Failure-based reasoning

James A. Riechel*

November 2, 2007

Abstract

We solve the game of tic-tac-toe using a case-based reasoning system, where a database of failed game positions or boards, and their immediate and obvious moves which led to failure, are maintained. The system begins with a null or empty database of failures, and only a single heuristic is implicit in our case-based reasoning system: play random moves excluding moves in our failure database which led to failure in the past. The system plays itself, alternatively playing moves for both X and O, and the failure database increases in size until it becomes stable and the game of tic-tac-toe is "solved."

1 Introduction

The hidden complexity of an apparently simple game like tic-tac-toe makes the game a valid subject for research in artificial intelligence. One measure of complexity in the domain of games is the number of unique games that can be played from the starting position to all possible end positions following the legal rules of the game. Probability theory, discrete mathematics, and combinatorics can measure this complexity of tic-tac-toe, given one assumption. We assume that all games complete with no unoccupied squares, even if 'X' or 'O' achieves an early victory before all nine squares are occupied. Given this assumption, there are

$$\left(\begin{array}{c} 9 \\ 5 \end{array} \right) 5! \left(\begin{array}{c} 4 \\ 4 \end{array} \right) 4! = 9! = 362,880$$

possible and unique games of tic-tac-toe. Over the course of a game, 'X' chooses five squares to play in out of nine, and he can choose to play those five moves in any order. Over the courses of a game, 'O' chooses four squares to play in out of the four remaining squares available to him, and he can choose to play those four moves in any order. There are exactly 362,880 possible and unique games of tic-tac-toe. A number of this order of magnitude is today considered tractable by computer scientists, but artificial intelligence research in tic-tac-toe is appropriate because it is both tractable and sufficiently complex.

*jamesriechel@gmail.com

2 Methodology

Our system employs no explicit heuristic to aim for any particular result in tic-tac-toe. It plays neither to win, lose, or to draw. Also, no type of search algorithm is used to aid in the selection of a move. The system knows how to play a legal game of tic-tac-toe, and the system knows whether the current game is still in progress, whether the current game has ended in a tie-game or draw, whether the current game has finished with 'X' as the victor, or whether the current game has ended with 'O' as the victor.

When a game is lost, or when a loss is unavoidable, a failure assignment is made. The failure assignment identifies the losing move in the critical position in which the losing move was made. The critical position and the losing move made in the critical position are added to the failure database.

On any given turn in any particular game of tic-tac-toe, whether it is X's turn to play, or whether it is O's turn to play, we generate all possible legal moves for the current player. From this list of all possible legal moves, moves which have led to failure in the past are excluded. Only a simple search of the failure database is required to exclude these moves. In this manner, our system, as best as possible, avoids failure. Failure in the context of tic-tac-toe is, of course, loss of the current game by the current player. Of the remaining available moves for play which have not led to failure in the past, one is chosen randomly and played.

2.1 Failure assignment

When a game is lost, or when loss is unavoidable, we identify the critical position and the losing move made in the critical position which led to the loss. The critical position and its losing move are then added to the failure database. There are two types of failure in tic-tac-toe. First, if a move is played, and then the opponent plays a move which immediately wins, the move which allowed the opponent to win is identified as a failure and added to the failure database along with the critical position. Second, if the current player has no moves available which don't lead to failure, then the previous move by the same player is identified as a failure and added to the failure database along with the critical position.

2.2 Choice of move

Given a current game or tic-tac-toe board, we first generate all possible legal moves for the current player. Next, for each possible and legal move available for play by the current player, we compare the current board and the current move under consideration to determine if the current board and the current move under consideration appear as a failure in the failure database. If the current board and the current move under consideration appear as a failure in the failure database, we exclude the current move under consideration as a choice for play.

If, as a result of this process, the current player is left will no moves which will not lead to failure, we record the last move by the current player as a failure, and regenerate the full list of legal moves for the current player, even though all of them have been previously identified by the system as failures. The system does not "resign," and the game continues. It is possible that even though all moves could lead to failure, the opponent may choose to draw instead. The system, alternatively playing moves for both 'X' and 'O', chooses random moves which have not led to failure in the past. If we assume the failure database represents all possible failures in tic-tac-toe, the system playing either for 'X' or for 'O', plays for a draw or better. A draw in tic-tac-toe is, of course, a tie-game, or what children call a "cat's game." Our system employs a failure-avoidance strategy. It may not always avoid loss, but, as best as possible, it plays for a draw or better.

So, legal and possible moves for the current player are excluded and possibly re-introduced as described above. Of the moves that remain, one is chosen randomly and played.

3 Implementation

In all cases, our system beings with an empty or null database of failures. The system then plays a specified number of games of tic-tac-toe, alternatively playing moves for 'X' and 'O' until each game is complete. If a game ends in a failure, that is, if either 'X' wins or 'O' wins, or a loss by 'X' or by 'O' is unavoidable, the critical position and the losing move made in the critical position are identified and added to the failure database.

Running statistics are collected and maintained, including: the total number of games, the number of tie-games, the number of times a game ends with 'X' as the victor, the number of times a game ends with 'O' as the victor, and the number of times a possible failure was avoided by excluding a move from play that might lead to failure. After all games are played, the failure database is written to a file, and the size of the failure database is reported along with the other running statistics.

4 Some theory

There are some theoretical bounds on the size of the failure database our system can create. 'X' has nine choices for play on his first move, 'O' has eight choices for play on his first move on a board that already has one 'X' placed in one of the nine squares, etc. Now assume that for the game of tic-tac-toe there exists a ratio $r \in [0.0, 1.0]$, where r is a ratio or percentage of moves available either to 'X' or to 'O' on any given turn in any given game that lead to failure. If such a ratio r exists, then we know, at least theoretically, the size of the failure database. Assume s is the theoretical size of our failure database.

$$s = 9r \begin{pmatrix} 9 \\ 0 \end{pmatrix} \begin{pmatrix} 9 \\ 0 \end{pmatrix} + 8r \begin{pmatrix} 9 \\ 1 \end{pmatrix} \begin{pmatrix} 8 \\ 0 \end{pmatrix} + 7r \begin{pmatrix} 9 \\ 1 \end{pmatrix} \begin{pmatrix} 8 \\ 1 \end{pmatrix} + 6r \begin{pmatrix} 9 \\ 2 \end{pmatrix} \begin{pmatrix} 7 \\ 1 \end{pmatrix} +$$

$$5\text{r} \begin{pmatrix} 9 \\ 2 \end{pmatrix} \begin{pmatrix} 7 \\ 2 \end{pmatrix} + 4r \begin{pmatrix} 9 \\ 3 \end{pmatrix} \begin{pmatrix} 6 \\ 2 \end{pmatrix} + 3r \begin{pmatrix} 9 \\ 3 \end{pmatrix} \begin{pmatrix} 6 \\ 3 \end{pmatrix} + 2r \begin{pmatrix} 9 \\ 4 \end{pmatrix} \begin{pmatrix} 5 \\ 3 \end{pmatrix} +$$

$$r \begin{pmatrix} 9 \\ 4 \end{pmatrix} \begin{pmatrix} 5 \\ 4 \end{pmatrix} = 19,107r$$

The maximum value of s occurs when $r = 1.0$. Call this maximum value s_{max}. Obviously, $s_{max} = 19,107$. In practice, and in our implementation, we know the experimental value of s, called s_{exp}, , and we can solve the above equation for the experimental value of r, r_{exp}.

$$r_{exp} = \frac{s_{exp}}{s_{max}}$$

Our experimental results are reported later, including the experimental values of s_{exp}. From these experimental results, we compute a value for r_{exp} for the game of tic-tac-toe, assuming the ratio r exists.

So, theoretically, in the worst case, for the game of tic-tac-toe, the failure database will be no larger than $19,107$. In this worst case, all possible moves for either 'X' or 'O' on any given turn in any given game lead to failure. This is a logical impossibility, since in any single completed game of tic-tac-toe, 'X' and 'O' can't both lose in the same game. Just think of $19,107$ as an upper-bound on the size of our failure database, s, in the game of tic-tac-toe. The actual size of the failure database is less than half its maximum size, and we report these results later.

5 Results

Our system was tested four times, each time starting with a null or empty failure database. In order, the system played 1,000 games, 10,000 games, 100,000 games, and 1,000,000 games of tic-tac-toe. Figure 1 shows, in table format, some of the recorded statistics for each of the four tests. Figure 2 shows the number of tie-games, X-victories, and O-victories as a function of the number of games played. Figure 2 is plotted on a logarithmic versus logarithmic scale, so all the data is visible.

It is obvious that our system learns how to play tic-tac-toe, or at least, it learns how not to lose. Notice that the statistics for X-victories and O-victories are the same when the system played either 100,000 games or 1,000,000 games, except for the number of failures avoided, since a different number of games

Games	1,000	10,000	100,000	1,000,000
tie-games	138	4,493	93,787	993,787
X-victories	558	3,350	3,824	3,824
O-victories	304	2,157	2,389	2,389
failures avoided	705	76,570	1,571,823	16,635,566

Figure 1: Performance

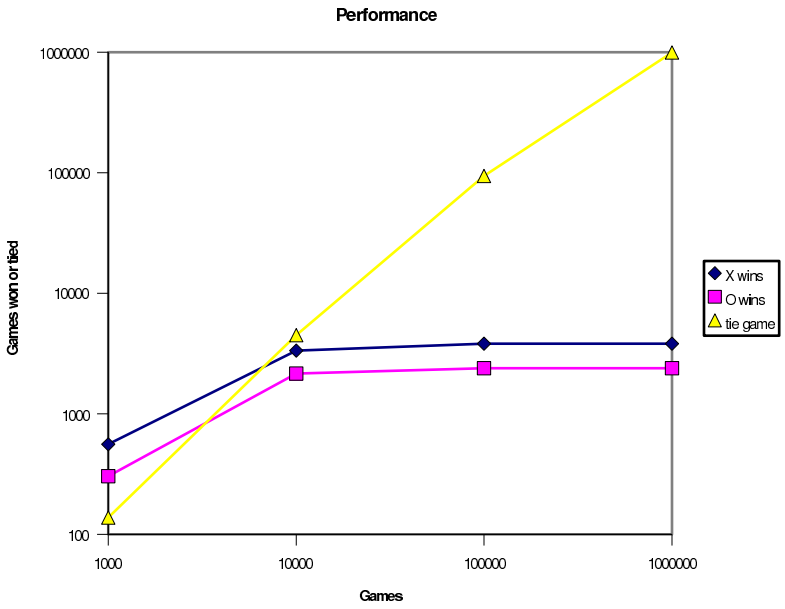

Figure 2: Performance (logarithmic vs. logarithmic)

were played in each case. When either 100,000 or 1,000,000 games were played, the usual result was a tie-game, and a huge number of failures were avoided in both cases. I cannot claim that there are only 3,824 unique X-victories in the game of tic-tac-toe, and I also cannot claim that there are only 2,389 unique O-victories in the game of tic-tac-toe, since our system chooses moves randomly, using a random number generator which I will not describe. Since the choice of moves is random, we cannot support either claim.

Figure 3 shows, in table format, the size of the failure database at the end of each experiment. Figure 4 plots the data from Figure 3. The failure database grows quickly and stabilizes.

Games	1,000	10,000	100,000	1,000,000
Size of failure database	862	5,656	6,409	6,409

Figure 3: s experimental

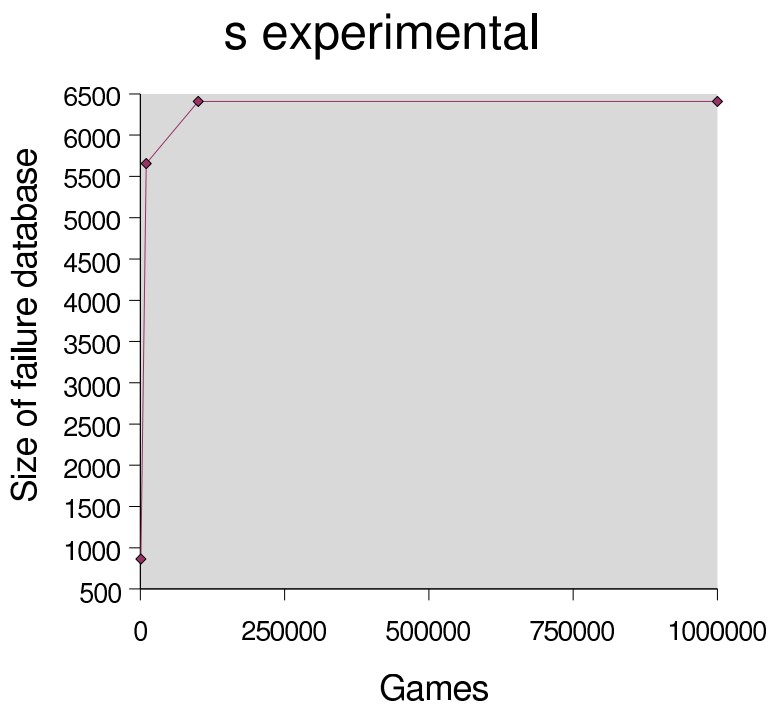

Figure 4: Games vs. Size of failure database

After playing 1,000,000 games, the failure database contains 6,409 unique failures. Again, even though the size of the failure database is the same when either 100,000 games or 1,000,000 games were played, I cannot claim that the failure

database of size 6,409 includes all possible and unique failures in the game of tic-tac-toe. Again, since moves are chosen randomly using a random number generator which I will not describe, the claim cannot be supported.

Remember, s_{exp} is our best estimate of the size of the failure database after our best experiment, so $s_{exp} = 6,409$. From s_{exp} we can compute the ratio r_{exp}, assuming the ratio r exists. Using the formula from the theory section, and using $s_{exp} = 6,409$, we find that $r_{exp} = .3354$ or $r_{exp} = 33.54\%$. In other words, if the ratio r exists, then whether it is X's turn to play, or O's turn to play, 33.54% of the legal moves available to 'X' or to 'O' on any given turn of any given game of tic-tac-toe, will result in a failure or loss of the game.

6 Summary

The methodology we employ in the domain of tic-tac-toe is generally applicable to many, if not all, domains of interest to researchers in artificial intelligence, and not just generally applicable to some, but not all, of the domains of games. Applications in every domain of interest to researchers in artificial intelligence have choices. Some of these choices lead to failure, however failure is defined. Whether the choices made are optimal or not, intelligent software should avoid making choices which lead to failure. One way to avoid failure is to not repeat the same or similar mistake made in the past. A database, however represented, can store these past mistakes, and the database can be accessed by intelligent software to determine if a current choice available to the intelligent software might lead to a failure. In this way, choices available to the intelligent software can be excluded, leaving only choices which, we hope, will not lead to failure because the given choice is not represented in some form in the failure database. After this process is completed, another artificial intelligence system can then make a choice between the remaining options, with the options leading to failure, hopefully, already removed.

A Failure database

The failure database, as implemented in our system, is a simple text file. For the purposes of example, Figure 5 shows the first three failures from the failure database in the experiment where 1,000 games of tic-tac-toe were played.

The first line of the failure database specifies the number of failures in the failure database. Then for each failure in the failure database, a unique tic-tac-toe board is specified along with the coordinates of the move which lead to failure in the specified tic-tac-toe board, (y, x). The coordinates are in $(row, column)$ format, where $y, x \in \{0, 1, 2\}$. A unique tic-tac-toe board is specified first by the player, 'X' or 'O,' whose turn it is to move, and then by the tic-tac-toe board itself. The entire file or failure database, in our implementation, is a plain text file.

```
862 failures
O to play
 X     O
 X  X  O
 O  X
(0, 1)
O to play
 X
 X  O
    O  X
(0, 2)
X to play
    X  X

    O  O
(1,1)
```

Figure 5: Sample failure database