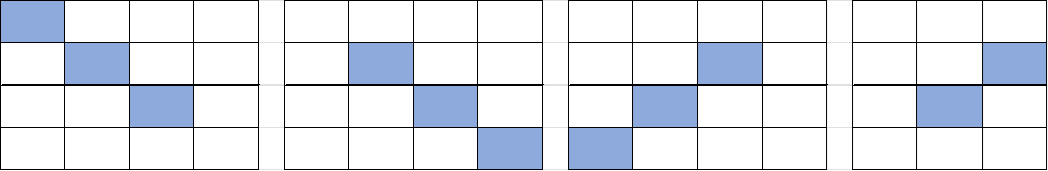
## Constraints

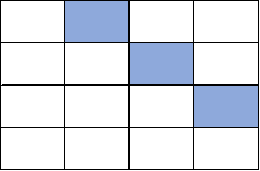
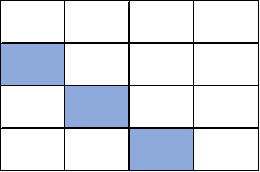
* ‘-’ value can be replaced by any of the three values, but not possible for other values if they exist in a given variable

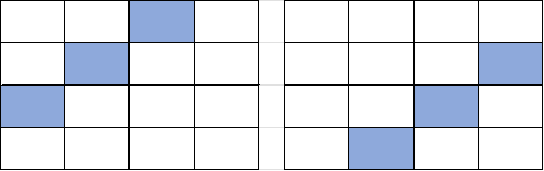
## Goal

Goal implementation is captured through the method: is\_endofgame in staticEval class.

* There will be various possible 3 consecutive cells in diagonal or row or columns. They are being validated as following:
  + In each column, validate 3 consecutive cells from top and from bottom
  + In each row, validate, first 3 consecutive cells or the last 3 cells
  + There will be following diagonals possible
    - Top left – Bottom right: top 3 and bottom 3 on this track
    - Top right – Bottom left: top 3 and bottom 3 on this track



* + - Exclude first column and last row:
      * Top left – Bottom right (starts with 2nd column)
      * 
    - Exclude last column and first row:
      * Top left - Bottom right (starts with first column, but 2nd row)
      * 
    - Similarly exclude last row and last column
      * Top right - Bottom left
    - Exclude first row and first column:
      * Top right - bottom left



# Algorithm

At high level the algorithm is encapsulated in three utility classes, depicted below. Main algorithm for initialization and game play in turns between alpha, beta and theta is achieved in minMaxAlgo. alphaBetaPruning hosts the algorithm for constraint satisfaction checks and recursively navigates to find the min/max values for the next move. staticEval is used by both the classes for testing the end of the game (or if goal is achieved).



## Core Logic - Player turns and automation

Flow is organized as following

* Alpha takes the first turn : ‘X’, Beta takes next: ‘O’ and than Theta: ‘T’
* When it is Alpha’s turn,
  + This is expected to be played by user
  + Using the above **min\_alpha\_beta**, program would recommend the move
* When it is Beta’s turn, invoke **max\_alpha\_beta**
* When it is Theta’s turn, invoke **max\_alpha\_theta**

Each of the **min/max method** **returns the location of cell** which is best to play for specific player who has the turn. In end handover the turn to next player

This keeps going in loop till end state is achieved. (is\_endofgame)

## Core Logic - Minimizing and Maximizing utility

On Alpha’s turn, note that it will invoke minimize alpha beta with utility value for alpha and beta. Objective is to find the edge (edges are all empty locations) which gives better value than it already has.

Note that it is exploring each cell 4 x 4 for everytime it gets invoked.

Min Alpha Beta further invokes max alpha beta to discover minimum utility value for alpha player as beta takes its turn in possible edges.

| for i in range(0, 4):  for j in range(0, 4):  if self.current\_state[i][j] == '-':  self.current\_state[i][j] = 'O'  (m, min\_i, min\_j) = alphaBetaPruning.min\_alpha\_beta(self,alpha, beta)  if m > maxv:  maxv = m  px = i  py = j  self.current\_state[i][j] = '-'   # Next two ifs in Max and Min are the only difference between regular algorithm and minimax  if maxv >= beta:  return (maxv, px, py)   if maxv > alpha:  alpha = maxv |
| --- |

For other min/max with alpha and theta, the pattern of code remains the same.

## Complexity

For every move, the search time complexity reduces as path to be traversed gets relaxed due to variables being assigned values and utility discovered is not matching the criteria.

However, worst case search complexity at the beginning is expected to be O(N^N)