Genetic Programming Technical Report
Space Invaders Player

Paul Magnus
with Ian Wilson

December 2017

1 Introduction

This project aims at creating a genetic programming based agent that can play the arcade game Space Invaders successfully. This problem is hard for a human to program since the exact parameters of when the player should move and shoot to optimize its playing are unknown and difficult to determine by normal methods.

2 Game Specifics

The Space Invaders game is a modified Java implementation from http://zetcode.com/tutorials/javagamestutorial/spaceinvaders/. All graphics and threads have been removed from the Java implementation to speed up the genetic programming system and

Figure 1: A snapshot of the Space Invaders game.

```
public class GameState {
    // Constructs a new GameState based on the
    // current board
    public GameState(Board board);

    // Returns the (x, y) position of the player
    public int[] getPlayerPosition();

    // Returns whether the player's shot still
    // exists in the game window
    public boolean playerShotExists();

    // Returns the (x, y) position of the player's
    // current shot
    public int[] getShotPosition();

    // Returns the (x, y) position of the nearest alien
    public int[] getAlienPosition();

    // Returns the distance to the nearest alien
    public long distanceToNearestAlien();

    // Returns the (x, y) position of the nearest bomb
    public int[] getBombPosition();

    // Returns the distance to the nearest bomb
    public long distanceToNearestBomb();
}
```

Figure 2: Header for the GameState class. Push can get information about the current state of the game from this class.

reduce possible compatibility issues. These features are added back in when we want to watch how the finished programs act.

This version of Space Invaders is slightly different than the standard arcade game in a few aspects. The score of the game is simply the number of remaining aliens and the goal is to have the lowest score possible, 0. In this way, all aliens are worth the same point value. This implementation does not include the bunkers of the standard game so dodging the bombs that the aliens drop is an important part of progressing beyond simple strategies. Also, the player cannot shoot again until the previous shot has either gone off of the screen or the shot has hit an alien. The aliens' bombs function the same as the player's shot. A snapshot of a game is shown in Figure 1.

2.1 Game State

In order to interface the Java implementation with Push we created a GameState class. Figure 2 shows a header for this Java class. The methods given in this header are all of the properties of the game that the evolved program can use to determine what to do.

3 Clojure, Java, and Push

In order to implement this project we required a mix of Clojure, Java, and Push programs. As stated above, the implementation of the game was written in Java. The overall genetic programming algorithm and evaluation of Push programs was written in Clojure. The evolved programs are in Push using Plush genomes.

For the evaluation of each Plush genome, the genome is first converted into a Push program by Clojure using translate.clj. This program is then sent to Java ten times, once for each of the random seeds that determine the randomness of the games. For each of these games, one animation step is computed then the Push program and the current state of the game, in the form of an instance of the GameState class are sent back to Clojure. The Push program is then interpreted based on the current game state and returns a list of two integers. The first indicates whether the player should move right (return of 1), move left (return of 2), or stay in its current position (return of 0). This first number is based on the top element of the **:string** stack. The second number indicates whether the player should shoot (return of 1) or not (return of 0). This is based on the top element of the **:boolean** stack.

4 New Push Instructions

In this implementation, the Push program is given the current GameState as an input and returns the result from the **:string** and **:boolean** stacks. The stacks used are **:integer**, **:boolean**, **:string**, **:input**. The **:input** stack is actually implemented as a map of data from the GameState. The creation of the **:input** map is shown in Figure 3. The elements of this map are used by the Push program through the use of the following Push instructions.

get_player_x	get_shot_x	get_alien_y	get_alien_dist
get_player_y	get_shot_y	get_bomb_x	get_bomb_dist
shot_exists	get_alien_x	get_bomb_y	

The shot_exists instruction pushes a boolean onto the **:boolean** stack and all of the other instructions push an integer onto the **:integer** stack. With the addition of the **:boolean** stack there are several new instructions that work with booleans and push booleans onto the **:boolean** stack based on other stacks.

bool_=	bool_dup	bool_not	int_<
bool_and	bool_or	int_=	int_>

```
(defn gs-to-map
  "This creates a map of all of the current gamestate
  information. This is used by all gamestate accessor methods
  in push."
  [gs]
  {:player_x (nth (vec (.getPlayerPosition gs)) 0)
   :player_y (nth (vec (.getPlayerPosition gs)) 1)
   :shot_exists (.playerShotExists gs)
   :shot_x (nth (vec (.getShotPosition gs)) 0)
   :shot_y (nth (vec (.getShotPosition gs)) 1)
   :alien_position (vec (.getAlienPosition gs))
   :bomb_position (vec (.getBombPosition gs))
   :alien_distance (.distanceToNearestAlien gs)
   :bomb_distance (.distanceToNearestBomb gs)
   })
```

Figure 3: Creation of the **:input** map from a GameState instance.

There are new **:exec** stack instructions to allow for use of logic by the Push programs. These are

exec_dup exec_if exec_do*range

All of the instructions besides the GameState instructions are the same as those described by https://faculty.hampshire.edu/lspector/push3-description.html.

There are some instructions that we considered but were not included in the submitted project. These instructions were generally removed since they seemed to be unnecessary and give the system too many options to try before it would find the correct solution.

int_flush	bool_flush	bool_swap	exec_pop
int_swap	bool_pop	exec_=	

The Push programs can also include the following literals: 0, 1, true, false, "Left", and "Right". Using the "Left" and "Right string literals is the only way to get a string on the **:string** stack which controls the motion of the player.

5 Plush Genomes

The evolution in this program evolves Plush Genomes rather than normal Push programs. These genomes consist of a list of maps containing **:instruction**, **:silent**, and **:close**. A sample individual with a genome is shown in Figure 4. The **:silent** and **:close** epigenetic markers function in the same way as discussed in class. The hope with these genomes was that they would help the system use the **:exec** stack instructions more effectively through intelligent use of parentheses. Unfortunately, as will be discussed later, this appears to have had little effect on the final result.

Several genetic operators were changed and some others created to work effectively with the Plush genomes. The function **uniform-addition** was replaced with **uniform-plush-addition**, shown in Figure 5, which creates a random Plush gene rather than a random Push

```
(def genome-example
  {:program '(true exec_if (int_+ bool_not) ())
  :error []
  :genome '({:instruction true :silent false :close 0}
            {:instruction false :silent true :close 0}
            {:instruction exec_if :silent false :close 0}
            {:instruction int_+ :silent false :close 0}
            {:instruction bool_not :silent false :close 1})
  :total-errors 0})
```

Figure 4: An individual with a plush genome. The **:program** element is the translated Push program based on the genome.

instruction. The crossover operation was updated to **plush-crossover**, shown in Figure 6, which takes two Plush genomes and uses uniform crossover on the two genomes to create a new child genome. Note that there is a 5% chance to increment or decrement the **:close** marker of any gene being used by way of the **evolve-close** function shown in Figure 7.

6 Lexicase Selection

Lexicase selection was added near the end of the implementation to try to improve the behavioural diversity of the population. Lexicase currently uses the 10 random seed game evaluation scores as the basis for selection. The hope was that in improving this diversity we would obtain better results. It does appear that the populations are more diverse after the introduction of Lexicase selection but this has not yielded better results as the diversity has not been enough for the system to discover more complex and effective solutions to playing the game than just sitting in one place while constantly shooting.

7 Results

Despite a variety of different settings for **max-generations**, **population-size**, and **max-initial-program-size** the system was unable to create a well-functioning game player. It consistently figures out within the first few generations that shooting as often as possible is a good strategy and so the best programs are those that end with a **true** on the top of the **:boolean** stack all of the time. These programs function better overall than all other simple programs since they manage to shoot many of the aliens before they are hit by one of the bombs. Unfortunately, the limited success of this simple strategy has proved problematic for evolving the system further. In fact, most of the time when Lexicase selection is not used, almost all programs in later generations will simply use this strategy. The addition of Lexicase selection increases the number of different kinds of programs a little but these programs tend to either move completely to the left or completely to the right while shooting as often as possible. Both of these strategies work better in a few test cases but are overall worse.

```
(defn random-plush-chance
  "Takes a list. 5% of the time, returns a list containing one
  ranodm element of the list. Otherwise, the empty list is
  returned."
  [lst chance]
  (if (<= (rand) chance)
    (list (make-random-plush-gene lst 0.05 2))
    '()))

(defn uniform-plush-addition
  "Randomly adds new instructions before every instruction
  (and at the end of the program) with some probability.
  Returns child program."
  [prog]
  (loop [parent prog
         child '()]
    ;; base case: if the parent is empty, return a random gene
    (if (empty? parent)
      (concat child (random-plush-chance instructions 0.05))
      (recur (rest parent)
             (concat
              child
              (random-plush-chance instructions 0.05)
              (list (first parent)))))))
```

Figure 5: The modified code for the uniform addition operator. This new operator incorporates Plush genomes.

```
(defn plush-crossover
  "Crosses over two programs (note: not individuals) using uniform
  crossover. Returns child program."
  [genome-a genome-b]
  (loop [g1 genome-a
         g2 genome-b
         child '()]
    ;; base case: if both are empty, return an empty list
    (if (and (empty? g1) (empty? g2))
      child
      (recur
       (rest g1)
       (rest g2)
       (concat
        child
        ;; choose which parent to pick from
        (choose-50%-chance
         (concat
          ;; check if parent is empty so as to
          ;; not return nil
          (if (empty? g1) '() (list (evolve-close (first g1) 0.05)))
          (if (empty? g2) '() (list (evolve-close (first g2) 0.05))))))))))
```

Figure 6: The updated uniform crossover operation. This includes the ability to slightly modify the **:close** marker of the genes being used in crossover.

```
(defn evolve-close
  "Gives a small chance for close to be randomized"
  [gene change-close]
  (if (> (rand) change-close)
    gene
    ; randomly increments or decrements the close value
    (let [close-gene (get gene :close)]
      (if (= close-gene 0)
        (assoc gene :close (inc close-gene))
        (assoc gene :close ((eval (rand-nth '(inc dec))) close-gene))))))
```

Figure 7: This function allows for a small random chance to increment or decrement the **:close** marker.

8 Future Possibilities

Genetic programming can still hold promise to solve this problem but with further modification. I plan to try to add in a measure of how long the player lasted in the game as a heuristic which can be used by Lexicase as a further test case for each game played. The heuristic will be measured by the number of frames that the player survived through. This should incentivise programs that can dodge the aliens' bombs to some extent even if these programs cannot manage to destroy many of the aliens. Hopefully with the combination of these other programs and the currently successful programs, the system will be able to figure out a good solution that can beat the SpaceInvaders game.