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

JORDI PUJOL RICARTE

# Automatic Configuration of Generalized Constructive Heuristics

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

Advisor: Prof. Dr. Marcus R. P. Ritt

Porto Alegre
January 2025

*“So Long, and Thanks for All the Fish”*

— DOUGLAS ADAMS

## AGRADECIMENTOS

Agredeço a minha família, que sempre me incentivou a conhecer mais sobre o mundo e me motiva a ser uma pessoa melhor a cada dia. Também agradeço os amigos que estiveram comigo durante os últimos anos e me deram suporte nos momentos mais difíceis, especialmente àqueles que me apoiaram direta ou indiretamente durante a escrita desse trabalho. Por fim, agradeço a Juliana Ferraro (*in memoriam*) por me incentivar a fazer o curso de Ciência da Computação na UFRGS, mesmo com todas as dificuldades existentes em ir sozinho estudar em um estado até então desconhecido por mim.

## ABSTRACT

Constructive algorithms represent one of the three fundamental approaches of metaheuristics, along with the modification and recombination strategies. In this context, the goal is to develop, step by step, a solution within a feasible solution space for a given problem. Examples of heuristics that employ this strategy include the Greedy Algorithm, Beam Search, Ant Colony Algorithm, among others. The diversity of available constructive algorithms, coupled with the nuances of metaheuristics, which are often stochastic, frequently complicates the selection of the most suitable algorithm for a specific problem. The same applies to design options, such as the definition of the neighborhood to be explored and the appropriate structures for efficient implementation. In light of these challenges, the field of automatic algorithm configuration emerges with the purpose of combining multiple heuristics to achieve more effective and comprehensive solutions, using various techniques to select the best heuristics and the best parameters for a given scenario.

This work proposes the use of automatic configuration techniques to combine multiple constructive heuristics, aiming to achieve the best possible combination for specific problems. Additionally, it intends to compare the results obtained with other solutions found in the literature, employing different problems for the evaluation of these metrics.



**Keywords:** Constructive Algorithms. Heuristics. Combinatorial Optimization. Context-free grammar.

**Configuração automática de Heurísticas Construtivas de propósito geral**

## RESUMO

Os algoritmos construtivos representam uma das três abordagens fundamentais das metaheurísticas, juntamente com as estratégias de modificação e recombinação. Nesse contexto, o objetivo é desenvolver, passo a passo, uma solução num espaço de soluções viáveis para um problema. Exemplos de heurísticas que empregam essa estratégia incluem o Algoritmo Guloso, a Busca por Raio, o Algoritmo de Colônia de Formigas, entre outros. A diversidade de algoritmos construtivos disponíveis, aliada às nuances das metaheurísticas, que muitas vezes são estocásticas, frequentemente dificulta a escolha do algoritmo mais adequado para um problema específico. O mesmo acontece com as opções de projeto, como a definição da vizinhança a ser explorada e das estruturas adequadas para uma implementação eficiente. Diante desses desafios, a área de configuração automática de algoritmos surge com o propósito de combinar várias heurísticas para alcançar soluções mais eficazes e abrangentes, empregando diversas técnicas para selecionar as melhores heurísticas e os melhores parâmetros para um determinado cenário.
Este trabalho propõe a utilização de técnicas de configuração automática para combinar múltiplas heurísticas construtivas, visando alcançar a melhor combinação possível para problemas específicos. Além disso, pretende-se comparar os resultados obtidos com outras soluções encontradas na literatura, empregando diferentes problemas para a avaliação dessas métricas.

**Palavras-chave:** Algoritmos construtivos. Heurísticas. Otimização combinatória. Gramáticas livres de contexto.

## LIST OF FIGURES

## LIST OF TABLES

**CONTENTS**

# 1 INTRODUCTION

Every day we encounter optimization problems in our lives. Whether finding the shortest path from our house to our workplace or deciding the best way to organize our daily tasks, we always seek an efficient way to solve our problems. In mathematics, the field of finding the optimal solution to a problem is called optimization. In computer science, the branch of optimization that aims to find the best solution in a finite set of possibilities is called combinatorial optimization (Papadimitriou; Steiglitz, 1998).

These problems can be expressed as a set $P \subseteq I \times S$ of instances and solutions. All the problems for which a solution can be verified in polynomial time belong to the complexity class NP. Some optimization problems can also be solved by algorithms with a polynomial time complexity, such as the Minimum Spanning Tree (Kruskal, 1956). These problems are elements of the complexity class P. Other problems in the NP class can not be solved by an algorithm with polynomial time complexity unless it is proven that P and NP are the same. In other words, no algorithm can solve them exactly in a polynomial time unless $P = NP$ (Papadimitriou, 1994). One example of NP-Hard combinatorial optimization problems is the Knapsack Problem (Martello; Toth, 1987).

Fortunately, not every problem needs to be solved precisely. Often, an approximate solution is sufficient for various practical applications. To achieve these solutions, many techniques have been developed to create approximations to the optimal solution of a problem, such as approximation algorithms (Williamson; Shmoys, 2011) and even machine learning (Zhou; Liu, 2021). Another widely used method for solving combinatorial optimization problems is the heuristic.

Heuristics are algorithms that find a solution to a problem in a reasonable time, with no formal guarantee of the quality of this solution. Even without these, those algorithms are widely used in science and industry, considering they can find satisfactory solutions for many applications. Today, multiple instances of famous problems have their best solutions encountered by heuristics, like the Traveling Salesman Problem (TSP) (Gutin; Punnen, 2007) and the Job-Shop Scheduling Problem (JSSP) (Applegate; Cook, 1991).

Heuristics can be divided into three main categories based on how they explore the solution space: constructing, modifying, or combining solutions. In this work, we will focus on the constructive heuristics. These algorithms build a solution step by step, adding elements to a partial solution until it is complete.

Constructive heuristics must be capable of exploring the solution space efficiently.

Many decisions must be made to accomplish this goal, such as finding the best element selection criteria and the construction method. Currently, many works in the literature propose using automatic configuration techniques to explore the space of heuristics, finding the best option for a specific problem.

In this work, we propose a grammar to represent a sub-space of constructive heuristics and the creation of a metaheuristic that can generate results close to state-of-the-art algorithms. Also, we want to create an interface to our metaheuristic that can be used for different problems without changing the main algorithm. Our main contributions are: The creation of a grammar that is able to represent multiple classical constructive heuristics and the definition and implementation of a metaheuristic that uses this grammar.

In Chapter 2, we will define a combinatorial problem, a Heuristic, and the algorithms used as a base for this work. In Chapter 3, we will introduce our proposed grammar and algorithm. In Chapter 4, we will explain the problems explored, the tuning scenario, the resulting algorithms, and their results.

# 2 HEURISTIC ALGORITHMS

In combinatorial optimization, a problem $P$ can be defined as a tuple $P = (I, S, f)$, where $I$ is an instance of a problem, $S$ is the set of complete solutions and $f$ is an objective function, typically defined as $f : S \rightarrow \mathbb{R}$. The goal is to find a solution $s \in S$ where $f(s)$ is optimal. These solutions $s$ are called optimal solutions, defined here as $s^*$. The value $f(s^*)$ is called the optimal value of the problem and will be referenced here as *OPT*. By optimal we mean that if the problem is a minimization problem, *OPT* will be the minimal value for $f$. If the problem is a maximization problem, *OPT* will be the maximal value for the function.

For some problems, finding a optimal solution is an easy task. For example, the shortest path problem can be solved in polynomial time using Dijkstra's algorithm (Dijkstra, 1959). However, for many of the optimization problems, finding $s^*$ is hard, and no known algorithm can find $s^*$ in a polynomial time.

Many heuristics use the concept of a solution space, where each solution is a point in the space. With this concept, we can see a heuristic as an algorithm that searches for a good solution in the space.

In constructive heuristics, it is possible to visualize the relation between the partial solutions as a Directed Acyclic Graph (DAG). In this DAG, the root is an empty solution, and each internal node is a partial solution, and each final node is a complete solution in the solution space. At each step, the algorithm chooses the next node to explore, based on some criteria. In the end, the choices made by the algorithm will be a path in the DAG.

## 2.1 Classical Constructive Heuritics

There are many constructive heuristics in the literature. Some strategies will use the concept of exploitation, exploring a small number of alternative paths and choosing the best solutions greedily. Others will use the concept of exploration, visiting a large number of alternatives to find the best solution even when their path is not the best. The following section will describe some of the heuristics studied in this work.

We will use the following concepts and notation from now on to describe the heuristics, expanding the concepts previously defined: $E$ is the set of all the elements in instance $I$. Solution $s$ is a subset of $E$ that represents a complete solution. The set of all complete solutions of an instance $I$ is defined as $S$. A set of partial solutions will be

defined as $\overline{S}$, where each element is a subset of $E$. A partial solution in this set will be referenced as $\overline{s}$ and $\overline{s}_i$ is a partial solution with $i$ elements. Note that $S \subseteq \overline{S}$ (in other words, a complete solution is also a partial solution). A partial solution with no elements is $\emptyset$.

The neighborhood of a partial solution $\overline{s}$ is $N(\overline{s})$, defined as the set of all reachable partial solutions from $\overline{s}$ by adding ($N^+(\overline{s})$) or removing ($N^-(\overline{s})$) one element. The set of elements such that $\overline{s} \cup \{e\} \in N^+(\overline{s})$ will be referenced as $E^+$. The element $e^*$ is the one that maximizes $f(\overline{s} \cup \{e^*\})$.

We will use this notation to define each algorithm in the next subsection.

### 2.1.1 Priority Algorithm

All the heuristics explored in this work rely on selecting an element in a good order. A good order is a sequence of elements that, when processed by the heuristic, produces a good result. To guarantee a good element selection in each algorithm, we will use the concept of Priority Algorithms (Borodin; Nielsen; Rackoff, 2003). A priority algorithm (Algorithm 1) is a heuristic that, when given a partial solution $\overline{s}$, returns a sequence $E^* \subseteq E$ sorted by some criterion. This criterion is a function $\sigma : E \times \overline{S} \to \mathbb{R}$ that evaluates the element $e$ based on the partial solution $\overline{s}$ and returns a value.

**Algorithm 1** Priority algorithm

**Require:** A partial solution $\overline{s}$, a function $\sigma$
**Ensure:** An sequence of elements $E^*$
1: $E^* = \{\}$
2: **for** $e \in E^+$ **do**
3: $\quad E^* = E^* \cup \{e^+ \mapsto \sigma(e^+, \overline{s})\}$
4: **end for**
5: **return** $E^*$

This concept will be used in the definition of the following algorithms.

### 2.1.2 Greedy Algorithm

A greedy algorithm (Algorithm 2) is the simplest constructive method used to solve a problem using a priority algorithm. It starts with an empty solution $\emptyset$ and, at each step, adds the element $e^*$ to $\overline{s}$. The algorithm stops when $\overline{s} \in S$.

**Algorithm 2** Greedy algorithm

**Require:** An instance $I$
**Ensure:** $\overline{s} \in S$
1: $\overline{s} \leftarrow \emptyset$
2: **while** $\overline{s} \notin S$ **do**
3: $e^* \leftarrow \arg\max_{e \in E^+} f(\overline{s} \cup \{e\})$
4: $\overline{s} \leftarrow \overline{s} \cup \{e^*\}$
5: $E \leftarrow E \setminus \{e^*\}$
6: **end while**
7: **return** $\overline{s}$

The greedy algorithm is deterministic and will always find the same solution for the same instance. A variation of this algorithm is the $\alpha$-greedy; Algorithm (3). In this version, we add a random factor $\alpha$ to the choice of the element. When added, the algorithm will choose the element $e^*$ with probability $\alpha$, otherwise, it will select a random element. The selection of a random element from a set will be referenced in algorithms and formulas from now on as $\in_R$.

**Algorithm 3** $\alpha$-Greedy Algorithm

**Require:** An instance $I$, a random factor $\alpha \in [0, 1]$
**Ensure:** $\overline{s} \in S$
1: $\overline{s} \leftarrow \emptyset$
2: **while** $\overline{s} \notin S$ **do**
3: **if** $X \sim U([0, 1]) < \alpha$ **then**
4: $e \leftarrow \arg\max_{e \in E} f(\overline{s} \cup \{e\})$
5: **else**
6: $e \leftarrow e \in_R E$
7: **end if**
8: $\overline{s} \leftarrow \overline{s} \cup \{e\}$
9: $E \leftarrow E \setminus \{e\}$
10: **end while**
11: **return** $\overline{s}$

Another common variation is to define a subset $B_k \subseteq E$ of the $k\%$ best elements at each step. In this case, the algorithm randomly chooses one element $e$ from $B_k$. This version is mentioned in the literacy as a $k$-greedy algorithm (4).

### 2.1.3 Pilot Method

The Pilot Method was defined by Duin and Voß (1999). It uses a constructive heuristic $H$ that, at each step, chooses the $e$ that gives the best complete solution $s$ based

**Algorithm 4** $k$-Greedy Algorithm

**Require:** An instance $I$, a integer $k \in [0, 1]$
**Ensure:** $\overline{s} \in S$
1: $\overline{s} \leftarrow \emptyset$
2: **while** $\overline{s} \notin S$ **do**
3: $\quad e \leftarrow e \in_R B_k$
4: $\quad \overline{s} \leftarrow \overline{s} \cup \{e\}$
5: $\quad E \leftarrow E \setminus \{e\}$
6: **end while**
7: **return** $\overline{s}$

on some simpler heuristic. Here, all the $e \in E^+$ are expanded at each step.

**Algorithm 5** Pilot Method

**Require:** An instance $I$, an Heuristic $H$
**Ensure:** $\overline{s} \in S$
1: $\overline{s} \leftarrow \emptyset$
2: **while** $\overline{s} \notin S$ **do**
3: $\quad e \leftarrow \arg\max_{e \in E} f(H(\overline{s} \cup \{e\}))$
4: $\quad \overline{s} \leftarrow \overline{s} \cup \{e\}$
5: $\quad E \leftarrow E \setminus \{e\}$
6: **end while**
7: **return** $\overline{s}$

Originally, the Pilot Method used the Greedy Algorithm as the simpler heuristic to evaluate the solutions, but the definition does not restrict the use of other heuristics. Some research was made trying to improve the heuristic performance and results, such as (Voß; Fink; Duin, 2005)

Another way to see the Pilot Method is as a Greedy Algorithm that uses $\sigma = f(H(\overline{s} \cup \{e\}))$. In other words, it uses the objective value of the solution found by a heuristic algorithm as the priority function.

### 2.1.4 Beam Search Algorithm

Beam Search was introduced by Lowerre (1976), when developing the speech recognition system Harpy. Even been developed in the Artificial Intelligence context, this algorithm has been widely used in the context of combinatorial optimization.

In this algorithm, we define a set $B$, called beam, with maximum size $n$ of partial solutions to keep track. At each step, for each $\overline{s} \in B$, a subset of $N^+(\overline{s})$ with size $m$ is evaluated to update $B$. At the end of the step, the algorithms keep the $n$ best solutions in

$B$. This update can be elitist or non-elitist. An update is called elitist when the algorithm also uses the partial solutions $\overline{s} \in B$ in the creation of the new beam.

When $\overline{s} \in S$, the algorithm removes $\overline{s}$ from $B$ and adds the complete solution to the set of solutions found. In the end, the best solution is returned.

**Algorithm 6** Beam Search

**Require:** An instance $I$, two integers $n$ and $m$
**Ensure:** $s^* \in S$
1: $B \leftarrow \{\emptyset\}$
2: **while** $B \neq \{\}$ **do**
3: $B' \leftarrow \{\}$
4: **for each** $\overline{s} \in B$ **do**
5: $B_{\overline{s}} \leftarrow$ select $m$ neighbors using $\sigma(N^+(\overline{s}))$
6: $B' \leftarrow B' \cup B_{\overline{s}}$
7: **end for**
8: **if** elitist **then**
9: $B \leftarrow$ select $n$ best solutions in $B' \cup B$
10: **else**
11: $B \leftarrow$ select $n$ best solutions in $B'$
12: **end if**
13: **for each** $\overline{s} \in B$ **do**
14: **if** $\overline{s} \in S$ **then**
15: $B \leftarrow B \setminus \{\overline{s}\}$
16: **end if**
17: **end for**
18: **end while**
19: **return** $s^*$

The number of solutions kept in $B$ is called beam-width, and the number of solutions evaluated in each step is called expansion-width or beam-factor. A Beam Search with beam-width $n$ and expansion-width $m$ will be referenced as Beam Search($n$, $m$).

This algorithm is a generalization of the Greedy Algorithm, where $n = 1$ and $m = |E|$.

### 2.1.5 Multi-Start Algorithm

Multi-Start Algorithm is the idea of running a heuristic multiple times until reaches a stop condition and returns the best solution found. The stop condition can be any condition that can be tracked between iterations. Some of the most common are the number of iterations, the number of iterations without improvement, a time limit, or a budget limit (number of essential operations).

**Algorithm 7** Multi-Start Algorithm

**Require:** An instance $I$, an integer $n$
**Ensure:** $s^* \in S$
1: **for** $i = 1$ to $n$ **do**
2: $\quad s' \leftarrow H(I)$
3: **end for**
4: **return** $s^*$

This concept is not exclusive to constructive methods and can be seen in many other non-constructive algorithms and heuristics, such as the Simulated Annealing (Kirkpatrick; Jr; Vecchi, 1983). It is possible to see that the inner heuristic inside must be non-deterministic, or at least, the initial condition in each iteration must be different from the previous ones. Otherwise, all the results produced by the heuristic will be the same.

### 2.1.6 Ant-Colony Optimization

Introduced by Dorigo, Maniezzo and Colorni (1996) and Dorigo, Caro and Gambardella (1999), this algorithm was designed for optimization in graph problems. Even so, many other problems can be solved using it.

Ant-Colony Optimization (ACO) is an algorithm designed to mimic the behavior of an ant colony finding the best path to a food source. When ants start the procedure to find a new source of food, they begin to search for food randomly, leaving a pheromone trail behind them. After find some food, the ant returns to the nest, reinforcing the pheromone trail. If no food is found by an ant, it will return to the colony without throwing these pheromones. Following that, other ants will follow the stronger trail, increasing the probability of finding the food. This process occurs multiple times until most of the ants follow the same trail. Paths that are not used will have their pheromone trail evaporated during the time.

Bringing the analogy to the algorithm, a colony of ants is a set $A$ of $n$ partial solutions (also called ants). At every iteration, called a walk, all the ants will construct a new solution using a non-deterministic greedy strategy, referred to from now on as $\tau$-Greedy.

For each step in their walk, all the $n$ ants will select an element $e$ with a probability $p(e)$ defined by the Equation 2.1. This equation uses the pheromone value $\tau_e$ deposited by other ants in previous iterations and the heuristic information $\eta_e$. In this equation, $\alpha$ and $\beta$ are real variables that represent the weight of each value in the probability.

$$p(e) = \frac{\tau_e^\alpha \eta_e^\beta}{\sum_{e' \in E} \tau_{e'}^\alpha \eta_{e'}^\beta} \tag{2.1}$$

When all the ants construct their solution in an interaction, the pheromone trail of each element is updated using the Equation 2.2, using an evaporation rate $\rho$.

$$\tau_e \leftarrow (1-\rho)\tau_e + \rho \sum_{k=1}^{n} \Delta\tau_e^k \tag{2.2}$$

Also, $\Delta\tau_e^k$ is defined for each ant using the Equation 2.3. We use $s_k$ to denote the solution found by the ant $k$.

$$\Delta\tau_e = \begin{cases} \frac{1}{f(s_k)}, & \text{if } e \in s_k \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

The application of these equations for a given $e$ and $\tau$ set can define the Algorithm 8. Using all of these operations, we can define one of many variations of the Ant-Colony Optimization as in Algorithm 9.

**Algorithm 8** Evaporate

**Require:** An set $\tau$, an Elements set $E$
1: **for** $e \in E$ **do**
2: $\quad \tau_e \leftarrow (1-\rho)\tau_e + \rho \sum_{k=1}^{n} \Delta\tau_e^k$
3: **end for**

**Algorithm 9** Ant-Colony Optimization

**Require:** An instance $I$, $(m, n_{ants}) \in \mathbb{Z}^+$ and $(\alpha, \beta, \rho) \in \mathbb{R}^+$
**Ensure:** $s^* \in S$
1: **for** $i = 1$ to $m$ **do**
2: $\quad$ **for** $k = 1$ to $n_{ants}$ **do**
3: $\quad\quad s_k \leftarrow \tau$-Greedy$(I)$
4: $\quad$ **end for**
5: $\quad$ evaporate($\tau$, $E$)
6: **end for**
7: **return** $s^*$

If we remove the biological analogy, the ACO can be seen as a variation of the multi-start algorithm. Instead of running the heuristic a single time for each interaction, ACO runs the heuristic $m$ times. The $\tau$-greedy algorithm is a simple $\alpha$-greedy algorithm that uses the pheromones system as a priority function. At the end of each iteration, the priority function is updated based on the solutions found.

## 2.2 Hybridizing Heuristics

As introduced in the previous section, some constructive heuristics have similar recipes: We build a solution step by step, using some priority algorithm to select the next element. Some of them keep more than one solution at the same time, while others run an internal heuristic multiple times, trying to find a better solution through repetition.

When we look at the multiple implementations of the greedy algorithm shown previously, including the $\tau$-greedy defined in Algorithm 9, we see that the only difference is the priority algorithm that selects the next element. Even further, the Beam Search algorithm, defined in Algorithm 6, can be seen as a generalization of the greedy algorithm allowing multiple solutions at the same time.

Finding the common ground between two or more heuristics allows us to mix them to create another that explores the solution space differently. This approach in the design of heuristics is called hybridization and has created some well-known heuristics, such as the Beam-ACO (Blum, 2005). When we compare the algorithms above, we can create a generic constructive heuristic, defined in Algorithm 10.

**Algorithm 10** Generic Constructive Heuristic

**Require:** An instance $I$, a $(m, n) \in \mathbb{Z}^+$, a priority algorithm $\sigma$
**Ensure:** A solution $s \in E$
**for** $i = 1$ to $m$ **do**
    Initialize $n$ different partial solutions
    **for** $k = 1$ to $n$ **do**
        **while** $s_k \notin S$ **do**
            Select $m$ elements to compare using $\sigma$
            Add the best element to the solution $s_k$
        **end while**
    **end for**
    Update $\sigma$
**end for**
**return** $s^*$

With this idea in mind, in the next chapter, we propose a new approach, using regular grammar to create a constructive metaheuristic.

# 3 A GRAMMAR-DRIVEN HYBRID HEURISTIC

The first step in developing a hybrid heuristic is to define an algorithmic space. To accomplish this, we need to define the algorithms in our space and the relation between them. When we observe the algorithms defined in the last section, they all follow the same pattern. Each one of them has a main algorithm and a priority algorithm. This simplicity allows us to represent all the addressed algorithms as regular grammar, using the concepts defined in Chomsky (1956). In this grammar, the terminals represent a concrete algorithm, and the non-terminals represent one type of algorithm. The use of formal grammar to create hybrid heuristics is not new and was used previously in works like Brum and Ritt (2018) and Pagnozzi and Stützle (2021).

One advantage of using a formal grammar to create hybrid heuristics is that we formalize a space of possible heuristics when using them. Applying the generated functions to a problem, we can define a quality value for each, and use this value to select the best heuristic in the space. To explore this space for a specific problem, we can use a tool such as irace (López-Ibáñez et al., 2016).

In the first section of this chapter, we will explore those similarities to create a grammar that expresses a generic constructive algorithm. Each symbol in this grammar will be a part of some algorithm explained above. Those symbols can express an existing algorithm or create a new one, mixing two or more. After that, we will explain our implementation, the use of object-oriented programming techniques to develop a generic solver using this grammar, and the process used to select good algorithms.

## 3.1 A Grammar to represent an algorithmic space

Our grammar is divided into a priority algorithm and a main algorithm. The priority algorithm will be the element selection algorithm used in the main algorithm. The main algorithm will be responsible for creating the solution returned at the end of the execution. The full grammar is explained below:

$$
\begin{array}{lcll}
\mathcal{H} & ::= & \mathcal{P}\,\mathcal{A} & \textit{Heuristic} \\
\mathcal{P} & ::= & greedy & \textit{Priority Algorithm} \\
 & | & random - greedy(\alpha, k) & \\
 & | & pheromones(\gamma, \rho) & \\
 & | & pilot(k) & \\
\mathcal{A} & ::= & \mathcal{S} & \textit{Main Algorithm} \\
 & | & multi - start(m, n)\,\mathcal{S} & \\
\mathcal{S} & ::= & greedy & \\
 & | & beam - search(b, e) &
\end{array}
$$

### 3.1.1 Priority algorithm

This non-terminal will be expanded in one of the four priority algorithms explained in the previous chapter. The greedy priority algorithm is the same as the one defined before.

The random greedy priority algorithm receives two parameters: $\alpha$ and k, The first parameter is the probability of selecting the best element. If this is not selected, k represents the proportional amount of elements to choose at random. In this case, the algorithm will select one between the $k\%$ best elements to choose with an equal probability, including the best one.

The pheromones selection represents the $\tau$-greedy Selection Algorithm 9. The parameter $\rho$ represents the evaporation value, while $\gamma$ is a combination of $\alpha$ and $\beta$. Considering $\gamma \in [a, b]$, both variables can be extracted from gamma using the Formula 3.1.

$$\alpha = \frac{\gamma - a}{b - a} \qquad \beta = \frac{b - \gamma}{b - a} \tag{3.1}$$

The parameter k in the pilot priority is the proportion of elements that will expand in the internal heuristic, such as in Random Greedy. In this grammar, we only use the greedy algorithm as the internal heuristic.

### 3.1.2 Main Algorithm

The greedy algorithm is Algorithm 2. The beam-search takes two parameters: the beam-width ($b$) and the expansion-width ($e$).

The multi-start algorithm will repeat one of the others heuristics up to $m$ times or $n$ iterations without improvement. Also, for each iteration, the multi-start algorithm can generate up to $k$ solutions. This last parameter is crucial to represent the ACO algorithm in our grammar.

## 3.2 Implementation

After defining the desired grammar, we develop a concrete version of our grammar. The implementation was made with C++ and is accessible in the project GitHub repository (Ricarte, 2024). We aim to create a library with all the functions and classes needed to implement a heuristic for a problem using our grammar. A program that uses this library must be capable of receiving a configuration file and an instance for an implemented problem and return the objective value of the solution.

It is possible to divide the implementation between the interface, the algorithm implementation, and the configuration reader. The interface is the part where the user will extend to solve its specific problem. The algorithm implementations follow a pattern that enables hybridization. Finally, the configuration reader is responsible for extracting the desired algorithm from an input file.

### 3.2.1 Interface

When creating the implementation, our main goal was to create a program that could be as close as possible to the problem definitions from Chapter 2. Because of this, we divide our interface into four generic classes: `Problem`, `Instance`, `Solution` and `Element` respectively.

As we desire to create a generic interface that can be used for many problems, we only implemented some control methods in each class. Other methods are defined as abstract and will be implemented for each problem, as described in the next chapter.

The `Problem` class calculates values such as the objective value statically, without any side effect for a solution. In our implementation, the problem will be a minimization problem. That means all the algorithms will try to minimize the objective value returned by the problem. When implementing a solution for a specific problem, the user must define the following methods:

- An `objectiveValue` function, that given a $\overline{s}$, returns $f(\overline{s})$;
- An `objectiveValue` function, that given a $\overline{s}$ and an $e^+$, returns $f(\overline{s} \cup \{e^+\})$;
- An `isValid` test, that returns true if partial solution $\overline{s} \cup \{e^+\}$ is valid;
- An `isComplete` test, that returns true if partial solution $\overline{s} \in S$.

The `Instance` class stores all the instance's specific values. This class will also be responsible for the interaction between a solution and an element. The user must implement four methods when solving a problem:

- A `initializeSolution` function, that returns an solution $\emptyset \in \overline{S}$;
- A `getCandidatesElements` function, that returns $N^+(\overline{s})$;
- The tests `isValid` and `isComplete`, which return the same values as the methods in the class `Problem`.

The `Solution` class will be the class that will store any data related to a specific iteration of an algorithm. It can be the set of elements in a single solution, the iteration between them, or any other solution-related value. This class also will be responsible for calculating the quality of an element in a given solution. The abstract methods defined in this class are:

- `getSolution`, which returns the elements in the solution sorted by insertion order;
- `getElementQuality`, which returns a Heuristic value for an $e \in E^+$ for the solution;
- `getObjectiveValue`, which returns the objective value of the solution;
- `clone` function who creates a clone of the solution.

The algorithm uses the element quality as the $\sigma$ function in the selector. This is necessary because, for many problems, the computation of the objective value is costly, so we must use a simpler function for evaluation.

The struct `Element` will represent the elements for a given problem. These elements will retrieve any information related to a single element in an instance. Also, the user must define a partial order for the elements. In other words, they need to be sorted by the $\leq$ operator.

### 3.2.2 Algorithm Implementation

#### *3.2.2.1 Selector*

The first part of our implementation is the class `Selector`. This generic class is a basic interface for all the priority algorithms. The three main methods used in this class are: `initialize`, `updateProbabilitiesIteration` and `selectElement`. The first method is used to initialize all the internal elements of the class; the second is responsible for updating all the probabilities used in the selection and is especially important for the implementation of Equation 2.2. The last method returns the $e \in E^+$ element given by the priority algorithm.

#### *3.2.2.2 Main Algorithm*

To allow the hybridization, we implement all the main algorithms to receive at least the same tuple of parameters: a problem, an instance, and a selector. Also, they all solve the problem as a minimization problem and return the best value using this criterion.

For the multi-start algorithm, we created a struct called `StoppingCriteria`, which keeps counting the number of iterations and iterations without improvement. A single method called `shouldStop` was implemented; it returns true if the algorithm reaches some stopping criterion. Another difference in the multi-start algorithm is that it receives another heuristic as a function parameter.

Internally, all algorithms follow the same structure as defined in Chapter 2.

### 3.2.3 Configuration Reader

Our goal with this implementation is to build an algorithm with an input that follows the predefined grammar. To implement this functionality, we translated our grammar to the JSON syntax as follows:

```
<output> ::=
 { "type": <algorithm>, "priority": <priority-config> }

<algorithm> ::=
 { "type": "greedy" } |
```

```
 { "type": "beamsearch", "beam-width": <integer>, "
    expansion-width": <integer> } |
 { "type": "iterated", "internal-algorithm": <internal-
    algorithm>, "stop": <stop-criteria>, "num-solutions": <
    integer> | (nothing) }

<internal-algorithm> ::=
 { "type": "greedy" } |
 { "type": "beamsearch", "beam-width": <integer>, "
    expansion-width": <integer> }

<stop-criteria> ::=
 { "max-iterations": <integer>, "max-no-improvement-
    iterations": <integer> }

<priority-config> ::=
 { "type": "greedy" } |
 { "type": "random", "alpha-value": <float>, "k-value": <
    float> } |
 { "type": "pheromone", "gamma-value": <float>, "rho-value
    ": <float> } |
 { "type": "pilot", "k-value": <float> }
```

To read the configuration, a parser was used to implement an automaton that creates a `Configuration` object while parsing the file. If the parser ends the reading in a valid state, the output will contain a valid heuristic to solve the implemented problem. The configuration is fully independent of the interface. So it does not carry any information about the problem.

# 4 EXPERIMENTAL RESULTS

To evaluate the heuristic, we implement two problems: the k-Labelled Spanning Forest Problem (KLSFP) and the Permutation Flowshop Scheduling Problem (PFSSP) with total completion time minimization. In this section, we will explain each one of them, our decisions when implementing the solution, and the obtained algorithm and results.

Two metrics was used to evaluate a heuristic: the objective value and the cost. For cost, we mean the number of main operations for the result. The operation used as the metric will be specified for each problem.

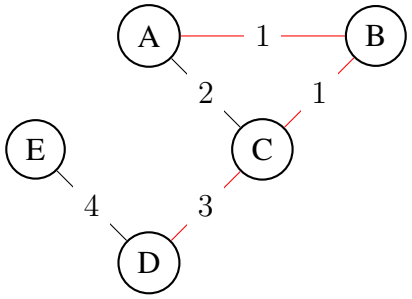
For every problem, we select a published constructive algorithm that can be expressed in the grammar and has competitive results to be the base for comparison. We execute the same tests for the obtained configuration and the base algorithm to evaluate the quality of the heuristics.

## 4.1 K-Labelled Spanning Forest

Introduced by Cerulli et al. (2014), the k-Labelled Spanning Forest is a a problem $\mathcal{P} = (G, k)$ defined as follow: $G = (V, E, L)$ is a undirected graph with a set of labels $L$, where each edge has exactly one label. The goal is to find a subgraph $G' = (V, E') \subseteq G$ with the edges from $k$ distinct labels that minimizes the number of connected components of $G'$.

Figure 4.1 shows an example instance of the problem with 5 nodes, 5 edges, 4 labels and $k = 2$. A solution that minimizes the object value for this instance would be a $G'$ containing the edges with labels 1 or 3, with two connected components.

Figure 4.1 – Example of KLSFP instance and solution for $k = 2$ (in red)




Using the notation introduced in Chapter 2, we can define each label as an element in $E$. A solution $s$ is the subgraph $G'$. The solution will complete when the number

of labels selected is the same as $k$. Then, our function $f(s)$ will return the number of connected components of $s$.

### 4.1.1 Implementation

In the implementation, the `Element` class represents a label. Besides the integer that represents the label, this class also keep a list of edges that have the label. To calculate the number of connected components in the `Solution`, we use the `UnionFind` data struct. That was used either in the element quality computation or the objective value computation. The `Instance` class keeps the graph information and a list of every label used in the solution. In this problem, the number of Union in the `UnionFind` is the main operation for the cost.

The Pilot method was used as the base algorithm. This algorithm was the state of the art on the problem in the past and can be achieved from our grammar. The scenario shown in Table 4.1 was used to train the model. We limit the cost of a configuration to the maximum cost obtained by the number of nodes on $G$ using the base algorithm. This decision limited the algorithmic space that the heuristic could reach but guaranteed that the resulting configuration would have a similar cost order as the base algorithm. We also compare the results with the current state-of-the-art algorithm (CBFS) proposed by Ritt (2024).

Table 4.1 – Parameters configuration.

| **Name** | **Type** | **Values/Range** |
|---|---|---|
| beam width | Integer | 2 to 8 |
| beam factor | Integer | 2 to 8 |
| max iterations | Integer | 0 to 20 |
| max no improvement | Integer | 0 to 20 |
| parallel solutions | Integer | 1 to 10 |
| random $\alpha$ | Real | 0 to 0.99 |
| random $k$ | Real | 0 to 0.99 |
| pilot $k$ | Real | 0 to 0.99 |
| pheromones $\gamma$ | Real | -1 to 1 |
| pheromones $\rho$ | Real | 0 to 1 |

### 4.1.2 Instances

The instances used were generated based on Pinheiro, Ravelo and Buriol (2022). These instances are randomly generated graphs with a specified number of nodes, edges, labels, and $k$ value. All the generated graphs have an edge density of $0.2$. This means that the number of edges is $20\%$ the number of all the possible edges for the graph. The number of labels is a ratio of the number of vertices.

We divided the instances into classes as shown in Table 4.2. For every class, 3 instances were generated for training and 10 for test.

Table 4.2 – KLSFP instaces classes

| **Nodes** | **Labels Ratio** | **Max Labels ($k$)** |
|---|---|---|
| 100 | $\frac{\|V\|}{4}, \frac{\|V\|}{2}, \frac{\|V\|}{1}, \frac{5\|V\|}{4}$ | 3, 6 |
| 200 | $\frac{\|V\|}{4}, \frac{\|V\|}{2}, \frac{\|V\|}{1}, \frac{5\|V\|}{4}$ | 3, 6, 12 |
| 300 | $\frac{\|V\|}{4}, \frac{\|V\|}{2}, \frac{\|V\|}{1}, \frac{5\|V\|}{4}$ | 2, 4, 9 |
| 400 | $\frac{\|V\|}{4}, \frac{\|V\|}{2}, \frac{\|V\|}{1}, \frac{5\|V\|}{4}$ | 3, 6, 12 |
| 500 | $\frac{\|V\|}{4}, \frac{\|V\|}{2}, \frac{\|V\|}{1}, \frac{5\|V\|}{4}$ | 3, 7, 15 |
| 1000 | $\frac{\|V\|}{4}, \frac{\|V\|}{2}, \frac{\|V\|}{1}, \frac{5\|V\|}{4}$ | 3, 7, 15 |

### 4.1.3 Results

The training returns as best configuration a Pilot Method with $k = 0.6024$ and the obtained results were grouped by Nodes, Labels, and Max Labels. Table 4.3 shows the average percentual difference and standard deviation between the objective value and cost of our configuration and the base algorithm. In both cases, a negative value means that our algorithm had a better performance when compared to the base algorithm.

Table 4.3 shows consistent values, with only two groups of instances showing different results from the base algorithm. At the same time, all the instances have a $40\%$ decrease in cost, showing that the algorithm keeps the same results with a better performance.

When comparing the same results with the CBFS algorithm, we have a more diverse result, as shown in Table 4.4 and in Figure 4.2. While most instances have similar values, some classes have a worse performance when compared to the current state-of-the-art algorithm, as shown in the Figure 4.3. Even so, we managed to have better results

Table 4.3 – Comparison of objective value and cost differences with the Pilot Method.

| **Nodes** | **Labels** | **Max** | **Value (%)** | **Cost (%)** |
|---|---|---|---|---|
| 100 | 25 | 3 | 0.00 ± 0.00 | -39.74 ± 0.52 |
| 100 | 50 | 3 | 0.00 ± 0.00 | -39.96 ± 0.25 |
| 100 | 50 | 6 | 0.00 ± 0.00 | -40.65 ± 0.08 |
| 100 | 100 | 6 | 0.00 ± 0.00 | -39.94 ± 0.10 |
| 100 | 125 | 6 | 0.00 ± 0.00 | -39.90 ± 0.08 |
| 200 | 50 | 3 | 0.00 ± 0.00 | -40.12 ± 0.09 |
| 200 | 100 | 6 | 0.00 ± 0.00 | -40.19 ± 0.12 |
| 200 | 200 | 6 | 0.00 ± 0.00 | -39.90 ± 0.04 |
| 200 | 250 | 6 | 0.00 ± 0.00 | -39.88 ± 0.01 |
| 200 | 250 | 12 | 0.00 ± 0.00 | -40.09 ± 0.02 |
| 300 | 75 | 2 | 0.00 ± 0.00 | -39.89 ± 0.05 |
| 300 | 75 | 4 | 0.00 ± 0.00 | -40.49 ± 0.10 |
| 300 | 150 | 4 | 0.00 ± 0.00 | -40.01 ± 0.08 |
| 300 | 300 | 9 | 0.00 ± 0.00 | -39.91 ± 0.01 |
| 300 | 375 | 9 | 0.00 ± 0.00 | -39.87 ± 0.02 |
| 400 | 100 | 3 | 0.00 ± 0.00 | -39.94 ± 0.06 |
| 400 | 200 | 6 | 0.00 ± 0.00 | -40.02 ± 0.02 |
| 4000 | 400 | 6 | 0.00 ± 0.00 | -39.85 ± 0.00 |
| 400 | 400 | 12 | 0.00 ± 0.00 | -39.92 ± 0.01 |
| 400 | 500 | 12 | -1.00 ± 3.16 | -39.87 ± 0.01 |
| 500 | 125 | 3 | 0.00 ± 0.00 | -39.96 ± 0.06 |
| 500 | 250 | 7 | 0.00 ± 0.00 | -40.05 ± 0.03 |
| 500 | 500 | 7 | 0.00 ± 0.00 | -39.84 ± 0.04 |
| 500 | 625 | 7 | 0.00 ± 0.00 | -39.81 ± 0.01 |
| 500 | 625 | 15 | 0.00 ± 0.00 | -39.86 ± 0.01 |
| 1000 | 250 | 3 | 0.00 ± 0.00 | -40.00 ± 0.03 |
| 1000 | 500 | 7 | 2.41 ± 8.01 | -39.88 ± 0.03 |
| 1000 | 1000 | 7 | 0.00 ± 0.00 | -39.80 ± 0.01 |
| 1000 | 1000 | 15 | 0.00 ± 0.00 | -39.85 ± 0.00 |
| 1000 | 1250 | 15 | 0.00 ± 0.00 | -39.79 ± 0.00 |

for four classes of instances.

Table 4.4 – Comparison of objective value and cost differences with CBFS.

| **Nodes** | **Labels** | **Max** | **Value (%)** | **Cost (%)** |
|---|---|---|---|---|
| 100 | 25 | 3 | 0.00 ± 0.00 | 372.28 ± 29.34 |
| 100 | 50 | 3 | 0.84 ± 2.22 | 789.65 ± 17.70 |
| 100 | 50 | 6 | 0.00 ± 0.00 | 1004.06 ± 4.34 |
| 100 | 100 | 6 | 3.23 ± 4.19 | 1250.40 ± 9.48 |
| 100 | 125 | 6 | 0.00 ± 0.00 | 1547.30 ± 18.84 |
| 200 | 50 | 3 | 0.00 ± 0.00 | 818.31 ± 12.81 |
| 200 | 100 | 6 | -5.00 ± 15.81 | 1657.43 ± 383.22 |
| 200 | 200 | 6 | 1.34 ± 2.37 | 2488.82 ± 21.15 |
| 200 | 250 | 6 | 0.00 ± 0.00 | 3047.58 ± 12.33 |
| 200 | 250 | 12 | 0.00 ± 0.00 | 5301.49 ± 14.08 |
| 300 | 75 | 2 | 0.00 ± 0.00 | 1540.96 ± 19.42 |
| 300 | 75 | 4 | 0.00 ± 0.00 | 1427.58 ± 202.38 |
| 300 | 150 | 4 | 0.00 ± 0.00 | 2128.80 ± 13.68 |
| 300 | 300 | 9 | -0.92 ± 5.20 | 3675.66 ± 22.53 |
| 300 | 375 | 9 | 4.64 ± 5.34 | 4446.20 ± 26.37 |
| 400 | 100 | 3 | 0.50 ± 1.58 | 1650.23 ± 21.18 |
| 400 | 200 | 6 | 1.46 ± 16.33 | 2642.44 ± 33.04 |
| 400 | 400 | 6 | 0.00 ± 0.00 | 4844.09 ± 0.00 |
| 400 | 400 | 12 | 0.00 ± 0.00 | 8131.21 ± 767.24 |
| 400 | 500 | 12 | 7.62 ± 16.16 | 6078.76 ± 32.74 |
| 500 | 125 | 3 | 0.43 ± 1.37 | 1995.52 ± 21.51 |
| 500 | 250 | 7 | -6.67 ± 36.18 | 3273.17 ± 39.64 |
| 500 | 500 | 7 | 2.03 ± 2.38 | 5907.37 ± 17.37 |
| 500 | 625 | 7 | 0.61 ± 0.52 | 7229.46 ± 9.82 |
| 500 | 625 | 15 | 0.00 ± 0.00 | 13443.71 ± 17.71 |
| 1000 | 250 | 3 | 0.17 ± 0.54 | 3796.33 ± 20.40 |
| 1000 | 500 | 7 | 3.70 ± 7.91 | 6283.27 ± 30.02 |
| 1000 | 1000 | 7 | 1.83 ± 1.53 | 11530.88 ± 24.33 |
| 1000 | 1000 | 15 | 0.00 ± 0.00 | 21364.75 ± 37.83 |
| 1000 | 1250 | 15 | -4.07 ± 7.25 | 15280.38 ± 118.95 |

The CBFS has a better cost performance, as shown in the Figure 4.4, with an average percentual difference of $4541.7\%$ compared to our configuration. As this algorithm uses multiple methods beside the constructive strategy, the CBFS is not in the algorithmic space covered in this work. This means that it is not possible to achieve this algorithm using the grammar defined in section 3.1.

Figure 4.2 – Box plot of average difference on objective value by instance class (Ours vs CBFS).

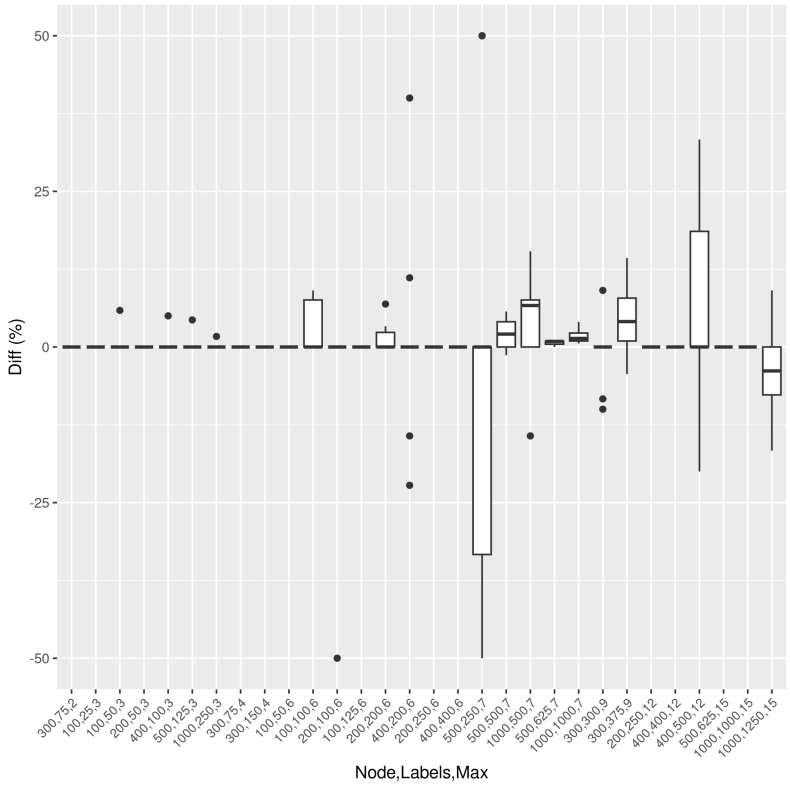



Figure 4.3 – Objective value comparison scatter plot. Ours (axis X) vs CBFS (axis Y).

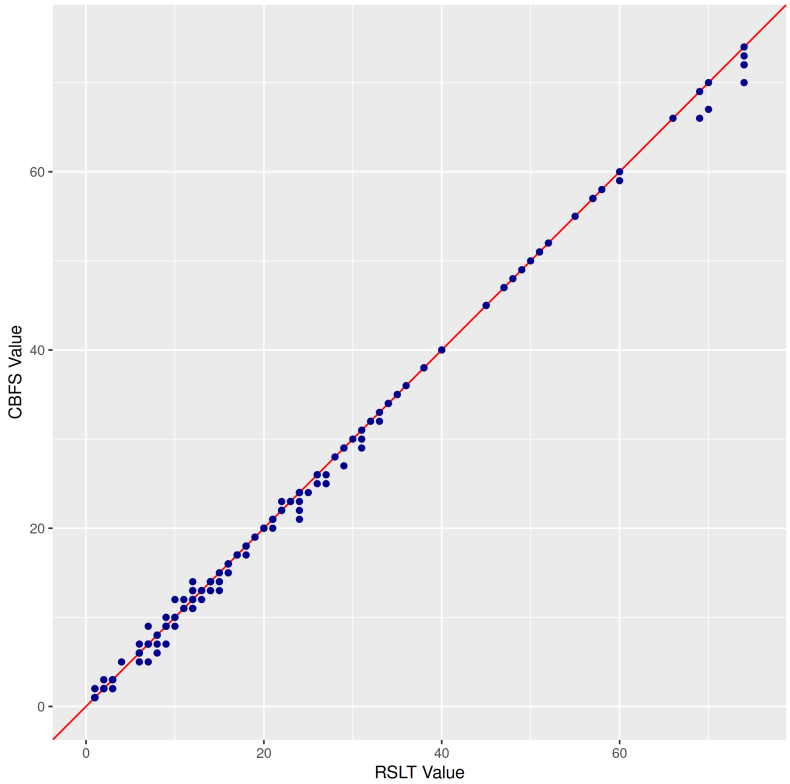
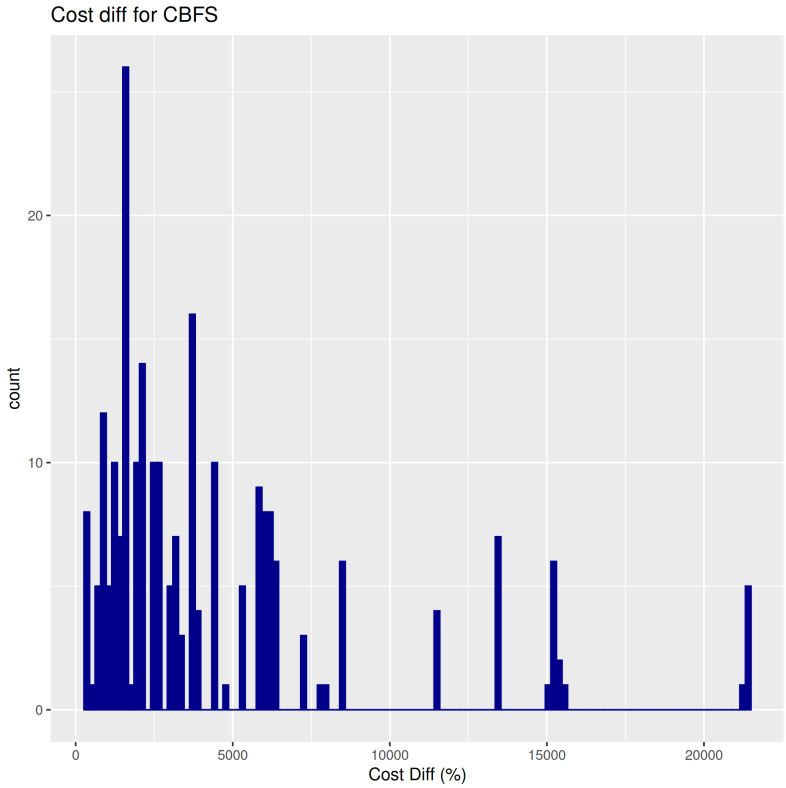


Figure 4.4 – Histogram of average difference on cost (Ours vs CBFS).




## 4.2 Flowshop Scheduling Problem

The problem was first defined by Johnson (1954). Since then, many variations and solutions have been proposed. The one used in this work is the Permutation Flowshop Scheduling Problem (PFSSP) with total completion time minimization, defined as follows: The PFSSP is a tuple $\mathcal{P} = (J, M, P)$ where

- J is a set of $n$ Jobs $J_1, ..., J_n$;
- M is a set of $m$ Machines $M_1, ..., M_m$;
- P is a set of processing time $p_{ji}$ of every job $j$ in every machine $i$.

Every machine can process up to one job at a time and any job can be made by two machines concurrently. Also, all the machines must process the jobs in the same order. We will use $C_i$ to express the total time spent to complete $J_i$, including the idle time between two machines. The problem objective is to create a permutation $\pi$ of Machines that minimizes $\sum_{i=1}^{n} C_i$.

In the notation presented in 2, the solution will be the permutation $\pi$. Each $J_j$ is an element in a solution, that will be complete when all the jobs are inserted. The function $f(\overline{s})$ will return the total time spent for every machine for the jobs already in $\overline{s}$.

### 4.2.1 Implementation

We use as the base for our implementation the algorithm developed by Brum, Ruiz and Ritt (2022) for this problem. Their Solution for the PFSSP uses a Beam Search with both beam-width and beam-factor as 3. This means that the algorithm keeps three solutions at each step, and expands three neighbors for each solution. The selection was a $\alpha$-Greedy with $\alpha = 0.8$.

The `Solution` class uses as base the implementation of a single solution used in (Brum; Ruiz; Ritt, 2022), there called a node. This implementation contains a vector of jobs that will be ordered during the execution, and values used to evaluate the elements at each iteration. The quality value of an element is defined using the heuristic defined by Fernandez-Viagas and Framinan (2017) for candidate nodes evaluation.

The `Element` class holds only the job number. In our `Instance` class, any job not selected at the moment of evaluation is valid and the solution is complete when all the jobs are selected. As the main operation, we choose both the computation of the evaluation heuristic and the computation of the objective value.

The tuning process uses the same configuration shown in Table 4.1. To avoid configurations where the cost is larger than the base algorithm cost, we limit the algorithm by budget in the tuning for the reasons expressed in Section 4.1.1. The formulas shown in Table 4.5 were defined using the cost obtained by replicating the base algorithm with our grammar and making a linear regression based on the number of Jobs.

Table 4.5 – Max Budget Formula based on Number of Jobs ($|J|$) and Number of Machines ($|M|$)

| **Number of Jobs ($\|J\|$)** | **Max Budget Formula** |
|---|---|
| $\|J\| \leq 20$ | maxBudget $= 700$ |
| $21 \leq \|J\| \leq 200$ | maxBudget $= 5418.36\,\|M\| + 189.1\,\|J\| - 31093.04$ |
| $\|J\| > 200$ | maxBudget $= 16537.1\,\|M\| + 262.73\,\|J\| - 102886.16$ |

### 4.2.2 Instances

For training the heuristic, we use variations of the benchmark instances defined by Taillard (1993). For tests, we use the original 120 benchmark instances of the paper.

These instances are divided into twelve classes, with ten instances per class. The classes are defined by a combination of a number of jobs $(20, 50, 100, 200)$ and a number of machines $(5, 10, 20)$. The time spent for each job in each machine is set randomly in

the interval $[1, 99]$.

### 4.2.3 Results

The resulting configuration has a main algorithm from the training is also a Beam Search, but with a beam-width of $2$ and beam-factor of $5$. The priority algorithm is a Random Greedy algorithm with $\alpha = 0.4695$ and $k = 0.6305$. Table 4.6 shows the average percentual difference between the results of our configuration and the base algorithm and the standard deviation of these values. For both value and cost, negative results means that our configuration had a better result then the compared one. The output values are divided by class of instances.

Table 4.6 – Comparison of average and standard deviation of values and cost differences for FSSP.

| **Jobs** | **Machines** | **Value (%)** | **Cost (%)** |
|---|---|---|---|
| 20 | 5 | -2.54 $\pm$ 3.65 | -32.76 $\pm$ 0.00 |
| 20 | 10 | -3.87 $\pm$ 3.41 | -32.76 $\pm$ 0.00 |
| 20 | 20 | -1.41 $\pm$ 3.06 | -32.76 $\pm$ 0.00 |
| 50 | 5 | -0.86 $\pm$ 2.20 | -33.23 $\pm$ 0.00 |
| 50 | 10 | -0.90 $\pm$ 2.12 | -33.23 $\pm$ 0.00 |
| 50 | 20 | 0.01 $\pm$ 1.12 | -33.23 $\pm$ 0.00 |
| 100 | 5 | 1.00 $\pm$ 1.92 | -33.31 $\pm$ 0.00 |
| 100 | 10 | -0.28 $\pm$ 1.60 | -33.31 $\pm$ 0.00 |
| 100 | 20 | -0.02 $\pm$ 2.14 | -33.31 $\pm$ 0.00 |
| 200 | 5 | 0.31 $\pm$ 1.22 | -33.33 $\pm$ 0.00 |
| 200 | 10 | 0.29 $\pm$ 1.53 | -33.33 $\pm$ 0.00 |
| 200 | 20 | -0.84 $\pm$ 1.35 | -33.33 $\pm$ 0.00 |
| 500 | 5 | 0.62 $\pm$ 0.83 | -33.33 $\pm$ 0.00 |
| 500 | 10 | 0.87 $\pm$ 0.76 | -33.33 $\pm$ 0.00 |
| 500 | 20 | 0.55 $\pm$ 1.10 | -33.33 $\pm$ 0.00 |

The results vary for each class, as shown in Figure 4.5. The heuristic had better results in smaller instances of the problem. As the size of the instances increases, the algorithm starts to return slightly worse objective values compared to the base algorithm, with the average difference never going over $1\%$. Even so, Table 4.7 and the Histogram in Figure 4.6 show that the results of our configuration were pretty similar to the compared one. When analyzing the cost of the solution, we can observe that all the classes have a third decrease in costs. Those data show that the expressive cost decrease provided by the automatic configuration may compensate for a small increase in the objective value for some instances.

Figure 4.5 – Box plot for average difference value by instance class (Ours vs Base).

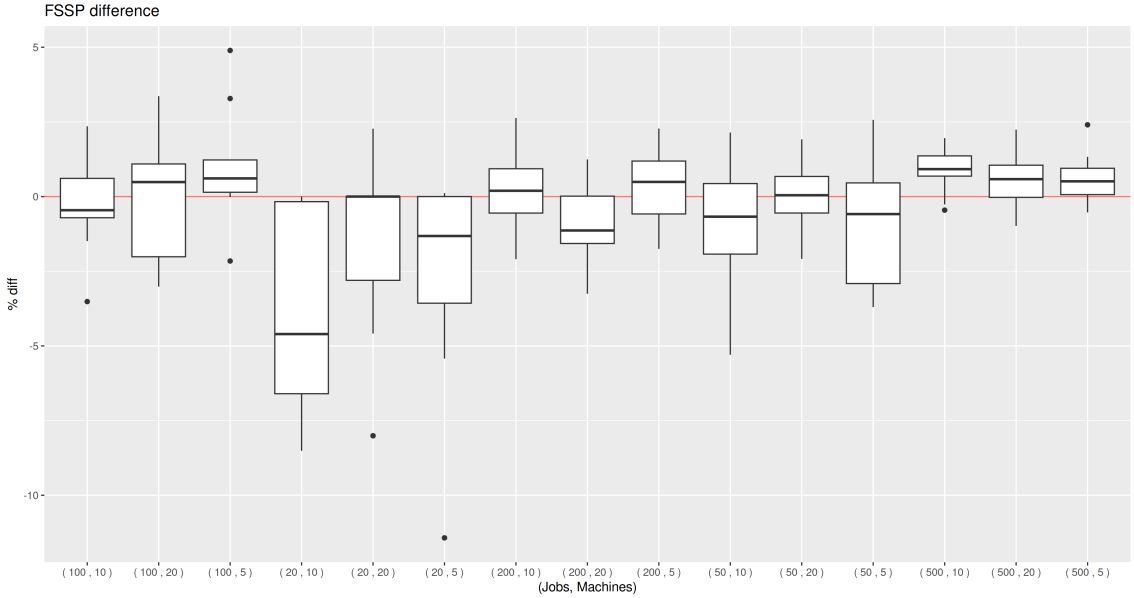
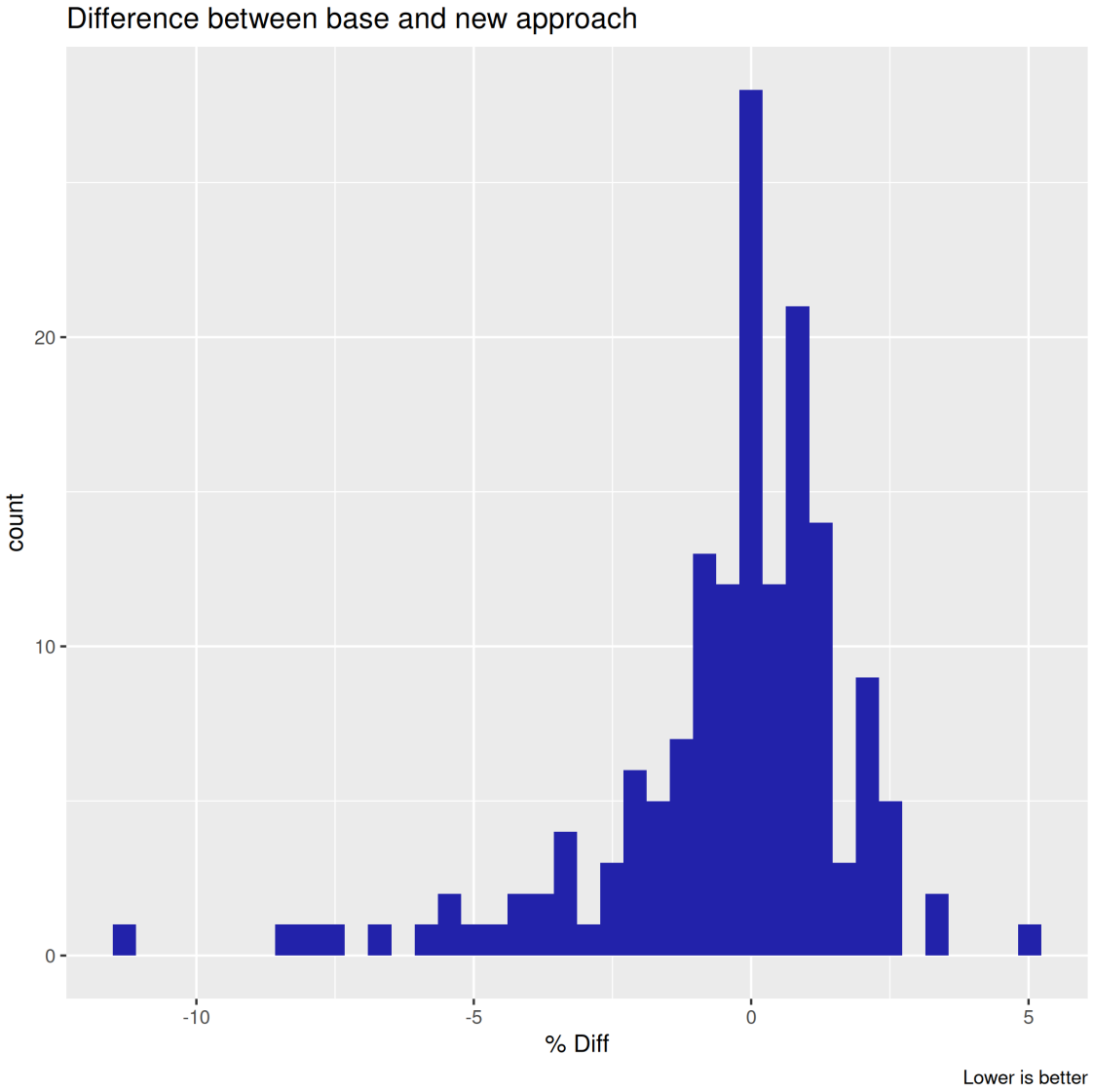



Table 4.7 – Values for FSSP

| | **Value** | **Instance** |
|---|---|---|
| Mean Cost Difference | $-33.19\%$ | |
| Worst Value Difference | $4.89\%$ | ta062 |
| Mean Value Difference | $-0.42\%$ | |
| Best Value Difference | $-11.43\%$ | ta010 |
| # Better | 72 | |
| # Equal | 12 | |
| # Worst | 76 | |

Figure 4.6 – Histogram of % difference (Ours vs Base).



# 5 CONCLUSION

The generated heuristic could achieve comparable results with the current state-of-the-art algorithms. Our algorithm could deliver similar results with a significant performance increase compared to the base algorithms in both problems. In the KLSFP, the output configuration had an average of $40\%$ decrease in the execution cost, returning the same result as the traditional Pilot method for almost every instance. In the PFSSP, the resulting beam search configuration decreased the number of essential operations by $33\%$, improving the objective value, especially in smaller instances, compared to the base algorithm. This shows that hybrid heuristics can achieve good results by being used to explore multiple heuristics and refine the existing ones.

Creating a generic library expressing multiple algorithms reduces the effort needed to solve a problem. This allows the user to focus on creating and optimizing operations in the problem, not in the strategy, knowing that, if right configured, the algorithm can return a good result.

The grammar in this work only covers a small subset of all constructive algorithms, limiting the possible configurations of the heuristic. This limits the results and performance that can be obtained. When comparing our algorithm results to CBFS, the last one still has better results using a fraction of the operations of the first one. Increasing the grammar with other algorithms is possible in future work. It is possible to expand the grammar in constructive and priority algorithms or allow new strategies to improve an existing solution, such as modifying it or recombining multiple solutions.

## REFERENCES

APPLEGATE, D.; COOK, W. A computational study of the job-shop scheduling problem. **ORSA Journal on Computing**, v. 3, n. 2, p. 149–156, 1991. Available from Internet: <https://doi.org/10.1287/ijoc.3.2.149>.

BLUM, C. Beam-aco - hybridizing ant colony optimization with beam search: An application to open shop scheduling. **Computers and Operations Research**, v. 32, p. 1565–1591, 6 2005. ISSN 03050548.

BORODIN, A.; NIELSEN, M. N.; RACKOFF, C. (incremental) priority algorithms. **Algorithmica (New York)**, v. 37, p. 295–326, 12 2003. ISSN 01784617.

BRUM, A.; RITT, M. Automatic design of heuristics for minimizing the makespan in permutation flow shops. In: IEEE. **2018 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.], 2018. p. 1–8.

BRUM, A.; RUIZ, R.; RITT, M. Automatic generation of iterated greedy algorithms for the non-permutation flow shop scheduling problem with total completion time minimization. **Computers & Industrial Engineering**, v. 163, p. 107843, 2022. ISSN 0360-8352. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0360835221007476>.

CERULLI, R. et al. The k-labeled spanning forest problem. **Procedia - Social and Behavioral Sciences**, Elsevier BV, v. 108, p. 153–163, 1 2014. ISSN 18770428.

CHOMSKY, N. Three models for the description of language. **IRE Transactions on Information Theory**, v. 2, n. 3, p. 113–124, 1956.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numer. Math. (Heidelb.)**, Springer Nature, v. 1, n. 1, p. 269–271, dec. 1959.

DORIGO, M.; CARO, G. D.; GAMBARDELLA, L. M. Ant algorithms for discrete optimization. **Artificial Life**, v. 5, p. 137–172, 4 1999. ISSN 1064-5462.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant system: optimization by a colony of cooperating agents. **IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)**, v. 26, p. 29–41, 2 1996. ISSN 1083-4419.

DUIN, C.; VOß, S. The pilot method: A strategy for heuristic repetition with application to the steiner problem in graphs. **Networks**, v. 34, n. 3, p. 181–191, 1999. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-0037%28199910%2934%3A3%3C181%3A%3AAID-NET2%3E3.0.CO%3B2-Y>.

FERNANDEZ-VIAGAS, V.; FRAMINAN, J. M. A beam-search-based constructive heuristic for the pfsp to minimise total flowtime. **Computers & Operations Research**, v. 81, p. 167–177, 2017. ISSN 0305-0548. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0305054816303288>.

GUTIN, G.; PUNNEN, A. P. (Ed.). **The Traveling Salesman Problem and Its Variations**. [S.l.]: Springer US, 2007. ISBN 978-0-387-44459-8.

JOHNSON, S. M. Optimal two- and three-stage production schedules with setup times included. **Naval Research Logistics Quarterly**, v. 1, n. 1, p. 61–68, 1954. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800010110>.

KIRKPATRICK, S.; JR, C. D. G.; VECCHI, M. P. Optimization by simulated annealing. **science**, American association for the advancement of science, v. 220, n. 4598, p. 671–680, 1983.

KRUSKAL, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. **Proceedings of the American Mathematical Society**, American Mathematical Society, v. 7, n. 1, p. 48–50, 1956. ISSN 00029939, 10886826. Available from Internet: <http://www.jstor.org/stable/2033241>.

LOWERRE, B. The harpy speech recognition system[ph. d. thesis]. 1976.

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

MARTELLO, S.; TOTH, P. Algorithms for knapsack problems. In: MARTELLO, S. et al. (Ed.). **Surveys in Combinatorial Optimization**. North-Holland, 1987, (North-Holland Mathematics Studies, v. 132). p. 213–257. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0304020808732377>.

PAGNOZZI, F.; STüTZLE, T. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems with additional constraints. **Operations Research Perspectives**, v. 8, p. 100180, 2021. ISSN 2214-7160. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S2214716021000038>.

PAPADIMITRIOU, C. **Computational Complexity**. Addison-Wesley, 1994. (Theoretical computer science). ISBN 9780201530827. Available from Internet: <https://books.google.com.br/books?id=JogZAQAAIAAJ>.

PAPADIMITRIOU, C.; STEIGLITZ, K. **Combinatorial Optimization: Algorithms and Complexity**. Dover Publications, 1998. (Dover Books on Computer Science). ISBN 9780486402581. Available from Internet: <https://books.google.com.br/books?id=cDY-joeCGoIC>.

PINHEIRO, T. F.; RAVELO, S. V.; BURIOL, L. S. A fix-and-optimize matheuristic for the k-labelled spanning forest problem. In: **2022 IEEE Congress on Evolutionary Computation, CEC 2022 - Conference Proceedings**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2022. ISBN 9781665467087.

RICARTE, J. P. **Grammar Constructive**. [S.l.]: GitHub, 2024. <https://github.com/jpricarte/grammar-constructive>.

RITT, M. The $k$-labeled spanning forest problem: instance analysis and effective heuristic solution. In: **Anais do LVI Simpósio Brasileiro de Pesquisa Operacional**. Fortaleza, Brasil: [s.n.], 2024.

TAILLARD, E. Benchmarks for basic scheduling problems. **European Journal of Operational Research**, v. 64, n. 2, p. 278–285, 1993. ISSN 0377-2217. Project Management anf Scheduling. Available from Internet: <https://www.sciencedirect.com/science/article/pii/037722179390182M>.

VOSS, S.; FINK, A.; DUIN, C. Looking ahead with the pilot method. **Annals of Operations Research**, Springer, v. 136, p. 285–302, 2005.

WILLIAMSON, D. P.; SHMOYS, D. B. **The design of approximation algorithms**. [S.l.]: Cambridge university press, 2011.

ZHOU, Z.; LIU, S. **Machine Learning**. Springer Nature Singapore, 2021. ISBN 9789811519673. Available from Internet: <https://books.google.com.br/books?id=ctM-EAAAQBAJ>.