UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Genetic Algorithms for solving combinatorial problems.

DEGREE FINAL WORK

Degree in Computer Engineering

Author: Juan Francisco Ariño Sales

Tutor: Federico Barber Sanchís

Course 2020-2021

Resum

The boolean satisfiability problem is a very important problem in computer science, affecting a great variety of fields, this work attempts to study how well genetic algorithms perform when applied to the SAT problem in comparison with some popular existing SAT solvers.
In order to study the genetic algorithms, the effects of the different genetic operators are analyzed through experimental evaluation, using this analysis the best performing genetic algorithms are selected for comparison with the SAT solvers.
The experiments performed indicate that hybrid genetic algorithms outperform pure genetic algorithms and local search algorithms, such as the SAT solvers analyzed, outperform them both when applied to SAT problems.

Paraules clau: algoritme genètic, satisfacibilidad booleana, operadors genètics, solucionador SAT

Resumen

El problema de satisfacibilidad booleana es un problema muy importante en el campo de la informatica ya que afecta a una gran variedad de campos, este trabajo intenta estudiar el comportamiento de los algoritmos aplicados a problemas SAT para luego compararlos con algunos de los solucionadores SAT existentes más populares.
Para poder estudiar el comportamiento del algoritmo genético, se analizan los efectos de los distintos operadores genéticos a partir de la evaluación experimental, utilizando dicho análisis se seleccionan los mejores algoritmos genéticos para compararlos con los solucionadores SAT.
Los experimentos llevados a cabo indican que los algoritmos genéticos hibridos funcionan mejor que los algoritmos genéticos puros y los algoritmos de busqueda local, como los solucionadores SAT analizados, funcionan mejor que ambos a la hora de solucionar problemas SAT.

Palabras clave: algoritmo genético, satisfacibilidad booleana, operadores genéticos, solucionador SAT

Abstract

El problema de satisfacibilidad booleana és un problema molt important en el camp de la informatica ja que afecta a una gran varietat de camps, este treball intenta estudiar com es comporten el algoritmes genetics aplicats a problemes SAT per a despres compararlos amb alguns dels solucionadors SAT existents més populars.
Per a poder estudiar el comportament de l'algoritme genètic, s'analitzen els efectes dels distints operadors genètics a partir de l'avaluació experimental, utilitzant la dit anàlisi se seleccionen els millors algoritmes genètics per a comparar-los amb els solucionadors SAT.
Els experiments duts a terme indiquen que els algoritmes genètics híbrids funcionen millor que els algoritmes genètics purs i els algoritmes de cerca local, com

els solucionadors SAT analitzats, funcionen millor que ambdós a l'hora de solucionar problemes SAT.

Key words: genetic algorithms, boolean satisfiability, genetic operators, SAT solver

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

The Boolean satisfiability problem (SAT) consists of determining an assignment of variables such that a given Boolean expression evaluates to True, or proving no such assignment can exist; It is a very famous problem in logic and computer science since it was the first problem proven to be NP-Complete, thus finding an efficient general SAT solver is bound to advance the P vs NP problem, which is considered by many to be one of the most important open problems in computer science and is actually one of the unsolved[1] seven millennium problems stated by the Clay Mathematics Institute on May 24, 2000.
This work attempts to study the behavior of genetic algorithms applied to a restricted version of the SAT problem called the 3-SAT problem, which is still NP-complete. It will attempt to do so by analyzing the behavior of genetic algorithms based on different operators and hyper-parameters tested on a series of 3-SAT problems, and then grading them against each other, the best ranking genetic algorithms found will then be compared with some of the most popular open source SAT solvers.

1.1 Objectives

The main objective of this work is to generate a series of genetic algorithms which can solve 3-SAT problem instances, these algorithms will be compared with some existing popular open source SAT solvers, in order to determine if a random search algorithm can compete with the local search used by most solvers.
A secondary objective is to determine the effect the different genetic operators and their parameters have on the evolution of the population, and how this affects both the exploration of the search space and the exploitation of the available solutions/individuals.
The point of the analysis of the different genetic operators and parameters stated as a secondary objective, is to determine the configuration of the genetic algorithms which will be compared with the SAT solvers explained in Section 2.5 in order to carry out the main objective.

1.2 Motivation

The study of the boolean satisfiability problem, and by extension P vs NP, is of great importance to computer scientists and the scientific community at large since any proof relating P and NP is bound to have a great impact on a wide range of fields, from computational theory and mathematics to economics and game theory, philosophy and many others.

[1] The Poincaré Conjecture is the only solved millennium problem

The boolean satisfiability problem also has practical applications on fields such as automatic theorem proving and circuit design, it even has some applications in planning for artificial intelligence.
Genetic algorithms are not usually applied to decision problems such as the SAT problem since they are best suited for optimization problems, nonetheless applying them to a decision problem poses an interesting challenge in converting either the algorithm or the problem.
This work converts the decision problem into an optimization one through the use of the *maxsat* fitness function explained in section 3.2, which allows for the incremental improvement of the fitness function needed by genetic algorithms.
It is also interesting to determine the effects of the different genetic operators on the behavior of the genetic algorithms, since this knowledge can help design better algorithms in the future.
The effect of the parameters is analyzed through the experimental evaluation of the algorithms on selected SAT instances, shown in chapter 5.
Though the problem studied in this work is the boolean satisfiability problem the genetic algorithm designed can be used on a wide variety of problems through minimal changes to the code, the genetic operators are also designed in a modular way so the algorithm can be constructed with any combination of them.

1.3 Outline

This work is organized as follows:

Chapter 2 provides an overview of the background information necessary to understand this work.
Among the required information one can find explanations for the boolean satisfiability problem, P vs Np and the field of computational complexity and finally completeness and hardness, in the sections 2.2, 2.3 and 2.4 respectively.
There is also a basic explanation for how genetic algorithms work in Section 2.1.
The SAT solvers which will be compared with the genetic algorithms are explained in Chapter 2, Section 2.5
Chapter 3 lays out a more in-depth description of how genetic algorithms work, along with detailed descriptions of the genetic operators and the different algorithms which can be used to carry out each operation.
Chapter 4 explains the specific details of the design of the *genetic_algorithm* class and the functionality of the developed implementation in Section 4.1, the database structure used to log the results of the experiments is shown in Section 4.2 and an overview of how the actual experiments are run can be found in Section 4.3.
In chapter 5 the effect of the different genetic operators on the *success rate* of the algorithm is analyzed through experimentation, the best combinations found are run on a harder subset of problems, these executions are analyzed again in order to select the genetic algorithm/s which will be compared with the SAT solvers explained in section 2.5.
Chapter 6 describes the final conclusions of this work.

CHAPTER 2
Preliminaries

2.1 Genetic Algorithms

A genetic algorithm is a meta-heuristic designed to mimic the behavior of natural selection, it does so by evolving a population of candidate solutions through the use of biologically inspired operators. First proposed by John Holland in 1975 (**10.5555/531075**), genetic algorithms are based on the process of natural selection, they simulate it, albeit in a pretty simplified manner, through the use of the different genetic operators:

- The *selection* operator simulates the selection process which occurs in nature in which fitter individuals have more probability of surviving and passing on their genes to the next generation.
- The *crossover* operator mimics the genetic information exchange which occurs when two creatures breed.
- The *mutation* operator simulates the "randomness" of the genetic information exchange process, when exchanging genetic information the resulting genetic string is not always a copy of the parents since random mutations have a probability of occurring.
- The *population replacement* operator simulates the life and death cycle in nature, where old individuals die off and get replaced by the young individuals.

Initially a population of random candidate solutions is generated, then the population is ranked using a fitness function which measures how well each individual in the population solves the problem; Once every individual has a fitness value associated, the individuals which will be used to generate the next population are selected based on said value and grouped into pairs, these pairs are called *parents* and the process of choosing them is called *selection*.
With the parents now selected begins the *crossover* process, for every parent pair the parent genes are split and recombined in order to form two new individuals, these new individuals are the *children* which will form the new generation.
Before the children are added to the new population they must undergo the *mutation* process, where there is a chance that some genes inside the individual's genetic code might change; After every individual has been mutated the old population is replaced, either partially or totally, by a new population made of the mutated children through a process called *replacement*; The new population is then ranked and the processes of selection, mutation and replacement are run again, this continues until a solution has been found or the maximum number of iterations for the algorithm is reached.

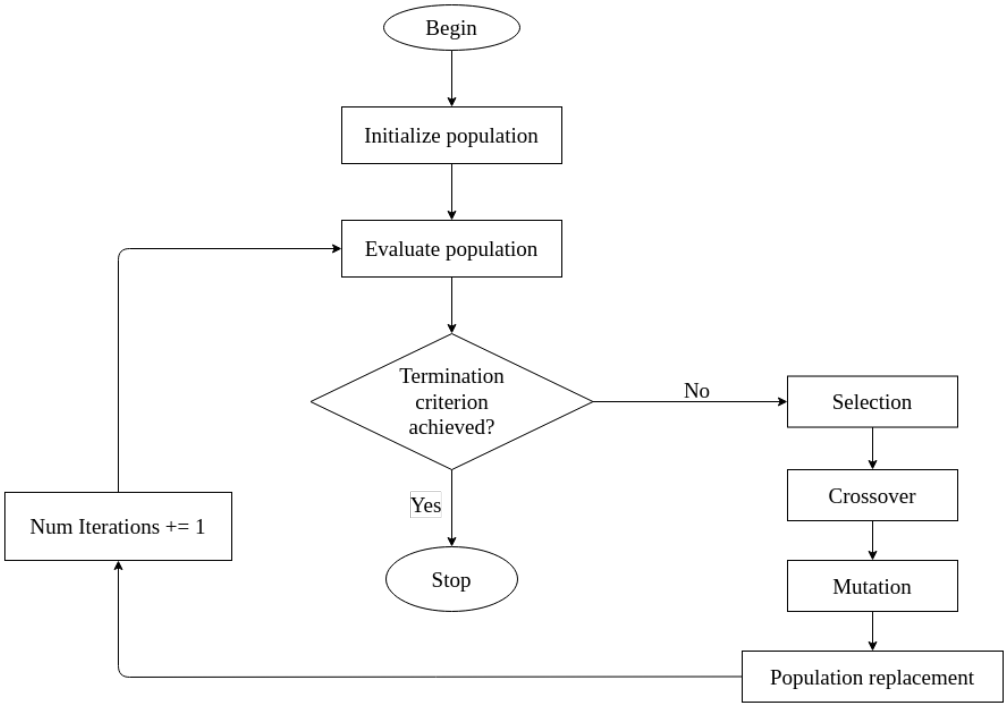

Figure 2.1: Genetic Algorithm diagram, own work based on (**genetic_algorithm**)

A genetic algorithm is a population based random search algorithm (**Cochran2011**) since the initial population is randomly generated, but it differs from other random search algorithms due to the fact that a genetic algorithm performs a guided random search through the use of the biological operators mentioned above; These biological operators mimic the processes of evolution and natural selection, albeit in a pretty simplified manner.

2.2 3-SAT Problem

The boolean satisfiability problem is the problem of determining, for a given boolean expression, if there exists an assignment of values, such that the formula evaluates to True, or proving no such assignment exists and therefore the formula will always evaluate to False. For example, having variables $\{x_1, x_2, x_3, x_4\}$ and the boolean expression:

$$(x_1) \wedge (x_1 \vee \overline{x_2}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee \overline{x_3} \vee x_4)$$

It is trivial to find an assignment of variables such that the given expression evaluates to True (eg. $\{True, True, False, False\}$), consequently this expression is said to be *satisfiable*, but for this other boolean expression:

$$(\overline{x_1}) \wedge (x_1 \vee \overline{x_2}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee \overline{x_3} \vee x_4)$$

There exists no such assignment, the equation will never be solved by tweaking the variable values, therefore it is an *unsatisfiable* expression.
Any algorithm designed to solve SAT instances must distinguish between satisfiable and unsatisfiable problems however, barring some trivial or contradictory expressions, this can only be done by attempting to solve the actual problems, since no one can assert that no solution exists if they haven't searched the whole solution space.
Due to the inherently random nature of Genetic Algorithms one can not be sure the whole

solution space has been searched and, in consequence, won't be able to assert with confidence that no solution exists, only that no solution has been found in the given amount of time, this is the main reason why this work will focus only on SAT instances which are known to be solvable.
Since genetic algorithms do not systematically try all solutions for a given problem, when attempting unsolvable 3-SAT instances the genetic algorithm would never stop, or it would only do so after its maximum number of iterations is reached, after which one can not prove that no solution exists, only that no solution has been found by the algorithm in the specified amount of time.
All the boolean SAT expressions used throughout this work are in conjunctive normal form or CNF, this means that all formulas are a conjunction ($\wedge$) of clauses (or a single clause), these clauses contain the variables and the disjunction ($\vee$) operator, they can also contain the negation ($\neg$) operator; Both of the boolean formulas shown above are in conjunctive normal form.
3-SAT is a restricted version of SAT where each clause has exactly three variables in it, the problem of determining the satisfiability of a 3-SAT formula in conjunctive normal form is also NP-Complete (**Karp2010**)

2.3 P vs NP

Computational complexity theory is a branch of mathematics and theoretical computer science, which tries to classify different mathematical problems according to the computational resources needed to solve them, usually the measured resources are time and storage space, though other complexity measures can also be used.
The analysis of complexity is done using a mathematical model of computation, this model allows us to analyze the performance of algorithms inside a theoretical computation machine, which means we can compare different algorithms without worrying about the details of their specific implementations.
The most common model of computation used is called a Turing machine (**turingm**), it is a useful model of computation when trying to study an algorithm's performance on real machines, since anything that can be calculated using a traditional computer can also be computed using a Turing machine.
Different types of Turing machines can be used to define different complexity classes, the most commonly used are *Deterministic Turing machines* which follow instructions sequentially, much like traditional computers, and *Non-deterministic Turing machines* which behaves stochastically, since at any given time step it can have more than one instruction and will choose which one to execute randomly.

Complexity classes are defined by establishing upper bounds to the resources available for the aforementioned Turing machines; For example the complexity class P is composed of all the problems which can be solved by a deterministic Turing machine in polynomial time, P is a class with a time constraint, the amount of space used is irrelevant for this categorization. There exist four fundamental classes based on the resources constrained:

- $DTIME[t(n)]$ is composed of all the problems which can be solved by a deterministic Turing machine in $t(n)$ amount of time
- $NTIME[t(n)]$ is composed of all the problems which can be solved by a non-deterministic Turing machine in $t(n)$ amount of time
- $DSPACE[s(n)]$ is composed of all the problems which can be solved by a deterministic Turing machine in $s(n)$ amount of space

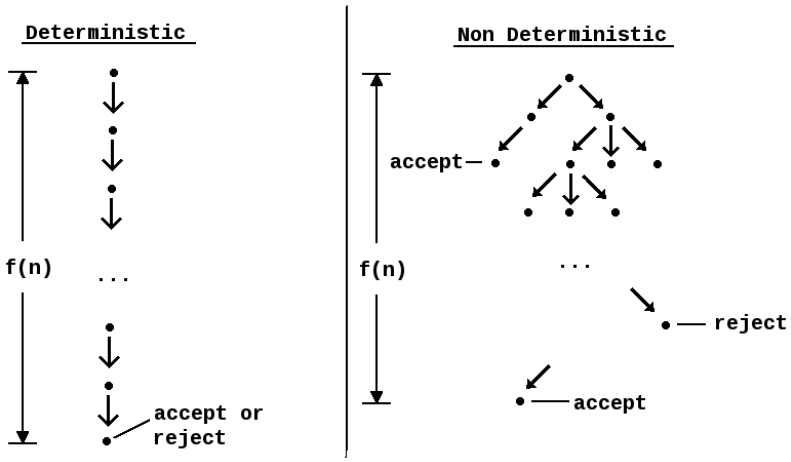

Figure 2.2: Left: Deterministic Turing machine computation tree. Right: Non-deterministic Turing machine computation tree. (**computation_trees**)

– $NSPACE[s(n)]$ is composed of all the problems which can be solved by a non-deterministic Turing machine in $s(n)$ amount of space

Using these fundamental classes the complexity class P can be defined as the union of all problems in $DTIME[t(n)]$ where t(n) is polynomial time, formally:

$$P = DTIME[n^{O(1)}] = \bigcup_{k \in \mathbb{N}} DTIME[n^k]$$

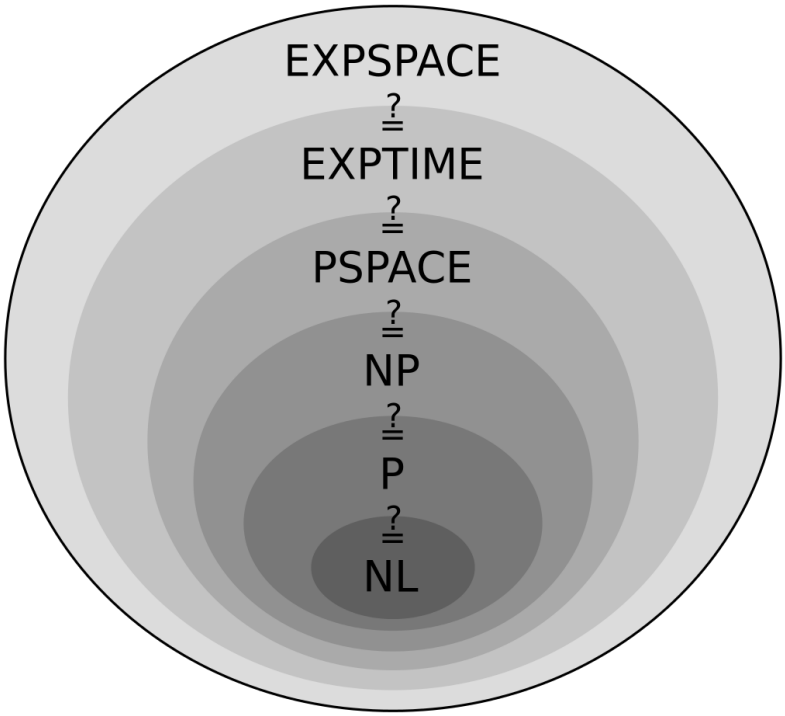

Figure 2.3: Relations between some of the canonical complexity classes (**complexity_classes**)

The boundaries between classes shown in figure 2.3 are not hard boundaries, which means all classes could theoretically be the same class, though there exists proof that at least one of the boundaries must not be equal.
This leads us to one of the main unsolved problems in computational theory, is **P** equal

to **NP**?, in other words, can all problems which can be verified in polynomial time by a deterministic Turing machine also be solved by that same deterministic machine in polynomial time?
This work attempts to study the application of genetic algorithms towards the search of this question, it does so by attempting to solve the 3-SAT problem, which is NP-Hard, through the use of these methods.
Due to the stochastic nature of the algorithms they cannot be used to determine any proof for unsolvable cases of the problem (explained in subsection 2.2), in spite of this, their behavior when applied to solvable instances, may yield insight into how stochastic processes can help improve local exploration methods.

2.4 Completeness and Hardness

Inside the different complexity classes there exists a certain type of problems which are said to be *Hard*, a problem P is considered hard if all other problems in the complexity class can be reduced to P using a polynomial time reduction; If problem A can be transformed into an instance of problem B in polynomial time, then problem A is polynomial time reducible to B.
A hard problem is at least as difficult to solve as the hardest problems in the complexity class, this means if there exists a solver for the hard problem, it can be used to demonstrate that all other problems in the complexity class can also be solved in the same amount of time, since they can be reduced to the hard problem in polynomial time. A problem does not need to be in a complexity class, for it to be considered hard in relation to that same complexity class.
If a problem is hard in relation to the same complexity class it is found in, that problem is considered *Complete*, the complete problems represent the hardest problems in that complexity class.

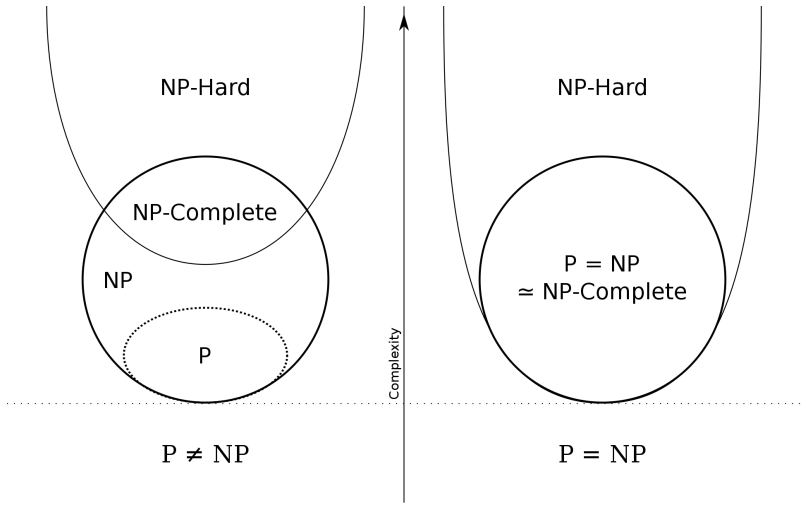

Figure 2.4: NP-Hard and NP-Complete boundaries (**complete_hard**)

The boolean satisfiability problem was the first problem proven to be NP-Complete (**Cook1971**), therefore finding a polynomial time algorithm which solves the SAT problem is akin to proving $P = NP$, the reverse is also true, if one can prove that no polynomial time algorithm for the SAT problem can exist, then $P! = NP$ is also proven.

2.5 SAT Solvers

Most of the modern SAT solvers, as evidenced by the contestants of the SAT-RACE[1] through the years (**Een2000**; **Gebser2007**; **Hamadi2009**; **Biere2017**), are based on two algorithms, the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm and the *conflict-driven clause learning* (CDCL) algorithm.

2.5.1. DPLL

The DPLL algorithm works by selecting a variable and splitting the problem into two simplified ones where said variable is assigned *True* and *False* respectively, then it chooses another variable and keeps splitting the problem recursively until a solution is found or all options have been checked (**Davis1962**).
The algorithm also uses some optimizations at each step in order to reduce the effective search space, these optimizations are the same as the ones explained in the *trivial case*, *remove unit variables* and *remove pure variables* sections which appear later in this work.

2.5.2. CDCL

The CDCL algorithm works by first selecting a variable and assigning the value *True* or *False* to it, this assignment is "remembered" in the *decision state*, then it applies *boolean constraint propagation* (BCP) to the resulting clauses, which analyzes the unsatisfied clauses in search for clauses where two of the three variables are evaluated to *False*, it assigns a value to the other variable such that it evaluates to *True*, the resulting clauses are then analyzed again, this procedure continues until no more assignments can be made.
After applying BCP it constructs an implication graph in order to find any conflicts in the variable assignments, if any conflict is found a new clause which negates the assignments that led to said conflict is derived and added to the clauses, then the algorithm backtracks to the *decision state* where the conflicting variable value is assigned.
If no conflicts are found in the implication graph the algorithm selects another variable and runs again, this process continues until all the variables have been assigned a value.

The SAT solvers shown below will be compared with the best performing genetic algorithms found throughout this work.

2.5.3. MiniSat

MiniSat is an extensible SAT solver first developed in 2003 by Niklas Eén and Niklas Sörensson at Chalmers University of Technology, Sweden (**Een2000**), the version used throughout this work is MiniSat v2.2.0, the second public release of MiniSat 2 which won the SAT-Race in 2006.
MiniSat is based on the CDCL algorithm though it improves the basic version by applying the techniques of watched literals and dynamic variable ordering.

– *Watched literals* improves the efficiency of the constraint propagator, reducing the amount of work that must be done during backtracking, this results in less time needed to check the satisfiability of the formulas (**Gent2006**).

[1] The SAT Race or SAT Competition is a yearly competition for SAT solvers organized by the *International Conference on Theory and Applications of Satisfiability Testing*

– *Dynamic variable ordering* alters the order in which the variables are instanced by tree-search algorithms, a good ordering is known to improve the efficiency of the search algorithms (**Bacchus1995**).

2.5.4. ManySat

ManySat is a portfolio-based SAT solver, first developed in 2008 by Y. Hamadi, S. Jabbour, and L. Sais it won best parallel SAT solver in the SAT-Race 2008 and silver and bronze medals in the SAT-Race 2010.
Portfolio-based SAT solvers run various algorithms in parallel, and stop as soon as any one of them reaches an answer, this allows them to exploit the multiprocessing capabilities of modern processors by running various algorithms concurrently, which helps offset the weaknesses of each specific algorithm thereby making the portfolio-based algorithm more robust.
ManySat uses variations of the DPLL and CDCL algorithms in its portfolio, each of the variations uses a specific set of parameters such that the generated strategies are orthogonal yet complementary (**Hamadi2009**).

2.5.5. zChaff

zChaff is an implementation of the *Chaff* algorithm, originally developed by Dr. Lintao Zhang it is currently being maintained by Yogesh Mahajan.
Chaff is a variation of the DPLL algorithm designed by researchers at Princeton University (**Moskewicz2001**), it improves the algorithm by using the *watched literals* technique and a decision heuristic called *Variable State Independent Decaying Sum* which changes the variable selection process, in addition to some other enhancements such as conflict clause deletion and early restarts.

2.5.6. Clasp

Clasp is a conflict-driven answer set solver developed in 2007 by Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub at the Institut für Informatik in Universität Potsdam, it is part of Potassco, the Potsdam Answer Set Solving Collection (**Gebser2007**).
Clasp and its variants won gold and silver medals during the SAT Races of 2009, 2011 and 2013, in all cases the medals were won in the crafted instances category.
Clasp's main algorithm works through conflict-driven nogood learning, it generates a set of *nogoods*, which are variables whose value has been set using the *Clark completion* procedure, it also creates an initial positive atom-body-dependency graph which is used to detect conflicts, then it applies the *unit clause rule* to said nogoods set, along with some decision heuristics, the process of generating nogoods and applying the unit clause rule is repeated until a conflict is found, in which case it backtracks to the nogood assignment which caused the conflict through the atom-body-dependency graph, or until no more assignments can be made.

2.5.7. Lingeling, Treengeling and Plingeling

Lingeling is a SAT solver developed in 2010 by Armin Biere at the Institute for Formal Models and Verification in Johannes Kepler University, Linz, Austria; Along with its variants Treengeling, Plingeling and others, it won gold, silver and bronze medals during the

SAT Races of 2010, 2011, 2013, 2014, 2016, 2017, 2018 and 2020, most of them in the parallel algorithms track with the variants Plingeling and Treengeling.
Lingeling uses a standard CDCL algorithm with various pre-processing techniques interleaved, these include SCC decomposition for extracting equivalences, failed literal probing, which assigns a value to a variable then propagates it, if it fails the opposite value is assigned, and lazy hyper binary resolution, which combines multiple resolution steps for the boolean formula into one (**Biere2010**).
Plingeling is a multi-threaded version of Lingeling, where each thread runs a separate SAT solver instance, it is thus a portfolio-based parallel SAT solver.
Treengeling is also a parallel solver which uses the same structure as Plingeling for parallel processing, the main differences between the two algorithms are the way the different SAT solvers are instanced in each thread and the information that is shared between said threads (**Biere2012**).

2.5.8. CaDiCaL

Cadical is also developed by Armin Biere at the Institute for Formal Models and Verification in Johannes Kepler University, Linz, Austria, it was first presented during the SAT Race 2017 (**Biere2017**), year in which it won gold medal in the category of Satisfiable+Unsatisfiable problems; Cadical also won another gold medal during the SAT Race 2018 in the Unsatisfiable problems category.
Cadical's main search loop is based on the CDCL algorithm but it expands on it by applying formula simplification methods at each time step, this procedure is called *in-processing* (**Jarvisalo2012**), the three in-processing methods used by Cadical are:

- *probing*: failed literal probing learns the binary clauses through hyper binary resolution, these clauses are later used to remove equivalent variables and duplicated binary clauses.
- *subsumption*: subsumption checks are used to remove the subsumed learned clauses, subsumed clauses are clauses which contain the same information, or a subset of it, than another clause, therefore they can be safely discarded without affecting the original formula.
- *variable elimination*: (bounded) variable elimination is used along with subsumption checks to eliminate variables from the clauses, or substitute them by an equivalent definition in order to simplify the formula.

Though this list covers many of the winners of the SAT race through the years, it does not include them all due to the time constraints and limited resources allowed for the development of this work.
Still, the list attempts to include a series of solvers based on different algorithms and different implementations/optimizations of the same algorithms in order to compare the genetic algorithm with the widest range of behaviors possible.
The analysis of the SAT solvers is also useful for the implementation of the genetic algorithm, since some of the local optimizations used by the solvers can be implemented in said algorithm.
These optimizations are used to reduce the solution space which must be searched, effectively turning the genetic algorithm into a hybrid genetic algorithm, their design and implementation for a genetic algorithm are shown in sections 3.8 and 2 respectively.

CHAPTER 3
Genetic Algorithms

This chapter and the subsequent ones offer a more in depth explanation of the actual design and implementation of the genetic algorithms and their operators used throughout this work, along with an experimental analysis of said implementations and posterior comparison with the SAT solvers explained in section 2.5.

3.1 Codification of individuals

The first step when designing a genetic algorithm is choosing an adequate method of representation for the solution domain, this means choosing a way of representing the candidate solutions to the problem which allows for the application of the biological operators needed.
Usually this is achieved by encoding the solution as a fixed size list of values, also called bit string representation (**Gottlieb**), though other encodings such as variable sized lists or tree-like representations may also be used, fixed size list representations are easier to work with since the genetic material is easier to split and align which allows for a simpler crossover operator, this makes it one of the most common representations for genetic algorithms (**Loviskova2015**; **Harmeling2000**; **Bhattacharjee2017**; **Gottlieb**).
The representation chosen for the SAT solutions is a binary encoding, each individual's genotype[1] is a single fixed size list where the values are binary digits, these digits are named *genes* and they represent the *True* (1) or *False* (0) values of the variables in the SAT instance, their position in the genotype list indicates which variable the value is assigned to. For the following boolean expression:

$$(x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee \overline{x_3})$$

A solution can be represented as a list of size 4 where each of the elements is either *True*, encoded as a 1 or *False*, encoded as a 0; Using this representation the solution S becomes S':

$$S = \{x_1 = True, x_2 = False, x_3 = False, x_4 = True\}$$
$$S' = \{1, 0, 0, 1\}$$

3.2 Fitness Function

Once the genotype of the individuals has been established, the next step when designing a genetic algorithm is choosing the fitness function which will be used to rank the

[1] The genotype is the complete set of genes of an individual

individuals based on how well they solve the problem; The individual's fitness value is called *phenotype* since it is considered to be the "physical" expression of the individual's genotype.
The fitness function chosen is based on the maximum satisfiability problem which tries to find the maximum number of clauses which can be solved in a boolean expression; The fitness value for each individual is equal to the number of clauses it solves for the given boolean expression. Formally:

$$f_{MAXSAT}(x) = c_1(x) + ... + c_m(x)$$

where $c_i(x)$ represents the truth value of the ith clause.
f_{MAXSAT} is one of the most common fitness functions used by genetic algorithms attempting to solve SAT problems (**Gottlieb**; **Marchiori1999**; **folino**).

3.3 Initial Population

The initial population is formed by randomly generating individuals until the established population size N is reached, the initial population size depends on the difficulty or size of the problem attempted, the population should be big enough to allow for the exploration of the whole solution space, an increase in population size carries with it an increase in the computational cost of each iteration therefore a balance must be found in order to solve the problems in an adequate amount of time.
For each individual a list of I random integers is selected from the discrete uniform distribution in the interval $[0, 1]$ using numpy's *randint* function[2] where I is equal to the number of variables in the boolean expression to be solved.
If any of the equation's variables has already been set through the problem-specific optimizations, the gene representing that variable is also set to the same value, once each individual's genome has been established they are added to the population list.

3.4 Selection

Selection is the process by which the individuals are chosen from the population in order to become the parents of the new generation, special care must be taken when selecting the parent pairs so as to maintain sufficient genetic diversity in the resulting children population and prevent premature convergence.
If too much emphasis is put on the fitness value, individuals with a high value can quickly dominate the population by filling it with copies of themselves, thereby reducing the effective search space of the algorithm by forcing it to focus only on the most effective solutions found up to that moment.
While this effect can be desirable once the algorithm is reaching the end of its running time, it is best to avoid it in the initial stages and instead try to maintain a high genetic diversity.
There exist many different selection algorithms each with their own characteristics and computational cost, some of them are explained below and will be used throughout this work as tunable parameters of the designed genetic algorithm.

[2] https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html

3.4.1. Random Selection

The individuals are randomly selected from the population independent of their fitness value, once an individual has been selected as a parent it is removed from the population so it cannot be chosen again.
The probability of each individual being selected is defined as:

$$P(i) = \frac{1}{N}$$

Where $P(i)$ is the probability of selecting the individual i and N is the total number of individuals in the population

3.4.2. Roulette Selection

Roulette selection assigns a normalized value to each individual equal to the individual's fitness value divided by the sum of all fitness values in the population, each individual is assigned a range between 0 and 1 the size of which depends on its normalized value, a random real number between 0 and 1 is chosen and the individual upon whose range it falls is selected as a parent, then the process begins anew.
This selection process is akin to assigning a slice of a wheel to each individual, with fitter individuals getting bigger slices, then spinning the wheel like a roulette and wherever the ball lands it chooses that individual.
The selection probability of each individual is proportionate to its fitness, formally:

$$P(i) = \frac{f(i)}{\sum_{j=1}^{N} f(j)}$$

where $f(i)$ is the fitness value for the individual i.

3.4.3. Rank Selection

Rank selection sorts the individuals in the population by fitness value and ranks them from best N to worst 1, then it assigns a fitness value to each individual equal to its rank divided by the total rank, finally another selection method must be used with the new fitness values, usually roulette selection.
The probability of selecting each individual can be defined as:

$$P(i) = \frac{rank(i)}{N * (N - 1))}$$

where $rank(i)$ is the rank assigned to the individual i.

3.4.4. Tournament Selection

In tournament selection, K individuals are randomly selected from the population, those individuals are compared amongst each other and the one with the highest fitness value is selected to become a parent; The process is repeated until enough parents are selected to form the next generation. The probability of each individual being selected as a parent is:

$$P(i) = \begin{cases} \frac{f(i)}{\sum_{j=1}^{k} f(j)} \; if \; i \in [1, n-k-1] \\ 0 \; if \; i \in [n-k, n] \end{cases}$$

where k is the tournament size and n is the number of times the tournament is repeated (**Jebari2013**).

3.4.5. Truncation Selection

Truncation selection sorts the individuals by their fitness value from best to worst, then selects the top $X\%$ to become the parents for the new generation, the children are all formed from combinations of those chosen individuals, self-mating is not allowed.

3.4.6. Stochastic Universal Sampling

In stochastic universal sampling first the total fitness value is calculated, this value is equal to the sum of all fitness values in the population, then said fitness value is divided by the total number of parents needed to create the next generation (np) resulting in the *distance* between points.
A random integer is chosen between 0 and *distance*, then $np - 1$ more points are chosen by adding the *distance* to the previous point starting from the random point.
Once all the points have been selected, the individuals are mapped to a line such that the size of the line segment for each individual is proportional to its fitness value, wherever the chosen points fall along that line, those individuals are selected to become the parents of the next generation.

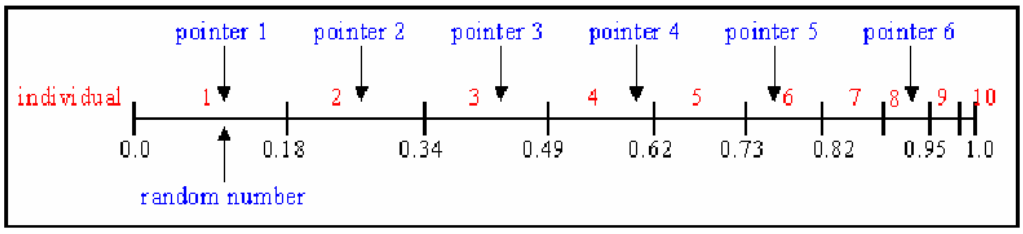

Figure 3.1: Stochastic Universal Sampling, individuals $1, 2, 3, 4, 6, 8$ are selected. (**PenchevaT2009**)

Stochastic universal sampling is very similar to roulette selection though it fixes one of its pitfalls by not allowing individuals with a very high fitness to dominate the process, this gives weaker members of the population a chance to be chosen.

3.4.7. Annealed Selection

Annealed selection (**Jyotishree**) is a blend of roulette wheel selection and rank selection, which tries to fix the shortcomings of both these methods by shifting from exploration (with rank selection) in the early stages of the algorithm gradually towards exploitation (with roulette wheel selection) as the algorithm reaches the maximum number of iterations allowed.
Annealed selection works by computing both the rank fitness value and the roulette fitness value for each individual, each value is multiplied by a factor which determines the importance of said value, these factors depend on the current generation value and are responsible for the gradual shift from exploration to exploration.
Formally:

$$f_a(i) = f_{rank}(i) * ra + f_{wheel}(i) * rb$$

where $f_a(i)$ is the final fitness value for individual i, $f_{rank}(i)$ is its rank fitness value and $f_{wheel}(i)$ is its roulette wheel fitness value.
ra is a factor which starts at 1 and decreases by $\frac{1}{N_{gen}}$ each generation, while rb starts at 0 and increases by $\frac{1}{N_{gen}}$ each generation (N_{gen} is the maximum number of iterations allowed). The probability for selecting an individual i as a parent is:

$$P_x(i) = \frac{f_a(i)}{\sum_{i=1}^{N} f_a(i)}$$

3.5 Crossover

Crossover in genetic algorithms is the process by which the genotype of the selected parents is recombined in order to generate new individuals, this process is analogous to the recombination of genetic material with occurs naturally during sexual reproduction in biology.
The purpose of the crossover operator is to attempt to generate novel offspring, theoretically with a higher fitness value than their parents though this isn't always achieved in practice, in order to explore the solution space through recombination of the existing solutions/individuals.
All the crossover operators used throughout this work operate on a bit-string representation of the genetic material of each individual, though other different genetic operators or variations of the ones shown in this paper can also be used to operate on different genetic representations.

3.5.1. Single-point Crossover

In single point crossover a natural number between 0 and L is randomly chosen, L is equal to the length of the genotype for the individuals, each parents gene string is split at the point chosen and the resulting halves are combined amongst each other with the resulting children each having some of their gene string coming from one parent and the rest from the other.

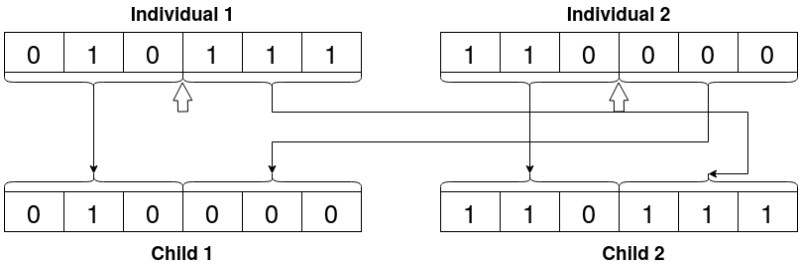

Figure 3.2: Single point crossover example with cutting point equal to 3

3.5.2. Two-point Crossover

Two point crossover is very similar in behavior to single point crossover but instead of choosing a single split point two of them are chosen, the children each are a copy of one

parents with the substring of their gene string which appears between the two chosen points interchanged with the other children/parent.

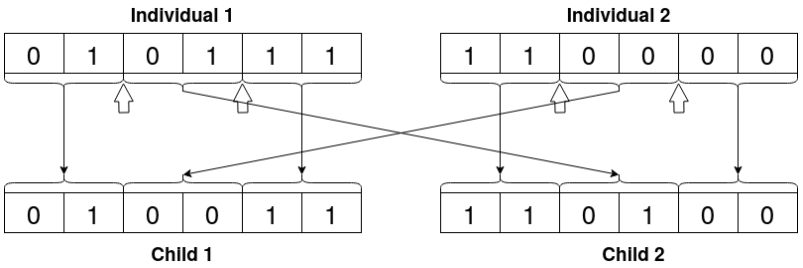

Figure 3.3: Two point crossover example with cutting points equal to 2 and 4

3.5.3. Sliding Window Crossover

Sliding window crossover starts with a fixed window size W established by the user, with the sliding window starting at point 0 up to point $W - 1$ the parents exchange the genetic material inside the window, the corresponding children are saved in a temporary list, the sliding window then moves one cell over and the parents interchange the genetic material in the new window forming two more children, once the sliding window reaches the end ($L - W - 1$) and all the different children have been added to the temporary list, the 2 fittest children with differing genotypes are selected, the rest are discarded.

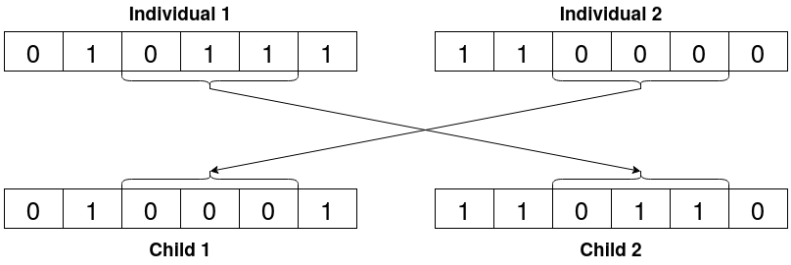

Figure 3.4: Sliding window crossover example with window length equal to 3

3.5.4. Random Map Crossover

Random map crossover first generates an array of size L formed natural numbers between 0 and 1, if the random map has a 0 in position i, then the parents interchange the gene at said position, if the random map instead has a 1 at position i the parents maintain the same gene.

3.5.5. Uniform Crossover

In uniform crossover genes are interchanged between parents uniformly. For each child half of their genetic material will come from one parent and half from the other in an alternating fashion; For example for child 1 the first gene will be from parent 1 the second from parent 2 the third from parent 1 again and so on, while the second child starts with the first gene from parent 2, the second from parent 1, etc...

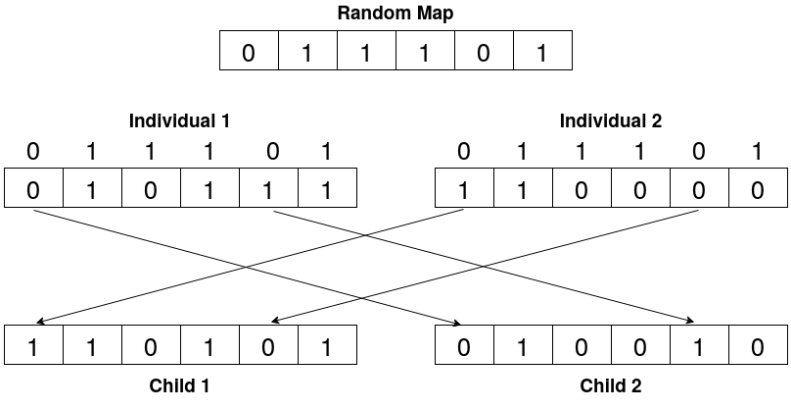

Figure 3.5: Random map crossover example with map $0, 1, 1, 1, 0, 1$

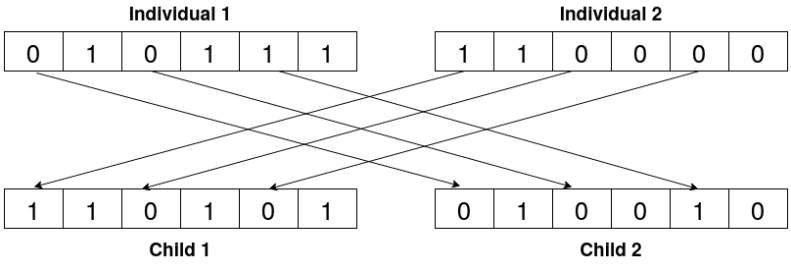

Figure 3.6: Uniform crossover example

3.6 Mutation

Mutation is the process by which the genotype of individuals generated through crossover is changed in order to introduce more genetic diversity in the population.
Maintaining genetic diversity is essential for the exploration of the solution space, if enough genetic diversity is lost the algorithm will focus too much on the dominating solutions and the exploration will get stuck in the local optimal solutions, in other words, the algorithm will focus too much on the exploitation of the available solutions while neglecting the exploration of the solution space.
The probability of the mutation process occurring in an individuals genotype is given by the parameter p_m, this value must not be set too high or the search of the solution space will degenerate to a random search, it also must not be set too low or the mutation process will barely have any effects on the population.
Since the representation used throughout this work is a bit-string representation, the mutation process can be done by flipping one or more genes inside the individuals genotype; Some of the different techniques used to achieve this are explained below.

3.6.1. Single Bit Flip

In single bit flip a random number between 0 and $L - 1$ is chosen, L is equal to the length of the gene string of the individuals, once the random number is chosen the bit indexed at that location in the gene string is flipped, if its value is 1 it is changed to 0 and vice-versa.

3.6.2. Multiple Bit Flip

Multiple bit flip is very similar to single bit flip with the main difference being the number of bits flipped in the gene string; In multiple bit flip a random number of bits between 0 and $L - 1$ is chosen to be flipped instead of only one.

3.6.3. Single Bit Greedy

Single bit greedy flips one bit, if the fitness of the new individual is higher than that of the original the new individual is returned, if its not higher then the bit is flipped back to its original state and the next bit in the sequence is flipped, this process continues until one of the individuals generated has higher fitness than the original or the last bit is reached with none being better, in which case the original individual is returned.

3.6.4. Single Bit Max Greedy

Single bit max greedy is a variant of single bit greedy, it works very similarly to single bit greedy but instead of returning the first individual better than the original, it stores all of the "improved" individuals in a list and once the algorithm reaches the end of the gene string, it returns the one with the highest fitness.

3.6.5. Multiple Bit Greedy

Multiple bit greedy also works very similarly to single bit greedy when it comes to flipping bits, but when an individual with better fitness than the original is generated it is not returned, instead the process continues flipping bits on the new improved individual starting by the bit right to the one just flipped. The process continues until the end of the gene string is reached, then the improved individual is returned, if no improved individual exists the original individual is returned.

3.6.6. FlipGA

FlipGA works exactly like multiple bit greedy, but when the end of the gene string is reached instead of returning it, the algorithm is run again using the new gene string as its input. The algorithm is run as many times as necessary until no further improvements can be obtained, once the algorithm is run and none of the new individuals generated are an improvement over the "original" individual, FlipGA stops and returns said individual.

3.7 Population Replacement

Once the array of individuals generated through crossover has undergone the mutation process they have to be added into the population, this presents a challenge, since the total size of the population must remain the same some individuals from the old population have to be selected for removal which can lead to the loss of some good solutions.
This loss of good solutions can be alleviated to some extent through the use of *Elitism* where a number of the best solutions in the population are added to the next generation in order to ensure they don't get replaced, the number of solutions saved is given as a percentage of the population using the parameter E_p. The two basic types of replacement strategies are *Generational* and *Steady-State* replacement:

– *Generational Replacement*: The whole population is replaced by the new individuals at each time step, unless elitism is active, then the top $X\%$ best solutions from the old population also carry over to the new one.

– *Steady-State Replacement*: Only a small fraction of the population (2 individuals) is replaced at each time step, elitism has no effect on steady-state replacement.

Generational replacement is faster at exploring the solution space than steady-state replacement but it can quickly lead to the loss of good solutions if elitism is not used and steps to ensure the new population generated is an improvement over the old one are not taken.
In steady-state replacement new individuals are immediately added to the population, which enables them to be selected as parents for the next children, while this greatly reduces the cost of each iteration it also means it is slower at exploring the solution space since only 0 to 2 new individuals can be generated at each time step, this can also lead to premature convergence if the algorithm focuses too much in the fittest individuals.
The population replacement techniques analyzed throughout this work are explained below, all the methods explained below can work as both generational replacement functions and as steady-state replacement functions, unless explicitly stated otherwise.

3.7.1. Random Replacement

In random replacement for each new individual added to the population, a random individual from the old population is removed. This method does not obtain great results in practice since it takes no care to not remove good individuals, instead choosing them randomly independent of their fitness.

3.7.2. Mu-Lambda Replacement

Mu-Lambda replacement (**Jyotishree2012a**) is derived from generational replacement but instead of replacing the old population with the new individuals, they are both mixed into a temporary population and ranked according to their fitness, then the best S individuals are selected as the new population, S is equal to the population length established in the GA's parameters.
A variant of mu-lambda replacement has also been implemented, in this variant the number of children generated through the selection, crossover and mutation operators is $S * 2$ and instead of mixing the old population with the children to rank them, it ranks only the children and selects the best $S - E_p * S$ individuals to form the new population, along with the individuals selected through elitism.

3.7.3. Parent Replacement

In parent replacement the parents get replaced by their offspring, therefore each individual is only allowed to breed once, while this helps in introducing genetic diversity to the population in can also lead to the loss of good solutions if a parent with a high fitness is replaced by its child with a lower fitness value.
This method is exclusively a steady-state replacement function and cannot be used for generational replacement.

3.7.4. Weak Parent Replacement

Weak parent replacement is derived from parent replacement with the main change being that instead of replacing the parents with their offspring, the four of them are added to a temporary list and ranked according to their fitness value, then the best two individuals are selected to be added back to the general population.
This method is exclusively a steady-state replacement function and cannot be used for generational replacement.

3.7.5. Delete Replacement

In delete replacement I individuals are chosen from the old population using the same selection function used during the selection process, then those individuals are deleted and I individuals from the children array are chosen in the same way, but instead of deleting them they are added to the population and the replacement process ends.
This algorithm reduces the convergence speed of the genetic algorithm which helps in avoiding premature convergence where the algorithm gets stuck in local optima.

3.8 Problem Specific Optimizations

Before running the genetic algorithm on the 3-SAT problem instance, some local search optimizations can be applied to the problem instance in order to try and reduce the size of the solution space which must be searched (**Bhattacharjee2017**), by applying local optimizations in combination with the genetic operators it becomes a hybrid genetic algorithm.

3.8.1. Trivial Case

If the problem instance contains only negated variables or only positive variables then we can safely assign the value 0 or 1 to all of them and the boolean equation will evaluate to *True*.
For example, in the boolean expression:

$$(x_1) \wedge (x_1 \vee x_2) \wedge (x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3 \vee x_4)$$

contains only positive variables therefore by setting them all to true $\{x_1, x_2, x_3, x_4\} = \{1, 1, 1, 1\}$ the expression is satisfied.

3.8.2. Remove Unit Variables

If the boolean expression contains *unit clauses*, these are clauses with a single variable in them, the value of the variable can be set to 0 or 1 accordingly, since the expression must be satisfied in order to satisfy the whole boolean expression, with the variable set, all clauses in which said variable is *True* can be removed from the expression and the negated form of the variable can also be removed from the rest of the clauses.
For example, in the boolean expression:

$$(x_1) \wedge (x_1 \vee \overline{x_2}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3} \vee x_4)$$

the first clause is a unit clause, we can safely assign a value to the variable it contains ($x_1 = 1$), then we can remove the clauses where it appears in its positive form and remove its negated form from the clauses where it appears, this results in the following boolean expression:

$$(x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee x_4)$$

3.8.3. Remove Pure Variables

If a variable appears only in negated form or only in positive form in the whole boolean expression, then said variable can be safely set to 0 (if negated) or 1 and all clauses which contain the variable can be removed from the expression.
For example, in the boolean expression:

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_4) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee x_4)$$

x_1 only appears in positive form in the expression, therefore we can set $x_1 = 1$ and remove all clauses where it appears, the resulting simplified boolean expression is:

$$(x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee x_4)$$

3.9 Conclusions

This chapter explains the characteristics and behavior of some of the most commonly used operators in genetic algorithms and expands on the information about how the algorithms work given in section 2.1.
The design of the implementation for the explained genetic operators is shown in chapter 4, while the actual code can be found in listing 2.
Chapter 5 contains the experimental evaluation of the operators along with the posterior analysis performed on said experiments; The final combination/s of genetic operators used in the algorithm/s is also selected and compared with the SAT solvers in the same chapter.
Although the amount of genetic operators analyzed is extensive, it is by no means a complete list of all the available operators, it would be convenient to expand the list of genetic operators analyzed in order to include other operators such as boltzmann selection (**Lee2003**), reduced surrogate crossover (**crossover_ops**) or partial shuffle mutation (**Abdoun2012**).

CHAPTER 4
Genetic Algorithm Design

4.1 The genetic algorithm class

The genetic algorithm is designed as a modular genetic algorithm where the selection, crossover, mutation and population replacement functions are all set as parameters when the genetic algorithm is initialized, along with all other parameters which said functions might need.
The genetic algorithm is designed to work with both generational and steady state replacement strategies, this can be set through a parameter during the initialization phase.
The parameters which need to be set before running the genetic algorithm are:

- *filename*: The name of the file containing the 3-SAT problem instance.
- *max_iterations*: The maximum number of iterations allowed.
- *pop_size*: Length of the population array.
- *elitism*: The number of best individuals which will carry over from one generation to the next.
- *steady_state_replacement*: Is a boolean which when set to True enables the steady state replacement strategy and disables the generational.
- *save_to_db*: Is a boolean which controls data logging to the database.
- *plot_distributions*: Is a boolean which controls if the distributions are logged when the algorithm is over.

After initializing the algorithm, before actually running it, some parameters have to be set through the function *set_params*, this function sets the selection, crossover, mutation and population replacement functions which will be used by the algorithm and their respective parameters.
The arguments of the *set_params* function are:

- *selection_func*: Sets the selection function.
- *crossover_func*: Sets the crossover function.
- *mutation_func*: Sets the mutation function.
- *replacement_func*: Sets the population replacement function.

– *mutation_rate*: Is the frequency with which the genes in the genetic string mutate, expressed as a real number between 0 and 1, both included.

– *truncation_proportion*: Is the number of best individuals which will be selected from the population to breed by the truncation selection method, expressed as a fraction of the population size.

– *tournament_size*: Is the number of individuals which will compete in each tournament in the tournament selection method.

– *crossover_window_len*: Is the length of the sliding window used by the sliding window crossover method, expressed as a fraction of the genetic string length.

– *num_individuals*: Is the number of individuals which will be selected for deletion in order to be replaced by the selected children in the delete replacement method, expressed as a fraction of the population size.

Once the genetic algorithm has been initialized and the parameters have been set, an initial population of random individuals is generated, then all the individuals are evaluated using the fitness function and stored in a tuple array alongside their fitness value.
The array containing the individuals and their fitness values is used as the *population* input in the selection functions.

Selection Function	**Parameters**
Random	population, num. parents
Roulette	population, num. parents
Roulette W/ Elimination	population, num. parents
Rank	population, num. parents
Truncation	population, num. parents, *truncation_proportion*
Tournament	population, num. parents, *tournament_size*
Stochastic Universal Sampling	population, num. parents
Annealed	population, num. parents, max. iterations, current iteration

Table 4.1: Selection function parameters

– *population* is the array where the population formed by all the individuals in the current generation is stored.

– *num. parents* is the number of parents which will be selected to breed, 2 if steady state replacement is used, and $PopulationLength - Elitism$ if generational replacement is used.

– *max iterations* is the maximum number of iterations allowed in the genetic algorithm.

The selection functions all return a list of parent tuples, where each tuple contains two individuals that have been selected as parents, these parent tuples are used as the input for all the crossover functions, where the genetic material of the two parents will be exchanged to form the children.

Crossover Function	Parameters
Single point	parent tuple
Two points	parent tuple
Sliding window	parent tuple, *crossover_window_len*
Random map	parent tuple
Uniform	parent tuple

Table 4.2: Crossover function parameters

The crossover functions receive one parent tuple as input and output one children tuple, then all the children tuples generated are put into a children array and fed into a function which navigates the children array one by one and feeds the corresponding genetic strings into the mutation function.

Mutation Function	Parameters
Single bit	gene string, *mutation_rate*
Multiple bit	gene string, *mutation_rate*
Single bit greedy	gene string, *mutation_rate*
Single bit max greedy	gene string, *mutation_rate*
Multiple bit greedy	gene string, *mutation_rate*
FlipGA	gene string, *mutation_rate*

Table 4.3: Mutation function parameters

All mutation functions receive one genetic string as their input and output another genetic string, they are also all dependent on the *mutation_rate* parameter.
The resulting modified genetic strings are all stored inside an array called *mchildren* which will be used as one of the inputs for the population replacement functions.

Population Replacement	Replacement mode	Parameters
Generational	generational	*elitism*, mchildren, population
Mu Lambda	generational, steady state	mchildren, population, population size
Mu Lambda Offspring	generational, steady state	mchildren, population size
Delete	generational, steady state	mchildren, population, *num_individuals*, selection method
Random	generational, steady state	mchildren, population
Parents	steady state	mchildren, parent tuple, population
Weak Parents	steady state	mchildren, parent tuple, population

Table 4.4: Population replacement function parameters

– *mchildren* is the array of mutated children genetic strings.

– *selection method* is the selection function which will be used to choose the individuals.

– *parent tuple* refers to the pair of individuals which were used to generate the children pair in *mchildren*.

The population replacement functions return an array (or set) of individuals which become the new population, the *current iteration* value is increased by one and the algorithm begins again by selecting parents from the new population.

4.2 Data logging and processing

The genetic algorithm is connected to a database where the information about the execution of said algorithm on a 3SAT problem instance can be logged; Logging the results to the database can be controlled by setting the boolean parameter *save_to_db* during the initialization of the genetic algorithm, if set to *True* the results will be logged.
The table *ga_run* stores the results of the execution, the genetic operators used by the algorithm (selection, crossover, ...) and the parameters used by said functions, it also stores information about the cost of the execution.
ga_run_generations stores the information about each individual generation used by the algorithm, while the table *ga_run_population* stores the population list containing all the individuals in each generation.
The database structure is shown in the diagram below:

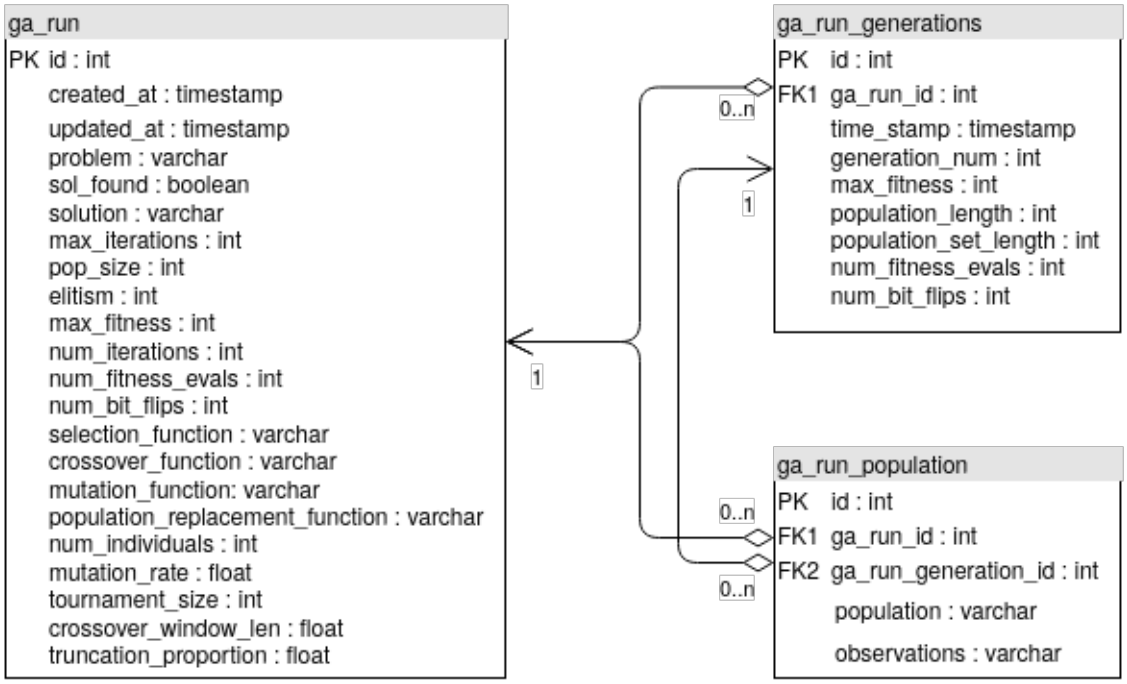

Figure 4.1: BBDD design schema

- *num_fitness_evals*: is a variable used to measure the cost of each generation, it records the number of fitness evaluations used by each genetic operator, the sum of which is the total cost of the generation.

- *num_bit_flips*: is also used to measure the cost of each generation, but it records the number of bit flips in each individuals genetic string used by the genetic operators.

- *population_set_length*: is the length of the population array when converted to a set, if no duplicates are allowed this value is equal to the *population_length*, this value is useful as a simple proxy of genetic diversity in the population.

The genetic algorithm can also calculate the phenotypic and genotypic variance and distributions in each generation.
The phenotypic distribution is the list of fitness values of each individual in the population, while the variance is a value calculated using the following formula:

$$\sigma_n^2 = \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X}^2)$$

where n is equal to the population length, X_i is the fitness value of individual i and $\overline{X}$ is the mean of all the fitness values in the population.
The genotypic distribution is a list of values where each value corresponds to the sum for the Hamming distances between each individuals gene string and the rest of the population divided by the population length, formally:

$$GV_i = \frac{\sum_{j=1,j\neq i}^{n} Dist_H(x_i, x_j)}{n}$$

where $Dist_H(x, y)$ returns the hamming distance between x and y, x_i is the genetic string of individual i and GV_i is the actual value stored in the list.
The hamming distance between two strings measures the number of substitutions needed to transform one string into the other.
The genotypic distributions are used throughout this work to measure the genetic diversity inside the different populations, the genotypic variance is calculated using the same formula as the phenotypic variance but substituting the fitness values for GV values.

4.3 Design of the experiments

Due to the prohibitively large computational cost, for the scope of this project, of running all the combinations of genetic operators and variations of their respective parameters on all the satisfiable uniform random 3-SAT problems in the SATLIB benchmark set, the experimentation part of this work is broken down into different phases.

During the genetic operator removal phase, shown in section 5.1, a small subset of "easy" problems is chosen in order to run all the different combinations of genetic operators on said problems, once all combinations have been run on the chosen problem subset the *success rate* of each operator is calculated by dividing the number of problems solved by the total number of problems attempted using the specified operator.
The success rate for each problem category is then multiplied by the weight assigned to that category, then they are all added up to form the final *score* of that operator, the genetic operators with a high enough score are then selected for phase two.
The second phase, described in section 5.1, is very similar to the first one, but the analysis is performed on a subset of 10 3-SAT problems with 100 variables each, and only the genetic operators chosen during the first phase will be run on them; An analysis of the effects of the combinations of the genetic operators on the success rate of the algorithm is also carried out during this phase.
During the third experimentation phase the 10 best combinations of genetic operators found during the second phase are run on a harder subset of 3-SAT problems with 175 variables each.
The analysis performed in this phase is different from that of phase 2, taking into account not only the success rate of the algorithms but also, their computational cost and the phenotypic and genotypic evolution of their populations.
The fourth and final phase is where the best combinations of genetic operators found during the third phase are compared with the SAT solvers explained in section 2.5.

4.4 Implementation

All of the algorithms and experiments in this work are written in python 3.7.7 due to it being the preferred language of the main researcher of this work is most comfortable with, which allows for a rapid iteration of the code necessary to implement the algorithms and run the experiments.
The python code is run on Debian 12.2 inside a docker container with the following dependencies installed through the python package manager pip3:

- *numpy*
- *matplotlib*
- *sklearn*
- *psycopg2-binary*
- *seaborn*

The database used is PostgreSQL 12.2 run on Debian 12.2 inside a docker container, another docker container with pgAdmin 4.18 on Alpine linux is also used to access and query the database.
A final docker container running Ubuntu 18.04 with Jupyter notebook and python 3.7.7 is also utilized to generate the different graphs and plots used throughout this work.
The deployment of all the containers except the one containing the Jupyter notebook is carried out through a docker-compose file, the Jupyter container is deployed using a shell script.
All of the code used can be found in Annex .2, as well as in a public git repository[1].

[1] https://github.com/punkyfer/3sat_genetic_algorithm

CHAPTER 5
Experimental Evaluation & Analysis

5.1 Genetic operator removal

During the phase of genetic operator removal, all of the different combinations of genetic operators which form the genetic algorithm are run on a small subset of "easy" problems from the SATLIB benchmark set; This is done in order to quickly eliminate the genetic operators which do not lead to good results before proceeding with a more in depth analysis, this step is necessary to reduce the computational cost of the subsequent experiments.
Each of the combinations of genetic operators is run many times on the same subset of 40 3-SAT problems, 10 problems with 20 variables, 10 with 50, 10 with 75 and finally 10 problems with 100 variables, all problems have been chosen from the SATLIB benchmark problem data set, one of the data sets used during the SAT-Race.
The measured value is the *success rate* which is equal to the number of problems solved divided by the total number of problems attempted, this value is commonly used when measuring genetic algorithms applied to SAT instances (**Gottlieb**; **Bhattacharjee2017**; **Marchiori1999**) since it offers a good representation of how well the algorithms will perform.
The success rate can be used in this work without making any adjustments since all the problems attempted are solvable, this requirement is needed due to the inefficiency of genetic algorithms with unsolvable instances explained in subsection 2.2.

All the experiments are run with the same parameters, except the experiments with *parents* and *weak parents* population replacement operators, which have higher number of maximum iterations (5000), the values used are listed below:

- *max_iterations* = 1000
- *pop_size* = 1000
- *elitism* = 0.01

The parameters needed by some genetic operators also remain constant throughout the different runs:

- *mutation_rate* = 0.05
- *crossover_window_len* = 0.4

	uf20	uf50	uf75	uf100	score
random	0.837	0.495	0.247	0.131	73.11
roulette	0.71	0.17	0.056	0.02	28.9
roulette w/ elimination	0.8	0.41	0.187	0.11	61.53
rank	0.81	0.42	0.156	0.09	57.9
tournament	0.74	0.33	0.107	0.049	44.23
stochastic	0.781	0.293	0.123	0.07	46.5
annealed	0.778	0.418	0.164	0.093	58.06
truncation	0	0	0	0	0

Table 5.1: Selection operators

- *num_individuals* = 0.4
- *tournament_size* = 5
- *truncation_proportion* = 1/3

	uf20	uf50	uf75	uf100	score
generational	0.87	0.472	0.177	0.1	64.27
mu lambda	0.987	0.562	0.274	0.172	85.59
mu lambda offspring	0.917	0.631	0.279	0.172	88.01
delete n	0.795	0.267	0.127	0.063	45.07
random	0.8	0.157	0.057	0.025	30.62
parents	0	0	0	0	0
weak parents	0	0	0	0	0

Table 5.2: Population replacement operators

From the results of the experiments shown in the tables above, it is not difficult to see that the selection operator *truncation*, as well as the population replacement operators *parents* and *weak parents* fail to find solutions for the easy problem instances, therefore it is highly unlikely that they will perform better on more difficult problems and so these operators can be safely removed.

5.2 Genetic operator analysis

The purpose of the second phase of the analysis is finding the best combinations of genetic operators, this is done by analyzing the *success rate* of each genetic operator when combined with a different genetic operator.
The table 5.3 shows the *success rate* of the genetic algorithm when using the specified combinations of *crossover* and *mutation* operators; The tables showing the other combinations of genetic operators can be found in tables 3 to 7 in Annex .1; All of the values shown in the specified tables are obtained by analyzing the runs of the algorithm on a subset of 10 3-SAT problems with 100 variables each.

Using the separated *success rates*, a value is assigned to each combination of genetic operators, this value is equal to the sum of the *success rates* for all the pairs of operators which can be formed using the specified genetic operators.

	two points	sliding window	random map	single point	uniform
single bit	0.1	0.137	0.1	0.016	0.015
multiple bit	0.08	0.2	0.033	0	0
flip ga	0.3	0.05	0.35	0.3	0.2
single bit greedy	0.075	0	0.1	0	0
single bit max greedy	0.1	0	0.15	0.1	0
multiple bit greedy	0.367	0	0.182	0.2	0.143

Table 5.3: Success rate obtained with the specified crossover and mutation operators on problems with 100 variables.

$$f(sel_{op}, cros_{op}, mut_{op}, rep_{op}) = sr(sel_{op}, cros_{op}) + sr(sel_{op}, mut_{op}) + sr(sel_{op}, rep_{op}) + sr(cros_{op}, mut_{op}) + sr(cros_{op}, rep_{op}) + sr(mut_{op}, rep_{op})$$

where $sel_{op}, cros_{op}, mut_{op}, rep_{op}$ are the selected operators for the *selection, crossover, mutation* and *population replacement* processes respectively, $sr(op_1, op_2)$ is the *success rate* calculated with the two specified genetic operators; For example:

$$f(random, two\_points, flip\_ga, generational) = 0.182 + 0.24 + 0.08 + 0.3 + 0.1 + 0.5 = 1.402$$

Using this formula, all combinations of genetic operators are assigned a value and then ranked in descending order, the best combinations of genetic operators are shown in table 5.4.

selection	crossover	mutation	population replacement	score
random	*two points*	*multiple bit greedy*	*mu lambda*	1.75
roulette elimination	*random map*	*flip ga*	*generational*	1.713
roulette elimination	*two points*	*flip ga*	*generational*	1.53
rank	*two points*	*multiple bit greedy*	*mu lambda*	1.44
roulette elimination	*sliding window*	*flip ga*	*generational*	1.415
random	*two points*	*flip ga*	*generational*	1.403
random	*random map*	*multiple bit greedy*	*mu lambda*	1.372
random	*two points*	*multiple bit greedy*	*mu lambda offspring*	1.371
random	*two points*	*flip ga*	*mu lambda*	1.353
random	*two points*	*multiple bit greedy*	*generational*	1.3
random	*two points*	*flip ga*	*mu lambda offspring*	1.288
random	*random map*	*flip ga*	*mu lambda*	1.21
random	*random map*	*multiple bit greedy*	*generational*	1.05
random	*random map*	*multiple bit greedy*	*mu lambda offspring*	1.04
random	*two points*	*single bit*	*generational*	0.592

Table 5.4: Top ranked combinations of genetic operators and randomly selected worse combination for comparison

5.3 In depth analysis

In this phase the top 14 ranked genetic algorithms found during the genetic operator analysis phase, as well as a randomly selected worse genetic algorithm, are run on a harder subset of 10 3-SAT problems with 175 variables each; The population size is increased to 2000 for these runs, but all the other parameters remain the same.
The randomly selected combination of genetic operators used for comparison with the top ranked genetic algorithms, along with its score is shown in table 5.4

Running time is not usually used as a computational cost measure due to the differences in the operating systems, programming language versions, hardware architecture, system load, and others, which might affect the running times and therefore give a distorted view of the algorithms.
Nonetheless, this work uses time as a proxy of computational cost to quickly eliminate genetic operator combinations with too high a cost, and offer a rough initial comparison of the cost of the different algorithms.
To offset the problems mentioned above which distort the running time of the algorithms all experiments have been run on the same system and each of the algorithms analyzed has also been run multiple times on each problem, this allows us to obtain the average

time cost across multiple runs which should offer a more accurate picture of the real running time of each algorithm.

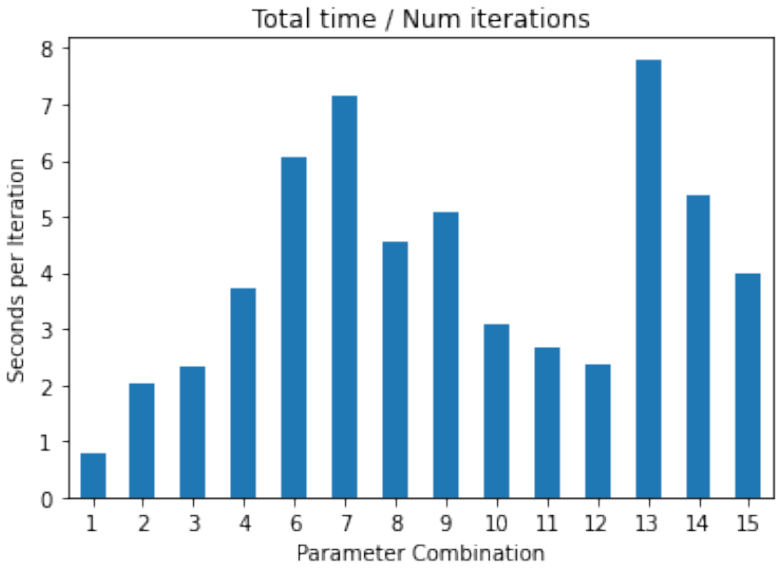

Figure 5.1: Time cost in seconds per iteration for the parameter combinations shown in table 5.5

Figure 5.1 shows the running time of each algorithm displayed in seconds/iteration, which is the number of seconds each algorithm takes to complete an iteration; The algorithm with *roulette w/ elimination* selection, *sliding window* crossover, *flip ga* mutation and *generational* replacement has been removed due to its extremely high running time (45.33 seconds), though a chart with its cost included can be seen in 1.

Table 5.5 shows the number assigned to each combination of genetic operators, along with the *success rate* of said combinations when applied to the chosen "hard" SATLIB problems with 175 variables.
Analyzing the evolution of the population in the different genetic algorithms can help explain the results obtained in table 5.5, this analysis is done using the average fitness of the population as the phenotype, and the average length of the population without duplicates divided by the total length of the population as the genotype.
The measurement used as the genotype is the length of the population array when converted to a set, this is a simplified proxy of the real genotypic diversity.
The real genotypic diversity is calculated using the manhattan distance between the individuals as explained in section 4.2, though it is not used in this case due to the extremely high computational cost of calculating it.
Figure 5.2 shows the phenotypic and genotypic evolution for all the selected genetic operator combinations in unsolved problems with 175 variables, the values displayed on the table are obtained by averaging all the unsolved runs with the specified genetic operator combination.
Single runs on some hard unsolved problems can be found in annex .1.

The worst combinations obtained in table 5.5 are the combinations numbered 12, 11, 9, 8, 7, 5, 2 and 1, which coincide with the graphs in figure 5.2 where the genetic diversity is quickly lost, this means the population is filled with copies of the best individuals which forces the algorithm to focus only on those solutions.
While this shift toward exploitation is desirable during the final stages of the algorithm, doing it too soon leads to a reduced exploration phase which means the algorithm will explore less of the solution, this increases its chances of getting stuck in local optimum points and reduces its chances of finding the global optimum.

number	selection	crossover	mutation	population replacement	success rate
1	*random*	*two points*	*single bit*	*generational*	0
2	*rank*	*two points*	*multiple bit greedy*	*mu lambda*	0.07
3	*random*	*two points*	*multiple bit greedy*	*mu lambda*	0.2
4	*roulette elimination*	*random map*	*flip ga*	*generational*	0.09
5	*roulette elimination*	*sliding window*	*flip ga*	*generational*	0
6	*roulette elimination*	*two points*	*flip ga*	*generational*	0.09
7	*random*	*two points*	*multiple bit greedy*	*generational*	0
8	*random*	*two points*	*multiple bit greedy*	*mu lambda offspring*	0
9	*random*	*two points*	*flip ga*	*mu lambda offspring*	0
10	*random*	*random map*	*multiple bit greedy*	*mu lambda*	0.2
11	*random*	*random map*	*multiple bit greedy*	*generational*	0
12	*random*	*random map*	*multiple bit greedy*	*mu lambda offspring*	0
13	*random*	*two points*	*flip ga*	*generational*	0.33
14	*random*	*random map*	*flip ga*	*mu lambda*	0.52
15	*random*	*two points*	*flip ga*	*mu lambda*	0.23

Table 5.5: Genetic operator combination id and success success rate of runs for problems with 175 variables.

Even though combination number 1 maintains a good phenotypic diversity during its evolution, its success rate is the worst of them all, since it fails to exploit its available solutions enough to find the answer to the problem stated.

The best combinations found in table 5.5 are 14, 13, 15, 3 and 10, in that order, but combinations 15 and 3 can be removed since they display a very high loss of diversity during the initial phase down to a stable small population of the fittest individuals, this can be seen in figure 5.2, this loss of diversity is also shown in figures 9 and 10 which show the behavior of the combinations on solved runs, as well as figures 2 to 8 which show unsolved runs.
The combinations numbered 15 and 3 instead focus mostly on the exploitation of those solutions, this makes them highly conditioned by the initial population generated since they barely explore any new solutions after the first stages of the algorithm.

Figures 5.3 and 5.4 show a discrepancy with combination number 13, since it does less fitness evaluations than combination number 13, yet it has a higher time cost per iteration, this higher time cost can be explained by the computational cost of each of the different operators between them.
The time cost of the implementation of each genetic operator is shown in table 5.6, while the difference between the population replacement operators is much greater, this has little effect on the final running time of each algorithm since the replacement function

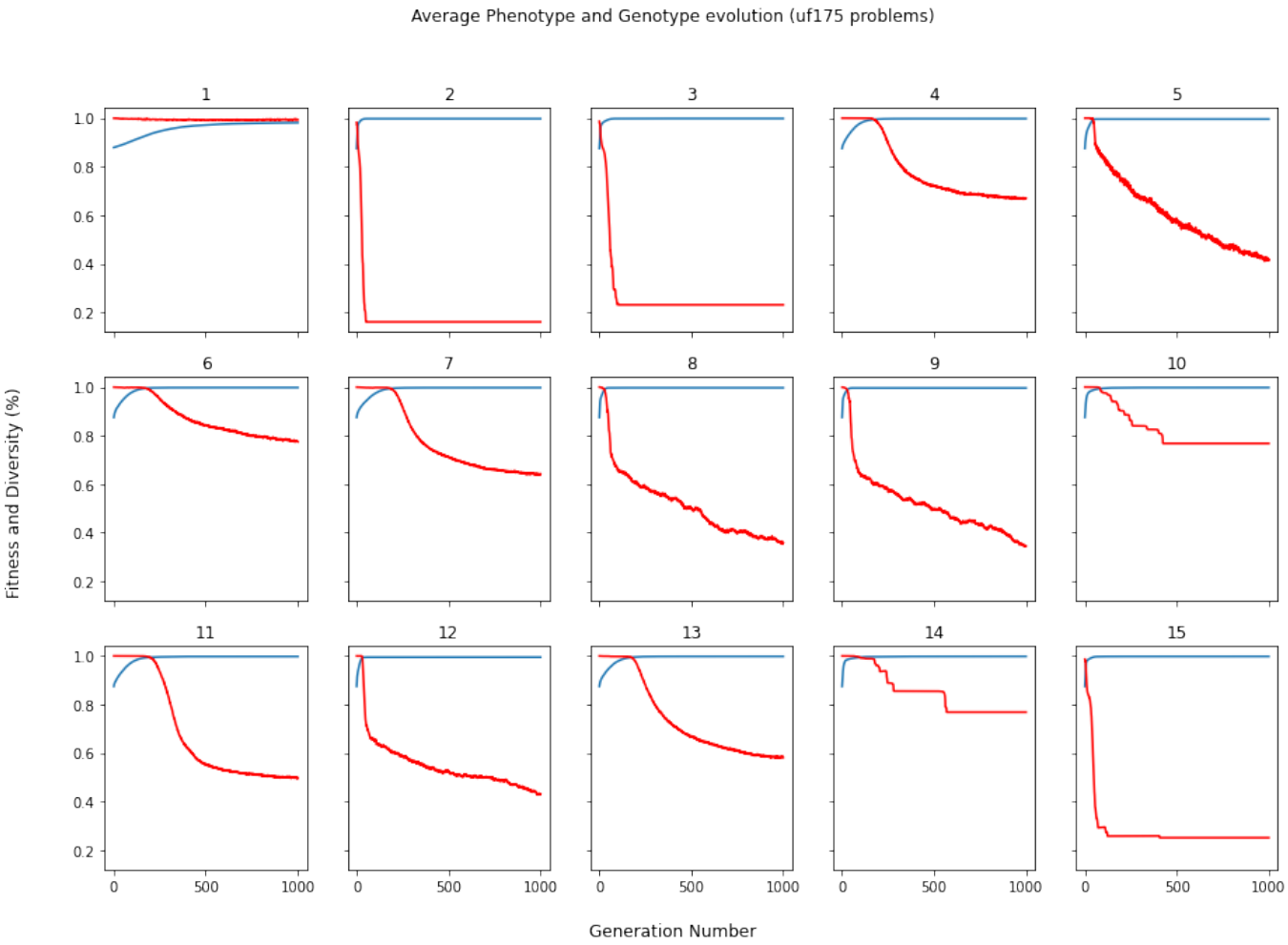

Figure 5.2: Average phenotypic and genotypic diversity for unsolved runs of problems with 175 variables

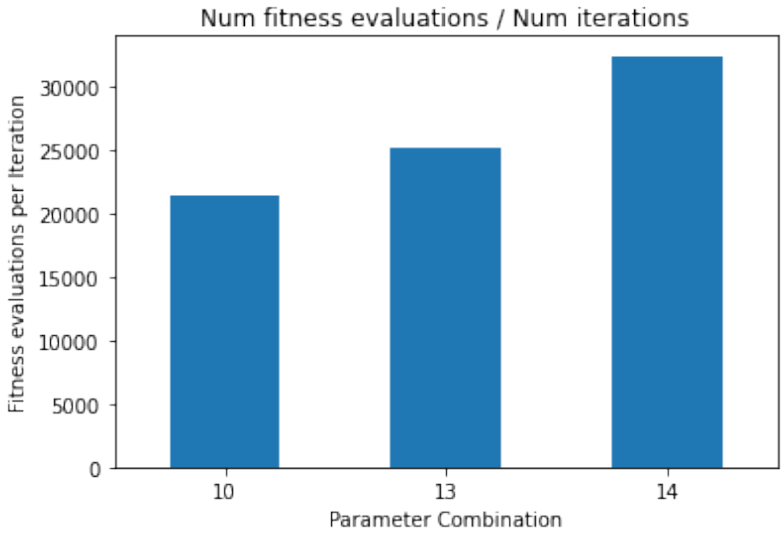

Figure 5.3: Average number of fitness evaluations per iteration for the specified genetic operator combinations

is only executed once per iteration, while the mutation operators get executed once per individual every iteration.

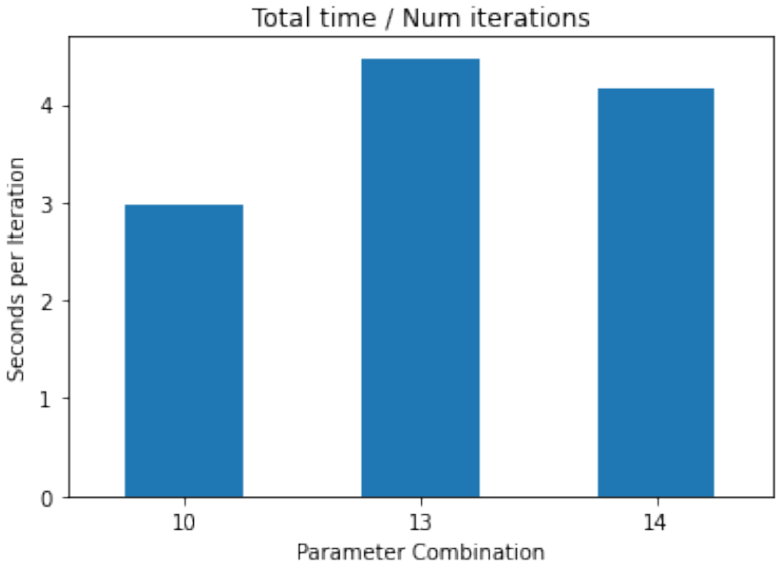

Figure 5.4: Average number of seconds per iteration for the specified genetic operator combinations

With a population size of 2000 the *two points* genetic operator is 0.0252 seconds slower than the *random map* operator, this difference is much larger than the difference of $32.75 * 10^{-5}$ seconds found between the population replacement operators.
This increase in execution time does not carry with it an increase in the computational power of the algorithm since combination number 13 obtains a worse success rate than combination number 14 which takes less time to run.
For this reasons combination number 13 is removed from the best combinations, the algorithms chosen to be compared with the SAT solvers are numbers 14 and 10.

genetic operator	time cost
two points	$58.62 * 10^{-5}$
random map	$57.36 * 10^{-5}$
mu lambda	$56.88 * 10^{-5}$
generational	$24.13 * 10^{-5}$

Table 5.6: Time cost of the specified genetic operators.

Combination number 10 can be seen as a simplified version of number 14 since they are virtually identical, they have the same genetic operators except for the mutation function, while combination number 14 uses a recursive version of the *multiple bit greedy* mutation operator called *flip ga*, combination number 10 uses the basic *multiple bit greedy* operator.
The use of *flip ga* increases the success rate of the algorithm, though it also carries with it an increase in execution time, as shown in table 5.6, due to the increased number of fitness evaluations and bit flips carried out every iteration, this can be seen in figures 5.3 and 5.5 respectively.

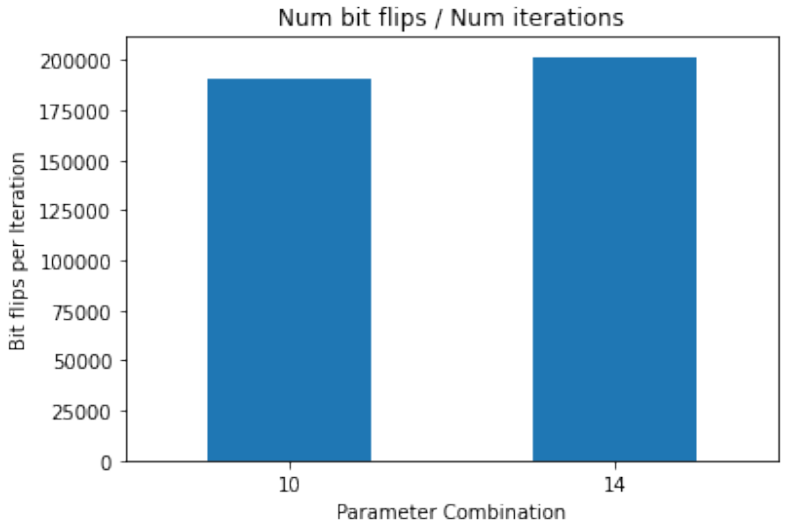

Figure 5.5: Average number of bit flips per iteration for the specified genetic operator combinations

5.4 Comparison with SAT solvers

All of the SAT solvers explained in section 2.5 are run on 100 problems of each problem size (20, 50, 75, 125, 150, 175, 200, 225, 250) from the SATLIB benchmark dataset.
Table 5.7 shows only the running time of the SAT solvers evaluated since all of them satisfy all of the SAT problems in the allotted amount of time, therefore they all have a success rate of 1.0.
Only the experiments on the harder problems are shown in table 5.7, this is due to the fact that the time spent by each solver on the easier problems is so small that for all events and purposes it can be considered 0, the remaining results are shown in table 8.

sat solver	uf150	uf175	200	225	250
cadical	0.02	0.07	0.18	0.51	1.55
clasp	0.01	0.02	0.05	0.16	0.58
lingeling	0.05	0.09	0.19	0.57	2.19
manysat	0.01	0.01	0.04	0.1	0.33
minisat	0.01	0.03	0.09	0.29	1.26
plingeling	0.06	0.07	0.13	0.23	0.52
treengeling	0.05	0.08	0.19	0.26	0.55
zchaff	0.02	0.04	0.2	0.68	4.97

Table 5.7: Average time cost (in seconds) for each problem of the specified size solved using the specified solver.

Tables 5.9 and 5.8 show the success rate and time cost of the selected genetic algorithms when applied to the same problems as the SAT solvers, though in this case only the easier problems are shown since they already offer enough information to compare the genetic algorithms with the SAT solvers.

genetic operator	uf20	uf50	75	100	uf125
flip ga	0.14	3.74	22.78	69.06	137.36
multiple bit greedy	0.17	4.24	13.88	33.71	62.79

Table 5.8: Average time cost (in seconds) for each problem of the specified size solved using the specified mutation operator.

genetic operator	uf20	uf50	75	100	uf125
flip ga	1	0.93	0.72	0.44	0.37
multiple bit greedy	1	0.85	0.59	0.37	0.27

Table 5.9: Success rate for the problems of the specified size solved using the specified mutation operator.

5.5 Conclusions

From the results obtained in tables 5.7, 5.8, 5.9 and 5.5 it is clear that genetic algorithms algorithms are no match for the analyzed SAT solvers, which are superior to the genetic algorithms for solving SAT instances in every way, from the success rate to the execution time.
Though the execution time for the genetic can be decreased it comes with a decrease in the success rate and vice versa.
The biggest difference in execution time comes from the stochastic nature of the genetic algorithms, even though the mutation operators selected try to mitigate this by applying more local exploration on each individual, the running time of the algorithm is still highly dependent on the initial population generated.
Another limiting factor is the fact that genetic algorithms have no memory of the previous populations, therefore it is possible for them to waste computing cycles by mutating the same solutions multiple times, only the mutation operator is mentioned since it's the process where the algorithm spends the most amount of time.
This could be limited to a certain extent by using a dictionary where the key is the individual and the value is the same individual after undergoing the mutation process, the dictionary can be used due to the fact that the mutation operators selected are deterministic processes, therefore for the same input they will always generate the same result; The randomness in the mutation process comes from the fact that an individual has a probability of not undergoing process.
Due to the time constraints imposed on this work, some optimizations which could improve the execution time of the genetic algorithms have not been implemented.
Chief among these optimizations is designing parallel implementations of the selection, crossover and mutation operators since these are the biggest computational costs and also the best suited for parallel processing, the population replacement operator is only executed once every iteration therefore a parallel is not needed.
Using a dictionary should decrease the computation time at an increased cost in the space needed to run the algorithm.
Some techniques from the field of hyper-parameter optimization could also be implemented to help search for a more optimum assignment of values to the parameters used by the genetic algorithm, this should increase the success rate of the algorithm.
Even though the genetic algorithms can be improved by adding memory, implementing

parallel operators or fine-tuning the parameters, through the analysis performed in this chapter it is shown that the biggest improvements to the algorithm occur when more local search is applied to the individuals generated.
This is in line with the results displayed in tables 5.7, 5.8, 5.9 and 5.5 which show the local search algorithms outperforming the genetic algorithms, so while the genetic algorithms can be improved by applying more local search, this defeats the purpose since using a pure local search algorithm will offer better results.

CHAPTER 6
Conclusions

This work aimed to study the behavior of genetic algorithms when applied to SAT problems in order to determine if they could be a viable alternative to the local search algorithms which currently dominate the SAT-Race.
Through the experimental evaluation carried out in chapter 5 it is demonstrated that the best genetic algorithms for solving SAT instances are the ones where most local search is performed, proving the usefulness of hybrid genetic algorithms.
Yet even the best genetic algorithms found through experimentation, under-perform when compared to the selected SAT solvers which all have a higher success rate and a lower execution time than the genetic algorithms.
While the results obtained are discouraging for genetic algorithms as SAT solvers, the research is nonetheless useful since it gives insight into the behavior of said algorithms and how the different genetic operators affect the exploration of the solution space.

Bibliography

Holland, John H (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press. ISBN: 0262082136.

Das, Amit Kumar and Dilip Kumar Pratihar (2018). "Performance improvement of a genetic algorithm using a novel restart strategy with elitism principle". In: *International Journal of Hybrid Intelligent Systems* 15.1, pp. 1–15. ISSN: 14485869. DOI: 10.3233/his-180257.

Cochran, James J. et al. (2011). "Random Search Algorithms". In: *Wiley Encyclopedia of Operations Research and Management Science*. DOI: 10.1002/9780470400531.eorms0704.

Karp, Richard M. (2010). *Reducibility among combinatorial problems*. DOI: 10.1007/978-3-540-68279-0_8.

Eleschinski2000 (2015). *File:Difference between deterministic and Nondeterministic.png - Wikimedia Commons*. URL: https://commons.wikimedia.org/wiki/File:Difference%7B%5C_%7Dbetween%7B%5C_%7Ddeterministic%7B%5C_%7Dand%7B%5C_%7DNondeterministic.png (visited on 08/24/2020).

Qef (2007). *File:Complexity subsets pspace.svg - Wikimedia Commons*. URL: https://commons.wikimedia.org/wiki/File:Complexity%7B%5C_%7Dsubsets%7B%5C_%7Dpspace.svg (visited on 08/24/2020).

Behnam Esfahbod (2017). *File:P np np-complete np-hard.svg - Wikimedia Commons*. URL: https://commons.wikimedia.org/wiki/File:P%7B%5C_%7Dnp%7B%5C_%7Dnp-complete%7B%5C_%7Dnp-hard.svg (visited on 08/24/2020).

Cook, Stephen A. (1971). *The complexity of theorem-proving procedures*. DOI: 10.1145/800157.805047.

Eén, Niklas and Sorensson Niklas (2000). "Extensible SAT Solver". In:

Gebser, Martin et al. (2007). "clasp: A conflict-driven answer set solver". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4483 LNAI, pp. 260–265. ISSN: 16113349. DOI: 10.1007/978-3-540-72200-7_23.

Hamadi, Youssef, Said Jabbour, and Lakhdar Sais (2009). "ManySAT: a Parallel SAT Solver". In: *Journal on Satisfiability, Boolean Modeling and Computation* 6.4, pp. 245–262. ISSN: 1574-0617. DOI: 10.3233/sat190070.

Biere, Armin (2017). "CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017". In: *Proceedings of SAT Competition 2017 - Solver and Benchmark Descriptions*, pp. 14–15. URL: http://fmv.jku.at/papers/Biere-SAT-Competition-2017-solvers.pdf.

Davis, Martin, George Logemann, and Donald Loveland (1962). "A machine program for theorem-proving". In: *Communications of the ACM* 5.7, pp. 394–397. ISSN: 15577317. DOI: 10.1145/368273.368557.

Gent, Ian P., Chris Jefferson, and Ian Miguel (2006). "Watched literals for constraint propagation in Minion". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4204 LNCS, pp. 182–197. ISSN: 16113349. DOI: 10.1007/11889205_15.

Bacchus, Fahiem and Paul Van Run (1995). "Dynamic variable ordering in CSPs". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 976, pp. 258–275. ISSN: 16113349. DOI: 10.1007/3-540-60299-2_16.

Moskewicz, M.W. et al. (2001). "Chaff: engineering an efficient SAT solver". In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pp. 530–535. DOI: 10.1145/378239.379017. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=%7B%5C&%7Darnumber=935565%7B%5C&%7Disnumber=20239.

Biere, Armin (2010). *Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010*. Tech. rep. August. Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, p. 4. URL: http://baldur.iti.uka.de/sat-race-2010/.

— (2012). *Lingeling and friends entering the SAT challenge 2012*. Tech. rep., p. 2.

Järvisalo, Matti, Marijn J.H. Heule, and Armin Biere (2012). "Inprocessing rules". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7364 LNAI, pp. 355–370. ISSN: 03029743. DOI: 10.1007/978-3-642-31365-3_28.

Gottlieb, Jens, Elena Marchiori, and Claudio Rossi (2002). *Evolutionary algorithms for the satisfiability problem*. Tech. rep. 1, pp. 35–50. DOI: 10.1162/106365602317301763.

Lovíšková, Jana (2015). "Solving the 3-SAT problem using genetic algorithms". In: *INES 2015 - IEEE 19th International Conference on Intelligent Engineering Systems, Proceedings*. Institute of Electrical and Electronics Engineers Inc., pp. 207–212. ISBN: 9781467379397. DOI: 10.1109/INES.2015.7329708.

Harmeling, Stefan (2000). *Solving Satisfiability Problems with Genetic Algorithms*. Tech. rep., pp. 206–213.

Bhattacharjee, Arunava and Prabal Chauhan (2017). "Solving the SAT problem using genetic algorithm". In: *Advances in Science, Technology and Engineering Systems* 2.4, pp. 115–120. ISSN: 24156698. DOI: 10.25046/aj020416.

Marchiori, Elena and Claudio Rossi (1999). "A Flipping Genetic Algorithm for Hard 3-SAT Problems". In: *Proc. of the {G}enetic and {E}volutionary {C}omputation {C}onf. {GECCO}-99*, pp. 393–400.

Folino, Gianluigi, Clara Pizzuti, and Giandomenico Spezzano (1998). "Combining Cellular Genetic Algorithms and Local Search for Solving Satisfiability Problems". In: *Proceedings ofTenth IEEE International Conference on Tools with Artificial Intelligence*, pp. 192–198.

Jebari, Khalid, Mohammed Mediafi, and Abdelaziz Elmoujahid (2013). "Parent Selection Operators for Genetic Algorithms". In: *International Journal of Engineering Research & Technology (IJERT)* 2.11, pp. 1141–1145.

Pencheva T, Atanassov K, and Shannon A (2010). "Modelling of a stochastic universal sampling selection operator in genetic algorithms using generalized nets". In: pp. 1–7.

Jyotishree, Jyotishree and Rakesh Kumar (2012a). "Chapter 6: SELECTION 6.1 Introduction". In: *Knowledge based operation and problems representation in genetic algorithms*. Kurukshetra, pp. 94–124. URL: http://shodhganga.inflibnet.ac.in/bitstream/10603/32680/16/16%7B%5C_%7Dchapter%206.pdf.

— (2012b). "Chapter 7 : REPLACEMENT". In: *Knowledge based operation and problems representation in genetic algorithms*, pp. 125–135.

Lee, Chang Yong (2003). "Entropy-boltzmann selection in the genetic algorithms". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 33.1, pp. 138–142. ISSN: 10834419. DOI: 10.1109/TSMCB.2003.808184.

A.J., Umbarkar and Sheth P.D. (2015). "Crossover Operators in Genetic Algorithms: a Review". In: *ICTACT Journal on Soft Computing* 06.01, pp. 1083–1092. ISSN: 09766561. DOI: 10.21917/ijsc.2015.0150.

Abdoun, Otman, Jaafar Abouchabaka, and Chakir Tajani (2012). "Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem". In: arXiv: 1203.3099. URL: http://arxiv.org/abs/1203.3099.

[1] Jennifer S. Light. When computers were women. *Technology and Culture*, 40:3:455–483, juliol, 1999.

[2] Georges Ifrah. *Historia universal de las cifras*. Espasa Calpe, S.A., Madrid, sisena edició, 2008.

[3] Comunicat de premsa del Departament de la Guerra, emés el 16 de febrer de 1946. Consultat a http://americanhistory.si.edu/comphist/pr1.pdf.

Appendices

.1 Annex A

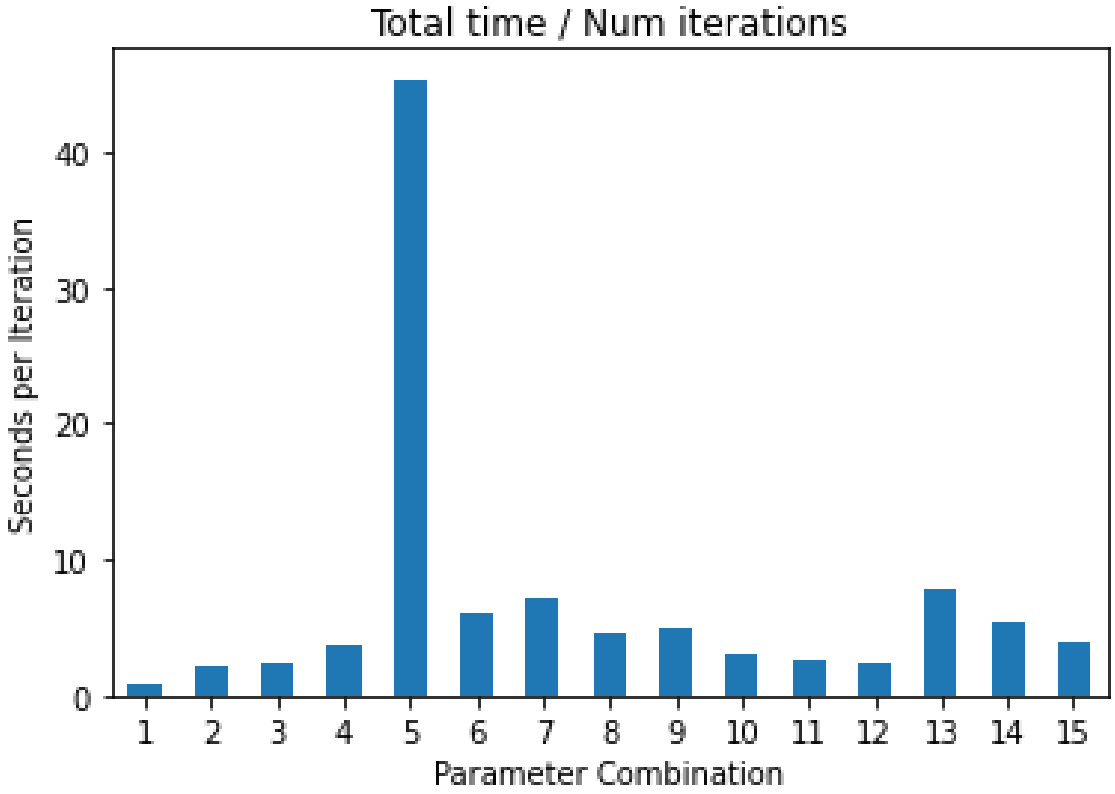

Figure 1: Time cost in seconds per iteration for all the parameter combinations shown in table 5.5

	uf20	uf50	uf75	uf100	score
single point	0.582	0.164	0.042	0.016	24.59
two points	0.905	0.595	0.27	0.173	85.4
sliding window	0.805	0.271	0.114	0.081	45.3
random map	0.905	0.606	0.28	0.15	84.4
uniform	0.818	0.317	0.129	0.068	48.69

Table 1: Crossover operators

	uf20	uf50	uf75	uf100	score
single bit	0.706	0.164	0.042	0.048	27.79
multiple bit	0.83	0.337	0.075	0.062	45.27
single bit greedy	0.723	0.272	0.133	0.05	43.03
single bit max greedy	0.793	0.322	0.119	0.071	47.98
multiple bit greedy	0.98	0.637	0.326	0.222	98.1
flip ga	1	0.75	0.433	0.133	103.27

Table 2: Mutation operators

	random	roulette elimination	rank	annealed	roulette	tournament	stochastic
single point	0.014	0.014	0.014	0.02	0	0.028	0.025
two points	0.182	0.067	0.2	0.1	0.1	0	0.1
sliding window	0.2	0.174	0.1	0.1	0.02	0.1	0.1
random map	0.08	0.162	0	0.133	0.1	0.1	0.1
uniform	0.067	0.041	0.05	0.1	0	0.25	0.083

Table 3: Success rate obtained with the specified selection and crossover operators on problems with 100 variables.

	random	roulette elimination	rank	annealed	roulette	tournament	stochastic
single bit	0.054	0.07	0.027	0.055	0.027	0.04	0.05
multiple bit	0.05	0.04	0.15	0.1	0	0	0
flip ga	0.24	0.4	0.333	0.2	0.05	0.2	0.2
single bit greedy	0.1	0.043	0.1	0.1	0	0	0
single bit max greedy	0.1	0.2	0.1	0.1	0	0	0
multiple bit greedy	0.37	0.133	0.133	0.2	0	0.1	0.3

Table 4: Success rate obtained with the specified selection and mutation operators on problems with 100 variables.

	random	roulette elimination	rank	annealed	roulette	tournament	stochastic
generational	0.08	0.163	0.04	0.08	0.06	0.05	0.08
mu lambda	0.19	0.1	0.1	0	0	0.1	0
mu lambda offspring	0.125	0.1	0.171	0.14	0	0	0
random	0	0.031	0	0.1	0	0	0
delete n	0	0	0	0	0	0.067	0.1

Table 5: Success rate obtained with the specified selection and population replacement operators on problems with 100 variables.

	two points	sliding window	random map	single point	uniform
generational	0.1	0.128	0.138	0.014	0.018
mu lambda	0.19	0.1	0.1	0.06	0
mu lambda offspring	0.183	0.2	0.14	0.1	0.1
random	0	0.154	0	0.014	0
delete n	0	0.1	0	0.02	0.08

Table 6: Success rate obtained with the specified crossover and population replacement operators on problems with 100 variables.

	single bit	multiple bit	flip ga	single bit greedy	single bit max greedy	multiple bit greedy
generational	0.075	0	0.5	0.1	0.2	0.2
mu lambda	0.06	0.05	0.25	0.1	0.1	0.45
mu lambda offspring	0.1	0.133	0.257	0.1	0.1	0.143
random	0.014	0.05	0.3	0	0.1	0.2
delete n	0.02	0	0.2	0	0	0.2

Table 7: Success rate obtained with the specified mutation and population replacement operators on problems with 100 variables.

sat solver	uf20	uf50	75	100	125
cadical	0	0.01	0	0	0.01
clasp	0	0	0	0.01	0
lingeling	0.01	0	0.02	0.04	0.04
manysat	0	0	0	0	0.01
minisat	0	0	0	0.01	0
plingeling	0.01	0.02	0.02	0.05	0.05
treengeling	0.01	0.02	0.02	0.05	0.05
zchaff	0	0	0	0	0.01

Table 8: Average time cost (in seconds) for each problem of the specified size solved using the specified solver.

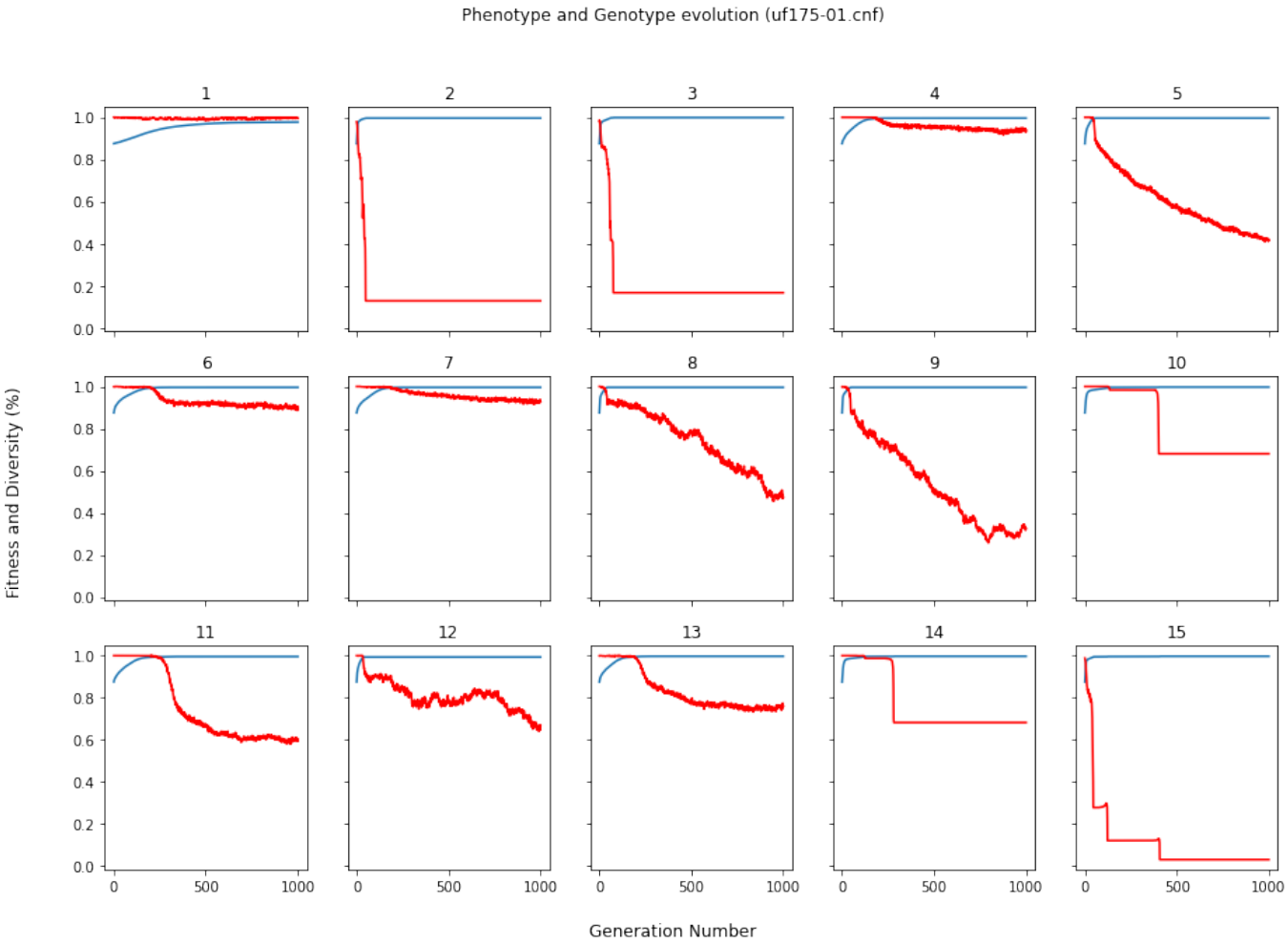

Figure 2: Phenotypic and genotypic diversity for unsolved runs of problem *uf175-01.cnf*

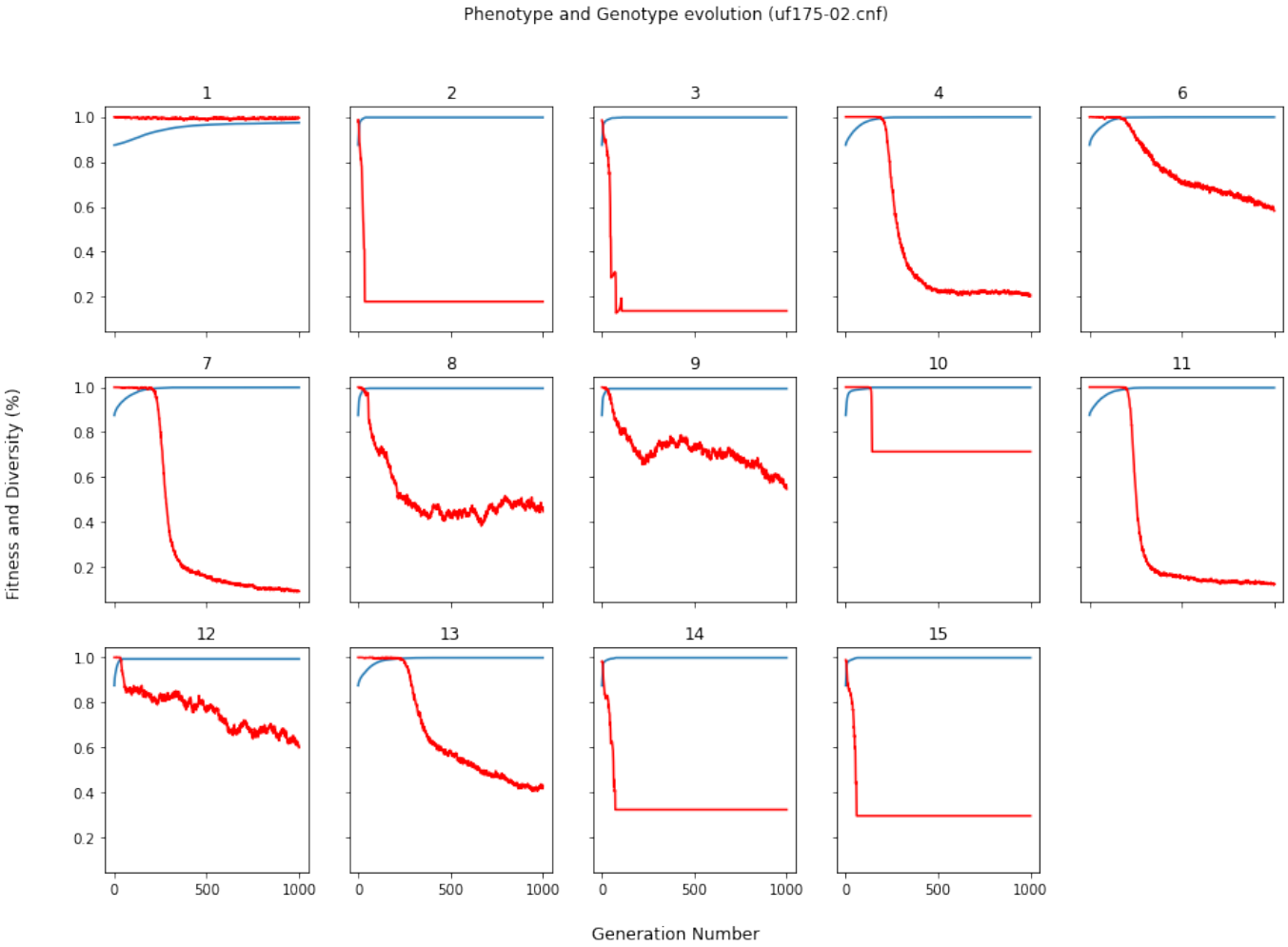

Figure 3: Phenotypic and genotypic diversity for unsolved runs of problem *uf175-02.cnf*

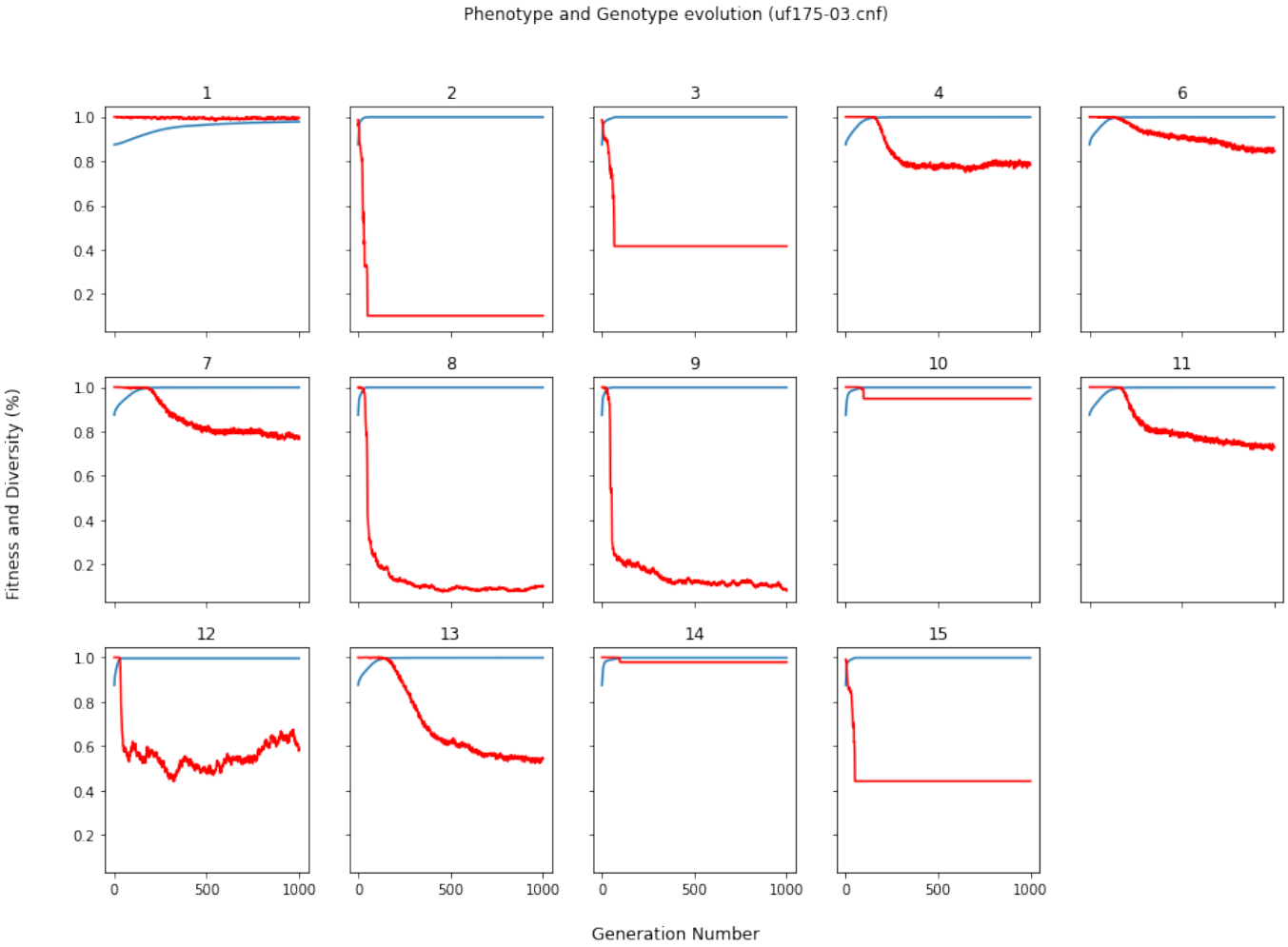

Figure 4: Phenotypic and genotypic diversity for unsolved runs of problem *uf175-03.cnf*

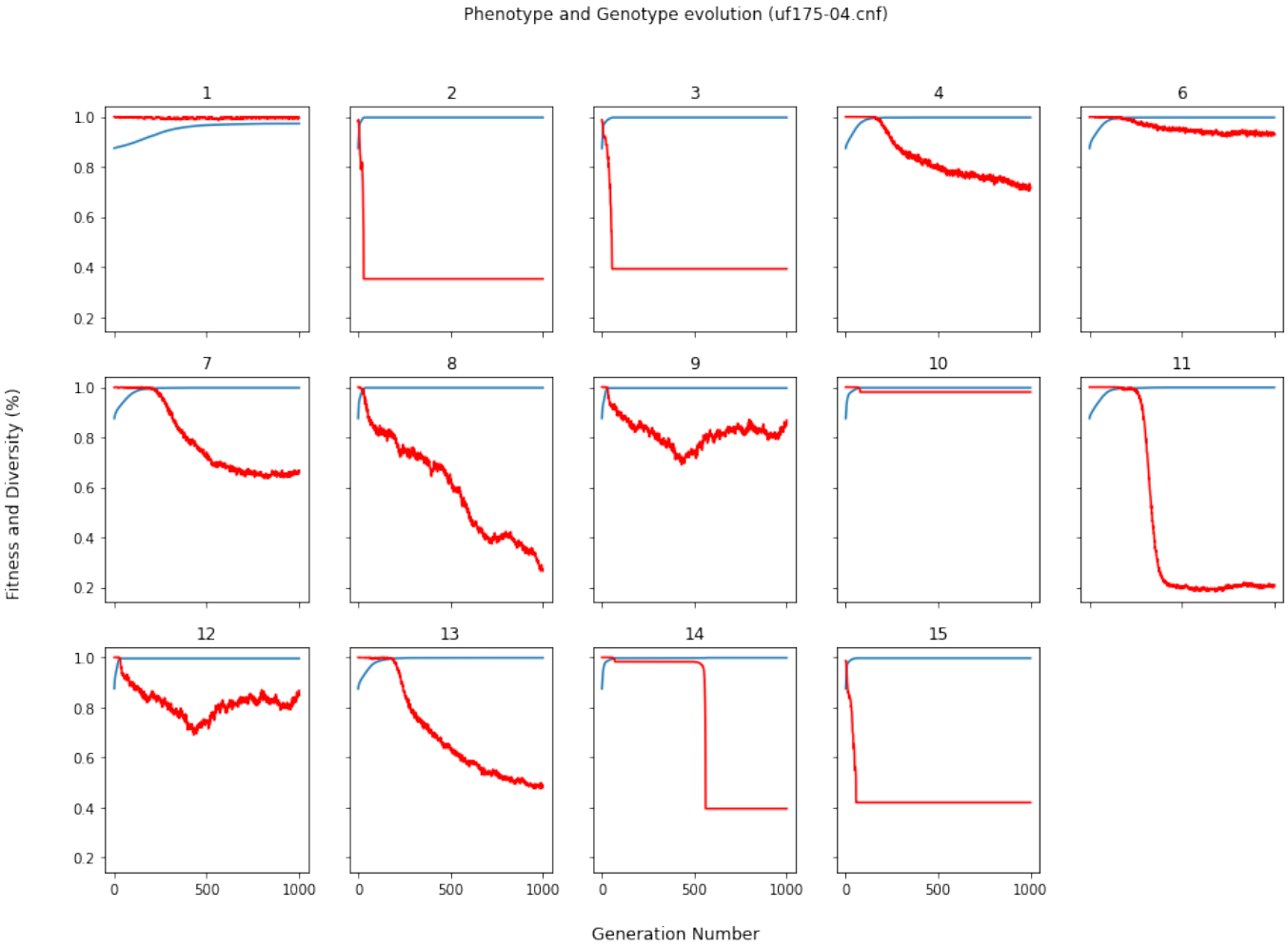

Figure 5: Phenotypic and genotypic diversity for unsolved runs of problem *uf175-04.cnf*

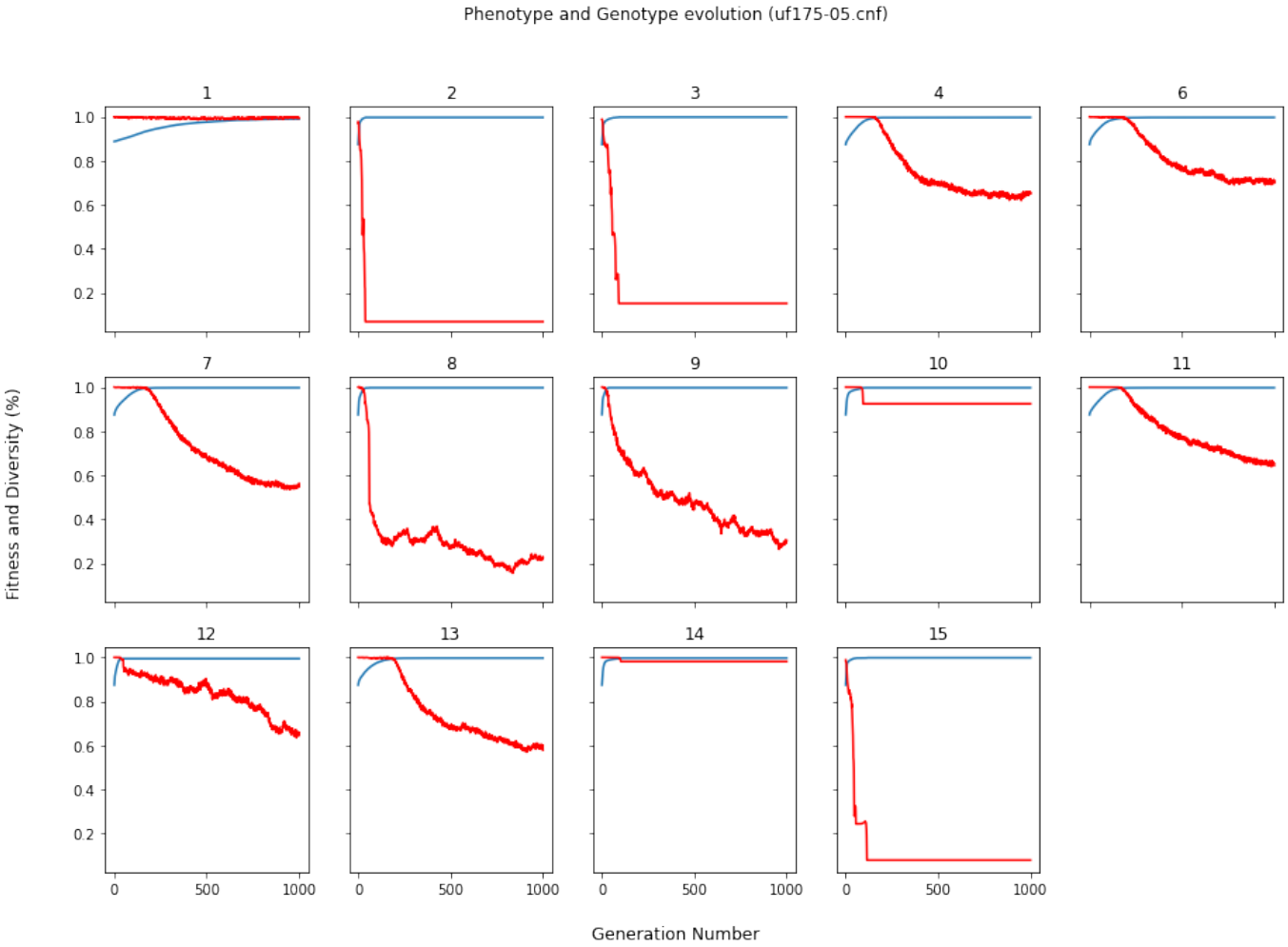

Figure 6: Phenotypic and genotypic diversity for unsolved runs of problem *uf175-05.cnf*

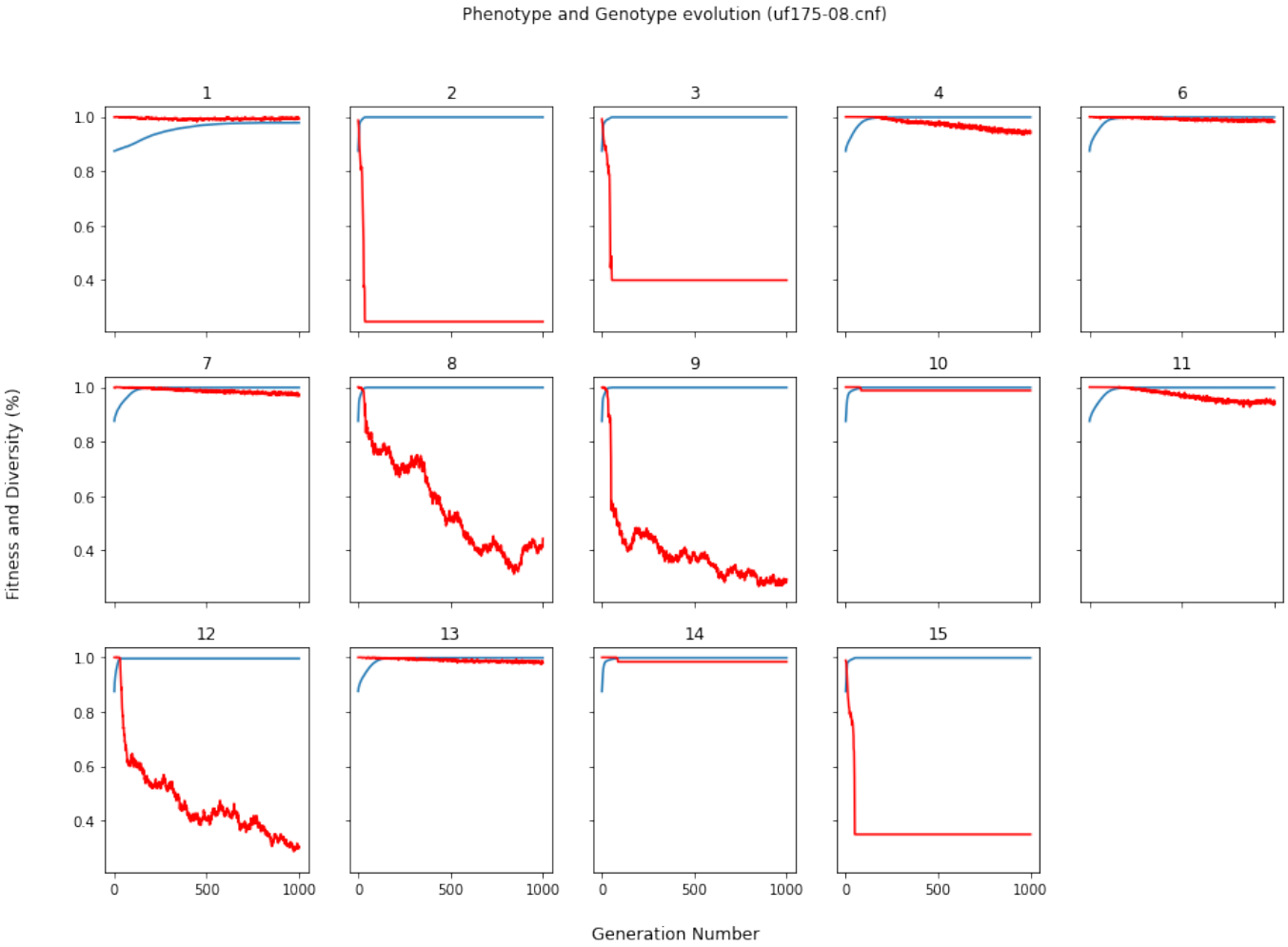

Figure 7: Phenotypic and genotypic diversity for unsolved runs of problem *uf175-08.cnf*

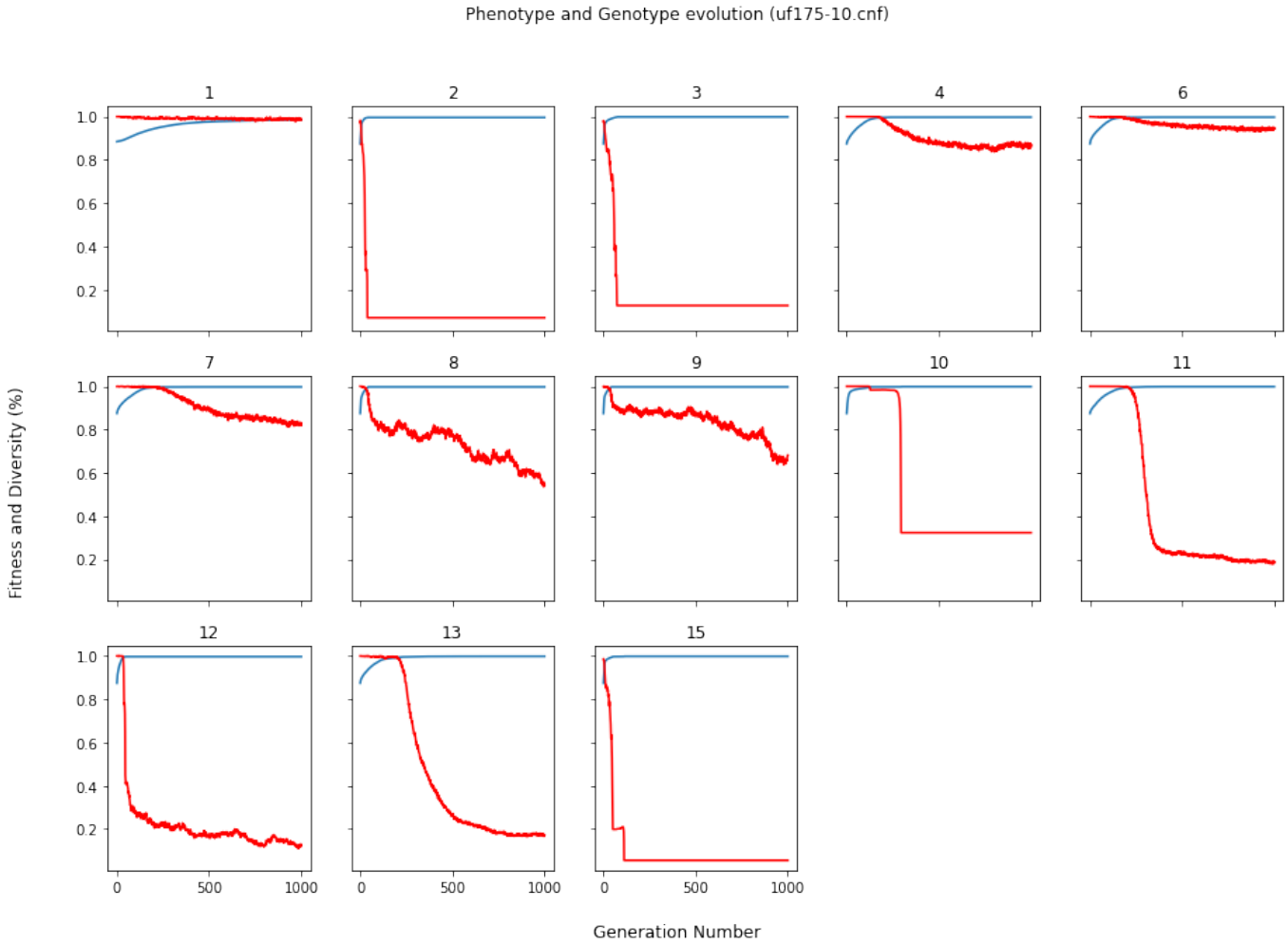

Figure 8: Phenotypic and genotypic diversity for unsolved runs of problem *uf175-10.cnf*

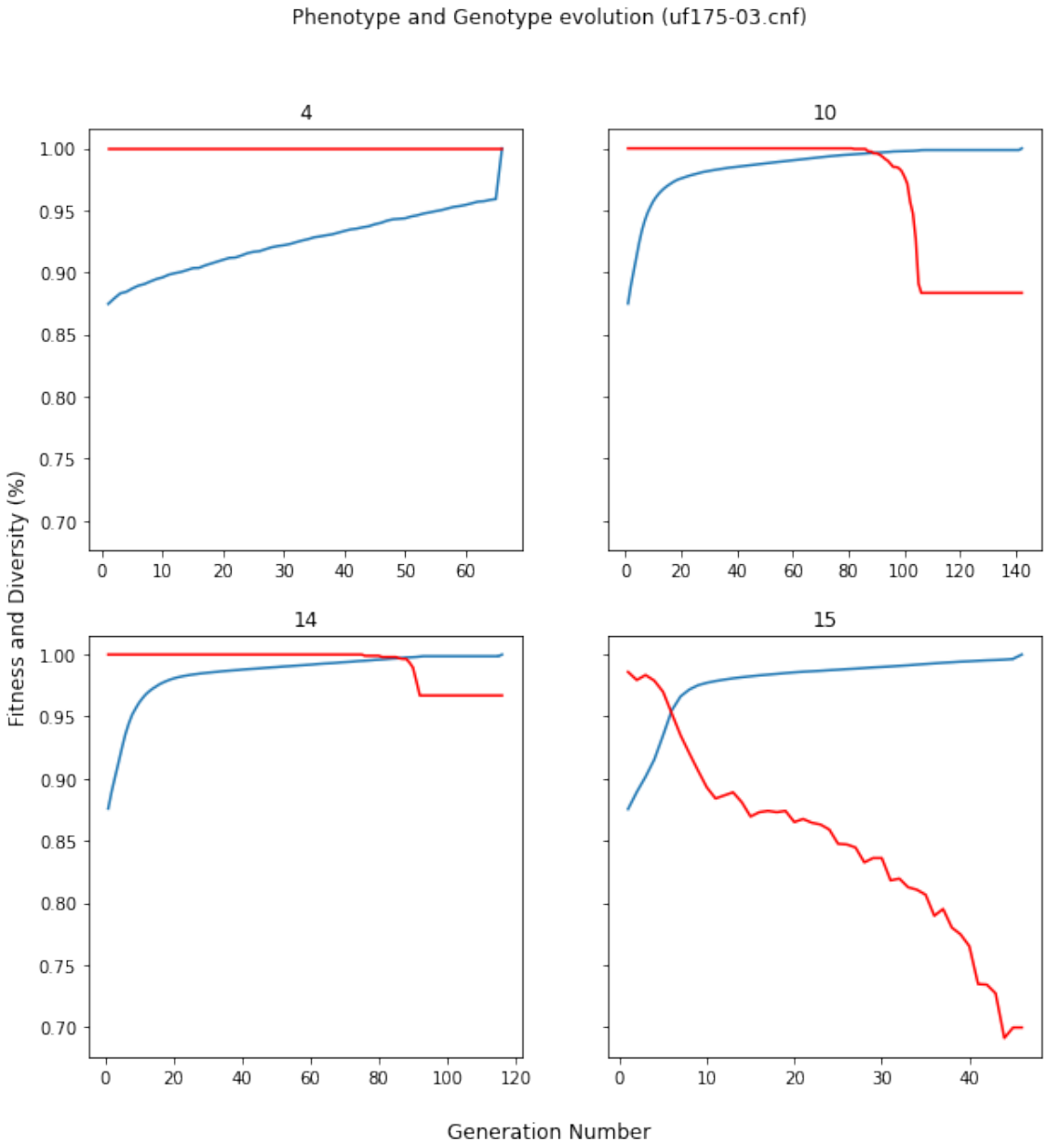

Figure 9: Phenotypic and genotypic diversity for solved runs of problem *uf175-03.cnf*

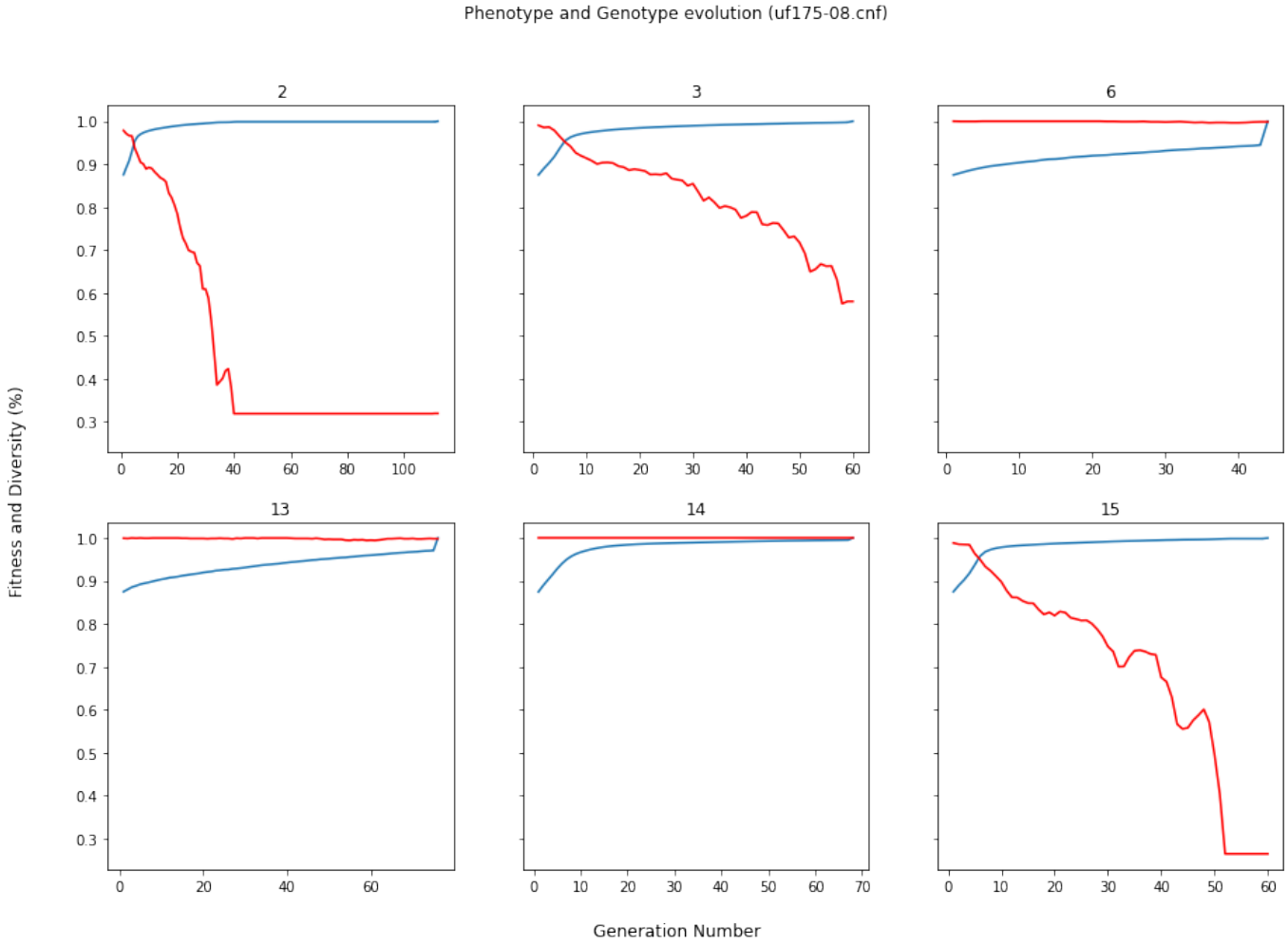

Figure 10: Phenotypic and genotypic diversity for solved runs of problem *uf175-08.cnf*

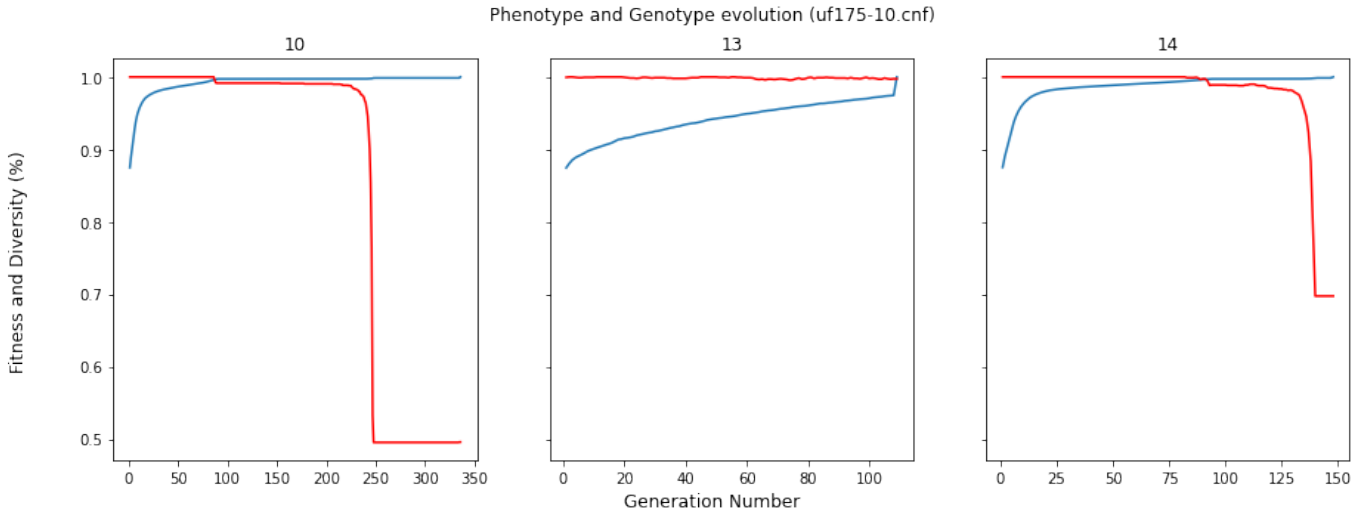

Figure 11: Phenotypic and genotypic diversity for solved runs of problem *uf175-10.cnf*

.2 Annex B

```
import functions as fn
import time
from datetime import datetime

############## CONSTANTS ##############
infinite = 2**31

############## GA CLASS ##############

class GeneticAlgorithm:

   def __init__(self, filename, max_iters=1000, pop_size=100, elitism=0.1,
       allow_duplicates=True,
      steady_state_replacement=False, save_to_db = True, plot_results = True
          , max_workers = 1):
      self.filename = filename
      self.num_vars, self.clauses = fn.read_problem(filename)
      self.set_vars = [infinite]*self.num_vars
      self.sol_value = fn.trivial_case(self.clauses)
      self.clauses, self.set_vars = fn.remove_pure_vars(self.clauses, self.
          set_vars)
      self.clauses, self.set_vars = fn.remove_unit_vars(self.clauses, self.
          set_vars)
      self.fitness_func = fn.maxsat_fitness
      self.selection_func = fn.roulette_selection
      self.selection_params = ()
      self.crossover_func = fn.single_point_crossover
      self.crossover_params = ()
      self.mutation_func = fn.single_bit_flip
      self.mutation_params = (0.1,)

      self.max_iters = max_iters
      self.pop_size = pop_size
      self.elitism = int(elitism*pop_size)
      self.log_level = None
      self.ret_cost = True
      self.steady_state_replacement = steady_state_replacement
      self.allow_duplicates = allow_duplicates
      self.save_to_db = save_to_db
      self.db_conn = None
      self.max_workers = max_workers
      if self.save_to_db:
         self.db_conn = fn.get_db_connection(dbname='genetic_algorithm',
             user='postgres', password='changeme')

      self.plot_results = plot_results

      fn.fitness_dict = {}
```

```
def set_log_level(self, log_level=None):
   self.log_level = log_level

def set_params(self, selection_func="roulette",
        crossover_func="single point", mutation_func="single bit",
            replacement_func="generational",
        mutation_rate=0.1, tournament_size=5, crossover_window_len=0.4,
            num_individuals=0.5, truncation_proportion=1/3):

   self.fitness_func = fn.maxsat_fitness

   self.mutation_rate=mutation_rate
   self.tournament_size=tournament_size
   self.truncation_proportion = truncation_proportion
   self.crossover_window_len = crossover_window_len
   self.selection_func_str = selection_func
   self.crossover_func_str = crossover_func
   self.mutation_func_str = mutation_func

   self.replacement_func = replacement_func
   if self.steady_state_replacement:
      self.num_individuals = 1
   else:
      self.num_individuals = int(self.pop_size*num_individuals)


   initial_pop_funcs = {
      'random':fn.random_population,
   }

   initial_pop_params = {
      'random':(self.num_vars, self.set_vars, self.pop_size, self.
         allow_duplicates,),
   }

   selection_funcs = {
      'random': fn.random_selection,
      'roulette':fn.roulette_selection,
      'roulette elimination':fn.roulette_selection_with_elimination,
      'rank':fn.rank_selection,
      'tournament':fn.tournament_selection,
      'stochastic':fn.stochastic_universal_sampling_selection,
      'annealed': fn.annealed_selection,
      'truncation':fn.truncation_selection
   }

   selection_params = {
      'random':[],
      'roulette':[],
      'roulette elimination':[],
      'rank':[],
      'tournament':[tournament_size,],
```

```
    'stochastic':[],
    'annealed':[self.max_iters, 0],
    'truncation':[truncation_proportion,]
}


crossover_funcs = {
    'single point':fn.single_point_crossover,
    'two points':fn.two_point_crossover,
    'sliding window':fn.sliding_window_crossover,
    'random map':fn.random_map_crossover,
    'uniform':fn.uniform_crossover
}

crossover_params = {
    'single point':(self.ret_cost,),
    'two points':(self.ret_cost,),
    'sliding window':(self.clauses, crossover_window_len,self.ret_cost
        ,),
    'random map':(self.ret_cost,),
    'uniform':(self.ret_cost,)
}


mutation_funcs = {
    'single bit':fn.single_bit_flip,
    'multiple bit':fn.multiple_bit_flip,
    'single bit greedy':fn.single_bit_greedy,
    'single bit max greedy':fn.single_bit_max_greedy,
    'multiple bit greedy':fn.multiple_bit_greedy,
    'flip ga':fn.flip_ga
}
mutation_params = {
    'single bit':(mutation_rate,self.ret_cost,),
    'multiple bit':(mutation_rate,self.ret_cost,),
    'single bit greedy':(mutation_rate,self.clauses,self.ret_cost,),
    'single bit max greedy':(mutation_rate,self.clauses,self.ret_cost,)
        ,
    'multiple bit greedy':(mutation_rate,self.clauses,self.ret_cost,),
    'flip ga':(mutation_rate,self.clauses,self.ret_cost,)
}

self.initial_pop_func = initial_pop_funcs["random"]
self.initial_pop_params = initial_pop_params["random"]

self.selection_func = selection_funcs[selection_func]
self.sel_params = selection_params[selection_func]

self.crossover_func = crossover_funcs[crossover_func]
self.cross_params = crossover_params[crossover_func]

self.mutation_func = mutation_funcs[mutation_func]
```

```
    self.mut_params = mutation_params[mutation_func]


def get_filename(self):
    filename = self.filename.split("/")[-1].replace('.cnf','') + "_"
    if self.allow_duplicates == False:
        filename += "set_"
    if self.steady_state_replacement:
        filename += "steady_state_"
    filename += self.selection_func_str.replace(' ','_') + "_"
    filename += self.crossover_func_str.replace(' ','_') + "_"
    filename += self.mutation_func_str.replace(' ','_') + "_"
    filename += self.replacement_func.replace(' ','_') + "_"
    filename += datetime.now().strftime("%m-%d-%Y_%H:%M:%S")
    return filename

def start_ga(self):
    # Generate initial population
    cur_iter = 0
    t_fitness_evals, t_num_flips = 0,0
    max_fitness = 0

    if self.plot_distributions:
        phenotype_distributions = []
        genotype_distributions = []
    populations = []
    pop_fitness_dict = {}

    if self.log_level=="time":
        t0 = time.time()

    population = self.initial_pop_func(*self.initial_pop_params)
    pop_set = fn.get_pop_set(population)

    if self.log_level=="time":
        t1 = time.time()
        print ("Initial population function: ", t1-t0)

    if self.save_to_db:
        ga_run_id = fn.add_ga_run(conn=self.db_conn, problem=self.filename.
            split('/')[-1],
          max_iterations=self.max_iters, pop_size=len(population), elitism
              =self.elitism,
          selection_function=self.selection_func_str, crossover_function=
              self.crossover_func_str, mutation_function=self.
              mutation_func_str,
          mutation_rate=self.mutation_rate, tournament_size=self.
              tournament_size, crossover_window_len=self.
              crossover_window_len, population_replacement_function=self.
              replacement_func, num_individuals=self.num_individuals,
              truncation_proportion=self.truncation_proportion, num_clauses
              =len(self.clauses))
```

```
        fn.add_ga_run_population(conn=self.db_conn, ga_run_id=ga_run_id,
            population=population, observation="Initial population")

    while (self.sol_value==-1 and cur_iter < self.max_iters):

        if self.selection_func_str=="annealed":
            self.sel_params[1] = cur_iter

        num_fitness_evals, num_flips = 0, 0

        if self.log_level=="time":
            t0 = time.time()
        pop_fitness, max_fitness, max_fit_indiv = fn.evaluate_population(
            population, self.clauses, self.fitness_func, pop_fitness_dict,
            self.max_workers)
        for i, indiv in enumerate(population):
            if type(indiv) == type(()):
                pop_fitness_dict[indiv]=pop_fitness[i][1]
            else:
                pop_fitness_dict[tuple(indiv)]=pop_fitness[i][1]

        if self.plot_results or self.save_to_db:
            fitness_arr, genes_arr = [], []
            for x in pop_fitness:
                fitness_arr.append(x[1])
                genes_arr.append(''.join([str(y) for y in x[0]]))

            phenotype_distributions.append(fitness_arr)
            genotype_distributions.append(genes_arr)

        populations.append(population)

        num_fitness_evals += len(population) # Since we evaluated the whole
            population
        if fn.maxsat_solution_found(self.clauses, max_fitness):
            if self.save_to_db:
                fn.add_ga_run_result(self.db_conn, ga_run_id, True,
                    max_fit_indiv, cur_iter, max_fitness, t_fitness_evals,
                    t_num_flips)
                fn.add_ga_run_generation(conn=self.db_conn, ga_run_id=
                    ga_run_id, generation_num=cur_iter,
                    max_fitness=max_fitness, population_length=len(population),
                        population_set_length=len(pop_set),
                    num_fitness_evals=num_fitness_evals, num_bit_flips=
                        num_flips, num_clauses=len(self.clauses), fitness_array
                        = fitness_arr)
                fn.add_ga_run_population(conn=self.db_conn, ga_run_id=
                    ga_run_id, population=population, observation="Solution
                    found")
                fn.close_db_connection(self.db_conn)

            if self.plot_results:
```

```
            self.plot_distributions(populations, genotype_distributions,
                phenotype_distributions)
        return (True, max_fit_indiv, cur_iter, max_fitness,
            t_fitness_evals, t_num_flips)
    if self.log_level=="time":
        t1 = time.time()
        print ("Calculate population fitness: ", t1-t0)


    if self.replacement_func=="mu lambda offspring":
        num_children = len(pop_fitness)*2
        if self.steady_state_replacement:
            num_children = 4
    elif self.steady_state_replacement or self.replacement_func in ["
        parents", "weak parents"]:
        num_children = 2
    else:
        num_children = len(pop_fitness)-self.elitism

    new_children = set()
    while len(new_children) < num_children:
        # Select parents
        if self.log_level=="time":
            t0 = time.time()

        parents = self.selection_func(pop_fitness, num_children, *self.
            sel_params)

        if self.log_level=="time":
            t1 = time.time()
            print ("Selection function: ", t1-t0)


        #Generate parent pairs
        if self.log_level=="time":
            t0 = time.time()

        parent_pairs = []
        while len(parents)>=2:
            p1 = fn.get_random_int(0, len(parents))
            parent_1 = parents.pop(p1)[0]
            p2 = fn.get_random_int(0, len(parents))
            parent_2 = parents.pop(p2)[0]
            parent_pairs.append([parent_1, parent_2])

        if self.log_level=="time":
            t1 = time.time()
            print ("Generate parent pairs: ", t1-t0)

        if self.log_level=="time":
            t0 = time.time()
        # Generate children through crossover
```

```
      children = []
      for parent_pair in parent_pairs:
         prechildren = self.crossover_func(parent_pair, *self.
             cross_params)
         if self.ret_cost:
            num_fitness_evals += prechildren[1]
            num_flips += prechildren[2]
            prechildren = prechildren[0]
         children += prechildren

      if self.log_level=="time":
         t1 = time.time()
         print ("Generate children: ", t1-t0)

      if self.log_level=="time":
         t0 = time.time()
      # Mutate children
      children = fn.mutate_population(children, self.mutation_func,
          self.mut_params, self.ret_cost, self.max_workers)
      if self.ret_cost:
         num_fitness_evals += children[1]
         num_flips += children[2]
         children = children[0]
      if self.log_level=="time":
         t1 = time.time()
         print ("Mutate population: ", t1-t0)

      if self.log_level=="time":
         t0 = time.time()

      if self.allow_duplicates:
         new_children = children
      else:
         for child in children:
            if tuple(child) not in population:
               new_children.add(tuple(child))
            if len(new_children)>=len(pop_fitness)-self.elitism:
               break


   # Replace population
   if self.replacement_func=="generational":
      population = fn.generational_replacement(self.allow_duplicates,
          self.elitism, new_children, pop_fitness)
   else:
      children_fitness, _, _ = fn.evaluate_population(new_children,
          self.clauses, self.fitness_func, pop_fitness_dict, self.
          max_workers)
      for i, indiv in enumerate(new_children):
         if type(indiv) == type(()):
            pop_fitness_dict[indiv]=children_fitness[i][1]
         else:
```

```
            pop_fitness_dict[tuple(indiv)]=children_fitness[i][1]
        num_fitness_evals += len(new_children)

        if self.replacement_func=="mu lambda offspring":
            population = fn.mu_lambda_replacement(self.allow_duplicates,
                children_fitness, self.pop_size)
        elif self.replacement_func=="mu lambda":
            population = fn.mu_lambda_replacement(self.allow_duplicates,
                pop_fitness+children_fitness, self.pop_size)
        elif self.replacement_func == 'delete n':
            population = fn.delete_n(self.allow_duplicates,
                children_fitness, pop_fitness, self.num_individuals, self.
                selection_func, self.sel_params)
        elif self.replacement_func == 'random replacement':
            population = fn.random_replacement(self.allow_duplicates,
                children_fitness, pop_fitness)
        elif self.replacement_func == 'parents':
            population = fn.parent_replacement(self.allow_duplicates,
                children_fitness, parent_pairs, pop_fitness_dict,
                pop_fitness)
        elif self.replacement_func == 'weak parents':
            population = fn.weak_parent_replacement(self.allow_duplicates
                , children_fitness, parent_pairs, pop_fitness_dict,
                pop_fitness)
    if self.log_level=="time":
        t1 = time.time()
        print ("Replace population: ", t1-t0)
        print("-----------------------------------")

    cur_iter += 1
    t_fitness_evals += num_fitness_evals
    t_num_flips += num_flips

    if cur_iter % 1 == 0:
        if self.log_level=='all':
            print ("Generation {}, Max Fitness {}/{}".format(cur_iter,
                max_fitness, len(self.clauses)))
            if self.allow_duplicates:
                pop_set = fn.get_pop_set(population)
                print ("Pop Len {}, Pop Set Len {}".format(len(population),
                    len(pop_set)))
            else:
                print ("Pop Len {}".format(len(population)))
        if self.save_to_db:
            fn.add_ga_run_generation(conn=self.db_conn, ga_run_id=
                ga_run_id, generation_num=cur_iter,
                max_fitness=max_fitness, population_length=len(population),
                    population_set_length=len(pop_set),
                num_fitness_evals=num_fitness_evals, num_bit_flips=
                    num_flips, num_clauses=len(self.clauses),
                fitness_array = fitness_arr)
```

```
                fn.add_ga_run_population(conn=self.db_conn, ga_run_id=
                    ga_run_id, population=population)

        if self.save_to_db:
            fn.add_ga_run_result(self.db_conn, ga_run_id, False, None, cur_iter
                , max_fitness, t_fitness_evals, t_num_flips)
            fn.add_ga_run_generation(conn=self.db_conn, ga_run_id=ga_run_id,
                generation_num=cur_iter,
                    max_fitness=max_fitness, population_length=len(population),
                        population_set_length=len(pop_set),
                    num_fitness_evals=num_fitness_evals, num_bit_flips=
                        num_flips, num_clauses=len(self.clauses),
                    fitness_array = fitness_arr)
            fn.add_ga_run_population(conn=self.db_conn, ga_run_id=ga_run_id,
                population=population, observation="No solution")
            fn.close_db_connection(self.db_conn)

        if self.plot_results:
            self.plot_distributions(populations, genotype_distributions,
                phenotype_distributions)
        return (False, [], cur_iter, max_fitness, t_fitness_evals, t_num_flips
            )


    def plot_distributions(self, populations, genotype_distributions,
        phenotype_distributions):
        gen_distribs = fn.get_genotypic_distribution(genotype_distributions,
            max_workers=10)
        norm_gen_distrib = fn.normalize_distributions(gen_distribs, max([max(x
            ) for x in gen_distribs]))
        norm_phen_distrib = fn.normalize_distributions(phenotype_distributions
            , len(self.clauses))
        #fn.plot_violin_graph(norm_phen_distrib, "Fitness percentage", "
            Phenotypic Distributions", self.get_filename(), "phenotype")
        #fn.plot_violin_graph(norm_gen_distrib, "Mean Hamming Distance
            percentage", "Genotypic Distributions", self.get_filename(), "
            genotype")
        fn.plot_means(norm_gen_distrib, norm_phen_distrib, self.get_filename()
            )
        #fn.animate_3d_distributions(populations, self.get_filename(), "PCA")
        #fn.animate_3d_distributions(populations, self.get_filename(), "SVD")
        #fn.animate_3d_distributions(populations, self.get_filename(), "NMF")
        #fn.animate_3d_distributions(populations, self.get_filename(), "
            FactorAnalysis")
        #fn.animate_3d_distributions(populations, self.get_filename(), "
            KernelPCA")
        #fn.animate_phenotypic_distributions(norm_phen_distrib, self.
            get_filename())
        #fn.animate_genotypic_distributions(norm_gen_distrib, self.
            get_filename())

    def get_run_average(self, num_runs=10):
```

```
        succ_runs, fail_runs = 0,0
        prev_ret = self.ret_cost
        self.ret_cost = True
        total_iters, total_fitness_evals, total_flips = [],[],[]
        for i in range(num_runs):
           sol_found, sol, num_iters, fitness, num_fitness_evals, num_flips =
               self.start_ga()
           if sol_found:
              succ_runs += 1
              total_iters += [num_iters]
              total_fitness_evals += [num_fitness_evals]
              total_flips += [num_flips]

        print ("-----------Problem-----------")
        print ("Filename: ", self.filename.split('/')[-1])
        print ("Location: ", '/'.join(self.filename.split('/')[:-1])+'/')
        print ("----------Parameters---------")
        sel_str = "Selection function: "+self.selection_func_str
        if self.selection_func_str=='tournament':
           sel_str += '(tournament_size = '+str(self.sel_params[0])+')'
        print (sel_str)
        cross_str = "Crossover function: " + self.crossover_func_str
        if self.crossover_func_str=='sliding window':
           cross_str += '(window_len = '+str(self.cross_params[1])+')'
        print (cross_str)
        mut_str = "Mutation function: " + self.mutation_func_str
        if self.mutation_func_str in ['single bit', 'multiple bit']:
           mut_str += '(mutation_rate = '+str(self.mut_params[0])+')'
        print (mut_str)
        print ("------------Results----------")
        print ("Success Rate: ", succ_runs/num_runs)
        print ("Average num iterations: ", sum(total_iters)/num_runs)
        print ("Average num fitness evaulations: ", sum(total_fitness_evals)/
            num_runs)
        print ("Average num flips: ", sum(total_flips)/num_runs)
        print ("----------------------------")
        self.ret_cost = prev_ret


############## PROGRAM #############
log_levels = ['all', 'time']
initial_pop_funcs = ['random']
fitness_funcs = ['maxsat']
selection_funcs = ['random', 'roulette', 'roulette elimination', 'rank', '
    tournament', 'stochastic', 'annealed', 'truncation']
crossover_funcs = ['single point', 'two points', 'sliding window', 'random
    map', 'uniform']
mutation_funcs = ['single bit', 'multiple bit', 'single bit greedy', '
    single bit max greedy', 'multiple bit greedy', 'flip ga']
replacement_funcs = ['generational', 'mu lambda', 'mu lambda offspring', '
    delete n', 'random replacement', 'parents', 'weak parents']
```

```
# Parents and Weak Parents replacement functions use
    steady_state_replacement

best_selection_funcs = ['random', 'roulette elimination', 'rank']
best_crossover_funcs = ['two points', 'random map']
best_mutation_funcs = ['multiple bit greedy', 'flip ga']
best_replacement_funcs = ['mu lambda', 'generational', 'mu lambda offspring
    ']

best_combinations = [0,1,1,0]

filename = 'uf75-02.cnf'
foldername = './data/uf75-325'
max_iters = 100
pop_size = 1000
elitism = 0.01
num_individuals_to_replace = 0.4
mutation_rate = 0.05
crossover_window_len = 0.4
tournament_size = 5
truncation_proportion = 1/3

for filename in fn.read_folder(foldername):
      gen_alg = GeneticAlgorithm(filename=filename, max_iters=max_iters,
          pop_size=pop_size, elitism=elitism,
      steady_state_replacement = False, allow_duplicates=True, save_to_db=
          False, plot_results=False, max_workers=1)
      gen_alg.set_params(selection_func=best_selection_funcs[
          best_combinations[0]], crossover_func=best_crossover_funcs[
          best_combinations[1]],
                mutation_func=best_mutation_funcs[best_combinations[2]],
                    replacement_func=best_replacement_funcs[
                    best_combinations[3]],
                mutation_rate=mutation_rate, tournament_size=tournament_size
                    ,
                crossover_window_len=crossover_window_len, num_individuals=
                    num_individuals_to_replace,
                truncation_proportion=truncation_proportion)
      gen_alg.set_log_level('none')

      sol_found, sol, iteration, fitness, num_fitness_evals, num_flips =
          gen_alg.start_ga()

      if sol_found:
         print ("Solution found in iteration {} > {}".format(iteration, sol)
             )
         print ("Num fitness evals: {}, Num bit flips: {}".format(
             num_fitness_evals, num_flips))
         print ("s SATISFIABLE")
      else:
         print ("Solution not found in {} iterations".format(iteration))
```

```
        print ("Max fitness found: {}/{}".format(fitness, len(gen_alg.
            clauses)))
        print ("s SOLUTION NOT FOUND")
```

Listing 1: Genetic Algorithm

```
import numpy as np
import random
import sys
import psycopg2
from datetime import datetime as dt
import os
from concurrent import futures
import itertools
import seaborn as sns
import matplotlib as mpl
mpl.rc('figure', max_open_warning = 0)
import matplotlib.pyplot as plt
import glob
import re
import subprocess
import scipy.stats as sp
import hashlib
from multiprocessing import Pool
import pandas as pd
import sklearn.decomposition as sd

############## CONSTANTS #############
infinite = 2**31
levene_constant = 0.05
fp_in = "./gifs/image_*.png"
fp_out = "./gifs/"


############## FUNCTIONS #############

def get_random_int(bot, top):
   return np.random.randint(bot, top)

def read_folder(folder_path):
   files = []
   for file in os.listdir(folder_path):
      if len(file)>4 and file[-4:]=='.cnf':
         files.append(folder_path+'/'+file)
   return files

def read_problem(filepath):
   with open(filepath, 'r') as fp:
      all_clauses = ""
      for line in fp.readlines():
         if line[0]=='c':
            continue
         if line[0]=='p':
```

```
            num_vars = int(line.split()[2])
            num_clauses = int(line.split()[3])
         else:
            all_clauses += " "+line.replace('\n','')

      clauses = [[] for x in range(num_clauses)]
      clause = 0
      for num in all_clauses.split():
         if num=='0':
            clause+=1
         elif num=='%':
            break
         else:
            clauses[clause].append(int(num))

   return num_vars, clauses

def get_pop_set(population):
   unique_pop = set()
   for indiv in population:
      unique_pop.add(str(indiv))
   return unique_pop

def get_indiv_count(population, indiv):
   str_pop = [str(x) for x in population]
   return str_pop.count(indiv)

def append_to(dest_list, elem):
   if type(dest_list) == type(set()):
      dest_list.add(elem)
   elif type(dest_list) == type([]):
      dest_list.append(elem)
   return dest_list

def animate_phenotypic_distributions(population_distributions, filename):
   min_f, max_f = 2**31, 0
   for distribution in population_distributions:
      max_d, min_d = max(distribution), min(distribution)
      if max_d > max_f: max_f = max_d
      if min_d < min_f: min_f = min_d

   for i, distribution in enumerate(population_distributions):
      fig, ax = plt.subplots()
      plot = sns.distplot(distribution, bins=20, norm_hist=True, ax=ax)
      ax.set_xlim(min_f, max_f)
      #ax.set_ylim(0, 100)
      plt.title("Generation {}".format(i))
      plt.xlabel("Fitness")
      plt.ylabel("Bin frequency")

      textstr = ['variance=%.3f' % (np.var(distribution), )]
      if i>0 and np.var(distribution)>0:
```

```
            statistic, result = sp.levene(distribution,
                population_distributions[i-1])
            # if levene result (p value) < 0.05 there is a difference between
                the variance of the populations
            textstr.append(r'levene p=%.3f' % (result))
            ttest,pval = sp.ttest_ind(distribution, population_distributions[i
                -1], equal_var=result > levene_constant)
            textstr.append(r't-test p=%.3f' % (pval))
            # if pval < threshold (0.05, 0.1) reject the null hypothesis of
                equal averages

        textstr = '\n'.join(textstr)
        # these are matplotlib.patch.Patch properties0
        props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)

        # place a text box in upper left in axes coords
        ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=14,
            verticalalignment='top', bbox=props)

        plt.savefig("gifs/image_{}.png".format(i))

    file_out = fp_out + 'phenotype_' + filename + '.gif'
    filenames = sorted(glob.glob( fp_in),key=lambda x:float(re.findall("
        ([0-9]+?)\.png",x)[0]))
    subprocess.call("convert -delay 50 " + " ".join(filenames) + " -loop 5 "
         + file_out, shell=True)
    for f in sorted(glob.glob(fp_in)):
        os.remove(f)


def animate_3d_distributions(populations, filename,
    dimensionality_reduction):

    if dimensionality_reduction == "PCA":
        pca = sd.PCA(n_components=3)
    elif dimensionality_reduction == "SVD":
        pca = sd.TruncatedSVD(3)
    elif dimensionality_reduction == "NMF":
        pca = sd.NMF(n_components=3, max_iter=100000)
    elif dimensionality_reduction == "FactorAnalysis":
        pca = sd.FactorAnalysis(n_components=3)
    elif dimensionality_reduction == "KernelPCA":
        pca = sd.KernelPCA(n_components=3)

    pandas_dfs = []
    max_ind = [0,0,0]
    min_ind = [2**31,2**31,2**31]
    for i, distribution in enumerate(populations):
        principalComponents = pca.fit_transform(distribution)
        dataframe = pd.DataFrame(data = principalComponents, columns=['
            principal component 1',
            'principal component 2', 'principal component 3'])
```

```
        pandas_dfs.append(dataframe)
        if dataframe.min()[0]<min_ind[0]: min_ind[0] = dataframe.min()[0]
        if dataframe.min()[1]<min_ind[1]: min_ind[1] = dataframe.min()[1]
        if dataframe.min()[2]<min_ind[2]: min_ind[2] = dataframe.min()[2]

        if dataframe.max()[0]>max_ind[0]: max_ind[0] = dataframe.max()[0]
        if dataframe.max()[1]>max_ind[1]: max_ind[1] = dataframe.max()[1]
        if dataframe.max()[2]>max_ind[2]: max_ind[2] = dataframe.max()[2]


    for i, principalDf in enumerate(pandas_dfs):

        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.set_ylim(min_ind[1]-1, max_ind[1]+1)
        ax.set_xlim(min_ind[0]-1, max_ind[0]+1)
        ax.set_zlim(min_ind[2]-1, max_ind[2]+1)
        plt.title("Generation {}".format(i))
        ax.set_xlabel("Principal Component 1")
        ax.set_ylabel("Principal Component 2")
        ax.set_zlabel("Principal Component 3")

        ax.scatter(principalDf['principal component 1'],principalDf['principal
            component 2'],
            principalDf['principal component 3'], c='r', marker='o')

        plt.savefig("gifs/image_{}.png".format(i))

    file_out = fp_out + dimensionality_reduction + '_analysis_' + filename +
        '.gif'
    filenames = sorted(glob.glob( fp_in),key=lambda x:float(re.findall("
        ([0-9]+?)\.png",x)[0]))
    subprocess.call("convert -delay 50 " + " ".join(filenames) + " -loop 5 "
        + file_out, shell=True)
    for f in sorted(glob.glob(fp_in)):
        os.remove(f)

def plot_violin_graph(pop_distributions, y_axis, title, filename, dist_type
    ):
    fig, ax = plt.subplots(figsize=(30,15))
    ax.violinplot(pop_distributions, showmeans=True, widths=0.7)
    plt.title(title)
    plt.xlabel("Generation number")
    plt.ylabel(y_axis)
    file_out = fp_out + dist_type + '_' + filename + '.png'
    plt.savefig(file_out)

def plot_means(genotype_distributions, phenotype_distributions, filename):
    fig, ax = plt.subplots(figsize=(30,15))

    ind_arr = [x for x in range(len(genotype_distributions))]
```

```
    gen_mean, gen_dev = get_mean_distributions(genotype_distributions)
    gen_lower, gen_upper = [], []
    for i in range(len(gen_mean)):
        gen_lower += [gen_mean[i]-gen_dev[i]]
        gen_upper += [gen_mean[i]+gen_dev[i]]

    phen_mean, phen_dev = get_mean_distributions(phenotype_distributions)
    phen_lower, phen_upper = [], []
    for i in range(len(phen_mean)):
        phen_lower += [phen_mean[i]-phen_dev[i]]
        phen_upper += [phen_mean[i]+phen_dev[i]]

    ax.fill_between(ind_arr, gen_upper, gen_lower, color="lightcyan")
    ax.plot(ind_arr, gen_mean, color="blue", lw=2, label="Genotype mean")
    #plt.plot(ind_arr, gen_mean, color ='g')

    ax.fill_between(ind_arr, phen_upper, phen_lower, color="lightyellow",
        alpha=0.7)
    ax.plot(ind_arr, phen_mean, color="brown", lw=2, label="Phenotype mean")

    ax.set_title("Genotypic vs Phenotypic mean evolution")
    ax.set_xlabel("Generation number")
    ax.set_ylabel("Mean percentage")
    ax.legend()
    file_out = fp_out + 'bbands_' + filename + '.png'
    plt.savefig(file_out)

def get_mean_distributions(pop_distributions):
    mean_dists = []
    deviation_dists = []
    for distrib in pop_distributions:
        mean_dists += [np.mean(distrib)]
        deviation_dists += [np.std(distrib)]
    return mean_dists, deviation_dists

def normalize_distributions(pop_distributions, max_value):
    normalized_dists = []
    for distribution in pop_distributions:
        new_dist = []
        for indiv in distribution:
            new_dist += [indiv/max_value]
        normalized_dists += [new_dist]
    return normalized_dists


def hamming_distance(chain1, chain2):
    chaine1 = hashlib.md5(chain1.encode()).hexdigest()
    chaine2 = hashlib.md5(chain2.encode()).hexdigest()
    return sum(c1 != c2 for c1, c2 in zip(chaine1, chaine2))

def get_hamming_distances(population, hamming_dict=None):
    pop = []
```

```
    for indiv in population:
        dist = 0
        for indiv2 in population:
            if indiv!=indiv2:
                if hamming_dict != None:
                    try:
                        ld = hamming_dict[tuple(indiv)+tuple(indiv2)]
                    except:
                        ld = hamming_distance(indiv, indiv2)
                        hamming_dict[tuple(indiv)+tuple(indiv2)] = ld
                else:
                    ld = hamming_distance(indiv, indiv2)
                dist += ld
        dist /= len(population)
        pop.append(dist)

    if hamming_dict != None:
        return pop, hamming_dict
    return pop


def get_genotypic_distribution(pop_distributions, max_workers=1):
    genotype_distributions = []

    if max_workers == 1:
        hamming_dict = {}
        for population in pop_distributions:
            pop, hamming_dict = get_hamming_distances(population, hamming_dict)
            genotype_distributions.append(pop)

    else:
        for i in range(0,len(pop_distributions), max_workers):
            p = Pool(processes=max_workers)
            data = p.map(get_hamming_distances, pop_distributions[i:i+10])
            p.close()
            genotype_distributions += data

    return genotype_distributions

def animate_genotypic_distributions(genotype_distributions, filename):
    min_f, max_f = 2**31, 0
    for pop in genotype_distributions:
        tmin = min(pop)
        tmax = max(pop)
        if tmin < min_f: min_f = tmin
        if tmax > max_f: max_f = tmax

    for i, distribution in enumerate(genotype_distributions):
        fig, ax = plt.subplots()
        plot = sns.distplot(distribution, bins=20, ax=ax)
        ax.set_xlim(min_f,max_f)
        #ax.set_ylim(0, len(distribution))
```

```
            plt.title("Generation {}".format(i))
            plt.xlabel("Hamming distance")
            plt.ylabel("Bin frequency")

            textstr = ['variance=%.3f' % (np.var(distribution), )]
            if i>0 and np.var(distribution)>0:
                statistic, result = sp.levene(distribution, genotype_distributions[
                    i-1])
                # if levene result (p value) < 0.05 there is a difference between
                    the variance of the populations
                textstr.append(r'levene p=%.3f' % (result))
                ttest,pval = sp.ttest_ind(distribution, genotype_distributions[i
                    -1], equal_var=result > levene_constant)
                textstr.append(r't-test p=%.3f' % (pval))
                # if pval < threshold (0.05, 0.1) reject the null hypothesis of
                    equal averages

            textstr = '\n'.join(textstr)
            # these are matplotlib.patch.Patch properties0
            props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)

            # place a text box in upper left in axes coords
            ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=14,
                verticalalignment='top', bbox=props)
            plt.savefig("gifs/image_{}.png".format(i))

        file_out = fp_out + 'genotype_' + filename + '.gif'
        filenames = sorted(glob.glob( fp_in),key=lambda x:float(re.findall("
            ([0-9]+?)\.png",x)[0]))
        subprocess.call("convert -delay 50 " + " ".join(filenames) + " -loop 5 "
             + file_out, shell=True)
        for f in sorted(glob.glob(fp_in)):
            os.remove(f)



############## DATABASE HELPER FUNCTIONS #############

def get_db_connection(dbname='genetic_algorithm', user='pyuser', password='
    123456'):
    conn, cursor = None, None
    try:
        conn = psycopg2.connect(database=dbname, user=user, password=password,
             host='192.168.1.4', port=5432)
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)

    return conn

def close_db_connection(conn):
    try:
        if conn is not None:
```

```
        conn.close()
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)

def add_ga_run(conn, problem, max_iterations, pop_size, elitism,
    selection_function, crossover_function,
    mutation_function, mutation_rate, tournament_size, crossover_window_len,
        population_replacement_function,
    num_individuals, truncation_proportion, num_clauses):
    new_id = -1
    try:
        with conn:
            cursor = conn.cursor()
            sql_string = "INSERT INTO ga_run (created_timestamp, problem,
                max_iterations, pop_size, elitism, \
                    selection_function, crossover_function, mutation_function,\
                    population_replacement_function, mutation_rate,
                        tournament_size, crossover_window_len, \
                    num_individuals, truncation_proportion, num_clauses) VALUES \
                    (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
                        RETURNING id;"
            cursor.execute(sql_string, (dt.now(), problem, max_iterations,
                pop_size, elitism, selection_function, crossover_function,
                mutation_function, population_replacement_function,
                mutation_rate, tournament_size, crossover_window_len,
                num_individuals, truncation_proportion, num_clauses))
            new_id = cursor.fetchone()[0]

    except (Exception, psycopg2.DatabaseError) as error:
        print (error)

    return new_id

def add_ga_run_result(conn, ga_run_id, sol_found, solution, num_iterations,
     max_fitness, num_fitness_evals, num_bit_flips):
    try:
        with conn:
            cursor = conn.cursor()
            sql_string = "UPDATE ga_run SET updated_timestamp = %s, sol_found =
                 %s, solution = %s, \
                num_iterations = %s, max_fitness = %s, num_fitness_evals = %s,
                    num_bit_flips = %s WHERE id = %s"
            cursor.execute(sql_string, (dt.now(), sol_found, solution,
                num_iterations,
                max_fitness, num_fitness_evals, num_bit_flips, ga_run_id))
    except (Exception, psycopg2.DatabaseError) as error:
        print (error)

def add_ga_run_generation(conn, ga_run_id, generation_num, max_fitness,
    population_length, population_set_length, num_fitness_evals,
    num_bit_flips, num_clauses, fitness_array):
    mean = float(np.mean(fitness_array))
```

```
    median = float(np.median(fitness_array))
    mode = sp.mode(fitness_array)
    mode_value = float(mode[0][0])
    mode_count = int(mode[1][0])
    deviation = float(np.std(fitness_array))
    try:
        with conn:
            cursor = conn.cursor()
            sql_string = "INSERT INTO ga_run_generations (time_stamp, ga_run_id
                , generation_num, max_fitness, \
                     population_length, population_set_length,
                         num_fitness_evals, num_bit_flips, num_clauses, \
                     mean, median, mode, mode_count, standard_deviation) VALUES
                          \
                     (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)"
            cursor.execute(sql_string, (dt.now(), ga_run_id, generation_num,
                max_fitness, population_length,
                       population_set_length, num_fitness_evals, num_bit_flips,
                           num_clauses, mean, median, mode_value,
                       mode_count, deviation))
    except (Exception, psycopg2.DatabaseError) as error:
        print (error)

def add_ga_run_population(conn, ga_run_id, population, ga_run_generation_id
    =None, observation=None):
    try:
        with conn:
            cursor = conn.cursor()
            sql_string = "INSERT INTO ga_run_population (ga_run_id,
                ga_run_generation_id, population, observations) VALUES (%s, %s,
                 %s, %s)"
            cursor.execute(sql_string, (ga_run_id, ga_run_generation_id,
                population, observation))
    except (Exception, psycopg2.DatabaseError) as error:
        print (error)


############## CNF SIMPLIFICATION ##############

def trivial_case(clauses):
    num_pos = 0
    num_neg = 0
    for clause in clauses:
        for num in clause:
            if num > 0: num_pos += 1
            else: num_neg += 1
    if num_neg == 0:
        return 1
    if num_pos == 0:
        return 0
    return -1
```

```
def remove_unit_vars(clauses, set_vars):
   """
      remove_unit_vars

      Finds clauses with a single variable in them
      and sets the variable so the clause equals True
   """
   unit_clauses = 0
   new_clauses = clauses[:]
   for i, clause in enumerate(clauses):
      if len(clause)==1:
         # Clause found
         unit_clauses += 1
         #new_clauses = clauses[:i]+clauses[i+1:]
         new_clauses.pop(i)
         # Set Variable
         if clause[0]>=0: set_vars[clause[0]-1] = 1
         else: set_vars[abs(clause[0])-1] = 0
         for j, new_clause in enumerate(new_clauses):
            if 0-clause[0] in new_clause:
               # Remove negative value from clauses since its False
               new_clauses[j].remove(0-clause[0])
         for new_clause in clauses:
            if clause[0] in new_clause:
               if new_clause in new_clauses:
                  # Remove clause since its solved
                  new_clauses.remove(new_clause)
         break
   if unit_clauses==0:
      return new_clauses, set_vars
   else:
      return remove_unit_vars(new_clauses, set_vars)

def remove_pure_vars(clauses, set_vars):
   """
      remove_pure_vars

      For every variable it searches through all the clauses,
      if the variable only appears in negated form, it sets it to False
      if the variable only appears in "positive" form, it sets it to True
      if the variable doesn't appear in the clauses, any value will do
   """
   new_clauses = clauses[:]
   for i in range(len(set_vars)):
      pos_var, neg_var = 0, 0
      for clause in clauses:
         if i+1 in clause: pos_var += 1
         elif -(i+1) in clause: neg_var += 1
      if pos_var > 0 and neg_var == 0:
         set_vars[i] = 1
      elif neg_var > 0 and pos_var == 0:
         set_vars[i] = 0
```

```
      elif neg_var == 0 and pos_var == 0:
         # Any value will do, since the variable doesn't appear in the
             formulas
         set_vars[i] = 0
      if set_vars[i] != infinite:
         for clause in clauses:
            if i+1 in clause:
               new_clauses.remove(clause)
            elif -(i+1) in clause:
               new_clauses.remove(clause)
   return new_clauses, set_vars

############## GENERATE INITIAL POPULATION #############

def random_population(num_vars, set_vars, pop_size, allow_duplicates):
   if allow_duplicates:
      population = []
   else:
      population = set()
   while(len(population)<pop_size):
      for p in range(pop_size):
         rpop = np.random.randint(2, size=num_vars)
         for j, var in enumerate(set_vars):
            if var != infinite:
               rpop[j] = var
         if allow_duplicates:
            population.append(rpop.tolist())
         else:
            population.add(tuple(rpop.tolist()))
   return population

def satisfy_clauses_population(num_vars, set_vars, pop_size, clauses,
    allow_duplicates):
   #NOT USED
   if allow_duplicates:
      population = []
   else:
      population = set()
   while len(population)<pop_size:
      for p in range(pop_size):
         indiv = [0]*num_vars
         for clause in clauses:
            rind = np.random.randint(len(clause))
            if clause[rind]>=0: indiv[clause[rind]-1] = 1
            else: indiv[abs(clause[rind])-1] = 0
         if allow_duplicates:
            population.append(indiv)
         else:
            population.add(indiv)
   return population

############## EVALUATE POPULATION #############
```

```
def evaluate_population(population, clauses, fitness_function, fitness_dict
    , max_workers=1):
   pop_fitness = []
   max_fitness = 0
   max_fit_indiv = None
   if max_workers == 1:
      for pop in population:
         try:
            if type(pop)==type(()):
               fitness = fitness_dict[pop]
            else:
               fitness = fitness_dict[tuple(pop)]
         except:
            fitness = fitness_function(clauses, pop)
         if fitness >= max_fitness:
            max_fitness = fitness
            max_fit_indiv = pop
         pop_fitness.append([pop, fitness])
   else:
      ex = futures.ProcessPoolExecutor(max_workers=max_workers)
      results = ex.map(fitness_function, itertools.repeat(clauses, len(
          population)), population)
      for i, fitness in enumerate(results):
         pop_fitness.append([population[i], fitness])
         if fitness >= max_fitness:
            max_fitness = fitness
            max_fit_indiv = population[i]
   return pop_fitness, max_fitness, max_fit_indiv


############## FITNESS FUNCTIONS #############
fitness_dict = {}
def maxsat_fitness(clauses, var_arr):
   # Since CNF formulas are of the shape (x1 OR x2 OR x3) AND (x3 OR -x2 OR
       -x1)
   # As soon as we find any True value inside a clause that clause is
       satisfied
   try:
      x = fitness_dict[str(var_arr)]
   except:
      t_clauses = 0
      for clause in clauses:
         for num in clause:
            if num >= 0:
               if var_arr[num-1]==1:
                  t_clauses += 1
                  break
            elif num <= 0:
               if var_arr[abs(num)-1]==0:
                  t_clauses += 1
                  break
      fitness_dict[str(var_arr)] = t_clauses
```

```
   return fitness_dict[str(var_arr)]

def float_fitness(clauses, var_arr):
   #NOT USED
   t_res = 1
   for clause in clauses:
      tmp_r = 0
      for num in clause:
         if num<0:
            if var_arr[abs(num)-1] == 0:
               tmp_r += 1
            else:
               tmp_r += 0
         else:
            tmp_r += var_arr[num-1]
      #if tmp_r >= 1: tmp_r = 1
      t_res *= tmp_r
   return t_res

def maxsat_solution_found(clauses, fitness):
   if fitness >= len(clauses): return True
   return False

############## SELECTION FUNCTIONS #############

def random_selection(population, num_parents):
   tmp_pop = population[:]
   parents = []
   for _ in range(num_parents):
      if len(tmp_pop)==0: tmp_pop = population[:]
      ind = np.random.randint(0, len(tmp_pop))
      parents.append(tmp_pop.pop(ind))
   return parents

def roulette_selection_with_elimination(population, num_parents):
   tmp_pop = population[:]
   parents = []
   for _ in range(num_parents):
      if len(tmp_pop)==0: tmp_pop = population[:]
      total_fitness = sum([y for x,y in tmp_pop])
      probabilities = [y/total_fitness if total_fitness>0 else 0 for x,y in
          tmp_pop]
      if total_fitness>0:
         rnum = random.random()
      else:
         rnum = 0
      sprob = 0
      for i, prob in enumerate(probabilities):
         if rnum >= sprob and rnum <= sprob+prob:
            parents.append(tmp_pop.pop(i))
            break
         sprob += prob
```

```
    return parents

def roulette_selection(population, num_parents):
    parents = []
    total_fitness = sum([y for x,y in population])
    probabilities = [y/total_fitness for x,y in population]
    for _ in range(num_parents):
        rnum = random.random()
        sprob = 0
        for i, prob in enumerate(probabilities):
            if rnum >= sprob and rnum <= sprob+prob:
                parents.append(population[i])
                break
            sprob += prob
    return parents

def rank_selection(population, num_parents):
    parents = []
    sorted_pop = sorted(population, key=lambda x: x[1])
    pop_rank = [[x, i+1] for i,x in enumerate(sorted_pop)]
    total_rank = sum([y for x,y in pop_rank])
    probabilities = [y/total_rank for x,y in pop_rank]
    for _ in range(num_parents):
        rnum = random.random()
        sprob = 0
        for i, prob in enumerate(probabilities):
            if rnum >= sprob and rnum <= sprob+prob:
                parents.append(pop_rank[i][0])
                break
            sprob += prob
    return parents

def truncation_selection(population, num_parents, proportion):
    num_indivs = int(len(population)*proportion)
    parents = []
    sorted_pop = sorted(population, key=lambda x: x[1])
    if num_indivs >= num_parents:
        for i in range(num_parents):
            parents.append(sorted_pop[i])
    elif num_indivs < num_parents:
        while len(parents)<num_parents:
            for i in range(num_indivs):
                parents.append(sorted_pop[i])
                if len(parents)>=num_parents:
                    break
    return parents



def tournament_selection(population, num_parents, tournament_size):
    parents = []
    for i in range(num_parents):
```

```
        twinner, tfitness = 0, 0
        for i in range(tournament_size):
            rnum = np.random.randint(len(population))
            if population[rnum][1] > tfitness:
                twinner = population[rnum][0]
                tfitness = population[rnum][1]
        parents.append([twinner,tfitness])
    return parents


def stochastic_universal_sampling_selection(population, num_parents):
    total_fitness = sum([x[1] for x in population])
    point_distance = total_fitness/num_parents
    start_point = int(np.random.uniform(0, point_distance))
    points = [start_point + i * point_distance for i in range(num_parents)]

    parents = []
    while len(parents) < num_parents:
        random.shuffle(population)
        fit_sum, point = 0, 0
        for indiv in population:
            if len(parents)==num_parents:
                break
            elif fit_sum+indiv[1]>points[point]:
                point+=1
                parents.append(indiv)
            fit_sum += indiv[1]
    return parents


def annealed_selection(population, num_parents, max_generations,
    generation_num):
    # Combination if rank and roulette wheel selection
    # Rank selection
    sorted_pop = sorted(population, key=lambda x: x[1])
    pop_rank = [[x, i+1] for i,x in enumerate(sorted_pop)]
    total_rank = int((pop_rank[-1][1]*(pop_rank[-1][1]-1))/2)
    # Roulette wheel selection
    total_fitness = sum([x[1] for x in sorted_pop])
    # Combination
    factor = (1/max_generations)*generation_num
    annealed_pop = []
    total_a_fitness = 0
    for i,x in enumerate(pop_rank):
        annealed_fitness = (sorted_pop[i][1]/total_fitness)*(1-factor)+(x[1]/
            total_rank)*(0+factor)
        annealed_pop.append([x[0], annealed_fitness])
        total_a_fitness += annealed_fitness
    parents = []

    while len(parents)<num_parents:
        rnum = np.random.uniform(0, total_a_fitness)
```

```
      fsum = 0
      for indiv in annealed_pop:
         if fsum+indiv[1]>=rnum:
            parents.append(indiv[0])
            break
         fsum += indiv[1]

   return parents


############## CROSSOVER FUNCTIONS #############

def single_point_crossover(parent_pair, ret_cost=False):
   parent1, parent2 = parent_pair
   fitness_evals, bit_flips = 0, 0
   cut_point = np.random.randint(len(parent1))
   children = [
      np.concatenate((parent1[:cut_point],parent2[cut_point:])).tolist(),
      np.concatenate((parent2[:cut_point],parent1[cut_point:])).tolist()
      ]
   if ret_cost:
      return children, fitness_evals, bit_flips
   else:
      return children

def two_point_crossover(parent_pair, ret_cost=False):
   parent1, parent2 = parent_pair
   fitness_evals, bit_flips = 0, 0
   cut_point_1 = np.random.randint(len(parent1)-1)
   cut_point_2 = np.random.randint(cut_point_1+1, len(parent1))
   children = [
      np.concatenate((parent1[:cut_point_1],parent2[cut_point_1:cut_point_2
          ],parent1[cut_point_2:])).tolist(),
      np.concatenate((parent2[:cut_point_1],parent1[cut_point_1:cut_point_2
          ],parent2[cut_point_2:])).tolist()
      ]
   if ret_cost:
      return children, fitness_evals, bit_flips
   else:
      return children

def sliding_window_crossover(parent_pair, clauses, crossover_window_len
    =0.4, ret_cost=False):
   parent1, parent2 = parent_pair
   fitness_evals, bit_flips = 0, 0
   window_len = int(crossover_window_len*len(parent1))
   max_fitness, max_i = [0,0], [0,0]
   bad_children = [[],[]]
   for i in range(len(parent1)-window_len):
      bad_children[0].append(np.concatenate((parent1[:i],parent2[i:i+
          window_len],parent1[i+window_len:])).tolist())
```

```
            bad_children[1].append(np.concatenate((parent2[:i],parent1[i:i+
                window_len],parent2[i+window_len:])).tolist())
            for t in range(2):
                fitness = maxsat_fitness(clauses, bad_children[t][i])
                fitness_evals += 1
                if fitness >= max_fitness[0]:
                    max_fitness[t] = fitness
                    max_i[t] = i
        if ret_cost:
            return [bad_children[0][max_i[0]], bad_children[1][max_i[1]]],
                fitness_evals, bit_flips
        else:
            return [bad_children[0][max_i[0]], bad_children[1][max_i[1]]]

    def random_map_crossover(parent_pair, ret_cost=False):
        parent1, parent2 = parent_pair
        fitness_evals, bit_flips = 0, 0
        rand_map = np.random.randint(2, size=len(parent1))
        child_1 = parent1[:]
        child_2 = parent2[:]
        for i, elem in enumerate(rand_map):
            if elem == 0:
                child_1[i] = parent2[i]
                child_2[i] = parent1[i]
                bit_flips += 2
        if ret_cost:
            return [child_1, child_2], fitness_evals, bit_flips
        else:
            return [child_1, child_2]

    def uniform_crossover(parent_pair, ret_cost=False):
        parent1, parent2 = parent_pair
        # Uses alternating bits, maybe change to a normal distribution
        fitness_evals, bit_flips = 0, 0
        child_1 = parent1[:]
        child_2 = parent2[:]
        for i in range(len(parent1)):
            if i%2==0:
                child_1[i] = parent2[i]
                child_2[i] = parent1[i]
                bit_flips += 2
        if ret_cost:
            return [child_1, child_2], fitness_evals, bit_flips
        else:
            return [child_1, child_2]

    ############## MUTATION FUNCTIONS #############

    def mutate_population(population, mutation_function, mutation_params,
        ret_cost=False, max_workers=1):
        fitness_evals, bit_flips = 0, 0
        new_pop = []
```

```
    if max_workers == 1:
        for i, pop in enumerate(population):
            newindiv = mutation_function(pop, *mutation_params)
            if ret_cost:
                fitness_evals += newindiv[1]
                bit_flips += newindiv[2]
                newindiv = newindiv[0]
            new_pop.append(newindiv)
    else:
        ex = futures.ProcessPoolExecutor(max_workers=max_workers)
        results = ex.map(mutation_function, population, itertools.repeat(
            mutation_params[0], len(population)), itertools.repeat(
            mutation_params[1], len(population)))
        if ret_cost:
            for indiv, fit_ev, bit_f in results:
                new_pop.append(indiv)
                fitness_evals += fit_ev
                bit_flips += bit_f
        else:
            new_pop=results

    if ret_cost:
        return new_pop, fitness_evals, bit_flips
    else:
        return new_pop


def single_bit_flip(individual, mutation_rate, ret_cost=False):
    fitness_evals, bit_flips = 0, 0
    indiv = individual
    rmut = random.random()
    if rmut <= mutation_rate:
        ind = np.random.randint(len(indiv))
        if indiv[ind] == 1: indiv[ind] = 0
        else: indiv[ind] = 1
        bit_flips += 1
    if ret_cost:
        return indiv, fitness_evals, bit_flips
    else:
        return indiv

def multiple_bit_flip(individual, mutation_rate, ret_cost=False):
    fitness_evals, bit_flips = 0, 0
    indiv = individual
    rmut = random.random()
    if rmut <= mutation_rate:
        num_bits = np.random.randint(len(indiv))
        for x in range(num_bits):
            ind = np.random.randint(len(indiv))
            if indiv[ind] == 1: indiv[ind] = 0
            else: indiv[ind] = 1
            bit_flips += 1
```

```
    if ret_cost:
        return indiv, fitness_evals, bit_flips
    else:
        return indiv

def single_bit_greedy(individual, mutation_rate, clauses, ret_cost=False):
    fitness_evals, bit_flips = 0, 0
    ind_fitness = maxsat_fitness(clauses, individual)
    fitness_evals += 1
    max_ind = individual
    rmut = random.random()
    if rmut <= mutation_rate:
        for j in range(len(individual)):
            indiv = individual[:]
            if indiv[j]==1: indiv[j]=0
            elif indiv[j]==0: indiv[j]=1
            bit_flips += 1
            fitness_evals += 1
            if maxsat_fitness(clauses, indiv)>ind_fitness:
                max_ind = indiv
                break
    if ret_cost:
        return max_ind, fitness_evals, bit_flips
    else:
        return max_ind

def single_bit_max_greedy(individual, mutation_rate, clauses, ret_cost=
    False):
    fitness_evals, bit_flips = 0, 0
    ind_fitness = maxsat_fitness(clauses, individual)
    fitness_evals += 1
    max_ind, max_fit = individual, ind_fitness
    rmut = random.random()
    if rmut <= mutation_rate:
        for j in range(len(individual)):
            t_indiv = individual[:]
            if t_indiv[j]==1: t_indiv[j]=0
            elif t_indiv[j]==0: t_indiv[j]=1
            tfit = maxsat_fitness(clauses, t_indiv)
            fitness_evals += 1
            bit_flips += 1
            if tfit>max_fit:
                max_ind = t_indiv
                max_fit = tfit

    if ret_cost:
        return max_ind, fitness_evals, bit_flips
    else:
        return max_ind

def multiple_bit_greedy(individual, mutation_rate, clauses, ret_cost=False)
    :
```

```
    fitness_evals, bit_flips = 0, 0
    ind_fitness = maxsat_fitness(clauses, individual)
    indiv = individual[:]
    rmut = random.random()
    if rmut <= mutation_rate:
        for j in range(len(indiv)):
            t_indiv = indiv[:]
            new_bit = 1
            if indiv[j]==1: new_bit=0
            t_indiv[j] = new_bit
            t_fitness = maxsat_fitness(clauses, t_indiv)
            fitness_evals += 1
            bit_flips += 1
            if t_fitness > ind_fitness:
                ind_fitness = t_fitness
                indiv = t_indiv
    if ret_cost:
        return indiv, fitness_evals, bit_flips
    else:
        return indiv

def flip_ga(individual, mutation_rate, clauses, ret_cost=False):
    fitness_evals, bit_flips = 0, 0
    indiv = individual[:]
    ind_fitness = maxsat_fitness(clauses, indiv)
    prev_fitness = ind_fitness-1
    rmut = random.random()
    if rmut <= mutation_rate:
        while(prev_fitness<ind_fitness):
            prev_fitness = ind_fitness
            for j in range(len(indiv)):
                t_indiv = indiv[:]
                new_bit = 1
                if t_indiv[j]==1: new_bit=0
                t_indiv[j] = new_bit
                t_fitness = maxsat_fitness(clauses, t_indiv)
                fitness_evals += 1
                bit_flips += 1
                if t_fitness > ind_fitness:
                    ind_fitness = t_fitness
                    indiv = t_indiv
    if ret_cost:
        return indiv, fitness_evals, bit_flips
    else:
        return indiv


############## POPULATION REPLACEMENT FUNCTIONS #############
def random_replacement(allow_duplicates, children_fitness, pop_fitness):
    new_pop = [x for x,y in pop_fitness]
    for _ in children_fitness:
        rand_ind = np.random.randint(0, len(new_pop))
```

```
            new_pop.pop(rand_ind)
        new_pop += [x for x,y in children_fitness]
        if allow_duplicates:
            return new_pop
        else:
            return set(new_pop)


    def weak_parent_replacement(allow_duplicates, children_fitness,
        parent_pairs, pop_fitness_dict, pop_fitness):
        new_pop = [x for x,y in pop_fitness]
        tmp_children = list(children_fitness)
        new_children = []
        for ppair in parent_pairs:
            new_pop.remove(ppair[0])
            aux = tmp_children[:2] + [[ppair[0],pop_fitness_dict[tuple(ppair[0])
                ]], [ppair[1],pop_fitness_dict[tuple(ppair[1])]]]
            tmp_children = tmp_children[2:]
            sorted_pop = sorted(aux, key=lambda x: x[1], reverse=True)
            if ppair[0]==ppair[1]:
                new_children += [x for x,y in sorted_pop[:1]]
            else:
                new_pop.remove(ppair[1])
                new_children += [x for x,y in sorted_pop[:2]]
        new_pop += new_children
        if allow_duplicates:
            return new_pop
        else:
            return set(new_pop)

    def parent_replacement(allow_duplicates, children_fitness, parent_pairs,
        pop_fitness_dict, pop_fitness):
        new_pop = [x for x,y in pop_fitness]
        remove_children = 0
        for ppair in parent_pairs:
            try:
                new_pop.remove(ppair[0])
            except: pass
            try:
                new_pop.remove(ppair[1])
            except: pass
        new_pop += [x for x,y in sorted(children_fitness, key=lambda x: x[1],
            reverse=True)][:len(pop_fitness)-len(new_pop)]
        if allow_duplicates:
            return new_pop
        else:
            return set(new_pop)

    def delete_n(allow_duplicates, children_fitness, pop_fitness,
        num_individuals, selection_method, selection_params):
        tmp_pop = pop_fitness[:]
        tmp_children = children_fitness[:]
```

```
    new_children = []
    for _ in range(num_individuals):
        to_kill = selection_method(tmp_pop, 1, *selection_params)[0]
        if len(tmp_children)>0:
            tmp_pop.remove(to_kill)
            to_add = selection_method(tmp_children, 1, *selection_params)[0]
            tmp_children.remove(to_add)
            new_children += [to_add]
        else:
            break

    new_pop = [x for x,y in tmp_pop] + [x for x,y in new_children]
    if allow_duplicates:
        return new_pop
    else:
        return set(new_pop)

def mu_lambda_replacement(allow_duplicates, pop_fitness, pop_size):
    new_population = []
    sorted_pop = sorted(pop_fitness, key=lambda x: x[1], reverse=True)
    new_population = [x for x,y in sorted_pop[:pop_size]]
    if not allow_duplicates: new_population = set(new_population)
    return new_population

def generational_replacement(allow_duplicates, elitism, children,
    pop_fitness):
    if elitism == 0:
        return children

    if allow_duplicates:
        new_population = []
    else:
        new_population = set()
    sorted_pop = sorted(pop_fitness, key=lambda x: x[1], reverse=True)
    if allow_duplicates:
        new_population = children + [x for x,y in sorted_pop[:elitism]]
    else:
        new_population = children
        for x,y in sorted_pop[:elitism]:
            new_population.add(x)
    return new_population
```

Listing 2: Auxiliary Functions