Comparative Implementation of Tic-Tac-Toe Using SWI-Prolog and OpenNars-AGI

Hsing-Chen (Tony) Lin

Abstract

This project explores the implementation of the Tic-Tac-Toe game using two distinct approaches: SWI-Prolog, a traditional logic programming environment, and OpenNars-AGI, a framework based on Artificial General Intelligence. The effectiveness, complexity, and performance of both implementations are analyzed and compared.

1 Introduction

Tic-Tac-Toe, a simple yet profound game, offers a fundamental platform for exploring various aspects of artificial intelligence systems, particularly in understanding and implementing rule-based decision-making and strategy development. This game is not only about placing 'X's and 'O's in a grid but involves recognizing patterns, optimizing decisions, and strategizing based on the opponent's moves.

In contrast, the choice to implement Tic-Tac-Toe in Prolog, a distinguished logic programming language, presents an excellent opportunity to fundamentally grasp how logical reasoning can be effectively applied to develop game algorithms.

2 Implementation in SWI-Prolog

In this section, we explore the implementation of a Tic-Tac-Toe AI robot using SWI-Prolog. We begin our implementation by dividing it into four main parts: Recognizing the Game Board, Win Conditions, Game Analysis, and Decision Making. Initially, we construct the game board as a dynamic list

of moves. We then delve into the methodologies for each part, starting with how the game recognizes board states, evaluates potential moves, determines win conditions, and finally, how decisions are made within the game.

2.1 Recognizing the Game Board

The game board in Tic-Tac-Toe is represented as a list of moves, where each move is denoted by `move(Player, X, Y)`, indicating that a player has placed a mark at coordinates (X, Y). During each turn, the move made by the player, specified by the coordinates and the player's identifier, is stored in the `GameBoard`. This list accumulates the history of moves made throughout the game. The robot recognizes the game board state by iterating over this list to check the positions and to determine the possible valid moves.

(2,0)	(2,1)	(2,2)
(1,0)	(1,1)	(1,2)
(0,0)	(0,1)	(0,2)

2.1.1 Evaluating Potential Moves

The predicate `list_valid_moves` ensures that Prolog evaluates every possible move on the board. By processing each column separately, it accumulates all potential moves into a single list, considering only the unoccupied cells.

```
% Generate a list of valid moves, this process should not
   be influenced by Prolog's decision-making.
list_valid_moves(Moves, GameBoard, Player) :-
    valid_moves_for_column(0, Moves1, [], GameBoard, Player
       ),
    valid_moves_for_column(1, Moves2, Moves1, GameBoard,
       Player),
    valid_moves_for_column(2, Moves, Moves2, GameBoard,
       Player).

% Calculate all valid moves for a specific column.
valid_moves_for_column(Column, ValidMoves, InitialMoves,
   GameBoard, Player) :-
    valid_move_for_cell(Column, 0, Moves1, InitialMoves,
       GameBoard, Player),
```

```
10      valid_move_for_cell(Column, 1, Moves2, Moves1,
            GameBoard, Player),
11      valid_move_for_cell(Column, 2, ValidMoves, Moves2,
            GameBoard, Player).
12
13  % Calculate a valid move for a specific cell.
14  valid_move_for_cell(Column, Row, UpdatedMoves, CurrentMoves
        , GameBoard, Player) :-
15      member(move(_, Column, Row), GameBoard) -> CurrentMoves
            = UpdatedMoves;
16      UpdatedMoves = [move(Player, Column, Row) |
            CurrentMoves].
```

2.2 Win Conditions

The predicate `win_condition` is specifically designed to evaluate the game board for any set of three consecutive marks made by the same player, which would indicate a win. This includes checks across rows, columns, and both diagonal lines. Here is how the win conditions are defined and recognized:

```
1   % Check if any row, column, or diagonal is completed by the
        same player
2   win_condition(Player, GameBoard) :-
3       (check_rows(Player, GameBoard);
4        check_columns(Player, GameBoard);
5        check_diagonals(Player, GameBoard)).
6
7   check_rows(Player, GameBoard) :-
8       (Row = 0; Row = 1; Row = 2),
9       member(move(Player, Row, 0), GameBoard),
10      member(move(Player, Row, 1), GameBoard),
11      member(move(Player, Row, 2), GameBoard).
12
13  check_columns(Player, GameBoard) :-
14      (Column = 0; Column = 1; Column = 2),
15      member(move(Player, 0, Column), GameBoard),
16      member(move(Player, 1, Column), GameBoard),
17      member(move(Player, 2, Column), GameBoard).
18
19  check_diagonals(Player, GameBoard) :-
```

```
20      (member(move(Player, 0, 0), GameBoard), member(move(
           Player, 1, 1), GameBoard), member(move(Player, 2, 2)
           , GameBoard)) ;
21      (member(move(Player, 0, 2), GameBoard), member(move(
           Player, 1, 1), GameBoard), member(move(Player, 2, 0)
           , GameBoard)).
```

The `win_condition` predicate functions by calling three separate predicates: `check_rows`, `check_columns`, and `check_diagonals`. Each of these predicates uses Prolog's `member` function to determine if a specific row, column, or diagonal line contains the same player's marker consecutively in all three positions. This logical check efficiently evaluates the board's state and determines if a player has won the game.

2.3 Game Analysis

The `analyze_game` predicate not only checks for immediate winning or losing conditions but also simulates potential future moves to anticipate the opponent's strategy.

2.3.1 Functionality of analyze_game:

- **Immediate Check:** Initially, the predicate checks if there is a winner in the current board configuration. If the player 'x' has already won, the function ends, reflecting a strategy to secure a win or avoid a loss immediately.
- **Strategic Planning:** If the game is still ongoing (i.e., no current winner), the function proceeds to the `continue_game_analysis` predicate.

```
1  % Analyze the game to decide the strategy
2  analyze_game(_, Game, _) :-
3      win_condition(Winner, Game),
4      Winner = x.
5  analyze_game(Player, Game, NextMove) :-
6      not(win_condition(_, Game)),
7      continue_game_analysis(Player, Game, NextMove).
```

2.3.2 Detailed Strategy Analysis:

- **List Possible Moves:** The `list_valid_moves` function is called to enumerate all possible moves that the player can make from the current game state.
- **Evaluate Moves:** Each move is evaluated by `search_game_analysis`, which simulates the game board after each potential move to foresee outcomes and determine the best strategic option.

```
continue_game_analysis(Player, Game, NextMove) :-
    list_valid_moves(Moves, Game, Player),
    search_game_analysis(Moves, Player, Game, NextMove).
```

2.3.3 Recursive Game Play:

- **Recursive Analysis:** The `search_game_analysis` and its helper, `search_game_analysis_x`, utilize a depth-first search (DFS) algorithm to recursively simulate future moves. Each potential move to the game board, and re-analyzing the results. This recursive depth helps us predict the consequences of moves several steps ahead, ensuring that strategies are both reactive and proactive.

```
search_game_analysis([], o, _, _). % Draw if no moves are
   possible.
search_game_analysis([], x, _, _).

search_game_analysis([Move|Rest], o, Game, NextMove) :-
    NextBoard = [Move | Game],
    search_game_analysis(Rest, o, Game, NextMove),
    analyze_game(x, NextBoard, _), !.

search_game_analysis(Moves, x, Game, NextMove) :-
    search_game_analysis_x(Moves, Game, NextMove).

search_game_analysis_x([Move|_], Game, Move) :-
    NextBoard = [Move | Game],
    analyze_game(o, NextBoard, _).
search_game_analysis_x([_|Rest], Game, NextMove) :-
    search_game_analysis_x(Rest, Game, NextMove).
```

This comprehensive approach to game analysis using Prolog's logical reasoning capabilities allows for sophisticated strategic planning, where we not only respond to the current game state but also plan several moves ahead, considering various possible future game scenarios.

2.4 Decision Making

The `make_decision` predicate introduces randomness by choosing randomly from a list of possible moves generated by the game analysis. This approach ensures that the game does not always follow the same predictable pattern, enhancing the robot's realism and making it more challenging for human opponents.

2.4.1 Process of Making Decisions:

- **Selection of Moves:** The function begins by ensuring that the list of possible moves is not empty. It then randomly selects one move from the list. This randomness introduces unpredictability into the game, simulating a more human-like decision-making process.
- **Application of the Move:** After selecting a move, it is applied to the current board to generate a new game state. This new state is then displayed using the `print_board` function.
- **Checking for a Win:** Once the move has been made, the game checks if this move results in a win for the player 'x'. If so, the game announces the player's victory; otherwise, it hands the turn over to the opponent.

```
% Make a decision based on possible moves
make_decision(PossibleMoves, CurrentBoard) :-
    not(PossibleMoves = []),
    length(PossibleMoves, NumberOfMoves),
    random(0, NumberOfMoves, SelectedMove),
    nth0(SelectedMove, PossibleMoves, Move),
    NextBoard = [Move | CurrentBoard],
    print_board(NextBoard),
    (win_condition(x, NextBoard) ->
        (write('I won. You lose.'), nl);
        opponent_turn(NextBoard), !).
```

2.5 Conclusion

This implementation illustrates the use of Prolog's logical inference capabilities to manage game states and decide optimal moves in Tic-Tac-Toe. By structuring the game logic in terms of rules and logical queries, Prolog allows for clear and concise representation of the game mechanics and decision-making processes.

3 Implementation in OpenNars-AGI

This section describes the implementation of Tic-Tac-Toe using the OpenNars-AGI framework, focusing on how NARS is utilized merely as an observer of the game to analyze possible truths of game states. Although NARS does not yet function as a fully autonomous AI agent within this project, its capabilities in reasoning and learning provide valuable insights into game strategy and win conditions. This part of the project was developed in two versions. The first version represented my initial ideas on implementation, which encountered several issues leading to inaccuracies in the inference results. The second version is a revised implementation that allows for more accurate predictions of potentially winning moves, although it still occasionally makes errors in judgment.

3.1 Recognizing the Game Board

In the initial version, the process of recognizing the game board focused primarily on the state of each cell within the grid. At the start, the game board was conceptualized as entirely empty, and specific Narsese rules were implemented to manage the occupancy state of each cell. These rules ensured that a cell is marked as empty if it is not occupied by either player.

```
1 // All cells are initially empty
2 <{x0y0, x0y1, x0y2, x1y0, x1y1, x1y2, x2y0, x2y1, x2y2} -->
      [empty]>.
3
4 //If a cell is marked as occupied by player1, it cannot
     also be marked as empty.
5 <<(*, {player1},{$1}) --> board> <=> (&&, (--, <(*, {
     player2},{$1}) --> board>), (--, <{$1} --> [empty]>))>.
```

```
<<(*, {player2},{$1}) --> board> <=> (&&, (--, <(*, {
   player1},{$1}) --> board>), (--, <{$1} --> [empty]>))>.
<<{$1} --> [empty]> <=> (&&, (--, <(*, {player1},{$1}) -->
   board>), (--, <(*,{player2},{$1}) --> board>))>.
```

3.2 Win Conditions

Win conditions were defined using a set of rules that identified potential winning lines (rows, columns, diagonals) on the board. The system was configured to detect these lines and evaluate whether they could potentially lead to a win based on the current placement of markers.

3.2.1 Defining Winning Lines and Conditions

```
// Initialize player win probability
<{player1} --> win>.  %0.5%
<{player2} --> win>.  %0.5%

// Ensure only one player can win at a time
<<{player1} --> win> ==> (--, <{player2} --> win>)>.
<<{player2} --> win> ==> (--, <{player1} --> win>)>.

// Define all potential lines that can result in a win
<(*, {x0y0}, {x0y1}, {x0y2}) --> sameline>.
<(*, {x1y0}, {x1y1}, {x1y2}) --> sameline>.
<(*, {x2y0}, {x2y1}, {x2y2}) --> sameline>.
<(*, {x0y0}, {x1y0}, {x2y0}) --> sameline>.
<(*, {x0y1}, {x1y1}, {x2y1}) --> sameline>.
<(*, {x0y2}, {x1y2}, {x2y2}) --> sameline>.
<(*, {x0y0}, {x1y1}, {x2y2}) --> sameline>.
<(*, {x0y2}, {x1y1}, {x2y0}) --> sameline>.

// Define Winning Condition
<(&&, <(*, {$1}, {$2}, {$3}) --> sameline>, <(*,{$p},{$1})
   --> board>,<(*,{$p},{$2}) --> board>,<(*,{$p},{$3}) -->
   board>) ==>  <{$p} --> win>>.
```

3.3 Game Strategy

In our Tic-Tac-Toe strategy, we utilize specific logical rules to enhance the NARS's ability to win or secure a draw. These rules are designed to increase the likelihood of winning by strategically analyzing the board and identifying the most advantageous moves.

3.3.1 Strategic Move Evaluation

We implement rules to detect when a player ('p') has two marks in a row with one space empty, offering a high chance of securing a win, and another rule where a player has one mark with two spaces open, setting up future winning opportunities. These strategic evaluations allow the AI to act proactively, maximizing winning probabilities and blocking opponent advances effectively.

N		
	W	

Scenario 1

```
1 //  Two in a Row with One Empty
2 <(&&, <(*, {$1}, {$2}, {$3}) --> sameline>, <(*,{$p},{$1})
     --> board>,<(*,{$p},{$2}) --> board>,<{$3} --> [empty]>)
      ==>  <{$p} --> win>>. %0.8;0.7%
3
4 // One Occupied and Two Empty
5 <(&&, <(*, {$1}, {$2}, {$3}) --> sameline>, <(*,{$p},{$1})
     --> board>,<{$2} --> [empty]>,<{$3} --> [empty]>) ==>
     <{$p} --> win>>. %0.6;0.3%
```

3.3.2 Offense and Defense Strategies

We use logical rules to activate offensive or defensive modes dynamically. A player marked as 'ready-to-win' triggers offensive maneuvers, while the robot adopts defensive tactics to thwart similar threats from opponents. This strategic flexibility enhances our robot's ability to respond to game developments effectively, increasing the chances of securing a win or preventing a loss, as illustrated in the provided scenarios.

N		W
W	W	
N		

Scenario 1

N		W
W	W	
N	N	

Scenario 2

```
//If anyone is ready to win
<(&&, <(*, {$1}, {$2}, {$3}) --> sameline>, <(*,{$s},{$1})
   --> board>,<(*,{$s},{$2}) --> board>,<{$3} --> [empty]>)
    ==> (&&, <(*, {$s}, {$3}) --> offense>, <{$s} --> ready
   -to-win>)>.
//Offensive Play Decision
<(&&, <{player1} --> ready-to-win>, <(*, {player1}, {$1})
   --> offense>) ==> <(*, {player1}, {$1}) --> nextstep>>.
//Defensive Play Decision
<(&&, <{player2} --> ready-to-win>, (--, <{player1} -->
   ready-to-win>), <(*, {player2}, {$1}) --> offense>) ==>
   <(*, {player1}, {$1}) --> nextstep>>.
//Increased Winning Probability
<(&&, <(*, {$s}, {$1}) --> nextstep>, <(*,{$s},{$1}) -->
   board>) ==> <{$s} --> win>>. %0.9%
```

3.4 Decision Making

In the Decision Making section, we calculate the truth values for all potential positions that could lead to a win for Player 1. We then identify the position with the highest truth value, which becomes our final choice for the next move.

```
<<(*,{player1},{x0y0}) --> board> ==> <{player1} --> win>>?
    %0.50;0.06%
<<(*,{player1},{x0y1}) --> board> ==> <{player1} --> win>>?
    %0.97;0.32%
<<(*,{player1},{x0y2}) --> board> ==> <{player1} --> win>>?
    %0.99;0.58%
<<(*,{player1},{x1y0}) --> board> ==> <{player1} --> win>>?
    %0.49;0.10%
<<(*,{player1},{x1y2}) --> board> ==> <{player1} --> win>>?
    %0.97;0.32%
```

3.5 Challenges Encountered

One of the significant challenges encountered was the handling of unexpected relational inferences. For example, consider the following NARS statements that define positions on the same line:

```
<(*, {A}, {B}) --> sameline>.
<(*, {A}, {C}) --> sameline>.
```

The NARS engine incorrectly inferred a direct relationship between B and C, as shown below:

```
<{B} <-> {C}>. %1.00;0.45%
```

This inference led to an unreasonable situation where a choice for one position erroneously suggested a choice for a non-aligned position, complicating the game strategy:

```
<(*,{player1},{B}) --> choose> <=> <(*,{player1},{C}) -->
    choose>
```

3.6 Adjustments and Enhancements

To address the challenges encountered with unexpected relational inferences, significant adjustments were made to the way the game board is recognized and processed within the NARS framework.

3.6.1 Recognizing the Game Board

To ensure accurate position recognition and avoid incorrect relational inferences, explicit definitions were assigned to each position on the game board regarding their x and y coordinates. These definitions ensure that each position's coordinates are recognized independently, minimizing the risk of incorrect relational reasoning based on shared positions or implied lines, and thus enhancing the system's ability to make accurate and logical decisions in the game.

Here are the NARS statements used to define the x and y coordinates:

```
<(*, {x0y0}, [x0]) --> Px>.
<(*, {x0y0}, [y0]) --> Py>.
<(*, {x1y0}, [x1]) --> Px>.
...
```

```
5
6 //Original rule
7 <(*, {x0y0}, {x0y1}, {x0y2}) --> sameline>.
8 <(*, {x1y0}, {x1y1}, {x1y2}) --> sameline>.
```

3.6.2 Win Conditions

I directly assign color attributes to positions immediately after a move. This approach replaces the previous method of using relational associations, such as the board definition, to connect players and positions. Now, each move directly marks the position with attributes (black, white).

```
1 // Define Winning Condition
2 <(&&, <{x0y0} --> [black]>, <{x0y1} --> [black]>, <{x0y2}
     --> [black]>) ==> <{player1} --> win>>.
3 <(&&, <{x1y0} --> [black]>, <{x1y1} --> [black]>, <{x1y2}
     --> [black]>) ==> <{player1} --> win>>.
4 <(&&, <{x2y0} --> [black]>, <{x2y1} --> [black]>, <{x2y2}
     --> [black]>) ==> <{player1} --> win>>.
5 ...
6
7 //Original rule
8 <(&&, <(*, {$1}, {$2}, {$3}) --> sameline>, <(*,{$p},{$1})
     --> board>,<(*,{$p},{$2}) --> board>,<(*,{$p},{$3}) -->
     board>) ==>  <{$p} --> win>>.
```

3.6.3 Game Strategy

```
1 //Increasing Winning Probabilities
2 // One Occupied and Two Empty
3 <(&&, <(*, {$1}, [$b]) --> $a>, <(*, {$2}, [$b]) --> $a>,
     <(*, {$3}, [$b]) --> $a>, <{$1} --> [black]>, <{$2} -->
     [empty]>, <{$3} --> [empty]>) ==> <{player1} --> win>>.
     %0.6;0.5%
4
5 // Two in a Row with One Empty
6 <(&&, <(*, {$1}, [$b]) --> $a>, <(*, {$2}, [$b]) --> $a>,
     <(*, {$3}, [$b]) --> $a>, <{$1} --> [black]>, <{$2} -->
     [black]>, <{$3} --> [empty]>) ==> <{player1} --> win>>.
     %0.7;0.7%
```

```
7
8  <(&&, <(*, {$1}, [$b]) --> $a>, <(*, {$2}, [$b]) --> $a>,
      <(*, {$3}, [$b]) --> $a>,<{$1} --> [white]>,<{$2} --> [
      white]>,<{$3} --> [empty]>) ==> (&&, <{player2} -->
      ready-to-win>, <(*, {player1}, {$3}) --> defense>)>.
9  //Offensive Play Decision
10 <(&&, <{player1} --> ready-to-win>, <(*, {player1}, {$1})
      --> offense>) ==> <(*, {player1}, {$1}) --> nextstep>>.
11 //Defensive Play Decision
12 <(&&, <{player2} --> ready-to-win>, (--, <{player1} -->
      ready-to-win>), <(*, {player2}, {$1}) --> offense>) ==>
      <(*, {player1}, {$1}) --> nextstep>>.
13
14 <(&&, <(*, {player1}, {$1}) --> nextstep>, <{$1} --> [black
      ]>) ==> <{player1} --> win>>.
```

3.7 Decision Making

```
1  <<{x0y0} --> [black]> ==> <{player1} --> win>>? %0.46;0.03%
2  <<{x0y1} --> [black]> ==> <{player1} --> win>>? %0.25;0.01%
3  <<{x0y2} --> [black]> ==> <{player1} --> win>>? %0.50;0.03%
4  <<{x1y0} --> [black]> ==> <{player1} --> win>>? %0.50;0.02%
5  <<{x1y2} --> [black]> ==> <{player1} --> win>>? %0.50;0.02%
6  <<{x2y0} --> [black]> ==> <{player1} --> win>>? %0.46;0.07%
7  <<{x2y1} --> [black]> ==> <{player1} --> win>>? %0.50;0.02%
8  <<{x2y2} --> [black]> ==> <{player1} --> win>>? %0.25;0.01%
```

4 Result(Nars)

0.50;0.03	0.50;0.02	0.25;0.01
0.25;0.01	W	0.50;0.02
0.46;0.03	0.50;0.02	0.46;0.03

Step 1

B		
	W	

Step 2

B	0.47;0.04	1.00;0.41
0.32;0.19	W	0.26;0.21
W	0.26;0.04	0.29;0.07

Step 3

B		B
	W	
W		

Step 4

B	W	B
0.32;0.19	W	0.88;0.12
W	0.95;0.25	0.25;0.16

Step 5

B	W	B
	W	
W	B	

Step 6

B	W	B
0.96;0.41	W	W
W	B	0.48;0.09

Step 7

B	W	B
B	W	W
W	B	

Step 8

B	W	B
B	W	W
W	B	W

Step 9

5 Comparison of Implementations

This section compares the two different implementations of Tic-Tac-Toe: SWI-Prolog and OpenNars-AGI. Each platform has its strengths and weaknesses, which are discussed below in terms of efficiency, realism, flexibility, and scalability.

5.1 Efficiency and Accuracy

The implementation in SWI-Prolog is more efficient and accurate for a game like Tic-Tac-Toe, which has a finite number of possible game states. Prolog functions like executing an algorithm, systematically exploring all possible moves to find every potential win scenario. This deterministic approach guarantees that all possibilities are considered, leading to a higher precision

in gameplay and decision-making.

5.2 Realism and Cognitive Simulation

Conversely, the OpenNars-AGI approach mirrors how humans learn and play games. It is not about brute force computation but about understanding and adapting to new information. In this implementation, Nars is taught the basics of the game—such as recognizing the game board and understanding the rules and strategy. This method is more aligned with true Artificial Intelligence, reflecting a learning and adapting system rather than just solving a problem. Nars, therefore, requires more reasoning time and does not guarantee a win or draw with certainty, but it offers a more human-like approach to gameplay.

5.3 Flexibility and Scalability

OpenNars-AGI exhibits greater flexibility and scalability. For example, in applications like chatbot technologies (e.g., ChatGPT), it is impractical to compute all possible answers. Instead, Nars can reason within limited information to derive conclusions, making it highly scalable for more complex or undefined scenarios. On the other hand, for games with fixed rules like Tic-Tac-Toe, Prolog's ability to methodically determine all possible outcomes makes it superior. However, Prolog's rigid, rule-based approach might limit its application in scenarios that require adaptation to a wide variety of inputs and conditions.

6 Conclusion

This comparative study of Tic-Tac-Toe implementations using SWI-Prolog and OpenNars-AGI has highlighted the distinct advantages and limitations of each approach within the context of artificial intelligence applications. The Prolog implementation demonstrated a high degree of efficiency and accuracy, owing to its capability to exhaustively explore all possible game states and outcomes. This deterministic approach ensures that the optimal moves are always selected, based on logical deduction from the current game state.

Conversely, the OpenNars-AGI implementation offered a glimpse into a more human-like approach to learning and playing games. Although it did not achieve the same level of performance consistency as the Prolog implementation, its ability to adapt and learn from new scenarios presents a compelling case for its use in more dynamic and less defined environments. This capability mirrors human cognitive processes more closely than the algorithmic precision seen in Prolog.

7 References

1. OpenNARS for Applications: A reasoning system for the AI demands of practical applications and an approach towards AGI. https://github.com/opennars/opennars/wiki.

2. Gym-TicTacToe: A simple Tic-Tac-Toe environment for OpenAI Gym with random and Q-learning agents. https://github.com/haje01/gym-tictactoe/tree/master.

3. Tic-Tac-Toe Game Example in SWI-Prolog. https://swish.swi-prolog.org/p/Tic-Tac-Toe.swinb.

4. Prolog Tic-Tac-Toe: An implementation of Tic-Tac-Toe in Prolog. https://github.com/cheery/prolog-tic-tac-toe/tree/master.