Joel Kemp
Topics in AI, Fall 2011
Programming Assignment Report
Tic Tac Toe Genetic Algorithm Solver

Tic Tac Toe Genetic Algorithm Solver

Introduction

The Genetic Algorithm (GA) solver for the Tic Tac Toe problem was implemented in Matlab 2011b. No optimization toolkits were used in the experiment. The solver can be run via the startup file tictactoe.m.

It is assumed that the standard rules for Tic Tac Toe are known and will not be discussed in this report.

The solver attempts to find a winning strategy for player X – although, winning strategies could also be found for the adversary: player O. The solver halts when a winning state has been found or when the number of iterations exceeds the threshold; at which point the closest possible solution is displayed (if it exists and is valid).

The system was built to model a basic Genetic Algorithm with the standard processes:

1. Initial Population Generation	(tictactoe.m)
2. Fitness Function	(fitness.m)
3. Selection	(selection.m)
4. Crossover	(crossover.m)
5. Mutation	(mutate.m)

The report touches on each of the processes in relation to the tic tac toe solver.

Initial Population Generation

Tests were run with initial population sizes of 2, 3, 4, and 9 solution strings/states. A solution is a 3x3 game board with randomly populated moves and non-moves (empty cells).

The game board is a multidimensional array of integers where each cell contains a number representing a player's move. The possible element values are as follows:

- 0 = empty cell
- 1 = player X's move
- 2 = player O's move

Internally, we use integers, however, the typical markers (characters) are shown when displaying the board's state in a grid-like fashion.

Fitness Function

The custom fitness function for this solver assigns a score to each of the states it is given. The score/fitness of the state is dictated by a few rules: whether or not the state has an appropriate number of played moves, and the state's proximity to a win condition.

A standard game of tic tac toe involves alternating moves/plays by players X and O. Hence, at any point in the game, game states should contain either an equal number of moves for both players or a difference of one marker. If a generated state contains an abnormal number of markers for a given player, then that state is given a fitness of zero to prevent its inclusion in the future generation of solutions.

Player X can win by getting three markers along a particular direction; in particular, there are three directions: horizontal, vertical, and diagonal. The state is evaluated in all three directions by maintaining a maximum score for each direction. This score represents the maximum number of markers in the direction.

In the horizontal direction, there are three rows where a win is possible. A sample game state looks as follows:

X		X
X	O	
O		O

Table 1 - Sample game state

The scores associated with each row would be 2, 1, and 0 from the top to bottom. Thus, the row most likely to yield a win would be the top row with a score of 2; this is the maximum score for the rows.

Similar analyses are done for vertical and diagonal directions. The maximum scores are then added together to yield an overall fitness for the state. The overall fitness for the game state shown in Table 1 would be 5 – as the maximum row score is 2; the maximum diagonal score is 1, and the maximum vertical (column) score is 2. The sum of these scores yields 5.

Selection

For the selection of the parents that would be used in generating offspring solutions, fitness statistics were computed for all of the original states and their associated fitness scores.

Roulette wheel selection was applied using the probabilities computed in this stage. The result of this selection is a set of solutions to be used in the steps of crossover and mutation.

Crossover

From the selection step, we obtained a set of fit parents to be used in producing offspring. The production of offspring would occur via the crossover process.

During crossover, a mate was randomly selected for each of the parent solutions. These two states were then crossed to produce a new, child state. The actual crossing involved randomly selecting a row from both states to be swapped. After the row swapping occurred, one of the new states would be chosen to be the offspring – typically, this was just the first state of the two in comparison.

At the end of the crossover process, we obtained a set of offspring – whose cardinality was equal to the cardinality of the set of initially generated solutions.

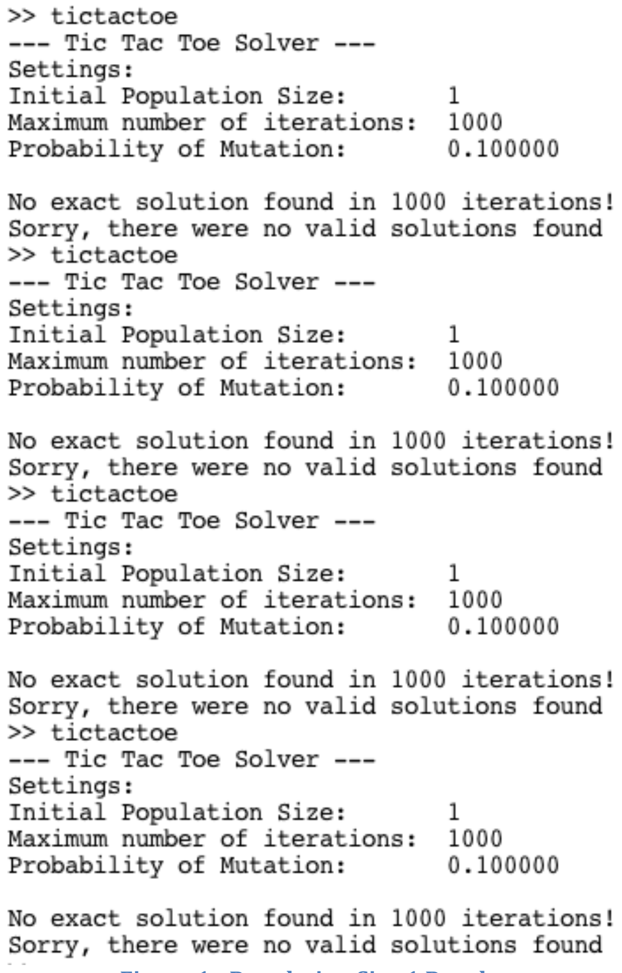
Mutation

Once the offspring solutions were generated, a probabilistic mutation occurred. Each state was given an equal likelihood for mutation; at which point another roulette wheel selection occurred to determine which child was mutated.

The process of mutation involved randomly choosing an un-played (blank) cell and playing X's marker in that location. In retrospect, this is a very simple mutation and could be expanded to modify more than a single state; in addition, the number of moves made (and possibly erased) could be increased to change the progression of the tic tac toe game states.

The result of the mutation process was a set of states similar to the set of offspring states where a single state had been slightly modified.

Results

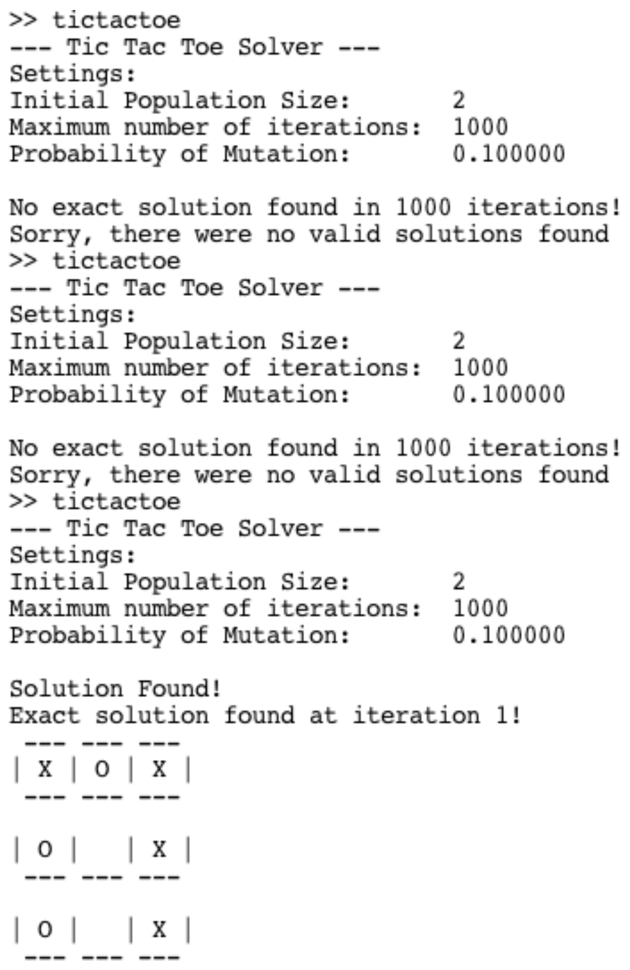


```
>> tictactoe
--- Tic Tac Toe Solver ---
Settings:
Initial Population Size:       1
Maximum number of iterations:  1000
Probability of Mutation:       0.100000

No exact solution found in 1000 iterations!
Sorry, there were no valid solutions found
>> tictactoe
--- Tic Tac Toe Solver ---
Settings:
Initial Population Size:       1
Maximum number of iterations:  1000
Probability of Mutation:       0.100000

No exact solution found in 1000 iterations!
Sorry, there were no valid solutions found
>> tictactoe
--- Tic Tac Toe Solver ---
Settings:
Initial Population Size:       1
Maximum number of iterations:  1000
Probability of Mutation:       0.100000

No exact solution found in 1000 iterations!
Sorry, there were no valid solutions found
>> tictactoe
--- Tic Tac Toe Solver ---
Settings:
Initial Population Size:       1
Maximum number of iterations:  1000
Probability of Mutation:       0.100000

No exact solution found in 1000 iterations!
Sorry, there were no valid solutions found
```

Figure 1 - Population Size 1 Results

With the population size equal to 1, it becomes highly unlikely that a solution can be found if not by an absolute miracle! As seen in Figure 1, the solver yields no solutions. In particular, the closest matches were not close at all – containing states where either both players won or there was an imbalance of the number of moves made by a particular player. These errors were made possible as a by-product of the randomness of the mutation and crossover processes.

Instead of showing erroneous solutions, the solver states that no valid solutions were found.

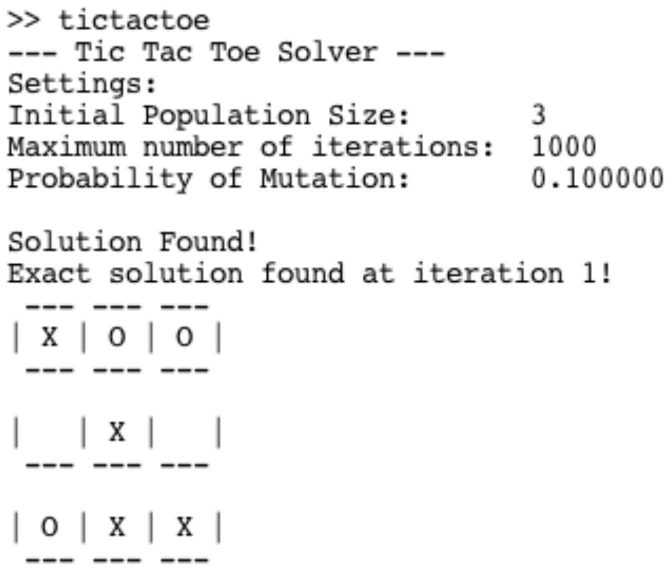


```
>> tictactoe
--- Tic Tac Toe Solver ---
Settings:
Initial Population Size:       2
Maximum number of iterations:  1000
Probability of Mutation:       0.100000

No exact solution found in 1000 iterations!
Sorry, there were no valid solutions found
>> tictactoe
--- Tic Tac Toe Solver ---
Settings:
Initial Population Size:       2
Maximum number of iterations:  1000
Probability of Mutation:       0.100000

No exact solution found in 1000 iterations!
Sorry, there were no valid solutions found
>> tictactoe
--- Tic Tac Toe Solver ---
Settings:
Initial Population Size:       2
Maximum number of iterations:  1000
Probability of Mutation:       0.100000

Solution Found!
Exact solution found at iteration 1!
 --- --- ---
| X | O | X |
 --- --- ---

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

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

Figure 2 - Population Size 2 Results

The experiment with an initial population size of 2 yielded a solution (at least). Of course, 2 strings is also not a lot to work with, and so the solver is unlikely to find valid solutions for the problem – even in a large number of iterations.

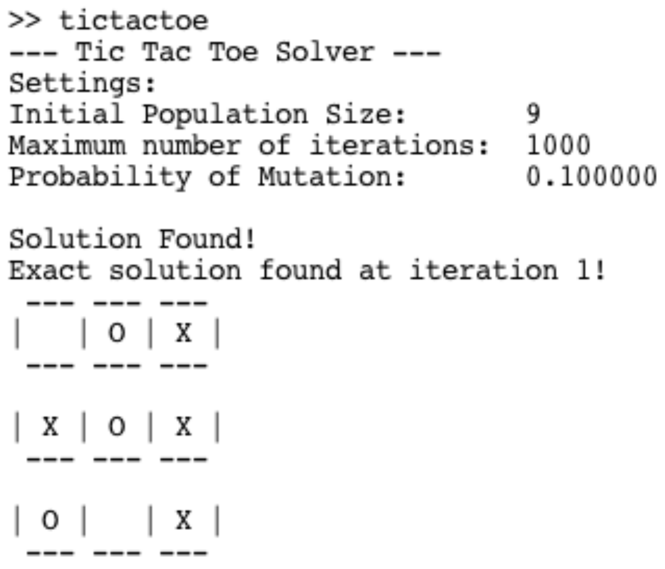
Of course, a solution was not found during the first 1000 iterations. After resetting the solver, a valid solution was luckily found and displayed to the user.



```
>> tictactoe
--- Tic Tac Toe Solver ---
Settings:
Initial Population Size:       3
Maximum number of iterations:  1000
Probability of Mutation:       0.100000

Solution Found!
Exact solution found at iteration 1!
 --- --- ---
| X | O | O |
 --- --- ---

|   | X |   |
 --- --- ---

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

Figure 3 - Population Size 3 Results

A population size of 3 game states/solutions yielded a valid solution on the first iteration. It is highly likely that a solution can be found when performing the genetic processes on numerous solutions with a large number of iterations.



```
>> tictactoe
--- Tic Tac Toe Solver ---
Settings:
Initial Population Size:       9
Maximum number of iterations:  1000
Probability of Mutation:       0.100000

Solution Found!
Exact solution found at iteration 1!
 --- --- ---
|   | O | X |
 --- --- ---

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

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

Figure 4 - Population Size 9 Results

The solver does an increasingly better job of finding results each time when the number of initial solutions is set to a size greater than 3. This can be seen from the valid solution found in the above figure.

Discussion

Again, the processes described above are repeated until the maximum number of iterations has been reached or a winning state has been found.

When we end up without a valid, close solution, it can be attributed to the very linear progression of the states towards a win-state. If and when all of the states are full (contain moves in all of the elements), there are no further mutations that can be made (in the current mutation function) to effectively produce varied offspring.

The modifications made to the solutions are not as severe as they could be – more drastic crossovers and mutations could create a wider range of results and even result in a reverse progression of the game for some states. In particular, mutation could be enhanced to erase several moves in addition to playing new moves, which would be similar to undoing one of the moves and backtracking to potentially better solutions.

This experiment shows the effectiveness of genetic algorithms for solving a seemingly simple problem of Tic Tac Toe.