A Tic Tac Toe Learning Machine Involving the Automatic Generation and Application of Heuristics

Thomas Abtey
SUNY Oswego

Abstract
Heuristics programs have been used to solve problems since the beginning of artificial intelligence research. The program described in this paper uses a simulation-based generation technique for developing winning heuristic moves in the game of tic tac toe, which it is able to apply automatically during a game session.

Keywords: Machine learning; rule bases, heuristics, game-playing, simulation, tic tac toe.
1. Introduction

Rule-based systems offer an easily-defined system to solve complex problems. In artificial intelligence, rule-based programs have been used for game-playing to provide an *if-then* set of predicates to develop reactionary behavior. These rules are contained as an antecedent/consequent pair, as in: if X, then Y [1].

Machine learning is a subfield of artificial intelligence with aims to design systems which alter their own characteristics based on experience. This includes behavior, in the sense of answering questions or performing a task (as in the case of this paper's game-playing program) [2].

The game of tic-tac-toe is a well-known, simple game consisting of a 3x3 grid of slots. Two players take turns filling in these slots with either an 'X' or an 'O' depending on who's turn it is – X-Player uses X's, while O-Player uses O's (see Figure 1.1 for example). X always goes first. The end of the game is determined when one of the players gets three of their letters lined up in a row, or all of the spots are filled up (this is a draw).

Figure 1.1, an in-progress game of tic-tac-toe:

The program (to be described in more detail below) is designed to use a series of simulations to pick and chose the winning strategies. This simulation includes a set of two agents which randomly choose which move they want to make next. From these simulations, rules are generated as lists of moves whereby the heuristic-machine-player may access and apply to play a game.

2. Related Research

There have been reported many successes in using rule-based systems to play zero-sum (games with perfect information) games like tic-tac-toe. A classic example is found in a paper by Arthur L. Samuel in 1959 [5]. The objective in that paper was to design a program that could beat its creator at a game of checkers. Checkers was chosen because it had a well-defined goal, all of the information regarding rules, pieces, and board were openly available, and because its simplicity allowed the program to be noted more on its learning ability than anything else. Through generalization of board states and a lookahead tree, the checkers game was able to play at a challenging level [5]. The writer's use of tic-tac-toe also allows a better focus on learning, rather than gameplay.

Self-play and exploration techniques (featured in more detail below) also show success in game-playing programs. By allowing the program to simulate games and develop its own strategy from an unintelligible form, valuable learning can occur [3, 4]. The writer’s program uses a large volume of randomized games to formulate its rulebase. An obvious next step to generating large databases of rules would be to add some
measure of value to each of the rules to determine their usefulness in context [7, 11]. This has been achieved by using reinforcement learning parameters to the rules to give them weight values [6] that judge how well a move or series of moves in a game will grant a win [4].

3. Approach

The tic-tac-toe playing program was designed in Common LISP (or CLISP), heavily used was the CLOS - Common LISP Object System [8]. Each of the agents created was an object of the same type - player. These players were assigned behaviors for playing the game in different ways. The random-machine-player was able to select its next move at complete random from a list of possible moves. The heuristic-machine-player could select from a previously generated rule-base to more intelligently chose its next move. The heuristic-learning-machine-player was the same as the heuristic-machine-player, except that after each game it would add a rule to its base if it had won the game. A human-player was simply able to accept a users input to chose the next move (this was used for experimentation purposes).

Learning took place through a simulation of n amount of games (where n was inputted by the experimenter) between two random-machine-players. Through these games, rules were generated from the list of moves played in a game where the machine had won.

4. Knowledge Representation

The program represented the game board as a list of values in the form of ‘(nw n ne w c e sw s se)’. A sketch of this board is featured below in figure 4.1. These values corresponded to compass relations as if the board were a geographic map.

Figure 4.1, board representation:

<table>
<thead>
<tr>
<th>nw</th>
<th>n</th>
<th>ne</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>c</td>
<td>e</td>
</tr>
<tr>
<td>sw</td>
<td>s</td>
<td>se</td>
</tr>
</tbody>
</table>

Plays are represented as lists of move combinations between two players, such that the list is of the form ‘(X1 O1 X2 O2 X3 O3 X4 O4 X5)’. This is to outline specifically who makes what move when, and to be able to determine who won first (by examining down the list).

Rules are plays which have shown to be a winning combination of moves.

5. Program Abstractions

Some psuedocode for the methods most salient to this paper’s topic of heuristic rule-base learning.

A random-play method to randomly select moves between two machines to create a
play list:

1. set **play** to nil
2. set **avail** to (nw n ne w c e sw s se)
3. set **play-so-far** to nil
4. set **player** to (x o x o x o x o x)
5. begin loop while **player** does not equal nil
6. if **player** equals x
7. then set **move** to a random move for x and add **move** to **play**
8. if **player** equals o
9. then set **move** to a random move for o and add **move** to **play**
10. destructively move to the next item in list **player**
11. end loop
12. set **player-so-far** to **play-so-far** with **move** at the end
13. return **play**

The random-play-and-learn method which calls for a full play between two random machines and decides whether its worth turning into a rule or not:

1. set **p** to the return value of **random-play**
2. set **result** to the return value of an analysis of **p**
3. if **result** equals a win
4. then add **result** as a rule to the rulebase

The add-rule method which takes a heuristic-machine-player and a play and adds the play as a rule to the heuristic-machine’s rulebase:

1. set **p** to a heuristic-machine-player
2. set **play** to a winning play
3. append **play** as a rule to the rulebase of **p**

The applicablep method which is returns a boolean value if the rule can be used for the current **play-so-far** list:

1. set **the-play** to the **rule**
2. if **the-play** matches **play-so-far**
3. then return true
4. else return nil

The make-heuristic-move method which choose from the available rules in the base for the next move to make.
1. set move to the next move from a rule
2. if move equals nil
3. then set move to a random move
4. remove move from *avail*
5. return move

The select-from-rule-base applies a rule (if there is one) for a given heuristic-machine-player p:

1. set rule-base to the heuristic-machine-player’s rulebase
2. loop while there is more rules to look at in rule-base
3. if a rule is applicable, select it
4. increment to next rule in rule-base list
5. end loop

6. Results

Some demonstrations of the before and after statistics of game wins, losses, and draws by the heuristic-learning-machine-player (who will always be X). The CLISP commands are of the form (demo-hlm-vs-random nlt ntt verbose) where demo-hlm-vs-random is the method call to play a game between a heuristic-learning-machine and a random-machine, nlt is the number of times to play in a simulation to generate rules, ntt is the number of times to play against a random machine to test learning statistics, and verbose is a boolean value of t or nil which is for debugging purposes and displaying game states.

First, a very simple demo displaying the board states of each play simulated or played:
(demo-hlm-vs-random 3 2 t)
HEURISTIC LEARNING MACHINE PLAYER ...
name = HLM
rules ...
.
(S C SW NW SE N E NE W)

O | O | O
---------
X | O | X
---------
X | X | X
W
(C E S W SW NW NE SE N)
O | X | X
HEURISTIC LEARNING MACHINE PLAYER ...
name = HLM
rules ...
(NW E W S SW NE SE C N)

stats before learning = ((W 1.0) (L 0.0) (D 0.0))

Now, a demonstration of a very large creation of rules (over 10,000 plays) and many games (1,000) to provide results of after-learning win rate:

(demo-hlm-vs-random 10000 1000 nil)
stats before learning = ((W 0.59) (L 0.287) (D 0.123))
stats after learning = ((W 0.662) (L 0.234) (D 0.104))

7. Discussion

As shown in the statistics above, substantial learning will lead to a substantial increase in winning rate. For the point of learning by experience and using those experiences to judge future situations correctly, it is quite a success. It is still not guaranteed to be a win or even close (in the 90th percentile), but is progress over simply randomly selecting moves.
8. Future Work

The total possible combinations for a full-length (filling all nine slots) tic-tac-toe game is 9! or 362880 [9]. Although only a portion of these would be definite winning plays for the machine, to enumerate every single possible game would take quite a while. But because some plays are simply rotations of one another in terms of board configuration, I think this number is much smaller for the needs of creating heuristics.

Interesting work has been with the help of a genetic algorithm design. Anurag Bhatt, Pratul Varshney, and Kalyanmoy Deb at the Kanpur Genetic Algorithms Laboratory in Kanpur, India have created a scheme for developing no-loss strategies. They have produced 72 no-loss strategies for tic-tac-toe [10]. The writer would like to find a way to incorporate their findings as heuristics in the tic-tac-toe program described in this paper.

A final word on future work -- the heuristics used by the machine are haphazardly listed, with no relation between them or weight values of any kind given to their successfulness in winning games. It would be reasonable to design the system to provide the very best rules at the top of the search results in a later addition to the program.

8. Conclusion

The statistics speak for themselves -- the machine was able to learn and apply its newfound rules to other board instances. Although not a perfect tic-tac-toe playing program, it does quite well against the other agents (the random machines) in the program. Any program can be more accurate, or a better game-player, and so if the author decided to continue on with the program’s engineering, there would still be alot of work to be done before it would be close to a no-loss player.