Homework 1: Implementation of a genetic algorithm using the Hill Climbing heuristic

1st Mihai Bojescu
Master in Artificial Intelligence and Optimisation
Faculty of Computer Science of University "Alexandru Ioan Cuza" of Iași
Iași, Romania
bojescu.mihai@gmail.com

***Abstract*—This document aims to detail the implementation of binary genetic algorithms, hill climbing algorithms and how the former could be hybridised using the latter.**

***Index Terms*—genetic algorithm, hill climber algorithm, nature inspired methods, hybridisation**

I. Introduction

This document aims to detail how binary genetic algorithms and hill climbing algorithms are implemented, and a method on how they could be hybridised in order to potentially achieve better results. The former represents a "weak" method that can be used to solve a pleothera of problems. The latter represents an effective algorithm of the local-search class. In this document, the hybrid algorithm is an extension of the binary genetic algorithm in which hillclimbing is used after n steps.

II. Hill Climbing algorithm

A. Definition

Hill climbing is a mathematical optimisation technique belonging to the family of local search. The algorithm is iterative and starts with an arbitrary solution to a given problem, which attempts to perfect by performing incremental changes.

The algorithm may regrettably become ensnared within local optima, precluding its ability to escape from such configurations. Nevertheless, it continues to find utility across diverse problem domains, owing to its computational efficiency and swift derivation of an acceptably effective solution.

B. Notations

Step:

$$Step \in \mathbb{Q}, Step \neq 0 \tag{1}$$

Acceleration:

$$Acceleration \in \mathbb{Q}, Acceleration \neq 0 \tag{2}$$

Precision:

$$Precision \in \mathbb{Q}, Precision \neq 0 \tag{3}$$

Iterations:

$$Iterations \in \mathbb{N}, Iterations \in [-1, +\inf) \tag{4}$$

Function to optimise:

$$f : \mathbb{R} \to \mathbb{R}, \tag{5}$$

C. High-level implementation

As a high-level view, the algorithm performs the following steps:

```
1) Picks an $x_{best}$ at random.
2) Set $iteration = 0$.
3) Set $score_{before} = NULL$.
4) Set $step_{best} = step$.
5) Set $score_{best} = f(x_{best})$.
6) While $iteration < iterations \vee score_{before} = NULL \vee |score_{best} - score_{before}| > precision$:
   a) For each step $step_i \in \{$
      $step_{best} * acceleration$,
      $step_{best} * -acceleration$,
      $step_{best} * 1/acceleration$,
      $step_{best} * -1/acceleration$
      $\}$:
      i) If $f(x_{best} + step_i) > score_{best}$, perform $step_{best} = step_i$, $score_{best} = f(x_{best} + step_i)$, $x_{best} = x_{best} + step_i$.
   b) Peform $iteration = iteration + 1$.
7) Return $x_{best}$.
```

D. Time complexity

The algorithm has a low time complexity of $\mathcal{O}(nlogn)$, thus it is usually implemented on systems that would need an aproximate solution, them often being realtime systems.

E. Continuous space hill climbing

Since the optima of a problem might not be found using a value x that are modified using discreet value, we will have to resume to modifying the value x in a continuous manner.

In order to perform the above, we will continuously use a parameter named $acceleration$, parameter which will improve x in smaller increments for each iteration. This way, the algorithm can find the optima of a given problem with a higher precision than a discreet space hill climber.

In order to perform this, we will call the f function on each $x * step_i$, where:

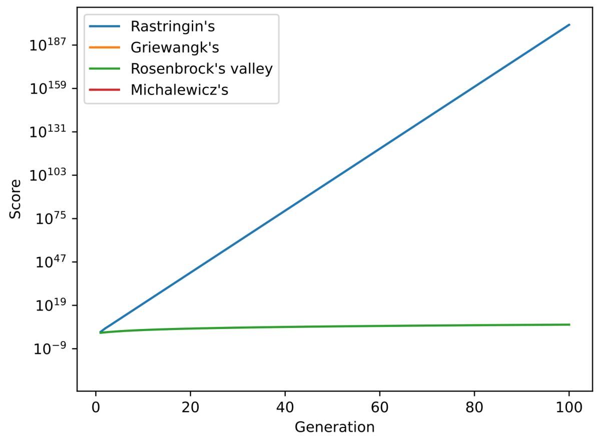
$$step_i \in \{ \quad step_{best} * acceleration, \quad step_{best} * -acceleration, \quad step_{best} * 1/acceleration, \quad step_{best} * -1/acceleration \quad \} \tag{6}$$

F. Algorithm Demonstrations and Graphical Analysis

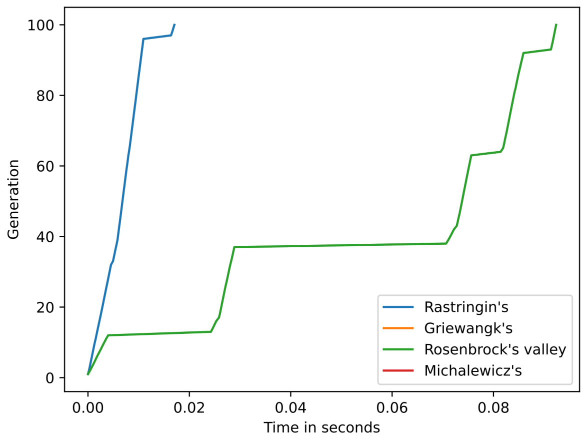
In this section the hillclimber algorithm will be applied on multiple functions:

1) Rastringin's function
2) Griewangk's function
3) Reosenbrock's valley function
4) Michalewicz's function

Given $dimension(x) = 2$:



With the following runtimes:



III. Genetic algorithms

A. Definition

A genetic algorithm represents a way to solve problems from the combinatorial, numerical, optimisation, timetabling and scheduling, robotics etc. spaces, in general being very weakly coupled to problems, thus can be used in multiple problem spaces with rather little modifications.

B. Notations

Individual:

$$Individual = (genes, value_{fitness}) \tag{7}$$

$$genes = \{x | x \in \{0, 1\}\} \tag{8}$$

$$value_{fitness} \in \mathbb{Q} \tag{9}$$

$$Individual_{decode} = (decode(genes), value_{fitness}) \tag{10}$$

Population:

$$Population = \{individual_1, individual_2, \dots\} \tag{11}$$

$$Population_{decoded} = \{ \quad individual_{1_{decoded}}, \quad individual_{2_{decoded}}, \quad \dots \quad \} \tag{12}$$

Population:

$$Population = \{individual_1, individual_2, \dots\} \tag{13}$$

Number of individuals in a population:

$$|Population| = n, n \in \mathbb{N}, n \geq 1 \tag{14}$$

Number of generations:

$$generation \in \mathbb{N} \tag{15}$$

Encode individual to binary function:

$$encode : x \to \{v | v \in \{0, 1\}\} \tag{16}$$

Decode individual from binary function:

$$decode : \{v | v \in \{0, 1\}\} \to x \tag{17}$$

Criteria function:

$$criteria : (\quad Population_{decoded}, \quad Population_{fitness}, \quad generation \quad) \to \{True, False\} \tag{18}$$

Fitness function:

$$fitness : individual_{decoded} \to \mathbb{Q} \tag{19}$$

Selection function:

$$select : (\; individual_{decoded},\; individual_{decoded} \;) \rightarrow (\; individual_{decoded},\; individual_{decoded} \;) \quad (20)$$

Crossover points:

$$bits_{crossover} = \{bit_{crossover} | bit_{crossover} \in \mathbb{N}\} \quad (21)$$

Mutation function:

$$mutate : individual \rightarrow individual \quad (22)$$

Mutation chance:

$$mutation \in \mathbb{Q}, mutation \in [0, 1] \quad (23)$$

C. *High-level implementation*

As a high-level view, the algorithm performs the following steps:

1) Generate a population of n binary-encoded individuals randomly.
2) While the criteria function $criteria$ does not return $True$, be the criteria number of generations, "fitness" values of the population or something else:
   a) Calculate the "fitness" of each individual in the population using the function $fitness$.
   b) Sort the population by the "fitness" value.
   c) Create a new and empty population set, $Population'$.
   d) For $n/2$ iterations:
      i) Perform the selection of parents for the next individuals using the $selection$ function.
      ii) Crossover the 2 selected parents at the $crossover$ bit, generating 2 children, $child_1$, $child_2$.
      iii) Mutate $child_1$, given the $mutate$ function and the $mutation$ chance using an internal mutation function.
      iv) Mutate $child_2$, given the $mutate$ function and the $mutation$ chance.
      v) Add $child_1$ and $child_2$ to $Population'$
   e) Perform $Population = Population'$
3) Return the most "fit" individual in the population, represented by the individual on which the $fitness$ function returned the highest score.

D. *Time complexity*

The time complexity of genetic algorithms can be difficult to determine as it depends on many factors such as the size of the population, the number of generations and the specifics of the algorithm. In general, this would be considered $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$.

E. *Fitness function*

The fitness function represents the function we want to optimise for. Due to the selection process of the genetic algorithm returning the 2 most "fit" individuals, and using them as seeds for the children of the next generation, this can be infered.

F. *Selection function*

The selection function aims to select 2 of the most "fit" individuals from the population. This function is provided by the user of the genetic algorithm in order to improve the flexibility of the algorithm.

G. *Crossover function*

The algorithm includes a crossover function. The crossover takes the 2 given individuals $parent_1$, $parent_2$ and performs the following:

1) For each $bit_{crossover} \in bits_{crossover}$:
   a) For child $child_1$, the $bit_i < bit_{crossover}$ bits from the parent $parent_1$'s genes and $bit_i \geq bit_{crossover}$ bits from the $parent_2$'s genes are borrowed.
   b) For child $child_2$, the $bit_i < bit_{crossover}$ bits from the parent $parent_2$'s genes and $bit_i \geq bit_{crossover}$ bits from the $parent_1$'s genes are borrowed.
   c) $parent_1, parent_2 = parent_2, parent_1$

For a parent $parent_1$ with $parent_1.genes = 01010101$, a parent $parent_2$ with $parent_2.genes = 10101010$, and crossover bits $bits_{crossover} = \{3\}$, the results would be:

1) $child_1.genes = 01101010$
2) $child_2.genes = 10010101$

H. *Mutation function*

The mutation function aims to increase the possibility of a greater solution and the chance of escaping the local optima of the final solution. In this function, each bit bit_i of the individual's genes is flipped when a probability $P(mutate) > mutation$ and the individual is returned.

Having this performed, the new individual might have a lower or higher "fitness" than the rest of the population, "fitness" which will be captured by the $fitness$ and later "judged" by the $select$ functions.

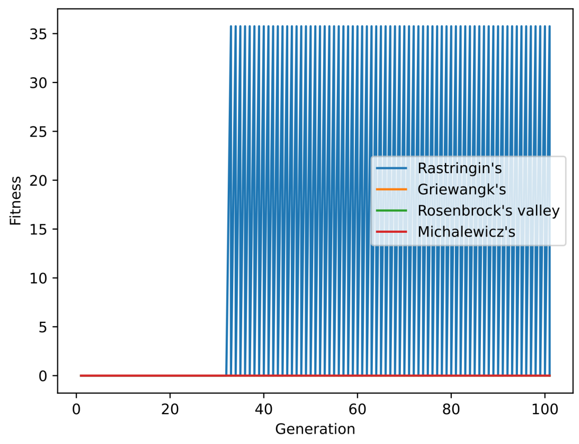
In case the individual performs lower due to the change, the individual is dropped. The other way around is also possible: in case the individual performs higher (possibly even significantly so), the individual is promoted by the $select$ function.

I. *Generations*

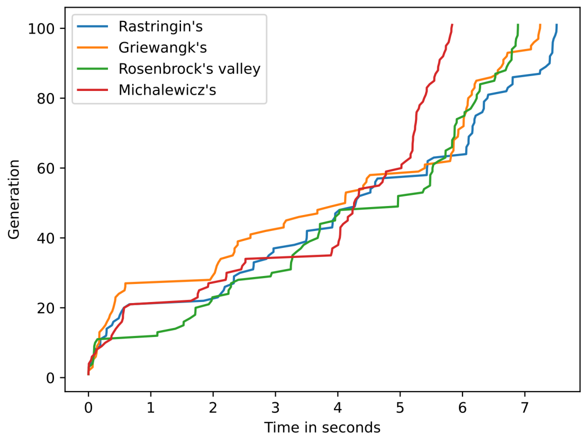
The algorithm does not aim to provide the best solution first-try, thus it needs time for it to improve. Given enough time (expressed in generations), it can provide better solutions, as the "fitness" of the individual increases.

J. *Algorithm Demonstrations and Graphical Analysis*

Idem to the preceeding section, when $dimension(x) = 2$, the algorithm offers the following results in logarithmic scale:



With the following runtimes:



IV. Hybridisation

A. *Motivation*

While the 2 algorithms can be used independently of each other and can in turn produce good results, they can also be merged into 1 single algorithm that performs both functions at the same time in the hopes that the solutions are found faster.

In general the Hybrid algorithm inherits from the binary genetic algorithm, while using the hillclimbing algorithm n times.

The algrorithm *might* deceive the user into thinking it is simple in complexity, but having to rely on both the binary genetic algorithm *and* the hillclimber algorithm in the same function calls brings higher complexity than the two algorithms taken without other dependencies.

B. *High-level implementation*

As a high-level view, the algorithm performs the following steps:

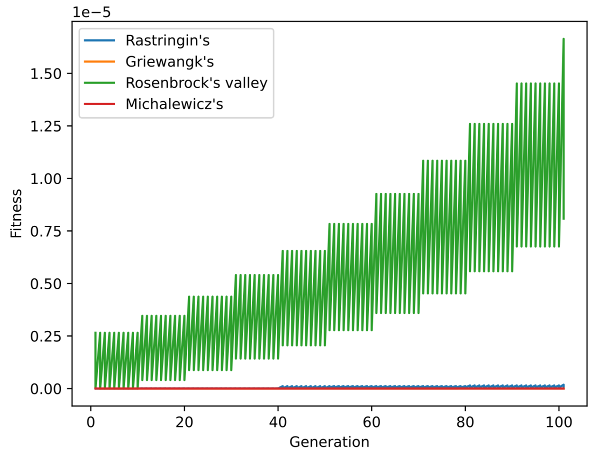
1) While $generations < generations_{limit}$:
   a) Run a step of the binary genetic algorithm, returning $Population_{decoded}$
   b) If the hillclimber algorithm should be run:
      i) For each $individual_{decoded} \in Population_{decoded}$:
         A) Run the hill climbing algorithm on the individual $individual_{decoded}$, returning $individual'_{decoded}$
         B) Run $individual_{decoded} = individual'_{decoded}$

C. *Time complexity*

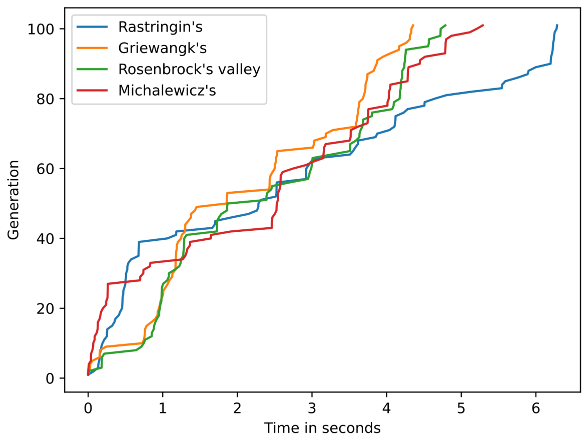
While the algorithm appears to have a low complexity at first, this would have the complexity $\mathcal{O}(n^2 + n)$, as both the genetic and the hill climbing algorithms are run in tandem.

D. *Algorithm Demonstrations and Graphical Analysis*

On the 4 test functions, the algorithm has given the following results when $dimension(x) = 2$:



In the following times:



E. *Testing*

During testing with a function $f : [0, 31] \to \mathbb{R}, f(x) = x^3 - 60 * x^2 + 900 * x + 100$ with an maxima of 10 in the given domain, the algorithm:

1) Did not escape the given boundaries (issue encountered with the genetic algorithm).
2) Did provide a better approximation of the solution (issue encountered with the hillclimber algorithm).

From the observations, the hybrid algorithm provided promising solutions. Surprinsingly, there were instances where it outperformed the binary genetic algorithm in timing.

V. Conclusions

Each presented algorithm has its use-cases. While some *can be* used in real-time systems due to their speed, they are prone to getting stuck in the local optima, and while others are really powerful, they are not fast enough to provide solutions in sensible time. A hybridisation of the two algorithms would partially mitigate the downsides, perhaps with added complexity.

References

[1] Luchian Henri; Eugen Croitoru. “Genetic algorithms (2022)”, Faculty of Computer Science of University “Alexandru Ioan Cuza” of Iași.

[2] Russell, Stuart J.; Norvig, Peter (2003). “Artificial Intelligence: A Modern Approach (2nd ed.)”, Upper Saddle River, New Jersey: Prentice Hall, pp. 111-114, ISBN 0-13-790395-2

[3] Banzhaf, Wolfgang; Nordin, Peter; Keller, Robert; Francone, Frank (1998). “Genetic Programming - An Introduction”. San Francisco, CA: Morgan Kaufmann. ISBN 978-1558605107.