Reusable genomes : Welcome to the green genetic algorithm world

Maurin Nadal, Guy Melançon

Rapport de recherche interne : RR-1469-13 Reusable genomes : Welcome to the green genetic algorithm world

Maurin Nadal

CNRS, LaBRI,UMR 5800
& INRIA Bordeaux Sud-Ouest,
F-33400 Talence, France
Email: nadal@labri.fr

Guy Melanon

CNRS, LaBRI,UMR 5800
& INRIA Bordeaux Sud-Ouest,
F-33400 Talence, France
Email: melancon@labri.fr

***Abstract*—The root motivation of this paper is to assist a novice user for graph drawing by generating different drawing for his data and let him choose the one which best fits his needs. We chose to use a genetic algorithm to generate these drawings. This paper focus on two main contributions which are ones of the first steps to achieve this goal. The first one consists in changing the way graph drawing GAs generally encode the genome. This allows a better abstraction between genotype and phenotype. In our project, each genome encodes a set of parameters for one vertex. Then a modified force-based genetic algorithm compute the resulting layout which is used to compute the fitness of each vertex genome. The second contribution is an enhancement of genetic algorithms (GAs) and cased-based genetic algorithms to make genomes more reusable. This evolution allows for an important increase in speed after a learning phase and also allows for an increase in the quality of the results and in the solution space coverage.**

I. Introduction

Genetic algorithms (GAs) are a well-known paradigm for solving complex problems. Many projects use them to efficiently explore multi-dimensional spaces and to optimize a defined fitness function. However, they require a significant quantity of computation, and the classical definition of a GA is sometimes not as efficient at addressing problems with a large solution space.
Furthermore, for problems presenting multiple similar instances, some basics computations must be performed at the beginning of each execution. If a quick answer is required, this process could be problematic, particularly for interactive GAs. Another problem is the limited size of simulated populations; compared to a "natural" GA, which utilizes a huge base of available genomes, a simulated GA is limited to the size of its genome pool. This problem can be addressed by keeping some interesting genomes in an auxiliary population, where the GA can pick genomes when a new generation is populated.
The target of this work is to present the evolution of a GA to make genomes reusable. The first step consists of adding contextual and behavioral information to each genome evaluation. The GA is then associated with a database to save the interesting genomes of each execution and to load certain good genomes before the algorithm begins.
This enhancement of the genomes allows them to be described in a new way. Genomes are not only containers for information that become a phenotype after the evaluation but also a part of a transformation function from a context to a behavior. If the added information is sufficiently precise, then it is possible to evaluate the new genome with a knowledge base by comparing the transformation associated with the genome to those saved in the database.
This method has been applied to graph drawing. The target of the implemented GA is to generate a different drawing for the same graph upon the user's request. The user can then choose the ones he likes, and the algorithm can further explore the space in this direction. This method provides several types of use to be considered: the learning phase, open-ended exploration, a quick calculation to answer the user's request, and the further enhancement of selected drawings. To address these multiple usages, the algorithm structure is completely defined in an xml file. These xml files are built from different base modules. Some are dedicated to interactions with the database, the management of the information contained in context and behavior, the fitness computation, and the population generation. An external server allows one to define different tasks (a set of graphs and a set of xml structure files to use) to automate massive multiple executions.
Figure 1 presents the elements added to the classical GA scheme to allow for genome reusability.

II. Related work

The main motivation of this work is to help novice users in the complex domain of graph drawing. This problem was well defined in the paper of Biedl et al. in 1998, "Graph Multidrawing: Finding Nice drawings without defining nice" [1]. The idea is that the novice user does not initially know what is a good graph for his problem. The proposed solution consists of offering several drawings of the same graph to the user to let him determine by consultation the one that is best for him. The subsequent problem highlight in this paper is the generation of multiple drawings that are both different and interesting.
Numerous projects have already used GAs for graph drawing.

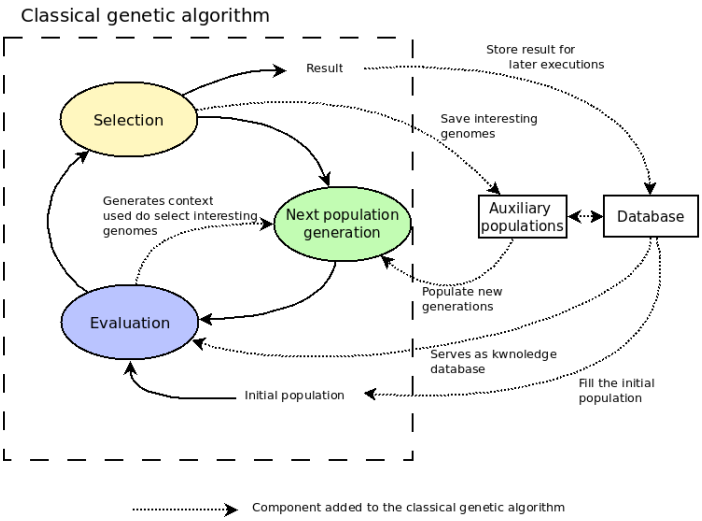
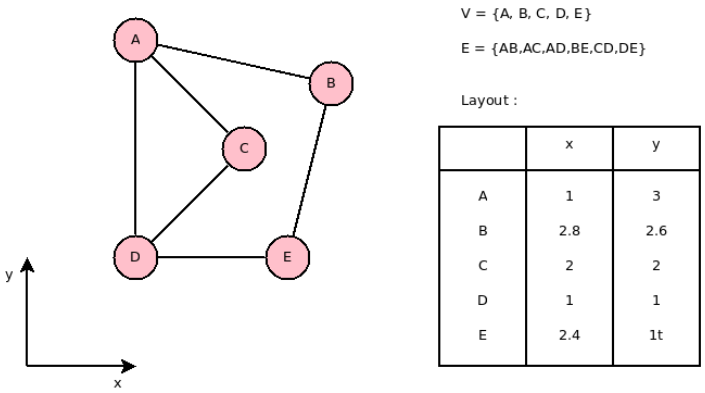


Fig. 1. Some elements have been added to the classical GA scheme



	x	y
A	1	3
B	2.8	2.6
C	2	2
D	1	1
E	2.4	1t

Fig. 2. A simple graph drawn and the corresponding layout.

The first such work was presented by Groves et al. in 1990 [2]. Some enhancements have been regularly published since this date ([3], [4], [5]), based mostly on the description given by Goldberg and Holland in 1988 and 1989 ([6], [7]). New GA features developed by the community have been added one by one to the graph drawing use case. However, some problems persist, mostly with the genetic operators and in the fitness evaluation. These problems will be detailed further in the next section.

Several enhancements that have been added to the first definition of the GA given by Goldberg et al. have influenced our work. The multi-objective evaluation efficiency highlighted for graph drawing by Barbosa et al. in 2001 [8] had a strong influence on the structure of the presented GA. The research on the variable probabilities for the genetic operator performed by Srinivas [9] and the specialization of these operators drive this project to allow for a fully configurable structure to the algorithm. We were also influenced by the work of Goldberg in 1991 on population sizing [10], which showed that the correct population size is complex to determine and depends on many algorithm parameters and on population convergence in particular.

Our system can also be seen as an extension of cased-based reasoning genetic algorithm introduced by Ramsey and Grefenstette in 1993 [11]. This concept was firstly designed to solve similar problems or to enhance adaptation to dynamic environments. The works of Louis et al. focus more on solving similar static problems [12], [13], which corresponds to what we do. The idea is that genetic algorithm efficiency could be increase by initializing the population with good solutions get from previous executions. One of the main contribution of this paper is to enhance this principle by extensively interact with saved solutions, to initialize the population, but also to automate the evaluation and to set up a curiosity guided exploration of space. Ashlock's work on hybridization of population [14] contains also lots of information about how to manage population and their hybridization.

Some ideas have also been taken from other branches of machine learning, particularly the open-endedness research concept presented by Lehman et al. in 2008 [15]. This idea consists of defining a fitness that is calculated from the novelty of the behavior obtained with a genome. This method allows for multiple types of exploration of the space and could also be used as part of a more complex fitness function to avoid local extrema.

As a database is also used in association with the GA, some techniques from the data mining community are very interesting. One method used in this paper, the kd-tree, as presented by Bentley in 1975 [16], is used to index k-dimensional data to easily obtain the neighborhood of a genome; the characteristics of this neighborhood will be more formally defined later in this paper. Classifiers and feature sub-set extraction are also studied but are not yet used.

III. Graph drawing and GAs

A graph is a pair formed from a set of vertices, named V, and a set of the edges linking them, named E. Drawing a graph consists of giving a position to each vertex, and a set of positions is named the "layout". Figure 2 presents a basic, simple graph with five vertices and six edges, with a layout that corresponds to the drawing. Several quality metrics have been defined on the layout; some of the main ones are the counts of the edge crossings and vertex overlapping, which must be minimized, and the ratio between the Euclidean distance in the layout divided by the topological graph distance (the count of the edges in the shortest path between the two vertices), which must be constant. However, there is no mathematical way to define a graph drawing as "beautiful", mainly because it depends upon the user and his needs.

GAs have been used many times for graph drawing, but as far as we know, in all GAs used for graph drawing, the genome directly encodes the layout as a pair of coordinates for each vertex. Some algorithms place also the vertices within a bounded grid to limit the size of the solution space. This approach gives rise to two major problems:

- With regard to the genetic operator and for crossovers in particular, the basic crossover of two layouts consist of taking the position of the first half of the vertices in the first parent and the remainder in the second parent. The major cons of this method are that

most of the properties of the layouts are lost in this process. Unless the two layouts are similar, the count of the edge crossings in the resulting layout may not related to the count of its parents. We make the hypothesis that there is a lack of abstraction between the genotype and phenotype.

- The second problem stems from the reusability of results. When the user wants to draw a graph, unless this graph has already been drawn, it is extremely difficult to used precedent computations. Even if a graph with the same count of vertices has already been drawn, if the set of the edge differs even slightly, then the layout will not be adapted. The entire algorithm must therefore be executed for each graph. This necessity is problematic if the user is waiting for the result of his request. In this case, the algorithm should give a quick answer so that it can at least determine the user's preferences about different types of drawings.

This project aims to correctly address these problems by adding two main contributions :

- To increase the abstraction between the genotype and phenotype, we associate a genome with each vertex, and this genome contains a set of parameters for a force-directed algorithm. This type of algorithm simulates a physical system in which each vertex is a mass and each edge is a spring. The system is simulated until stabilization is achieved and vertices reach their final positions. The parameters consist of the mass of the vertex and the coefficient of the forces applied to it (attraction, repulsion, gravity, and shaking that consists of a random movement). The algorithm used to draw the graph is GEM (Graph EMbedder algorithm), presented by Frick et al. in 1995 [17], with a slight modification to consider a value for each parameter for each vertex. This means that the genotype encode a set of these six parameters (and by the become totally independent of the number of vertices of the graph) and the phenotype is the position of the vertex in the layout resulting of GEM computation.
- For each evaluation of a genome, we associate the context of the evaluation and its behavior after or during the evaluation. This information is stored in a database. Then, in later executions, it will be possible to quickly obtain a set of interesting genomes for some context to obtain quick results for a precise request. The context and behavior will be more formally defined in the following section. This is an extension of cased based genetic algorithm, which already used a problem context to identify similar problems already resolved. The main contribution consists to add the behavior to allow new interactions with auxiliary population, as automatic evaluation and curiosity guided exploration.

A third contribution, which is more a required feature of our framework, consists to use an extremely configurable algorithm. As there is multiple ways to exploit the information attached to evaluated genomes and auxiliary population, each execution of the genetic algorithm can be completely defined by the user. That means to choose how the initial population is created, how each generation is population, the fitness definition, the way to select and to use genomes contained in auxiliary population. Our framework allow to execute multiples executions and compare them easily in order to determine which ones are most effective to achieve our goal.

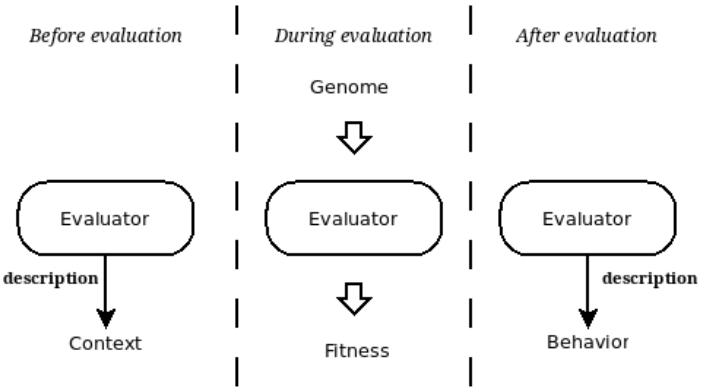


Fig. 3. The context and behavior describe the evaluation environment before and after the evaluation.

IV. Context and behavior

This section defines the context and the behavior more formally. The main objective of these two elements is to find interesting genomes for precise situations. As is shown in Figure 3, the context is generated before the evaluation and describes the environment in which the genome will be evaluated and the behavior is the state of the environment after the evaluation. This last description should be focused on the impact of the genome during the evaluation. Of course, the content of those two elements depends on the use case for which the GA is applied.

A. Definitions

- **Context**: The context must describe, as precisely as possible, the situation in which the genome will be evaluated. It should be calculated before the genome's evaluation, and it must be independent of the genome. For the graph use case, the context is formed from topological metrics, such as the degree, betweenness centrality, closeness, and clustering coefficient of the vertex that will be associated to the genome. All of these metrics are computed from the graph topology (the vertices and edges) and do not require any layout (which is the position of each vertex within a drawing). To the extent possible, the context information must be independent of the execution-related information, which will not be liable in any other execution (as, for example, the ID of the node in a graph).
- **Behavior**: The behavior contains information about the phenotype of the genome. It should describe how a genome "reacts" when it is used in this specific context. In the graph use case, the behavior is formed from graphical metrics that are computed from the layout. The main metrics are the average length of

the node's edges, the average distance of the closest vertices, and the standard deviation of the ratio between the Euclidian distance and graph distance between this node and the other nodes.

The aim of this information is to make possible the selection of interesting genomes in later executions. Interesting genomes will be those that produce a good result in a similar context. For this selection to work, the context and behavior must satisfy certain conditions.

B. Context and behavior quality

Numerous contexts and behaviors could be considered for a use case. The following measures can help the designer to select those that are the most interesting to him. Some are also related to the GAs and the structures of the genomes.

- **Reproducibility**: This parameter could be considered to be the most important quality. Reproducibility means that the same genome used in the same context will always produce the same behavior. This result is not always possible, such as when the phenotype of a genome depends on other genomes that are evaluated at the same time. Nevertheless, the behaviors obtained with a genome/context pair must be as close to each other as possible.
- **Continuity**: This measure could be considered to be an extension of reproducibility: it means that a genome that is reused in a similar context will produce a similar behavior. This parameter is most important when we want to explore some new context because the database will not contain a perfect match.
- **Quantitative comparability**: Context and behavior should be easily and quickly comparable. The result of the comparison must be numeric to be used in selection and evaluation algorithms.
- **Expressiveness**: The context and behavior must precisely describe the evaluation situation and phenotype of the genome. In fact, two different evaluation situations must be associated with different contexts. For example, in a graph drawing use case, if the context is only formed by the degree of the vertex, then it is largely insufficient because two different vertices sharing the same degree can have a different role in the graph. The same constraint is present between the phenotype and behavior.

C. GAs and genome constraints

To efficiently use the context and the behavior, the GA design must satisfy two constraints based on the genome definition:

- **Genome universality**: Any genome must be usable in any context. This constraint must be verified to allow for genome reusability. For example, in graph drawing, the GA that uses a genome formed by the layout of the graph (x and y coordinate for a two-dimensional layout) will not satisfy this constraint because the genome will be usable only for graphs with the same number of vertices.
- **Genome abstractness**: Genomes and the genes that form them must have a sufficiently abstract link with the behavior. The aim of this constraint is to conserve some behavioral property during crossover and mutation. For example, in graph drawing, if the genome is formed by the position of each vertex, then most of the graphical properties will be lost during crossing over with different genomes. In our case, each vertex has a genome, and it contains information about how the vertex will interact with others during the drawing (the coefficient of the force that will be applied to it).

V. GA STRUCTURE

Once the context and behavior have been associated to a genome, it becomes possible to save them apart from the main population to reuse them later. This possibility offers multiple ways to enhance the GA depending on which genomes are saved and when and how they are reused. It is necessary to be able to easily modify the structure of the algorithm to exploit efficiently these multiple possibility.

To achieve this goal, the library uses an xml description in which all of the modules and their interactions are defined. By editing this description, the user will choose how the initial population is generated, and then for each step of the algorithm, he will choose how the step ends, how the population for the next generation is generated, and how the fitness of each genome is computed. The choice of xml external file allow to easily execute multiple execution with the same structure to compare them to others afterwards.

As shown in Figure 4, a description file is made from several blocks. Each block can have different parameters and can allow for a certain type of child block. The root of the structure is always a "GA" block.

A strong separation is made between blocks that are independent from the use case and those that are dependent, mostly to compute the fitness or to select the genomes in an interesting way for the evaluation environment. Sixty-three blocks are currently implemented: 50 for the GA core and 13 for the graph drawing use case. Of course, new blocks can be easily added. An inheritance structure is defined between the blocks; a child slot can accept abstract blocks, and they are filled by any implementation of those abstract blocks.

A. geneticAlgorithm and step blocks

The two most important blocks are the **geneticAlgorithm** and **step** blocks.

The first block is the root at any xml description file, and there is always only one per file. Several subblocks could or must be added to it as follows:

- A population initializer, which determines how the first population is generated. Some genomes can come from the database.
- At least one step (there could be more) that defines the manner in which the algorithm works during the execution. A more precise description will follow.

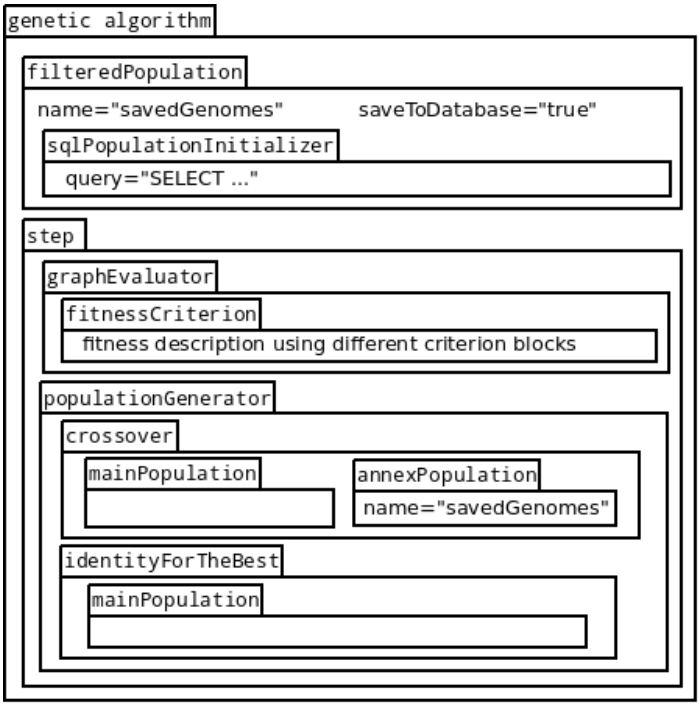


Fig. 4. A simplified GA structure with one auxiliary population and one step

- Some auxiliary population, where genomes can be saved during an execution. It is possible to add a population initializer to fill the population before the launch of the algorithm and a filter to delete or make regular actions on genomes, which is mostly used to delete useless genomes and duplicates. The last possible child is an indexer, which is used to change the manner in which the genomes are picked in the population, allowing for the possibility of contextual picking.

The second block is the **step**. This block was first designed to allow for the evolution of genetic operator probabilities along the GA, inspired by the work of Srinivas [9]. The final design of this block allows one to define the entire workflow of the algorithm with the following subblocks:

- A population generator that is similar to the population initializer but also allow generators, which need a previous population. The user will choose the chance for crossovers, mutations, and every other operator that can be defined for his use case. The method for picking genomes is also very important and must be precisely defined.
- Some terminators, which define when the step will end. The most classical terminator is the **generationNumberTerminator**, but there are also terminators based on the average score, the convergence of the population, or some Boolean factor, which allows the user to define any Boolean operation he needs to determine when a step must end.
- An evaluator used to evaluate genomes and compute fitness. This evaluator must be implemented by the user, and it is entirely dependent on the use case. For the graph drawing use case, a **tulipEvaluator** has been defined to interpret the parameters contained in the genomes and to generate the layout. The fitness computation is based on criteria that allow the user to easily change the fitness definition from one step to another.
- Some genome selectors, which indicate at the end of each generation whether a genome must be saved before deleting. This decision could be made based on the computation of the context, the behavior, the score, the distance between this new behavior and the ones already saved, or even randomly. This selection is applied to every evaluated genome, so it is important that it has a low computational cost.
- A population manager to allow the algorithm to simulate island evolution or any type of parallel evolution.

VI. Auxiliary population and GA configuration

This section presents the basic principle of the enhancements added to the GA. The auxiliary population is explained first, and then, different ways to simulate multi-objective evaluations are shown.

A. Auxiliary population

The auxiliary population is the root of the GA enhancement of this project. This population stores certain genomes so that they can be reused in the current execution or stored in the database. The following three modules are needed to add an auxiliary population to the GA:

1) An auxiliary population to store saved genomes.
2) A selector to indicate when an evaluated genome should be saved. Multiple types of selectors exist, and new ones can be added easily.
3) A genome dealer to insert genomes picked from the auxiliary population during the generation of the new population. These genomes can be used during crossover and mutation or can be added to the main population unchanged, as defined in the population generator block of each step.

A basic usage of auxiliary population consists of defining the following steps for the algorithm:

1) The first step consists of populating an auxiliary population with interesting genomes. This process typically begins with a random population. This step is close to the classical GA; the only difference is that depending on the selector, any evaluated genome could be saved in the auxiliary population.
2) The second step consists of a wide exploration around the saved population. The population generator is mainly based on the saved population. It is also possible to keep a random part of the main population in the next generation to allow for the appearance of unexpected behaviors.
3) The last step consists of exploring the close neighborhood of the saved population by making crossovers and small mutations in the saved genomes. This step had a high convergence to obtain a good final

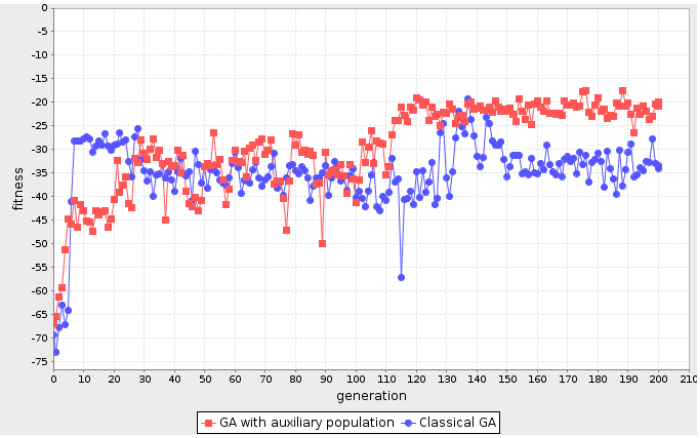


Fig. 5. The system is simulated until stabilization is achieved and vertices reach their final positions.

population. This last population and saved population could be saved in the database if they are interesting (for example, in terms of novelty).

As Figure 5 shows, using an auxiliary population allows for a slight increase of the score and a better stability of the fitness.

B. Multiple objectives evaluation

Multiple objectives evaluations are now fairly common in GAs. In our library, the fitness of a genome must be a simple scalar. However, different methods allow one to simulate multiple fitness levels efficiently. These three methods will be presented in the following section:

1) Multiple auxiliary populations
2) Step with different fitness definitions
3) Non-linear fitness

1) Multiple auxiliary populations: The first and most simple method to simulate multi-objective fitness is to define one auxiliary population by an objective. For each objective, a population will save the best genomes using an associated selector. Then, a part of the main population will come from this auxiliary population.

The user can select the influence of this auxiliary population in the population generator. In the case where a strong specialization is wanted, it is possible to include crossing over or mutation only with the genomes that come from the auxiliary population. However, it is often interesting to cross good genomes for different objectives to make a thorough exploration of the solution space.

It is also possible to fill the auxiliary populations from genomes taken from the database at the beginning of the algorithm to accelerate the process.

2) Step with different fitness definitions: When adding a step in the structure of the GA, the user must define how the fitness is computed. The fitness is typically the same for all steps, but it is also possible to define completely different fitness levels to create an evolution gap. To increase the chance to explore a new part of the space with these gaps, it could be interesting to always randomly save a small part of the main population before a gap.

It is possible to use a different auxiliary population for each step to separate each type of ”good” genome. Then, two approaches are possible for population generation: promote good genomes for the current fitness or use the genomes from previous steps to build a melting pot of all specializations.

3) Non-linear fitness: Much freedom is given to the user for fitness definition. It uses criteria that allow one to describe any operation tree using values obtained from the context, behavior, and fitness as leaves.

All of the basic operations are available (addition, multiplication, exponents). It is also possible to select the maximum of a series of arguments and to use bandpass filter. These last two features provide alternative solutions for simulating multi-objective evaluations. Of course, the following fitness can also be part of a more complex fitness.

Min/Max fitness: When multiple fitness levels are available, it is possible to use a maximum to associate a genome with its maximal fitness. For this to work correctly, the fitness must have been normalized previously.

Adaptive fitness: Bandpass filters allow the user to define complex operations. They allow one to define a first function, to determine whether the filter is active or blocks the signal, and a second function that corresponds to the signal. These filters allow the user to define complex fitness levels, which can be applied to different situations:

- Defining some bonus fitness when a certain score is achieved so that different fitness levels with increasing complexities can be defined.
- Adapting the fitness to the context and the behavior. It is often useful to adapt the evaluation of a genome depending on the context of its evaluation. In graph drawing, an important vertex with a high degree must be drawn differently from a little vertex that is far from the center of the graph.

An advanced application of adaptive fitness will be explained in the next section with the auto-evaluator.

VII. Advanced use of context and behavior

As we have observed, by configuring the GA correctly, it is possible to assess different problems. However, with the increasing amount of genomes available, it could be hard to pick a good genome for a crossover, particularly if the fitness of genomes depends on context. To address this problem, the population can be indexed to more easily find interesting genomes. This technique is the main topic of the next paragraph. Later in this section, we will give more details about other possibilities for context indexing.

A. Context indexing

Two types of indices have been implemented in the library. The first one is an exact indexer, which provides quick access to all genomes sharing the exact same context for which a genome is needed. This type of index is possible in graph drawing because the same graph is drawn several times, as

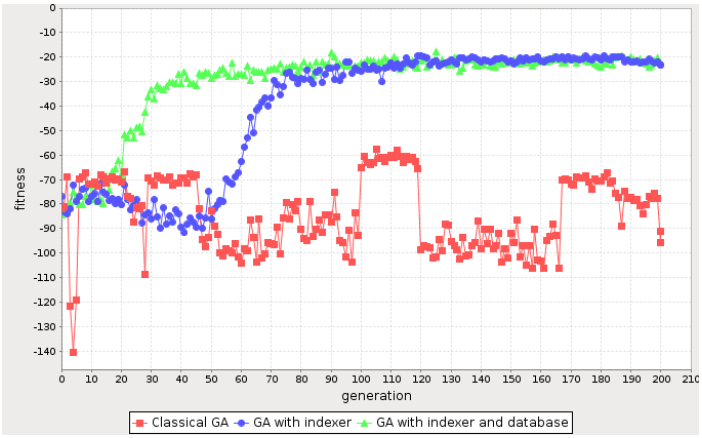


Fig. 6. Score comparison on a larger graph (46 nodes) of a classical GA (red), a GA using an indexer (blue) and a GA using an indexer and the database (green).

the context of a vertex depends only on the topology of the graph, which does not change from one evaluation to another. Of course, this method cannot always be used, but it allows for a good specialization of the genome for a precise context. This index could be as efficient as the data structure allows (for example, in graph drawing, the ID of the node and the graph name are key for the context of this node).

The second type of index allows one to efficiently find the neighborhood of a context to search for genomes that work well in a similar situation. We use a kd-tree as an indexer, for which the user describes the dimensions he wants with criteria. The kd-tree is a variation of a tree allowing for multi-dimensional data indexation. It was introduced by Bentley et al. in 1975 [18], [16].

The index can be attached to a population and modify the manner in which genomes are picked in the population. As each genome is generated for a defined context, the context is given as a parameter of the picking function. This allows one to have large auxiliary populations without the risk of making picking too difficult.

As shown in Figure 6, the classical genetic algorithm is not enough efficient for more complex graphs. Using an index for genome picking slightly slows down the algorithm but increases the maximal score obtained by a genome. When the database is used, the GA converge much quicker to the best score achieved for this graph.

1) Behavior targeting: It is also possible to index a part of the behavior. This could be useful when a certain type of behavior is targeted. It becomes possible to try certain genomes that obtain the desired type of behavior, even if the context is different. This process also allows one to gather a significant amount of information about how genomes and context are linked to produce the behavior.

B. Curiosity-driven exploration

It is sometimes difficult to efficiently explore a large portion of the solution space, particularly when it presents many local maxima. In this situation, getting good space coverage could be really interesting for many applications. The context and behavior can be used to obtain a score depending on the novelty introduced by a genome.

We employ the kd-tree indexer to achieve this goal. The kd-tree allows us to quickly obtain the neighborhood (in terms of the context and behavior) of an evaluated genome. Then, the novelty introduced by the genome corresponds to the distance to its first (or its k-first) neighbors.

It then becomes possible to execute a certain type of GA, where the fitness is defined by the novelty to explore space. By immediately adding the interesting genomes to the indexed population (the one used to compute the novelty), we are also sure that each selected genome is different from the other genomes in the population.

An interesting enhancement of this last execution consists of using the novelty divided by a basic fitness to undergo a sparse exploration in areas that are associated with a bad score and a precise exploration in interesting areas.

This curiosity criterion could also be used in an exploratory step, for example, with a basic first step that makes a classic exploration of the space. This step ends with a generation counter or a threshold on the convergence of the population. Then, the next step is based mainly on the first population and uses a bounded curiosity criterion to explore a limited part of the space around the saved population of the first step. A last step could then work on a refinement of all of the genomes that were found since the beginning of the algorithm.

Moreover, as we will see in the next section, it is important to obtain a wide space coverage in the database to allow for the auto-evaluation of fitness. This curiosity criterion is often used to address this problem.

C. Auto-evaluation of fitness

One of the most complicated tasks in the GA is to define a good fitness function. This problem becomes impossible in the context of an interactive GA, where what is good depends mostly on the user and thus could not be defined at the creation of the algorithm. However, even in non-interactive GAs, a simple function is often not sufficiently expressive to describe a complete fitness that can identify whether a genome is fit in any given situation.

This problem is omnipresent in graph drawing, where it is not possible to define what is a beautiful. This definition changes from one user to another and depends on the usage of the graph and the type of information to be extracted. However, it is possible to find some graph drawings that are considered to be good by most of the community. These graphs make a set of good examples.

It then becomes possible to fill the database with those examples and to create a similarity fitness (by comparing the target drawing to the drawing obtained with the GA) to give a score to each genome (the drawn graph must be one of the example set). Once this fitness has been defined, a learning phase can begin. The curiosity criterion is used to obtain good space coverage, where ”good” means that the maximal distance between two contexts/behaviors is less than the mean of the similarity fitness of the two corresponding genomes.

Once the database has been filled, it is possible to define a new auto-evaluation of the fitness. This last step works by finding a set of neighbors for the genome to evaluate and then inferring the score of this genome and the score of the neighboring genomes.

This type of fitness is quite adaptive because the way to evaluate the behavior depends first on the context. Moreover, it is also possible to make this evaluation on a chosen subset of genomes when the user wants to obtain a certain result.

1) Interactive auto-evaluation: For an interactive GA, it could be interesting to couple the basic fitness and precedent user evaluation to infer the score of a genome. This method has two major advantages:

- The auto-evaluator learns the preferences of the user.
- The distance from a genome and the genomes that have already been evaluated by the user can provide hints regarding the selection of the genome to later be evaluated by the user and can identify the ones that will bring more information to the auto-evaluator.

However, this type of interactive auto-evaluation has not been integrated into the project.

VIII. Conclusion and future works

This paper presents two main contributions, one which concern general genetic algorithm. It consists on an extension of case-based genetic algorithm by adding also behavior related information to evaluated genomes and using auxiliary populations which interact with main population during an execution. Doing so open a lot of possibility on how to drive the genetic algorithm. We also presented how our genetic algorithm is configurable to exploit efficiently these numerous possibilities.
The second contribution is more centred on GA used for graph drawing. We define a new genome encoding paradigm for this context. Instead of encoding a whole layout, genome encode a set of parameters for only one vertex. This allow a better abstraction between genotype (a set of floating parameter) and phenotype (the position in the layout), which is interesting for keeping good traits through genetic operation. This make also the genome independent with the graph drawn (and mostly its vertex number).

However, there is still more work to be conducted, and four major needs have been identified. First, all of these techniques must be applied extensively to our graph drawing use case. This application will allow for a better understanding of how each technique works and how they must be used to maximize their efficiency. Different parameters must be tested, and it also important to couple some of these parameters to learn how they interact.

The second point is focused on data mining. After several executions, the database contains a significant amount of information about the problem, such as how context and behavior are linked and how certain types of genes affect the phenotype. Moreover, to set up a good GA, it is important that the user obtains information about how an execution worked.
A monitoring interface is already present in the control client, but it must be enhanced and must present an easily readable summary of what was appended during the execution. This information will allow the user to determine the precise efficiency of a GA structure.

Another path to explore is the different ways to implement the auto-evaluator. The result obtained with the kd-tree could be enhanced by using a non-linear classifier, such as a decision tree. The solution space is not always continuous, and linear classifiers are not adapted in certain situations. The data mining community is very active in this field and has presented numerous solutions that could be useful for this research.

The last topic to consider is the interactive part of the GA. Due to the structure of the library, this consideration should be possible without heavy modifications. The most complicated task will be to use the user evaluation that will be stored in the database in an efficient manner.

References

[1] T. Biedl, J. Marks, and K. Ryall, "Graph multidrawing: Finding nice drawings without defining nice," *Graph Drawing*, 1998.

[2] L. Groves, Z. Michalewicz, P. V. Elia, and C. Z. Janikow, "Genetic algorithms for drawing directed graphs," *for Intelligent Systems,*, 1990.

[3] A. Markus, "Experiments with Genetic Algorithms for Displaying Graphs," pp. 62–67, 1991.

[4] C. Kosak, J. Marks, and S. Shieber, "Automating the Layout of network diagrams with specified visual organization," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 3, pp. 440–454, 1994.

[5] U. A. Gen, D. Grafos, and D. T. Eloranta, "TimGA: A Genetic Algorithm for Drawing Undirected Graphs," *Computer*, vol. 9, no. 2, pp. 155–171, 2001.

[6] D. Goldberg and J. Holland, "Genetic algorithms and machine learning," *Machine Learning*, pp. 95–99, 1988.

[7] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, addison we ed., A. Wesley, Ed., 1989.

[8] H. Barbosa and A. Barreto, "An interactive genetic algorithm with co-evolution of weights for multiobjective problems," *Proceedings of the Genetic and Evolutionary ...*, pp. 203–210, 2001.

[9] M. Srinivas and L. Patnaik, "Adaptive probabilities of crossover and mutation in genetic algorithms," *Systems, Man and Cybernetics, IEEE ...*, vol. 24, no. 4, pp. 656–667, 1994.

[10] D. Goldberg, K. Deb, and J. Clark, "Genetic algorithms, noise, and the sizing of populations," *Complex systems*, 1991.

[11] C. Ramsey and J. Grefenstette, "Case-based initialization of genetic algorithms," ... *Conference on Genetic Algorithms*, 1993.

[12] S. Louis and J. Johnson, "Solving similar problems using genetic algorithms and case-based memory," ... *International Conference on Genetic Algorithms*, 1997.

[13] S. Louis and J. McDonnell, "Learning with case-injected genetic algorithms," *Evolutionary Computation, IEEE ...*, vol. 13, no. 0704, 2004.

[14] D. Ashlock, K. Bryden, and S. Corns, "Small population effects and hybridization," ... *Computation, 2008. CEC ...*, 2008.

[15] J. Lehman and K. Stanley, "Exploiting open-endedness to solve problems through the search for novelty," *Artificial Life*, vol. 11, no. Alife Xi, p. 329, 2008.

[16] J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, 1975.

[17] A. Frick, A. Ludwig, and H. Mehldau, "A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration)," *Graph Drawing*, 1995.

[18] J. Friedman, J. Bentley, and R. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on ...*, vol. 1549, no. July, 1977.