


Probabilistic Tic-Tac-Toe in ProbLog (and Python)

Djanira dos Santos Gomes, Madeleine Gignoux, & Nano Batistoni

June 28, 2024

Abstract

Probabilistic Tic-Tac-Toe is an online game developed by Cameron Sun [Sun24]. An optimal solution exists, and was discovered by Louis Abraham [Abr24]. *ProbLog* is a probabilistic logic programming language developed at KU Leuven [Gro20a]. In this paper, we model Probabilistic Tic-Tac-Toe in ProbLog, and implement strategies for playing the game, which we test against a random and the optimal strategy. For instructions on how to play around with our model: `python3 probtictactoe.py -h`.

 The code is available at the [GitHub repository here](#).

Contents

1	Introduction	2
2	Building the Model	3
3	Strategies	5
3.1	Aggressive versus Defensive Optimization	5
3.2	Candidate Selection	6
3.3	Restricted Winning Conditions	6
3.4	General Losing Conditions	6
3.5	Querying the ProbLog Model	6
3.6	Example of a Selection Procedure	7
4	Results	8
4.1	Playing against the Random Strategy	9
4.2	Playing against the Optimal Strategy	9
5	Conclusion	10
A	Python Implementation	12
A.1	High-level overview	12
A.2	Some implementation details	12
	References	14

1 Introduction

Probabilistic Tic-Tac-Toe is an online game developed by Cameron Sun [Sun24] and posted on the forum Hacker News on June 11th, 2024 [igp24]. The gameplay is similar to *Tic-Tac-Toe* (otherwise known as *Noughts and Crosses*), where two players, one playing as crosses and the other playing as noughts, compete to obtain three of their marks in a row on a 3×3 game board. *Probabilistic Tic-Tac-Toe* expands on this idea, but assigning a probability distribution to each cell on the game board of a *good*, *bad*, or *neutral* event happening. Both players are aware of these probabilities, and make choices for which cell to pick based upon these probabilities. When a cell is chosen by a player, an event happens. If a *good* event happens, the player's mark is put on the cell; if a *bad* event happens, the opponent's mark is put on the cell; if a *neutral* event happens, neither mark is put on the cell. The probability distribution stays the same throughout one game. The game is won in the same way as normal Tic-Tac-Toe.

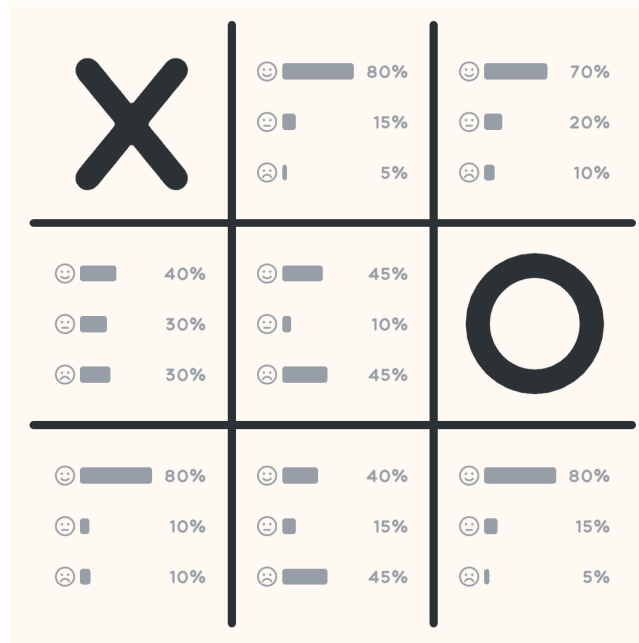


Figure 1: The game board of *Probabilistic Tic-Tac-Toe* after a couple moves (from [Sun24]).

The goal of this paper is first and foremost to create a model of this game in ProbLog: this is discussed in Section 2. ProbLog is a probabilistic logic programming language developed at KU Leuven [Gro20a]. Using this model, we can provide the current state of the board, as well as provide evidence and query the model to make choices about which cell should be selected next, which will allow us to program different strategies for playing the game. We will write about different strategies for playing the game in Section 3. The proposed optimal strategy for playing the game exists, and was discovered by Louis Abraham [Abr24]. In Section 4 we compare our strategies with the optimal strategy and a random agent by simulating the game in Python. Finally, we draw our conclusions in Section 5. A description of the Python implementation can be found in Appendix A.

2 Building the Model

We now discuss how we chose to model Probabilistic Tic-Tac-Toe in ProbLog. To start with, it is important to consider how we will use the model, as this influences how we will encode Probabilistic Tic-Tac-Toe in ProbLog. Recall that the game board of Probabilistic Tic-Tac-Toe consists of nine cells, which each have their own probability distribution of *good*, *bad*, and *neutral* outcomes, which we refer to as the *grid* of probabilities. Next, we have the current *state* of the game, which changes as cells are chosen, and a *good* or *bad* event happens. The basic idea of the model is to take in the probability grid and current state of the game, and look forwards a number of steps to discover the possible future configurations of the board, and compute the move that maximizes the probability of winning, or minimizes the probability of losing.

To start with, we encode the current state of the board with the following facts. Suppose we want to figure out which move to make for the board shown in Figure 1. To do so, we would instantiate this board with the following.

```
board(1,x,0). board(2,n,0). board(3,n,0). board(4,n,0). board(5,n,0).  
          board(6,o,0). board(7,n,0). board(8,n,0). board(9,n,0).
```

These facts have the form `board(C,P,B)` where the *C* stands for the position of the cell (starting from the top left), the *P* stands for the value in the current cell where *x* stands for a cross, *o* stands for a nought, and *n* stands for an empty cell, and finally the *B* denotes that we are looking at state *B* of the board, which to begin with is `0` or the *true* initial state of the current board. Later, we will discuss rules for propagating the state forward probabilistically.

Next we add atoms corresponding to whose turn it is, and how many turns forwards we want to look. Theoretically, the same cell can be attempted repeatedly, but return the *neutral* event every time, meaning that the game never ends. In order to avoid an infinite loop, we assume a *finite horizon*, meaning that we only look a few steps forwards at every turn. We believe that this choice will not effect the model too much, as it is not realistic for a player to look forward more steps than the number of steps needed to win in Tic-Tac-Toe. Due to the probabilistic implementation of the game, including the opponent's turns, it is possible to win in three turns (as opposed to five to six turns in deterministic Tic-Tac-Toe). Thus, we look three moves ahead, described by the following facts.

```
turn(x,1). turn(o,2). turn(x,3).
```

This expresses that the crosses player will play next, followed by the noughts player and then the crosses player as is common in Tic-Tac-Toe.

The next thing we encode are the probabilities of a *good*, *bad*, or *neutral* event happening for a given cell. To do this, we use a predicate `cell` filled in with the cell number (1-9), whether the event is good, bad, or neutral, and the step *A* of the board. For example, for the board shown in Figure 1, we would instantiate the following facts for the second cell.

```
0.80 :: cell(2,good,A); 0.05 :: cell(2,neutral,A); 0.15 :: cell(2,bad,A)  
:- turn(_,A).
```

The reason that we make these probabilities dependent on the turn, is the aforementioned condition that after a cell is attempted, the event resets, but the probability distribution stays the same.

We now define rules to propagate the state of the board forwards. Note that these rules are only applied during the computation of the probabilities of winning during the *current* turn: the program needs to consider possible developments of the game, but actual updates and moves are applied outside of it - see Appendix A for the implementation details.

We express that if a board has a neutral cell in state A, then if this cell is selected by the player on turn A+1, it will get their mark at turn A+1 with the probability of that cell having the good event happen on that turn. Moreover, we have to add the possibilities that the cell becomes the opponent's mark when the bad event happens on that turn, and finally that the cell stays neutral if then neutral event happens, or the cell is not played.

```
board(C,P,B) :- board(C,n,A), cell(C,good,B), turn(P,B), play(C,B), B is A+1.
board(C,o,B) :- board(C,n,A), cell(C,bad,B), turn(x,B), play(C,B), B is A+1.
board(C,x,B) :- board(C,n,A), cell(C,bad,B), turn(o,B), play(C,B), B is A+1.
board(C,n,B) :- board(C,n,A), cell(C,good,B), turn(_,B), play(C,B), B is A+1.
board(C,P,B) :- board(C,P,A), turn(_,B), \+play(C,B), B is A+1.
```

As seen in this code, there is a play predicate. This predicate is used to ensure that a certain cell can change only if the current player selects that cell to play. Although part of the model, the probability distribution of play is dependent on the strategy employed, as perhaps some cells are likelier to be played than other. For this paper, a uniform distribution will be assumed.

Next, we also add predicates expressing win conditions and lose conditions with respect to the crosses player. There are several possibilities for winning and losing, each corresponding to a different board configuration. A player wins (resp. loses) whenever the board is in a winning (resp. losing) configuration; moreover, if we win or lose at a turn A, then we “stop looking forward”, as the match is over.

```
win1(A) :- board(1,x,A),board(2,x,A),board(3,x,A).
win2(A) :- board(4,x,A),board(5,x,A),board(6,x,A).
win3(A) :- board(7,x,A),board(8,x,A),board(9,x,A).
win4(A) :- board(1,x,A),board(4,x,A),board(7,x,A).
win5(A) :- board(2,x,A),board(5,x,A),board(8,x,A).
win6(A) :- board(3,x,A),board(6,x,A),board(9,x,A).
win7(A) :- board(3,x,A),board(5,x,A),board(7,x,A).
win8(A) :- board(1,x,A),board(5,x,A),board(9,x,A).

lose1(A) :- board(1,o,A),board(2,o,A),board(3,o,A).
lose2(A) :- board(4,o,A),board(5,o,A),board(6,o,A).
lose3(A) :- board(7,o,A),board(8,o,A),board(9,o,A).
lose4(A) :- board(1,o,A),board(4,o,A),board(7,o,A).
lose5(A) :- board(2,o,A),board(5,o,A),board(8,o,A).
lose6(A) :- board(3,o,A),board(6,o,A),board(9,o,A).
lose7(A) :- board(3,o,A),board(5,o,A),board(7,o,A).
lose8(A) :- board(1,o,A),board(5,o,A),board(9,o,A).
```

```

win(A)  :- win1(A); win2(A); win3(A); win4(A); win5(A); win6(A); win7(A);
          win8(A).
lose(A) :- lose1(A); lose2(A); lose3(A); lose4(A); lose5(A); lose6(A); lose7(A);
          lose8(A).

win(B)  :- win(A),   turn(_,B), B is A+1.
lose(B) :- lose(A),  turn(_,B), B is A+1.

```

Finally, our ProbLog model will contain evidence of certain things happening, such as a certain board positions. Ultimately, we will query this program for desired probabilities. Both the evidence and the queries are strategy-dependent and will be discussed in the following section.

3 Strategies

Our strategies are written in Python and decide what to query the ProbLog program with. We defined four different strategies, which can be organized as shown in Table 1.

Whenever it is our algorithm’s turn (i.e. the crosses’ turn), we use a strategy to decide on a subset of candidate cells to play, and choose the *best* cell. The game simulator then applies the move by sampling a good, bad, or neutral event from the probability distribution for that cell.

	Fast	Conquer-the-Board
Aggressive	Winning Fast	Aggressive CB
Defensive	Tying Fast	Defensive CB

Table 1: The four strategies

The approaches used to select the set of candidates are defined by the *Fast* and *Conquer-the-Board* methods; the approach used to select the best cell from those candidates is decided by choosing between the *Aggressive* and *Defensive* methods. The strategies are used as an optimization of the efficiency of the program: instead of computing our chances for each possible available cell, we consider only a small subset of cells that we consider promising.

3.1 Aggressive versus Defensive Optimization

The main idea of the aggressive approach is to maximize winning: from the given set of candidates (selected through either the Winning Fast or Aggressive Conquer-the-Board approach), it selects the cell maximizing the probability of *winning* within three¹ turns.

The defensive approach aims for a tie, that is, it avoids losing. From the given set of candidates (selected through Tying Fast or Defensive Conquer-the-Board), it selects the cell that minimizes the probability of *losing* within three turns.

¹We consider winning in (at most) three turns as a default winning condition, for the same reason as choosing a finite horizon of three turns: we’d like to look forwards as little as possible, and in Tic-Tac-Toe it’s not realistic to look forward (many) more steps than the minimal number of turns required to win.

3.2 Candidate Selection

The Winning Fast strategy, and its defensive analogue Tying Fast, were designed to describe an intuitive way of playing and aims to finish the game fast. We try to place crosses adjacent to cells that already contain a cross, in order to get three crosses in a row as fast as possible. More specifically, the Winning Fast strategy selects those cells as candidates that are adjacent to *at least two* other cells already containing a cross.

The Conquer-the-Board strategy could be considered opposite to Winning Fast and Tying Fast in its selection of candidate cells: we try to 'spread out' our crosses, by selecting cells located further from those that already contain a cross. More specifically, the Aggressive Conquer-the-Board strategy selects those that are adjacent to *at most one* cell already containing a cross. This approach is aimed more at blocking opportunities from the opponent, since it selects those cells that aren't surrounded by crosses yet, i.e. these cells are surrounded by neutral cells and cells marked by noughts.

3.3 Restricted Winning Conditions

In order to further optimize efficiency in the aggressive approach, we restrict the set of candidate cells obtained in our previous selection procedure, based on their ability to help us win.

First, recall that both winning and losing are defined relative to the turn number. For instance, the predicate `win(4)` is true if the game is won by the cross player in four or fewer turns. Recall for both winning and losing, we define a separate predicate for each specific configuration of the board that describes a winning, resp. losing situation. An example of a winning configuration is a board of which the first row consists of crosses. This was discussed in Section 2.

Now for each candidate cell, we consider the winning configurations that this cell participates in, and that can be reached from the current state. If the placement of a cross in a cell can't participate in any winning configuration, we disregard that cell as a candidate. For the selected set of candidate cells, the winning conditions will be restricted to those that can actually be reached by playing one of those cells.

3.4 General Losing Conditions

For the defensive approach, we consider all possible losing configurations in the current state, rather than those in which the candidate cell participates. Although we might try to win by aiming for a specific winning configuration, it does not seem natural to prevent losing by avoiding a specific losing configuration.

3.5 Querying the ProbLog Model

Given our list of candidate cells, we will either maximize our probability of winning (aggressive), or minimize our probability of losing (defensive). We first update the ProbLog

program to account for the current state.

- The updated board configuration is defined using the result of our previous move and the opponent's subsequent move.
- In case of the aggressive strategy, the clauses defining the updated winning conditions are defined in terms of the relevant configurations (those including any of the candidate cells). In case of the defensive strategy, the clauses defining the updated losing conditions are defined in terms of all reachable losing configurations given the current state.
- The currently playable options are given by all available cells; the ProbLog program uses these to compute possible future moves by both itself and the opponent. We define these as a uniform distribution for every turn, meaning that we make the simplifying assumption that the opponent plays randomly, and that that our future moves will be random. For a strategy which predicts the moves of the opponent, one could define a different distribution on plays whenever it's the opponent's turn. The distribution given by the initial state - in which all nine cells are available - would be described as follows:

```
1/9 :: play(1,A); 1/9 :: play(2,A); 1/9 :: play(3,A); 1/9 :: play(4,A);
1/9 :: play(5,A); 1/9 :: play(6,A); 1/9 :: play(7,A); 1/9 :: play(8,A);
1/9 :: play(9,A) :- turn(_,A).
```

For each turn this distribution is defined separately, which expresses that at each turn a different cell is selected.

For each candidate cell, we run the updated ProbLog program with evidence stating that we played that cell, and a query for either `win(3)`, the winning predicate for three turns (aggressive), or `lose(3)`, the losing predicate for three turns (defensive). We finally choose the cell that either maximizes (aggressive) or minimizes (defensive) the probability for the corresponding query that gets returned by our ProbLog program.

3.6 Example of a Selection Procedure

1	2	3	x	o	x
4	5	6	n	n	o
7	8	9	x	o	n

Table 2: Board indices (left) and example configuration (right)

The strategies are best illustrated with an example. Consider the configuration displayed in Table 2. It shows a state of the board in which cells 1, 3, and 7 have been marked with a cross; cells 2, 6, and 8 have been marked with a nought, and the other cells are available. Our selection proceeds as follows:

1. **Candidate selection:** We select a subset of candidate cells.

- The Fast strategy would select cells 4 and 5, since these are adjacent to at least two cells marked with a cross.
- The Conquer-the-Board strategy would select cell 9, since this is the only cell adjacent to at most one of the cells marked with a cross.

2. Pruning based on winning conditions (aggressive only):

- Suppose we applied the Fast strategy. Our candidates are 4 and 5. The winning configurations in which cell 4 participates are those with crosses in the first column (indices 1, 4, and 7) or crosses in the second row (indices 4, 5, and 6). However, the latter one is unreachable due to the nought in cell 6.

For cell 5, the winning configurations are those with crosses in the second column, second row, or one of the diagonals. The diagonals are the only reachable winning configurations from this situation. Since both cells can still lead us to a win, we continue with both.

- If we had chosen the Conquer-the-Board strategy, cell 9 could still result in a win through the diagonal, and so we'd continue with cell 9.

3. Check if candidate list is empty: If no candidates are left, we can't win the game anymore, but we can still tie; we choose an available cell at random.

4. Otherwise: cell selection from candidates: Suppose we applied the Fast strategy; our candidates are still 4 and 5. After updating the board (as defined in Section 3.5), we do the following for each cell:

- In case of the Aggressive approach, we include the evidence of choosing the current candidate and query `win(3)`.
- In case of the Defensive approach, we include the evidence of choosing the current candidate and query `lose(3)`.

Finally, we choose the best cell as defined in Section 3.5.

4 Results

To test our different strategies, we simulated 1000 games per strategy against a random strategy which selects random available cells and Louis Abraham's optimal strategy [Abr24]. As a quick note, we want to mention that the way we use ProbLog is slow, and we did not spend much time optimising our code, other than running the simulation for each strategy in a different thread. For example, this simulation of a total of 8000 games (1000 games per our strategy vs. opposing strategy possible pairs) took about 5 hours to run on an M1 MacBook Pro. If you want to test our code using the GitHub repository here, please keep this in mind.

4.1 Playing against the Random Strategy

The results of our strategy played against the random strategy are shown in Table 3. We also show Sun’s results from [Sun24] in Table 4.

	Win rate	Loss rate	Tie rate
Win Fast	70.6%	26.4%	3.0%
Conquer Board Aggressive	66.7%	29.3%	4.0%
Tie Fast	54.3%	40.9%	4.8%
Conquer Board Defensive	58.0%	35.6%	6.4%

Table 3: Our strategies vs. random strategy, over 1000 trials

	Win rate	Loss rate	Tie rate
Regular AI	66%	28%	6%

Table 4: Sun’s regular AI vs. random strategy, from [Sun24], over 10000 trials

In accordance with our expectations, the aggressive strategies, which focused on winning, clearly outperform the defensive strategies, which focused on tying. The defensive strategies do however achieve a higher percentage of ties than the aggressive ones, as we expected. Still, each of our strategies did better than an entirely random one.

Win Fast slightly outperforms the aggressive Conquer-the-Board strategy, both in terms of win rate and loss rate. As for the defensive strategies focusing on minimizing the probability of losing, it appears that the defensive Conquer-the-Board does slightly better than Tie Fast, with 64.4% non-losses against 59.1%.

Overall, Win Fast seems to slightly outperform Sun’s regular AI, while aggressive Conquer-the-Board roughly matches it. Our defensive strategies however perform significantly worse, although they at least outperform a completely random one.

4.2 Playing against the Optimal Strategy

Our results against Louis Abraham’s optimal strategy [Abr24] are shown in Table 5. We also show the results of Cameron Sun’s testing from [Sun24] in Table 6.

	Win rate	Loss rate	Tie rate
Win Fast	36.1%	60.1%	3.8%
Conquer Board Aggressive	38.0%	58.2%	3.8%
Tie Fast	29.0%	67.6%	3.4%
Conquer Board Defensive	32.0%	61.9%	6.1%

Table 5: Our strategies vs. optimal strategy by [Abr24], over 1000 trials

	Win rate	Loss rate	Tie rate
Regular AI	33.0%	59.5%	7.5%
Random strategy	24.7%	71.2%	4.1%

Table 6: Results from [Sun24], playing against the optimal strategy by [Abr24], over around 750-800 trials

Against Louis Abraham’s strategy, our Win Fast and aggressive Conquer-the-Board strategies managed to slightly outperform Sun’s regular AI in terms of win rate, while they matched its loss rate because of their lower tie rate.

However, the defensive strategies performed rather poorly. Tie Fast in particular only managed to slightly outperform a completely random strategy, which in Sun’s testing over around 800 trials achieved a win rate of 24.7% and a tie rate of 4.1%. Tie Fast only improves the win rate to 29.0%. On the other hand, the defensive Conquer-the-Board strategy managed to almost match Sun’s regular AI’s win rate of 33.0% against the optimal strategy.

Overall, the aggressive approach performs significantly better than the defensive approach: trying to maximize the probability of winning leads to better results - even in terms of losses - than trying to minimize the probability of losses. For candidate cell selection, the Conquer-the-Board approach performs slightly better than the Fast approach. Overall, considering our rather small sample size, Win Fast and aggressive Conquer-the-Board seem to perform about the same, with defensive Conquer-the-Board trailing far behind and Tie Fast being our worst performing strategy.

5 Conclusion

In this 1.5-week project, we got a hands-on experience learning the programming language *ProbLog* by modelling the game *Probabilistic Tic-Tac-Toe* by Cameron Sun [Sun24]. This project introduced us to logic programming, as well as a bit of game theory, complexity theory, and refreshed our probability theory. Specifically when talking about ProbLog, we ran into a lot of difficulties in how to express the game, but overall gained a stronger understanding of how ProbLog works as a result, and created a model of the game which we presented in Section 2.

We will briefly mention some of the learning experiences we had about modelling in ProbLog. To begin with, the central piece of the game is the state of the board, and we thought an obvious implementation would be to instantiate `board(x,n,n,n,n,o,n,n,n,0)` for the board shown in Figure 1. However, looking just three moves forwards builds an incredibly large tree of possible states, and is extremely slow. To circumvent this, on advice of our classmates and teacher, we instead split up the board predicate into nine predicates, one for each cell, which is a lot faster.

Another room for improvement is in how the strategies implement the play predicate. We chose to only implement the cells that are available at the *current* turn in the ProbLog program. As a consequence, the program assumes the same uniform distribution for following turns. This is not entirely correct: the available cells should be updated along with the propagation

of the model for the computation of probabilities by the ProbLog program. However, this resulted in difficulties that arose from the possibility of a neutral event happening after choosing a cell. Alternatives that we considered lead to errors concerning stratified negation, which prevents the grounding of the program.

The effect of the non-updated available cells skews the results obtained by *model counting*; the program considers all future states that are reachable from the current state and counts how many of those satisfy the query. Because the available cells are not updated during these computations, the program will consider models in which a cell, that was available at the current turn, gets selected during multiple turns. None of the rules that update the board will then fire: the cell gets selected, but already contains a value. The win and lose predicate will not be satisfied in such models.

As a consequence, those branches do not count towards the number of wins in aggressive strategies, and not towards the number of losses in defensive strategies, i.e., the probabilities outputted by ProbLog are lower than the actual probabilities. We assume that this does not influence the outcomes of the strategies too much, since we compute the probabilities of our query being satisfied for each of the moves that we consider. Given the uniform distribution, the probabilities should be skewed by the same ratio for all possible moves, which means that our algorithm should still select the optimal move.

This project did not consist just of modelling. We also looked at ProbLog as a Python package from [Gro20b], and saw how ProbLog can be run inside of and in conjunction with Python to give us a way of updating the state and facts dynamically, rather than only once using the evidence function in ProbLog. Using our model, we were able to make two offensive and two defensive strategies playing Probabilistic Tic-Tac-Toe come to life in Section 3. Moreover, using Python to facilitate making moves and building the necessary ProbLog programs, we were able to test our strategies in Section 4. The Python implementation details are discussed in Appendix A.

Ultimately, we are happy with how our strategies performed. As an optimal strategy already exists, we have a baseline to see how well our strategies perform against the best strategy, as well as against the random strategy, which we can view as a sort of *worst* strategy (without intentionally trying to lose). Our best performing strategies appear to slightly outperform Cameron Sun’s regular AI. Still, the main goal of the project was modelling the game of Probabilistic Tic-Tac-Toe in ProbLog and gaining hands-on experience with ProbLog, especially given that an optimal strategy has already been found, so again we are happy with this.

In terms of future work, there are two avenues of improvement. One is making the ProbLog code stronger, and the other is to gain more knowledge on the topic of Game Theory. To the first point, we previously discussed the difficulties with how to encode the play predicate. Finding a way to encode these and propagate these forwards naturally would increase the accuracy of the model, and as a result, increase the accuracy of the strategies. The other method of improvement is to take a course in game theory, and learn about zero-sum games, minimax, and more, to have a better intuition of how to define strategies. Finally, it could be interesting to try to recreate the probabilities that the optimal strategy of Louis Abraham [Abr24] uses to make decisions about which cell to attempt in ProbLog.

A Python Implementation

Here, we briefly discuss how we implemented Probabilistic Tic-Tac-Toe in Python. Recall, as discussed in Section 2, that we only use our ProbLog model to compute the probabilities of winning or losing, given that we play certain cells, from a given state of the game - everything else, including updating the state of the game and therefore actually playing the game, is done in Python.

A.1 High-level overview

Our Python implementation consists of three main components that interact with each other. See Figure 2 for a high-level overview of these interactions.

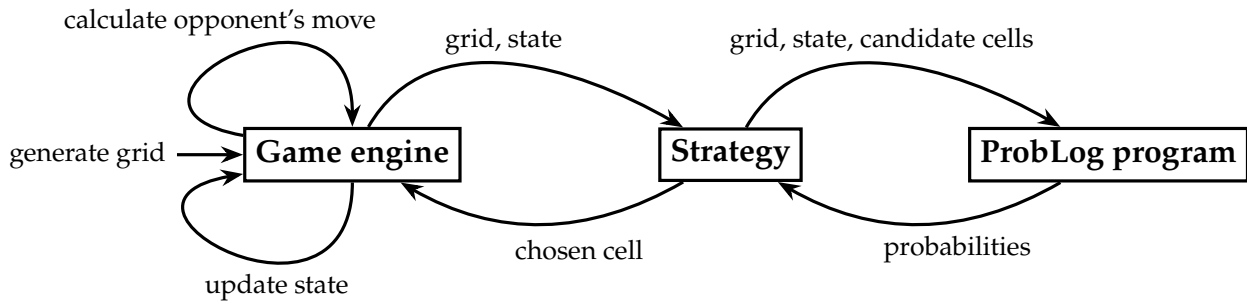


Figure 2: High-level overview of how a single game is played

- A game simulator, which runs a given number of instances of Probabilistic Tic-Tac-Toe playing with one of our given strategies against either Louis Abraham’s optimal one, or a completely random agent. This is where the state of the game is updated. For each run, a game board is randomly generated. For this section of the code, we used Louis Abraham’s code from [Abr24] to generate the board, update its state, and for the optimal strategy.
- Concrete strategies, which given a grid and its state compute the optimal cell to be played next. This is done as described in Section 3: at a high level, a strategy selects a subset of candidate cells, updates the ProbLog model, queries it for the probability of winning or losing given that a candidate cell is played, and selects the optimal cell to be played and sent back to the main game engine.
- An interface for our ProbLog model, which the strategies can partially update with new information, and query for the probability of winning or losing, giving as evidence that a certain cell has been played. This was described in Section 2.

A.2 Some implementation details

We now briefly describe some more technical implementation details. All the main components described above are represented as objects. Here, we provide a fragment of their interface.

```

class Game:

    def __init__(self, games, opposing):
        pass # see code

    def simulate(self, player, strategy):
        gameswon_x, gameswon_o, games_tied = 0, 0, 0
        for _ in range(self.games):
            grid = louis.generate_grid()
            s = strategy(grid)
            initial_state = (None,) * 9
            result = self._play_one_game(grid, player, initial_state, s)
            # accumulate results
        return gameswon_x, gameswon_o, games_tied

class Strategy(ABC):

    def __init__(self, grid, max_turns = 3):
        self.grid = grid
        self.max_turns = max_turns
        self.problog_program = ProbLogProgram(grid, max_turns)

    def run(self, state):
        ### pipeline: update ProbLog program -> query it -> find optimal cell
        pass # see code

class ProbLogProgram:

    _program = {} # internal representation, to allow for easier access to
        different sections of the ProbLog program

    def __init__(self, grid, max_turns = 3):
        self._grid(grid)
        self._init_turns(max_turns)
        self._moves()
        self._end_conditions()

    def query(self, query, evidence = None):
        pass # see code

    def update_board(self, new_board):
        pass # see code

    def update_end_conditions(self, *conditions):
        pass # see code

    def update_play(self, prob_dist):
        pass # see code

```

Strategy consists of an implementation of the Strategy pattern, to allow for variations of a family of algorithms: namely, choosing a cell from a grid and a state, which is precisely what the run function does. As the body of the run function was largely the same for all

our concrete strategies, and the steps of the algorithm were also the same at a high level, we later refactored run into the parent abstract Strategy class as a template method, and implemented certain steps of the algorithm differently for each of our concrete strategies.

Finally, it is worth mentioning that the ProbLogProgram class uses our custom implementation of parts of the KU Leuven ProbLog library [Gro20b]. More specifically, we did not use their Python data structures as we found it hard to keep track of all their types, given Python’s dynamic type system. Moreover, their AnnotatedDisjunction function did not allow for an empty body, which we thought we needed for an earlier version of our ProbLog model. This prompted us to write a small library of functions to build ProbLog statements, which is contained in problog_utils.py. While we think that our library has the advantage of very good readability, all it does is basically string formatting: all the syntactic building blocks of a ProbLog program are represented as strings, and there can therefore be no type checking. All the functions must be used carefully, and as described in their documentation, to avoid runtime errors in ProbLog. The ProbLog program is in the end naturally built from a string using PrologString.

References

- [Gro20a] KU Leuven DTAI Research Group. *ProbLog*. 2020. URL: <https://dtai.cs.kuleuven.be/problog/> (visited on June 19, 2024).
- [Gro20b] KU Leuven DTAI Research Group. *ProbLog*. 2020. URL: https://dtai.cs.kuleuven.be/problog/tutorial/advanced/01_python_interface.html (visited on June 26, 2024).
- [Abr24] Louis Abraham. *Solving Probabilistic Tic-Tac-Toe*. 2024. URL: <https://louisabraham.github.io/articles/probabilistic-tic-tac-toe> (visited on June 19, 2024).
- [igp24] igpay. *Show HN: Probabilistic Tic-Tac-Toe*. 2024. URL: <https://news.ycombinator.com/item?id=40635397> (visited on June 19, 2024).
- [Sun24] Cameron Sun. *Probabilistic Tic-Tac-Toe*. 2024. URL: <https://www.csun.io/2024/06/08/probabilistic-tic-tac-toe.html> (visited on June 19, 2024).