University of Sheffield

Rigorous Runtime Analysis of Evolutionary Algorithms for the Nonogram NP-hard Problem

Aleksandra Kulbaka

Supervisor: Pietro Oliveto

A report submitted in fulfilment of the requirements
for the degree of BSc in Artificial Intelligence and Computer Science

in the

Department of Computer Science

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Aleksandra Kulbaka

Signature: Aleksandra Kulbaka

Abstract

The report evaluates the performance of Evolutionary Algorithms on the Nonogram NP-hard problem. It proposes two new fitness functions that perform better than the one described in the literature. The upper bounds on the expected runtime of the (1+1)Evolutionary Algorithm and Randomised Local Search proved in the report are equivalent or smaller than the upper bounds on the runtime of problem-specific algorithms for most of the analysed subclasses. That allows us to predict that even the simplest bio-inspired approaches can successfully compete with the problem-specific methods when applied to the Nonogram problem.

COVID-19 Impact Statement

The lockdown imposed because of COVID-19 caused additional challenges for the completion of this project. At the time that the project was being developed, all teaching was conducted online, and university buildings were closed. All project meetings were shifted to email correspondence and video meetings.

Acknowledgements

I would like to thank my project supervisor, Dr Pietro Oliveto, for his constant support, motivation and trust as well as the knowledge and the enthralment about the field he has passed to me during the time we worked on this project. This helped me tremendously in staying focused and excited about the research and finishing it on time.
I would also like to thank my parents for motivation, listening to me worrying about not finishing this project on time during every conversation we had this year, and not letting me give up.
Finally I would like to thank my friends, Amber, Andreas, Chris, Charlotte, Hayley, Iga, Jack, Karola, Marialena and Nikki (alphabetical order!) for encouraging me to apply for this project and constant support during this year.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Evolutionary Algorithms (EAs) find high-quality solutions to optimisation problems by applying the principles of evolution found in nature [26]. They do not require any problem-specific knowledge; therefore, they can be applied to many (even not well understood) problems, and they are highly successful in practice [19]. EAs perform particularly well on Combinatorial Optimization Problems [17]. One of such problems is solving Nonograms - picture logic puzzles [5]. The goal is to fill a grid with black and white cells based on the rows' and columns' descriptions. The problem is NP-Hard; thus, one cannot expect that any algorithm solves all the instances of the problem in polynomial time. Empirical results have suggested that evolutionary algorithms can be competitive in practice with problem-specific algorithms [14, 22].

The main challenge with EAs is to design an appropriate fitness function to mathematically evaluate and compare different solutions. Since computing a fitness value for a given instance is the most time-consuming step of the Evolutionary Algorithm, the expected running time is usually measured as a number of these evaluations [17]. Hence, choosing the most efficient fitness function is crucial to obtain the good performance. However, creating the correct fitness function is very difficult, especially for the complex problems.

Another challenging step is to predict how much time an EA will need to output the solution. Research on correctness and time complexity of such algorithms is relatively new. The first theoretical result of runtime analysis of an Evolutionary Algorithm (the (1+1)EA) was given by Mühlenbein in 1992 [17]. That is mainly due to many random decisions and the complexity of fitness functions used by those algorithms, which make the analysis very difficult to conduct.

1.1 Aims and Objectives

The main aim of this project is to examine whether Evolutionary Algorithms can compete with problem-specific methods when applied to the Nonogram problem. First, an appropriate fitness function will be created. Next, the performance, in terms of the expected running

time, of the considered Evolutionary Algorithms on the Nonogram problem will be analysed. The runtime analysis will be conducted on subclasses of the problem that can be solved in polynomial time. The results will be then compared with the problem-specific methods described in Chapter 2.

1.2 Overview of the Report

The structure of this report is as follows. Chapter 2 defines the theory behind Nonograms and bio-inspired algorithms. It also aims to describe the complexity of the Nonogram problem and show why and when a bio-inspired optimisation can be successfully applied to this problem. Chapter 3 lists the main aims of this report, defines the algorithms used in the analysis, and provides mathematical methods to bound the expected running time of the Evolutionary Algorithms. Chapter 4 describes the encoding of the Nonograms and defines the problem's subclasses solvable in polynomial time. It also defines and compares three fitness functions proposed in this report with the standard function taken from the literature. Chapter 5 proves the upper bounds on the expected running time of the (1+1)Evolutionary Algorithm and Randomized Local Search with proposed fitness functions on all the described Nonongrams' subclasses. Finally, Chapter 6 summarizes the results and provides further research questions.

Chapter 2

Literature Survey

The literature survey begins by defining all the fundamental concepts and theories that are necessary to understand the report. Section 2.3 describes how bio-inspired algorithms work. Next, the current research on the Nonogram problem is overviewed. Finally, the literature regarding applications of both traditional approaches and bio-inspired algorithms to the Nonogram problem is summarized and compared.

2.1 Algorithms

An algorithm can be defined as a sequence of computational steps which transform the input into the output. It is used to solve well-specified computational problems (Section 2.2). A correct algorithm halts with the correct output for every input instance [7].

The algorithm is **deterministic** if for a given input it always produces the same output by going through the same set of states. Algorithm is described as **randomized** if its behavior is determined not only by its input, but also by the values produced by a random-number generator [13].

Usually there exist many algorithms for a given problem. They may differ in: the amount of time (time complexity) or memory space (space complexity) needed for solving a problem, the quality of the result (correct or approximate), or the probability of success [7]. Hence, when choosing the right algorithm, the performance analysis of one or more of these features is crucial. In this report, the main focus is on the Time Complexity Analysis.

2.1.1 Time Complexity Analysis

Time Complexity Analysis allows us to predict how much computational time an algorithm requires. In this report, runtime of algorithm X on instance I means the number of elementary operations in the RAM model X takes on I for problem specific algorithms [24], while for Evolutionary Algorithms it means the number of fitness function evaluations [19].

For the deterministic algorithms, we will concentrate on the **worst-case running time** - the longest running time of an algorithm X for any input of size n. For the randomised algorithms, we will consider the **expected running time** - an expectation of the running time over the distribution of values returned by the random number generator [7].

We will express the running time using the asymptotic notation. This enables us to capture how the runtime changes with regard to the input size n, and to compare the relative performance of different algorithms. Main types of asymptotic notation are [7, 24]:

- ***O*-notation** - to denote asymptotic upper bound.
  $O(g(n)) = \{\ f(n)$: there exist constants $0 < c$ and n_0 such that
  $0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0\}$
- **Ω-notation** - to denote asymptotic lower bound.
  $\Omega(g(n)) = \{\ f(n)$: there exist constants $0 < c$ and n_0 such that
  $0 \leq c * g(n) \leq f(n)$ for all $n \geq n_0\}$
- **Θ-notation** - to denote asymptotic tight upper and lower bounds.
  $\Theta(g(n)) = \{\ f(n)$: there exist constants $0 \leq c_1 \leq c_2$ and n_0 such that
  $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0\}$

2.1.2 Approximation Algorithms

Approximation Algorithms are useful when the solution to the combinatorial optimization problem needs to be found quickly, but it does not have to be exact (near-optimal suffices). They trade accuracy for time by returning a solution which is provably close to the optimum. The most important property of approximation algorithms is the worst-case approximation ratio ($p(n)$) which describes how close to the optimum the solutions returned by an algorithm are in the worst case [2].

$p(n)$-approximation algorithm for a minimization problem

For $p(n) \geq 1$, the $p(n)$-approximation algorithm, for any instance I, must:

- run in polynomial time,
- return the solution $A(I)$ such that $A(I) \leq p(n) \times OPT(I)$ [21].

$p(n)$-approximation algorithm for a maximization problem

For $p(n) \leq 1$, the $p(n)$-approximation algorithm, for any instance I, must:

- run in polynomial time,
- return the solution $A(I)$ such that $A(I) \geq p(n) \times OPT(I)$ [21].

2.2 Computational Problems

Informally, computational problems are the questions or problems that a computer may be able to answer. Given an input, a computational problem is defined by the property that the output has to satisfy, e.g., "given number a and b, output their highest common factor" [27]. The main types of computational problems are [6, 27]:

- Decision Problems - finding the "Yes/No" answer to the problem,
- Search Problems - finding the solution (if it exists),
- Counting Problems - finding the number of possible solutions to the problem,
- Optimization Problems - finding the best possible solution to the problem.

2.2.1 Computational Complexity

Computational Complexity theory enables us to classify and compare the difficulty of the problems. The class of the problems that can be solved by an algorithm in polynomial time is known as P. Tasks requiring exponential time to be solved are in class EXP. Problems whose given solution can be verified in polynomial time belong to class NP. Even though formally $P \subseteq NP$, it is commonly believed that P $\neq$ NP [30].

The problem X is considered to be **NP-hard** if it is at least as difficult as any problem in NP. This property is proven by showing a polynomial reduction from any NP problem to X [17]. The problem is **NP-complete** if it is NP-hard and belongs to class NP. If one was able to solve any NP-Complete problem Y in polynomial time, then this would prove that $NP = P$ (because of polynomial reduction from any NP problem to Y).

An efficient algorithm for an NP-complete problem has not been found. Unless $P = NP$, which is unlikely, one cannot expect any polynomial-time algorithm to find the optimal solution to all the instances of any NP-complete problem. Therefore, to obtain a **satisfactory solution** in a reasonable time, approaches like approximation (Section 2.1.2), randomization, or heuristics are used. One of the examples combining all these techniques is Evolutionary Algorithms, which have proven to be effective at finding high quality solutions for NP-Complete problems, especially those in Combinatorial Optimization [17].

2.3 Bio-inspired Algorithms

Biologically inspired computing aims to solve problems by taking inspiration from biological processes like evolution, immune systems, the behaviour of swarm colonies, etc [8, 29]. They have shown to be effective even if there is little knowledge or understanding of a problem, the solution must be found quickly, or the problem is too hard to be solved with a problem-specific algorithm.

2.3.1 Basic Concepts

The focus of this dissertation project is on Evolutionary and Genetic Algorithms. They take inspiration from Natural Evolution for the automatic generation of solutions to hard problems, and can be easily applied to any problem once: a suitable genetic representation of the solution domain is defined, a fitness function used to evaluate sample solutions is designed, and the genetic operators are chosen.

Genetic Representation

A set of potential solutions is called a population. A single candidate for a solution is called an individual, and the data it contains - a chromosome that comprises genes [29]. One of the most common genetic representations is an array of bits (a bit-string) where the individual is represented as a sequence of ones and zeroes. One means that the particular gene is in the final solution while the genes represented by zeros are excluded [29]. Binary encoding can be used for many optimisation problems, for example the Knapsack Problem (see: Figure 2.1) [25]. Other representations include trees, sequences of real values or letters [29].

Index of item	Weight	Value
1	3	10
2	1	5
3	2	9
4	1	3
5	4	20

Bit-string	Taken Items
[0, 0, 0, 0, 0]	None
[0, 0, 0, 0, 1]	No. 5
[0, 1, 0, 0, 0]	No. 2
[1, 0, 1, 0, 0]	No. 1 & No. 3
[1, 1, 1, 1, 1]	All

Figure 2.1: Binary Encoding of the Knapsack Problem

Fitness Function

Fitness Function quantitatively measures how close a particular solution is to the optimum (how fit a solution is). Its evaluation is often the most time-consuming step of an evolutionary algorithm - the expected running time is usually measured as a number of these evaluations; thus, it should always be implemented efficiently [15].

Genetic Operators

Selection is used to ensure that better individuals make relatively more offspring. The most common is a roulette wheel selection (also called a fitness proportionate selection). In this method, the probability of an individual to make an offspring is directly proportional to its fitness value with respect to the population fitness - the higher the fitness value of an individual, the more likely it is to reproduce [29].

Crossover is an operator which imitates the recombination of genetic information from two parents to generate new offspring. The crossover methods mentioned in this report include

single-point and two-point crossover. In single-point crossover a crossover point is chosen randomly on both parents' chromosomes. Genes on to the right of this point are swapped, resulting in offspring carrying the genetic information from both parents. Two-point crossover works similarly, except that two crossover points are chosen and the genetic information between them is swapped (see: Figure 2.2) [29].

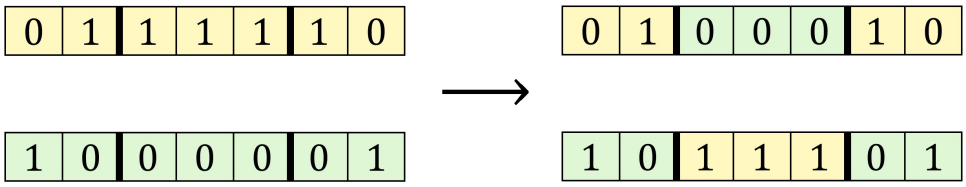


Figure 2.2: Two-point crossover

Mutation is responsible for maintaining genetic diversity in the next generations of an offspring. The most common mutation method used on a bit-string is a standard bit mutation, where each bit is flipped with probability p_m (e.g. $1/N$ where N is a length of a bit-string) [19, 29].

2.3.2 Genetic and Evolutionary Algorithms

A **Genetic Algorithm** starts by generating a population: it creates a set of possible solutions to the problem. Next, the current population is evaluated using a fitness function, and the individuals with the highest scores are chosen to be the parents of the new solutions with higher probability than individuals with lower scores. The following steps are repeated until a satisfactory solution is found or a time-budget limit has expired: the set of parents is recombined by using an appropriate crossover process and mutated by using an appropriate mutation function. The next generation of parents may or may not be chosen using a designated replacement strategy (i.e., Steady-State versus Generational Genetic Algorithms). **Evolutionary Algorithms** work the same as Genetic Algorithms except that they do not use a crossover operator (i.e. they imitate asexual evolution) [10, 19]:

Algorithm 1: The Steady-State Evolutionary Algorithm

```
Generate a population;
Evaluate fitness of the population;
Choose a subset of parents from the current population according to the employed
  selection operator;
while the optimal solution has not been found do
    Mutate set of parents using an appropriate mutation operator;
    Evaluate fitness of the mutated population;
    Choose the next generation of parents using an appropriate replacement strategy;
end
```

2.3.3 Performance of Evolutionary Algorithms

Both empirical results and theoretical studies have shown that the Evolutionary Algorithms can compete with the problem-specific algorithms on many different problems, including the maximum matching or partitioning problem [11, 12]. As stated previously, the fitness function largely contributes to the total runtime of EAs; thus, in order to achieve a competitive running time, the search space and fitness function need to be designed appropriately. For example, the efficient problem representation for the Eulerian cycles problem improved the runtime of the (1+1)EA from $O(m^4)$ to $\Theta(m\ log\ m)$ [12].

However, choosing the appropriate fitness function is very challenging and the best solution may not be obvious. As proof, the vertex cover problem may be considered. Jansen, Oliveto and Zarges [12] proved that even though the vertex-based representation seems to be intuitive in this case, changing it to the edge-based and adding more information to the fitness function allows the (1+1)EA to guarantee 2-approximations in polynomial time rather than arbitrarily bad ones in the worst case. Therefore, when designing the EA, multiple fitness functions should be analysed in terms of the overall performance.

2.4 Nonograms

Nonograms are a popular type of picture logic puzzles. They have a form of an $N \times M$ grid of cells which need to be filled in according to the description. N indicates the number of rows, and M - the number of columns (See: Figure 2.3). Description consists of the numbers on the left describing each row and on the top describing each column. Each number represents the size of a black cell block in a given row or column. If there are 2 or more numbers, the consecutive blocks of black cells must be separated by at least 1 white cell.

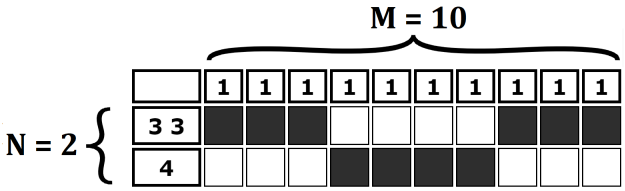
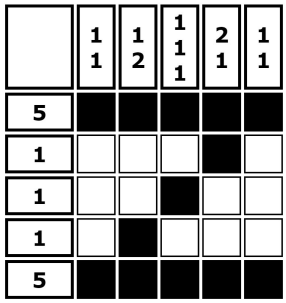
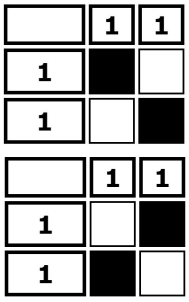


Figure 2.3: Defining the size of a Nonogram

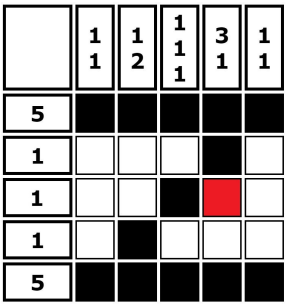
A solvable Nonogram has at least one possible solution. There exist instances that have a unique solution (see: Figure 2.4a) or multiple solutions (Figure 2.4b). Nonograms with no feasible solutions (Figure 2.4c) are unsolvable.



(a) Solvable 5×5 Nonogram



(b) Non-unique 2×2 Nonogram



(c) Unsolvable 5×5 Nonogram

Figure 2.4: Examples of Nonograms

2.4.1 NP-completeness

The general problem of determining whether the Nonogram is solvable or not is **NP-complete**. It belongs to the NP class - there exists a poly-time certifier which, given the Nonogram instance and sample solution, can verify whether the solution is correct in polynomial time. An example certifier would sum the consecutive blocks of black cells in every column and row, and compare these numbers to the Nonogram description. Proof that the Nonogram problem is NP-hard by using polynomial reduction from the 3D matching problem was given by Ueda and Nagao [28]. In the same paper [28], the authors showed that the problem of determining whether a solution to an instance of the Nonogram problem is unique or not is also NP-complete.

2.4.2 A measure of the difficulty of Nonograms

The difficulty of Nonograms ranges from the simple instances which can be solved by using only logical rules, to the highly complicated. Batenburg and Kosters created the first classification system which is widely used in the literature regarding the Nonogram problem. They divided Nonograms into 2 subclasses: simple and non-simple [4].

Simple Nonograms

Simple Nonograms can be solved by applying a set of logical rules, each involving just a single line (row or column). To solve simple Nonograms, the following SIMPLE-SOLVER algorithm has been proposed in the paper [4]:

Algorithm 2: SIMPLE-SOLVER

```
while the value of any cell is missing do
    (H-SWEEP) for each row do
        (SETTLE) Iterate through all the possible combinations of filling the line and
          for each cell i, set its pixel value to 1 (or 0) if and only if that cell has value 1
          (or 0) in every combination;
    end
     (V-SWEEP) for each column do
        Apply SETTLE;
    end
end
```

The total number of SWEEP operations needed for solving the Nonogram can be then used as a difficulty measure. Its value is between 1 and $M * N + 1$. The running time of the SIMPLE-SOLVER was not explicitly stated in any of the papers so it has been computed in this report based on the running times of its components mentioned by the authors [3, 4]. The computations described in Appendix A.1 showed that the total running time of the SIMPLE-SOLVER on a simple Nonogram instance is $\boldsymbol{O(N^2 * M^5 + N^5 * M^2)}$. However, in most of the cases, this upper bound is not tight. This running time implies that all the simple Nonograms can be solved in polynomial time.

Non-simple Nonograms

When applying SIMPLE-SOLVER to a non-simple Nonogram, a partially filled solution will be obtained. To solve non-simple Nonograms, the following (p,q)-SOLVER was proposed [4]:

Algorithm 3: (p,q)-SOLVER

```
Choose p and q such that 1 ≤ p ≤ M and 1 ≤ q ≤ N;
while the value of any cell is missing do
    for arbitrary chosen p rows and q columns do
        Find all the combinations of values for intersections of p and q (configurations);
        Apply SETTLE to p rows and q columns and keep track of those configurations
          that can be extended in every line;
        Fix unknown intersection of pixels that are the same in all of them;
    end
    if at least 1 pixel can be fixed then
        Apply SIMPLE-SOLVER;
    end
end
```

A Nonogram that can be solved by (p, q)-SOLVER, but not by any (p', q')-SOLVER, where $p' \leq p$, $q' \leq q$ and $(p', q') \neq (p, q)$, is called (p, q)-hard. The difficulty measure for non-simple Nonograms is the minimum number of successful (p, q)-intersections needed to solve the puzzle. If the Nonogram is unsolvable, its difficulty is ∞. The running time of (p,q)-SOLVER is $\boldsymbol{O(2^{p*q})}$ which is polynomial in Nonogram's size if p and q are fixed. However, if $p = M$ and $q = N$, the (p,q)-SOLVER is nothing more than a brute force algorithm which generates all the possible solutions taking exponential time in the worst case ($O(2^{M*N})$) [4].

Distribution of simple and non-simple Nonograms

In the same paper [4], Batenburg and Kosters analysed the correlation between the percentage of simple, (2,2)-hard, and unsolvable by (p,q)-SOLVER Nonograms and the percentage of black pixels in the solved instance. They generated 1000 random instances for every integer percentage. These results are shown in Figure 2.5. Based on the graph, it can be derived that most simple Nonograms have a high percentage of black cells - the more constraints, the easier it is to find "obvious" values in each line; thus, SIMPLE-SOLVER can find a complete solution.

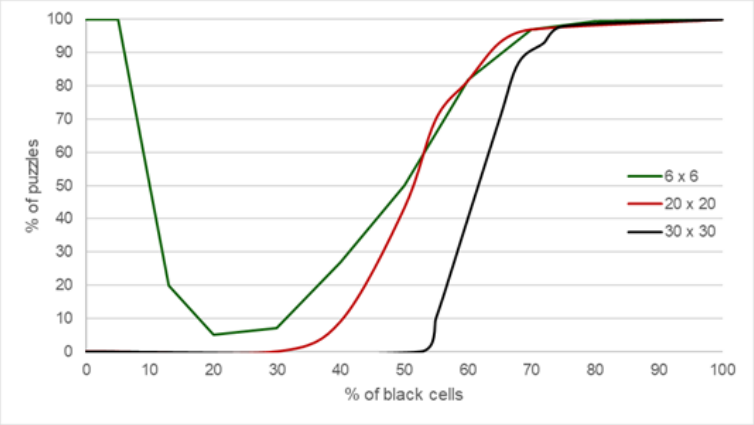


Figure 2.5: Distribution of simple Nonograms [4]

2.5 Problem-specific Algorithms for the Nonogram Problem

2.5.1 Brute Force

The simplest, but the least effective way to solve Nonograms is to exhaustively search the whole search space by using Depth First Search, for example. The algorithm takes one row at a time, generates all the possible combinations of black and white cells satisfying row constraints, and combines them with possible solutions to the previous rows. In Figure 2.6 that process has been shown for the first row (with all the other rows solved). Once the complete solutions are generated, the algorithm evaluates their correctness and returns the one fulfilling all the constraints (see: Figure 2.7).

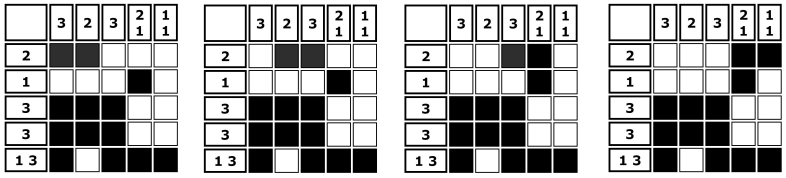


Figure 2.6: Generating solutions for the first row

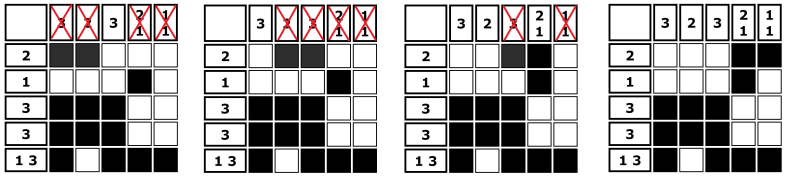


Figure 2.7: Verifying the generated solutions

Even though DFS guarantees to find a solution (if it exists), the overall performance is poor because of the size of the search space [30]. Generating and evaluating all the solutions results in an exponential time complexity ($\boldsymbol{O(2^{N*M})}$). Wiggers [30] showed that this algorithm outperforms other techniques like Genetic Algorithms only on small instances (up to 6×6).

2.5.2 Logic Reasoning

There exist many different logical approaches to deduce the location of the particular pixels based on the description. All algorithms which use only logical rules to produce the solution are referred to in this report as Logical Solvers. They may differ in the implementation (different set of logical rules, different order of applying them, etc.), but their success rate is similar [4]. Examples of Logical Solvers are SIMPLE-SOLVER and (p,q)-SOLVER described in Section 2.4.2. As stated previously, the total running time of SIMPLE-SOLVER on a simple Nonogram instance is $\boldsymbol{O(N^2 * M^5 + N^5 * M^2)}$ and the running time of (p,q)-SOLVER is $\boldsymbol{O(2^{p*q})}$ [4].

Logical Solvers can efficiently produce correct solutions to simple and (p,q)-hard Nonograms (for relatively small p and q); hence, they can be successfully applied to the majority of small and medium instances (up to 20×20), and instances with a high percentage of black cells. However, they are inefficient when applied to Nonograms with multiple possible solutions and those with a small number of constraints (see: Figure 2.8).

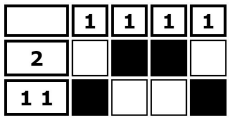


Figure 2.8: 2×4 Nonogram which cannot be solved by Logical Solver

2.5.3 Multiple Techniques

There also exist algorithms which combine logic, brute-force and heuristics techniques. The method proposed by Yu, Li, and Chen [31] combines logic with chronological backtracking (depth-first search variation). By applying logical rules before visiting the next node in the search tree, the algorithm outperforms both the simple GA and DFS described in [30]. This shows that even small improvements to the single technique algorithm may produce much better results.

More complex but also one of the most efficient algorithms used on Nonograms is the BGU solver designed by Berend, Pomeranz, Rabani and Raziel [5]. It first uses a Line Solver (like SIMPLE-SOLVER) to extract all the information using pure logic. If no more data can be obtained from the description, the algorithm guesses the pixels' colours using probing - it chooses the least probable guesses which, by quickly leading to contradictions, enables cutting branches from the search tree and reduces the search space. If no contradiction is encountered, the BGU uses heuristics to choose the colour of the particular cells. Even though the running time of this algorithm is bounded by $\boldsymbol{O(N^M)}$, in practice it outperformed seven the most efficient Nonogram solvers on all the 2,491 tests [5].

2.6 General-purpose Algorithms for the Nonogram Problem

2.6.1 Simple Genetic Algorithm

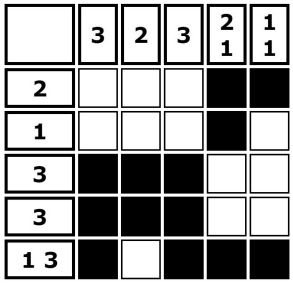
A unique approach was proposed by Wiggers [30] when he transformed a Nonogram into the corresponding Satisfiability Problem instance (see: Figure 2.9) and ran the GA on the SAT problem. To obtain more information about the fitness of the solutions, he used an averaged fitness function (see: Table 2.1). This was obtained by redefining the AND-operator as an AVG-operator and OR-operator as MAX-operator. In the implementation, the author used a two-point crossover and roulette wheel selection operator.

X	Y	Z	Fitness value
1	1	1	1
1	1	0	1
1	0	1	1
1	0	0	0.5
0	1	1	0.5
0	1	0	0.5
0	0	1	0.5
0	0	0	0

Table 2.1: Averaged fitness function for SAT: (X AND Y) OR (X AND Z) [30]

One problem encountered by Wiggers [30] when using the GA was the algorithm often got stuck in the local optima. This returned a solution close to the correct one and required a

manual restart. Additionally, the process of converting Nonograms into the SAT instance was memory and time-consuming, especially when working on big instances. However, in general, the Simple Genetic Algorithm significantly outperformed the DFS in all of his three first successful runs for all the bigger instances of Nonogram problem (7×7 and larger) [30].



(a) Nonogram instance

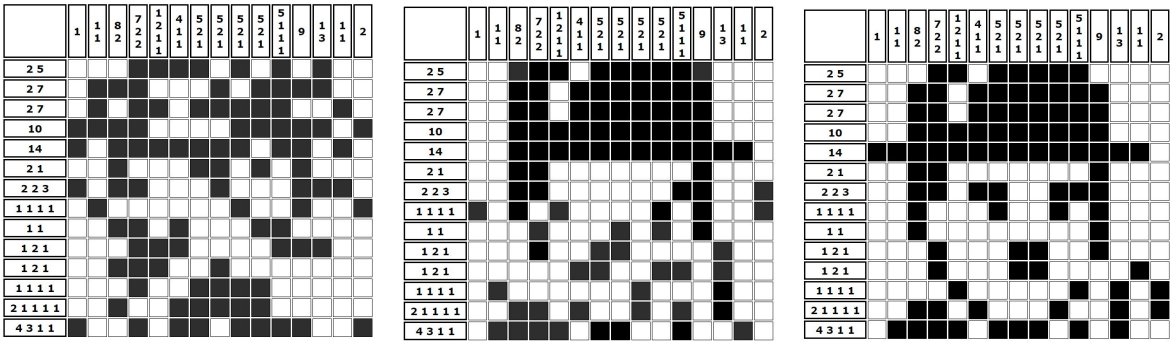
$$
\begin{aligned}
&\text{AND}\\
&\quad (X_{11} \text{ AND } X_{12} \text{ AND } \neg X_{13} \text{ AND } \neg X_{14} \text{ AND } \neg X_{15})\\
&\quad \text{OR}\\
&\quad (\neg X_{11} \text{ AND } X_{12} \text{ AND } X_{13} \text{ AND } \neg X_{14} \text{ AND } \neg X_{15})\\
&\quad \text{OR}\\
&\quad (\neg X_{11} \text{ AND } \neg X_{12} \text{ AND } X_{13} \text{ AND } X_{14} \text{ AND } \neg X_{15})\\
&\quad \text{OR}\\
&\quad (\neg X_{11} \text{ AND } \neg X_{12} \text{ AND } \neg X_{13} \text{ AND } X_{14} \text{ AND } X_{15})\\
&\text{AND}\\
&\quad (X_{21} \text{ AND } \neg X_{22} \text{ AND } \neg X_{23} \text{ AND } \neg X_{24} \text{ AND } \neg X_{25})\\
&\quad \text{OR}\\
&\quad (\neg X_{21} \text{ AND } X_{22} \text{ AND } \neg X_{23} \text{ AND } \neg X_{24} \text{ AND } \neg X_{25})\\
&\quad \text{OR}\\
&\quad (\neg X_{21} \text{ AND } \neg X_{22} \text{ AND } X_{23} \text{ AND } \neg X_{24} \text{ AND } \neg X_{25})\\
&\quad \text{OR}\\
&\quad (\neg X_{21} \text{ AND } \neg X_{22} \text{ AND } \neg X_{23} \text{ AND } X_{24} \text{ AND } \neg X_{25})\\
&\quad \text{OR}\\
&\quad (\neg X_{21} \text{ AND } \neg X_{22} \text{ AND } \neg X_{23} \text{ AND } \neg X_{24} \text{ AND } X_{25})\\
&\text{AND}\\
&\ldots\\
&\text{AND}\\
&\quad (X_{51} \text{ AND } \neg X_{52} \text{ AND } X_{53} \text{ AND } X_{54} \text{ AND } X_{55})
\end{aligned}
$$

(b) SAT representation of the Nonogram

Figure 2.9: Transformation of a Nonogram into a SAT instance

2.6.2 Canonical Genetic Algorithm

Salcedo-Sanz et al. [22] proposed a few different bio-inspired approaches for the Nonogram Problem. First, they used the Canonical Genetic Algorithm (roulette wheel selection, single-point crossover, gene-wise mutation [23]). They encoded $N \times M$ Nonogram instances into binary strings of length $N * M$ so that the first M bits corresponded to the first row, the next M bits - to the second row, etc. The search space was of size $\{0, 1\}^{N*M}$. The objective function was defined as $g(X) = N * M - f(X)$ where $f(X)$ was the sum of differences between the desired number of 1's and the actual number of 1's in each row and column, and between the desired and the actual number of 0's in every row and column. The algorithm was tested on single Nonogram instance. The solution produced after 50,000 fitness evaluations is shown in Figure 2.10a. Since this approach did not produce the correct result, the authors [22] proposed a set of improvements that will be described in the following sections.



(a) (b) (c)

Figure 2.10: The result of 3 different GAs applied to the same Nonogram

2.6.3 EA with Advanced Encoding and Special Operators

The Evolutionary Algorithm with Advanced Encoding and Special Operators is a special case of the EA where all the individuals are feasible in columns (but not necessarily in rows), so the search space reduces to $\{1,0\}^N$. The algorithm makes use of heuristics, which creates solutions feasible in columns - satisfying all the columns' constraints. It chooses the initial-filled square at random between the first (usually 1^{st} bit) and last possible square (computed from column constraints). EA operators are implemented in a special way to avoid the situation where they would produce offspring that is infeasible in columns. A mutation changes the position of the alleles (a group of C cells where C is the number of conditions in a given column) instead of genes (single cells), and a crossover swaps columns from two parents to produce a new individual. The fitness function is similar to the one used in the Canonical GA ($g(X) = N * M - f(X)$) except that now $f(X)$ is just a sum of differences between the desired and actual number of zeroes and ones respectively in every row. The performance of this EA on a given test instance after 50,000 fitness evaluations was better than the canonical GA (Figure 2.10b). However, it still did not solve the given test puzzle.

2.6.4 EA and Local Search

The most efficient and successful algorithm proposed by Salcedo-Sanz et al. [22] was a hybrid algorithm combining an Evolutionary Algorithm with Advanced Encoding and Special Operators with a Hill-Climbing. That algorithm was the only one that managed to solve the given test Nonogram (Figure 2.10c). It is worth emphasizing that the three algorithms described above were tested only on a single complex and big (15×15) example, so it is not possible to deduce their performance on smaller instances.

2.6.5 Other Approaches

Other bio-inspired approaches to the Nonogram problem include Dynamic Inertia Weight Particle Swarm Optimization (DIW-PSO) [20], or more complex EA with ternary encoding,

Diversity-Guided Mutation and special initialization [1]. This Evolutionary Algorithm significantly outperformed the EA described in [22] (Section 2.6.4) in all of the 16 tests conducted by the authors [1].

2.7 Summary of the Techniques

The problem-specific solving techniques described in this report are summarized in Table 2.2. These methods are efficient mainly on the smaller instances or simple Nonograms with a high percentage of black cells.

Algorithm	**Runtime**	**What it can be used on?**
DFS	$O(2^{N*M})$	all instances
SIMPLE-SOLVER	$O(N^2 * M^5 + N^5 * M^2)$	only simple Nonograms
(p,q)-SOLVER	$O(2^{p*q}) = O(2^{N*M})$	all instances
BGU Solver	$O(N^M)$	all instances

Table 2.2: A comparison of problem-specific solving techniques

Most Genetic Algorithms used for the Nonogram Problem work well on small or medium-size instances. Some of the more advanced and complex EAs can solve bigger Nonograms, especially those with a smaller number of constraints. The runtime analysis of any of the Genetic or Evolutionary Algorithms on the Nonogram problem has not yet been conducted. The algorithms and their efficiency were tested experimentally, mostly on different instances. Hence, with these results, it is not possible to explicitly predicate which bio-inspired approach is the best to use for solving the Nonogram problem. In this project, for the first time, rigorous statements regarding the performance of EAs for this problem will be provided.

Chapter 3

Requirements and Analysis

This Chapter sets aims for the project (Section 3.1), justifies and describes algorithms which will be used in the analysis (Section 3.2), and summarizes the most commonly used techniques for analysing the runtime of Evolutionary Algorithms (Section 3.3).

3.1 Aims of the Project

The main aim of this project is to create an appropriate fitness function and analyse the performance of Evolutionary Algorithms for the Nonogram problem. The analysis will be performed on the (1+1)Evolutionary Algorithm and Random Local Search. The results of this project may provide a context for further analysis of more complex Evolutionary Algorithms applied to the Nonogram problem. The work is divided into two main parts - Design (3.1.1) and Runtime Analysis (3.1.2).

3.1.1 Design

Since the (1+1)Evolutionary Algorithm and Random Local Search have not been used to solve Nonograms, they will need to be designed specifically for this problem. Therefore, before the mathematical analysis, the following steps need to be completed:

1. Define an appropriate problem representation,
2. Create the fitness function for the (1+1)EA and RLS,
3. Identify the subclasses of the Nonogram problem which are solvable in polynomial time.

3.1.2 Runtime Analysis

Once all the steps described in Section 3.1.1. are completed, it will be possible to start analysing the runtime of the defined algorithms. The project aims to:

4. Derive an upper bound on the expected running time of RLS on the Nonograms' subclasses which are solvable in polynomial time,

5. Derive an upper bound on the expected running time of the (1+1)EA on the Nonograms' subclasses which are solvable in polynomial time,

6. Evaluate and check the correctness of the obtained mathematical proofs.

3.2 Algorithms chosen for the analysis

Since the runtime analysis of EAs for a specific problem is very challenging, it is often convenient to first consider the behaviour of a simplified Evolutionary Algorithm and then assess the performance of more sophisticated EAs based on these results [19]. That is why the analysis will be performed on the (1+1)Evolutionary Algorithm and Random Local Search - one of the simplest yet powerful Evolutionary Algorithms.

3.2.1 The (1+1)Evolutionary Algorithm

The simplest version of an Evolutionary Algorithm is the (1+1)EA where the initial and the succeeding populations are of size 1. The genetic representation of the solution domain is usually a bit-string. The algorithm creates an initial bit-string x of length N by setting each bit to 1 with probability $\frac{1}{2}$, and evaluates it using a fitness function. Then, it repeats the following steps: it creates a new bit-string x' by flipping each bit of x independently with probability $p_m := 1/N$. Next, it evaluates x' using a fitness function and chooses x' to be the new parent generation if its fitness value is at least as good as the value of x [10, 18].

Algorithm 4: The (1+1)EA

Choose randomly an initial bit-string $x \in \{0,1\}^N$;
while *the optimal solution has not been found* **do**
    Compute x' by flipping independently each bit in x with probability $p_m := 1/N$;
    Replace x by x' iif $f(x') \geq f(x)$ where f is a fitness function $f : \{0,1\}^N \longrightarrow \mathbb{R}$;
end

3.2.2 Randomized Local Search

Randomized Local Search is even simpler than the (1+1)EA: instead of flipping each bit of the parent bit-string with probability $p_m := 1/N$, it flips exactly 1 bit chosen at random in each iteration [16, 18].

Algorithm 5: RLS

Choose randomly an initial bit-string $x \in \{0,1\}^N$;
while *the optimal solution has not been found* **do**
    Compute x' by flipping the i^{th} bit of x, chosen uniformly at random;
    Replace x by x' iif $f(x') \geq f(x)$ where f is an fitness function $f : \{0,1\}^N \longrightarrow \mathbb{R}$;
end

3.3 Analysing the runtime of Bio-inspired Algorithms

The research about rigorous runtime analysis of Evolutionary Algorithms is relatively young. Until the 1990s, the main theoretical findings about EAs dealt only with their convergence and behaviour in a single iteration. The first runtime analysis was conducted by Mühlenbein in 1992. In his research, he used the fact that Evolutionary Algorithms are a subclass of randomized algorithms and, thus, argued they should be analysed in the same way [17]. The main methods used for the runtime analysis of stochastic search algorithms are described below.

3.3.1 Artificial Fitness Levels

This is a very simple technique to calculate the upper bounds on the runtime of the stochastic search algorithms. To apply this method, we need to partition the search space $S = \{0,1\}^n$ of size 2^n into m disjoint fitness-based sets A_1, A_2, ..., A_m such that $A_1 \cup A_2 \cup ... \cup A_m = S$ and $A_i \cap A_j = \emptyset$ for $i \neq j$. Partitions are of increasing fitness: $f(A_1) < f(A_2) < ... < f(A_m)$. A_m should contain only the global optimum (or global optima) [14].

Elitist algorithms such as the (1+1)EA or RLS only accept solutions that belong to levels with higher or equal fitness to the current level solution, and never go back to the previous levels. In the worst case, all fitness levels are visited. Artificial fitness-level technique uses that fact to bound the expected time to reach the optimum as the sum of lower bounds s_i on the probability of leaving each level A_i [14]. In order for an upper bound to be tight, the fitness-based partition must be used with not too many partitions and with a high probability of leaving the current level [17].

Theorem 1 (**Artificial Fitness Levels [14]**). *Let $f : S \to \mathbb{R}$ be a fitness function, $A_1, ..., A_m$ be a fitness-based partition of f, and $s_1, ...s_{m-1}$ be lower bounds on the corresponding probabilities of leaving the respective fitness levels for a level of better fitness. Then the expected runtime of an elitist algorithm using a single individual is $\mathbb{E}[T_{A,f}] \leq \sum_{i=1}^{m-1} 1/s_i$.*

To show the usage of the AFL in practice, we derive the expected optimisation time of the (1+1)EA on a simple function called $OneMax(x) := \sum_{i=1}^{n} x_i$ which counts the number of one-bits in a bit-string.

Lemma 1. *The expected runtime of the (1+1)EA on $OneMax$ is $O(n\ log(n))$.*

Proof. Let's partition the search space into $n+1$ sets $A_0, ..., A_n$ such that A_i contains all the bit-strings x with i ones and $n-i$ zeroes ($OneMax(x) = i$). To reach the level of higher fitness it is necessary to flip a zero-bit into a one and leave the remaining bits unchanged. Since, for level A_i, the probability of reaching the higher level is bounded by s_i:

$$s_i \geq (n-i) * \frac{1}{n} * (1 - \frac{1}{n})^{n-1} \geq \frac{n-i}{en} \tag{3.1}$$

where the second inequality holds because $(1 - 1/n)^{n-1} \geq 1/e$ for all $n \in N$.
Then, by using AFL:

$$\mathbb{E}[T_{(1+1)EA,OneMax}] \leq \sum_{i=0}^{n-1} \frac{en}{n-i} = en \sum_{i=1}^{n} \frac{1}{i} = O(n\ log(n)) \tag{3.2}$$

□

3.3.2 Drift Analysis

Development of drift analysis significantly improved the understanding of Evolutionary Algorithms by either simplifying the existing results or obtaining the more precise ones. Quoting an example from Lengler's "Drift Analysis" [9], the original proof by Droste, Jansen, and Wegener that the (1+1)EA takes time $O(n\ log(n))$ on all linear functions needed 7 pages, while Doerr, Johannsen, and Winzen could reduce the proof to a single page using drift analysis.

The idea behind this method is to predict the long-term behaviour of a stochastic process X by measuring the expected progress towards a target in a single step. Let X_k be a random variable representing the current state of the process at step k over a finite set of states S. In order to apply Drift Analysis, the following measures need to be defined [14]:

- a distance function $d : S \to \mathbb{R}_0^+$ such that $d(X_k) = 0$ that assigns a non-negative real number to every state if and only if X_k is a target point,
- the initial distance $d(X_0)$ or the initial expected distance $\mathbb{E}[d(X_0)]$,
- the expected change in distance in one step: a drift $\Delta(X_k) = \mathbb{E}[d(X_{k+1}) - d(X_k)|X_k]$.

Depending on a problem and a fitness function, different drift theorems can be applied.

Additive Drift Theorem

Additive Drift Theorem derives both upper and lower bounds on the runtime by considering the worst-case decrease in distance. If at each time step k the drift is **at least** $-\sigma$, the expected number of steps to reach the target is **at most** $\mathbb{E}[d(X_0)]/\sigma$. Similarly, if the drift is **at most** $-\sigma$, the the expected number of steps to converge is **at least** $\mathbb{E}[d(X_0)]/\sigma$ [14, 19].

Theorem 2 (**Additive Drift Theorem for Upper Bounds [19]**). *For stochastic process X and distance function d, let $T := min\{t \geq 0 : d(X_t) = 0\}$ be the time to reach the target. If*

$$\forall k\ \mathbb{E}[d(X_k) - d(X_{k+1}) \mid d(X_k) > 0] \ \geq \ \sigma \tag{3.3}$$

then for $\sigma > 0$:

$$\mathbb{E}[T] \leq \frac{\mathbb{E}[d(X_0)]}{\sigma}. \tag{3.4}$$

Theorem 3 (**Additive Drift Theorem for Lower Bounds [19]**). *For stochastic process X and distance function d, let $T := min\{t \geq 0 : d(X_t) = 0\}$ be the time to reach the target. If*

$$\forall k \ \mathbb{E}[d(X_k) - d(X_{k+1}) \mid d(X_k) > 0] \ \leq \ \sigma \tag{3.5}$$

then for $\sigma > 0$:

$$\mathbb{E}[T] \geq \frac{\mathbb{E}[d(X_0)]}{\sigma}. \tag{3.6}$$

The main drawback of this method is that the derived bound is not tight if the expected decrease in distance changes in different areas of the search space [14, 19]. As an example of this, let's consider OneMax problem again.

Lemma 2. *The expected running time of the (1+1)EA on OneMax using Additive Drift Theorem is $O(n^2)$.*

Proof. In order to derive an upper bound, we need to find the worst case improvement. That case occurs when the current search point is optimal, except for one zero-bit. The maximum decrease in distance that may be achieved in a single step is Y_t - $Y_{t+1} = 1$. In order to move to the optimum, the (1+1)EA needs to flip the zero bit into a one and leave the others unchanged. Hence, the drift is bounded by:

$$\Delta_t \geq 1 * \frac{1}{n} * (1 - \frac{1}{n})^{n-1} \geq \frac{1}{en} := \sigma \tag{3.7}$$

By applying Theorem 2 (Additive Drift Theorem for Upper Bounds [19]), we get:

$$\mathbb{E}[T] \leq \frac{\mathbb{E}[d(X_0)]}{\sigma} = \frac{n/2}{1/en} = \frac{en^2}{2} = O(n^2) \tag{3.8}$$

Where the expected distance $\mathbb{E}[d(X_0)] = n/2$ due to random initialisation. □

Multiplicative Drift Theorem

In cases when the amount of progress depends on the distance from the target, Multiplicative Drift Theorem gives better bounds on the runtime.

Theorem 4 (**Multiplicative Drift Theorem [14, 9]**). *Let $\{X_t\}_{t \in \mathbb{N}_0}$ be random variables describing a Markov process over a finite state space $S \subseteq \mathbb{R}$. Let T be the random variable that denotes the earliest point in time $t \in N_0$ such that $X_t = 0$. If there exist δ, c_{min}, $c_{max} > 0$ such that for all $t < T$,*

$$\mathbb{E}[X_t - X_{t+1} \mid X_t] \ \geq \ \delta X_t \tag{3.9}$$

$$c_{min} \leq X_t \leq c_{max}, \tag{3.10}$$

then

$$\mathbb{E}[T] \leq \frac{2}{\delta}\ ln\left(1 + \frac{c_{max}}{c_{min}}\right) \tag{3.11}$$

Let's use this method to derive an upper bound on the runtime of the (1+1)EA on OneMax.

Lemma 3. *The expected running time of the (1+1)EA on OneMax using Multiplicative Drift Theorem is $O(n\ ln(n))$.*

Proof. Let X_t be the number of zero-bits in a bit-string at time t, representing the distance to the optimal solution. Increases in distance are not accepted due to elitism. If a zero-bit is flipped and nothing else, the distance decreases by one. By considering the above, the expected decrease in distance at one step can be bounded by:

$$\mathbb{E}[X_{t+1} \mid X_t]\ \leq\ X_t - 1 * \frac{X_t}{en} = X_t\left(1 - \frac{1}{en}\right) \tag{3.12}$$

Then the drift can be computed as follows:

$$\mathbb{E}[X_t - X_{t+1} \mid X_t]\ \geq\ X_t - X_t\left(1 - \frac{1}{en}\right) = \frac{1}{en}X_t := \delta X_t \tag{3.13}$$

By bounding $1 = c_{min} \leq X_t \leq c_{max} = n$ and applying the Theorem 4 we get:

$$\mathbb{E}[T]\ \leq\ \frac{2}{\delta}\ ln\left(1 + \frac{c_{max}}{c_{min}}\right) = 2en\ ln\left(1 + \frac{n}{1}\right) = O(n\ ln(n)) \tag{3.14}$$

□

Chapter 4

Design

Chapter 4 shows how the Nonogram's instance and its constraints will be encoded. Next, it explains why the runtime analysis of the Evolutionary Algorithms on the general instance of Nonograms is redundant, and describes the subclasses that the runtime analysis will be performed on. Finally, Section 4.3 describes fitness functions proposed in this report, and compares them with the function taken from the literature.

4.1 Problem representation

To apply RLS and the (1+1)EA to the Nonogram problem, genetic representation of the solution domain and the fitness function must be defined. Both of them are based on the definitions from the "Teaching Advanced Features of Evolutionary Algorithms using Japanese Puzzles" paper [22].

In this report, the puzzle will be encoded as a binary string of length $N * M$ (first M bits corresponding to the first row, next M bits – to the second row, etc.). Zero means that the given cell is white, and one - that it is black. Figure 4.1 shows the solved Nonogram and its corresponding bit-string.

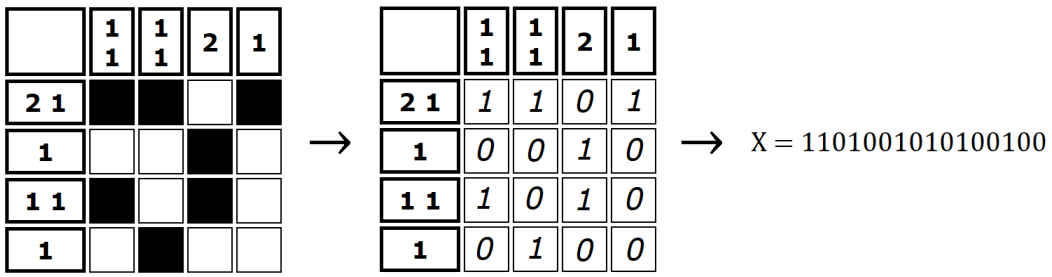


Figure 4.1: Solved Nonogram instance and the corresponding bit-string

The fitness function must operate on the list of constraints for each row and column, and correctly rank the solutions based on their correctness. Therefore, the Nonogram's description needs to be encoded as well. The rows constraints will be stored in an $N \times M$ matrix r and columns constraints - in an $M \times N$ matrix c. A maximum number of constraints in each line is the number of cells in that line. Each constraint corresponds to one entry in the matrix. If there are no more constraints to add, the remaining entries in the matrix are equal to zero. The process of creating those matrices is shown in Figure 4.2. In that example there are only two constraints for the first row: 2 and 1; hence, $r_{1,1} = 2$, $r_{1,2} = 1$ and $r_{1,3}$ and $r_{1,4}$ are both equal to 0.

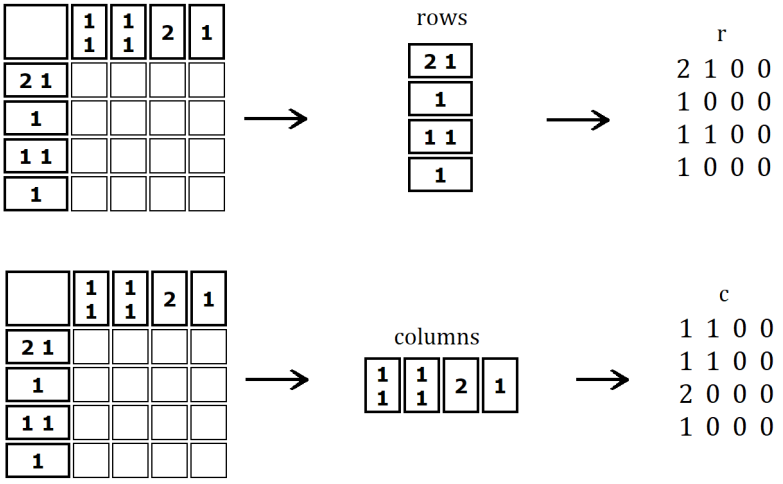


Figure 4.2: An example of creating matrices r and c

For each Nonogram instance, matrices similar to c and r will be created: $N \times M$ matrix X_r to store the information about the black cell blocks in each row, and $M \times N$ matrix X_c to store the information about the black cell blocks in each column (see: Figure 4.3). A solution is optimal if X_r is equal to r, and X_c is equal to c.

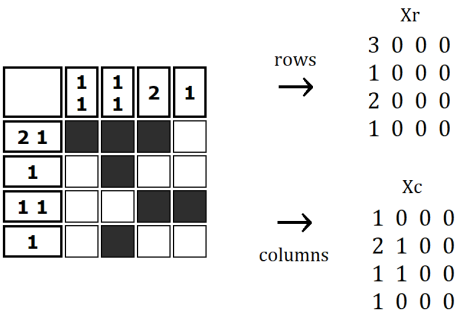


Figure 4.3: An example of creating matrices X_r and X_c

4.2 Subclasses of Nonograms solvable in polynomial time

Since the Nonogram problem is NP-Hard, there is no polynomial-time algorithm that would find the solution to every instance. Hence, the running time of the (1+1)EA on any instance is bounded by $O((NM)^{NM})$ [19] and the running time of RLS is ∞ (it may never find the solution) [19]. We may be able to prove tighter upper bounds on the runtime by considering only the subclasses of Nonograms that can be solved in polynomial time.

We will consider three main subclasses of Nonograms: fully filled, empty and single-line. In the following sections, the description of each subclass is followed by the proof that a given subclass can be solvable in polynomial time. This is done by deriving the total running time of the DFS and SIMPLE-SOLVER on a given set of instances. Since the authors [5] did not specify the implementation of the BGU Solver, the tight upper bound on the runtime is impossible to derive.

4.2.1 Empty Nonograms

The instances with no black cells are the simplest subclass of the Nonogram problem. The $N \times M$ Nonogram is considered in this report as empty if the rows constraints matrix r and columns constraints matrix c are zero matrices (see: Figure 4.4).

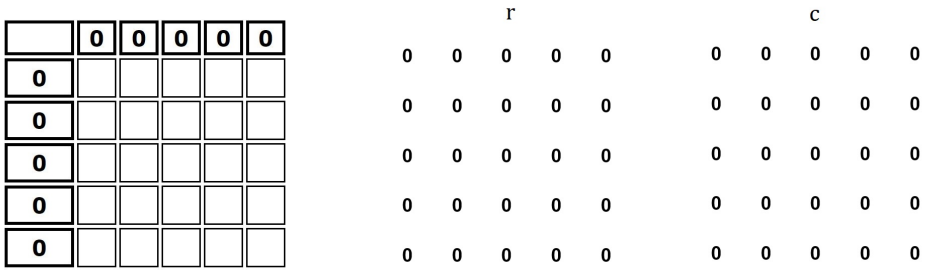


Figure 4.4: Empty Nonogram alongside the constraints matrices

DFS takes one row at a time, and generates all the possible combinations of black and white cells satisfying the row constraints (in $O(2^M)$ time). It then combines them with possible solutions to the previous rows. Since all the constraints are zeroes, there is only one combination satisfying a constraint for every row. Hence, the time needed for combining all the possible solutions is constant ($O(1)$). The total running time of the DFS is the time needed for generating all the possible solutions for each row multiplied by the number of rows: $O(N * 2^M)$. **SIMPLE-SOLVER** solves the Empty Nonogram using one H-SWEEP operation. For each of N rows, it applies SETTLE operation which takes $O(M^3)$ time (see: Appendix A.1). Hence, the total running time is bounded by $O(N * M^3)$.

4.2.2 Fully Filled Nonograms

The Nonograms with all the cells filled are another polynomial-time subclass of the problem. They are referred to in this report as fully filled. Row constraints matrix r of every Nonogram in this subclass has one constraint of size M for each row. Column constraints matrix c has one constraint of size N for each row. The remaining entries are zeroes. An example of the fully filled Nonogram alongside the constraints matrices is shown in Figure 4.5.

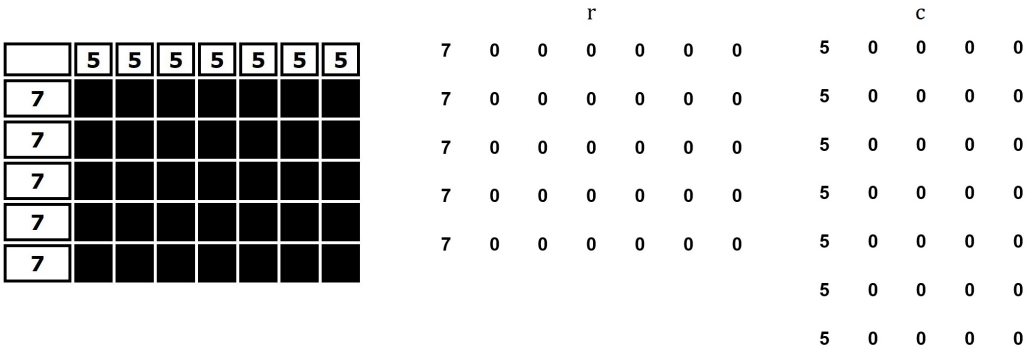


Figure 4.5: Fully filled Nonogram alongside the constraints matrices

As for the empty Nonogram subclass, only one possible combination for filling each row of the fully filled Nonogram exists. Thus, the running time of the **DFS** and **SIMPLE-SOLVER** will be the same as in the previous section: $O(N * 2^M)$ and $O(N * M^3)$ respectively.

4.2.3 Vertical Single-line Nonograms

Nonograms in this subclass can have various descriptions, but they must be of size $N \times 1$. Their rows constraints matrices are of size $N \times 1$, where each entry has a value of 1 or 0. Their column constraint matrices are of size $1 \times N$. Figure 4.6 shows the instance of the vertical single-line Nonogram.

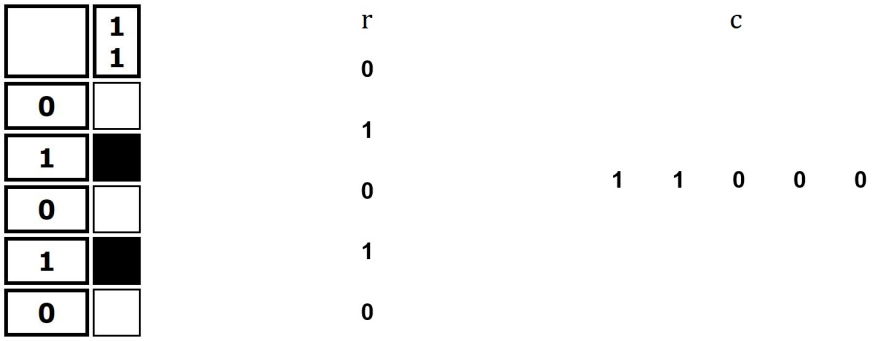


Figure 4.6: Vertical single-line Nonogram alongside the constraints matrices

Since each row is of size 1, **DFS** generates the solution for it in $O(1)$ time. The total running time of this algorithm is the time needed for generating all the possible solutions for each row multiplied by the number of rows: $O(N)$. **SIMPLE-SOLVER** solves the vertical single-line Nonogram using one H-SWEEP operation. For each of the N rows, it applies SETTLE operation which takes $O(1^3) = O(1)$ time. Therefore, the total running time is the same as for the DFS: $O(N)$.

4.2.4 Horizontal Single-line Nonograms

Nonograms in this subclass are of size $1 \times M$. As in the previous case, their descriptions may vary. Their row constraint matrices consist of 1 row with M numbers. Their column constraints matrices are of size $M \times 1$, where each entry has a value of 1 or 0. The horizontal single-line Nonogram is shown in Figure 4.7.

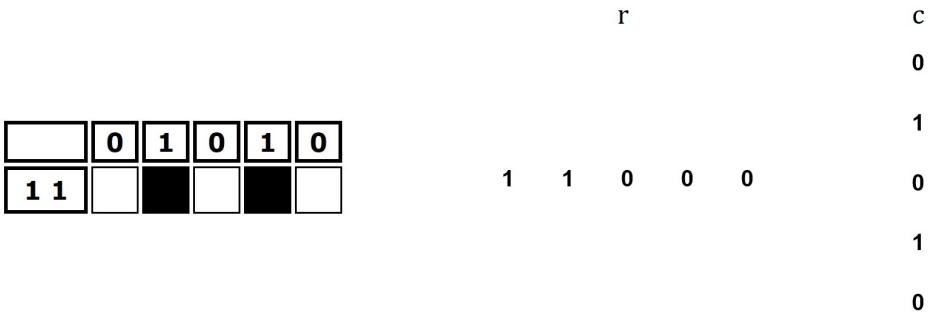


Figure 4.7: Horizontal single-line Nonogram alongside the constraints matrices

Even though the horizontal and vertical single-line Nonograms seem to be equally easy to solve, the algorithms need much more time to generate a solution for horizontal instances. **DFS** needs $O(2^M)$ to generate all the possible solutions to one row. **SIMPLE-SOLVER** solves the horizontal instance using at most one H-SWEEP ($O(M^4)$) and one V-SWEEP ($O(M)$) operation. Hence, the total running time is bounded by $O(M^4)$.

Subclass	**DFS**	**SIMPLE-SOLVER**
Empty	$O(N * 2^M)$	$O(N * M^3)$
Fully Filled	$O(N * 2^M)$	$O(N * M^3)$
Vertical Single-line	$O(N)$	$O(N)$
Horizontal Single-line	$O(2^M)$	$O(M^4)$

Table 4.1: Problem-specific techniques on different Nonogram's subclasses

4.3 Fitness Function

The fitness function determines correctness, efficiency and the running time of Evolutionary Algorithms. In the following sections, four fitness functions will be described and evaluated. Two of them are taken from or based on the "Teaching advanced features of Evolutionary Algorithms using Japanese puzzle" paper [22]. Fitness functions described in Section 4.3.3 and Section 4.3.4 are proposed in this report.

4.3.1 $g(X)$ - fitness function proposed by Salcedo-Sanz et al. [22]

The fitness function proposed by Salcedo-Sanz et al. [22] is defined as $g(X)$ (Equation 4.1). $f(X)$ is the sum of differences between the desired number of ones and the actual number of ones in each row and column, and between the desired and the actual number of zeroes in every row and column:

$$g\left(X\right) = N * M - f\left(X\right) \tag{4.1}$$

$$f\left(X\right) = f_1\left(X\right) + f_2\left(X\right) + f_3\left(X\right) + f_4\left(X\right) \tag{4.2}$$

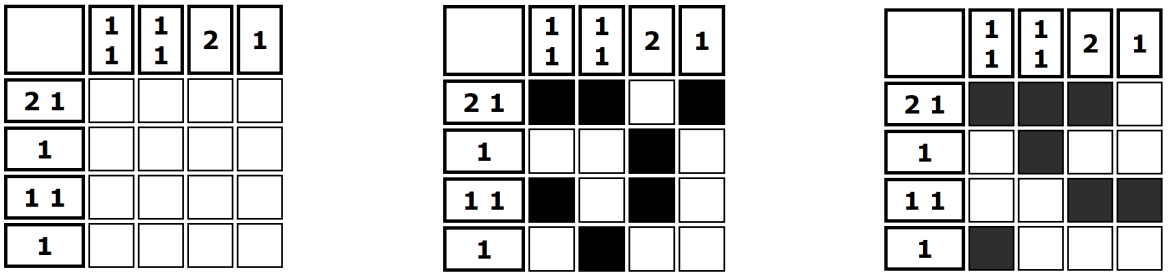
$$f_1(X) \;=\; \sum_{i=1}^{N} \left| \sum_{p=1}^{M} r_{i,p} \;-\; \sum_{p=1}^{M} X r_{i,p} \right| \tag{4.3}$$

$$f_2(X) \;=\; \sum_{j=1}^{M} \left| \sum_{p=1}^{N} c_{j,p} \;-\; \sum_{p=1}^{N} X c_{j,p} \right| \tag{4.4}$$

$$f_3(X) \;=\; \sum_{i=1}^{N} \left| (M \;-\sum_{p=1}^{M} r_{i,p}) \;-\; (M \;-\; \sum_{p=1}^{M} X r_{i,p}) \right| \tag{4.5}$$

$$f_4(X) \;=\; \sum_{j=1}^{M} \left| (N \;-\sum_{p=1}^{N} c_{j,p}) \;-\; (\;N \;-\sum_{p=1}^{N} X c_{j,p}) \right| \tag{4.6}$$

However, the approach taken by Salcedo-Sanz et al. [22] does not work well for all the Nonogram instances. It only checks for a correct number of black cells in each row and column, but not for their correct position. Therefore, in case of multiple constraints in each row or column, this approach may fail. For example, for the Nonogram defined in Figure 4.8a, both the correct solution (Figure 4.8b) and the solution shown in Figure 4.8c have the same maximum value of fitness function (16), while it is clear that only one solution is correct (4.8b). In the worst-case, for $p \times p$ Nonograms (where p is odd), $g(X)$ may classify the solution with $(p-1)^2$ incorrect cells as the optimal one (see: Figure 4.9).



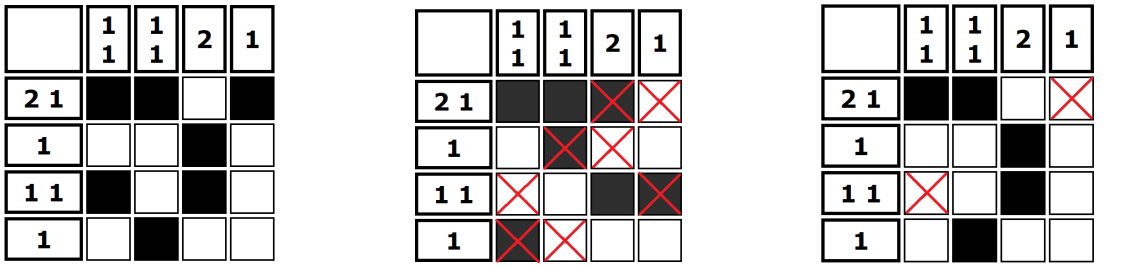
(a) (b) (c)

Figure 4.8: $g(X)$ incorrectly identifying the optimal solution

(a) Optimal Solution (b) Incorrect Solution

Figure 4.9: Solutions classified by $g(X)$ as the optimum

This fitness function not only classifies the wrong solutions as correct but also leads to incorrect results. In Figure 4.10a the correct solution with the fitness function value 16 has been shown. Example 4.10b also has a fitness value equal to 16 even though 8 cells need to be changed to reach the correct solution. The solution shown in Figure 4.10c is closer to the correct one, only 2 cells need to be changed, however, its fitness function is only 8. Hence, the algorithm would choose the 4.10b instance to be in the parent generation with a higher probability than the instance in Figure 4.10c.



(a) Optimal solution (b) Example with 8 wrong cells (c) Example with 2 wrong cells

Figure 4.10: $g(X)$ leading to the incorrect optimal solution

This might also be the reason why the Canonical Genetic Algorithm described in Section 2.6.2 [22] performed poorly on the given test instance. The returned solution was not correct but its fitness function was relatively high ($g(X) = 160$ with $f_1(X) = f_3(X) = 11$, $f_2(X) = f_4(X) = 14$ where the maximum value of fitness function was $N * M = 14 * 15 = 210$).

4.3.2 $g'(X)$ - simplified fitness function $g(X)$

$$g'(X) = N * M - f'(X) \tag{4.7}$$

$$f'(X) = f_1(X) + f_2(X) \tag{4.8}$$

It is worth noticing that for every X, $f_1(X) = f_3(X)$ and $f_2(X) = f_4(X)$. It follows that $f(X)$ (Equation 4.2) can be rewritten as $2 * (f_1 + f_2)$. Multiplying a function by a positive factor only stretches (or shrinks) the graph of the function but it does not change the relationship between different points on the graph. For functions $f(X) = 2 * (f_1 + f_2)$ and $k(X) = f_1 + f_2$, for every x_1, x_2 if $f(x_1) \leq f(x_2)$ then $k(x_1) \leq k(x_2)$ and if $f(x_1) \geq f(x_2)$ then $k(x_1) \geq k(x_2)$.

$$f_3(X) = \sum_{i=1}^{N} \left| (M - \sum_{p=1}^{M} r_{i,p}) - (M - \sum_{p=1}^{M} X r_{i,p}) \right| = \sum_{i=1}^{N} \left| \sum_{p=1}^{M} X r_{i,p} - \sum_{p=1}^{M} r_{i,p} \right| \tag{4.9}$$

$$f_1(X) = \sum_{i=1}^{N} \left| \sum_{p=1}^{M} r_{i,p} - \sum_{p=1}^{M} X r_{i,p} \right| = \sum_{i=1}^{N} \left| \sum_{p=1}^{M} X r_{i,p} - \sum_{p=1}^{M} r_{i,p} \right| = f_3(X) \tag{4.10}$$

$$f_4(X) = \sum_{j=1}^{M} \left| (N - \sum_{p=1}^{N} c_{j,p}) - (N - \sum_{p=1}^{N} X c_{j,p}) \right| = \sum_{j=1}^{M} \left| \sum_{p=1}^{N} X c_{j,p} - \sum_{p=1}^{N} c_{j,p} \right| \tag{4.11}$$

$$f_2(X) = \sum_{j=1}^{M} \left| \sum_{p=1}^{N} c_{j,p} - \sum_{p=1}^{N} X c_{j,p} \right| = \sum_{j=1}^{M} \left| \sum_{p=1}^{N} X c_{j,p} - \sum_{p=1}^{N} c_{j,p} \right| = f_4(X) \tag{4.12}$$

$$f(X) = f_1(X) + f_2(X) + f_3(X) + f_4(X) = 2 * f_1(X) + 2 * f_2(X) \tag{4.13}$$

That means that $g'(X)$ (Equation 4.7) and $g(X)$ (Equation 4.1) work identically; thus, all the bounds on the running time derived for $g'(X)$ hold for $g(X)$. That is why in order to access the performance of $g(X)$, we will conduct the runtime analysis on the simpler fitness function $g'(X)$.

4.3.3 $k(X)$ - fitness function for multiple constraints

We propose a new fitness function $k(X)$ (Equation 4.14) that deals with multiple constraints in a row. It computes the difference in size of the black cell blocks in each row and column between the constraint matrices and the matrices of the sample solution X.

$$k(X) = N * M - k'(X) \tag{4.14}$$

$$k'(X) = k_1(X) + k_2(X) \tag{4.15}$$

$$k_1(X) = \sum_{i=1}^{N} \sum_{p=1}^{M} |r_{i,p} - Xr_{i,p}| \tag{4.16}$$

$$k_2(X) = \sum_{j=1}^{M} \sum_{p=1}^{N} |c_{j,p} - Xc_{j,p}| \tag{4.17}$$

Considering the examples given in Figure 4.10, the solution in Figure 4.10b would now have a fitness function equal to 8, while the instance shown in Figure 4.10c – 12. Hence, the algorithm would now choose the 4.10c instance to be in the parent generation with a higher probability than the instance in Figure 4.10b. The example above shows that $k(X)$ works well on the complex and partially solved instances (on the contrary to $g(X)$ and $g'(X)$).

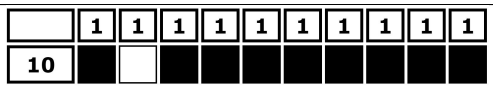
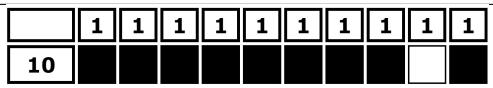
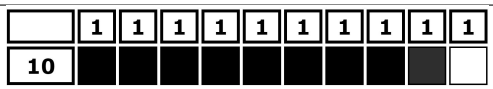
ID	Nonogram	$k(X)$
1		$10 - 9 - 8 - 1 = -8$
2		$10 - 2 - 1 - 1 = 6$
3		$10 - 1 - 1 = 8$

Table 4.2: $k(X)$ on different partial solutions

However, $k(X)$ may be inefficient when applied to Nonograms with a single constraint for each row or column. Table 4.2 shows that $k(X)$ differently rates three partial solutions for a fully filled 1×10 Nonogram with the same number of correct cells. In the example in row 1, the algorithm is looking for 1 cell block of length 10 but it gets 2 blocks of lengths 1 and 8. Hence, it seems that 9 more cells need to be added to the first block and all the cells need to be removed from the second block. In case of a solution in row 2, the algorithm gets 2 blocks of length 8 and 1, so it needs to add 2 cells to the first block and remove the second block. The last solution would be selected as the best out of these examples - only 1 black

cell needs to be added. However, all the solutions are equally good and ideally, the fitness function should return the same value for all of them.

4.3.4 $p(X)$ - fitness function checking the correct position of the cells blocks

This fitness function addresses the issues spotted in the previous examples. It attempts to optimise the time needed for solving simple Nonograms by imitating the Logic Solver. The main rule is to set the position of one constraint block in a row/column at a time. For each row and column, the fitness function takes the first constraint and computes the minimum start and the maximum end-point for this black cell block. It then checks if the block is inside the derived range. To ensure that there is only one block of the required length, the white cells between the black cells are largely penalised. Once the first block is at the correct position, the fitness function moves to the next block and repeats the procedure until all the blocks are placed. For a constraint $r_{i,j}$, the fitness function computes the range using the following formula. For all $r_{p,j} \neq 0$:

$$start_{i,j} = min \left\{ \sum_{p=0}^{i-1} (r_{p,j} + 1), end_{i-1,j} \right\} \tag{4.18}$$

$$end_{i,j} = M - \sum_{p=i+1}^{M} (r_{p,j} + 1) \tag{4.19}$$

where $end_{i-1,j}$ is the end position of the previous black block. Once the start and the end points are identified, the cells inside this range are validated using the following formula:

$$p(X) = M - (\left| r_{i,j} - b_{start_{i,j},end_{i,j}} \right| + 2 * w_{start_{i,j},end_{i,j}}) \tag{4.20}$$

where $b_{start_{i,j},end_{i,j}}$ is the sum of black cells inside the range and $w_{start_{i,j},end_{i,j}}$ is the sum of all the while cells which are between the black cells inside the range. $p(X)$ rates the solutions much more accurately than $k(X)$ - see: Table 4.3. However, there is still space for improvement: an ideal fitness function should give an equal score to all these partial solutions.

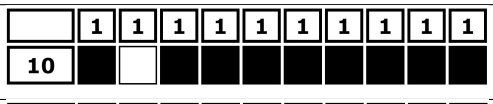
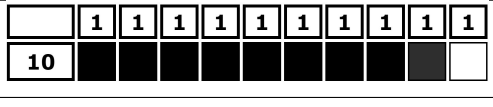
ID	Nonogram	$p(X)$
1		$10 - (\lvert 10 - 9 \rvert + 2) = 7$
2		$10 - (\lvert 10 - 9 \rvert + 2) = 7$
3		$10 - (\lvert 10 - 9 \rvert) = 9$

Table 4.3: $p(X)$ on different partial solutions

4.3.5 Comparison of the proposed fitness functions

The function $g(X)$ classifies solutions based on the total sum of black cells in each row and column. We have shown that this approach is incorrect, as it misclassifies some search points as an optimum. It is also inefficient - half of its computations are redundant and can be reduced to $g'(X)$. An algorithm that uses $g'(X)$ will be the fastest, but it will return the wrong solution quite often. $k(X)$ - fitness function designed in this report - looks at the amount and size of black cells blocks in each row and column. It correctly classifies the search points. However, the correct bits need to be added in the right order (from left to right). Therefore, the algorithm will probably take a long time to find the optimum, even for the simplest subclasses like fully filled Nonograms. Finally, $p(X)$ checks the correct position and size of one black cell block at a time for each row and column. It correctly rates the partial solutions and it may work faster on some subclasses than $k(X)$ because of the added knowledge to the function. The hypotheses regarding the runtime will be verified in the following chapter using the mathematical analysis.

Chapter 5

Runtime Analysis

In this chapter we derive the upper bounds on the runtime of the (1+1)EA and RLS with three different fitness functions on the fully filled, empty and single-line Nonograms. Since these algorithms work by first converting the Nonogram instance into a bit-string, the expected running times for horizontal and vertical single-line Nonongrams are the same. That is why we bound the runtime only for one of these subclasses - vertical single-line Nonograms.

5.1 Fitness function $g'(X)$

As described in section 4.3.1, $g(X)$ (and, thus, also $g'(X)$) checks only for the appropriate number of one-bits in each row and column that can be added to the solution in any order.

5.1.1 Fully filled Nonograms

The (1+1)Evolutionary Algorithm

Theorem 5. *The expected running time of the $(1+1)EA$ with the fitness function $g'(X)$ on any $N \times M$ fully filled Nonogram is $O(NM\ log(NM))$.*

Proof. Since a fitness function looks at both column and row constraints, the fitness value of a solution increases by 2 every time a zero-bit is flipped into a one-bit. Hence, the problem can be reduced to OneMax (Section 3.3.1) where fitness value increases by 1 with every zero-bit flipped into a one-bit. As proved in Lemma 1, the expected runtime of the (1+1)EA on OneMax with a bit-string of length NM is $O(NM\ log(NM))$. □

Randomized Local Search

Theorem 6. *The expected running time of RLS with the fitness function $g'(X)$ on any $N \times M$ fully-filled Nonogram is $O(NM\ log(NM))$.*

Proof. Let's divide the search space $\{0,1\}^{NM}$ into $NM+1$ disjoint sets A_0, A_1, ..., A_{NM} where A_i contains search points with i correct one-bits. A_{NM} contains the optimum with all

the one-bits. To reach the level of higher fitness it is necessary to flip one of the $NM - i$ zero-bits into a one-bit. RLS flips one bit in every step, leaving the other bits unchanged. Hence, for level A_i, the probability of reaching the higher level is bounded by s_i:

$$s_i \geq (NM - i) * \frac{1}{NM} = \frac{NM - i}{NM} \tag{5.1}$$

Then, by using Theorem 1 (Artificial Fitness Levels [14]):

$$\mathbb{E}[T_{RLS}] \leq \sum_{i=0}^{NM-1} \frac{NM}{NM - i} = NM \sum_{i=1}^{NM} \frac{1}{i} = O(NM\ log(NM)) \tag{5.2}$$

□

5.1.2 Empty Nonograms

The (1+1)Evolutionary Algorithm

Theorem 7. *The expected running time of the $(1 + 1)EA$ with the fitness function $g'(X)$ on any $N \times M$ empty Nonogram is $O(NM\ log(NM))$.*

Proof. The fitness of a solution increases by 2 with every one-bit flipped into a zero-bit. Therefore, similarly to the previous section, the problem can be reduced to OneMax on a bit-string of length NM and we can again apply Lemma 1. □

Randomized Local Search

Theorem 8. *The expected running time of RLS with the fitness function $g'(X)$ on any $N \times M$ empty Nonogram is $O(NM\ log(NM))$.*

Proof. Let's divide the search space $\{0,1\}^{NM}$ into $NM + 1$ disjoint sets A_0, A_1, ..., A_{NM} where A_i contains search points with i correct zero-bits. A_{NM} contains the optimum with all the zero-bits. To reach the level of higher fitness it is necessary to flip a one-bit into a zero-bit. Since, for level A_i, the probability of reaching the higher level is bounded by s_i:

$$s_i \geq (NM - i) * \frac{1}{NM} = \frac{NM - i}{NM} \tag{5.3}$$

Then, by using Theorem 1 (Artificial Fitness Levels [14]):

$$\mathbb{E}[T_{RLS}] \leq \sum_{i=0}^{NM-1} \frac{NM}{NM - i} = NM \sum_{i=1}^{NM} \frac{1}{i} = O(NM\ log(NM)) \tag{5.4}$$

□

5.1.3 Single-line Nonograms

When using AFL (Section 3.3.1), we cannot partition the search space based on the number of the correct bits in a bit-string, because solutions with different number of correct bits may

have the same fitness value (see: Table 5.1).

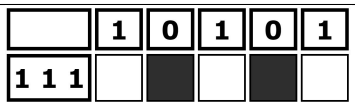
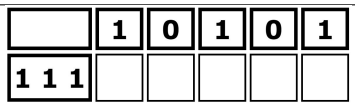
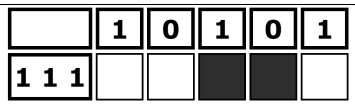
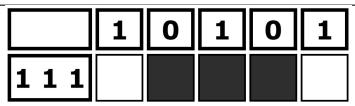
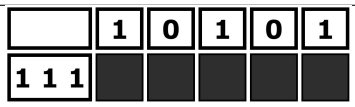
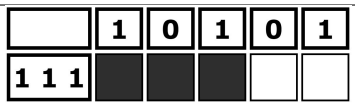
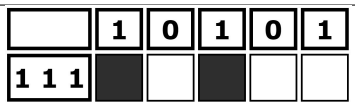
$A_i = g'(X)$	**Nonogram**	no. of wrong 1s	no. of wrong 0s
$A_0 = -1$		2	3
$A_0 = -1$		0	3
$A_1 = 1$		1	2
$A_1 = 1$		2	2
$A_1 = 1$		2	0
$A_2 = 3$		1	1
$A_2 = 3$		0	1

Table 5.1: $g'(X)$ on different partial solutions

One possible way of creating a fitness-based partition is to use the actual value of the fitness function which depends on the number of constraints violated. Let's divide the search space $\{0,1\}^{NM}$ into k disjoint sets A_1, A_1, ..., A_k where the last level contains all the solutions with an optimal fitness value M. Level A_1 contains all the solutions with the smallest fitness value $(-|M-2a|)$. Solutions at this fitness level violate all the top constraints. A total sum of black cells in the model solution is denoted as $a := \sum_{p=1}^{M} Xr_p$. Since these instances violate all the top constraints, they have $M-a$ black cells (and following the row description they should have had a). Hence, the total value of the fitness function is $g'(X) = M-M-|M-2a| = -|M-2a|$. The fitness function can improve by 2 at each level (each cell in an appropriate place may violate one less row and one less column description). Thus, by solving $-|M-2a|+2k = M$, we notice that a total number of levels in this fitness-based partition is linear in M:

$$k = M + \frac{|M-2a|}{2} = O(M) \tag{5.5}$$

In most cases, in order to move from level A_i to A_{i+1} it is necessary to swap one incorrect bit and leave the others unchanged. However, sometimes flipping a bit to fit the column

constraints, results in violating more row constraints - thus, the fitness value remains the same (see: Figure 5.1). Hence, it is necessary to both flip an appropriate zero-bit and an appropriate one-bit.

Now we need to compute the exact number of incorrect one and zero-bits at each level A_i. As shown in Figure 5.1, these numbers may vary on each level. It can be derived that for each level i, the amount of incorrect zero-bits is at most $k-i$. For levels where $k-i > M/2$, in order to improve, it is sufficient to just flip any zero-bit into a one-bit. For other levels, it may be necessary to flip a zero-bit into one and a one-bit into zero-bit. Finally, in order to move from level A_{k-1} to the optimum (A_k), in the worst case we must flip a one-bit and a zero-bit.

Lemma 4. *At any level A_i, the probability of improving to level A_{i+1} is greater or equal to the probability of flipping both one of the $k-i$ one-bits and one of the $k-i$ zero-bits.*

The (1+1)Evolutionary Algorithm

Theorem 9. *The expected running time of the (1+1)EA with the fitness function $g'(X)$ on any single-line Nonogram is $O(M^2\ log(M))$.*

Proof. Following Lemma 4, the probability s_i of moving from level A_i to level A_{i+1} is bounded by the probability of flipping one of the $k-i$ one-bits and one of the $k-i$ zero-bits in a single step while leaving the other bits unchanged:

$$s_i \geq \frac{(k-i)}{M} * \frac{(k-i)}{M} * (1-\frac{1}{M})^{M-2} \geq \frac{(k-i)^2}{eM^2} \tag{5.6}$$

Then, by using Theorem 1(Artificial Fitness Levels [14]):

$$\mathbb{E}[T_{(1+1)EA}] \ \leq \ \sum_{i=0}^{k-1} \frac{eM^2}{(k-i)^2} = eM^2 \sum_{i=1}^{k} \frac{1}{i^2} \ \leq \ eM^2 \sum_{i=1}^{k} \frac{1}{i} = O(M^2 log(k)) = O(M^2 log(M)) \tag{5.7}$$

where the last equality holds because $k = O(M)$ □

Randomized Local Search

Theorem 10. *The expected running time of RLS with the fitness function $g'(X)$ on any single-line Nonogram is $O(M^2\ log(M))$.*

Proof. As previously, using Lemma 4, we assume that in order to improve from level A_i to level A_{i+1}, RLS must flip one of the $k-i$ one-bits and one of the $k-i$ zero-bits. RLS mutates exactly one bit in a single step so it must flip an appropriate one-bit in one step, which leaves the fitness value unchanged, and a zero-bit in the consecutive step, which improves the

fitness of the solution. Therefore, the probability of moving from level A_i to level A_{i+1} can be bounded by s_i:

$$s_i \geq \frac{(k-i)}{M} * \frac{(k-i)}{M} = \frac{(k-i)^2}{M^2} \tag{5.8}$$

Then, by using Theorem 1 (Artificial Fitness Levels [14]):

$$\mathbb{E}[T_{RLS}] \leq \sum_{i=0}^{k-1} \frac{M^2}{(k-i)^2} = M^2 \sum_{i=1}^{k} \frac{1}{i^2} \leq M^2 \sum_{i=1}^{a} \frac{1}{i} = O(M^2\ log(k)) = O(M^2 log(M)) \tag{5.9}$$

where the last equality holds because $k = O(M)$

□

5.2 Fitness function $k(X)$

As described in section 4.3.3, $k(X)$ correctly identifies the optimal solutions. However, in order for the fitness function to improve, the next correct cells need to be added in the specific order: from left to right.

5.2.1 Fully filled Nonograms

The (1+1)Evolutionary Algorithm

Let's first consider a single-line fully filled $1 \times M$ Nonogram. To increase the fitness, we need to add one-bits in the specific order.

Lemma 5. *The expected running time of the $(1+1)EA$ with the fitness function $k(X)$ on any single-line fully-filled Nonogram is $O(M^2)$.*

Proof. We can divide the search space into into $M+1$ disjoint sets A_0, A_1, ..., A_M using different fitness function values. Level A_0 contains the empty Nonogram with fitness of $-M$ (all rows and columns constraints violated). Solutions at level A_1 satisfy 1 row and 1 column constraint - which increases the fitness by 2. Following this pattern, all the points in level A_i have fitness value of $2i - M$. The last level A_M contains only the optimal solution of fitness M.

In order to bound the running time, we need to compute the probability of leaving the level A_i to the level A_{i+1}. In order to move from level A_0 to level A_1, an algorithm needs to swap any of M zero-bits into a one-bit. However, to go to level A_2, it is not sufficient to just swap any of the remaining $M-1$ zero-bits. As shown in Table 5.2, this action may leave the fitness level unchanged. This happens when the algorithm swaps the zero-bit which is not adjacent to any one-bit and, thus, creates a new black cell block.

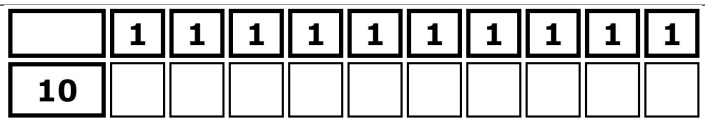
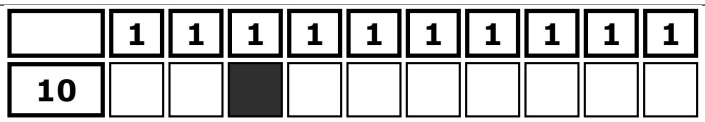
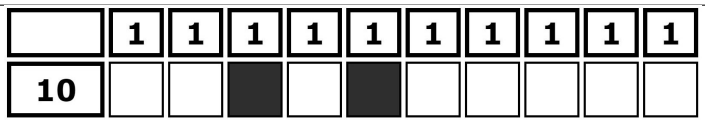
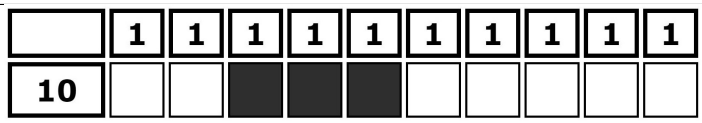
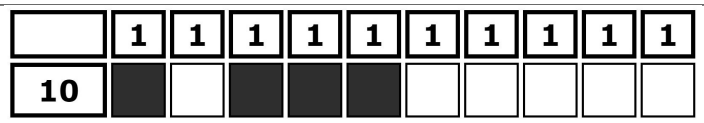
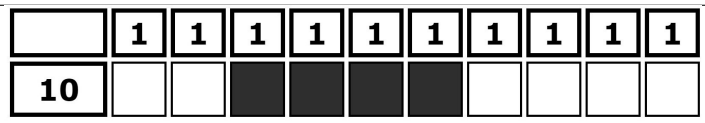
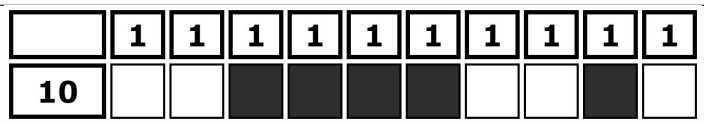
ID	Nonogram	$k(X)$
1		$10 - 10 - 10 * 1 = -10$
2		$10 - 9 - 9 * 1 = -8$
3		$10 - 9 - 1 - 8 * 1 = -8$
4		$10 - 7 - 7 * 1 = -4$
5		$10 - 9 - 3 - 6 * 1 = -8$
6		$10 - 6 - 6 * 1 = -2$
7		$10 - 6 - 1 - 5 * 1 = -2$

Table 5.2: $k(X)$ on different partial solutions

We can increase the fitness only by increasing the size of the current black cell block. Therefore, in order to move from level A_i to level A_{i+1}, we need to swap any of the zero-bits adjacent to the current black cell block. The number of these bits does not depend on the level but on the position of the existing one-bits. If it is in the middle, we can swap a bit on the left or a bit on the right. Otherwise, we can add one-bit only on one side. Since, for level A_i, the probability of reaching the higher level is bounded by s_i:

$$s_i \geq \frac{1}{M} * (1 - \frac{1}{M})^{M-1} \geq \frac{1}{eM} \tag{5.10}$$

Then, by using Theorem 1 (Artificial Fitness Levels [14]):

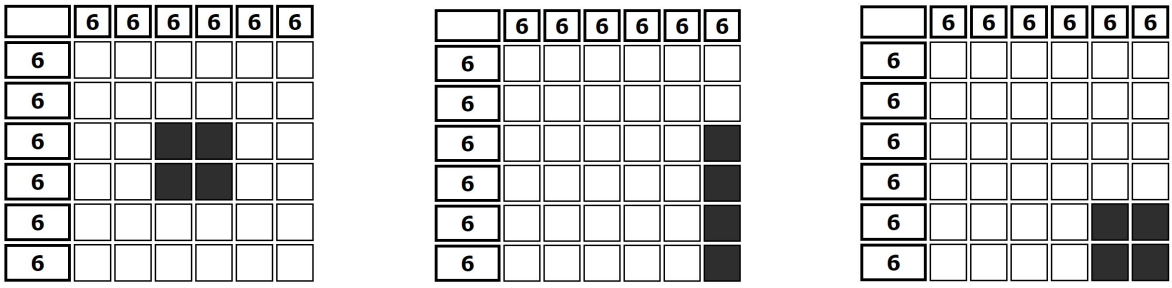
$$\mathbb{E}[T_{(1+1)EA}] \leq \sum_{i=0}^{M-1} \frac{eM}{1} = eM \sum_{i=1}^{M} 1 = eM^2 = O(M^2) \tag{5.11}$$

□

We can use the knowledge from the previous example to derive the bound on the running time of the (1+1)EA on any $N \times M$ fully filled Nonogram.

Theorem 11. *The expected running time of the* $(1+1)$*EA with the fitness function* $k(X)$ *on any* $N \times M$ *fully-filled Nonogram is* $O((NM)^2)$.

Proof. We can divide the search space into $MN + 1$ levels of different fitness values. At level A_0, all the rows and columns constraints are violated; thus, the fitness is equal to $-MN$. Level A_{MN} contains only the optimal solution with the fitness value of MN.



(a) 12 adjacent cells (b) 6 adjacent cells (c) 5 adjacent cells

Figure 5.1: Number of adjacent cells to the black-cell block of the same size

Depending on a position and size of the black cell block, we may have different number of adjacent cells to swap (see: Figure 5.1). In order to derive the upper bound on the runtime, we assume that is the worst case we have a constant number of bits to flip (b). Hence, for level A_i, the probability of reaching the higher level is bounded by s_i:

$$s_i \geq \frac{b}{NM} * (1 - \frac{1}{NM})^{NM-1} \geq \frac{b}{eNM} \tag{5.12}$$

Then, by using Theorem 1 (Artificial Fitness Levels [14]):

$$\mathbb{E}[T_{(1+1)EA}] \leq \sum_{i=0}^{NM-1} \frac{eNM}{b} = eNM \sum_{i=1}^{NM} \frac{1}{b} = \frac{e(NM)^2}{b} = O((NM)^2) \tag{5.13}$$

□

Randomized Local Search

Theorem 12. *The expected running time of RLS with the fitness function* $k(X)$ *on any* $N \times M$ *fully filled Nonogram is* $O((NM)^2)$.

Proof. Let's divide the search space into $MN + 1$ levels of different fitness values. At level A_0, all the rows and columns constraints are violated; thus, the fitness is equal to $-MN$. Level A_{MN} contains only the optimal solution with the fitness value of MN. Assuming that in the worst case we have a constant number of bits to flip (b), the probability of leaving the level A_i is bounded by s_i:

$$s_i \geq \frac{b}{NM} \tag{5.14}$$

Then, by using Theorem 1 (Artificial Fitness Levels [14]):

$$\mathbb{E}[T_{RLS}] \leq \sum_{i=0}^{NM-1} \frac{NM}{b} = NM \sum_{i=1}^{NM} \frac{1}{b} = \frac{(NM)^2}{b} = O((NM)^2) \tag{5.15}$$

□

5.2.2 Empty Nonograms

Lemma 6. *For Empty Nonograms, k(X) looks only at the total sum of all the one-bits in the solution; therefore, it behaves exactly like g'(X).*

Proof. Since the constraints matrices Xr and Xc are both zero-matrices, the fitness function can be rewritten as follows:

$$k_1(X) = \sum_{i=1}^{N}\sum_{p=1}^{M} |r_{i,p} - Xr_{i,p}| = \sum_{i=1}^{N}\sum_{p=1}^{M} Xr_{i,p} = \sum X \tag{5.16}$$

$$k_2(X) = \sum_{j=1}^{M}\sum_{p=1}^{N} |c_{j,p} - Xc_{j,p}| = \sum_{j=1}^{M}\sum_{p=1}^{N} Xc_{j,p} = \sum X \tag{5.17}$$

$$k'(X) = k_1(X) + k_2(X) = 2\sum X \tag{5.18}$$

$$k(X) = N * M - k'(X) = NM - 2\sum X \tag{5.19}$$

As shown in Equation 5.19, the fitness value decreases by 2 for every one-bit in a solution. We can swap one-bits to zero-bits in any order, because $k(X)$ (just like $g'(X)$) looks only at the total sum of all the one-bits in the solution. □

The (1+1)Evolutionary Algorithm

Theorem 13. *The expected running time of the $(1+1)EA$ with the fitness function $k(X)$ on any $N \times M$ empty Nonogram is $O(NM\ log(NM))$.*

Proof. See: Lemma 6 and Theorem 7. □

Randomized Local Search

Theorem 14. *The expected running time of RLS with the fitness function $k(X)$ on any $N \times M$ empty Nonogram is $O(NM\ log(NM))$.*

Proof. See: Lemma 6 and Theorem 8. □

5.2.3 Single-line Nonograms

When solving fully-filled Nonograms, we have to add one-bits in a sequence. In case of empty Nonograms - it is sufficient to remove one-bits without any order. However, when solving general case of single-line Nonogram, this may not be enough. Table 5.3 shows the examples where adding or removing a one-bit in one step does not increase the fitness value. In order to improve, particular one-bit need to be added and particular one-bit need to be removed.

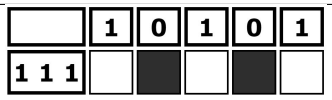
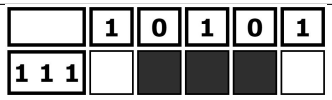
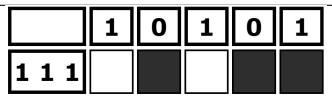
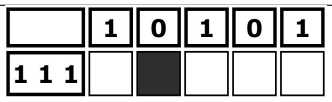
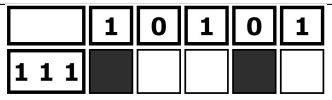
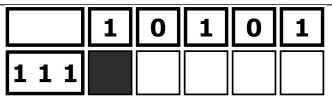
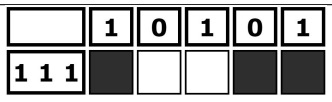
$k(X)$	**Nonogram**	**Example**
$k(X) = -1$		ex. 1
$k(X) = -3$		adding correct one-bit to ex.1
$k(X) = -1$		adding correct one-bit to ex.1
$k(X) = -1$		adding correct zero-bit to ex.1
$k(X) = 1$		ex. 2
$k(X) = 1$		adding correct zero-bit to ex.2
$k(X) = 1$		adding correct one-bit to ex.2

Table 5.3: $k(X)$ on a single-line Nonogram

The (1+1)Evolutionary Algorithm

Theorem 15. *The expected running time of the* $(1+1)EA$ *with the fitness function* $k(X)$ *on any* $1 \times M$ *Nonogram is* $O(M^3)$.

Proof. We pessimistically assume that at any step, in order to improve, a one-bit needs to be added $(1/M)$, a one-bit needs to be removed $(1/M)$, and all the other bits remain unchanged $((1-\frac{1}{M})^{M-2})$. Thus, the expected decrease in distance (Δ_t) is the same in all the areas of the search space:

$$\Delta_t \geq \frac{1}{M} * \frac{1}{M} * (1-\frac{1}{M})^{M-2} \geq \frac{1}{eM^2} := \sigma \tag{5.20}$$

Because of the constant drift, we can apply the Additive Drift Theorem (Theorem 2):

$$\mathbb{E}[T_{(1+1)EA}] \leq \frac{\mathbb{E}[d(X_0)]}{\sigma} \leq \frac{M}{1/eM^2} = O(M^3) \tag{5.21}$$

Where the expected distance is **at most** $\mathbb{E}[d(X_0)] = M$. □

Randomized Local Search

Theorem 16. *The expected running time of RLS with the fitness function $k(X)$ on any $1 \times M$ Nonogram is $O(M^3)$.*

Proof. Let's divide the search space into L disjoint levels such that every level A_i contains solutions with the same fitness value. The fitness function increases by at least 1 so the maximum number of levels $L \leq max(k(X)) - min(k(X))$, where $max(k(X))$ is M and $min(k(X))$ can be computed by bounding $k_2(X)$ and $k_1(X)$:

$$k_2(X) = \sum_{j=1}^{M} |c_{j,1} \ - \ Xc_{j,1}| \leq \sum_{j=1}^{M} 1 = M \tag{5.22}$$

As for every j, c_j and Xc_j can be either 0 or 1.

$$k_1(X) \ = \ \sum_{p=1}^{M} |r_{1,p} \ - \ Xr_{1,p}| \ \ = b * M \tag{5.23}$$

Since the total number of black cells cannot be greater than M, a number of constraints or a size of each of the constraint should be constant.

$$k'(X) = k_1(X) + k_2(X) = (b+1) * M \tag{5.24}$$

$$k(X) = N * M - k'(X) = -b * M \tag{5.25}$$

where $b \in \mathbb{R}$. Hence, the number of levels (L) is linear:

$$L \leq max(k(X)) - min(k(X)) = (b+1)M = O(M) \tag{5.26}$$

In order to improve, the particular one-bit need to be added in one step and the particular one-bit need to be removed in the consecutive step. The probability of this event can be bounded as:

$$s_i \geq \frac{1}{M} * \frac{1}{M} = \frac{1}{M^2} \tag{5.27}$$

Then, by using Theorem 1 (Artificial Fitness Levels [14]):

$$\mathbb{E}[T_{RLS}] \leq \sum_{i=0}^{L-1} \frac{M^2}{1} = M^2 \sum_{i=1}^{(b+1)M} 1 = M^2 * (b+1)M = O(M^3) \tag{5.28}$$

□

5.3 Fitness function $p(X)$

5.3.1 Fully filled Nonograms

The function $p(X)$ decreases the fitness value by one for any black cell missing in the solution and by two for every white cell between the blocks of black cells (See: Table 5.4). Thus, it can be derived that, similarly to $k(X)$, one-bits need to be added in a particular order.

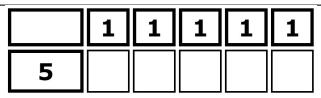
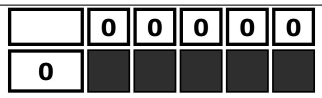
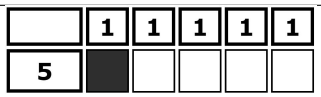
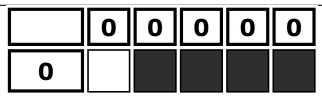
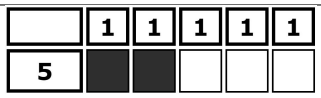
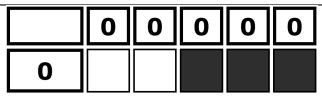
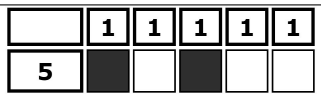
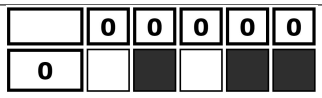
$p(X)$	**Filled Nonogram**	$p(X)$	**Empty Nonogram**
-5		-5	
-3		-3	
-1		-1	
-3		-3	

Table 5.4: $p(X)$ on Empty and Fully-Filled Nonograms

Randomized Local Search

Theorem 17. *The expected running time of RLS with the fitness function $p(X)$ on any $N \times M$ fully-filled Nonogram is ∞.*

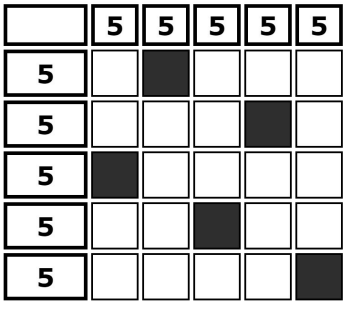


Figure 5.2: Solution where RLS is stuck

Proof. Because the penalty for every white cell between black cells is two times greater than the reward for increasing the number of black cells, there exist partial solutions where it is impossible to insert or remove a one-bit without decreasing the fitness value (see: Figure 5.2).

In these cases RLS is stuck forever, because it can only mutate a single bit in one step. Let A be en event where RLS gets stuck forever. Since we have shown that for any $N \times M$ fully-filled Nonogram, there exists at least one such instance, Figure 5.2, $p(A) > 0$. To compute the total expected time, we can use the conditional expectation:

$$\mathbb{E}[T_{RLS}] = \mathbb{E}[T|A] * p(A) + \mathbb{E}[T|\neg A] * p(\neg A) \geq \mathbb{E}[T|A] * p(A) = \infty * p(A) = \infty \tag{5.29}$$

□

The (1+1)Evolutionary Algorithm

Lemma 7. *The expected running time of the* $(1+1)EA$ *with the fitness function* $p(X)$ *on any* $N \times M$ *fully filled Nonogram is* $O((NM)^{NM})$.

Proof. The (1+1)EA will never get stuck, because regardless of the solution, there is always a positive probability ($p \geq 1/NM^{NM}$) that the algorithm will flip all the incorrect bits in one step. Therefore, by the waiting time argument:

$$\mathbb{E}[T_{(1+1)EA}] \leq \frac{1}{p} = O(NM^{NM}) \tag{5.30}$$

However, in many cases where RLS is stuck, the (1+1)EA can improve by flipping a constant number of bits b in one step with probability $p_i \geq 1/(NM)^b$. For example, it can improve the solution in Figure 5.2 by flipping two bits. Therefore, usually the exponential upper bound on the runtime ($O(NM^{NM})$) is not tight. □

Lemma 8. *The expected running time of the* $(1+1)EA$ *with the fitness function* $p(X)$ *on any single-line* $1 \times M$ *fully filled Nonogram is* $O((M)^2)$.

Proof. To increase the fitness, we need to increase the size of the current black cell block by swapping any of the zero-bits adjacent to the current black cell block. The number of these bits depends on the position of the existing one-bits. If it is in the middle, we can add a one-bit to the left or a one-bit to the right. Otherwise, we can add one-bit only on one side. Thus, the expected decrease in distance in one step (Δ_t) is constant in all the areas of the search space:

$$\Delta_t \geq \frac{1}{M} * (1 - \frac{1}{M})^{M-1} \geq \frac{1}{eM} := \sigma \tag{5.31}$$

By applying the Additive Drift Analysis for Upper Bounds (2):

$$\mathbb{E}[T_{(1+1)EA}] \leq \frac{\mathbb{E}[d(X_0)]}{\sigma} \leq \frac{M}{1/eM} = \frac{eM^2}{1} = O(M^2) \tag{5.32}$$

Where the expected distance is at most $\mathbb{E}[d(X_0)] = M$.

□

5.3.2 Empty Nonograms

Since $p(X)$ not only penalises the cells of the wrong colour but also the white cells between the black cells, the one-bits need to be removed in the particular order (see: Table 5.4). To increase the fitness, we need to decrease the size of the existing black cell blocks by removing the most top-left, top-right, bottom-left or bottom-right one-bits. This ensures that no penalties for the while cells between the black blocks are added. Therefore, at any step, there is at least one one-bit which can be flipped in order for the fitness function to increase.

The (1+1)Evolutionary Algorithm

Theorem 18. *The expected running time of the $(1+1)EA$ with the fitness function $p(X)$ on any $N \times M$ empty Nonogram is $O((NM)^2)$.*

Proof. The initial distance to the optimum ($\mathbb{E}[d(X_0)]$) is at most NM - when all the cells are black. Assuming that at every step we can flip at least one black cell to improve, the drift Δ_t and the expected runtime $\mathbb{E}[T]$ can be bounded as:

$$\Delta_t \geq \frac{1}{NM} * (1 - \frac{1}{NM})^{NM-1} \geq \frac{1}{eNM} := \sigma \tag{5.33}$$

$$\mathbb{E}[T_{(1+1)EA}] \leq \frac{\mathbb{E}[d(X_0)]}{\sigma} \leq \frac{NM}{1/eNM} = \frac{e(NM)^2}{1} = O((NM)^2) \tag{5.34}$$

□

Randomized Local Search

Theorem 19. *The expected running time of RLS with the fitness function $p(X)$ on any $N \times M$ empty Nonogram is $O((NM)^2)$.*

Proof. Let's divide the search space into $MN+1$ levels of different fitness values. At level A_0, all the rows and columns constraints are violated; thus, the fitness is equal to $-MN$. Level A_{MN} contains only the optimal solution with the fitness value of MN. Assuming that in the worst case we need to flip exactly one one-bit to improve, the probability of leaving the level A_i is bounded by s_i:

$$s_i \geq \frac{1}{NM} \tag{5.35}$$

Then, by using Theorem 1 (Artificial Fitness Levels [14]) we derive the same bound as in the case of the (1+1)EA:

$$\mathbb{E}[T_{RLS}] \leq \sum_{i=0}^{NM-1} \frac{NM}{1} = NM \sum_{i=1}^{NM} 1 = O((NM)^2) \tag{5.36}$$

□

5.3.3 Single-line Nonograms

In case of a single-line Nonogram, in order to improve, we often need to add and remove one-bits in a single step. Hence, the upper bound on the running time for $p(X)$ is the same as for $k(X)$.

The (1+1)Evolutionary Algorithm

Theorem 20. *The expected running time of the* $(1+1)EA$ *with the fitness function* $p(X)$ *on any* $1 \times M$ *Nonogram is* $O(M^3)$.

Proof. We assume that at any step, in order to improve, a one-bit need to be added $(1/M)$, a one-bit need to be removed $(1/M)$, and all the other bits remain unchanged $((1-\frac{1}{M})^{M-2})$. Hence, the expected decrease in distance (Δ_t) can be bounded by:

$$\Delta_t \geq \frac{1}{M} * \frac{1}{M} * (1-\frac{1}{M})^{M-2} \geq \frac{1}{eM^2} := \sigma \tag{5.37}$$

Applying the Additive Drift Theorem (Theorem 2):

$$\mathbb{E}[T_{(1+1)EA}] \leq \frac{\mathbb{E}[d(X_0)]}{\sigma} \leq \frac{M}{1/eM^2} = O(M^3) \tag{5.38}$$

□

However, on the contrary to $k(X)$, this bound is usually not tight - it depends on whether it is possible to deduce the value of all the cells based only on the row constraints.

Definition 1. Single-line Nonogram is referred in this report as Deducible if it is possible to deduce the value of all the cells based only on the row constraints.

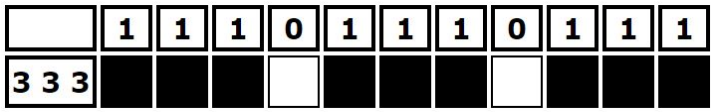


Figure 5.3: Deducible 1×11 Nonogram

Theorem 21. *The expected running time of the* $(1+1)EA$ *with the fitness function* $p(X)$ *on any single-line deducible* $1 \times M$ *Nonogram is* $O(M^2)$.

Proof. For any deducible single-line Nonogram with r constraints of size at most b, the (1+1)EA correctly identifies the start and the end position of every black cell block. Time needed for swapping all the zero-bits into one-bits for a constraint of size b is equivalent to the time the (1+1)EA needs for solving a fully-filled $1 \times b$ Nonogram where each bit is flipped with probability $1/M$:

$$\Delta_t \geq \frac{1}{M} * (1 - \frac{1}{M})^{M-1} \geq \frac{1}{eM} := \sigma \tag{5.39}$$

$$\mathbb{E}[T_{(1+1)EA}] \leq \frac{\mathbb{E}[d(X_0)]}{\sigma} \leq \frac{b}{1/eM} = ebM = O(bM) \tag{5.40}$$

Where the expected distance is at most $\mathbb{E}[d(X_0)] = b$. Therefore, the total running time can be bounded as $O(rbM)$. For constant b and $r := O(M)$, the running time is $O(M^2)$. If $b := M$ and $r := 1$ (fully filled), we obtain $O(M^2)$, which is the bound we proved in Section 5.3.1. □

Randomized Local Search

Theorem 22. *The expected running time of RLS with the fitness function $k(X)$ on any $1 \times M$ Nonogram is $O(M^3)$.*

Proof. We assume that RLS improves in 2 steps, flipping a one-bit $(1/M)$, which leaves the fitness value unchanged, and then flipping a zero-bit $(1/M)$. Hence the expected decrease in distance in 2 steps can be bounded by:

$$\Delta_{2t} \geq \frac{1}{M} * \frac{1}{M} = \frac{1}{M^2} \tag{5.41}$$

To get the expected decrease in distance in a single step (Δ_t), we need to divide Δ_{2t} by 2:

$$\Delta_t = \frac{\Delta_{2t}}{2} \geq \frac{1}{2M^2} := \sigma \tag{5.42}$$

Applying the Additive Drift Theorem (Theorem 2):

$$\mathbb{E}[T_{RLS}] \leq \frac{\mathbb{E}[d(X_0)]}{\sigma} \leq \frac{M}{1/2M^2} = O(M^3) \tag{5.43}$$

□

Chapter 6

Conclusions and Discussion

The main goal of this project was to assess the performance of Evolutionary Algorithms on the Nonogram problem and compare it with the problem-specific methods. First, we explored and summarized the most well-known algorithms used for solving Nonograms. We designed three fitness functions for the (1+1)EA and RLS, and compared their correctness. Finally, we conducted mathematical analysis to derive the upper bounds on the expected running time of for the proposed functions. All the goals stated in Chapter 3 were achieved.

6.1 Main Findings

6.1.1 Runtime Analysis

We analysed the running time of the (1+1)EA and RLS with $g'(X)$, $k(X)$ and $p(X)$ on three different Nonogram subclasses. As predicted, using $g'(X)$ results in the best runtime on any of the analysed subclasses. That shows how important it is to analyse not only the time performance but also the correctness of the fitness function.

Nonogram	$\boldsymbol{g'(X)}$	$\boldsymbol{k(X)}$	$\boldsymbol{p(X)}$
Empty	$O(NM \ \ log(NM))$	$O(NM \ \ log(NM))$	$O((NM)^2)$
Fully Filled	$O(NM \ \ log(NM))$	$O((NM)^2)$	∞
Single-Line	$O(M^2 \ \ log(M))$	$O(M^3)$	$O(M^3)$

Table 6.1: Randomised Local Search

$k(X)$ is strictly slower than $g'(X)$ on fully filled and single-line Nonograms because in order for the fitness value to increase, the bits need to be swapped in the correct order. Surprisingly, this does not hold for empty Nonograms: in this case the order does not matter; therefore, the running time of $k(X)$ is the same as $g'(X)$.

Since $p(X)$ gives penalty not only for the incorrect number of black cells, but also for any white cells that break the black cell blocks, one or more one-bits in the solutions need to be added and/or removed in the specific order regardless of the Nonogram instance. Therefore,

Nonogram	$g'(X)$	$k(X)$	$p(X)$
Empty	$O(NM\ \ log(NM))$	$O(NM\ \ log(NM))$	$O((NM)^2)$
Fully Filled	$O(NM\ \ log(NM))$	$O((NM)^2)$	$O((NM)^{NM})$
Single-Line	$O(M^2\ \ log(M))$	$O(M^3)$	$O(M^3)$

Table 6.2: The (1+1)Evolutionary Algorithm

the upper bound on the expected runtime greatly varies depending on the subclass. It is quadratic for empty Nonograms, cubic for single-line Nonograms and exponential or infinite for fully filled Nonograms.

6.1.2 Fitness function choice

Even though $g'(X)$ has the smallest upper bounds derived, it misclassifies the sub-optimal solutions as optimal; thus, it is not an appropriate choice for the fitness function. $k(X)$ correctly identifies the optimal solutions, but it often misclassifies the close-to-optimum solutions as very bad and, therefore, it takes longer time to find the correct solution. $p(X)$ works well on the single-line Nonograms achieving even the quadratic upper bound on the running time on some instances. However, because of the added penalties for white cells, in order to improve, more than one bit must be flipped in one step. Hence, RLS with $p(X)$ may get stuck and never find the correct solution.

That is why, based on the analysis results, the most appropriate fitness function for RLS is $k(X)$. When it comes to the (1+1)EA, the best choice is unclear. Based on the analysis, $k(X)$ has the same or strictly smaller upper bound on any of the subclasses than $p(X)$. However, because of the added knowledge, $p(X)$ may perform better on some instances than $k(X)$ in practice. To clearly state which fitness function is better for the (1+1)EA, the lower bounds on the runtime need to be derived.

Nonogram	DFS	SIMPLE - SOLVER	RLS	the (1+1)EA
Empty	$O(N * 2^M)$	$O(N * M^3)$	$O(NMlog(NM))$	$O(NMlog(NM))$
Fully Filled	$O(N * 2^M)$	$O(N * M^3)$	$O((NM)^2)$	$O((NM)^2)$
Single-Line (H)	$O(2^M)$	$O(M^4)$	$O(M^3)$	$O(M^3)$
Single-Line (V)	$O(M)$	$O(M)$	$O(M^3)$	$O(M^3)$

Table 6.3: A comparison of the Nonogram solving techniques

6.1.3 Performance of the Evolutionary Algorithms

RLS and the (1+1)EA are faster than DFS on all of the analysed subclasses except the vertical single-line Nonograms. Assuming that $M = \Theta(N)$, bio-inspired techniques have the same performance as the SIMPLE-SOLVER on empty and fully filled instances ($O((NM)^2) =$

$\Theta(O(N * M^3)))$. In case of single-line Nonograms - SIMPLE SOLVER solves the vertical instances in linear time, but it needs $O(M^4)$ time to find an optimum for any horizontal Nonogram. RLS and the (1+1)EA solve any single-line Nonogram in $O(M^3)$ time. The project's results prove that the upper bounds on the expected runtime of even the simplest bio-inspired algorithms are equivalent to or strictly smaller than the upper bounds on the runtime of the problem-specific algorithms for most of the analysed subclasses. That allows us to predict that since the (1+1)EA and RLS work better on simple instances, they may also perform better on the more complicated Nonograms than the problem-specific techniques.

6.2 Further Research Questions

Due to time constraints, this research project could not cover everything related to the runtime analysis of Evolutionary Algorithms for the Nonogram problem. We hope that the results achieved in this project provide the solid basis for further analysis of this topic. Below we provide the research questions that can be examined in the future:

1. Derive the lower bounds on the performance of RLS and the (1+1)EA,
2. Compare experimentally the performance of $g'(X)$, $k(X)$ and $p(X)$,
3. Check if there exist subclasses of Nonograms where Evolutionary Algorithms succeed while deterministic approaches fail,
4. Find more subclasses of Nonograms where RLS or the (1+1)EA fail,
5. Find the worst-case approximation ratio of RLS and the (1+1)EA for the Nonogram problem,
6. Verify whether the hypothesis that Evolutionary Algorithms in general perform better on Nonograms with smaller number of constraints holds.

Bibliography

[1] Alkhraisat, H., and Rashaideh, H. Dynamic inertia weight particle swarm optimization for solving nonogram puzzles. *International Journal of Advanced Computer Science and Applications (IJACSA) 7*, 10 (2016), 277–280.

[2] Aspnes, J. Approximation Algorithms, 2014.

[3] Batenburg, K., and Kosters, W. Solving Nonograms by combining relaxations. *Pattern Recognition 42* (08 2009), 1672–1683.

[4] Batenburg, K. J., and Kosters, W. A. On the difficulty of Nonograms. *ICGA Journal 35*, 4 (2012), 195–205.

[5] Berend, D., Pomeranz, D., Rabani, R., and Raziel, B. Nonograms: Combinatorial questions and algorithms. *Discrete Applied Mathematics 169* (2014), 30–42.

[6] Bogdanov, A. Lecture 2: P vs NP, hierarchy theorems. In *Computational Complexity*. ITCS, Tsinghua University, 2007. Online Course.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[8] Darwish, A. Bio-inspired computing: Algorithms review, deep analysis, and the scope of applications. *Future Computing and Informatics Journal 3*, 2 (2018), 231 – 246.

[9] Doerr, B., and Neumann, F., Eds. *Drift Analysis*. Springer International Publishing, Cham, 2020, pp. 89–131.

[10] Droste, S., Jansen, T., and Wegener, I. On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science 276*, 1-2 (2002), 51–81.

[11] Friedrich, T., he, J., Hebbinghaus, N., Neumann, F., and Witt, C. On improving approximate solutions by evolutionary algorithms. In *2007 IEEE Congress on Evolutionary Computation* (10 2007), pp. 2614 – 2621.

[12] Jansen, T., Oliveto, P. S., and Zarges, C. Approximating vertex cover using edge-based representations. In *Proceedings of the Twelfth Workshop on Foundations of Genetic Algorithms XII* (2013), FOGA XII '13, Association for Computing Machinery, p. 87–96.

[13] Kleinberg, J., and Tardos, E. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[14] Lehre, P. K., and Oliveto, P. S. Theoretical Analysis of Stochastic Search Algorithms. *CoRR abs/1709.00890* (2017).

[15] Mallawaarachchi, V. How to define a Fitness Function in a Genetic Algorithm?, 2017. Online.

[16] Neumann, F., and Wegener, I. Randomized local search, evolutionary algorithms, and the minimum spanning tree problem. *Theoretical Computer Science 378*, 1 (2007), 32 – 40.

[17] Neumann, F., and Witt, C. *Bioinspired computation in combinatorial optimization - Algorithms and their computational complexity*. Springer, 2010.

[18] Neumann, F., and Witt, C. On the runtime of randomized local search and simple evolutionary algorithms for Dynamic Makespan Scheduling. *CoRR* (2015).

[19] Oliveto, P. Lectures 6 - 10. In *COM3105 Advanced Algorithms*. University of Sheffield, 2020. Course.

[20] Ortiz-García, E. G., Salcedo-Sanz, S., Pérez-Bellido, Á. M., Carro-Calvo, L., Portilla-Figueras, A., and Yao, X. Improving the performance of evolutionary algorithms in grid-based puzzles resolution. *Evolutionary Intelligence 2*, 4 (2009), 169.

[21] Raskhodnikova, S. Lecture 1: Introduction. The Set Cover Problem: linear programming, deterministic rounding. In *CSE 597: Approximation Algorithms*. College of Engineering, PennState, 2017. Online Course.

[22] Salcedo-Sanz, S., Portilla-Figueras, J. A., Ortíz-García, E. G., Pérez-Bellido, Á. M., and Yao, X. Teaching advanced features of evolutionary algorithms using Japanese puzzles. *IEEE Transactions on Education 50*, 2 (2007), 151–156.

[23] Sudhoff, S. Lecture 2: Canonical Genetic Algorithms. In *ECE 630 Special Topics: Engineering Analysis and Design Using Genetic Algorithms*. College of Engineering, Purdue University, 2007. Online Course.

[24] Sudholt, D., and Oliveto, P. Lecture 2: Runtime and Asymptotic Notation. In *COM1009 Introduction to Algorithms and Data Structures*. University of Sheffield, 2019. Course.

[25] Evolutionary Algorithms: Genetic Algorithms, 2020. Online.

[26] Genetic algorithms and evolutionary algorithms – introduction, 2020. Online.

[27] Trevisan, L. Lecture 2: P vs NP, deterministic hierarchy theorem. In *CS 254 – Computational Complexity*. Cs, Stanford University, 2010. Online Course.

[28] Ueda, N., and Nagao, T. NP-completeness results for NONOGRAM via Parsimonious Reductions, 1996.

[29] Walker, D. Lecture 2: Evolutionary Algorithms. In *COM3524 Bio-inspired Computing*. University of Sheffield, 2020. Course.

[30] Wiggers, W., and van Bergen, W. A comparison of a genetic algorithm and a depth first search algorithm applied to Japanese nonograms. In *Twente student conference on IT* (2004).

[31] Yu, C.-H., Lee, H.-L., and Chen, L.-H. An efficient algorithm for solving nonograms. *Applied Intelligence 35*, 1 (2011), 18–31.

Appendices

Appendix A

Detailed Computations

A.1 Computing the running time of SIMPLE-SOLVER

To compute the running time of SIMPLE-SOLVER, we need to know the time needed to perform one SWEEP operation and the maximum number of times this operation need to be performed in order to solve a simple Nonogram.

Assuming working on a bit-string of length l, the SETTLE operation applies Fix operation to each cell in a string which is not yet fixed - at most $O(l)$ times. The complexity of the computation of Fix is $O(k * l^2)$, where k is a number of constraints for a given bit-string. We may assume that $k \leq \lceil l/2 \rceil$. Hence the total runtime of SETTLE operation on the string of length l is $O(l^4)$ [3, 4].

H-SWEEP applies the SETTLE operation to all rows of the Nonogram - the running time is $O(N * M^4)$. V-SWEEP applies the SETTLE operation to all columns of the Nonogram - the running time is $O(M * N^4)$.

The maximum number of alternating H-SWEEP and V-SWEEP operations is $M * N + 1$. Hence the total running time of SIMPLE SOLVER can be expressed as
$O(((M * N + 1)/2) * N * M^4 + ((M * N + 1)/2) * M * N^4) = O(N^2 * M^5 + N^5 * M^2)$.