Information Flocking Applied to Genetic Programming Visualization

Matthieu Macret
School of Interactive Arts and Technology
Simon Fraser University, Surrey,
B.C., Canada V3T 0A3
mmacret@sfu.ca

ABSTRACT

Genetic programming is an evolutionary technique used in optimization problem. Its dynamic nature makes it difficult to visualize. Indeed, hundred of candidate solutions are evolved per generation. There could be relationships between these candidate solutions. Phenomenon such as code growth can also happen. Information flocking uses the emergence of motion to visualize the dynamics of varying datasets. It makes possible to observe the data at different levels. In this paper, we apply information flocking techniques to genetic programming visualization and intend to demonstrate its qualities.

Categories and Subject Descriptors

H.5 [**Information Systems**]: Information Interfaces And Presentation (I.7); I.2.11 [Artificial Intelligence] Distributed Artificial Intelligence

Keywords

time-varying information visualization, artificial life, genetic programming, motion, boids

1. INTRODUCTION

2. BACKGROUND

2.1 Genetic Programming

Genetic Programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task [7]. It has been used successfully to evolve electronic circuits [12] or deep space network antennas [5].

The algorithm has six main steps. Figure 1 shows the simplest algorithm used in GP. Of course, it exists numerous variations of each step.

2.1.1 Initial Population

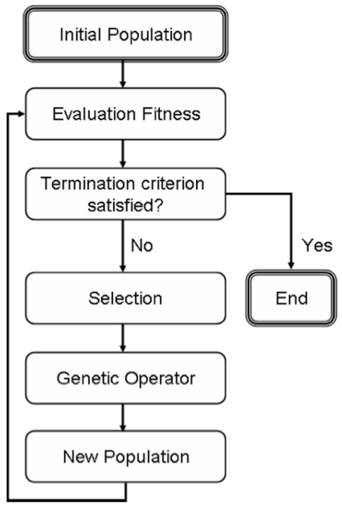


Figure 1: Genetic programming principle

In GP, each individual, which is a candidate solution to the problem, is represented by a tree. During the initialization of the algorithm, the first population is randomly created.

2.1.2 Evaluation Fitness

Each individual is then evaluated using a *fitness function*. This function is a metric to measure how close a individual (or candidate solution) is to the optimal solution.

2.1.3 Termination criteria and selection

The termination criteria could be a specified level of fitness or a maximum number of generation. It this criterion is reached, the algorithm stops. If not, a selection of the individuals with the best fitness is made.

2.1.4 Genetic operators

These selected individuals are combined and modified thanks to genetic operators to get a new generation of individuals. The most common genetic operators in GP are:

- Crossover: two individuals exchange subtrees to create two new individuals,
- Mutation: a subtree of an individual is replaced by a randomly-generated tree,

- Reproduction: an individual is copied into the new generation.

Individuals are again evaluated and the process iterates until the termination criterion is satisfied.

2.1.5 *Evolution parameters*

In GP, a large number of parameters can be adjusted:

- number of generations,
- number of individuals,
- maximum / minimum depth or size for the trees,
- cross-over rate,
- mutation rate,
- choice of the initialization method, choice of the mutation/crossover type,
- etc.

The choice of these parameters is very important because two different configurations can lead to two different evolutions. One configuration can make the GP algorithm to converge and another not.

2.2 Code growth

Despite GP's early successes, it has not scaled well to larger problems. A major factor in this failure has been code growth [14]. Code growth can be described as the tendency of programs generated using GP to grow much larger than is functionally necessary. The growth is out of proportion to improvements in performance and consists almost exclusively of code which does not contribute to performance.

Code growth is a serious issue for GP because it requires the use of resources (for storage and evaluation of the extraneous code) which is disproportional to the performance of the programs being evolved. Thus, understanding and controlling the causes of code growth is a fundamental task if GP is to remain a viable technique.

Currently there are several theories explaining the cause of code growth. Each of these theories depends, to some extent, on the presence of inviable code and the generally destructive nature of the crossover operation. However, each is distinct. It is also worth nothing that these theories are not mutually contradictory. Thus, any or all of them could apply to GP.

2.2.1 *Destructive theory*

McPhee and Miller have argued that code growth occurs to protect programs against the potentially destructive effects of operations other than selection for standard GP, this means crossover [9]. Intuitively, removing a section of code from a reasonably functional program and replacing it with code from a different program would rarely be expected to increase the first program's performance. Several studies have confirmed this intuition showing that few crossovers increase fitness and many lower fitness. This, on average crossover is a neutral or destructive operation. However, it is also clear that a crossover which only affects inviable code cannot be destructive. It is likely that a crossover affecting only inoperative code is also less apt to be damaging than a crossover affecting operative code, but more difficult to prove. Thus, individuals which contain large amounts of inviable code and relatively small amounts of viable code are less likely to have offspring damaged by crossover, and therefore enjoy an evolutionary advantage.

2.2.2 *Distribution theory*

Langdon and Poli have argued that code growth is partially caused by the distribution of semantically equivalent solutions in the solution space [8]. A particular solution can be represented by many semantically equivalent, but syntactically different, programs. The existence of inviable and inoperative code guarantees that for any given program size there are many more larger versions of the solution than there are smaller ones. They argue that as a search progresses it is likely to find progressively larger solutions because there are more of them. Thus, it is possible that some code growth is simply due to the distribution of solutions in the search space, rather than an evolutionary influence.

2.2.3 *Removal bias theory*

It has already been shown that the inviable code always forms subtrees of the entire syntax tree [15]. Thus, the average subtree size of the inviable code is strictly less than the subtree size of the entire syntax tree, assuming the syntax tree is pseudo self-similar. This in turn means that the selection of a relatively small branch for removal increases the likelihood that the branch consists of inviable code. By definition, subtree exchange in inviable code will not change the fitness of the offspring. So, removing a smaller than average subtree makes it more likely that only inviable code will be affected and the offspring will have the same fitness as its parent. Because changes in invisibles code do not affect fitness, the source and size of the replacement branch will not affect the fitness of the new program. Thus there is no corresponding bias in favor of smaller (or larger) replacement branches. Fitness neutral operations will favor the removal of small branches and replacement with average sized branches creating a general increase in size of fitness neutral offspring. Additionally, because most crossovers which are not neutral are destructive, these fitness neutral offspring will be favored during selection and the average program size will increase even in the absence of fitness increase. We refer to this cause of code growth as *removal bias* because it is caused by a bias which favors the removal of small subtrees.

However, all of these three theories have not been proved. Visualizing the GP process could be a good way to give credit to one or another theory.

2.3 Challenge in GP

One of the central element in GP is the fitness function. It directly influences the shape of the fitness landscape and the convergence of the algorithm to an optimal solution [6].

2.3.1 *Fitness landscape*

What is a fitness landscape? First, imagine the space of all possible programs that can be generated by a particular genetic programming system applied to a particular problem. With each program is associated a fitness, a real number which reflects how well this program solves the problem. Then, imagine that this space of all possible programs is mapped into the x-y plane. Finally, imagine that the fitness of each one of these programs is plotted on the z-axis. This creates a surface where the peaks are the locations of programs with poor fitness, and the basins show the locations of the programs with good fitness. Discovering the best solution to the problem then becomes equivalent to searching over this landscape for the deepest basin.

The ruggedness of the landscape has a direct bearing on the difficulty of the problem. Evolutionary techniques such as genetic programming will have increasing difficulty in locating the deepest basins on landscapes that display greater ruggedness. It is perhaps easiest to visualize in the opposite case, that of a landscape with a single large basin, the bottom of which is the best solution. In this case, it is relatively easy for the adaptive processes to proceed directly to the bottom of the basin. In the opposite case, where the landscape is quite rugged, perhaps even locally or globally discontinuous, the individuals may get trapped on local minima without every finding the (or a) global minimum, or the population may simply wander aimlessly across the landscape, in which case the adaptive process has degenerated into random search.

Designing the fitness function to solve a specific problem leading to rugged enough fitness landscape is one of the challenge in the GP system's design. Visualizing the GP process could be a good way to get an idea about the ruggedness of the landscape and therefore give clues for the designer to improve the fitness function.

2.3.2 GP parameters optimization
In GP, there is a large number of parameters that could be adjusted (See section 2.1.5). The choice of a good configuration is very important to make the GP system to converge. For example, a too small population size could slow down the evolution, making the system too slow to converge. A too high mutation rate could completely bias the evolution and prevent the system to converge. A too high mutation rate could allow *big jumps* in the fitness landscape and thus could possibly lead the system to miss the deepest basins (i.e the optimal solutions).

Visualizing the GP process could assess the GP parameters configuration and also help the designer to optimize it.

2.4 Visualization in genetic programming
J. M. Daida et al. presented some methods to visualize the structure of trees that occur in genetic programming [1]. These methods allow for the inspection of structure of entire trees even though several thousands of nodes may be involved. The methods also scale to allow for the inspection of structure for entire populations and for complete trials even though millions of nodes may be involved.

Figure 2 shows a example of visualization proposed in their work. Each graphic represents a population summary for one generation. Generation 1 is shown in the upper left corner and the summary from generation 23, in the lower right corner. As a reference, a gray circle around each summary represents depth level 26.

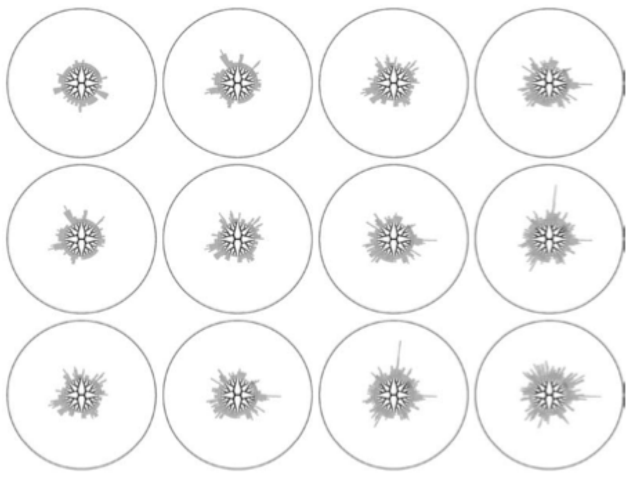


Figure 2: Visualization of the first 12 generations. Each graphic represents a population summary for one generation. Generation 1 is shown in the upper left corner; the summary from generation 23, in the lower right corner. As a reference, a gray circle around each summary represents depth level 26.

However, this type of visualization is static and do not explain the dynamics of GP. It gives a global idea of the evolution but do not allow a exploration at the individual level for example. It is also difficult to visualize the code growth phenomenon and even more difficult to get some clues about the reason of it.

2.5 Time varying data visualization
The following list compares alternative, commonly used methods for visualizing time-varying datasets.

2.5.1 Static State Replacement
These techniques represent data value updates by instantly replacing a static world with another version, mostly because the visualization generation requires considerable calculation efforts. The disadvantage of this approach is that the continuous sequences can be ineffectively perceived as discrete steps [16].

2.5.2 Time-Series Plots
These approaches typically employ time- series plotting, connecting sets of static states that are mapped in space and time with simple curves, stacks or timelines, such as stock market chart line diagrams, web usage bar charts, or line and river metaphors [3].

2.5.3 Static State Morphing
Temporal data can be filtered by directly changing the selected time period, a method also called dynamic queries [13]. Static state morphing techniques interpolate the visualization between different fixed states that accurately represent

the data values within discrete time intervals. Notably, it is the objectÕs motion and not the nature of the motions that carries the informational values. These techniques also differ from information flocking because they require some pre-computation of the static states, and thus are unable to visualize real-time data.

2.5.4 Control Applications

Control applications are like online aggregations and can be classified as anytime algorithms that can produce a meaningful approximate result at any time during their execution. Instead of requiring a dedicated amount of time to build up the scene, a control project gradually streams representative data objects to the visualization system [4].

2.5.5 Equilibrium Attainment

Force-directed diagrams and self- organizing maps show many conceptual similarities with information flocking, as they are internally controlled by local interactions and only reach a state of equilibrium after a specific adaptation time [2]. However, most force-directed methods visualize static datasets and require pre-computed data similarity matrices to determine the spring strengths between pairs of points. Notably, force-directed visualizations only represent informational values by individual distance differences, and do not generate sets of recognizable behaviors as the motion characteristics carry no specific meaning.

2.6 Information flocking

Moere carried out research that demonstrate how principles of self-organisation and behavior simulation can be used to represent dynamic data evolutions by extending the concept of information flocking to time-varying dataset. *Information flocking* is the application to information visualization of mathematical flocking simulation [10]. These swarming movements are generated by behavior rules introduced by Reynolds [11], who successfully modeled the movements of so-called *boids* (or bird-objects) within a *flock*.

The three original flocking rules are the followings:

1. Collision avoidance: Pull away before crashing with other boids nearby.
2. Velocity matching: Attempt to move with the same speed as the neighbors in the flock.
3. Flock centering: Attempt to move toward the center of the flock as the boid perceives.

In Moere's work, these rules are extended with two data-dependent *clustering* rules:

1. Data similarity: Attempt to stay close to those boids that have experienced a similar data value evolution (i.e. similar relative change of stock market prices) during the current database timeframe.
2. Data dissimilarity: Attempt to stay away from boid with dissimilar data values in the current database timeframe.

Moere claimed that implementing two separate data-dependant behavior rules with the traditional flocking behavior rules makes possible to generate a more controllable flocking behavior which can still be stabilized or adapted. Even if the behavior of a flock never attains equilibrium, it could be stabilized by manually fine-tuning both the different weighting factor values for each rules and their threshold distances.

Moere concluded that his research had shown that the original information flocking principles of spacial clustering extended with dynamically generated motion typologies is able to represent similarities in time-varying datasets, or how data evolves over time, both for individual and groups of data objects.

3. INFORMATION FLOCKING APPLIED TO GENETIC PROGRAMMING VISUALIZATION

3.1 Motivation

As seen in section 2.1, GP is a dynamic system. Information flocking has shown potential to convey this feeling of dynamics in the visualization but also makes possible to represent how data evolves over time.

A large number of individuals per generation are evolved. The strong and effective perceptual grouping effect of Information flocking could be used to cluster the individuals in a meaningful way. It could make possible to identify interesting patterns and relationships.

Moreover, GP has several reading levels:

1. the whole evolution from the first generation to the last generation,
2. the generation,
3. the individual

Information flocking has shown some potential to visualize data at different levels. It seems interesting to use this flocking feature to visualize the different levels of a GP system.

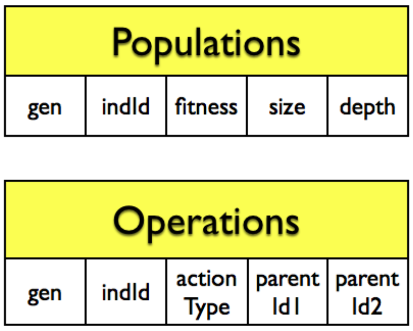
Finally, flocking and GP are two systems based on emergence. In flocking systems, motion emerges from behaviors rules embedded in individuals. In GP, convergence to an optimal solution emerges from selection and recombination of individuals. This common foundation reinforces the coherence of our visualization process.

3.2 Design

In this section, we describe how we intend to use information flocking to visualize the dynamics of GP.

3.2.1 Data

The data we are using comes from one of our other project which aims to use GP to optimize the parameters of sound synthesizers. In order to get the complete description of the evolution, we are crossing two datasets which are stored into two tables within a database (see Figure 3).



Populations				
gen	indId	fitness	size	depth

Operations				
gen	indId	action Type	parent Id1	parent Id2

Figure 3: Database architecture

The first one contains the population of every generations. Each entry is a individual characterizes by:

- the generation it belongs to (gen),
- its unique id (indId),
- its fitness (fitness),
- its size (size),
- its depth (depth).

The second table contains the actions performed by the genetic operators (selection, cross-over, mutation):

- the generation it is applied to (gen),
- the individual id it is applied to (indId),
- the type of genetic operator (type),
- the first parent id if it is a crossover or the individual id on which the operator is applied if it is a mutation or empty if selection (parentId1),
- the first parent id if it is a crossover or empty otherwise (parentId2),

The population of the first generation is initialized with the population table. To get the next generations, the genetic operators are applied on the previous generation. The actions performed by the genetic operators on the previous generation are obtained by requests on the actions table. The individuals characteristics are updated by requests on the population table.

3.2.2 *Environnement*

For the first implementation, we choose to use a 3D environment. The advantage of 3D on 2D is that there is one more dimension. An usual generation in GP contains hundreds of individuals. We are expecting that one more dimension could improve the clustering power of the flocking algorithm and make the emergent patterns and relationships more visible.

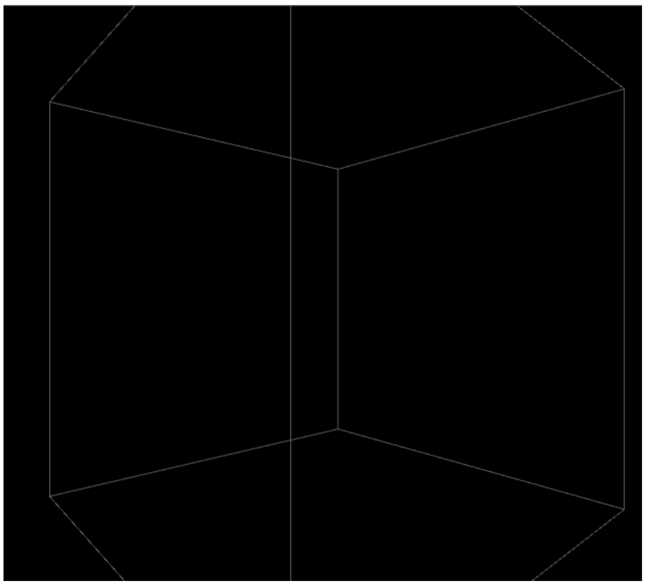


Figure 4: Cube environment

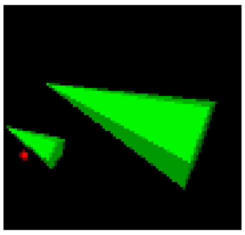


Figure 5: Example of an current individual

Figure 4 shows the black cube environment. We chose black as background color because it facilitates the visualization and differentiation of colored individuals. It is also less tiring for user than brighter color.

3.2.3 *Particule*

Each individual is represented with a cone. The color encodes the fitness of the individual. We chose a color gradient from red to green for a fitness from low to high. Green is a positive color and represents a high fitness and red a negative color and represents a low fitness. Moreover, green and red are cognitively easy to differentiate.

The tree size and depth of the individual are used to define the shape of the cone (see Figure 5). It would makes possible to compare at first size the relative size between two individuals.

3.2.4 *Genealogy visualization*

In our visualization, we wanted to represent the genealogy of the individuals. For a given generation, we represent not only the current population but also the directs parents of the individuals. To differentiate the parents from the current individuals, the parents are drawn with white lines contour (see Figure 6.

Crossover are represented with two blue lines linking the

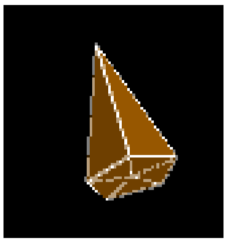


Figure 6: Example of an parent

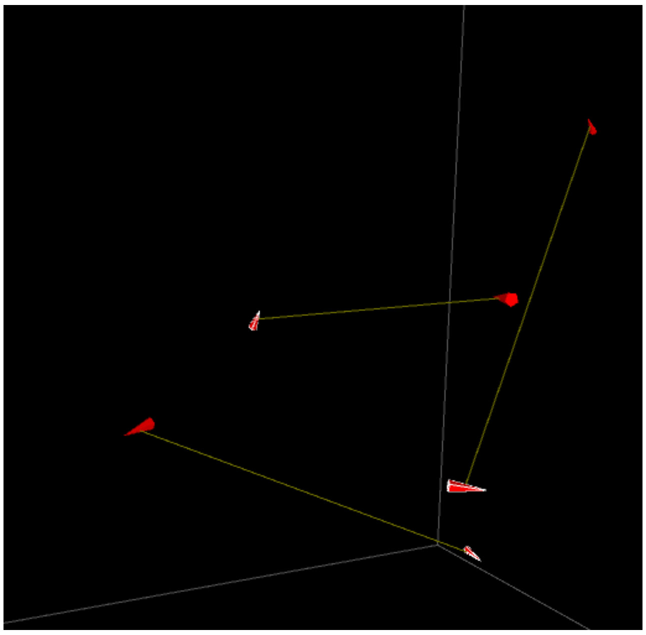


Figure 7: Mutation representation

parents to their offspring (see Figure 8). Mutation are represented with one yellow line linking the original and mutated individual (see Figure 7).

3.2.5 *Interactivity*

The user can interact with the visualization:

- The camera can be moved and zoomed to explore the 3D space,
- The visualization can be paused,
- The user can activate or deactivate the genealogy visualization,
- The user can increment the current generation,
- The user can click on an individual to get its characteristics and visualize its tree and its parents' trees.

This limited interactivity facilitates the switching between the different reading level. For example, the user can focus on visualizing the data dynamics deactivating the genealogy visualization or she can focus on the individual level following one specific individual on the screen and clicking on one individual to get more precise data.

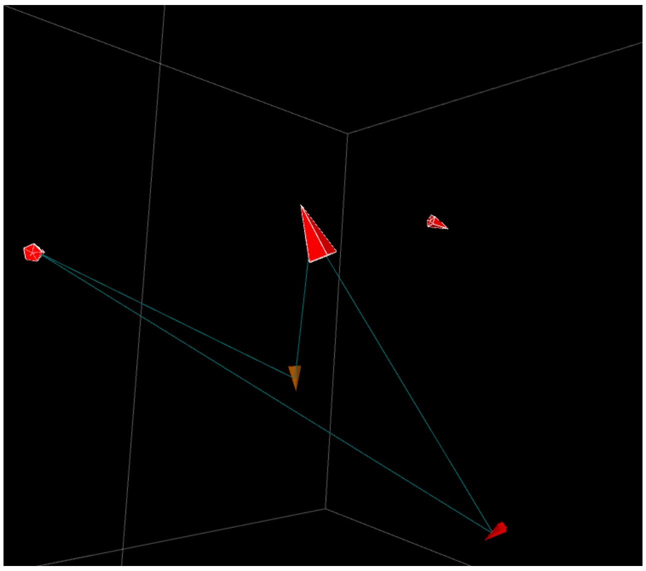


Figure 8: Crossover representation

3.2.6 *Behavior rules*

The direction and speed of each boid A with position $\overrightarrow{p_A}$, is dependent on all the boids X with position $\overrightarrow{p_X}$ in its neighborhood. Following flocking rules apply, which are executed for each single boid and at each single frame.

Collision Avoidance. Pull away before crashing with other boids nearby. If the distance between the boids is within the collision avoidance range d_{CA} , calculate a directional vector pointing away, of which the strength is inversely proportional to the distance between the boids.

$$||\overrightarrow{p_X} - \overrightarrow{p_A}|| <= d_{CA} \rightarrow \overrightarrow{v_{CA}} = \sum_X \frac{|\overrightarrow{p_X} - \overrightarrow{p_A}|}{||\overrightarrow{p_X} - \overrightarrow{p_A}||}$$

Velocity Matching. Attempt to move with about the same speed as the neighbors in the flock. If the distance between the boids is smaller than the velocity matching threshold d_{VM}, boid A should attempt to take over the direction of boid X.

$$d_{CA} < ||\overrightarrow{p_X} - \overrightarrow{p_A}|| <= d_{VM} \rightarrow \overrightarrow{v_{VM}} = \sum_X \overrightarrow{v_X}$$

Flock Centering. Attempt to move toward the center of the flock as the void perceives it. If the distance between $\overrightarrow{p_X}$ and $\overrightarrow{p_A}$ is smaller than the flock centering limit d_{FC}, boid X should try to direct itself towards void A.

$$d_{CA} < ||\overrightarrow{p_X} - \overrightarrow{p_A}|| <= d_{FC} \rightarrow \overrightarrow{v_{FC}} = \sum_X |\overrightarrow{p_X} - \overrightarrow{p_A}|$$

These three traditional flocking principles are solely based upon the relative spatial positions of the pairs of boids, and are extended with two separate data-dependent clustering rules. These rules rely on the relative change q_{change} in the fitness value from the parents for crossover (q_{parent}) or original individual for mutation ($q_{original}$) to the current individual ($q_{current}$). Only one of both rules is valid for each pair of boids A and X, depending on whether the similarity between the two data alterations is below a specific threshold $t_{threshold}$. In addition, a weight factor W_{DS} between boid A and X is introduced, that is proportional to their data alteration similarity and will influence the strength of the resulting attraction or repulsion force.

Equation 1 shows how q_{change} is calculated when the individual comes from a mutation.

$$\frac{q_{current}(A) - q_{original}(A)}{q_{original}(A)} = q_{change}(A) \tag{1}$$

Equation 2 shows how q_{change} is calculated when the individual comes from a crossover.

$$\frac{q_{current}(A) - \frac{q_{parent1}(A)+q_{parent2}(A)}{2}}{q_{original}(A)} = q_{change}(A) \tag{2}$$

Equation 3 and 4 show how the weight factor W_{DS} between boid A and X is calculated.

$$q_{change}(A,X) = |q_{change}(A) - q_{change}(X)| \tag{3}$$

$$w_{DS}(A,X) = \frac{|q_{threshold} - q_{change}(X)|}{q_{threshold}} \tag{4}$$

Data Similarity. Attempt to stay close to those boids that have experienced a similar data value evolution during the current generation. The data alteration similarity, in this case the relative change of fitness, is determined by calculating the difference between the data value evolutions of the boids. The strength of the attracting force is proportional to the distance between the boids and the similarity between the two data value evolutions.

$$\left.\begin{array}{r} ||\overrightarrow{p_X}-\overrightarrow{p_A}||<=d_{DS} \\ q_{change}(A,X)<q_{threshold} \end{array}\right\} \rightarrow \overrightarrow{v_{VM}}=\sum_X w_{DS}(A,X)|\overrightarrow{p_X}-\overrightarrow{p_A}|*||\overrightarrow{p_X}-\overrightarrow{p_A}|| \tag{5}$$

Data Dissimilarity. Attempt to stay away from boids with dissimilar data values in the current generation. This rule is similar to the previous data similarity influence, except that the repulsion force and the distance between the boids are inversely proportional.

$$\left.\begin{array}{r} ||\overrightarrow{p_X}-\overrightarrow{p_A}||<=d_{DD} \\ q_{change}(A,X)>q_{threshold} \end{array}\right\} \rightarrow \overrightarrow{v_{VM}}=\sum_X w_{DS}(A,X)\frac{|\overrightarrow{p_X}-\overrightarrow{p_A}|}{||\overrightarrow{p_X}-\overrightarrow{p_A}||} \tag{6}$$

$0<w_{CA},w_{VM},w_{FC},w_{DS},w_{DD}<1$ are weights applied to Collision Avoidance, Velocity Matching, Flock Centering, Data Similarity and Data Dissimilarity behaviors respectively.

$\overrightarrow{v_A}, \overrightarrow{v_{CA}}, \overrightarrow{v_{VM}}, \overrightarrow{v_{FC}}, \overrightarrow{v_{DS}}, \overrightarrow{v_{DD}}, \overrightarrow{v_A}$ denote the accumulated velocity vectors.

$$||\overrightarrow{v_X}|| > 1 \rightarrow normalize(\overrightarrow{v_X})$$

$$\overrightarrow{v_A}=-w_{CA}*\overrightarrow{v_{CA}}+w_{VM}*\overrightarrow{v_{VM}}+w_{FC}*\overrightarrow{v_{FC}}+w_{DS}*\overrightarrow{v_{DS}}-w_{DD}*\overrightarrow{v_{DD}}$$

The behavior of a flock never attains equilibrium, and expresses continuing characteristics but also unpredictable patterns. Accordingly, it is necessary to stabilize this behavior as much as possible by fine-tuning both the different weighting factor values and the threshold distances. The exact numerical values of these variables are determined through a process of trial and error, as the application designer is mostly unable to foresee the exact outcomes of the simultaneously applied local interactions.

4. PROTOTYPE IMPLEMENTATION

The prototype was implemented with Processing with the graphical library: toxiclibs. The database used was PostgreSQL.

5. DISCUSSION

Applying information flocking to visualize GP, we were expecting to outlay dynamic patterns such as a sudden improvement of the fitness for a group of individuals. Using the interactivity feature, we could analyze the reasons of this improvement checking the trees of the individuals and their parents. It could be a good way to understand which mutation or crossover were useful to improve the fitness.

The number of flocks could also be interpreted as an indicator for the *ruggedness* of the fitness landscape. An unique flock could indicate a very flat landscape and a large number of flocks a very rugged landscape.

With this visualization, it is possible to look at the whole evolution. Increasing the generation, the user can see the formation of flock and the change in the global color. For example, if the algorithm is converging, the global color should gradually change to green. It is also possible to observe some interesting event such as sudden improvement or decrease in fitness.

With this visualization, it is also possible to focus on the individual level. For example, if a individual is emerging from the flocks, the user can get all the characteristics of this individual as well as its tree and parent's tree.

Code growth could also be observed with the global size increase of the individuals. Looking at the generations when this phenomenon is amplifying and comparing the trees before and after mutation or crossover could make possible to confirm one of the theory exposed in section 2.2.

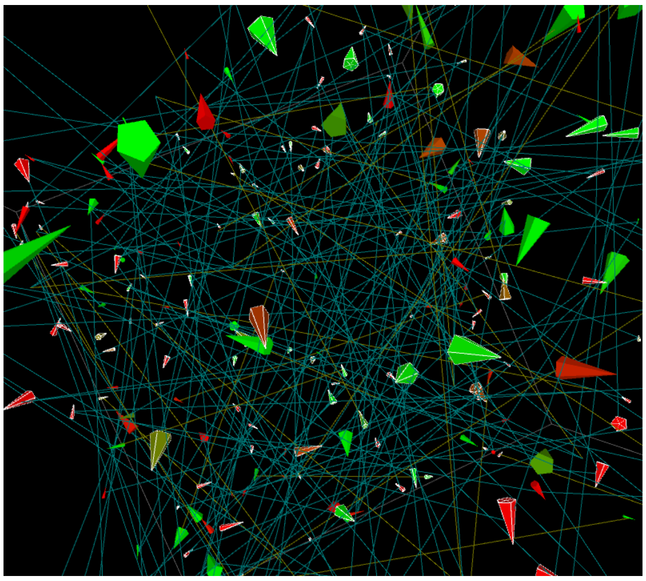


Figure 9: Screenshot of a large population

However, our implementation was good enough to make possible all these expected behaviors. Figure 9 shows a screenshot taken with a large population. The genealogy representation seems not viable when the population is too big. It becomes impossible to read the relationship between the individual.

The implementation of the behavior rules was too poor to evaluate their performance to cluster the individual and identify interesting patterns. It was very difficult to find a good set of weight coefficients to stabilize the system and make it interesting.

Moreover, the choice of a 3D environment . With a black background, it is really difficult to get a real sense of depth. Therefore, the comparison between the size of two individuals is bias. Navigating in the 3D space was not as easy as expected.

6. CONCLUSIONS AND FUTURE WORKS

Information flocking seems to have potential to visualize GP dynamics. However, the current implementation is not good enough to draw conclusions.

We intend to switch back to a 2D environment which is easier to navigate and to read. In addition of mapping the tree size and depth to the size of the boid, we are also thinking about mapping it to its speed. It could result, for instance, to a global slow down of the population with the code growth. A lot of work has also to be on the proper implementation of the behavior rules to be able to evaluate their performance.

7. REFERENCES

[1] J. Daida, A. Hilss, D. Ward, and S. Long. Visualizing tree structures in genetic programming. In *Genetic and Evolutionary ComputationâĂŤGECCO 2003*, pages 211–211. Springer, 2003.

[2] P. Eades. A heuristic for graph drawing. *Congressus numerantium*, 42:149–160, 1984.

[3] S. Havre, E. Hetzler, P. Whitney, and L. Nowell. Themeriver: Visualizing thematic changes in large document collections. *Visualization and Computer Graphics, IEEE Transactions on*, 8(1):9–20, 2002.

[4] J. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. Haas. Interactive data analysis: The control project. *Computer*, 32(8):51–59, 1999.

[5] W. Imbriale. Evolution of the large deep space network antennas. *Antennas and Propagation Magazine, IEEE*, 33(6):7 –19, dec 1991.

[6] K. Kinnear Jr. Fitness landscapes and difficulty in genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 142–147. IEEE, 1994.

[7] J. Koza and R. Poli. Genetic programming. *Search Methodologies*, pages 127–164, 2005.

[8] W. Langdon and R. Poli. An analysis of the max problem in genetic programming. *Genetic Programming*, 1(997):222–230, 1997.

[9] N. McPhee and J. Miller. Accurate replication in genetic programming. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309. Citeseer, 1995.

[10] G. Proctor and C. Winter. Information flocking: Data visualisation in virtual worlds using emergent behaviours. In *Virtual Worlds*, pages 168–176. Springer, 1998.

[11] C. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

[12] C. Santini, R. Zebulumi, M. Pacheco, M. Vellasco, and M. Szwarcman. Evolution of analog circuits on a programmable analog multiplexer array. In *Aerospace Conference, 2001, IEEE Proceedings.*, volume 5, pages 2301 –2308 vol.5, 2001.

[13] B. Shneiderman. Dynamic queries for visual information seeking. *Software, IEEE*, 11(6):70–77, 1994.

[14] T. Soule. *Code growth in genetic programming.* PhD thesis, University of Idaho, 1998.

[15] T. Soule and J. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 781–786. IEEE, 1998.

[16] B. Tversky, J. Morrison, and M. Betrancourt. Animation: can it facilitate? *International journal of human-computer studies*, 57(4):247–262, 2002.