LEONARDO DE ALMEIDA E BUENO

BIASED RANDOM-KEY GENETIC ALGORITHM FOR WAREHOUSE RESHUFFLING

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2018

Leonardo de Almeida e Bueno

BIASED RANDOM-KEY GENETIC ALGORITHM FOR WAREHOUSE RESHUFFLING

A M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science with emphasis in Operational Research

Advisor: Ricardo Martins de Abreu Silva

Recife
2018

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB 4-1758

B928b Bueno, Leonardo de Almeida e.
Biased random key genetic algorithm for warehouse reshuffling / Leonardo de Almeida e Bueno. – 2018.
139f.: fig.

Orientador: Ricardo Martins de Abreu Silva
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn. , Recife, 2018.
Inclui referências e apêndices.

1. Ciência da computação. 2. Pesquisa operacional. 3. Otimização. I. Silva, Ricardo Martins de Abreu. (Orientador). II. Titulo.

004 CDD (22. ed.) UFPE-MEI 2018-112

Leonardo de Almeida e Bueno

Biased Random Key Genetic Algorithm for Warehouse Reshuffling

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 08/08/2018.

BANCA EXAMINADORA

____

Prof. Dr. Silvio de Barros Melo
Centro de Informática/UFPE

____

Prof. Dr. Rodrigo Gabriel Ferreira Soares
Departamento de Estatística e Informática/ UFRPE

____

Prof. Dr. Ricardo Martins de Abreu Silva
Centro de Informática / UFPE
(Orientador)

I dedicate this thesis to my parents, who invested so much in my education. Without their wisdom, support and inspiration I would never achieve this or any step on my life.

ACKNOWLEDGEMENTS

I would like to express my special appreciation and thanks to my advisor Professor Dr. Ricardo Martins de Abreu Silva, for accepting me as a graduate student in his group even though I was a working student with less time available for the Masters. Ricardo was always giving me all the scientific freedom I wanted, encouraging me to new researches, and criticizing my ideas and results.

I would like to thank Miguel Domingos de Santana Wanderley and the Professors Dr. Cleber Zanchettin and Dr. Adriano Lorena Inacio Oliveira for the amazing collaboration that resulted in a published paper about deep learning. I am also grateful to Professor Dr. Abel Guilhermino da Silva Filho and his advisees Eronides Felisberto da Silva Neto and Hilson Gomes Vilar de Andrade for the collaborations and all the support, advice and encouragement given.

I had the opportunity to supervise two excellent students, Diogo Pereira de Morais and Danilo Dias Pena during their undergraduate senior projects. Their persistence and motivation contributed greatly to this project.

Many thanks to Mariana Alves Moura for the help with programming, the fruitful discussion, and the advice regarding my research.

Thanks a lot to all the friends I made during the masters: Antonio Luís do Rego Luna Filho, Raimundo Martins Leandro Junior, Diocleciano Dantas Neto, Eudes da Silva Barboza, Rodrigo Gomes de Souza, and many others. It's been a great joy to study and spend free time with you. Thanks as well to Ben Qureshi for the fantastic illustrations made.

I greatly appreciate all the support I received from Professor Dr. Jeroen Bergmann of the Natural Interactions Lab, and from my previous co-workers and employers at Tomus, that accepted my studies, motivated me on my personal research and supported me on busy moments.

A special thanks to my family. I am very grateful to my parents for all of the support and sacrifices that they've made on my behalf. My mother that always wisely advised me in the universe of academia and research, as well as on my personal life, and my father who always inspired my journey to improve the lives of those surrounding me. I am also very grateful to my siblings, aunts, and cousins for their patience on hearing my struggles with my work and my studies.

I would like to give an special thank my girlfriend who gave me the motivational push and the emotional support needed to conclude this research.

Ultimately, I am grateful to the Center for Informatics of the Federal University of Pernambuco and to all people and institutions that directly or indirectly participated in this phase of my academic life that I am pleased to conclude now.

"It is not enough to have a good mind; the main thing is to use it well."
(DESCARTES; ARIEW, 2000)

ABSTRACT

Due to its strategical importance, the efficient stock management in a warehouse presents several challenges that can be approached using optimization methods. In this universe, frequently explored problems are ambient dimensioning, department organization and layout, stock organization and layout, pilling design, product storage and recovery methodology. Design and operation imprecisions and failures can result in large delays in the product delivery or even in missing items in final client stocks. Among the main causes of missing items in inventories, there are the incongruity between storage capacity and refilling frequency; infrequency, delay, or nonexistence of product restitution in shelves; inexact or wrong inventories; storages with the inadequate organization, package disruption and scarce availability; poor storage layout and inefficient operational services. To determine the optimized product stocking is a problem frequently approached in the literature throughout the decades. However, the increasing need or changes in the storage, increase the importance of other problem: the sequence of movement to obtain a particular stock organization, given the current organization of the items. This problem is known as stock rearrangement, stock shuffling, or stock reshuffling. The optimization of package reshuffling in large warehouses directly impacts the profits. Large warehouses need, very frequently, to reorganize stock because of: seasonality, market changes, logistics, and other factors. Certain types of products have higher demand during specific periods of the year. Products on sale may leave the stock faster, new products may have higher output. All these are examples that justify a frequent stock reshuffling. Warehouse stock reshuffling consists of repositioning items by moving them sequentially. Several studies aim to solve reshuffling problems by applying exact methods. However, due to the complexity of the problem, only heuristics result in practical solutions. This study investigates how to optimize unit-load warehouse reshuffling in multiple empty locations scenarios. Traditional heuristics are reviewed and an evolutionary programming approach is proposed for the unit-load warehouse reshuffling problem. Experimental results indicate the proposed heuristic perform satisfactorily in terms of computational time and is able to improve solution quality upon benchmark heuristics.

Keywords: Optimization. Evolution Strategy. Genetic Algorithms. Warehouse Reshuffling. Logistics.

RESUMO

Devido à sua importância estratégica, a gestão eficiente de grandes armazéns apresenta diversos desafios que podem ser resolvidos via métodos de otimização. Neste universo, são frequentemente explorados pela literatura os problemas de: dimensionamento de ambientes, organização e layout de departamentos e estoques, padrão de empilhamento, metodologia de armazenamento e recuperação de produtos. Imprecisões e falhas de projeto e operação de armazéns podem resultar em grandes atrasos na entrega de produtos e até na falta de itens em inventários de clientes finais. Entre as causas principais de falta de inventário se encontram: incongruência entre capacidade e frequência de abastecimento; infrequência, atraso ou inexistência de reposição de artigos em prateleiras; inventário inexato ou errado; armazenamento com organização inadequada, rompimento de embalagens ou pouca disponibilidade; mal projeto do estoque e serviços operacionais ineficientes. Determinar a forma otimizada de armazenamento de produtos é um problema que vem sido estudado há décadas, porém, a cada vez mais frequente necessidade de mudança nos estoques trouxe um novo problema à tona: a sequência de movimento para obtenção de uma organização em particular, dado o estado atual das cargas no estoque. Este problema é conhecido como reorganização de estoque. Otimizar a reorganização de itens em grandes armazéns impacta diretamente e de forma positiva os rendimentos. Grandes armazéns frequentemente necessitam de reorganizações por motivos sazonais, de mercado, logísticos, etc. Determinados tipos de produtos tem maior demanda em uma época do ano do que em outras, produtos postos em promoção vão ser liquidados e vão sair do estoque mais rapidamente, novos produtos são recebidos constantemente nos depósitos, todos esses são exemplos que requerem uma reorganização frequente no estoque. Reorganização de pacotes em centros de distribuição consiste em reposicionar itens movendo-os sequencialmente. Vários estudos da literatura se propõem a solucionar problemas de reorganização de pacotes aplicando métodos exatos. No entanto, devido à complexidade do problema, apenas heurísticas obtém tempos de processamento viáveis para aplicações reais. Este estudo investiga como otimizar a reorganização de centros de distribuição de cargas unitárias em cenários onde existem múltiplas localizações vazias. Heurísticas tradicionais são revisadas e uma abordagem de programação evolucionária é proposta para o problema. Resultados experimentais indicam que a heurística proposta tem desempenho satisfatório em termos de tempo computacional e é capaz de melhorar a qualidade das soluções em comparação com heurísticas de referência.

Palavras-chave: Otimização. Computação Evolucionária. Algoritmos Genéticos. Reorganização de armazéns. Logística.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS AND ACRONYMS

ρ_a	Probability of inherit allele from first parent
ρ_e	Probability of inherit allele from elite parent
p	Population size
p_e	Elite population percentage
p_m	Mutant population percentage
$MAXGEN$	Maximum number of generations
$maxDist$	Maximum Distance
API	Application Programming Interface
ASRS	Automated Storage/Retrieval Systems
BRKGA	Biased Random-Key Genetic Algorithm
CP	Convergence Population Fraction
CSV	Comma-Separated Values
DCs	Distribution Centers
DE	Differential Evolution
DP	Dynamic Programming
DU	Distance Unit
EA	Evolutionary Algorithms
GA	Genetic Algorithm
GRASP	Greedy Randomized Adaptive Search Procedure
GRH	General Reshuffling Heuristic
H3	Heuristic 3
I/O	Input/Output
IRACE	Iterated Racing for Automatic Algorithm Configuration
K	Number of separated populations
MinGW	Minimalist GNU for Windows
NP-Hard	Non-deterministic Polynomial acceptable

P/D	Pickup/Drop-off
PSO	Particle Swarm Optimizer
RKGA	Random-Key Genetic Algorithm
RWW	Rearrange-While-Working
S/R	*Storage/Receive*
SA	Simulated Annealing
SGA	Simple Genetic Algorithm
SI	Shuffling with Insertion
SLAP	Storage Location Assignment Problem
SNN	Shuffling with Nearest Neighbor Heuristic
SSD	Solid State Drive

CONTENTS

1 INTRODUCTION

> *Rene Descartes: Divide each difficulty into as many parts as is feasible and necessary to resolve it.*

1.1 MOTIVATION

The supply chain is the collection of resources and methods required to plan, execute and control the production, storage, and delivery of goods and services from the origins to the final consumers. It involves several key activities and processes that must be completed in a cost-effective and timely manner to efficiently deliver products to the clients (ASGARI et al., 2016).

The whole chain is composed of a series of operators specialized in a specific step of the process. As an example, a manufacturer that fabricates products in a different country from the consumer market. From the manufacturer until the consumer, the items will be produced, transported, stored, distributed, and accessed by end consumers (ASGARI et al., 2016). Operators take roles in each of these phases, and they are all dependent on the other operators in the supply chain. An example of the flow of articles in such a chain is depicted in Figure 1.

The overall performance of a supply chain depends on its design and operation. Number, location, and capacities of manufactures, warehouses, Distribution Centers (DCs), and retailers; inventory control methodologies, storage facilities, and service quality; access to suppliers, transporters, resellers, distributors, are individual aspects that have important roles in the chain (RAJGOPAL, 2016).

Warehouses and large distribution centers are an essential part of the product supply chain. Design and operation imprecisions and failures can result in large delays in the product delivery or even in missing items in final client stocks. The study conducted by Corsten e Gruen (2004) over a population of 71,000 consumers in 29 countries indicate that clients will recur to other suppliers between 21% and 41% of the times, if they find a missing item in the inventory, resulting in a loss of at least 4% for a retailer.

Some of the main reasons for missing items in inventories are the incongruity between storage capacity and refilling frequency (replenishment); infrequency, delay, or nonexistence of product restitution in shelves; inexact or wrong inventory control; storages with an inadequate organization, package disruption and scarce availability; poor storage layout and inefficient operational services (GRUEN et al., 2002). Delays in one point of the supply chain can result in considerable losses for a final retailer. Losses for poor storage can represent up to 10% of the final losses due to stock faults. This means, at any point where there is a storage for raw materials or manufactured products, there is an

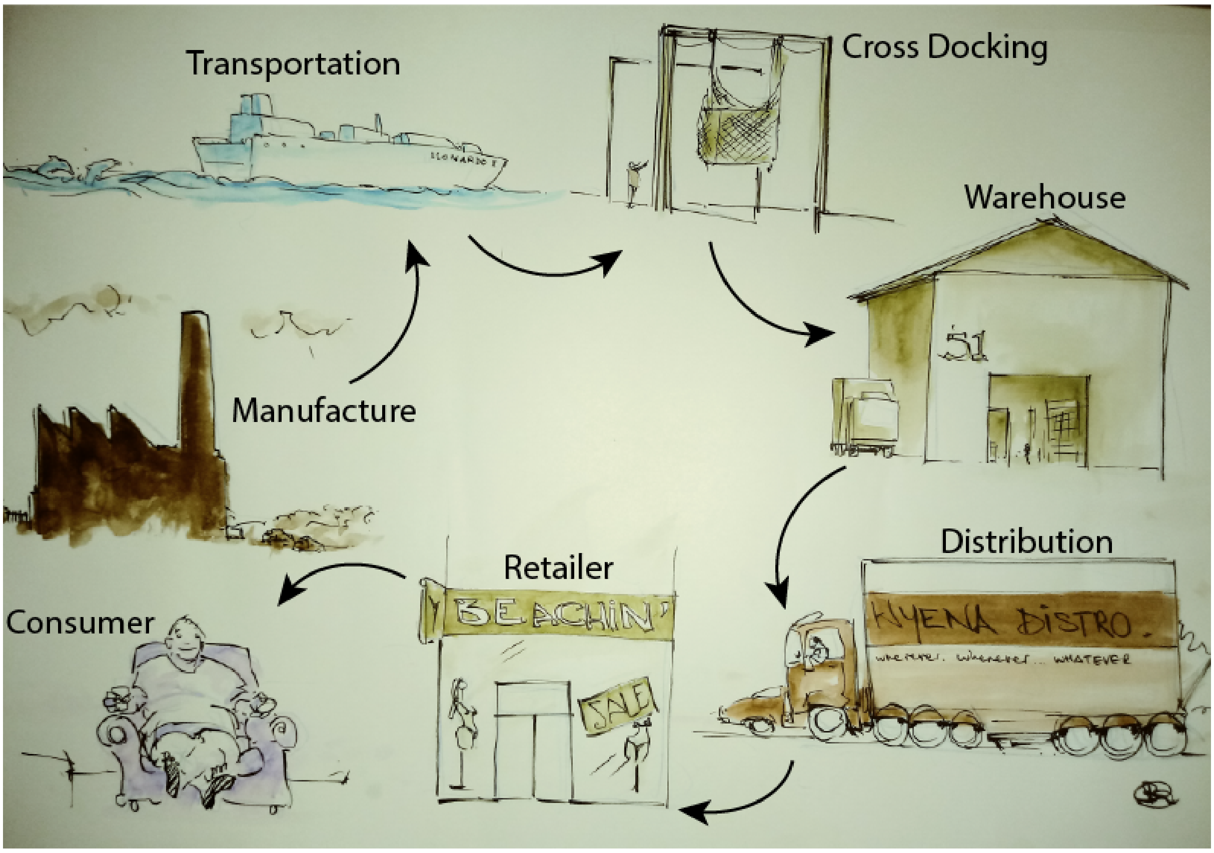

Figure 1 – Flow of items in a supply chain.

opportunity to improve the timing in which the orders are fulfilled, by optimizing the storage.

Several points in the supply chain include warehouses, DCs, and storages including the manufacturer, the transporter, and the distributor. The storages can have racks systems, individual product placements, container terminals, among others. Therefore, it is possible to improve the flow of products from the manufacturer to final users by improving the design and operation of storages, of any type. Due to its strategical importance, the efficient stock management in a warehouse contains several problems that can be approached using optimization methods. In this universe, frequently explored problems are ambient dimensioning, department organization and layout, stock organization and layout, pilling design, product storage and retrieval methodology (GU; GOETSCHALCKX; MCGINNIS, 2007), (GU; GOETSCHALCKX; MCGINNIS, 2010).

An efficient storage operation within a supply chain greatly requires an effective organization of the stock. A disorganized storage will have products in unassigned locations, resulting in losses of time for storage and retrieval, unnecessary use of tools and equipment, inadequate use of space, additional replacements of items, and low productivity, resulting in profit losses and affecting the competitiveness of the organization and of those that rely on it.

One of the most frequently studied problems within this context is the efficient Storage Location Assignment Problem (SLAP). This class of problems is defined as "the assignment of locations of products inside a storage in order to minimize the costs related to handling the items during daily operation" (HAUSMAN; SCHWARZ; GRAVES, 1976). These problems can be found from shelves systems in final products to container terminals, and pallet racks. Fortunately, SLAP has been largely investigated and is solved using different policies for location assignments (GU; GOETSCHALCKX; MCGINNIS, 2007) intended to minimize travel distances, travel time, or energy required to access locations and items. The studies by Gu, Goetschalckx e McGinnis (2007), Koster, Le-Duc e Roodbergen (2007), Roodbergen e Vis (2009), and Gu, Goetschalckx e McGinnis (2010) provide extensive reviews of the warehouse operational problems, including the most commonly used policies to solve the SLAP.

In most of the cases, these policies are based on the item demand, and it is inevitable that demand profiles change over time (KOSTER; LE-DUC; ROODBERGEN, 2007). The demand profiles can change due to competition, new products in the market, product maturity or seasonality (CARLO; GIRALDO, 2012). Consequently, the best arrangement of the items in a stock changes with time.

To determine the new best arrangement, the new demand profiles are used and the SLAP problem is solved once again. This process creates a new problem: the sequence of movements to efficiently obtain a particular stock organization, given the current organization of the item in the storage. This problem is known as stock rearrangement, stock shuffling, or stock reshuffling. The reshuffling activities' frequency varies. Daily, weekly, monthly, quarterly and semiannual reshuffling policies are adopted depending on the type and size of the warehouse and the supplied demand profile.

The optimization of storage reshuffling in warehouses directly impacts the profits by keeping the storage best arranged to the demand and consequently reducing losses due to delays in product storage and recovery operations. The reshuffling can be especially important for large warehouses with larger storage units. In these scenarios, improvements between 8–15% in storage and retrieval converts in savings of up to \$500,000 per year based on a 2011 evaluation (TREBILCOCK, 2011). This costs should be balanced by the reshuffling costs, that include manpower (in manual storages) and electricity (in automatized storages). In both cases, the reshuffling costs can be minimized through the reduction in the total time needed for the process.

1.2 PROBLEM STATEMENT

As described in Christofides e Colloff (1973) the reshuffle problem is to find a sequence of movements to be executed that will transform the initial arrangement of **K** items in a storage (I_K) to some specified final arrangement (F_K), and that will minimize the total cost involved.

In the warehouse reshuffling universe, the items that are relocated may be stored in pallets, as in most warehouses and in the reserve area of DCs, or in totes as in mini load Automated Storage/Retrieval Systems (ASRS) and in the forward area of DCs where picking occurs (KOSTER; LE-DUC; ROODBERGEN, 2007). The items may be distributed in several aisles, and frequently the access to these aisles is controlled and regulated by proper entrance and exits, traffic direction, and even driving speeds. As a result, accidents are avoided when storing and recovering packages (GU; GOETSCHALCKX; MCGINNIS, 2007).

To simplify the design and analysis, this study focuses on a system where items are palletized and stored in a single rack that is served by a single material handling equipment as the one depicted in Figure 2. Without loss of generality, the main assumptions restricting the problem studied are:

1. Items are carried as unit-loads;
2. Each location may store only one item;
3. Each item has a unique storage location (i.e., dedicated storage policy); each copy of an item is treated as a unique item that has a specific location in the initial and final storage configurations;
4. The initial and final storage configurations are known;
5. Reshuffling is performed by a single material handling equipment;
6. The travel distance between any two storage locations is assumed to be known;
7. Only one rack (i.e., one side of the aisle) served by the *Storage/Receive* (S/R) machine is considered;
8. Every item is directly accessible from the aisle (i.e., a single-deep aisle);
9. The Input/Output (I/O) point is known and considered as a location in the rack;
10. All moves can be completed in the time available;
11. The objective is to minimize the total movement cost measured as the distance traveled for both loaded and unloaded movements.

The objective function in the last assumption can be easily modified from travel distance to travel time by incorporating the travel speed, acceleration/deceleration, and Pickup/Drop-off (P/D) times. Alternatively, if P/D's are to be incorporated, a fixed distance-penalty may be added for each P/D. Travel distance metrics may also be altered to correspond with different storage layouts.

The studied methodologies, though, are directly implementable for manual or automated warehouses and can be later expanded to consider double-handling of materials

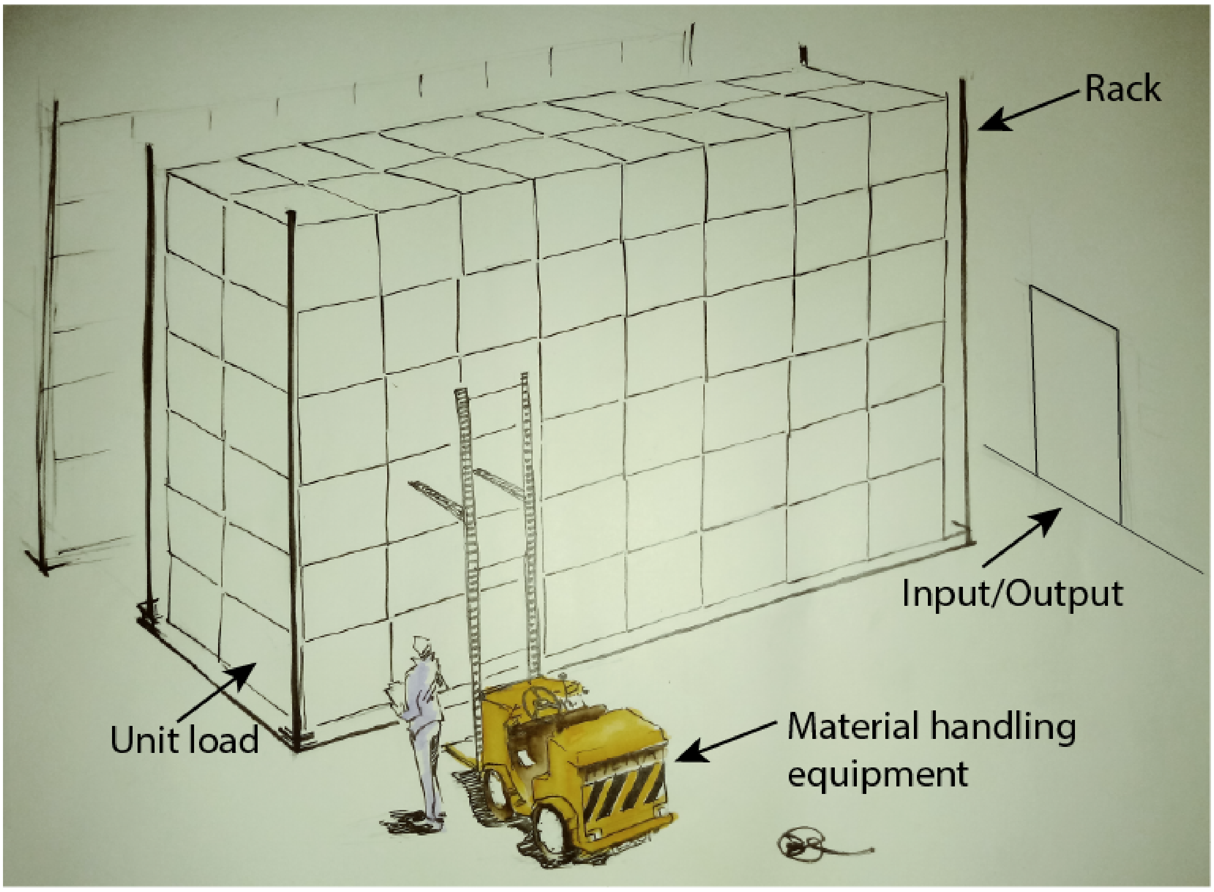

Figure 2 – Single rack with material handling equipment.

and heterogeneous loads. The main modeling assumptions used in this study are consistent with the traditional assumptions in the warehousing reshuffling literature (PAZOUR; CARLO, 2015), (GIRALDO, 2011). Each of these assumptions can be relaxed, generating new studies to identify strategies to approach the derived problems.

Figure 3 depicts a sample reshuffling problem in which four items (A–D) require repositioning. The required solution is the order of movements to be executed by the material handling machine to reshuffle an item from the initial storage configuration depicted in Figure 3 (a), to the final storage configuration depicted in Figure 3 (b).

Items A, B, and C in Figure 3 are referred to as cycle items because to reposition any of these items the other items need to be moved. The final location of item A is initially occupied by item C. The final location of item C is initially occupied by item B, and the final location of item B is occupied by item A. Therefore, to reposition these items, it is necessary to break the cycle moving one item from its initial location to an intermediary location different from its final location. This additional step allows moving sequentially the remaining items in the cycle to their final location before moving the first item to its final location. A set of items is classified as cycle items when the set's initial locations are equal to the set's final locations. A larger set may be decomposed into a union of disjoint subsets that denote individual cycles. The cycles are a property of the problem that was initially identified in the study of Christofides e Colloff (1973) and is frequently used in the literature to simplify the problem modeling and the design of solutions.

The remaining item (D) is a non-cycle item because it is not part of any cycles as it can be directly moved from its initial location (Loc 1) to its final location (Loc 5).

In addition to the items, the problem contains two open locations (represented by 0_1 and 0_2 in the initial and $0'_1$ and $0'_2$ in the final configurations). The I/O point is assumed to be at the bottom leftmost location (i.e., location 0, labeled as Loc 0 in Figure 3). A possible solution to this problem is to move unloaded from the I/O point (Loc 0) to the initial location of item B. Then reposition item B from its initial location (Loc 4) to the open location identified as 0_1 in Loc 5. Next, move unloaded to the initial location of item C (Loc 0), pick up item C in and move it to its final location (Loc 4). Then move unloaded to item A (Loc 3) and move it to its final location (Loc 0). At that point, item B can be moved from location 5 to its final location (Loc 3), followed by the repositioning of item D (From location 1 to location 5). For a solution to be feasible, an item cannot be moved to a location unless the location is open. However, the open location changes as the items are being reshuffled. Consequently, there are multiple feasible solutions to the sample problem, which increase exponentially with the number of items and open locations considered.

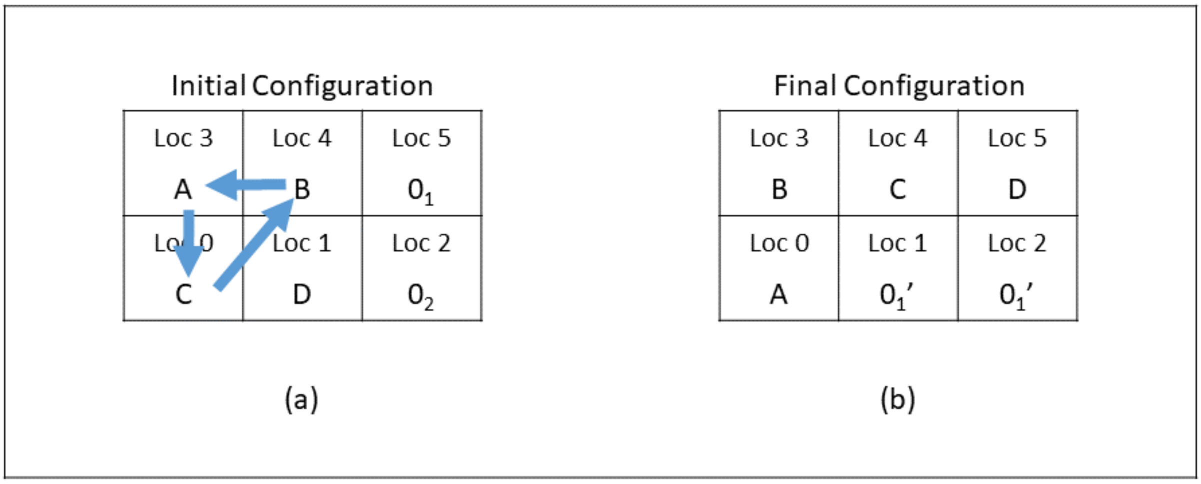

Figure 3 – The initial (a) and final (b) configurations for a sample reshuffling problem.

1.3 STATEMENT OF THE CONTRIBUTIONS

This study has as main contributions:

1. A genetic algorithm based on the BRKGA metaheuristic (GONÇALVES; RESENDE, 2011) for solving unit-load warehouse reshuffling problems in large storages;

2. Validation of the proposed heuristic by benchmark comparison with recent literature heuristics successfully applied to the problem;

3. A warehouse reshuffling scenario generator for benchmark testing of reshuffling optimization algorithms.

1.4 ORGANIZATION OF THE DISSERTATION

The remainder of the dissertation is organized as follows:

- Chapter 2 presents a review of relevant literature pertaining to the reshuffling problem;

- Chapter 3 introduces genetic algorithms and their random-key variations used in this project and describes the methods used to apply the metaheuristic in reshuffling problems;

- Chapter 4 presents the experiments performed to adjust the parameters of the reshuffling heuristic built;

- Chapter 5 analyzes the experimental results obtained with the developed heuristic in comparison with literature benchmark solutions.

- Chapter 6 presents the conclusions extracted from the study and future research.

2 LITERATURE REVIEW

The concept of warehouse reshuffling was initially proposed by Christofides e Colloff (1973) who referred to it as "*warehouse rearrangement*". This study assumes problems as exemplified in Figure 4. The problems have one empty location within the warehouse (represented by O_1 in the figure) and all items contained in *cycles* (exemplified in the figure with one cycle with items A, B and C, and one cycle with items D and E). Furthermore, it is assumed that items in a cycle are moved sequentially (i.e., once an item that is part of a cycle is moved, the remainder of the items in the cycle have to be moved). The paper hypothesizes that the position of the open locations remains fixed throughout the problem since only cycles are considered and that the cycles must be executed separately, one after the other. The same open locations will be available before and after the reshuffling.

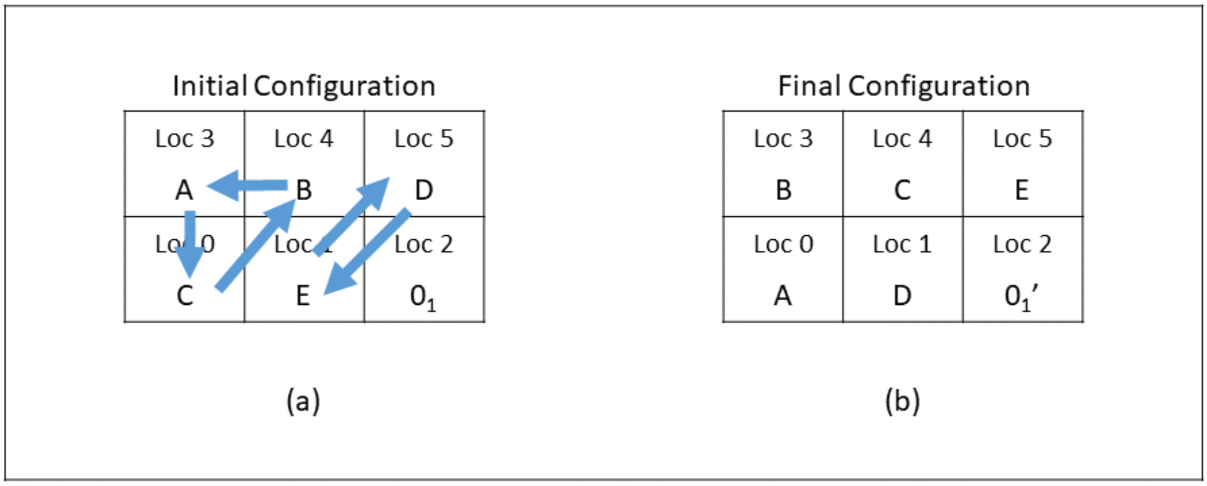

Figure 4 – The initial (a) and final (b) configurations for a sample reshuffling problem with open location and two cycles.

The authors propose a two-stage algorithm that will sequence load movements by minimizing the travel costs required to rearrange the products in a dedicated warehouse. The first stage identifies how each of the cycles can be repositioned, whereas the second stage uses Dynamic Programming (DP) to determine the sequence in which the cycles are moved. The DP algorithm by Christofides e Colloff (1973) is capable of finding the optimum solution for the simplified problem scenario, but, as later found by Muralidharan, Linn e Pandit (1995), the problem with non-cycle items is Non-deterministic Polynomial acceptable (NP-Hard), and the solution space for their DP-based method grows exponentially with the number of cycles and empty locations such that the algorithm becomes impractical. As illustrated in Figure 5, the problem addressed here is similar to the problem studied in Christofides e Colloff (1973), but relaxing the assumptions of having only one empty location and only cycles that must be executed sequentially. By relaxing these assumptions the problem becomes more complex as open locations change throughout the reshuffling process and non-cycle items need to be considered individually.

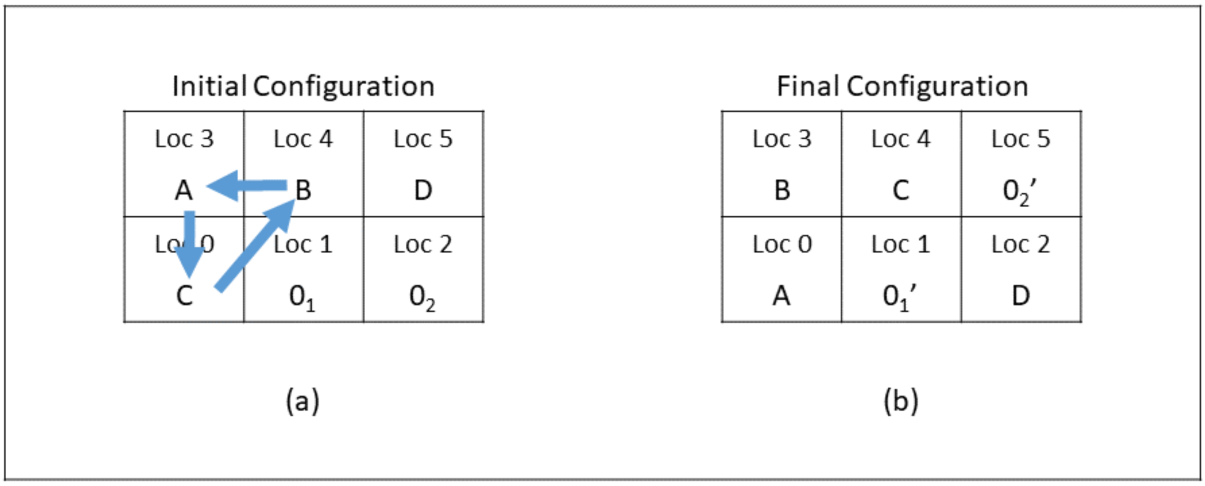

Figure 5 – The initial (a) and final (b) configurations for a sample reshuffling problem with two open locations, one cycle, and one non-cycle item.

The first sub-optimum solution applied to this problem was proposed by Muralidharan, Linn e Pandit (1995). In this study, the problem was formulated as a Precedence Constrained Selective Asymmetric Traveling Salesman Problem and, given the computational complexity of the problem, the authors proposed two heuristics: the Shuffling with Nearest Neighbor Heuristic (SNN) and the Shuffling with Insertion (SI). Based on simulation results the authors conclude that using idle times to update the warehouse configuration increases the storage/retrieval process efficiency. One important assumption used in by Muralidharan, Linn e Pandit (1995) is that the open location for each item is preassigned and available, which allows them to focus only on minimizing unloaded travel.

Chen, Langevin e Riopel (2011) focuses on relocating items in a warehouse by simultaneously deciding which items are to be relocated and their relocation destination in order to satisfy the required throughput during peak periods. A mathematical model for the problem and two heuristics are presented, a two-stage heuristic and a Tabu Search. Since Chen, Langevin e Riopel (2011) considers the destination as a variable of their problem, the nature of their problem is different than the one studied in this project.

Carlo e Giraldo (2012) introduces the Rearrange-While-Working (RWW) strategy. The RWW concept is to reposition pallets by storing them in a different location than where they were retrieved from. Hence, upon retrieving an item, a decision of where to store it is made considering the open locations, the desired final location of the item, and the set of retrieval movements required to serve a predetermined number of orders. Genetic Algorithm is used to find the sequence of repositions that minimizes the total travel costs. However, as the scheduled retrievals may not suffice to complete the rearrangement of all items, a polynomial-time heuristic called Heuristic 3 (H3), similar to the SNN heuristic in Muralidharan, Linn e Pandit (1995), was used to estimate the remaining work after serving all orders by performing reshuffling. H3 assumes that non-cycle items are moved before cycle items and that items in a cycle are repositioned sequentially. Since the H3 is sub-optimum, to compare its results with optimum solutions, Giraldo (2011) proposes a dynamic programming algorithm based on the branch and bound approach. This DP algorithm could not be applied in real scale problems due to its exponential-time complexity.

More recently, Pazour e Carlo (2015) proposed 4 different mathematical models for reshuffling operating policies. These models include the original formulation by Christofides e Colloff (1973), and additional formulations where non-cycle items are also handled. It was indicated that the formulation where cycle items are treated sequentially before non-cycle items, returns the best results. The new formulations proposed by Pazour e Carlo (2015) are a good reference for optimum solutions to small-scale problems. However, as the problem scale increases, these solutions also demand impractical processing times. To overcome these limitations Pazour e Carlo (2015) proposes the GRH, which is based on the H3 but relaxes the assumption that non-cycles are moved before cycles. This is achieved by introducing a parameter τ that allows breaking nearby cycles in between non-cycle movements. In addition, Pazour e Carlo (2015) proposes a Simulated Annealing (SA) adapted from the one elaborated in Wilhelm e Ward (1987). It was found that the GRH algorithm results in better solutions than the benchmark heuristic H3 with similar processing times. The SA approach reported respectable solutions in small and medium scales. However, Pazour e Carlo (2015) demonstrated statistically that the algorithm is not scalable to large problems.

As the most successful approaches reported so far in the literature for the reshuffling problem studied, H3 from Carlo e Giraldo (2012) and the GRH from Pazour e Carlo (2015) were used as inspiration for a Biased Random-Keys Genetic Algorithm (BRKGA) reshuffling decoder and as benchmark solutions. The BRKGA is a genetic algorithm recently successfully applied in several combinatorial applications. Chapter 3 details the main concepts behind this metaheuristic with its differences to the classical genetic algorithms, and introduce the modifications added to solve reshuffling problems.

3 METHODS

As detailed in Chapter 2, several heuristic approaches were suggested to solve reshuffling problems. The main meta-heuristic paradigm applied to these problems was the Simulated Annealing (SA). However, as shown in Pazour e Carlo (2015), the SA approach evaluated significantly fewer candidate solutions once the scale of the problem grew. Even after running for 10 hours, only approximately 13% of the candidate solutions for instances with 100 locations were considered. For this reason, the authors decided not to increase run-times for the heuristic.

To overcome the apparent limitation of the SA, this study proposes the use of a Genetic Algorithm. As observed in the literature, even though for some problems the SA paradigm has better performance, such as in learning fuzzy cognitive map (GHAZANFARI et al., 2007), and integrated process routing and scheduling (BOTSALI, 2016), for combinatorial problems similar to the warehouse reshuffling, the Genetic Algorithm (GA) paradigm resulted in significantly better performance, especially with increasing problem sizes (MANIKAS; CAIN, 1996), (NAIR; SOODA, 2010), (ADEWOLE et al., 2012).

Within the GA heuristics available, this study focuses on using the Biased Random-Key Genetic Algorithm because of the significant performance gain reported from this approach in comparison with more traditional GAs (MOURA, 2018).

The following sections detail the main references and developed methods of this study. The BRKGA heuristic is described and each of the features added for solving reshuffle problems are introduced. Two heuristics are analyzed as inspirations for the BRKGA reshuffling decoder. The first reference heuristic for the reshuffling problem is the H3 from Carlo e Giraldo (2012), which implicitly assumes that items not in a cycle are repositioned before items in cycles, that items within a cycle are repositioned sequentially, and that double handling is not considered other than to break cycles. The implicit assumptions in H3 are those atone with the current rule-of-thumb in practice. Next, it is the GRH from Pazour e Carlo (2015), which relaxes these assumptions to obtain better results than those of Carlo e Giraldo (2012). The development is completed with the stopping criteria used to reduce the processing time of the Reshuffling BRKGA.

3.1 GENETIC ALGORITHM

Genetic Algorithms were introduced by Holland (1975) as a particular class of Evolutionary Algorithms (EA). These algorithms use techniques inspired by the Darwinian evolutionary biology (CHARLES, 1859) as inheritance, mutation, natural selection, and sexual reproduction using crossover. As explained by Goldberg (2006), Genetic Algorithms use computer models of the natural evolutionary processes as a tool to solve optimization problems. Although several variations exist within the proposed models in the literature, all have in common the concept of simulating a population of individuals with different characteristics, determined by their genes. Some characteristics are favorable for the environment in which they are inserted, while others are not. The GA transfers a group of the best performing solutions to a problem (fittest individuals in an environment) to a new population in a process analogous to the natural selection. These individuals can suffer modifications through genetic operations (mutation and crossover) in their chromosomes. The main idea is on the course of subsequent iterations, the worse individuals are discarded, therefore only the best individuals in the population remain.

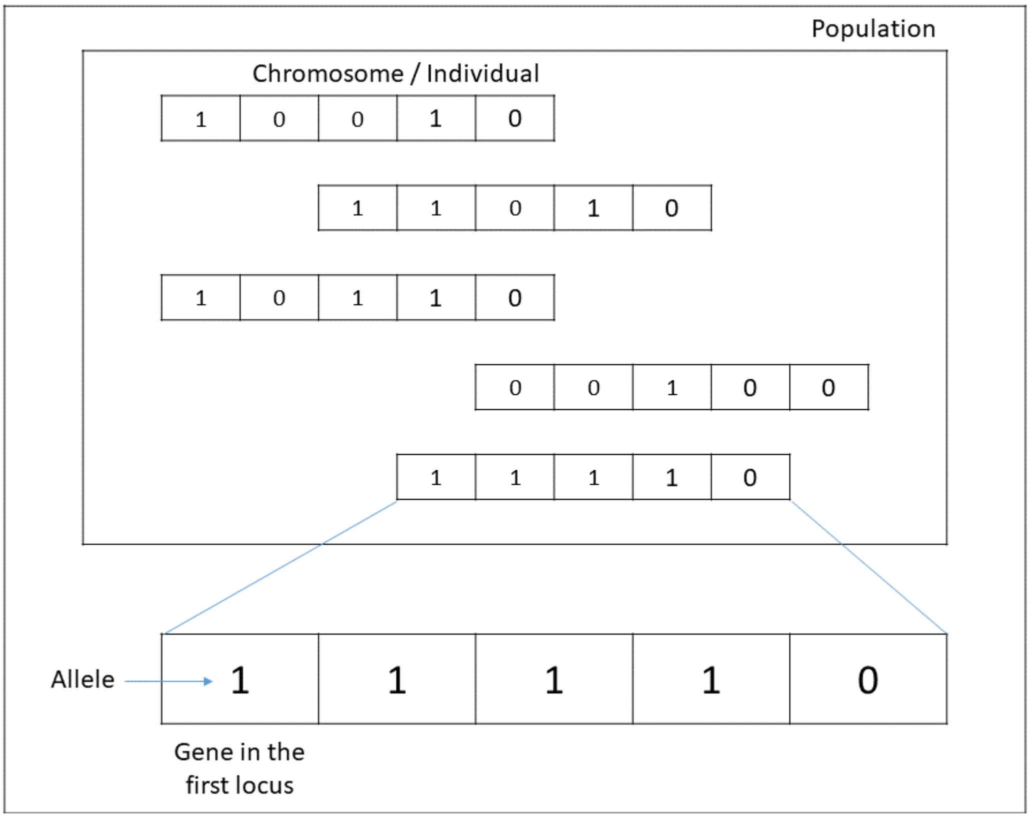

Figure 6 – Terminology used in genetic algorithms.

As shown in Figure 6, Genetic Algorithms are inspired by evolutionary biology terms combined with optimization concepts. One solution to the problem is referred to as an individual. Each individual is associated with a n size vector named *chromosome*. Each entry in the vector is known as *gene*, an its value is referred as *allele*. The position each gene occupies in the chromosome is called *locus*. The first entry in the gene vector is referred to as the first locus. Each gene represents a characteristic of the individual. A group of individuals (chromosomes) forms a population.

The selection operator picks the chromosomes that will take part in the reproduction process to combine their characteristics and generate new individuals. An objective function is applied to quantify the *fitness* of each individual (solution) in a population in the given evaluation environment (problem). The solutions better adapted for the problem (fittest), usually have higher probabilities of being selected for reproduction, transmitting their characteristics to future generations.

The crossover operator combines two parent chromosomes to create a new child chromosome by imitating a biological sexual reproduction of organisms. One example of a crossover operation is illustrated in Figure 7. This example operator is known as point crossover, in which a cutting point is determined to divide the parent's chromosomes into two parts. Two offspring are then generated by receiving one part from the first parent and another part from the second parent.

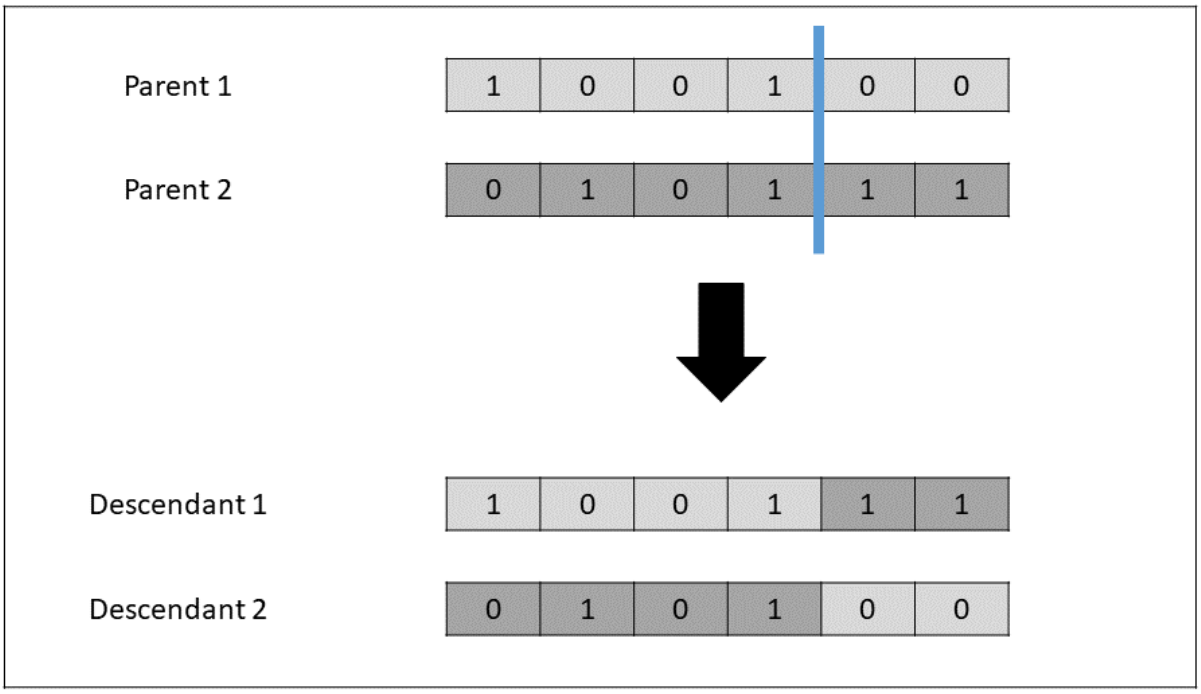

Figure 7 – Example of point crossover.

The mutation operator changes the values of some randomly selected alleles increasing the diversity in the population. This avoids a quick convergence to a local optimum. A mutation operator, for example, can randomly select a locus and alter its associated allele. Considering a Simple Genetic Algorithm (SGA), where chromosomes are represented by binary vectors, if the allele has a 0 value, it will become a 1, and vice-versa. Figure 8 illustrates this procedure. In this example, the second locus had its allele altered from 1 to 0.

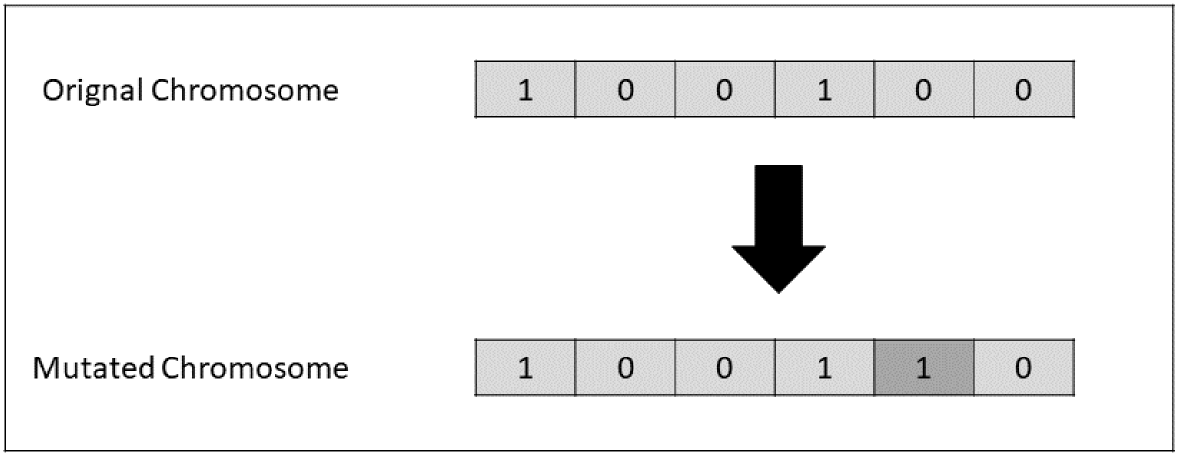

Figure 8 – Example of mutation operation.

The last and most important aspect of the GA is to define the objective function to quantify the fitness of each chromosome from the information contained in its genes. This is problem-specific and will greatly vary according to the algorithm design.

A pseudo-code of a traditional genetic algorithm is detailed in Algorithm 1.

Algorithm 1 Generic pseudo code for Genetic Algorithms

1: **procedure** GENETIC ALGORITHM
2:     Initialize starting population P;
3:     **while** Stopping criteria not met **do**
4:         Evaluate fitness for each individual in P;
5:         Select parents for reproduction;
6:         Perform reproduction via crossover;
7:         Perform mutation;
8:         Generate new population;
9:     **end while**
10:     **return** Fittest individual in the population
11: **end procedure**

At the second line, a starting population is initialized. This is usually done by randomly generating individuals. Next, from the third line to the ninth line, the main evolutionary process occurs. At the fourth line, the fitness of each individual is quantified using the objective function. At the fifth line, the selection operator is applied to pick parents to participate in the reproduction process at the sixth line. At the seventh line, mutation is applied to increase the diversity in the solutions. At the eighth line, the new offspring and mutant individuals are combined to form the next generation. This procedure is repeated until a stopping criterion is reached. This stopping criterion can be a maximum number of generations (iterations), a threshold number of generations with no improvements, among others. Several stopping criteria are analyzed in Zielinski, Peters-Drolshagen e

Laur (2005). Finally, at the tenth line, the algorithm returns the best solution found.

A common problem is the generation of inviable solutions after the application of mutation and crossover operations. In a sequencing problem, where a solution is represented by the permutation of some values without repetitions, the point crossover exemplified in Figure 9 would generate two inviable children solutions, because repetitions would occur. To overcome this problem in the GA, many authors developed algorithms highly dependent on the problems they proposed to solve (GOLDBERG; LINGLE et al., 1985), (GREFENSTETTE et al., 1985), (GREFENSTETTE, 1987), (CLEVELAND; SMITH, 1989). Intending to create a genetic alternative without this inviability problem, (BEAN, 1994) proposed the random-key strategy shown in Section 3.1.1

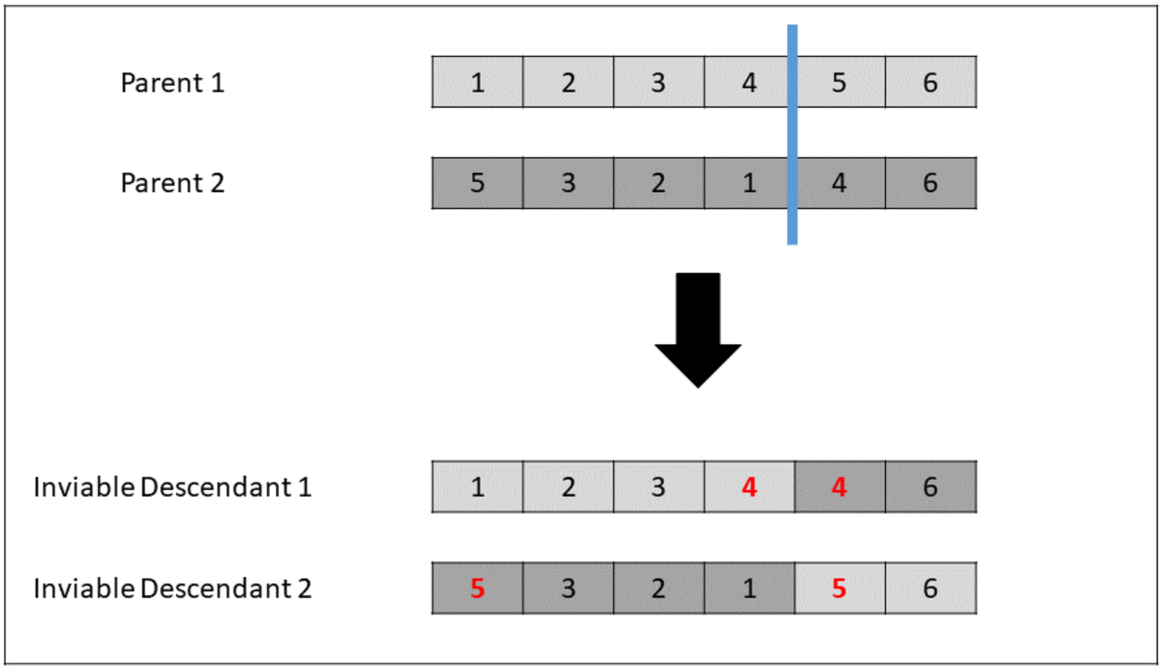

Figure 9 – Example of inviable offspring generated by point crossover.

3.1.1 Random-Key Genetic Algorithm

A Random-Key Genetic Algorithm (RKGA) is an evolutionary metaheuristic for combinatorial optimization problems introduced by (BEAN, 1994). The RKGA is based on the solution representation through a vector of n random keys, in which each key is a real number randomly generated according to a uniform distribution in the continuous interval [0,1).

The solutions (chromosomes) represented by the random-key vectors pass through a decoder responsible for mapping the keys into a viable solution for the problem and return its cost (fitness). The mapping process is illustrated in Figure 10, where on the left side is the continuous n-dimensional unit hypercube and on the right side is the solution space for the problem. The decoder located between both spaces connects each random-key vector to a problem solution and calculates its fitness.

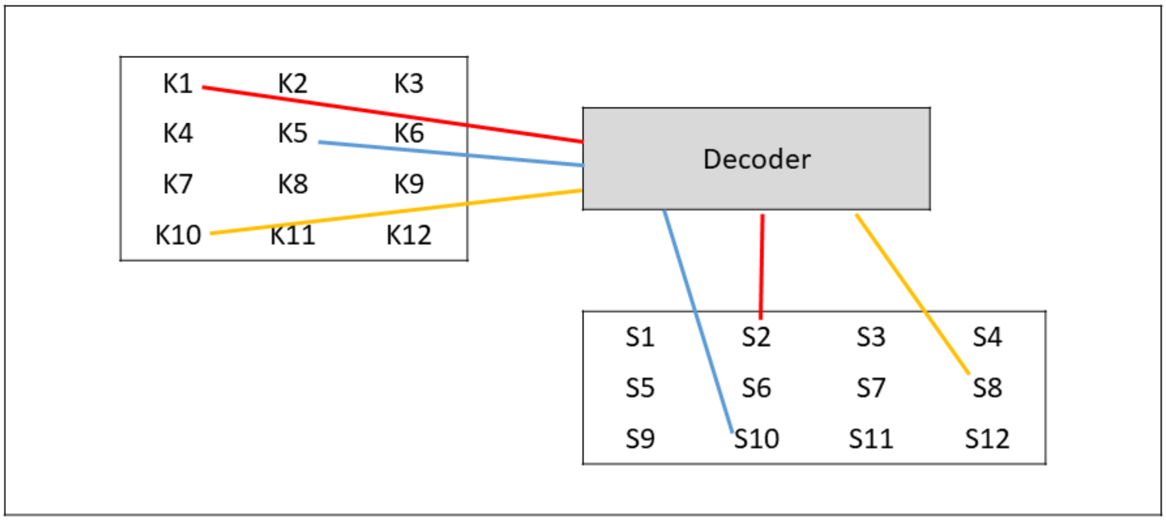

Figure 10 – Decoder used to map solutions in the random-key hypercube to solutions in the solution space where fitness is computed.

The decoding process is exemplified in the Algorithm 2. In this example, the decoder has to convert the random-key vector into an integer vector of length 6 with values varying from 0 to 100. The problem has a constraint that forces 3 vector positions of the solution to be 0.

Algorithm 2 Example Decoder

1: **procedure** DECODER(*chromosome*)
2: Copy the random-key vector represented by *chromosome* into the new vector *keys*;
3: Sort in increasing order the vector *keys*;
4: Multiply the first 3 sorted elements in *keys* to 100 and convert them to integer;
5: Verify in the initial *chromosome* the index of the first 3 sorted elements in *keys*;
6: Define a new vector *solution* of length 6 and attribute the integer values to the first 3 sorted elements in *keys* at the indexes found in the previous step.;
7: Define the remaining positions of the next vector as 0;
8: Calculate fitness of vector *solution*;
9: **return** fitness of vector *solution*
10: **end procedure**

Through this process, the decoder receives the random-key vector (the chromosome) at the first line; creates the vector *keys* with a sorted copy of the chromosome at the second and third lines; obtains the corresponding integer values at the fourth line; and at the fifth line verifies the corresponding indexes in the original chromosome of the first 3 sorted elements in vector *keys*. At the sixth line, the algorithm attributes the 3 integer values obtained previously to the original indexes of the keys. The remaining positions of the final vector are set to 0 at the seventh line. This new vector is the decoded solution and can be used to find the corresponding fitness to the problem at the eighth line. Following this process, the decodification process always results in a viable solution. One example of this decodification process is numerically illustrated in Figure 11.

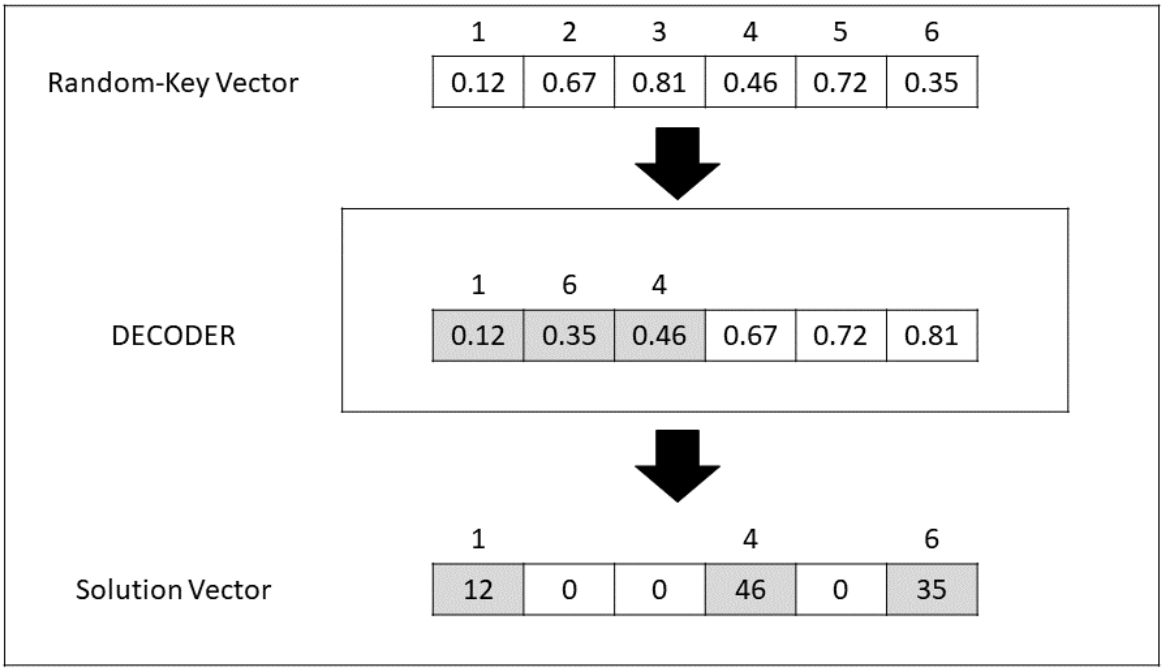

Figure 11 – RKGA randon-key decodification example.

In the RKGA, mutation and crossover operator are applied in the random-key vector before the fitness evaluation, not affecting the decoding process. By elaborating a decoder that always converts the random-key vector into viable solutions, the resultant algorithm does not produce non-viable solutions.

The evolution of the population (set of Population size (p) random-key vectors) is done based on the Darwinian principle, in which the fittest individuals have higher chances of passing their genetic information to future generations. This is due to higher chances of selection to generating offspring in reproduction phases and being copied as elite individuals.

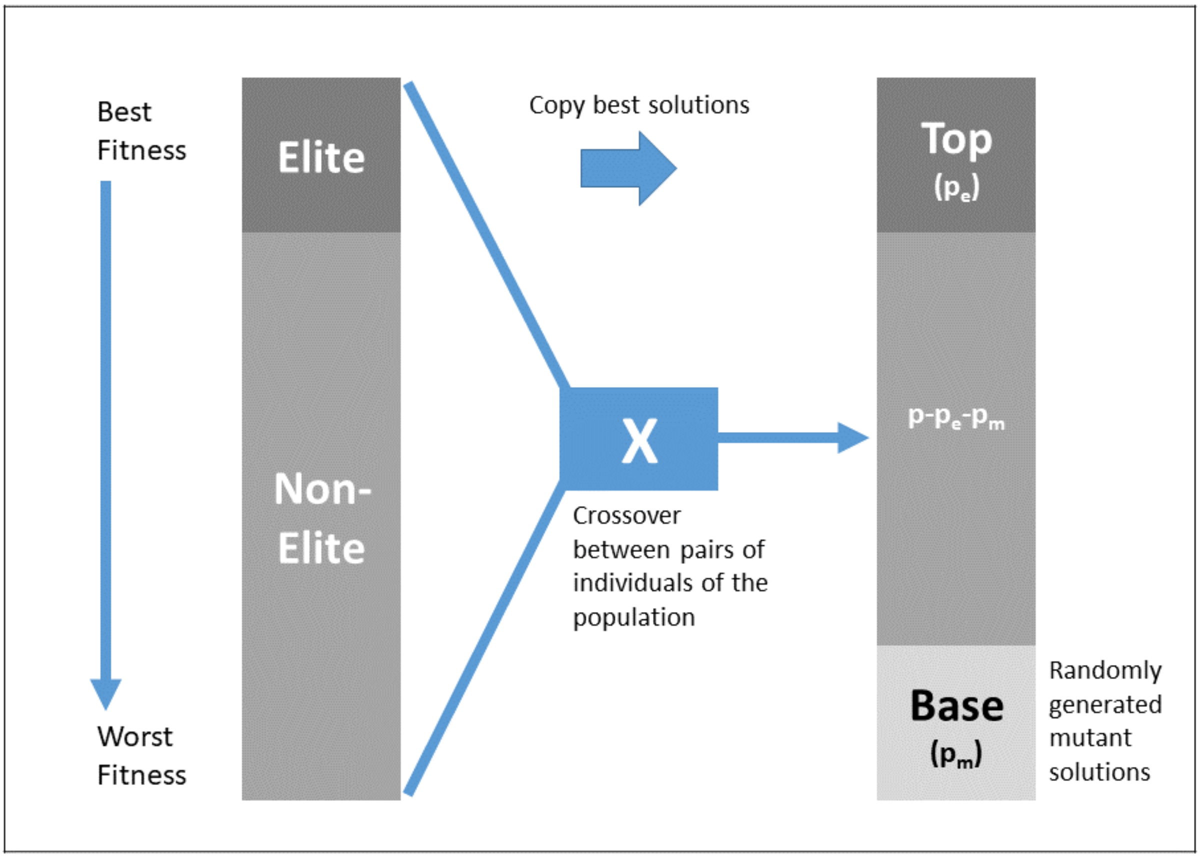

Figure 12 – Creation of new generation in the RKGA.

As illustrated in Figure 12, the p individuals of the population are divided into two groups at the end of each generation: the Elite population percentage (p_e) with the best solutions in the population, where $p_e < p/2$, and the non-elite group. The elite individuals are copied to the next population, applying the Darwinian elitism. Next, a Mutant population percentage (p_m) is generated and added to the future generation to guarantee diversity. A mutant individual is just a random-key vector generated in the same way as initial individuals are generated. Finally, to complete the new population, the remaining p - p_e - p_m individuals are generated combining pairs of randomly selected parents in the current population. The parents are combined using the uniformly parametrized crossover proposed by Spears e Jong (1995), illustrated in Figure 13.

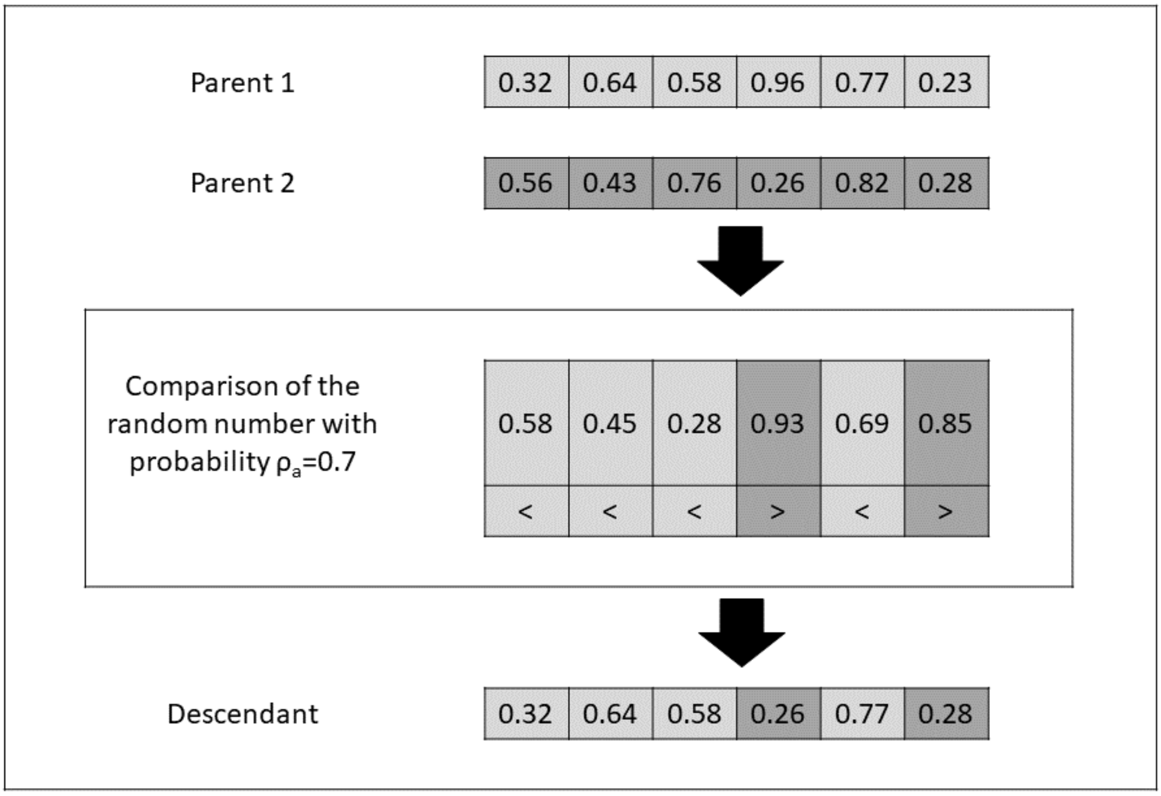

Figure 13 – Parametrized uniform crossover.

For this crossover process, at each chromosome position, a random number is generated and compared with the Probability of inherit allele from first parent (ρ_a) (parameter of the algorithm). If the number is lower than ρ_a the allele of the first parent (a) is inherited by the offspring. Otherwise, the allele of the second parent is inherited by the offspring. In Figure 13, given $\rho_a = 0.7$, if the random number is smaller than 0.7, the offspring receives the allele of parent a. Otherwise, it receives the allele of parent b.

The RKGA runs until a stopping criterion is met, then it returns the best solution found so far.

3.1.2 Biased Random-Key Genetic Algorithm

The BRKGA is a variant metaheuristic of RKGA proposed by (GONÇALVES; RESENDE, 2011). As illustrated in Figure 14, the dynamic evolution of BRKGA is similar to that of RKGA. The population is divided into elite and non-elite groups. The elite group is copied to the next generation. A number p_m of new individuals is randomly generated and added to the new generation. The main innovation in comparison with the RKGA is in the selection of the parents for the crossover operation. The BRKGA always opts for one elite parent (pe) crossing with one non-elite parent. In some cases, the second parent is selected from the entire population. This characteristic makes the BRKGA biased towards elitism. The repetition of parents is allowed in the reproduction phase, allowing then one parent to have more than one offspring. Since $p_e < p/2$, the probability of one elite individual being selected for crossover $(1/p_e)$ is larger than a non-elite individual $(1/(p - p_e))$. Therefore, increasing the chances of elite individuals to pass their genetic material to future generations.

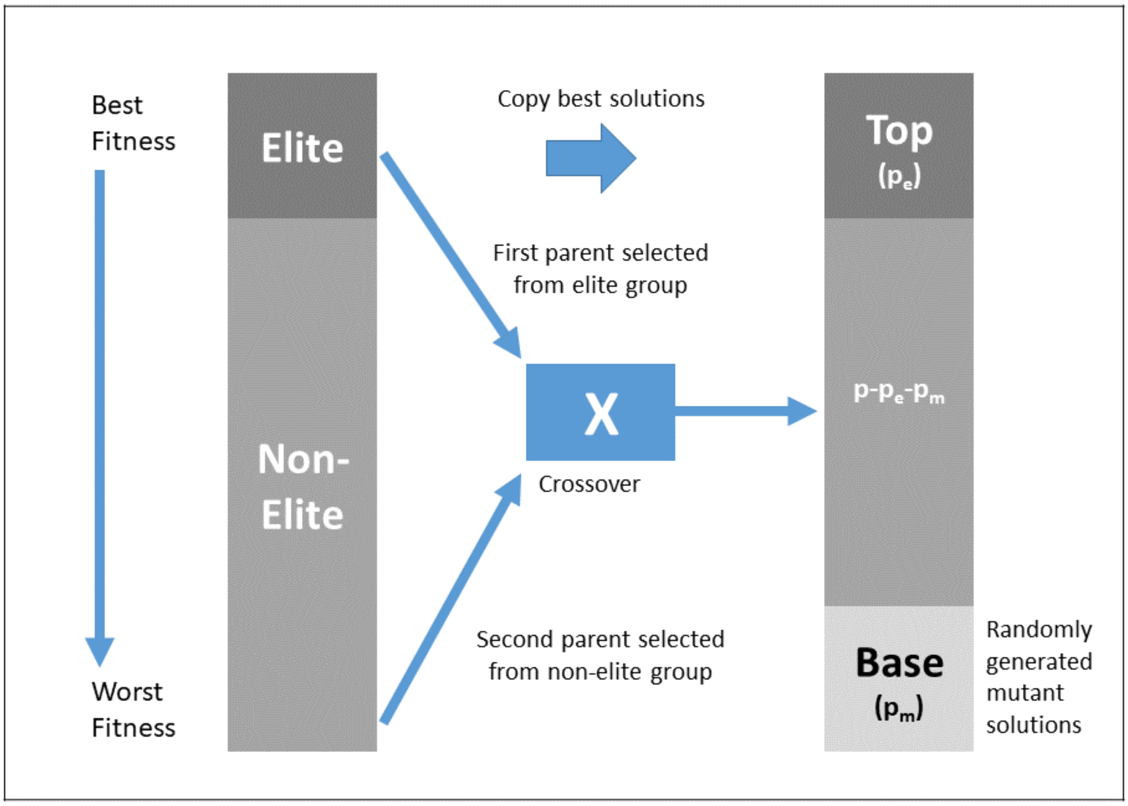

Figure 14 – Creation of new generation in the BRKGA.

The same way as in the RKGA, the BRKGA always applies the previously described uniformly parametrized crossover from (SPEARS; JONG, 1995). The only modification is that the probability ρ_a of inheriting an allele from the first parent (a) is always larger than 0.5. Considering that the first parent is always an elite one, setting $\rho_a > 0.5$ results in a higher chance of the offspring to inherit genes from an elite parent, adding a bias toward elite genes that was not present in the original RKGA.

The BRKGA pseudocode is described in Algorithm 3. Initially, in line 2, a population P is started. At line 4, the fitness of each individual is calculated using the solution decoder. At line 5, the population is sorted according to the individuals' fitness. The elite and non-elite groups of the population are divided in line 6. At line 7, the elite individuals are copied to the new population, while in line 8 the selection of parents for crossover is performed. Lines 9 and 10 show the application of the mutation and crossover operators. Finally, the new population is generated in line 11. The evolutionary process runs until the stopping criteria is met, and the algorithm returns the best solution found.

Algorithm 3 BRKGA Pseudocode

```
procedure BRKGA
    Randomly generate initial population P;
    while Stopping criteria not met do
        Evaluate fitness of each individual in P using the Decoder;
        Sort population P in increasing order of fitness values;
        Divide P into elite and non-elite groups;
        Copy elite individuals of current population to next generation;
        Select an elite parent to crossover with a second parent from the non-elit population;
        Perform the parametrized uniform crossover;
        Generate new mutants;
        Update next population;
    end while
    return Returns best individual in the population
end procedure
```

According to the study, the BRKGA was built as a general search metaheuristic capable of finding optimal or near-optimal solutions to hard combinatorial optimization problems (TOSO; RESENDE, 2015). As a general metaheuristic, the BRKGA clearly separates the problem-dependent from the problem-independent parts. As illustrated in Figure 15, the evolutionary part of the algorithm has no knowledge of the problem and seeks to operate only in the random-keys domain. The only problem-dependent part is the decoder, responsible for mapping the random-key vectors into viable solutions and calculating their fitness. This way, to use the BRKGA, it is only needed to define a decoder suitable for the studied problem and to adjust the execution parameters.

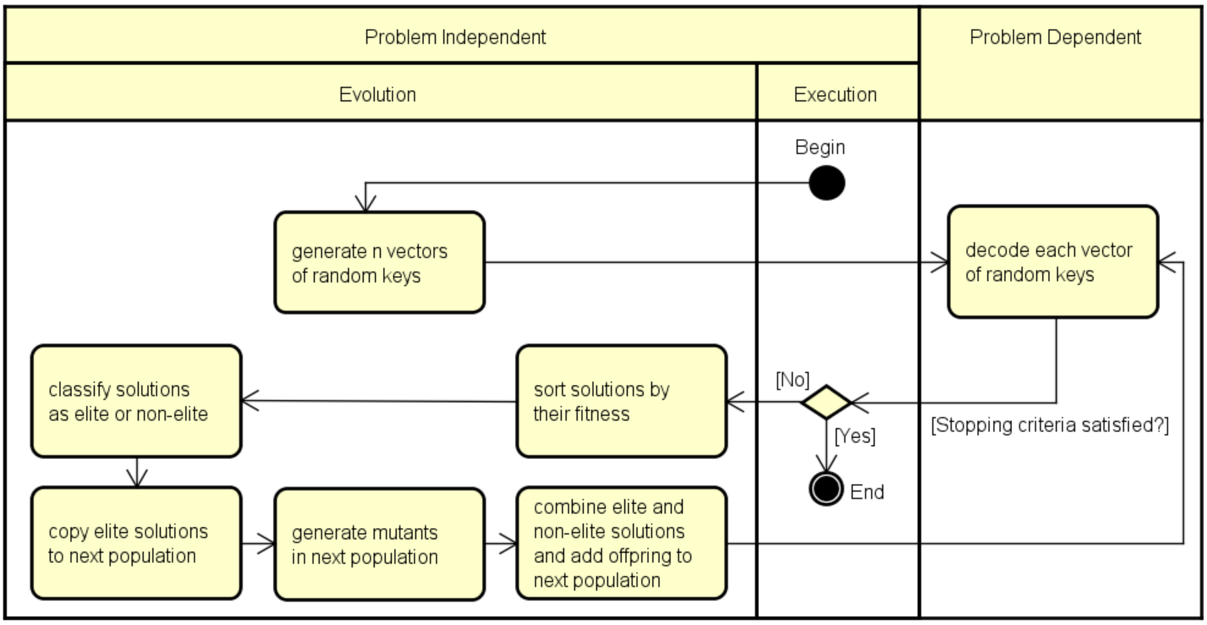

Figure 15 – Flowchart of a Biased Random-Key Genetic Algorithm.

The main advantage of using a Random-Key Genetic Algorithm, either BRKGA or RKGA, is the re-usability and ease of modeling and maintenance since the evolutionary parts are independent of the problem domain, which is not always true for other Genetic Algorithms in the literature. For the BRKGA, the modification added in comparison with the RKGA, resulted in considerable performance improvements as found in Gonçalves e Resende (2011) and Gonçalves, Resende e Toso (2014). According to the authors, the elitism bias results in greedy characteristics similar to those found in the semi-greedy heuristic of Hart e Shogan (1987) and in the Greedy Randomized Adaptive Search Procedure (GRASP) (FEO; RESENDE, 1995). The greedy characteristics improved, on average, the solutions found in comparison with pure random constructive methods. As with other traditional genetic algorithms, the BRKGA has the disadvantage of having a high number of parameters and a high computational cost, being more recommended for harder problems. It is important to notice that the decoder is one of the most important operational parts of the algorithm. Therefore, its performance highly impacts the final performance of the heuristic in a given problem.

To use the BRKGA this study focused on three main aspects:

1. **Decoder:** Responsible for converting the Random-Key Vectors into viable solutions and calculating their fitnesses;
2. **Parameter Configuration:** The tuning of the several parameters to guarantee the best performance in the studied problems;
3. **Stopping Criteria:** Responsible for limiting the processing time and guaranteeing

the solution of the problem is found within a viable time.

The following sections describe each of these aspects.

3.2 RESHUFFLE BRKGA

3.2.1 Decoder

To build a reshuffling decoder for the BRKGA, the best-performing reshuffling heuristics in the literature are used as references. The following subsections analyze these heuristics and describe how the decoder was built based on them.

3.2.1.1 Heuristic H3

As described in Chapter 2, Carlo e Giraldo (2012) proposes a reshuffling heuristic called H3, which is similar to the shuffling with nearest neighbor heuristic in Muralidharan, Linn e Pandit (1995). The H3 heuristic may be summarized as follows:

Algorithm 4 H3 Heuristic

1: **procedure** H3(Initial and Final location of elements, Movement cost matrices)
2: FinalCost = 0. ▷ Init variable for final cost
3: **while** Final organization was not reached **do**
4: **while** Exist items whose final location is open **do** ▷ Move non-cycle items
5: Reposition the item whose final location is open and is closest to item position; Draws are settled by favoring the load closest to S/R machine.
6: Using cost matrices, add to FinalCost unloaded cost of moving S/R to item's initial location.
7: Using cost matrices, add to FinalCost loaded cost of moving item from initial to final location
8: **end while**
9: Move to the closest open location the item closest to the S/R that is not in its final position and its final position is currently occupied. ▷ Break a cycle
10: Using cost matrices, add to FinalCost unloaded cost of moving S/R to item's initial location.
11: Using cost matrices, add to FinalCost loaded cost of moving item from initial to open location
12: **end while**
13: **return** Reshuffling steps, FinalCost
14: **end procedure**

By carefully examining H3 one can note two implicit assumptions: (1) cycles will be moved after non-cycles; and (2) cycles are moved sequentially (i.e., once a cycle is started, it is finished). The first implicit assumption is associated with lines 4 and 5. Notice that H3 starts with all items whose final location is open. Hence, by definition of a cycle, all items that are part of a cycle are left to be repositioned at the end. The second implicit assumption is associated with line 9. After breaking a cycle (line 9) there will be exactly one item that meets the criteria for line 4. Therefore, the *while* loop in line 4 will continue to be repeated for the all items in the cycle before moving to the next cycle. While the items are repositioned, the unloaded costs of moving the S/R machine unloaded to the item initial position, and moving loaded to the final or intermediary position are calculated in lines 6, 7, 10 and 11 using the movement cost matrices. The final cost and the reshuffling steps are returned in line 13.

To exemplify how the H3 approach solves unit-load reshuffle problems, the algorithm applied to the problem illustrated in Figure 16.

Initial Configuration

Loc 3	Loc 4	Loc 5
A	B	D
Loc 0	Loc 1	Loc 2
C	0_1	0_2

Final Configuration

Loc 3	Loc 4	Loc 5
B	C	$0_2'$
Loc 0	Loc 1	Loc 2
A	$0_1'$	D

Chebyshev Loaded Cost Matrix

g_{ij}	Loc 0	Loc 1	Loc 2	Loc 3	Loc 4	Loc 5
Loc 0	0	1	2	1	1	2
Loc 1	1	0	1	1	1	1
Loc 2	2	1	0	2	1	1
Loc 3	1	1	2	0	1	2
Loc 4	1	1	1	1	0	1
Loc 5	2	1	1	2	1	0

Figure 16 – The initial and final configurations and Chebyshev cost matrix for a sample reshuffling problem with two open locations, one cycle, and one non-cycle item.

One solution to this problem according to H3 results in the steps illustrated in Figure 17. In this figure, the star represents the position of the material handling equipment before executing the movements, and the arrow represents the next loaded movements to be performed.

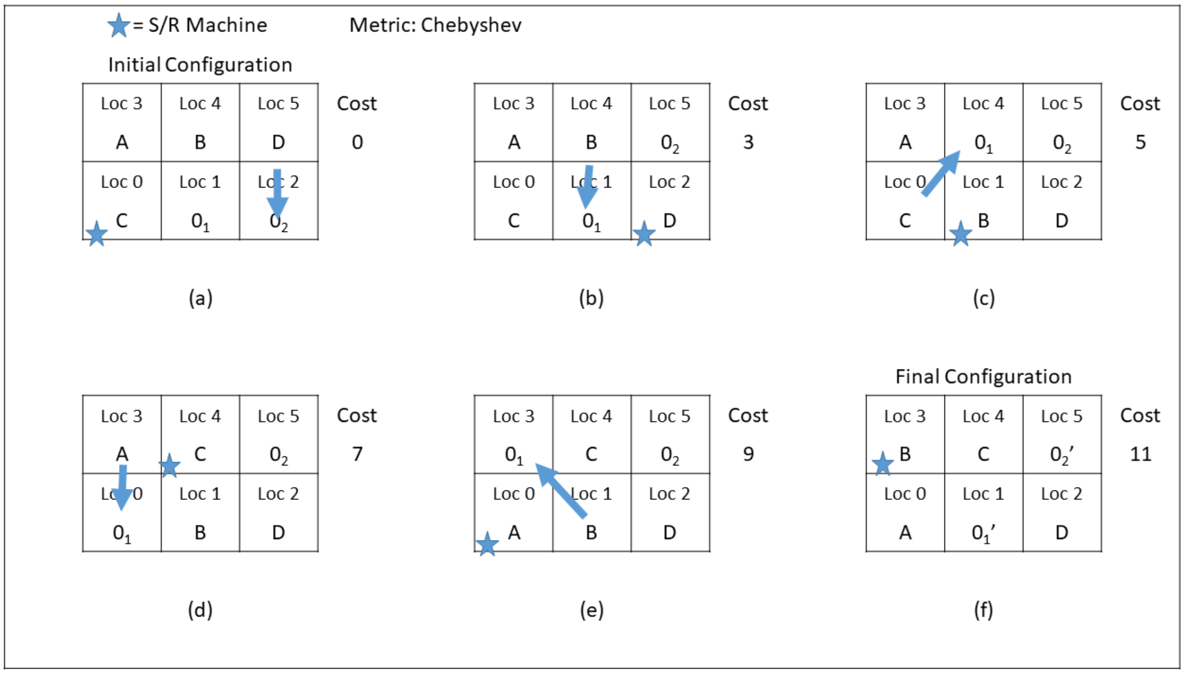

Figure 17 – Reshuffle solution using H3. Initial storage organization and non-cycle movement (a), movement to break the cycle (b), subsequent movements to reorganize the cycle elements (c - e), the final desired organization (f).

The final sequence of movements for this example are listed in Table 1.

Table 1 – Reshuffle solution for example problem using H3.

Move	**Item Moved**	**Move Cost**	**Total Cost**	**Location of items** $C - O_1 - O_2 - A - B - D$
0 - 5	none	2	2	As above
5 - 2	D	1	3	$C - O_1 - D - A - B - O_2$
2 - 4	none	1	4	As above
4 - 1	B	1	5	$C - B - D - A - O_1 - O_2$
1 - 0	none	1	6	As above
0 - 4	C	1	7	$O_1 - B - D - A - C - O_2$
4 - 3	none	1	8	As above
3 - 0	A	1	9	$A - B - D - O_1 - C - O_2$
0 - 1	none	1	10	As above
1 - 3	B	1	11	$A - O_1 - D - B - C - O_2$

For the C++ implementation of the H3 algorithm, refer to Appendix A.

3.2.1.2 Heuristic GRH

Pazour e Carlo (2015) proposes a reshuffling heuristic similar to H3 but relaxing the assumption that non-cycles are moved before cycles. This is achieved by including a parameter τ that permits cycles to be broken while there are still non-cycles to be relocated. The GRH heuristic may be summarized as follows:

Algorithm 5 GRH Heuristic

1: **procedure** GRH(Initial and Final location of elements, Movement cost matrices, τ)
2: Define set C_c with cycles in the problem
3: FinalCost = 0. ▷ Init variable for final cost
4: **while** Final organization was not reached **do**
5: Identify item (q) with final position occupied stored closest to the S/R machine's current position that has loaded movement cost from initial location to an open location $\leqslant \tau$ OR whose ending position is currently open.
6: **if** item is part of cycle ($q \in C : C \in C_c$) **then** ▷ Break nearby cycle
7: Move item q (for which loaded movement cost from the initial location to an open location $\leqslant \tau$) and remove the cycle from the list of all cycles ($C_c = C_c \setminus C$).
8: Using cost matrices, add to FinalCost unloaded cost of moving S/R to item's initial location.
9: Using cost matrices, add to FinalCost loaded cost of moving the item from initial to open location.
10: **else if** item (q) has ending position is currently open **then** ▷ Move non-cycle item
11: Move item to its final position
12: Using cost matrices, add to FinalCost unloaded cost of moving S/R to item's initial location.
13: Using cost matrices, add to FinalCost loaded cost of moving the item from initial to the final location.
14: **else** ▷ Break distant cycle
15: Move to the closest open location the item closest to the S/R that is not in its final position and its final position is currently occupied.
16: Using cost matrices, add to FinalCost unloaded cost of moving S/R to item's initial location.
17: Using cost matrices, add to FinalCost loaded cost of moving the item from initial to open location.
18: **end if**
19: **end while**
20: **return** Reshuffling steps, FinalCost
21: **end procedure**

As stated in the original paper (PAZOUR; CARLO, 2015), the H3 is a specific case of the GRH when $\tau = 0$. This can be observed in the algorithm. At line 5, if $\tau = 0$, no items will be found to meet the criteria of travel distance from starting location of q to an open location $\leqslant \tau$. In this case, all identified items will be non-cycles, meeting the conditions for moving non-cycle items (lines 10 to 13). Only when no items are identified in line 4 the cycles will be broken (lines 14 to 17). This behavior is exactly that of H3. However, when $\tau > 0$, nearby cycles will be broken before non-cycle items (lines 6 to 9). This relaxation of the previous assumptions allows the heuristic to find new solutions with a similar processing time of the H3. At each item movement, the final cost of the reshuffle is updated with the cost of moving the S/R machine unloaded to the item's initial position, and then moving loaded to the final or intermediary position (lines 8, 9, 12, 13, 16 and 17). The final cost and the reshuffling steps are returned in line 20.

Since each problem may require a different τ, the authors also propose running the GRH iteratively with different values of τ ($\tau \geqslant 0$) and reporting the best objective value. Values between 0 and 20 were found to be more appropriate for scenarios up to 400 locations and distances calculated by Chebyshev metric (maximum between horizontal and vertical distances) (PAZOUR; CARLO, 2015).

The GRH algorithm starts by identifying all cycles in the problem. This step can be performed using the polynomial-time algorithm also proposed in Pazour e Carlo (2015) and summarized in Algorithm 6.

Algorithm 6 Polynomial-time algorithm to identify cycles

procedure CYCLES(Initial (I_k) and Final (F_k) location of elements $k \in K$)
    Define $L = \{k \in K : I_k \neq F_k \bigcap F_k \neq OPEN\}$.
    Initialize set index $i = 0$.
    if $L \neq \emptyset$ **then**
        $i = 1$. ▷ Increase cycle index
        $k = l \in L$. ▷ Select an item from set L
        $C_i = \{k\}$. ▷ Set C_i only includes item k
    end if
    while $L \neq \emptyset$ **do** ▷ While there are cycles
        Select $k' \in K$ such that $I'_k = F_k$. ▷ Select the item currently located in item k's final location
        if $k' \ni L$ **then** ▷ C_i is not a cycle
            $L = L \setminus C_i$. ▷ Remove the elements in C_i from L
            $k = l \in L$. ▷ Select an item from the new set L
            $C_i = \{k\}$. ▷ Set C_i only includes item k
        else if $k' \in C_i$ **then** ▷ Cycle C_i identified
            $L = L \setminus C_i$. ▷ Remove the elements in C_i from L
            $i = i + 1$. ▷ Increase cycle index
            $k = l \in L$. ▷ Select an item from the new set L
            $C_i = \{k\}$. ▷ Set C_i only includes item k
        else ▷ $k' \ni C_i$
            $C_i = C_i \bigcup k'$ ▷ Add k' to set C_i
            $k = k'$
        end if
    end while
    return Cycles C_i
end procedure

Using the initial storage configuration I_k, the final storage configuration F_k, the set of items to be reshuffled K, and the open locations *OPEN*, Algorithm 6 identifies the cycles when their number is unknown. The algorithm starts with a subset of items $L \in K$ containing the items that require reshuffling and whose final location is initially occupied by another item.

A solution for the problem of Figure 16 according to GRH results in the steps illustrated in Figure 18. In this figure, the star represents the position of the material handling equipment before executing the movements, and the arrow represents the next loaded movements to be performed. The solution was evaluated with a $\tau = 1$.

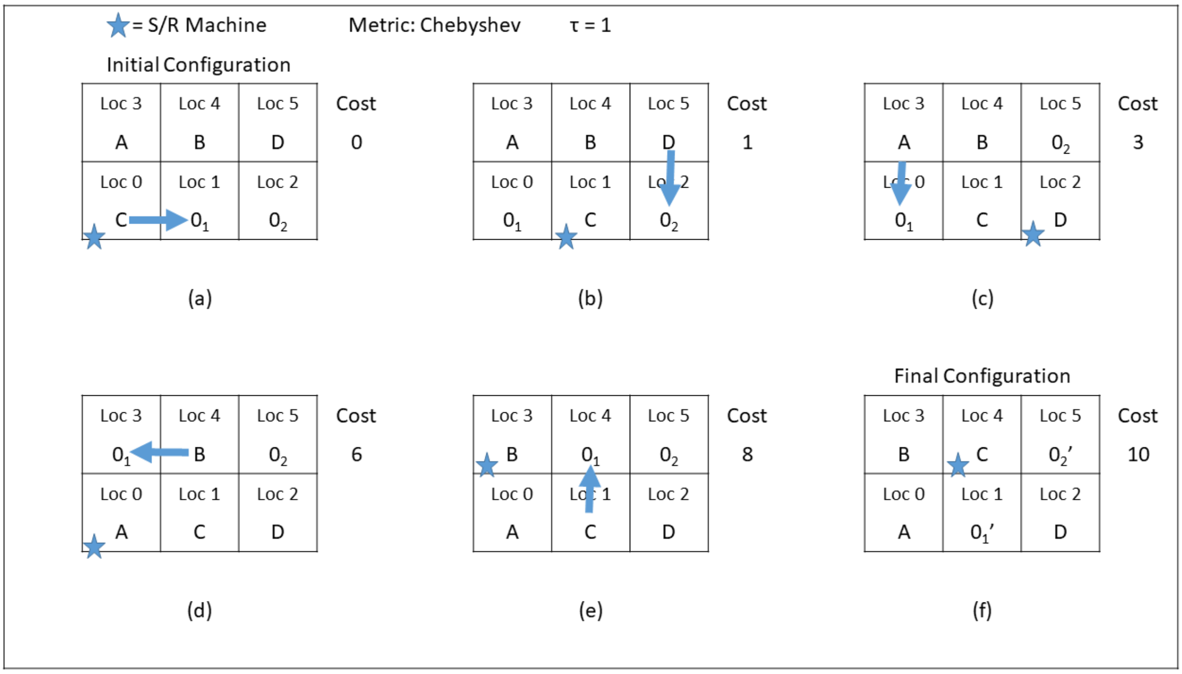

Figure 18 – Reshuffle solution using GRH. Initial storage organization and non-cycle movement (a), movement to break the cycle (b), subsequent movements to reorganize the cycle elements (c - e), the final desired organization (f).

The final sequence of movements for this example are listed in Table 2. As can be observed, the combination of cycle break and non-cycle movements allowed the GRH to find a better solution to the problem when compared with the H3. Total final cost of 10 distance units in comparison with the 11 distance units found by the H3.

Table 2 – Reshuffle solution for example problem using GRH.

Move	Item Moved	Move Cost	Total Cost	**Location of items** $C - O_1 - O_2 - A - B - D$
0 - 1	C	1	1	$O_1 - C - O_2 - A - B - D$
1 - 5	none	1	2	As above
5 - 2	D	1	3	$O_1 - C - D - A - B - O_2$
2 - 3	none	2	5	As above
3 - 0	A	1	6	$A - C - D - O_1 - B - O_2$
0 - 4	none	1	7	As above
4 - 3	B	1	8	$A - C - D - B - O_1 - O_2$
3 - 1	none	1	9	As above
1 - 4	C	1	10	$A - O_1 - D - B - C - O_2$

For the C++ implementation of the GRH algorithm, refer to Appendix A. And for the C++ implementation of the polynomial-time cycles algorithm, refer to Appendix B.

3.2.1.3 Reshuffle Decoder

The GRH heuristic (PAZOUR; CARLO, 2015) adds an advantageous flexibility in comparison to the H3 heuristic (CARLO; GIRALDO, 2012), and maintains the excellent performance of the previous. However, by defining a fixed τ to be applied through all the reshuffling process, it reduces the explored universe by not considering using larger and smaller values of τ in different moments of the reshuffling.

To take advantage of the performance characteristics and increase the explored solution universe in order to find better reshuffling configurations, this study proposes a BRKGA heuristic with decoder based on the GRH heuristic by Pazour e Carlo (2015).

The core of the GRH is maintained in the decoder, the main improvement offered is to use the random-keys of the BRKGA chromosome to dynamically adapt the τ during the reshuffling process. Before each movement decision, the τ value is readjusted by an allele in the chromosome.

The GRH uses the τ at each loop to decide whether to move a non-cycle element to its final position, to break a nearby cycle, or to break a distant cycle. To maintain this behavior, each random-key in the chromosome is multiplied by a constant factor to form the τ. To guarantee the maximum flexibility for the decoder, the constant factor used is the maximum loaded cost (g_{MAX}) of the given problem. This factor can be obtained when the loaded travel cost matrix for the problem is calculated. An example of such a matrix is illustrated in the Christofides e Colloff (1973) study and is also used in the mathematical models proposed by Pazour e Carlo (2015).

The chromosome should be long enough to contain keys for all movements performed, but not too long, otherwise, the performance will be greatly affected. Knowing that the

GRH heuristic is based on the H3, it is reasonable to use the latter to define upper bound for movements to obtain the final configuration.

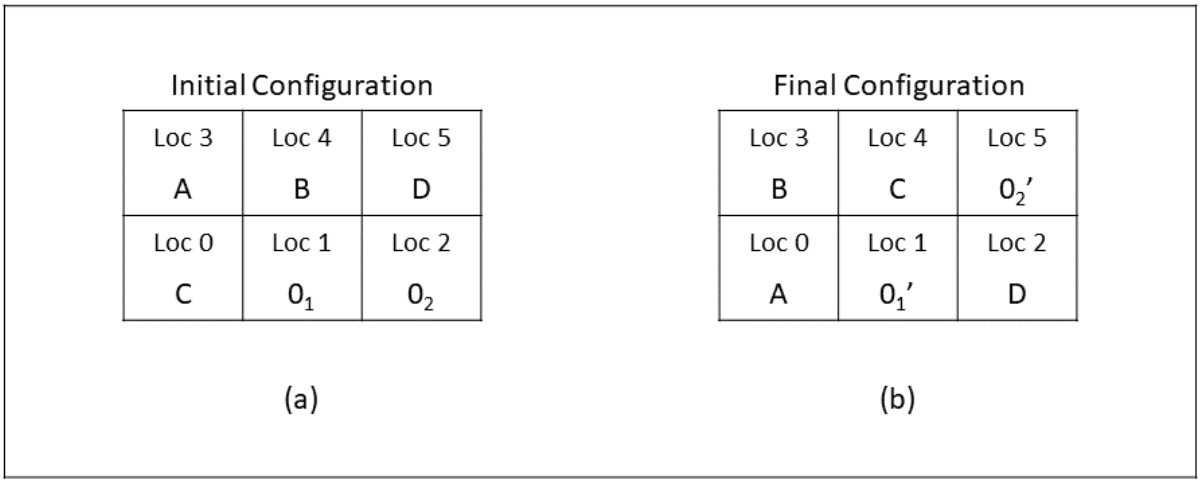

Figure 19 – The initial (a) and final (b) configurations for a sample reshuffling problem to be solved using H3 heuristic.

The H3 moves all non-cycle elements, and later adds one movement per cycle in other to break the cycle by opening one position and starts relocating the rest of the items as non-cycles. To illustrate this breaking behavior applied in the example of Figure 19, consider the Figures 20(a) to (f). In Figure 20(a) the storage is in its initial organization. The first movement is to relocate the non-cycle item D to its final position. After this step, a series of movements is needed to relocate the cycle items A (Loc 3), B (Loc 4), and C (Loc 0), to their respective final positions. The first step is to break the cycle by moving the element closest to the S/R to the open position closer to its final position. In this case, it is to move element C from Loc 0 to Loc 1 (Figure 20(b)). This movement frees Loc 0 for relocating item A from Loc 3 to Loc 0 (Figure 20(c)). Now that Loc 3 was freed, item B can be relocated there from Loc 4 (Figure 20(d)). Finally, Loc 4 is open for relocating item C (Figure 20(e)). The final organization is depicted in Figure 20(f). The final sum of movements is 1 non-cycle item relocation + 1 cycle break + 3 cycle item relocations = 5 movements = total of items to be relocated + total number of cycles to break.

Note that there are several possible ways of solving that same scenario. Nevertheless, the heuristic increases the probability of finding the optimum solution by greedily searching shortest distances. This is done by choosing the break-movement based on the shortest distance between the element intermediate position and its final position and solving draws by selecting elements closest to the S/R current position.

The sequential cycle relocation used in this routine was originally introduced in Christofides e Colloff (1973) and demands one movement to break each cycle, and one movement to

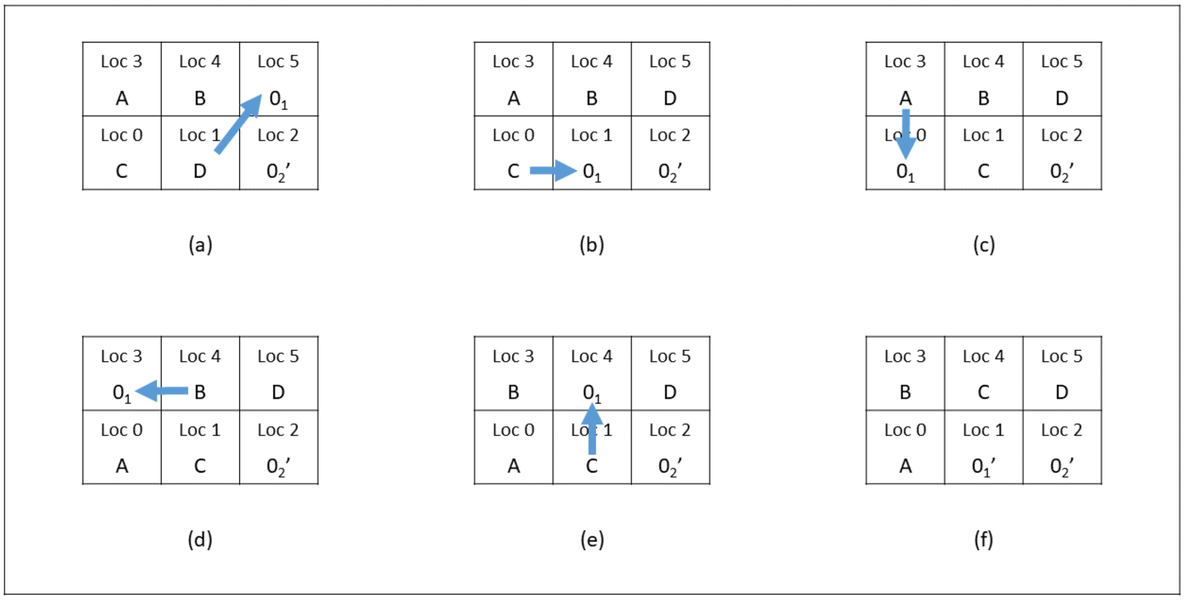

Figure 20 – Sample problem solution using H3. Initial storage organization and non-cycle movement (a), movement to break the cycle (b), subsequent movements to reorganize the cycle elements (c - e), the final desired organization (f).

relocate each item within a cycle. When considering the additional movements to relocate non-cycle items, at the end, the H3 relocation process requires one movement per item to be relocated k_{MAX} (total number of elements to be reshuffled) plus one break movement per cycle c_{MAX} (total number of cycles identified). So the upper bound for the relocation movements is:

$$maxMoves = k_{\text{MAX}} + c_{\text{MAX}} \tag{3.1}$$

Even though the GRH heuristic changes the order of cycle and non-cycle relocations, it does not add movements to the final relocation process. For this reason, the upper limit for relocation movements is maintained. This limit can now be used as the size of the chromosome for the reshuffling BRKGA.

For the example scenario in Figure 20, considering the loaded movements cost matrix calculated using Chebyshev metric (largest between horizontal and vertical distances) over a rack with unitary distances, a chromosome, and its translation into τ values are illustrated in Figure 21.

The BRKGA reshuffling decoder may be summarized as follows:

Algorithm 7 BRKGA Reshuffling Decoder

procedure ReshuffleDecoder(Initial and Final location of elements, Movement cost matrices, Chromosome)
    Define set C_c with cycles in the problem
    Initiate using first gene locus (first allele in the chromosome
    FinalCost = 0. ▷ Init variable for final cost
    while Final organization was not reached OR chromosome is over **do**
        Update τ using current chromosome allele
        Identify item (q) with final position occupied stored closest to the S/R machine's current position that has loaded movement cost from initial location to an open location $\leqslant \tau$ OR whose ending position is currently open.
        if item is part of cycle ($q \in C : C \in C_c$) **then** ▷ Break nearby cycle
            Move item q (for which loaded movement cost from the initial location to an open location $\leqslant \tau$) and remove the cycle from the list of all cycles ($C_c = C_c \setminus C$).
            Using cost matrices, add to FinalCost unloaded cost of moving S/R to item's initial location.
            Using cost matrices, add to FinalCost loaded cost of moving the item from initial to open location.
        else if item (q) has ending position is currently open **then** ▷ Move non-cycle item
            Move item to its final position
            Using cost matrices, add to FinalCost unloaded cost of moving S/R to item's initial location.
            Using cost matrices, add to FinalCost loaded cost of moving the item from initial to the final location.
        else ▷ Break distant cycle
            Move to the closest open location the item closest to the S/R that is not in its final position and its final position is currently occupied.
            Using cost matrices, add to FinalCost unloaded cost of moving S/R to item's initial location.
            Using cost matrices, add to FinalCost loaded cost of moving the item from initial to open location.
        end if
        Move to next gene locus (get next allele)
    end while
    return Reshuffling steps, FinalCost
end procedure

Chebyshev Loaded Cost Matrix

g_{ij}	Loc 0	Loc 1	Loc 2	Loc 3	Loc 4	Loc 5
Loc 0	0	1	2	1	1	2
Loc 1	1	0	1	1	1	1
Loc 2	2	1	0	2	1	1
Loc 3	1	1	2	0	1	2
Loc 4	1	1	1	1	0	1
Loc 5	2	1	1	2	1	0

Example Chromosome

0.32	0.64	0.58	0.96	0.77	0.23

x

(g_{MAX} = 2)

=

Translated Chromosome into τ values

0.64	1.22	1.16	1.92	1.54	0.46

Figure 21 – Example chromosome for reshuffling.

As stated previously, the core of the GRH is maintained in the decoder. At the first line, the decoder receives the chromosome with the random-keys, the initial and final organization of the storage, and the movement cost matrices. The initial and final organization of the storages are vectors that relate the elements with their locations in the storage. The movement cost matrices include the loaded and unloaded travel cost matrices calculated using the distance metric defined in the problem. At the second line, the algorithm identifies all the cycles in the problem using the polynomial-time algorithm previously used in the GRH. At the third line, the indexes are set to start calculating τ using the first allele in the chromosome. At the fourth line the condition of the *while* loop is set to guarantee a final organization is reached or the possible configurations for τ (values calculated from the chromosome) are exhausted. This new decision is not strictly necessary since the chromosome size is calculated to contain the maximum allowed movement number. In any case, it is used for safety measurement. In line 6 the main modification in comparison with the GRH is introduced. At this point, the algorithm uses the chromosome to update the τ value. From line 7 to line 20 the decoder uses exactly the same procedure of the GRH algorithm. In line 7 the algorithm searches for an item close to the material handling equipment that is either a non-cycle item whose final location is available or part of a nearby cycle. In lines 8 to 11, if an item is found and it is part of a nearby cycle, it is moved to an intermediary open position close to its final location, and the cycle is removed from the list of all cycles. In lines 12 to 15, if an item is found and it is a non-cycle, the item is moved to its final location. In lines 16 to 19, if no other item is selected to move in the previous steps, the item closest to the material handling equipment that has its final position occupied is moved to the closest open location. In line 14, the indexer of the gene in the chromosome is updated in order to change the τ for the following movement decision. As previously done in the H3 and the GRH, at each item movement the final cost of the reshuffle is updated by adding the costs of moving the S/R machine unloaded to the item's initial position, and moving loaded to the final or intermediary position (lines 8, 9, 12, 13, 16 and 17). After achieving the final organization, the decoder returns the final reshuffling cost and the execution stops in line 23. This reshuffling cost, which depends on the cost matrices used and the reshuffling movements, is the fitness of the chromosome.

For the C++ implementation of the Reshuffle BRKGA Decoder, refer to Appendix A.

3.2.2 Stopping Criteria

Several studies investigated the best the upper bound generations to ensure convergence of the evolutionary algorithm. As pointed out in Safe et al. (2004), traditionally three termination conditions have been employed for Genetic Algorithms:

- An upper limit on the number of generations;
- An upper limit on the number of evaluations of the fitness function;
- An evaluation of the chance of extremely low chances of achieving significant changes in the next generations.

The authors of the study discuss that a choice of sensible settings for the first two alternatives demands significant knowledge about the problem to allow estimation of reasonable search length. In contrast, the third alternative is alternative and does not require such knowledge. For this approach, two variants are applied. Genotypical and phenotypical stopping criteria. The former ends when the population reaches a certain threshold with respect to the chromosomes in the populations. A number of genes converged to a certain value in a percentage of the population, for example. The phenotypical criterion, on the other hand, measures the algorithm progress achieved in terms of the results of the chromosomes, which may be expressed as the fitness values of the population. Though adaptive, these stopping criteria raise difficulties concerning the establishment of appropriate values for their associated parameters.

The study of adaptive termination methods was further deepened in Zielinski, Peters-Drolshagen e Laur (2005). The study executed an extensive evaluation of eleven stopping criteria on Differential Evolution (DE) and Particle Swarm Optimizer (PSO) algorithms. It was found that maximum distance criterion *MaxDist* and combined criterion *ComCrit* are the most promising stopping criteria for differential evolution algorithms. For PSO algorithms, the distribution-based maximum distance criterion *MaxDistQuick* and the combined criterion lead to more reliable convergence behaviors.

In the *MaxDist* criterion, the allowed Maximum Distance (*maxDist*) between the fitness of every chromosome in the population is calculated through the Equation 3.2.

$$maxDist = f(x_i) - f(\mathbf{x_{Best}}) \tag{3.2}$$

Where $\mathbf{x_{Best}}$ is the individual with the best fitness in the population, and x_i are the other individuals of the population. To terminate the execution, the criterion considers that the heuristics converged when:

$$maxDist \leq \begin{cases} m, & if \quad f(x_{\text{Best}}) = 0 \\ m \cdot f(x_{\text{Best}}), & if \quad f(x_{\text{Best}}) \neq 0 \end{cases} \tag{3.3}$$

The combined criterion *ComCrit* waits for an average improvement of the algorithm to stagnate for t generation before the *MaxDist* criterion is analyzed. The *MaxDistQuick* evaluates the *MaxDist* only in a Convergence Population Fraction (CP), instead of the whole population. For the *MaxDistQuick* criterion to converge, the top CP individuals of the population should have a maximum distance from the best chromosome lower than m according with Equation 3.3.

In addition to these findings, the study presents several recommendations concerning suitable stopping criteria for evolutionary algorithms based on the performance variations observed. In general, for evolutionary algorithms, Zielinski, Peters-Drolshagen e Laur (2005) suggests an $m = 0.001$. The parameter CP used in the *MaxDistQuick* criterion is more dependent on the specific model. The authors found that for PSO algorithms $0.3 \leqslant CP \leqslant 0.6$ results in a good cost-benefit between processing time to analyze the convergence and the final time the algorithm is allowed to run. Values under 0.3 were found to have a higher risk of premature convergence because the fraction of the population is too small to guarantee significant statistical certainty of genetic diversity. While values over 0.6 did not result in a significant reduction in final processing time because the fraction of the population is too large to analyze using sorting algorithms.

Based on these findings and recommendations, the Maximum number of generations (*MAXGEN*) criterion (where the algorithm stops after reaching a maximum number of generations allowed) and the *MaxDistQuick* stopping criterion were applied in the reshuffle BRKGA to reduce processing time. The *MaxDistQuick* criterion was combined with the *MAXGEN* criterion in order to ensure an upper limit of generation executed in case the population does not converge quickly. The *MaxDistQuick* was used because it is unexpected that the whole population converges to a similar phenotypic solution in the case of the BRKGA, rendering ineffective the use of the basic *MaxDist* criterion. This is due to the way the mutant population is generated. As previously explained, the BRKGA randomly generates mutants the same way individuals are generated for the first population. The impact of such mutation procedure is that these mutants have no genetic relationship with the rest of the population, lowering the chances of resulting in similar phenotypes. In this case, the *MaxDistQuick* can be evaluated only over the elite and generated fractions of the population and avoid the mutants. The *MaxDistQuick* also benefits from the fact that the BRKGA already sorts the whole population using the fitness, facilitating the evaluation in a fraction of the population. Because of the multiobjective character of the parameter CP of the *MaxDistQuick* that needs to result in better solutions (tending to higher values) but also limit the processing time (tending to lower values) it was manually adjusted in the parameter tuning phase described in Chapter 4.

4 PARAMETER CONFIGURATION

Before running the final experiments where the reshuffling BRKGA is evaluated in comparison with the H3 heuristic (CARLO; GIRALDO, 2012) and the GRH heuristic (PAZOUR; CARLO, 2015), it is necessary to create an instance generator for reshuffling problems and to configure the several parameters present in the heuristics.

Section 4.1 describes how the reshuffling scenarios are represented for the code. This section also introduces a parser for the input files and an algorithm developed for this study to generate the reshuffling scenarios. Since no real data was available for the experiments, all the scenarios were created using the developed generator.

Section 4.2 introduces the Iterated Racing (Irace) technique used for automatic parameter configuration, while Sections 4.3 and 4.4 describes how the iterated racing technique was applied to configure the GRH and the reshuffling BRKGA respectively. The parameter tunning used the Iterated Racing for Automatic Algorithm Configuration (IRACE) library (LÓPEZ-IBÁÑEZ et al., 2016) developed for the R computing Environment (R Core Team, 2015).

To run the IRACE, the primary step was to create the configuration files defining the tunned parameters (including type, variation range, and initial configurations), the rules and constraints the parameters should comply to, the instance list used for the adjustment, as well as the connection with the optimized algorithm and its cost function.

The GRH was configured using the IRACE to guarantee the original τ values proposed by the authors in the original paper are valid for the scenarios tested in this study. The authors suggest $0 \geqslant \tau \leqslant 20$ for scenarios up to 400 locations and Chebyshev distance metric Pazour e Carlo (2015).

The reshuffling BRKGA was configured via IRACE with only *MaxDistQuick* termination criterion to ensure best solution quality results. Since the IRACE process is single-objective and was not designed to improve both the processing time and the solution quality of the tunned heuristics, in Section 4.5 the parameters of the *MaxDistQuick* stopping criteria used in the BRKGA were manually tunned. This additional step intends to build a BRKGA with low processing time and high-quality results.

4.1 SCENARIO REPRESENTATION

In order to facilitate the description of testing cases and provide standard inputs for the heuristics, all the scenarios were described using the following information for the reshuffling independent of the used algorithm. These are:

- **imax:** the number of storage locations;

- **startPos:** location of S/R when reshuffling starts actuating. Positive numbers are actual locations and negative numbers indicate the algorithm to start the S/R where the best first move starts;
- $\mathbf{I_k}$**:** initial location *i* of item *k*;
- $\mathbf{F_k}$**:** final location *i* of item *k*;
- $\mathbf{g_{ij}}$**:** matrix of cost of loaded movement from location *i* to location *j*. This matrix can be asymmetric as the ones used in Christofides e Colloff (1973) to represent complex aisle systems in storages;
- $\mathbf{d_{ij}}$**:** matrix of cost of unloaded movement from location *i* to location *j*. This matrix can be asymmetric and with lower values than the g_{ij} matrix as the ones used in Christofides e Colloff (1973) to represent higher accelerations when unloaded.

For a CSV example of the input file, refer to Appendix B.

4.1.1 Parser

Along with the reshuffle scenario input file, a parser class was written to be used by all the tested heuristics. The parser not only reads the input file to identify the previous parameters, but also finds:

- **Cc:** The cycles using the polynomial-time algorithm to identify all cycles found in Pazour e Carlo (2015);
- $\mathbf{g_{MAX}}$**:** the maximum loaded travel cost;
- $\mathbf{d_{MAX}}$**:** the maximum unloaded travel cost;
- $\mathbf{k_{MAX}}$**:** number of elements to rearrange;
- $\mathbf{o_{MAX}}$**:** number of open positions in the scenario;
- $\mathbf{I_i}$**:** initial item *k* stored in location *i* ($I_i(i) \in 0 \ldots k_{MAX}$-1 for items, $I_i(i)$ = -1 for open locations);
- $\mathbf{F_i}$**:** final item *k* stored in location *i* ($I_i(i) \in 0 \ldots k_{MAX}$-1 for items, $I_i(i)$ = -1 for open locations);
- $\mathbf{OI_o}$**:** initial location *i* of open position *o* ($OI_o(o) \in 0 \ldots i_{MAX}$-1);
- $\mathbf{OF_o}$**:** final location *i* of open position *o* ($OF_o(o) \in 0 \ldots i_{MAX}$-1);

A C++ implementation of the parser can be found in Appendix B.

4.1.2 Scenario Generation

To standardize the generation of testing scenarios an automatic generator was created. This code receives inputs for:

- **Size of storage (Imax):** the number of storage locations;
- **Utilization (U):** percentage of storage locations occupied by item. 0% > U > 100%;
- **Organization (O):** percentage of items that do not change positions during reshuffling. O > 100%;
- **Final Open Locations (FO):** if the open locations remain in the same positions in the end configuration of the storage ("equal"), or if the final configurations of the open locations are randomly repositioned ("random");
- **Start location (S):** location of S/R when reshuffling starts actuating. Can be: "random" (starts at random location); "none" (starts at the same position of the best first item to move as defined by the algorithm); or with fixed value S, where S < Imax;
- **Columns (Cols):** number of columns in the rack. This parameter defines the rack organization. So a 20 x 20 rack is a rack with 400 items and 20 columns. Through this parameter, different rack organization can be obtained;
- **Loaded Movement Metric (D):** metric used to calculate the cost of moving a loaded S/R between different rack locations in terms of distance. Can be: "random", the distances are randomly attributed with maximum value Imax and minimum value a random number between 1 and Imax/2; "euclidean", the distances are rectilinear and calculated using rack organization; "chebyshev", the distances are calculated using Chebyshev metric (largest between horizontal and vertical distances) on the rack organization; "cityblock", the distances are calculated using Manhattan metric (sum of horizontal and vertical distances) on the rack organization;
- **Unloaded Movement Factor (UD):** factor used to calculate the cost of moving an unloaded S/R machine in relation to the distance between rack locations. Can be: "random", the cost of moving an unloaded S/R is the distance between racks multiplied by a random factor between 0.1 and 0.99; "equal", the cost of moving an unloaded S/R is the same as the distance between racks as used by the loaded cost.

All the distances evaluations consider the locations with dimensions of 1 Distance Unit (DU).

To guarantee a reliable random distribution of the storage configurations generated, the Algorithm 8 was used. At line 1, the algorithm receives as input the size of the storage (Imax), the utilization (U), the organization (O) and the final configuration of the open positions (FO). At line 2, the parameters ranges are verified. At line 3 the number of items in the storage (kmax) is calculated using the storage size and the utilization. At line 4, a list i_list of random integer ranging from 0 to Imax is created. At line 5, the initial configuration of the items is obtained from the first kmax items in i_list. Lines 6 to 10 define if the final configuration of the open positions (FO) coincide with the initial position or if they are randomly reshuffled. If FO is random, all the list i_list will be reshuffled to obtain the final configuration of the storage. Otherwise, if FO is coincident, only the first kmax items in i_list will be reshuffled to obtain the final storage configuration. After defining FO, at lines 11 to 13 the algorithm randomly swaps pairs of allowed reshuffled items in i_list until the number of different location between the first kmax elements in the list is greater than the desired organization limit defined by O. Finally, at line 14 the final configuration of the storage Fk is obtained from the first kmax elements of the reshuffled i_list.

Algorithm 8 Algorithm to generate different initial and final configuration of storage

```
 1: procedure StorageConfigurationGenerator(Imax, U, O, FO)
 2:     Check parameters
 3:     kmax = Imax * U
 4:     Create random list i_list of size Imax
 5:     Create list of initial locations Ik from first kmax items in i_list
 6:     if FO = "random" then
 7:         Max Reshuffle Index = Imax.
 8:     else if FO = "equal" then
 9:         Max Reshuffle Index = kmax.
10:     end if
11:     while Equal elements in first kmax elements in i_list is greater than kmax * O do
12:         Swap items in two random indexes of i_list
13:     end while
14:     Create list of final locations Fk from first kmax items in i_list
15: end procedure
```

The quality of the random distribution of this algorithm is guaranteed by the random number generator used in the code. This study relies on the widely used Mersenne Twister pseudorandom number generator (MATSUMOTO; NISHIMURA, 1998) due to its fast generation of high-quality pseudorandom integers.

A Python 3.4 implementation of the scenario generator can be found in Appendix B.

4.2 AUTOMATIC PARAMETER CONFIGURATION

Frequently optimization algorithms require the fine-tuning of a large number of parameters in order to perform well. Sometimes, these parameters can be adjusted manually until an acceptable configuration is reached. Nevertheless, when the number of parameters increases, the increasing amount of possible parameter combinations makes tunning difficult.

Several techniques were suggested throughout the years addressing this problem and automatizing the parameter selection in the best manner. Recently the *Iterated Racing* technique is receiving more attention in the scientific community for successfully automatically configuring several algorithms (LÓPEZ-IBÁÑEZ et al., 2016). This section will describe this technique.

4.2.1 Iterated Racing

The IRACE is an automatic parameter configuration technique recently applied in several literature problems such as traveling salesman with time windows (LÓPEZ-IBÁÑEZ et al., 2013), simultaneous slot allocation (PELLEGRINI; CASTELLI; PESENTI, 2012), flow shops (BENAVIDES; RITT, 2015), placement of virtual machines (STEFANELLO et al., 2015), on-line bin packing (YARIMCAM et al., 2014), image binarization (MESQUITA et al., 2015), real-time train routing selection (SAMA et al., 2016), bike sharing re-balancing (DELL et al., 2016), energy planning (JACQUIN; JOURDAN; TALBI, 2014), class scheduling (NANNEN; EIBEN, 2006), time series discretization (ACOSTA-MESA et al., 2014), finite state machines construction (CHIVILIKHIN; ULYANTSEV; SHALYTO, 2016), and others.

The race concept was initially described by (MARON; MOORE, 1997) as a machine learning technique to compare different models and find the statistically superior. Later the technique was adopted by (BIRATTARI et al., 2002) to configure parameters in optimization algorithms.

The IRACE has three main phases that repeat until a stopping criterion is met:

1. Sampling of new configurations according to a truncated normal distribution for continuous parameters, and according to a discrete probability for categorical parameters;

2. Selection of the best configuration among the new samples through a racing process;

3. Updating the sampling distribution to increase the probability that the best configurations are selected.

In the IRACE, each configurable parameter is associated with a sampling distribution independent of other parameters. Constraints and conditions are applied for the generation of each parameter. Continuous parameters use a truncated normal distribution, while categorical parameters use the discrete probability function described in López-Ibáñez et al. (2011) and López-Ibáñez et al. (2016). Ordinal parameters are considered integers. To update the sampling distributions, the average and standard deviation are adjusted in normal distributions, and the probability is altered in discrete distributions. The update of the sampling distribution is based on the best configurations so far, creating a type of elitism where the chances of selection of parameters close to best configurations increase.

The new configurations for the parameters are sampled from the distributions, the best are selected through racing. The configured models run until they reach: a minimum number of survival configurations; a maximum number of used instances; or a maximum computational limit B defined as a maximum computational time or ran experiment (execution of one configured model in one testing instance). Algorithm 9 details the IRACE pseudocode.

Algorithm 9 Iterated Racing Pseudocode

1: **procedure** ITERATEDRACING($I = \{I_1, I_2, ...\} \sim \mathcal{I}, X, C(\theta, i) \in \mathbb{R}, B$)
2: $\quad \Theta_i \leftarrow GenerateUniformDistribution(X)$
3: $\quad \Theta_{elite} \leftarrow Race(I, \Theta_1, B_1, C)$
4: $\quad j \leftarrow 1$
5: $\quad$ **while** $B^{used} \leq B$ **do**
6: $\qquad j \leftarrow j + 1$
7: $\qquad \Theta_{new} \leftarrow GenerateSample(X, \Theta_{elite})$;
8: $\qquad \Theta_j \leftarrow \Theta_{new} \cup \Theta_{elite}$
9: $\qquad \Theta_{elite} \leftarrow Race(I, \Theta_j, B_j, C)$;
10: $\quad$ **end while**
11: $\quad$ **return** Θ_{elite}
12: **end procedure**

At line 1, the algorithm receives as input:

1. The testing instances I, sampled from the problem space $\mathcal{I}$, over which the candidate models run;
2. The parameters X which will be automatically configured;
3. A cost function C to determine the quality of each configuration;
4. A computational limit B that is usually either a maximum execution time or a maximum number of experiments.

For the first iteration, the initial set of candidate configurations is sampled from a uniform distribution of each parameter's space X (line 2). Next, the best configurations are found through a race (line 3). At each iteration of the race, the configurations are applied to a problem instance and are evaluated according to the average cost C. Then the results are compared through a statistical test, that can be either a Friedman test

(FRIEDMAN, 1937), or a Student's t-test (STUDENT, 1908). If there is statistical evidence that some candidate configurations performed better than others, the worst configurations are discarded and the best configurations are tested in the next instance. At each new iteration, a new group of candidate configurations is generated through the sample distributions updated in the previous iteration (line 7). At line 8, the new candidates are combined with the best candidates from the previous iteration to form a new testing group. The new group races again in line 9 to determine the best solutions of the group. The procedure runs until the predefined computational limit is reached. In the end, the algorithm returns the best configuration found.

The Iterated Racing algorithm makes use of the *race* procedure summarized in Algorithm 10.

Algorithm 10 Racing procedure in irace

1: **procedure** RACE(I, Θ_{it}, B, C, I)
2: $B_{it} = SampleInstances(B)$
3: $\Theta_{elite} = \Theta_{it}$
4: **while** $B^{used} \leq B_{it}$ **do**
5: $b_i = SampleInstances(B_{it})$
6: $Execute(\Theta_{elite}, b_i)$
7: Identify non-dominant configurations $\Theta_{non-dominant}$ using statistical test
8: $\Theta_{elite} = \Theta_{elite} \setminus \Theta_{non-dominant}$ ▷ Eliminate non-dominant configurations
9: **end while**
10: **return** Θ_{elite}
11: **end procedure**

At line 1, the *race* algorithm receives as input:

1. The testing instances I, sampled from the problem space $\mathcal{I}$, over which the candidate models run;
2. The set of candidate configurations Θ to be tested;
3. The parameters X which will be automatically configured;
4. A cost function C to determine the quality of each configuration;
5. A computational limit B that is usually either a maximum execution time or a maximum number of experiments.

The algorithm initially sets all the input configurations as elite (line 2). Next, the algorithm samples a number of instances to be used for the racing process (line 3). After that, the algorithm enters the loop of executions to identify elite configurations while there are instances to run (line 4). At line 5, a subset of the racing instances is sampled for the current iteration. At line 6 the algorithm executes the configuration in a subset of instances b_i. The results of these executions are statistically analyzed in line 7 to identify non-dominant configuration. The poor-performing configurations are eliminated in line 8, remaining only the elite configurations. After executing these steps until no more instances are available to execute, the algorithm returns the best configurations found in line 10.

4.3 GRH PARAMETER TUNING

The GRH tunning aims to find the best configuration and confirm if the interval of $0 < \tau < 20$ for problems with up to 400 locations used in the original paper is reasonable. The parameters for the IRACE execution were:

- **τ:** Real values between 0 and 40;
- **Computational Limit:** 15,000 iterations;
- **Scenarios:** 1,296 instances formed from the combination[1] of the following factors:
  - **Rack Size:** 9 (3 x 3), 100 (10 x 10), 400 (20 x 20);
  - **Utilization:** 50%, 80%, 95%;
  - **Organization:** 0%, 50%, 85%;
  - **Start Location:** Coincide, Random, 0;
  - **Final Open Locations:** Coincide, Random;
  - **Loaded Move Cost:** Euclidean, Chebychev, Manhattan, Random;
  - **Unloaded Move Factor:** 1, Random.

The best configuration found by the IRACE for the GRH in the given scenarios was $\tau = 22$. The second best configuration found was $\tau = 13.6267$. The best configurations are not too distant from the ones used in the original paper, so the final experimental tests to compare all the heuristics can be performed using the GRH with $0 < \tau < 25$, guaranteeing the best τ is used.

See all configuration files in the Appendix C. Detailed results can be found in the GitHub link: https://github.com/FaridLeoBueno/Warehouse-Reshuffling.

4.4 BRKGA PARAMETER TUNING

The BRKGA tuning aims to find the configuration that is best suitable for the reshuffling process. This step was performed before the parameter configuration of the *MaxDistQuick* criterion, because the IRACE process is designed to adjust the parameters to improve only solution quality. If the stopping criterion is adjusted using the same method, the best configuration found by the IRACE would decrease the time performance of the

[1] Factors are combined to form each scenario. Example scenario: Rack 100, Utilization 50%, Organization 0%, Start 0, Final Open Locations Random, Loaded Cost Chebyshev, Unloaded Factor 1.

algorithm to increase the search and consequently the chances of finding better solutions. In this case, only the number of maximum generations was set to be configured by the IRACE. For reshuffling BRKGA, the IRACE parameters were defined as follows:

- **Population size (p):** Integer value between 1 and 100;
- **Elite population fraction (p_e):** Real value between 0 and 1;
- **Mutant population fraction (p_m):** Real value between 0 and 1;
- **Probability of inherit allele from elite parent (ρ_e):** Real between 0 and 1;
- **Number of separated populations (K):** Integer value between 1 and 5;
- **Maximum number of generations ($\boldsymbol{MAXGEN}$):** Integer between 50 and 3,000;
- **Top individuals exchanged between populations (X_NUMBER):** Integer value between 2 and 5;
- **Generation interval to exchange top individuals between populations (X_INTVL):** Integer value between 30 and 300.

As described in the paper (TOSO; RESENDE, 2015), the BRKGA parameters have the following constraints:

$$p_e + p_m \leqslant 1 \tag{4.1}$$

$$p_e * p \geqslant 1 \tag{4.2}$$

$$X\_NUMBER * K \leqslant p_e * p \tag{4.3}$$

$$X\_INTVL \leqslant MAXGEN \tag{4.4}$$

To reduce the processing time, but guarantee good generalization of the tunned BRKGA, the algorithm ran for 4,000 iterations on the 432 scenarios from the combinations[2] of the following factors:

- **Rack Size:** 100 (10 x 10);
- **Utilization:** 50%, 80%, 95%;
- **Organization:** 0%, 50%, 85%;

[2] Factors are combined to form each scenario. Example scenario: Rack 100, Utilization 50%, Organization 0%, Start 0, Final Open Locations Random, Loaded Cost Chebyshev, Unloaded Factor 1.

- **Start Location:** Coincide, Random, 0;
- **Final Open Locations:** Coincide, Random;
- **Loaded Move Cost:** Euclidean, Chebychev, Manhattan, Random;
- **Unloaded Move Factor:** 1, Random.

These experiments are the same ones used for the GRH, but executing only on racks with 100 locations. This size of racks reduces the total processing time of the *iterated racing* process without reducing much of the problem complexity. The expectation is that the best configuration found for this size will also perform well when scaled to larger scenarios.

The best configurations found by the IRACE for the BRKGA in the given scenarios were:

Table 3 – Best BRKGA automatic parameter configurations ranked according to the solution quality.

Ranking					**Parameters**			
	p	p_e	p_m	ρ_e	K	MAXGEN	X_NUMBER	X_INTVL
1	78	0.1625	0.2631	0.3122	4	2982	2	40
2	77	0.1458	0.3402	0.3317	4	2987	2	46
3	84	0.1714	0.2281	0.4196	4	2895	2	33
4	87	0.2497	0.1856	0.4032	4	2889	2	32
5	87	0.1318	0.2655	0.3220	4	2906	2	37

Since the configuration 1 was the best ranked, it was used in the rest of this study with one modification. The MAXGEN values found was 2,982. To simplify the algorithm operation, a maximum value of 3,000 was applied.

See all configuration files in Appendix C.

4.5 BRKGA STOPPING CRITERIA TUNING

As presented in Chapter 3, the maximum number of generations executed *MAXGEN* and the *MaxDistQuick* stopping criterion were applied in the reshuffle BRKGA to reduce processing time. The stopping criterion was combined with the maximum number of generations executed *MAXGEN* in order to ensure an upper limit of generations executed in case the *MaxDistQuick* criterion does not converge. The *MaxDistQuick* was used to benefit from the fact that the BRKGA already sorts the whole population using the fitness, facilitating the evaluation in a fraction of the population.

The *MaxDistQuick* criterion has two important parameters. The maximum distance threshold m and the fraction of the population to be evaluated CP. For the criterion to converge, the best CP*p individuals of the population should have a maximum distance from the fittest individual lower than $m * f(x_{\text{Best}})$. Following the suggestions of the original study (ZIELINSKI; PETERS-DROLSHAGEN; LAUR, 2005) for evolutionary algorithms, the parameter m was set to 0.001. The study also recommends $0.3 \leqslant CP \leqslant 0.6$.

The CP parameter has a direct impact on the moment of convergence of the algorithm. Small CP means that only a small fraction of the top of the total population will be used to evaluate the phenotypical diversity of the individuals. Therefore, small CP can result in premature convergence, since the diversity of a few of the fittest individuals of the population can more easily converge. In contrast, large CP means that a large fraction of the top of the population needs to have phenotypical similarity to allow termination of the algorithm. Therefore, a large CP can delay the convergence until a larger size of the population slowly converges to similar fitness values. The delay allows the algorithm to search longer for a better solution before termination.

To decide which CP to use for the BRKGA, a convergence analysis test was performed. The tests evaluated the impact of the parameter CP on the solution quality and the convergence (measured using the generation in which the algorithm was terminated at each execution). The best CP would result in a reduction of processing time while maintaining good results in solution quality.

The test was executed on the same 432 scenarios used for the BRKGA automatic parameter tuning. In each execution of the BRKGA a different seed was used for the random number generator. The 5 seeds were taken from the decimal places of π and can be seen in Table 4. The evaluated CP were: 0.3; 0.45; 0.6. The fraction values are smaller than the non-mutant population of the tunned BRKGA (since the best p_m found by the irace was $p_m = 0.2631$, the non-mutant fraction of the population is $1 - p_m = 0.7369$). All values are also within the optimum range found in the original study (ZIELINSKI; PETERS-DROLSHAGEN; LAUR, 2005).

Table 4 – Seeds for the random number generator for convergence analysis.

1415926535	8979323846	2643383279	5028841971	6939937510

The next section analyzes the results to select the best CP for the developed heuristic.

4.5.1 Comparison Between Stopping Criteria Configurations

In order to compare the test results and select the most suitable CP for the reshuffling BRKGA, it was used as performance measures the average quality of solutions ($\overline{Z}$) and the average number of executed generations until algorithm termination ($\overline{Gen}$). Table 5 outlines the obtained convergence results.

Find all results in the GitHub link: https://github.com/FaridLeoBueno/Warehouse-Reshuffling.

Table 5 – Comparison between convergence configurations with respect to solution quality $\overline{Z}$ and generation executed until termination $\overline{Gen}$.

Property	**Configurations**		
	CP = 0.30	**CP = 0.45**	**CP = 0.60**
$\overline{Z_{\text{MIN}}}$	1224.12	1217.51	**1217.15**
$\overline{Z_{\text{MAX}}}$	1248.17	1234.80	**1234.20**
$\overline{Z}$	1235.86	1226.51	**1226.09**
Ave. S.D.	10.36	7.30	**7.19**
$\overline{Gen_{\text{MIN}}}$	**150.28**	2324.95	2551.49
$\overline{Gen_{\text{MAX}}}$	**1093.71**	2529.64	2659.90
$\overline{Gen}$	**543.44**	2438.01	2599.54
Ave. S.D.	**422.80**	98.25	52.98
%conv	**99.54%**	23.15%	15.05%

From Table 5 it is noticeable that the configuration CP = 0.30 converges early almost 100% of executions. This observation combined with the fact that the configuration yields worse solution qualities in all measures is an indication of premature convergence. On the other hand, configurations CP = 0.45 and CP = 0.60 have very similar solution qualities and convergence.

In order to confirm if the results of the three configurations are significantly different in terms of quality and performance, the Non-parametric *Friedman* Test was used with a significance level of 0.05.

Solution Quality

For the solution quality test, the hypotheses were defined as:

H_0 : The configurations have the same statistical quality;

H_1 : The configurations have different statistical quality;

If the result of *p-Value* is lower than 0.05, the null hypothesis that the approaches were defined as equal, is rejected and it is possible to assume with 95% of certainty that there was a difference between at least one pair of the analyzed samples.

The detailed results are listed in Table 6, where the lowest ranking indicates the best configuration, and the highest ranking is the worst configuration.

Table 6 – Friedman Tests for convergence configurations solution qualities.

Friedman Test	
F-Value:	353.274
p-Value:	1.110e-16
Average Ranking	
Configuration	Ranking
CP = 0.30	2.432
CP = 0.45	1.803
CP = 0.60	**1.764**

From these results, the *p-Value* obtained was lower than 0.05, confirming the configurations have solutions statistically different. The test also ranked the configuration CP = 0.60 as the best.

Since the null hypothesis of the *Friedman* test was rejected, the *Nemenyi* post-hoc test was applied to compare data at each execution and measure the significance difference between them. As in the previous test, if *p-Value* $\geqslant 0.05$, the configurations have similar results statistically, while *p-Value* < 0.05 indicate significant statistical difference between the solutions. The test results are listed in Table 7.

Table 7 – Nemenyi Post-hoc Test for convergence configurations solution qualities.

Nemenyi Post-hoc Test	
CP = 0.30 X CP = 0.45	
Z-value:	20.668
p-Value:	0.000
p-value adjusted	0.000
CP = 0.30 X CP = 0.60	
Z-value:	21.962
p-Value:	0.000
p-value adjusted:	0.000
CP = 0.45 X CP = 0.60	
Z-value:	1.293
p-Value:	0.196
p-value adjusted:	0.588

From the post-hoc test results it is possible to confirm that both configurations CP = 0.45 and CP = 0.60 are statistically different from CP = 0.30, with both comparisons having *p-Value* under 0.05. Since they are also better ranked in the *Friedman* test, we can discard the latter configuration as inappropriate for the project.

The comparison between CP = 0.45 and CP = 0.60, on the other hand yields a *p-Value* = 0.196 > 0.05. This means these approaches are statistically equivalent and it is not possible to decide between them based only on the quality results.

To decide between these options and ensure the best balance between solution quality and processing time given by the stopping criteria, a statistical analysis of the generations executed until termination was performed.

Executed Generations until termination

To analyze the best configuration in terms of the executed generations until termination, the hypothesis for the *Friedman* Test were defined as:

$$H_0: \quad \text{The configurations have similar performance;}$$

$$H_1: \quad \text{The configurations have different performance;}$$

The test results are detailed in Table 8, where again the statistical difference can be evaluated using *p-Value*.

Table 8 – Friedman Tests for convergence configurations solution performance.

Friedman Test	
F-Value:	1911.301
p-Value:	1.110e-16
Average Ranking	
Configuration	Ranking
CP = 0.30	**1.210**
CP = 0.45	2.353
CP = 0.60	2.436

From this test, the CP = 0.30 has the better rank. This result is expected since almost 100% of its executions had early convergence.

As in the quality test, the null hypothesis was rejected and the *Nemenyi* post-hoc test was applied to evaluate the statistical difference between a pair of samples. The results are listed in Table 9

Table 9 – Nemenyi Post-hoc Test for convergence configurations solution performance.

Nemenyi Post-hoc Test	
CP = 0.30 X CP = 0.45	
Z-value:	37.572
p-Value:	0.000
p-value adjusted	0.000
CP = 0.30 X CP = 0.60	
Z-value:	40.296
p-Value:	0.000
p-value adjusted:	0.000
CP = 0.45 X CP = 0.60	
Z-value:	2.723
p-Value:	0.006
p-value adjusted:	0.019

The post-hoc test of the executed generations once again confirm both configurations CP = 0.45 and CP = 0.60 are statistically different from CP = 0.30. The comparison between configurations CP = 0.45 and CP = 0.60 has $p\text{-}Value = 0.006 < 0.05$, which indicates the execution times of these configurations are also significantly different.

Configurations CP = 0.45 and CP = 0.60 have similar solution qualities but significantly different execution times, in order to speed up the final simulation, the option CP = 0.45 was selected for being better ranked in execution generations.

4.6 FINAL RESHUFFLING BRKGA CONFIGURATION

After the parameter adjustment phase, the BRKGA used for reshuffle problems has the following configuration:

- **BRKGA Configuration:**
  - **Population size (p):** 78;
  - **Elite population fraction (p_e):** 0.1625;
  - **Mutant population fraction (p_m):** 0.2631;
  - **Probability of inherit allele from elite parent (ρ_e):** 0.3122;
  - **Number of separated populations (K):** 4;
  - **Maximum number of generations ($\boldsymbol{MAXGEN}$):** 3,000;
  - **Top individuals exchanged between populations (X_NUMBER):** 2;
  - **Generations interval to exchange top individuals between populations (X_INTVL):** 40;

- **Stopping Criteria Configuration:**
  - **Maximum phenotypical distance between individuals ($\boldsymbol{m}$):** 0.001;
  - **Convergence Population Fraction (CP):** 0.45.

5 EXPERIMENTAL ANALYSIS

To compare the quality and performance of the developed heuristics, the reference heuristics and the reshuffle BRKGA were executed within all the scenarios suggested by Carlo e Giraldo (2012) and the additional scenarios formed in this study. Tables have been created to analyze and compare the data in order to evaluate the contributions of this study.

The tables detail the results in terms of relative solution quality and execution time. Each table registers the average of the obtained results, as well as the percent comparison between heuristics in each of the operating environment tested.

Statistical tests were performed on the results and computational times to evaluate the relevance of the proposed algorithm. The non-parametric *Friedman* test (FRIEDMAN, 1937) was used in combination with the *Nemenyi* post-hoc test (NEMENYI, 1963) to compare the different heuristics.

5.1 COMPUTATIONAL ENVIRONMENT

All algorithms were coded in C++11 (LANGUAGES, 2011) using Eclipse Neon 3 IDE for C/C++ Developers (Eclipse Contributors, 2016) and Minimalist GNU for Windows (MinGW) 64bits Release 5.0. The experiments were run on an Asus K43E personal computer with a 2,30 GHz Intel Core i5 2410M Processor, 8GB RAM DDR3, and 256GB Solid State Drive (SSD), operating on Windows 10 Pro 64bits. The algorithms were developed based on the Application Programming Interface (API) for the BRKGA proposed in Toso e Resende (2015).

5.2 EXPERIMENTAL DESIGN

The final performance of the heuristics was analyzed through the full factorial experimental expanded from the scenarios used by Carlo e Giraldo (2012) by adding variation in the final locations of open positions as seen in Pazour e Carlo (2015). The final scenarios resulted from the combinations of the following factors:

- **Rack Size:** 9 (3 x 3) ,100 (10 x 10), 400 (20 x 20);
- **Utilization:** 50%, 80%, 95%;
- **Organization:** 0%, 50%, 85%;
- **Start Location:** 0;
- **Final Open Locations:** Coincide, Random;

- **Loaded Move Cost:** Euclidean, Chebychev;
- **Unloaded Move Factor:** 1.

Five instances of each combination were generated for the experiments, resulting in a total of 540 instances. Each scenario ran 10 times. In each execution of the BRKGA a different seed was used for the random number generator. The 10 seeds were taken from the decimal places of π and can be seen in Table 10. For each run, the GRH had the τ parameter variated with integer numbers from 0 to 25, passing through the optimum values found during the tuning process.

Table 10 – Seeds for the random number generator.

1415926535	8979323846	2643383279	5028841971	6939937510
5820974944	5923078164	8628034825	3421170679	8214808651

In the end, the average values of the obtained results and computational times were calculated, as well as the percentage of iterations that reached the maximum distance stopping criteria.

5.3 RESULTS

The analysis of the performance of the heuristics is performed in each operating environment of the problem. This approach is used to verify the impact of the design assumptions when optimizing different problems. For example, it is expected that larger scales scenarios provide more opportunities for the flexibility of the BRKGA to find better solutions in comparison to the benchmark approaches.

The following analysis evaluates the impact of each operating environment by solution quality and runtime. The tables display data averaging the results of all executions over all instances of each scenario combination. To facilitate comparisons with references, the tables display the results following the design used in the literature. The reported parameters are:

- $\overline{Z}$**:** Average cost found by heuristic;
- $\overline{\%diffGRH}$**:** Average of percentual difference between best results found by BRKGA and GRH in each scenario;
- **%best** Z**:** Percentage of instances the BRKGA found a solution as good as or better than GRH;
- $\overline{RT}$**:** Average run-time of the heuristic for one instance;
- $\overline{Gen}$**:** Average end generation of the BRKGA for each scenario;

- **%conv:** Percentage of instances in which the BRKGA converged early due to the *maxDist* stopping criteria;

The tables with detailed results and computational times can be found in the GitHub link: https://github.com/FaridLeoBueno/Warehouse-Reshuffling.

Table 11 – The average quality results of BRKGA, GRH, and H3 with respect to each operating environment.

Operating Environment		$\overline{Z_{\mathbf{H3}}}$	$\overline{Z_{\mathbf{GRH}}}$	$\overline{Z_{\mathbf{BRKGA}}}$	$\overline{\%diffGRH}$	**%best** Z
Average of all instances		1203.36	1011.79	946.87	6.73 ± 6.37	91.67
Rack Size	Small (9)	14.83	14.14	13.05	4.73 ± 8.51	83.33
	Medium (100)	418.33	371.03	330.36	9.68 ± 4.75	97.22
	Large (400)	3176.92	2650.20	2494.17	5.77 ± 3.90	94.44
Final Open Locations	Random	1040.49	910.01	873.74	4.96 ± 4.65	88.89
	Equal	1366.23	1113.58	1020.00	8.49 ± 7.35	94.44
Utilization	50%	753.53	647.48	608.64	5.59 ± 6.66	94.44
	80%	1266.15	1063.93	998.29	6.73 ± 5.77	88.89
	95%	1590.39	1323.97	1233.69	7.86 ± 6.63	91.67
Organization	0%	1826.19	1919.76	1711.06	8.86 ± 7.65	91.67
	50%	1267.86	918.36	860.93	6.34 ± 5.26	100.00
	85%	422.46	290.83	268.62	4.98 ± 5.49	83.33
Distance Metric	Chebyshev	1136.69	955.78	892.36	6.56 ± 6.79	92.59
	Euclidean	1270.02	1067.81	1001.38	6.89 ± 5.99	88.89

As indicated in Table 11, in all the analyzed scenarios the BRKGA found, in average, better solutions than the benchmark heuristics, resulting in an average improvement of the solution quality of 6.73%. Observing the boxplot in Figure 22, it is clear that in all the scenarios the BRKGA found better solutions in at least 75% of the instances. The improvement was more relevant in scenarios with coincident final open locations where there was an average improvement of 8.49% with some instances having over 25% improvement. This result is particularly important because these scenarios were also the ones which the GRH had higher improvements over the H3. Apparently, the nearby cycle break used in the GRH and in the BRKGA is a relevant technique for handling such scenarios. In scenarios with 0% of organization, the most complex cases, the solutions found were significantly better than those of the GRH, resulting in 8.86% improvement in solution quality, also with some instances having over 25% improvement. This result demonstrates the potential of the BRKGA in finding better solutions in the most extreme cases. This result is very relevant because the situation of a very low organization is when the storage has no policies to organize the stock and needs to implement one. In this case, the BRKGA is significantly better than the best heuristics in the literature.

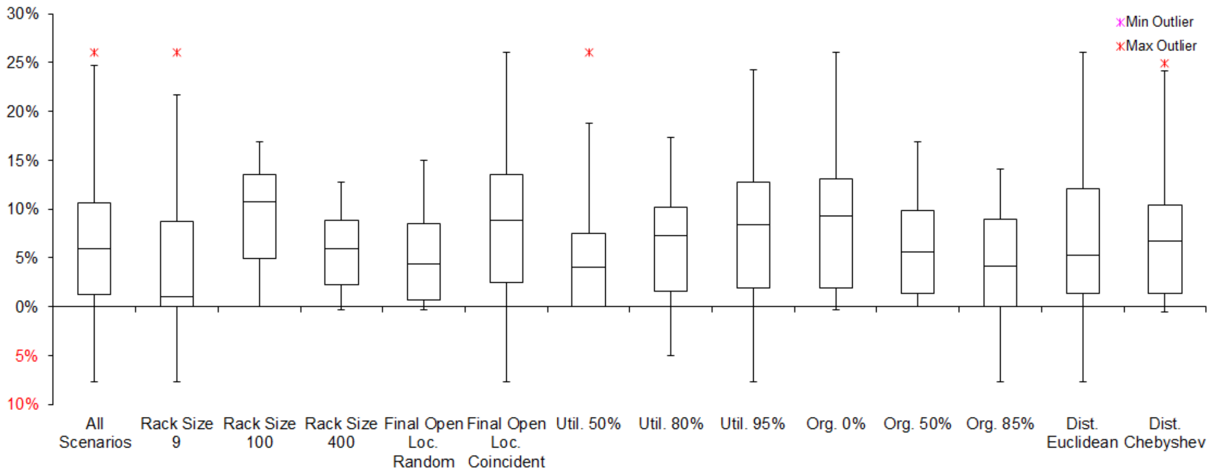

Figure 22 – Boxplot of the average of the percentile difference between best results found by BRKGA and GRH with respect to each operating environment.

Another very important result is the improvement of 7.86% in scenarios with high utilization. These cases are very relevant because in these scenarios the GRH improvements over the H3 were not as significant as in scenarios with lower utilization. These results indicate that the added flexibility of the nearby cycle distance configuration of the BRKGA decoder added a strong tool to handle scenarios with fewer open locations.

On average, in 8.33% of the instances, the BRKGA found worse solutions than the GRH. This behavior seems to be partially due to premature convergence. In other words, the stopping criteria may be terminating the BRKGA before it searches enough the solution space and finds better solutions for the problem than the ones found by the GRH. As can be interpreted from the processing time and stopping generations of the BRKGA reported in Table 12, the genetic algorithm converged early in average 43.28% of the executions. This interpretation can also be observed in the specific operating environment of 50% utilization. In these cases, the BRKGA only converged early in 23.38% of the executions, the lowest convergence percentage in all operating environments. As a result, the BRKGA found better instances in 100% of the analyzed scenarios. This early convergence capability was added to reduce the processing time to a practical range of operation in larger scenarios. As a result, the average runtime of large scenarios is about 30 minutes, within the 1-hour limit frequently applied in the literature. Nevertheless, the early convergence apparently created a problem that was not observed during the convergence analysis phase, because the results were not compared against the GRH results at that moment. To evaluate the influence of the stopping criterion in the final results, it was calculated the linear correlation between $\%bestZ$ and $\%conv$. On a scale of 1 to -1, where 0 indicates no correlation, the obtained correlation was -0.12. Therefore the early convergence has low correlation with the low performance of the BRKGA in certain scenarios. In other words, the early convergence does not have a negative effect on the

BRKGA solution quality.

Future research is needed in order to ensure the early convergence does not negatively affect the performance and to better balance the trade-off between solution quality and processing time. One approach to be studied is to add a minimum threshold of generations iterated before analyzing the *MaxDistQuick* criterion to terminate the execution. The genetic algorithm can be forced to run 1% of the total generations before allowing convergence, for example. This would force the algorithm to search for more solutions before returning the final results.

Since the early convergence is not responsible for the degraded performance of the BRKGA in specific scenarios, the problem may be when searching the solution space. A solution found by the GRH is equivalent to a chromosome with all allele with the same values, resulting in a reshuffling process where the τ value is the same in all movement decisions. This situation is highly unlikely to be generated when creating individuals using random generators. One technique that could be tested to avoid such issue is to add the best solution found by the GRH in the initial population of the BRKGA. This would create an elite individual in the initial population that could be genetically enhanced throughout the iterative process. This proposed modification may add a bias towards GRH elite solutions that may need to be compensated with larger diversity in the population. For this reason, the modified BRKGA with the addition of GRH elite individuals may need to have the parameters again tuned through the IRACE process.

There is a noticeable variation in processing times. For a small scenario, the BRKGA used on average 0.675s. For a medium scenario, the BRKGA ran on average for 2m 59.145s. In large scenarios, the BRKGA required on average 29m 56.672s. From these results, we observe that for an increase of 11.11 in the scenario size (from small to medium scenarios) the BRKGA runtime increased 265.57 times, while an increase of 4 times in the scenario size (from medium to large scenarios) the BRKGA runtime increased about 9.78 times. As expected, larger scenarios had larger runtimes. However, the larger scenarios also have lower convergence rates, which can explain the significant difference between the execution times of small and medium scenarios. These observations demonstrate the capability of the stopping criteria in reducing the processing time.

The quickest scenarios were the ones with 0% of organization with an average runtime of 6m 58.626s, while the longest scenarios were the ones with 50% organization with 15m 1.083s. From these results, retailers and warehouse managers can derive organization policies that could either allow the storage to have lower organization before starting reshuffling or having more frequent reshuffling activities while the storage has over 50% organization.

Table 12 – The average runtime results of BRKGA, GRH, and H3
with respect to each operating environment.

Operating Environment		$\overline{RT}_{\mathbf{H3}}$	$\overline{RT}_{\mathbf{GRH}}$	$\overline{RT}_{\mathbf{BRKGA}}$	$\overline{Gen}$	**%conv**
Average of all instances		4.65	5.38	643830.61 ± 943256.55	1388.07	43.28
Rack Size	Small (9)	0.03	0.05	674.57 ± 1370.14	229.5	93.89
	Medium (100)	0.60	0.69	179144.97 ± 181486.40	2404.1	20.72
	Large (400)	12.94	15.42	1751672.27 ± 881562.09	2546.71	15.24
Final Open Locations	Random	5.71	7.19	574476.28 ± 999985.35	1406.1	53.72
	Equal	3.34	3.58	713184.93 ± 886837.64	2047.4	32.85
Utilization	50%	4.77	5.86	668832.35 ± 969424.66	1433.0	35.42
	80%	5.71	6.82	682476.20 ± 1007102.74	1822.51	26.70
	95%	3.09	3.47	580183.27 ± 871581.73	1924.7	24.44
Organization	0%	2.62	4.86	408626.19 ± 737550.91	1387.0	36.96
	50%	7.38	8.29	871083.48 ± 1067656.36	1958.1	23.38
	85%	3.58	3.00	651782.15 ± 962332.07	1835.2	26.22
Distance Metric	Chebyshev	5.49	4.56	643341.82 ± 936529.19	1686.6	44.70
	Euclidean	5.28	4.49	644319.39 ± 958731.37	1766.9	41.86

To ensure the interpretations extracted from the average data are not distorted, statistical hypothesis analysis using *Friedman* test combined with the *Nemenyi* post-hoc test were performed.

5.4 STATISTICAL ANALYSIS

5.4.1 Solution Quality

To prove the BRKGA had a significant statistical difference in comparison with the other heuristics in terms of solution quality ($\overline{Z}$), the *Friedman* test was performed considering a significance level of 0.05. It was assumed as the hypothesis for the statistical analysis that:

$$H_0: \quad \text{The heuristics have similar solution quality;}$$

$$H_1: \quad \text{The heuristics have different solution quality.}$$

The test results are detailed in Table 13, where the statistical difference can be evaluated using *p-value*.

Table 13 – Friedman Test for solution quality ($\overline{Z}$) results of BRKGA, GRH, and H3 with respect to each operating environment.

Friedman Test	
F-Value:	263.974
p-Value:	1.110e-16
Average Ranking	
Heuristic	Ranking
BRKGA	**1.181**
GRH	1.954
H3	2.866

From this test, the BRKGA has the better rank. This result is expected since 91.67% of the scenarios the BRKGA found significantly better solutions than the best benchmark approach.

From Table 13 we observe that the *p-Value* is lower than the confidence level. In other words, the null hypothesis was rejected. From these results, the *Nemenyi* post-hoc test was applied to evaluate the statistical difference between a pair of samples. The results are listed in Table 14

Table 14 – Nemenyi Post-hoc Test for solution quality ($\overline{Z}$) results of BRKGA, GRH, and H3 with respect to each operating environment.

Nemenyi Post-hoc Test	
BRKGA X GRH	
Z-value:	5.681
p-Value:	1.336e-08
p-value adjusted	4.007e-08
BRKGA X H3	
Z-value:	12.384
p-Value:	0.000
p-value adjusted:	0.000
GRH X H3	
Z-value:	6.838
p-Value:	8.022e-12
p-value adjusted:	2.407e-11

Observing that the p-Values are all lower than 0.05, the post-hoc test of the solution qualities confirms that the heuristics are statistically different from each other.

5.4.2 Runtime

The *Friedman* test was also performed using the runtime results to prove significant processing time difference between the heuristics. The test considered a significance level of 0.05. It was assumed as the hypothesis for the statistical analysis that:

$$H_0 : \quad \text{The heuristics have similar runtimes;}$$

$$H_1 : \quad \text{The heuristics have different runtimes.}$$

The test results are detailed in Table 15, where again the statistical difference can be evaluated using *p-value*.

Table 15 – Friedman Test for runtime ($\overline{RT}$) results of BRKGA, GRH, and H3 with respect to each operating environment.

Friedman Test	
F-Value:	437.761
p-Value:	1.110e-16
Average Ranking	
Heuristic	Ranking
BRKGA	3.000
GRH	1.731
H3	**1.269**

From this test, the BRKGA has the worst rank. This result was expected because the BRKGA executes several times the adapted GRH algorithm as a decoder for the chromosomes.

As in the quality test, the null hypothesis was rejected, indicating that the heuristics are statistically different in terms of processing time. The *Nemenyi* post-hoc test was applied to evaluate the statistical difference between a pair of heuristics. The results are listed in Table 16

Table 16 – Nemenyi Post-hoc Test for runtime ($\overline{RT}$) results of BRKGA, GRH, and H3 with respect to each operating environment.

Nemenyi Post-hoc Test	
BRKGA X GRH	
Z-value:	9.322
p-Value:	0.000
p-value adjusted:	0.000
BRKGA X H3	
Z-value:	12.724
p-Value:	0.000
p-value adjusted:	0.000
GRH X H3	
Z-value:	3.402
p-Value:	0.001
p-value adjusted:	0.002

The post-hoc test confirmed the statistical difference between the heuristics in terms of processing time.

6 CONCLUSIONS AND FUTURE RESEARCH

From the warehouse strategy where the item locations are reassigned to create a new layout configuration that will improve product picking and putting-away performance, storage reshuffling is the procedure to move items from the original to the final configuration.

This study had as major goals to introduce a Biased Random-Keys Genetic Algorithm (BRKGA) to solve unit-load single handled reshuffling problems and quantify its results in common scenarios studied in the literature for the warehouse reshuffling problem against the most recent and successful benchmark references, heuristic H3 (CARLO; GIRALDO, 2012) and the General Reshuffling Heuristic (GRH) (PAZOUR; CARLO, 2015).

The designed BRKGA uses as decoder an adaptation of the General Reshuffling Heuristic (GRH), the best-published reshuffling heuristic in the literature. To do so, the chromosome of the BRKGA dynamically modify the τ parameter used by the GRH as a threshold to select nearby cycles to break. This adaptation results in an added flexibility of the nearby cycles distance threshold and allows the heuristic to search the reshuffling solution in a broader solution space.

Using a scenario-generator created to generate reshuffle scenarios with different sizes, utilization percentage, organization percentage, distance metrics, initial material handling position, final configuration of open locations, and rack design, an exhaustive full factorial experiment was executed to compare the heuristics and to quantify the effect that the BRKGA design assumptions and different operating environments have on performance.

Based on statistical tests, the BRKGA proved to be significantly different from the previously published reshuffling heuristics. From the experimental results, it was concluded that the reshuffling BRKGA outperforms the benchmark heuristics in all scales and operating environments. By analyzing the experiments, it was observed that the reshuffling BRKGA outperforms the GRH on average by 6.73%. For scenarios with coincident final open locations, which the GRH had significant improvement over the H3, the previously best reshuffling heuristic, the BRKGA improved 8.49% the solution qualities. In scenarios with 0% organization, the BRKGA outperformed the GRH by 8.86%, while in scenarios with 95% utilization the BRKGA outperformed the reference in 7.86%. These results indicate the potential of the technique to solve highly complex scenarios.

In 8.33% of the tested scenarios, the BRKGA could not find a solution better or equivalent to the GRH possibly due to space search problems. Solutions to this issue may include adding the best solution found by the GRH in the initial population of the BRKGA, this way the genetic algorithm would have a reference elite individual to enhance on. To complement the previous approach, it is possible to improve the stopping criteria and reduce the chances of premature convergence by adding a threshold of executed generations before verifying the *MaxDistQuick* stopping criterion.

Another issue to be studied more carefully is the processing time. Although the BRKGA had average run-time in larger scenarios of about 30 minutes (significantly less than one hour per instance, as assumed in the literature as a practical solution time), the processing times increase with the size of the scenarios, which may limit the application of the algorithm in real scale scenarios. To guarantee scalability to larger scenarios, the algorithm complexity should be evaluated in future research. An eventual processing time limitation may be solved by applying the BRKGA-Levy-LS introduced in Moura (2018) which had better performance than the canonical BRKGA applied in this study. Another approach to be studied is the study of another convergence criterion that is better suitable for the BRKGA and the reshuffling problems.

6.1 FUTURE RESEARCH

Future research may include:

- Introducing GRH solution as elite individual in the initial population of the BRKGA to improve performance;
- Improving convergence of the BRKGA by adding a threshold of minimum number of generations executed before evaluation of *QuickMaxDist* stopping criterion, or investigating better criterion for the heuristic;
- Testing reshuffling using the BRKGA-Levy-LS (MOURA, 2018);
- Analyzing the impact of different distance metrics in the final solutions;
- Analyzing the quality of the BRKGA in scenarios with only one open location;
- Optimizing scenarios with heterogeneous items and storage location sizes;
- Optimizing scenarios with multiple material handlers working together;
- Optimizing scenarios with multiple intermediary movements until conducting item to final location;
- Considering reshuffling using a dynamic SLAP;
- Combining reshuffling policies with RWW from Carlo e Giraldo (2012).

REFERENCES

ACOSTA-MESA, H.-G.; RECHY-RAMÍREZ, F.; MEZURA-MONTES, E.; CRUZ-RAMÍREZ, N.; JIMÉNEZ, R. H. Application of time series discretization using evolutionary programming for classification of precancerous cervical lesions. *Journal of biomedical informatics*, Elsevier, v. 49, p. 73–83, 2014.

ADEWOLE, A.; OTUBAMOWO, K.; EGUNJOBI, T.; NG, K. A comparative study of simulated annealing and genetic algorithm for solving the travelling salesman problem. *International Journal of Applied Information Systems*, v. 4, p. 6–12, 10 2012.

ASGARI, N.; NIKBAKHSH, E.; HILL, A.; FARAHANI, R. Z. Supply chain management 1982–2015: a review. *IMA Journal of Management Mathematics*, v. 27, n. 3, p. 353–379, 2016. Disponível em: <http://dx.doi.org/10.1093/imaman/dpw004>.

BEAN, J. C. Genetic algorithms and random keys for sequencing and optimization. *ORSA journal on computing*, INFORMS, v. 6, n. 2, p. 154–160, 1994.

BENAVIDES, A. J.; RITT, M. Iterated local search heuristics for minimizing total completion time in permutation and non-permutation flow shops. In: *ICAPS*. [S.l.: s.n.], 2015. p. 34–41.

BIRATTARI, M.; STÜTZLE, T.; PAQUETE, L.; VARRENTRAPP, K. A racing algorithm for configuring metaheuristics. In: MORGAN KAUFMANN PUBLISHERS INC. *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. [S.l.], 2002. p. 11–18.

BOTSALI, A. R. Comparison of simulated annealing and genetic algorithm approaches on integrated process routing and scheduling problem. *International Journal of Intelligent Systems and Applications in Engineering*, İsmail SARITAŞ, p. 101 – 104, 2016.

CARLO, H. J.; GIRALDO, G. E. Toward perpetually organized unit-load warehouses. *Computers and Industrial Engineering*, v. 63, n. 4, p. 1003 – 1012, 2012. ISSN 0360-8352. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0360835212001611>.

CHARLES, D. On the origin of species by means of natural selection. *Murray, London*, 1859.

CHEN, L.; LANGEVIN, A.; RIOPEL, D. A tabu search algorithm for the relocation problem in a warehousing system. *International Journal of Production Economics*, v. 129, n. 1, p. 147 – 156, 2011. ISSN 0925-5273. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0925527310003506>.

CHIVILIKHIN, D.; ULYANTSEV, V.; SHALYTO, A. A. Modified ant colony algorithm for constructing finite state machines from execution scenarios and temporal formulas. *Automation and Remote Control*, Springer, v. 77, n. 3, p. 473–484, 2016.

CHRISTOFIDES, N.; COLLOFF, I. The rearrangement of items in a warehouse. *Operations Research*, v. 21, n. 2, p. 577–589, 1973. Disponível em: <https://doi.org/10.1287/opre.21.2.577>.

CLEVELAND, G. A.; SMITH, S. F. Using genetic algorithms to schedule flow shop releases. In: *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989. p. 160–169. ISBN 1-55860-066-3. Disponível em: <http://dl.acm.org/citation.cfm?id=645512.657259>.

CORSTEN, D.; GRUEN, T. Stock-Outs Cause Walkouts. *Harvard Business Review*, v. 82, n. 5, p. 26–28, 2004. Disponível em: <http://www.redi-bw.de/db/ebsco.php/search.ebscohost.com/login.aspx?direct=true&db=buh&AN=12932512&lang=de&site=ehost-live>.

DELL, M.; IORI, M.; NOVELLANI, S.; STÜTZLE, T. et al. A destroy and repair algorithm for the bike sharing rebalancing problem. *Computers & Operations Research*, Elsevier, v. 71, p. 149–162, 2016.

DESCARTES, R.; ARIEW, R. *Philosophical Essays and Correspondence*. Hackett Pub., 2000. (Hackett Classics Series). ISBN 9780872205024. Disponível em: <https://books.google.com.br/books?id=F3Ob74iLXwMC>.

Eclipse Contributors. *Eclipse documentation - Eclipse Neon*. 2016. Disponível em: <http://help.eclipse.org/neon/index.jsp>.

FEO, T. A.; RESENDE, M. G. Greedy randomized adaptive search procedures. *Journal of global optimization*, Springer, v. 6, n. 2, p. 109–133, 1995.

FRIEDMAN, M. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the american statistical association*, Taylor & Francis, v. 32, n. 200, p. 675–701, 1937.

GHAZANFARI, M.; ALIZADEH, S.; FATHIAN, M.; KOULOURIOTIS, D. Comparing simulated annealing and genetic algorithm in learning fcm. *Applied Mathematics and Computation*, v. 192, n. 1, p. 56 – 68, 2007. ISSN 0096-3003. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0096300307002949>.

GIRALDO, G. E. *Metodología Para la Reorganización Perpetua de Almacenes*. Dissertação (Mestrado) — University of Puerto Rico, Mayaguez, 2011.

GOLDBERG, D. E. *Genetic algorithms*. [S.l.]: Pearson Education India, 2006.

GOLDBERG, D. E.; LINGLE, R. et al. Alleles, loci, and the traveling salesman problem. In: LAWRENCE ERLBAUM, HILLSDALE, NJ. *Proceedings of an international conference on genetic algorithms and their applications*. [S.l.], 1985. v. 154, p. 154–159.

GONÇALVES, J. F.; RESENDE, M. G. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, Springer, v. 17, n. 5, p. 487–525, 2011.

GONÇALVES, J. F.; RESENDE, M. G.; TOSO, R. F. An experimental comparison of biased and unbiased random-key genetic algorithms. *Pesquisa Operacional*, SciELO Brasil, v. 34, n. 2, p. 143–164, 2014.

GREFENSTETTE, J.; GOPAL, R.; ROSMAITA, B.; GUCHT, D. V. Genetic algorithms for the traveling salesman problem. In: *Proceedings of the first International Conference on Genetic Algorithms and their Applications*. [S.l.: s.n.], 1985. p. 160–168.

GREFENSTETTE, J. J. Incorporating problem specific knowledge into genetic algorithms. In: _____. *Genetic Algorithms and Simulated Annealing*. London: [s.n.], 1987. p. 42–60.

GRUEN, T.; CORSTEN, D.; BHARADWAJ, S.; AMERICA, G. M. of. *Retail Out-of-stocks: A Worldwide Examination of Extent, Causes and Consumer Responses*. Grocery Manufacturers of America, 2002. Disponível em: <https://books.google.co.uk/books?id=zxAPHwAACAAJ>.

GU, J.; GOETSCHALCKX, M.; MCGINNIS, L. F. Research on warehouse operation: A comprehensive review. *European Journal of Operational Research*, v. 177, n. 1, p. 1 – 21, 2007. ISSN 0377-2217. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0377221706001056>.

GU, J.; GOETSCHALCKX, M.; MCGINNIS, L. F. Research on warehouse design and performance evaluation: A comprehensive review. *European Journal of Operational Research*, v. 203, n. 3, p. 539 – 549, 2010. ISSN 0377-2217. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0377221709005219>.

HART, J. P.; SHOGAN, A. W. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, Elsevier, v. 6, n. 3, p. 107–114, 1987.

HAUSMAN, W. H.; SCHWARZ, L. B.; GRAVES, S. C. Optimal storage assignment in automatic warehousing systems. *Management Science*, v. 22, n. 6, p. 629–638, 1976. Disponível em: <https://doi.org/10.1287/mnsc.22.6.629>.

HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975. Second edition, 1992.

JACQUIN, S.; JOURDAN, L.; TALBI, E.-G. Dynamic programming based metaheuristic for energy planning problems. In: SPRINGER. *European Conference on the Applications of Evolutionary Computation*. [S.l.], 2014. p. 165–176.

KOSTER, R. de; LE-DUC, T.; ROODBERGEN, K. J. Design and control of warehouse order picking: A literature review. *European Journal of Operational Research*, v. 182, n. 2, p. 481 – 501, 2007. ISSN 0377-2217. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0377221706006473>.

LANGUAGES, J. . S. for P. *ISO/IEC 14882:2011*. [S.l.], 2011. Disponível em: <https://www.iso.org/standard/50372.html>.

LÓPEZ-IBÁÑEZ, M.; BLUM, C.; OHLMANN, J. W.; THOMAS, B. W. The travelling salesman problem with time windows: Adapting algorithms from travel-time to makespan optimization. *Applied Soft Computing*, Elsevier, v. 13, n. 9, p. 3806–3815, 2013.

LÓPEZ-IBÁÑEZ, M.; DUBOIS-LACOSTE, J.; CÁCERES, L. P.; BIRATTARI, M.; STÜTZLE, T. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, Elsevier, v. 3, p. 43–58, 2016.

LÓPEZ-IBÁÑEZ, M.; DUBOIS-LACOSTE, J.; STÜTZLE, T.; BIRATTARI, M. *The irace Package: Iterated Race for Automatic Algorithm Configuration*. Université Libre de Bruxelles, 2011.

MANIKAS, T.; CAIN, J. *Genetic Algorithms vs. Simulated Annealing: A Comparison of Approaches for Solving the Circuit Partitioning Problem.* [S.l.], 1996.

MARON, O.; MOORE, A. W. The racing algorithm: Model selection for lazy learners. In: *Lazy learning.* [S.l.]: Springer, 1997. p. 193–225.

MATSUMOTO, M.; NISHIMURA, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, ACM, v. 8, n. 1, p. 3–30, 1998.

MESQUITA, R. G.; SILVA, R. M.; MELLO, C. A.; MIRANDA, P. B. Parameter tuning for document image binarization using a racing algorithm. *Expert Systems with Applications*, Elsevier, v. 42, n. 5, p. 2593–2603, 2015.

MOURA, M. A. *Algoritmo Genético de Chaves Aleatórias Via Distribuição de Levy Para Otimização Global.* Dissertação (Mestrado) — Federal University of Pernambuco, Recife, Brazil, 2018.

MURALIDHARAN, B.; LINN, R. J.; PANDIT, R. Shuffling heuristics for the storage location assignment in an as/rs. *International Journal of Production Research*, Taylor & Francis, v. 33, n. 6, p. 1661–1672, 1995. Disponível em: <http://dx.doi.org/10.1080/00207549508930234>.

NAIR, T. R. G.; SOODA, K. Comparison of genetic algorithm and simulated annealing technique for optimal path selection in network routing. *CoRR*, abs/1001.3920, 2010. Disponível em: <http://arxiv.org/abs/1001.3920>.

NANNEN, V.; EIBEN, A. E. A method for parameter calibration and relevance estimation in evolutionary algorithms. In: ACM. *Proceedings of the 8th annual conference on Genetic and evolutionary computation.* [S.l.], 2006. p. 183–190.

NEMENYI, P. *Distribution-free Multiple Comparisons.* Tese (Doutorado) — Princeton University, 1963.

PAZOUR, J. A.; CARLO, H. J. Warehouse reshuffling: Insights and optimization. *Transportation Research Part E: Logistics and Transportation Review*, v. 73, p. 207 – 226, 2015. ISSN 1366-5545. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1366554514001914>.

PELLEGRINI, P.; CASTELLI, L.; PESENTI, R. Metaheuristic algorithms for the simultaneous slot allocation problem. *IET Intelligent Transport Systems*, IET, v. 6, n. 4, p. 453–462, 2012.

R Core Team. *R: A Language and Environment for Statistical Computing.* Vienna, Austria, 2015. Disponível em: <https://www.R-project.org/>.

RAJGOPAL, J. *Supply Chains: Definitions & Basic Concepts.* [S.l.], 2016.

ROODBERGEN, K. J.; VIS, I. F. A survey of literature on automated storage and retrieval systems. *European Journal of Operational Research*, v. 194, n. 2, p. 343 – 362, 2009. ISSN 0377-2217. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0377221708001598>.

SAFE, M.; CARBALLIDO, J.; PONZONI, I.; BRIGNOLE, N. On stopping criteria for genetic algorithms. In: BAZZAN, A. L. C.; LABIDI, S. (Ed.). *Advances in Artificial Intelligence – SBIA 2004*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 405–413. ISBN 978-3-540-28645-5.

SAMA, M.; PELLEGRINI, P.; D'ARIANO, A.; RODRIGUEZ, J.; PACCIARELLI, D. Ant colony optimization for the real-time train routing selection problem. *Transportation Research Part B: Methodological*, Elsevier, v. 85, p. 89–108, 2016.

SPEARS, W. M.; JONG, K. D. D. *On the virtues of parameterized uniform crossover*. [S.l.], 1995.

STEFANELLO, F.; AGGARWAL, V.; BURIOL, L. S.; GONÇALVES, J. F.; RESENDE, M. G. A biased random-key genetic algorithm for placement of virtual machines across geo-separated data centers. In: ACM. *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. [S.l.], 2015. p. 919–926.

STUDENT. The probable error of a mean. *Biometrika*, v. 6, n. 1, p. 1–25, 1908. Disponível em: <http://dx.doi.org/10.1093/biomet/6.1.1>.

TOSO, R. F.; RESENDE, M. G. A c++ application programming interface for biased random-key genetic algorithms. *Optimization Methods and Software*, Taylor & Francis, v. 30, n. 1, p. 81–93, 2015.

TREBILCOCK, B. *Resolve to Reslot Your Warehouse, Modern Materials Handling*. 2011. <http://www.mmh.com/issue_archive/2011/mmh_11_05.pdf>. Accessed: 2017-09-30.

WILHELM, M. R.; WARD, T. L. Solving quadratic assignment problems by 'simulated annealing'. *Iie Transactions*, v. 19, p. 107–119, 03 1987.

YARIMCAM, A.; ASTA, S.; ÖZCAN, E.; PARKES, A. J. Heuristic generation via parameter tuning for online bin packing. In: IEEE. *Evolving and Autonomous Learning Systems (EALS), 2014 IEEE Symposium on*. [S.l.], 2014. p. 102–108.

ZIELINSKI, K.; PETERS-DROLSHAGEN, D.; LAUR, R. Stopping criteria for single-objective optimization. 01 2005.

APPENDIX A – HEURISTICS

A.1 H3 HEURISTIC

Listing A.1 – GiraldoH3.h

```
/**
 * @file GiraldoH3.h
 * @version 1.0
 * @author Leonardo Bueno
 * @date May 9, 2018
 *
 *************************************************************************
 *
 * @brief: Declarations for Heuristic H3 from German Giraldo's article
 * "Toward perpetually organized unit-load warehouses", 2012
 *
 *************************************************************************
 * @section Revisions:
 *
 * Revision: 1.0   May 9, 2018    Leonardo Bueno
 * * Original version based on German Giraldo's article:
 *      "Toward perpetually organized unit-load warehouses" 2012
 *
 *************************************************************************/

#ifndef GiraldoH3_H
#define GiraldoH3_H

#include <list>
#include <vector>
#include <algorithm>
#include "ReshuffleScenarioParser.h"

#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

class GiraldoH3
{
private:
    std::vector<int> OIo;         // the initial location of item k
    std::vector<std::vector<double>> dij;    // distance to travel from location i
        to j.
    std::vector<std::vector<double>> gij;    // distance to travel from location i
        to j.
    std::vector<int> Ik;          // the initial location of item k
    std::vector<int> Fk;          // the final location of item k;
    std::vector<int> Ii;          // Initial items in each location i
    std::vector<int> Fi;          // Final items in each location i
    int startPos;                 // Starting position of S/R Machine

public:
    GiraldoH3(const ReshuffleScenarioParser &scenario);   // Constructor
```

```
    // prints shortest path from s
    double bestPath(bool print);

    void printMovement(const int initialLoc, const int finalLoc,
            const int element, const std::vector<int> &trackItem,
            const double moveCost, const std::vector<int> &items) const;

    void printIntVectorSequence(const std::vector<int> &vec)  const;
};

#endif /* GiraldoH3_H */
```

Listing A.2 – GiraldoH3.cpp

```
/**
 * @file GiraldoH3.cpp
 * @version 1.0
 * @author Leonardo Bueno
 * @date May 9, 2018
 *
 ***********************************************************************
 *
 * @brief: Implements Heuristic H3 from German Giraldo's article
 * "Toward perpetually organized unit-load warehouses", 2012
 *
 ***********************************************************************
 * @section Revisions:
 *
 * Revision: 1.0    May 9, 2018     Leonardo Bueno
 * * Original version based on German Giraldo's article:
 *       "Toward perpetually organized unit-load warehouses" 2012
 *
 ***********************************************************************/

#include<bits/stdc++.h>
#include "GiraldoH3.h"

using namespace std;
# define INF 0x3f3f3f3f

GiraldoH3::GiraldoH3(const ReshuffleScenarioParser &scenario)
{
    this->OIo = scenario.getOIo();
    this->Ik = scenario.getIk();
    this->Fk = scenario.getFk();
    this->Ii = scenario.getIi();
    this->Fi = scenario.getFi();
    this->gij = scenario.getGij();
    this->dij = scenario.getDij();
    this->startPos = scenario.getStartPos();
}

// Prints shortest paths from src to all other vertices
double GiraldoH3::bestPath(bool printKeyFlag)
{
    const unsigned int kmax = Ik.size();
    int currentLoc = startPos;
    int auxItem;
    int moveLoc = -1;
    int emptyIdx = -1;
    unsigned int o; // Index for open locations
    unsigned int k; // Index for items
    double totalCost = 0;
    double minCostToEmpty = INF;
    double minCostToSR = INF;

    std::vector<int> emptyLoc(OIo);          // Tracks empty locations
    std::vector<int> currentPos(Ik);         // Track item locations
    std::vector<int> TrackIi(Ii);            // Track storage modifications
```

```

    while(currentPos != Fk)
    {
        minCostToEmpty = INF;
        minCostToSR = INF;
        moveLoc = -1;
        emptyIdx = -1;

        // identify the item (q) stored closest to the S/R machine
        // current position whose ending position is currently open.
        for(o = 0; o < emptyLoc.size(); o++)
        {
            if((Fi[emptyLoc[o]] >= 0) && (currentLoc != currentPos[Fi[emptyLoc[o]]]))
            {
                if((currentLoc >= 0))
                {
                    if((dij[currentLoc][currentPos[Fi[emptyLoc[o]]]] < minCostToSR))
                    {
                        moveLoc = currentPos[Fi[emptyLoc[o]]];
                        minCostToSR = dij[currentLoc][moveLoc];
                        emptyIdx = o;
                    }
                }
                else
                {
                    moveLoc = currentPos[Fi[emptyLoc[o]]];
                    emptyIdx = o;
                }
            }
        }

        // Perform movement
        if(moveLoc < 0)
        {
            minCostToSR = INF;
            minCostToEmpty = INF;
            moveLoc = -1;
            emptyIdx = -1;

            // identify the item (q) stored closest to the S/R machine
            // current position whose ending position is currently occupied
            // and move it to the open position closest to its final position
            for(o = 0; o < emptyLoc.size(); ++o)
            {
                for (k = 0; k < kmax; ++k)
                {
                    // Item using this position is not in final position
                    if( (currentPos[k] != Fk[k]))
                    {
                        // Has minimum moving cost to empty position under threshold
                        if(gij[currentPos[k]][emptyLoc[o]] < minCostToEmpty)
                        {
                            moveLoc = currentPos[k];
                            emptyIdx = o;
```

```
                                minCostToEmpty = gij[moveLoc][emptyLoc[o]];

                                if((currentLoc >= 0))
                                {
                                    minCostToSR = dij[currentLoc][moveLoc];
                                }
                            }
                            else if((gij[currentPos[k]][emptyLoc[o]] ==
                                minCostToEmpty) &&
                                    (currentLoc >= 0) &&
                                    (dij[currentLoc][currentPos[k]] < minCostToSR))
                            {
                                moveLoc = currentPos[k];
                                minCostToSR = dij[currentLoc][moveLoc];
                            }
                        }
                    }
                }
            }

            if(moveLoc >= 0)
            {
                if((currentLoc != moveLoc) && (currentLoc >= 0))
                {
                    // Move vehicle from initial position to the position of the
                        element to be moved
                    totalCost += dij[currentLoc][moveLoc];

                    if(printKeyFlag)
                    {
                        printMovement(currentLoc, moveLoc, -1, TrackIi, totalCost,
                            emptyLoc);
                    }
                }

                // Now the initial position is the position of the element
                // to be moved and the final position is the empty position
                currentLoc = moveLoc;
                moveLoc = emptyLoc[emptyIdx];

                // Move item to it's final position
                totalCost += gij[currentLoc][moveLoc];

                currentPos[TrackIi[currentLoc]] = emptyLoc[emptyIdx];

                // Swap item on each location
                auxItem = TrackIi[moveLoc];
                TrackIi[moveLoc] = TrackIi[currentLoc];
                TrackIi[currentLoc] = auxItem;

                emptyLoc[emptyIdx] = currentLoc;

                if(printKeyFlag)
                {
                    printMovement(currentLoc, currentPos[TrackIi[moveLoc]],
                            TrackIi[moveLoc], TrackIi, totalCost, emptyLoc);
                }
```

```

            // Now vehicle is in the final position
            currentLoc = moveLoc;
        }
    }

    return totalCost;
}

void GiraldoH3::printMovement(const int initialLoc, const int finalLoc,
        const int element, const std::vector<int> &trackItem,
        const double moveCost, const std::vector<int> &items) const
{
    std::cout << initialLoc << "\t" << finalLoc << "\t";
    if(element == -1){
        std::cout << "none\t\t"<< moveCost << "\t\t";
    }
    else{
        std::cout << element <<"\t\t"<< moveCost << "\t\t";
    }

    printIntVectorSequence(trackItem);

    std::cout << "\t";

    printIntVectorSequence(items);

    std::cout << std::endl;
}

void GiraldoH3::printIntVectorSequence(const std::vector<int> &vec)  const
{
    for (std::vector<int>::const_iterator i = vec.begin(); i != vec.end(); ++i)
        std::cout << *i << ' ';
}
```

Listing A.3 – mainGiraldoH3.cpp

```
/**
 * @file mainGiraldoH3.cpp
 * @version 1.0
 * @author Leonardo Bueno
 * @date 2018
 *
 ***********************************************************************
 *
 * @brief: Main file for executing German Giraldo's
 *  Reshuffling Heuristic 3 (H3)
 *
 * This code treats the following parameters
 * OBS: Parameters should be in this order
 *
 * FILE.csv printBool tau
 *
 * Where:
 * - FILE.csv   - The reshuffle scenario
 * - printBool  - True prints final solution, false prints only the final cost
 *
 ***********************************************************************
 * @section Revisions:
 *
 * Revision: 1.0   2018    Leonardo Bueno
 * * Original version based on German Giraldo's article:
 *       "Toward perpetually organized unit-load warehouses" 2012
 *
 ************************************************************************/

#include <iostream>
#include "ReshuffleScenarioParser.h"
#include "GiraldoH3.h"
#include <climits>
#include <time.h>
#include <math.h>

#define DEFAULT_SCENARIO_FILE   ((char*)"scenarios\\scenario_Imax100Util50Org0.csv")
#define DEFAULT_PRINT_KEY   false

int main(int argc, char* argv[]) {
    char* scenarioFilePtr = DEFAULT_SCENARIO_FILE;
    bool printKey = DEFAULT_PRINT_KEY;

    std::cout << "\r\nGiven parameters: ";
    for(int argCount = 0; argCount < argc; argCount++)
    {
        std::cout << argv[argCount] << " ";
    }
    std::cout << "\r\n";

    if(argc >= 2)
    {
        scenarioFilePtr = argv[1];
    }
```

```

    char *findCSV = NULL;
    findCSV = strstr (scenarioFilePtr,".csv");

    if(!findCSV)
    {
        // Tell the user how to run the program
        std::cerr << "Usage: " << argv[0] << " FILE.csv true (printBool)" << std::endl;
        /* "Usage messages" are a conventional way of telling the user
         * how to run a program if they enter the command incorrectly.
         */
        return 1;
    }

    if(argc >= 3)
    {
        std::stringstream ss(argv[2]);

        if(!(ss >> std::boolalpha >> printKey))
        {
            // Tell the user how to run the program
            std::cerr << "Usage: " << argv[0] << " FILE.csv true (printBool)" << std::endl;
            /* "Usage messages" are a conventional way of telling the user
             * how to run a program if they enter the command incorrectly.
             */
            return 1;
        }
    }

    std::string filePath(scenarioFilePtr);
    ReshuffleScenarioParser scenario(filePath, printKey);

    // H3
    clock_t start = clock();

    GiraldoH3 H3_cs1(scenario);
    double h3Results = H3_cs1.bestPath(printKey);

    unsigned long int h3_milliseconds_since_start = (( clock() - start ) * 1000) / CLOCKS_PER_SEC;
    std::cout << std::endl << "H3:\t" << h3Results;
    std::cout << "\tRuntime = "<<h3_milliseconds_since_start<< "ms" << std::endl;
    std::cout << std::endl << h3Results;

    // Output results
    std::ofstream outfile("reshuffleResultsH3.csv", ios::out | ios::app);

    if (outfile.is_open())
    {
        outfile << argv[1];
        outfile << "," << h3Results << "," << h3_milliseconds_since_start; // H3 Results

        outfile << "\n";
        outfile.close();
```

```
    }

    return 0;
}
```

A.2 GRH HEURISTIC

Listing A.4 – PazourGRH.h

```
/**
 * @file PazourGRH.cpp
 * @version 1.0
 * @author Leonardo Bueno
 * @date May 10, 2018
 *
 ************************************************************************
 *
 * @brief: Declarations for Heuristic GRH from Jennifer Pazour's article
 *  "Warehouse reshuffling: Insights and optimization", 2015
 *
 ************************************************************************
 * @section Revisions:
 *
 * Revision: 1.0    May 10, 2018    Leonardo Bueno
 * * Original version based on Jennifer Pazour's article:
 *      "Warehouse reshuffling: Insights and optimization" 2015
 *
 ************************************************************************/

#ifndef PazourGRH_H
#define PazourGRH_H

#include <list>
#include <vector>
#include <algorithm>
#include "ReshuffleScenarioParser.h"

#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

class PazourGRH
{
private:
    std::vector<int> OIo;          // the initial location of item k
    std::vector<std::vector<double>> dij;    // distance to travel from location i to j.
    std::vector<std::vector<double>> gij;    // distance to travel from location i to j.
    std::vector<std::vector<int>> Cc;        // Cc - set of items that belong to cycle c, indexed on c.
    std::vector<int> Ik;           // the initial location of item k
    std::vector<int> Fk;           // the final location of item k;
    std::vector<int> Ii;           // Initial items in each location
    std::vector<int> Fi;           // Final items in each location
    int startPos;

public:
    PazourGRH(const ReshuffleScenarioParser &scenario);  // Constructor

    // prints shortest path from s
```

```
    double bestPath(double closeDistanceThreshold, bool printKeyFlag);

    void printMovement(const int initialLoc, const int finalLoc, const int
        element, const std::vector<int> &trackItem, const double moveCost, const
        std::vector<int> &items) const;

    void printIntVectorSequence(const std::vector<int> &vec)  const;
};

#endif /* PazourGRH_H */
```

Listing A.5 – PazourGRH.cpp

```
/**
 * @file PazourGRH.cpp
 * @version 1.0
 * @author Leonardo Bueno
 * @date May 10, 2018
 *
 ************************************************************************
 *
 * @brief: Implements Heuristic GRH from Jennifer Pazour article
 *  "Warehouse reshuffling: Insights and optimization", 2015
 *
 ************************************************************************
 * @section Revisions:
 *
 * Revision: 1.0    May 10, 2018     Leonardo Bueno
 * * Original version based on Jennifer Pazour article:
 *       "Warehouse reshuffling: Insights and optimization" 2015
 *
 *************************************************************************/

#include<bits/stdc++.h>
#include "PazourGRH.h"

using namespace std;
#define INF 0x3f3f3f3f

PazourGRH::PazourGRH(const ReshuffleScenarioParser &scenario)
{
    this->OIo = scenario.getOIo();
    this->Cc = scenario.getCc();
    this->Ik = scenario.getIk();
    this->Fk = scenario.getFk();
    this->Ii = scenario.getIi();
    this->Fi = scenario.getFi();
    this->gij = scenario.getGij();
    this->dij = scenario.getDij();
    this->startPos = scenario.getStartPos();
}

double PazourGRH::bestPath(double closeDistanceThreshold, bool printKeyFlag)
{
    const unsigned int kmax = Ik.size();
    int currentLoc = startPos;
    int auxItem;
    int moveLoc = -1;
    int emptyIdx = -1;
    int moveCycle = -1;
    unsigned int o; // Index for open locations
    unsigned int c; // Index for cycle in Cc
    unsigned int k; // Index for items
    double totalCost = 0;
    double minCostToEmpty = INF;
    double minCostToSR = INF;

    std::vector<int> emptyLoc(OIo);                // Tracks empty locations
```

```
    std::vector<int> currentPos(Ik);              // Track item locations
    std::vector<int> TrackIi(Ii);                 // Track storage modifications
    std::vector<std::vector<int>> TrackCc(Cc);    // Track cycle breaks

    while(currentPos != Fk)
    {
        minCostToEmpty = closeDistanceThreshold;
        minCostToSR = INF;
        moveLoc = −1;
        emptyIdx = −1;
        moveCycle = −1;

        // identify the item (q) stored closest to the S/R machine
        // current position that is either part of a cycle that can be
        // broken with less than tau distance units (i.e., travel distance
        // from starting location of q to an open location <= tau)
        // OR whose ending position is currently open.
        for(o = 0; o < emptyLoc.size(); o++)
        {
            // (Break nearby cycle) Move item q
            // (for which travel distance from starting location of q to an open
                location 6 s)
            // and remove the cycle from the list of all cycles.
            // Find item with minimum cost to move to empty location
            for(c=0; c < TrackCc.size(); c++)
            {
                for(k=0; k<TrackCc[c].size(); k++)
                {
                    // Item has moving cost to empty lower than minimum found so
                        far
                    // And item is not in final position
                    if((gij[currentPos[TrackCc[c][k]]][emptyLoc[o]] <=
                        minCostToEmpty) &&
                            (currentPos[TrackCc[c][k]] != Fk[TrackCc[c][k]]))
                    {
                        if((currentLoc >= 0) &&
                                (dij[currentLoc][currentPos[TrackCc[c][k]]] <
                                    minCostToSR))
                        {
                            moveLoc = currentPos[TrackCc[c][k]];
                            emptyIdx = o;
                            moveCycle = c;
                            minCostToSR = dij[currentLoc][moveLoc];
                        }
                        else if(moveLoc < 0)
                        {
                            moveLoc = currentPos[TrackCc[c][k]];
                            emptyIdx = o;
                            moveCycle = c;
                        }
                    }
                }
            }

            // identify the item (q) stored closest to the S/R machine
            // current position whose ending position is currently open.
            if((Fi[emptyLoc[o]] >= 0) && (currentLoc != currentPos[Fi[emptyLoc[o
```

```
            ]]]))
        {
            if((currentLoc >= 0))
            {
                if((dij[currentLoc][currentPos[Fi[emptyLoc[o]]]] < minCostToSR))
                {
                    moveLoc = currentPos[Fi[emptyLoc[o]]];
                    emptyIdx = o;
                    minCostToSR = dij[currentLoc][moveLoc];
                }
            }
            else
            {
                moveLoc = currentPos[Fi[emptyLoc[o]]];
                emptyIdx = o;
            }
        }
    }

    // (Break cycle far away) Move the item closest to the S/R
    // (which requires repositioning) to the closest open location
    if(moveLoc < 0)
    {
        minCostToSR = INF;
        minCostToEmpty = INF;
        moveLoc = -1;
        emptyIdx = -1;

        for(o = 0; o < emptyLoc.size(); ++o)
        {
            for (k = 0; k < kmax; ++k)
            {
                // Item using this position is not in final position
                if( (currentPos[k] != Fk[k]))
                {
                    // Has minimum moving cost to empty position under threshold
                    if(gij[currentPos[k]][emptyLoc[o]] < minCostToEmpty)
                    {
                        moveLoc = currentPos[k];
                        emptyIdx = o;
                        minCostToEmpty = gij[moveLoc][emptyLoc[o]];

                        if((currentLoc >= 0))
                        {
                            minCostToSR = dij[currentLoc][moveLoc];
                        }
                    }
                    else if((gij[currentPos[k]][emptyLoc[o]] == minCostToEmpty) &&
                            (currentLoc >= 0) &&
                            (dij[currentLoc][currentPos[k]] < minCostToSR))
                    {
                        moveLoc = currentPos[k];
                        minCostToSR = dij[currentLoc][moveLoc];
                    }
```

```
                        }
                    }
                }
            }

            // Perform movement
            if(moveLoc >= 0)
            {
                if((currentLoc != moveLoc) && (currentLoc >= 0))
                {
                    // Move vehicle from initial position to the position of the
                        element to be moved
                    totalCost += dij[currentLoc][moveLoc];

                    if(printKeyFlag)
                    {
                        printMovement(currentLoc, moveLoc, -1, TrackIi, totalCost,
                            emptyLoc);
                    }
                }

                // Now the initial position is the position of the element to be
                    moved
                // and the final position is the empty position
                currentLoc = moveLoc;
                moveLoc = emptyLoc[emptyIdx];

                // Move item to it's final position
                totalCost += gij[currentLoc][moveLoc];

                currentPos[TrackIi[currentLoc]] = emptyLoc[emptyIdx];

                // Swap item on each location
                auxItem = TrackIi[moveLoc];
                TrackIi[moveLoc] = TrackIi[currentLoc];
                TrackIi[currentLoc] = auxItem;

                emptyLoc[emptyIdx] = currentLoc;

                if(printKeyFlag)
                {
                    printMovement(currentLoc, currentPos[TrackIi[moveLoc]],
                            TrackIi[moveLoc], TrackIi, totalCost, emptyLoc);
                }

                // Now vehicle is in the final position
                currentLoc = moveLoc;

                if(moveCycle >= 0)
                {
                    TrackCc.erase(TrackCc.begin() + moveCycle);
                }
            }
        }

    return totalCost;
}
```

```cpp

void PazourGRH::printMovement(const int initialLoc, const int finalLoc, const int element,
        const std::vector<int> &trackItem, const double moveCost,
        const std::vector<int> &items) const
{
    std::cout << initialLoc << "\t" << finalLoc << "\t";
    if(element == -1){
        std::cout << "none\t\t"<< moveCost << "\t\t";
    }
    else{
        std::cout << element <<"\t\t"<< moveCost << "\t\t";
    }

    printIntVectorSequence(trackItem);

    std::cout << "\t";

    printIntVectorSequence(items);

    std::cout << std::endl;
}

void PazourGRH::printIntVectorSequence(const std::vector<int> &vec)  const
{
    for (std::vector<int>::const_iterator i = vec.begin(); i != vec.end(); ++i)
        std::cout << *i << ' ';
}
```

Listing A.6 – mainPazourGRH.cpp

```
/**
 * @file mainPazourGRH.cpp
 * @version 1.0
 * @author Leonardo Bueno
 * @date 2018
 *
 ***********************************************************************
 *
 * @brief: Main file for executing Jennifer Pazour's
 *  General Reshuffling Heuristic (GRH)
 *
 * This code treats the following parameters
 * OBS: Parameters should be in this order
 *
 * FILE.csv printBool tau
 *
 * Where:
 * - FILE.csv    - The reshuffle scenario
 * - printBool   - True prints final solution, false prints only the final cost
 * - tau         - Distance to break nearby cycles - Default 0
 *
 ***********************************************************************
 * @section Revisions:
 *
 * Revision: 1.0    2018     Leonardo Bueno
 * * Original version based on Jennifer Pazour's article:
 *       "Warehouse reshuffling: Insights and optimization" 2015
 *
 ***********************************************************************/

#include <iostream>
#include "ReshuffleScenarioParser.h"
#include "PazourGRH.h"
#include <climits>
#include <time.h>
#include <math.h>
using namespace std;

#define DEFAULT_SCENARIO_FILE   ((char*)"\\scenarios\\scenario_Imax100Util50Org0.
    csv")
#define DEFAULT_GRH_TAU 0.0
#define DEFAULT_PRINT_KEY   false

int main(int argc, char* argv[]) {
    char* scenarioFilePtr = DEFAULT_SCENARIO_FILE;
    double grh_Tau = DEFAULT_GRH_TAU;
    bool printKey = DEFAULT_PRINT_KEY;

    std::cout << "\r\nGiven parameters: ";
    for(int argCount = 0; argCount < argc; argCount++)
    {
        std::cout << argv[argCount] << " ";
    }
    std::cout << "\r\n";

```

```
    if(argc >= 2)
    {
        scenarioFilePtr = argv[1];
    }

    char *findCSV = NULL;
    findCSV = strstr (scenarioFilePtr,".csv");

    if (!findCSV)
    {
        // Tell the user how to run the program
        std::cerr << "Usage: " << argv[0] << " FILE.csv true (printBool) 30.0 (Tau) " << std::endl;
        /* "Usage messages" are a conventional way of telling the user
         * how to run a program if they enter the command incorrectly.
         */
        return 1;
    }

    if(argc >= 3)
    {
        std::stringstream ss(argv[2]);

        if(!(ss >> std::boolalpha >> printKey))
        {
            // Tell the user how to run the program
            std::cerr << "Usage: " << argv[0] << " FILE.csv true (printBool) 30.0 (Tau) " << std::endl;
            /* "Usage messages" are a conventional way of telling the user
             * how to run a program if they enter the command incorrectly.
             */
            return 1;
        }
    }

    if(argc >= 4)
    {
        grh_Tau = atof(argv[3]);
    }

    std::string filePath(scenarioFilePtr);
    ReshuffleScenarioParser scenario(filePath, printKey);

    // GRH
    clock_t start = clock();

    PazourGRH GRH_cs1(scenario);
    double grhResults = GRH_cs1.bestPath(grh_Tau, printKey);

    unsigned long int grh_milliseconds_since_start = (( clock() - start ) * 1000) / CLOCKS_PER_SEC;
    std::cout << std::endl << "GRH:\t" << grhResults;
    std::cout << "\tRuntime = "<<grh_milliseconds_since_start<< "ms" << std::endl;
    std::cout << std::endl << grhResults;

    // Output results
```

```
    std::ofstream outfile("reshuffleResultsGRH.csv", ios::out | ios::app);

    if (outfile.is_open())
    {
        outfile << argv[1];
        outfile << "," << grhResults << "," << grh_milliseconds_since_start << ",
            " << grh_Tau; // GRH Results

        outfile << "\n";
        outfile.close();
    }

    return 0;
}
```

A.3 BRKGA RESHUFFLE DECODER

Listing A.7 – ReshuffleDecoder.h

```
/**
 * @file reshuffleDecoder.h
 * @version 1.0
 * @author Leonardo Bueno
 * @date 2018
 *
 ***********************************************************************
 *
 * @brief: Declaration for Reshuffle BRKGA Decoder
 *
 ***********************************************************************
 * @section Revisions:
 *
 * Revision: 1.0   2018    Leonardo Bueno
 * * Original version based on BRKGA C++ API Sample Code:
 * A C++ APPLICATION PROGRAMMING INTERFACE FOR
 * BIASED RANDOM-KEY GENETIC ALGORITHMS
 * RODRIGO F. TOSO AND MAURICIO G.C. RESENDE, 2011
 *
 ***********************************************************************/
#ifndef RESHUFFLEDECODER_H
#define RESHUFFLEDECODER_H

#include <list>
#include <vector>
#include <algorithm>
#include "ReshuffleScenarioParser.h"

class ReshuffleDecoder {
public:
    ReshuffleDecoder(const ReshuffleScenarioParser &scenario);
    ~ReshuffleDecoder();

    double decode(const std::vector< double >& chromosome) const;
    double decode(const std::vector< double >& chromosome, bool printKeyFlag)
        const;
    int chromosomeSize(void) const;
    void printMovement(const int initialLoc, const int finalLoc, const int
        element,
            const std::vector<int> &trackItem, const double moveCost,
            const std::vector<int> &items) const;
    void printIntVectorSequence(const std::vector<int> &vec)  const;
    void printKey(const std::vector< double >& chromosome) const;

private:
    std::vector<std::vector<double>> gij;   // distance to travel from location i
         to j.
    std::vector<std::vector<double>> dij;   // distance to travel unloaded from
        location i to j.
    std::vector<std::vector<int>> Cc;       // Cc - set of items that belong to
        cycle c, indexed on c.
    std::vector<int> Ik;        // the initial location of item k
```

```
    std::vector<int> Fk;          // the final location of item k
    std::vector<int> Ii;          // Initial items in each location
    std::vector<int> Fi;          // Final items in each location
    std::vector<int> OIo;         // the open locations
    double gmax;
    int startLoc;
};

#endif /* RESHUFFLEDECODER_H */
```

Listing A.8 – ReshuffleDecoder.cpp

```
/**
 * @file reshuffleDecoder.cpp
 * @version 1.0
 * @author Leonardo Bueno
 * @date 2018
 *
 ************************************************************************
 *
 * @brief: Methods for Reshuffle BRKGA Decoder
 *
 ************************************************************************
 * @section Revisions:
 *
 * Revision: 1.0    2018     Leonardo Bueno
 * * Original version based on BRKGA C++ API Sample Code:
 * A C++ APPLICATION PROGRAMMING INTERFACE FOR
 * BIASED RANDOM-KEY GENETIC ALGORITHMS
 * RODRIGO F. TOSO AND MAURICIO G.C. RESENDE, 2011
 *
 ************************************************************************/

#include "ReshuffleDecoder.h"
#include <vector>
#include <tuple>
#include <string>
#include <set>
#include <cmath>
#include <climits>
#include <fstream>
#include <iostream>
#include <sstream>
using namespace std;


#define CALC_MOVES_IN_CYCLE(kmax, cmax, imax)    (kmax+cmax)
# define INF 0x3f3f3f3f

ReshuffleDecoder::ReshuffleDecoder(const ReshuffleScenarioParser &scenario)
{
    this->OIo = scenario.getOIo();
    this->Cc = scenario.getCc();
    this->Ik = scenario.getIk();
    this->Fk = scenario.getFk();
    this->Ii = scenario.getIi();
    this->Fi = scenario.getFi();
    this->gij = scenario.getGij();
    this->dij = scenario.getDij();
    this->gmax = scenario.getGmax();
    this->startLoc = scenario.getStartPos();
}

ReshuffleDecoder::~ReshuffleDecoder(void)
{
}

```

```
int ReshuffleDecoder::chromosomeSize(void) const
{
    return CALC_MOVES_IN_CYCLE(Ik.size(), Cc.size(), Ii.size());
}

double ReshuffleDecoder::decode(const std::vector< double >& chromosome) const
{
    return decode(chromosome, false);
}

double ReshuffleDecoder::decode(const std::vector< double >& chromosome, bool printKeyFlag) const
{
    const unsigned int kmax = Ik.size();
    int currentLoc = this->startLoc;
    int auxItem;
    int moveLoc = -1;
    int emptyIdx = -1;
    int moveCycle = -1;
    int allele;      // Index for chromosome alleles
    unsigned int o; // Index for open locations
    unsigned int c; // Index for cycle in Cc
    unsigned int k; // Index for itens
    double totalCost = 0;
    double minCostToEmpty = INF;
    double minCostToSR = INF;

    if(printKeyFlag){
        std::cout << "From\tTo\tItem Carried\tMoveCost\tPositions" << std::endl;
    }

    std::vector<int> emptyLoc(OIo);                // Tracks empty locations
    std::vector<int> currentPos(Ik);               // Track item locations
    std::vector<int> TrackIi(Ii);                  // Track storage modifications
    std::vector<std::vector<int>> TrackCc(Cc);     // Track cycle breaks

    for(allele = 0; (currentPos != Fk) && (allele < chromosomeSize()); allele++)
    {
        // Tau is now calculated using gmax and the current allele
        minCostToEmpty = this->gmax * chromosome[allele];
        minCostToSR = INF;
        moveLoc = -1;
        emptyIdx = -1;
        moveCycle = -1;

        // identify the item (q) stored closest to the S/R machine
        // current position that is either part of a cycle that can be
        // broken with less than tau distance units (i.e., travel distance
        // from starting location of q to an open location <= tau)
        // OR whose ending position is currently open.
        for(o = 0; o < emptyLoc.size(); o++)
        {
            // (Break nearby cycle) Move item q
            // (for which travel distance from starting location of q to an open location 6 s)
            // and remove the cycle from the list of all cycles.
            // Find item with minimum cost to move to empty location
```

```
        for(c=0; c < TrackCc.size(); c++)
        {
            for(k=0; k<TrackCc[c].size(); k++)
            {
                // Item has moving cost to empty lower than minimum found so far
                // And item is not in final position
                if((gij[currentPos[TrackCc[c][k]]][emptyLoc[o]] <= minCostToEmpty) &&
                         (currentPos[TrackCc[c][k]] != Fk[TrackCc[c][k]]))
                {
                    if((currentLoc >= 0) && (dij[currentLoc][currentPos[TrackCc[c][k]]] < minCostToSR))
                    {
                        moveLoc = currentPos[TrackCc[c][k]];
                        emptyIdx = o;
                        moveCycle = c;
                        minCostToSR = dij[currentLoc][moveLoc];
                    }
                    else if(moveLoc < 0)
                    {
                        moveLoc = currentPos[TrackCc[c][k]];
                        emptyIdx = o;
                        moveCycle = c;
                    }
                }
            }
        }

        // identify the item (q) stored closest to the S/R machine
        // current position whose ending position is currently open.
        if((Fi[emptyLoc[o]] >= 0) && (currentLoc != currentPos[Fi[emptyLoc[o]]]))
        {
            if((currentLoc >= 0))
            {
                if((dij[currentLoc][currentPos[Fi[emptyLoc[o]]]] < minCostToSR))
                {
                    moveLoc = currentPos[Fi[emptyLoc[o]]];
                    emptyIdx = o;
                    minCostToSR = dij[currentLoc][moveLoc];
                }
            }
            else
            {
                moveLoc = currentPos[Fi[emptyLoc[o]]];
                emptyIdx = o;
            }
        }
    }

    // (Break cycle far away) Move the item closest to the S/R
    // (which requires repositioning) to the closest open location
    if(moveLoc < 0)
    {
        minCostToSR = INF;
```

```
            minCostToEmpty = INF;
            moveLoc = -1;
            emptyIdx = -1;

            for(o = 0; o < emptyLoc.size(); ++o)
            {
                for (k = 0; k < kmax; ++k)
                {
                    // Item using this position is not in final position
                    if( (currentPos[k] != Fk[k]))
                    {
                        // Has minimum moving cost to empty position under
                            threshold
                        if(gij[currentPos[k]][emptyLoc[o]] < minCostToEmpty)
                        {
                            moveLoc = currentPos[k];
                            emptyIdx = o;
                            minCostToEmpty = gij[moveLoc][emptyLoc[o]];

                            if((currentLoc >= 0))
                            {
                                minCostToSR = dij[currentLoc][moveLoc];
                            }
                        }
                        else if((gij[currentPos[k]][emptyLoc[o]] ==
                            minCostToEmpty) &&
                                (currentLoc >= 0) &&
                                (dij[currentLoc][currentPos[k]] < minCostToSR))
                        {
                            moveLoc = currentPos[k];
                            minCostToSR = dij[currentLoc][moveLoc];
                        }
                    }
                }
            }
        }

        // Perform movement
        if(moveLoc >= 0)
        {
            if((currentLoc != moveLoc) && (currentLoc >= 0))
            {
                // Move vehicle from initial position to the position of the
                    element to be moved
                totalCost += dij[currentLoc][moveLoc];

                if(printKeyFlag)
                {
                    printMovement(currentLoc, moveLoc, -1, TrackIi, totalCost,
                        emptyLoc);
                }
            }

            // Now the initial position is the position of the element to be
                moved
            // and the final position is the empty position
            currentLoc = moveLoc;
```

```
                moveLoc = emptyLoc[emptyIdx];

                // Move item to it's final position
                totalCost += gij[currentLoc][moveLoc];

                currentPos[TrackIi[currentLoc]] = emptyLoc[emptyIdx];

                // Swap item on each location
                auxItem = TrackIi[moveLoc];
                TrackIi[moveLoc] = TrackIi[currentLoc];
                TrackIi[currentLoc] = auxItem;

                emptyLoc[emptyIdx] = currentLoc;

                if(printKeyFlag)
                {
                    printMovement(currentLoc, currentPos[TrackIi[moveLoc]],
                            TrackIi[moveLoc], TrackIi, totalCost, emptyLoc);
                }

                // Now vehicle is in the final position
                currentLoc = moveLoc;

                if(moveCycle >= 0)
                {
                    TrackCc.erase(TrackCc.begin() + moveCycle);
                }
            }
        }

    return totalCost;
}

void ReshuffleDecoder::printMovement(const int initialLoc, const int finalLoc,
    const int element,
        const std::vector<int> &trackItem, const double moveCost,
        const std::vector<int> &items) const
{
    std::cout << initialLoc << "\t" << finalLoc << "\t";
    if(element == -1){
        std::cout << "none\t\t"<< moveCost << "\t\t";
    }
    else{
        std::cout << element <<"\t\t"<< moveCost << "\t\t";
    }

    printIntVectorSequence(trackItem);

    std::cout << "\t";

    printIntVectorSequence(items);

    std::cout << std::endl;
}

void ReshuffleDecoder::printIntVectorSequence(const std::vector<int> &vec)  const
{
```

```
    for (std::vector<int>::const_iterator i = vec.begin(); i != vec.end(); ++i)
        std::cout << *i << ' ';
}

void ReshuffleDecoder::printKey(const std::vector< double >& chromosome) const
{
    decode(chromosome, true);
}
```

Listing A.9 – mainReshuffleBRKGA.cpp

```
/**
 * @file mainReshuffleBRKGA.cpp
 * @version 1.0
 * @author Leonardo Bueno
 * @date 2018
 *
 ***********************************************************************
 *
 * @brief: Main file for executing Reshuffle BRKGA
 *
 * This code treats the following parameters
 * OBS: Parameters should be in this order
 *
 * FILE.csv printBool seed P pe pm rhoe k maxgen X_NUMBER X_INTVL distP
 *
 * Where:
 * - FILE.csv   - The reshuffle scenario
 * - printBool  - True prints final solution, false prints only the final cost
 * - seed       - Long unsigned used as seed for the random generator
 * - P          - Size of population - Default 78
 * - pe         - Elite fraction of the population - Default 0.1625
 * - pm         - Mutant fraction of the population - Default 0.2631
 * - rhoe       - Probability of inheriting allele from elite - Default 0.3122
 * - k          - Number of independent populations - Default 4
 * - maxgen     - Maximum number of generations - Default 3000
 * - X_NUMBER   - Number of exchanged top individuals - Default 2
 * - X_INTVL    - Generation period to exchange individuals - Default 40
 * - distP      - Fraction of population for distance convergence - Default 0.45
 *
 ***********************************************************************
 * @section Revisions:
 *
 * Revision: 1.0   2018    Leonardo Bueno
 * * Original version based on BRKGA C++ API Sample Code:
 * A C++ APPLICATION PROGRAMMING INTERFACE FOR
 * BIASED RANDOM-KEY GENETIC ALGORITHMS
 * RODRIGO F. TOSO AND MAURICIO G.C. RESENDE, 2011
 *
 ***********************************************************************/

#include <iostream>
#include "ReshuffleDecoder.h"
#include "ReshuffleScenarioParser.h"
#include "MTRand.h"
#include "BRKGA.h"
#include <climits>
#include <time.h>
#include <string.h>

#define DEFAULT_SCENARIO_FILE                    ((char*)"scenarios\\
    scenario_Imax400Util95Org50.csv")
#define DEFAULT_PRINT_KEY                        false

#define DEFAULT_POPULATION                       (78)
#define DEFAULT_POPULATION_ELITE_FRACTION        (0.1625)
```

```
#define DEFAULT_POPULATION_MUTANT_FRACTION          (0.2631)
#define DEFAULT_PROBABILITY_INHERITANCE_FROM_ELITE  (0.3122)
#define DEFAULT_INDEPENDENT_POPULATIONS             (4)
#define DEFAULT_NUMBER_OF_THREADS                   (4)
#define DEFAULT_RANDOM_SEED                         (14159265)

#define DEFAULT_INDIVIDUAL_EXCHANGE_COUNT           (2)
#define DEFAULT_MAX_GENERATIONS                     (3000)
#define DEFAULT_INDIVIDUAL_EXCHANGE_GEN             (40)

#define DEFAULT_MAXDIST                             (0.001)
#define DEFAULT_MAXDIST_POLULATION_PERCENTAGE       (0.45)

/*
 * Quick Maximum Distance evaluation as defined in:
 * "Stopping Criteria for Single-Objective Optimization",
 * Karin Zielinski, Dagmar Peters, and Rainer Laur
 * 2007
 */
bool quickMaxDistConverged(BRKGA< ReshuffleDecoder, MTRand > &alg, double popPercentage, double distTolerance)
{
    double best = alg.getBestFitness();
    unsigned j = alg.getP()*popPercentage - 1;

    for(unsigned i = 0; i < alg.getK(); ++i)
    {
        if(alg.getPopulation(i).getFitness(j) > best+best*distTolerance)
        {
            return false;
        }
    }

    return true;
}

int main(int argc, char* argv[])
{
    char* scenarioFilePtr = DEFAULT_SCENARIO_FILE;
    bool printKey = DEFAULT_PRINT_KEY;

    unsigned p = DEFAULT_POPULATION;                              // size of population
    double pe = DEFAULT_POPULATION_ELITE_FRACTION;                // fraction of population to be the elite-set
    double pm = DEFAULT_POPULATION_MUTANT_FRACTION;               // fraction of population to be replaced by mutants
    double rhoe = DEFAULT_PROBABILITY_INHERITANCE_FROM_ELITE;     // probability that offspring inherit an allele from elite parent
    unsigned K = DEFAULT_INDEPENDENT_POPULATIONS;                 // number of independent populations
    unsigned MAXT = DEFAULT_NUMBER_OF_THREADS;                    // number of threads for parallel decoding
    long unsigned rngSeed =DEFAULT_RANDOM_SEED;                   // seed to the random number generator

    unsigned X_INTVL = DEFAULT_INDIVIDUAL_EXCHANGE_GEN;           // exchange best
```

```
    individuals at every X_INTVL generations
    unsigned X_NUMBER = DEFAULT_INDIVIDUAL_EXCHANGE_COUNT;        // exchanged top individuals

    unsigned MAX_GENS = DEFAULT_MAX_GENERATIONS;                  // maximum number of generations
    double m = DEFAULT_MAXDIST;                                   // Distance from best solution to assume convergence
    double distP = DEFAULT_MAXDIST_POLULATION_PERCENTAGE;         // Percentage of population with distance smaller than m to assume convergence

    std::cout << "\r\nGiven parameters: ";
    for(int argCount = 0; argCount < argc; argCount++)
    {
        std::cout << argv[argCount] << " ";
    }
    std::cout << "\r\n";

    if(argc >= 2)
    {
        scenarioFilePtr = argv[1];
    }

    char *findCSV = NULL;
    findCSV = strstr (scenarioFilePtr,".csv");

    if(!findCSV)
    {
        // Tell the user how to run the program
        std::cerr << "Usage: " << argv[0] << " FILE.csv (printBool) randomSeed P pe pm rhoe k MAX_GENS X_NUMBER X_INTVL" << std::endl;
        /* "Usage messages" are a conventional way of telling the user
         * how to run a program if they enter the command incorrectly.
         */
        return 1;
    }

    if(argc >= 3)
    {
        std::stringstream ss(argv[2]);

        if(!(ss >> std::boolalpha >> printKey))
        {
            // Tell the user how to run the program
            std::cerr << "Usage: " << argv[0] << " FILE.csv (printBool) randomSeed P pe pm rhoe k MAX_GENS X_NUMBER X_INTVL" << std::endl;
            /* "Usage messages" are a conventional way of telling the user
             * how to run a program if they enter the command incorrectly.
             */
            return 1;
        }
    }

    if(argc >= 4)
    {
        rngSeed = (unsigned long)atol(argv[3]);
    }
```

```
    if(argc >= 5)
    {
        p = (unsigned)atoi(argv[4]);
    }
    if(argc >= 6)
    {
        pe = atof(argv[5]);
    }
    if(argc >= 7)
    {
        pm = atof(argv[6]);
    }
    if(argc >= 8)
    {
        rhoe = atof(argv[7]);
    }
    if(argc >= 9)
    {
        K = (unsigned)atoi(argv[8]);
    }
    if(argc >= 10)
    {
        MAX_GENS = (unsigned)atoi(argv[9]);
    }
    if(argc >= 11)
    {
        X_NUMBER = (unsigned)atoi(argv[10]);
    }
    if(argc >= 12)
    {
        X_INTVL = (unsigned)atoi(argv[11]);
    }
    if(argc >= 13)
    {
        distP = atof(argv[12]);
    }

    std::string filePath(scenarioFilePtr);
    ReshuffleScenarioParser scenario(filePath, printKey);

    unsigned long int start = (unsigned long int)clock();
    ReshuffleDecoder decoder(scenario);         // initialize the decoder
    unsigned n = decoder.chromosomeSize();                      // size of
        chromosomes

    MTRand rng(rngSeed);                 // initialize the random number generator

    // initialize the BRKGA-based heuristic
    BRKGA< ReshuffleDecoder, MTRand > algorithm(n, p, pe, pm, rhoe, decoder, rng,
         K, MAXT);

    unsigned generation = 0;         // current generation
    unsigned bestGeneration = 0;         // current generation
    double bestFitness = 0;
    std::vector< double > bestChromosome;
    double curFitness = 0;
    do {
```

```
        algorithm.evolve(); // evolve the population for one generation

        if((++generation) % X_INTVL == 0) {
            algorithm.exchangeElite(X_NUMBER);  // exchange top individuals
        }
        curFitness = algorithm.getBestFitness();
        if(bestFitness != curFitness)
        {
            bestFitness = curFitness;
            bestGeneration = generation;

            if(printKey)
            {
                std::cout << "At generation " << generation <<" best solution found has objective value = "
                        << bestFitness << std::endl;
            }
        }
    } while ( (generation < MAX_GENS) &&
            (!quickMaxDistConverged(algorithm, distP, m))
    );

    bestChromosome = algorithm.getBestChromosome();

    if(printKey)
    {
        std::cout << "Best Chromosome Key = "<< std::endl;
        for (std::vector<double>::const_iterator i = bestChromosome.begin(); i != bestChromosome.end(); ++i)
            std::cout << *i << ',';
        std::cout << std::endl;
        std::cout << "Best Chromosome Decoded = "<< std::endl;
        decoder.printKey(bestChromosome);
        std::cout << std::endl;
    }

    unsigned long int brkga_milliseconds_since_start = (unsigned long int)(( (unsigned long int)clock() - start ) * 1000) / CLOCKS_PER_SEC;
    std::cout << std::endl << "BRKGA:\t" << bestFitness << "\tGeneration = " << bestGeneration;
    std::cout << "\tRuntime = "<<brkga_milliseconds_since_start<< "ms" << "\tEnd Generation = "<< generation << std::endl;
    std::cout << std::endl << bestFitness;

    // Output results
    std::ofstream outfile("reshuffleResultsBRKGA.csv", ios::out | ios::app);

    if (outfile.is_open())
    {
        outfile << argv[1];
        outfile << "," << bestFitness << "," << brkga_milliseconds_since_start << "," << bestGeneration; // GRH Results
        outfile << "," << rngSeed; // Random Seed
        outfile << "," << p << "," << pe << "," << pm << "," << rhoe << "," << K; // BRKGA Parameters
        outfile << "," << MAX_GENS << "," << X_INTVL << "," << X_NUMBER; // Execution Parameters
```

```
            outfile << "," << generation; // Conversion Generation

            outfile << "\n";
            outfile.close();
        }

    return 0;
}
```

APPENDIX B – SCENARIO GENERATION AND PARSING

B.1 EXAMPLE SCENARIO FILE

imax	9								
startPos	0								
Ik	6	5	1	2					
Fk	2	6	5	1					
gij	0	1	2	1	1	2	2	2	2
	1	0	1	1	1	1	2	2	2
	2	1	0	2	1	1	2	2	2
	1	1	2	0	1	2	1	1	2
	1	1	1	1	0	1	1	1	1
	2	1	1	2	1	0	2	1	1
	2	2	2	1	1	2	0	1	2
	2	2	2	1	1	1	1	0	1
	2	2	2	2	1	1	2	1	0
dij	0	1	2	1	1	2	2	2	2
	1	0	1	1	1	1	2	2	2
	2	1	0	2	1	1	2	2	2
	1	1	2	0	1	2	1	1	2
	1	1	1	1	0	1	1	1	1
	2	1	1	2	1	0	2	1	1
	2	2	2	1	1	2	0	1	2
	2	2	2	1	1	1	1	0	1
	2	2	2	2	1	1	2	1	0

Figure 23 – Example of CSV reshuffling Scenario outputted by ScenarioGenerator.py.

B.2 SCENARIO PARSER

Listing B.1 – ReshuffleScenarioParser.h

```
/**
 * @file ReshuffleScenarioParser.h
 * @version 1.0
 * @author Leonardo Bueno
 * @date April 18, 2018
 *
 ***********************************************************************
 *
 * @brief: Declares class ReshuffleScenarioParser
 *
 ***********************************************************************
 * @section Revisions:
```

```
 *
 * Revision: 1.0    2018     Leonardo Bueno
 * * Original version based on scenarios used in Jennifer Pazour's article:
 *       "Warehouse reshuffling: Insights and optimization" 2015
 *
 ***********************************************************************/

#ifndef ReshuffleScenarioParser_H
#define ReshuffleScenarioParser_H

#include <algorithm>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

class ReshuffleScenarioParser{
private:
    int startPos;               // Reshuffle start position
    unsigned int imax;          // Set of storage locations, indexed on i, j = 0,
         1, 2, ..., |I|.
    unsigned int omax;          // Number of empty positions
    unsigned int kmax;          // Set of items, indexed on k = 1, 2, ..., |K|.
    unsigned int cmax;          // Number C of sets of cycles, indexed on c = 1,
        2, ..., |C|.
    unsigned int nmax;          // Items not in a cycle
    double gmax;                // Max loaded cost
    double dmax;                // Max unloaded cost
    std::vector<int> Ik;        // The initial location of item k
    std::vector<int> Fk;        // The final location of item k;
    std::vector<int> Ii;        // Initial items in each location
    std::vector<int> Fi;        // Final items in each location
    std::vector<int> OIo;       // The open locations
    std::vector<int> OFo;       // The open locations
    std::vector<int> N;         // Number N of items that do not belong to a
        cycle (i.e., non-cycle items), indexed on k.
    std::vector<std::vector<int> > Cc;         // Set of items that belong to
        cycle c, indexed on c.
    std::vector<std::vector<double> > dij;     // Unloaded cost to travel from
        location i to j.
    std::vector<std::vector<double> > gij;     // Loaded Cost to travel from
        location i to j.
    std::vector<std::vector<double> > fields; // Variable used to store the csv
        fields

public:
    ReshuffleScenarioParser(string csvFileName, bool print);
    const vector<int> getIk(void) const;
    const vector<int> getFk(void) const;
    const vector<int> getIi(void) const;
    const vector<int> getFi(void) const;
    const vector<int> getOIo(void) const;
    const vector<int> getOFo(void) const;
    const vector<vector<double> > getDij(void) const;
```

```
    const vector<vector<double> > getGij(void) const;
    const vector<vector<int> > getCc(void) const;
    const unsigned int getImax(void) const;
    const unsigned int getOmax(void) const;
    const unsigned int getKmax(void) const;
    const int getStartPos(void) const;
    const double getGmax(void) const;
    const double getDmax(void) const;
};

#endif //ReshuffleScenarioParser_H
```

Listing B.2 – ReshuffleScenarioParser.cpp

```
/**
 * @file ReshuffleScenarioParser.h
 * @version 1.0
 * @author Leonardo Bueno
 * @date April 18, 2018
 *
 ***********************************************************************
 *
 * @brief: Implements class ReshuffleScenarioParser
 *
 ***********************************************************************
 * @section Revisions:
 *
 * Revision: 1.0   2018    Leonardo Bueno
 * * Original version based on scenarios used in Jennifer Pazour's article:
 *      "Warehouse reshuffling: Insights and optimization" 2015
 *
 ***********************************************************************/

#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <algorithm>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "ReshuffleScenarioParser.h"

using namespace std;

#define DEFAULT_STARTPOS    -1

#define LBL_COL         0
#define DATA_COL        0

#define IMAX_LINE       0
#define STARTPOS_LINE   1
#define IK_LINE         2
#define FK_LINE         4
#define GIJ_LINE        6

#define DIJ_LINE(imax)  (GIJ_LINE + imax + 1)

ReshuffleScenarioParser::ReshuffleScenarioParser(string csvFileName, bool print)
{
    // ReshuffleScenarioParser receives the name of csv file
    ifstream csvFile(csvFileName);  // Reading csv file
    unsigned int fieldCounter;      // field counter
    unsigned int j = 0;             // column counter
    unsigned int i = 0;             // item counter

    if (csvFile.is_open())
    {
```

```
        // Reading elements from every line and pushing into field vector of
            vectors
        string line;
        while (getline(csvFile, line))
        {
            stringstream sep(line);
            string field;
            fields.push_back(vector<double>());
            while (getline(sep, field, ','))
            {
                if(std::any_of(field.begin(), field.end(), ::isdigit))
                {
                    fields.back().push_back(stod(field));
                }
            }
        }
    }
    else
    {
        throw std::invalid_argument("Cannot open CSV file");
    }

    // Getting imax from csv file
    imax = fields[IMAX_LINE][DATA_COL];

    // Getting start position from csv file
    startPos = DEFAULT_STARTPOS;
    if(fields[STARTPOS_LINE].size())
    {
        startPos = fields[STARTPOS_LINE][DATA_COL];
    }

    //Checking errors csvFile startPos, Ik, Fk, Imax and Omax
    if(startPos >= (int)(imax))
    {
        throw std::invalid_argument("Invalid csv: startPos >= (imax)");
    }
    if((fields[IK_LINE].size() - DATA_COL) >= (imax))
    {
        throw std::invalid_argument("Invalid csv: Ik.size() >= (imax)");
    }
    if((fields[FK_LINE].size() - DATA_COL) >= (imax))
    {
        throw std::invalid_argument("Invalid csv: Fk.size() >= (imax)");
    }
    if(fields[IK_LINE].size()!=fields[FK_LINE].size())
    {
        throw std::invalid_argument("Invalid csv: Ik.size() != Fk.size()");
    }

    // Extracting Ik from fields(full csv in a matrix)
    for(fieldCounter=0; fieldCounter<fields[IK_LINE].size(); fieldCounter++)
    {
        Ik.push_back(fields[IK_LINE][fieldCounter+DATA_COL]);
        Fk.push_back(fields[FK_LINE][fieldCounter+DATA_COL]);
    }

```

```
    //  Defining Kmax
    kmax = Ik.size();

    // Find empty positions
    for (i = 0; i<imax; ++i)
    {
        Ii.push_back(-1);
        Fi.push_back(-1);

        // If position is not occupied as initial location of any item, it is initially empty
        if (std::find(Ik.begin(), Ik.end(), i) == Ik.end())
        {
            OIo.push_back(i);
        }
        // If position is not occupied as final location of any item, it is finally empty
        if (std::find(Fk.begin(), Fk.end(), i) == Fk.end())
        {
            OFo.push_back(i);
        }
    }

    // Number of empty positions
    omax = OIo.size();

    // Polynomial-time algorithm to identify cycles from Jennifer Pazour's article
    // "Warehouse reshuffling: Insights and optimization" 2015

    // Subset of items that contains all items that require reshuffling and
    // whose final location is initially occupied by another item.
    std::vector<int> L;
    int k; // Item under investigation
    int k_; // Item currently located item k's final location
    for (k = 0; k<(int)kmax; ++k)
    {
        Ii[Ik[k]] = k;
        Fi[Fk[k]] = k;

        // Element might be part of a cycle
        if ((Fk[k] != Ik[k]) && (std::find(Ik.begin(), Ik.end(), Fk[k]) != Ik.end()))
        {
            L.push_back(k);
        } else  // Element is not in a cycle
        {
            N.push_back(k);
        }
    }

    if(L.size() > 0)
    {
        i = 0;
        k = L[0];
        Cc.resize(i+1);
        Cc[i].push_back(k);
```

```
    }

    while(L.size() > 0)
    {
        k_ = std::distance(Ik.begin(), std::find(Ik.begin(), Ik.end(), Fk[k]));
        // if k_ not in L, then Ci is not a cycle;
        if(std::find(L.begin(), L.end(), k_) == L.end())
        {
            for (int index = (int)Cc[i].size(); index > 0; index--)
            {
                k=Cc[i][0];
                Cc[i].erase(Cc[i].begin(),Cc[i].begin()+1);
                N.push_back(k);
                L.erase(std::remove(L.begin(), L.end(), k), L.end());
            }

            if(L.size() > 0)
            {
                k = L[0];
                Cc[i].resize(0);
                Cc[i].push_back(k);
            }
        }
        else
        {
            // if k_ is Ci, then you have identified cycle Ci;
            if(std::find(Cc[i].begin(), Cc[i].end(), k_) != Cc[i].end())
            {
                L.erase(std::remove_if(L.begin(), L.end(),
                        [&] (int item) ->
                        bool {return std::find(Cc[i].begin(), Cc[i].end(), item)
                            != Cc[i].end();}),
                        L.end());

                i++;
                if(L.size() > 0)
                {
                    k = L[0];
                    Cc.resize(i+1);
                    Cc[i].push_back(k);
                }
            }
            // Identifying Ci
            else
            {
                Cc[i].push_back(k_);
                k = k_;
            }
        }
    }
    cmax = i;          //C = set of cycles, indexed on c = 1, 2, ... , |C|.

    nmax = N.size();          //Items not in a cycle

    // Extracting gij and dij
    gij.resize(imax, vector<double>(imax));   // Allocating gij (imax size)
    dij.resize(imax, vector<double>(imax));   // Allocating dij (imax size)
```

```
    gmax = 0;
    dmax = 0;

    for(fieldCounter=0;fieldCounter<imax;fieldCounter++)
    {
        std::vector<double> auxFieldLineGij(fields[fieldCounter+GIJ_LINE]);
        if(auxFieldLineGij.size() < imax)
        {
            throw std::invalid_argument("Invalid csv: gij line < imax");
        }

        std::vector<double> auxFieldLineDij(fields[fieldCounter+DIJ_LINE(imax)]);
        if(auxFieldLineDij.size()-DATA_COL < imax)
        {
            throw std::invalid_argument("Invalid csv: dij line < imax");
        }

        for(j=0; j<imax; j++)
        {
            gij[fieldCounter][j] = auxFieldLineGij[j+DATA_COL];

            //Checking errors in gij
            if(gij[fieldCounter][j]<0)
            {
                throw std::invalid_argument("Invalid csv: empty or incoherent
                    value gij");
            }

            if(gij[fieldCounter][j] > gmax)
            {
                gmax = gij[fieldCounter][j];
            }

            dij[fieldCounter][j] = auxFieldLineDij[j+DATA_COL];

            //Checking errors in dij
            if(dij[fieldCounter][j]<0)
            {
                throw std::invalid_argument("Invalid csv: empty or incoherent
                    value dij");
            }

            if(dij[fieldCounter][j]>gij[fieldCounter][j])
            {
                throw std::invalid_argument("Invalid csv: empty or incoherent
                    value dij");
            }

            if(dij[fieldCounter][j] > dmax)
            {
                dmax = dij[fieldCounter][j];
            }
        }
    }

    if(print)
```

```
    {
        std::cout << "imax: " << imax << endl;
        std::cout << "kmax: " << kmax << endl;
        std::cout << "Start Position: " << startPos << endl;

        std::cout << "Ik: ";
        for (std::vector<int>::const_iterator i = Ik.begin(); i != Ik.end(); ++i)
            std::cout << *i << ' ';
        std::cout <<endl;

        std::cout << "Fk: ";
        for (std::vector<int>::const_iterator i = Fk.begin(); i != Fk.end(); ++i)
            std::cout << *i << ' ';
        std::cout <<endl;

        std::cout << "Cycles: " << cmax << endl;
        for(unsigned c = 0; c<cmax; c++)
        {
            for(unsigned item=0; item < Cc[c].size(); item++)
            {
                std::cout << Cc[c][item] << ' ';
            }
            std::cout <<endl;
        }

        std::cout << "Non Cycles: " << nmax << endl;
        for (std::vector<int>::const_iterator i = N.begin(); i != N.end(); ++i)
            std::cout << *i << ' ';
        std::cout <<endl;

        std::cout << "Empty Locations: " << omax << endl;
        for (std::vector<int>::const_iterator i = OIo.begin(); i != OIo.end(); ++i)
            std::cout << *i << ' ';
        std::cout <<endl;
    }

}

const vector<int> ReshuffleScenarioParser::getIk(void) const
{
    return Ik;
}
const vector<int> ReshuffleScenarioParser::getFk(void) const
{
    return Fk;
}
const vector<int> ReshuffleScenarioParser::getIi(void) const
{
    return Ii;
}
const vector<int> ReshuffleScenarioParser::getFi(void) const
{
    return Fi;
}
const vector<int> ReshuffleScenarioParser::getOIo(void) const
{
```

```
    return OIo;
}
const vector<int> ReshuffleScenarioParser::getOFo(void) const
{
    return OFo;
}
const vector<vector<double> > ReshuffleScenarioParser::getDij(void) const
{
    return dij;
}
const vector<vector<double> > ReshuffleScenarioParser::getGij(void) const
{
    return gij;
}
const vector<vector<int> > ReshuffleScenarioParser::getCc(void) const
{
    return Cc;
}

const int ReshuffleScenarioParser::getStartPos(void) const
{
    return startPos;
}
const unsigned int ReshuffleScenarioParser::getImax(void) const
{
    return imax;
}
const unsigned int ReshuffleScenarioParser::getOmax(void) const
{
    return omax;
}
const unsigned int ReshuffleScenarioParser::getKmax(void) const
{
    return kmax;
}

const double ReshuffleScenarioParser::getGmax(void) const
{
    return gmax;
}
const double ReshuffleScenarioParser::getDmax(void) const
{
    return dmax;
}
```

B.3 SCENARIO GENERATOR

Listing B.3 – ScenarioGenerator.py

```
'''
Created on May 18, 2018

@author: Leonardo Bueno
'''
from __future__ import print_function
import sys
import os
sys.path.insert(0, os.path.dirname(os.path.realpath(__file__)))
import random
import numpy as np
import csv
from scipy.spatial.distance import squareform
from scipy.spatial.distance import pdist

def whireOutCsvFile(outputDict):
    with open(outputDict["outputFile"], 'w') as csvfile:
        outFile_ = csv.writer(csvfile, dialect='excel', quotechar='"',
                              quoting=csv.QUOTE_NONE, lineterminator = '\n')

        # Write imax
        writtenValue = ["imax"]
        writtenValue.extend([outputDict["imax"]])
        outFile_.writerow(writtenValue)

        # Write Start Position
        writtenValue = ["startPos"]
        writtenValue.extend([outputDict["startPos"]])
        outFile_.writerow(writtenValue)

        # Write Ik
        writtenValue = ["Ik"]
        writtenValue.extend(outputDict["Ik"])
        outFile_.writerow(writtenValue)

        writtenValue = []
        outFile_.writerow(writtenValue)

        # Write Fk
        writtenValue = ["Fk"]
        writtenValue.extend(outputDict["Fk"])
        outFile_.writerow(writtenValue)

        writtenValue = []
        outFile_.writerow(writtenValue)

        # Write gij
        writtenValue = ["gij"]
        for i in range(outputDict["imax"]):
            writtenValue.extend(outputDict["gij"][i])
            outFile_.writerow(writtenValue)
            writtenValue = [""]
```

```python

        outFile_.writerow(writtenValue)

        # Write dij
        writtenValue = ["dij"]
        for i in range(outputDict["imax"]):
            writtenValue.extend(outputDict["dij"][i])
            outFile_.writerow(writtenValue)
            writtenValue = [""]

def listCompare(x, y):
    count = 0
    for i in range(0, len(x)):
        if x[i] == y[i]:
            count += 1
    return count

def generateReshuffleScenario(imax, util_prct, org_prct, outputFile,
                              startPos = -1, cols = 1, distanceMetric = 'random',
                              unloadedDistanceMetric = 'random', finalOpenPositions = 'random'):

    util_prct = util_prct/100;
    org_prct = org_prct/100;

    if(imax < 2):
        raise ValueError("Invalid parameter: imax < 2")
    if((util_prct <= 0) or (util_prct >= 1.0)):
        raise ValueError("Invalid parameter: utilization >= 100% or utilization <= 0%")
    if((org_prct >= 1.0)):
        raise ValueError("Invalid parameter: organization >= 100%")
    if(".csv" not in outputFile):
        raise ValueError("Invalid parameter: output file is not .csv")

    if(startPos == "none"):
        startPos = -1;
    elif(startPos == "random"):
        startPos = random.randint(0, imax-1);
    elif(startPos.isdigit()):
        startPos = int(startPos);
    else:
        raise ValueError("Invalid parameter: startPos is not \"none\", \"random\", or \"digit\"")

    if(imax < startPos):
        raise ValueError("Invalid parameter: imax < startPos")

    kmax = np.int(imax*util_prct);

    equalLocation = np.int(kmax * org_prct);

    outputDict = {}
    outputDict["imax"] = imax
    outputDict["startPos"] = startPos
    outputDict["outputFile"] = outputFile

```

```
    i_list = list(np.random.permutation(imax))

    outputDict["Ik"] = list(i_list[:kmax]);

    if(finalOpenPositions == "random"):
        maxIndex = imax;
    elif(finalOpenPositions == "equal"):
        maxIndex = kmax;
    else:
        raise ValueError("Invalid parameter: finalOpenPositions is not \'equal\' nor \'random\'")

    while (listCompare(list(i_list[:kmax]),outputDict["Ik"]) > equalLocation):
        index1 = random.randint(0, maxIndex-1);
        index2 = random.randint(0, maxIndex-1);
        i_list[index1], i_list[index2] = i_list[index2], i_list[index1]

    outputDict["Fk"] = list(i_list[:kmax]);

    if(distanceMetric == "random"):
        #consider using randint to generate dij and gij with integer intervals
        gij_min = random.uniform(1, imax/2)

        gij = np.random.uniform(low=gij_min, high=imax, size=(imax, imax))
        np.fill_diagonal(gij, 0)
    else:
        if(cols > imax):
            raise ValueError("Invalid parameter: cols > imax")

        imaxHVList = [(int(i%cols), int(i/cols)) for i in range(imax)]
        gij = squareform(pdist(imaxHVList, distanceMetric))

    outputDict["gij"] = gij

    if(unloadedDistanceMetric == 'equal'):
        dij_deduction = 1;
    elif(unloadedDistanceMetric == 'random'):
        dij_deduction = random.uniform(0.1, 0.99)
    else:
        raise ValueError("Invalid parameter: unloadedDistanceMetric is not \'equal\' nor \'random\'")

    dij = dij_deduction * gij
    np.fill_diagonal(dij, 0)
    outputDict["dij"] = dij

    print("Imax: " + str(imax));
    print("Start Pos: " + str(startPos));
    print("Kmax: " + str(kmax));
    print("Organization: " + str(org_prct*100) + "%");
    print("Equal Locations: " + str(equalLocation));
    print("Final open positions: " + finalOpenPositions);
    print("Ik: " + str(outputDict["Ik"]));
    print("Fk: " + str(outputDict["Fk"]));
    print("Columns: " + str(cols));
    print("Distance Metric: " + distanceMetric);
    print("Unloaded Distance Metric: " + unloadedDistanceMetric);
```

```
    whireOutCsvFile(outputDict)

if __name__ == '__main__':
    try:
        imax = np.int(sys.argv[1]);
        util_prct = np.double(sys.argv[2]);
        org_prct = np.double(sys.argv[3]);

        outputFile = "scenario.csv"
        if(len(sys.argv) >= 5):
            outputFile = sys.argv[4];

        startPos = -1;
        if(len(sys.argv) >= 6):
            startPos = sys.argv[5];

        cols = 1;
        if(len(sys.argv) >= 7):
            cols = np.int(sys.argv[6]);

        distanceMetric = 'random';
        if(len(sys.argv) >= 8):
            distanceMetric = sys.argv[7];

        unloadedDistanceMetric = 'random';
        if(len(sys.argv) >= 9):
            unloadedDistanceMetric = sys.argv[8];

        finalOpenPositions = 'random';
        if(len(sys.argv) >= 10):
            finalOpenPositions = sys.argv[9];

        generateReshuffleScenario(imax, util_prct, org_prct, outputFile, startPos
            ,
                                            cols, distanceMetric, unloadedDistanceMetric,
                                                finalOpenPositions);

    except AssertionError:
        raise ValueError("Invalid parameter")
```

APPENDIX C – IRACE CONFIGURATION AND RESULTS

C.1 IRACE FILES FOR GRH PARAMETER TUNNING

C.1.1 Parameters

Listing C.1 – Parameters for GRH Irace execution

```
# name              switch               type      values
tau                 ""                   r         (0, 40)
```

C.1.2 Restrictions

No restrictions were applied for the GRH configuration

C.1.3 Evaluation Function

Listing C.2 – Evaluation Function for GRH Irace execution

```
#!/usr/bin/python
###############################################################################
# This script is the command that is executed every run.
# This script is run in the execution directory (execDir, --exec-dir).
#
# PARAMETERS:
# argv[1] is the candidate configuration number
# argv[2] is the instance ID
# argv[3] is the seed
# argv[4] is the instance name
# The rest (argv[5:]) are parameters to the run
#
# RETURN VALUE:
# This script should print one numerical value: the cost that must be minimized.
# Exit with 0 if no error, with 1 in case of error
###############################################################################

import datetime
import os.path
import re
import subprocess
import sys

exe = "C:\TunningBRKGA\iracePazour\PazourGRH\BuildPazourGRH.exe"
fixed_params = "false -1"

if len(sys.argv) < 5:
    print ("\nUsage: ./target-runner.py <candidate_id> <instance_id> <seed>")
    print ("<instance_path_name> <list of parameters>\n")
    sys.exit(1)

def target_runner_error(msg):
    now = datetime.datetime.now()
    print(str(now) + " error: " + msg)
    sys.exit(1)

# Get the parameters as command line arguments.
candidate_id = sys.argv[1]
instance_id = sys.argv[2]
seed = sys.argv[3]
instance = sys.argv[4]
cand_params = sys.argv[5:]

# Define the stdout and stderr files.
out_file = "c" + str(candidate_id) + "-" + str(instance_id) + ".stdout"
err_file = "c" + str(candidate_id) + "-" + str(instance_id) + ".stderr"

if not os.path.isfile(exe):
    target_runner_error (str(exe) + " not found")
if not os.access(exe, os.X_OK):
    now = datetime.datetime.now()
    print(str(now) + " error: " + str(exe) + " is not executable")

```

```
# Build the command, run it and save the output to a file,
# to parse the result from it.
#
# Stdout and stderr files have to be opened before the call().
#
# Exit with error if something went wrong in the execution.

command = [exe] + [instance] +fixed_params.split() +cand_params

outf = open(out_file, "w")
errf = open(err_file, "w")
return_code = subprocess.check_call(command, stdout = outf, stderr = errf)
outf.close()
errf.close()

if return_code != 0:
    now = datetime.datetime.now()
    print(str(now) + " error: command returned code " + str(return_code))
    sys.exit(1)

if not os.path.isfile(out_file):
    now = datetime.datetime.now()
    print(str(now) + " error: output file "+ out_file  +" not found.")
    sys.exit(1)
# This is an example of reading a number from the output.
# It assumes that the objective value is the first number in
# the first column of the last line of the output.

lastline = [line.rstrip('\n') for line in open(out_file)][-1]

# from http://stackoverflow.com/questions/4703390
numeric_const_pattern = r"""
     [-+]? # optional sign
     (?:
         (?: \d* \. \d+ ) # .1 .12 .123 etc 9.1 etc 98.1 etc
         |
         (?: \d+ \.? ) # 1. 12. 123. etc 1 12 123 etc
     )
     # followed by optional exponent part if desired
     (?: [Ee] [+-]? \d+ ) ?
     """
rx = re.compile(numeric_const_pattern, re.VERBOSE)

cost = rx.findall(lastline)[0]
print(cost)

os.remove(out_file)
os.remove(err_file)

sys.exit(0)
```

C.1.4 Scenario

Listing C.3 – Scenario for GRH Irace execution

```
################################################## -*- mode: r -*- #####
## Scenario setup for GRH Iterated Race (iRace).
######################################################################

## File that contains the description of the parameters.
parameterFile = "./parameters.txt"

## Directory where the programs will be run.
execDir = "./exec-dir/"

## File to save tuning results as an R dataset, either absolute path
## or relative to execDir.
logFile = "./irace.Rdata"

## Directory where tuning instances are located, either absolute path or
## relative to current directory.
trainInstancesDir = "../Instances"

## File with a list of instances and (optionally) parameters.
trainInstancesFile = "instances-list.txt"

## A file containing a list of initial configurations.
configurationsFile = "configurations.txt"

## The script called for each configuration that launches the program to be
## tuned.
targetRunner = "./target-runner.py"

## The maximum number of runs (invocations of targetRunner) that will
## performed. It determines the (maximum) budget of experiments for the tuning.
maxExperiments = 15000

## Enable/disable deterministic algorithm mode, if enabled irace
## will not use an instance more that once in each race. Note that
## if the number of instances provided is less than firstTest, no
## statistical test will be performed.
deterministic = 1
```

C.2 IRACE FILES FOR RESHUFFLE BRKGA PARAMETER TUNNING

C.2.1 Parameters

Listing C.4 – Parameters for Reshuffle Brkga Irace execution

```
# name        switch     type   values
p             ""         i      (10, 100)
pe            ""         r      (0,1)
pm            ""         r      (0,1)
rhoe          ""         r      (0,1)
K             ""         i      (1, 4)
MAX_GENS      ""         i      (50, 3000)
X_NUMBER      ""         i      (2, 5)
X_INTVL       ""         i      (30, 300)
```

C.2.2 Restrictions

Listing C.5 – Restrictions for Reshuffle Brkga Irace execution

```
pe+pm > 1
pe*p < 1
X_NUMBER*K > pe*p
X_INTVL > MAX_GENS
```

C.2.3 Evaluation Function

Listing C.6 – Evaluation Function for Reshuffle BRKGA Irace execution

```
#!/usr/bin/python
###############################################################################
# This script is the command that is executed every run.
# This script is run in the execution directory (execDir, --exec-dir).
#
# PARAMETERS:
# argv[1] is the candidate configuration number
# argv[2] is the instance ID
# argv[3] is the seed
# argv[4] is the instance name
# The rest (argv[5:]) are parameters to the run
#
# RETURN VALUE:
# This script should print one numerical value: the cost that must be minimized.
# Exit with 0 if no error, with 1 in case of error
###############################################################################

import datetime
import os.path
import re
import subprocess
import sys

exe = "C:\TunningBRKGA\iraceBRKGA\BuildBRKGA\BuildBRKGA.exe"
fixed_params = "false"

if len(sys.argv) < 5:
    print ("\nUsage: ./target-runner.py <candidate_id> <instance_id> <seed>")
    print ("<instance_path_name> <list of parameters>\n")
    sys.exit(1)

def target_runner_error(msg):
    now = datetime.datetime.now()
    print(str(now) + " error: " + msg)
    sys.exit(1)

# Get the parameters as command line arguments.
candidate_id = sys.argv[1]
instance_id = sys.argv[2]
seed = sys.argv[3]
instance = sys.argv[4]
cand_params = sys.argv[5:]

# Define the stdout and stderr files.
out_file = "c" + str(candidate_id) + "-" + str(instance_id) + ".stdout"
err_file = "c" + str(candidate_id) + "-" + str(instance_id) + ".stderr"

if not os.path.isfile(exe):
    target_runner_error (str(exe) + " not found")
if not os.access(exe, os.X_OK):
    now = datetime.datetime.now()
    print(str(now) + " error: " + str(exe) + " is not executable")

```

```
# Build the command, run it and save the output to a file,
# to parse the result from it.
#
# Stdout and stderr files have to be opened before the call().
#
# Exit with error if something went wrong in the execution.

command = [exe] + [instance] +fixed_params.split() +cand_params

outf = open(out_file, "w")
errf = open(err_file, "w")
return_code = subprocess.check_call(command, stdout = outf, stderr = errf)
outf.close()
errf.close()

if return_code != 0:
    now = datetime.datetime.now()
    print(str(now) + " error: command returned code " + str(return_code))
    sys.exit(1)

if not os.path.isfile(out_file):
    now = datetime.datetime.now()
    print(str(now) + " error: output file "+ out_file +" not found.")
    sys.exit(1)
# This is an example of reading a number from the output.
# It assumes that the objective value is the first number in
# the first column of the last line of the output.

lastline = [line.rstrip('\n') for line in open(out_file)][-1]

# from http://stackoverflow.com/questions/4703390
numeric_const_pattern = r"""
    [-+]? # optional sign
    (?:
        (?: \d* \. \d+ ) # .1 .12 .123 etc 9.1 etc 98.1 etc
        |
        (?: \d+ \.? ) # 1. 12. 123. etc 1 12 123 etc
    )
    # followed by optional exponent part if desired
    (?: [Ee] [+-]? \d+ ) ?
    """
rx = re.compile(numeric_const_pattern, re.VERBOSE)

cost = rx.findall(lastline)[0]
print(cost)

os.remove(out_file)
os.remove(err_file)

sys.exit(0)
```

C.2.4 Scenario

Listing C.7 – Scenario for Reshuffle BRKGA Irace execution

```
###################################################### -*- mode: r -*- #####
## Scenario setup for GRH Iterated Race (iRace).
############################################################################

## File that contains the description of the parameters.
parameterFile = "./parameters.txt"

## Directory where the programs will be run.
execDir = "./exec-dir/"

## File to save tuning results as an R dataset, either absolute path
## or relative to execDir.
logFile = "./irace.Rdata"

## Directory where tuning instances are located, either absolute path or
## relative to current directory.
trainInstancesDir = "../Instances"

## File with a list of instances and (optionally) parameters.
trainInstancesFile = "instances-list.txt"

## A file containing a list of initial configurations.
configurationsFile = "configurations.txt"

## The script called for each configuration that launches the program to be
## tuned.
targetRunner = "./target-runner.py"

## The maximum number of runs (invocations of targetRunner) that will
## performed. It determines the (maximum) budget of experiments for the tuning.
maxExperiments = 4000
```