PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

Ministry of Higher Education and Scientific Research

UNIVERSITY OF SAAD DAHLEB BLIDA

Faculty of sciences

Computer Science Department

MASTER'S THESIS

In Computer Science

Option : Natural Language Processing

Toward an improvement of the genetic algorithm: application to Max-SAT problem

By

Ait Hellal Noureddine

Members of the thesis committee

Dr. HIRECHE C	Supervisor
Dr. MEZZI M	President
Mr. KAMECHE H	Examinator

Defended on June 25, 2023

Dedication

I thank my family for their continuous support and encouragement.

I am also thankful for my friends for being such great company throughout this.

Noureddine AIT HELLAL .

Blida, July 8, 2023.

Acknowledgment

First, I would like to express my admiration and gratitude to my advisor, Mrs. Hireche, for her invaluable guidance and assistance. Her hard work and dedication were instrumental in the success of this work.

I would like to thank the president for the honor of chairing the defense jury.

I thank the examinators, for providing me the chance to learn from their judgment and expertise.

Abstract

A considerable amount of research has been conducted on the optimization of genetic algorithms, what lead to a wide range of different genetic operators and solution representations; however, parameter tuning, which is a very crucial part of the good performance of the algorithm, is rarely discussed. This step can be considered a search for a set of good parameters that maximize the performance of the algorithm.

Taking this into consideration, we can say that this process is a double search search for good parameters, then search for a solution. In this work, we propose a genetic algorithm that combines the two search spaces, where a chromosome does not only represent a solution to the problem at hand but a set of genetic parameters too.

This method achieved a 4% increase in performance compared to the classic genetic algorithms. This result was reached after comparing the performance of our methods with all the combinations of genetic operators found in the state of the art, taking into consideration the temporal and hardware limitations.

Furthermore, this search space unification led to the elimination of the need to initialize parameters for each problem, which made it problem-independent.

Keywords: NP-completeness, MAX-SAT, Evolutionary Algorithms, Genetic Algorithm, Parameter tuning.

Résumé

Plusieurs travaux ont été menés sur l'optimisation des algorithmes génétiques, ce qui a conduit à un vaste ensemble d'opérateurs génétiques et différentes représentations de solutions. Cependant, le paramètrage de ces opérateurs, qui est une étape cruciale pour assurer la bonne performance de l'algorithme, est rarement discuté. Cette étape peut être considérée comme la recherche d'un ensemble de paramètres efficaces qui maximisent la performance de l'algorithme. En prenant cela en considération, nous pouvons dire que ce processus est une double recherche (recherche des bons paramètres, puis recherche d'une solution).

Dans ce mémoire, nous proposons un algorithme génétique qui combine les deux espaces de recherche. Où un chromosome ne représente pas seulement une solution au problème posé, mais aussi une série de paramètres génétiques.

Cette méthode a permis une amélioration des performances de 4 % par rapport aux algorithmes génétiques classiques. Ce résultat a été obtenu après avoir comparé les performances de nos méthodes avec toutes les combinaisons d'opérateurs génétiques de l'état de l'art, en tenant compte des limitations temporelles et matérielles.

En outre, cette unification de l'espace de recherche a permis d'éliminer la nécessité d'initialiser les paramètres pour chaque problème, ce qui a rendu la méthode indépendante du problème.

Mot-clés : NP-complétude, MAX-SAT, Algorithmes évolutionnaires, Algorithme Génétique, Paramétrage.

ملخص

تم إجراء قدر كبير من البحوث حول تحسين الخوارزميات الجينية ، مما أدى إلى مجموعة واسعة من العمليات الجينية المختلفة وتمثيلات الحلول ؛ ومع ذلك، نادراً ما تتم مناقشة ضبط المعلمات، وهو جزء مهم جداً للأداء الجيد للخوارزمية. يمكن اعتبار هذه الخطوة بحثاًا عن مجموعة من المعلمات الجيدة التي تزيد من أداء الخوارزمية.

مع أخذ ذلك في الاعتبار، يمكننا القول أن هذه عملية بحث مزدوج (البحث عن معلمات جيدة، ثم البحث عن حل). في هذا البحث، نقترح خوارزمية جينية تجمع بين مساحتي البحث، حيث لا يمثل الحل حلاً للمشكلة المطروحة فحسب، بل يمثل أيضاًا مجموعة من المعلمات الجينية.

حققت هذه الطريقة زيادة في الأداء بنسبة ٤٪ مقارنة بالخوارزميات الجينية التقليدية. تم الوصول إلى هذه النتيجة بعد مقارنة أداء هذه الطريقة مع جميع الطرق الموجودة في أحدث التقنيات، مع مراعاة القيود الزمنية والأجهزة.

زيادةً على ذلك، أدى توحيد مساحة البحث هذا إلى إلغاء الحاجة إلى تهيئة المعلمات لكل مشكلة، مما جعلها مستقلة عن المشكلة.

االكلمات الرئيسية: NP-completenes، MAX-SAT ، الخوارزميات التطورية ، الخوارزمية الجينية ، المعلمات

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ACO Ant colony optimization. 6

BCGA binary coded genetic algorithm. 14, 15

BNB Branch and bound algorithm. 20, 21

CDCL Conflict driven clause learning. 21, 22

CGA Classic genetic algorithm. x, xi, 15, 24, 25, 30, 41, 43–46, 48–50, 52

DPLL DavisPutnam-Logemann-Loveland algorithm. 21, 22

DRCRX Dominant random confusion resolution crossover. 47–49

EA Evolutionary algorithm. 17

EMO Electromagnetism like optimization. 7

EP Evolutionary programming. 8

FEP Fast Evolutionary programming. 8

GA Genetic algorithm. x, xi, 8, 11, 13, 14, 24, 25, 32, 33, 35, 42, 43, 46, 47, 49, 50, 52

GSA Gravitational search algorithm. 7

IPGA Integrated parameter genetic algorithm. 25, 29–31, 41, 42, 46–50

IWD Intelligent Water Drops. 7

MAXSAT Satisfiability maximization problem. 20, 32, 36

PSO Particle Swarm Optimization. 5, 8

RCGA Real coded genetic algorithm. 15, 30

RCRX random confusion resolution crossover. 47, 49

RFD River formation dynamics. 7

SAGA Self adaptive genetic algorithm. 13

SAT Satisfiability problem. 19–21, 23, 33, 41, 43

SRX simple resolution crossover. 47

SUS Stochastic universal sampling. 10

WMAXSAT weighted Satisfiability maximization problem. 20

Chapter 1

General introduction

Problem solving is the task of finding a solution to a certain problem. Problem solvers are not only tasked with finding such a solution but also doing so in a fast and reliable way. Simple problems only require defining a process that generates a valid solution. For NP-complete problems, where defining such a process is not possible, alternate methods of solving are required.

The straight-forward approach is to test all possible solutions by brute force; even though this will eventually reach a definite answer, given a large enough problem, this becomes quickly unmanageable. Guiding this extensive search by leveraging problem-specific characteristics can enhance this method. This type of solver is called an exact solver.

Another approach is to use metaheuristics, which are general problem solvers and easy to use. The drawback of these methods is that they are non-deterministic, so reaching the most optimal solution is not guaranteed.

Generally, when discussing the performance of an algorithm, we only consider its behavior after its execution starts; this makes sense as a good solver is the one that reaches the best solution in the shortest time. What this fails to capture is the considerable amount of time spent selecting a good set of parameters for the problem at hand.

This work focuses on the optimization of parameters search of genetic algorithm where this project argues that parameter search can be eliminated as a preliminary step of the genetic algorithm.

To demonstrate the performance of the proposed method, experiments were conducted on hlthe SATLIB satisfiability problem benchmark [1]as it is one of the most established NP-complete problems.

This document is organised as follows; First, an overview of state-of-the-art meta-heuristics focusing on the genetic algorithm is presented, followed by the satisfiability problem and examples of exact solvers. The proposed optimized genetic algorithm is presented in Chapter 3, the implementation details are discussed in Chapter 4. Chapter 5

showcases the analysis of the experimental results. Finally, a general conclusion and some perspectives are presented.

Chapter 2

Metaheuristics and Genetic algorithms

2.1 Introduction

Solving NP-complete problems can be a real challenge and present a serious handicap when processed by current machines. Indeed, the existing exact methods explore the whole search space in order to find a solution. Nonetheless, a large search space can generate a combinatorial explosion and a considerable overrun of calculation time that no machine can support.

Metaheuristics are stochastic problem solvers that rely on a random component as a mechanism to generate potential solutions that are iteratively optimized in order to have the best possible solution.

In this section, we will give a general overview of existing methods, their advantages, and their drawbacks.

2.2 Overview of Metaheuristics

Generally,a problem corresponds to a set S of possible solutions and a function f allowing to measure the quality of the elements of this set. In other words, a problem is defined as a set of potential solutions named search space where the validity of each solution is determined by an objective function.

When faced with an optimization problem, the goal is to find the best possible solution. This is equivalent to a maximization (minimization) of the fitness function, where the best solution is the global maximum (minimum). Because of the distribution of good solutions in the search space, the algorithm may get stuck in local optima. This issue may be caused by a bad parameter choice.

In case of an optimization problem, the set of potential solutions is a search space and the objective function is called a fitness function, where the fitness is a continuous value rather than a binary value. This is important as the optimization process is gradual; with

a binary evaluation, there is no difference in quality between two false solutions.

As an example, when looking for the shortest path between a point A and a point B, S is the set of all possible paths and the fitness function determines the quality of each solution, in this case with the length of the path.

The difficulty of solving this kind of problem is directly related to the search space size which is often very large or even infinite.
Figure 2.1 shows the increase of search space size in function of the number of nodes in the previous example

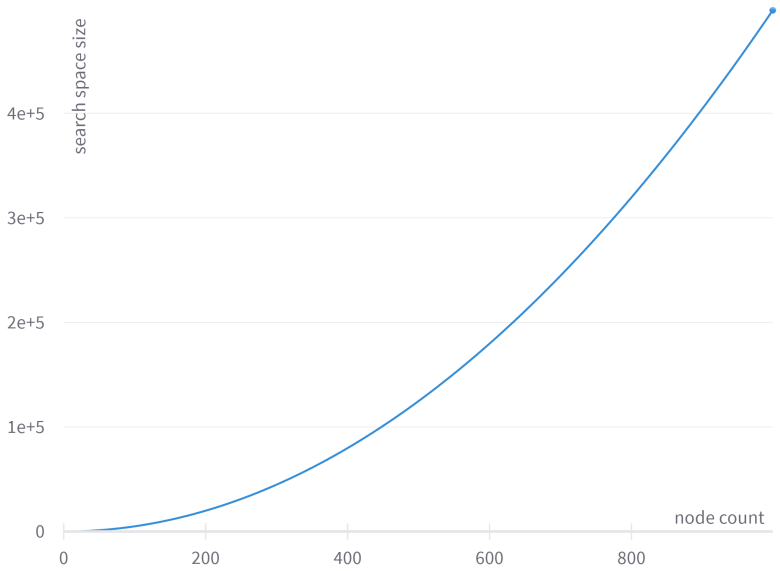


Figure 2.1: Shortest path search space size

Metaheuristics appeared as an alternative to exact methods ,which explore the whole space search looking for the best solution, in order to find a compromise between solution quality and execution time. The goal is to optimize the solution as much as possible while guaranteeing an acceptable calculation time[2].

Metaheuristics are optimization algorithms that, generally, generate a single or a set of random potential solutions and optimize them at each iteration. These algorithms consider the optimization version of a given problem, where a stochastic process is guided by a fitness function in order to search the potential solution space.

Nowadays, a plethora of metaheuristics exist and can be categorized as follows:

- Physics based metaheuristics
- Swarm based metaheuristics
- Evolutionary algorithms

2.3 Swarm based metaheuristics

Swarm based intelligence refers to the fact that natural or artificial objects, systems, or insects devoid of any intelligence can have collectively intelligent behavior [3].

2.3.1 Particle Swarm Optimization

Proposed by Eberhart and Kennedy (1995), particle swarm optimization (PSO) is one of the first proposed swarm based algorithms. It is an evolutionary optimization calculation technique based on the concept of swarm intelligence[4].

In PSO, the social behavior is modeled by a mathematical equation that guides particles during their displacement process. The displacements are influenced by the inertia, cognitive, and social component. Each of these components reflects a part of the equation.

A particle is described with a velocity and a position vector.

Let P^i a particle, P^ix is the position of P at the i^{th} iteration,P^iv is the velocity of P at the i^{th} iteration.(Equations 2.1 and 2.2 describe the position and velocity updates for the $i+1$ iteration, respectively).

$$P^{i+1}x = P^ix + P^iv \tag{2.1}$$

$$P^{i+1}v = WP^iv + C_1R_1(Px_{max} - P^ix) + C_2R_2(Gx_{max} - P^ix) \tag{2.2}$$

Where R_1 and R_2 are random values between 0 and 1, W determines how much of the previous velocity the particle keeps,C_1 and $C2$ control the effects of local and global best, respectively, and W, C_1 and $C2$ are hyperparameters,

The algorithm starts with a random set of particles, after evaluation, Gx_{max} and Px_{max} are determined and represent the global best and local best, respectively. Each particle is updated according to the described equations above.

High velocity values cause large steps in the particles' movement, thus limiting local exploration. This problem was mitigated by clamping P^iv to a maximal value Max_v according to 2.3[5]:

$$P^iv = \begin{cases} P^iv \text{ if } P^iv < Max_v \\ Max_v \text{ otherwise} \end{cases} \tag{2.3}$$

Even though this method makes the particles more controllable, if all the velocities become equal to Max_v the particle continue to conduct searches within a hypercube and will probably remain in the optima but will not converge in the local area[5].

2.3.2 Ant Colony Optimization

Ant colony optimization (ACO) takes inspiration from the foraging behaviour of ant colonies, where ants indicate trails leading to food using pheromone deposits. [6] This behaviour is used as indirect communication of path desireability between ants. [7]

ACO implements artificial ants as agents that iteratively construct solutions stochasticly, taking in consideration the pheromone component in the construction (the pheromone is called a pheromone value). [7]

ACO considers combinatorial optimization problems only, where a problem P is formulated as follows (Equation 2.4 describes the components of a combinatorial optimization problem) :

$$P = (S, \Omega, f) \tag{2.4}$$

Where S is a search space defined over a set of decision variables X_i, a feasible solution adheres to a set of constraints Ω, f is the fitness function,[8]

A decision variable x_i takes a discrete value $v_i^j \in D_i$, where $x_i = v_i^j$ is a solution component c_i^j. This notion is important, as a pheromone value τ_i^j is an evaluation of the component c_i^j.

For each iteration of ACO, an artificial ant goes through the following operations:

- Construct a solution
- Local search (optional)
- Update pheromone

A solution is constructed based on the following probability distribution (Equation 2.5 represents the probability of a solution component c_i^j being chosen given the the current solution s):

$$p(c_i^j|s) = \frac{{\tau_{ij}}^\alpha \cdot \eta(c_i^j)^\beta}{\sum_{l=0}^{|N(s)|} {\tau_{il}}^\alpha \cdot \eta(c_i^l)^\beta} \tag{2.5}$$

Equation 2.6 updates the pheromone deposit of a solution component.

$$\tau_{ij} = (1 - P)\tau_{ij} + \sum_{k=0}^{m} g(s_k) \tag{2.6}$$

g calculates the pheromone value laid by an ant k for a component c_i^j,P is the evaporation rate.

The above solution construction and pheromone update were proposed in the Ant System (AS) method, other methods can be used. Local search is an optional step used to refine the solutions.

2.4 Physics based metaheuristics

Physics based optimization methods leverage already established physical principles, such as gravitational force in gravitational search (GSA), and Electromagnetic force in electromagnetism-like optimization (EMO), other physical phenomena were used in algorithms like Intelligent Water Drops (IWD) and River Formation Dynamics (RFD).

2.4.1 Gravitational search algorithm

Gravitational search algorithm (GSA) represents solutions as masses that interact according to gravity and motion laws. The mass of each object (solution) is proportional to its fitness, so agents with higher fitness have a higher influence on the position of other solutions,[9] pulling them towards a region of potentially higher optimality. On the other hand, a heavier object resists changes in motion. Therefore, the trajectory of more optimal solutions is less affected, making less optimal solutions follow the path to a more optimal region.

Let x_i be a point in an n-dimensional space where n is the number of decision variables $x_i = (x_i^0, x_i^1 \ldots, x_i^d, \ldots, x_i^n)$. M_i, the mass of the i^{th} agent (point). At each iteration, the force exerted on the i^{th} agent by all the other masses is calculated (Equation 2.7 describes the force exerted by all masses on the i^{th} agent [9]).

$$F_i^d = \sum_{j=0}^{N} r.F_i^j \tag{2.7}$$

With r a random value $r \in [0, 1]$, F_{ij}^d the force exerted by mass j on mass i in the dimension d, given by the Equation 2.8 , and N the number of agents[9].

$$F_{ij}^d = G\frac{M_i \cdot M_j}{R + \varepsilon}(x_j^d - x_i^d) \tag{2.8}$$

To update the agents, first the new acceleration is calculated, followed by a position update[9](Equation 2.9 describes the acceleration of the i^{th} mass).

$$a_i^d = \frac{F_i^d}{M_i} \tag{2.9}$$

2.5 Evolutionary algorithm

Inspired by evolutionary theory, evolutionary algorithms represent a class of optimization algorithms based on population optimization. A random initial population is iteratively optimized, evolving towards a higher-performing population, using operators inspired by natural evolution. Each generation is then created as a recombination of a selected set

of parents from the previous population, where the selection process favors fitter parents. Some algorithms inject randomness into a generation by mutating individuals.

2.5.1 Evolutionary programming

Evolutionary programming (EP) is a stochastic search algorithm inspired by biological evolution and natural selection. In EP, an initial population of random solutions is generated. Each candidate solution is evaluated based on an objective function[10]. Selection is performed, where individuals with higher fitness values have a greater chance of being selected as parents for reproduction. The selected individuals undergo reproduction through mutation. The new offspring replace some individuals in the current population; in other words, the new generation is a subset of the union of the mutated solutions and the previous population. The process continues iteratively, with the population evolving over generations. The algorithm terminates based on a halting condition[11].

EP is known to have slow convergence. Methods like fast evolutionary programming (FEP) were proposed[11] to tackle this issue by using a mutation operator based on a Cauchy distribution rather than the Gaussian distribution used in the original algorithm. This choice is based on the claim that Cauchy mutation results in longer jumps than Gaussian mutation; other research results support this claim. A similar use case of Cauchy mutation was used in other algorithms, like PSO.

Genetic algorithms (GAs) are another type of evolutionary algorithm that is similar to EP in that both use mutation and selection operators, but GA differs in that it uses a genetically inspired structure to represent solutions.

2.6 Genetic algorithm

Genetic algorithm (GA)is a population-based algorithm inspired by the theory of evolution. Its principle is that, on an initial randomly generated population (set of agents), representing solutions (chromosomes) to a given problem, where each decision variable is a gene, GA applies a set of genetic operators to generate the next generation of solutions[12].

After evaluating the fitness of each solution, a subset of the current generation called parents is selected based on their fitness (selection operator), a crossover operator is applied to parent pairs to generate new offspring (solutions). Finally, each chromosome is slightly modified with the mutation operator. This process is repeated until a satisfactory solution is found or a stopping condition is reached.

Algorithm 1 presents the general structure of GA:

Notice that mutation and crossover are parameterized. These parameters control how mutation and crossover are performed and can have a significant impact on the algorithm's behavior and performance.

Algorithm 1 Genetic Algorithm

Input: A problem instance P
Output: Best solution reached

```
G ← create_initial_generation()
while stopping condition and problem not solved do
    G_evaluated ← evaluate_solution(G, P)
    G_parents ← selection(G_evaluated)
    G_new ← crossover(G_parents, parameters)
    G_new_mutated ← mutate(G_newparameters)
    G ← G_new_mutated
end while
return G.best_solution
```

Several versions of genetic algorithm exist, depending on the genetic operators choices. This will be discussed in the following sections.

2.6.1 Selection

During the selection process, the parents of the next generation are chosen based on their fitness scores. The goal is to generate more fit solutions by allowing only good solutions to influence the next generation.

Some selection methods result in aggressive selection strategies[13], which reduce diversity. A low selective pressure on the other hand, results in slow convergence.

Selection methods can be grouped into the following categories:

2.6.1.1 fitness proportional selection

A selection method is said to be fitness proportional if the selection probability of a solution is determined by its score. Outliers with relatively high scores compared to the rest of the population are heavily selected, resulting in a less diverse population[14], which may lead to stagnation. These methods are advantageous due to their simplicity.

Given a population G^i of size n, to generate generation G^{i+1} of the same size, a set of parents $P \in G^i$ are selected, The following selection methods can be used:

- **Roulette selection**

  Roulette selection divides a wheel into N shares proportional to the fitness of each solution,and then the wheel is randomly rotated to select the parents of the next generation. To address the issues of fitness proportional selection, a modified version of roulette selection (rank selection) was developed[13].

  The selection probability of a solution g is calculated as follows (Equation 2.10 describes the probability of selection for an agent g using roulette selection[13])

$$p(g) = \frac{f(g)}{\sum_{j=0}^{n} f(g_j)} \tag{2.10}$$

Where f is the fitness function.

- **Stochastic universal sampling**

  Stochastic universal sampling (SUS) is a variant of roulette selection. The main difference is that SUS selects parents using a set of selection pointers rather than sequential selection. SUS assigns solutions to contiguous sections of varying lengths based on their fitness score.To position the selection pointers, a random position is generated for the first one in the range $[0, 1/N]$,and the subsequent pointers are equally spaced by $1/N$ [15]. (Figure 2.2 represents the selection process using stochastic universal sampling).

Figure 2.2: Stochastic universal sampling[15]

2.6.1.2 Ordinal based selection

Ordinal based selection methods try to eliminate the premature convergence problem caused by fitness proportional methods by using alternative ordering techniques.

- **Rank selection**

  Rank selection is a variant of roulette selection where, selection probability is a function of solution rank rather than fitness value. Solutions are ranked in the ascending order of their fitness; the most fit solution takes rank of n and the worst a rank of 1.(Equation 2.11 describes the selection probability of selection for an agent g using rank selection[13])

$$p(g) = \frac{f(rank(g))}{n \times (n-1)} \tag{2.11}$$

- **Tournament selection**

  To select the gene pool for the next generation, a parent is chosen based on a fitness comparison of randomly selected agents from the last population. A tournament generally contains a pair of agents, but there is no theoretical limit to tournament size[16].

2.6.2 Crossover

Any agent in GA is either generated in the initial generation or is the result of a combination of two solutions from the previous generation. Combining two good solutions potentially results in a better solution is the main idea of crossover.

After selecting the parents of the next generation, these are combined two by two to generate new agents using one of the methods described below.

Given two parents P_1 and P_2, the crossover operation results in two children C1 and C2.

- **Single point crossover**
  Given a crossover point k where $k \in [1, m-1]$ (m is the number of decision variables), the two parents, P_1 and P_2, are split at the point k, this results in two contiguous parts for each parent. A child solution combines different parts (at their respective places) of the parents to generate a chromosome[17] (Figure 2.3 represents the selection process using single point crossover).

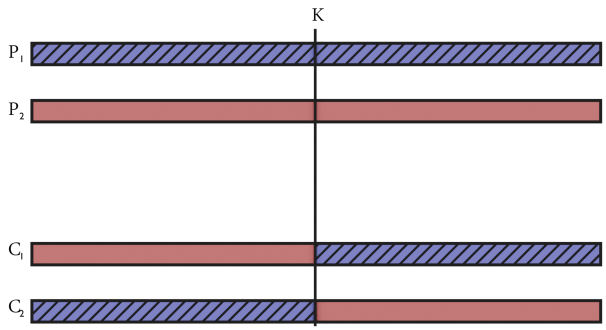


Figure 2.3: single point crossover

- **K-point crossover**
  Given K crossover points where $\forall k \in K, k \in [1, m-1]$ (m is the number of decision variables), each parent is split into $K+1$ sequences. C_1 and C_2 are constructed from parent sequences by alternating the donor parent at each k point, starting with a different parent for each child to ensure that C_1 and C_2 are different[17] (Figure 2.4 represents the selection process using k-point crossover).

- **Mask crossover**
  Mask crossover defines a binary vector M of length m, where m is the number of decision variables, the mask value determines which parent contributes each bit. Knowing that a crossover operation results in two offspring, we can use the inverse mask to generate the second solution. After each generation, a new mask is generated randomly [17] (Figure 2.5 represents the selection process using mask crossover).

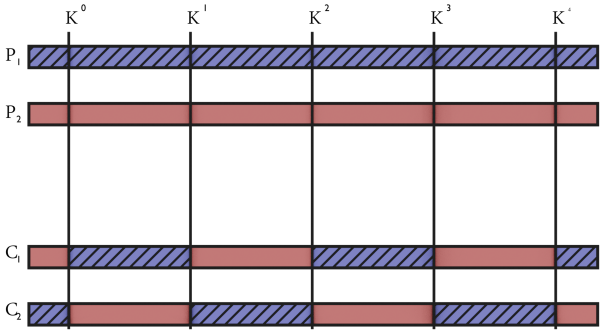


Figure 2.4: K-point crossover

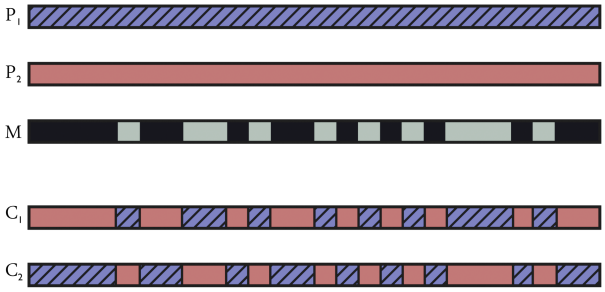


Figure 2.5: Mask crossover

2.6.3 Mutation

Mutation operator randomly edits parts of the solution, with the goal of maintaining solution diversity and prevent premature convergence[18].
The mutation rate controls the number of mutations that occur. This rate is generally set to a low probability; a higher mutation rate causes big changes in the solutions, and the search becomes equivalent to a random search.

Many mutation functions exist among which;

- **Bit-flip mutation**
  In bit-flip mutation, a set of random bits $A \in P$ is selected according to the mutation rate, where P is the current solution,A' is the set of new values for A.

$$A'_i = \begin{cases} 1 \text{ if } A_i = 0 \\ 0 \text{ if } A_i = 1 \end{cases} \tag{2.12}$$

- **Inversion mutation**
  After selecting two random points p_1 and p_2 in the solution, where $p_2 > p_1$, the order of the contiguous bits in the range $[p_1, p_2]$ is reversed[19] (Figure 2.6 represents the mutation of a solution S using inversion mutation).

- **Scramble mutation**
  Like inversion mutation, a subset of contiguous bits in the chromosome is selected to

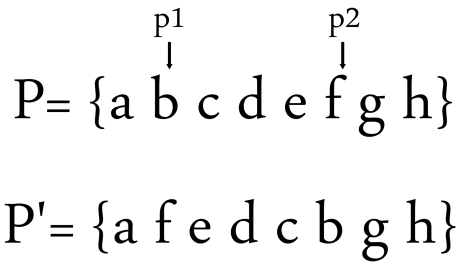


Figure 2.6: Inversion mutation

be mutated by a random shuffle[19] (Figure 2.7 represents the mutation of a solution S using scramble mutation).

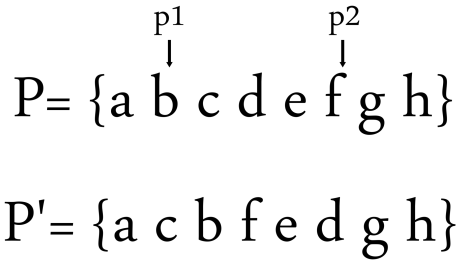


Figure 2.7: Scramble mutation

2.7 Adaptive genetic algorithm

Knowing that parameter selection is a tedious procedure, self-adaptive genetic algorithm (SAGA) tries to optimize this process by implementing individual-level self-adaptation by defining some of GA's parameters as agent parameters, where the selection process does not only select for fitter solutions but also for the parameters that lead to better solutions, as a way to optimize parameter search[20].

SAGA integrates the crossover method, mutation probability, and a noise range (noise is used as a secondary means of mutation). The crossover method parameter determines which crossover algorithm is in use. Another particularity in the way SAGA implements these parameters is the way the parameters are operated upon[20];

SAGA splits the genetic operations into solution operations and parameter operations; first crossover and mutation are applied to the parents parameters, generating the offspring's parameters; then the offspring's solution is calculated according to its parameters. The following pseudo code gives a more detailed description(Algorithm 2 is a pseudocode implementation of SAGA[20]).

SAGA resulted in a similar performance to a simple genetic algorithm with a tuned set of parameters [20].

Algorithm 2 Adaptive Genetic Algorithm

```
initialize generation (G)
while not solved do
    PP ← select parent pairs(G)
    new_G ← empty set
    for each P1, P2 in PP do
        S.param ← crossover_parameters(P1.param,P2.param)
        S.param ← mutate_parameters(S.param)
        S.solution ← crossover_solution(P1.solution,P2.solution, S.param)
        new_G.add(S)
    end for
    G ← new_G
end while
return G.best_solution
```

2.8 Real coded genetic algorithms

A large set of real-life problems deal with continuous parameters, but GA's deal exclusively with binary coded parameters. To be able to use binary coded genetic algorithms (BCGA), we have to either translate the problem into binary encoding or use a genetic algorithm compatible with real coded solutions; this requires crossover and mutation operators compatible with real valued solutions[21].

The first proposed methods were based on a binary encoding; given a problem with a search space $S = S_1 \times S_2 \times \cdots \times S_n$, with n the number of decision variables, s is a solution, $s = s_1, s_2, ...s_n$ with $s_i \in [a_i, b_i]$, (a_i b_i the lower and upper bound of the i^{th} value respectively), an encoding function $cod(s) = x$ that transforms a real value s to a binary string of size l (Figure 2.8 represents the Gray encoding of a binary soluiton S into a solution S')[21].

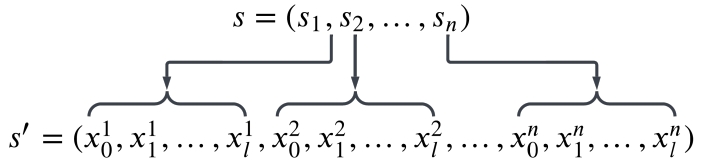


Figure 2.8: Binary encoding

Even though binary encoding is advantageous because of the ability to leverage already existing GA operators, it becomes less performant and especially taxing the bigger the search space; the length of the binary strings increases, increasing with it the size of the binary search space. On the other hand, having a range $[a_i, b_i]$ that does not cover all the possible binary string codifications results in values outside the acceptable range. For example, to codify a parameter $S_i \in [0, 8]$, it requires at least 4 bits, with the lower bound 0000 and the upper bound 1000, but a crossover or a mutation can result in a value outside this range(1001,1010,...,1111). These conflicts can be mitigated by discarding

invalid solutions, assigning a low fitness value , or remapping these solutions to valid ones, this problem becomes more pronounced the larger the set of invalid solutions.

Moreover, the structure of binary numbers can be an issue. Two adjacent values can differ in many bits, this becomes an issue when the algorithm converges close to a maximum. For example, considering that 8 is the best solution, to converge from a solution 0111 to 1000 takes a complete change in all solution bits. Even the distance between the two values is small, this makes it difficult for the algorithm to converge to the best solution. A better distance metric that expresses this issue is the hamming distance[21], where the distance between two strings is the number of bits where they differ. The previous example is a special case where every bit of the two strings differ, called the hamming cliff.

As a solution to the above problem, we can use gray coding instead of binary coding, where the hamming distance between two consecutive gray coded integers is equal to one.

The gray coding of a binary string $H = \{h_0, h_1 \dots h_l\}$ to a string $H' = \{h'_0, h'_1, \dots, h'_l\}$ is done as follows(Equation 2.13) :

$$h'_i = \begin{cases} h_0 \text{ if } i = 0 \\ h_{i-1} \oplus h_i \text{ if } i > 0 \end{cases} \tag{2.13}$$

The decoding process is as follows(Equation 2.14):

$$h_i = \begin{cases} h'_0 \text{ if } i = 0 \\ h_{i-1} \oplus h'_i \text{ if } i > 0 \end{cases} \tag{2.14}$$

Because of the stated drawbacks of BCGA, a real coded genetic algorithm (RCGA) was proposed. Parameters are directly codified as real values, and this method implements a set of genetic operators. As the CGA operators can not be used on real values, RCGA is a less complex solution because a codification step is not needed. Contrary to BCGA, the precision of the parameters is not bounded by the used codification but by the computer's characteristics.
The following are the most notable genetic operators relating to RCGA:

2.8.1 Crossover

Given two parents $P_1 = \{p_0^1, p_1^1, \dots, p_n^1\}$ and $P_2 = \{p_0^2, p_1^2, \dots, p_n^2\}$, the resulting offspring is $H_k = \{h_0^k, h_1^k, \dots, h_n^k\}$, and generally $k \in \{1, 2\}$. In addition to CGA's crossover methods RCGA implements the following operators:

- **Flat crossover**
  For each gene h_i^k in the offspring H_k, a random value is chosen from the set $\{p_i^1, p_i^2\}$[21].

- **Arithmetical crossover**
  The two offspring are calculated according to the following equations.

  Equations 2.15 and 2.16 describe the arithmetical crossover of two parents c^1 and c^2 resulting in offspring H_1 and H_2 respectively[21].

$$\forall h_i^1 \in H_1, h_i^1 = \lambda c_i^1 + (1-\lambda)c_i^2 \tag{2.15}$$

$$\forall h_i^1 \in H_2, h_i^2 = \lambda c_i^2 + (1-\lambda)c_i^1 \tag{2.16}$$

  With λ can be a constant value $\lambda \in [0,1]$ (called uniform arithmetical crossover) or defined in function of the generation count (called non-uniform arithmetical crossover).

- **Linear crossover**
  Contrary to the above crossover methods, linear crossover generate three offspring then chooses the two best agents to propagate to the next generation.
  Equations 2.17,2.18 and 2.19 describe the linear crossover of two parents c^1 and c^2 resulting in three offspring where two are randomly chosen[21]:

$$\forall h_i^1 \in H_1, h_i^1 = \frac{1}{2}c_i^1 + \frac{1}{2}c_i^2 \tag{2.17}$$

$$\forall h_i^2 \in H_2, h_i^2 = \frac{3}{2}c_i^1 + \frac{1}{2}c_i^2 \tag{2.18}$$

$$\forall h_i^3 \in H_3, h_i^3 = -\frac{1}{2}c_i^1 + \frac{3}{2}c_i^2 \tag{2.19}$$

2.8.2 Mutation

Let $C = \{c_0, c_1, \ldots, c_n\}$ be a chromosome where each gene is defined in a range $c_i \in [a_i, b_i]$. The following mutation methods can be applied on each gene; Let c_i' be the result of mutating c_i.

- **Random mutation**
  c_i' takes a random value in the range $[a_i, b_i]$

- **Non-uniform mutation**
  Equation 2.20 describes the non-uniform mutation of a gene c_i[22]

$$c_i' = \begin{cases} ci + \Delta(t, b_i - c_i) \text{ if} \tau = 0 \\ ci - \Delta(t, c_i - a_i) \text{ if} \tau = 1 \end{cases} \tag{2.20}$$

  with t the number of the current generation $t \in [0, g_{max}]$, τ a random value $\tau \in 0, 1$ determining the shifting direction, the function $\Delta(l, y)$ calculates the shifting distance for the value c_i.

Equation 2.21 calculates the shifting distance given the current generation and shifting range[22].

$$\Delta(t, y) = y(1 - r^{(1-\frac{t}{g_{max}})^b}) \tag{2.21}$$

with r a random value $r \in [0, 1]$, and b a user determined value controlling the degree of dependency on the generation count, and $\Delta(t, y) \in [0, y]$.
This method starts out with large shift values and gradually favours local search the closer the current generation is from g_{max}.

- **Muhlenbein's mutation**
  Equation 2.22 describes the Muhlenbein mutation of a gene c_i[21]

$$c_i' = c_i \pm rang_i \cdot \gamma \tag{2.22}$$

where $rang_i$ defines the mutation range, and it is normally set to $0.1 \cdot (b_i - a_i)$ γ is calculated as follows(Equation 2.23):

$$\gamma = \sum_{k=0}^{15} \alpha_k 2^{-k} \tag{2.23}$$

with $\alpha_i \in \{0, 1\}$ randomly generated with the probability $p(\alpha_i = 1) = \frac{1}{16}$[21]

2.9 Parameter tuning

Considering the sensitivity of metaheuristics to parameter choice, determining the best set of parameters becomes paramount for good performance. Specifically for Evolutionary algorithms where mutation probability, crossover points, and generation size are the main parameters of concern.

Considering the computational cost of tuning these values by trial and error, several methods of parameter tuning were developed.

Meta evolutionary algorithms tune parameters by considering parameter tuning as a problem by itself, hens meta EA, where a solution represents a set of parameters, and a solution is evaluated by running anEA with the given parametersEA[23]. This method is computationally demanding because each evaluation requires a full run of the EA[24].

Another approach is Sequential Parameter Optimization, where a set of parameter vectors is first tested several times to determine their efficacy, and a new set of vectors is generated based on a regression model constructed from the previous set of parameters [24].

2.10 Conclusion

Through this chapter, we have introduced the basic concepts of metaheuristics and their different types, such as swarm-based and physics-based metaheuristics.

Evolutionary algorithms were discussed, and especially the genetic algorithm, which is the subject of this research work, was put forward.

Chapter 3

Satisfiability problem: definition and solvers

3.1 Introduction

The Boolean satisfiability problem is a widely established problem with both theoretical and practical significance. SAT is the first problem to be proven to be NP-complete, and a wide range of practical problems can be formulated as SAT problems, like model checking, automatic test pattern generation, automated theorem proving, and many more. With SAT being the first NP-complete problem to be proven, there exists a large body of work mapping problems to SAT, which means that advancement in solving SAT problems translates to all related problems. This chapter introduces the fundamentals of SAT and some of its solvers.

3.2 Satisfiability problem; Definition and MAX-SAT variant

The goal of SAT problem is to determine whether a boolean formula P in Conjunctive Normal Form (CNF) is satisfiable. I.e : A conjunction of clauses C, where each clause is a disjunction of literals $c = (\alpha_0 \vee \alpha_2 \dots \alpha_n)$ and each literal is a propositional variable or its negation. Its formal definition is as fallows;

- **Instance:** A set of m clauses and n variables forming a CNF formula.
- **Question:** Is the CNF formula satisfiable? IS there any assignment of the variables that satisfies the instance?

For example, the following boolean formula P is satisfiable, because given the assignment S, all clauses of P are satisfied.

$$P = (\alpha_1 \vee \neg\alpha_2 \vee \alpha_3) \wedge (\neg\alpha_3 \vee \alpha_1) \wedge (\alpha_2 \vee \alpha_3) \tag{3.1}$$

$$S = \{\alpha_1 = 1, \alpha_2 = 0, \alpha_3 = 1\} \tag{3.2}$$

Due to the combinatorial explosion that may occur while solving a SAT instance, less complex variants have been introduced, from which we cite MAX-SAT for Maximum SAT.

In MAX-SAT, instead of finding a boolean assignment that satisfies all the clauses, MAX-SAT looks for the maximum number of satisfiable clauses in a given instance. In other words, MAX-SAT is the optimization version of SAT, whose formal definition is :

- **Instance:** A set of m clauses and n variables forming a CNF formula.
- **Question:** What is the maximum number of satisfied clauses?

More general versions of SAT exist, like Weighted Partial MAXSAT (WMAXSAT) or PM-sat. The generalization of SAT permits the use of solvers across different problems, WMAXSAT assigns a weight to each clause representing its importance, WMAXSAT solvers can be used for MAXSAT, as an instance of the former with an equal weight for every clause is equivalent to a MAXSAT instance, the solvers below can be used to solve MAXSAT instances as they solve a more general problem.

3.3 SAT solvers

Since the SAT problem is considered the backbone of NP-complete problems, a plethora of solvers have been developed for solving it. These solvers can be organized into two major classes as follows:

3.3.1 Complete solvers

Complete solvers explore the entire search space of a given problem. This type of algorithm reaches a definitive answer at the end of its execution. It either finds the solution to the problem or prove that the problem can not have any solution.

Complete solvers traverse the search space using a search tree structure, where a solution is constructed by incrementally assigning a truth value to decision variables.

3.3.1.1 Branch and bound solvers

BNB methods use inference rules to extend the current solution; as one variable assignment is made, another may be logically inferred. For example, assigning false to a variable α while having the following clause in the SAT instance $C = \alpha \vee \beta$ makes beta trivially

assigned to true, as any other assignment makes it unsatisfied[25]. Algorithm 3 presents a typical implementation of the BNB algorithm [25].

Algorithm 3 A typical BnB algorithm for Max-SAT

Input: A CNF formula Φ with n the number of variables of Φ
Output: (UB, I_{UB}) with UB the best solution found and I_{UB} the corresponding assignment

1: $UB \leftarrow \sum_{c_j \in \Phi} w_j$
2: $I_{UB} \leftarrow \emptyset$
3: $I \leftarrow \emptyset$
4: **repeat**
5: $I \leftarrow$ assignment extension(Φ, I)
6: $LB \leftarrow \sum_{c_j \in \Phi|I} w_j +$ count remaining conflicts$(\Phi|I)$
7: **if** $LB \geq UB$ **then**
8: backtrack()
9: **else if** $|I| = n$ **then**
10: $UB \leftarrow \sum_{c_j \in \Phi|I} w_j$
11: $I_{UB} \leftarrow I$
12: backtrack()
13: **else**
14: $l \leftarrow$ select new decision()
15: $I \leftarrow I \cup \{l\}$
16: **end if**
17: **until** $|I| > 0$
18: **return** (UB, I_{UB})

Given a partial assignment α, a conflict is a contradiction reached by a unit propagation of α, where two unit clauses assign different values for the same variable, this means that the assignment α does not satisfy the SAT instance.

3.3.1.2 DavisPutnam-Logemann-Loveland algorithm

Similarly to BNB, DavisPutnam-Logemann-Loveland (DPLL) explores the search tree by iteratively expanding the partial solution, either by a truth value assignment or a Unit propagation. Unit propagation procedure consists of assigning a truth values that makes the unit clause true. A unit is a clause with all but one of its literals equal to 0.
In case of a conflict, DPLL backtracks to the last assignment and reverts all following assignments.
The following implementations written in a recursive form(Algorithm 4 is a typical implementation of the DPLL algorithm [26]).

3.3.1.3 Conflict driven clause learning

Conflict-Driven Clause Learning (CDCL) is an extension DPLL algorithm. It follows the same general structure as a DPLL solver but introduces additional techniques to

Algorithm 4 DPLL algorithm

```
1: function DPLL-RECURSIVE(F, τ)
2:     Input: A CNF formula F and a partial assignment τ
3:     Output: SAT/UNSAT, depending on whether there exists an assignment extending τ that satisfies F
4:     while ∃ unit clause ∈ F do
5:         ℓ ← the unset literal in the unit clause
6:         τ[ℓ] ← true
7:     end while
8:     if F contains the empty clause then
9:         return UNSAT
10:    end if
11:    if all clauses in F are satisfied then
12:        Output τ
13:        return SAT
14:    end if
15:    ℓ ← some unset literal (based on variable ordering heuristic)
16:    if DPLL-recursive(F, τ[ℓ] ← true) = SAT then
17:        return SAT
18:    end if
19:    return DPLL-recursive(F, τ[ℓ] ← false)
20: end function
```

improve efficiency. When a conflict is encountered during the search process, Instead of immediately backtracking to a previous decision level and undoing assignments, CDCL performs a clause learning procedure[27].

The clause learning procedure aims to identify a new clause, called a conflict clause, that captures the cause of the conflict. This clause enables the solver to detect conflicts earlier in subsequent assignments[27].
Algorithm 5 is a typical implementation of the CDCL algorithm [27].

Contrary to the backtracking of DPLL, CDCL can backtrack to previous assignments other than the latest assignment; this property is called non-chronological backtracking.

3.3.2 Incomplete solvers

With larger and more complex problem instances, the complete solvers can no longer scale, resulting in a significant increase in running time and a combinatorial explosion. This has led researchers to develop new approaches for finding incomplete solutions in reasonable time, in other words, to try to find a compromise between exhaustive search and temporal effectiveness.

These algorithms do not explore the entire search space. They are stochastic and guided by heuristics to reach a mostly optimized solution. These solvers are generally metaheuristics. Among the plethora of existing metaheuristics, some principle ones are

Algorithm 5 Typical CDCL algorithm

Input: A CNF formula ϕ and a partial assignment ν
Output: SAT/UNSAT

```
1: function CDCL(φ, ν)
2:     if UnitPropagation(φ, ν) = CONFLICT then
3:         return UNSAT
4:     end if
5:     dl ← 0                                              ▷ Decision level
6:     while not AllVariablesAssigned(φ, ν) do
7:         (x, v) ←PickBranchingVariable(φ, ν)             ▷ Decide stage
8:         dl ← dl + 1          ▷ Increment decision level due to new decision
9:         ν ← ν ∪ (x, v)
10:        if UnitPropagation(φ, ν) = CONFLICT then        ▷ Deduce stage
11:            β ←ConflictAnalysis(φ, ν)                   ▷ Diagnose stage
12:            if β < 0 then
13:                return UNSAT
14:            else
15:                Backtrack(φ, ν, β)
16:            end if
17:            dl ← β       ▷ Decrement decision level due to backtracking
18:        end if
19:    end while
20:    return SAT
21: end function
```

presented within the first chapter.

3.4 Conclusion

Through this chapter, we have introduced the problem of Boolean Satisfiability as well as the interest that is brought to it by the scientific community.

We have presented some of the most popular SAT algorithms and solvers whose two main categories have been defined (complete and incomplete solvers).

Chapter 4

Integrated Parameter Genetic Algorithm

4.1 Introduction

Parameter initialization is one of the drawbacks of GAs, where the parameter search time is a considerable cost in the overall search process.
Through this work, we propose a new improved version of the genetic algorithm where parameters are integrated into the solution itself with the idea that a good solution is the one that leads to a better solution.
In this chapter, we present the main concepts developed for this new approach.

4.2 Parameter integration

The proposed methods rely on the idea of gene-level parameter integration. A solution, is represented, not just as a value assignment of problem variables but also as an evolutionary preference at the gene level. An agent may have different parameters for each gene which gives the potential for a granular optimization of the genetic parameters. Another advantage of this method is the dynamic nature of the integrated parameters, where these values have the potential to adapt, favoring values that maximize the fitness, where the optimal parameters may change depending on the current generation, where such thing is not possible in CGA.

This method leads to a wide range of possible implementations of genetic operators. For example, crossing two solutions with different crossover expressions leads to conflicts that require managing (this will be discussed in later sections), and a number of different operators will be presented.

Granted, this adds a layer of complexity that needs to be justified with better performance or desirable characteristics; these methods are tested against a canonical imple-

mentation of GA.

4.3 Agent definition

In classical GA (CGA), an agent is simply a solution, where each variable is mapped to a gene and the set of all genes is a chromosome.
In the proposed integrated parameter genetic algorithm (IPGA), each gene is equipped with two values: an expression value and a resilience value. The expression value is used in the crossover operator (a full description of its use is given in section 3.4.1.), and resilience is the inverse of mutation probability.
In this method, each variable expresses its mutation probability and its expression; the genetic parameters are integrated on a variable level.
Example : Given a problem instance P with n variables, an agent G is defined as follows(Equation 4.1 represents the structure of an IPGA agent):

$$G = \begin{pmatrix} s_0 & s_1 & \dots & s_i & s_n \\ e_0 & e_1 & \dots & e_i & e_n \\ r_0 & r_1 & \dots & r_i & r_n \end{pmatrix} \tag{4.1}$$

where s_i e_i r_i represent the solution, expression, and resilience of the variable at index i, respectively. An agent for a problem of n variables is a chromosome of shape $n \times 3$.

The integrated parameters are used mainly in the genetic operators, which leads to a diverse way of implementing them. The following is the set of developed genetic operators:

4.4 Genetic operators

This section presents the proposed IPGA's genetic operators.

4.4.1 Crossover

IPGA relies on a modified masked crossover operator. As described in the state-of-the-art section, masked crossover uses a mask vector to determine which of the two parents expresses each gene (variable).
In IPGA, each parent provides its own expression making the crossover process ambiguous, where a choice has to be made about which parent should express its gene.

Let E^1 and E^2 be expression vectors of P_1 and P_2, respectively.
E^1 and E^2 are said to be compatible if, $E^1 = \neg E^2$, the first parent expresses a gene while the other parent does not, as in figure 4.1. Contrarily, $E^1_i = E^2_i$ is a state of confusion (see figure 4.1), a decision should be taken on which parent should express its gene(i.e.

confusion resolution)(Figure 4.1 illustrates the case of confusion in parent expressions in the crossover process).

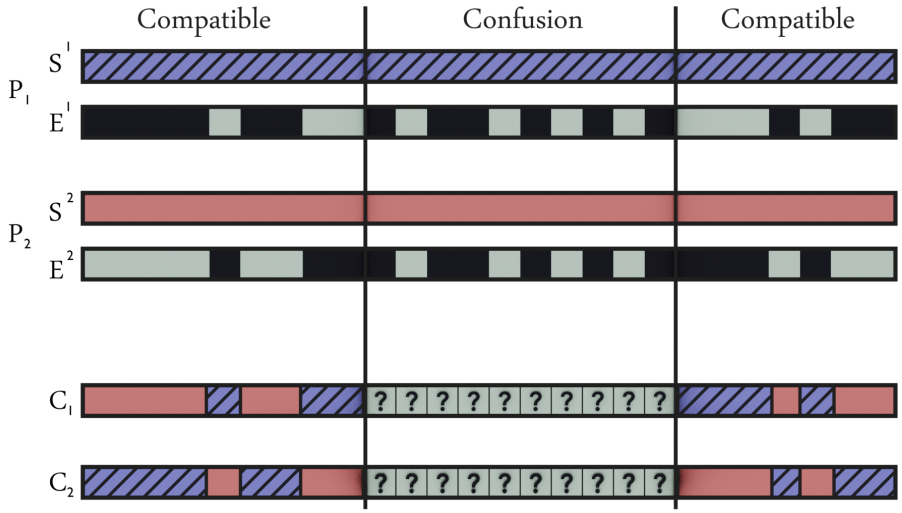


Figure 4.1: IPGA crossover, confusion vs compatible genes

4.4.2 Confusion resolution methods

A gene i is a confusion point if $E_i^1 = E_i^2$. Confusion points C is a set of gene indexes where $|C| < n$ and $\forall c \in C, c \in [0, n-1], E_c^1 = E_c^2$,with n the number of genes.
The choice of which parent's gene to express is done following resolution methods.

4.4.2.1 Simple resolution

As each parent pair results in two offspring, each child can be assigned a parent's expression as the final expression vector (Figure 4.2 is an example of simple confusion resolution).

4.4.2.2 Random confusion resolution

To resolve the confusion, two new expression vectors V^1 and V^2 are created, these vectors are the crossover masks after resolving the confusions, in this case a random parent will be selected to be expressive for each confusion point Equations 4.2 and 4.3 calculate the expression vectors after random confusion resolution:

$$\forall i \in [0, n-1], V_i^1 = ((E_i^1 \oplus E_i^2) \wedge E_i^1) \vee (\neg(E_i^1 \oplus E_i^2) \wedge a) \tag{4.2}$$

$$\forall i \in [0, n-1], V_i^2 = ((E_i^1 \oplus E_i^2) \wedge E_i^2) \vee (\neg(E_i^1 \oplus E_i^2) \wedge a) \tag{4.3}$$

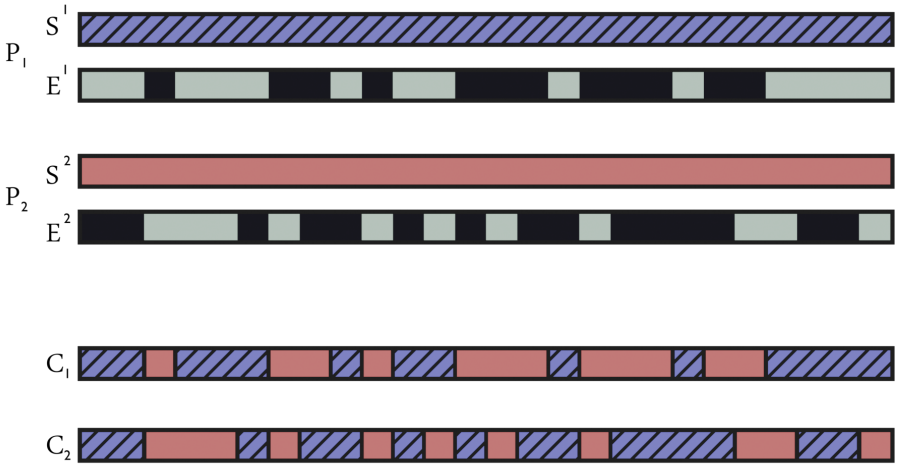


Figure 4.2: Simple confusion resolution

With a a random boolean value. In this case, each child C^1 and C^2 has a main parent. If $V_i = 1$, the main parent expresses its gene; otherwise, the secondary parent expresses its gene. This method can be implemented as follows(Algorithm 6)

Algorithm 6 Random confusion resolution crossover

Input: Two parent agents $P1$ and $P2$
Output: A new offspring agent

$newAgent \leftarrow$ Agent()
$F \leftarrow \text{XOR}(P1.e, P2.e)$
$C[F] \leftarrow P1.e[F]$
$C[\neg F] \leftarrow$ random()
for each g, c in $newAgent, C$ **do**
    if c **then**
        $gene.s \leftarrow P1.s$
        $gene.r \leftarrow P1.r$
    else
        $gene.s \leftarrow P2.s$
        $gene.r \leftarrow P2.r$
    end if
    $gene.e \leftarrow c$
end for
return $newAgent$

Figure 4.3 is an example of random confusion resolution crossover.

4.4.2.3 Resilience based confusion resolution

Instead of randomly selecting the expressive parent, we can select for a resilience characteristic. This method is a corrective method for the resilience accumulation problem,

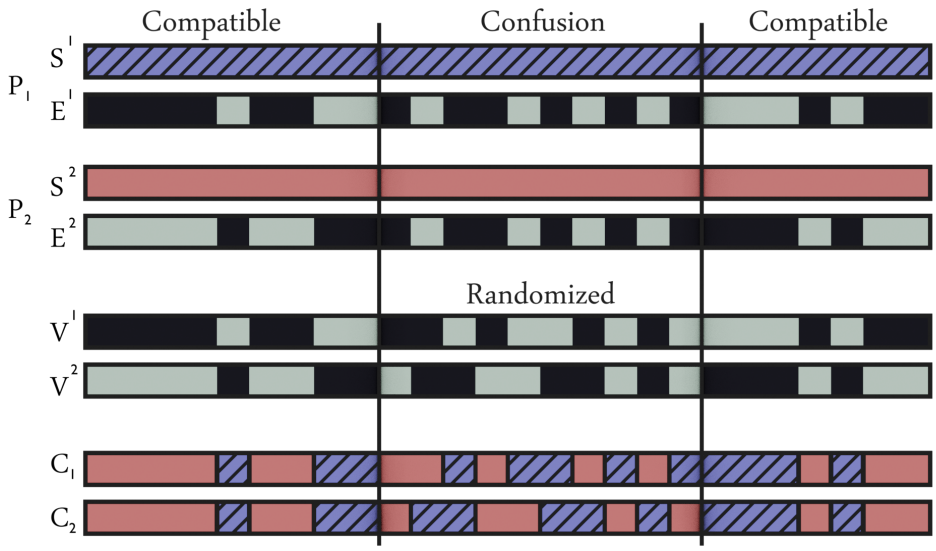


Figure 4.3: Random confusion resolution

where the agent with a lower resilience value is prioritized. Equations 4.4 and 4.5 calculate the expression vectors after random resilience based confusion resolution.

$$\forall i \in [0, n-1], V_i^1 = ((E_i^1 \oplus E_i^2) \wedge E_i^1) \vee (\neg(E_i^1 \oplus E_i^2) \wedge (R_i^1 > R_i^2))) \tag{4.4}$$

$$\forall i \in [0, n-1], V_i^2 = ((E_i^1 \oplus E_i^2) \wedge E_i^2) \vee (\neg(E_i^1 \oplus E_i^2) \wedge (R_i^2 >= R_i^1)) \tag{4.5}$$

With R^1 and R^2 resilience vectors(Algorithm 7 shows an implementation of resilience based confusion resolution).

Algorithm 7 Resilience based confusion resolution

Input: Two parent agents $P1$ and $P2$
Output: A new offspring agent

```
1: newAgent ← Agent()
2: F ← XOR(P1.e, P2.e)
3: C[F] ← P1.e[F]
4: C[¬F] ← compare(P1.r, P2.r)
5: for each g, c in newAgent, C do
6:     if c then
7:         gene.s ← P1.s
8:         gene.r ← P1.r
9:     else
10:        gene.s ← P2.s
11:        gene.r ← P2.r
12:    end if
13:    gene.e ← c
14: end for
15: return newAgent
```

4.4.2.4 Dominance resolution

The objective of an expression vector is to determine which of the two parents has better genes in order to propagate them to the next generation. This method proposes that a fitter solution carries a better expression vector.

A parent is dominant if its score is greater than the score of the other parent; a dominant parent's expression is used as the final expression vector for both offspring.

This method can be used in conjunction with random confusion resolution or resilience-based confusion resolution.

4.4.3 Mutation

IPGA mutation is based on bit flip mutation. As an agent is a value assignment of problem variables equipped with resilience and expression vectors, each one of these values should be affected by the mutation. This triplet (s_i, e_i, and r_i) can be treated as one unit; The mutation affects the three values at the same time (referred to as coupled mutation). Algorithm 8 shows an implementation of coupled mutation method

Algorithm 8 Coupled mutation

Input: An Agent A with n genes
Output: The mutated Agent A'

```
1: A' ← copy of A
2: for i ← 1 to n do
3:     if random() < 1 − A_i.r then
4:         A'_i.s ← ¬A_i.s
5:         A'_i.e ← ¬A_i.e
6:         A'_i.r ← resilience_mutate(A_i.r)
7:     end if
8: end for
9: return A'
```

Another method is to treat each value as a separate mutation case; the mutation of each value is determined separately. (referred to as disjoined mutation).

Algorithm 9 shows an implementation of disjoined method.

In both methods, the three values are mutated: expression and solution are binary values, and resilience is a real value.

When a gene i is determined to be mutated, s_i and e_i can be simply mutated as follows(Equations 4.6 and 4.7 describe the solution mutation and expression mutation, respectively):

$$s'_i = \neg s_i \tag{4.6}$$

$$e'_i = \neg e_i \tag{4.7}$$

Algorithm 9 Disjoined mutation

Input: An agent *Agent*
Output: An agent *Agent* with some genes mutated

```
1: for all genes gene in Agent do
2:     if random() < 1 − gene.r then
3:         gene.s ← ¬ gene.s
4:     end if
5:     if random() < 1 − gene.r then
6:         gene.e ← ¬ gene.e
7:     end if
8:     if random() < 1 − gene.r then
9:         gene.r ← resilience_mutate(gene.r)
10:    end if
11: end for
12: return Agent
```

But resilience is a real-coded value $r_i \in [0, 1]$, a simple mutation is not possible. This value can be mutated according to real-valued mutation methods described in section 2.8. This difference is noted by the use of resilience_mutate function in the above pseudocode.

In addition to the RCGA mutations, normal and Cauchy mutations can also be used. These methods shift the chromosome x by a value α sampled from a probability distribution. To fit the use case of resilience mutation, very high or low values of resilience should be avoided; thus, we add a lower and upper bound Lb Ub to limit the range of mutation.

Algorithm 10 resiliance mutation

Input: resiliance value x
Output: mutated resiliance value x'

```
1: x' ← min(max(x + alpha, Lb), Ub)
2: return x'
```

Comparisons between the performance of normal and cauchy mutations showed that cauchy mutation is more capable of escaping local optima and leading to a more diverse population[28].

4.4.4 Selection

The selection methods of CGA are directly applicable to IPGA, as the selection process is not affected by IPGA's structure.

4.5 Conclusion

Through this chapter, we proposed an improved version of the genetic algorithm called IPGA where the crossover and mutation parameters are integrated into the solution itself,

unifying the problem and genetic parameter search spaces. This representation lead to the development of its own set of genetic operators.

The test results of IPGA are discussed in chapter 6.

Chapter 5

Solver Implementation

5.1 Introduction

This chapter discusses important details and notions used while implementing the proposed MAXSAT solver.
Firstly, a simple solver implementation is mentioned then contrasted with the chosen optimized implementation. A fast and reliable framework is crucial for the implementation and testing of the proposed methods.

5.2 Testing system architecture

The simplest approach for implementing test algorithms is to create an independent implementation for each solving method and then proceed with the tests. Using this approach leads to slow development as genetic algorithms are based on operators with numerous implementations. On the other hand, Such an implementation is surely redundant, as genetic algorithms share the structure presented in figure 5.1:

Any genetic algorithm is implemented in the above general structure; in some cases, the order of operations between mutation and crossover is reversed. A better testing system is needed to be able to rapidly iterate through ideas. A good testing system should enable users to rapidly iterate through the experimentation process, from implementation to execution. A non-optimized or complicated implementation, coupled with hardware limitations, leads to longer execution times and, thus, a slower testing framework.

Another important aspect of a testing system is the logging and comparing of results. Each test run should produce a trace describing the implementation (used operators), the used parameters, and the obtained result, with the goal of future comparison.

Leveraging the special structure of GAs, we can define a general framework to test different implementations by abstracting general processes into a library. This way, we can optimize the performance using parallel computing and other methods without adding

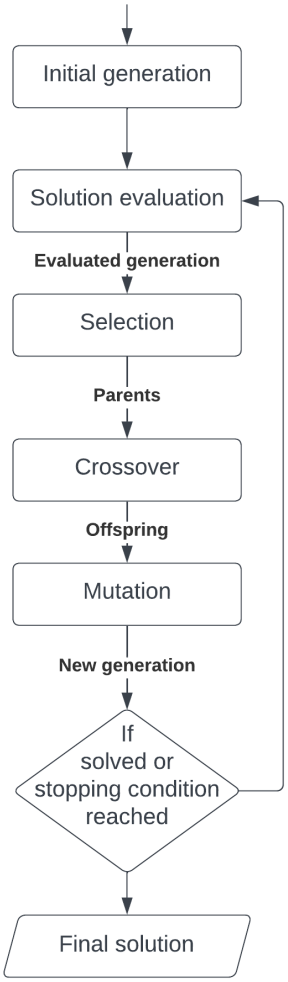


Figure 5.1: Flow chart of GA algorithm

unnecessary complexity on the development end.

Figure 5.2 describes the general structure of the testing system.

- A solver takes a set of operators and a SAT instance as input;
- Operators (selection, crossover, mutation, and evaluation) are defined as functions;
- A selection function takes a set of agents as input and returns a set of agent couples (parents);
- Crossover takes a set of agent pairs and returns the new generation (the result of parents crossover);
- Mutation takes an agent as input and returns it after mutation;
- The evaluation function defines the used fitness function; and a SAT instance is an implementation of the SAT solution evaluation described earlier.

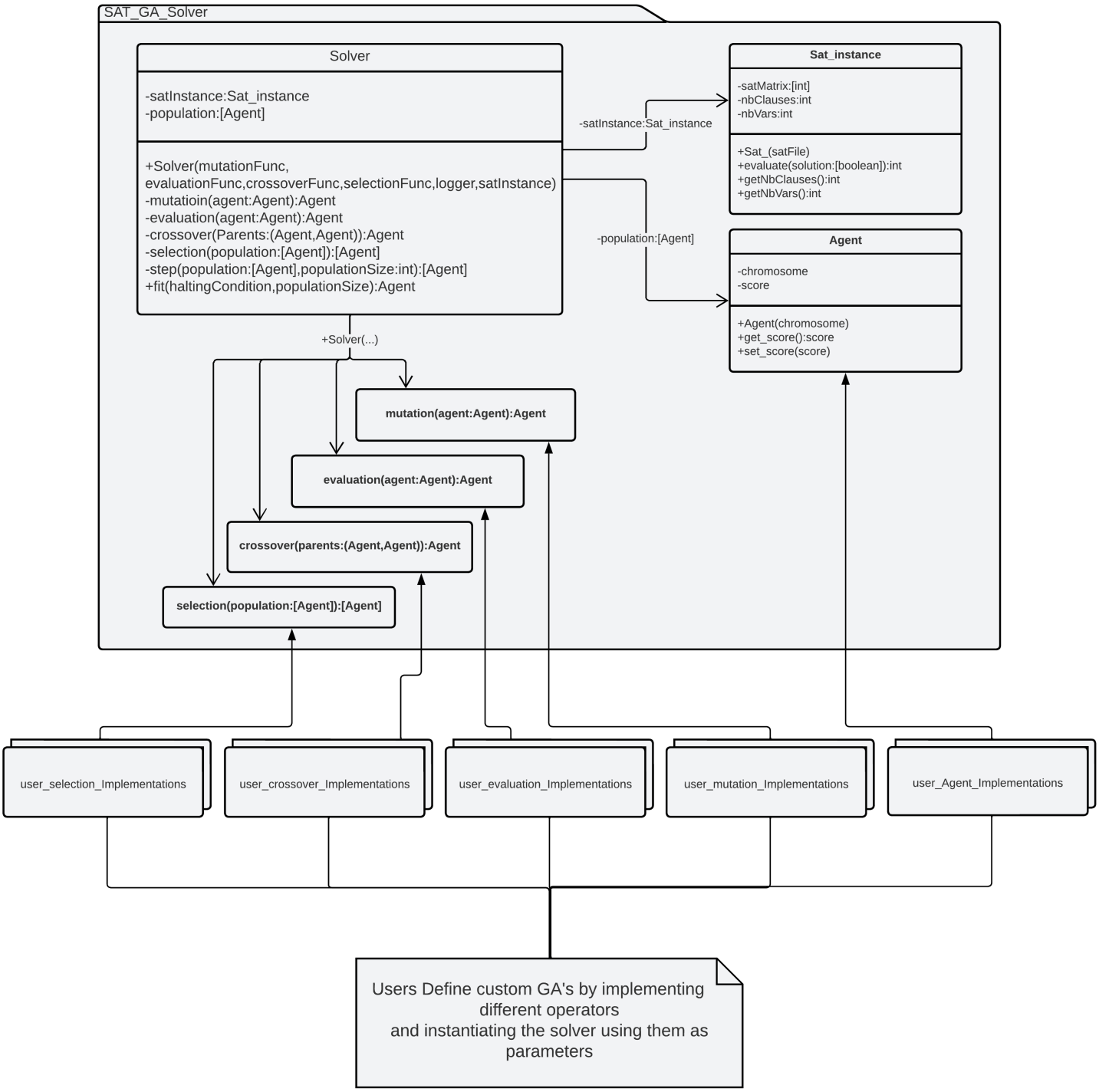


Figure 5.2: Project structure

5.3 parallelism

Parallelism is the process of splitting a task into independently computed subprocesses with the goal of reducing execution time. For example, calculating the fitness value of a solution in a generation depends only on the solution itself; thus, the evaluation step of a GA is a parallelizable process. On the other hand, selection and crossover can not be parallelized, as the crossover process depends on the selected parents (a serial process).

The final goal of optimization is to minimize execution time with the available hardware. In addition to the time performance gains associated with parallelism, we can manage heavier loads and more efficiently exploit the used hardware.
Parallelism was achieved in this project using Dask.

Dask provides a scalable parallelization framework where multiple computers can be used simultaneously as one compute unit called a cluster. This structure is scalable and easily intergrated. It uses the same API as Python's numpy and pandas, which influenced the use of numpy as the implementation of choice for Sat instances and solution representations.

Workers are the main computation units in Dask; they can be defined on one single machine (threads or cores) or in a network of machines. Computation load is distributed across workers using a scheduler; the number of used workers depends on hardware specs, and the number of used machines.

5.4 Logging

Relying on a statistical representation of algorithm performance is maybe enough when analyzing simple methods, where a general view of the performance is satisfactory to compare and understand the functionality of different implementations. Generally, performance visualization is represented as a change in score over generation. Even though this representation shows the overall performance, it does not show the difference in behavior between algorithms; two algorithms may have the same score/generation graph while behaving in very different ways.

To better understand the proposed methods, a per-variable analysis is needed. As these methods behave on a variable level, a log should show the effects of different operators even if the performance is similar; a per-variable level analysis shows changes in trends and stagnation. On the other hand, visualizing the algorithm's behavior in this manner helps in the development phase, where it will be easier to recognize implementation errors. To do so, we used heatmaps.

Heatmaps have a wide range of uses, from webpage performance to DNA expression analysis. Heatmaps represent variable values as a color intensity in a two-dimensional plot; they show the relationship between variables through changes in intensity patterns.

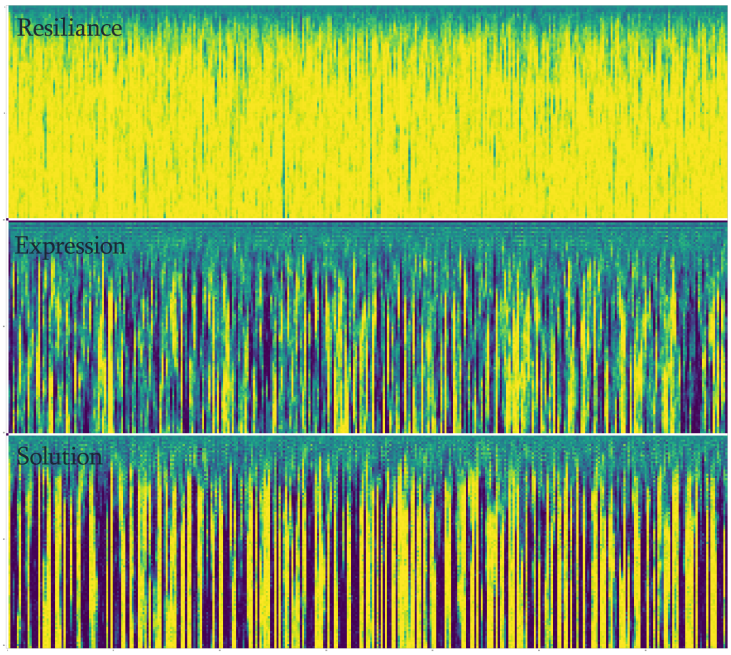


Figure 5.3: Heatmap example

Heatmaps were recognized as a valuable visualization tool for optimization algorithms[29]. Figure 5.3 is an example of a heatmap output.

The heatmap is split into three sections: resilience, expression, and a solution section. Each row in a section represents a generation, and each column represents a variable. Two solutions can be compared by comparing their heatmap logs, these logs give insight into the changing trends throughout the generations.

5.5 MAX-SAT implementation and instance pre-processing

The max-sat problem, as discussed in the state of the art, is an optimization problem with the goal of maximizing the number of satisfied clauses.
Implementing a solution for such a problem requires not only an efficient solver but also an efficient representation of the problem instance. Furthermore, solution verification is a crucial part of any metaheuristic, including genetic algorithms. The pre-processing of MAXSAT instances and the optimization methods are discussed in this section.

Most satisfiability benchmarks codify instances in ".cnf" files,the following is an example of a ".cnf" file :

```
c <comments>
p cnf 4 3
-1 3 -4 0
1 4 0
```

```
-2 3 4 0
```

Each file includes a parameter line that starts with p and contains two consecutive integers n and m, variable count and clause count, respectively.
In this example, $n = 4$ and $m = 3$, each clause is defined by a subset of variables $a \in \alpha$ denoted by their indices, where a negative index means that the variable is negated, "0" is used as a delimiter between clauses, and a cnf file may contain comment lines prefixed by a lowercase "c".

The above instance represent the following boolean formula:

$$(\neg\alpha_1 \vee \alpha_3 \vee \neg\alpha_4) \wedge (\alpha_1 \vee \alpha_4) \wedge (\neg\alpha_2 \vee \alpha_3 \vee \alpha_4) \tag{5.1}$$

Instance files are compiled into a matrix representation, where an instance P is a matrix of size $n * m$, where $P^i_j \in \{-1, 0, 1\}$. If $P^i_j = 0$, the i^th variable is not included in the j^th clause, $P^i_j = 1$, the i^th variable is included in the j^th clause and if $P^i_j = -1$ the negation of the i^th variable is included in the j^th clause.

$$P = \begin{pmatrix} -1 & 0 & 1 & -1 \\ 1 & 0 & 0 & 1 \\ 0 & -1 & 1 & 1 \end{pmatrix} \tag{5.2}$$

This choice will be justified in the following section. Upon analysis of the used benchmark, we found that 99% of values in instance matrices are zero.

Arithmetic operations on a matrix require storage and access to each value; thus, a large matrix requires both more memory and execution time to operate on. This lead us to use a sparse matrix. In sparse matrix, we can leverage its structure to optimize execution time and memory use, where non-zero values are stored in an alternate structure, leading to a more compact data representation and thus using less memory.
There are multiple matrix compression algorithms to optimize both time and memory. A method's performance is not only related to the compressed matrix but also to the use case, where methods optimize for specific arithmetic operations or a problem-specific matrix structure. Compressed sparse row (CSR) is one of the most commonly used compression methods, as it performs well with non-regularly structured data. In the table below, we present the most commonly used methods and a brief description(see table 5.1).

To choose the best method for our purposes, in the next section, each described method will be tested and then compared.

method	description
DOK	stores each non zero value as a dictionnary of keys where a key is the index of the value
LIL	list of lists where each index in the main list represents a row containing a list of column/value nodes
COO	a list of row column value triplets of non zero values
CSR	compressed sparse row. Decompose the matrix into value column index and non zero count matricies
CSC	compressed sparse column. Same representation as CSR but column wise

Table 5.1: Sparse matrix compression methods

A solution S with $|S| = n$, is an assignment of a boolean value to each variable, $S_i \in \{0, 1\}$, if $S_i = 0$, the i^th variable is set to false, if $S_i = 1$, the i^th variable is set to true.

5.5.1 Optimization of solution evaluation

To iterate faster over proposed methods, a fast evaluation method is needed, whereas a slow implementation limits the size and number of tests we can run.In this section, multiple approaches are presented and tested.
First, intuitively, we can determine the number of satisfied clauses given a boolean value assignment S by simply verifying the satisfiability of each clause one by one, reminding that a clause is satisfied if one literal is true. Algorithm 11 shows an intuitive implementation of solution evaluation:

To optimize the implementation, the evaluation method is rewritten with matrix operations. Such an approach is beneficial considering the limitations of using for loops in a high-level language like Python. Even though using a lower-level language may lead to better performance. The benefits of the tools available in a high-level language justify its use.

This method evaluates a solution as follows:

$$V = C + P \cdot S' \tag{5.3}$$

A clause i is satisfied if $V_i > 0$ where C is the number of variables in a clause,$C_i = \|V_i\|$ and $C_i > 0$, P instance matrix and S' is calculated as follows:

Algorithm 11 Count the number of satisfied clauses in a CNF formula

Input: A CNF formula P with n variables
Input: An assignment S of truth values to the variables in P
Output: The number of clauses in P that are satisfied by S

```
satisfied_count ← 0
for each clause C in P do
    for i in [0, n] do
        if (C_i = 1 and S_i = 1) or (C_i = −1 and S_i = 0) then
            satisfied_count ← satisfied_count + 1
            break
        end if
    end for
end for
return satisfied_count
```

$$S' = \begin{cases} 1 \text{ if } S = 1 \\ -1 \text{ if } S = 0 \end{cases} \tag{5.4}$$

This method is tested using the sparse matrix compression algorithms described in the previous section.
The test goes as follows: Define a set of cnf instances with an increasing number of clauses ranging from 10 to 40000 clauses; these instances are created by truncating the "bmc-ibm-7.cnf" instance to a size starting with 10 and with 1000 clause increments; a test solution is randomly generated; and the execution time is averaged over 3 runs for each instance/solution.

Based on the test result, with clause counts <20000, the best method is sparse_csr, with higher values, sparse_csc becomes marginally better, because most instances are smaller than 20000 clauses and the difference between csc and csr for instances larger than 20000 is negligible. sparse_csr is The beneficial method to be used.

Further optimization is possible; the chosen method can be used in parallel. Even though this may result in improved performance, the fact that the performance is already very good makes the complexity of implementation unnecessary.

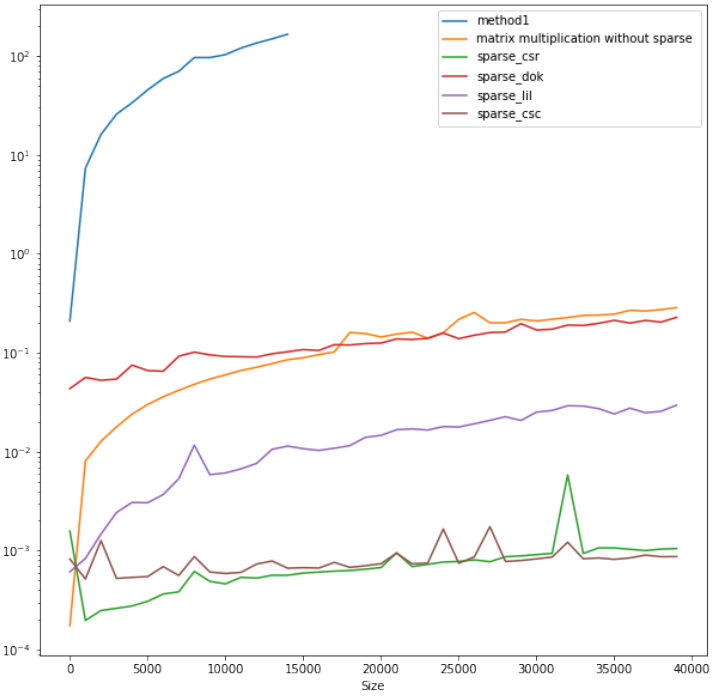


Figure 5.4: Performance test results (log scale)

5.6 Conclusion

Throught this chapter, we have defined a framework for implementing and testing the proposed approach. In the next chapter, we will expose and discuss the experiments results .

Chapter 6

Experimetns and results

6.1 Introduction

In this section, we describe the experiments and results of the proposed methods, all tests are done on a well established benchmark To be able to compare the results with the established methods, a set of CGAs is tested against the same benchmarks (results in the CGA tests section). And finally, the results of IPGA are discussed and compared against the best-performing CGA in the IPGA tests.

6.2 Testing environment and Benchmarks description

All the tests presented in this chapter are done in Desktop PC with Intel Core i5 (2.30 GHz) with 4 cores and 8 GB DDR3 RAM.

The experiments were done on a set of benchmarks that cover a large set of different characteristics in the structure of a SAT problem.
The SATLIB benchmark, which is widely used [30][31], contains 13 instances; the table below shows each instance and its number of clauses and number of variables(Table 6.1) Because of hardware limitations, testing on instances 6,8,9,10,11 and 12 was not possible.

File name	number of variables	number of clauses
bmc-ibm-1.cnf	9686	55870
bmc-ibm-2.cnf	2810	11683
bmc-ibm-3.cnf	14930	72106
bmc-ibm-4.cnf	28161	139716
bmc-ibm-5.cnf	9396	41207
bmc-ibm-6.cnf	51639	368352
bmc-ibm-7.cnf	8710	39774
bmc-ibm-10.cnf	59056	323700
bmc-ibm-11.cnf	32109	150027
bmc-ibm-12.cnf	39598	194778
bmc-ibm-13.cnf	13215	65728
bmc-galileo-8.cnf	58074	294821
bmc-galileo-9.cnf	63624	326999

Table 6.1: SATLIB benchmark instance characteristics

6.3 Genetic algorithm experiments

The goal of these experiments is to establish a comparison baseline to compare IPGA's results against.
Because of the interconnected nature of GA's operators, different combinations of operators result in different behaviors and, thus, different performance. To have a comprehensive baseline, all the combinations of the following operators need to be tested:

- One point crossover
- Two point crossover
- Kpoint crossover
- Mask crossover
- Roulette selection
- Tournament selection
- Rank selection
- Bitflip mutate
- Inversion mutate
- Scramble mutation

6.3.1 Genetic algorithm parameter choice

The one-point, two-point, and k-point crossovers have the crossover points as parameters. These parameters are fixed to the following random points for each operator. Figure 6.2 lists the names of the crossover operators and their parameter:

The bitflip mutation takes a mutation probability as input, and the following mutation probabilities are tested. Figure 6.3 lists the names of the mutation operators and their parameters:

Operator name	points
onepoint_cross_07	0.7
kpoint_cross_p	{0.2,0.4,0.6,0.8}
twopoint_cross_01_05	{0.1,0.5}

Table 6.2: Crossover operator names

Operator name	Mutation probability
bitflip_02	0.2
bitflip_002	0.02
bitflip_03	0.3

Table 6.3: Mutation operator names

Other parameters were chosen while keeping the hardware limitations in mind: The population size is fixed at 100 agents, and the maximum number of generations is fixed at 100 generations.

6.3.2 Genetic algorithm test results

The following are the test results for the bmc-ibm benchmark. Tables 6.4 and 6.5 exhibit the score and mean score reached by each combination of CGA operators per test file and their execution time per generation, score is calculated as the percentage of satisfied clauses:

After an analysis of the results, we can say that the best operator combination is mask crossover, bit flip mutation, and tournament selection, with a mutation probability of 0.02.

Table 6.6 gives the ranking of each operator based on the maximum score reached using a GA, including that operator.

Additional information regarding the benchmark can be extracted by comparing the differences in average max score for each instance (named mean in the results table). This value can be interpreted as an indicator of the difficulty of each problem instance; given two SAT instances, the one with a higher average max score is the easier one.
Based on this observation, we can rank the instances based on their difficulty(Figure 6.1 gives the problem instance ranking):

On the other hand, the max score value is negatively correlated with both variable count and clause count, -0.62 and -0.58, respectively; in other words, the more variables and clauses in an instance, the lower the maximum score.

cross	mutate	select	bmc-ibm-1.cnf			bmc-ibm-2.cnf			bmc-ibm-3.cnf			bmc-ibm-4.cnf		
			score	mean	time(s)	score	mean	time(s)	score	mean	time(s)	score	mean	time(s)
kpoint_cross_p	bitflip_002	tournament_4	0.8306	0.8270	0.1767	0.8620	0.8539	0.1683	0.8306	0.8266	0.0004	0.8177	0.8146	0.6183
		roulette	0.7871	0.7775	0.1867	0.8055	0.7881	0.0950	0.7975	0.7888	0.2017	0.7908	0.7845	0.3600
		rank	0.8187	0.8123	0.1767	0.8464	0.8360	0.1567	0.8171	0.8133	0.2000	0.8099	0.8062	0.3483
	bitflip_003	tournament_4	0.8222	0.8165	0.2800	0.8567	0.8454	0.0950	0.8221	0.8179	0.5433	0.8136	0.8088	0.3600
		roulette	0.7893	0.7785	0.2717	0.8043	0.7850	0.1583	0.7956	0.7905	0.2067	0.7937	0.7846	0.3567
		rank	0.8087	0.8028	0.1783	0.8349	0.8277	0.0967	0.8144	0.8091	0.2017	0.8058	0.8013	0.4817
	bitflip_02	tournament_4	0.7941	0.7844	0.4100	0.8149	0.7954	0.1050	0.8012	0.7941	0.3600	0.7972	0.7889	0.5550
		roulette	0.7899	0.7769	0.1867	0.8082	0.7848	0.1300	0.7987	0.7886	0.2067	0.7913	0.7838	0.3550
		rank	0.7931	0.7802	0.2017	0.8076	0.7917	0.0950	0.7991	0.7918	0.2050	0.7934	0.7867	0.5333
	inversion	tournament_4	0.8102	0.7989	0.1217	0.8387	0.8202	0.0833	0.8160	0.8069	0.1333	0.8068	0.7996	0.3600
		roulette	0.7904	0.7774	0.2067	0.8079	0.7862	0.1467	0.7980	0.7893	0.1317	0.7906	0.7845	0.2133
		rank	0.7999	0.7888	0.1667	0.8245	0.8057	0.1233	0.8069	0.7986	0.1267	0.8032	0.7935	0.4100
	scramble	tournament_4	0.8150	0.8009	0.2167	0.8380	0.8183	0.0817	0.8137	0.8049	0.5050	0.8063	0.7990	0.2233
		roulette	0.7926	0.7765	0.1617	0.8072	0.7852	0.0817	0.7970	0.7887	0.1300	0.7947	0.7851	0.3800
		rank	0.8007	0.7891	0.1700	0.8267	0.8059	0.1483	0.8078	0.7993	0.2550	0.8014	0.7919	0.2250
mask_cross	bitflip_002	tournament_4	0.8517	0.8454	0.1650	0.8738	0.8653	0.1567	0.8460	0.8368	0.3533	0.8269	0.8230	0.3850
		roulette	0.7945	0.7799	0.1783	0.8021	0.7804	0.1133	0.8245	0.8199	0.2200	0.7905	0.7840	0.4283
		rank	0.8350	0.8265	0.2683	0.8548	0.8446	0.1000	0.7977	0.7914	0.3400	0.8171	0.8113	0.3833
	bitflip_003	tournament_4	0.8352	0.8281	0.2733	0.8626	0.8497	0.0967	0.8304	0.8244	0.2183	0.8176	0.8135	0.3817
		roulette	0.7910	0.7792	0.1733	0.8007	0.7796	0.1017	0.8184	0.8125	0.2283	0.7923	0.7846	0.4667
		rank	0.8228	0.8101	0.1667	0.8491	0.8320	0.1567	0.7966	0.7903	10.3533	0.8088	0.8031	0.3850
	bitflip_02	tournament_4	0.7984	0.7824	0.1583	0.8163	0.7960	0.1583	0.8032	0.7941	0.2217	0.7998	0.7892	0.4467
		roulette	0.7893	0.7765	0.1850	0.8102	0.7861	0.1050	0.8017	0.7925	0.2267	0.7933	0.7855	0.3950
		rank	0.7969	0.7800	0.1733	0.8136	0.7913	0.1117	0.7982	0.7884	0.3200	0.7949	0.7867	0.4467
	inversion	tournament_4	0.8302	0.8141	0.1417	0.8524	0.8281	0.0850	0.8246	0.8142	0.1467	0.8121	0.8025	0.3367
		roulette	0.7941	0.7810	0.1267	0.8089	0.7861	0.0983	0.8061	0.7976	0.1567	0.7920	0.7847	0.2567
		rank	0.8068	0.7942	0.1300	0.8288	0.8084	0.1450	0.7979	0.7895	0.3050	0.8013	0.7935	0.2517
	scramble	tournament_4	0.8243	0.8076	0.1233	0.8507	0.8293	0.0850	0.8238	0.8132	0.1517	0.8157	0.8068	0.3567
		roulette	0.7890	0.7791	0.2333	0.8129	0.7850	0.1083	0.8101	0.7984	0.1533	0.7906	0.7838	0.2550
		rank	0.8104	0.7974	0.1350	0.8302	0.8090	0.0850	0.8036	0.7908	0.1633	0.8018	0.7953	0.3250
onepoint_cross_07	bitflip_002	tournament_4	0.8309	0.8264	0.2667	0.8636	0.8545	0.0967	0.8287	0.8255	0.4183	0.8164	0.8137	0.4250
		roulette	0.7887	0.7766	0.1717	0.8011	0.7839	0.0950	0.7970	0.7900	0.2017	0.7915	0.7853	0.3567
		rank	0.8169	0.8111	0.2283	0.8472	0.8360	0.1300	0.8181	0.8136	0.2000	0.8073	0.8045	0.4367
	bitflip_003	tournament_4	0.8233	0.8173	0.1700	0.8541	0.8443	0.1567	0.8234	0.8185	0.2117	0.8112	0.8076	0.3550
		roulette	0.7867	0.7774	0.1733	0.8085	0.7829	0.0917	0.8007	0.7931	0.2017	0.7917	0.7843	0.4617
		rank	0.8123	0.8040	0.2650	0.8418	0.8292	0.0950	0.8145	0.8075	0.3533	0.8063	0.8007	0.3533
	bitflip_02	tournament_4	0.7926	0.7839	0.1717	0.8134	0.7956	0.1583	0.8028	0.7942	0.3333	0.7961	0.7889	0.3550
		roulette	0.7920	0.7779	0.2650	0.8073	0.7819	0.0967	0.7971	0.7887	0.2050	0.7945	0.7849	0.3583
		rank	0.7882	0.7805	0.1683	0.8191	0.7912	0.1100	0.7990	0.7924	0.3700	0.7947	0.7870	0.4733
	inversion	tournament_4	0.8129	0.7982	0.1200	0.8415	0.8192	0.1000	0.8161	0.8065	0.1250	0.8064	0.7980	0.2150
		roulette	0.7871	0.7779	0.1683	0.8066	0.7827	0.0783	0.7967	0.7888	0.1283	0.7904	0.7844	0.3300
		rank	0.8096	0.7953	0.2267	0.8330	0.8103	0.1417	0.8054	0.7981	0.5850	0.7980	0.7918	0.2200
	scramble	tournament_4	0.8106	0.7986	0.1250	0.8451	0.8229	0.0833	0.8115	0.8037	0.1317	0.8077	0.7985	0.2233
		roulette	0.7881	0.7774	0.1350	0.8052	0.7816	0.1683	0.8004	0.7888	0.1350	0.7923	0.7851	0.3317
		rank	0.8070	0.7925	0.1250	0.8298	0.8087	0.1500	0.8043	0.7967	0.1333	0.8013	0.7927	0.2300
twopoint_cross_01_05	bitflip_002	tournament_4	0.8338	0.8267	0.1600	0.8649	0.8559	0.0850	0.8300	0.8266	0.2067	0.8188	0.8150	0.5150
		roulette	0.7896	0.7787	0.1183	0.8066	0.7871	0.1267	0.7978	0.7897	0.2600	0.7917	0.7846	0.3500
		rank	0.8169	0.8105	0.1600	0.8477	0.8389	0.0900	0.8186	0.8133	0.1933	0.8099	0.8048	0.3417
	bitflip_003	tournament_4	0.8258	0.8189	0.1750	0.8504	0.8412	0.1100	0.8204	0.8166	0.3100	0.8127	0.8086	0.5167
		roulette	0.7834	0.7747	0.1700	0.8132	0.7879	0.1500	0.7980	0.7893	0.1900	0.7912	0.7837	0.3517
		rank	0.8101	0.8052	0.1683	0.8363	0.8252	0.1450	0.8128	0.8080	0.2633	0.8056	0.8017	0.6167
	bitflip_02	tournament_4	0.7944	0.7830	0.1150	0.8178	0.7964	0.1417	0.8001	0.7934	0.3100	0.7956	0.7897	0.5467
		roulette	0.7937	0.7783	0.1150	0.8125	0.7839	0.0883	0.7970	0.7894	0.2050	0.7912	0.7845	0.3500
		rank	0.7819	0.7819	0.1283	0.8118	0.7911	0.0967	0.8000	0.7924	0.2183	0.7945	0.7870	0.3517
	inversion	tournament_4	0.8125	0.7991	0.1317	0.8390	0.8203	0.0733	0.8153	0.8065	0.1250	0.8095	0.8006	0.2133
		roulette	0.7864	0.7757	0.1250	0.8155	0.7882	0.0867	0.7984	0.7888	6.2767	0.7912	0.7851	0.3417
		rank	0.8037	0.7910	0.5250	0.8314	0.8089	0.0717	0.8047	0.7966	0.1817	0.8022	0.7940	0.2083
	scramble	tournament_4	0.8097	0.7991	0.1600	0.8453	0.8258	0.1483	0.8142	0.8059	0.1300	0.8055	0.7970	0.2100
		roulette	0.7855	0.7753	0.1600	0.8003	0.7831	0.0867	0.7995	0.7889	0.2500	0.7931	0.7846	0.3300
		rank	0.8033	0.7908	0.1350	0.8245	0.8015	0.0700	0.8078	0.8005	0.1600	0.8021	0.7938	0.2250

Table 6.4: CGA test results

cross	mutate	select	bmc-ibm-5.cnf			bmc-ibm-7.cnf			bmc-ibm-11.cnf			bmc-ibm-12.cnf			bmc-ibm-13.cnf		
			score	mean	time(s)	score	mean	time(s)	score	mean	time(s)	score	mean	time(s)	score	mean	time(s)
kpoint_cross_p	bitflip_002	tournament_4	0.8465	0.8420	0.1617	0.8388	0.8342	0.1683	0.8246	0.8217	0.3983	0.8204	0.8184	0.3250	0.8324	0.8287	0.1933
		roulette	0.8096	0.7907	0.1617	0.7945	0.7823	0.2283	0.8015	0.7945	0.9717	0.7996	0.7944	0.3300	0.8026	0.7903	0.3450
		rank	0.8347	0.8277	0.2867	0.8225	0.8171	0.1600	0.8164	0.8139	0.4050	0.8129	0.8095	0.3367	0.8212	0.8164	0.1850
	bitflip_003	tournament_4	0.8375	0.8329	0.1633	0.8303	0.8226	0.2533	0.8189	0.8159	0.4033	0.8170	0.8134	0.3267	0.8263	0.8201	0.1883
		roulette	0.8034	0.7837	0.2533	0.7994	0.7853	0.1650	0.8005	0.7954	0.4017	0.8004	0.7933	0.3233	0.7992	0.7884	0.1667
		rank	0.8300	0.8214	0.1583	0.8212	0.8120	0.1833	0.8148	0.8102	1.0317	0.8096	0.8064	0.3267	0.8171	0.8096	0.1867
	bitflip_02	tournament_4	0.8123	0.7954	0.1650	0.8006	0.7904	0.1950	0.8038	0.7987	0.4050	0.8023	0.7964	0.3283	0.8034	0.7946	0.1917
		roulette	0.8047	0.7837	0.1633	0.7969	0.7840	0.1600	0.8024	0.7949	0.9783	0.7999	0.7923	0.3350	0.7956	0.7876	0.1867
		rank	0.8085	0.7904	0.2950	0.8045	0.7874	0.1600	0.8039	0.7968	0.4067	0.8023	0.7946	0.3383	0.7999	0.7908	0.1717
	inversion	tournament_4	0.8243	0.8107	0.1200	0.8192	0.8063	0.1150	0.8146	0.8049	0.2400	0.8128	0.8047	0.4317	0.8185	0.8083	0.1183
		roulette	0.8064	0.7824	0.1100	0.7951	0.7818	0.0983	0.8024	0.7945	0.2450	0.7984	0.7926	0.4350	0.7979	0.7880	0.1183
		rank	0.8197	0.8034	0.2517	0.8107	0.7958	0.1150	0.8072	0.8012	0.2400	0.8033	0.7977	0.3833	0.8085	0.7994	0.1217
	scramble	tournament_4	0.8270	0.8126	0.1133	0.8190	0.8047	0.1317	0.8121	0.8065	0.7017	0.8101	0.8032	0.4633	0.8137	0.8047	0.1233
		roulette	0.8088	0.7860	0.1133	0.7963	0.7841	0.1200	0.8001	0.7942	0.2400	0.7995	0.7929	0.0333	0.7988	0.7881	0.2050
		rank	0.8247	0.8078	0.2167	0.8071	0.7960	0.1300	0.8098	0.8021	0.2383	0.8057	0.8000	0.2133	0.8086	0.7970	0.1217
mask_cross	bitflip_002	tournament_4	0.8584	0.8522	0.2667	0.8528	0.8459	0.2567	0.8317	0.8285	0.4683	0.8310	0.8272	0.5367	0.8480	0.8425	0.2800
		roulette	0.8073	0.7879	0.1800	0.7975	0.7851	6.1717	0.8004	0.7937	0.5433	0.8024	0.7931	0.3633	0.7985	0.7871	0.2000
		rank	0.8424	0.8344	0.1983	0.8360	0.8281	0.1733	0.8252	0.8197	0.4417	0.8203	0.8154	0.3667	0.8326	0.8250	0.2000
	bitflip_003	tournament_4	0.8258	0.8117	68.0600	0.8378	0.8317	0.2083	0.8266	0.8214	0.5033	0.8236	0.8197	0.3617	0.8321	0.8274	0.3367
		roulette	0.8089	0.7866	0.1800	0.7951	0.7812	0.1683	0.8028	0.7940	0.4467	0.7998	0.7926	0.3633	0.7971	0.7876	0.2050
		rank	0.8331	0.8243	0.1950	0.8243	0.8157	0.1667	0.8181	0.8119	0.4667	0.8147	0.8098	0.5400	0.8193	0.8137	0.1983
	bitflip_02	tournament_4	0.8145	0.7961	0.1900	0.8032	0.7910	0.1717	0.8067	0.7985	0.5217	0.8013	0.7958	0.5167	0.8041	0.7940	0.4017
		roulette	0.8095	0.7819	0.1667	0.7962	0.7821	0.2467	0.8022	0.7942	0.4400	0.7996	0.7922	0.3617	0.8006	0.7894	0.2133
		rank	0.8055	0.7889	0.1867	0.8004	0.7866	0.1750	0.8046	0.7966	0.4433	0.8020	0.7947	0.5267	0.8030	0.7920	0.2100
	inversion	tournament_4	0.8391	0.8228	0.1283	0.8336	0.8169	0.1250	0.8200	0.8121	0.6083	0.8165	0.8095	0.2500	0.8276	0.8160	0.1400
		roulette	0.8031	0.7859	0.2933	0.7945	0.7825	0.1500	0.8023	0.7951	0.2933	0.7966	0.7918	0.4050	0.7987	0.7891	0.2133
		rank	0.8268	0.8103	0.1300	0.8108	0.7952	0.1267	0.8082	0.8014	0.3017	0.8050	0.7984	0.2500	0.8135	0.8018	0.1400
	scramble	tournament_4	0.8346	0.8207	0.2350	0.8313	0.8135	0.1817	0.8226	0.8132	0.5900	0.8175	0.8083	0.2450	0.8275	0.8147	0.1433
		roulette	0.8044	0.7858	0.1300	0.8002	0.7844	0.1267	0.8006	0.7947	0.3000	0.8021	0.7930	0.4450	0.7974	0.7883	0.2183
		rank	0.8237	0.8062	0.1233	0.8089	0.7970	0.1250	0.8099	0.8020	0.4300	0.8071	0.8002	0.2467	0.8123	0.8023	0.1383
onepoint_cross_07	bitflip_002	tournament_4	0.8460	0.8408	0.1617	0.8381	0.8324	0.1567	0.8249	0.8224	0.4067	0.8205	0.8181	0.3383	0.8352	0.8317	0.1833
		roulette	0.8059	0.7811	0.3050	0.7958	0.7814	0.1500	0.7997	0.7940	1.0267	0.7985	0.7933	0.3233	0.7978	0.7883	0.2733
		rank	0.8324	0.8263	0.1600	0.8247	0.8192	0.1650	0.8158	0.8126	0.4067	0.8135	0.8108	0.3283	0.8201	0.8154	0.1833
	bitflip_003	tournament_4	0.8380	0.8326	0.1633	0.8283	0.8226	0.1533	0.8207	0.8169	0.6067	0.8169	0.8138	0.3317	0.8240	0.8205	0.1850
		roulette	0.8053	0.7850	0.2783	0.7919	0.7803	0.1650	0.8027	0.7950	0.4267	0.8012	0.7910	0.4967	0.7985	0.7893	0.1833
		rank	0.8272	0.8173	0.1650	0.8186	0.8119	0.1833	0.8147	0.8093	0.4083	0.8119	0.8072	0.3350	0.8158	0.8100	0.2617
	bitflip_02	tournament_4	0.8159	0.7943	0.1617	0.8062	0.7911	0.1817	0.8040	0.7984	0.4150	0.8028	0.7961	0.3267	0.8050	0.7935	0.2500
		roulette	0.8032	0.7841	0.2800	0.7969	0.7834	0.1600	0.8015	0.7947	0.6400	0.7994	0.7925	0.3333	0.8008	0.7891	0.1850
		rank	0.8084	0.7910	0.1633	0.8009	0.7869	0.1667	0.8026	0.7968	0.4083	0.8014	0.7946	8.1367	0.8018	0.7910	0.2617
	inversion	tournament_4	0.8288	0.8121	0.1083	0.8234	0.8063	0.1133	0.8143	0.8082	0.2367	0.8103	0.8045	0.2067	0.8183	0.8058	0.1233
		roulette	0.8067	0.7842	0.1750	0.7952	0.7832	0.1333	0.8007	0.7939	0.2383	0.7982	0.7933	0.2067	0.7974	0.7885	0.1283
		rank	0.8190	0.8019	0.1150	0.8111	0.7974	0.1117	0.8071	0.8013	0.2417	0.8037	0.7993	0.2267	0.8060	0.7978	0.1383
	scramble	tournament_4	0.8285	0.8117	0.1150	0.7965	0.7829	0.1083	0.8162	0.8081	0.7900	0.8117	0.8057	0.2117	0.8203	0.8074	0.1233
		roulette	0.8104	0.7853	0.2383	0.8194	0.8024	0.1117	0.8043	0.7946	0.2467	0.7990	0.7931	0.2083	0.7971	0.7874	0.1400
		rank	0.8178	0.8008	0.1133	0.8085	0.7960	0.1033	0.8095	0.8026	0.2517	0.8079	0.7997	0.2067	0.8096	0.7998	0.1200
twopoint_cross_01_05	bitflip_002	tournament_4	0.8463	0.8404	0.1483	0.8368	0.8326	0.1650	0.8245	0.8219	0.3983	0.8205	0.8180	0.3267	0.8321	0.8290	0.1717
		roulette	0.8049	0.7827	0.1567	0.7954	0.7827	0.2233	0.8028	0.7961	0.3950	0.7976	0.7918	0.3200	0.7992	0.7886	0.1767
		rank	0.8365	0.8296	0.2100	0.8255	0.8190	0.1600	0.8164	0.8131	0.4483	0.8137	0.8104	0.7500	0.8205	0.8151	0.2833
	bitflip_003	tournament_4	0.8381	0.8325	0.2250	0.8307	0.8238	0.1583	0.8206	0.8177	0.5383	0.8152	0.8127	0.3117	0.8253	0.8195	0.2000
		roulette	0.8101	0.7864	0.1617	0.8005	0.7856	0.1867	0.8035	0.7955	0.3967	0.7980	0.7920	0.3117	0.7986	0.7894	0.1650
		rank	0.8279	0.8187	0.2167	0.8184	0.8123	0.1967	0.8144	0.8093	0.5417	0.8107	0.8069	0.6417	0.8199	0.8111	0.1800
	bitflip_02	tournament_4	0.8089	0.7946	0.2267	0.8037	0.7908	0.1750	0.8034	0.7981	0.4067	0.8014	0.7969	0.3250	0.8065	0.7943	0.1850
		roulette	0.8066	0.7830	0.1567	0.7961	0.7834	0.1567	0.7985	0.7937	0.5000	0.7979	0.7921	0.3200	0.7973	0.7886	0.1850
		rank	0.8083	0.7902	0.1617	0.8032	0.7876	0.1900	0.8040	0.7972	0.3950	0.7993	0.7939	0.6617	0.7978	0.7908	0.2067
	inversion	tournament_4	0.8286	0.8110	0.1183	0.8196	0.8060	0.1633	0.8140	0.8058	0.2450	0.8118	0.8049	0.2267	0.8174	0.8083	0.1167
		roulette	0.8040	0.7843	0.1983	0.7973	0.7843	0.1100	0.8021	0.7954	0.3767	0.8001	0.7923	0.2050	0.8011	0.7901	0.4767
		rank	0.8225	0.8064	0.1383	0.8085	0.7958	0.2083	0.8083	0.8033	0.2350	0.8043	0.8000	0.4167	0.8089	0.7996	0.1167
	scramble	tournament_4	0.8293	0.8160	0.1067	0.8219	0.8067	0.1217	0.8140	0.8062	0.2467	0.8110	0.8030	0.2183	0.8168	0.8059	0.1183
		roulette	0.7986	0.7824	0.1350	0.7946	0.7825	0.1133	0.8024	0.7952	0.3250	0.8032	0.7927	0.2117	0.7999	0.7898	0.1150
		rank	0.8279	0.8099	0.1917	0.8076	0.7975	0.1667	0.8073	0.8016	0.3383	0.8085	0.8002	0.4917	0.8091	0.8005	0.4050

Table 6.5: CGA test results

	Operator	Max score
Crossover	mask	0.873833776
	twopoint_01_05	0.864931952
	onepoint_07	0.863648036
	kpoint_p	0.862021741
mutate	bitflip_002	0.873833776
	bitflip_02	0.864931952
	bitflip_003	0.862620902
	inversion	0.852435162
	scramble	0.850723273
select	tournament_4	0.873833776
	rank	0.854831807
	roulette	0.824494494

Table 6.6: Operator ranking based on max achieved score

Instance	Average max score
bmc-ibm-2.cnf	0.828086393
bmc-ibm-5.cnf	0.820450813
bmc-ibm-7.cnf	0.811569954
bmc-ibm-13.cnf	0.810898957
bmc-ibm-11.cnf	0.809831564
bmc-ibm-3.cnf	0.808810871
bmc-ibm-12.cnf	0.807188575
bmc-ibm-1.cnf	0.804829026
bmc-ibm-4.cnf	0.801525953

Figure 6.1: Instance difficulty ranking based on average max score

6.4 Integrated parameter genetic algorithm experiments

In this work, a set of genetic operators were proposed, to be able to effectively test the performance of each operator, and the behavior of IPGA in general, and taking into consideration the hardware and temporal limitations, a limited test set of operators was selected. similarly to the CGA tests, due to the interlinked relation between genetic operators, a set of genetic algorithms was created to test all the possible combinations of the selected operators for testing. These GA's are tested firstly against the bmc-ibm benchmark, and then a comparison of the general performance of the operators against their CGA counterparts. Then, the effect of the proposed methods on the diversity and the overall behavior of the algorithm is explored.

Because of the large number of experiments, all mutation operators use coupled mutation only.

The following operators are tested(Table 6.7 lists the IPGA tested operators):

The number of generations and the population size are fixed to the same values tested in CGA.

	Operator	Name
Crossover	DRCRX	dominant random confusion resolution crossover
	RCRX	random confution resolution crossover
	RBRX	resiliance based resolution crossover
	SRX	simple resolution crossover
Mutation	CM	Cauchy mutation
	NM	normal mutation
	NUM	non uniform mutation
	RM	random mutation
Selection	tournament select 4	
	rank select	
	roulette select	

Table 6.7: IPGA operators

The following are the test results for each test file: the max score and mean score of the final generation are recorded, and the top 10% of the solutions are highlighted in green. Figures 6.8 and 6.9 give the score and mean score reached by each combination of IPGA operators per test file and their execution time per generation:

Through these experimental results, we can state that the best operator combination is dominant random confusion resolution with Cauchy mutation and tournament selection. Figure 6.10 gives ranking of each proposed operator based on the maximum score reached using a GA including that operator:

As shown in the IPGA table, using simple crossover resulted in a lower score than its confusion-resolving counterparts; both RCRX and DRCRX performed 8% better than SRX, which shows the importance of confusion resolution.
Further, SRX methods stagnate at a low score close to the score of the initial generation, which indicates that the GA is not converging.

Figure 6.2 exposes the reached diversity using the proposed crossover methods in function of the generation averaged over 3 runs per crossover method for 200 generations on the instance bmc-ibm-2.cnf.

Examining the solution diversity chart6.2, we observe that the SRX methods' generation diversity stays static, maintaining a high diversity. This proves that the simple crossover method is not effective and does not lead to a convergence.

On the other hand, from the diversity chart 6.2, we can see that RCRX maintains a higher diversity than DRCRX throughout the GA's lifetime. This was expected, as

cross	mutate	select	bmc-ibm-1.cnf			bmc-ibm-2.cnf			bmc-ibm-3.cnf			bmc-ibm-4.cnf		
			score	mean	time(s)	score	mean	time(s)	score	mean	time(s)	score	mean	time(s)
DRCRX	CM	tournament_select_4	0.8754	0.8713	0.7367	0.9002	0.8947	0.2317	0.8594	0.8525	1.2883	0.8401	0.8378	1.7400
		roulette_select	0.8263	0.8188	0.7283	0.8539	0.8438	0.2867	0.8238	0.8142	0.9650	0.8126	0.8086	1.8317
		rank_select	0.8600	0.8556	0.7700	0.8892	0.8821	0.2767	0.8477	0.8397	1.2483	0.8358	0.8318	1.9500
	NUM	tournament_select_4	0.8337	0.8251	0.7733	0.8552	0.8472	0.2533	0.8259	0.8162	1.3333	0.8178	0.8131	2.5917
		roulette_select	0.8026	0.7945	0.7733	0.8361	0.8122	0.2867	0.8078	0.7947	0.9933	0.8030	0.7967	1.9817
		rank_select	0.8298	0.8222	0.7450	0.8517	0.8401	0.2450	0.8232	0.8117	1.2483	0.8142	0.8096	1.9917
	NM	tournament_select_4	0.8756	0.8694	0.7317	0.8917	0.8847	0.2983	0.8552	0.8479	1.0333	0.8369	0.8344	1.8467
		roulette_select	0.8296	0.8217	0.7750	0.8440	0.8338	0.2633	0.8223	0.8121	1.1033	0.8135	0.8087	1.9600
		rank_select	0.8600	0.8539	0.7600	0.8847	0.8777	0.2933	0.8488	0.8410	1.0217	0.8304	0.8267	1.9167
	RM	tournament_select_4	0.8245	0.8183	0.7267	0.8526	0.8428	0.2750	0.8223	0.8119	0.9767	0.8140	0.8080	1.8583
		roulette_select	0.8027	0.7944	0.7900	0.8242	0.8079	0.2733	0.8074	0.7942	0.9600	0.7992	0.7933	1.8583
		rank_select	0.8188	0.8116	0.6117	0.8506	0.8388	0.2683	0.8234	0.8070	1.2300	0.8093	0.8043	1.8750
RCRX	CM	tournament_select_4	0.8717	0.8667	0.0833	0.8960	0.8900	0.2633	0.8575	0.8508	0.8783	0.8381	0.8353	1.8800
		roulette_select	0.7937	0.7805	0.7683	0.8072	0.7913	0.2100	0.7958	0.7814	0.9700	0.7894	0.7822	1.9033
		rank_select	0.8534	0.8471	0.0783	0.8766	0.8692	0.2567	0.8421	0.8330	0.8350	0.8257	0.8214	1.7200
	NUM	tournament_select_4	0.8342	0.8264	0.6633	0.8565	0.8447	0.2833	0.8311	0.8202	0.9133	0.8181	0.8129	1.9700
		roulette_select	0.7883	0.7770	0.7400	0.8203	0.7948	0.2867	0.8021	0.7839	0.9933	0.7918	0.7841	1.9650
		rank_select	0.8204	0.8140	0.7750	0.8446	0.8321	0.2717	0.8199	0.8067	0.8983	0.8105	0.8047	2.0167
	NM	tournament_select_4	0.8697	0.8648	0.0917	0.8951	0.8897	0.2833	0.8563	0.8468	1.0450	0.8394	0.8371	1.9417
		roulette_select	0.7895	0.7785	0.0933	0.8098	0.7964	0.2950	0.7961	0.7830	0.8817	0.7842	0.7788	1.8650
		rank_select	0.8470	0.8409	0.1667	0.8733	0.8658	0.2483	0.8440	0.8316	1.0467	0.8259	0.8221	1.8217
	RM	tournament_select_4	0.8280	0.8197	0.0817	0.8558	0.8439	0.2067	0.8255	0.8116	0.9633	0.8125	0.8081	1.9817
		roulette_select	0.7920	0.7801	0.0398	0.8113	0.7852	0.2583	0.7939	0.7775	1.0200	0.7922	0.7840	1.9900
		rank_select	0.8166	0.8062	0.7272	0.8391	0.8262	0.2350	0.8142	0.8020	0.8233	0.8059	0.8006	1.8383
RBRX	CM	tournament_select_4	0.7950	0.7812	0.6533	0.8169	0.7962	0.2917	0.8589	0.8526	0.9683	0.7921	0.7866	2.1783
		roulette_select	0.7868	0.7762	0.7167	0.7989	0.7797	0.2733	0.8002	0.7853	0.9583	0.7905	0.7835	2.1383
		rank_select	0.7876	0.7779	0.6017	0.8111	0.7875	0.2783	0.8448	0.8349	0.9633	0.7932	0.7855	2.1083
	NUM	tournament_select_4	0.7890	0.7782	0.6167	0.7963	0.7766	0.2833	0.8332	0.8214	0.9850	0.7896	0.7833	2.2483
		roulette_select	0.7898	0.7770	0.7350	0.8083	0.7852	0.2850	0.7973	0.7793	1.2983	0.7927	0.7836	2.2017
		rank_select	0.7847	0.7766	0.6483	0.8040	0.7809	0.2833	0.8229	0.8113	0.9850	0.7924	0.7846	2.3067
	NM	tournament_select_4	0.7918	0.7808	0.7600	0.8109	0.7906	0.2833	0.8583	0.8500	1.0267	0.7922	0.7853	2.2133
		roulette_select	0.7866	0.7768	0.6500	0.8114	0.7872	0.2917	0.7985	0.7831	1.0283	0.7910	0.7844	2.3533
		rank_select	0.7905	0.7784	0.7683	0.8081	0.7887	0.2850	0.8414	0.8317	1.0967	0.7923	0.7850	2.3500
	RM	tournament_select_4	0.7932	0.7789	0.7217	0.8086	0.7872	0.2700	0.8290	0.8161	0.9817	0.7958	0.7855	1.9100
		roulette_select	0.7869	0.7778	0.6133	0.8125	0.7836	0.2750	0.7984	0.7807	3.9500	0.7913	0.7841	2.0667
		rank_select	0.7880	0.7774	0.7150	0.8005	0.7836	0.2767	0.8202	0.8069	0.9783	0.7930	0.7854	1.9033
SRX	CM	tournament_select_4	0.7900	0.7787	0.8300	0.8107	0.7889	0.2350	0.8012	0.7845	1.3350	0.7938	0.7871	1.9683
		roulette_select	0.7896	0.7770	0.8850	0.8058	0.7828	0.2967	0.8035	0.7800	1.0467	0.7918	0.7842	1.9950
		rank_select	0.7926	0.7795	0.8400	0.8044	0.7843	0.2967	0.7984	0.7802	1.3433	0.7926	0.7847	2.0183
	NUM	tournament_select_4	0.7902	0.7767	0.2683	0.8063	0.7874	0.2917	0.8000	0.7798	1.1100	0.7939	0.7848	2.1483
		roulette_select	0.7866	0.7770	0.0800	0.8067	0.7821	0.2400	0.8000	0.7785	0.3967	0.7923	0.7843	1.8733
		rank_select	0.7897	0.7776	0.8383	0.8078	0.7846	0.2267	0.7984	0.7798	1.2217	0.7917	0.7847	1.8167
	NM	tournament_select_4	0.7930	0.7812	0.3650	0.8155	0.7905	0.3883	0.8027	0.7821	1.7733	0.7953	0.7858	2.1500
		roulette_select	0.7933	0.7781	0.0167	0.8064	0.7848	0.2500	0.7972	0.7802	1.7833	0.7946	0.7855	3.1783
		rank_select	0.7916	0.7788	0.0133	0.8027	0.7831	0.3783	0.8011	0.7809	0.0417	0.7937	0.7852	3.2017
	RM	tournament_select_4	0.7948	0.7784	0.6650	0.8161	0.7884	0.2350	0.8000	0.7810	0.9733	0.7934	0.7855	1.9433
		roulette_select	0.7879	0.7771	0.6883	0.7984	0.7813	0.2483	0.7975	0.7813	1.0817	0.7944	0.7847	2.1533
		rank_select	0.7868	0.7778	0.6400	0.8062	0.7850	0.2717	0.8020	0.7816	1.0733	0.7928	0.7844	0.1267

Table 6.8: IPGA test results

dominance crossover favors the expression of the higher-performing parent instead of favoring a different parent for the two resulting offspring. Even though a higher diversity is considered a positive characteristic as it maintains variety in the gene pool, What helps avoid premature convergence.

Despite this, DRCRX reached better score values6.4. It could be suggested that despite DRCRX having lower diversity, it still maintains its useful diversity, where useful diversity is defined as diversity that leads to good solutions.

Figure 6.11 illustrates the difference in score and mean score between CGA and IPGA.

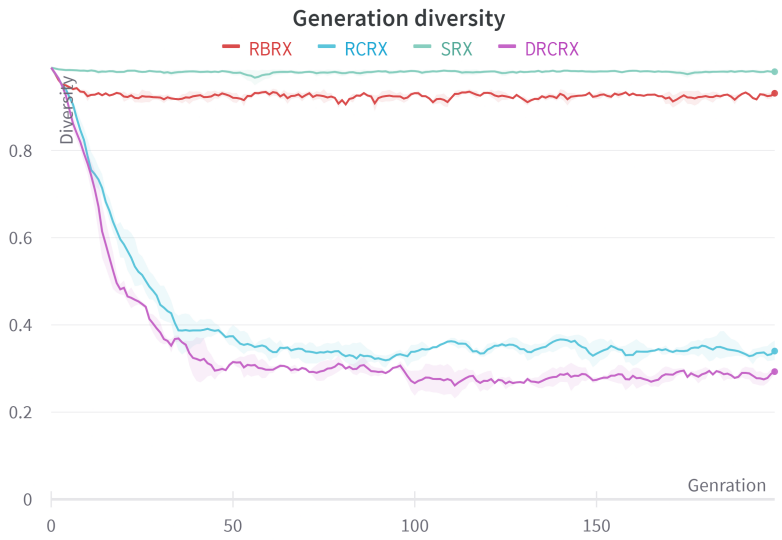


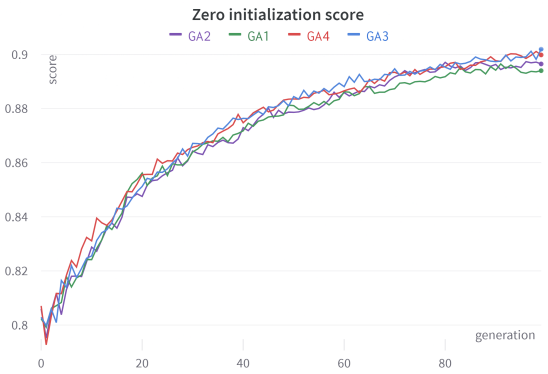
Figure 6.2: Generation diversity

This difference is calculated as the difference between the best score and mean score value for each test instance.

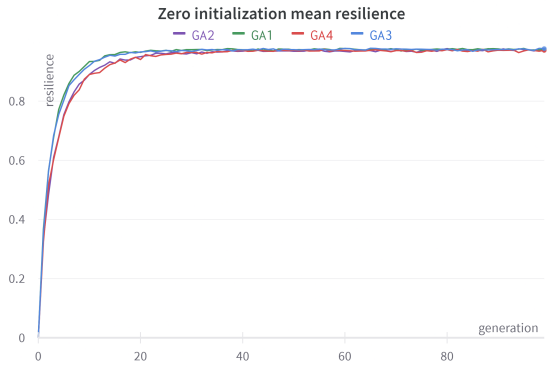
Even though resilience minimization was effective in keeping the resilience value lower than the other methods, but this lead to a limited convergence compared to DRCRX and RCRX. This method was proposed to limit the accumulation of resilience, this happens when a gene gets mutated, if it receives a higher resilience value it becomes less likely to be mutated in the later generations. Having an upper limit helped mitigate this issue, but from the empirical results we can see that it has a large effect on the GAs performance. On the other hand comparing the performance of IPGA to CGA shows that IPGA performed 4%better on both the average score of the last generation and the max score(see figure 6.11). This disparity maybe due to a suboptimal parameter initialization, as CGA depends highly on the mutation probability and the crossover method used, but the mean resilience of the best performing IPGA method converged to 0.98, which is equivalent to a mutation probability of 0.02 used in the best performing CGA, what suggests that this disparity in score is actually not a parameter issue.

One of IPGA's characteristics is the ability to converge to optimal parameters without a definite initialization; IPGA initializes the resilience and expression vectors to random values in the initial generation, this can be showcased by initializing these vectors to bad values, in this case to zero.

As chart 6.3a shows, even though the resilience and expression vectors were initialized to zero in the first generation, the genetic algorithms listed in 6.12 converged to score values comparable to the randomly initialized tests seen in chart 6.4. The zero initialization mean resilience chart 6.3b shows all the methods starting with 0 mean resilience as all the population is initialized to zero, and all methods showed a trend towards a stagnation



(a) IPGA zero init score



(b) IPGA zero init mean resiliance

Figure 6.3: IPGA zero initialization results

around 0.98 mean resilience. Furthermore, we notice that GAs that use Cauchy mutation reached stagnation faster than GAs using normal mutation.

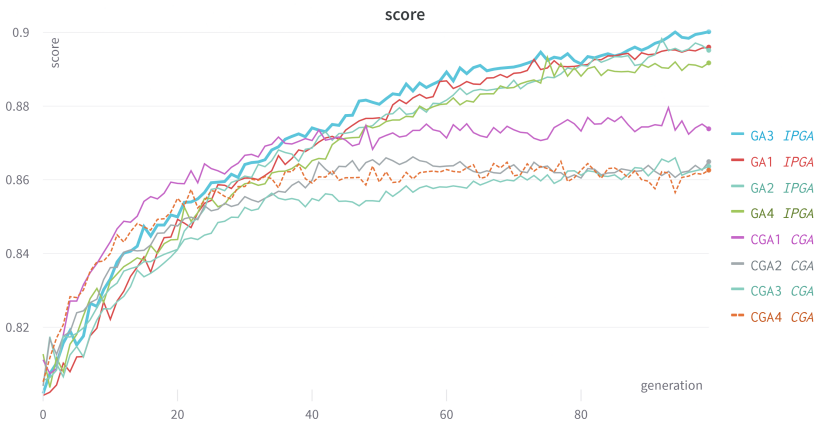


Figure 6.4: IPGA and CGA score trends

Finally, the added complexity of IPGA lead to a 0.57 seconds slower average generation execution time compared to CGA, this is caused by the added computation of both the mutation of expression and resilience vectors, and the crossover resolution process.

cross	mutate	select	bmc-ibm-5.cnf			bmc-ibm-7.cnf			bmc-ibm-13.cnf		
			score	mean	time(s)	score	mean	time(s)	score	mean	time(s)
DRCRX	CM	tournament_select_4	0.8744	0.8715	0.8500	0.8685	0.8636	0.8300	0.8676	0.8633	1.1167
		roulette_select	0.8397	0.8314	0.9850	0.8323	0.8249	0.6317	0.8263	0.8211	1.2933
		rank_select	0.8688	0.8632	0.6950	0.8602	0.8548	0.9250	0.8592	0.8529	1.1583
	NUM	tournament_select_4	0.8462	0.8357	1.0217	0.8378	0.8307	0.7683	0.8317	0.8264	1.1067
		roulette_select	0.8178	0.8071	0.6900	0.8117	0.8003	0.9783	0.8120	0.8053	1.0683
		rank_select	0.8351	0.8263	0.9000	0.8313	0.8224	0.7267	0.8288	0.8227	1.2083
	NM	tournament_select_4	0.8757	0.8707	0.6817	0.8670	0.8635	0.8950	0.8624	0.8593	1.1417
		roulette_select	0.8366	0.8277	1.0500	0.8329	0.8248	0.9817	0.8325	0.8232	1.2533
		rank_select	0.8637	0.8595	0.6650	0.8604	0.8539	0.5950	0.8541	0.8505	1.2300
	RM	tournament_select_4	0.8342	0.8240	0.8600	0.8349	0.8260	0.8283	0.8294	0.8222	1.1117
		roulette_select	0.8215	0.8045	0.7083	0.8114	0.8000	0.7733	0.8104	0.8019	1.1200
		rank_select	0.8331	0.8235	0.9783	0.8248	0.8172	0.9233	0.8193	0.8142	1.0883
RCRX	CM	tournament_select_4	0.8707	0.8672	0.6817	0.8691	0.8652	0.8600	0.8651	0.8609	1.0467
		roulette_select	0.8104	0.7930	0.8250	0.8084	0.7919	0.6800	0.7971	0.7907	1.0267
		rank_select	0.8536	0.8491	0.9050	0.8561	0.8498	0.9333	0.8483	0.8427	1.1617
	NUM	tournament_select_4	0.8435	0.8365	0.9733	0.8381	0.8296	0.6850	0.8324	0.8272	1.1100
		roulette_select	0.7964	0.7822	0.8767	0.7950	0.7814	0.9533	0.7998	0.7911	1.1817
		rank_select	0.8361	0.8237	0.8117	0.8246	0.8189	0.8650	0.8241	0.8180	0.8583
	NM	tournament_select_4	0.8663	0.8607	0.6600	0.8686	0.8642	0.6700	0.8619	0.8584	1.1017
		roulette_select	0.8022	0.7910	0.9100	0.7916	0.7820	0.9917	0.7995	0.7912	0.9133
		rank_select	0.8563	0.8514	0.9867	0.8505	0.8449	0.7000	0.8486	0.8428	1.2333
	RM	tournament_select_4	0.8381	0.8272	0.6717	0.8326	0.8257	0.8517	0.8294	0.8242	1.0183
		roulette_select	0.7991	0.7806	0.8300	0.7916	0.7805	0.7183	0.7982	0.7890	0.9483
		rank_select	0.8302	0.8198	0.6933	0.8245	0.8168	0.8583	0.8231	0.8146	1.1433
RBRX	CM	tournament_select_4	0.8108	0.7913	1.1883	0.8031	0.7884	1.1333	0.8067	0.7916	1.3667
		roulette_select	0.8004	0.7834	1.1167	0.7964	0.7829	0.7367	0.8015	0.7891	1.4567
		rank_select	0.8064	0.7843	0.9600	0.7999	0.7862	1.0717	0.7988	0.7906	1.2933
	NUM	tournament_select_4	0.8109	0.7845	1.0167	0.7963	0.7835	0.7283	0.7980	0.7908	1.3833
		roulette_select	0.8089	0.7854	0.7483	0.7975	0.7837	0.9800	0.7994	0.7884	1.1400
		rank_select	0.7991	0.7837	0.8083	0.8009	0.7851	0.7283	0.7976	0.7888	1.0850
	NM	tournament_select_4	0.8099	0.7918	1.6067	0.8030	0.7860	1.4583	0.7993	0.7894	1.8967
		roulette_select	0.8058	0.7854	1.5167	0.7921	0.7818	1.4100	0.7973	0.7880	2.0483
		rank_select	0.8036	0.7851	0.9017	0.7991	0.7863	0.8050	0.7986	0.7895	1.2550
	RM	tournament_select_4	0.8104	0.7866	0.9700	0.7982	0.7837	0.6583	0.7970	0.7887	1.1300
		roulette_select	0.8126	0.7832	0.8950	0.7999	0.7829	0.9583	0.7974	0.7883	1.1933
		rank_select	0.8045	0.7852	0.7950	0.7971	0.7828	0.8533	0.7990	0.7881	0.9783
SRX	CM	tournament_select_4	0.8185	0.8066	0.8033	0.8002	0.7860	1.1083	0.8024	0.7913	1.3650
		roulette_select	0.8069	0.7844	1.0850	0.7971	0.7830	1.0017	0.7974	0.7883	1.2883
		rank_select	0.8070	0.7880	0.9850	0.7934	0.7840	0.7150	0.8000	0.7900	1.0233
	NUM	tournament_select_4	0.8018	0.7865	1.0067	0.7978	0.7839	1.0317	0.7975	0.7893	1.1317
		roulette_select	0.8026	0.7834	0.9183	0.8023	0.7840	0.8267	0.7969	0.7883	1.0650
		rank_select	0.8150	0.7845	0.8117	0.7990	0.7838	1.0283	0.7972	0.7889	1.0950
	NM	tournament_select_4	0.8119	0.7975	1.3550	0.7990	0.7858	0.7317	0.7987	0.7902	1.7667
		roulette_select	0.8020	0.7850	0.7983	0.7980	0.7821	1.2867	0.7995	0.7883	1.6717
		rank_select	0.8059	0.7845	1.2867	0.7949	0.7839	0.7433	0.7998	0.7903	1.1133
	RM	tournament_select_4	0.8036	0.7870	0.6967	0.8006	0.7830	0.8350	0.7999	0.7898	1.1133
		roulette_select	0.8027	0.7850	0.8867	0.7955	0.7828	0.9167	0.8048	0.7882	1.1167
		rank_select	0.8041	0.7845	0.7567	0.7998	0.7842	0.6767	0.7975	0.7903	1.0650

Table 6.9: IPGA test results

	Operator	Max score
crossover	DRCRX	0.9002
	RCRX	0.8960
	RBRX	0.8589
	SRX	0.8185
mutation	CM	0.9002
	NM	0.8951
	NUM	0.8565
	RM	0.8558
selection	tournament_select_4	0.9002
	rank_select	0.8892
	roulette_select	0.8539

Table 6.10: IPGA operator ranking

File	bmc-ibm-1.cnf		bmc-ibm-2.cnf		bmc-ibm-3.cnf		bmc-ibm-4.cnf	
Difference between CGA and	score	mean	score	mean	score	mean	score	mean
IPGA(IPGA-CGA)	0.0350	0.0443	0.0412	0.0455	0.0598	0.0595	0.0224	0.0241
File	bmc-ibm-5.cnf		bmc-ibm-7.cnf		bmc-ibm-13.cnf		Average	average
Difference between CGA and	score	mean	score	mean	score	mean	score	mean
IPGA(IPGA-CGA)	0.0405	0.0490	0.0330	0.0370	0.0352	0.0359	0.0382	0.0422

Table 6.11: Difference in IPGA and CGA performance

GA Name	crossover	mutation	selection
GA1	RCRX	CM	tournament select 4
GA2	RCRX	NM	tournament select 4
GA3	DRCRX	CM	tournament select 4
GA4	DRCRX	NM	tournament select 4

Table 6.12: Zero init test GA names

6.5 Conclusion

Throughout this chapter, experiments were processed on a set of well established benchmarks. Several combination of genetic parameters were tested in order to define a baseline of performance for later comparison. Once done, the proposed Integrated Parameter Genetic Algorithm was tested and compared with the classical genetic algorithm. The obtained results showed a measurable increase in performance proving the effectiveness of this method.

Chapter 7

General Conclusion

In this thesis, we have proposed an improved genetic algorithm by integrating the algorithm's parameters into the problem search space.

This work is the consequence of a broad analysis of the optimization process. Considering parameter tuning as a preliminary step of the genetic algorithm was the motivation behind parameter integration.

This approach eliminates the need for tuning the crossover and mutation parameters adding two vectors named expression and resilience respectively. The first dictates the participation of a parent in the crossover process and the second represents an agents resistance to mutation.
This idea led to different implementations of genetic operators, all combinations of these operators were tested against a well-established benchmark to test their performance. The experiments conducted showed the impact of the proposed solution. Indeed, in addition to the considerable increase in performance, the tests showed that this method is successful in eliminating the need for parameter tuning, which saves a considerable amount of time, in addition to GA becoming a problem-independent algorithm.

In the future, an analysis of resilience and expression trends will be useful to further understand the behavior of this algorithm.

Finally, as the concept of parameter integration is not limited to genetic algorithms, implementing it on other metaheuristics can be considered the next step in our research.

Bibliography

[1] Ofer Shtrichman. "Tuning SAT checkers for bounded model checking". In: *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer. 2000, pp. 480–494.

[2] Michel Gendreau, Jean-Yves Potvin, et al. *Handbook of metaheuristics*. Vol. 2. Springer, 2010.

[3] Zahra Beheshti and Siti Mariyam Hj Shamsuddin. "A review of population-based meta-heuristic algorithms". In: *Int. J. Adv. Soft Comput. Appl* 5.1 (2013), pp. 1–35.

[4] Masaaki Suzuki. "Adaptive Parallel Particle Swarm Optimization Algorithm Based on Dynamic Exchange of Control Parameters". In: *American Journal of Operations Research* 6.5 (2016).

[5] Siti Sophiyati Yuhaniz Dian Palupi Rini Siti Mariyam Shamsuddin. "Particle Swarm Optimization: Technique, System and Challenges". In: *International Journal of Computer Applications* 14.1 (2011).

[6] Mohammed Moness. "H. Youness, A. Ibraheim, M. Moness, and M. Osama, "An Efficient Implementation of Ant Colony Optimization on GPU for the Satisfiability Problem," 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015". In: Mar. 2015.

[7] Marco Dorigo and Thomas Stützle. "Ant Colony Optimization: Overview and Recent Advances". In: *Handbook of Metaheuristics* (2018).

[8] Thomas Stu¨tzle Marco Dorigo Mauro Birattari. "Ant Colony Optimization". In: *IEEE COMPUTATIONAL INTELLIGENCE MAGAZINE* 1.4 (2006).

[9] Esmat Rashedi, Hossein Nezamabadi-pour, and Saeid Saryazdi. "GSA: A Gravitational Search Algorithm". In: *Information Sciences* 179.13 (2009). Special Section on High Order Fuzzy Sets, pp. 2232–2248. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2009.03.004. URL: https://www.sciencedirect.com/science/article/pii/S0020025509001200.

[10] Thomas Bäck and Hans-Paul Schwefel. "An overview of evolutionary algorithms for parameter optimization". In: *Evolutionary computation* 1.1 (1993), pp. 1–23.

[11] Xin Yao, Yong Liu, and Guangming Lin. "Evolutionary programming made faster". In: *IEEE Transactions on Evolutionary Computation* 3.2 (1999), pp. 82–102. DOI: 10.1109/4235.771163.

[12] Seyedali Mirjalili. "Evolutionary algorithms and neural networks". In: *Studies in computational intelligence*. Vol. 780. Springer, 2019.

[13] Khalid Jebari. "Selection Methods for Genetic Algorithms". In: *International Journal of Emerging Sciences* 3 (Dec. 2013), pp. 333–344.

[14] Peter JB Hancock. "An empirical comparison of selection methods in evolutionary algorithms". In: *Evolutionary Computing: AISB Workshop Leeds, UK, April 11–13, 1994 Selected Papers*. Springer. 2005, pp. 80–94.

[15] Tania Pencheva, Krassimir Atanassov, and Anthony Shannon. "Modelling of a stochastic universal sampling selection operator in genetic algorithms using generalized nets". In: *Proceedings of the tenth international workshop on generalized nets, Sofia*. 2009, pp. 1–7.

[16] Artem Sokolov and Darrell Whitley. "Unbiased tournament selection". In: *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. 2005, pp. 1131–1138.

[17] Anant J Umbarkar and Pranali D Sheth. "Crossover operators in genetic algorithms: a review." In: *ICTACT journal on soft computing* 6.1 (2015).

[18] Siew Mooi Lim et al. "Crossover and mutation operators of genetic algorithms". In: *International journal of machine learning and computing* 7.1 (2017), pp. 9–12.

[19] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. "A review on genetic algorithm: past, present, and future". In: *Multimedia Tools and Applications* 80 (2021), pp. 8091–8126.

[20] Juha Kivijärvi, Pasi Fränti, and Olli Nevalainen. "Self-Adaptive Genetic Algorithm for Clustering". In: *Journal of Heuristics* 9.2 (Mar. 2003), 113–129. ISSN: 1381-1231. DOI: 10.1023/A:1022521428870. URL: https://doi.org/10.1023/A:1022521428870.

[21] F. Herrera, M. Lozano, and J. L. Verdegay. "Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis". In: *Artif. Intell. Rev.* 12.4 (Aug. 1998), 265–319. ISSN: 0269-2821. DOI: 10.1023/A:1006504901164. URL: https://doi.org/10.1023/A:1006504901164.

[22] Aki Sorsa, Riikka Peltokangas, and Kauko Leiviska. "Real-coded genetic algorithms and nonlinear parameter identification". In: *2008 4th International IEEE Conference Intelligent Systems*. Vol. 2. IEEE. 2008, pp. 10–42.

[23] John J Grefenstette. "Optimization of control parameters for genetic algorithms". In: *IEEE Transactions on systems, man, and cybernetics* 16.1 (1986), pp. 122–128.

[24] Selmar K Smit and Agoston E Eiben. "Comparing parameter tuning methods for evolutionary algorithms". In: *2009 IEEE congress on evolutionary computation*. IEEE. 2009, pp. 399–406.

[25] André Abramé and Djamal Habet. "Ahmaxsat: Description and Evaluation of a Branch and Bound Max-SAT Solver". In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (Dec. 2015). DOI: 10.3233/SAT190104.

[26] Cezar-Constantin Andrici and Ştefan Ciobâcă. "Who Verifies the Verifiers? A Computer-Checked Implementation of the DPLL Algorithm in Dafny". In: *arXiv preprint arXiv:2007.10842* (2020).

[27] Joao Marques-Silva, Inês Lynce, and Sharad Malik. "Conflict-driven clause learning SAT solvers". In: *Handbook of satisfiability*. IOS press, 2021, pp. 133–182.

[28] Kuo-Torng Lan and Chun-Hsiung Lan. "Notes on the distinction of Gaussian and Cauchy mutations". In: *2008 Eighth International Conference on Intelligent Systems Design and Applications*. Vol. 1. IEEE. 2008, pp. 272–277.

[29] Andy Pryke, Sanaz Mostaghim, and Alireza Nazemi. "Heatmap Visualization of Population Based Multi Objective Algorithms". In: Jan. 2006, pp. 361–375. ISBN: 978-3-540-70927-5. DOI: 10.1007/978-3-540-70928-2_29.

[30] Noureddine Bouhmala, Karina Hjelmervik, and Kjell Ivar Øvergård. "Combining An Evolutionary Algorithm With The Multilevel Paradigm For The Simulation Of Complex System." In: *ECMS*. 2013, pp. 753–757.

[31] Marc Thurley. "sharpSAT-counting models with advanced component caching and implicit BCP". In: *SAT* 4121 (2006), pp. 424–429.