

Parallel Genetic Algorithm for Regression

Paulo Santos and Maria Fidalgo

Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa
`{fc47806,fc49034}@alunos.fc.ul.pt`

Abstract. The first sentence of an abstract should clearly introduce the topic of the paper so that readers can relate it to other work they are familiar with. However, an analysis of abstracts across a range of fields show that few follow this advice, nor do they take the opportunity to summarize previous work in their second sentence. A central issue is the lack of structure in standard advice on abstract writing, so most authors don't realize the third sentence should point out the deficiencies of this existing research. To solve this problem, we describe a technique that structures the entire abstract around a set of six sentences, each of which has a specific role, so that by the end of the first four sentences you have introduced the idea fully. This structure then allows you to use the fifth sentence to elaborate a little on the research, explain how it works, and talk about the various ways that you have applied it, for example to teach generations of new graduate students how to write clearly. This technique is helpful because it clarifies your thinking and leads to a final sentence that summarizes why your research matters.

Keywords: Parallel Genetic Programming · Regression · Island Model.

1 Introduction

Genetic Algorithms (GAs) are metaheuristic searching algorithms where the main idea lies in following the same principles as Natural Selection and Evolution in Biology [12]. That is, the algorithms work around a *population* of potential solutions for the problem, the *individuals*. The population changes during the execution of the algorithm, mimicking the evolution of a real population of living beings, from generation to generation, where the fitter individuals are more likely to survive and reproduce.

The algorithm is composed by three main operations: Fitness, Crossover, Mutation and Selection. Fitness measures how good an individual is; Crossover (or sexual reproduction) generates a new solution based on two existing ones, simulating the breeding of two individuals; and Mutation (or asexual reproduction) produces random changes to an individual; Selection determines how the individuals are chosen for the Crossover and Mutation [7].

Due to the characteristics of the GAs, applications are often related to optimization [12], such as the Traveling Salesman Problem [5], classification [2], decision making [4] and prediction [3].

Prediction can be accomplished through Regression Analysis, which is the task of modeling a random variable Y as a function of a vector of random variables X . This may be translated as the task of finding the mathematical expression best suited to explain Y . Regression models presuppose the existence of constants, called *parameters*, that are to be estimated from the data [11].

However, when using a GA for this type of problem, it is not the parameters that we want to estimate but the whole mathematical expression. Therefore, we will be looking for the one that minimizes the error between the value provided by the model and the actual value.

In this work, we explore different parallelizations of the Genetic Algorithm for regression, using the toxicity dataset [6]. We will start by introducing the algorithm and its operations in further detail. Next, we will present the several approaches studied:

- Sequential
- Adaptive Sequential
- Trivial Parallelization
- Island Parallelization

Finally, we will demonstrate the experimental evaluation and compare with each other.

2 Genetic Algorithm

The implementation of a GA typically starts with a population of random individuals. Then, the population is evaluated, and reproduction operations take place [13]. Bellow, we present the pseudo-code for the genetic algorithm. The *AddElitesToNewPopulation* is an optional but often used step, where a designated percentage of the best individuals are directly transfered to the new population (without any modification). This helps the algorithm to converge faster and prevents it from losing the best solutions [10].

Input: $Max_{gen}, Pop_{size}, P_{cross}, P_{mut}, P_{reproduction}$

Output: S_{best}

$Population \leftarrow InitializePopulation(Pop_{size}, nodes_{func}, nodes_{term})$

$MeasureFitness(Population)$ ▷ Evaluation of the population

$S_{best} \leftarrow GetBestSolution(Population)$

$Generation_i \leftarrow 0$

while $Generation_i \neq Max_{gen}$ **do**

$Generation_i \leftarrow Generation_i + 1$

$NewPopulation \leftarrow \emptyset$

$AddElitesToNewPopulation(Population, NewPopulation)$ ▷ Optional

step

while $Size(NewPopulation) < Pop_{size}$ **do**

$Ind_1, Ind_2 \leftarrow SelectForCrossover(Population, P_{reproduction})$

$NewGeneration \leftarrow NewGeneration \cup Mutate(Crossover(Ind_1, Ind_2, P_{cross}), P_{mut})$

```

    MeasureFitness(NewPopulation)
     $S_{best} \leftarrow GetBestSolution(NewPopulation)$ 
    Population  $\leftarrow NewPopulation$ 
  return  $S_{best}$ 

```

3 Approach

3.1 Encoding

Originally, binary encoding was used to encode the solution [13]. That is, an individual was represented by a vector where each entrance corresponded to some feature being true for that particular individual. Later, other types of encoding were proposed in order to represent more sophisticated types of individuals, like the value encoding, permutation encoding, and trees [10]. We chose the last one for our work, since our goal is to encode expressions, that can be directly mapped to Abstract Syntax Trees.

Choosing an Abstract Syntax Tree for the encoding allows us to easily generate random mathematical expressions with an immutable tree. Each node of the tree either represents a binary operator ($+$, $-$, $*$, *wholdivision*) node or a constant node, which can be a variable or value (any integer from -1000 to 1000).

The variable names were generated according to the dataset: from x_1 to x_n , where n is the number of features from the dataset.

3.2 Fitness

The fitness function is the main responsible for the efficiency of the GA, so it must be wisely selected. In order to perform regression on the given data, it is necessary to bind the variables from the expression being evaluated and the corresponding data values. After that, the error for each data entry is computed. Below there is an algorithm line-up for the fitness measurement.

```

Input: Data, DataOutput, Expression, Popsize
Output: RMSE
errors  $\leftarrow 0$ 
BindedData  $\leftarrow BindDataToExpression(Data, Expression)$ 
for DataRow $i$  in BindedData do
    errors  $\leftarrow errors + (DataOutput_i - ExpEval(BindedData_i, Expression))^2$ 
RMSE  $\leftarrow \sqrt{\frac{errors}{Pop_{size}}}$ 
return RMSE

```

Normally, the fitness function is the objective function to be maximized throughout the execution. Instead, we opted to try to minimize the fitness. That decision is based on the fact that our goal is to minimize the error. We chose to use the Rooted Mean Squared Error (RMSE) as the measure for the error between the data output and the predicted output. It is easy to see that this operation is quite heavy, and depends a lot on the size of the dataset. To alleviate the load, we used an approximate function evaluation [?] for the first 250

generations. The approximate approach is based on the division of the training set in parts and assign each to each generation, in turns.

3.3 Selection

There are many different strategies to choose the individuals for crossover and mutation, like tournament selection or rank selection. Nevertheless, we chose the most popular one, known as proportionate selection (or roulette-wheel selection). In this strategy, the probability of an individual being chosen is directly proportionate to its fitness [10].

3.4 Crossover and Mutation

For crossover and mutation there also are multiple strategies that can be chosen. For the sake of simplicity, we opted by following the simpler approaches: one-point crossover and the uniform mutation [10].

One-point crossover is accomplished by randomly selecting a point from both individuals selected and assigning the first part of the first parent to the first part of the offspring and the second part of the second parent to the second part of the offspring.

The uniform mutation consists of randomly choosing one node and changing its inner value uniformly from the available values. A constant node could have its inner content changed with a new value or variable, while a binary operator node would have its operation changed to a random one (each equally likely of being chosen).

Note that, in the context of our work, the point is one of the nodes of the tree. So, to choose the node randomly, we developed the algorithm that follows:

3.5 Parallelization

4 Implementation Details

4.1 Abstract Syntax Tree

As mentioned above, Abstract Syntax Trees allow us to generate random mathematical expressions. Moreover, by using a Java expression builder¹, it was possible to easily map values to variables of a given expression and obtain the result from it, which is useful when calculating the fitness of the tree.

4.2 Adaptive Genetic Algorithm

The Adaptive implementation of a GA diverges from the regular implementation by not having the mutation and crossover rates fixed. Instead, the mutation

¹ exp4j - Expression Builder from String for Java

and crossover rates are dynamic, and adapt depending on the progression made by each operation in the last generation [8].

The new offsprings are generated by crossing parents using an Adapted Fitness Proportionate Selection. Since new offsprings are always generated by crossing two individuals, the crossover rate will influence the probability each individual has of being chosen. This has been attained by using the absolute value of the normal probability density function, and multiplying it by the sum of a third of the amount of population with the crossover rate. The crossover rate is an integer value between $-populationSize$ and $populationSize$.

$$\left| \left| \frac{1}{\sigma\sqrt{2\pi}} * e^{-\frac{(-\mu)^2}{2\sigma^2}} * (popSize / 3) + crossoverRate \right| \right| \% popSize$$

The mutation rate controls the probability of an individual to suffer a mutation. This value is a value between 0.05 and 1.0.

Both rates are updated according to the progressions they obtained for each individual. The chosen mutation rate offset is 0.05 and the crossover rate offset is $populationSize * 0.025$.

```

if mutationProgress < crossOverProgress then
    mutationRate := min(1.0, mutationRate + mutationOffset)
    crossoverRate := max(-popSize, crossoverRate - mutationOffset)
if mutationProgress > crossOverProgress then
    mutationRate := max(mutationOffset, mutationRate - mutationOffset)
    crossoverRate := min(popSize, crossoverRate + mutationOffset)

```

4.3 Parallel Versions Implementation

4.3.1 Parallel Population Sorting

Due to parallel mergesort ease of implementation and advantages of execution time [9], the sorting of the population by fitness has been implemented with a parallel mergesort with linear insertion sort when the amount of population to compute is smaller than an offset of 7. The population is splitted by half until the offset is reached and the insertion sort and merge algorithms' are computed.

4.3.2 ForkJoin

TODO:

4.3.3 Phaser

The Phaser approach uses phasers in order to introduce a synchronization point between all the threads running.

First of, there has been made the decision of creating N threads proportional to the amount of available processors of the machine. A smaller value would not take advantage of all the processing power available, whereas a higher value would have a negative impact on the performance, since the CPU scheduler

would give each one of those $N > availableProcessors$ some share of CPU time.

Every thread is responsible for a portion of the population. In order to maintain consistency in the population over the several generations, and between different threads, it is required a synchronization point, the phaser.

The synchronization point has been introduced before and after the transition from the old population to the new one. This is a requirement since the GA operations cannot happen until every single expression from the new population has been introduced in the current population.

Additionally, it was also introduced synchronization points before and after sorting the population. After every single individual has transitioned from the old to the new population, the thread with *threadId* := 0 launches the Parallel Population Sorting algorithm. Once the algorithm finishes, every thread gets through the synchronization point and a new generation starts.

The phaser runs the regular GA approach to compute each individual.

4.3.4 Island

In order to take advantage of the amount of available processors we have also implemented an Island Model for the GA[14].

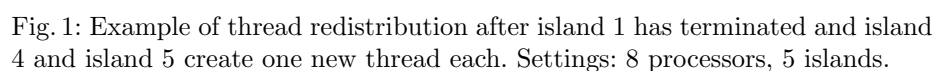
In our approach we took into account the quantity of available processors to create and hold the computation of the islands. Each island has its own portion of the population, with size *populationSize/amountIslands*, and is independent of the other islands. Consequently, synchronization points are not required. Occasionally - every 20 generations - the best individual of an island is sent to a random island and waits in a *ConcurrentLinkedQueue* until the sort starts, when the last (worst) individual is replaced with the new one.

A naive approach would be to create one island per available processor. Our implementation allows us to create a variable amount of islands between 1 and the amount of available processors.

If the specified amount of islands is smaller than the amount of processors then the remaining available processors will be evenly distributed through the islands, allowing these to contain inner parallelization. The inner parallelization allows the island to split the population with the amount of threads it has been assigned and compute each generation quicker. As in the Phaser approach, phasers were also used here to create synchronization in the points mentioned earlier. In this case the island is responsible for calling the Parallel Population Sorting algorithm.

When an island completes its computation, it redistributes the upcoming available threads with the rest of the islands starting by the end. This way we are able to reuse the processors once they're free. The redistribution is implemented using a *ConcurrentLinkedQueue* $< Pair < Integer, Integer > >$, where the island sends a message to create a new thread to another active island containing $< islandId, newAmountThreads >$.

Every island runs the regular GA approach to compute each individual.



5.1 Experimental Setup

The results have been gathered by running each approach previously presented 30 times, with a population of 1.000 for 500 generations. An individual, expression tree, was able to achieve $2^{10} + 1$ nodes when created and up to $2^{15} + 1$ nodes after crossing with another individual. For first 250 generations the data set has been split in half and each part was tested intercalated.

FALTA COMPARACAO LINEAR COM ADAPTATIVA !!!!

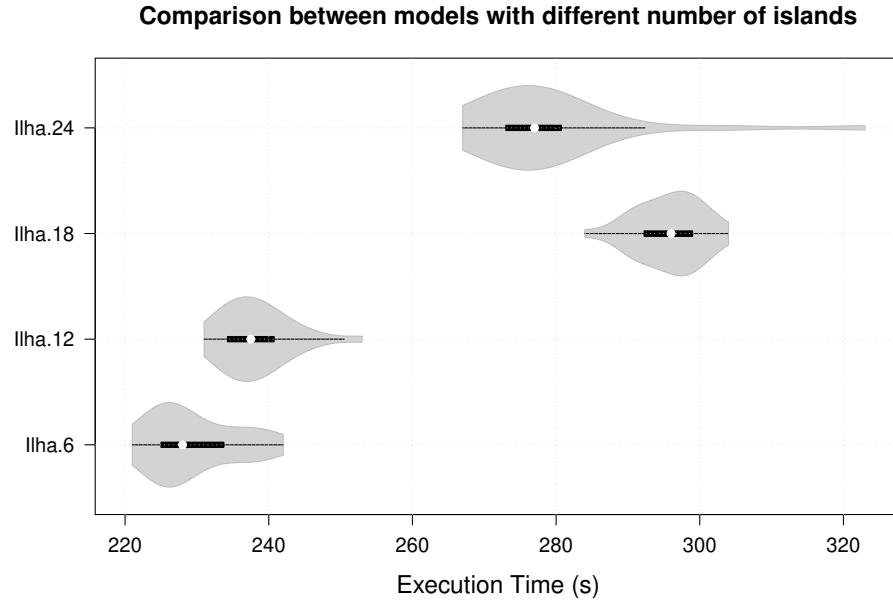


Fig. 2: Comparison of the performance of the islands approach with different number of islands .

The Figure 2 demonstrates the execution time when using different amount of islands.

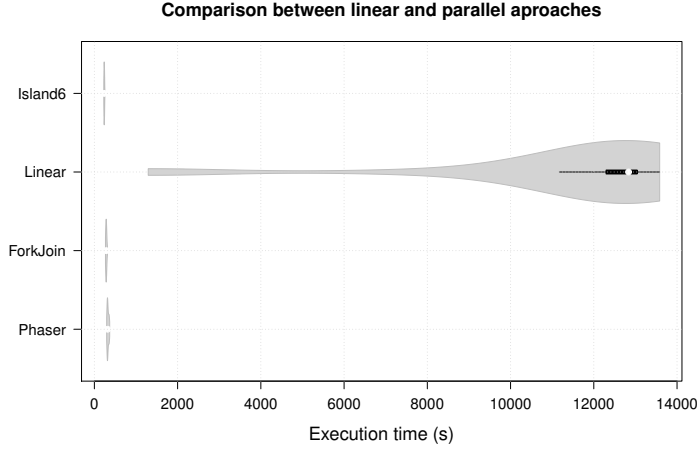
We're able to see that there is a significant difference in execution time between the islands 6,12 and islands 18,24.

Since the islands 6 and 12 are multiple of 24, they can respectively have 4 and 2 threads for inner parallelization. Both of these islands will supposedly finish at the same time, and therefore there won't be a need of redistributing the Threads.

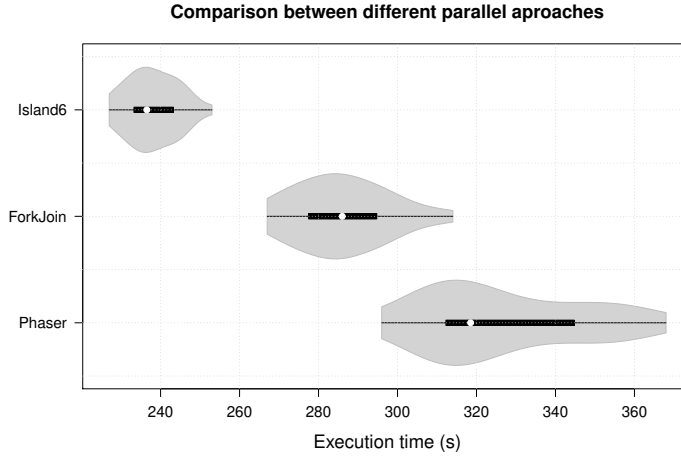
On the other hand the island 18, as it is not a multiple of 24, there are going to be islands with 2 threads, and islands with only 1 thread. The execution time will significantly be slower since the islands with only 1 thread represent a bottleneck in our system. However, because of the thread redistribution we are able to amortise the execution time allowing it to achieve a performance similar to the island 24 approach.

The island 24 features has an unexpected performance. Has the population is splitted evenly between the 24 islands it was expected to behave similarly to other island approaches with multiples of 24, however this is not the case. Our assumption relies on the Parallel MergeSort implemented. Each island is going to create and execute tasks to perform the sorting operation. The thread pool executor, when executed, creates N amount of threads corresponding to the amount of available processes. By having N amount of islands running, and each one of them creates a pool of 24 threads to compute the sorting algorithm,

by the end we'll have $N * N$ threads. The CPU will have to schedule all of these threads, which implies an issue in terms of performance.



(a) Execution time comparison of linear implementation versus parallel implementations



(b) Execution time comparison of the parallel implementations

Fig. 3: Comparison in execution time of the different implemented approaches.

The Figure 3 (a) represents the difference between the linear approach and the speedup obtained from the parallel approaches. The linear approach required up to 50x the time from the best parallel approach implemented to accomplish the computation.

On the Figure Figure 3 (b) we have a comparison of the execution time between the best island approach and the ForkJoin and Phaser approach.

Through the data gathered we're able to see that the island approach takes on average less than 80 seconds than the Phaser approach. If we recall the Phaser implementation, it contains synchronization points, for example, before and after sorting the population. By having 24 inner threads, in every generation, every thread is required to wait for others in order to advance. This situation has an impact in performance, which is clearly visible on the Phaser violion plot. On the other hand, has the island6 only contains 4 inner threads, the amount of waiting required for each other is far less than waiting for 24 inner threads.

The island approach in comparison with the ForkJoin approach demonstrates a difference on average of 40 seconds probably due to the costs of creating tasks every generation. Even though these costs are low, when multiplied by the population size and amount of generations, it may have an impact on execution time later on.

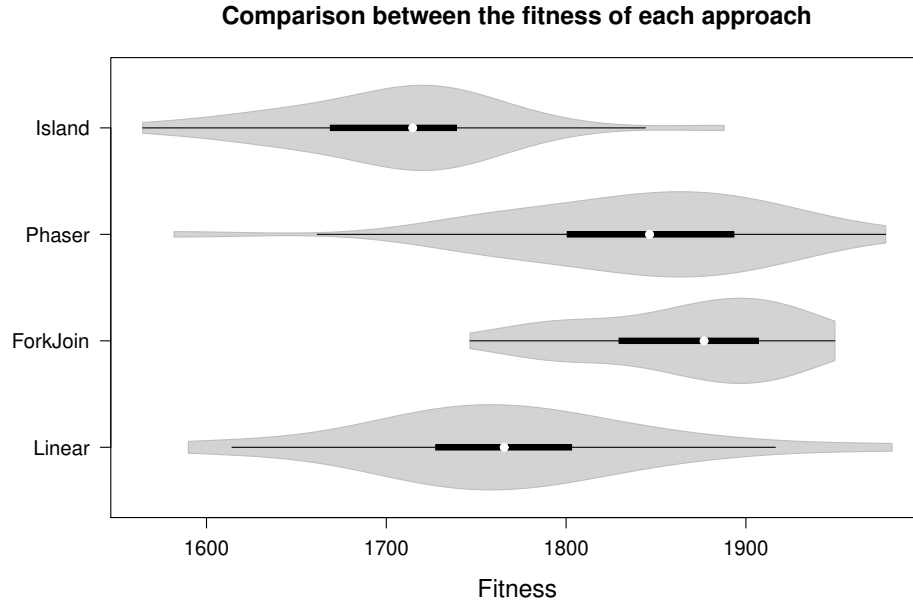


Fig. 4: Comparison of the fitness achieved for each approach.

The Figure 4 addresses the differences between the approaches implemented.

Due to the nature of separability and independence of individuals between islands [14] we can point the small improvement of fitness in the Island approach. The remaining parallel approaches have obtained a fitness higher than the linear approach. This may have happened due to the random nature of the generated mathematical expressions.

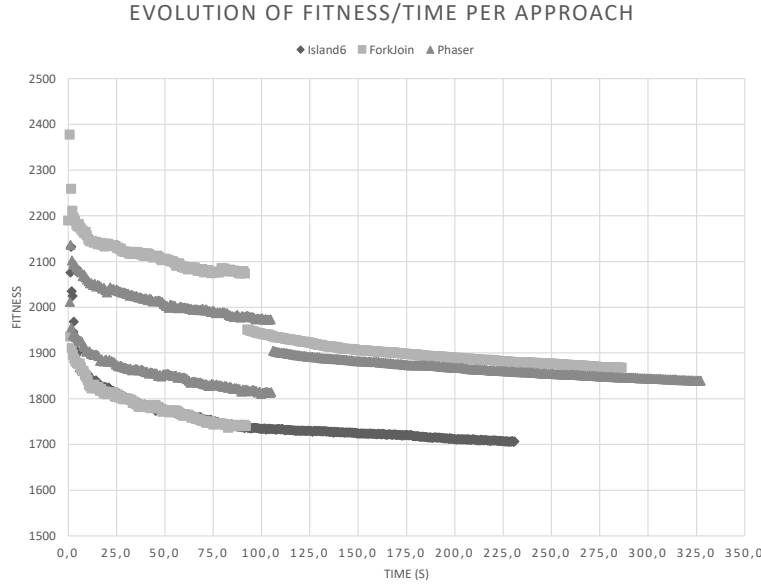


Fig. 5: Comparison of fitness evolution through time on the parallel approaches

The Figure 5 above shows how the fitness values improve over time for each approach.

During the first 250 generations the training set is splitted in half and tested in alternate. Therefore a mathematical expression which was the best individual last generation may not be as good in the current one influencing the overall fitness by arising it. In the figure, the ForkJoin and Phaser approaches both have 2 lines in the beginning which represent the data split. The island approach didn't have much difference between generations and therefore it only contains one line.

Through the results we are able to evaluate that the island approach achieves a better fitness earlier than the other parallel approaches.

5.3 Discussion

Here you should discuss the results on a high level. For instance, based on our results, the parallelization of the merge-sort is relevant as no other parallel work occurs at the same time, and the complexity $O(N \log(N))$ can have a large impact when the number of individuals is high.

6 Related Work

Several implementations of the genetic algorithm were made throughout the years. We will shortly talk about two implementations and how they fit in the scope of our work.

Dominic and Willis [1] developed a MATLAB toolbox, GPTIPS, which is able to perform regression through genetic programming. The main difference between their approach and ours is that they do not explore the parallelism of the algorithm, focusing on the usability of the toolbox. Moreover, they chose to include nonlinear operators, that we decided to leave out. Our work is, therefore, important to whom intends to develop a fast approach of the genetic classifier. Additionally, GPTIPS requires the purchase of a payed software (MATLAB), available to a less broader population.

Jenetics [15] is another genetic programming implementation. It is a Java library designed to abstract different concepts within the genetic programming panorama, such as Gene, Genotype and Chromosome, allowing it to serve a vast spectrum of domains. This library implements the Java Stream Interface and provides ForkJoin Parallelization. This is, therefore, a generic purpose implementation for genetic algorithm. On the other hand, in our work we provided a study of genetic programming specific for regression, where other parallelization techniques were able to achieve better results than ForkJoin, like the Island Models.

7 Conclusions

Here you should resume the major conclusions taken from discussion. Ideally, these should align with the objectives introduced in the introduction.

You should also list the future work, i. e., tasks and challenges that were outside your scope, but are relevant.

Acknowledgements

First Author wrote the part of the program implemented the phasers. Second Author implemented the MergeSort in parallel.

Both authors wrote this paper, with First Author focusing on the introduction, related work and conclusions while the Second Author focused on approach and evaluation.

Each author spent around 30 hours on this project.

References

1. Dominic, P., Leahy, D., Willis, M.: Gptips:an open source genetic programming toolbox for multigene symbolic regression. *Lecture Notes in Engineering and Computer Science* **2180** (12 2010)

2. Espejo, P.G., Ventura, S., Herrera, F.: A survey on the application of genetic programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **40**(2), 121–144 (March 2010). <https://doi.org/10.1109/TSMCC.2009.2033566>
3. Etemadi, H., Rostamy, A.A.A., Dehkordi, H.F.: A genetic programming model for bankruptcy prediction: Empirical evidence from iran. *Expert Systems with Applications* **36**(2), 3199–3207 (2009)
4. George, A., Rajakumar, B.R., Binu, D.: Genetic algorithm based airlines booking terminal open/close decision system. In: *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*. pp. 174–179. ICACCI '12, ACM, New York, NY, USA (2012). <https://doi.org/10.1145/2345396.2345426>, <http://doi.acm.org/10.1145/2345396.2345426>
5. Grefenstette, J., Gopal, R., Rosmaita, B., Van Gucht, D.: Genetic algorithms for the traveling salesman problem. In: *Proceedings of the first International Conference on Genetic Algorithms and their Applications*. pp. 160–168 (1985)
6. Krawiec, K., Moraglio, A., Hu, T., Etaner-Uyar, A., Hu, B.: Genetic Programming: 16th European Conference, EuroGP 2013, Vienna, Austria, April 3–5, 2013, *Proceedings. Lecture Notes in Computer Science*, Springer Berlin Heidelberg (2013), <https://books.google.pt/books?id=6Y-5BQAAQBAJ>
7. Langdon, W.B., Qureshi, A.: Genetic programming-computers using” natural selection” to generate programs. In: *WC1E 6BT*. Citeseer (1995)
8. Lin, W.Y., Lee, W.Y., Hong, T.P.: Adapting crossover and mutation rates in genetic algorithms. *J. Inf. Sci. Eng.* **19**, 889–903 (09 2003)
9. Manwade, K.: Analysis of parallel merge sort algorithm. *International Journal of Computer Applications* **1** (02 2010). <https://doi.org/10.5120/401-597>
10. Martins, B.M.M.: Gacuda-a genetic algorithm framework over gpus (2016)
11. Rawlings, J.O., Pantula, S.G., Dickey, D.A.: *Applied regression analysis: a research tool*. Springer Science & Business Media (2001)
12. Sivanandam, S., Deepa, S.: Genetic algorithm optimization problems. In: *Introduction to Genetic Algorithms*, pp. 165–209. Springer (2008)
13. Whitley, D.: A genetic algorithm tutorial. *Statistics and computing* **4**(2), 65–85 (1994)
14. Whitley, D., Rana, S., Heckendorn, R.: The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology* **7** (12 1998)
15. Wilhelmstötter, F.: *Jenetics, Library user’s manual 4.3* (2018), <https://books.google.pt/books?id=6Y-5BQAAQBAJ>