Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Application of Genetic Algorithms for the optimization of combinatorial problems

DEGREE FINAL WORK

Degree in Computer Engineering

Author: Juan Francisco Ariño Sales
Tutor: Federico Barber Sanchís

Course 2020-2021

Abstract

????

Key words: ?????, ????? ?????, ?????????????

Contents

List of Figures

List of Tables

Abstract

????

Key words: ?????, ????? ?????, ????????????

Contents

List of Figures

List of Tables

Abstract

????

Key words: ?????, ????? ?????, ?????????????

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

1.1 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 expression exists; 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.

1.1.1. 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 **??** in order to carry out the main objective.

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

1.1.2. Outline

This work is organized as follows:

Chapter **??** provides an overview of the background information necessary to understand this work, among such background 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 **??**, **??** and **??** respectively, there is also a basic explanation for how genetic algorithms work in Section **??**.
The SAT-solvers which will be compared with the genetic algorithms are explained in Chapter **??**, Section **??**
Chapter **??** 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 **??** explains the specific details of the design of the *genetic_algorithm* class and the functionality of the developed implementation in Section **??**, the database structure used to log the results of the experiments is shown in Section **??**. (TODO: Describe Analysis and Conclusions chapters)

CHAPTER 2
Preliminaries

2.1 Preliminaries

2.1.1. 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.
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 (Karp 2010)

2.1.2. 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, invented in 1936 by mathematician Alan Turing, a Turing machine is a theoretical machine which consists of four parts:

- An infinite *tape* divided into cells
- A *head* that can write and read symbols on the tape and can also move left or right along the tape
- A *state register* that stores the state of the Turing machine
- A finite table of *instructions*, which tell the head what to do based on the state and the symbols on the tape

The machine then operates on the tape, its behavior is defined in the instructions table where instructions are indexed by state and symbol, there are three types of possible instructions:

- Erase or write a symbol on the tape
- Move the head left or right one cell
- Change state.

The machine will begin by reading the first symbol on the tape and then executing instructions until it halts.
A Turing machine 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*, for each combination of state and symbol in the instructions table there exists only one instruction.

- *Non-deterministic Turing machines*, for each combination of state and symbol there can exist more than one instruction in the instructions table, therefore it is non-deterministic since it is not possible to know exactly the next state of the machine based on the current state and the tape symbol.

Though a non-deterministic Turing machine can be fully replicated using a deterministic Turing machine, it is nonetheless a useful abstraction because it allows us to generate computation trees with many branches, and as long as any one of these branches reaches an accepting state, the machine will halt and accept, it will only reject once all the branches have reached their limit.
This is in contrast with deterministic Turing machines where the computation tree is a single branch with nodes following each other sequentially; A non-deterministic Turing machine can be recreated using a deterministic Turing machine by exploring the whole computation tree generated by the non-deterministic machine using tree traversal algorithms, but the resulting machine is much more convoluted and difficult to reason about.

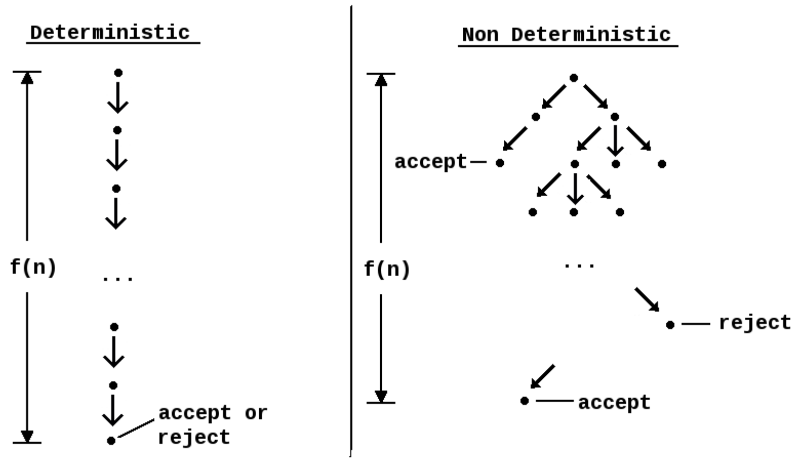

Figure 2.1: Left: Deterministic Turing machine computation tree. Right: Non-deterministic Turing machine computation tree

Different 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

- $NSPACE[s(n)]$ 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:

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

The canonical complexity classes are defined as follows:

- $L = DSPACE[logn]$ (logarithmic space)
- $NL = NSPACE[logn]$ (logarithmic space)
- $P = DTIME[n^{O(1)}] = \bigcup_{k \in \mathbb{N}} DTIME[n^k]$ (polynomial time)
- $NP = NTIME[n^{O(1)}] = \bigcup_{k \in \mathbb{N}} NTIME[n^k]$ (polynomial time)
- $PSPACE = DSPACE[n^{O(1)}] = \bigcup_{k \in \mathbb{N}} DSPACE[n^k]$ (polynomial space)
- $E = DTIME[2^{O(n)}] = \bigcup_{k \in \mathbb{N}} DTIME[k^n]$ (exponential time, linear exponent)
- $NE = NTIME[2^{O(n)}] = \bigcup_{k \in \mathbb{N}} NTIME[k^n]$ (exponential time, linear exponent)
- $EXPTIME = DTIME[2^{n^{O(1)}}] = \bigcup_{k \in \mathbb{N}} DTIME[2^{n^k}]$ (exponential time)
- $NEXPTIME = NTIME[2^{n^{O(1)}}] = \bigcup_{k \in \mathbb{N}} NTIME[2^{n^k}]$ (exponential time)
- $EXPSPACE = DSPACE[2^{n^{O(1)}}] = \bigcup_{k \in \mathbb{N}} DSPACE[2^{n^k}]$ (exponential space)

Some of the boundaries between complexity classes are naturally established by the differences between the models of computation used, a deterministic Turing machine can be considered as a version of a non-deterministic Turing machine which explores branches sequentially, this relationship is represented as $DTIME[t(n)] \subseteq NTIME[t(n)]$ and $DSPACE[s(n)] \subseteq NSPACE[s(n)]$, one can also safely assume $DTIME[t(n)] \subseteq DSPACE[t(n)]$ and $NTIME[t(n)] \subseteq NSPACE[t(n)]$ since a Turing machine can only write at most one new symbol on the tape at every time step.
Other boundaries have been proven, for example Savitch's theorem proves (Savitch 1970):

$$NSPACE[s(n)] \subseteq DSPACE[(s(n))^2]$$

The relationship $NPSPACE = PSPACE$ is deduced as a corollary of the theorem; Having this relationship, other theorems related to boundaries in the complexity hierarchy can also be proven (TODO: add reference to lec16.pdf??):

- $NP \subseteq PSPACE$
- $PSPACE \subseteq EXPTIME$

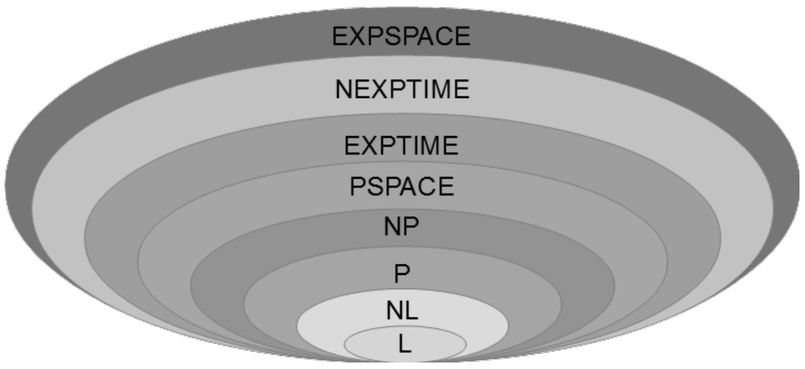

Figure 2.2: Relations between some of the canonical complexity classes

After applying these theorems the following hierarchy of complexity classes is obtained:

$$P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME \tag{2.1}$$

The boundaries between classes shown in the hierarchy above (2.1). are not hard boundaries, which means all three classes could in theory be the same class, though there exists proof that at least one of the boundaries must not be equal (TODO: add reference to time hierarchy theorem??); This leads us to one of the main unsolved problems in computational theory, is **P** equal to **NP**?, in other terms, 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? Any proof demonstrating the relationship or its negation will have a great impact on the field of computational theory and many others such as mathematics, philosophy, game theory or algorithm analysis to name a few.

2.1.3. 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, (TODO: add example of hard problem which is not complete).
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.

The boolean satisfiability problem was the first problem proven to be NP-Complete (Cook 1971), 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

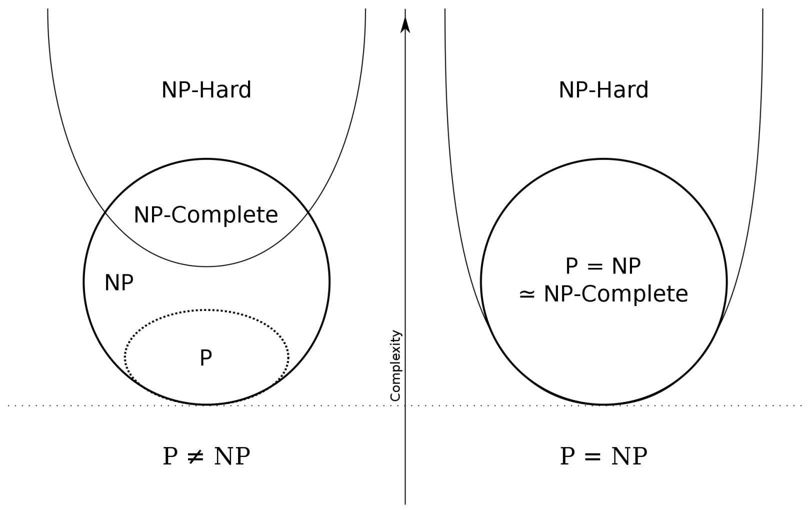

Figure 2.3: NP-Hard and NP-Complete boundaries

prove that no polynomial time algorithm for the SAT problem can exist, then $P! = NP$ is also proven.

2.1.4. 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.
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.

A genetic algorithm is a population based random search algorithm (Cochran et al. 2011) 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.

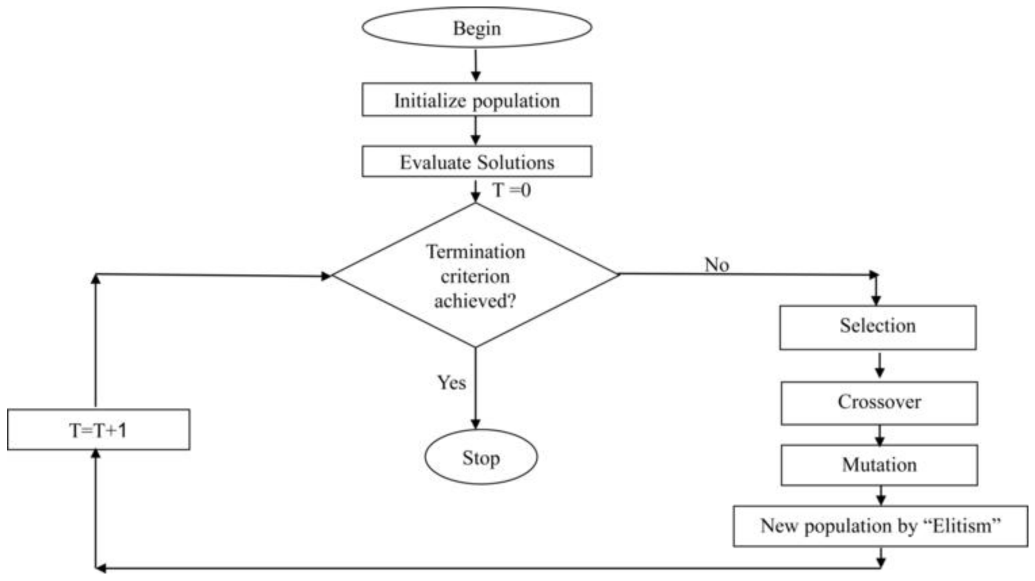

Figure 2.4: Genetic Algorithm steps

2.1.5. SAT Solvers

Most of the modern SAT-solvers are based on two algorithms, the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm and the *conflict-driven clause learning* (CDCL) algorithm.

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.

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.

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[1] 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**).

– *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**).

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**).

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*

[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*

Sum which changes the variable selection process, in addition to some other enhancements such as conflict clause deletion and early restarts.

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.

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**).

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 *inprocessing* (**Jarvisalo2012**), the three inprocessing 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.

CHAPTER 3
Genetic Algorithm Design

3.1 Genetic Algorithm Design

3.1.1. 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, 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.
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':

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

3.1.2. 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 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"

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

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.

3.1.3. Initial Population

The initial population is formed by randomly generating individuals until the established population size is reached, for each individual a list of N random integers is selected from the discrete uniform distribution in the interval $[0, 1]$ using numpy's randint function[2] where N 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.
Throughout this work two different ways of storing the population are used, one way stores it in a *set* and the other in an *array*, with the main difference between them being that arrays allow duplicate entries while sets do not, this leads to a greater computational cost when working with sets since individuals have to be regenerated until they are different from the existing ones, in order to reduce this cost lower population sizes are used for sets.

3.1.4. 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

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

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.

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.

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:

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

where k is the tournament size and n is the number of times the tournament is repeated (Jebari, Mediafi, and Elmoujahid 2013).

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.

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.

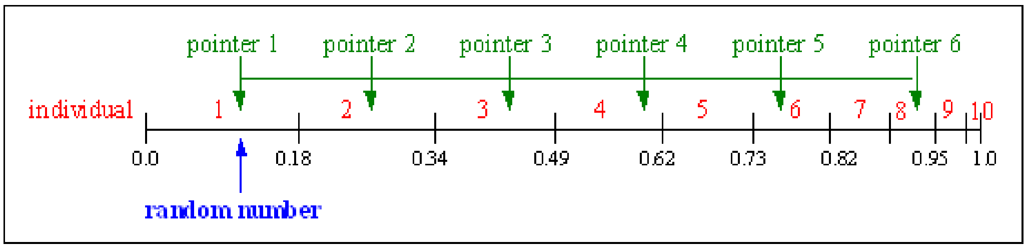

Figure 3.1: Stochastic Universal Sampling, individuals 1, 2, 3, 4, 6, 8 are selected

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.

Annealed Selection

Annealed selection (Jyotishree and Kumar 2012a) 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:

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

3.1.5. 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.

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.

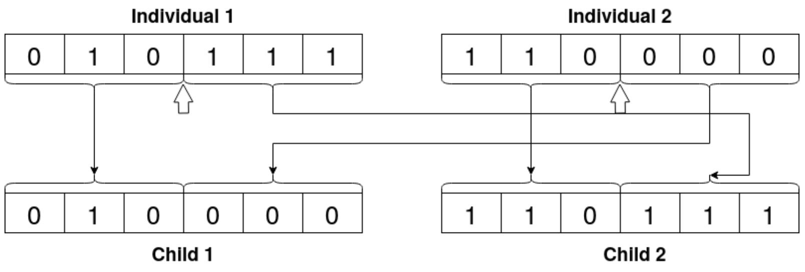

Figure 3.2: Single point crossover example with cutting point equal to 3

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.

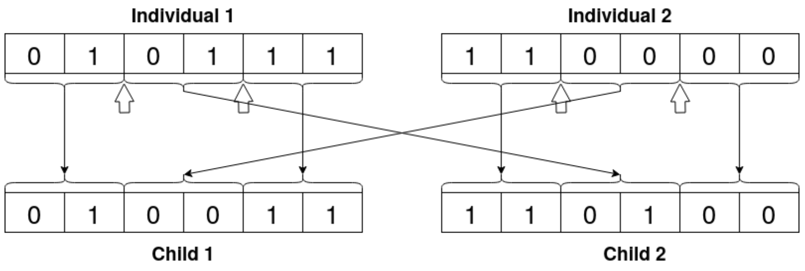

Figure 3.3: Two point crossover example with cutting points equal to 2 and 4

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.

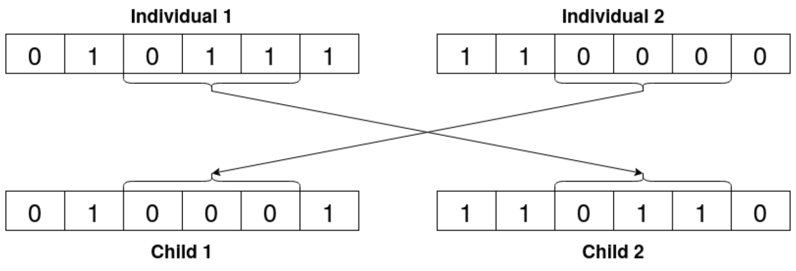

Figure 3.4: Sliding window crossover example with window length equal to 3

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.

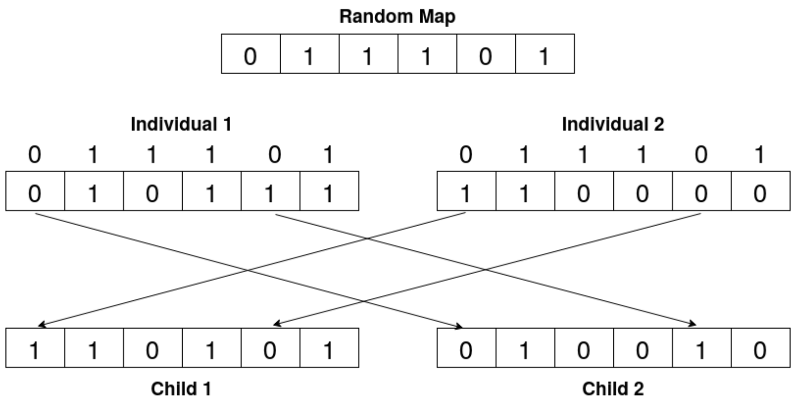

Figure 3.5: Random map crossover example with map 0, 1, 1, 1, 0, 1

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...

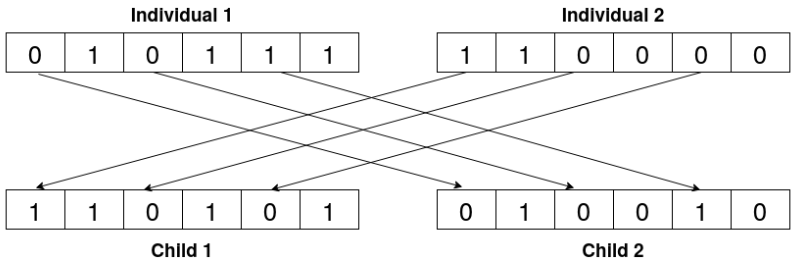

Figure 3.6: Uniform crossover example

3.1.6. 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.

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.

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.

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.

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.

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.

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.

3.1.7. 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.

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.

Mu-Lambda Replacement

Mu-Lambda replacement (Jyotishree and Kumar 2012b) 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.

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.

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.

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.

3.1.8. 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 (Bhattacharjee and Chauhan 2017).

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:

$$(x_1) \land (x_1 \lor x_2) \land (x_2 \lor x_3 \lor x_4) \land (x_1 \lor x_2 \lor x_4) \land (x_1 \lor x_2 \lor x_3 \lor x_4)$$

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.

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:

$$(x_1) \land (x_1 \lor \overline{x_2}) \land (x_2 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_2} \lor \overline{x_4}) \land (\overline{x_1} \lor x_2 \lor \overline{x_3} \lor x_4)$$

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:

$$(x_2 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor x_4)$$

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:

$$(x_1 \lor x_2 \lor \overline{x_3}) \land (x_1 \lor \overline{x_2} \lor x_4) \land (x_2 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_2} \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor x_4)$$

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:

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

CHAPTER 4

Implementation

4.1 Implementation

4.1.1. 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 of the algorithm.
Another important parameter in the genetic algorithm initialization function is a boolean called *allow_duplicates*, this value determines the type of storage for the population, if duplicate individuals are allowed, the population is stored in an array, if duplicate individuals are not allowed, the population is instead stored in a set; All the functions which can be used by the genetic algorithm are able to work with both types of population storage.
The parameters which need to be set for the initialization of 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.
- *allow_duplicates*: Determines if the population is a set or an array
- *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 or set (if duplicates aren't allowed) 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.

4.1.2. 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:

- *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.

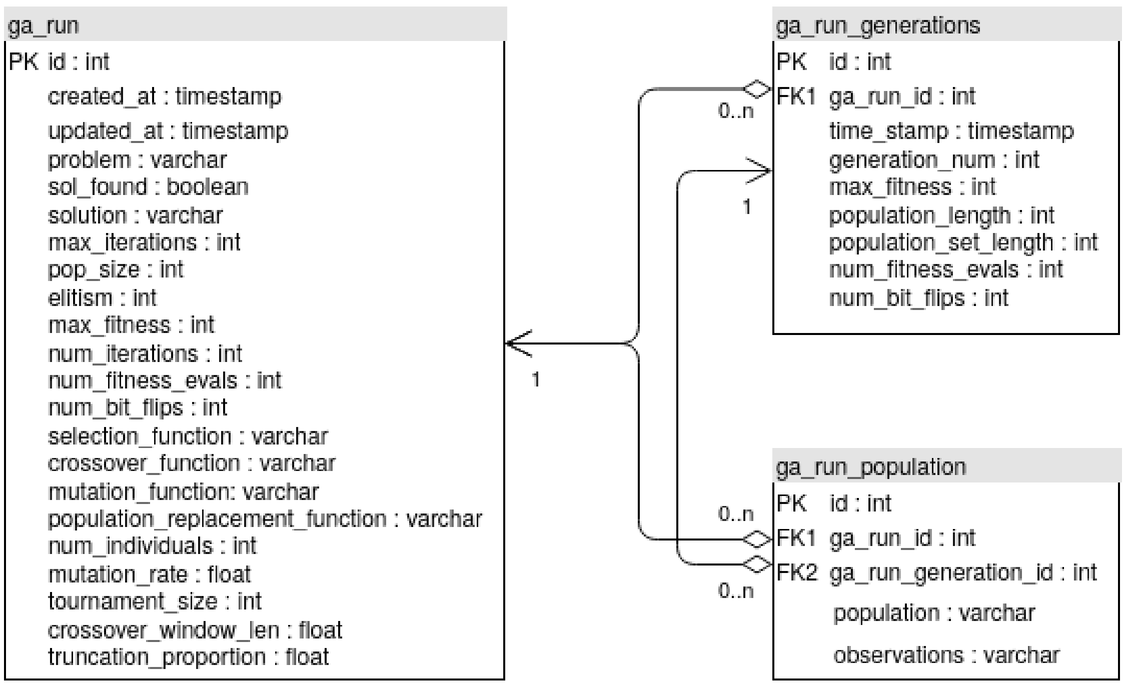

Figure 4.1: BBDD design schema

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.

4.1.3. Experiment design

(TODO: Add experiment design)

CHAPTER 5
Analysis

CHAPTER 6
Conclusions

Bibliography

[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`.