UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

# A data structure for spanning tree optimization problems

**Marco Aurélio Lopes Barbosa**

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

**Marco Aurélio Lopes Barbosa**

# A data structure for spanning tree optimization problems

Thesis submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Doctor in Science. *EXAMINATION BOARD PRESENTATION COPY*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Alexandre Cláudio Botazzo Delbem

**USP – São Carlos**
**March 2019**

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

B238d

Barbosa, Marco Aurélio Lopes
A data structure for spanning tree optimization problems / Marco Aurélio Lopes Barbosa; orientador Alexandre Cláudio Botazzo Delbem. -- São Carlos, 2019.
94 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2019.

1. Estrutura de dados. 2. Árvores geradoras. 3. Algoritmos evolutivos. 4. Busca local. I. Delbem, Alexandre Cláudio Botazzo, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

**Marco Aurélio Lopes Barbosa**

# Uma estrutura de dados para problemas de otimização de árvores geradoras

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Alexandre Cláudio Botazzo Delbem

**USP – São Carlos**
**Março de 2019**

# ABSTRACT

BARBOSA, M. A. L. **A data structure for spanning tree optimization problems**. 2019. 94 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Spanning tree optimization problems are related to many practical applications. Several of these problems are NP-Hard, which limits the utility of exact methods and can require alternative approaches, like metaheuristics. A common issue for many metaheuristics is the data structure used to represent and manipulate the solutions. A data structure with efficient operations can expand the usefulness of a method by allowing larger instances to be solved in a reasonable amount of time. We propose the 2LETT data structure and uses it to represent spanning trees in two metaheuristics: mutation-based evolutionary algorithms and local search algorithms. The main operation of 2LETT is the exchange of one edge in the represented tree by another one, and it has $O(\sqrt{n})$ time, where $n$ is the number of vertices in the tree. We conducent qualitative and quantitative evaluations for 2LETT and other structures in the literature. For the main operation of edge exchange in evolutionary algorithms, the computational experiments show that 2LETT has the best performance for trees with more than 10,000 vertices. For local search algorithms, 2LETT is the best option to deal with large trees with large diameters.

**Keywords:** Dynamic tree data structures, spanning trees, evolutionary algorithms, local search algorithms.

# RESUMO

BARBOSA, M. A. L. **Uma estrutura de dados para problemas de otimização de árvores geradoras**. 2019. 94 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Os problemas de otimização de árvores geradoras estão relacionados a muitas aplicações práticas. Vários desses problemas são NP-difícies, o que limita a utilidade de métodos exatos e pode exigir abordagens alternativas, como metaheurísticas. Um questão relevante para muitas metaheurísticas é a estrutura de dados usada para representar e manipular as soluções. Uma estrutura de dados com operações eficientes pode aumentar a utilidade de um método, permitindo que instâncias maiores sejam resolvidas em um período de tempo razoável. Propomos a estrutura de dados 2LETT e a usamos para representar árvores geradoras em duas metaheurísticas: algoritmos evolutivos baseados em mutações e algoritmos de busca local. A operação principal da 2LETT é a troca de uma aresta na árvore representada por outra aresta. Esta operação tem tempo de $O(\sqrt{n})$, onde $n$ é o número de vértices na árvore. Conduzimos avaliações qualitativas e quantitativas para 2LETT e outras estruturas na literatura. Para a principal operação de troca de arestas em algoritmos evolutivos, os experimentos computacionais mostram que a 2LETT possui o melhor desempenho para árvores com mais de 10.000 vértices. Para algoritmos de busca local, o 2LETT é a melhor opção para lidar com árvores grandes com grandes diâmetros.

**Palavras-chave:** Estrutura de dados de árvores dinâmicas, árvores geradoras, algoritmos evolutivos, algoritmos de busca local.

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| 2LETT | 2-level Euler tour tree |
| DCMSTP | Degree constrained minimum spanning tree problem |
| EA | Evolutionary algorithm |
| ILS | Iterated local search |
| LS | Local search |
| MSTC | Minimum spanning tree problem with conflict constraints |
| MSTP | Minimum spanning tree problem |
| NDDR | Node-depth-degree representation |
| TS | Tabu search |
| TSP | Travelling salesman problem |
| TT | Tabu thresholding |

# LIST OF SYMBOLS

$d(v)$ — The depth of the vertex $v$ in a rooted tree

$deg(v)$ — The degree of the vertex $v$

$E$ — Edge set of a graph

$E_T$ — Edge set of $T$

$G = (V, E)$ — A graph with vertex set $V$ and edge set $E$

$n$ — Number of vertex in a graph

$m$ — Number of edges in a graph

$T$ — A tree of a graph

$\overline{T}$ — The complement graph of $T$

$uTv$ — The path from $u$ to $v$ in $T$

$V$ — Vertex set of a graph

$V_T$ — Vertex set of $T$

# CONTENTS

CHAPTER

1

# INTRODUCTION

A spanning tree of an undirected graph $G$ is an acyclic connected subgraph of $G$ whose edges span all vertices of $G$ (BONDY; MURTY, 2011). A spanning tree optimization problem consists in finding a spanning tree of a given graph that respects a set of restrictions and is optimal according to an objective function. These problems are significant because they are related to many practical applications in areas such as telecommunications (HU, 1974), transportation (GLOVER; KLINGMAN, 1975), and phylogenetics (FELSENSTEIN, 2003).

Although some spanning tree problems can be solved in polynomial time, as the minimum spanning tree problem (PRIM, 1957; KRUSKAL, 1956), many are NP-Hard, such as the degree-constrained minimum spanning tree problem (GAREY; JOHNSON, 1990), and the minimum spanning tree with conflict constrains problem (DARMANN; PFERSCHY; SCHAUER, 2009). For the latter, some research using metaheuristics is made, since exact algorithms are unable to solve large instances in reasonable time.

Each metaheuristic has its design issues, but a common concern for most of them are the underlying data structures used to store and manipulate solutions to the problem. An efficient data structure can make a method faster and also enable it to solve large instances in a reasonable time. For example, Fredman *et al.* (1995) presents a comparison of data structures for local search algorithms for the classical traveling salesman problem. Their results show that the efficient data structures significantly outperformed the less efficient one, and could solve instances ten times larger in a reasonable time. In this work, we study efficient spanning trees data structures for two metaheuristics: mutation-based evolutionary algorithms and local search algorithms.

An evolutionary algorithm keeps a population of individuals that change over time (MICHALEWICZ, 1996; JONG, 2006). An individual encodes a problem solution, and new individuals (offspring) are created from existing individuals (parents) through

mutation and crossover operators. The mutation operator generates an offspring by a small modification in a parent. By its turn, the crossover operator combines two or more parents in order to generate at least one offspring. The offspring compete with existing individuals to remain in the population as it continually changes until a criterion is reached. In the end, the best individual in the population, according to a fitness (objective) function, is returned as the problem solution. In mutation-based evolutionary algorithms, only mutation operators are used. A common mutation for spanning trees is the exchange of one of its edges by another one.

A local search algorithm starts from an initial solution and iteratively replaces it according to a neighborhood structure and an acceptance criterion (PAPADIMITRIOU; STEIGLITZ, 1998). A neighborhood of a solution $T$ is the set of all solutions that are somehow related to $T$. For example, the one-edge exchange neighborhood for a spanning tree $T$ is the set of all valid solutions that can be obtained from $T$ by exchanging one of its edges by another one. The acceptance criterion is used to determine when a neighbor solution replaces the current one. In general, better solutions are always accepted.

Noticeably, the main issue in common with evolutionary and local search algorithms is the edge exchange operation. We shall discuss how this can be a bottleneck for both metaheuristics, and how important it is to improve it to obtain faster algorithms. For evolutionary algorithms, another important aspect is to avoid copying parents when generating offspring (DELBEM; LIMA; TELLES, 2012). Once an offspring is created, its parent can coexist with it in the population, which prevents us from obtaining an offspring by just modifying the parent. Most commonly is copying the parent before generating the offspring, but copying it can be expensive, which would compromise the running time of the algorithm.

The edge exchange operation can be implemented by predecessor arrays (BONDY; MURTY, 2011), a tree data structure often used by graph algorithms, requiring time $O(n)$ where $n$ is the number of vertices in the tree. A more sophisticated alternative is to use a dynamic tree data structure, like the Euler tour tree which has operations in time $O(\log n)$ (TARJAN; WERNECK, 2007). However, for evolutionary algorithms, it would need to be adapted to work with a population of solutions, and as far as we know, this has not been done yet. We discuss using dynamic trees in local search algorithms in Chapter 5.

Another data structure with sublinear edge exchange operation is NDDR (DELBEM; LIMA; TELLES, 2012), which is a spanning forest data structure designed ad hoc to be used in EA's, and carefully planned to address the two previously mentioned issues. The innovative aspect that helps NDDR to overcome those problems is the decomposition of trees in substructures that can be shared between parent and offspring (this approach is called structural sharing in persistent data structure literature (KAPLAN, 2004)). This resulted in operators having average time $O(\sqrt{n})$, which is a great asymptotic improvement

compared to linear time implementations, as predecessor arrays.

## 1.1 Contributions

We rely on both Euler tour trees and substructures decomposition to develop a new data structure called 2LETT, which can perform edge exchange operations in time $O(\sqrt{n})$ in worst-case. For evolutionary algorithms, it uses structural sharing as NDDR. We organize this work in three phases as follows:

1. The design of 2LETT and its evaluation compared to NDDR and predecessor arrays in evolutionary algorithms.

   First, we consider direct spanning tree representations and their search operators. Then we define two mutation operators based on edge exchange, as well as discuss important aspects that should be addressed in order to implement them. We review predecessor arrays and NDDR, and we describe the 2LETT structure. We evaluate the structures qualitatively and experimentally.

   Our main contribution here is the development of the 2LETT structure, which presented the best performance for the mutation that exchanges any two valid edges for graphs with more than $10,000$ vertices. Another interesting contribution is the realization that predecessor arrays, even having the worst asymptotic time, had a remarkable performance, even comparable to efficient structures. They were the best structure for random trees with less than $10,000$ vertices.

2. Exploration of local search algorithm involving one-edge and two-edges exchange neighborhood.

   In this phase, we investigate how to efficiently implement local search algorithms for the one-edge and two-edges exchange neighborhoods. We evaluate both approaches for the minimum spanning tree with conflict constrains problem, concluding that the local search using the two-edge exchange neighborhood can be implemented efficiently and yield better results.

3. Adaptation of 2LETT for a local search algorithm.

   Finally, in this phase we show how to adapt 2LETT to work without structural sharing, making it faster for local search algorithms. We compare its performance with predecessor arrays and link-cut trees, a dynamic tree data structure. We also investigate the difference of using two-edges exchange versus a variable number of edges exchange neighborhood in the degree-constrained minimum spanning tree problem.

## 1.2 Thesis outline

The thesis is organized as follows. In Chapter 2 we discuss the main concepts used in the research. In Chapter 3 we present the 2LETT data structure and compare it with NDDR and predecessor arrays in the context of evolutionary algorithms. In Chapter 4 we investigate how to efficiently implement local search for one-edge and two-edges exchange neighborhoods. In Chapter 5 we show how to adapt 2LETT to local search algorithms. Finally, in Chapter 6 we present some conclusions and directions for future research.

CHAPTER

2

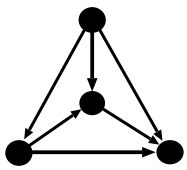
# CONCEPTS REVIEW

In this chapter, we present the fundamental concepts used throughout the work. We start in section 2.1 with graph definitions and notation. In section 2.2 we describe some optimization spanning tree problems and then discuss evolutionary algorithms, a metaheuristics commonly used to solve hard problems, in section 2.3. Finally, we present persistent data structures in section 2.4, which are essential for the proposal of our structure in Chapter 3.

## 2.1 Graphs

A *graph* $G$ is an ordered pair $(V,E)$ where $V$ is a finite set of *vertices*, and $E$ is a finite set of *edges*. Each edge $e$ is a pair of vertices, called the *extremes* of $e$. If the order of vertices is meaningful, the graph is called a *directed graph* (see Figure 1) or *digraph*, otherwise the graph is an *undirected graph* (see examples in Figure 3). The edges $(u,v)$ and $(v,u)$ represent the same edge in undirected graphs.

Figure 1 – A directed graph or digraph.



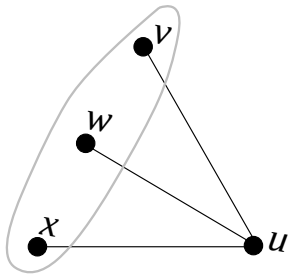
Source: Elaborated by the author.

We denote the size of $V$ by $n$, and the size of $E$ by $m$, i.e., $|V| = n$ and $|E| = m$.

We say that an edge $(u,v)$ is *incident* to the vertices $u$ and $v$. Also, $v$ is *adjacent* to $u$, and vice versa. The *degree* of a vertex $u$, denoted by $deg(u)$, is the number of edges incident to it, and the neighborhood of $u$ is the set of its adjacent vertices. Figure 2

highlights the neighborhood of a vertex $u$ in a graph $G$. Vertices $v$, $w$ and $x$ are adjacent to $u$, therefore $deg(u) = 3$.

Figure 2 – Neighborhood of a vertex $u$.




Source: Elaborated by the author.

A *path* from a vertex $x$ to a vertex $y$ in a graph $G = (V,E)$ is a sequence of vertices $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ such that $x = v_0$, $y = v_k$ and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The *length* of a path is its number of edges. We say that the path contains the vertices $v_0, v_1, \ldots, v_k$, and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. If there is a path $p$ from $u$ to $u'$, we say that $u'$ is *reached* from $u$ through $p$, or $u \overset{p}{\rightsquigarrow} u'$ if the graph is directed.

A graph is *complete* if, for each pair of distinct vertices $u$ and $v$, the vertex $v$ is adjacent to $u$. See examples of complete graphs for $n = 3, 4, 5$ in Figure 3.

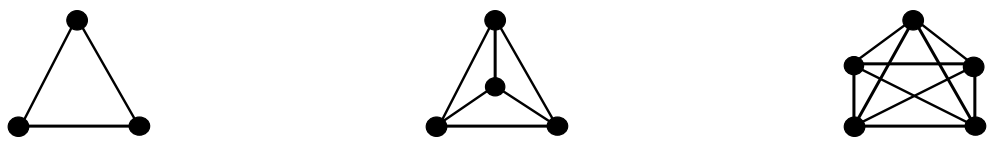
For a graph $G = (V,E)$, an *independent set* is a set $S$ of vertices such that, for every two vertices in $S$, there is no edge connecting the two. In other words, each edge in $V$ has at most one endpoint in $S$ (see Figure 4).

In a directed graph, a path $\langle v_0, v_1, \ldots, v_k \rangle$ makes a *cycle* if $v_0 = v_k$ and the path contains at least an edge. In an undirected graph, a path $\langle v_0, v_1, \ldots, v_k \rangle$ makes a cycle if $k > 0$, $v_0 = v_k$ and all edges in the path are distinct. A graph without cycles is *acyclic* (see an example in Figure 1).

A graph $H = (V', E')$ is a *subgraph* of a graph $G = (V,E)$ if $V' \subseteq V$ and $E' \subseteq E$. If $V' = V$, the subgraph is called *spanning* (see Figure 5).

A *connected component* of an undirected graph $G = (V,E)$ is a maximal subgraph

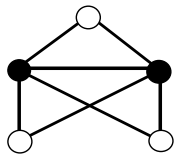
Figure 3 – Complete graphs for $n = 3, 4, 5$.



(a) $n = 3$ (b) $n = 4$ (c) $n = 5$

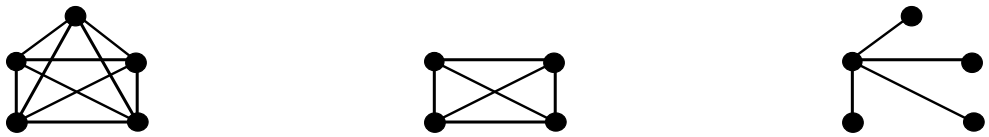
Source: Elaborated by the author.

Figure 4 – Independent set for a graph: the white vertices make an independent set.



Source: Elaborated by the author.

Figure 5 – Subgraphs.



(a) A graph $G$ (b) A subgraph of $G$ (c) Spanning subgraph of $G$

Source: Elaborated by the author.

$H = (V', E')$ of $G$ such that, for each pair of vertices $u, v \in V'$, we have that $v$ is reachable from $u$ in $H$. A graph is *connected* if it is only one connected component.

A *forest* is an acyclic undirected graph not necessarily connected. A *tree* is a connected acyclic undirected graph (see Figure 5c). Therefore, each component of a forest is a tree.

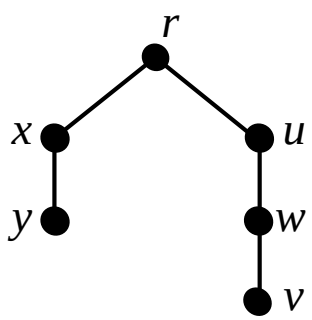
A *spanning tree* $T = (V_T, E_T)$ of an undirected connected graph $G = (V, E)$ is an acyclic connected subgraph of $G$ such that $V_T = V$ (see Figures 5a and 5c). When $G$ has weights on its edges, the *weight of a spanning tree* is the sum of the weights in its edges. A *minimum spanning tree (MST)* of a graph $G$ is a spanning tree with minimum weight. We denote by $\overline{T}$ the complement of $T$ in $G$, i.e., $\overline{T} = (V_T, E \setminus E_T)$. Given two distinct vertices $u$ and $v$ in $V$, we denote the unique path between $u$ and $v$ in $T$ by $uTv$.

The *depth* of a vertex $u$ in a tree $T$ with root $r$ is the length of the path $uTr$, denoted by $d(v)$. The *height* of a tree is the maximum depth of any of its vertices. The *diameter* of a tree is the longest distance between any pair of vertices in it.

From the root $r$ of a tree, for any edge $(u, w) \in vTr$, we say $u$ is the *predecessor* or *parent* of $v$, denoted by $p(v)$, if $u$ is closer to $r$ than $w$. In this case, we also say $w$ is a *child* of $u$. The vertices that are reachable from a vertex $u$ by repeatedly going from child to parent are called *ancestors*. Similarly, the vertices reached by repeatedly going from parent to child are known as *descendants*. A *u-subtree* is a subtree rooted in $u$ containing all the descendants of $u$ in $T$. For example, for the tree in Figure 6, the vertex $u$ is the parent of $w$ and $w$ is child of $u$. Also, $r$, $u$ and $w$ are all ancestors of $v$, while $x, y, u, w$ and $v$ are all descendants of $r$. The $u$-subtree has $w$ and $v$ as its descendants.

In the next section, we present some spanning tree optimization problems.

Figure 6 – A tree rooted in $r$.




Source: Elaborated by the author.

## 2.2 Spanning tree optimization problems

The *minimum spanning tree problem (MSTP)* is a classical optimization problem (PAPADIMITRIOU; STEIGLITZ, 1998). Given an undirected graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}$, the MSTP consists in finding a spanning tree with minimum weight. In other words, we need to find an acyclic connected subgraph $T = (V, E_T)$ with minimum $w(T) = \sum_{e \in E_T} w(e)$.

While there are some polynomial algorithms to solve the MSTP, among which the best known are Prim's algorithm (PRIM, 1957) and Kruskal's algorithm (KRUSKAL, 1956), many spanning tree problems are NP-Hard. Next, we discuss two such problems, as well as a related benchmark problem. They are used in various experimental evaluations through the next chapters.

### *Degree-constrained minimum spanning tree problem*

The *degree-constrained minimum spanning tree problem (DCMSTP)* is similar to MSTP, but has an additional restriction: given a value $v_d \geq 2$, for each vertex $v$ of $G$, $v$ can have at most $v_d$ (NARULA; HO, 1980) neighbours in $T$. Despite the similarity with MSTP, this restriction makes the problem to be NP-hard (GAREY; JOHNSON, 1990). Note that when $v_d = 2$ for all vertices of $G$, DCMSTP becomes similar to the classical traveling salesman problem (TSP) (LAWLER, 1985), which is used in the evaluation in Chapter 5.

### *Minimum spanning tree problem with conflict constraints*

The *minimum spanning tree problem with conflict constraints (MSTC)* is also similar to MSTP (DARMANN; PFERSCHY; SCHAUER, 2009). Given an undirected graph $G = (V, E)$, a cost function $w : E \to \mathbb{R}^+$, and a set of conflicting edges $C \subset E \times E$, the MSTC consists in finding a spanning tree $T$ of $G$ with a minimum cost $\sum_{e \in E(T)} w(e)$

so that $T$ is conflict free, i.e., $T$ contains no pair of edges in $C$. We use this problem in the evaluation in Chapter 4.

### One-tree max problem

The *one-max tree problem* is a benchmark problem defined by Rothlauf, Goldberg and Heinzl (2002) and is used to evaluate spanning tree representations. The objective is to find a given tree $T_{obj}$. A solution $T$ is evaluated by the objective function $\frac{|E_{T_{obj}} \cap E_T|}{n-1}$, that is, the function value is proportional to the number of common edges between $T$ and $T_{obj}$. If $T$ has no edges in common with $T_{obj}$, then the function value is 0, and if $T = T_{obj}$, then the function value is 1. This problem is used in the evaluation in Chapter 3.

Next, we discuss evolutionary algorithms, a metaheuristic used to solve hard spanning tree problems.

## 2.3 Evolutionary algorithms

An *evolutionary algorithm* (EA) is a population-based metaheuristic, inspired in nature's principles, and largely used to solve optimization problems (GEN; CHENG, 1997; MICHALEWICZ, 1996).

An EA maintains a population of promising individuals, where each individual represents a possible solution for the problem. New individuals, called *offspring*, are created from existent individuals, called *parents*, through search operators like mutation and crossover. In *mutation* operation, an offspring is created from a small modification in a parent. In *crossover* operation (also known as *recombination*), two or more parents are combined to generate a new offspring. New individuals compete with existent individuals to remain in the population of solutions (JONG, 2006).

Each individual is evaluated by a *fitness function* that indicates the quality of the solution related to that individual for the problem. More promising individuals (with better fitness) are *selected* more often to be used to generate new individuals and to stay in the population. This intent to keep the characteristics of good solutions in the population (MICHALEWICZ, 1996). The population is modified until a predetermined criterion is satisfied. The best solution from the population is then returned as the answer from the EA.

Next section, we discuss the process of selection and how it affects an EA behavior. After that, we discuss some important aspects for a representation of individuals and their search operators.

### 2.3.1 Selection

Selection mechanisms are used to choose individuals to reproduce, as well as to choose individuals to remain in the population. Two factors need to be balanced in a selection process: selection pressure and diversity (GEN; CHENG, 1997).

*Selection pressure* is the tendency to select individuals with better fitness, guiding an EA in the search for global optimum, while the *diversity* is essential to provide a good exploration of the search space. Too much selection pressure decreases diversity and can cause a premature convergence to a local optimum. On the other hand, too little selection pressure makes convergence being too slow (GEN; CHENG, 1997).

Among the main selection strategies are the uniform, truncation, tournament, and fitness proportionate (JONG, 2006). In *uniform selection*, all individuals have the same probability of being chosen, regardless of their fitness value. This kind of selection is commonly used together with the truncation selection. In *truncation selection*, the $n$ individuals with the best fitness value are chosen, where $n$ is usually the population size. When combining uniform and truncation selections, the former is used to choose individuals for reproduction, and the latter is used to choose individuals to remain in the population.

*Tournament selection* chooses the individual with the best fitness among $k$ randomly chosen individuals. When $k = 1$, it behaves exactly like the uniform selection. Something interesting about the tournament selection is the possibility of adjusting the selection pressure by modifying the value of $k$. The higher the value of $k$, the higher the selection pressure. Usually, $k$ takes values from 2 to 10.

In *fitness proportionate selection*, each individual has a probability of being chosen that is proportional to its fitness. The proportionate selection was widely used in the first evolutionary algorithms, but its use was mostly replaced by tournament selection.

### 2.3.2 Representations and search operators

Two of the most important decisions to make when designing an EA are choosing a representation for solutions and defining its search operators (RAIDL; JULSTROM, 2003). Naturally, the operators strongly depend on the solutions representation.

A representation can be classified as direct or indirect (ROTHLAUF, 2006). In *direct representations*, individuals directly represent the values of variables of a candidate solution for the problem (decision space). In *indirect representations*, individuals are usually encoded as a string of values (*genotype*), and a mapping function is used to convert to/from solutions in the decision space (*phenotype*).

Rothlauf (2006) analyzes some trade-offs in the design of direct and indirect repre-

sentations. There are many general indirect representations and search operators defined for them, so the design concern for indirect representations is to define a genotype to phenotype mapping function. On the other hand, there are no general search operators for direct representations, since different problem domains require different representations. As a consequence, the main concern to design them is the definition of specific search operators. According to Rothlauf (2006), neither design issue is easier than the other.

Two important properties of search operators, independent of representation type, is locality for mutation and heritability for crossover (CAMINITI; PETRESCHI, 2005; ROTHLAUF, 2006). A mutation operator has high *locality* if the individual created by mutation is similar to its parent. As for indirect representations, the similarity must be related to the phenotype, since the fitness function evaluates the phenotype – not the genotype –, even though the mutation operator is applied over the genotype.

For crossover operator, *heritability* means that new individuals generated by it are a combination of the substructures of their parents, and so it adds few new substructures to the population. Moreover, as well as for locality in indirect representations, the feature must be evaluated over the phenotype, not over the genotype.

For evolutionary algorithms, we propose a structure designed for direct representations. We shall discuss our proposal and other direct representations in Chapter 3. Next, we present a common indirect representation and its search operators, as well as two employments of that for spanning trees.

### Fixed size arrays

Fixed size arrays are a usual type of indirect representation. They can contain binary numbers, integers, or real numbers. Since they have already been extensively studied, there are many mutation and crossover operators for them.

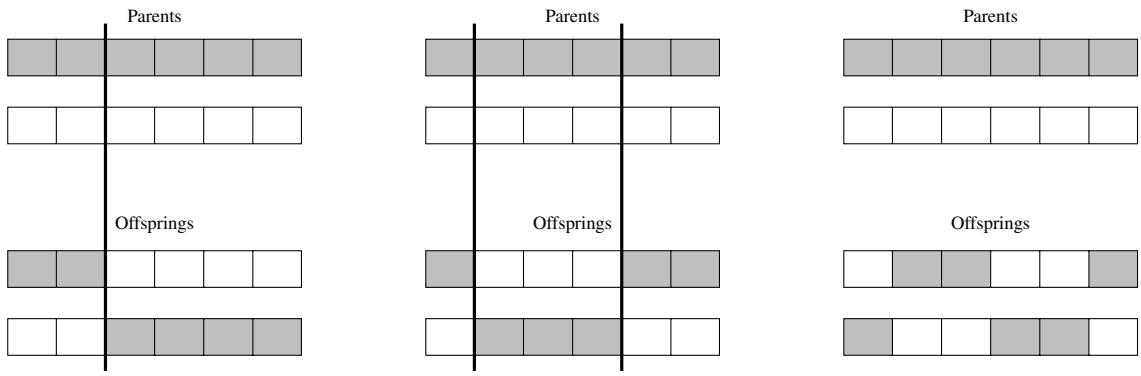
The *single-point mutation* establishes a probability for each position of the array to be selected and then have its value modified (GEN; CHENG, 1997). For binary arrays, this means to invert a value. For real or integer number arrays, values can be replaced according to a distribution.

Some popular recombination operators for fixed size arrays are (JONG, 2006):

(i) *Single-point crossover*: it determines a random point on both parents, then the right part of that point is swapped between the parents, resulting in two offspring (Figure 7a);

(ii) *Two-point crossover*: it determines two points on the parents, and the part between those points are swapped between the parents (Figure 7b);

Figure 7 – Examples of recombination for fixed size arrays




(a) Single-point (b) Two-point (c) Uniform

Source: Elaborated by the author.

(iii) *k-point crossover*: a generalization of the two-point crossover, the $k$-point crossover determines $k$ points, and the parts between them are swapped between the parents in order to generate the offspring;

(iv) *Uniform crossover*: each position of the offspring can inherit the value of any of the parents (Figure 7c).

Fixed size arrays are used by many indirect spanning tree representation. Some of them, like Prüfer numbers (PRUFER, 1918) and Dandelion code (THOMPSON; PAULDEN; SMITH, 2007), are based on the result of (CAYLEY, 1889), which shows that there are $2^{n-2}$ labeled trees on $n$ vertices. They encode an individual as an array of size $n-2$, with values in the range $[1,n]$, and defines a bijective function that maps each array to a unique tree and vice-versa. One advantage of this scheme is that, on complete graphs, only valid trees can be represented.

The encoding/decoding function for both Prüfer numbers and Dandelion code can be implemented in linear time. However, they differ in the quality of their operation. While Prüfer numbers present low locality and heritability, making an EA using them to be similar to a random search (GOTTLIEB; JULSTROM; RAIDL, 2001), Dandelion code presents high locality and heritability (THOMPSON; PAULDEN; SMITH, 2007), making it more appropriated for EAs than Prüfer numbers.

Next, we discuss local search algorithms, another metaheuristic used to solve hard spanning tree problems.

## 2.4 Persistent data structures

A persistent data structure is opposed to an ephemeral (non-persistent) one. In an *ephemeral data structure*, the update operations change the internal state of the structure, making it impossible to access again its previous state. In a *persistent data structure*, the update operations create a new version of the structure, in order to allow queries and changes to both versions, the previous and the current one (DRISCOLL *et al.*, 1989; OKASAKI, 1998).

The literature about persistent data structures can roughly be classified into three categories (KAPLAN, 2004):

1. General transformations to convert ephemeral data structures into persistent ones.
2. Strategies to convert particular data structures, such as lists and trees, into persistent data structures.
3. Design of algorithms using persistent data structures.

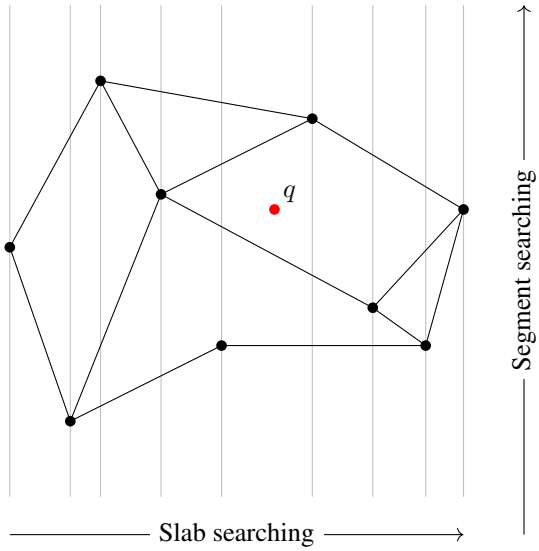
Now we discuss using persistent data structures in a computational geometry problem and two combinatorial optimization methods. Given a set $S$ of $n$ segments, representing a polygonal subdivision of the plane, the *planar point location problem* consists in preprocessing the set $S$ so that, given a sequence of points, the polygon containing each point can be determined quickly on-line (DOBKIN; LIPTON, 1976).

One can solve this problem as follows: we divide the space by drawing a vertical line to each extreme of each segment of $S$. We call the region between two consecutively vertical lines of *slab*. Then we use two searches to identify the polygon containing a point $q$. In the first search, we use the coordinate $x$ of $q$ to identify the vertical slab containing $q$. In the second one, we use the coordinate $y$ of $q$ to identify the segment directly over $q$ in the slab (see Figure 8).

If the slabs and segments of each slab are stored in binary search trees, where slabs are sorted from left to right, and the segments are sorted from bottom to top, the execution time for each query is $O(\log n)$. The execution time for preprocessing depends on the way that trees are constructed. If a separated binary search tree is constructed for each slab, then the preprocessing time in the worst case is $\Omega(n^2)$, since $\Omega(n)$ segments can intersect $\Omega(n)$ slabs (SARNAK; TARJAN, 1986).

It is important to note that consecutive slabs are different in only some segments. This difference can be either adding or removing some segments. To construct all slabs, each one based on the previous one (except for the first), it is necessary $2n$ insertion and removing operations. This scheme reduces the planar point location problem to an efficient persistent sorted set structure.

Figure 8 – Slab and segment search. First the slab containing $q$ is found, and then the segment directly over $q$.




Source: Elaborated by the author.

Sarnak and Tarjan (1986) describe a simple implementation of a persistent binary search tree that allows insertions and removals in amortized time $O(\log n)$, and it uses only $O(1)$ in amortized space. Through this implementation, we can get a structure to the planar point location problem, with $O(n)$ in space complexity, $O(n \log n)$ in preprocessing time, and queries in time $O(\log n)$. This technique is general enough to be applied to many other search problems in geometry (BOROUJERDI; MORET, 1995), which explains the large number of works in the literature. On the other hand, in optimization, they are yet uncommon, and we only found two works: (BATTITI, 2002) and (DELBEM; LIMA; TELLES, 2012).

Battiti (2002) proposes using persistent data structures in history-sensitive heuristic algorithms, which use information collected in previous phases of the algorithm in order to guide future searches. For example, in a Tabu Search, we could keep a list of some solutions previously found in order to prevent that they are again explored. In this context, Battiti uses a persistent set structure to store a collection of solutions (from a search space of binary strings of size $L$). Since a new solution is obtained from a current solution by inserting or removing an element, i.e., by modifying a bit, the persistent structure allows that a new solution uses only $O(1)$ in space. Battiti's method is optimum regarding space complexity because it just requires $O(L+t)$ to store all solutions generated in $t$ iterations.

Delbem, Lima and Telles (2012) developed a structure called NDDR, which aims speeding up the mutation operation in evolutionary algorithms for network-design problems. The result is that, although other equivalent structures need time $O(n)$, the authors

achieved an average time of $O(\sqrt{n})$ by using structural sharing, a technique used by many persistent structures. We discuss NDDR in details in Chapter 3, as well as present our proposal and compare both.

Our structure, like NDDR, is based on structural sharing. Therefore, we now discuss structural sharing, as well as other general methods to turn a structure into persistent.

### 2.4.1 Marking data structures persistent

Throughout this section, we consider that each time a structure is updated, it is labeled with a monotonically increasing time, which is used to access the structure version in that time.

The *fat node technique* (DRISCOLL *et al.*, 1989) associates the history of changes to each node of the structure. The old values are never deleted, so nodes can become arbitrary "fat". The history of each node is organized through a balanced search tree, where the structure version is used as the key. The additional cost for updating is $O(\log m)$, where $m$ is the number of stored versions, since all updates must be stored into the tree. The additional space for modifying is $O(1)$, which corresponds to the space to keep the new data. The access to each node has a multiplicative factor of $O(\log m)$.

The *path copying technique* (DRISCOLL *et al.*, 1989) makes a copy of a node before updating it, and then propagates the modification recursively to all nodes that referred the previous version of that modified node. The modification propagation stops when it reaches the root since no nodes refer to it. Root versions are stored into an array indexed by the structure version. Unchanged nodes are shared between the old and new version of the structure, which is called *structural sharing* in recent works, as Puente (2017).

Access has an additional cost of $O(\log m)$, because it requires to find the correct version of the root node. This cost is much smaller than the multiplicative factor of $O(\log m)$ required by the fat node technique. The time for updating and the extra space in the worst case is $O(n)$, since one update may require copying the entire structure. The path copying technique works well for balanced structures, for the number of nodes involved in modification is small.

Driscoll *et al.* (1989) describe a strategy to combine both the fat node and the path copying techniques, in order to acquire access in time $O(1)$ and modification in time and space $O(1)$. Their approach requires that all nodes be referred by a maximum constant number of vertices.

The most general technique to make an ephemeral data structure to become persistent is by simulating the computer memory so that each writing operation in the memory generates a version of the structure. Since the memory can be seen as an array, the techniques to make arrays to become persistent are notably important.

Dietz (1989) describes an efficient method to make an array persistent, whose idea is visualizing an array of size $n$ as a fat node with $n$ fields. The list of pairs version-value of each field is stored in a van Emde Boas tree (BOAS, 1975). This structure allows access to an element of the array in time $O(\log\log m)$, and modification in time $O(\log\log m)$, using space $O(m)$, where $m$ is the number of modifications.

Next, we describe a more recent and simple way to make an array persistent.

### 2.4.2 Persistent arrays

A *persistent array* gives two operations:

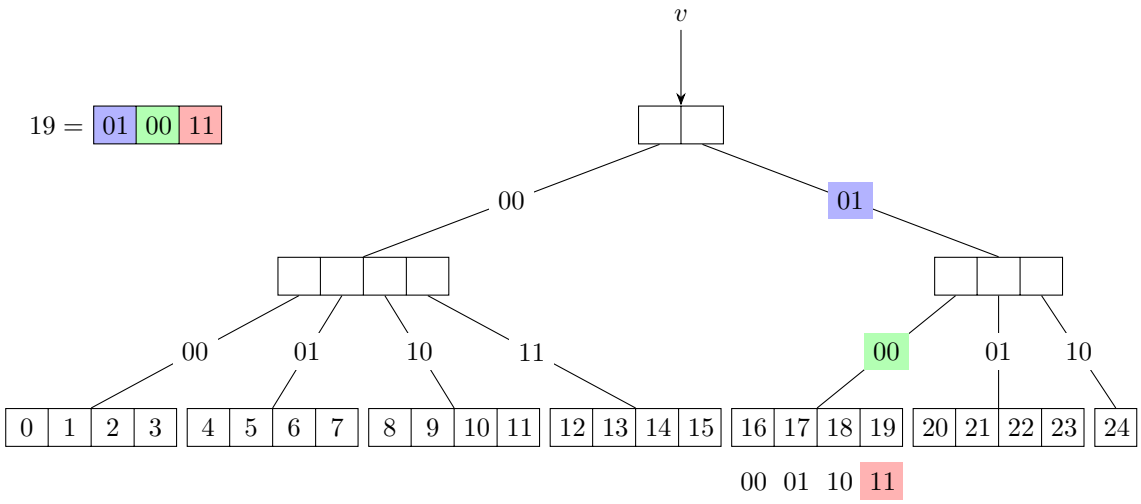
1. $\textsc{Get}(v,i)$: it returns the element in the position $i$ of the array $v$;
2. $\textsc{Assoc}(v,i,x)$: it returns a new array $v'$ which is similar to $v$, but with $x$ in the position $i$.

In this section, we describe a strategy to implement persistent arrays based in a trie, whose concept was conceived by Briandais (1959), and later named by Fredkin (1960). A *trie*, whose name derived from the word *retrieval*, in *information retrieval systems*, is a sorted tree, whose keys are strings of an alphabet. Each element of the alphabet corresponds to a branch in each node so that the position of a node in the tree defines which key is related to it. This way, the keys do not need to be stored in the tree. All descendants of a node have a common prefix of the key. The root represents an empty prefix.

If the keys are a sequence of bits (a string with only 0's and 1's), we have a *bitwise trie*. Many researchers studied the bitwise tries. Recently, the interest has increased because of Bagwell's works (BAGWELL, 2000; BAGWELL, 2001; BAGWELL, 2003).

We now describe how the operations Get, and Assoc of a persistent array can be implemented using bitwise tries. We assume that all keys have the same number of bits and, consequently, all leaves are at the same level. The procedure $\textsc{Get}(v,i)$ works as a common querying procedure in trees. Each bit of $i$ is used to reach a branch in a node of the tree. The last bit of $i$ indicates the position in the leaf which contains the value related to $i$ in $v$. The procedure $\textsc{Assoc}(v,i,x)$ is implemented by copying the path, i.e., the nodes of the path up to the leaf which contains the value related to $i$ are copied, and the value related to $i$ is modified to $x$. The new root is returned as a result of the procedure. The execution time of Get and Assoc is $O(h)$, where $h$ is the tree height. Since the ramification factor of the tree is 2, and all leaves are in the same level, the tree height is $\lceil \log_2 n \rceil$, where $n$ is the number of leaves (the array size). This way, the time of execution of Get and Assoc is $O(\log_2 n)$.

Figure 9 – Example of GET$(v, 19)$ for a persistent array $v$ with 2 branching bits. The key 19 is divided into 3 strings of 2 bits. The two most significant bits are used as indices in the root to determine the node of level 1. The next two bits are used as indices in the node of level 1 to determine the leaf node. The last two bits are used as indices in the leaf node to determine the value related to the key 19.




Source: Elaborated by the author.

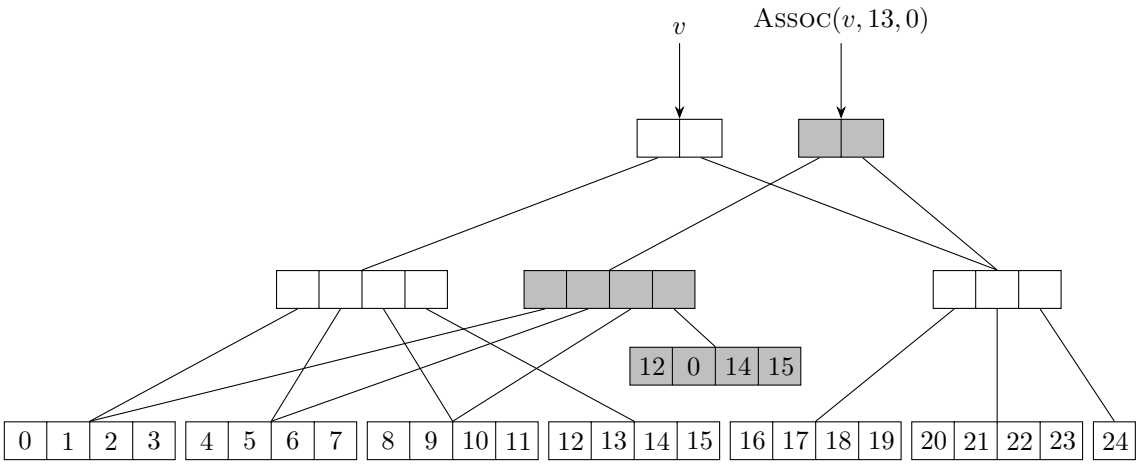
In practice, it is common to divide the bits of the keys into groups, increasing the ramification factor of the tree and decreasing its height. Moreover, this strategy improves the use of processor cache, since the values inside the nodes are grouped, which gives an advantage to references nearby.

When a key is divided into groups of $b$ bits, each leaf has at most $2^b$ branches. We refer to the value $b$ as *branching bits*. The operation of procedures GET and ASSOC when $b > 1$ is similar to that described previously, but instead of using a bit to choose the branch in a node, we use $b$ bits. In this case, we suppose that each key has $(h+1)b$ bits, where $h$ is the tree height. Figures 9 and 10 show examples of GET e ASSOC for a bitwise trie with height $h = 2$ and $b = 2$ branching bits.

For a value $b > 1$, the execution time for GET is $O(\log_{2^b} n)$, and for ASSOC is $O(2^b \log_{2^b} n)$ (the cost of copying $\log_{2^b} n$ nodes containing $2^b$ elements). Although the value of $b$ is constant related to $n$, it can be significant in practical applications. A small value for $b$ increases the tree height, which increases the time of GET and, consequently, disadvantages using cache, but also decreases the number of elements that should be copied in ASSOC. On the other hand, a large value for $b$ decreases the tree height, which decreases the time of GET and benefits using cache, but also increases the number of elements that must be copied in ASSOC.

A common value used in practical situations for the branching bits is 5. In this case, each node would have at most 32 branches. If we consider an address space of 40

Figure 10 – Example of Assoc$(v, 13, 0)$ to a persistent array $v$ with 2 branching bits. Each position $i$ of the array was initialized with the value $i$. The path from the root up to the leaf node to the key 13 is determined using the same idea of the procedure Get. A copy of each node in the path is created (highlighted nodes) and the value related to the key 13 in the copy of the leaf node is modified to 0. Notice that $v$ and Assoc$(v, 13, 0)$ share the most of the nodes.




Source: Elaborated by the author.

bits (1 Terabyte), the maximum tree height with $b = 5$ would be $\log_{2^5} 2^{40} = 8$. For this reason, some researchers consider Get and Assoc as having constant time when $b = 5$ (PUENTE, 2017).

CHAPTER

3

# DATA STRUCTURES FOR MUTATION-BASED EVOLUTIONARY ALGORITHMS

This chapter is a slightly modified copy of the paper "Data Structures for Direct Spanning Tree Representations in Mutation-based Evolutionary Algorithms", submitted to IEEE Transaction on Evolutionary Computation journal. The paper had the collaboration of Letícia Rodrigues Bueno, Assistant Professor at Federal University of ABC.

Optimization methods for spanning tree problems may require efficient data structures. The Node Depth Degree Representation (NDDR) has achieved relevant results for direct spanning tree representation together with evolutionary algorithms. Its two mutation operators have average time $O(\sqrt{n})$, where $n$ is the number of vertices of the graph, while similar operators implemented by predecessor arrays, a typical tree data structure, have time $O(n)$. Dynamic trees are also relevant when investigating tree representations since they have low time complexity, but there is no proper extension of them for evolutionary algorithms. Using aspects of both a dynamic tree and NDDR (Euler tours and structural sharing), we propose a data structure called 2LETT. The time of its mutation operators is $O(\sqrt{n})$ in the worst case. Experiments with the mutation operators using 2LETT, predecessor arrays, and NDDR are carried out for graphs with up to $300,000$ vertices. For the mutation operator that exchanges any two valid edges, the predecessor array presents the better performance for random trees with less than $10,000$ vertices; while 2LETT has the best performance for trees with more than $10,000$ vertices.

## 3.1 Introduction

A *spanning tree* of an undirected connected graph $G$ is an acyclic connected subgraph of $G$ that includes all vertices of $G$. *Network design problems* (KORTE; VYGEN, 2008) are a broad class of optimization problems, many of them requiring to find spanning trees. The *minimum spanning tree problem* is a well-known network design problem. Although it is solved in polynomial time by greedy algorithms (KRUSKAL, 1956; PRIM, 1957), simple variants of it are computationally hard as, for example, if we add constraints on the vertex degree, on the diameter, or on the number of leafs (GAREY; JOHNSON, 1990). These problems are related to many applications in areas such as distribution network reconfiguration, transportation, and phylogenetics, therefore there is great interest in efficient methods to solve them.

Several real-world problems involve large-scale networks, but exact algorithms can solve only small instances of NP-Hard problems. As a consequence, heuristic methods as evolutionary algorithms (EAs) have been investigated to deal with larger instances of network design problems.

Two important design aspects of EAs applied to large-scale networks are the representation of the individuals and the search operators. The representation can be *direct* or *indirect*. In the former, the individuals directly represent the values of variables of a candidate solution for the problem (decision space). In the latter, the individuals are usually encoded as a string of values (genotype), and a mapping function is used to convert to/from solutions in the decision space (phenotype).

Rothlauf (2006) analyzes some trade-offs in the design of direct and indirect representations of EAs. There are many general indirect representations and associated search operators, where the main concern in designing them is to define a genotype to phenotype mapping function. On the other hand, there are no general search operators for direct representations, since different problem domains require different representations. As a consequence, the main concern to design them is the definition of specific search operators.

Li (2001) highlights that implementing direct representations is as important as designing them. Once the search operators are defined, it still remains to choose, among many options, a data structure to implement them, which can be challenging and have a huge impact in the performance of an EA.

Consider two mutation operators, namely CHANGE-ANY and CHANGE-PRED, which are generalizations of mutation operators proposed in (DELBEM; LIMA; TELLES, 2012). Both methods execute simple tasks in trees: CHANGE-ANY returns a tree from exchanging an edge by another one that is not in the given tree, and CHANGE-PRED returns a tree from modifying the predecessor of a vertex in a given rooted tree.

These operators can be implemented in time $O(n)$, where $n$ is the number of vertices of the tree, using predecessor array, a common tree representation used by many graph algorithms. However, in the data structure named "node-depth-degree representation" (NDDR for short) proposed in (DELBEM; LIMA; TELLES, 2012), they can have average time $O(\sqrt{n})$. One aspect of NDDR that enables the efficient implementation of these operations are the decomposition of a tree in substructures, which can be shared by parent and offspring trees, when mutation is applied.

The basic operations of CHANGE-ANY and CHANGE-PRED can also be implemented in sublinear time using dynamic tree data structures (TARJAN; WERNECK, 2007), however, as far we know, they were not yet adapted to efficiently work with a population of trees, as required for EAs.

In this paper we focus on designing and comparing efficient data structures for implementing CHANGE-ANY and CHANGE-PRED operators. Inspired by a dynamic tree data structure and by NDDR, we propose a new data structure, called 2LETT. In this structure, we represent trees by Euler tours and store them in two-level arrays, which results in CHANGE-PRED and CHANGE-ANY having worst-case time $O(\sqrt{n})$. By using two-level arrays, substructures are shared by parent and offspring trees when mutation is applied, as NDDR does (DELBEM; LIMA; TELLES, 2012). Also, we show an implementation of predecessor arrays using two-level arrays instead of linear arrays, which allows CHANGE-PRED to have average time $O(\sqrt{n})$.

We compare the performance of three data structures: predecessor arrays, NDDR and 2LETT for graphs with up to $300,000$ vertices. The comparison points out that 2LETT is the most efficient for trees with more than $10,000$ vertices, while predecessor arrays are the most efficient for random trees with less than $10,000$ vertices. We also show that the running time of 2LETT, in contrast to the others, does not rely on tree diameter, which makes it useful in handling trees with larger diameters.

The paper is organized as follows. Section 3.2 reviews the search operators for direct spanning tree representations, formally defines CHANGE-ANY, and CHANGE-PRED and analyzes some important implementation aspects. Section 3.3 describes predecessor arrays and a way to improve them by using two-level arrays. Section 3.4 briefly reviews NDDR data structure and discuss its innovative aspects and limitations. Section 3.5 proposes the 2LETT data structure. Section 3.6 makes a qualitative evaluation of the structures. Section 3.7 presents a comparison of computational results: *i*) the efficiency of CHANGE-ANY and CHANGE-PRED implemented for the three data structures and *ii*) the performances of three implementations of an EA, one for each structure, solving the one-max tree problem. Finally, Section 3.8 presents our conclusions.

## 3.2 Search operators for direct representations

We start with some definitions. Let $T = (V_T, E_T)$ be a spanning tree of a connected graph $G = (V, E)$, where $V$ and $V_T$ are the vertex sets, and $E$ and $E_T$ are the edge sets. An edge is an unordered pair, which we denote by $(u, v)$, where both $u$ and $v$ are in the same vertex set. Also, consider $r$ a special vertex of $V_T$, called the *root* of $T$.

We denote by $\overline{T}$ the complement of $T$ in $G$, i.e., $\overline{T} = (V_T, E \setminus E_T)$. Given two distinct vertices $u, v \in V$, we denote the unique path between $u$ and $v$ in $T$ by $uTv$. For any vertex $v \in V$, the *depth* of $v \in T$, denoted by $d(v)$, is the number of edges along the path $vTr$. Consequently, $d(r) = 0$.

For any edge $(u, w)$ in $vTr$, we say $u$ is the *predecessor* or *parent* of $w$, denoted by $p(w)$, if $u$ is closer to $r$ than $w$. In this case, we also say $w$ is a child of $u$. The vertices that are reachable from a vertex $u$ by repeatedly going from child to parent are called *ancestors* of $u$. Similarly, the vertices reached from $u$ by repeatedly going from parent to child are known as *descendants* of $u$. A *v-subtree* is a subtree rooted in $v$ and contains all the descendants of $v$ in $T$.

Although there are many indirect spanning tree representations (see the studies in (RAIDL; JULSTROM, 2003; ROTHLAUF, 2006), and a comparison in (CARRANO *et al.*, 2007; SOAK; JEON, 2010)), there are only a few direct representations such as LI; BOUCHEBABA's one (LI; BOUCHEBABA, 1999; LI, 2001), Edge sets (RAIDL; JULSTROM, 2003), NetDir (ROTHLAUF, 2006), and NDDR (DELBEM; LIMA; TELLES, 2012).

Let us present and compare now the search operators defined for these direct representations. The mutation operator is basically the same for all of them: for a tree $T$, remove an edge from $T$ and add an edge to $T$ chosen from $\overline{T}$ in such a way that the resulting structure is still a tree or, more explicitly, in a way that does not create cycles. This operation is called *edge exchange*.

Li and Bouchebaba (1999) propose a version of edge exchange where a path or a subtree is added to the tree, instead of a single edge, which can be seen as a sequence of edge exchanges of some edges selected beforehand. Raidl and Julstrom (2003) propose an heuristic version of edge exchange for Edge sets, in which edges with small weights have preference to be added. NDDR (DELBEM; LIMA; TELLES, 2012), in its turn, has defined a version of edge exchange that changes only the predecessor of a vertex.

Regarding the crossover operator, except for NDDR which does not use one, its takes two trees (the parents) from the population and generates a new tree (an offspring) combining the parents. The crossover of Li and Bouchebaba (1999) is basically a sequence of edge exchanges that add some edges taken from another tree. On the other hand, the crossover of Edge sets (RAIDL; JULSTROM, 2003) is more complex and defined by the

following three steps:

(i) it adds all common edges with both parent trees to the new tree;

(ii) progressively it adds the remaining edges to the new tree, if they do not create cycles;

(iii) if the new tree is not still connected and spanning, then it tries to add edges that are not in any of the parent trees.

In addition, the authors propose a heuristic version which gives preference to edges with small weights in the steps (ii) and (iii).

Rothlauf (2006) also defines for NetDir an elaborated crossover as follows:

(i) it divides arbitrarily the vertices of the graph in two sets $V_1$ and $V_2$;

(ii) it chooses arbitrarily a parent tree, and adds all of its edges with both vertices in $V_1$ to the new tree;

(iii) it adds all the edges with both vertices in $V_2$ from other parent tree. Notice that the result so far is a disconnected tree;

(iv) finally, it tries to randomly add edges from the parent trees until the new tree is spanning and connected.

We now discuss the data structures used in the implementation of these search operators. For Edge sets, Raidl and Julstrom (2003) note that the edges can be stored into an array or a hash table, and highlight that the latter allows insertion, deletion, and lookup of individual edges in constant time. They also suggest the representation of a tree through an adjacent list in order to implement the mutation operator in linear time. Rothlauf (2006) does not discuss any data structure aspect for NetDir, nevertheless, the mutation operator can be implemented in the same way as for Edge sets. The crossover operator of both NetDir and Edges sets can be implemented in linear time using the data structure union-find, making union by rank and path compression (CORMEN *et al.*, 2009, p. 561 – 581).

Li and Bouchebaba (1999) use adjacent list to implement the main operations of the search operators in time $O(n^2)$. Later, Li (LI, 2001) described how to reduce the time complexity of the same main operations to linear time using a predecessor array.

Delbem, Lima and Telles (2012) designed a specific data structure for NDDR data structure, which we will also refer to as NDDR. Their implementation of the search operators has average time of $O(\sqrt{n})$, which is a significant asymptotic improvement over linear time implementations.

Notice that, even though the mutation operators of the representations are based on edge exchanges, their authors use different data structures to implement them. Although only Li and Bouchebaba (1999) performed a comparison between two of them, an experimental comparison can enable us to understand better which data structure is more recommended according to specific conditions. Notwithstanding a better asymptotic time than others have, an algorithm can have a poorer practical performance for instances with specific characteristics and sizes.

In order to present an experimental comparison, we start by defining the mutation operators that we compare in the context of different data structures in Section 3.7.

### 3.2.1 CHANGE-ANY *and* CHANGE-PRED

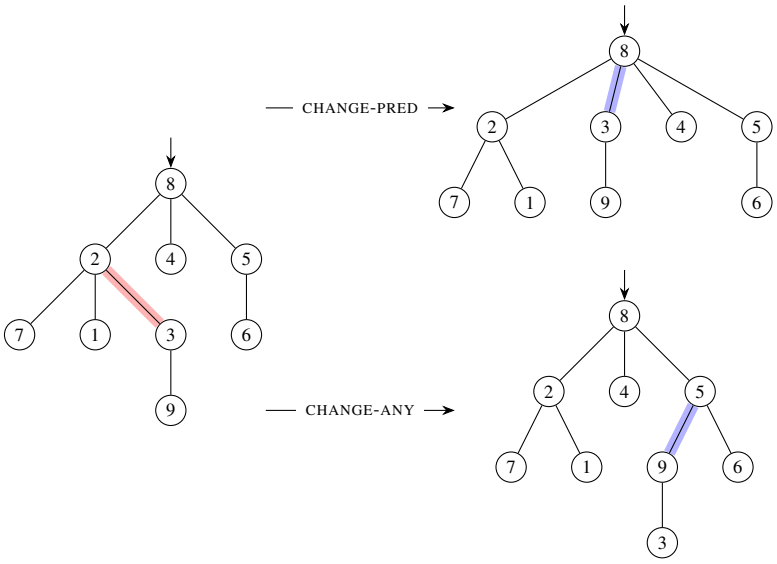
Given a rooted spanning tree $T$ of a graph $G$, the mutation operator CHANGE-ANY generates a new tree from $T$ by removing one of its edges and adding an edge from $\overline{T}$. That is, it receives $T$ as input and returns a triple $(T', e, f)$ where $e \in E_{\overline{T}}$ is the edge to insert, $f \in E_T$ is the edge to remove, and $T' = (V_T, E_{T'})$ is the resulting spanning tree, where $E_{T'} = (E_T \setminus \{f\} \cup \{e\})$. The mutation operator CHANGE-PRED generates a new tree from an existing one by changing the predecessor of a vertex, so it is a restricted version of CHANGE-ANY. If CHANGE-PRED is applied over an edge $f = (v, p(v))$, then there must be an edge $e = (v, p'(v)) \in E_{T'}$, where $p'(v)$ is the predecessor of $v$ in the new tree $T'$, and $p(v) \neq p'(v)$. These definitions are generic enough so that each data structure can use the best approach to select the edges to insert and remove. Figure 11 shows an example of CHANGE-ANY and CHANGE-PRED.

Now we highlight two important aspects for the implementation of CHANGE-ANY and CHANGE-PRED and give an overview of some approaches to tackle them.

1. **Preserving the parent tree:** considering that the parent tree and/or its offspring tree can both be in the population, the parent should not be changed. This reverberates in the mutation operators, since they are required to return new trees. The most common way to generate the offspring trees and preserve the parent is to copy it before applying the mutation operator. However, this has linear time so it can be unacceptable when the copying operation dominates the mutation time.

   The vast literature of persistent data structures can help on this. A data structure is persistent if each operation generates a new version of the structure, allowing the old and new versions to coexist (KAPLAN, 2004). There are general approaches to make ephemeral (non-persistent) data structures to become persistent, and NDDR uses one of them: *structural sharing*. The idea is to decompose a structure in substructures in such a way that, when an operator is applied, most of the substructures can be shared between the old and the new version of the structure.

Figure 11 – CHANGE-ANY and CHANGE-PRED examples. The predecessor of the vertex 3 is modified by CHANGE-PRED, which replaces the edge $(3,2)$ by $(3,8)$. CHANGE-ANY replaces the edge $(3,2)$ by $(9,5)$. Notice that while in CHANGE-PRED always a single predecessor is changed, many can be changed in CHANGE-ANY.




Source: Elaborated by the author.

To the best of our knowledge, at the present time, NDDR is the only spanning tree representation using structural sharing to reduce the mutation time.

2. **Selecting edges to insert and remove, and ensuring a valid new tree:** if an edge $(u,v)$ is chosen to be inserted before selecting an edge to remove, then the latter must be selected from the path $uTv$. If the enumeration of the edges in the path is necessary, then this approach has time of $O(n)$, since the path may contain all the edges of the tree. Nevertheless, this can be efficient for trees with small diameters. Differently from CHANGE-ANY, CHANGE-PRED can avoid the enumeration of edges for graphs with general diameter since only $(u,p(u))$ or $(v,p(v))$ can be removed. On the other hand, if the edge $(u,v)$ is chosen to be removed before selecting an edge to add, this one must reconnect tree $T' = (V_T, E_T \setminus (u,v))$. If this is done by verifying all edges in $\overline{T}$, then the time is $O(m)$, where $m$ is the number of edges in the graph. Therefore, an efficient approach should optimize this last step.

   The same decomposition used by NDDR for structural sharing (that deals with the first issue of preserving the parent) also enables an efficient solution for the second issue of selecting edges to insert and remove and generate a valid new tree. NDDR first selects the edge to add, for example $(u,v)$, and then selects the edge to remove by enumerating part of the edges along the path $uTv$. All these steps are bounded by the mean size of the decomposed substructures, which is $O(\sqrt{n})$. See Section 3.4 for further discussions about this.

An alternative approach to the one used by NDDR is the use of dynamic tree data structures. A *dynamic tree data structure* maintains spanning trees that change overtime through edges insertions and deletions, and answers connectivity queries (TARJAN; WERNECK, 2007). Some dynamic tree data structures support these operations in time $O(\log n)$, though they are not designed to solve the parent copy issue. Thus, in this case they need to be adapted.

Section 3.5 presents our proposal, which is based on a dynamic tree data structure and structural sharing, and it has time $O(\sqrt{n})$ for the mutation operators in the worst case.

## 3.3 Predecessor Array

When using a predecessor array, a tree $T$ with root $r$ is represented by associating its vertices to their predecessors. In other words, we store the edges $(v, p(v))$ into an array $\forall v \in V_T$, where $v \neq r$. The NIL value is used to represent the absence of value for the predecessor of $r$. If the vertices are arbitrarily numbered $1, 2, \ldots n$, then the predecessors can be stored into an array of size $n$, where each predecessor $p(v)$ is stored in the index $v$ and accessed in constant time.

Although the predecessor array is a widely used tree representation in graph algorithms in general, it is apparently not so popular for EAs, having only a few works such as (RAIDL; DREXEL, 2000; LI, 2001) in direct spanning tree representation. Moreover, some works (PALMER; KERSHENBAUM, 1994; KRISHNAMOORTHY; ERNST; SHARAIHA, 2001) conclude that it is not an adequate indirect representation because traditional operators, such as one-point and two-point crossovers, can generate invalid offspring.

Let $A$ be a predecessor array representing a spanning tree $T$. The interface to use $A$ consist in three procedures: IS-ANCESTOR, SET-PRED, and MAKE-ROOT. For two vertices $u$ and $v$ in $V_T$, IS-ANCESTOR$(A, u, v)$ returns TRUE if $u$ is ancestor of $v$, and FALSE otherwise, which is done by verifying if $u$ is in the path $vTr$. SET-PRED$(A, u, v)$ simply sets the predecessor of $u$ as $v$. MAKE-ROOT$(A, u)$ makes $u$ the root of $T$, which is done by reversing the predecessors along the path $uTr$.

The time of IS-ANCESTOR and MAKE-ROOT is $O(h)$, where $h$ is the tree height. In the worst case, when $h = n$, the time is $O(n)$. However, since the expected height (and diameter) of a random tree is $O(\sqrt{n})$ (RéNYI; SZEKERES, 1967; SZEKERES, 1983), the average time is $O(\sqrt{n})$. Moreover, the time of SET-PRED is $O(1)$.

Next we describe how CHANGE-PRED and CHANGE-ANY can be implemented, and we show how the use of structural sharing enables the average time of CHANGE-PRED to

be $O(\sqrt{n})$.

### 3.3.1 Implementing CHANGE-ANY and CHANGE-PRED

Once we select a random edge $(u,v)$ from $\overline{T}$ to CHANGE-PRED to add into $T$, the edge to be removed from the tree must either be $(u,p(u))$, or $(v,p(v))$, that is, the predecessor of either $u$ or $v$ must be modified while preventing creation of cycles. If $u$ is an ancestor of $v$, then we change the predecessor of $v$ by SET-PRED($T$, $v$, $u$). Similarly, if $v$ is an ancestor of $u$, then we change the predecessor of $u$ by SET-PRED($T$, $u$, $v$). If $u$ is not an ancestor of $v$ nor is $v$ an ancestor of $u$, then we choose with same probability between the two previous cases.

In CHANGE-ANY, we select a random edge $(u,v)$ from $\overline{T}$ to add into $T$, and we select an edge to remove from the path $uTv$. In order to simplify the enumeration of the edges in $vTu$, we make $u$ as the root of the tree using MAKE-ROOT($T$, $u$). Afterward, we find the path from $v$ to $u$ following the chain of predecessors starting in $v$ up to $u$, and we select a random edge $(x,p(x))$ from this path to be removed, where $x \neq u$. Finally, we add edge $(u,v)$ by SET-PRED($T$, $u$, $v$) and we remove edge $(x,p(x))$ by SET-PRED($T$, $x$, NIL), making $x$ the new root of the tree.

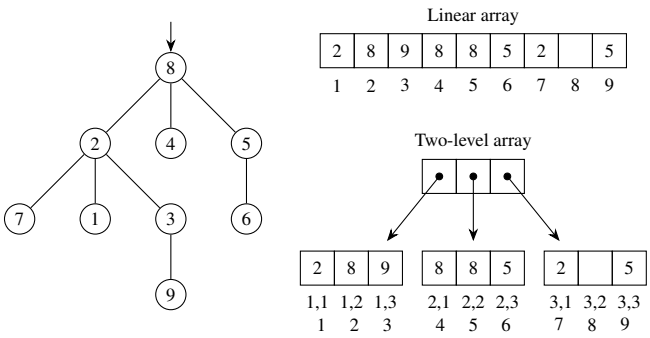
One could argue that the time of CHANGE-PRED and CHANGE-ANY is $O(h)$, but since they need to perform a copy of the input array to return a new tree, we get $\Omega(n)$. Comparing with NDDR's average time of $O(\sqrt{n})$, it may seem that predecessor arrays are not a good choice but, as we shall see in Section 3.7, it is indeed competitive for quite large random graphs. This comes from the fact that predecessor arrays are a simple and compact data structure and its implementation has small constant factors. Besides, predecessor arrays are a good baseline when comparing to more advanced data structures such as NDDR and our proposed structure 2LETT.

### 3.3.2 Improving CHANGE-PRED

We could improve the average time of CHANGE-PRED by using structural sharing: instead of storing the predecessors in a linear array, we store them in a two-level array. Let $A$ be an array with $n$ elements and let $s$ be the split factor, for $1 < s < n$. Suppose, without loss of generality, that $n$ is a multiple of $s$. We split $A$ in $\frac{n}{s}$ subarrays with $s$ consecutive elements each, and store the pointers to the subarrays in a two-level array $B$. The first level of $B$ stores the pointer to the subarrays, and the second level stores the content of the subarrays. This way, for each index $i$, the element $A[i]$ is stored in $B[\lceil \frac{i}{s} \rceil][i - s(\lceil \frac{i}{s} \rceil - 1)]$. We call this scheme of storing predecessors in a two-level array as 2LPredecessor. Figure 12 shows how predecessors are stored in linear and two-level arrays.

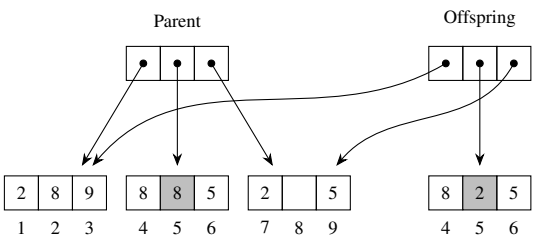
Using structural sharing, the unchanged subarrays can be shared between the

Figure 12 – Predecessors stored in linear and two-level arrays. The split factor for the two-level array is 3. The predecessor of each vertex is stored in the corresponding position, for example, the predecessor of 7 is store in the position 7 in the linear array and in the position $(3,1)$ in the two-level array.




Source: Elaborated by the author.

Figure 13 – Subarray sharing in CHANGE-PRED of 2LPredecessor. The offspring is created by changing the predecessor of the vertex 5 from 8 to 2. Only one new subarray was created to accommodate the modification, while the other subarrays are shared between the parent and the offspring.




Source: Elaborated by the author.

input and output tree of CHANGE-PRED. To do so we adapt CHANGE-PRED and SET-PRED. Before executing CHANGE-PRED, we create a copy of the first level of the predecessor two-level array, which makes all subarrays to be shared between the input and output trees. This operation has time of $O(\frac{n}{s})$. As the subarrays are shared, every time a predecessor is changed we need to create a copy of the subarray that stores that predecessor, and make the modification in this copy. This makes the time of SET-PRED to be $O(s)$ instead of $O(1)$. CHANGE-PRED only call SET-PRED once, so the time of CHANGE-PRED is $O(\frac{n}{s}+s+h)$, where $h$ is the cost of IS-ANCESTOR calls. If $s = O(\sqrt{n})$, then $O(\frac{n}{s}+s+h) = O(\sqrt{n}+h)$. Considering $h = O(\sqrt{n})$ for random trees, the average time of CHANGE-PRED is $O(\sqrt{n})$. Notice that the two-level arrays do not improve the average time of CHANGE-ANY, because CHANGE-ANY uses MAKE-ROOT, which in turn makes many calls to SET-PRED. Figure 13 shows an example of the subarray sharing.

When implementing, we have chosen $s$ as the smallest power of 2 greater than $\sqrt{n}$. This allows us replacing the division in the index calculation of the two-level array with

bit shifting operations, which is much faster.

## 3.4 NDDR

*Node-depth-degree* (NDDR) is a spanning forest representation and data structure based on the concepts of paths and depth (DELBEM; LIMA; TELLES, 2012), and it is an improvement of the NDE representation (DELBEM *et al.*, 2004). Both representations were successfully used in evolutionary algorithms applied to many problems (LIBRALAO *et al.*, 2005; LIMA; ROTHLAUF; DELBEM, 2008; MANSOUR *et al.*, 2010; SANTOS; DELBEM; BRETAS, 2008; DELBEM; LIMA; TELLES, 2012). NDDR was designed to represent not only spanning trees, but also spanning forests, although a version of NDDR suitable only for spanning trees with linear mutation time was recently proposed in (LIMA *et al.*, 2016). In fact, modeling a spanning tree as a spanning forest is the key to achieve sublinear mutation time for spanning trees.

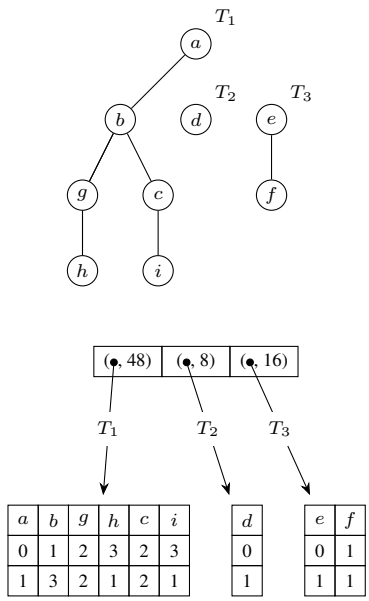
Now we describe how a forest is represented in NDDR and how the mutation operators work. Next we explain the structure to represent spanning trees, and the implementation of CHANGE-ANY and CHANGE-PRED for it. We also point some limitations of the structure.

Given a spanning forest $F$ from a graph $G = (V, E)$, each tree $T \in F$ is rooted at an arbitrary vertex and represented independently by an array of triples: vertex, depth, and degree (NDD). A NDD array has two properties: (i) the vertices in every subtree are consecutive, and (ii) the root of every subtree precedes the subtree vertices. A NDD array can be obtained by a depth-first search (CORMEN *et al.*, 2009), which adds each vertex and its properties to the end of an array the first time the vertex is visited. Forest $F$ is therefore represented by an array of pairs with each pair $(p_T, deg_G(T))$ corresponding to a tree $T$ of $F$, where $p_T$ is a pointer to the NDD array of $T$, and $deg_G(T)$ is the amount of edges of $G$ incident to vertices of $T$. Although we no longer refer to the depth and degree values of the NDD array and to $deg_G(T)$, these values are important to ensure the improved time of the search operators. Figure 14 shows an example of a forest from a graph represented by NDDR.

### 3.4.1 OP1 *and* OP2

Two mutation operators are defined for NDDR, namely OP1 and OP2. OP1 requires four operands: a tree $T_{from}$, a vertex $u$ of $T_{from}$ different from the root, a tree $T_{to}$, and a vertex $v$ of $T_{to}$. The trees $T_{from}$ and $T_{to}$ can be the same, only respecting that $u$ is not an ancestor of $v$ in order to prevent cycles. OP1 extracts the $u$-subtree from $T_{from}$ and inserts it as a subtree of $v$ in $T_{to}$. This operation replaces the edge $(u, p(u))$ by $(u, v)$.

Figure 14 – A forest of the complete graph with 9 vertices represented by NDDR. The forest has three trees: $T_1$, $T_2$, $T_3$. Each tree is stored into an NDD array, which is drawn as a matrix with each column representing a triple vertex, depth, and degree.




Source: Elaborated by the author.

Similarly, Op2 requires the same operands as Op1 plus an vertex $w$ that is different from the root and is ancestor of $u$. The Op2 extracts the $w$-subtree of $T_{from}$, makes $u$ as its root, and inserts it as a subtree of $v$ in $T_{to}$. This operation replaces the edge $(w, p(w))$ by $(u, v)$. Notice that, when $u = w$, Op2 does the same as Op1. See (DELBEM; LIMA; TELLES, 2012) for details regarding the creation of NDD arrays for the new trees.

The operands for Op1 are found in three steps:

(i) first, the tree $T_{from}$ and the vertex $u$ from $T_{from}$ are randomly selected, respecting that $u$ is not the root of $T_{from}$ and has at least an incident edge not in $F$;

(ii) an edge $(u, v)$ not in $F$ is randomly selected;

(iii) finally, starting in $v$, the tree $T_{to}$ is determined.

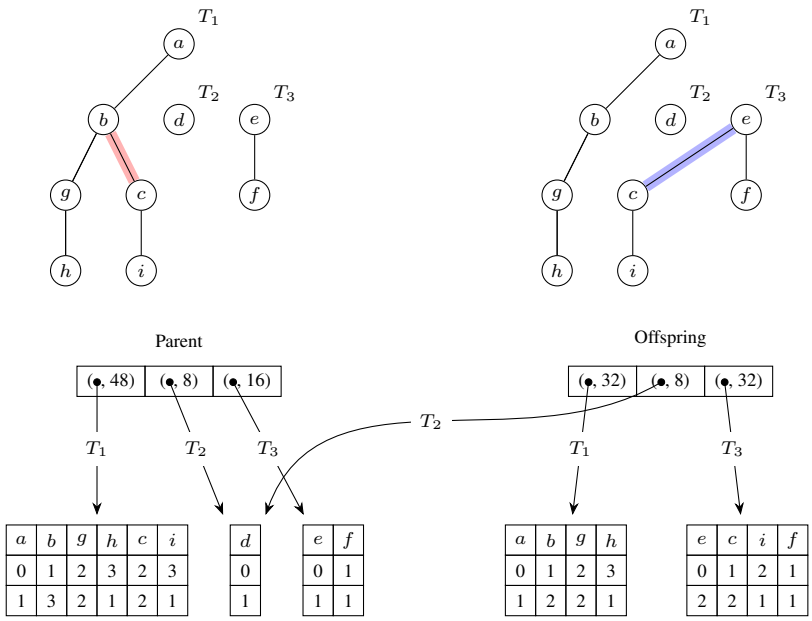
The operands for Op2 are determined likewise:

(i) first, the tree $T_{from}$ and a vertex $u$ from $T_{from}$ are randomly selected, respecting that $u$ is not the root of $T_{from}$ and has at least an incident edge not in $F$;

(ii) an arbitrary vertex $w$, different from the root of $T_{from}$, is selected among the vertices in the path from $u$ to the root of $T_{from}$;

(iii) an edge $(u, v)$ not in $F$ is randomly selected, and from $v$, the tree $T_{to}$ is determined.

Figure 15 – Subtree sharing in NDDR. At most two trees are modified by mutations, while the other trees can be shared by parent and offspring. Here, the offspring is created by replacing $(b,c)$ by $(c,e)$.




Source: Elaborated by the author.

The time of Op1 and Op2 is composed by determining the operands and creating the new forest. Delbem, Lima and Telles (2012) describe an implementation to determine the operands in time $O(t+|T_{from}|)$, where $t$ is the number of trees in $F$. The only trees changed by Op1 and Op2 are $T_{from}$ and $T_{to}$, so the others can be shared by $F$ and the new forest (see Figure 15). Thus, the time to create the new forest is $O(t+|T_{from}|+|T_{to}|)$, since it takes $O(|T_{from}|+|T_{to}|)$ to copy and modify $T_{from}$ and $T_{to}$, and it takes $O(t)$ to copy the pointers to the shared trees. Therefore, the total time of Op1 and Op2 is $O(t+|T_{from}|+|T_{to}|)$. If the $n$ vertices of the graph are uniformly distributed in the $t$ subtrees, then $O(t+|T_{from}|+|T_{to}|) = O(t+\frac{2n}{t})$, which has the minimum value of $O(\sqrt{n})$ when $t = \lceil\sqrt{n}\rceil$.

Although the requirement that the vertices of the graph should be uniformly distributed into the subtrees may seem too strong, Delbem, Lima and Telles (2012) show that, if $t = \lceil\sqrt{n}\rceil$, the average size of $T_{from}+T_{to}$ is $O(\sqrt{n})$ after many applications of Op1 and Op2.On the other hand, this requires that the probability of selecting any pair of subtrees as $T_{from}$ and $T_{to}$ should be the same, and NDDR selection method does not assure that. We discuss this in the end of the section.

In order to represent general forests in NDDR, Delbem, Lima and Telles (2012) show how to decompose any forest into a forest with $O(\sqrt{n})$ trees. We now describe this process for forests with one tree.

### 3.4.2 One-tree forests

Let $T = (V_T, E_T)$ be a spanning tree of a graph $G = (V, E)$. We decompose $T$ in $\lceil\sqrt{n}\rceil + 1$ parts as follows:

(i) One connected subgraph of $T$ with $\lceil\sqrt{n}\rceil$ vertices:

$$T^* = (V_T^*, E_T^*)$$

(ii) $\lceil\sqrt{n}\rceil$ connected subgraphs (the subtrees) of $(V_T, E_T \setminus E_T^*)$:

$$\begin{aligned} T_1 &= (V_{T_1}, E_{T_1}) \\ T_2 &= (V_{T_2}, E_{T_2}) \\ &\dots \\ T_{\lceil\sqrt{n}\rceil} &= (V_{T_{\lceil\sqrt{n}\rceil}}, E_{T_{\lceil\sqrt{n}\rceil}}) \end{aligned}$$

The tree $T$ is represented by a pair $(S, F)$, where $S$ is a NDD array representing $T^*$, and $F$ is a NDDR structure representing $T_1, T_2, \dots T_{\lceil\sqrt{n}\rceil}$, with each vertex of $T^*$ being the root of one of the trees $T_1, T_2, \dots, T_{\lceil\sqrt{n}\rceil}$.

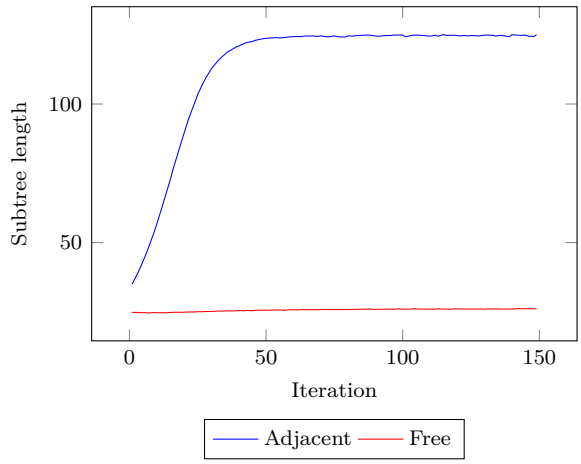
Both Op1 and Op2 are modified to work with this representation. The original procedure Op1 (as well as Op2) is either applied to the forest $F$, choosing $T_{from}$ and $T_{to}$ from $\{T_1, T_2, \dots T_{\lceil\sqrt{n}\rceil}\}$, or to $T^*$, choosing $T_{from} = T_{to} = T^*$. This version for forests with one tree keeps the same average time. Notice that this version of Op1 is exactly the procedure CHANGE-PRED, and this version of Op2 is the procedure CHANGE-ANY.

### 3.4.3 Limitations

Although these versions of Op1 and Op2 have the same average time as the original ones, they have some intrinsic limitations. One of them is that some exchanges of edges cannot be done, specifically the edges that connect vertices in $T^*$ cannot be exchanged with other edges, which in turn implies that the search may not find some trees. Another limitation regards the decomposition in balanced trees, making it impracticable for some trees such as, for example, a path. In this case, one or two subtrees would have most of the vertices, which compromises the time of the search operators. We now describe in details this issue and show some experimental results that highlight its implications.

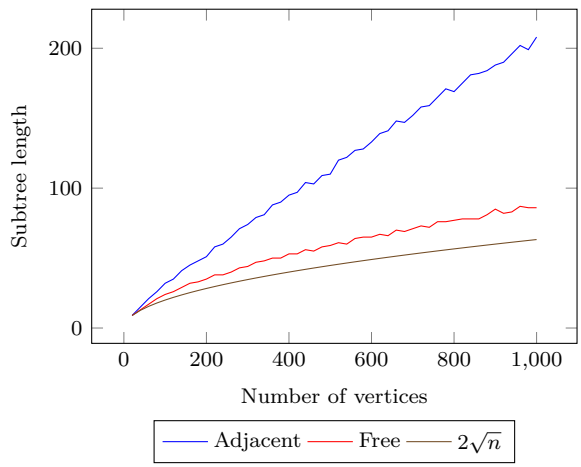
Suppose now, without loss of generality, that $G$ is a complete graph. After selecting $T_{from}$ and $u$, an edge $(u, v)$ is chosen. There are edges from $u$ to all vertices of $G$, and the probability of choosing any edge is the same, so the subtrees with more vertices have more chance of containing vertex $v$ and, consequently, they are more likely to be selected as $T_{to}$. This bias leads to a situation in which most of the vertices are in a single subtree.

Figure 16 – Increase of sum of selected subtree sizes for Op2. Initially, Op2 is applied to a random spanning tree of a complete graph with 100 vertices, and then applied successively to the resulting tree. The abscissa is the mutation sequence number. The ordinate is the sum of sizes of the subtrees $T_{from}$ and $T_{to}$. Free NDDR converges near to $2\sqrt{100}$, while adjacent NDDR converges approximately to 120.




Source: Elaborated by the author.

Figure 17 – Sum of selected subtrees sizes in the first execution of Op2 for a random tree. The abscissa refers to $n$, and the ordinate to the sum of sizes of $T_{from}$ and $T_{to}$. The growing rate of the subtree length of free NDDR is closer to $2\sqrt{n}$ than that of adjacent NDDR.

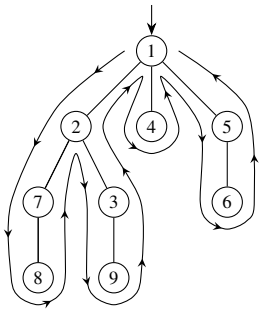



Source: Elaborated by the author.

Figures 16 and 17 illustrate it by comparing the NDDR's behaviour using its original selection method (here named "adjacent") with an unbiased strategy (that we called "free"), described as follows: first select $T_{from}$ and $u$, and then independently select $T_{to}$ and $v$ (since $G$ is complete, $(u,v)$ has to be in $G$). As seen in the figures, the sum of sizes of the selected subtrees by the adjacent strategy grow faster than $O(\sqrt{n})$.

Although the bias of adjacent selection strategy was highlighted by a sequence of random operations, an objective function rewarding forests with balanced subtrees could

Figure 18 – Euler tour of a tree. The tour starts and ends in vertex 1, the tree root. For simplicity, each edge $(u,v)$ is shown here as $uv$.




$\langle 12, 27, 78, 87, 72, 23, 39, 93, 32, 21, 14, 41, 15, 56, 65, 51 \rangle$

Source: Elaborated by the author.

overcome the bias, making the adjacent strategy viable.

## 3.5 2LETT

An *Euler tour* of a graph is a trail that visits every edge exactly once. To represent a tree using an Euler tour, we consider edge $(u,v)$ distinct of edge $(v,u)$, so a tree is represented by an Euler tour that starts and ends in the root, and visits all of its edges (see an example in Figure 18). The representation of trees by Euler tours was first used in parallel graph algorithms and, later, in the implementation of a dynamic tree data structure (HENZINGER; KING, 1995; TARJAN, 1997).

A *dynamic tree* is an abstract data type that maintains a forest that changes through insertions and deletions of edges. A dynamic tree supports many operations, such as:

(i) LINK($F$, $u$, $v$): given a forest $F$ and two vertices $u$ and $v$, it joins two trees containing the vertices $u$ and $v$ by adding edge $(u,v)$ to $F$, assuming $u$ and $v$ in distinct trees.

(ii) CUT($F$, $u$, $v$): it disconnects the tree containing the vertices $u$ and $v$ by removing edge $(u,v)$ from $F$, assuming $(u,v)$ in $F$.

Dynamic trees can be implemented by predecessor arrays. In this case, for LINK, $v$ is made root of its tree and the predecessor of $v$ is set to $u$. Regarding CUT, if $u$ is the predecessor of $v$, then the predecessor of $v$ is set to NIL, otherwise the predecessor of $u$ is set to NIL. The same observation made in Section 3.3 regarding the implementation of CHANGE-ANY and CHANGE-PRED is valid for LINK and CUT as well: they have time proportional to the height of the trees, which means that the time depends on the trees

topology. Representing trees by Euler tours overcomes this issue since a tour is always linear, regardless of tree topology.

This linearization also has an interesting property that helps implementing the dynamic tree abstract data. For a given vertex $v$, the tour corresponding to $v$-subtree, called *$v$-subtour*, is a contiguous sequence of edges along the tour. When $v$ is not the tree root, the $v$-subtour is delimited by the edges that connects $v$ to its predecessor, namely $(p(v), v)$ and $(v, p(v))$. For example, the sequence $\langle 39, 93 \rangle$ in Figure 18, between – but not including – 23 and 32, corresponds to 3-subtree, and the empty sequence between 39 and 93 corresponds to the 9-subtree. This enables the implementation of LINK and CUT by Euler tours to be reduced to a constant number of concatenation and split operations in tours.

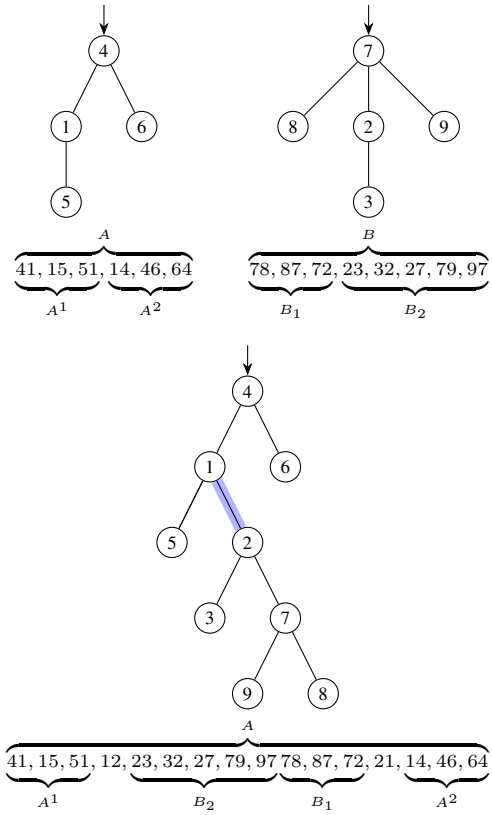
Next, we describe a slightly modified version of the methods LINK and CUT described in (TARJAN, 1997). First, consider LINK$(F,\ u,\ v)$. Let $T_1$ and $T_2$ be trees such that $u \in V_{T_1}$ and $v \in V_{T_2}$. Also, let $A$ and $B$ be the tours representing $T_1$ and $T_2$, respectively. We split $A$ into $A^1$ and $A^2$, and $B$ into $B^1$ and $B^2$. The split of $A$ is made just after the end of the $u$-subtour, and the split of $B$ is made just before the beginning of $v$-subtour. Afterward, we construct a tour by concatenating $A^1, [(u,v)], B^2, B^1, [(v,u)], A^2$ (see Figure 19 for an example). Notice that, concatenating $B^2$ and $B^1$ makes $v$ the root of $T_2$, which is then inserted as $u$-subtree.

Regarding CUT$(F,\ u,\ v)$, let $T$ be the tree containing $u$ and $v$ and $A$ the tour representing it. We split $A$ before and after of both $(u,v)$ and $(v,u)$ in $A^1, [(u,v)], A^2, [(v,u)], A^3$. The cut produces two trees represented by $A^2$, and by $A^1$ concatenated with $A^3$ (see an example in Figure 20).

Both LINK and CUT can be implemented in constant time by storing the tours in doubly linked lists. However, in this case, querying if two vertices are in the same tree would take linear time. Thus, usually the tours are stored in splay trees (SLEATOR; TARJAN, 1985; TARJAN, 1997), a kind of self-adjusting binary search trees. Each edge of a tour becomes a node in a splay tree so that an in-order traversal corresponds to the linear order of the edges in the tour. The amortized time of split and concatenation in splay trees is $O(\log n)$, which gives the same time for LINK and CUT, as well as for querying if two vertices are in the same tree.

Although using splay trees to store Euler tours for EAs seems promising, it would still need to copy parents before mutations. We could think about a combination with structural sharing, however, splay trees are circular structures requiring pointers from each node to its parents, and algorithms combining both turn to be too complex, increasing the practical execution time. Therefore, we explore an alternative approach.

Figure 19 – Example of LINK operation in an Euler tour of a tree. The vertices 1 and 2 are connected to form a new tree.



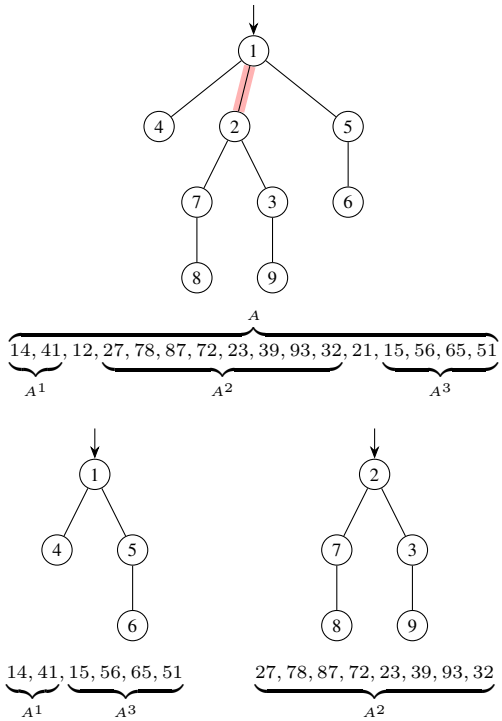

Source: Elaborated by the author.

### 3.5.1 Euler tours in two-level arrays

A two-level structure simplifies storing the tours and using structural sharing in concatenation and split operations. The structure is similar to the two-level array presented in Section 3.3.2, but instead of requiring all subarrays to have the same length, their size can be in the range $[\frac{\lceil\sqrt{n'}\rceil}{2}, 2\lceil\sqrt{n'}\rceil]$, where $n' = 2(n-1)$, and $n$ is the number of vertices in the represented forest. Also, if a two-level array has only one subarray, it can have less than $\frac{\lceil\sqrt{n'}\rceil}{2}$ elements. We first describe how a sequence of values is stored in this structure, and how split and concatenation work. Subsequently, we describe an enhancement in order to implement LINK and CUT efficiently. We refer this enhanced structure as 2LETT, which stands for 2-Level arrays Euler Tour Trees. A similar structure was already used to represent tours (vertex permutations) for the traveling salesman problem (FREDMAN *et al.*, 1995).

A sequence $S = \langle a_1, a_2, \ldots, a_{n-1}, a_n \rangle$ is stored in a two-level array $A$ with $|A| = O(\sqrt{n})$ subarrays, where each subarray $A[i]$, for $1 \leq i \leq |A|$, contains consecutive elements of $S$ such that the concatenation of $A[1], A[2], \ldots, A[|A|]$ equals to $S$. That way, each element $a_k$

Figure 20 – Example of CUT operation in an Euler tour of a tree. The edge $(1,2)$ is removed resulting in two trees.




Source: Elaborated by the author.

of $S$ is stored in $A[i][j]$, where

$$\sum_{p=1}^{i-1} |A[p]| < k \leq \sum_{p=1}^{i} |A[p]|, \text{ and } j = i - \sum_{a=1}^{i-1} |A[a]|.$$

We say that the linear index $k$ corresponds to the two-level index $(i,j)$ of $A$. Computing the mapping from $k$ to $(i,j)$ for a specific value of $k$ can be done in $O(\sqrt{n})$, by first probing a value for $i$ in the range $[1,|A|]$, and then calculating $j$.

Before defining the split and concatenation procedures, we define the auxiliary method CONCAT-ARRAY. It takes the linear arrays $X$ and $Y$ as parameters, where $|X| \leq 2\lceil\sqrt{n'}\rceil$ and $|Y| \leq 2\lceil\sqrt{n'}\rceil$, and produces one or two new linear arrays resulting from the concatenation of $X$ and $Y$. The length of the output array(s) should not exceed the maximum allowed value for subarrays in a two-level array. Therefore, if $|X|+|Y| \leq 2\lceil\sqrt{n'}\rceil$, we return the array $[X[1],X[2],\ldots,X[|X|],Y[1],Y[2],\ldots,Y[|Y|]]$. Otherwise, we concatenate $X$ and $Y$ to create a new array $Z$, and we split $Z$ into $Z^1$ and $Z^2$ such that their size differs in at most 1. The time of CONCAT-ARRAY is $O(\sqrt{n})$.

The definition of CONCAT$(A,B)$, which concatenates the two-level arrays $A$ and $B$ and produces a new two-level array $C$, is defined in three cases:

(i) If $|A| = 1$ and $|A[1]| < \frac{\lceil\sqrt{n'}\rceil}{2}$, then we call CONCAT-ARRAY$(A[1],B[1])$. If it returns

one array $X$, then

$$C = [X, B[2], B[3], \ldots, B[|B|]].$$

Otherwise, CONCAT-ARRAY returns two arrays $X^1$ and $X^2$, then

$$C = [X^1, X^2, B[2], B[3], \ldots, B[|B|]].$$

(ii) If $|B| = 1$ and $|B[1]| < \frac{\lceil \sqrt{n'} \rceil}{2}$, then we call CONCAT-ARRAY$(A[|A|], B[1])$. If it returns one array $X$, then

$$C = [A[1], A[2], \ldots, A[|A|-1], X].$$

Otherwise, CONCAT-ARRAY returns two arrays $X^1$ and $X^2$, then

$$C = [A[1], A[2], \ldots, A[|A|-1], X^1, X^2].$$

(iii) Otherwise,

$$C = [A[1], A[2], \ldots, A[|A|], B[1], B[2], \ldots, B[|B|]].$$

Since at most two new subarrays are created, namely $X^1$ and $X^2$, and all unchanged subarrays are shared by $A$, $B$ and $C$, the time of CONCAT is $O(\sqrt{n})$.

The method SPLIT-BEFORE$(A, i, j)$ splits a two-level array $A$ into $B$ and $C$ before $A[i][j]$, and has two cases:

(i) If $j = 1$, which means $A[i][j]$ is the first element of $A[i]$, then

$$B = [A[1], A[2], \ldots, A[i-1]], \text{ and}$$
$$C = [A[i], A[i+1], \ldots, A[|A|]].$$

(ii) Otherwise, we first split $A[i]$ before $A[i][j]$ into $X$ and $Y$, and then we construct $B$ and $C$ as follows:

- If $i = 1$ then $B = [X]$. Otherwise,

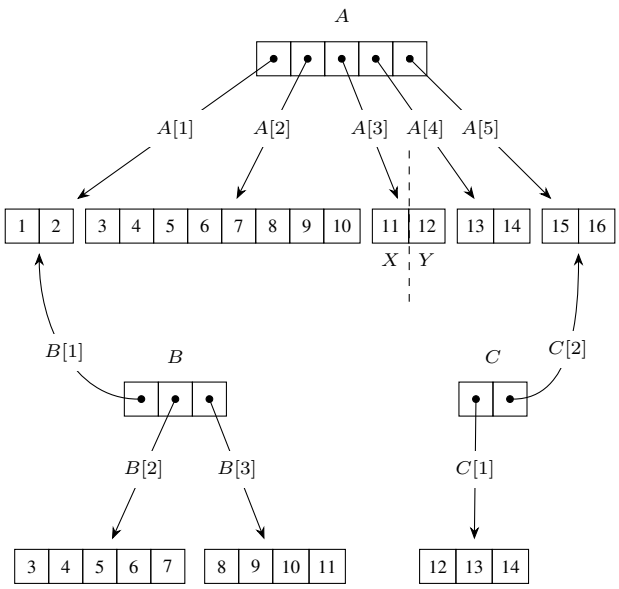
$$B = \text{CONCAT}([A[1], A[2], \ldots A[i-1]], [X]).$$

- If $i = |A|$ then $C = [Y]$. Otherwise,

$$C = \text{CONCAT}([Y], [A[i+1], A[i+2], \ldots A[|A|]).$$

Figure 21 shows an example of SPLIT-BEFORE. The two-level arrays $A$, $B$ and $C$ have $O(\sqrt{n})$ subarrays. At most four subarrays (two in the end of $B$ and two in the beginning of $C$) are created and they have length $O(\sqrt{n})$. Since all the unchanged subarrays are shared by $A$, $B$ and $C$, SPLIT-BEFORE has time $O(\sqrt{n})$. The method SPLIT-AFTER can be implemented likewise with the same time.

Figure 21 – Split of a two-level array. The two-level array $A$ is split before the index $(3,2)$. In order to respect size bounds, $X$ was concatenated with $A[2]$ and then split, resulting in $B[2]$ and $B[3]$. The array $Y$ was concatenated with $A[4]$ resulting in $C[1]$.




Source: Elaborated by the author.

### 3.5.2 Implementing LINK and CUT

For the implementation of LINK and CUT, we need to find the indices for splitting. Specifically, we need to find the beginning and the end of a vertex subtour. Let $A$ be a 2LETT instance representing a spanning tree $T$, we set procedures START$(A,v)$ and END$(A,v)$ to return the linear index of the beginning and the end of the $v$-subtour in $A$.

If $v$ is the root of $T$, then the $v$-subtour is the full tour. Otherwise, the $v$-subtour is delimited by $(p(v),v)$ and $(v,p(v))$, and we need an efficient way to find these edges. A linear search in each subarray $A[i]$ would require $O(n)$ time. Therefore, we enhance $A$ by storing a set $S[i]$ with each subarray $A[i]$ such that a vertex $v$ is in $S[i]$ if and only if there is an edge with $v$ in $A[i]$. We explain later how such a set can be implemented, but now suppose that querying if a vertex is in a set $S[i]$ takes constant time. Thereby, to find the edge $(p(v),v)$ we query which set $S[i]$, for $i$ in $\langle 1,2,\ldots,|A|\rangle$, contains the vertex $v$, and then we perform a linear search in $A[i]$ frontwards to find the index $j$ for which $A[i][j] = (p(v),v)$. We find $(v,p(v))$ in a similar way, but we search in the reverse order, i.e., we search $i$ in $\langle |A|,|A|-1,\ldots,1\rangle$ and $A[i]$ backwards. Finding $i$ takes $|A| = O(\sqrt{n})$, and finding $(p(v),v)$ or $(v,p(v))$ in $A[i]$ also takes $O(\sqrt{n}) = |A[i]|$. Thus, START and END has time $O(\sqrt{n})$.

Regarding LINK and CUT, although they are supposed to deal with forests, each 2LETT instance represents a single tree. Therefore, we define them as follows: CUT disconnects a given tree, and LINK connects two given trees.

More specifically, given a 2LETT instance $A$ representing a spanning tree $T$ and two vertices $u$ and $v$ of $T$, the method CUT$(A,u,v)$ returns two 2LETT instances representing the two trees obtained by removing $(u,v)$ from $T$. The occurrences of $(u,v)$ and $(v,u)$ delimit either the $u$-subtour or the $v$-subtour. We verify the two possibilities, and make the proper sequence of splits and concatenations. The implementation requires a constant number of calls to START, END, CONCAT, SPLIT-BEFORE and SPLIT-AFTER, resulting in CUT having time $O(\sqrt{n})$.

Given two 2LETT instances $A$ and $B$, representing two spanning trees $T_1$ and $T_2$, and two vertices $u$ and $v$ where $u$ is a vertex of $T_1$ and $v$ is a vertex of $T_2$, the method LINK$(A,u,B,v)$ returns a new 2LETT instance representing the tree obtained by linking $T_1$ and $T_2$ through the edge $(u,v)$. We find the $u$-subtour and $v$-subtour using START and END, and the corresponding split indexes. Subsequently, we make the proper sequence of splits and concatenations. As CUT, this requires a constant number of calls to procedures with time $O(\sqrt{n})$, thus LINK has time $O(\sqrt{n})$.

It remains to describe how to implement the enhancement necessary for START and END. Recall that we need to associate a vertex set to each subarray of a 2LETT instance. We could use hash tables taking the vertices as keys, which would have linear average time for insertion, deletion and querying, and would require linear space in the worst case. Since the number of elements of each hash table would be $O(\sqrt{n})$, the use of hash tables would not increase the asymptotic time and space for the 2LETT structure. However, in our preliminary experiments, they had a poor practical performance.

Therefore, we used bit-arrays. We assume that the vertices are numbered $1,2,\ldots,n$, and, for each subarray $A[i]$ of a 2LETT instance $A$, we associate a bit-array $S[i]$ with length $n$, where $X[i][v]$ is set to TRUE if $A[i]$ contains an edge with vertex $v$, and FALSE otherwise. So querying if a vertex is in a subarray $A[i]$ is $O(1)$.

A new bit-array is needed when a new subarray is created. Although creating new subarrays using splits and concatenations takes $O(\sqrt{n})$, creating a new bit-array takes $O(n)$, since it requires the initialization of $O(n)$ bits. However, if it was already initialized, it requires only time $O(\sqrt{n})$ to insert (or remove) the vertices of the corresponding subarray. Thus, we allocate and initialize all bit-arrays beforehand and put them in a pool. Each time a new bit-array is asked, we pick one from the pool in constant time. Similarly, each time a bit-array is discarded (the corresponding subarray is not referenced anymore), we set all its bits TRUE to FALSE in $O(\sqrt{n})$, and return it back to the pool in constant time.

Comparing to hash tables, bit-arrays increase the memory of each tree from $O(n)$ to $O(n^{1.5})$, and the extra memory required by an offspring from $O(\sqrt{n})$ to $O(n)$. Besides that, the time to initialize a population of trees increases from $O(p \cdot n)$ to $O(p \cdot n^{1.5})$, where $p$ is the number of individuals in the population. Despite all this, bit-arrays make 2LETT very efficient in practice.

### 3.5.3 Implementing CHANGE-PRED and CHANGE-ANY

Let $A$ be a 2LETT instance for a spanning tree $T$. The implementation of CHANGE-PRED($A$) is conceptually equivalent to the one for predecessor arrays. We first select an edge $(u,v)$ from $\overline{T}$. If $u$ is ancestor of $v$, we call CUT($A,v,p(v)$), which generates two trees $B$ and $C$, and then we call LINK($B,u,C,v$). Similarly, if $v$ is ancestor of $u$, we call CUT($T,u,p(u)$), which generates two trees $B$ and $C$, and then we call LINK($B,u,C,v$). If neither $u$ is ancestor of $v$ nor $v$ is ancestor of $u$, we randomly choose one of the two previous cases. In order to determine if a vertex $u$ is an ancestor of a vertex $v$, we verify if START($A,u$) $\leq$ START($A,v$) $\leq$ END($A,u$), i.e., we verify if $v$ is in the $u$-subtour. Thus, the time of CHANGE-PRED is $O(\sqrt{n})$.

Regarding CHANGE-ANY($A$), although the 2LETT structure enables efficient insertion and deletion of edges into/from the tree, it still remains defining how to select them well. This is easily solved for CHANGE-PRED, because once we choose an edge $(u,v)$ to insert, we only need to look for $(u,p(u))$ and $(v,p(v))$ as the only possible edges to remove. On the other hand, for CHANGE-ANY, any edges in the path from $u$ to $v$ can be removed, which implies in a path with $O(n)$ edges. Consequently, we take an approach that is viable in practice but has some bias: we try to select edges to add and remove stochastically, and if this fails, we make the edge exchange by calling CHANGE-PRED.

More specifically, we start by selecting an edge $(u,v)$ from $T$, where $v = p(u)$. Afterwards, we try to find an edge $(x,y)$, where $x$ is part of the $u$-subtour and $y$ is not (we refer this as a *subtour restriction*). Subsequently, we call CUT($A,u,v$), which returns two trees $B$ and $C$, and then we call LINK($B,x,C,y$) to reconnect them. If this is not possible, we call CHANGE-PRED($A$). In order to select $(x,y)$, we verify if the length of the $u$-subtour is smaller than $\frac{|A|}{2}$. If so we select $x$, otherwise we take $y$. Next we try $k$ times to select the edge $(x,y)$ from the adjacent list of the previously selected vertex, and we verify if the vertices respect the subtour restriction. Deciding which vertex, $x$ or $y$, must be selected first is essential, because choosing it from the smaller set increases the chance of the another vertex to respect the subtour restriction. If we use a constant value for $k$, then the time of CHANGE-ANY is $O(\sqrt{n})$. We have found that $k = 5$ provides a good trade off between running time versus probability of finding an edge $(x,y)$.

### 3.5.4 Limitations

2LETT has a bias in CHANGE-ANY, which has different implications in the results depending on the type of problem being solved. It can be positive if the fallback to CHANGE-PRED leads to better solutions and faster convergence, or negative otherwise. See Section 3.7.3 for further information regarding the one-tree max problem.

## 3.6 Qualitative evaluation

Table 1 presents a summary about the structures. We can consider the Predecessor as the baseline to compare the other structures. Although its design does not take into account the issues discussed in Section 3.2.1, it is a compact and simple structure to implement and has good practical performance. Moreover, it has linear time for both operations and requires linear memory. And, finally, its mutation operations have no restrictions, being able to perform any valid edge exchange.

Table 1 – Comparison of the Structures Regarding Complexity of Memory for an Initial Individual and Creation of an Offspring; Complexity of Time, and; Restrictions for the Operations.

| Structure | Memory | | Time | | Restrictions | |
|---|---|---|---|---|---|---|
| | Initial | Offspring | CHANGE-ANY | CHANGE-PRED | CHANGE-ANY | CHANGE-PRED |
| Predecessor | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | No | No |
| 2LPredecessor | $O(n)$ | $O(\sqrt{n})$ | — | Average $O(\sqrt{n})$ | — | No |
| NDDR | $O(n)$ | Average $O(\sqrt{n})$ | Average $O(\sqrt{n})$ | Average $O(\sqrt{n})$ | Yes | Yes |
| 2LETT | $O(n^{1.5})$ | $O(n)$ | $O(\sqrt{n})$ | $O(\sqrt{n})$ | Yes | No |

The 2LPredecessor is a modification of Predecessor that aims to avoid copying the parent in CHANGE-PRED. It not only reduces the running time, but also the extra memory needed for each offspring: only one new subarray needs to be allocated. Its implementation requires more work than Predecessor, but it is still simple.

NDDR has average time of $O(\sqrt{n})$ for the operations, which is related to the extra memory required by an offspring: both time and memory are proportional to the size of $T_{from}$ and $T_{to}$. As discussed in Section 3.4.3, the operators have restrictions, which has implications in the search. The implementation of NDDR is complex because, besides the sharing of subtrees scheme, it requires saving the history of changes in order to find the mutation operands.

Finally, the worst-case of 2LETT is $O(\sqrt{n})$, and its CHANGE-PRED has no restrictions, as in Predecessor and 2LPredecessor. On the other hand, in order to achieve the time that CHANGE-ANY has, we fall back to CHANGE-PRED after some failed attempts to find an edge to add (see Section 3.5.4). The initial memory for an individual is larger than for Predecessor, but only $O(\sqrt{n})$ memory is accessed in the offspring creation. The implementation of 2LETT is complex as well, therefore it is for the sake of comprehension that we provide detailed information.

## 3.7 Experimental evaluation

This section presents some experimental results of CHANGE-ANY and CHANGE-PRED implemented for Predecessor, 2LPredecessor, NDDR, and 2LETT.

NDDR usually requires global memory of $O(n^2)$ (DELBEM; LIMA; TELLES, 2012), for a global $n \times n$ matrix is used for finding mutation operands, even though only one row in the matrix is used in each iteration (the one related to vertex $p$ of $T_{from}$). Therefore, when implementing the adjacent strategy, we could decrease the global memory to $O(n)$ by using an $1 \times n$ matrix, thus reducing the overall use of the processor cache, and improving the practical performance. On the other hand, the free strategy does not require global memory. The implementation described in (DELBEM; LIMA; TELLES, 2012) depends on a unspecified constant value $k$, which we set as $k = 1$.

For the structures relying on structural sharing, we use reference counting to keep track of the shared structures. We associate a reference counter to each structure, thus each time a new pointer to the structure is created the counter is incremented, and each time a pointer to a structure is deleted, the counter is decremented. When the counter reaches zero, the structure memory is deallocated, except for the bit-arrays used by 2LETT, which is returned to the pool.

We have observed in preliminary experiments that the structures performance depends on how much data is stored on them. Increasing the memory of a structure makes more pressure on processor cache, which decreases performance. For example, we first used 64-bits integer to store each predecessor, but we changed to 32-bits integer and got a performance improvement. Therefore, we used smaller types wherever possible.

We coded in Rust 1.31.1 and compiled with the command `cargo build --release`. We have run the experiments in a Debian GNU/Linux 9.7 system with Intel Xeon X5670 2.93GHz processor and 32GB of RAM, using only one core for the experiments. The source code are available under a free software license at <https://github.com/malbarbo/ea-tree-repr>.

### 3.7.1 Random trees

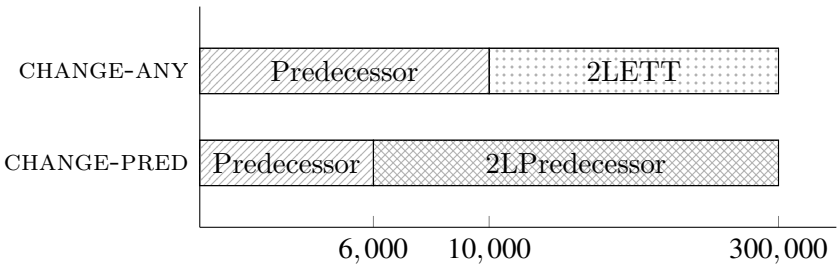
We first evaluate how the running time of mutation operators for random trees changes according to the number of vertices. To generate the random trees we used the following random walk algorithm from (BRODER, 1989): a particle starts in a random vertex and, at each iteration, the particle moves to another randomly chosen adjacent vertex. When the particle visits a vertex for the first time, the edge that leads to the vertex is added to the tree. The algorithm stops when all vertices are visited.

We executed this process 100 times: for each value of $n \in \{1,000, 2,000, ..., 50,000\}$, we generated a random tree of the complete graph in $n$ vertices, and we applied iteratively the mutation operator $10,000$ times. In each iteration, the operator was applied over the resulting tree from the previous iteration. Figure 22 shows the mean time of each operator to run one time.

We use free NDDR to avoid linear time. Recall that the time of NDDR, 2LETT, and 2LPredecessor over random trees is $O(\sqrt{n})$. The plot lines are as expected, behaving as a square function. On the other hand, the plot line for Predecessor shows slight bends due to the increase of memory working set and the associated increase in cache misses. Notice that Predecessor deals with $O(n)$ memory cells to perform each mutation, because it needs to copy the parent tree, while the other structures work with $O(\sqrt{n})$ memory cells. We shall see in the next experiment that increasing $n$ makes cache misses affect the other structures as well.

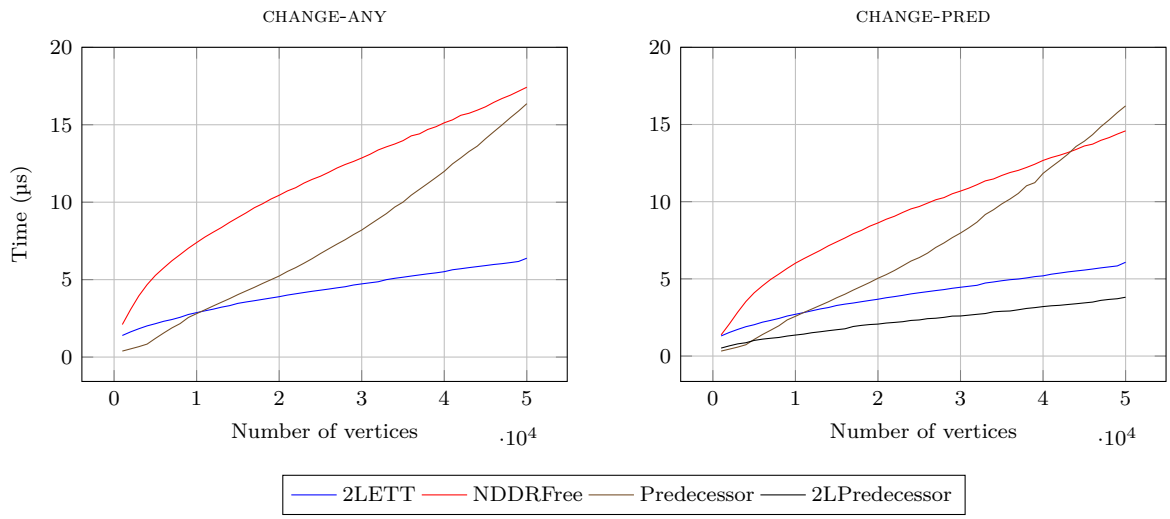
The difference between the running time of CHANGE-ANY and CHANGE-PRED is small for Predecessor, and 2LETT, but more evident for NDDR. Also, Predecessor is the most efficient data structure for CHANGE-ANY for trees with up to $10,000$ vertices, and for CHANGE-PRED for trees with up to $6,000$. For trees with more than $10,000$ vertices, 2LETT is the most efficient for CHANGE-ANY, while for trees with more than $6,000$ vertices, 2LPredecessor is the most efficient for CHANGE-PRED. Remark that the Predecessor is faster than NDDR for both methods for trees with up $\approx 40,000$ vertices, which suggests that the hidden terms in NDDR average time are larger than those in Predecessor and 2LETT complexity times.

In order to evaluate the efficiency of the data structures for larger graphs, we have run the same experiment for $n \in \{6,000, 12,000, \ldots, 300,000\}$. See Figure 23 for the results. Notice that the linear execution of Predecessor is more evident, since the cache effect has stabilized. On the other hand, the cache seems to affect more the running time of NDDR and 2LETT for trees with number vertices from $\approx 50,000$ to $\approx 125,000$, making the running time appears to be superlinear in this range. The running time of NDDR is more affected, while the running time of 2LETT seems regular for trees with more than $\approx 125,000$. The conclusion is that the Predecessor is still competitive with NDDR, even for trees with $300,000$ vertices. In addition, the performance of 2LETT and 2LPredecessor comparing to the others have not changed. The following diagram summarizes the most efficient structure for random trees according to the results from the experiments:



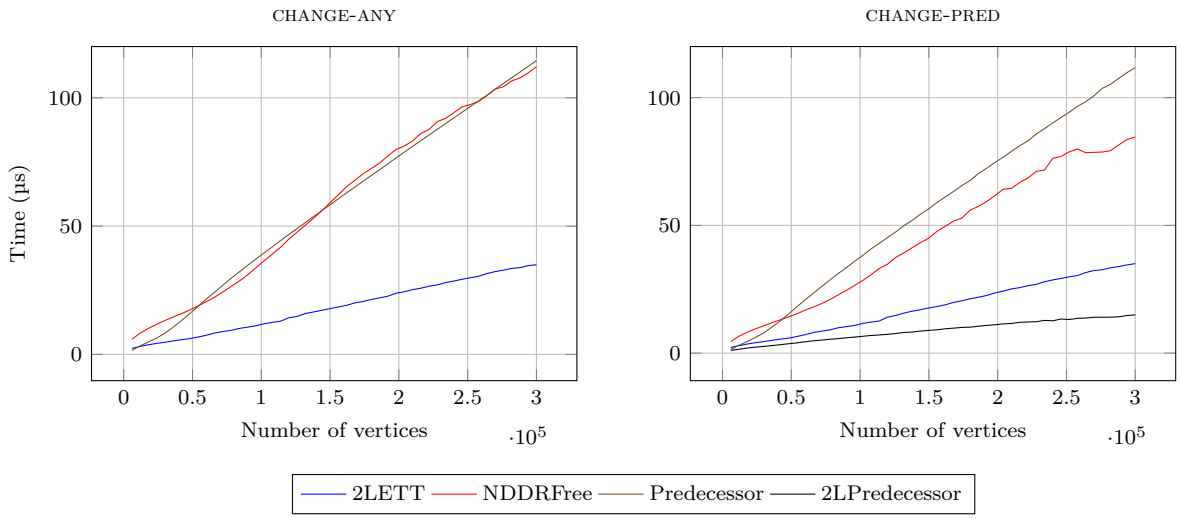

For all these experiments, we used free NDDR, i.e., the free selection strategy for NDDR. Now, we discuss the experiments with the adjacent selection strategy. To avoid its bias, the experiments were run for the following class of graphs: for a given $n$, select a

Figure 22 – Running time of CHANGE-ANY and CHANGE-PRED over random trees of complete graphs. The abscissa has the number of vertices of the tree. The ordinate shows the running time in microseconds. For 2LETT, NDDRFree, and 2Predecessor the time is $O(\sqrt{n})$, while for Predecessor the running is $O(n)$.



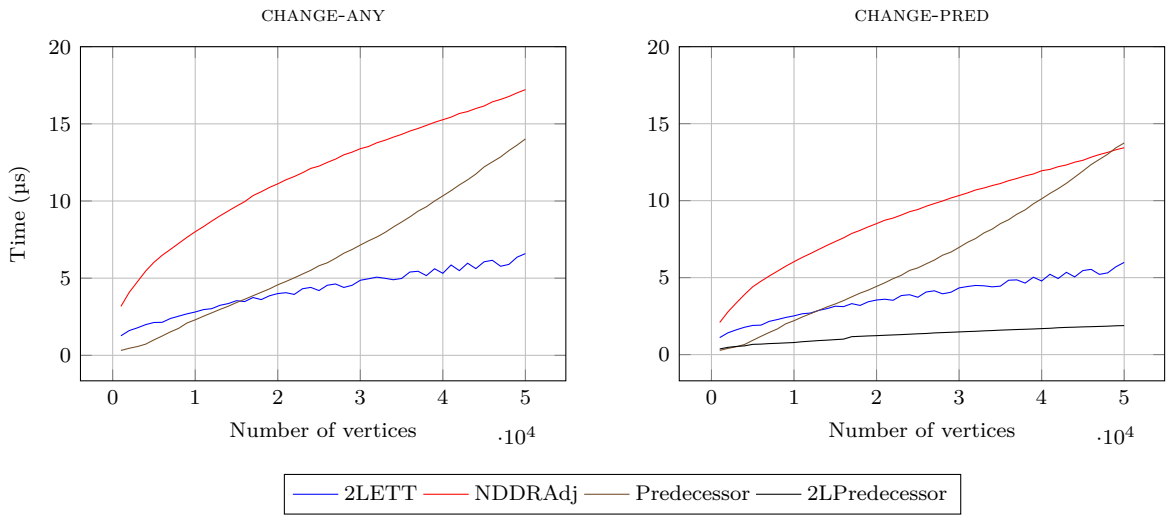

Source: Elaborated by the author.

Figure 23 – Running time of CHANGE-ANY and CHANGE-PRED over random trees from complete graphs. The abscissa has the number of vertices of the tree. The ordinate shows the running time in microseconds. Cache misses affect 2LETT, and especially NDDR.




Source: Elaborated by the author.

Figure 24 – Running time of CHANGE-ANY and CHANGE-PRED for NDDR best-case inputs. The abscissa has the number of vertices of the tree. The ordinate shows the running time in microseconds. Overall, the results are similar to those for random trees.




Source: Elaborated by the author.

vertex $r$ and create $\lceil\sqrt{n}\rceil$ groups of vertices by randomly distributing all vertices to them, except $r$. The difference in the number of vertices between any two groups must be at most 1, making each group have $O(\sqrt{n})$ vertices. Subsequently, for each group, we create edges connecting each pair of vertices inside the group. Finally, we choose one vertex from each group to connect to $r$.

The experiments were executed as before but now, when NDDR is created, we carefully placed all vertices that are in the same group in the same NDDR subtree. The vertex $r$ and its adjacent vertices are placed in $T^*$. Since there are no edges between vertices of two distinct groups, all mutation will affect only one subtree. Therefore, the number of vertices in each subtree will remain the same, avoiding adjacent selection bias. This effectively makes NDDR for one-tree forest works as it does for forests with $O(\sqrt{n})$ disjoint trees, where each forest has $O(\sqrt{n})$ vertices, which is the best case for NDDR.

We used $n \in \{1,000, 2,000, ..., 50,000\}$, and the results (see Figure 24) are similar to those from free strategy shown in Figure 22. This is expected, since NDDR makes a balanced division of subtrees which provides to average case scenarios a resembling performance to that obtained in best cases.

2LETT results in Figure 24 are similar as well to those in Figure 22, except for a greater variance that creates a jagged plot line. On the other hand, Predecessor and 2LPredecessor have a better running time than before. Recall that general random trees have mean diameter of $O(\sqrt{n})$, but the diameter of the random trees used in this last experiment is smaller, specifically $O(\sqrt[4]{n})$, which makes Predecessor and 2LPredecessor run faster. Next, we provide a more in-depth discussion of the impact of diameter on the

performance of structures.

### 3.7.2 Impact of tree diameter

In order to generate random trees with a specified diameter $d < n$, we modified the random walk algorithm described earlier as follows: start the tree with a random path with $d$ edges, select a random vertex from the path and continue as in the random walk algorithm, just avoiding to add edges that increase the tree diameter. In the end, the tree should have diameter $d$.

We executed this process 100 times: for a complete graph in $5,000$ vertices, we generated 50 values for the diameter $d$, evenly distributed across the range $[2,4,999]$. For each value of $d$, we created a random tree with diameter $d$, and we executed the mutation $10,000$ times. Unlike the previous experiments, each mutation was performed in the original tree, not in the tree resulting from the previous application of the mutation. This way, every time the mutation was done in a tree with diameter $d$.

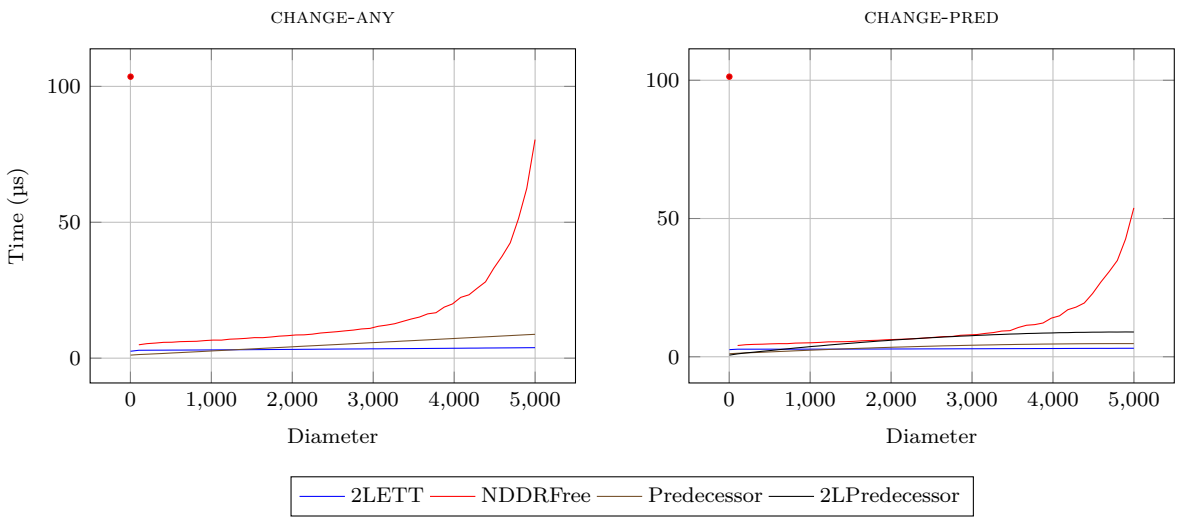
Figure 25 shows the mean time of each operator to run one time. As we can see, the running time of NDDR and Predecessor increases according to the diameter, unless for one exception for NDDR when $d = 2$, which has the highest running time. If $d = 2$ the tree has one central vertex connected to all other vertices and this is the worst case for NDDR, because its division results in all subtrees, except one, containing only one vertex, and one subtree containing all other vertices. Another extreme case for NDDR is $d = n - 1$, which results in all subtrees, except two, with one vertex and the other vertices distributed between the two other subtrees.

The increase in the running time of NDDR and Predecessor seems to grow at the same rate up to $d \approx 3,000$, but is much faster for NDDR for larger values of $d$. Note also that the execution time of the 2LETT is independent on the tree diameter. We can conclude that, while the Predecessor is better for trees with up to $10,000$ vertices and random diameters, it begins to show its limitations for larger diameters, even for trees with a smaller number of vertices, being overcame by 2LETT in these cases. Additionally, 2LETT performance advantage for larger diameter increases as the number of vertices increases, so it comes out to be the best option.

### 3.7.3 Search space exploration

Now we evaluate the characteristics of the data structures when the operators are used inside an EA. We consider a simple benchmark problem called One-max tree (ROTHLAUF; GOLDBERG; HEINZL, 2002), and evaluate the differences in the search performed by each implementation. The objective in the One-max tree problem is finding a given tree $T_{obj}$, which can be determined randomly or by hand. The fitness of an individual

Figure 25 – Impact of the tree diameter in the running time of CHANGE-ANY and CHANGE-PRED. The number of vertices is fixed in $5,000$ and the diameter changes across the range $[2-4,999]$ (abscissa). The ordinate shows the running time in microseconds. Only 2LETT is unaffected by the diameter variation.




Source: Elaborated by the author.

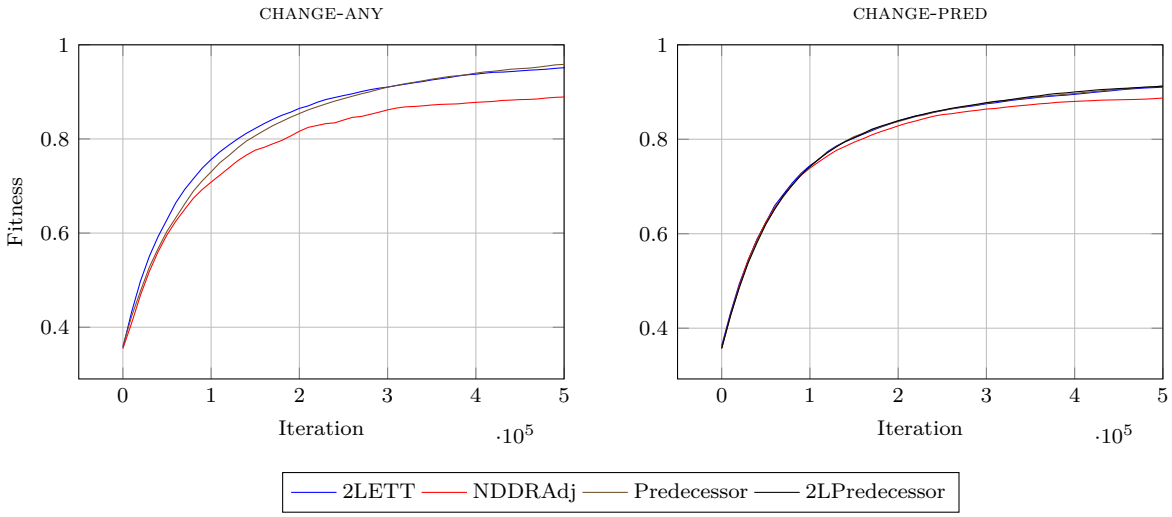
$T$ is defined as $\frac{|E_{T_{obj}} \cap E_T|}{n-1}$, that is, the fitness is proportional to the number of common edges between $T$ and $T_{obj}$. If $T$ has no edges in common with $T_{obj}$, then the fitness is 0, and if $T = T_{obj}$, then the fitness is 1.

The EA was set as follows. The population size was set for 10 randomly initialized individuals. We arbitrarily select the parent and apply the mutation operator. If the offspring was not in the population, the individual with the smaller fitness is replaced. The execution stops after $5 \cdot 10^5$ iterations. The problem instance is a class of randomly connected graphs with $5,000$ vertices and $15,000$ edges, and a tree arbitrarily constructed from those graphs. We have run the algorithm 100 times and got the average fitness of the best individual in the population at each iteration. See Figure 26 for the results.

Notice that the fitness for CHANGE-PRED increases faster than for CHANGE-ANY for the initial iterations, but it slows down after that. A better solution is found by CHANGE-ANY in the end. Also, the progress of fitness in CHANGE-PRED is the same for Predecessor, 2LPredecessor, and 2LETT. This is expected, because they all implement the same process. On the other hand, the progress of NDDR is smaller, which is also expected because, once $T^*$ is defined its vertices cannot change anymore, implying in a search space that cannot be completely explored and some trees that cannot be found. In fact, when we allow EA's to run until finding the tree, only the one using NDDR was not able to find it, running indefinitely.

Regarding CHANGE-ANY, the implementation for 2LETT falls back to CHANGE-PRED when it is not able to find two edges to exchange, which makes CHANGE-ANY

Figure 26 – Evolution of the best individual's fitness for an EA to the one-max tree problem. The abscissa has the iteration number. The ordinate shows the fitness. CHANGE-PRED presents the best results for initial iterations and CHANGE-ANY for final iterations. NDDR search is the restricted one.




Source: Elaborated by the author.

strictly better at initial iterations and strictly worse in final iterations than CHANGE-ANY of Predecessor.

## 3.8 Final remarks

In this chapter we saw that data structures are a key aspect in the implementation of search operations for direct spanning tree representations in EAs. Based on the most common search operator in the literature, namely edge exchange, we defined the mutation operators CHANGE-ANY and CHANGE-PRED and identified two important implementation issues for them: preserving the parent tree, and selecting the edges to exchange in such a way as to ensure a valid offspring. We proposed the 2LETT data structure to implement those operators and performed a qualitative and experimental evaluation of 2LETT, NDDR, and predecessor arrays.

Regarding the structure Predecessor, even having the worst asymptotic time when compared to other structures, its simplicity makes it very efficient in practice. The experiments show that it is the most efficient structure for random trees with up to $10,000$ vertices and remains competitive with NDDR for trees with up to $300,000$ vertices. Besides that, 2LPredecessor, the version using structural sharing, has the best running time for CHANGE-PRED for random trees with more than $6,000$ vertices.

The structure 2LETT overcomes the other structures regarding asymptotic time: only $O(\sqrt{n})$ in the worst-case scenario. Furthermore, the practical efficiency is also the

best for general graphs, being the only structure whose running time is independent of tree diameter. Although CHANGE-ANY may fall back to call CHANGE-PRED, that happens unusually and does not affect its effectiveness.

About NDDR, the experiments show that its practical running time is inferior to the other structures, even considering the best-case scenario. Despite this, NDDR may still be the best approach for particular situations not addressed in our experiments. Therefore, we point some possible limitations that require attention when implementing it: 1) inputs for which the trees have either diameter 2 or close to the maximum diameter; 2) a possible bias in the procedure to select operands for mutation operators, and; 3) restriction on the search performed by operators for one-tree forests.

Overall, the results suggest that, when tackling a difficult spanning tree problem, it is a good option to start with Predecessor, for its simplicity and efficiency for random trees. If the initial experiments indicate that CHANGE-PRED is more promising than CHANGE-ANY, then changing to 2LPredecessor can increase the algorithm efficiency, and it is a simple extension to do. If the input trees are larger and/or have larger diameters, then 2LETT is more rewarding, although it is also more complicated to implement.

In Chapter 5 we explore the use of 2LETT in a local search algorithm which operates in a single solution in each iteration, removing the need for structural sharing from the 2LETT structure. However, before exploring this local search algorithm, which uses a neighborhood defined by a variable number of edge exchange, we discuss in Chapter 4 two simpler neighborhoods based on one-edge and two-edge exchanges and show efficient implementations that do not require advanced data structures.

CHAPTER

4

# ONE-EDGE AND TWO-EDGES EXCHANGE NEIGHBORHOODS

This chapter is based on the paper presented at the Brazilian Symposium on Operations Research 2017 (BARBOSA; DELBEM, 2017), which is part of this doctorate's research.

The minimum spanning tree problem and its variants are well studied in the literature and have many practical applications, especially in the design of networks. We study a variant that considers edge conflict constraints by forbidding the existence of some pairs of edges in the tree, which makes the problem NP-Hard. There are already many complexity results and exact algorithms for the problem, but only a preliminary study about heuristic algorithms. We present an iterated local search algorithm and a new neighborhood structure for the problem. For a set of standard instances, the iterated local search using the new neighborhood found all the known optimal solutions, and improved the result for three instances, thus outperforming other heuristic algorithms.

## 4.1 Introduction

The *minimum spanning tree problem* (MSTP) is widely known in the combinatorial optimization area. Several practical problems, especially those related to the design of networks, can be modeled as variants of MSTP. Many of them are NP-Hard such as the quadratic minimum spanning tree problem (ASSAD; XU, 1992) and the capacitated minimum spanning tree problem (VOSS, 2009).

We study a variant of MSTP called the *minimum spanning tree problem with conflict constraints* (MSTC). Given a graph $G = (V, E)$, a cost function $w : E \to \mathbb{R}^+$, and a set $C \subset E \times E$ of conflicting edges pair, the MSTC consists in finding a spanning tree $T$ of $G$ with a minimum cost $\sum_{e \in E(T)} w(e)$ so that $T$ is conflict free, i.e., $T$ contains no pair

of edges in $C$.

A concept often used to define the MSTC is the *conflict graph*. In a conflict graph $\hat{G} = (E, C)$, its vertices correspond to the edges of $G$, and its edges correspond to the conflicts in $C$. This way the edges of a conflict free spanning tree of $G$ correspond to an independent set in $\hat{G}$.

Darmann, Pferschy and Schauer (2009) first introduced the MSTC and presented results about the complexity, showing that it is NP-hard even if the conflict graph is a union of paths of size 2. Other complexity results are later presented by Darmann *et al.* (2011).

Zhang, Kabadi and Punnen (2011) show that the problem can be solved in polynomial time if the conflict graph is a collection of disjoint cliques. They also show exact algorithms, feasibility tests and a preliminary study about heuristics for the problem. The heuristics proposed were: local search (LS), tabu search (TS) and tabu thresholding (TT), all off them based on the one-edge exchange neighborhood (discussed in section 4.2).

Samer and Urrutia (2015) proposed two new formulations and a branch and cut approach to solving the problem. One formulation is based in subtour elimination constraints (SECs) and another in odd-cycle inequalities. The formulation based in SECs found new certificates of feasibility and optimality, as well as dual bounds stronger than those in (ZHANG; KABADI; PUNNEN, 2011).

Bittencourt, Campêlo and Dias (2016) presented a strategy using vertex labeling to remove cycles. Their initial experiments resulted in a good computational performance and found that three open instances were unfeasible (SAMER; URRUTIA, 2013; SAMER; URRUTIA, 2015).

Although newer models have been more robust, they still miss the optimal solutions for some feasible instances of benchmark sets. Moreover, the gap between lower and upper bounds for some instances is large. One possibility to try decreasing these gaps is through heuristic algorithms. To the best of our knowledge, the work of Zhang, Kabadi and Punnen (2011) seems to be the only using heuristics approach. However, preliminary results show large gaps for instances with known optimal solutions, and also no gap reduction for the other instances.

We propose an iterated local search algorithm and a neighborhood structure for MSTC. In experiments using a benchmark set, our approach found all known optimal values as well as decreased the gap for three instances with an unknown optimal solution.

We organized this chapter as follows: in section 4.2 we present two neighborhood structures and describe our algorithm. In section 4.3 we discuss the results of our computational experiments and in section 4.4 we present the final remarks.

## 4.2 Edge exchange neighborhoods

A neighborhood structure usually used in spanning tree problems is the *one-edge exchange neighborhood* (ZHANG; KABADI; PUNNEN, 2011). For a given tree $T$, it consists of all valid trees that can be obtained from $T$ by replacing one of its edges. The replacing process can be seen in two ways:

(i) An edge $e$ is removed from $T$, which creates two connected components. Then an edge $f \neq e$ connecting both components is added to $T$, and;

(ii) An edge $f$ is added to $T$, which creates a cycle. Then an edge $e \neq f$ is removed from this cycle.

Notice that both strategies preserve the property of the solution being a tree, i.e., a connected subgraph without cycles. Now we give a formal definition for this neighborhood.

Let $T \subset E$ be the set of edges of a spanning tree of a graph $G = (V, E)$. Also, let $\overline{T} = E \setminus T$ be the set of edges of $E$ which are not in $T$. We define the one-edge exchange neighborhood as $N_1(T) = \{(T \setminus \{e\}) \cup \{f\} : e \in E, f \in \overline{T}$ and $(T \setminus \{e\}) \cup \{f\}$ contains no cycles$\}$. $N_1$ is called as *2-exchange* by Zhang, Kabadi and Punnen (2011).

We can extend this concept to a neighborhood based in the exchange of two edges: $N_2(T) = \{(T \setminus \{e_1, e_2\}) \cup \{f_1, f_2\} : \{e_1, e_2\} \in \binom{T}{2}, \{f_1, f_2\} \in \binom{\overline{T}}{2}$ and $(T \setminus \{e_1, e_2\}) \cup \{f_1, f_2\}$ contains no cycles$\}$. $N_2$ is not so much used in the literature (see uses in (RIBEIRO; SOUZA, 2002; BUI; DENG; ZRNCIC, 2012)), possibly because of its size. While the size of $N_1$ is $O(nm)$, the size of $N_2$ is $O(n^2m^2)$. Despite this, we implement a local search using $N_2$ that is efficient in practice.

In order to define a search in $N_1$ and $N_2$ for MTSC, we need to deal with unfeasible solutions. We use the strategy of penalizing conflicts in the objective function proposed by Zhang, Kabadi and Punnen (2011). The objective function is defined as: $obj(T) = w(T) + \alpha c(T)$, where $w(T)$ is the sum of costs of all edges in $T$, $\alpha$ is a large constant value, and $c(T) = |\{(e, f) : \{e, f\} \in C$ and $\{e, f\} \in T\}|$ is the number of violated restrictions (from restrictions set $C$).

The procedures ONE-EXCHANGE and TWO-EXCHANGES (see Algorithm 1) describe searching the neighborhoods $N_1$ and $N_2$, respectively. Next, we describe how they can be efficiently implemented.

### 4.2.1 Local search implementation

For each edge $e$ we define an attribute $e.cost$ with the cost of adding $e$ to the current incumbent solution. The sets $T$ and $\overline{T}$ are represented by arrays, and their indices

**Algorithm 1** – Searching the neighborhoods $N_1$ and $N_2$.

ONE-EXCHANGE($T$, $\overline{T}$)

1 **for each** edge $e \in T$ **do**
2 $\quad T' = T \setminus \{e\}$
3 $\quad$ **if** $\exists f \in \overline{T}$ such that $T' \cup \{f\}$ is a tree **and** $obj(T' \cup \{f\}) < obj(T)$ **then**
4 $\quad\quad T = (T \setminus \{e\}) \cup \{f\}$
5 $\quad\quad \overline{T} = (\overline{T} \setminus \{f\}) \cup \{e\}$
6 $\quad\quad$ **return** IMPROVED
7 **return** NOT-IMPROVED

TWO-EXCHANGES($T$, $\overline{T}$)

1 **for each** pair of edges $(e_1, e_2) \in \binom{T}{2}$ **do**
2 $\quad T' = T \setminus \{e_1, e_2\}$
3 $\quad$ **if** $\exists\ (f_1, f_2) \in \binom{\overline{T}}{2}$ such that $T' \cup \{f_1, f_2\}$ is a tree **and** $obj(T' \cup \{f_1, f_2\}) < obj(T)$ **then**
4 $\quad\quad T = (T \setminus \{e_1, e_2\}) \cup \{f_1, f_2\}$
5 $\quad\quad \overline{T} = (\overline{T} \setminus \{f_1, f_2\}) \cup \{e_1, e_2\}$
6 $\quad\quad$ **return** IMPROVED
7 **return** NOT-IMPROVED

are used to access the edges, providing exchanges in time $O(1)$ (lines 4 and 5 in ONE-EXCHANGE and TWO-EXCHANGES). The attribution in line 2 is conceptual and has no need to be performed. Line 3 in both procedures is implemented as a loop testing each edge or pair of edges.

The objective function evaluation can be done through the incremental strategy proposed by Zhang, Kabadi and Punnen (2011), computing only the cost of modifications in the incumbent solution. For this purpose, we define an attribute $e.conf$ for each edge $e$ in order to store the number of conflicting edges with $e$ in $T$, i.e., $e.conf = |\{f : f \in T \text{ and } \{e, f\} \in C\}|$. Given the value of $obj(T)$, and the edges $e \in T$ and $f \in \overline{T}$, the objective function of $T' = (T \setminus \{e\}) \cup \{f\}$ can be calculated in $O(1)$ by $obj(T') = w(T) + f.cost - e.cost + \alpha C(T')$, where

$$C(T') = \begin{cases} c(T) + f.conf - e.conf - 1, & \text{if } \{e, f\} \in C \\ c(T) + f.conf - e.conf, & \text{otherwise.} \end{cases}$$

In this case $T' \in N_1(T)$. The incremental evaluation is done likewise for $N_2(T)$ . When an edge $f$ is removed from a tree (resp., added into a tree), the values $e.conf$ are decreased (resp., increased) for each edge $e$ such that $\{f, e\} \in C$. Since the number of conflicts of an edge is $O(m)$, the execution time for this operation is $O(m)$.

In order to verify if an exchange of edges generates a valid tree (line 3 in the procedures) in time $O(1)$, we run a depth-first search (DFS) in $T$ before line 1. For each edge $e \in T$, we define two attributes: the start time $e.desc$, and the end time $e.term$, which represents the time stamps as described by Cormen *et al.* (2009). We say a vertex $v$ is a *descendant* of a vertex $u$ in DFS tree if $u.desc \leq v.desc$ and $v.term \leq u.term$. The execution time of DFS for trees is $O(n)$.

We discuss now the exchange of an edge (the strategy is similar for two edges). Let $u$ and $v$ be the extremes of an edge $e$ such that $u$ has been visited before $v$ during DFS, i.e., $u.desc < v.desc$. In subgraph $T' = T \setminus \{e\}$, the descendants of $v$ in DFS tree form a connected component and the other vertices form another. Given the extremes $x$ and $y$ of an edge $f$ such that $x.desc < y.desc$, we can verify in time $O(1)$ if $f$ connects those two connected components of $T'$: if $y$ is descendant of $v$ in DFS tree ($y$ is inside a connected component), and $x$ is not descendant of $v$ in DFS tree ($x$ is inside the other component), then $f$ connects the two connected components of $T'$.

Since line 3 can be done in $O(1)$, the execution time of DFS is $O(n)$, and updating $conf$ takes time $O(m)$, the total execution time of ONE-EXCHANGE is $n+m+nm = O(nm)$ and, of TWO-EXCHANGES is $n+m+n^2m^2 = O(n^2m^2)$.

In TWO-EXCHANGES, the loop in line 3 is the biggest cost with time $O(m^2)$. In order to improve the procedure, we use a strategy to decrease the number of pairs of edges that need to be verified. We can notice that the subgraph $T' = T \setminus \{e_1, e_2\}$ (line 2) has three connected components. Denote them by $A$, $B$, and $C$. Before line 3, we classify all edges of $\overline{T}$ according to their extremes as follows: if the extremes of an edge $e$ are in two connected components $A$ and $B$, then we add $e$ to a set $G_{AB}$. The other cases are likewise: if extremes are in $A$ and $C$, we add the edge to a set $G_{AC}$, and if they are in $B$ and $C$, we add the edge to a set $G_{BC}$. The edges with extremes in the same connected component are not added to any set. This way, there is no need to test all pairs of $\binom{\overline{T}}{2}$, because we only have to test the pairs which connect the $A$, $B$ and $C$ components: $G_{AB} \times G_{AC} \cup G_{AB} \times G_{BC} \cup G_{AC} \times G_{BC}$. Even though this strategy does not affect the asymptotic complexity of TWO-EXCHANGES, it is indeed very efficient in our experiments.

We use these two local search in an iterated local search algorithm that we describe next.

### 4.2.2 Iterated local search

The main idea of an *iterated local search* is to perform a local search iteratively and take advantage of the solutions found in previous iterations (LOURENÇO; MARTIN; STÜTZLE, 2010). The algorithm keeps an incumbent solution $T$ and the best current solution $T_b$. At each iteration, a local search is performed from $T$ and, if the result is better

than $T_b$, it replaces $T_b$. Before the next iteration, $T$ "suffers" a perturbation in order to leave the local minimum. This perturbation must be balanced, not too strong; otherwise, the new search neighborhood could not be related to the previous one, thus discarding the characteristics obtained in previous searches. On the other hand, the perturbation should not be too weak; otherwise, the algorithm will remain at the same local minimum.

An ILS using ONE-EXCHANGE or TWO-EXCHANGES to solve MSTC is described in Algorithm 2. In the next section, we present the results from the experimental evaluation of that algorithm.

**Algorithm 2** – Iterated local search

ILS($E$, $T$, $P$, *Iters*)

1 $T_b = T$
2 $\overline{T} = E \setminus T$
3 **repeat** *Iters* times
    ▷ Local Search
4    execute EXCHANGE($T, \overline{T}$) while the returned value is IMPROVED
5    **if** $obj(T) < obj(T_b)$ **then**
6        $T_b = T$
    ▷ Perturbation
7    Arbitrarily select a subset $A \subset T$ with size $P$
    and a subset $B \subset \overline{T}$ such that $(T \setminus A) \cup B$ is a tree
8    $T = (T \setminus A) \cup B$
9    $\overline{T} = (\overline{T} \setminus B) \cup A$
10 **return** $T_b$

## 4.3 Experimental evaluation

We present in this section the results of a computational evaluation of the iterated local search (Algorithm 2) applied to neighborhoods $N_1$ and $N_2$, which we denote by $B_1$ and $B_2$, respectively. We use the benchmark instances proposed by Zhang, Kabadi and Punnen (2011). The instances are divided in types 1 and 2, and each one is defined by a graph $G = (V, E)$ and a conflict set $C$, identified by a label $|V|$-$|E|$-$|C|$. Type 2 instances were generated to be feasible and have known optimal solutions. Type 1 instances have difficult cases: among 85 instances, 35 were determined to be unfeasible, 11 are feasible, and up to now, the remaining instances have unknown feasibility. Among the 11 feasible instances, 9 of them have known optimal solutions.

The algorithms were implemented in Rust version 1.16 with the command `cargo build --release` for compilation. Experiments were carried using only one core on a

Table 2 – Performance to find feasible solutions for type 1 instances.

| | | | $B_1$ | | $B_2$ | |
|---|---|---|---|---|---|---|
| $\|V\|$ | $\|E\|$ | $\|C\|$ | Obj | T (s) | Obj | T (s) |
| 50 | 200 | 199 | 1200 | 0,000 | **1192** | 0,000 |
| 50 | 200 | 398 | **1215** | 0,000 | 1230 | 0,000 |
| 50 | 200 | 597 | **1379** | 0,000 | 1409 | 0,000 |
| 50 | 200 | 995 | **1676** | 0,000 | 1805 | 0,010 |
| 100 | 300 | 448 | **4842** | 0,010 | 4934 | 0,030 |
| 100 | 300 | 897 | **6095** | 0,050 | 6362 | 0,060 |
| 100 | 500 | 1247 | **5848** | 0,020 | 6159 | 0,045 |
| 100 | 500 | 2495 | **8083** | 0,020 | 8758 | 0,040 |
| 100 | 500 | 3741 | **9221** | 0,040 | 10354 | 0,060 |
| 200 | 600 | 1797 | **14674** | 2,255 | 15449 | 1,310 |
| 200 | 800 | 3196 | **27478** | 0,150 | 28800 | 0,245 |

Source: Research data.

Debian GNU/Linux 8.7 system with Intel Core i5-3320M 2.6Mhz processor and 8GB of RAM.

Each algorithm was executed 30 times. We show the results with the median run time in seconds and the median value of the objective function. We also present box plots to help visualizing the distribution of the objective function value of solutions.

Each execution starts with a solution generated by Kruskal's minimum spanning tree algorithm (KRUSKAL, 1956), for which we use random weights for the edges in order to get different initial solutions. We use the number of edges of the instance to set the number of iterations of ILS, i.e., $Iters = |E|$. The perturbation consisted of replacing three edges in the incumbent solution.

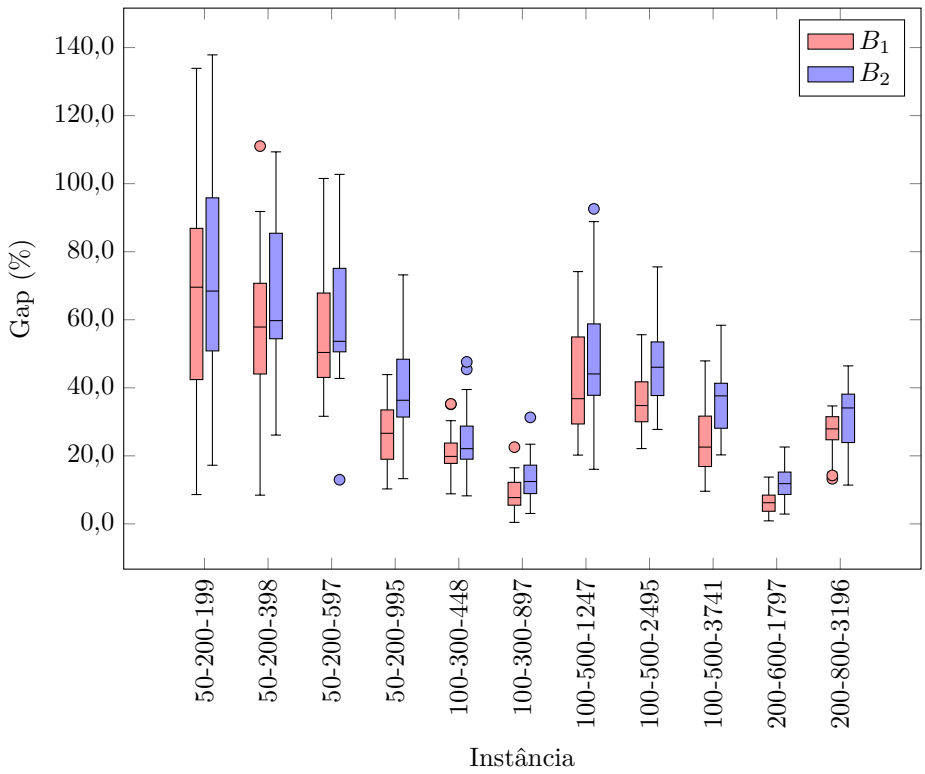
### 4.3.1 Feasible solutions

Since finding feasible solutions, namely spanning trees without conflicting edges, for MSTC is NP-hard, it is interesting to evaluate the ability of algorithms to find feasible solutions quickly.

We set both algorithms to finish as soon as a feasible solution was found; otherwise, they executed up to a maximum number of iterations.

The results for type 1 feasible instances are presented in Table 2 and Figure 27. In Table 2 notice that $B_1$ and $B_2$ find feasible solutions quickly but, in general, $B_1$ is faster and find better solutions than $B_2$. Only for the case 200-600-1797, the algorithm $B_1$ did not find feasible solutions in all its executions (it missed 7). This makes its execution time larger than the one of $B_2$, since it executed up to the maximum number of iterations in those cases. Only for the case 50-200-199, the median solution found by $B_1$ was worst than the one found by $B_2$.

Figure 27 – Gaps for feasible solutions for type 1 instances.




Source: Elaborated by the author.

Figure 27 shows the value distribution of objective function for the solutions found in 30 executions of $B_1$ and $B_2$. At the ordinate, we have the gap to the optimal solution for instances of 50 and 100 vertices, and the gap to the best known solution for the others (see Table 4). The gap of a solution $s$ to a solution $b$ is given by $100 \cdot (obj(b) - obj(s))/obj(b)$. Notice that the gap between solution values for graphs with the same number of vertices and edges decreases as the number of constraints increases. It happens because the increase of restrictions decreases the number of feasible solutions, which also decreases the mean value of the objective function (we suppose a uniform decrease in the distribution of the values of the objective function).

Results for type 2 instances are presented in Table 3. Since they are similar to those for type 1 instances, we omit a diagram. Notice that the execution time of $B_2$ increases more than the one of $B_1$ as the size of instances increases. This is expected because neighborhood $N_2$ is larger than $N_1$ (as discussed in subsection 4.2.2).

### 4.3.2 Quality of the solutions

Next, we evaluate the ability of algorithms $B_1$ and $B_2$ to find good solutions. Results are presented in Table 4 and Figure 28. In Table 4, we present not only the median of objective function values, but also the best value found in the 30 executions for $B_1$ and $B_2$

Table 3 – Performance to find feasible solutions for type 2 instances.

| | | | $B_1$ | | $B_2$ | |
|---|---|---|---|---|---|---|
| $\|V\|$ | $\|E\|$ | $\|C\|$ | Obj | T (s) | Obj | T (s) |
| 50 | 200 | 3903 | **1722** | 0,010 | 1844 | 0,020 |
| 50 | 200 | 4877 | **2150** | 0,010 | 2205 | 0,020 |
| 50 | 200 | 5864 | **2378** | 0,010 | 2434 | 0,030 |
| 100 | 300 | 8609 | **7507** | 0,030 | 7534 | 0,090 |
| 100 | 300 | 10686 | **8030** | 0,060 | 8040 | 0,160 |
| 100 | 300 | 12761 | **8169** | 0,040 | 8182 | 0,170 |
| 100 | 500 | 24740 | **12731** | 0,100 | 12890 | 0,340 |
| 100 | 500 | 30886 | **11392** | 0,115 | 11542 | 0,470 |
| 100 | 500 | 36827 | **11564** | 0,090 | 11686 | 0,565 |
| 200 | 400 | 13660 | **17758** | 0,100 | 17897 | 0,340 |
| 200 | 400 | 17088 | **18713** | 0,140 | 18784 | 0,375 |
| 200 | 400 | 20469 | **19189** | 0,130 | 19274 | 0,530 |
| 200 | 600 | 34504 | **20827** | 0,315 | 20895 | 1,535 |
| 200 | 600 | 42930 | **18128** | 0,340 | 18162 | 1,685 |
| 200 | 600 | 50984 | **20864** | 0,405 | 20954 | 2,615 |
| 200 | 800 | 62625 | **39911** | 0,520 | 40049 | 3,340 |
| 200 | 800 | 78387 | **38170** | 0,595 | 38352 | 4,505 |
| 200 | 800 | 93978 | **38819** | 0,620 | 39100 | 4,485 |
| 300 | 600 | 31001 | **43721** | 0,380 | **43721** | 9,990 |
| 300 | 600 | 38216 | **44280** | 0,455 | 44394 | 1,325 |
| 300 | 600 | 45310 | **43206** | 0,465 | 43347 | 2,320 |
| 300 | 800 | 59600 | **43125** | 0,785 | 43247 | 4,420 |
| 300 | 800 | 74500 | **42377** | 0,930 | 42436 | 5,230 |
| 300 | 800 | 89300 | **44164** | 1,085 | 44270 | 5,225 |
| 300 | 1000 | 96590 | **71675** | 1,210 | 71734 | 7,995 |
| 300 | 1000 | 120500 | **76345** | 1,415 | **76345** | 14,125 |
| 300 | 1000 | 144090 | **78880** | 1,610 | **78880** | 20,535 |

Source: Research data.

(column named "Best"). Column "Model / Obj" presents the value of optimal solutions obtained by an exact method. Since an optimal solution is not known yet for the three last instances, we show the best-known bounds.

As before, the execution time of $B_2$ is larger than the one of $B_1$. But, for this experiment, $B_2$ overcame $B_1$ regarding the quality of solutions. Considering the median of the executions, algorithm $B_1$ found optimal solutions for two instances, and $B_2$ found them for five instances. Considering the best value among executions, algorithm $B_1$ found optimal solutions for four instances, and $B_2$ found them for all instances with known optimal values. For other instances with unknown optimal values, $B_2$ could overcome the best-known solutions not only regarding the median but also in respect to the best value, this way decreasing the gap for the lower bound. For instance 100-500-3741, the gap decreased from 28,99% to 13,09%, for instance 200-600-1797 it decreased from 6,33% to 3,99%, and for instance 200-800-3196, it decreased from 5,07% to 3,43%.

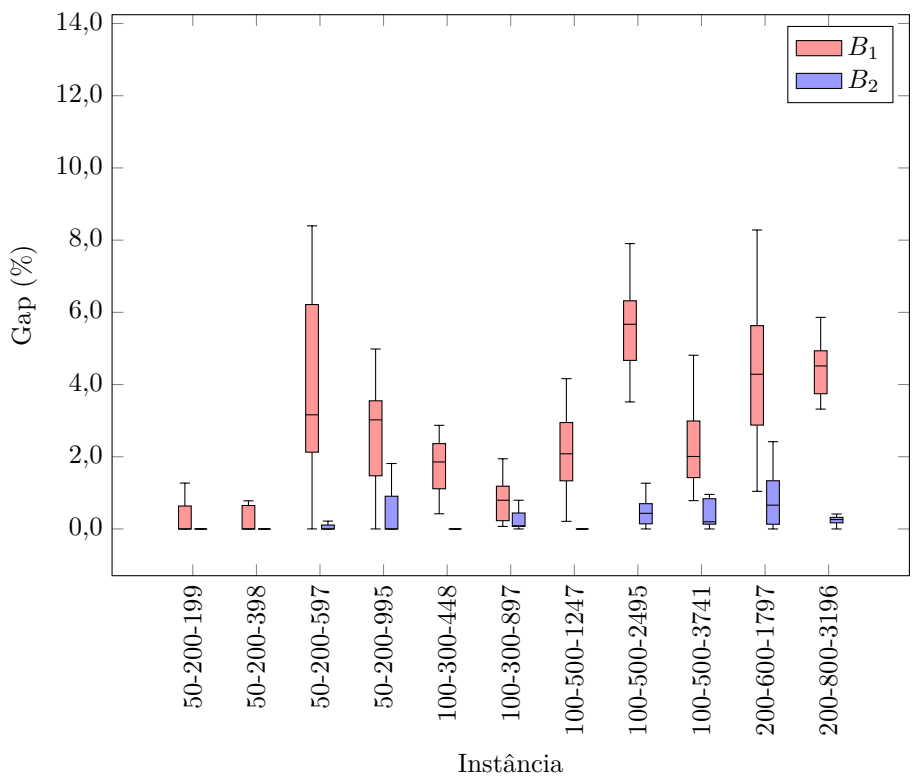
Notice that $B_2$ is consistent in finding good solutions (see Figure 28). For instances

Table 4 – Quality of solutions for type 1 instances.

| | | | Model | $B_1$ | | | $B_2$ | | |
|---|---|---|---|---|---|---|---|---|---|
| $\|V\|$ | $\|E\|$ | $\|C\|$ | Obj | Obj | Best | T (s) | Obj | Best | T (s) |
| 50 | 200 | 199 | **708** | **708** | **708** | 0,090 | **708** | **708** | 0,470 |
| 50 | 200 | 398 | **770** | **770** | **770** | 0,100 | **770** | **770** | 0,460 |
| 50 | 200 | 597 | **917** | 946 | **917** | 0,110 | 918 | **917** | 0,450 |
| 50 | 200 | 995 | **1324** | 1364 | **1324** | 0,120 | **1324** | **1324** | 0,460 |
| 100 | 300 | 448 | **4041** | 4118 | 4058 | 0,380 | **4041** | **4041** | 3,230 |
| 100 | 300 | 897 | **5658** | 5704 | 5662 | 0,410 | 5663 | **5658** | 2,425 |
| 100 | 500 | 1247 | **4275** | 4364 | 4284 | 1,205 | **4275** | **4275** | 8,590 |
| 100 | 500 | 2495 | **5997** | 6347 | 6125 | 1,410 | 6024 | **5997** | 8,200 |
| 100 | 500 | 3741 | [6538 - 9207] | 7675 | 7582 | 1,510 | 7538 | **7523** | 7,225 |
| 200 | 600 | 1797 | [13264 - 14161] | 14418 | 13959 | 3,315 | 13915 | **13815** | 29,465 |
| 300 | 800 | 3196 | [20744 - 21852] | 22453 | 22193 | 6,950 | 21536 | **21480** | 62,480 |

Source: Research data. The values for "Model / Obj" column are from Bittencourt, Campêlo and Dias (2016) and Samer and Urrutia (2015).

Figure 28 – Gaps for type 1 instances.




Source: Elaborated by the author.

Table 5 – Comparison of heuristics for type 1 instances.

| | | | LS | | TT | | TS | | $B_1$ | | $B_2$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\|V\|$ | $\|E\|$ | $\|C\|$ | Obj | T (s) | Obj | T (s) | Obj | T (s) | Obj | T (s) | Obj | T (s) |
| 50 | 200 | 199 | **708** | 0,031 | 735 | 0,434 | 711 | 0,513 | **708** | 0,090 | **708** | 0,470 |
| 50 | 200 | 398 | 797 | 0,053 | 789 | 0,519 | 785 | 0,497 | **770** | 0,100 | **770** | 0,460 |
| 50 | 200 | 597 | - | 0,088 | 1044 | 0,522 | 1086 | 0,428 | 946 | 0,110 | **917** | 0,450 |
| 50 | 200 | 995 | 1424 | 0,125 | 1721 | 0,397 | 1629 | 0,506 | 1364 | 0,120 | **1324** | 0,460 |
| 100 | 300 | 448 | 4102 | 0,392 | 4316 | 2,831 | 4207 | 2,772 | 4118 | 0,380 | **4041** | 3,230 |
| 100 | 300 | 897 | - | 0,769 | - | 2,381 | - | 2,625 | 5704 | 0,410 | **5663** | 2,425 |
| 100 | 500 | 1247 | 4293 | 0,941 | 4913 | 5,797 | 4539 | 7,756 | 4364 | 1,205 | **4275** | 8,590 |
| 100 | 500 | 2495 | 6603 | 1,275 | 7959 | 6,206 | 6812 | 7,484 | 6347 | 1,410 | **6024** | 8,200 |
| 100 | 500 | 3741 | - | 1,728 | 10066 | 5,681 | 8787 | 6,544 | 7675 | 1,510 | **7538** | 7,225 |
| 200 | 600 | 1797 | - | 6,431 | - | 37,282 | - | 31,744 | 14418 | 3,315 | **13915** | 29,465 |
| 300 | 800 | 3196 | - | 10,463 | - | 49,369 | - | 46,082 | 22453 | 6,950 | **21536** | 62,480 |

Source: Research data. The values for "LS", "TT" and "TS" column are from Zhang, Kabadi and Punnen (2011).

50-200-199 and 50-200-398, $B_2$ found the optimal solution in all executions. Except for instances 50-200-995, 100-500-3741, and 200-800-3196, all solutions have a gap smaller than 2%.

Next section, we discuss results for type 2 instances, and we compare our approach with other heuristics.

### 4.3.3 Comparison with other heuristics

In this section we compare $B_1$ and $B_2$ with heuristic algorithms LS, TT and TS, proposed by Zhang, Kabadi and Punnen (2011) (described in section 4.1). We highlight that the authors described results from the heuristics as preliminary. Their experiments were executed in a Dell computer with processor Intel Pentium 4 3.4 GHz and a workstation Dell with processor Intel Xeon 2.0 GHz.

Results for type 1 instances are shown in Table 5. Zhang, Kabadi and Punnen (2011) does not provide information about the number of executions for each algorithm, neither if execution times and objective function values are the best found or the median (or average) of many executions. For $B_1$ and $B_2$, the values are the median of execution time and the median of objective function values. Notice that values in columns "Obj" and "T (s)" for $B_1$ and $B_2$ in Table 5 are the same in Table 4, and they are repeated to ease the comparison. A value "-" in column "Obj" for LS, TT and TS mean the algorithm found no feasible solution.

Notice that heuristics LS, TT and TS found no feasible solutions for some instances. For cases 100-300-448 and 100-500-1247, algorithm LS found better solutions than $B_1$, for case 50-200-199, algorithms LS and $B_1$ found solutions with the same value, and for the

remaining cases, $B_1$ overcomes LS. Algorithm $B_1$ overcomes TT and TS in all cases, and $B_2$ overcomes LS, TT, and TS in all cases.

In order to make a comparison of execution times, we need to adjust the times, since experiments were executed in different computers. By using websites CpuBenchmark [1] and UserBenchmark [2], we establish a limit of 5 times for the performance (superior) of the computer in which we executed our tests in relation to those used by Zhang, Kabadi and Punnen (2011). Adjusted execution times for LS, TT and TS (originally reported time divided by 5) are presented in columns "T (s)". Considering this new information, $B_1$ and LS present the best times and are competitive with each other. $B_2$ presents times more elevated, but it is competitive with TT and TS.

Results for type 2 instances are presented in Table 6. Column "Optimal" contains values of the optimal solutions for each instance. All heuristics found optimal solutions for all instances ($B_1$ and $B_2$ found optimal solutions in all executions). These results are similar to those obtained for type 1 instances, highlighting $B_1$ with total execution time 68% better than LS, i.e., the second fastest heuristic.

## 4.4 Final remarks

We proposed an iterated local search and a neighborhood for the minimum spanning tree problem with conflict constraints. Also, we presented neighborhood $N_1$ described in (ZHANG; KABADI; PUNNEN, 2011), which is based on the exchange of an edge. Our proposed neighborhood $N_2$ differs by doing the exchange of two edges. We described efficient local search implementations in both neighborhoods.

Additionally, we presented the results of computational experiments for a set of benchmark sets. The iterated local search can identify feasible solutions quickly either in neighborhood $N_1$ or in $N_2$. The algorithm which uses neighborhood $N_2$ found optimal solutions for all instances for which the optimal value is already known, and it improved the best-known value for three other instances. These results overcome those obtained by other heuristics.

[1] <https://www.cpubenchmark.net/compare.php?cmp[]=1315&cmp[]=1077&cmp[]=817> (comparison of "Single Thread Rating" values)

[2] <http://cpu.userbenchmark.com/Compare/Intel-Pentium-D-340GHz-vs-Intel-Core-i5-3320M/m5820vsm402> (comparison of "Average User Bench" values)

Table 6 – Comparison of heuristics for type 2 instances.

| | | | | Time (s) | | | | |
|---|---|---|---|---|---|---|---|---|
| $\|V\|$ | $\|E\|$ | $\|C\|$ | Optimal | LS | TT | TS | $B_1$ | $B_2$ |
| 50 | 200 | 3903 | 1636 | 0,266 | 0,444 | 0,578 | 0,200 | 0,510 |
| 50 | 200 | 4877 | 2043 | 0,331 | 0,491 | 0,847 | 0,230 | 0,510 |
| 50 | 200 | 5864 | 2338 | 0,500 | 0,419 | 0,684 | 0,200 | 0,490 |
| 100 | 300 | 8609 | 7434 | 0,953 | 3,125 | 3,516 | 0,660 | 2,390 |
| 100 | 300 | 10686 | 7968 | 1,066 | 2,378 | 3,272 | 0,620 | 2,400 |
| 100 | 300 | 12761 | 8166 | 1,081 | 3,328 | 3,453 | 0,770 | 2,510 |
| 100 | 500 | 24740 | 12652 | 6,009 | 6,647 | 8,450 | 2,550 | 7,960 |
| 100 | 500 | 30886 | 11232 | 7,491 | 6,513 | 7,344 | 2,965 | 9,310 |
| 100 | 500 | 36827 | 11481 | 9,938 | 4,960 | 6,641 | 2,570 | 8,155 |
| 200 | 400 | 13660 | 17728 | 0,863 | 12,441 | 13,260 | 2,155 | 10,840 |
| 200 | 400 | 17088 | 18617 | 0,966 | 11,013 | 16,922 | 2,310 | 10,935 |
| 200 | 400 | 20469 | 19140 | 2,169 | 9,884 | 11,669 | 1,970 | 11,550 |
| 200 | 600 | 34504 | 20716 | 13,063 | 25,632 | 36,132 | 7,040 | 33,050 |
| 200 | 600 | 42930 | 18025 | 12,163 | 22,900 | 42,117 | 7,210 | 31,290 |
| 200 | 600 | 50984 | 20864 | 14,606 | 26,541 | 53,729 | 6,795 | 33,095 |
| 200 | 800 | 62625 | 39895 | 55,135 | 43,810 | 51,391 | 12,970 | 59,930 |
| 200 | 800 | 78387 | 37671 | 59,366 | 47,335 | 46,657 | 18,045 | 67,170 |
| 200 | 800 | 93978 | 38798 | 56,188 | 39,410 | 46,176 | 11,495 | 65,745 |
| 300 | 600 | 31001 | 43721 | 8,002 | 40,106 | 49,069 | 5,225 | 49,945 |
| 300 | 600 | 38216 | 44267 | 4,253 | 49,394 | 67,401 | 9,125 | 50,960 |
| 300 | 600 | 45310 | 43071 | 2,553 | 31,135 | 43,360 | 8,110 | 54,320 |
| 300 | 800 | 59600 | 43125 | 33,144 | 88,516 | 93,951 | 13,620 | 105,210 |
| 300 | 800 | 74500 | 42292 | 28,803 | 59,757 | 83,932 | 17,360 | 110,600 |
| 300 | 800 | 89300 | 44114 | 9,200 | 81,076 | 107,364 | 20,595 | 113,275 |
| 300 | 1000 | 96590 | 71562 | 140,117 | 106,301 | 131,958 | 28,630 | 177,240 |
| 300 | 1000 | 120500 | 76345 | 145,935 | 114,238 | 125,920 | 21,105 | 191,390 |
| 300 | 1000 | 144090 | 78880 | 125,441 | 123,320 | 142,421 | 26,210 | 205,030 |

Source: Research data. The values for "LS", "TT" and "TS" column are adapted from Zhang, Kabadi and Punnen (2011).

CHAPTER

5

# DATA STRUCTURES FOR VARIABLE LOCAL SEARCH

In this chapter, we explore the use of 2LETT in local search algorithms. We first discuss in section 5.1 the relation of the edge exchange operations in evolutionary algorithms as described in Chapter 3 and local search algorithms. Then, in section 5.2 we analyze the performance of predecessor arrays and 2LETT when used in local search algorithms. Finally, we present our final remarks in section 5.3.

## 5.1 Edge exchange

In Chapter 3, we showed that efficient edge exchange mutation operators requires advanced data structures. On the other had, similar operations in the local search algorithm discussed in Chapter 4 are implemented using simple data structures. We discuss in this section why this is the case.

Consider an individual in an EA. Let us trace back the sequence of edge exchanges that lead to this individual from the initial solution and observe how each one was done. We can see that each edge exchange was randomly chosen, without comparing any possible alternative. Therefor, the main computation time to produce an individual is related to find and do one edge exchange.

For local search, the process is different. Each edge exchange is chosen by comparing many options. This scheme allows some computation time to be amortized between all available edge exchange options. Indeed, this is what is done in one-edge and two-edges exchange in Chapter 4, where a pre-processing with $O(n)$ time allows each possible edge exchange to be considered in $O(1)$ time. However, this strategy can be unfeasible if there are few candidate edge exchanges, and so the pre-processing would not be amortized.

This last observation brings the question of which kind of neighborhood would require efficient data structures because a pre-processing scheme would not work. We take $N_1$ and $N_2$ and generalizes it to $N_k(T) = \{T' : T \text{ and } T' \text{ differs in at most } k \text{ edges}\}$. A neighborhood like $N_k$ has been explored in many problems, including in TSP, where the best algorithms (LAWLER, 1985) use it. The first issue with $N_k$ it is size. In the TSP literature, the exploration of $N_k$ is made viable restricting the search. We propose a similar restriction that requires efficient data structures.

Given a spanning tree $T$ of a graph $G = (V, E)$, we restrict the search on $N_k$ to trees than can be obtained from $T$ by a sequence of exchanges $(e_1, f_1), (e_2, f_2), \ldots, (e_k, f_k)$, were the edges $e_1, e_2, \ldots, e_k$ are add to $T$ and the edges $f_1, f_2, \cdots, f_k$ are removed from $T$ and $e_1$ and $f_1$ has a vertex in common, $f_1$ and $e_2$ has a vertex in common and so on.

## 5.2 Data structures performance

We now consider how predecessor arrays and 2LETT perform in searching the $N_k$ neighborhood. The cost to do an edge exchange is $O(\sqrt{n})$ in the average case for predecessor array and $O(\sqrt{n})$ in the worst case for 2LETT.

Although 2LETT without structural sharing is more efficient its asymptotic time did not change. On the other hand, the time for predecessor array decreased from $O(n)$ in the worst case (the array needs to be copied in EA) to $O(\sqrt{n})$ in the average case. Predecessor array was already the best option for random trees with less than $10,000$ vertices. In local search algorithms, we expect it to continue to the best option for random trees for even more vertices.

We also expected that 2LETT to continue to be the best option for trees with large diameter (the worst case for predecessor arrays), but only for very large trees.

## 5.3 Final remarks

We have proposed the $N_k$ neighborhood and discussed how the predecessor array and 2LETT would perform in searching it. The predecessor array has an advantage in random trees while 2LETT in large trees with large diameters.

CHAPTER

6

# CONCLUSIONS

In this thesis, we addressed the problem of designing an efficient data structure for evolutionary and local search algorithms applied to optimization spanning tree problems. Based on the concepts of Euler tours and structural sharing we proposed the 2LETT data structure. We evaluated qualitatively and quantitative the 2LETT and other structures from literature.

In Chapter 3, we analyzed various direct spanning tree representations for evolutionary algorithms and identified the most common mutation operator: the edge exchange. Based on it we defined the two mutation operators CHANGE-ANY and CHANGE-PRED. We evaluated the implementation of these operators using predecessor arrays, NDDR and 2LETT. Predecessor arrays are the most efficient structure for CHANGE-ANY for random trees with less than $10,000$. When dealing with a hard spanning tree problems, it is a good idea to start with predecessor for its simplicity and efficiency. Our proposal of using structural sharing to improve predecessor arrays for CHANGE-PRED, the 2LPredecessor, has the best performance for this operation for random trees with more than $6,000$ vertices. Finally, 2LETT has the best time for the mutation operators, $O(\sqrt{n})$ in the worst-case. Besides, it is the only structure that the running time is independent of the diameter of the tree and has the best performance for CHANGE-ANY for trees with more than $10,000$ vertices. For NDDR we identified three aspects that must be carefully analyzed before using it: 1) trees that cannot be split into equally sized subtrees; 2) a possible bias in the procedure to select operands for mutation operators, and; 3) restriction on the search performed by the operators for one-tree forests.

In Chapter 4 we considered the one-edge and the two-edges exchange neighborhood and its implementation. We showed how to implement a search in each neighborhood in such a way that each neighbor is checked in $O(1)$ time, without using an advanced data structure. Besides, our implementation of the search for two-edges exchange uses a strategy that avoids testing some exchanges that are unfeasible, improving its practical

performance. We used an ILS with each neighborhood to solve the minimum spanning tree problem with conflict constraints. The version using two-edges exchange found all known optimum solutions for a benchmark set and improved the best-known solutions for the other cases with unknown optimum.

In Chapter 5 we investigate a local search algorithm in a neighborhood defined by a variable number of edge exchanges. We showed that, unlike the one-edge and two-edges exchanges neighborhood, the exploration of the variable number of edge exchanges neighborhood requires efficient data structures. For its simplicity, the predecessor arrays still have the best performance for random trees, while 2LETT is only adequate for large trees with large diameters.

## 6.1 Directions for future research

We list in this section some topics that can be further investigated.

Recall that the practical performance of the studies structures is sensitive to implementations details. Specifically for evolutionary algorithms, one aspect that can be better explored is the use of a pool of memory for all structures, not only for the vertex set.

Another direction for future research is the application of the local search algorithm using the variable number of edge exchanges to other problems. Using predecessor array can be very efficient for problems that have trees solutions with small diameters, like the minimum spanning tree with diameter constraint. On the other hand, 2LETT can be efficient for the opposite, problems with trees solutions with large diameters, like the minimum number of branch vertices problem and the Hamiltonian cycle problem.

# BIBLIOGRAPHY


ASSAD, A.; XU, W. The quadratic minimum spanning tree problem. **Naval Research Logistics (NRL)**, Wiley Online Library, v. 39, n. 3, p. 399–417, 1992. Citation on page 69.

BAGWELL, P. **Fast and space efficient trie searches**. [S.l.], 2000. Available: <http://infoscience.epfl.ch/record/64394>. Citation on page 34.

_____. **Ideal hash trees**. [S.l.], 2001. Available: <http://infoscience.epfl.ch/record/64398>. Citation on page 34.

_____. Fast functional lists. In: PEñA, R.; ARTS, T. (Ed.). **Implementation of Functional Languages**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 34–50. ISBN 978-3-540-44854-9. Citation on page 34.

BARBOSA, M. A. L.; DELBEM, A. C. B. Uma busca local iterada para o problema da Árvore geradora mínima sob restrições de conflitos. In: **Anais do XLIX SBPO**. [S.l.]: SOBRAPO, 2017. p. 1–12. Citation on page 69.

BATTITI, R. Partially persistent dynamic sets for history-sensitive heristic. **Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges: Papers Related to the DIMACS Challenge on Dictionaries and Priority Queues (1995-1996) and the DIMACS Challenge on Near Neighbor Searches (1998-1999)**, v. 59, p. 1, 2002. Citation on page 32.

BITTENCOURT, Y. B.; CAMPÊLO, M.; DIAS, F. C. S. Formulação MTZ para o problema da Árvore geradora mínima sob restrições de conflito. In: **Anais do XLVIII SBPO**. [S.l.: s.n.], 2016. p. 2441–2448. Citations on pages 70 and 78.

BOAS, P. van E. Preserving Order in a Forest in Less Than Logarithmic Time. In: **Proceedings of the 16th Annual Symposium on Foundations of Computer Science**. Washington, DC, USA: IEEE Computer Society, 1975. (SFCS '75), p. 75–84. Citation on page 34.

BONDY, A.; MURTY, U. **Graph Theory**. [S.l.]: Springer London, 2011. (Graduate Texts in Mathematics). Citations on pages 19 and 20.

BOROUJERDI, A.; MORET, B. M. Persistency in computational geometry. In: **Proceedings of the Canadian Conference on Computational Geometry (CCCG)**. [S.l.: s.n.], 1995. p. 241–246. Citation on page 32.

BRIANDAIS, R. D. L. File searching using variable length keys. In: **Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference**. New York, NY, USA: ACM, 1959. (IRE-AIEE-ACM '59 (Western)), p. 295–298. Citation on page 34.

BRODER, A. Generating random spanning trees. In: **Proceedings of the 30th Annual Symposium on Foundations of Computer Science**. [S.l.]: IEEE Computer Society, 1989. (SFCS '89), p. 442–447. Citation on page 61.

BUI, T. N.; DENG, X.; ZRNCIC, C. M. An Improved Ant-Based Algorithm for the Degree-Constrained Minimum Spanning Tree Problem. **IEEE Transactions on Evolutionary Computation**, v. 16, n. 2, p. 266–278, 4 2012. ISSN 1089-778X, 1089-778X, 1941-0026. Available: <http://ieeexplore.ieee.org/document/5910378/>. Citation on page 71.

CAMINITI, S.; PETRESCHI, R. String coding of trees with locality and heritability. In: WANG, L. (Ed.). **Computing and Combinatorics**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 251–262. Citation on page 29.

CARRANO, E. G.; FONSECA, C. M.; TAKAHASHI, R. H. C.; PIMENTA, L. C. A.; NETO, O. M. A preliminary comparison of tree encoding schemes for evolutionary algorithms. In: **2007 IEEE International Conference on Systems, Man and Cybernetics**. [S.l.: s.n.], 2007. p. 1969–1974. Citation on page 40.

CAYLEY, A. A theorem on trees. **Quart. J. Math.**, v. 23, p. 376–378, 1889. Citation on page 30.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms, Third Edition**. 3rd. ed. [S.l.]: The MIT Press, 2009. ISBN 978-0-262-03384-8. Citations on pages 41, 47, and 73.

DARMANN, A.; PFERSCHY, U.; SCHAUER, J. Determining a minimum spanning tree with disjunctive constraints. In: **Algorithmic Decision Theory**. [S.l.]: Springer, Berlin, Heidelberg, 2009. p. 414–423. Citations on pages 19, 26, and 70.

DARMANN, A.; PFERSCHY, U.; SCHAUER, J.; WOEGINGER, G. Paths, trees and matchings under disjunctive constraints. **Discrete Applied Mathematics**, Elsevier, v. 159, n. 16, p. 1726–1735, 2011. Citation on page 70.

DELBEM, A. C. B.; CARVALHO, A.; POLICASTRO, C. A.; PINTO, A. K. O.; HONDA, K.; GARCIA, A. C. Node-depth encoding for evolutionary algorithms applied to network design. In: DEB, K. (Ed.). **Genetic and Evolutionary Computation – GECCO 2004**. [S.l.]: Springer Berlin Heidelberg, 2004. (Lecture Notes in Comput. Sci., v. 3102), p. 678–687. Citation on page 47.

DELBEM, A. C. B.; LIMA, T. W. de; TELLES, G. P. Efficient forest data structure for evolutionary algorithms applied to network design. **IEEE Trans. Evolutionary Computation**, v. 16, n. 6, p. 829–846, 12 2012. Citations on pages 20, 32, 38, 39, 40, 41, 47, 48, 49, and 61.

DIETZ, P. F. Fully persistent arrays. In: DEHNE, F.; SACK, J.-R.; SANTORO, N. (Ed.). **Algorithms and Data Structures**. [S.l.]: Springer Berlin Heidelberg, 1989, (Lecture Notes in Computer Science, 382). p. 67–74. Citation on page 34.

DOBKIN, D.; LIPTON, R. J. Multidimensional Searching Problems. **SIAM Journal on Computing**, v. 5, n. 2, p. 181–186, Jun. 1976. ISSN 0097-5397, 1095-7111. Available: <http://epubs.siam.org/doi/10.1137/0205015>. Citation on page 31.

DRISCOLL, J. R.; SARNAK, N.; SLEATOR, D. D.; TARJAN, R. E. Making data structures persistent. **Journal of Computer and System Sciences**, v. 38, n. 1, p. 86–124, 1989. Citations on pages 31 and 33.

FELSENSTEIN, J. **Inferring Phylogenies**. [S.l.]: Sinauer, 2003. ISBN 9780878931774. Citation on page 19.

FREDKIN, E. Trie memory. **Commun. ACM**, ACM, New York, NY, USA, v. 3, n. 9, p. 490–499, Sep. 1960. Citation on page 34.

FREDMAN, M. L.; JOHNSON, D. S.; MCGEOCH, L. A.; OSTHEIMER, G. Data Structures for Traveling Salesmen. **Journal of Algorithms**, v. 18, n. 3, p. 432–479, May 1995. ISSN 0196-6774. Citations on pages 19 and 54.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability; A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1990. Citations on pages 19, 26, and 38.

GEN, M.; CHENG, R. **Genetic algorithms and engineering design**. [S.l.]: Wiley, 1997. (Wiley series in engineering design and automation). ISBN 9780471127413. Citations on pages 27, 28, and 29.

GLOVER, F.; KLINGMAN, D. Finding minimum spanning trees with a fixed number of links at a node. In: ROY, B. (Ed.). **Combinatorial Programming: Methods and Applications**. Dordrecht: Springer Netherlands, 1975. p. 191–201. Citation on page 19.

GOTTLIEB, J.; JULSTROM, B. A.; RAIDL, G. R. Prüfer numbers: A poor representation of spanning trees for evolutionary search. In: **Proceedings of the Genetic and Evolutionary Computation Conference**. [S.l.]: Morgan Kaufmann, 2001. p. 343–350. Citation on page 30.

HENZINGER, M. R.; KING, V. Randomized Dynamic Graph Algorithms with Polylogarithmic Time Per Operation. In: **Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing**. New York, NY, USA: ACM, 1995. (STOC '95), p. 519–527. Citation on page 52.

HU, T. Optimum communication spanning trees. **SIAM Journal on Computing**, v. 3, n. 3, p. 188–195, 1974. Citation on page 19.

JONG, K. A. D. **Evolutionary computation - a unified approach.** [S.l.]: MIT Press, 2006. I-IX, 1-256 p. ISBN 978-0-262-04194-2. Citations on pages 19, 27, 28, and 29.

KAPLAN, H. Persistent data structures. In: MEHTA, D. P.; SAHNI, S. (Ed.). **Handbook of data structures and applications**. [S.l.]: Chapman & Hall/CRC, 2004, (Chapman & Hall/CRC Computer and Information Science Series). Citations on pages 20, 31, and 42.

KORTE, B.; VYGEN, J. Network Design Problems. In: **Combinatorial Optimization: Theory and Algorithms**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 491–525. Citation on page 38.

KRISHNAMOORTHY, M.; ERNST, A. T.; SHARAIHA, Y. M. Comparison of Algorithms for the Degree Constrained Minimum Spanning Tree. **Journal of Heuristics**, v. 7, n. 6, p. 587–611, 11 2001. Citation on page 44.

KRUSKAL, J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. **Proceedings of the American Mathematical Society**, v. 7, n. 1, p. 48–50, 1956. Citations on pages 19, 26, 38, and 75.

LAWLER, E. **The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization**. [S.l.]: John Wiley & sons, 1985. (Wiley-Interscience series in discrete mathematics and optimization). Citations on pages 26 and 84.

LI, Y. An Effective Implementation of a Direct Spanning Tree Representation in GAs. In: **Applications of Evolutionary Computing**. [S.l.]: Springer, Berlin, Heidelberg, 2001. (Lecture Notes in Comput. Sci.), p. 11–19. Citations on pages 38, 40, 41, and 44.

LI, Y.; BOUCHEBABA, Y. A new genetic algorithm for the optimal communication spanning tree problem. In: **Artificial Evolution**. [S.l.]: Springer, Berlin, Heidelberg, 1999. (Lecture Notes in Comput. Sci.), p. 162–173. Citations on pages 40, 41, and 42.

LIBRALAO, G. L.; PEREIRA, F. C.; LIMA, T. W.; DELBEM, A. C. B. Node-depth encoding for evolutionary algorithms applied to multi-vehicle routing problem. In: ALI, M.; ESPOSITO, F. (Ed.). **Innovations in Applied Artificial Intelligence**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. (Lecture Notes in Comput. Sci., v. 3533), p. 557–559. Citation on page 47.

LIMA, T. W. d.; DELBEM, A. C. B.; SOARES, A. d. S.; FEDERSON, F. M.; JUNIOR, J. B. A. L.; BAALEN, J. V. Node-depth phylogenetic-based encoding, a spanning-tree representation for evolutionary algorithms. part i: Proposal and properties analysis. v. 31, p. 1–10, 2016. Citation on page 47.

LIMA, T. W. de; ROTHLAUF, F.; DELBEM, A. C. B. The node-depth encoding: Analysis and application to the bounded-diameter minimum spanning tree problem. In: **Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation**. [S.l.]: ACM, 2008. (GECCO '08), p. 969–976. Citation on page 47.

LOURENÇO, H. R.; MARTIN, O. C.; STÜTZLE, T. Iterated local search: Framework and applications. In: **Handbook of metaheuristics**. [S.l.]: Springer, 2010. p. 363–397. Citation on page 73.

MANSOUR, M. R.; SANTOS, A.; LONDON, J. B.; DELBEM, A. C. B.; BRETAS, N. G. Node-depth encoding and evolutionary algorithms applied to service restoration in distribution systems. In: **IEEE PES General Meeting**. [S.l.: s.n.], 2010. p. 1–8. Citation on page 47.

MICHALEWICZ, Z. **Genetic Algorithms + Data Structures = Evolution Programs**. [S.l.]: Springer, 1996. (Artificial intelligence). Citations on pages 19 and 27.

NARULA, S. C.; HO, C. A. Degree-constrained minimum spanning tree. v. 7, n. 4, p. 239–249, 1980. Citation on page 26.

OKASAKI, C. **Purely Functional Data Structures**. New York, NY, USA: Cambridge University Press, 1998. Citation on page 31.

PALMER, C. C.; KERSHENBAUM, A. Representing trees in genetic algorithms. In: **Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence**. [S.l.: s.n.], 1994. v. 1, p. 379–384. Citation on page 44.

PAPADIMITRIOU, C.; STEIGLITZ, K. **Combinatorial Optimization: Algorithms and Complexity**. [S.l.]: Dover Publications, 1998. (Dover Books on Computer Science Series). ISBN 9780486402581. Citations on pages 20 and 26.

PRIM, R. C. Shortest connection networks and some generalizations. **The Bell System Technical Journal**, v. 36, n. 6, p. 1389–1401, 10 1957. Citations on pages 19, 26, and 38.

PRUFER, H. Neuer bewis eines satzes uber permutationnen. **Arch. Math. Phys.**, v. 27, p. 742–744, 1918. Citation on page 30.

PUENTE, J. P. B. Persistence for the masses: RRB-vectors in a systems language. v. 1, p. 16:1–16:28, 08 2017. ISSN 2475-1421. Available: <http://doi.acm.org/10.1145/3110260>. Citations on pages 33 and 36.

RAIDL, G. R.; DREXEL, C. A predecessor coding in an evolutionary algorithm for the capacitated minimum spanning tree problem. In: **Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference**. [S.l.: s.n.], 2000. p. 309–316. Citation on page 44.

RAIDL, G. R.; JULSTROM, B. A. Edge sets: an effective evolutionary coding of spanning trees. v. 7, n. 3, p. 225–239, 2003. Citations on pages 28, 40, and 41.

RIBEIRO, C. C.; SOUZA, M. C. Variable neighborhood search for the degree-constrained minimum spanning tree problem. **Discrete Applied Mathematics**, v. 118, n. 1, p. 43–54, 4 2002. ISSN 0166-218X. Available: <http://www.sciencedirect.com/science/article/pii/S0166218X01002554>. Citation on page 71.

ROTHLAUF, F. **Representations for genetic and evolutionary algorithms**. 2nd ed. ed. [S.l.]: Springer, 2006. Citations on pages 28, 29, 38, 40, and 41.

ROTHLAUF, F.; GOLDBERG, D. E.; HEINZL, A. Network random keys: A tree representation scheme for genetic and evolutionary algorithms. v. 10, n. 1, p. 75–97, 3 2002. Citations on pages 27 and 65.

RéNYI, A.; SZEKERES, G. On the height of trees. **Journal of the Australian Mathematical Society**, v. 7, n. 04, p. 497–507, Nov. 1967. ISSN 1446-7887, 1446-8107. Citation on page 44.

SAMER, P.; URRUTIA, S. Um algoritmo de branch and cut para árvores geradoras mínimas sob restrições de conflito. In: **Anais do XLV SBPO**. [S.l.: s.n.], 2013. p. 2521–2532. Citation on page 70.

_____. A branch and cut algorithm for minimum spanning trees under conflict constraints. **Optimization Letters**, v. 9, n. 1, p. 41–55, 2015. Citations on pages 70 and 78.

SANTOS, A. C. D.; DELBEM, A. C. B.; BRETAS, N. G. A multiobjective evolutionary algorithm with node-depth encoding for energy restoration. In: **2008 Fourth International Conference on Natural Computation**. [S.l.: s.n.], 2008. v. 6, p. 417–422. Citation on page 47.

SARNAK, N.; TARJAN, R. E. Planar point location using persistent search trees. **Commun. ACM**, v. 29, n. 7, p. 669–679, 1986. Citations on pages 31 and 32.

SLEATOR, D. D.; TARJAN, R. E. Self-adjusting Binary Search Trees. **J. ACM**, v. 32, p. 652–686, Jul. 1985. Citation on page 53.

SOAK, S.; JEON, M. The property analysis of evolutionary algorithms applied to spanning tree problems. **Applied Intelligence**, v. 32, n. 1, p. 96–121, 02 2010. Citation on page 40.

SZEKERES, G. Distribution of labelled trees by diameter. In: CASSE, L. R. A. (Ed.). **Combinatorial Mathematics X**. [S.l.]: Springer Berlin Heidelberg, 1983. (Lecture Notes in Mathematics), p. 392–397. ISBN 978-3-540-38694-0. Citation on page 44.

TARJAN, R. E. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. v. 78, n. 2, p. 169–177, 08 1997. ISSN 0025-5610, 1436-4646. Citations on pages 52 and 53.

TARJAN, R. E.; WERNECK, R. F. Dynamic trees in practice. In: **Experimental Algorithms**. [S.l.]: Springer, Berlin, Heidelberg, 2007. (Lecture Notes in Comput. Sci.), p. 80–93. Citations on pages 20, 39, and 44.

THOMPSON, E.; PAULDEN, T.; SMITH, D. The Dandelion Code: A New Coding of Spanning Trees for Genetic Algorithms. **IEEE Transactions on Evolutionary Computation**, v. 11, n. 1, 2 2007. ISSN 1089-778X. Citation on page 30.

VOSS, S. Capacitated minimum spanning trees. In: FLOUDAS, C. A.; PARDALOS, P. M. (Ed.). **Encyclopedia of Optimization**. Boston, MA: Springer US, 2009. p. 347–357. Citation on page 69.

ZHANG, R.; KABADI, S. N.; PUNNEN, A. P. The minimum spanning tree problem with conflict constraints and its variations. **Discrete Optimization**, v. 8, n. 2, p. 191–205, 2011. Citations on pages 70, 71, 72, 74, 79, 80, and 81.

# ALPHABETICAL INDEX

ICMC USP
SÃO CARLOS