Genetic Cycles: Creating a GP Gamer with Flair

Richard Banister
CSC 4510 Machine Learning
7 May 2007

Purpose

The goal of this project was to use a genetic programming approach to automatically generate a game playing artificial intelligence, then to see how changing certain parameters of the generation process affected the "personality" of the resulting programs. The game in question is TRON (alternately known as Lightcycles or Worm), a game in which two players move through an enclosed arena, filling each space they pass through with a wall. If a player hits any wall they explode and their opponent wins. The game was chosen because of its uncertain and open-ended nature. There is no perfect game play algorithm that will guarantee victory every time. However the game is simple enough (one can either turn or go straight, essentially) to make experimentation feasible.

The main goal was to explore how the input parameters of a genetic program affect its output, and to see if the output can be guided in a certain direction by changing the input. How would one generate a player that acts aggressively? How could one generate a defensive player? And will generated players simulate "human play" – that is, can they be mistaken for human players? To investigate these questions, a game engine and a genetic programming API needed to be obtained, genetic programming parameters needed to be determined and programmed, and populations needed to be evolved experimentally.

Engine

The BMTron engine is a simple 2d lightcycles game. It is multi-threaded in nature, running animation and player decisions in different concurrent threads. It features two computer AI players, deemed Computer 1 and Computer 2, being of lower skill and higher skill respectively. It also comes with an experimental third computer player, which supposedly uses a learning algorithm. Computer 3 was sparsely documented and probably not very functional, so it was discarded and replaced with the GP player. Up to four human or computer players can play at a time. The engine also allows for team play, but this option was not considered for the development of evolved programs.

API

The JGAP package was selected as the genetic programming engine. JGAP, previously focusing exclusively on genetic algorithms, recently introduced genetic programming features. Since these features are still quite young, the documentation is in many cases absent or insufficient. However, it was functional and was being used in several applications. For this project JGAP was run with Java 1.5.0_06. JGAP manages a population of individual evolved programs, handling selection, evolution and mutation based on simple guidance from the programmer. It is up to the programmer to provide a fitness function (by extending the abstract FitnessFunction class), commands (by using prefabricated CommandGene classes or implementing new ones) and any variables and constants the program can use. Commands are the genes of the chromosome – the methods that can be combined and reorganized to implement the program.

First Steps

First the BMTron engine would have to be modified for use by JGAP. The only way to measure the fitness of a TRON player is to watch it play. However, running each individual in a large population through the multi-threaded engine, at tens of seconds per match, would be an enormous time consideration. To speed things up, a new mode, dubbed “Invisible Bike” mode, was created; in this mode, the game is no longer multi-threaded but instead moves each player in turn, once per round, and executes games in the background. In this manner, games can now take place in a matter of milliseconds.

Next, JGAP was integrated with the engine and a population specification was created. Each chromosome (individual program) in the population contains a GPPlayer object, which can be passed into a BMTron game to play against the computer. The chromosome itself contains the command sequence that makes up the player’s “personality.” The fitness function instantiates the GPPlayer in the chromosome and runs a game with it. The outcome of the game determines the chromosome’s fitness. The first fitness measurement depended solely on how long the player stayed alive. A game could last, at most, 875 rounds before every square of the grid becomes filled with walls. So the fitness was rated as a score from 0 to 1.0 reflecting the percentage rounds stayed alive out of 875.

The initial possible commands given to the population were `wallAhead()` and `turnRight()`. Due to complications with JGAP, each command had to essentially be a procedure – functions returning a value could not be used. This ruled out even simple statements like `if (`*`condition`*`) then (`*`action1`*`) else (`*`action2`*`)`, since *`condition`* had to be either a variable or the boolean return value of a function. (incidentally, booleans were not fully implemented in JGAP and had to be written by the author in the course of experimenting with the project) So each conditional statement had to be essentially hard-coded – e.g. `if (there's a wall ahead) then (`*`action1`*`)` – removing some of the potential flexibility of a chromosome. Future generations of JGAP may fix this, or enable the fix given better documentation.

Initially small populations with limited chromosome depth were used. These values were increased as the system started producing working candidates. Also, no mutation was allowed over the course of the project. Mutation in genetic programming in JGAP is complicated and was considered outside the scope of the project.

Patterns

The first successful evolved programs made simple patterns. In particular, they formed increasing left-turn spirals. The first program turned only left, making a tight spiral until running headlong into a wall. This program also had a quirk whereby it would panic if it happened to run up alongside an enemy wall, turning back on itself and crashing. This unforced error would be the first idiosyncrasy, but certainly not the last, shown by evolved programs in the project.

There are a couple of aspects of this algorithm that are worth noting. The first, which was quite shocking at first, was that the spiral was formed from left turns. But the command set at that point only included `turnRight()` – `turnLeft()` hadn’t been implemented yet. The program had discovered an unintentional and unknown feature of the engine – the ability to "turn one's head." Due to requirements of the engine, the moving algorithm runs and makes turning decisions in full every turn. In a single turn each player queues a single direction to move for that turn, then at the end of the turn the engine moves each player in the direction they indicated. It is impossible (or at least not feasible within the scope of the project) to queue more than one action. However, because the player only advances at the end of the round, the player can turn any number of times before the round ends. So the player can call `turnRight()` once to turn right, or three times to turn left. The more interesting aspect of that is that while the player's "head" is turned, its perspective changes as well. So the player can look in any direction and check the conditions before making its move decision.

The other interesting aspect of the algorithm is that it overcame an error in the if-then programming. The `wallAhead()` method was supposed to perform an action if there was a wall directly in front of the player; however, the method actually did that action if there wasn't a wall in front of the player (to avoid confusion, use of the incorrect method in code listings will be written as `!wallAhead()`). This makes it more difficult (and confusing, if you're a human!) to program seemingly simple logical decisions, e.g. `if (wallAhead()) turnRight()`. The computer had no trouble with this; it had no knowledge of what `wallAhead()` was supposed to do, it just managed to combine it with other commands to form something successful.

An extensive analysis of the evolved program has not been done. It appears to be somewhat bloated, being rather large and probably containing a lot of redundant commands. It is included in Appendix A. At this point there were also errors in the code which made the printed algorithm slightly untrustworthy. The frequently appearing "`&1`" is a placeholder for whatever the program executes at that point, but which was never replaced due to exceptions that were being thrown during code generation. It's not clear what happens in those places. It is likely turnRight(), but it could be a subroutine or nothing at all.

The next evolved program, after making some minor tweaks to the code and adding turnLeft(), was just a simple modification of the spiral. However, in this case the player, upon encountering a wall, would turn right and follow it. This gave a significantly better fitness rating, although the player was still subject to certain quirks and the program rarely won any matches.

The second spiral program is notable for its extreme simplicity, compared to its predecessor:

```
sub {
      if (!wallAhead()) then turnLeft();
      if (!wallAhead()) then turnLeft();
      turnRight();
}
```

Simple Algorithm

The next step was to increase the capabilities of the GP player. `goStraight()`, `moveTowards()`, and `moveAway()` entered the commands list. At this point the fitness function also changed. It now awarded additional points if the GP player won the match, and the score averaged over five games, to account for lucky or unfortunate rounds.

After several evolutions and experimentation `goStraight()` was removed from the commands list, since unfavorable results were being produced, even after evolving many generations. It's hard to say why its addition would cause problems, but it can be seen that its presence doesn't add any functionality – going straight can be simulated by either doing nothing or turning one way and then the other. Moreover, the sequence `if (!wallAhead()) goStraight()` is essentially meaningless and superfluous, and worse, `if (wallAhead()) goStraight()` is suicidal. Since there was no need for the function it was removed and curiously performance increased immediately.

With the added commands, the system evolved the following algorithm:

```
sub {
      if (!wallAhead()) then turnLeft();
      if (!wallahead()) then moveTowards();
      turnRight();
}
```

Note first its very simple nature. It resembles the second spiral algorithm quite closely. Accordingly, the player behaves in a simple manner. A little analysis shows why the algorithm is successful. In the case of an open space in front of the player, it turns left. If there's open space to the left, it turns towards the opponent. Then it turns right. But if there's a wall in front of the player, it just turns right. The opening of a typical game shows this. First the player moves down while the opponent moves up. Since there is nothing but open space in front of the player, it turns its head left. There's open space there, too, so it turns towards the opponent. In this case, that's left also, so it doesn't turn any more. Then it turns right, which faces it down, and the round passes. This happens until the opponent is more up than left, at which point the player, in the second part of its routine, turns up. It then turns immediately right, avoiding running into its own tail.

This has the overall effect of putting the player in an avoidance pattern. Not direct avoidance, by turning always away from the opponent, but trying to move laterally. The right turn at a wall prevents basic collisions. But a side effect is that the GP player tends to stay on the outside wall for a good portion of the game. Once it hits the wall, it tends to stick to it – due to the canceling nature of the left and right turns, and the skipping of `moveTowards()` due to a wall at the left – until it reaches another wall at which point it turns. In this way the GP player avoids the computer, waiting for it to make a mistake of fall into some kind of trap.

It's not exciting play, and it's not human play, but it's successful against the computer player, which is why the program evolved in this way. This algorithm had a high success rate against Computer 1 (which it trained against), winning a high percentage of games, but was not as successful against Computer 2. A human player should easily be able to figure out the pattern and set traps against the walls, guaranteeing victory every time.

Further Experiments

The process of command addition and fitness modification continued. The fitness function first changed in the following manner: the player is awarded additional fitness points depending on what the opponent crashed into (boundary wall: +0.0; opponent wall: +0.25; player wall: +1.0). The time bonus was adjusted to give greater reward for longer games in the losing case, and for shorter games in the winning case. The programs were also trained against Computer 2, to see how different opponents bred different players. A notable algorithm evolved from these changes, and its code is provided in Appendix B (in its original as well as a simplified form). The program shows some hints of "bloodthirstiness" – seemingly going after the other player. It tended to stay close to Computer 2, often mimicking its erratic movements and sometimes forcing it into traps. In the evolution process, this algorithm earned a remarkable score of 2.483 out of a possible 3.0. This score seems rather large and improbable, and running the program against Computer 2 in the graphical engine didn't always show such great performance, so it is possible the program got lucky in the simulator. However, the program was still able to compete well against the computer and win a significant percentage of games. The algorithm did not fare as well against a human player, having obvious exploitable gaps. See Appendix C for screenshots of the program in action.

Conclusions

There is clearly much more work that can be done in this area. This project only scratched the surface of the issue and came up with some intriguing results, and set the stage for further study and experimentation. There are some conclusions that can be drawn already. First, evolved program personalities are highly depended on fitness functions. The behavior of the generated players changed significantly, from patterns to evasion to aggressiveness, as the fitness measurement was changed. Second, genetic programming techniques are good at exploring the system and bringing out previously unknown features. This was seen in the first generated programs which managed to turn left having only the ability to turn right. As such, applications in testing come to mind, where a genetic program could detect errors in a system. Third, in order to simulate human play, there probably needs to be a way to set a multi-turn strategy. Determining an action from start to finish in each round proved to be severely limiting in many ways. In the process of experimentation it became clear that the human mind develops overall strategies of play and does not determine its next action from round to round. This is why it was easy to examine a genetic program's behavior and defeat it consistently – because the mind can set a goal and follow through with it. Simulating this proved to be too difficult within the system used here, but in the future that avenue can be explored. And lastly, the behavior of a genetic player depends greatly on the opponent it trains against. Players that were evolved using Computer 1 played well against Computer 1, but not against Computer 2 or a human. Players that evolved against Computer 2 didn't fare very well against Computer 1 or a human. It would be very interesting to see how the programs could evolve given skilled human opponents.

The project didn't necessarily create masterful game players, but it was a good start down the road to understanding genetic techniques and creating players with those techniques. With the potential addition of strategies, or a simulation thereof, and with the addition of new commands and fitness measurements, the system should be able to evolve a competitive player in the future.

Glossary of Commands

`sub {}` – A subroutine. Allows the program to execute commands in sequence.

`turnRight()` – Faces the player immediately right.

`turnLeft()` – Faces the player immediately left.

`goStraight()` – Faces the player in the direction of motion.

`moveTowards()` – Faces the player in the direction that will move it closest to the opponent player.

`moveAway()` – Faces the player in the direction that will move it furthest from the opponent player.

`moveFollow()` – Faces the player in the direction the opponent is facing.

`randomMove()` – Faces the player in a random direction that has no wall directly in line.

`wallAhead()` – Returns true if there is a wall in the direction the player is facing.

`wallAheadInFive()` – Returns true if there is a wall within the next 5 spaces in front of the player.

Sources

BMtron 1.2
Mikhail A. Kryshen (Open Source)
http://kryshen.pp.ru/games/bmtron.html

JGAP 3.2
Neil Rotstan, Klaus Meffert, et al. (Open Source)
http://jgap.sourceforge.net/

Java Development Kit
Gosling, et al.
http://java.com

Appendix A
First Spiral Program (In Original Form)

```
sub {
      sub {
            sub {
                  turnRight();
                  if (!wallAhead()) then (&1);
                  if (!wallAhead()) then (&1);
                  turnRight();
            }
            sub {
                  turnRight();
                  if (!wallAhead()) then (&1);
                  turnRight();
            }
            sub {
                  if (!wallAhead()) then (&1);
                  turnRight();
                  turnRight();
            }
      }
      turnRight();
      sub {
            sub {
                  if (!wallAhead()) then (&1);
                  if (!wallAhead()) then (&1);
                  turnRight();
            }
            if (!wallAhead()) then (&1);
            sub {
                  if (!wallAhead()) then (&1);
                  if (!wallAhead()) then (&1);
                  turnRight();
                  turnRight();
            }
            sub {
                  turnRight();
                  turnRight();
                  if (!wallAhead()) then (&1);
                  turnRight();
            }
      }
}
```

Appendix B
Crazy Program

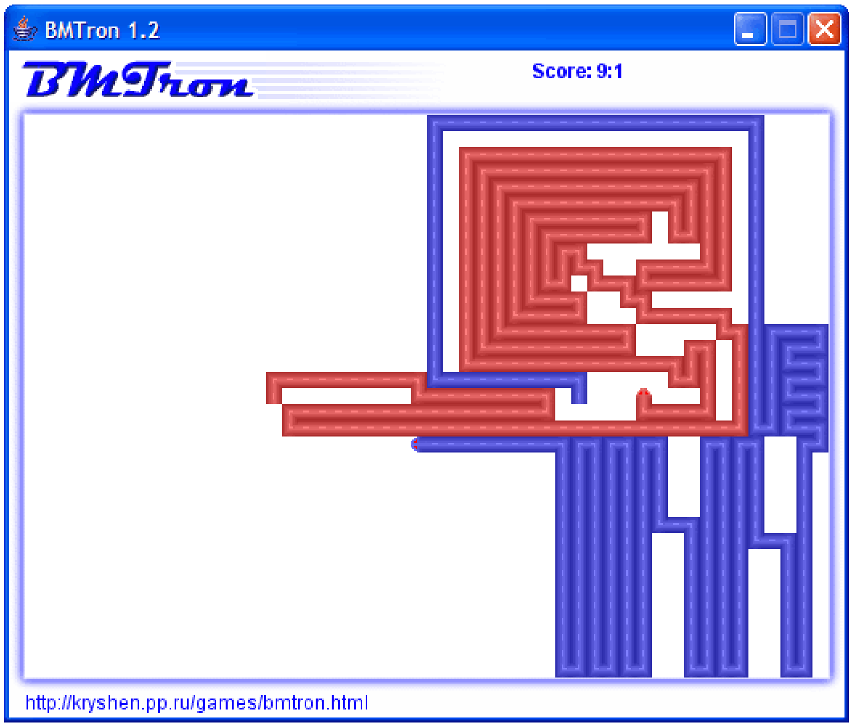
Original:

```
if (wallAheadInFive()) then {
      moveTowards();
      turnLeft();
      turnRight();
}

if (wallAhead()) then
      if (wallAheadInFive()) then {
                  turnRight();
                  if (wallAhead()) then turnLeft();
                  if (wallAhead()) then moveAway();
                  if (wallAheadInFive()) then turnLeft();
                  turnRight();
                  turnLeft();
                  moveAway();
                  if (wallAhead()) then
                        if (wallAheadInFive()) then turnRight();
                  if (wallAheadInFive()) then
                        if (wallAhead()) then moveTowards();
      }

if (wallAheadInFive()) then
      if (wallAhead()) then turnRight();
if (wallAheadInFive()) then
      if (wallAhead()) then turnRight();

if (wallAheadInFive()) then
      if (wallAheadInFive()) then
            if (wallAhead()) then {
                        moveTowards();
                        moveTowards();
                        moveAway();
                        moveTowards();
            }
```

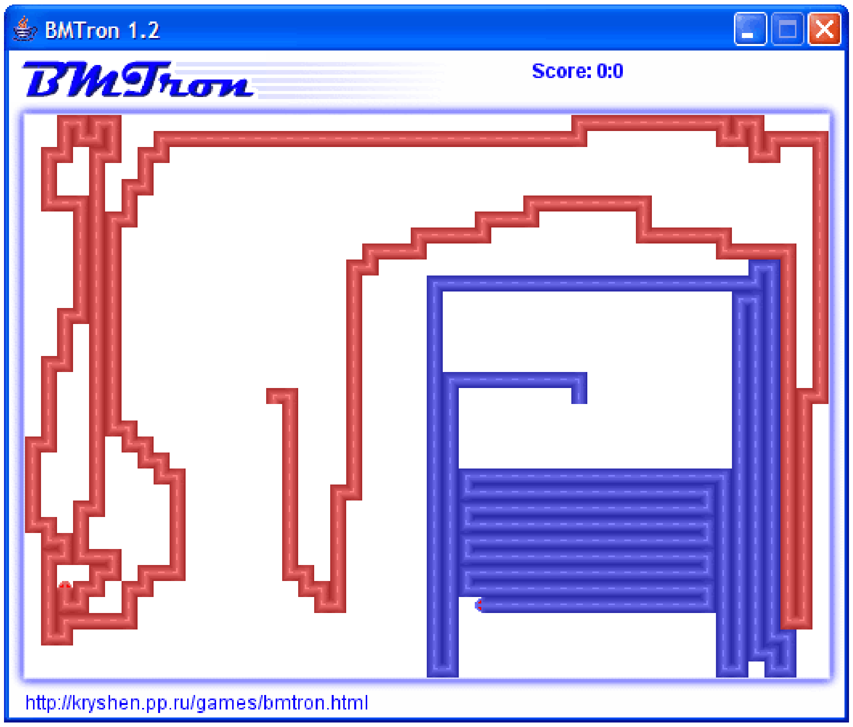
Simplified:

```
if (wallAheadInFive()) then moveTowards();

if (wallAhead()) then {
            moveAway();
            if (wallAhead()) then turnRight();
            if (wallAhead()) then moveTowards();
}

if (wallAhead()) then turnRight();
if (wallAhead()) then turnRight();
if (wallAhead()) then moveTowards();
```

Appendix C
Crazy Program Screenshots



About to win against Computer 2



About to win against Computer 1

Read Me

The source code and executable programs used in this project are provided on CD. All of the code and libraries necessary to run BMTron and the Evolver are stored in the `bmtron-1.2` directory. The `JGAP` folder contains source code and libraries for compiling JGAP; it has been modified from its 3.2 release, and running the system with the 3.2 release may not work correctly or at all.

There are also a number of project "freezes." When an interesting program evolved the entire structure was zipped up and archived. This was done because further changes in the program meant that stored GPPlayer objects would not be de-serializeable. To see previously generated players in action those archives can be unzipped and run in the standard way.

JGAP can output visualizations of individual chromosomes. Portraits of interesting evolved programs are stored in `bmtron-1.2/dist`, along with `.obj` files containing the programs themselves (although they will not be loadable, for the reasons mentioned above).

<u>Source files worth noting</u>: (all in `bmtron-1.2/src/org/bm/bmtron`)
`Evolver.java, GPPlayer.java`: Most important for understanding the project.
`TronCommand.java, GP*.java`: Generally useable GP commands
`Game.java, Field.java`: Heavily and recklessly modified in the course of the project.
`*.code`: Interesting generated player code. Not always readable.

<u>Installation</u>: copy the JGAP and bmtron-1.2 directories to hard disk.

<u>To Run BMTron</u>:
1. Change directory to `bmtron-1.2`
2. type "`e`" (to execute `e.bat`)

<u>To Evolve a Player</u>:
1. Change directory to `bmtron-1.2`
2. type "`evo`" (to execute `evo.bat`)
3. The population will evolve until fitness > 2.5 OR 1,000 generations. This may take a while. This and other parameters can be changed in `Evolver.java`.

<u>To Compile BMTron</u>: (including the genetic player)
1. Compiling BMTron requires Apache Ant. Obtain and install it.
2. Change directory to `bmtron-1.2`
3. type "`make`" (which executes `make.bat`)

<u>To Compile JGAP</u>:
1. Compiling JGAP also requires Ant.
2. Change directory to `JGAP`
3. type "`compile`" (to execute `compile.bat`)