UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUSTAVO DELAZERI

Generating Heuristics for the Quadratic Unconstrained Binary Optimization Problem using Automatic Algorithm Configuration

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dr. Marcus Ritt

Porto Alegre
April 2023

ABSTRACT

In this work, we study an automated approach to the design of heuristic algorithms to solve the Quadratic Unconstrained Binary Optimization Problem (QUBO). Our approach is an extension of the work of Souza and Ritt (2018), who represent a design space of algorithms by a context-free grammar which is later explored by an algorithm configurator. We extend this work by introducing a new platform for design space exploration of heuristics for QUBO with a modular architecture and new components. Using this platform and the grammar-based approach, we were able to find algorithms that are very competitive and sometimes better than the state of the art in commonly used instances in the literature.

Keywords: Operations Research. Binary Quadratic Optimization. Automated Algorithm Design. Automatic Algorithm Configuration.

Gerando Heurísticas para o Problema de Otimização Binária Quadrática Irrestrita usando Configuração Automática de Algoritmos

RESUMO

Neste trabalho, estudamos uma abordagem automatizada para o projeto de algoritmos heurísticos para resolver o Problema de Otimização Binária Quadrática Irrestrita (QUBO). Nossa abordagem é uma extensão do trabalho de Souza and Ritt (2018), que representam um espaço de design de algoritmos por uma gramática livre de contexto que posteriormente é explorada por um configurador de algoritmos. Estendemos este trabalho introduzindo uma nova plataforma para exploração do espaço de design de heurísticas para QUBO com uma arquitetura modular e novos componentes. Usando essa plataforma e a abordagem baseada em gramática, conseguimos encontrar algoritmos muito competitivos e às vezes melhores do que o estado da arte em instâncias comumente usadas na literatura.

Palavras-chave: Pesquisa Operacional. Otimização Quadrática Binária. Design Automático de Algoritmos. Metaheuristicas. Configuração Automática de Algoritmos.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AC	Algorithm Configurator
GLS	Generalized Local Search
GRASP	Greedy Randomized Adaptive Search Procedure
IG	Iterated Greedy Algorithm
ILS	Iterated Local Search
IR	Iterated Racing
MWSS	Maximum Weight Stable Set
QUBO	Quadratic Unconstrained Binary Optimization
VNS	Variable Neighborhood Search

CONTENTS

1 INTRODUCTION

Given a matrix $Q \in \mathbb{R}^{n \times n}$, the Quadratic Unconstrained Binary Optimization Problem (QUBO) asks to find a binary vector $\mathbf{x} \in \{0,1\}^n$ that maximizes the quadratic form $\mathbf{x}^\intercal Q \mathbf{x}$. Several works in the literature use QUBO as a modeling framework. Rosenberg (2016), for example, shows how QUBO can be used to find an optimal arbitrage opportunity given a set of financial assets and the conversion rate between them. Milne, Rounds and Goddard (2017) describe how the problem of identifying a subset of independent and high-informative features in a credit scoring data set can be modeled as a QUBO. Neukart et al. (2017) use QUBO to optimize traffic flow: given a set of possible routes each car can take, they try to find assignments of cars to routes that minimize the overall congestion. A similar approach is used by Ohzeki et al. (2019), but their goal is to avoid collisions between automated vehicles transporting materials in manufacturing facilities. Punnen (2022) describes several other examples of applications.

It is known that QUBO is NP-Hard, and several polynomial-time reductions from NP problems to QUBO can be found in the literature (PUNNEN, 2022). We give as an example a reduction from the *maximum weight stable set problem* (MWSS). Given a graph $G = (V, E)$[1] and a function $w : V \to \mathbb{R}$ that assigns a weight to each vertex $v \in V$, MWSS asks to find a stable set[2] $S \subseteq V$ with maximum sum of weights. The sum of weights of a set S is defined as

$$s(S) = \sum_{v_i \in S} w(v_i)$$

Every subset $S \subseteq V$ can be encoded as a binary vector $\mathbf{x} = [x_1, \ldots, x_n]^\intercal$ such that $x_i = 1$ if, and only if, $v_i \in S$. It is easy to see that, for every stable set $S \subseteq V$ and its corresponding binary vector $\mathbf{x}$, $s(S) = \sum_{v_i \in V} w(v_i) x_i$ always holds. Maximizing $f(\mathbf{x}) = \sum_{v_i \in V} w(v_i) x_i$, however, is not enough, because not every set is a stable set. To account for that, we include a penalty term $M \sum_{(v_i, v_j) \in E} x_i x_j$. If M is big enough, every vector $\mathbf{x}$ representing a stable set will be better than one that does

[1] We assume an implicit enumeration of the vertices in V, .i.e., $V = \{v_1, \cdots, v_n\}$.

[2] A stable set of a graph $G = (V, E)$ is a subset $S \subseteq V$ such that no two vertices $v_1, v_2 \in S$ are connected by an edge.

not represent a stable set. Hence, to solve MWSS it suffices to maximize

$$f(\mathbf{x}) = \sum_{v_i \in V} w(v_i)x_i - M \sum_{(v_i,v_j)\in E} x_i x_j$$

It is possible to create a matrix $Q \in \mathbb{R}^{|V|\times|V|}$ where each diagonal entry Q_{ii} holds the value of $w(v_i)$ and, for each edge $(v_i, v_j) \in E$, we set entry Q_{ij} to M. Other entries are set to 0. One can check that $f(\mathbf{x}) = \mathbf{x}^\intercal Q\mathbf{x}$ for every binary vector $\mathbf{x}$. Therefore, f is an instance of QUBO.

As is common in NP-Hard problems, exact methods are useful to obtain optimal solutions for small instances, but when dealing with large instances, heuristic approaches are often the most effective option. The literature of heuristic procedures for QUBO is vast and covers most of the standard approaches, including local search (BOROS; HAMMER; TAVARES, 2007), Tabu Search (WANG et al., 2011), Simulated Annealing (KATAYAMA; NARIHISA, 2001), and Genetic Algorithm (LODI; ALLEMAND; LIEBLING, 1999). As a matter of fact, if we consider all the works cited in Kochenberger et al. (2014), Dunning, Gupta and Silberholz (2018), and Punnen (2022) more than 40 distinct heuristic procedures can be identified.

The design of heuristic algorithms is often laborious and based on a trial-and-error approach. The main steps of the design cycle usually consist in implementing a heuristic method and evaluating it experimentally. Based on the obtained results, new components are included or removed from the heuristic, and a new evaluation is performed. The experimental evaluation normally takes a considerable amount of time, which inhibits a wider exploration of the design space. As a consequence, good combinations of components may be overlooked or not even tested. Besides some good practices established by the optimization research community (KENDALL et al., 2016)(JOHNSON, 2002), there is no guiding theory on how to develop good heuristic algorithms, and the result of the development process often depends a lot on the experience of the designer.

The reasons mentioned above motivate the study of automated approaches to the design space of heuristics. These approaches usually consist of the specification of a design space of algorithms and of a set of training instances. Some strategy is then employed to explore the design space and find an algorithm that performs well on the training instances. The standard approach in the literature is to represent the design space by a context-free grammar, where different algorithms can be instantiated by

the repeated application of the grammar rules. There is no standard approach, however, for exploring the design space. Some works encode all the rules of the grammar as parameters and then use automatic algorithm configuration methods to search for good algorithms (MASCIA et al., 2014). Other works treat algorithms as valid expressions in the language encoded by the grammar and define a set of rules on how expressions can be combined to form new valid expressions. These operators are then used by a search procedure (e.g., a genetic algorithm) to find good algorithms (CASEAU; SILVERSTEIN; LABURTHE, 2004). Other approaches exist, but a complete account is out of scope.

In this thesis, we contribute to the literature of automated algorithm design by extending the work of Souza and Ritt (2018), who propose a context-free grammar that encodes a diverse design space of algorithms for QUBO, which is later explored by an algorithm configurator. More specifically, this thesis presents the following contributions:

- A modular and easy-to-extend platform for design space exploration of heuristics to QUBO built on top of the code base of Souza and Ritt (2018);
- The implementation of new components in the platform mentioned above;
- The definition of a new context-free grammar that includes these new components;
- New automatically-generated algorithms that are very competitive with the state of the art.

This thesis is the third in a series of works in the fields of heuristic design and automatic algorithm configuration. In the first work, we proposed a heuristic algorithm for a variation of the Traveling Salesperson problem where a truck and a set of drones must be coordinated to deliver a set of parcels to clients as fast as possible (DELAZERI; RITT, 2021). Throughout the project, we experienced all the difficulties mentioned above regarding the design process. In a second work, we studied surrogate models for tuning optimization algorithms (DELAZERI; RITT; SOUZA, 2023). This project introduced us to the field of automatic algorithm configuration, and we had the opportunity to learn some of its main methods and tools. All of this brings us to this work, where we use our experience developing heuristics and working with automatic algorithm configuration to explore a topic at the intersection of both fields.

We close this section by giving an outline of what follows. In Section 2 we introduce some notation and review the literature on heuristic algorithms for QUBO. We also cover some notions and tools of automatic algorithm configuration and review some works in the field of automated design of algorithms. Section 3 introduces the context-free grammar that describes our design space, and gives details about each of the implemented algorithmic components. In Section 4 we describe the platform that was developed to support this work. Section 5 presents computational experience to validate the efficiency of our platform and the performance of the algorithms automatically generated. Lastly, in Section 6 we conclude our work.

2 BACKGROUND

In this section, we give the necessary background to understand this thesis. Section 2.1 defines the notation that will be used along the text. Section 2.2 presents a brief survey on some state-of-the-art heuristic approaches to QUBO. In Section 2.3 we give a short background on automatic algorithm configuration and in Section 2.4 we review related works on automated approaches to algorithm design.

2.1 Notation and Conventions

An *instance* of QUBO is a function $f(\mathbf{x}) = \mathbf{x}^\intercal Q\mathbf{x}$, where $Q \in \mathbb{R}^{n\times n}$ is a matrix of real numbers and $\mathbf{x} \in \{0,1\}^n$ is a vector of binary variables, i.e., $\mathbf{x} = [x_1, \ldots, x_n]$. Similarly, we can also work with an instance in its expanded form $f(\mathbf{x}) = \sum_{i\in[n]} \sum_{j\in[n]} q_{ij}x_ix_j$, where q_{ij} denotes the entry in Q in the ith row and jth column. Throughout the text we often refer to an entry q_{ij} of Q as *coefficient* q_{ij}, and we use the letter n to refer to the size of an arbitrary instance (i.e., the number of variables in $\mathbf{x}$).

It is common in the literature to assume that the matrix Q associated to an instance f is symmetric. This assumption is valid because, given an a matrix $Q \in \mathbb{R}^{n\times n}$ and a vector $\mathbf{x} \in \{0,1\}^n$, it is easy to check that $\mathbf{x}^\intercal Q\mathbf{x} = \mathbf{x}^\intercal Q^\intercal \mathbf{x}$. Therefore,

$$\mathbf{x}^\intercal Q\mathbf{x} = \frac{1}{2}(\mathbf{x}^\intercal Q\mathbf{x} + \mathbf{x}^\intercal Q^\intercal \mathbf{x}) = \mathbf{x}^\intercal \hat{Q}\mathbf{x}$$

where $\hat{Q} = \frac{1}{2}(Q + Q^\intercal)$, which is symmetric.

A *solution* to an instance f is an assignment of values to each variable x_i in $\mathbf{x}$. It is common to try to improve a solution $\mathbf{x}$ by changing the value of one of its variables. This operation is called a *flip*. More precisely, to flip a variable x_i means replacing its value by its complement, i.e., $x_i = 1 - x_i$. The *1-flip neighborhood* of a solution $\mathbf{x}$ is the set of all solutions obtained by flipping a single variable of $\mathbf{x}$. When facing the decision of flipping a variable x_i or not, we often consider the change in the objective value caused by the flip. We denote this quantity by $\Delta(\mathbf{x}, i)$. More precisely, if $\mathbf{x}'$ is the solution we obtain by flipping x_i in $\mathbf{x}$, then $\Delta(\mathbf{x}, i) = f(\mathbf{x}') - f(\mathbf{x})$.

2.2 Heuristic Approaches

In this section, we give an overview of the main heuristic approaches that can be found in the literature. As stated in the first chapter, the literature on heuristic algorithms for QUBO is vast, and a complete and fair account of all the work that has been done is out of the scope. In the next paragraphs, we give more details about heuristics that directly influence this work. For a more complete treatment, the interested reader is referred to Punnen (2022).

2.2.1 Multistart Tabu Search

Palubeckis (2004) proposed 5 different multistart strategies to solve QUBO, named MST1, MST2, MST3, MST4 and MST5. Relevant to this thesis are MST2 and MST3, which we explain in the following paragraphs. Both algorithms are based on the repeated application of two components: a constructive heuristic and a tabu search. At each iteration, a solution is created using the constructive heuristic and then refined using the tabu search. The tabu search used in both algorithms is the same, the main difference between both algorithms is the constructive heuristic.

The tabu search procedure, denoted here by TS_0, consists of a simple short-term memory and operates on the 1-flip neighborhood (i.e., each move is a flip of a variable). The tabu tenure constant is set to $\min\{20, n/4\}$ and the termination criterion is a maximum number of iterations (there is no termination based on stagnation). If after applying a move we obtain a solution $\mathbf{x}$ that is better than the best solution found by the top-level algorithm (in this case, MST2 or MST3) so far, we try to improve it with a local search. Just like the tabu search, this local search also operates on the 1-flip neighborhood. It iterates over each variable x_i in order (i.e., from x_1 to x_n), and checks if the objective value can be increased by flipping x_i. If that is the case, we flip x_i and proceed to analyze the next variable (i.e., x_{i+1}). If any flip occurred while analyzing variables x_1 to x_n, we start another iteration. Otherwise, the local search ends.

2.2.1.1 MST2

As it was stated above, MST2 works by repeatedly constructing new solutions and refining them with TS_0. At each iteration, the solution returned by TS_0 is passed as an argument to the constructive heuristic, which we explain in the next paragraphs.

Let f be an instance and $\mathbf{x}$ a solution to f that is given as input to MST2's constructive heuristic. The construction of a new solution works as follows: first, we construct a set I^* containing the indices of the variables in f that are likely to change their values if $\mathbf{x}$ were to be transformed into an optimal solution. Second, we apply a steepest ascent algorithm that decides the value of the variables whose index is in I^*. Variables whose index is not included in I^* are set to 0.

We first show how I^* is constructed. Let $I = [n]$ be a set containing the indices of the variables of f. The indices to be included in I^* are selected probabilistically, and the probability of an index $i \in I$ being included is based on a measure of how much its associated variable contributes to the objective function (i.e., how much $f(\mathbf{x})$ would change if x_i were to be flipped in $\mathbf{x}$). To compute this probability, we first assign to each index i a score e_i calculated as follows

$$e_i = \begin{cases} 1 - \frac{\Delta(\mathbf{x},i)}{\delta_{\min}}, & \Delta(\mathbf{x},i) \leq 0 \text{ and } \delta_{\min} < 0, \\ 0, & \Delta(\mathbf{x},i) = \delta_{\min} = 0, \\ 1 + \lambda_{mst2}\frac{\Delta(\mathbf{x},i)}{\delta_{\max}}, & \Delta(\mathbf{x},i) > 0. \end{cases}$$

where $\delta_{\min} = \min_{i\in[n]} \Delta(\mathbf{x},i)$, $\delta_{\max} = \max_{i\in[n]} \Delta(\mathbf{x},i)$, and λ_{mst2} is a tuning factor. This formula assigns scores between 0 and λ_{mst2}. Indices associated with variables that decrease the objective value when flipped receive low scores, while the opposite is true for indices whose associated variable can increase the objective value. An index $x_k \in I$ is picked with probability $e_k / \sum_{i\in I} e_i$. Each time we pick an index k, we flip its associated variable in $\mathbf{x}$ (i.e. $\mathbf{x} := [x_1, x_2, \ldots, x_{k-1}, 1 - x_k, \ldots, x_n]$) and remove it from I. Before picking another index, we have to update the score values $e_i, i \in I$ considering the new solution $\mathbf{x}$. We stop this process when $|I^*|$ is $\max\{10, \lfloor \alpha_{mst2} n \rfloor\}$, where α_{mst2} is a parameter.

We now describe the steepest ascent algorithm that decides the final value of the variables whose index is in I^*. It takes as input the same solution $\mathbf{x}$ received as input by the procedure that constructed I^*. First, all variables whose index is not

in I^* are set to 0 (i.e., $x_i := 0$, for all $i \in [n] \setminus I^*$.) Next, at each iteration, the value of a variable whose index is in I^* is decided as follows: let $i = \arg\max_{i \in I^*} \Delta(\mathbf{x}, i)$. If $\Delta(\mathbf{x}, i) < 0$, then x_i is set to 0 in $\mathbf{x}$, otherwise it is set to 1. We remove i from I^* and go to the next iteration. Once I^* becomes empty, we return $\mathbf{x}$.

2.2.1.2 MST3

The constructive algorithm employed by MST3 builds on ideas that are similar to the ones employed in MST2. Let f be an instance, $\mathbf{x}$ be a solution to f where all variables are set to 0, and $I = [n]$ be a set with the indices of the variables of f. To each index $i \in I$ we assign a measure of importance c_i. The value of c_i is calculated based on the coefficients q_{ij} of f (Section 2.1). More precisely, $c_i := \sum_{j \in [n]} q_{ij}$, the sum of all coefficients associated to x_i.

The algorithm executes $|I|$ iterations, and at each iteration the value of a variable is decided as follows: first, we sort the indices in I in descending order according to the value c_i, and the first $\min\{\mu, |I|\}$ indices (μ is a parameter) are added to a set I'. To each index $i \in I'$ we associate a probability $p_i = |c_i| / \sum_{j \in I'} |c_j|$ and, using these probabilities, we randomly select an index $k \in I'$. If $c_k > 0$, we set x_k to 1 in $\mathbf{x}$, otherwise we set it to 0. The index k is then removed from I and, if x_k was set to 0, we subtract from each c_i the coefficients associated with x_k, i.e., $c_i := c_i - 2q_{ik}, i \in I$. Once I becomes empty, we return $\mathbf{x}$.

2.2.2 Iterated Tabu Search

Palubeckis (2006) proposed an iterated tabu search procedure that combines the tabu search TS_0 described above and a perturbation strategy. The idea is to, starting from an initial random solution $\mathbf{x}$, 1) improve $\mathbf{x}$ with TS_0 until a local minimum is reached, and then 2) escape from this local minimum using the perturbation strategy. Steps 1) and 2) are repeated until the termination criterion is met.

We now describe the employed perturbation strategy, here denoted $PERT_0$. Let f be an instance, $\mathbf{x}$ be the solution to be perturbed, $I = [n]$ be a set with the indices of the variables of f, and $\tilde{r}$ be an integer randomly selected from the interval $[d_1, d_2 n]$[1]. The value of $\tilde{r}$ represents the perturbation size (i.e., the number

[1]The values of d_1 and d_2 are parameters.

of variables that will be flipped). We repeat the following steps until $\tilde{r}$ variables are flipped:

1. Rank all the indices $i \in I$ in decreasing order according to the value of $\Delta(\mathbf{x}, i)$;
2. select the b indices with better rank according to 1) and create a set J with them;
3. randomly pick, with uniform probability, an index $k \in J$, flip x_k in $\mathbf{x}$, and remove it from I.

$PERT_0$ always takes as input a solution that is optimal in the 1-flip neighborhood. As a result, $\Delta(\mathbf{x}, i) < 0$ for all $i \in [n]$. When we rank variables in step 1) according to the value of $\Delta(\mathbf{x}, i)$, we are actually prioritizing the variables that cause the smallest decrease in the objective function when flipped in $\mathbf{x}$. Instead of just flipping the first $\tilde{r}$ variables according to the rank of step 1), we introduce some degree of randomization into the process in steps 2) and 3). The degree of randomization is controlled by the parameter b.

2.2.3 Diversification-driven Tabu Search

Glover, Lü and Hao (2010) proposed a diversification-driven tabu search. It is based on the repeated application of two components: a simple tabu search and a perturbation method that works based on information collected from a pool of high-quality solutions.

The employed tabu search procedure, denoted here by TS_1, works in the 1-flip neighborhood. Each time a move is applied to a variable x_i (i.e., the value of x_i is changed to its complementary value), the tabu tenure associated to this variable is set to $c+\text{rand}(10)$[2]. If the tabu tenure of a variable does not allow the application of a move, but applying this move would result in a solution that is better than the best known solution, we apply the move anyway. The termination criterion is a maximum number of non-improving iterations.

The perturbation method, here denoted by $PERT_1$, uses a set $E = \{\mathbf{x}^1, \ldots, \mathbf{x}^e\}$ containing e[3] locally optimal solutions in the 1-flip neighborhood. Based on the solutions stored in E, two vectors are created: ω_i and ϕ_i, for $i \in [n]$. The former holds

[2]The value of c is a parameter and rand(10) is a function that randomly selects a number between 1 and 10.

[3]The value of e is a parameter.

the number of times the variable x_i has been flipped and the latter holds the total number of solutions stored in E whose value for x_i is 1. Given a solution $\mathbf{x}$ to be perturbed, we assign to each variable a score based on the statistics collected in the vectors ω and ϕ. The score s_i of a variable x_i is

$$s_i = \frac{\phi_i(e - \phi_i)}{e^2} + \beta\left(1 - \frac{\omega_i}{\Omega}\right)$$

where β is a parameter and $\Omega := \max_{i \in [n]} \omega_i$. The score function has two terms. The first term assigns high scores to variables that do not have a strong tendency towards a value in solutions in E. For example, if a variable x_i assumes the value 1 in half of the solutions in E, then

$$s_i = \frac{\phi_i(e - \phi_i)}{e^2} = \frac{\frac{e}{2}(e - \frac{e}{2})}{e^2} = \frac{1}{4}$$

, which is the highest possible score this term can attribute to a variable. On the other hand, if a variable x_i assumes the same value in all elite solutions, then this terms gives to it a score of 0. The second term gives high scores to variables that are seldom flipped. Glover, Lü and Hao (2010) believe that changing the value of such variables helps leaving a local minimum. To control the importance of this term in the final score, the parameter β is used.

All the variables are sorted in decreasing order based on the score value, and to each variable x_i we assign a probability P_i proportional to how its score s_i is ranked among the other variables. For example, suppose that x_i is the kth highly-scored variable. Its associated probability is

$$P_i = \frac{k^{-\lambda}}{\sum_{j \in [n]} j^{-\lambda}}$$

where λ controls how the rank obtained by a variable impacts its probability of being selected. Higher values of λ make the probabilities assigned to the variables with highest and lowest score more distant to each other. Using the probabilities P_i, we randomly pick ${\gamma_g}^4$ variables and flip their values.

We are now ready to explain the algorithm. It starts by building a set E of e locally optimal solutions as follows: an initial random solution $\mathbf{x}^0$ is constructed, improved with TS_1 and added to E. In the next iterations, a solution is randomly

[4]The value of γ_g is a parameter.

selected from E, perturbed by $PERT_1$, improved by TS_1 and added to E. This process is repeated until $|E| = e$. After E is built, we proceed to the main loop, where at each iteration a solution $\mathbf{x}$ is randomly selected from E, perturbed by $PERT_1$, and improved by TS_1. Let $\mathbf{x}'$ be the resultant solution. If $\mathbf{x}'$ is different from all the solutions already in E and is better than the worst solution in E, than $\mathbf{x}'$ replaces the worst solution in E. At the end, the best solution in E is returned.

2.2.4 Path-relinking

Wang et al. (2012) proposed an algorithm based on the recombination of high-quality solutions. The algorithm uses two components: a path-relinking procedure and a tabu-search procedure equivalent to TS_1.

Path-relinking is an intensification strategy based on the recombination of two solutions. The main idea is to combine two solutions $\mathbf{x}^a$ and $\mathbf{x}^b$ to form a series of intermediary solutions $S = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^p\}$ that are, in some sense, "between" $\mathbf{x}^a$ and $\mathbf{x}^b$.

Let $\mathbf{x}^a$ and $\mathbf{x}^b$ be two different solutions to an instance f of QUBO, and P be an empty set. Path-relinking starts by computing $D = \{i : i \in [n] \text{ and } x_i^a \neq x_i^b\}$, i.e., a set with the indices of the variables where $\mathbf{x}^a$ and $\mathbf{x}^b$ differ. The method takes $|D| - 1$ iterations to complete, and at the kth iteration an index $t \in D$ is selected and added to P. A new solution $\mathbf{x}^k = [x_i^k = x_i^b, i \in P;\ x_i^k = x_i^a, i \in [n] \setminus P]$, which is in the "path" between $\mathbf{x}^a$ and $\mathbf{x}^b$, is created and added to an ordered set S. Finally, we remove t from D and go to the next iteration.

At the end of all iterations, we have a sequence of intermediary solutions. Wang et al. (2012) discard all solutions $\mathbf{x}^k \in S$ such that the Hamming distance between $\mathbf{x}^k$ and $\mathbf{x}^a$ or between $\mathbf{x}^k$ and $\mathbf{x}^b$ is less than $\gamma|D|$, where γ is a real number between 0 and 0.5. From the remaining solutions, the one with the best objective value is returned.

The strategy used to select an index $t \in D$ to construct $\mathbf{x}^k$ was not specified in the description above. Wang et al. (2012) experiment with two strategies: 1) t is randomly selected and 2) t is the index that, when flipped in $\mathbf{x}^{k-1}$, gives the biggest increase (or the smallest decrease) in the objective function.

We now explain the algorithm. It starts by building an ordered set $E =$

$\{\mathbf{x}^1, \ldots, \mathbf{x}^r\}$ of r high-quality solutions[5]. A high-quality solution is constructed in two steps: 1) a random solution is created and 2) is then improved by TS_1. After E is built, we enter in the main loop. At each iteration, we select a pair of distinct solutions $\mathbf{x}^a, \mathbf{x}^b \in E$ and apply path-relinking from $\mathbf{x}^a$ to $\mathbf{x}^b$. The solution returned by path-relinking is then improved by TS_1. Let $\mathbf{x}^p$ be the solution returned by TS_1. If $\mathbf{x}^p$ is different from all solutions in E and is better than some solution in E, then replace the worst solution in E by x^p. We repeat this same process again, but now applying path-relinking from $\mathbf{x}^b$ to $\mathbf{x}^a$. We iterate inside the main loop until there is no pair of distinct solutions in E that was not processed. Once this happens, we rebuild the set E and start again.

2.3 Automatic Algorithm Configuration

Let A be a parameterizable algorithm with parameters $p_1, p_2, \ldots, p_m$. A *configuration* of A is a valid assignment of values to each parameter p_i, and is denoted by θ. The set of all valid configurations is called the *configuration space* of A, and is denoted by Θ.

Each parameter p_i has an associated domain D_i. For our purposes, parameters can be classified into three classes, depending on their domain. If the domain D_i of a parameter p_i is a subset of the integer or real numbers, we call p_i an *integer parameter* or *real parameter*. On the other hand, if D_i is a set of discrete values with no defined ordering relation between its elements (e.g., $D_i = \{\text{red}, \text{green}, \text{blue}\}$), then we call p_i a *categorical parameter*.

In Automatic Algorithm Configuration we are usually interested in finding a configuration $\theta \in \Theta$ that maximizes the performance of A on a particular set of instances. For example, if A is a heuristic algorithm for the Traveling Salesperson Problem, we would like to find a configuration that minimizes the average tour length obtained on a set of Euclidean instances. The literature describes a variety of algorithms that aim to achieve this goal, the so called algorithm configurators (ACs). Some examples of algorithm configurators are ParamILS (HUTTER et al., 2009), GGA (ANSÓTEGUI; SELLMANN; TIERNEY, 2009), SMAC (LINDAUER et al., 2022) and irace (LÓPEZ-IBÁÑEZ et al., 2016). The last one is used in this work, and we give an overview of its functionality in the next section.

[5]The value of r is a parameter.

2.3.1 irace

The irace package (LÓPEZ-IBÁÑEZ et al., 2016) is an implementation of Iterated Racing (IR), a method for algorithm configuration. Let A be an algorithm we want to optimize and I be a set of training instances. IR starts by randomly creating[6] a population of configurations. At each iteration, all the configurations in the population participate in a selection procedure called racing, which filters the best configurations (the remaining ones are discarded). New configurations that are similar to the surviving ones are then sampled and included in the population.

We start by describing the racing process. Suppose we have a population of configurations $C = \{\theta_1, \theta_2, \ldots, \theta_w\}$ and let $I' \subseteq I$ be the set of instances. At each step of the race, we execute all configurations in C over a single instance $i \in I'$. After a predetermined number of steps, we apply a statistical test (e.g., Friedman test (CONOVER, 1998)) to check if any configuration in the race performed statistically worse than any other configuration on the instances seen so far. If that is the case, then the configuration is eliminated from the race. The race continues until the number of surviving configurations is less than or equal to a predetermined constant, the instance set is exhausted, or the maximum number of algorithm evaluations is exceeded.

After each race, we have a set of surviving configurations, which we call *elite configurations*. As stated above, based on these we sample new configurations to add to the population. To generate a new configuration, first we randomly select an elite configuration. Each parameter of a configuration has an associated probability distribution that is biased towards values that are close to its value. Hence, to sample a configuration similar to the selected elite we can sample one parameter value at a time.

2.4 Automated Algorithm Design

A design space is usually defined based on one or more templates of algorithms. Consider, for example, the template of a Greedy Randomized Adaptive Search Procedure (GRASP) (RESENDE; RIBEIRO, 2003). It consists of basically

[6] To randomly create a configuration we proceed parameter-wise: a value is uniformly sampled from the domain of each parameter p_i.

two components: a greedy constructive heuristic and a local search. The exact form of these components is not specified, so different combinations of them give form to a design space. In practice, however, this design space can be considerably more complex, because each component may have its own design space.

```
1 <grasp> ::= GRASP(<construction>, <search>)
2 <construction> ::= CONSTRUCTION_1 | CONSTRUCTION_2
3 <search> ::= SEARCH_1 | SEARCH_2
```

Figure 2.1 – A simple design space of GRASP algorithms.

As it was stated in the introduction, the standard approach to represent the design space is to use a context-free grammar. Figure 2.1 gives an example of simple design space for GRASP algorithms. The instantiation of an algorithm can be viewed as the repeated application of the grammar rules until all that rests are terminals. The expression `GRASP(CONSTRUCTION_1, SEARCH_2)`, for instance, represents a GRASP that uses the constructive heuristic `CONSTRUCTION_1` and the local search `SEARCH_2`.

A common strategy to explore the grammar is by using an Algorithm Configurator (AC). As we saw in Section 2.2, an AC takes a target algorithm A and a set of training instances I_{train}, and it tries to find a configuration that maximizes the performance of A on I_{train}. It does so by repeatedly evaluating configurations on instances in I_{train} and, based on the evaluations, decide which configurations to maintain and which to discard. This workflow implies the existence of two things: 1) a parametric representation of the grammar and 2) a platform that takes a configuration and an instance, builds the algorithm represented by the configuration and evaluates it on the instance. The parametric representation is usually easy to obtain, and there are methods in the literature to do this conversion automatically. Regarding the platform, some works pack all components in a single application, while others write the source code of the specified algorithm, compile and execute it on the fly. In the next paragraphs, we review some works that explore the ideas described so far in this section.

Mascia et al. (2013) propose a grammar describing a design space of Iterated Greedy (IG) algorithms for the permutation flow-shop scheduling problem. An IG algorithm iteratively "destructs" the current solution and then "reconstructs" it into

a full solution using a greedy heuristic. The form of the operations of destruction and reconstruction depends on the problem being solved. Besides the grammar, they also provide a method to convert grammars into parameters. They use this method to create a parametric representation of their grammar and then explore it using irace. In their system, algorithms are implemented, compiled and executed on the fly.

Marmion et al. (2013) propose a grammar encoding a design space of Generalized Local Search (GLS) algorithms to solve the permutation flow-shop with weighted tardiness problem. Their template allows the instantiation of popular algorithms, like Simulate Annealing (KIRKPATRICK; GELATT; VECCHI, 1983), Random Iterative Improvements (HOOS; STÜTZLE, 2005), and Variable Neighborhood Search (HANSEN; MLADENOVIĆ, 2001). A GLS consists of basically three components: a search procedure (which can be another GLS, thus the template is recursive), a perturbation method, and an acceptance criterion. At each iteration, a given solution is perturbed by the perturbation method, improved by the search procedured and the acceptance criterion decides if it accepts the solution or stays with the current one. They convert the grammar into parameters using the method of Mascia et al. (2013), and then use irace to explore the design space. In the same way as Mascia et al. (2013), algorithms are implemented, compiled and executed on the fly during the execution of irace. In their platform, algorithms are built using ParadisEO (DREO et al., 2021), an open-source C++ framework that provides a library of reusable components.

Mascia et al. (2014) propose two grammars describing a design space of IG algorithms, one of for the 1-dimensional bin-packing problem and the other for the permutation flowshop with weighted tardiness. They also introduce an algorithm to automatically convert a grammar into a parametric representation. The algorithm is implemented in a tool called *grammar2code* that automatically generates the parametric representation of a grammar specified in XML format. This tool is also responsible for generating source code given the parametric representation of an algorithm, which is then compiled and executed. Both design spaces are explored using irace.

López-Ibáñez, Marmion and Stützle (2017) propose a grammar describing a design space of GLS algorithms for the permutation flow-shop with weighted tardiness, QUBO, and the travelling salesperson problem with time windows. Parts

of the grammar derive components that are general and can be used to instantiate algorithms for all the three problems. Other parts, however, derive problem-specific components. Following Mascia et al. (2014), they use *grammar2code* to generate the parametric representation of the grammar and the source code of an algorithm given its parametric representation. They also use irace to explore the design space.

More close to this thesis is the work of Souza and Ritt (2018), who built their own platform. They extracted components from state-of-the-art heuristics to solve QUBO and developed a platform that is not restricted to algorithms following a specific template. They proposed a grammar that encodes a diverse design space of algorithms and allows the instantiation of state-of-the-art algorithms, like the ones proposed in Palubeckis (2004), Palubeckis (2006), Glover, Lü and Hao (2010) and Wang et al. (2012). They constructed a parametric representation of the grammar following the ideas of Mascia et al. (2013) and Mascia et al. (2014), and explored it using irace. As a result, they found new algorithms that are very competitive with the state-of-the-art for two instance domains.

3 A GRAMMAR FOR HEURISTIC ALGORITHMS

In this chapter we specify a context-free grammar that compactly encodes a space of heuristic algorithms. We also show how the grammar can be converted to a parametric representation.

3.1 Grammar Definition

The grammar encoding our design space is shown in Figure 3.1. It comprises 10 rules and 28 distinct terminals and is based on the grammar proposed by Souza and Ritt (2018), with some modifications.

The first modification consists in splitting the search procedures into two levels. In the first level we have strong metaheuristics usually used in state-of-the-art algorithms. In the second level, we have search procedures that are typically used as sub-components of the components in the first level. For example, the popular algorithm proposed by Palubeckis (2006) is basically an iterated local search (ILS) in combination with a simple tabu search (STS). We hope that with this modification the search procedure used to explore the design space will spend more time evaluating promising algorithms.

The second modification consists in the introduction of new components. Inspired by the work of Festa et al. (2002), we introduced a variable neighborhood search (VNS), a variable neighborhood search with path relinking (VNSPR), and a GRASP with path-relinking (GRASPPR). Inspired by the work of Palubeckis (2004), we introduced two new construction methods: PT2 which was used in the algorithm MST2, and SP which was used in the algorithm MST3 (Section 2.2.1).

In the following sections we give more details about each component. Pseudocode for some components is provided in Appendix A.

```
<start> ::= <search> | <construction> | <recombination>
<search> ::= ILS(<intensification>, <pert>)
        | ILSE(<intensification>, <pert>)
        | VNS(<intensification>, <pert>)
        | VNSPR(<intensification>, <pert>, <improvement>)
<intensification> ::= LS(<improvement>)
        | NMLS(<improvement>)
        | VNS(<intensification>, <pert>)
        | <ts>
<ts> ::= STS | RTS
<improvement> ::= FI | FI-RR | BI
        | SI | SI-PARTIAL | SI-PARTIAL-RR
<perturbation> ::= RANDOM(<step>)
        | LEAST-LOSS(<step>)
        | DIVERSITY(<step>)
<step> ::= UNIFORM | GAUSSIAN | EXPONENTIAL | GAMMAM
<construction> ::= GRASP(<constructor>, <intensification>)
        | GRASPR (<constructor>, <intensification>, <improvement>)
<constructor> ::= ZERO | HALF | PT2 | SP
<recombination> ::= RER(<intensification>, <improvement>)
```

Figure 3.1 – Context-free grammar encoding our design space.

3.1.1 Search

Our platform implements four search components, namely ILS, ILSE, VNS, and VNSPR. These templates were inspired by the metaheuristics usually employed

in state-of-the-art algorithms, like the ones described in Section 2.2.

ILS follows the template of an Iterated Local Search algorithm, as described by Palubeckis (2006). It requires the specification of `perturbation` and `intensification` components. Algorithm 1 illustrates the implementation.

ILSE stands for Iterated Local Search Elite, and was inspired by the iterated local search template used by Glover, Lü and Hao (2010). Besides the standard components of an iterated local search, it also requires the specification of e, the reference set size. Algorithm 2 depicts the implementation.

VNS follows the basic structure of a variable neighborhood search, as described by Festa et al. (2002), with one small modification. Festa et al. (2002) create a random solution at the beggining of each iteration. In our implementation, this initial solution is the result of a perturbation of the current solution. We can replicate the behavior of Festa et al. (2002) by selecting `RANDOM` as our perturbation method. In summary, VNS requires the specification of the maximum neighborhood size $kmax$ and of `intensification` and `perturbation` components. Algorithm 3 depicts the general scheme.

VNSPR stands for Variable Neighborhood Search with Path Relinking. This template is inspired by an algorithm proposed by Festa et al. (2002). It has the same basic structure of a VNS, but with the addition of a reference set holding high-quality solutions that is updated using path-relinking. The path-relinking procedure implemented in our platform is the same one described in Section 2.2.4, and it accepts different strategies for choosing the next variable in the path from $\mathbf{x}^a$ to $\mathbf{x}^b$. These strategies are implemented by an `improvement` component, hence it is also necessary to specify one. In addition, a value for γ must also be provided. Algorithm 4 gives a high-level view of VNSPR.

3.1.2 Construction

Our platform implements two constructive algorithms: a GRASP and a GRASP with path-relinking (GRASPPR). The implementation of GRASP follows the template described by Festa et al. (2002), as Algorithm 5 shows. Its instantiation requires the specification of `constructor` and `intensification` components.

The GRASPPR template, depicted in Algorithm 6, was inspired by one of the algorithms proposed by Festa et al. (2002). It employs the same intensification

strategy used in VNSPR, where high-quality solutions are kept in a reference set E and path-relinking is used to combine solutions in this set. Hence, in addition to GRASP parameters, GRASPPR also requires an `improvement` component and a value for γ.

3.1.3 Constructor

The grammar specifies 4 constructors, namely ZERO, HALF, PT2, and SP. The last two were extracted from Palubeckis (2004). More specifically, PT2 is the construction method employed in MST2 and SP is the construction method employed in MST3 (see Section 2.2 for a description of these two methods). PT2 requires the specification parameters α_{mst2} and λ_{mst2}, while SP requires a value for μ.

The ZERO constructor is an adaptation of the greedy constructive heuristic proposed by Festa et al. (2002) to the context of QUBO. The main idea is to start with a solution where all variables are set to zero and to greedily select a variable that, when flipped, improves the objective value. More specifically, we start by creating a solution $\mathbf{x}$ where all variables are set to 0, initializing a set $I = [n]$ containing the indices of all variables, and selecting a value α from the interval $[0, 1)$ with uniform probability. At each iteration, we try to set a variable to 1. To do so, first we compute $w_{min} = \min_{i \in I} \Delta(\mathbf{x}, i)$ and $w^{max} = \max_{i \in I} \Delta(\mathbf{x}, i)$. These two values are used to compute a cut-off value $\mu = \max\{0,\ w_{min} + \alpha(w^{max} - w_{min})\}$. Using μ, we create a restricted candidate set $I' = \{i : i \in I \text{ and } \Delta(\mathbf{x}, i) > \mu\}$. If $I' = \emptyset$, we stop the process and return x. Otherwise, we randomly pick an index $k \in I'$, flip x_k in $\mathbf{x}$, remove k from I and go to the next iteration.

The HALF constructor follows the same approach, but now all variables are initialized at 0.5, and at each iteration we greedily choose a variable to fix to 0 or 1.

3.1.4 Recombination

Our grammar only comprises one recombination algorithm, RER, which stands for Repeated Elite Recombination. This template is inspired by the algorithm proposed by Wang et al. (2012). It requires the specification of an intensifica-

tion procedure and the size r of the reference set. Since RER uses path-renlinking, an `improvement` component and a value for γ must also be specified. Algorithm 7 illustrates the template.

3.1.5 Intensification

Like search procedures, intensification strategies take as input a solution and then try to improve it. The difference is that an intensification strategy executes for a short amount of time, usually limited by a maximum number of iterations. Our platform implements five intensification procedures.

LS is a simple Local Search procedure based on repeated calls to an `improvement` component. As is explained in Section 3.1.6, an `improvement` component receives a solution $\mathbf{x}$ and tries to find a variable in $\mathbf{x}$ that, when flipped, improves the objective value. If no such variable exists (i.e., $\mathbf{x}$ is locally optimal in the 1-flip neighborhood), the search stops. Algorithm 8 illustrate this component.

NMLS, which stands for Non-Monotonic Local Search, does basically the same thing, but with a probability p_{nmls} a random variable is flipped instead of the variable indicated by the `improvement` component. It was observed during experiments that high values of p_{nmls} may cause the search to go on forever, since a random move usually deteriorates the objective value and subsequent calls to `improvement` can easily find improvements (just by undoing the flips randomly made, for example). To avoid this kind of behavior, we introduce a parameter $smax$ to terminate the execution after some period of stagnation. Algorithm 9 illustrates the logic just described.

The VNS component is the same one described in Algorithm 3. When used as an `intensification` component, we have to take care to limit the time spent inside the main loop. We do that by setting the termination criterion to a maximum number of iterations $imax$.

Lastly, our platform also implements two tabu search procedures, STS and RTS. Both tabu searches work in the 1-flip neighborhood, exploring the variables in order (from x_1 to x_n). A short-term memory is used to avoid flipping the same variable twice in a short interval of time. Following Glover, Lü and Hao (2010), however, we allow a tabu move to be executed if it results in a solution that is better than any solution found during the search. STS and RTS differ only in one

aspect: STS always selects the best improving variable and flip it, while RTS, with a probability p_{rts}, selects a random variable to flip. Otherwise it proceeds as an STS.

STS and RTS share a set of parameters related to tabu tenure and termination criteria (maximum number of iterations or maximum stagnation). Regarding tabu tenure, we implemented 4 strategies: in strategy t_1 tabu tenure is a specified constant t_v. In strategy t_2 tabu tenure is $(nt_p)/100$. Strategy t_3 sets tabu tenure as equal to n/t_d. Lastly, in strategy t_4 tabu tenure is n/t_d+rand$(1, t_c)$, where rand$(1, t_c)$ is a function that randomly picks a number from the interval $[1, t_c]$ with uniform probability. Regarding the maximum number of iterations, there are 3 available strategies. Strategy i_1 allows at most i_v iterations, strategy i_2 allows at most n iterations, and strategy i_3 allows an unlimited number of iterations. Regarding the maximum number of non-improving moves (i.e., stagnation), strategy s_1 allows at most s_v non-improving iterations, strategy s_2 allows at most n and strategy s_3 does not impose any limitation.

3.1.6 Improvement

Given a solution $\mathbf{x}$, the purpose of an improvement procedure is to find a variable x_i that, when flipped, improves the objective function. Our platform implements 6 improvement strategies. All strategies investigate variables in ascending order (i.e., from x_1 to x_n).

Improvement strategy FI returns an improving neighbor as soon as it finds one. FI-RR does the same thing, but starts the exploration from the point the previous exploration ended, in a round-robin fashion. BI returns the best neighbor and SI returns a random improving neighbor. SI-PARTIAL also returns a random improving neighbor, but it only considers the first $(n \cdot f)/100$ variables in the exploration, where f is a parameter. SI-PARTIAL-RR extends SI-PARTIAL with a round-robin exploration (as is done in FI-RR).

When an improvement strategy is used inside path-relinking (see sections 2.2.4 and 3.1.4, for example), the improvement method must work on a subset D of the variables. Not only that, but it must return a variable even if this variable will not improve the objective function. In this case, FI, FI-RR and BI will return the variable in D that when flipped causes the smallest decrease in the objective value. In contrast, SI, SI-PARTIAL, and SI-PARTIAL-RR return will return a random

variable.

3.1.7 Perturbation

The purpose of a perturbation strategy is to select a subset of the variables of a solution and flip their values. Our platform implements three different perturbation strategies. Perturbation LEAST-LOSS is the same one described in Palubeckis (2006). It requires the specification of the size of the candidate set of variables b. Perturbation DIVERSITY, on the other hand, is the one described in Glover, Lü and Hao (2010), and it requires a value for the selection importance factor λ and for the frequency contribution β. Lastly, flipped variables in perturbation RANDOM are selected with uniform probability. All perturbation methods require the specification of a `step` component, which determines the size of the perturbation and is explained below.

3.1.8 Step

The purpose of a step component is to select the size of a perturbation (i.e.the number of variables that will be flipped). Our platform implements four strategies. Three of those strategies, namely UNIFORM, GAUSSIAN and EXPONENTIAL, select a value from a given interval according to a probability distribution. Let $[d_1, d_2]$ be the interval. The UNIFORM strategy samples a value uniformly from this interval. The GAUSSIAN strategy samples a value from a normal distribution with mean $(d_1+d_2)/2$ and standard deviation d_2-d_1. The EXPONENTIAL strategy samples a value from an exponential distribution with $\lambda = 1/((d_2 + d_1)/2)$.

Differently from the first three strategies, the GAMMAM strategy selects a fix perturbation size, n/g, where g is a positive integer.

3.2 Parameters

As sections 3.1.1 to 3.1.8 pointed out, some components require the specification of numerical and categorical parameters, besides the subcomponents. The existence of these parameters was hidden in the grammar presented in Figure 3.1

for presentation purposes. To account for that, we present a Table 3.1, which summarizes all the parameters mentioned in the last sections.

Table 3.1 – List of all parameters.

Parameter	Type	Component	Description
t	Categorical	<ts>	Strategy for tabu tenure
t_v	Integer	<ts>	Tabu tenure is t_v
t_p	Integer	<ts>	Tabu tenure is $(t_pN)/100$
t_d	Real	<ts>	Tabu tenure is N/t_d
t_c	Integer	<ts>	Tabu tenure is $N/t_d + rand(1, t_c)$
s	Categorical	<ts>	Strategy for maximum stagnation
s_v	Integer	<ts>	Maximum stagnation is s_v
s_m	Integer	<ts>	Maximum stagnation is s_mN
i	Categorical	<ts>	Strategy for maximum number of iterations
i_v	Integer	<ts>	Maximum number of iterations is t_v
p_{nmls}	Real	NMLS	Probability of flipping a random variable
p_{rts}	Real	RTS	Probability of a random move
f	Real	SI-PARTIAL [-RR]	Size of the partial exploration
d_1	Integer	<pert>	Min. perturbation size
d_2	Integer	<pert>	Max. perturbation size is N/d_2
g	Integer	GAMMAM	Perturbation size is N/g
b	Integer	LEAST-LOSS	Candidate variables for perturbation
β	Real	DIVERSITY	Frequency contribution
λ	Real	DIVERSITY	Selection important factor
e	Integer	ILSE	Reference set size for ILSE
r	Integer	RER	Reference set size for RER
γ	Real	RER VNSPR GRASPPR	Distance scale for path-relinking
μ	Integer	SP	Upper bound on the number of candidate variables
α_{mst2}	Real	PT2	Multiplier used to compute the size of I^*
λ_{mst2}	Integer	PT2	Tuning factor of the measure of attractiveness of a variable
$imax$	Real	VNS	Maximum number of iterations
$kmax$	Integer	VNS VNSPR	Maximum neighborhood size
$smax$	Integer	NMLS	Maximum stagnation

3.3 Parametric Representation

As stated in Section 2.4, algorithm configurators usually work on a finite space of parameters. As a result, the space encoded by the grammar has to be converted to a parametric representation. To solve this problem, we follow the ap-

proach proposed by Mascia et al. (2013), which Souza and Ritt (2018) also follow. It basically consists in representing by categorical parameters the decisions on how to apply a rule (e.g., which constructor to choose in the rule `<constructor>`). Sometimes a decision depends on previous decisions (e.g. we only have to use the rule `<constructor>` if `<algorithm>` is chosen to be `<construction>`). These situations can be encoded by conditional parameters, a type of parameter that is only activated if the attached conditions are satisfied.

In the next paragraphs, we explain how we encoded the derivations of the grammar depicted in Figure 3.1 using the approach of Mascia et al. (2013).[1] Table 3.2 summarizes all the parameters that will be mentioned. The first and second columns show the name of the parameter and its domain. The third column shows the main condition required for its activation. We say main condition because sometimes a parameter depends on another conditional parameter. For example, the parameter **pert1** depends on the parameter **search** assuming the value LS, ILSE, VNS or VNSPR, which in turn depends on the parameter **start** assuming the value SEARCH.

We start by encoding the first rule of the grammar as a parameter: a categorical parameter named **start** is introduced, which can assume the value SEARCH, CONSTRUCTION, or RECOMBINATION. Since we always have to make a decision regarding this rule, no matter the derivation, the parameter has no attached conditions. If the value of **start** is RECOMBINATION, the algorithm must be RER, so we do not have to encode any decision. If **start** is SEARCH or CONSTRUCTION, however, we have to choose between more than one option. We add parameters **search** and **construction** to take care of that. Parameter **search** is only activated when **start** is set to SEARCH and can assume the values ILS, ILSE, VNS, or VNSPR, while parameter **construction** depends on **start** being set to CONSTRUCTION and can assume the values GRASP or GRASPPR.

Every possible derivation of the grammar will require at some point the selection of an intensification procedure, so we add parameter **intens1**, without any attached conditions, to encode this decision. If **intens1** is set to VNS, an additional intensification procedure is required, so we add a separate parameter, named **intens2**, that is only activated when the value of **intens1** is VNS. To limit the recursion depth, we do not allow VNS in this second choice of intensification

[1] Parameters that are specific to a component (Table 3.1) are not considered here, but they can be included in the parametric representation following a similar logic.

procedure.

Most intensification procedures, with the exception of TS, require an improvement method.[2] As a result, the parameter controlling this decision must restrict its activation to situations where the chosen intensification procedure is not TS. Parameter **improv1** takes care of that. A second improvement method may be necessary if the high-level procedure requires path-relinking, as is the case with VNSPR, GRASPPR and RER. Parameter **improv2** deals with this case.

If **start** assumes the value CONSTRUCTION, we have to choose a constructor method, thus parameter **constructor** is added. In the same way, some search procedures require a perturbation method (i.e., ILS, ILSE, VNS, VNSPR), and parameter **pert1** encodes this decision. If the intensification procedure is VNS, another perturbation method is required, and this decision is encoded by parameter **pert2**. Perturbation strategies LEAST-LOSS and DIVERSITY require the specification of step components, and parameters **step1** and **step2** encode these decisions.

Finally, we add some parameters to decide which tabu search will be selected, STS or RTS. Since there are two parameters encoding intensification procedures (i.e.**intens1** and **intens2**), we need two parameters to deal with each case. Parameter **ts1** is active only when **intens1** is TS, while **ts2** is active only when **intens2** is active and set to TS.

[2]Although VNS does not directly require one, the intensification procedure embedded in it (selected by parameter **intens2**) will require an improvement method.

Table 3.2 – Main categorical parameters of the parametric representation.

Parameter Name	Options	Main Conditions
start	SEARCH CONSTRUCTION RECOMBINATION	-
search	LS ILSE VNS VNSPR	**start** == SEARCH
construction	GRASP GRASPPR	**start** == CONSTRUCTION
intens1	LS NMLS TS VNS	-
intens2	LS NMLS TS	**intens1** == VNS
improv1	FI FI-RR BI RI SI SI-PARTIAL SI-PARTIAL-RR	**intens1** is not TS
improv2	FI FI-RR BI RI SI SI-PARTIAL SI-PARTIAL-RR	(**start** == RECOMBINATION) or (**construction** == GRASPPR) or (**search** == VNSPR)
constructor	ZERO HALF PT2 SP	**start** == CONSTRUCTION
pert1	RANDOM LEAST-LOSS DIVERSITY	**search** in {LS, ILSE, VNS, VNSPR}
pert2	RANDOM LEAST-LOSS DIVERSITY	**intensification** == VNS
step1	UNIFORM GAUSSIAN EXPONENTIAL GAMMAM	**pert1** in {LEAST-LOSS, DIVERSITY}
step2	UNIFORM GAUSSIAN EXPONENTIAL GAMMAM	**pert2** in {LEAST-LOSS, DIVERSITY}
ts1	STS RTS	**intens1** == TS
ts2	STS RTS	**intens2** == TS

4 A PLATFORM FOR DESIGN SPACE EXPLORATION

This section describes the platform that was built to support this work.

4.1 Architecture

Souza and Ritt (2018) introduced in the literature a platform named AutoBQP[1], which implements their approach. This platform, however, has some limitations that inhibit further research on the topic, which we intend to overcome with our new platform. We discuss some of these limitation below:

- AutoBQP is fully coupled to the grammar and its parametric representation, so minimal changes to the design space require a modification to the source code. In addition, algorithms are specified by their parametric representation, which is not pleasant for humans to work with and, consequently, makes the countless tests and small experiments carried out during a research project more laborious.
- AutoBQP generates source code for the specified algorithm, which must be compiled and run. In the author's experience, this approach adds unnecessary complexity to the research work, as the artifacts produced during compilation can be cumbersome to manage.
- AutoBQP only allows termination based on maximum run time, which is a commonly used criterion in the research literature for reporting results. However, this criterion is highly dependent on the conditions of the desktop or the server running the algorithm, especially when the algorithm uses all available resources, such as memory and CPU. It is not uncommon to see significantly different results on the same platform for two runs of the same algorithm due to background processes stealing some of the algorithm's processing cycles. Consequently, reproducing results from previous experiments can be challenging.

The platform we built was written in C++ in an object-oriented style. The grammar presented in Figure 3.1 has 28 distinct terminals, hence our platform has 28 different components. From these 28 components, 23 come from the code base

[1] <https://github.com/souzamarcelo/AutoBQP>

```
class Constructor {
public:
  virtual void construct(Solution &S) = 0;
};
```

Figure 4.1 – Constructor interface.

Table 4.1 – Components in our platform and its respective interfaces.

Interface	**Components**
Algorithm	LS NMLS ILS ILSE VNS VNSPR STS RTS GRASP GRASPPR RER
Constructor	ZERO HALF PT2 SP
Improvement	FI FI-RR BI SI SI-PARTIAL SI-PARTIAL-RR
Perturbation	RANDOM LEAST-LOSS DIVERSITY
Step	UNIFORM GAUSSIAN EXPONENTIAL GAMMAM

of Souza and Ritt (2018). The implementation of the constructors `PT2` and `SP` was extracted from the work of Dunning, Gupta and Silberholz (2018), while the implementation of VNS, VNS with path-relinking and GRASP with path-relinking is ours.

Each component is implemented as a class, and each class implements an interface, depending on its function. Figure 4.1 gives an example. The interface Constructor defines a method called `construct` that takes as an argument a reference to a `Solution` object. All components in our platform whose goal is to construct a new solution must implement this interface. Our platform defines five interfaces, and Table 4.1 shows which components implement which interface. Note that, although the grammar makes a distinction between search and intensification procedures, in our platform they implement the same interface, because their function is the same (i.e., they take a solution and try to improve it).

An important aspect of our implementation is the way components interact

```
class ConstructorFactory {
public:
  static Constructor *get_constructor(const json &description) {
    std::string type = description["type"].get<std::string>();
    if (type == "ZERO")
      return new ZERO();
    else if (type == "HALF")
      return new HALF();
    else if (type == "PT2")
      return new PT2(description["alpha"].get<double>(),
                     description["lambda"].get<double>());
    else if (type == "SP")
        return new SP(description["mu"].get<int>());
    else{
      // Code removed for clarity
    }
  }

```

Figure 4.2 – Constructor factory.

with each other. An iterated local search (ILS), for example, requires the specification of an intensification procedure (which, as Table 4.1 shows, implements the interface Algorithm). We solve this problem by using factory methods, a popular design pattern in software engineering. Each type of component (e.g., `constructor`) has an associated factory class. This factory class has a static factory method that, given a JSON description of the component, returns a pointer to an object with the same type of the interface implemented by the component. Figure 4.2 gives an example.

As Figure 4.2 implies, components are typically constructed using the information contained in a JSON object. Figure 4.3 shows the JSON description of a VNS. Note how `improvement` and `perturbation` components are embedded in the description. Figure 4.4 shows the implementation of a constructor that takes such a JSON description and constructs a VNS component.

By embedding JSON objectives insided each other, we can instantiate complex algorithms like the one proposed by Glover, Lü and Hao (2010) and explained in Section 2.2.3. Figure 4.5 shows its JSON description.

```
{
  "search": {
    "type": "LS",
    "description": {
      "improvement": {
        type": "BI"
      }
     }
   },
   "perturbation": {
     "type": "RANDOM"
    },
    "kmax": 100
}
```

Figure 4.3 – JSON description of a VNS.

```
VNS::VNS(const json &description){
  search = SearchFactory::get_search(description["search"]);
  pert = PerturbationFactory::get_perturbation(description["perturbation"]);
  kmax = description["kmax"].get<unsigned>();
}
```

Figure 4.4 – VNS constructor.

4.2 Flip Evaluation

The most fundamental operation of practically all heuristic procedures is to check the change in the objective function obtained by flipping a variable from 0 to 1 and vice versa. Hence, the efficiency of an implementation is strongly influenced by the complexity of that operation. At the backbone of our platform there is a fast flip evaluation technique extracted from the code base of Souza and Ritt (2018).

Given a solution $\mathbf{x}$ and solution $\mathbf{x}'$ in the one-flip neighborhood of $\mathbf{x}$, naively computing $f(\mathbf{x}') - f(\mathbf{x})$ results in a time complexity of $O(n^2)$. Put simply, the implemented flip evaluation technique works by keeping in memory a vector Δ of size n holding the change in the objective value obtained by flipping any variable, hence checking whether or not flipping a variable would results in a better solution can be done in constant time. Every time a variable is actually flipped, Δ can be updated in $O(n)$. For more details about this technique we refer the reader to Alidaee, Kochenberger and Wang (2010).

```
{
"algorithm": {
    "type": "SEARCH",
    "description": {
        "type": "ILSE",
        "description": {
            "search": {
                "type": "STS",
                "description": {
                    "t": "t4",
                    "td": 100,
                    "tc": 10,
                    "s": "s2",
                    "sm": 20,
                    "i":"i3"
                }
            },
            "perturbation": {
                "type": "DIVERSITY",
                "description": {
                    "beta": 0.3,
                    "plambda": 1.2,
                    "step": {
                        "type": "GAMMAM",
                        "description": {
                            "g": 4
                        }
                    }
                }
            },
            "r": 8
        }
    }
}
}
```

Figure 4.5 – JSON specification of the algorithm proposed by Glover, Lü and Hao (2010)

4.3 Termination Criteria

It is common practice in the literature to use wall-clock time as the termination criterion of an algorithm. The performance of an algorithm as measured by wall-clock time, however, is sensitive to the conditions of the underlying platform (e.g., CPU usage by other processes), which can make it difficult to reproduce obtained results. To account for that, we implemented in our platform a termination criterion based on the number of basic operations realized during an execution. This mechanism consists in a regression model that converts a desired time limit into a maximum number of basic operations.

There are three basic operations that, together, dominate the running time of an algorithm. These operations are listed below, together with their time complexity:

1. Creating a new solution, which has complexity $O(n^2)$.
2. Flipping a variable of a solution, which has complexity $O(n)$.
3. Checking the change in the objective function caused by flipping a variable, which has complexity $O(1)$.

The time complexity of the first two operations is due to the update of internal data structures.

To build our regression model, we selected 10 algorithms and 25 instances and executed each algorithm on each instance 5 times with different seed values. The time limit of each run depends on the size and density of the instance, and for our set of 25 instances it varied from 150 seconds to 2250 seconds. During each run, we kept track of the amount of basic operations executed and, with this information, we built a plot associating running time to an estimate of the number of steps taken by the algorithm. Consider an instance f with n variables, an algorithm A, and a time limit t given in seconds. Also, let c_1 be the number new solutions created during the execution of A on f for t seconds (operation 1, in the list above), c_2 the number of flips (operation 2), and c_3 the number of times the change in the objective function caused by a flip was checked (operation 3). We consider that the number of steps s completed by algorithm A during its execution for t seconds is

$$s = c_1 n^2 + c_2 n + c_3 \tag{4.1}$$

We computed this quantity for each of the 1250 runs and plotted against the running time. As Figure 4.6 shows, there is a strong correlation between these two quantities. We used this correlation to build our regression model that, given a time limit, returns an estimate s' of the number steps. Our hypothesis is that $s \approx a \cdot t^b$, for real numbers a and b. We computed a linear regression of our hypothesis in the log-log space, i.e., $\log_{10}(s) \approx b \log_{10}(t) + \log_{10}(a)$, and obtained the following model:

$$s' = 10^9 t^{0.93} \tag{4.2}$$

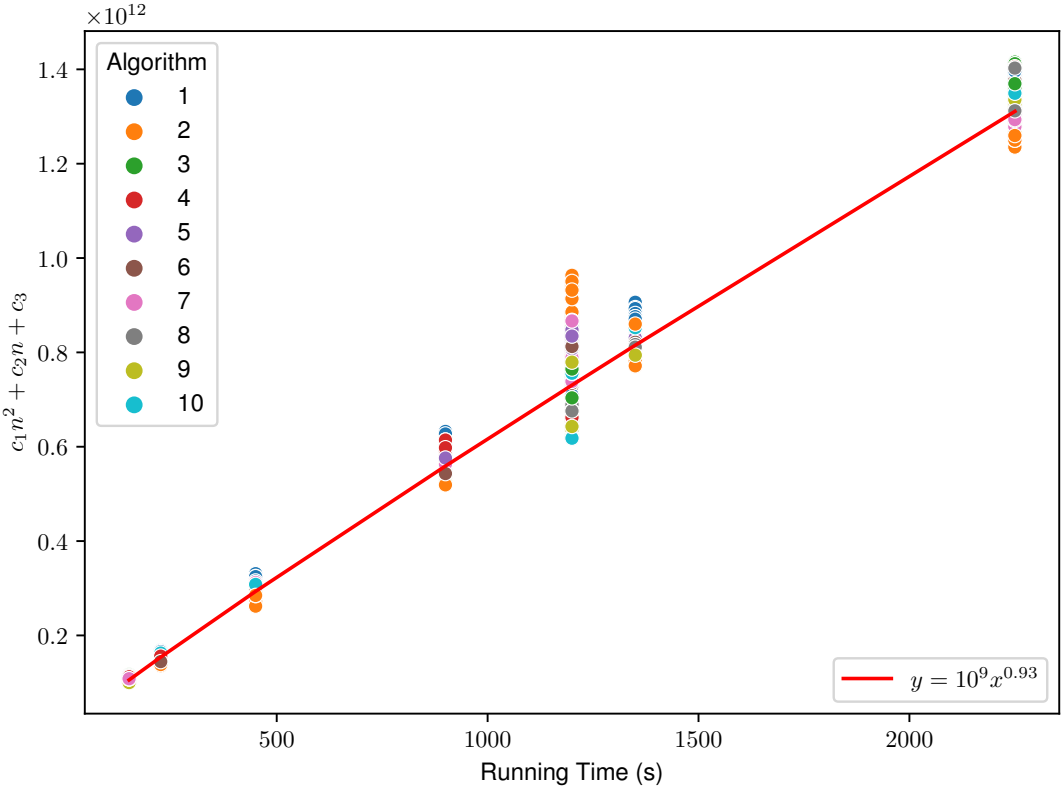

Figure 4.6 – Data collected in the 1250 runs and the regression curve. Each dot represents a combination of algorithm, instance and seed value. Dots are colored according to the algorithm they represent.

In summary, our platform allows the specification of a desired time limit, which is converted to a maximum number of steps completed by the algorithm according to Equation 4.2. Once the actual number of steps (as defined by Equation 4.1) exceeds the maximum number, execution is stopped.

A small experiment was conducted to validate the new termination criterion. We selected five of the 10 algorithms and executed them on 10 of the 25 instances mentioned above. We ran each algorithm on each instance using the termination

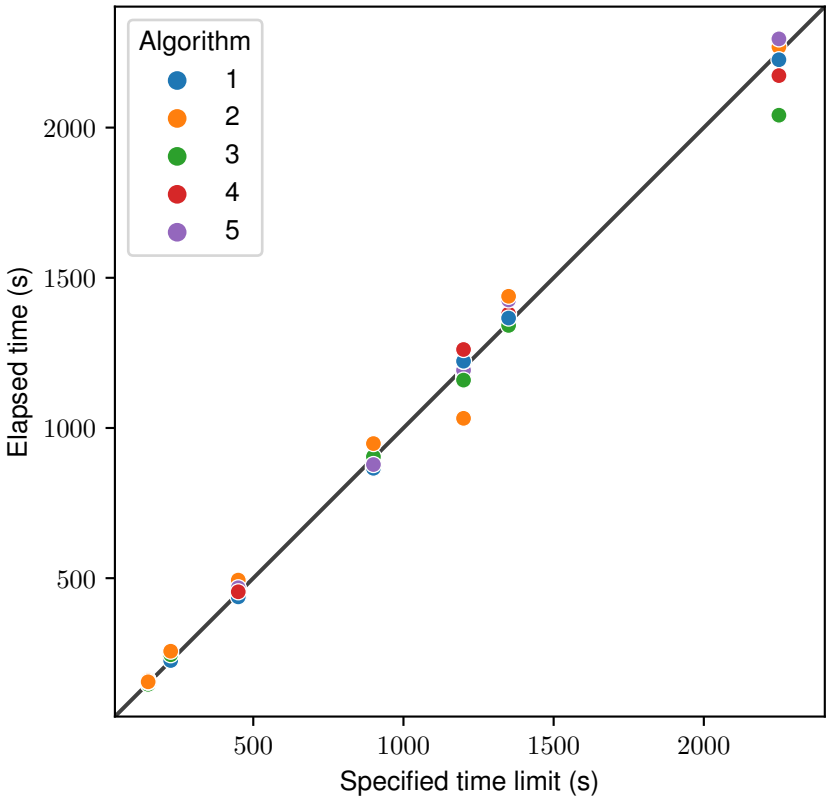

Figure 4.7 – Relation between the specified running time and the time taken to execute the estimated number of steps. Each dot represents a combination of algorithm and instance. Dots are colored according to the associated algorithm.

Table 4.2 – Average absolute and relative difference between the objective value obtained by both termination criteria across all 10 instances.

Algorithm	**Abs. Difference**	**Rel. Difference**
1	-32.8	-0.00009
2	0.2	0.00006
3	29.4	0.00024
4	-0.7	-0.00021
5	32.7	0.00006

criterion just described. The specified maximum running times were the same ones used to produce the data displayed in Figure 4.6.

Figure 4.7 shows how the specified time limit (x-axis) relates to the time spent executing the estimated number of steps (y-axis) for each of the 50 runs. As we can see, both times are about the same, which constitutes evidence that the regression model works and the overhead caused by the implementation is small.

It is also important to make sure that the new termination criterion does not deteriorate solution quality. To verify that, we executed again the same 50 combinations of algorithm and instance, but now using wall-clock time as the termination criterion. All 50 runs occurred in parallel, using 12 cores.

For each of the 50 combinations of algorithm and instance, let s_i denote the objective value obtained in the ith run when the termination criterion was the estimated number of steps. Similarly, let w_i be the equivalent quantity when the termination criterion was wall-clock time. Table 4.2 shows the average of the absolute difference, $w_i - s_i$, and the relative difference, $\frac{w_i - s_i}{s_i}$, grouped by algorithm (i.e., each line is the average of 10 values). As we can see, both termination criteria obtain similar results on average.

5 EXPERIMENTS

In this section we present some computational experience. All the experiments presented in the next sections were executed on a AMD Ryzen 9 3900X 12-Core Processor with 32 GB of available memory. Our platform was compiled using GCC 9.4.0 with the flag -O3.

5.1 Instances

All the instances used in our experiments were created using the generator described by Palubeckis (2006). They have sizes ranging from 3000 to 7000 variables, densities varying from 50% to 100%, and integer coefficients sampled uniformly from the interval $[-100, 100]$.

Table 5.1 – Composition of the test set regarding instance size.

Instance Size	**Number of Instances**
3000	5
4000	5
5000	5
6000	3
7000	3

Table 5.2 – Composition of the training and validation sets regarding instance size.

Instance Size	**Number of Instances**
3000	4
4000	4
5000	8
6000	8
7000	8

As is common in the machine learning literature, we splitted our instances into three sets: training, validation and test. Our test set consists of 21 instances commonly used in the literature to benchmark heuristics for QUBO (PALUBECKIS, 2006) (SOUZA; RITT, 2018). Its composition regarding instance sizes is depicted in Table 5.1. Our training and validation sets have 32 instances each and share the same composition regarding instance sizes. Table 5.2 illustrates the composition. More importantly, all three sets are disjoint.

The composition of the test set differs from the other two sets because in previous experiments we noticed that algorithms found by irace consistently performed worse than state-of-the-art algorithms on bigger instances. We hope that by increasing the number of large instances in the training phase, algorithms with this problem can be more easily identified and discarded by the AC.

In all experiments the termination criterion is running time as measured by a wall clock (i.e., we did not use the feature described in Section 4.3), and the maximum running time is determined by the size of the instance. Table 5.3 presents the maximum running time in seconds for each instance size.

Table 5.3 – Time limit for each instance size.

Instance Size	**Maximum Running Time (s)**
3000	150
4000	300
5000	600
6000	900
7000	1500

5.2 Results

In the next three sections, we present experiments to validate our work. Section 5.2.1 presents computational experience to assess the performance of our platform. In Section 5.2.2 we present the results of the exploration of our design space using irace. Lastly, in Section 5.2.3 we analyze the algorithms found in Section 5.2.2.

Table 5.4 – Selected algorithms used in the comparison between our platform and MQLib.

Algorithm	MQLib	Our Platform
Wang et al. (2012)	-	*wang*
Palubeckis (2006)	*PAL06*	*p06*
Glover, Lü and Hao (2010)	*GLO10*	*d2ts*
Palubeckis (2004)	*PAL04T2*	-
Palubeckis (2004)	*PAL04T3*	-

5.2.1 Performance

To make sure the algorithms in our platform are competitive to what is best in the literature, we make some comparisons against MQLib[1], a platform that implements several heuristics for the Max-cut and QUBO problems. MQLib is a product of the work of Dunning, Gupta and Silberholz (2018), where they make a large-scale study comparing the performance of 37 heuristics evaluated on 3296 instances from different domains. We selected four heuristics that had top performance on instances created using the generator mentioned above and compared them against three state-of-the-art algorithms that can be instantiated in our platform.

Table 5.4 shows which algorithms were selected. The first column shows the work who proposed the algorithm, the second column indicates how the algorithm is identified by MQLib, and the third column shows how the algorithm is identified in our platform. MQLib does not provide an implementation of the algorithm proposed by Wang et al. (2012). Similarly, our platform is not able to instantiate exactly the algorithms proposed by Palubeckis (2004).

We executed each of the seven algorithms on five instances five times (using different seeds). The five instances we used were sampled from the test set, one for each instance size. For each execution, we recorded the best objective value obtained during the execution and the time taken to obtain it. All 175 executions happened in parallel in 12 cores.

[1] <https://github.com/MQLib/MQLib>

Table 5.5 – Absolute gap of selected algorithms.

Algorithm	**Average Gap**
wang	52
d2ts	80.56
p06	99.32
PAL06	124.16
PAL04T2	262.6
GLO10	789.52
PAL04T3	2266.6

Given the objective value $objv$ obtained by an algorithm on an instance i, the absolute gap is the difference between the best known value of instance i and $objv$. The absolute gap can be viewed as a measure of solution quality, and we use it here to compare the quality of solutions found by our platform to MQLib's. In Table 5.5, the average absolute gap obtained by each algorithm on its 25 executions is displayed. We can see that the first three smallest absolute gaps are from algorithms of our platform. Not only that, but algorithms *p06* and *d2ts* have better results than their counterparts implemented in MQLib.

Table 5.6 – Time to best of selected algorithms.

Algorithm	**Average Time to Best**
PAL06	202.7
p06	210.4
PAL04T2	247.7
d2ts	251.3
wang	268.4
GLO10	325.6
PAL04T3	363.6

We also evaluate how fast, on average, an algorithm takes to arrive at the best solution it could find. Table 5.6 shows these values. The average time taken for algorithms in our platform to arrive at the best solution are very similar to what we observe in MQLib's algorithms. In particular, MQLib's *PAL06* tends to arrive

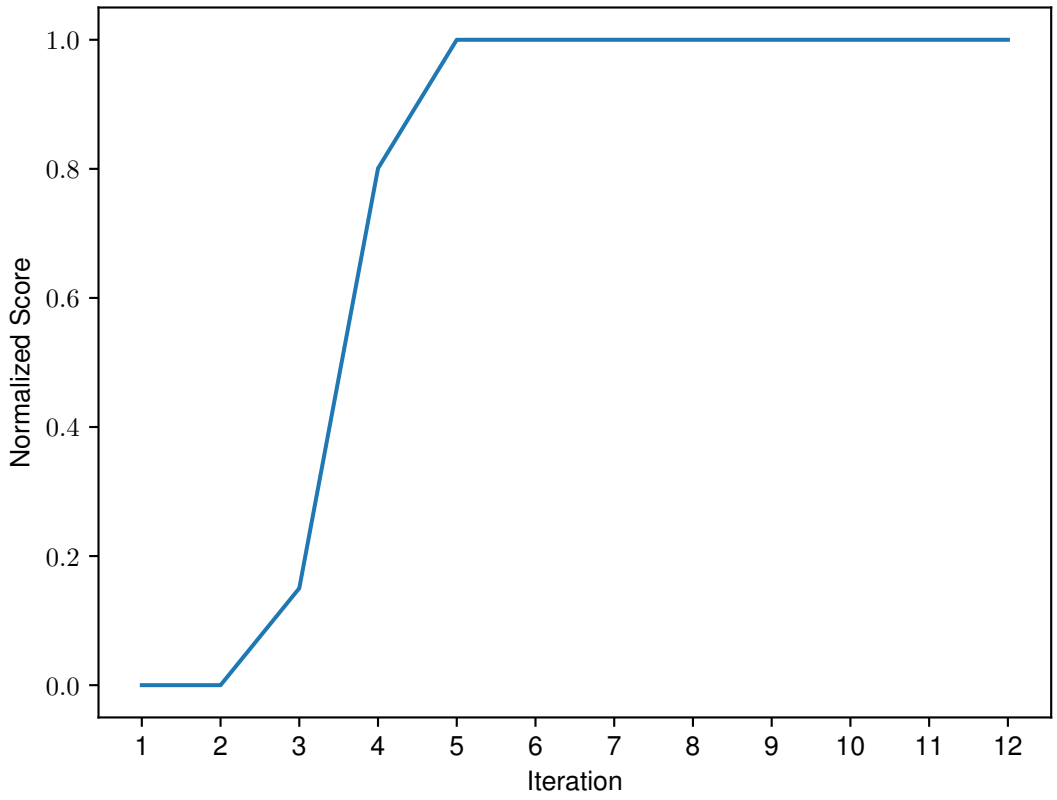

Figure 5.1 – Performance evolution of configurations found by irace.

at the best solution faster than its counterpart in our platform. On the other hand, we observe the opposite behavior when we consider *GLO10* and *d2ts*.

In summary, the results above are good evidence that our platform is at least efficient enough to produce results comparable to the literature.

5.2.2 Optimization Process

Using the parametric representation described in Chapter 4, we search the space of algorithms using irace. We ran irace with the training set mentioned in Section 5.1 and a budget of 7000 runs[2]. It took 6 days and 17 hours to complete its execution using 12 cores, and a total of 12 iterations were performed.

Before evaluating the configurations found by irace on the test set, we perform some analyses to check for a possible overfitting on the training data. To do that, when irace finished its execution we collected all elite configurations returned at

[2]The term *budget* refers to the maximum number of algorithm executions irace is allowed to make.

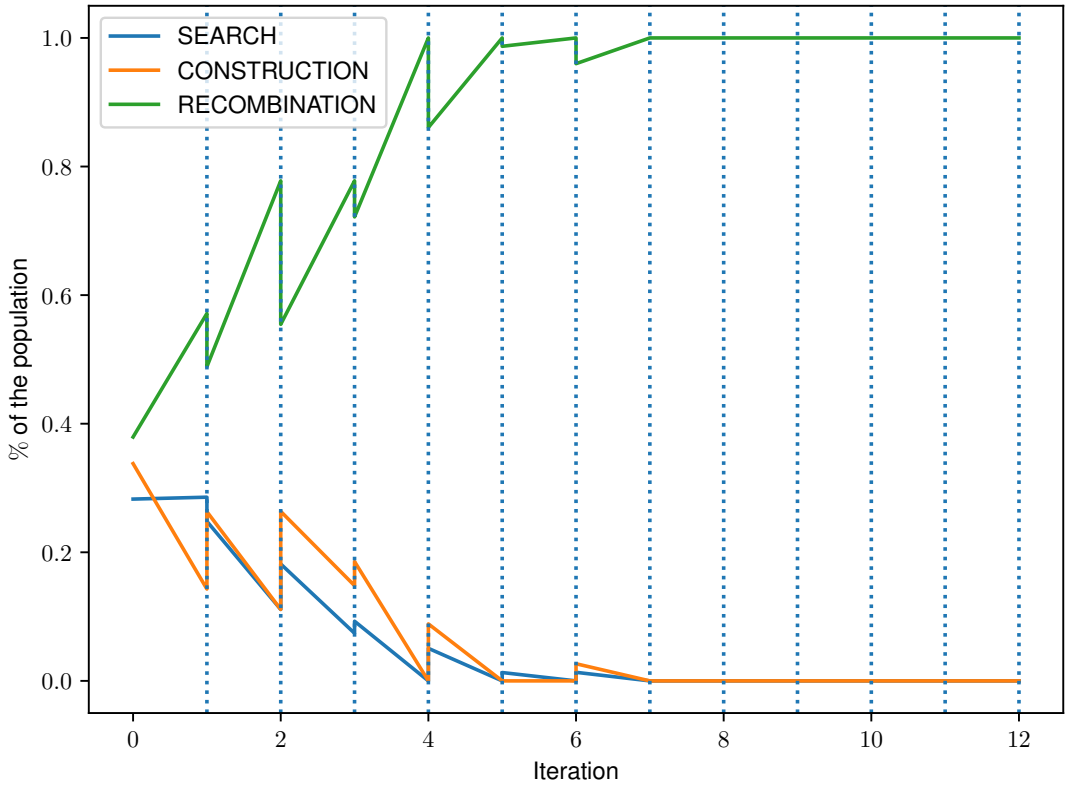

Figure 5.2 – Composition of the population of configurations maintained by irace along the 12 iterations.

the end of each iteration and evaluated them on the validation set, replicating each execution 3 times. Since irace produced 30 elite configurations along its execution, this experiment resulted in 2880 runs, which were executed in parallel using 12 cores. Using the data of this experiment, we associated to each elite configuration a score: the average objective value obtained considering all 96 executions (32 instances times 3 replications).

Figure 5.1 shows the score, mapped to the interval $[0, 1]$, obtained by the best elite configuration of each iteration. As we can see, the population of elite configurations maintained by irace consistently obtain better scores as the iteration count increases. This trend, however, stops at iteration 5, and from this point on the score value stagnates. One possible reason for this behavior is the homogeneity of the population of configurations. We expand on this hypothesis in the next paragraphs.

Configurations can be classified according to the type of algorithm they represent. As we can see by looking at the grammar (Figure 3.1), the top level rule defines three types of algorithms, i.e., search, construction, and recombination. Figure 5.2 shows the composition of the population of configurations at the beginning and at the end of each iteration. Each line shows how much of the population is

composed by algorithms of a certain type. Consider the green line, for example. At Iteration=1, one point shows how much of the population at the end of iteration 1 is composed of recombination algorithms. The other point shows the same information, but now about the population at the beginning of iteration 2. Remember that, as it was explained in Section 2.3.1, at the start of each iteration irace samples new configurations and adds them to the population of elite configurations. Between the start and the end of an iteration the population is filtered through racing.

As we can see, at the beginning of the first iteration the population has approximately the same number of configurations of each type, due to the random sampling. At the end of the first iteration, however, almost 60% of the population is composed of recombination algorithms. This means that the racing process ended up eliminating several configurations of type SEARCH and CONSTRUCTION. At the beginning of iteration 2 the proportion of search and construction algorithms increases a bit because of the new sampled configurations, but most of these configurations are again eliminated by racing as the point representing the end of iteration 2 shows. This trend causes the sampling process to be biased towards recombination algorithms and, as a result, from iteration 5 on the population is composed almost entirely of recombination algorithms. A population with less diversity may help to explain the stagnation in performance on the validation set.

We now turn our attention to Table 5.7, which displays some information about the 30 elite configurations. The first column gives an identifier for the algorithm (each configuration represents an algorithm), the second column shows at which iteration the algorithm was first sampled, the third column shows the rank the algorithm obtained in the validation set, and the fourth column shows how the algorithm was ranked by irace (algorithms that are not part of the final set of elites do not receive a rank). Lastly, the last column shows the type of the algorithm.

By looking at the last column we can see that almost all elite configurations are recombination algorithms. As a matter of fact, just the first two iterations produced some elite configurations that are search or construction algorithms, and recombination algorithms represent approximately 86% of all elite configurations. This is expected, considering the composition of the population of configurations maintained by irace depicted in Figure 5.2.

By looking at the second column, we can see that few of the 30 elite configurations were sampled at the end of the search. The histogram in Figure 5.3

Table 5.7 – Elite configurations found by irace.

Algorithm	First Iteration	Rank on Validation Set	irace Rank	Algorithm Type
i527	5	1	3	RECOMBINATION
i447	4	2	2	RECOMBINATION
i345	3	3	-	RECOMBINATION
i57	1	4	-	RECOMBINATION
i744	9	5	5	RECOMBINATION
i383	4	6	-	RECOMBINATION
i490	5	7	-	RECOMBINATION
i454	4	8	-	RECOMBINATION
i24	1	9	-	RECOMBINATION
i250	2	10	-	RECOMBINATION
i517	5	11	4	RECOMBINATION
i717	8	12	1	RECOMBINATION
i371	3	13	-	RECOMBINATION
i325	3	14	-	RECOMBINATION
i508	5	15	-	RECOMBINATION
i776	11	16	6	RECOMBINATION
i33	1	17	-	RECOMBINATION
i163	2	18	-	RECOMBINATION
i422	4	19	8	RECOMBINATION
i234	2	20	-	RECOMBINATION
i568	6	21	-	RECOMBINATION
i533	5	22	7	RECOMBINATION
i203	2	23	-	CONSTRUCTION
i136	1	24	-	CONSTRUCTION
i365	3	25	-	RECOMBINATION
i75	1	26	-	RECOMBINATION
i167	2	27	-	RECOMBINATION
i12	1	28	-	SEARCH
i225	2	29	-	SEARCH
i129	1	30	-	RECOMBINATION

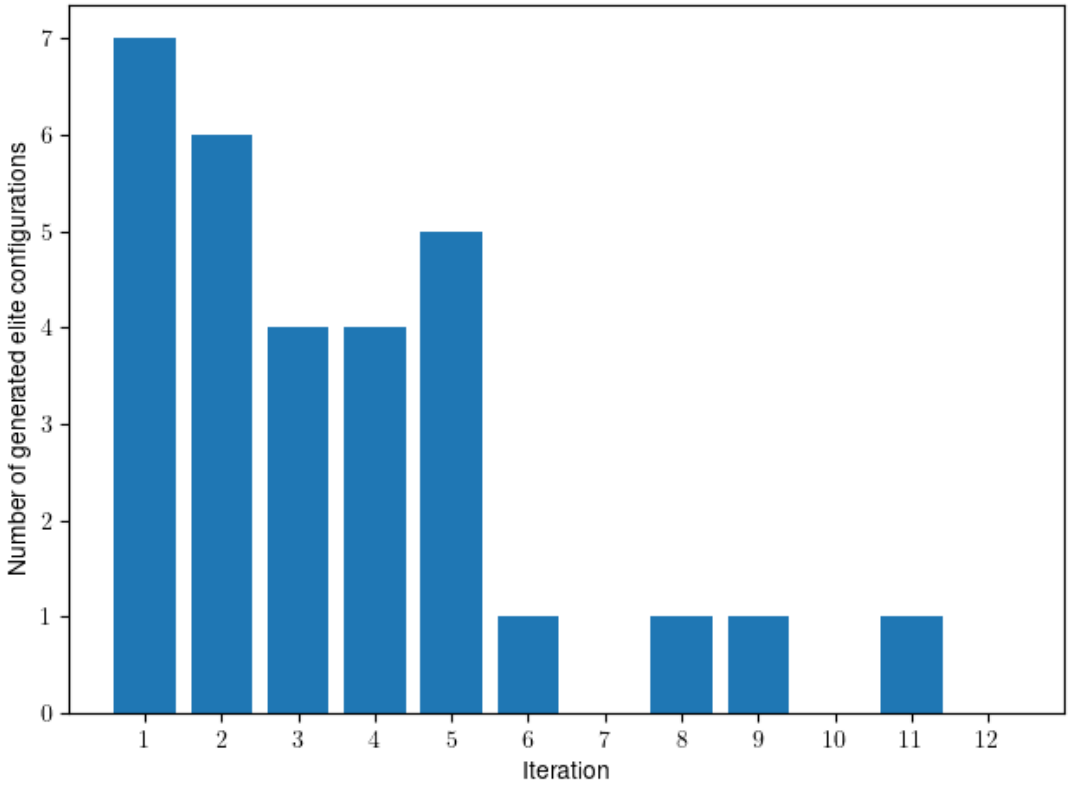

Figure 5.3 – Distribution of elite configurations considering the iteration in which they were sampled.

illustrates this behavior. From iteration 6 to 12, only 4 new elite configurations were sampled, which means that the majority of the high-quality solutions found by irace was sampled between iterations 1 and 5.

In summary, our study indicates that the quality of the configurations found by irace did not deteriorate over time due to overfitting. The stagnation on the score value, however, may be a signal that a budget as high as 7000 does not necessarily result in better configurations.

5.2.3 Benchmark

In this section we compare some of the algorithms found by irace with state-of-the-art algorithms that can be instantiated in our platform. We selected the first five best algorithms as ranked by irace and the first five algorithms with best performance on the validation (see Table 5.7). It turns out that there is an overlap between these two sets of algorithms, so we end up with 7 algorithms.

In addition to these seven algorithms, we also consider the three algorithms

used in Section 5.2.1, namely *wang*, *d2ts*, and *p06*, and the best algorithm found by Souza and Ritt (2018), which here we name *hhpal*. We executed the 11 algorithms on all 21 test instances 15 times in parallel using 12 cores. In order to compare the heuristics, we consider the same metrics used in the large-scale study of Dunning, Gupta and Silberholz (2018), which we explain below. Let h be any of the 11 considered heuristics:

- **First-equal percentage**: is the percentage of instances for which the mean objective value obtained by h across the 15 replicates was no worse than the mean solution of any of the other 10 heuristics.
- **First-strict percentage**: is the percentage of instances for which the mean objective value obtained by h across the 15 replicates was strictly better than the mean solution of any of the other 10 heuristics.
- **Best achieved percentage**: is the percentage of instances for which h achieved the best-known solution in at least one replicate.
- **Worst-of-15 deviation**: let w_i^h denote the worst objective value obtained by h on instance i among the 15 replicates divided by the best value obtained by any heuristic on i. The value of the metric is the average of w_i^h across all test instances i.
- **Mean-of-15 deviation**: let m_i^h denote the mean objective value obtained by h on instance i among the 15 replicates divided by the best value obtained by any heuristic on i. The value of the metric is the average of m_i^h across all test instances i.
- **Best-of-15 deviation**: let b_i^h denote the best objective value obtained by h on instance i among the 15 replicates divided by the best value obtained by any heuristic on i. The value of the metric is the average of b_i^h across all test instances i.
- **Average rank**: let r_i^h be the rank heuristic h obtained on instance i among all heuristics according to the mean value obtained across the 15 replicates (in the case of ties, all tied heuristics are given the minimum rank). Rank 1 indicates the best mean performance on instance i, while rank 11 indicates the worst performance. The value of this metric is the average of r_i^h across all test instances i.

Table 5.8 shows the value that each considered algorithm obtained for each metric.

The rows are ordered according to the values of the columns, from left to right.

By looking at the last column we can see that six out of the seven algorithms with better average rank were found by irace, including the top scorer in this metric. The **Best-Achieved** column shows that all algorithms found by irace obtained the best known value in at least one replicate in all instances. In particular, algorithm i447 is very competitive since, as the **First-Equal** column indicates, it obtained the best mean value across all 15 replicates in 76.2% of the 21 instances. The picture changes a bit when we analyze the **Worst-of-15 Deviation** column. The best heuristic according to this metric is *hhpal*, followed by *d2ts*, i447, *p06*, and *wang*. This means that, although algorithms found by irace are usually competitive (as indicated by the first column), on some fraction of the instances their performance is notably bad.

Table 5.8 – Comparison between state-of-the-art algorithms and the algorithms found by irace.

Heuristic	First-Equal (%)	First-Strict (%)	Best-Achieved (%)	Worst-of-15 Deviation (%)	Mean-of-15 Deviation (%)	Best-of-15 Deviation (%)	Average Rank
i447	76.2	9.5	100	99.998586	99.999615	100	2.23
hhpal	76.2	4.8	100	99.999188	99.999657	100	2.66
i345	66.7	4.8	100	99.998366	99.999303	100	3.09
i527	66.7	0	100	99.998153	99.999490	100	2.61
wang	61.9	0	100	99.998287	99.999509	100	3.61
i517	61.9	0	100	99.997517	99.999477	100	3.14
i717	61.9	0	100	99.997485	99.999330	100	3.52
i744	61.9	0	100	99.997400	99.999142	100	4.14
p06	61.9	0	95.2	99.997510	99.999551	99.9	4.04
d2ts	52.4	0	100	99.9988	99.999620	100	3.76
i57	52.4	0	100	99.997727	99.999456	100	3.47

6 CONCLUSIONS

In this work we studied an automated approach to the design of algorithms for the Quadratic Unconstrained Binary Optimization Problem. We extend the work of Souza and Ritt (2018), where they use a context-free grammar to encode a design space of algorithms that is explored by an algorithm configurator.

In Section 3, we introduced an extension of the grammar proposed by Souza and Ritt (2018), with new components and other modifications aiming to help the algorithm configurator to spend more time on promising algorithms. We also proposed a parametric representation for this new grammar, which was explained in Section 3.3. To support our work, we developed a platform for design space exploration of heuristics to QUBO. Our platform comprises 28 algorithmic components and is easy to extend with new components. Besides the modular architecture, its main features are a fast technique to evaluate the value of flipping variables and the possibility of terminating executions based on the number of executed operations, facilitating the production of reproducible research. In computational experiments we showed that algorithms instantiated in our platform are as efficient as standard implementations of the literature.

Using our platform and irace, we explored the design space encode by the grammar. Our results show a great dominance of recombination algorithms over search and constructions algorithms in all 12 iterations completed by irace. We performed experiments to check for a possible deterioration in performance due to overfitting, but our results indicate that this did not happen. Additionally, we observed that the optimization process converged quite quickly to a few high-quality solutions, which indicates that a smaller budget could achieve similar results.

Finally, we compared the best algorithms found by irace to state-of-the-art algorithms that can be instantiated in our platform using seven performance metrics. As our experiments show, algorithms found by irace are, if not better, at least competitive to the state-of-the-art algorithms.

As future work, we would like to use the insights presented in Section 5.2.2 to make the configuration process more efficient. As we stated before, there is a strong convergence toward recombination algorithms, but search algorithms are the majority between the state-of-the-art approaches we analyzed. Hence, why irace does not spend more time on the space of search algorithms is still an open

question. We would also like to define a new design space that generalizes the neighborhood size on which most algorithms operate. It is known that working on large neighborhoods is computationally expensive, but new works on closed-form formulas for evaluating r-flips may turn such explorations possible in practice (see Wang (2022), for example).

REFERENCES

ALIDAEE, B.; KOCHENBERGER, G.; WANG, H. Theorems supporting r-flip search for pseudo-boolean optimization. **International Journal of Applied Metaheuristic Computing**, v. 1, n. 1, p. 93–109, January 2010.

ANSÓTEGUI, C.; SELLMANN, M.; TIERNEY, K. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In: GENT, I. P. (Ed.). **Principles and Practice of Constraint Programming - CP 2009**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 142–157. ISBN 978-3-642-04244-7.

BOROS, E.; HAMMER, P. L.; TAVARES, G. Local search heuristics for quadratic unconstrained binary optimization (QUBO). **Journal of Heuristics**, v. 13, n. 2, p. 99–132, February 2007.

CASEAU, Y.; SILVERSTEIN, G.; LABURTHE, F. Learning hybrid algorithms for vehicle routing problems. **Theory and Practice of Logic Programming**, v. 1, p. 779–806, June 2004.

CONOVER, W. J. **Practical Nonparametric Statistics**. 3. ed. Nashville, TN: John Wiley & Sons, 1998. (Wiley Series in Probability and Statistics).

DELAZERI, G.; RITT, M. Fast heuristics for traveling salesman problems with multiple flying sidekicks. In: **2021 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.: s.n.], 2021. p. 1365–1371.

DELAZERI, G.; RITT, M.; SOUZA, M. de. Comparing Surrogate Models for Tuning Optimization Algorithms. In: **Learning and Intelligent Optimization: 16th International Conference, LION 16, Milos Island, Greece, June 5–10, 2022, Revised Selected Papers**. Berlin, Heidelberg: Springer-Verlag, 2023. p. 347–360. ISBN 978-3-031-24865-8. Available from Internet: <https://doi.org/10.1007/978-3-031-24866-5_26>.

DREO, J. et al. Paradiseo: From a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of paradiseo. In: CHICANO, F. (Ed.). **Proceedings of the Genetic and Evolutionary Computation Conference Companion**. New York, NY, USA: Association for Computing Machinery, 2021. p. 1522–1530. ISBN 9781450383516.

DUNNING, I.; GUPTA, S.; SILBERHOLZ, J. What Works Best When? A Systematic Evaluation of Heuristics for Max-Cut and QUBO. **INFORMS Journal on Computing**, Institute for Operations Research and the Management Sciences (INFORMS), v. 30, n. 3, p. 608–624, August 2018.

FESTA, P. et al. Randomized heuristics for the max-cut problem. **Optimization Methods and Software**, Informa UK Limited, v. 17, n. 6, p. 1033–1058, January 2002.

GLOVER, F.; Lü, Z.; HAO, J.-K. Diversification-driven tabu search for unconstrained binary quadratic problems. **4OR**, Springer Science and Business Media LLC, v. 8, n. 3, p. 239–253, Janurary 2010.

HANSEN, P.; MLADENOVIĆ, N. Variable neighborhood search: Principles and applications. **European Journal of Operational Research**, v. 130, n. 3, p. 449–467, May 2001.

HOOS, H. H.; STÜTZLE, T. **Stochastic Local Search**. 1. ed. San Francisco: Morgan Kaufmann, 2005. ISBN 978-1-55860-872-6.

HUTTER, F. et al. ParamILS: An Automatic Algorithm Configuration Framework. **Journal of Artificial Intelligence Research**, v. 36, p. 267–306, October 2009.

JOHNSON, D. A theoretician's guide to the experimental analysis of algorithms. In: GOLDWASSER, M.; JOHNSON DAVID ANDMCGEOCH, C. (Ed.). **Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges**. Providence, RI: American Mathematical Society, 2002. v. 59, p. 215–250. ISBN 978-1-4704-4017-6.

KATAYAMA, K.; NARIHISA, H. Performance of simulated annealing-based heuristic for the unconstrained binary quadratic programming problem. **European Journal of Operational Research**, v. 134, n. 1, p. 103–119, Octorber 2001.

KENDALL, G. et al. Good Laboratory Practice for optimization research. **Journal of the Operational Research Society**, v. 67, n. 4, p. 676–689, April 2016.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. **Science**, American Association for the Advancement of Science (AAAS), v. 220, n. 4598, p. 671–680, may 1983.

KOCHENBERGER, G. et al. The unconstrained binary quadratic programming problem: a survey. **Journal of Combinatorial Optimization**, v. 28, n. 1, p. 58–81, July 2014. ISSN 1573-2886.

LINDAUER, M. et al. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. **Journal of Machine Learning Research**, v. 23, n. 54, p. 1–9, 2022. Available from Internet: <http://jmlr.org/papers/v23/21-0888.html>.

LODI, A.; ALLEMAND, K.; LIEBLING, T. M. An evolutionary heuristic for quadratic 0–1 programming. **European Journal of Operational Research**, v. 119, n. 3, p. 662–670, December 1999.

LÓPEZ-IBÁÑEZ, M. et al. The irace package: Iterated racing for automatic algorithm configuration. **Operations Research Perspectives**, v. 3, p. 43–58, 2016.

LÓPEZ-IBÁÑEZ, M.; MARMION, M.-E.; STÜTZLE, T. Technical Report, **Automatic Design of Hybrid Metaheuristics from Algorithmic Components**. 2017. Available from Internet: <https://iridia.ulb.ac.be/IridiaTrSeries/link/IridiaTr2017-012.pdf>.

MARMION, M.-E. et al. Automatic Design of Hybrid Stochastic Local Search Algorithms. In: BLESA, M. J. et al. (Ed.). **Hybrid Metaheuristics**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 144–158. ISBN 978-3-642-38516-2.

MASCIA, F. et al. From Grammars to Parameters: Automatic Iterated Greedy Design for the Permutation Flow-Shop Problem with Weighted Tardiness. In: NICOSIA, G.; PARDALOS, P. (Ed.). **Learning and Intelligent Optimization**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 321–334. ISBN 978-3-642-44973-4.

MASCIA, F. et al. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. **Computers & Operations Research**, v. 51, p. 190–199, 2014. ISSN 0305-0548.

MILNE, A.; ROUNDS, M.; GODDARD, P. White paper, **Optimal feature selection in credit scoring and classification using a quantum annealer**. 2017.

NEUKART, F. et al. Traffic flow optimization using a quantum annealer. **Frontiers in ICT**, Frontiers Media SA, v. 4, dec. 2017.

OHZEKI, M. et al. Control of automated guided vehicles without collision by quantum annealer and digital devices. **Frontiers in Computer Science**, Frontiers Media SA, v. 1, nov. 2019.

PALUBECKIS, G. Multistart Tabu Search Strategies for the Unconstrained Binary Quadratic Optimization Problem. **Annals of Operations Research**, Springer Science and Business Media LLC, v. 131, n. 1-4, p. 259–282, October 2004.

PALUBECKIS, G. Iterated Tabu Search for the Unconstrained Binary Quadratic Optimization Problem. **Informatica**, Vilnius University Press, v. 17, n. 2, p. 279–296, January 2006.

PUNNEN, A. P. (Ed.). **The Quadratic Unconstrained Binary Optimization Problem**. 1. ed. Switzerland: Springer International Publishing, 2022. ISBN 978-3-031-04519-6.

RESENDE, M. G. C.; RIBEIRO, C. C. Greedy Randomized Adaptive Search Procedures. In: GLOVER, F.; KOCHENBERGER, G. A. (Ed.). **Handbook of Metaheuristics**. Boston, MA: Springer US, 2003. p. 219–249. ISBN 978-0-306-48056-0.

ROSENBERG, G. White paper, **Finding optimal arbitrage opportunities using a quantum annealer**. 2016.

SOUZA, M. de; RITT, M. Automatic Grammar-Based Design of Heuristic Algorithms for Unconstrained Binary Quadratic Programming. In: LIEFOOGHE, A.; LÓPEZ-IBÁÑEZ, M. (Ed.). **Evolutionary Computation in Combinatorial Optimization**. Cham: Springer International Publishing, 2018. p. 67–84. ISBN 978-3-319-77449-7.

WANG, H. New results on closed-form formulas for evaluating r-flip moves in quadratic unconstrained binary optimization. **SSRN Electronic Journal**, 2022.

WANG, Y. et al. Backbone guided tabu search for solving the UBQP problem. **Journal of Heuristics**, v. 19, n. 4, p. 679–695, March 2011.

WANG, Y. et al. Path relinking for unconstrained binary quadratic programming. **European Journal of Operational Research**, v. 223, n. 3, p. 595–604, December 2012.

APPENDIX A — COMPONENTS

Data: Initial solution x

Result: Best found solution x^*

$x^* \leftarrow x$;

while *Termination criteria* **do**

  $x \leftarrow \text{pert}(x)$;

  $x \leftarrow \text{intensification}(x)$;

  if $f(x) < f(x^*)$ **then**

    $x^* \leftarrow x$;

  end

end

Algorithm 1: ILS

Data: Initial solution x

Result: Best found solution x^*

$x^* \leftarrow x$;

$E \leftarrow \emptyset$;

while $|E| \neq e$ **do**

  $x \leftarrow \text{intensification}(x)$;

  $E = E \cup \{x\}$;

  $x =$ generate random solution;

end

while *Termination criteria* **do**

  $x \leftarrow$ select random element from E;

  $x \leftarrow \text{pert}(x)$;

  $x \leftarrow \text{intensification}(x)$;

  $x^w \leftarrow$ worst solution in E;

  if $f(x) < f(x^w)\, and\, x \notin E$ **then**

    $E = E \cup \{x\} \setminus \{x^w\}$;

  end

  if $f(x) < f(x^*)$ **then**

    $x^* \leftarrow x$;

  end

end

Algorithm 2: ILSE

Data: Initial solution x

Result: Best found solution x^*

$x^* \leftarrow x$;

while *Termination Criteria* **do**

  $x \leftarrow \text{pert}(x)$;

  $x' \leftarrow x$;

  $k \leftarrow 1$;

  while $k \leq k_{max}$ **do**

    $\Delta \leftarrow$ randomly select k distinct integers between 1 and N;

    $x'_i \leftarrow 1 - x'_i, i \in \Delta$;

    $x' \leftarrow \text{intensification}(x')$;

    if $f(x') < f(x)$ **then**

      $x \leftarrow x'$;

      $k \leftarrow 1$;

      if $f(x') < f(x)$ **then**

        $x \leftarrow x'$;

        if $f(x) < f(x^*)$ **then**

          $x^* \leftarrow x$;

        end

      end

    else

      $x' \leftarrow x$;

      $k \leftarrow k + 1$;

    end

  end

end

Algorithm 3: VNS

```
Data: Initial solution x
Result: Best found solution x*
x* ← x;
E ← ∅;
while Termination Criteria do
    x ← pert(x);
    x' ← x;
    k ← 1;
    while k ≤ k_max do
        Δ ← randomly select k distinct integers between 1 and N;
        x'_i ← 1 − x'_i, i ∈ Δ;
        x' ← intensification(x');
        if f(x') < f(x) then
            x ← x';
            k ← 1;
            if f(x') < f(x) then
                x ← x';
                if f(x) < f(x*) then
                    x* ← x;
                end
            end
        else
            x' ← x;
            k ← k + 1;
        end
    end
    if |E| = 0 then
        E ← E ∪ {x};
    else
        x' ← select random element from E;
        x ← PathRelinking(x, x');
        x^w ← worst solution in E;
        if f(x) < f(x^w) and x ∉ E then
            E = E ∪ {x} \ {x^w};
        end
        if f(x) < f(x*) then
            x* ← x;
        end
    end
end
```

Algorithm 4: VNSPR

Data: Initial solution x
Result: Best found solution x^*
$x^* \leftarrow x$;
while *Termination criteria* **do**
    $x \leftarrow \text{constructor}(x)$;
    $x \leftarrow \text{intensification}(x)$;
    if $f(x) < f(x^*)$ **then**
        $x^* \leftarrow x$;
    end
end

Algorithm 5: GRASP

Data: Initial solution x

Result: Best found solution x^*

$x^* \leftarrow x$;

$E \leftarrow \emptyset$;

while *Termination criteria* **do**

  $x \leftarrow \text{constructor}(x)$;

  $x \leftarrow \text{intensification}(x)$;

  if $f(x) < f(x^*)$ **then**

    $x^* \leftarrow x$;

  end

end

if $|E| = 0$ **then**

  $E \leftarrow E \cup \{x\}$;

else

  $x' \leftarrow$ select random element from E;

  $x \leftarrow \text{PathRelinking}(x, x')$;

  $x^w \leftarrow$ worst solution in E;

  if $f(x) < f(x^w)\, and x \notin E$ **then**

    $E = E \cup \{x\} \setminus \{x^w\}$;

  end

  if $f(x) < f(x^*)$ **then**

    $x^* \leftarrow x$;

  end

end

Algorithm 6: GRASPPR

```
Data: Initial solution x
Result: Best found solution x*
x* ← x;
while Termination criteria do
    E ← empty array;
    while E.size ≠ b do
        x = generate random solution;
        x ← intensification(x);
        if x not in E then
            Append x to E
        end
    end
    tag(i) ← TRUE, i ∈ [N];
    novel ← TRUE;
    while novel do
        novel ← FALSE;
        C ← E;
        I ← ∅;
        for i, j ∈ [N], i < j and (tag(i) or tag(i)) do
            x ← PathRelinking(C(i), C(j));
            x ← intensification(x);
            w ← index of worst solution in E;
            if f(x) < f(E(w)) and x not in E then
                E(w) ← x;
                novel ← TRUE;
                I ← I ∪ {w};
            end
            if f(x) < f(x*) then
                x* ← x;
            end
            x ← PathRelinking(C(j), C(i));
            x ← intensification(x);
            w ← index of worst solution in E;
            if f(x) < f(E(w)) and x not in E then
                E(w) = x;
                novel ← TRUE;
                I ← I ∪ {w};
            end
            if f(x) < f(x*) then
                x* ← x;
            end
        end
        tag(i) ← FALSE, i ∈ [N] \ I;
        tag(i) ← TRUE, i ∈ I;
    end
end
```

Algorithm 7: RER

Data: Initial solution x

Result: Best found solution x^*

$x^* \leftarrow x$;

$change \leftarrow TRUE$;

while *change* **do**

    $i \leftarrow \text{improvement}(x)$;

    $x_i \leftarrow 1 - x_i$;

    if $f(x) < f(x^*)$ **then**

        $x^* \leftarrow x$;

    else

        $change \leftarrow FALSE$;

    end

end

Algorithm 8: LS

Data: Initial solution x
Result: Best found solution x^*
$x^* \leftarrow x$;
$change \leftarrow TRUE$;
while $stagnation < smax$ *and* $change$ **do**
    $rand \leftarrow FALSE$;
    if $rand\_real(0,1) < p$ **then**
        $i \leftarrow rand\_int(1, N)$;
        $rand \leftarrow TRUE$;
    else
        $i \leftarrow$ improvement(x);
    end
    $x_i \leftarrow 1 - x_i$;
    if $f(x) < f(x^*)$ **then**
        $x^* \leftarrow x$;
    else
        $st \leftarrow st + 1$;
        if $rand$ **then**
            $change \leftarrow FALSE$;
        end
    end
end

Algorithm 9: NMLS