Goal-Oriented Adversarial Movement Behaviours with Genetic Programming

HIT3046 AI for Games - Research Report

Charlotte Pierce

Abstract

Genetic programming has been successfully applied in numerous game-based contexts. Though a popular construct in game development, industry tends to favour manually coded rather than generated behaviour trees despite their demonstrated performance.

This work applies genetic programming to determine a behaviour tree capable of moving an agent through an environment to a goal location whilst avoiding detection.

Tests using five alternate game maps show the potential of the technique to solve this problem.

1 Introduction

The class of evolutionary algorithm defined as genetic programming (GP) can be viewed as the process of iteratively improving a system design and architecture (Martin (2001)). Poli et al. (2008) describes the technique as follows:

> "[genetic programming] automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance."

This work attempts to apply GP to the task of finding a program capable of moving an agent through an adversarial environment towards a target location. To complete this task the agent must successfully move around obstacles and avoid areas of detection.

The merits of using GP as a tool for developing strategies in general has been acknowledged by Sipper et al. (2007). The randomness of the GP process is often responsible for improving the results of the technique compared to strictly deterministic methods. This is due to the fact that random elements increase the scope of candidate solutions (Simpson et al. (1994)).

Behaviour trees, such as those often used in game development, can be represented in the same data structure as that used by GP to represent candidate solutions. However, creating behaviour trees for dynamic environments is a difficult task, and the games industry tends to adopt traditional algorithms such as A* and min-max (Perez et al. (2011)).

Perez et al. (2011) demonstrated the viability of GP as an alternative to standard AI techniques by creating a bot capable of navigating unseen levels in a platformer game. The potential of genetically programmed behaviour trees has also been shown by Lim et al. (2010), who used GP to create behaviour trees capable of playing DEFCON, defeating human opponents in over 50% of games.

This work is closely related to work by Sondahl (2005), where GP was applied to find agents capable of moving to a goal location within a maze. Agents were provided with three 'commands' and four 'reporters'. Commands included instructions for the agent to move for-

ward, turn left and right, to wait, and a conditional `ifelse` statement. Reporters comprised simple boolean queries regarding the location of walls (i.e., whether there is a wall in front, to the right and to the left of the agent). GP was successfully assigned the task of finding the correct composition of commands and reporters, represented as a tree-like structure, to navigate through a given maze.

The applicability of GP to game-like environments has been widely investigated. For example, Hauptman & Sipper (2005) used GP to generate endgame chess players capable of defeating other manually programmed artificial intelligences.

GP has also been applied to generate cooperative agents. The technique was used by Luke et al. (1998) to create agents capable of working together as a soccer team. Using a similar approach to Sondahl (2005), the algorithm was provided with a number of commands and reporters, and GP was applied to determine a suitable composition of these elements. Results from this work were used to control soft-bots[1] in RoboCup, and were able to defeat multiple human-coded teams.

Section 2 will detail the experimental process used to apply GP within the context of the problem described. The results of these experiments are discussed in Section 3 with potential further work described in Section 4.

2 Method

2.1 Environment

A tile-based, grid-like environment was used. Each environment must contain a single agent and target location, and may contain a single guard and multiple obstacles (i.e., non-traversable tiles). No weighting is defined between tiles; adjacent tiles are a standard distance of 1 from each other.

Environments are defined in plain text files, the contents of which represent a map literally. Different characters are used to indicate tile types. Listings 1, 2, 3, 4 and 5 show the five test maps that were used in this work.

2.2 Guards

Each tile surrounding a guard is flagged with a 'detection' variable, indicating that an agent will be caught if they move onto that tile.

A guard can be set to move randomly within the environment given a provided walk length. The actual walk length will be two times the length specified so that the guard moves between two points on the map. This avoids situations where the guard moves to an edge of the environment and stops.

Alternatively, a guard can be stationary.

2.3 Program Trees

The programs generated by GP are typically represented as tree structures (Colton (2004)). Each program tree is comprised of a number of nodes, each of which represents a single query or action. Agent actions are restricted to four movements: `north`, `south`, `east` and `west`. When executed these actions move the agent a single tile in the specified direction.

Queries relate to one of three areas: location of obstacles, location of the guard, and direction of the target location. Obstacle and guard location queries are limited to a single factor, being whether they can be 'seen' in a specific direction (north, south, east or west) within a distance of 1. Queries regarding the location of the guard operate similarly, but a larger distance is allowed - agents are able to 'see' a guard within a distance of three tiles. Queries regarding the direction of the target location are not limited by distance.

[1] Soft-bots compete in a simulated soccer environment.

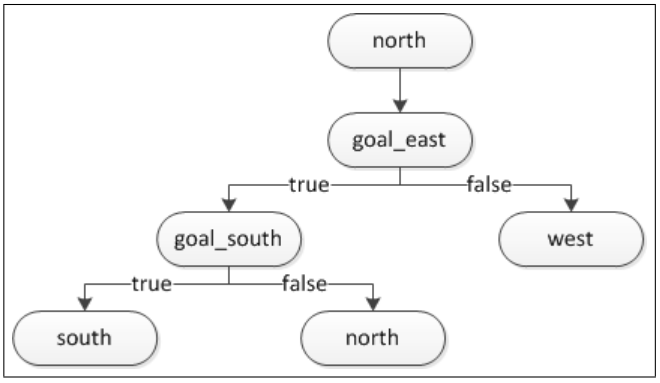

Figure 1: Simple program tree

A conditional `ifelse` structure forms the last element of valid program syntax. Only simple boolean queries such as those described can be used within this structure. Figure 1 shows an example of a simple program tree combining all described syntactical elements.

2.4 Selection

For all experiments roulette wheel (a.k.a. fitness proportionate) selection was used. The fitness measure is defined by Equation 1.

$$fitness = max\_steps - agent\_distance\_from\_goal \quad (1)$$

2.5 Experiment Parameters

Two classes of experiments were performed. During the first all guards (where they exist) were set to remain stationary; for the second, only maps containing guards (i.e., maps 2, 4 and 5) were used, and guards were given a random walk length of 20 steps.

Initial populations were generated randomly with each agent being given a random program tree with a length randomly selected in the range $[5, 10]$.

Agents were given a step limit (i.e., value of max_steps) of 25, after which their performance was recorded. This prevented inefficient agents from being assigned unfairly high fitness values, or non-moving agents from disrupting the simulation.

All experiments employed a population size of 100, and ran for 100 iterations.

3 Results

Figures 2 and 3 show the average distance from the goal of the candidate population at each iteration for environments where guard movement was turned off and on respectively. Both figures show a positive result, where the average distance from the goal is reduced over time.

Also shown by both figures is a tendency to reach a performance plateau at a distance of 4. This indicates that agents from these populations are frequently caught by the guard, as guards are generally initialised around this distance from the goal.

The effects of guard movement in terms of problem complexity can also be seen. Figure 2 shows a high-performing population was generated for map 2. However, once guard movement was turned on for the same map, average performance saw a significant reduction (see Figure 3).

This was not the case for map 4, where performance was significantly improved once guard movement was turned on. This is likely a result of the stochastic nature of GP, indicating that with a sufficient population size and number of iterations a strong result could be achieved for each map.

Figures 4 and 5 show the performance of the best agent at each iteration. Performance where guard movement was not allowed show a higher variability than those where guard movement was turned on. This could indicate an insufficiency of GP in generating a solution for those environments, or simply be a side-affect of the random guard walks generated for this particular experiment.

Whether guard movement is on or off, the

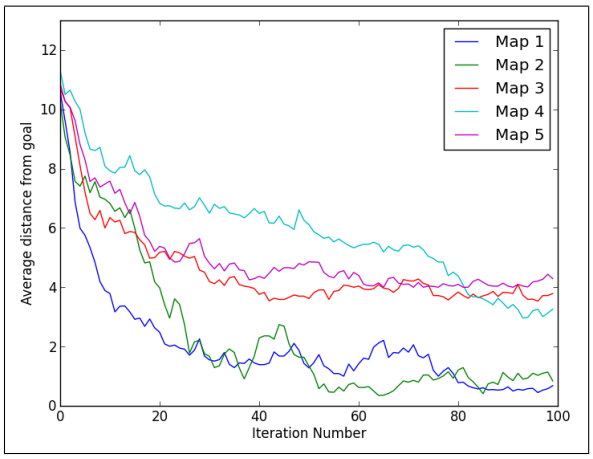

Figure 2: Average distance from goal; guard movement off

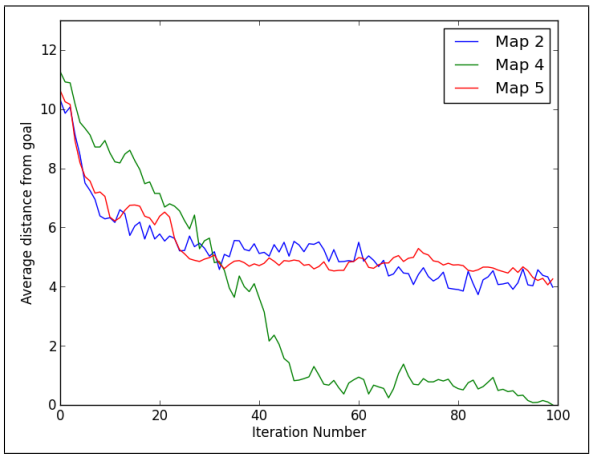

Figure 3: Average distance from goal; guard movement on

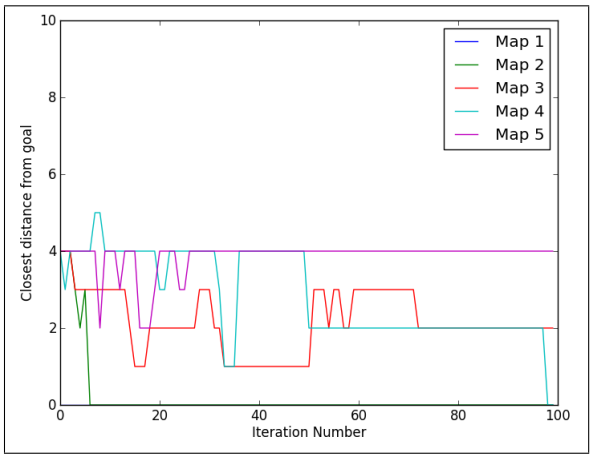

Figure 4: Closest distance to goal; guard movement off

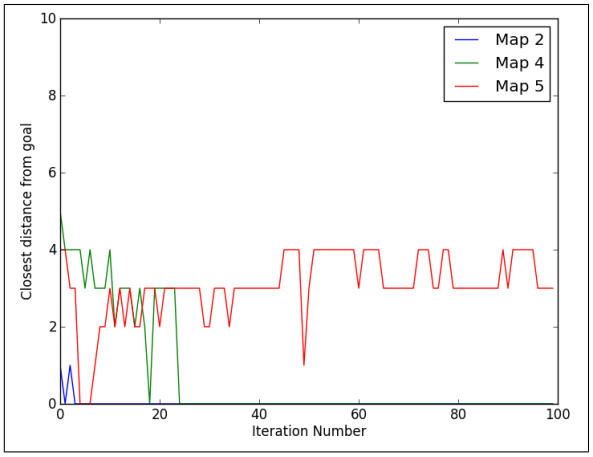

Figure 5: Closest distance to goal; guard movement on

data shows that populations which end up performing well overall (i.e., on average) tend to contain a high-performing, or even perfect candidate during an earlier population. It is likely that the high average performance is then a result of elements from that single candidate slowly becoming dominant throughout the population.

Generally, the best agents generated by GP when guard movement was turned off did not perform well in unfamiliar contexts. This is a result of overtraining, where the generated program tree was too specific to the test map used. The randomness introduced when guard movement was turned on increased performance of agents on alternate maps, but not significantly.

A third experiment was run over map 5, using a population of 1000 and a guard walk length of 20. The results of this test, shown in Figure 6, demonstrate the effects of a larger population when using GP. Within few iterations a perfect solution had been found, and the average performance of each population improved significantly over time. Unlike previous tests, performance reached a plateau at a distance of 1 rather than 4, with most non-perfect agents reaching the step limit rather than getting caught.

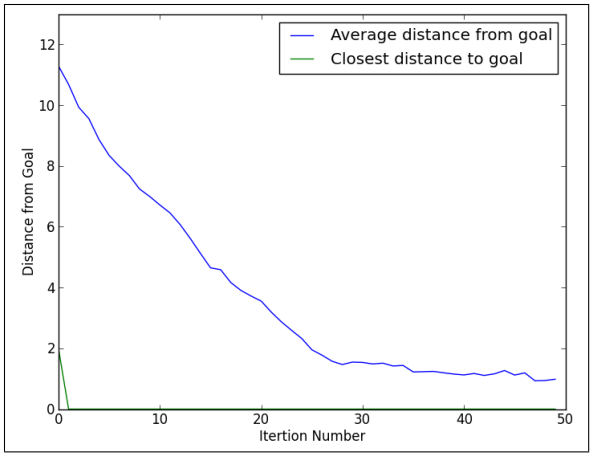

Figure 6: Performance of GP on map 5 using a population of 1000

4 Discussion

Given the time requirements of running experiments using GP, the extent of tests was limited. It is expected that further tests, particularly with larger population sizes over more iterations, would show higher performance results on all environments. The resulting agents would likely transfer more effectively to other environments if generated with guard movement turned on.

Alternate fitness measures may also affect the final results. For example, the fitness measure currently being used does not consider how an agent's run was ended (i.e., did the agent reach the step limit, was the agent caught etc.), or the expected path of the agent given obstacles and guard location. If this information were to be considered a more accurate and effective fitness measure could be defined.

The role of parameter selection in GP is widely recognised, but it is only recently that the influence of initial population generation has been considered Maaranen et al. (2007). Further experiments could test the effects of alternate initialisation methods.

The flexibility and scope of candidate solutions could be improved with additions to the set of valid syntax. Extensions to existing functionality could include allowing comparators (i.e., '<', '>', '=' and '!=') and boolean operators (`OR` and `AND`). Furthermore, limiting the depth of program trees would result in more concise, reusable solutions.

Lastly, culling methods could be applied to candidate populations to remove low performing agents from the gene pool.

5 Conclusion

This work has shown the potential of GP in determining the composition of agent behaviours in an adversarial environment. Results from basic experiments were positive, showing a steady increase in the performance of solutions over time in environments featuring both static and randomly moving guards.

It is expected that further testing would continue to improve results, particularly in the performance of generated agents in an unfamiliar environments.

Listing 1: Test map 1

```
# . = traversable tile
# x = non-traversable tile
# o = agent starting position
# G = goal position
# 1 = guard position
...........
...........
...........
...........
o..........
..........G
...........
...........
...........
...........
```

Listing 2: Test map 2;
Tile types are the same as defined in Listing 1

```
..........
..........
..........
..........
o.........
.....1....G
..........
..........
..........
..........
```

Listing 3: Test map 3;
Tile types are the same as defined in Listing 1

```
..........
..........
..........
.....x.....
o....x.....
.....x....G
.....x.....
..........
..........
..........
```

Listing 4: Test map 4;
Tile types are the same as defined in Listing 1

```
..........
..........
..........
.....x.....
o....x.....
.....x....G
.....x.....
..........
......1....
..........
```

Listing 5: Test map 5;
Tile types are the same as defined in Listing 1

```
..........
......1....
..........
.....x.....
o....x.....
.....x....G
.....x.....
..........
..........
..........
```

References

Colton, S. (2004), 'Genetic programming (lecture material)', `http://www.doc.ic.ac.uk/~sgc/teaching/pre2012/v231/lecture17.html`. [Online; accessed 04-June-2013].

Hauptman, A. & Sipper, M. (2005), *GP-endchess: Using genetic programming to evolve chess endgame players*, Springer.

Lim, C.-U., Baumgarten, R. & Colton, S. (2010), Evolving behaviour trees for the commercial game defcon, *in* 'Applications of Evolutionary Computation', Springer, pp. 100–110.

Luke, S. et al. (1998), 'Genetic programming produced competitive soccer softbot teams for robocup97', *Genetic Programming* **1998**, 214–222.

Maaranen, H., Miettinen, K. & Penttinen, A. (2007), 'On initial populations of a genetic algorithm for continuous optimization problems', *Journal of Global Optimization* **37**(3), 405–436.

Martin, M. C. (2001), Visual obstacle avoidance using genetic programming: First results, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)'.

Perez, D., Nicolau, M., ONeill, M. & Brabazon, A. (2011), Evolving behaviour trees for the mario ai competition using grammatical evolution, *in* 'Applications of Evolutionary Computation', Springer, pp. 123–132.

Poli, R., Langdon, W. W. B. & McPhee, N. F. (2008), *Field Guide to Genetic Programming*, Lulu Enterprises Uk Limited.

Simpson, A. R., Dandy, G. C. & Murphy, L. J. (1994), 'Genetic algorithms compared to other techniques for pipe optimization', *Journal of Water Resources Planning and Management* **120**(4), 423–443.

Sipper, M., Azaria, Y., Hauptman, A. & Shichel, Y. (2007), 'Designing an evolutionary strategizing machine for game playing and beyond', *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* **37**(4), 583–593.

Sondahl, F. (2005), 'Genetic programming maze rules', `http://www.cs.northwestern.edu/~fjs750/netlogo/final/mazerules.html`. [Online; last accessed 10-June-2013].