CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM MATEMÁTICA E COMPUTACIONAL

k-IN-A-TREE: AN INTEGER PROGRAMMING APPROACH

LUCAS SALDANHA FERREIRA

Supervisor: Vinicius Fernandes dos Santos
Universidade Federal do Estado de Minas Gerais

BELO HORIZONTE
SEPTEMBER 2024

LUCAS SALDANHA FERREIRA

k-IN-A-TREE: AN INTEGER PROGRAMMING APPROACH

Ph.D. Thesis presented to Programa de Pós-graduação em Modelagem Matemática e Computacional do Centro Federal de Educação Tecnológica de Minas Gerais, as a partial requirement for the title of Ph.D. in Computational and Mathematical Modelling.

Research Area: Computational and Mathematical Modelling

Research Topic: Graphs

Supervisor: Vinicius Fernandes dos Santos
Universidade Federal do Estado de Minas Gerais

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM MATEMÁTICA E COMPUTACIONAL
BELO HORIZONTE
SEPTEMBER 2024

Ferreira, Lucas Saldanha
F383k K-IN-A-TREE: an integer programming approach / Lucas Saldanha Ferreira. – 2024.
xviii, 120 f.

Tese de doutorado apresentada ao Programa de Pós-Graduação em Modelagem Matemática e Computacional.
Orientador: Vinícius Fernandes dos Santos.
Tese (doutorado) – Centro Federal de Educação Tecnológica de Minas Gerais.

1. Teoria dos grafos – Teses. 2. Computação gráfica – Teses. 3. Programação linear – Teses. 4. Algoritmos – Teses. I. Santos, Vinícius Fernandes dos. II. Centro Federal de Educação Tecnológica de Minas Gerais. III. Título.

CDD 519.6

Elaboração da ficha catalográfica pela bibliotecária Jane Marangon Duarte, CRB 6ª 1592 / Cefet/MG

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
COORDENAÇÃO DO PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM MATEMÁTICA E COMPUTACIONAL

"K-IN-A-TREE: AN INTEGER PROGRAMMING APPROACH".

Tese de Doutorado apresentada por **Lucas Saldanha Ferreira**, em 16 de outubro de 2024, ao Programa de Pós-Graduação em Modelagem Matemática e Computacional do CEFET-MG, e aprovada pela banca examinadora constituída pelos professores:

Prof. Dr. Vinícius Fernandes dos Santos
Centro Federal de Educação Tecnológica de Minas Gerais

Prof. Dr. Rafael Augusto de Melo
Universidade Federal da Bahia

Prof. Dr. Cristiano Arbex Valle
Universidade Federal de Minas Gerais

Profª. Drª. Elisangela Martins de Sá
Centro Federal de Educação Tecnológica de Minas Gerais

Prof. Dr. Flávio Vinícius Cruzeiro Martins
Centro Federal de Educação Tecnológica de Minas Gerais

Visto e permitida a impressão.

Prof. Dr. Thiago de Souza Rodrigues
Coordenador do Programa de Pós-Graduação Stricto Sensu em Modelagem Matemática e Computacional

To my parents, who ensured that I never went without, always providing everything I needed.

To the Bros++, for the unfiltered support, motivation, and occasionally questionable advice.

To my Dota crew, for the endless entertainment and lifelong friendship.

In particular, to my life partner, Livia Dias, for her (mostly) unwavering support and love throughout this journey.

And finally, to my ever-faithful furballs, Ugli and Kissy, for always being by my side — quite literally — as I worked on this.

Acknowledgements

The authors thank and acknowledge Professor **Cristiano Arbex Valle** from DCC/UFMG for making his personal optimization framework available.

CNPq, **CAPES**, **FAPEMIG**, **CEFET-MG** and **UFMG** for their support.

“One in two PhD students experiences psychological distress; one in three is at risk of a common psychiatric disorder. This estimate is significantly higher than those obtained in any other profession.”

Levecque, Katia, et al. "Work organization and mental health problems in PhD students." Research Policy 46.4 (2017): 868-879.

Resumo

Chudnovsky e Seymour (2010) propuseram o algoritmo THREE-IN-A-TREE, que resolve o seguinte problema em tempo polinomial: dado um grafo simples e finito e três vértices fixos, verificar se existe uma árvore induzida contendo esses vértices. Neste trabalho, lidamos com uma generalização natural desse problema, denominada k-IN-A-TREE. Este problema pergunta se um grafo contém uma árvore induzida que abrange k vértices predeterminados. Quando k faz parte da entrada, o problema é conhecido por ser NP-completo, se $k \geq 4$ é um número fixo, sua complexidade está aberta. Entretanto existem algoritmos eficientes para casos restritos, como grafos livres de garras, grafos com circunferência pelo menos k e grafos cordais. Neste trabalho apresentamos formulações de programação inteira mista para o problema em questão e mostramos que instâncias com até 25.000 vértices podem ser resolvidas em tempo computacional razoável. Também agregamos mais de três anos de tempo computacional em experimentos projetados para explorar quais variáveis impactam o tempo de execução das várias instâncias geradas; essas análises fornecem um perfil para o problema k-IN-A-TREE em sua forma mais genérica.

As contribuições deste trabalho incluem demonstrar que o k-IN-A-TREE pode ser eficientemente resolvido usando Programação Linear Inteira (ILP), permitindo a solução de instâncias de tamanhos relativamente grandes. Além disso, apresentamos análise detalhada que foi conduzida para identificar e caracterizar as principais variáveis que influenciam a dificuldade computacional do problema, fornecendo dados valiosos sobre como a estrutura do grafo, tamanho, conectividade dos vértices e a distribuição dos elementos obrigatórios, afetam a dificuldade. Outra contribuição significativa é a demonstração de que o problema pode ser resolvido de forma eficaz em intervalos de tempo que seriam insuficientes para resolver outros problemas combinatórios complexos, como o Problema do Caixeiro Viajante (TSP). Isso destaca a tratabilidade computacional do problema em ambientes com restrições de tempo, onde outros problemas de complexidade semelhante permanecem mais desafiadores. Essas contribuições avançam o entendimento do problema, estabelecendo um novo marco para a habilidade de resolver o problema e abrindo caminho para futuras explorações teóricas e práticas.

Palavras-chave: k-IN-A-TREE. Subgrafo induzido. Programação inteira. THREE-IN-A-TREE.

Abstract

Chudnovsky and Seymour (2010) proposed the THREE-IN-A-TREE algorithm, which solves the following problem in polynomial time: given three fixed vertices in a simple finite graph, check whether an induced tree containing these vertices exists. In this paper, we deal with a generalization of this problem, referred henceforth as k-IN-A-TREE. The k-IN-A-TREE asks whether a graph contains an induced tree spanning k given vertices. When k is part of the input, the problem is known to be NP-complete. If $k \geq 4$ is a fixed given number, its complexity is an open question, although there are efficient algorithms for restricted cases such as claw-free graphs, graphs with girth at least k, and chordal graphs. We present mixed-integer programming formulations for this problem and show that instances with up to $25{,}000$ vertices can be solved in reasonable computational time. We also aggregate over three years of computational time in experiments designed to explore which variables impact the execution time of the several generated instances; these analyses provide a profile for the k-IN-A-TREE problem in its most generic form.

The contributions of this work include demonstrating that the k-IN-A-TREE problem can be efficiently solved using Integer Linear Programming (ILP), allowing for the solution of instances of relatively large sizes. In addition, a detailed analysis was conducted to identify and characterize the key variables influencing the computational hardness of the problem, providing valuable insights into how the graph's structure, including factors such as size, vertex connectivity, and the distribution of mandatory elements, affects its solvability. Another significant contribution is the demonstration that the problem can be effectively solved within time intervals that would be insufficient for solving other complex combinatorial problems, such as the Traveling Salesman Problem (TSP). This highlights the computational tractability of k-IN-A-TREE in time-constrained environments, where other similarly complex problems remain more challenging. These contributions advance the understanding of the problem, setting a new benchmark for its solvability and paving the way for future theoretical and practical explorations.

Keywords: k-IN-A-TREE. Induced Subgraph. Integer programming. THREE-IN-A-TREE.

List of Figures

List of Tables

List of algorithms

List of abbreviations and acronyms

TSP	Traveling Salesman Problem
LP	Linear Programming
ILP	Integer Linear Programming
FVS	Feedback Vertex Set
MTZ	Miller-Tucker-Zemlin
MWIF	Maximum Weighted Induced Forests
MWFVS	Minimum Weighted FVS
DFJ	Dantzig-Fulkerson-Johnson
UFMG	Universidade Federal do Estado de Minas Gerais
DCC/UFMG	Departamento de ciência da computação da UFMG
CEFET-MG	Centro Federal de Educação Tecnológica de Minas Gerais

Contents

1 Introduction

Graphs are a fundamental structure in both theoretical computer science and applied fields, providing versatile ways to model relationships and interactions in diverse domains. From representing communication networks, biological systems, and social interactions to optimizing transportation and logistical networks, graphs are vital tools for modeling, understanding, studying, and solving both real-world and theoretical problems. Inside the study of graphs, the area dedicated to the detection of subgraphs and of structural properties of graphs plays a crucial role in understanding the properties of the target problems or meta-problems.

One important problem in this area is the detection of specific substructures within graphs, which often serve as indicators of key properties or constraints. One example is the THREE-IN-A-TREE problem, proposed by Chudnovsky and Seymour (CHUDNOVSKY; SEYMOUR, 2010), which involves determining whether a simple graph contains an induced subtree that spans three given vertices. Formally, given a graph $G = (V, E)$ and three vertices $u, v, w \in V$ the problem asks whether there exists an induced subtree T of G that includes these three vertices. This problem holds theoretical significance and serves as a building block for understanding more complex graph properties.

The algorithm proposed by Chudnovsky and Seymour solves the THREE-IN-A-TREE problem in polynomial time. It has since been used to test for the presence of certain forbidden subgraphs, such as thetas and pyramids. In subsequent work, Lévêque (LÉVÊQUE et al., 2009) extended this approach, demonstrating that the algorithm could be applied to detect entire classes of forbidden induced subgraphs. More recently, Chang (CHANG; LU, 2015) employed the THREE-IN-A-TREE framework to identify specific structures in their recognition algorithm for even-hole-free graphs, further illustrating the broad applicability of this problem.

This thesis explores a natural and significant generalization of this problem: the k-IN-A-TREE problem. This generalized problem asks whether, given a graph, there exists an induced subtree that spans k chosen vertices, where $k \geq 3$. While the case for $k = 1$ and $k = 2$ can be shown to be trivial, the difficulty of the problem grows rapidly with a larger k. When the number of fixed vertices k is part of the input, the problem is known to be NP-Complete (DERHY; PICOULEAU, 2009), posing significant computational challenges. When, instead, k is fixed, the complexity is unknown for $k \geq 4$.

This work introduces and compares three distinct integer programming formulations for solving the k-IN-A-TREE problem. Through a series of computational experiments, we demonstrate the scalability of our formulations, solving instances with up to $25{,}000$ vertices

within reasonable computational times. These contributions lay a new understanding for the practical solvability of the problem and also improve the general knowledge of the hardness of the problem for different combinations of variables.

1.1 Main contributions

Most existing studies on the k-IN-A-TREE problem and related graph problems focus on specific instances or subclasses of graphs, such as claw-free graphs, cliques, or other restricted graph families. These studies have provided valuable insights into particular cases, often leading to specialized algorithms that exploit the structural properties of these restricted graph types. In contrast, this thesis takes a broader approach by addressing the k-IN-A-TREE problem in its more general setting. Instead of restricting the problem to specific graph classes, it explores a generalized framework where the problem can arise in any arbitrary graph without structural constraints. This shift in focus allows for a more comprehensive understanding of the problem's complexity and solvability, independent of predefined graph families/structures.

By extending the study to this unrestricted version of the k-IN-A-TREE problem, the work presented here offers more widely applicable insights across diverse graph structures. This differentiation from previous studies highlights the novelty of this thesis, as it tackles the computational challenges associated with solving the problem in its full generality. This allowed us to make several key contributions to the study of the problem, particularly in computational approaches, complexity analysis, and comparative performance with other combinatorial optimization problems. The main contributions are as follows:

Integer programming approach for large instance sizes: A significant contribution of this work is the demonstration that the k-IN-A-TREE problem can be efficiently solved using integer programming techniques in practice. By formulating the problem as an integer linear program (ILP), the authors were able to solve instances of the problem up to relatively large sizes. This approach provides a systematic and scalable method for tackling the problem and highlights the effectiveness of ILP in addressing specific structural properties of graph-based problems.

Characterization of problem hardness variables: A detailed analysis was conducted to identify and characterize the key variables influencing the problem's computational hardness. This characterization contributes to a broader understanding of how the problem's structure, including factors such as graph size, vertex connectivity, and the distribution of mandatory elements, affects its solvability. Identifying these critical variables may provide valuable insights for theoretical analysis and practical problem-solving strategies.

Efficiency in time-constrained environments: Another contribution is demonstrating that the problem can be effectively solved within time intervals smaller than expected, when

compared with instances of the same size being applied to other graph related problems such as the traveling salesman or the Steiner tree. This finding underscores the comparative computational tractability of the k-IN-A-TREE problem and positions it as a problem that can be tackled efficiently under time constraints where other problems with similar theoretical computational complexity remain more challenging. This result can have implications for developments requiring rapid solutions in dynamic or resource-limited environments.

Together, these contributions advance the field's understanding of the k-IN-A-TREE problem, establishing a new benchmark for its solvability, not only broadening the scope of practical applications but also lays the groundwork for future explorations. These contributions are grounded in carefully designed experiments and their observations. Given the exploratory nature of this work, the focus is not on providing mathematical proofs but rather on investigating the behavior of the problem to gain insights and build a solid understanding before dedicating further effort to formal mathematical demonstrations.

1.2 Thesis' outline

This thesis is organized as follows:

Chapter 1, the current chapter, introduces the key topics and establishes the context for the research developed. It begins with a motivational introduction, highlighting the significance and challenges of the k-in-a-tree problem in graph theory. This chapter presents an overview of the main contributions and outlines the structure of the thesis, providing a roadmap for the subsequent chapters.

Chapter 2 lays the theoretical groundwork for the thesis by reviewing the fundamental concepts and principles of graph theory, with a particular focus on the properties of graphs and trees that are essential for developing the forthcoming integer linear programming models. It also comprehensively reviews the relevant literature, focusing on the original THREE-IN-A-TREE problem and the k-IN-A-TREE problem. This review also explores related computational problems, highlighting their similarities and differences with the main problem. By examining these analogous problems, the chapter identifies potential strategies and insights that may apply to the current study. Finally it establishes the core terminology and notation that will be consistently used throughout the thesis, ensuring clarity and precision in the research presentation. For readers unfamiliar with graph theory or those needing a refresher, this chapter serves as a valuable starting point to build the necessary background to fully understand and engage with the mathematical models and computational strategies introduced in the subsequent chapters.

Chapter 3 focuses on constructing and elucidating the mathematical models proposed to solve the problem. These models are formulated as integer optimization models, designed to ensure that the conditions stipulated by the problem are met while searching

for suitable solutions. Given that the problem generates a binary outcome – either there is at least one possible solution, or there is none – the models are specifically tailored to address this yes/no scenario. Additionally, it discusses the rationale behind the model design, including the selection of constraints and variables that effectively capture the essence of the problem.

Chapter 4 describes the tests conducted and their results. The proposed models are thoroughly analyzed and compared with proper analysis and potential explanations for the observed results. Insights into the behavior of both the models and the problem are extracted from the data. These insights reveal aspects of the problem's inherent complexity independent of the solution method. The chapter also attempts to isolate and dissect variables related to the problem's computational hardness, allowing for an extensive discussion of their impacts. Through comparisons and visual analysis, this chapter seeks to elucidate the dynamics of the problem, enhancing the reader's comprehension of the factors that influence the hardness and feasibility of the k-IN-A-TREE problem.

Chapter 5 synthesizes the conclusions and accomplishments of the research, articulating the implications of the findings discussed in previous chapters. It emphasizes the advancements and contributions made by the study to the field, highlighting how the initial research questions are addressed. It discusses the theoretical and practical significance of the results, noting some limitations incurred on the work. Furthermore, this chapter outlines potential future work, suggesting directions for subsequent research to build upon the findings laid by this thesis. This includes possible improvements to the methodologies employed and new areas of investigation opened by the findings.

Appendix A presents a collection of all comparison charts used to evaluate and study the algorithms developed in this research and infer information from the problem itself. While the most relevant charts are presented and discussed in the main text, this appendix serves as a repository for all generated comparisons, including those not included in the main discussion, due to redundancy or lack of additional insights. By gathering these results in one place, Appendix A provides a convenient reference for further examination, ensuring that the full spectrum of data is accessible for future analysis or spot checks.

2 Problem definition and related works

This chapter provides an overview of essential definitions and notations, including fundamental principles of graph theory, such as the formal definitions of induced subgraphs, trees, and cliques, among others. This review is crucial, as it establishes the foundational properties of the concepts that will be extensively utilized in subsequent chapters. The discussion will specifically focus on characteristics pertinent to this thesis.

Furthermore, this chapter formally defines the k-IN-A-TREE problem, setting the stage for the modeling and analysis in later chapters. It also includes a review of related works and discusses similar problems in the literature, providing context for the current study. The chapter concludes with a detailed presentation of notations, assumptions, and terminologies that will persist throughout this work. This is of paramount importance for the correct interpretation of the concepts and methodologies discussed in future chapters.

2.1 An overview of relevant graph concepts

This section provides a concise introduction to fundamental graph definitions and terminologies. It is designed to acquaint readers who may be unfamiliar with these terms and their implications. Additionally, it directs the attention of those well-versed in graph theory to the most crucial properties relevant to each discussed situation. This approach ensures all readers have a solid foundation and clear understanding as they progress through the discussions in the subsequent sections.

2.1.1 Simple graphs

A simple graph is a type of graph where no two vertices can have more than one edge between them, and no edge can connect a vertex to itself. This means that a simple graph contains no loops (edges that begin and end at the same vertex) and no multiple edges between any pair of distinct vertices. Formally, a simple graph $G = (V,E)$ consists of a set V of vertices and a set E of edges, where each edge is an unordered pair of distinct vertices from V. Simple graphs are commonly used in graph theory because they represent basic relationships, making them suitable for analyzing foundational graph properties and algorithms.

On the other hand, a complex graph refers to any graph that includes additional features not present in simple graphs. This could involve multiple edges between the same pair of vertices (a multigraph) and/or edges that start and end at the same vertex (loops). These added complexities may be needed for modeling more intricate systems where the

relationships between entities may not be simple. Complex graphs allow for a more nuanced representation of these systems, facilitating the development of algorithms tailored to handle such detailed structures (ZHANG; CHARTRAND, 2006).

2.1.2 Connectivity

Another concept needed for our discussion includes the fundamental concepts of acyclicity and connectivity, which underpin our subsequent definition of a tree.

A graph is considered connected if, for every pair of vertices in the graph, there exists at least one path that links them. In other words, a connected graph is one where every vertex is reachable from any other vertex within the graph. This property ensures that there is no isolated subgroup of vertices within the graph; all vertices are part of a single cohesive structure. By definition, a graph consisting of a single vertex is inherently connected, as there are no other vertices to connect. When a graph contains two or more vertices, connectivity exists if and only if there is at least one edge directly or indirectly linking each vertex to every other vertex.

As expected, a graph that does not satisfy this criterion is termed disconnected. In a disconnected graph, there exists at least one pair of vertices for which no path can be found connecting them. This can occur if the graph is divided into multiple disjoint subgraphs, where each subgraph is connected within itself but has no edges linking it to other subgraphs. Additionally, a graph with multiple vertices but no edges is also disconnected, as no paths exist between any of the vertices.

Formally, an undirected graph $G = (V, E)$ is defined as disconnected if there exists a pair of vertices u,$v \in V$ such that no path exists between u and v. If, for every pair of vertices u,$v \in V$, a path does exist, the graph is defined as connected. Specifically, an undirected graph G is defined as disconnected if it contains at least two vertices for which no path exists connecting them as endpoints (GROSS; YELLEN, 2004).

Figure 1 shows both a connected and a disconnected graph. The disconnected graph has one vertex (y_2) and one subgraph ($y_8 3, y_8$) that are not connected to the other by any paths, thus rendering it disconnected.

2.1.3 Aciclicity

In graph theory, a cycle is a path that consists of a sequence of edges and vertices such that the starting and ending vertices are the same, effectively forming a loop (GROSS; YELLEN, 2004). More specifically, a cycle is a closed path where the only repeated vertex is the first and last vertex of the sequence. This concept is crucial in distinguishing different types of graph structures, particularly in determining whether a graph contains any cyclical patterns.

(a) Connected (b) Disconnected

Figure 1 – Example of a connected and a disconnected graph

A simple cycle is a specific type of cycle in which no vertex or edge is repeated along the path, except for the vertex at which the cycle begins and ends (EVEN, 2011). In a simple cycle, the path traverses distinct edges and vertices, ensuring that the cycle is not composed of any redundant loops or repetitions. For clarity and consistency throughout this thesis, the term "cycle" will exclusively refer to a simple cycle unless specified otherwise.

An acyclic graph is defined as a graph that contains no cycles. In other words, it is a graph in which there is no way to start at a vertex and follow a sequence of edges that eventually loops back to the starting vertex without repeating edges. Acyclic graphs are significant in various applications, including tree structures, which will be explored later, where the absence of cycles is a defining characteristic.

Figure 2 illustrates examples of both cyclic and acyclic graphs. The example of a cyclic graph presented includes three distinct cycles:

- $C_0 = \{y_2, y_5, y_7, y_4, y_2\}$
- $C_1 = \{y_2, y_5, y_7, y_6, y_4, y_2\}$
- $C_2 = \{y_4, y_6, y_7, y_4\}$

(a) Cyclic (b) Acyclic

Figure 2 – Example of both cyclic and acyclic graphs

2.1.4 Tree

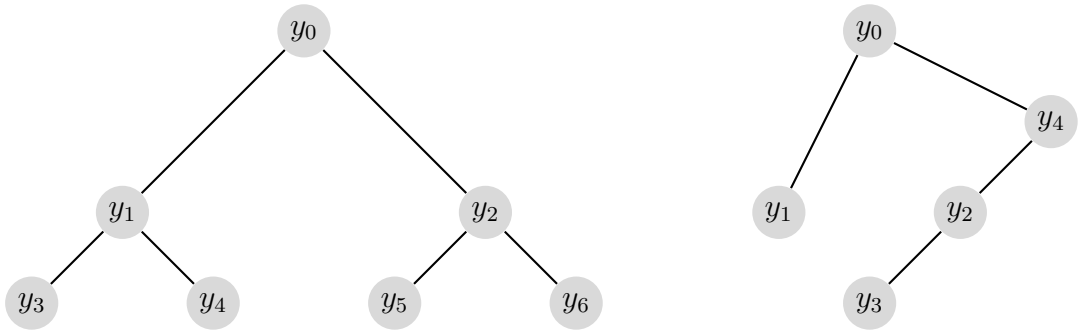
The concept of a tree is a fundamental one in graph theory, serving as the basis for many algorithms and theoretical results discussed in this thesis as it is present in the name of the problem. A tree is defined as an acyclic-connected graph; hence, every acyclic-connected graph qualifies as a tree, and vice versa (DIESTEL, 2005). Alternatively, a tree can also be characterized as an undirected graph in which any two vertices are connected by exactly one path (DIESTEL, 2005). The absence of cycles ensures no redundant connections between vertices, while the connectiveness guarantees a path linking every pair of vertices in the graph. This dual property is what makes trees particularly useful for representing hierarchical structures, such as organizational charts, file systems, and decision processes, where a clear, non-redundant connection between elements is essential.

The structural properties of trees can be quantitatively expressed as where $V(G) - E(G) = 1$, where $V(G)$ and $E(G)$ represent the number of vertices and edges, respectively (GROSS; YELLEN, 2004). This relationship is a direct consequence of defining a tree as an undirected graph where a unique path connects each pair of vertices. This property serves as the foundation for the algorithms discussed in this paper.

For a finite graph, this definition implies a connected graph with $n - 1$ edges, which is also equivalent to a finite graph without simple cycles and $n - 1$ edges (WEST et al., 1996); these alternative definitions will play a crucial role in the integer modeling in this thesis. Graphs with no vertices are not included in these considerations.

There are several types of tree, one noteworthy that will be explored later is the binary tree. It consists of a root node connected to two subtrees, one to the "left" and the other to the "right," each of which can be a single node or another binary tree. Binary trees are commonly used in algorithms for searching, sorting, and structuring hierarchical data, and can be balanced or unbalanced depending on the distribution of nodes.

Figure 3 illustrates various types of trees, including a binary tree with seven vertices and six edges and a path with five vertices and four edges.



(a) Binary tree (b) A path (every path is a tree)

Figure 3 – Two different types of trees

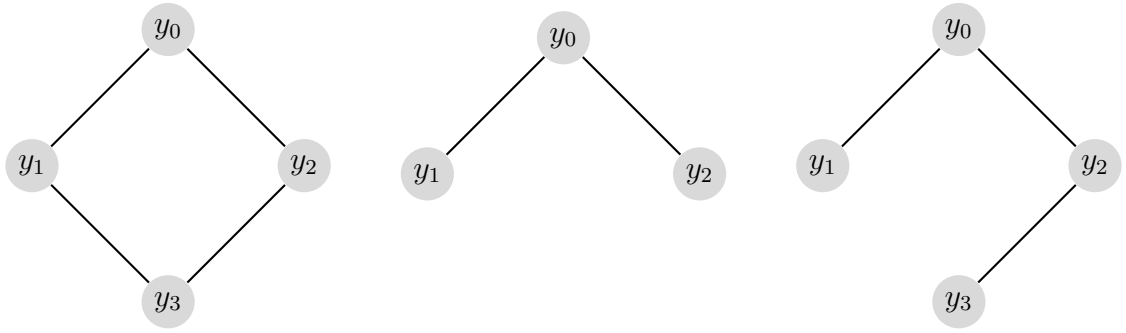
2.1.5 Induced subgraph definition

Another essential concept in graph theory that warrants discussion is the notion of an induced subgraph, which is central to the problem addressed in this thesis.

An induced subgraph of a graph consists of a selected subset of the vertices of the original graph, along with all the edges that connect these vertices in the original graph (EVEN, 2011). To clarify, H is considered an induced subgraph of G if the following conditions are both satisfied:

1. $V(H) \subseteq V(G)$
2. $E(H) = \{uv \in E(G) \mid u,v \in V(H)\}$.

Where $V(X)$ denotes the set of vertices of a graph. $E(X)$ denotes the set of edges of a graph.



(a) Original graph G (b) An induced subgraph (c) A non-induced subgraph

Figure 4 – Positive and negative examples of induced subgraphs

Figure 4 shows an example of an induced subgraph and a subgraph that is not induced. Per definition, if both vertices y_1 and y_3 are in the solution and there is an edge connecting them in the original graph, that edge should also exist in an induced subgraph.

There are many algorithmic questions of interest concerning the existence of an induced subgraph of some specific type containing some form of mandatory vertices, but almost all of them seem to be NP-complete (BIENSTOCK, 1991).

2.1.6 Induced subtree definition

An induced subtree is a graph that is both an induced subgraph of a graph G and a tree, complying to the restrictions in both sections 2.1.5 and 2.1.4, respectively.

2.2 Problem definition

This section formally defines the core problem addressed in this thesis. While the ultimate focus of this thesis is on the k-IN-A-TREE problem, understanding the principles

and solutions associated with the THREE-IN-A-TREE problem is crucial, as it serves as the conceptual and methodological basis for the generalization explored in this work. Therefore, this section is divided into two subsections: the first provides a detailed explanation of the THREE-IN-A-TREE problem, setting the stage for the k-IN-A-TREE problem in sequence.

2.2.1 The THREE-IN-A-TREE problem

Probably the most relevant related work, the THREE-IN-A-TREE inspired the k-IN-A-TREE problem. The problem statement reads: given a simple graph and three arbitrary vertices belonging to that graph, check whether an induced subgraph that is both a tree and contains the three chosen vertices exists, or more formally:

THREE-IN-A-TREE
INSTANCE: A graph G and three vertices v_1, v_2, v_3 of G.
QUESTION: Is there $X \subseteq V(G)$ with $v_1, v_2, v_3 \in X$ such that the subgraph induced by X is a tree ?

The problem requires a binary (yes/no) response. Figure 5 illustrates an example instance of the problem, with the three input vertices specified as y_1, y_5, and y_9. A potential solution, X, is depicted in blue and consists of the induced subgraph and a tree containing $X = \{y_0, y_1, y_4, y_5, y_7, y_9\}$. It is crucial to note that y_2 cannot be used to connect y_1 to y_9 instead of using y_0 and y_7 as this would form the cycle y_1, y_2, y_4, which would invalidate the solution since it would not meet the tree definition, this scenario is demonstrated in Figure 6. Moreover, removing the edge (y_4, y_2) to prevent the aforementioned cycle would still result in an invalid solution, as the resultant graph would no longer be an induced subgraph of the original, as shown in Figure 7.

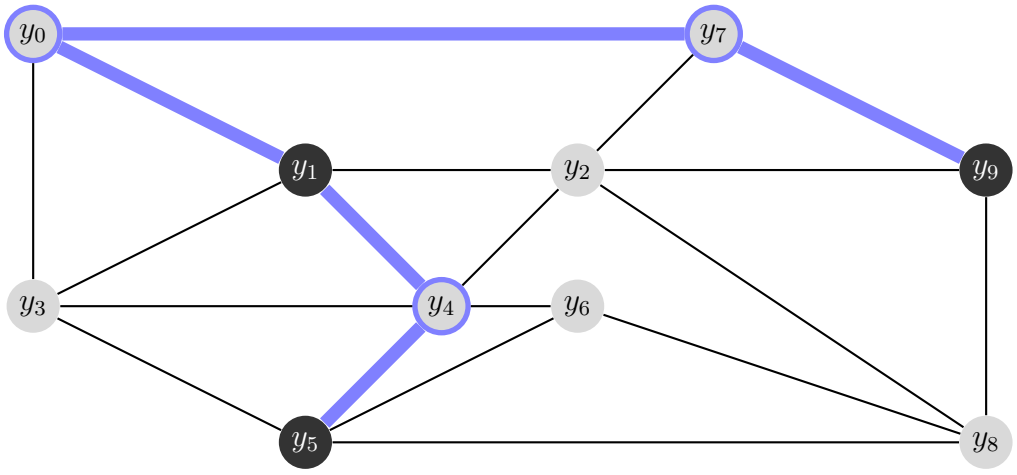


Figure 5 – A THREE-IN-A-TREE problem instance. Mandatory vertices are highlighted in black. One possible solution is highlighted in blue

As mentioned, THREE-IN-A-TREE is fundamentally a feasibility problem. The objective is straightforward: either finding one viable solution or confirming that no possible solutions

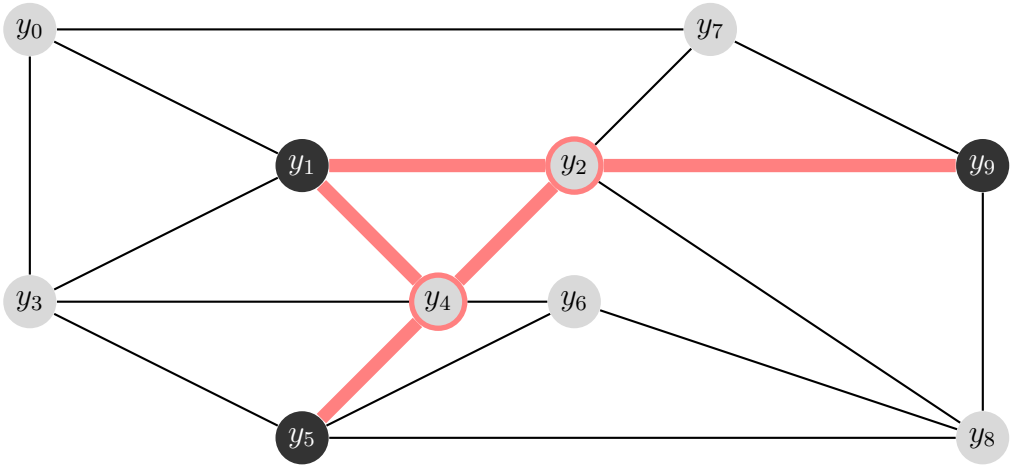


Figure 6 – A THREE-IN-A-TREE problem instance. Mandatory vertices are highlighted in black. The highlighted induced subgraph is not a valid solution as it is not a tree.

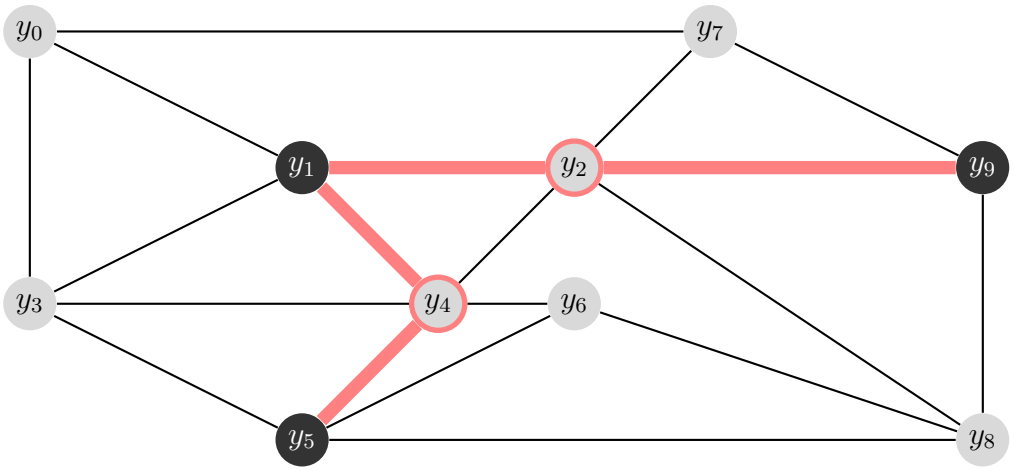


Figure 7 – A THREE-IN-A-TREE problem instance. Mandatory vertices are highlighted in black. The highlighted subgraph is not a valid solution as it is not an induced subgraph.

exist. These problems do not require finding the minimal subgraph or maximizing the number of vertices, allowing for multiple valid solutions under the given conditions. Figure 8 illustrates another possible positive solution for the same graph depicted in Figure 5. Finding either solution would equally classify the instance as having a solution without any of those being superior to the other.

Initially approached by Chudnovsky and Seymour (CHUDNOVSKY; SEYMOUR, 2010), they proposed an algorithm that solves this problem in polynomial time. The problem was shown to be solvable in time $O(|V(G)|^4)$ (CHUDNOVSKY; SEYMOUR, 2010), later improved to $O(|V(G)|^2 \log |V(G)|)$ by Lai et al. (LAI; LU; THORUP, 2020). For most graphs, one would expect a "yes" answer for the THREE-IN-A-TREE problem, but there are interesting graphs for which the answer is "no," as noted in the original paper. The technique was inspired by Chudnovsky's previous research on deciding if a graph contains a theta (CHUDNOVSKY; KAPADIA, 2008), and it was used for some formulations of her work on Berge

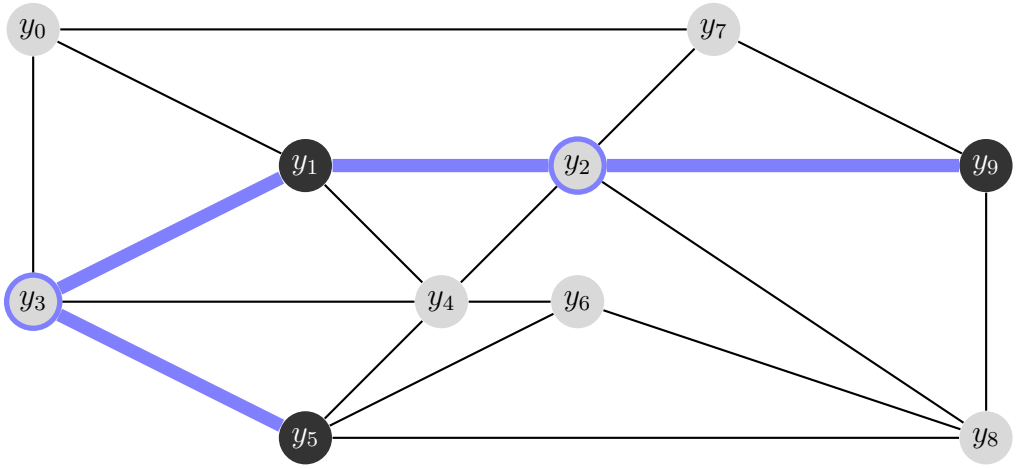


Figure 8 – Another solution for the same THREE-IN-A-TREE problem shown in Figure 5

graphs (CHUDNOVSKY* et al., 2005). The original authors also used this algorithm as a tool for testing for certain forbidden induced subgraphs such as thetas and pyramids. Lévêque showed that the THREE-IN-A-TREE algorithm could be used to test for whole classes of forbidden induced subgraphs (LÉVÊQUE et al., 2009). While Chang used the tool to help recognize even-hole-free graphs (CHANG; LU, 2015).

Unfortunately, most of these works and algorithms cannot be directly applied to the generalized version of the problem, as the number of mandatory vertices is unknown. However, as a subset of the problem approached in this thesis, the THREE-IN-A-TREE problem shares many similarities and insights that helped dissect its more generic form.

2.2.2 The k-IN-A-TREE problem

A natural generalization of the THREE-IN-A-TREE problem would be to remove the limitation of three vertices. The k-IN-A-TREE problem consists of checking for the existence of an induced subtree that contains k chosen vertices. We define the k-IN-A-TREE problem formally as:

k-IN-A-TREE
INSTANCE: A graph G and a set $K \subseteq V(G)$.
QUESTION: Is there an induced subgraph T of G, with $K \subseteq V(T)$ such that T is a tree?

It is essentially the same feasibility problem from the previous subsection: our objective is to identify whether any solution exists. For $k = 1$ and $k = 2$, the problem can be solved by just checking whether the vertices belong to the same connected component. For $k = 3$ Chudnovsky found a polynomial solution (CHUDNOVSKY; SEYMOUR, 2010). Thus, this problem is more interesting when k is any natural $k > 3$. When the number of fixed vertices k is part of the input, the problem is known to be NP-Complete (DERHY; PICOULEAU, 2009).

When k is fixed instead, the complexity is unknown for $k \geq 4$. Gomes et al. studied the parameterized complexity of the problem and, despite not answering the central question about the complexity for a fixed k, they showed that it is unlikely that the complexity can be detached from the input parameters; furthermore, complexities smaller than $O(n^k)$ are, most likely, unattainable (GOMES et al., 2020).

Addressing NP-hard problems typically involves several strategies: developing exact algorithms that are effective but only feasible for small-sized problems, creating approximation or heuristic algorithms that provide reasonably good solutions within acceptable time frames, or identifying specific conditions or special cases of the problem where more effective or exact heuristics can be applied (CRESCENZI; KANN; HALLDÓRSSON, 1995). Within this latter approach, certain algorithms capitalize on specific graph types and properties to enhance the proposed problem's management or variants. For instance, the literature includes efficient algorithms for specific subgraphs containing k prescribed vertices, such as testing for an induced cycle when $k \geq 2$ (BIENSTOCK, 1991), a minimum induced tree for $k \geq 3$ (DERHY; PICOULEAU, 2009), and an induced path for $k \geq 3$ (HAAS; HOFFMANN, 2006).

For restricted cases of the k-IN-A-TREE problem, there are efficient algorithms in the literature, such as claw-free graphs (FIALA et al., 2012), graphs of the girth of at least k (LIU; TROTIGNON, 2010) and chordal graphs (SANTOS; SILVA; SZWARCFITER, 2015). All those works presented specialized algorithms and formulations that took advantage of unique and well-formulated properties of the respective cases to deliver polynomial solutions for specific situations.

Those papers have provided deep insights into potential behaviors of the problem under varied conditions and have been instrumental in shaping the research direction. However, as those approaches largely depend on pre-existing knowledge about the graph's properties, they could not be directly modified or adapted to address the generalized version of the problem tackled in this thesis, which seeks to solve the problem without relying on any prior knowledge about the graph's specific properties. This challenge highlights a central focus of the thesis — addressing the problem in its full generability thoroughly and systematically.

The k-IN-A-TREE problem has been rarely approached as a generic problem. Few studies assume nothing on the input and seek to understand the profile of the problem and not the worst-case complexity for a pre-determined sub-problem of interest. This makes it difficult to determine an initial approach from an algorithmic perspective and to identify which variables have the most significant impact on execution time. As mentioned before, in some situations, the complexity of the problem is unknown; when k is part of the input, the problem is known to be NP-Complete; and there are studies with efficient sub-problems solutions, but as far as we know, there is no empirical study aimed in investigating which properties of

the graph or the mandatory vertices affect the hardness of the instance.

2.3 Related problems

This subsection examines several problems that share similarities with the current study. Many of these related issues have served as the foundation or inspiration for developing the algorithm proposed in this thesis. While not extensively covered in existing literature, the k-IN-A-TREE problem finds its nearest parallels in these correlated problems. As a result, these analogous problems serve as the most pertinent sources of literature and have significantly shaped our approach's conceptual and methodological foundations. This analysis situates our work within the larger field context and traces the development of the algorithmic solutions that stem from these related studies.

2.3.1 The traveling salesman problem

An important problem closely related to the one under study is the Traveling Salesman Problem (TSP), particularly concerning the integer formulations found in existing literature. The initial research phase on our problem began with adaptations of integer TSP formulations, this approach provided insights, spurred further research, and led to the development of our current formulations.

The TSP asks how to determine the optimal route (typically the shortest) for visiting a specified set of cities and then returning to the starting city. Recognized as an NP-hard problem, it plays a significant role in combinatorial optimization, operations research, theoretical computer science, and graph theory.

The Traveling Salesman Problem can be formally defined as follows: Given a complete graph $G = (V, E)$ where V represents the set of cities and E represents the set of direct routes between each pair of cities, and a weight function $w : E \to \mathbb{R}^+$ that assigns a non-negative cost to travel between each pair of cities, the goal is to find a Hamiltonian cycle (a cycle that visits each city exactly once and returns to the starting city) with the minimum total cost. Considering x_{ij} a binary variable that indicates if the path from i to j is part of the solution or not, the TSP can be formulated as

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} w_{ij} x_{ij}, \\
\text{subject to} \quad & \sum_{i=1, i \neq j}^{n} x_{ij} = 1 \quad \forall j, \\
& \sum_{j=1, j \neq i}^{n} x_{ij} = 1 \quad \forall I, \\
& u_i - u_j + n x_{ij} \leq n - 1 \quad \forall i \neq j,\ i, j \in \{2, \ldots, n\}, \\
& x_{ij} \in \{0, 1\}, \quad \forall i, j \in N.
\end{aligned}
$$

Here, the second set of constraints ensures that each city is entered precisely once, and the first set ensures that each city is left exactly once. The subtour elimination constraint (third) is necessary to ensure that the solution does not consist of multiple disjoint subtours, thus forming a single tour covering all cities. It is fundamentally a minimization problem that requires starting and finishing at a designated vertex, having visited each other vertex exactly once(APPLEGATE et al., 2006). It is typically represented by a complete graph, meaning every pair of vertices is connected by an edge. In cases where no direct path exists between two nodes, an arbitrarily long edge may be added to ensure the graph remains complete, thereby not altering the optimal tour. The essence of solving the TSP is to find a Hamiltonian cycle of the minimum possible weight (HASSIN; RUBINSTEIN, 2000). The stipulation of returning to the starting city does not alter the computational complexity of the problem (ALLENDER et al., 2009).

Additionally, the TSP can be expressed as an integer linear program (DANTZIG, 2016). Among the known formulations, the Miller-Tucker-Zemlin (MTZ)(MILLER; TUCKER; ZEMLIN, 1960) and the Dantzig-Fulkerson-Johnson (DFJ)(DANTZIG; FULKERSON; JOHNSON, 1954) formulations are particularly noteworthy. While the DFJ formulation is generally considered stronger, the MTZ formulation retains its utility under specific conditions. The initial formulation explored in this thesis is an adaptation of the MTZ constraints, details of which will be elaborately discussed in subsequent chapters.

2.3.2 Minimum Spanning Tree problem

Another similar problem that has been solved using integer programming is the Minimum Spanning Tree (MST) problem. It consists of finding a spanning tree whose sum of edge weights is as small as possible, a problem that has many formulations and has been extensively studied.

Given a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$ assigned to each edge, the objective of the MST problem is to find a spanning tree of the graph such that the sum of the weights of the edges in the tree is minimized. A spanning tree is a subset of G that is a tree and includes every vertex of G, with the minimum possible number of edges.

Several algorithms are known for solving the MST problem efficiently, with the two most famous being Kruskal's algorithm and Prim's algorithm. Kruskal's Algorithm approaches the problem by adding edges in order of increasing weight, ensuring that no cycles are formed. It utilizes a disjoint-set data structure to keep track of the components in the growing forest to ensure that each edge connects two distinct components (KRUSKAL, 1956). Prim's algorithm, alternatively, starts from an arbitrary vertex and grows the MST by adding the cheapest edge from the tree being built to a vertex not yet in the tree. It can be efficiently implemented using priority queues (PRIM, 1957). Both algorithms run in polynomial time and are classic examples of greedy algorithms, which build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit.

The MST problem is also theoretically significant in studying algorithmic graph theory, serving as a prototype for problems and algorithms in the more general class of matroids. Its study has led to important advancements in understanding greedy algorithms and has applications in telecommunications and transportation. Overall, the Minimum Spanning Tree problem presents interesting theoretical challenges and offers substantial practical utility by providing optimal solutions for connecting networks in the most economical way (BONDY; MURTY et al., 1976).

2.3.3 The Steiner tree problem

In the context of weighted graphs, consider an undirected graph $G = (V, E)$ with non-negative edge weights c. Let $S \subseteq V$ denote a subset of vertices known as terminals. A Steiner Tree is defined as a tree within G that spans all vertices in S. The corresponding optimization challenge is to find a minimum-weight Steiner Tree (GILBERT; POLLAK, 1968). This problem can be viewed as a generalization of both the non-negative shortest path problem and the minimum spanning tree problem (PAOLINI; STEPANOV, 2013).

This problem is categorized as NP-hard, highlighting its computational complexity due to the exponential number of potential configurations as the inclusion of Steiner vertices increases. The computational challenge lies in determining which vertices, if any, should be added to the subset of terminals to minimize the total edge weight of the spanning tree, which makes the problem notably more complex than the classic MST problem. Specifically, when the Steiner tree contains exactly two terminals, it simplifies to the shortest path problem. Conversely, if all vertices in G are terminals, the problem becomes equivalent to finding the minimum spanning tree. Although solutions for both the shortest path and minimum

spanning tree problems exist within polynomial time, the Steiner tree problem is NP-complete, indicating that its optimization variant is NP-hard (CRESCENZI; KANN; HALLDÓRSSON, 1995).

Despite its NP-hard status, certain restricted cases of the Steiner tree problem can be addressed in polynomial time. In practice, various Steiner tree problem variants are solvable efficiently, even for large-scale real-world applications such as circuit layout and network design. These practical applications often involve adaptations, leading to numerous variants of the Steiner tree problem (HWANG, 1976; IVANOV; TUZHILIN, 1994). These include algorithms inspired by the previously mentioned Kruskal's and Prim's MST algorithms, adapted to incorporate potential Steiner points effectively. Additionally, advanced preprocessing techniques simplify the graph by removing unnecessary vertices and edges, thereby reducing the problem size and computational demand. These strategies aim to produce good-enough solutions within a feasible time frame, acknowledging the trade-off between exactitude and practicality in solving NP-hard problems (WARME; WINTER; ZACHARIASEN, 2000; DU; SMITH; RUBINSTEIN, 2000).

The Steiner Tree problem has significantly inspired some of the formulations developed in this work, particularly due to its tree constraint and the requirement to include certain terminal vertices within such trees, mirroring several restrictions of the k-IN-A-TREE problem.

2.3.4 Feedback Vertex Set & Maximum Weighted Induced Forest problems

Another similar problem is the Feedback Vertex Set (FVS) problem, a fundamental problem in graph theory and computer science, particularly in the fields of network analysis and algorithm design. The problem involves identifying a set of vertices within a graph such that, by removing these vertices, the remaining subgraph contains no cycles. In other words, the objective is to transform a given graph into a forest (a disjoint union of trees) since trees are acyclic connected graphs. This problem is theoretically intriguing and has practical applications in various areas, including circuit testing, deadlock prevention in operating systems, and reducing complexity in network analysis (FESTA; PARDALOS; RESENDE, 1999).

Formally, given a graph $G = (V, E)$, where V represents the vertices and E the edges, a feedback vertex set is a subset $V' \in V$ such that the subgraph $G - V$ (obtained by removing the vertices in V' and their incident edges from G) is acyclic. The computational challenge lies in determining the minimum feedback vertex set, the smallest set V' that achieves this. The problem is known to be NP-complete, indicating that no efficient algorithm is likely to exist for solving all instances of this problem optimally within polynomial time (BRUNETTA; MAFFIOLI; TRUBIAN, 2000).

In practical scenarios, algorithms and heuristics are developed to approximate the

minimum feedback vertex set, particularly for large-scale graphs typical in real-world applications. Techniques from various areas of computer science, such as approximation algorithms, fixed-parameter tractability, and heuristic methods, are applied to manage and simplify the complexities involved in finding feedback vertex sets. This has led to a rich body of research aiming to balance between the computational efficiency and the accuracy of the solutions provided by these algorithms (CHEN et al., 2007).

While the FVS problem seeks a set of vertices whose removal results in a cycle-free graph, it directly correlates with the tree and induced subgraph restriction from the k-IN-A-TREE problem; in practice, the set of candidate solutions for FVS is a superset of solutions for k-IN-A-TREE. Melo studied the minimum weighted FVS (MELO; QUEIROZ; RIBEIRO, 2021), presenting formulations that are a good starting point for exploring possible k-IN-A-TREE heuristics.

Another good starting point is the Maximum Weighted Induced Forests (MWIF) problem, which is equivalent to the Minimum Weighted FVS (MWFVS) problem, often considered a dual problem. Both address issues of optimizing weights under specific structural constraints within a graph but from opposing perspectives. The MWIF problem seeks to maximize the sum of the weights of the vertices in an induced forest, while the MWFVS problem aims to minimize the sum of the weights of vertices in a feedback vertex set.

To understand the equivalence between these problems, consider a graph $G = (V, E)$ with vertex weights assigned by a function $w : V \to \mathbb{R}^+$. The goal of the MWIF problem is to select a subset of vertices $V' \subseteq V$ such that the induced subgraph $G[V']$ is a forest (i.e., acyclic), and the sum of the weights of the vertices in V' is maximized. Formally, the objective is to maximize $\sum_{v \in V'} w(v)$. Conversely, in the MWFVS problem, the objective is to remove the smallest-weighted subset of vertices $V'' \subseteq V$ such that the remaining subgraph $G - V''$ is also a forest. The goal here is to minimize $\sum_{v \in V''} w(v)$.

These problems are equivalent because solving one provides a solution to the other. Specifically, if V'' is an optimal solution to the MWFVS problem, then $V' = V \setminus V''$ will be an optimal solution to the MWIF problem. This is because the total weight of the graph $\sum_{v \in V} w(v)$ is fixed, and maximizing the weight of the induced forest V' is equivalent to minimizing the weight of the vertex set V'' removed to create that forest. In other words, the weight of V' in the MWIF problem is $\sum_{v \in V} w(v) - \sum_{v \in V''} w(v)$, which maximizes $\sum_{v \in V'} w(v)$ by minimizing $\sum_{v \in V''} w(v)$ (RAZGON, 2006).

This duality extends to algorithmic strategies and heuristics used to tackle these problems, especially in large and complex graphs where exact solutions are computationally infeasible. Techniques such as greedy algorithms, dynamic programming, and approximation algorithms that efficiently solve one problem can often be adapted to address the other.

Thus, in the same manner, that FVS formulations are good research starting points for

the k-IN-A-TREE, the same applies to MWFVS. In particular, we sought inspiration from Melo and his co-authors, who expanded on their previous work and compared five approaches to the problem through extensive computational experiments (MELO; RIBEIRO, 2022).

2.4 Notations

In this section, we outline the key notations used throughout the thesis ensuring clarity and consistency in the upcoming chapters.

Let $V(G)$ and $E(G)$ represent the set of vertices and edges, respectively, of a graph G. An edge between vertices i and j is denoted by $\{i, j\}$.

Henceforth, we assume that any graph G that is an input for the k-IN-A-TREE is always connected. If we don't, then either:

- all the mandatory vertices are on the same connected component, and therefore, we can ignore all other non-connected components as they cannot be a part of any tree containing all mandatory vertices, or
- there are at least two vertices in the set of mandatory vertices that are in different connected components, making the solution trivial as it is impossible to find an induced subtree that contains all mandatory vertices.

Since those situations simplify the problem to a connected component or make the problem trivial, we will assume that G is always connected.

As stated before, the k-IN-A-TREE problem is a decision problem; the objective is to decide if at least one possible tree exists. An answer is positive (negative) if a feasible solution is (not) found; if an k-IN-A-TREE problem instance is positive, it is called a yes-instance. Otherwise, it is a no-instance.

Trivial instances are defined as those where the solution to the problem is readily apparent. Examples include graphs in which mandatory elements are located within disconnected components or cases where the mandatory elements themselves already form a tree or a clique. These instances are less engaging from a research perspective because the solution can be determined simply by examining the mandatory vertices, effectively disregarding most other graph elements. This straightforward approach eliminates the complexity typically associated with more challenging graph configurations.

Finally we also assume that all graphs are finite, undirected, and simple unless stated otherwise.

3 Integer programming modeling of the problem

The efficacy of our approach hinges on the robustness of the modeling techniques employed. This chapter is dedicated to a detailed exposition of the developed integer programming models that are central to addressing the research problem efficiently. The formulations presented herein are engineered to solve the problem within a feasible computational timeframe. This chapter will not only elucidate several choices that were made when building the equations but also reference the inspirations that have shaped the construction of the constraints. To maintain clarity and coherence across multiple model formulations, all equations have been numbered and will be consistently referred to by these identifiers throughout the chapter.

This chapter introduces three integer programming formulations aimed at the efficient resolution of the k-IN-A-TREE problem, which fundamentally assesses the existence of a solution rather than its enumeration. Given that the sought solution is an induced tree, our models enforce acyclicity and connectivity through different strategies, since those are essential characteristics of any admissible solution. We first present a compact formulation, followed by two other formulations that contain an exponential number of constraints.

3.1 MTZ-based formulation

The first formulation is based on the Miller-Tucker-Zimler (MTZ) formulation of the Travelling Salesman Problem (TSP) (MILLER; TUCKER; ZEMLIN, 1960), which uses a directed graph. Accordingly, an arborescence (a rooted tree) is a feasible solution, which contains all vertices in K.

In regards to an induced subgraph that characterizes a solution to the k-IN-A-TREE problem, we change its definition slightly. If vertices i and j are part of the induced subgraph, then at least one arc among (i,j) and (j,i) must also be part of the subgraph.

The original MTZ formulation for the TSP requires the definition of an arbitrary vertex from which a tour starts. Accordingly, we define a vertex as the arborescence root. As only vertices in K are guaranteed to be part of the arborescence, we arbitrarily choose $r \in K$ to be the root.

Our MTZ-based formulation involves the following zero-one decision variables:

- $x_{ij} = 1$ if arc (i,j) is part of the arborescence, zero otherwise,
- $y_i = 1$ if vertex i is part of the arborescence, zero otherwise,

We also make use of decision variables u_i, which are required by the MTZ constraints and whose meaning is explained later in this section.

Let $n = |V|$ be the number of vertices in D. Let $A = \{(i,j) \in V \times V : \{i,j\} \in E(G)\}$. The compact formulation, denoted as **P1**, is given by:

$$\min \sum_{i=1}^{n} y_i - \sum_{(i,j)\in A} x_{ij} \tag{1}$$

subject to:

$$y_i = 1 \qquad \forall i \in K \tag{2}$$

$$x_{ij} \leq y_i \qquad \forall \{i,j\} \in E \tag{3}$$

$$x_{ij} \leq y_j \qquad \forall \{i,j\} \in E \tag{4}$$

$$x_{ij} + x_{ji} \geq y_i + y_j - 1 \qquad \forall \{i,j\} \in E \tag{5}$$

$$x_{ij}, x_{ji} \in \mathbb{B} \qquad \forall \{i,j\} \in E \tag{6}$$

$$y_i \in \mathbb{B} \qquad \forall i \in V \tag{7}$$

$$\sum_{i=1, i\neq r}^{n} x_{ir} = 0 \tag{8}$$

$$\sum_{i\in N(j)} x_{ij} = y_j \qquad \forall j \in V, j \neq r \tag{9}$$

$$u_r = 1 \tag{10}$$

$$2 \leq u_i \leq n \qquad \forall i \in V, i \neq r \tag{11}$$

$$u_i - u_j + 1 \leq n(1 - x_{ij}) \qquad \forall (i,j) \in A \tag{12}$$

The objective function (1) is the difference between the number of vertices and the number of arcs in the arborescence. Any connected arborescence T containing $|T|$ vertices has exactly $|T| - 1$ arcs, which implies an objective value of 1. Any acyclic but not connected graph composed of at least two arborescences must have an objective value greater than 1 as each arborescence contributes 1 to the objective value. As long as acyclicity is enforced, any tree (acyclic and connected), if it exists, must be optimal concerning minimizing (1).

Constraints (2) assure that all mandatory vertices (those in K) must be included in any feasible solution. Constraints (3)-(5) ensure that the solution is a proper induced subgraph. Constraints (3) and (4) make sure that an edge can only be selected if both its vertices are selected. Accordingly, constraints (5) make sure that if an arc is selected than its two endpoints must also be selected. Constraints (6)-(7) define variables bounds.

Constraints (8)-(9) enforce the arborescence structure. Constraint (8) ensures that the arborescence is rooted at r, and constraints (9) ensure that any selected vertex is

reached by exactly one incoming arc.

Constraints (10)-(12) ensure acyclicity through a slight modification of the original MTZ constraints for the TSP. The constraints enforce that if $x_{ij} = 1$, then u_j must be at least a unit higher than u_i. This ensures that any cycle induces an unfeasible solution as for any path $i, j, \ldots, k$ in the arborescence, we must have $u_i < u_j < \ldots < u_k$.

Applying the formulation to any problem, the smallest possible value for the objective function will be 1, and this value will be achieved if, and only if, there is at least one feasible solution for the problem. The algorithm execution is halted if the objective function reaches 1 as the problem is decided.

This formulation was based on the MTZ subtour elimination known to produce weak relaxations (LANGEVIN; SOUMIS; DESROSIERS, 1990; PADBERG; SUNG, 1991). It is reasonable to assume that our adaptation could be strengthened with adaptations of known techniques such as tightening the relaxation (SHERALI; DRISCOLL, 2002; DIETZ; ADAMS; YANG, 2019) lifting the subtour elimination constraints (DESROCHERS; LAPORTE, 1991; KARA; LAPORTE; BEKTAS, 2004) or adding valid inequalities (BEKTAŞ; GOUVEIA, 2014), our objective is not to optimize the formulation's performance. Instead, this paper aims to explore the problem itself and gain insights into its behavior. Therefore, we opted for the simpler version to maintain clarity and focus on the problem's characteristics. It is also noteworthy to mention that (4) is guaranteed by (9); still, we chose to keep it for consistency, as it has no significant impact on performance and simplifies the description of upcoming formulations that will rely on constraints (2)-(7).

3.2 Ensuring acyclicity with an exponential number of constraints

The MTZ-based formulation introduced earlier is polynomially bounded in relation to its number of variables and constraints. Nevertheless, for the TSP at least, the MTZ constraints are well known for providing weaker linear relaxation bounds than traditional subtour elimination constraints (LAPORTE, 1992). In this section, we show how the MTZ constraints can be replaced with an alternative formulation based on an exponential number of constraints.

We consider again the original undirected graph G with edge set E where a feasible solution to the k-IN-A-TREE problem is a single (undirected) tree containing all vertices in K, instead of an arborescence. Similarly to the previous case, our new formulation employs decision variables y_i and x_{ij}. As before, variable y_i indicates if the vertex i is on the solution or not and edge $\{i, j\}$ is part of the solution if exactly one of x_{ij} or x_{ji} have value 1. Note that since variables x_{ij} and x_{ji} correspond to the same edge, and we enforce $x_{ij} + x_{ji} \leq 1$

(constraint (14)). The careful reader may notice that creating only one variable per edge would be enough in this case. Indeed, this is the way we implemented it, but for a unified notation for all formulations, we decided to keep one variable for each ordered pair.

The formulation below, henceforth referred to as **P2**, is given by minimizing (1) subject to (2)-(7) and

$$\sum_{i \in S} y_i \leq |V(S)| - 1 \qquad \forall\, S \subseteq V \,:\, G[S] \text{ contains at least one cycle.} \tag{13}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall\, \{i,j\} \in E(G). \tag{14}$$

The objective function (1) is the subtraction between the number of vertices and the number of edges in the subgraph. In an acyclic graph (1) will represent the number of connected components on the graph, with 1 being optimal when minimizing (1), which indicates an acyclic connected graph, also known as a tree. This can be rationalized as any connected arborescence T that contains $|T|$ vertices must have precisely $|T| - 1$ arcs, implying a value of 1 for the objective function. On the other hand, any acyclic but not connected graph with at least two arborescences must have an objective value greater than one, as each arborescence contributes 1 to the objective value.

Note that despite the change in interpretation of variables x_{ij}, (2)-(7) still enforce the property of being an induced subgraph. Combined with constraint (13), have the effect of causing every subgraph S of G that contains at least one cycle to have, at most, $|S| - 1$ vertices that are in the solution, effectively rendering the solution acyclic. However, due to the impractical number of restrictions that Constraint (13) would generate for even moderately sized graphs, we employ a Branch-and-cut algorithm (PADBERG; RINALDI, 1987) (PADBERG; RINALDI, 1991) for separating these constraints and solving **P2**.

Let $\tilde{y}_i, i \in V$ and $\tilde{x}_{ij}, \{i,j\} \in E$ be the values assigned to each decision variable in the partial integer solution at the current node of the Branch-and-cut tree of **P2**. Let $\tilde{G} = (\tilde{V}, \tilde{E})$ be a subgraph of G such that $i \in \tilde{V} \,:\, \tilde{y}_i > 0$ and $\{i,j\} \in \tilde{E} \,:\, \tilde{x}_{ij} > 0$ (*i.e.* the induced subgraph defined by the current solution at the node). In the process of separating constraints (13) we use depth-first search (DFS) in $\tilde{G}$ to locate existing cycles. Upon finding a cycle, the DFS execution is stopped, the corresponding constraint is added, and the algorithm restarts with the new constraint, limiting all cycle elements from simultaneously being part of future solutions.

The pseudo-code for verifying the existence of a cycle, adding the relevant restriction, and triggering a new process to find a solution for the integer problem is shown in Algorithm 1.

Using this approach, only one constraint can be added per iteration. This also means that only constraints that were eventually violated during the algorithm execution are added since the DFS is applied only to $\tilde{G}$, a candidate solution, at each step. Although there is

Algoritmo 1: Minimization problem with DFS for detecting cycles

```
Data: Graph G
Result: A cycle in the graph G, if one exists
Function DFS(G, v):
    Mark v as visited (grey);
    foreach For each neighbor w of v in graph G do
        if w is not visited then
            DFS(G,w);
        else
            return cycle(v, w);
        end
    end
    Mark v as completely explored (black);
    return;
Function Main(G):
    Solve(G);
    cycle = DFS(G, k_0);
    if cycle exists then
        addCycleConstraint(cycle);
        Main(G);
    else
        return;
    end
```

potential for an exponential number of constraints, we expect that, for most of the instances, the algorithm will complete its execution, having added only a small fraction of those, which may reduce the execution time. For performance reasons, the execution is halted if the objective function ever comes to the value of 1 as, by that point, the problem is already decided, as there is at least one solution for the problem characterizing a positive answer for that instance.

3.3 Ensuring connectivity with an exponential number of constraints

In the previous formulation, we had a minimization problem in which acyclicity is enforced with constraints (13). The other alternative, denoted as **P3**, is to use constraints to enforce connectivity, with the same objective function swapped to a maximization problem, as by doing that, any optimal solution that reaches an objective value of 1 is guaranteed to be acyclic.

Let $\delta^+(W) = \{\{i,j\} \in E(G) : \{i,j\} \cap W \neq \emptyset$ and $\{i,j\} \setminus W \neq \emptyset\}$ be the set of neighbors of W, let $k_0 \in K$ be an arbitrarily chosen starting vertex and $k_i \in (K - k_0)$.

Formulation **P3**, is defined as:

$$\max \sum_{i=1}^{n} y_i - \sum_{(i,j)\in A} x_{ij} \tag{15}$$

subject to (2)-(7), (14) and

$$\sum_{\{i,j\}\in\delta^+(W)} x_{ij} + x_{ji} \geq 1 \qquad \forall\, W \subset V(G), k_0 \in V(G) \setminus W, K \cap W \neq \emptyset. \tag{16}$$

The objective of Constraint (16) is to separate $V(G)$ into multiple sets of two connected complementary partitions. For each set of connected complementary partitions, k_0 will be in one component and k_i in the other. Enforcing this for every (k_0, k_i) pair results in a restriction that ensures that k_0 is connected to all k_i, and by consequence, that all elements of K are connected.

Similarly to **P2**, this formulation (16) can be achieved by a potentially exponential number of constraints. As done before, we attempted to improve it by reducing those potentially exponential constraints to the problem of finding the maximum flow in a directed graph.

3.3.1 Max-flow min-cut

The Max Flow Min Cut Theorem is a fundamental concept in network flow theory. The theorem articulates a relationship between a flow network's maximum flow and the minimum cut. Specifically, it states that in a flow network, the maximum value of the flow from a source node s to a sink node t is equal to the minimum capacity that, when removed from the network, would completely sever the source from the sink.

Mathematically speaking, given a network $G = (V,E)$ with a capacity $c(u,v)$ on each edge $(u,v) \in E$, the maximum flow f from s to t is calculated by maximizing the sum of the flows on all edges leading out of the source, subject to the capacity constraint $f(u,v) \leq c(u,v)$ and flow conservation constraint at each vertex (excluding s and t). Formally defined as:

$$\max \quad f = \sum_{v:(s,v)\in E} f(s,v) \tag{17}$$

subject to:

$$f(u,v) \leq c(u,v) \quad \forall (u,v) \in E \tag{18}$$

$$\sum_{v:(u,v)\in E} f(u,v) = \sum_{v:(v,u)\in E} f(v,u) \quad \forall u \in V \setminus \{s,t\} \tag{19}$$

Conversely, a 'cut' in the network is defined as a partition of V into two disjoint subsets S and T with $s \in S$ and $t \in T$ such that removing the edges crossing from S to T disconnects t from s. The capacity of a cut is the sum of the capacities of the edges crossing the cut, and the minimum cut is the cut whose capacity is minimal among all such cuts.

$$c(S,T) = \sum_{u\in S, v\in T} c(u,v) \tag{20}$$

The Max Flow Min Cut Theorem states that the maximum value of the flow from s to t is equal to the capacity of the minimum cut separating s and t:

$$\max f = \min c(S,T) \tag{21}$$

The Max Flow Min Cut theorem can be algorithmically implemented using polynomial approaches like the Ford-Fulkerson (FORD; FULKERSON, 1956) method or its optimized version, the Edmonds-Karp algorithm (EDMONDS; KARP, 1972). These algorithms iteratively search for augmenting paths (paths where additional flow can be pushed) and adjust the flows until no more augmenting paths can be found, at which point the flow value equals the capacity of the minimum cut. The minimum cut on equation (21) will be used on the next section to define a new constraint for the nominal problem.

3.3.2 Applying Max-flow min-cut to enforce connectivity

For separating constraints (16), the Max-flow Min-cut algorithm is used at the integer solution on the branch-and-bound tree of **P3**. Since polynomial algorithms are known for that problem, there is a clear path forward. For the implementation itself, we used the Ford-Fulkerson algorithm (FORD; FULKERSON, 1956).

To build the necessary flow graph $\tilde{G}$, we use a weighted version of G; if an edge is not present at the current integer solution, its weight is 1; otherwise, it is ∞. To define the source and the sinks let $C = (S,T)$ be a (k_0,k_i)-cut, a partition of $V(\tilde{G})$ such that $k_0 \in S$ and one $k_i, i \neq 0 \in T$. The cut-set X_C of a cut C is, in a Max-flow problem, the set of edges that connect the k_0 (flow source) part of the cut to k_i (flow sink).

$$X_C := \{(k_0,k_i) \in E(K) \ : \ k_0 \in S, k_i \in T\} = (S \times T) \cap E(K) \ \forall i \neq 0. \tag{22}$$

After calculating the cut capacities for each (k_0,k_i) pair, as defined on equation (21), we add the corresponding restrictions if its capacity is smaller than ∞. Adding at most $|K|-1$

constraints at each step. When the minimal cut of all (k_0,k_i) pairs are connected, the same process can be applied again on the next executions. This process only takes place after all of k is connected since solving for K usually also solves for several other vertices as part of the process, saving some execution time.

Once again, the formulation can potentially generate an exponential number of constraints during its execution. But, as before, they are not likely to be added before the algorithm answers the problem. However, a critical distinction in this formulation is that the algorithm execution cannot be stopped when 1 is reached as a value of the objective function. Multiple disconnected subgraphs can achieve this value, such as a simple cycle and a single isolated vertex.

Algorithm 2 shows the proposed solution. The algorithm used to solve the max-flow min-cuts problem was the Ford–Fulkerson algorithm (FORD; FULKERSON, 1956) (HEINEMAN; POLLICE; SELKOW, 2016).

Algoritmo 2: Maximization problem with Max Flow constraints

Data: G
Result: Cuts C to be added in current solution
Function `Main(`G`)`**:**
    Solve(G);
    if *Solution is connected* **then**
        return;
    else
        foreach *Pair* (k_0, $k_{n\neq 0}$) **do**
            possibleCut = MaxFlow(k_0, k_n);
            if *possibleCut.isViolated()* **then**
                AddViolatedCut(possibleCut);
            end
        end
    end
    Main(G);
    return;

To elucidate the proposed algorithm, an illustrative example is provided. Consider the graph depicted in Figure 9 as the input for this demonstration. When the algorithm is applied without connectivity constraints, one potential outcome is shown in Figure 10. This result represents an induced subgraph that includes all mandatory elements and is maximal with respect to $V(G) - E(G)$, comprising 7 vertices and 2 edges, achieving an objective function value of 5. However, it lacks connectivity due to the absence of constraints to enforce this property. It is noteworthy that alternative solutions could achieve the same objective function value, such as the exclusion of a vertex from the current solution. For illustrative purposes,

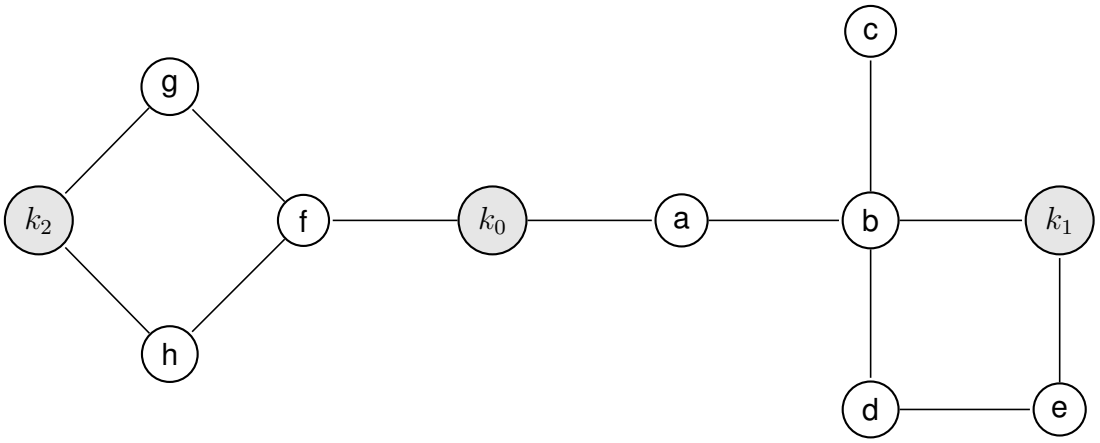


Figure 9 – Hypothetical input graph to exemplify the usage of Max-flow min-cuts for generating connectivity constraints

this specific configuration is selected.

As the solution is computed, a new graph is generated, as shown in Figure 11. This graph mirrors the original input graph but adjusts the edge weights based on their presence in the partial solution. Subsequently, the minimum cut algorithm is applied to each pair (k_0, k_n), with k_0 as the source and k_n as the sink, in this case the pairs (k_0, k_1) and (k_0, k_2). The computed minimal cuts are illustrated in red in the figure, with the cut between (k_0, k_1) valued at 1 and (k_0, k_2) at 2. It is important to note that (k_0, k_2) has other possible min-cuts with the same value, but we chose the best one for exemplification.

Following the identification of these cuts, constraints are imposed on the model to ensure that at least one edge from each cut is included in all subsequent solutions. Specifically, the constraints added are:

$$x_{ab} \geq 1 \tag{23}$$

and

$$x_{fg} + x_{fh} \geq 1 \tag{24}$$

With these constraints in place, the algorithm is re-executed, and this iterative process continues until a conclusive solution is derived. For this specific example, no more steps would be needed since enforcing those two constraints would mean adding b to the solution, which would enforce the connection with k_1, and adding either g or h would also cause the other constraints to enforce a connection with k_2, thus solving the problem, both g and h would not be added simultaneously as this would reduce the value of the objective function. However, for larger graphs and as needed, this process would only generate another intermediary solution that would go through the process as many times as needed.

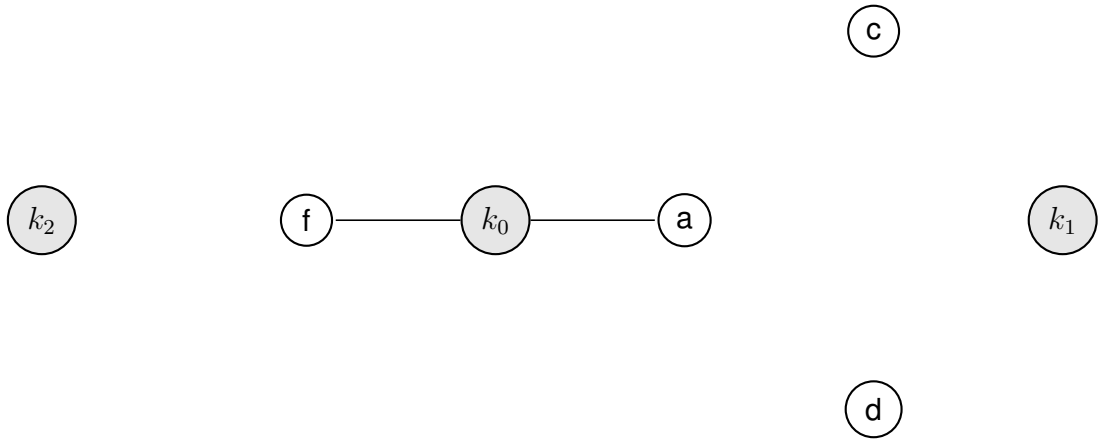


Figure 10 – Hypothetical partial solutions graph to exemplify the usage of Max-flow min-cuts for generating connectivity constraints

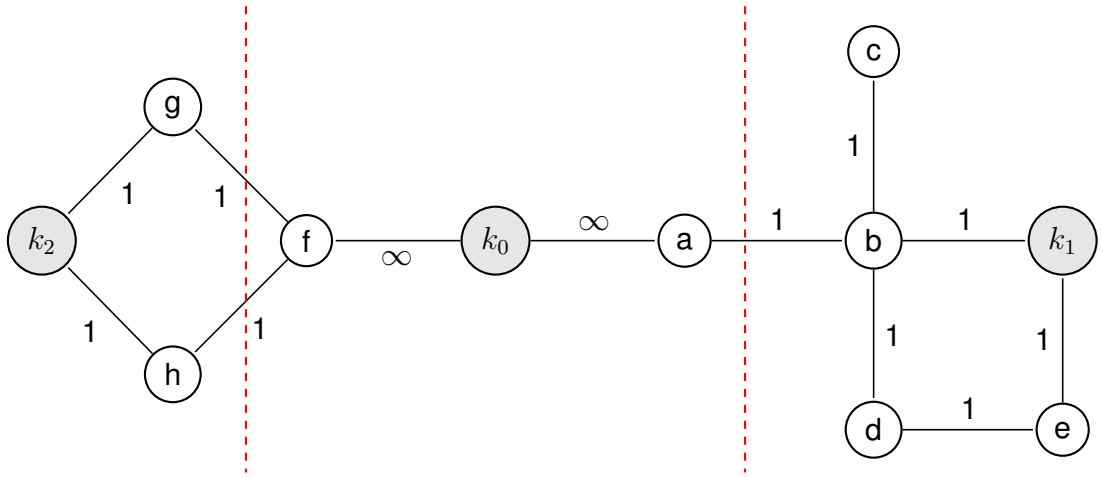


Figure 11 – Hypothetical cuts to exemplify the usage of Max-flow min-cuts for generating connectivity constraints

4 Testing the formulations and analyzing the problem behavior

This chapter is devoted to evaluating and discussing the results obtained from the models proposed in the preceding chapter. It details a series of experiments designed to examine the models' performance and facilitate a comparative analysis of the formulations. The problem in question is multifaceted and possibly has a variety of factors influencing its behavior and computational hardness; notably, the size of the graph and the number of mandatory vertices emerge as primary suspects. The interaction between these factors adds an additional layer of complexity to the problem, and it appears that the proportion of mandatory vertices relative to the total number of vertices is an element of significance. In light of these considerations, the experiments conducted are methodically designed to yield insights into how each factor contributes to the overall complexity and behavior of the problem.

For solving all three formulations, we used CPLEX 12.8.0, with default configurations and 16 threads. The code was implemented in C++, and all experiments were executed on the same machine - an Intel Xeon W-2175 with 512 GB of RAM.

An answer is positive (negative) if a feasible solution is (not) found. All positive instances were validated; however, negative instances were not verified on our test benches as there is no known mechanism to perform this action in a reasonable time span. Negative instances were validated in a few handcrafted small instances to ensure that the algorithm was running properly.

The algorithms were applied to publicly available benchmark instances from other well-known graph problems, such as the graph coloring problem (TRICK, 2008) and the Steiner Tree Problem (KOCH; MARTIN; VOSS, 2001), with necessary modifications to fit our nominal problem. However, the algorithms solved the majority of instances in under one second, which, while demonstrating efficiency, rendered these results less insightful for our purposes. This can likely be attributed to specific structural features in the benchmark instances that are pertinent to their original problems but either do not impact or even simplify the k-IN-A-TREE problem. As these results did not provide meaningful contributions to the current discussion, they will not be further explored in this thesis.

Given that the benchmark instances from other graph problems did not yield meaningful insights, we opted to explore a different approach: generating random graphs, which are described in detail in the following section.

4.0.1 Generating random graph instances

Preliminary empirical results using random instances revealed that these instances posed a more significant challenge for the algorithm than the benchmark problems; the random instances demonstrated higher effort and execution time from our algorithms. Based on these initial findings, we devised a method for automating the generation of random graph instances to evaluate the algorithms under more challenging conditions. This allowed for precise control over the graph size n, the cardinality of set K, and the approximate edge density. Additionally, we could dictate whether the graph incorporated a predetermined solution or if the solution's presence is purely random.

The generation methodology was inspired by the Erdős–Rényi model (ERDŐS; RÉNYI et al., 1960), as we developed a variation of it tailored to meet the needs of our study. The following steps were undertaken to ensure the generation of graph structures that conformed to our objectives:

Graph Initialization: We predefined the number of vertices. Each vertex was initialized with no edges to set the foundational structure of the graph.

Mandatory Vertices Assignment: The first two vertices in every instance were designated as mandatory members of K. Subsequently, additional vertices were randomly assigned to K based on a predetermined probability, reflecting the intended proportion of mandatory vertices.

Edge Creation: The edges between consecutive vertices were established based on a predetermined probability, which guided the overall density of the graph. While this probability influenced the expected average density, the actual edge density of specific instances could deviate due to randomness, aligning with the design intent that density should average to the desired level across multiple instances.

Partitioning and Connectivity Checks: The vertices in K were partitioned such that vertices with even indices were disconnected from those with odd indices, thereby ensuring a partition within $|K|$. The mandatory classification of the first two vertices guaranteed at least one vertex in each partition. A shadow-graph containing only the vertices from K was utilized to monitor connections and perform cycle checks whenever a new edge within K was proposed. Blocks due to this cycle prevention were recorded as 'Block counts' for the involved vertices.

Ensuring Connectivity: A Breadth-First Search (BFS) was employed to identify disconnected components. If the graph was not fully connected, an iterative process was implemented to link these components by connecting a vertex from that unconnected partition to a member of K — preferably one with a high block count — ensuring that all components were also connected to K. This step was carefully managed to prevent cycle formation and was repeated until full connectivity was achieved. Measures were in place to

prevent potential infinite loops where no suitable connections could be established.

Final Validation: A Depth-First Search (DFS) was conducted on the shadow-graph to ensure it remained disconnected as required. This step was crucial to verify that even and odd members of K were not interconnected, maintaining the experiment's integrity.

Special Considerations for Predefined Solutions: In experiments intended to test yes-instances, as discussed in subsequent sections, the initial graph configuration included predefined edges to represent a solution. Vertices involved in these predefined connections were excluded from random connectivity procedures to preserve the solution's integrity. Details of how each solution was constructed are elaborated in their respective sections.

Handling Invalid Instances: In certain cases, particularly when generating yes-instances and predefined solutions, the random graph generation process may produce invalid instances that do not meet the problem's requirements. For example, when aiming to generate purely random yes-instances, the approach was to indiscriminately generate random instances and filter out the no-instances. When generating some predefined solution configurations, the algorithm could also encounter deadlock situations where it couldn't add new elements without violating some constraint, which would cause the process to abort automatically.

This methodology ensured the generation of random graphs that met our experimental requirements and presented randomized and challenging scenarios for testing. It is pertinent to note that the connection density within K and any hidden solutions were slightly lower than the overall intended density due to the imposed block constraints, a variance considered negligible for the purposes of data collection.

4.1 Exploring the influence of graph size and density

To test our algorithms, we varied the parameters in our random graph generator, creating a series of graph instances with differing numbers of vertices, edge probabilities, and mandatory vertices; for this first batch of experiments, we did not intentionally introduce any known solution. This approach ensured diverse graph instances with different properties, allowing us to examine their impact on algorithm performance and problem hardness. Importantly, our generator was designed to avoid trivial instances, such as those featuring mandatory elements in disconnected components or those already forming a tree or clique structure.

We evaluated the algorithms using graph sizes $n = 50$, $n = 500$, $n = 5{,}000$, and $n = 25{,}000$. The selection of the first three sizes followed a simple tenfold progression, while the largest size was chosen based on the maximum manageable within our computational constraints and the time available for processing.

The sizes tested for $|K|$ were $4, 12, 36, \max(0.01n, 3), \log n,$ and $\sqrt{n}$. The first three values represent a basic incremental series, beginning at 4, which marks the smallest unsolved case for our problem domain. The latter three sizes were selected to maintain a proportionate relationship between the size of the instance and the subset K, based on various functional relationships.

For edge densities, we used configurations of $6n$, $n \log n$, $n\sqrt{n}$, and $\binom{N}{2}/2$. The $6n$ edges configuration, corresponding to an average degree of three, represents the sparsest yet nontrivial structure, as the problem transitions from trivial to NP-complete between 2-regular and 3-regular graphs (DERHY; PICOULEAU, 2009). The $n \log n$ edges configuration aligns with the connectivity threshold, providing a graph structure that is rich yet sparse, as extensively discussed in extremal graph theory (BOLLOBÁS, 1998). The configuration of $n\sqrt{n}$ edges ensures a highly connected graph with a typically constant diameter, including any sparse subgraph that satisfies certain conditions (JANSON; RUCINSKI; LUCZAK, 2011). Lastly, a density of $\binom{N}{2}/2$ results in a highly interconnected network, where each pair of vertices shares a linear number of common neighbors, greatly reducing the number of potential induced trees.

For each parameter combination of $|K|$, edge probability, and graph size, thirty problem instances were generated. The algorithms were then applied to these instances within a set time limit of 10 minutes per instance. This time constraint was empirically determined to fit the scope of our experiments within the desired timeframe. Since all instances with $n < 1{,}000$ were solved trivially, we focus our discussion on the results for $n = 5{,}000$ and $n = 25{,}000$.

The results of these experiments will be detailed in subsequent subsections through a series of tables. The tables will feature columns denoted by *# yes inst* and *# no inst*, indicating the number of yes-instances and no-instances solved by the algorithms, respectively. The column *# timeouts* will report the count of instances that exceeded the time constraint. Furthermore, *yes / no inst. mean time* will present the average execution times for the solved instances, with σ representing the standard deviation of these times.

4.1.1 Results for **P1**

The results for algorithm **P1** are shown in Table 1, demonstrating that the algorithm in question had difficulties scaling from $n = 5{,}000$ to $n = 25{,}000$ within the allotted time constraint. The table shows a trend where, as the number of edges and the density of the graph increase, the algorithm encounters greater execution times and increased number of timeouts. For $n = 5{,}000$ **P1** performs relatively well, solving all instances up to $|K| = n \log n$, for the other two $|K|$ values, higher densities started causing more timeouts.

As the instance size increases to $n = 25{,}000$, the decline in the algorithm's perfor-

mance becomes more evident. There is a significant drop in the success rate across all edge densities, accompanied by a sharp increase in the mean computation time. This behavior suggests that **P1** encounters substantial difficulties when dealing with higher-density instances and larger problem sizes, as indicated by the frequent timeouts and the greater variance in solution times, reflected in the higher standard deviation values. However, it is crucial to exercise caution when interpreting these deviations. In some cases, low deviation values stem from a limited number of successful samples. For instance, in the $|K| = n \log n$ case with an edge density of $6n$, the standard deviation of 8 only accounts for two successful instances, as the others timed out. Furthermore, since the completion times are close to the timeout threshold of 600 seconds, all successful instances are expected to exhibit similar runtimes, leading to a low variance.

Overall **P1** presented the slowest execution time for the performed experiments of the three proposed algorithms. Still, it could reliably solve instances of size five thousand, always returning an answer in a matter of minutes. It was also capable of solving a few instances with twenty-five thousand vertices in less than ten minutes. However, the density and size of K directly influenced the capacity of the algorithm doing so.

Table 1 – Comparison of **P1** for $n = 5{,}000$ and $n = 25{,}000$

		P1 with $n = 5{,}000$					**P1 with** $n = 25{,}000$				
$\|K\|$	edge density	# yes inst.	# no inst.	# timeout	yes inst. mean time	no inst. mean time	# yes inst.	# no inst.	# timeout	yes inst. mean time	no inst. mean time
4	$6n$	27	3	0	32 ($\sigma = 17$)	40 ($\sigma = 19$)	17	1	12	487 ($\sigma = 141$)	580 ($\sigma = 0$)
	$n \log n$	29	1	0	31 ($\sigma = 14$)	42 ($\sigma = 12$)	20	1	9	565 ($\sigma = 154$)	574 ($\sigma = 0$)
	$n\sqrt{n}$	25	5	0	32 ($\sigma = 19$)	44 ($\sigma = 17$)	18	2	10	507 ($\sigma = 148$)	532 ($\sigma = 21$)
	$\binom{N}{2}/2$	19	11	0	55 ($\sigma = 22$)	57 ($\sigma = 25$)	1	0	29	499 ($\sigma = 0$)	-
12	$6n$	22	8	0	87 ($\sigma = 35$)	91 ($\sigma = 45$)	20	2	8	517 ($\sigma = 142$)	584 ($\sigma = 22$)
	$n \log n$	22	8	0	95 ($\sigma = 32$)	102 ($\sigma = 68$)	18	4	8	511 ($\sigma = 131$)	562 ($\sigma = 31$)
	$n\sqrt{n}$	18	12	0	98 ($\sigma = 45$)	115 ($\sigma = 71$)	12	0	18	532 ($\sigma = 98$)	-
	$\binom{N}{2}/2$	14	16	0	112 ($\sigma = 52$)	147 ($\sigma = 95$)	6	0	24	515 ($\sigma = 45$)	-
36	$6n$	21	9	0	195 ($\sigma = 80$)	215 ($\sigma = 114$)	18	6	6	499 ($\sigma = 97$)	522 ($\sigma = 65$)
	$n \log n$	19	11	0	221 ($\sigma = 80$)	248 ($\sigma = 104$)	20	1	9	521 ($\sigma = 84$)	388 ($\sigma = 0$)
	$n\sqrt{n}$	20	10	0	245 ($\sigma = 97$)	254 ($\sigma = 122$)	11	0	19	530 ($\sigma = 75$)	-
	$\binom{N}{2}/2$	15	15	0	296 ($\sigma = 141$)	320 ($\sigma = 195$)	2	0	28	574 ($\sigma = 18$)	-
$\log n$	$6n$	20	10	0	68 ($\sigma = 32$)	76 ($\sigma = 41$)	15	2	13	522 ($\sigma = 84$)	561 ($\sigma = 8$)
	$n \log n$	25	5	0	71 ($\sigma = 28$)	79 ($\sigma = 46$)	18	0	12	501 ($\sigma = 84$)	-
	$n\sqrt{n}$	21	9	0	79 ($\sigma = 32$)	77 ($\sigma = 48$)	12	0	18	544 ($\sigma = 30$)	-
	$\binom{N}{2}/2$	8	22	0	99 ($\sigma = 40$)	109 ($\sigma = 55$)	0	0	30	-	-
max (1%,3)	$6n$	27	3	0	268 ($\sigma = 114$)	295 ($\sigma = 117$)	0	0	30	-	-
	$n \log n$	21	9	0	299 ($\sigma = 120$)	340 ($\sigma = 156$)	0	0	30	-	-
	$n\sqrt{n}$	24	6	0	314 ($\sigma = 108$)	328 ($\sigma = 142$)	0	0	30	-	-
	$\binom{N}{2}/2$	16	10	4	445 ($\sigma = 190$)	498 ($\sigma = 156$)	0	0	30	-	-
$\sqrt{n}$	$6n$	18	10	2	325 ($\sigma = 147$)	401 ($\sigma = 108$)	0	0	30	-	-
	$n \log n$	20	6	4	368 ($\sigma = 146$)	480 ($\sigma = 98$)	0	0	30	-	-
	$n\sqrt{n}$	16	10	4	398 ($\sigma = 167$)	499 ($\sigma = 101$)	0	0	30	-	-
	$\binom{N}{2}/2$	4	16	10	420 ($\sigma = 102$)	540 ($\sigma = 80$)	0	0	30	-	-

4.1.2 Results for **P2** and **P3**

Table 2 provides a detailed comparison of the performance of algorithms P3 and P2 for different edge densities and instance sizes when $n = 5{,}000$. A key observation is that **P3** consistently outperforms **P2** in terms of mean computation time across all edge densities. For instance, in the $|K| = 4$ cases, the mean time for **P3** to solve the instances is considerably lower, with differences such as 10 seconds versus 36 seconds for the $6n$ edge density. This trend persists as edge densities increase, indicating that **P3** handles smaller and sparser graph structures faster than **P2**.

As edge density increases, the performance gap between P3 and P2 becomes even more significant. In the $|K| = 36$ cases, the difference in mean computation time widens even more. For example, for $n\sqrt{n}$ **P3** solves instances in 81 seconds, while **P2** requires 187 seconds. This pattern holds across the table, suggesting that while **P3** experiences an increase in computation time as the graph density grows, it remains more efficient than **P2** across all densities. One final noteworthy insight is that **P2** not only requires more time to solve instances but also shows greater variance in solution times (i.e., higher standard deviation values). For example, in the $\binom{N}{2}/2$ case with $\sqrt{n}$, **P3** has a mean time of 180 seconds with a standard deviation of 52, whereas **P2** has a larger standard deviation of 101 seconds for a mean time of 306 seconds.

Table 2 – Comparison of **P3** and **P2** for $n = 5{,}000$

		P3					**P2**				
$\|K\|$	edge density	# yes inst.	# no inst.	# timeout	yes inst. mean time	no inst. mean time	# yes inst.	# no inst.	# timeout	yes inst. mean time	no inst. mean time
4	$6n$	27	3	0	10 ($\sigma = 3$)	11 ($\sigma = 1$)	27	3	0	36 ($\sigma = 12$)	42 ($\sigma = 22$)
	$n \log n$	29	1	0	9 ($\sigma = 1$)	12 ($\sigma = 3$)	29	1	0	42 ($\sigma = 14$)	49 ($\sigma = 20$)
	$n\sqrt{n}$	25	5	0	11 ($\sigma = 3$)	11 ($\sigma = 3$)	25	5	0	42 ($\sigma = 18$)	51 ($\sigma = 26$)
	$\binom{N}{2}/2$	19	11	0	11 ($\sigma = 2$)	11 ($\sigma = 5$)	19	11	0	60 ($\sigma = 11$)	92 ($\sigma = 36$)
12	$6n$	22	8	0	29 ($\sigma = 9$)	27 ($\sigma = 12$)	22	8	0	82 ($\sigma = 30$)	101 ($\sigma = 56$)
	$n \log n$	22	8	0	31 ($\sigma = 9$)	31 ($\sigma = 12$)	22	8	0	81 ($\sigma = 16$)	104 ($\sigma = 55$)
	$n\sqrt{n}$	18	12	0	32 ($\sigma = 11$)	33 ($\sigma = 10$)	18	12	0	76 ($\sigma = 18$)	108 ($\sigma = 79$)
	$\binom{N}{2}/2$	14	16	0	32 ($\sigma = 12$)	39 ($\sigma = 17$)	14	16	0	91 ($\sigma = 41$)	136 ($\sigma = 84$)
36	$6n$	21	9	0	70 ($\sigma = 16$)	85 ($\sigma = 27$)	21	9	0	146 ($\sigma = 58$)	184 ($\sigma = 84$)
	$n \log n$	19	11	0	70 ($\sigma = 21$)	94 ($\sigma = 32$)	19	11	0	175 ($\sigma = 61$)	233 ($\sigma = 81$)
	$n\sqrt{n}$	20	10	0	81 ($\sigma = 25$)	95 ($\sigma = 36$)	20	10	0	187 ($\sigma = 68$)	245 ($\sigma = 95$)
	$\binom{N}{2}/2$	15	15	0	84 ($\sigma = 25$)	99 ($\sigma = 41$)	15	15	0	214 ($\sigma = 94$)	298 ($\sigma = 98$)
$\log n$	$6n$	20	10	0	15 ($\sigma = 5$)	15 ($\sigma = 10$)	20	10	0	47 ($\sigma = 18$)	48 ($\sigma = 21$)
	$n \log n$	25	5	0	13 ($\sigma = 7$)	15 ($\sigma = 6$)	25	5	0	47 ($\sigma = 11$)	49 ($\sigma = 24$)
	$n\sqrt{n}$	21	9	0	15 ($\sigma = 6$)	16 ($\sigma = 8$)	21	9	0	49 ($\sigma = 29$)	44 ($\sigma = 19$)
	$\binom{N}{2}/2$	8	22	0	13 ($\sigma = 7$)	19 ($\sigma = 11$)	8	22	0	56 ($\sigma = 29$)	79 ($\sigma = 32$)
max (1%,3)	$6n$	27	3	0	72 ($\sigma = 21$)	80 ($\sigma = 23$)	27	3	0	195 ($\sigma = 80$)	204 ($\sigma = 92$)
	$n \log n$	21	9	0	80 ($\sigma = 26$)	89 ($\sigma = 22$)	21	9	0	199 ($\sigma = 88$)	263 ($\sigma = 99$)
	$n\sqrt{n}$	24	6	0	99 ($\sigma = 22$)	103 ($\sigma = 28$)	24	6	0	187 ($\sigma = 91$)	285 ($\sigma = 108$)
	$\binom{N}{2}/2$	17	13	0	107 ($\sigma = 23$)	133 ($\sigma = 27$)	17	13	0	242 ($\sigma = 93$)	284 ($\sigma = 104$)
$\sqrt{n}$	$6n$	19	11	0	147 ($\sigma = 47$)	188 ($\sigma = 42$)	19	11	0	245 ($\sigma = 68$)	287 ($\sigma = 93$)
	$n \log n$	20	10	0	163 ($\sigma = 49$)	199 ($\sigma = 45$)	20	10	0	268 ($\sigma = 87$)	312 ($\sigma = 102$)
	$n\sqrt{n}$	16	14	0	162 ($\sigma = 52$)	212 ($\sigma = 56$)	16	14	0	253 ($\sigma = 90$)	305 ($\sigma = 94$)
	$\binom{N}{2}/2$	4	26	0	180 ($\sigma = 52$)	211 ($\sigma = 51$)	4	26	0	306 ($\sigma = 101$)	364 ($\sigma = 121$)

Table 3 compares the performance of algorithms **P3** and **P2** for instances where $n =$

$25{,}000$. A key observation is that both algorithms face significant computational challenges at this larger instance size, but **P3** continues to outperform **P2** in terms of mean computation time across all edge densities. For example, in the case of $|K| = 4$ and an edge density of $6n$ on yes-instances, **P3** achieves a mean computation time of 197 seconds, while **P2** takes much longer at 501 seconds. This consistent trend of **P3** being faster is evident across all densities.

As edge density increases, the performance gap between P3 and P2 widens, particularly in denser configurations such as $|K| = 36$. For instance, in the $n \log n$ case, **P3** completes no-instances in 208 seconds on average, compared to **P2**, which requires 520 seconds. Moreover, the gap is not only in computation time but also in solution consistency. **P2** shows higher standard deviations across many cases, indicating greater variability in performance, particularly for denser edge sets.

Furthermore, it is essential to note that **P2** also experiences more frequent timeouts in this larger instance size, especially in the denser edge sets such as $\sqrt{n}$ with $\binom{N}{2}/2$, **P2** times out in 26 instances. In contrast, **P3** completes all of them successfully.

Table 3 – Comparison of **P3** and **P2** for $n = 25{,}000$

		P3					**P2**				
$\|K\|$	edge density	# yes inst.	# no inst.	# timeout	yes inst. mean time	no inst. mean time	# yes inst.	# no inst.	# timeout	yes inst. mean time	no inst. mean time
4	$6n$	29	1	0	197 ($\sigma = 50$)	189 ($\sigma = 0$)	29	1	0	501 ($\sigma = 98$)	272 ($\sigma = 0$)
	$n \log n$	25	5	0	193 ($\sigma = 51$)	203 ($\sigma = 56$)	25	5	0	520 ($\sigma = 54$)	510 ($\sigma = 48$)
	$n\sqrt{n}$	23	7	0	201 ($\sigma = 52$)	202 ($\sigma = 53$)	23	7	0	474 ($\sigma = 105$)	512 ($\sigma = 89$)
	$\binom{N}{2}/2$	17	13	0	215 ($\sigma = 54$)	210 ($\sigma = 61$)	17	13	0	499 ($\sigma = 79$)	514 ($\sigma = 86$)
12	$6n$	22	8	0	191 ($\sigma = 50$)	193 ($\sigma = 23$)	22	8	0	524 ($\sigma = 80$)	532 ($\sigma = 99$)
	$n \log n$	21	9	0	190 ($\sigma = 51$)	187 ($\sigma = 33$)	21	9	0	501 ($\sigma = 101$)	541 ($\sigma = 87$)
	$n\sqrt{n}$	21	9	0	204 ($\sigma = 52$)	200 ($\sigma = 47$)	21	9	0	501 ($\sigma = 88$)	533 ($\sigma = 84$)
	$\binom{N}{2}/2$	14	16	0	202 ($\sigma = 41$)	210 ($\sigma = 51$)	14	16	0	499 ($\sigma = 87$)	548 ($\sigma = 96$)
36	$6n$	20	10	0	208 ($\sigma = 55$)	212 ($\sigma = 67$)	20	6	4	544 ($\sigma = 36$)	532 ($\sigma = 54$)
	$n \log n$	20	10	0	208 ($\sigma = 52$)	222 ($\sigma = 52$)	20	8	2	520 ($\sigma = 74$)	536 ($\sigma = 62$)
	$n\sqrt{n}$	22	8	0	211 ($\sigma = 63$)	222 ($\sigma = 63$)	20	6	4	486 ($\sigma = 142$)	528 ($\sigma = 62$)
	$\binom{N}{2}/2$	11	19	0	219 ($\sigma = 57$)	230 ($\sigma = 64$)	11	10	9	516 ($\sigma = 85$)	521 ($\sigma = 65$)
$\log n$	$6n$	20	10	0	210 ($\sigma = 57$)	210 ($\sigma = 62$)	20	10	0	512 ($\sigma = 77$)	522 ($\sigma = 98$)
	$n \log n$	18	12	0	200 ($\sigma = 53$)	214 ($\sigma = 59$)	18	12	0	508 ($\sigma = 55$)	511 ($\sigma = 33$)
	$n\sqrt{n}$	15	15	0	212 ($\sigma = 29$)	208 ($\sigma = 51$)	15	15	0	554 ($\sigma = 49$)	548 ($\sigma = 65$)
	$\binom{N}{2}/2$	6	24	0	208 ($\sigma = 55$)	224 ($\sigma = 54$)	6	24	0	551 ($\sigma = 91$)	522 ($\sigma = 65$)
$\sqrt{n}$	$6n$	17	13	0	300 ($\sigma = 67$)	287 ($\sigma = 62$)	17	0	13	578 ($\sigma = 44$)	-
	$n \log n$	18	12	0	294 ($\sigma = 69$)	302 ($\sigma = 57$)	8	6	16	514 ($\sigma = 68$)	532 ($\sigma = 48$)
	$n\sqrt{n}$	16	14	0	308 ($\sigma = 66$)	304 ($\sigma = 79$)	9	1	20	557 ($\sigma = 39$)	447 ($\sigma = 0$)
	$\binom{N}{2}/2$	7	23	0	342 ($\sigma = 49$)	341 ($\sigma = 73$)	0	4	26	-	566 ($\sigma = 68$)
max (1%,3)	$6n$	17	13	0	336 ($\sigma = 62$)	379 ($\sigma = 74$)	7	2	21	541 ($\sigma = 87$)	562 ($\sigma = 3$)
	$n \log n$	2	28	0	402 ($\sigma = 79$)	411 ($\sigma = 83$)	2	3	25	523 ($\sigma = 45$)	540 ($\sigma = 26$)
	$n\sqrt{n}$	9	19	2	414 ($\sigma = 93$)	419 ($\sigma = 96$)	9	0	21	506 ($\sigma = 68$)	-
	$\binom{N}{2}/2$	8	8	14	480 ($\sigma = 98$)	496 ($\sigma = 85$)	0	0	30	-	-

Both **P2** and **P3** formulations performed better than **P1**. With similar times for instances of size five thousand, the algorithms performed faster and with less deviation. However, the difference between them can be seen more clearly in instances where $k = 25{,}000$. On this data, we can infer that the **P3** is less susceptible to increases in the size of K and

performs in a more stable and faster way for those circumstances. The overall time seems to be less than half of the time used by **P2** while wielding the same results and even solving some instances that **P2** could not solve in the time limit. Based on this, it can be inferred that the **P3** approach is faster for random graphs with the presented densities.

Despite the data presented, we cannot statistically infer that one algorithm is strictly better than the other across all possible instances. The results provided reflect the performance for the specific datasets tested, and while **P3** consistently shows faster solution times and fewer timeouts in this context, these observations cannot be generalized without further rigorous analysis. The primary objective of this comparison is to evaluate the algorithms' capabilities in solving the given instances, not to declare a definitive winner. By having multiple algorithms available, we can better understand whether certain behaviors, such as longer computation times or increased variance in solution times, are intrinsic to the structure of the problem itself or are related to the individual strengths and limitations of the algorithms.

4.2 Exploring the influence of |K|

From the results presented in the last section, it became clear that the size of K influences the execution time significantly. To study this influence on a deeper level, the dimension of n was fixed at $25{,}000$, and more experiments were performed. Random instances with K sizes varying from 10 to $7{,}000$ in steps of 10 (*i.e.*: 10, 20, 30, ..., $7{,}000$) were generated and presented to both **P2** and **P3**. K max size was set at $7{,}000$ as creating non-trivial instances (instances where K does not induce a cycle by itself) became increasingly difficult. This experiment was performed for the four possible edge densities in the previous section. On another note, it is important to bring the reader's attention to the scale of the charts when going through the data; they have different scales on the Y-axis otherwise, this would cause some of them to be hard to read given the absolute difference in values between them.

To better understand the k-IN-A-TREE problem and complexity, we want to evaluate four main points:

1. How does the size of n influence the execution time?
2. How does the size of K influences the execution time?
3. Are positive instances harder to solve than negative instances (or the opposite)?
4. If the instance is positive, do any particular properties of the solution influence the execution time?

The first point was explored in the previous section, and the second and third points are assessed in this section; the last point is discussed in the next section. We generated k-IN-A-TREE problem instances crafted to provide insights on those questions. To solve

those instances, the previously mentioned algorithms were also implemented in C++, using CPLEX 12.8.0. The implementations were then used to solve all the instances designed to explore different properties of the problem and provide further information on the variables that may influence the hardness of the problem. Once again, the same densities explored in the last section were reused in this set of experiments.

The instances were divided into two datasets detailed in Table 4, collectively named random datasets. The objective of those datasets is to provide insights into how the percentage of mandatory elements in one instance affects the execution time. For that purpose, the number of vertices was fixed with $n = 25{,}000$, and the number of mandatory elements varied from 10 to $7{,}000$ in steps as indicated on the *# instances* column. The two datasets represented in the table are:

d-random: This dataset comprises randomly generated instances; they have no special properties, nor are the instances guaranteed to be positive or negative. Due to the nature of the problem, as the size of K increases, if n is kept constant, the generated instances are more likely to be negative. This is a property of the problem, and we did not influence this in any way. However, as explained before, this dataset contains no trivially negative instances. The objective of this dataset is to provide a baseline execution time chart that is comparable to other datasets. By having this data, we can assess if adding or removing a property from the generated instances made them easier or harder to solve from an execution time perspective when compared to equivalent purely random instances.

d-random-yes: This dataset also comprises randomly generated instances; however, all instances are positive. In other words, it is known that there is at least one solution for each of these instances. There are no guarantees that the known solution is the only solution, nor are the algorithms expected to find the known solutions. This dataset was created to verify how random positive instances compare to random solutions. Since random solutions are mostly negative for large enough $|K|$, this comparison can be, to some extent, extrapolated to a positive vs. negative comparison. To generate these instances, we took a brute force approach, generating several instances, solving them, and picking only positive ones.

This provided us with tens of thousands of instances to solve. Both algorithms were used to solve each dataset. In all executions, CPLEX ran at the default configurations but was limited to 16 threads. All instances labeled as positive by the algorithms were validated. However, negative instances were not verified on our test benches as there is no known mechanism to perform this action within a reasonable time. However, we validated several handcrafted small negative instances to ensure the algorithm ran correctly. Due to the number of data points generated on the results, we will present a simple average of the numbers for each data point in the upcoming charts. The curious reader wondering how long it took to execute all those tests might be surprised to know that the computational time used exceeded four hundred days.

Table 4 – Random datasets

Name	edge densities	$\|K\|$ size	# instances	Notes
d-random	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 10 End 7,000 Step: 10 Total: 700	5 per combination 14,000 total	Randomly generated instances
d-random-yes	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 10 End 7,000 Step: 10 Total: 700	5 per combination 14,000 total	Randomly generated instances All instances are yes-instances

4.2.1 Influence of |K| in **P2**

Solving the previously mentioned **d-random** dataset with **P2** indicated that $|K|$ influenced the execution time as much as the input size. In Figure 12, the average execution time for each tested $|K|$ is presented on 4 bar charts, each representing a different graph density.

All densities share some common characteristics, notably a sharp increase in execution time on the lower values of $|K|$, followed by a small growth rate after a certain threshold. This threshold seems to happen roughly around $|K| = 900$, $|K| = 1{,}000$, $|K| = 2{,}000$ and $|K| = 2{,}800$ for each density in decreasing order. After the threshold, they behave differently from each other. The instances with $\binom{n}{2}/2$ density (blue) keep increasing their execution time, going from $3{,}500s$ to $5{,}100s$ between the threshold and the final value. The instances with density $n\sqrt{n}$ behave similarly, but the growth rate after reaching the threshold is lower, going from $3{,}200s$ to $4{,}000s$. The interesting comparisons begin with the instances of density $n \log n$. The execution time seems to plateau after the threshold and hovers around $3{,}000s$; however, there is a lot of variance between the data points to draw a firm conclusion with a visual analysis only. Finally, the instances with density $6n$ seem to have an inverse behavior; the execution time goes into a downward trend after the threshold, this is the only chart were the final execution time is inferior to most values around the threshold.

To better visualize and compare the somewhat noisy data from this experiment, every 10 data points were averaged and plotted in a single line chart. This was done to smooth the variations further and allow for better visualizations with less data pollution. The result can be seen in Figure 13. The changes and trends of the execution time are more apparent in this visualization, and further analysis can be made. Despite the faster growth of the denser instances, the execution times overlap in the $2.200 - 4.000$ range, only becoming distinct again after that value, with the red line ($n\sqrt{n}$) lingering longer with the lower density instances, not being clearly slower until $|K| = 5.500$. In this specific interval, there are points where the denser instance is the fastest; although this is not a pattern, it is interesting to

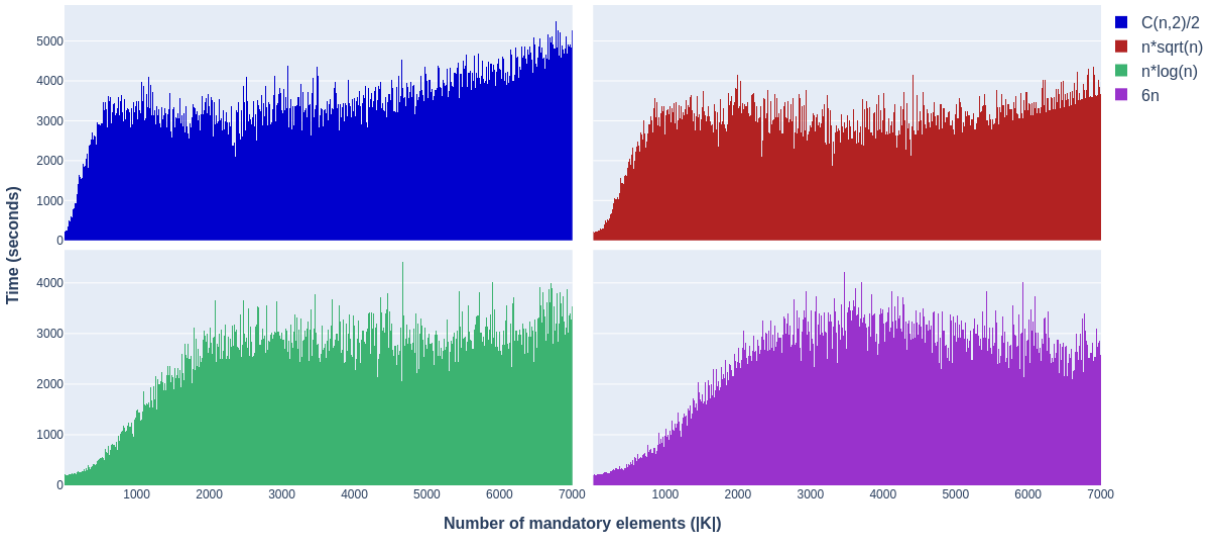


Figure 12 – Execution time versus of the size of K for **P2** using instances with $n = 25{,}000$, and decreasing edge density of $\binom{n}{2}/2$, $n\sqrt{n}$, $n \log n$ and $6n$ from left-to-right top-to-bottom

notice that there is no clear winner across the whole x-axis spectrum.

However, despite the mixing of the lines in the middle ranges, we see a tendency of the denser instance to stay on top most of the time, even by only a tiny margin. This is not true for the second denser instance, which is frequently positioned as the fastest. Despite the larger number of edges, the higher-density instance does not seem harder to solve in that range. Thus, it is safe to assume that either the computational cost of representing more edges is not more significant than the intrinsic cost of finding an answer or the representation cost is higher but is being compensated by answers that are, on average, easier to find in denser instances.

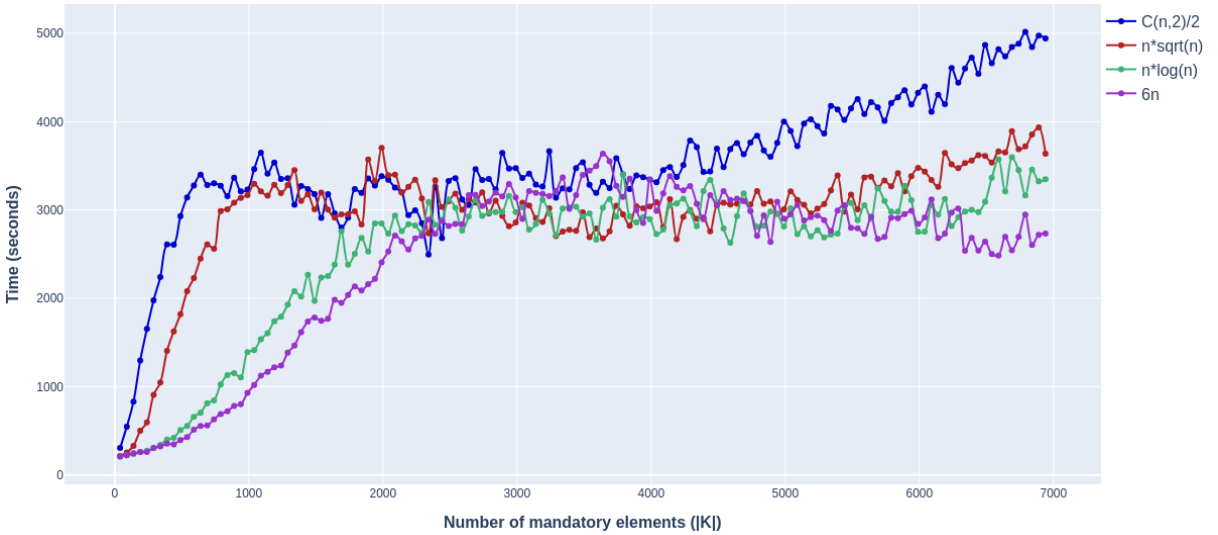


Figure 13 – Execution time versus of the size of the K for **P2** using instances with $n = 25{,}000$

From this data, it can be inferred that for the algorithm **P2**, given this fixed graph size

of $n = 25{,}000$, increasing the size of $|K|$ will make the instance harder or equally challenging on average. Beyond that, denser instances are either more complex or similarly tricky to solve.

4.2.2 Influence of |K| in **P3**

Presenting the same **d-random** dataset to **P3** further suggested the stronger than linear correlation in the lower values of $|K|$ and the existence of the threshold seen with **P2**. Similar to the previous experiment, after the threshold, the behavior changes. But this time, increasing the size of K seems to continually decrease the execution time with a rate that varies with each density. The data is represented in four bar charts in Figure 14.

Once more, all densities share common characteristics: the execution time initially rises as the size of $|K|$ increases to a certain point, in which the execution time seems to plateau for a while, followed by a decline that persists until the end of the experiment. Each of these three steps has a different rate for each density, but they seem somewhat correlated. The faster the initial ascension, the sooner the plateau is reached, the narrower the stationary section is, and the quicker the descent is. However, the decline always seems to be slower in rate than the ascension. The instances with $\binom{n}{2}/2$ density (blue) appeared to reach the threshold around $K = 800$ while the others reached it around $K = 1{,}100$, $K = 2{,}300$ and $K = 3{,}100$ in descending density order.

The two denser instances also seemed to get a fourth state, a second plateau at the end of the test. In this state, they achieved a visually stable configuration with less variance than other parts of the experiment and an execution time similar to the lower values of K - the lower values on the whole chart. Both $\binom{n}{2}/2$ (blue) and $n\sqrt{n}$ (red) seemed to reach it at around $|K| = 5{,}000$. Despite not reaching this state, both $n \log n$ (green) and $6n$ showed tendencies of going through the same process at a slower, more spread out rate, and it is reasonable to assume that extrapolating the data beyond $|K| = 7{,}000$ would eventually lead to the same situation.

As with the previous chart, another graphical visualization of the data was generated by averaging every ten points on the last dataset. The line chart representation of this new data is available in Figure 15. In this new chart, it is easier to see that despite the apparent differences between the data points, the maximum execution time did not vary by much; it happened on different values of $|K|$ for each density, with the denser instances happening sooner. Furthermore, the only moment on the chart that different densities get mixed is by the end, after $|K| = 5.000$, when the two denser instances get bundled and become indistinguishable and are later joined by $n \log n$ at the very end. Altering the edge density seems to make the most difficult instances happen with a smaller size of K and narrow the curve, diminishing its deviation concerning the maximum value. Thus turning the increasing and decreasing of the execution time steeper concerning the x-axis. It also affects

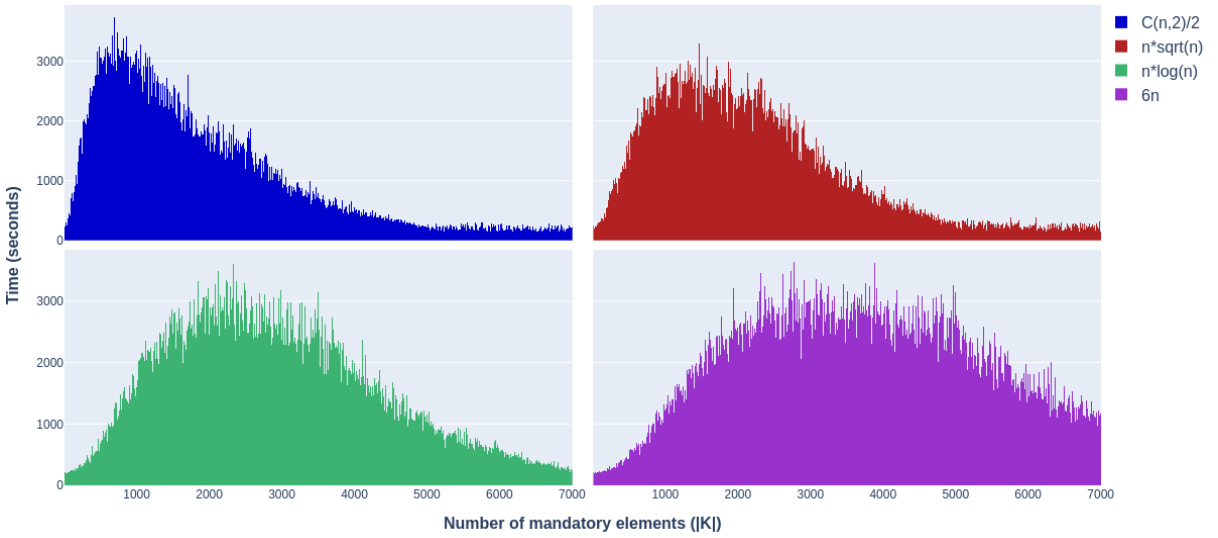


Figure 14 – Execution time versus of the size of the K for **P3** using instances with $n = 25{,}000$, and decreasing edge density of $\binom{n}{2}/2$, $n\sqrt{n}$, $n \log n$ and $6n$ from left-to-right top-to-bottom

the ramp-up with a stronger correlation than the ramp-down.

On these charts, there is a clear descending tendency and a baseline execution time that is reached on instances with a sufficiently large $|K|$. The baseline value is so similar between different cases with different densities and with so little deviation that it could indicate an association with the intrinsic costs of the code implementation of the algorithm and not the logic or the instances themselves.

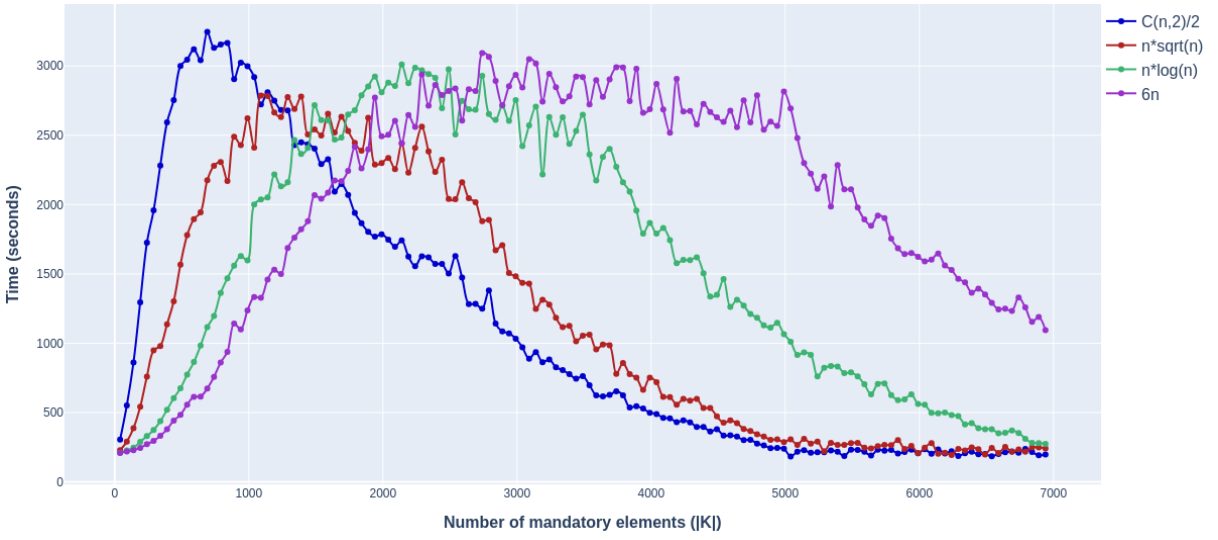


Figure 15 – Execution time versus of the size of the K for **P3** using instances with $n = 25{,}000$

It can also be inferred that for the algorithm **P3**, given this fixed graph size of $n = 25{,}000$, increasing the size of $|K|$ by itself will not make the instance harder. It may even make it easier. Despite being larger and requiring more memory to be stored, denser instances are not more complex to solve overall; that only happens on smaller values of K relative

to the total graph size (n) while sparser instances are more complicated at higher values of $|K|$ relative to the whole graph size. The total complexity seems to be a factor of both variables combined without one overpowering the other significantly. Those hypotheses will be discussed and explored in further experiments.

4.2.3 Comparing the influence of |K| between **P2** and **P3**

P2 and **P3** presented different behaviors when solving the same **d-random** dataset; it is clear that the influence of $|K|$ on the difficulty of the instance affects each one of those implementations uniquely. In this section, there will be a discussion of those differences and what new insights can be extracted about the problem itself and how it behaves.

Figure 16 shows a side-by-side comparison of both algorithms' results for each density individually in four line charts. The first obvious visualization is that algorithm **P3** dominated **P2** in all densities. At any moment in any chart, **P3** performed similarly if not better than **P2**. That being said, it is interesting to notice that, in all densities, they behave similarly at the beginning of the experiment, the lower values of $|K|$. The divergence seems to begin, on a visual inspection, near the threshold mentioned in the previous section, and that threshold varies from density to density. From that point onwards, **P3** has a significant decrease in the execution time, while **P2** keeps growing at a slower rate when compared to the initial growth rate.

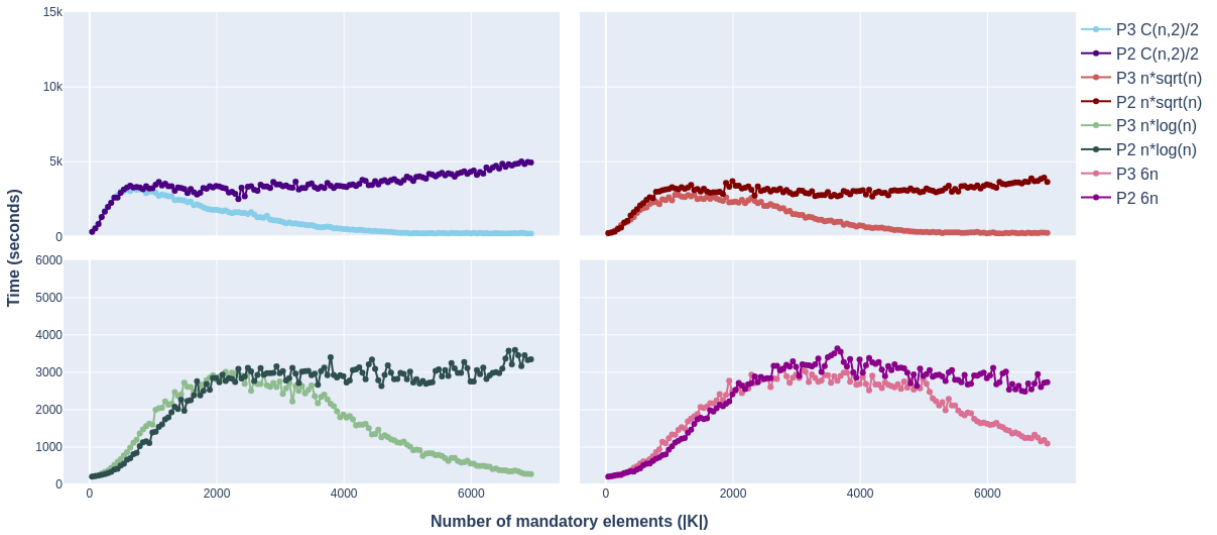


Figure 16 – Comparing execution time versus of the size of the K for **P2** and **P3** using instances with $n = 25{,}000$, and decreasing edge density of $\binom{n}{2}/2$, $n\sqrt{n}$, $n \log n$ and $6n$ from left-to-right top-to-bottom

Figure 17 shows a comparison of all execution for both algorithms **P2** and **P3** on the same chart. With this plot, it becomes even clearer the dominance of **P3**, after $|K| = 1{,}500$ the downward tendencies of the execution times ensure that **P3** lines are always below their equivalents for **P2** even overpowering more straightforward instances in no time. On the

range between $1{,}800 < |K| < 2{,}200$ there is some grouping of all execution times, and in the range $1{,}800 < |K| < 3{,}500$, there is some grouping of two densities for **P3** and all four densities **P2**, however even in this grouping situations **P3** presents a tendency of staying in the low end of the group, unlike the groupings explored on previous sections where some amount of variance allowed for punctual changes in the ordering.

Another noticeable tendency is how **P3** presents a clear downward path and a converging value for all densities if one were to extrapolate the data mentally. **P2**, on the other hand, presents an upward pattern with each density diverging from the other in the same mental extrapolation. Despite that, the execution times for the previously mentioned threshold values are very similar between both algorithms. In general, the first stage of each plot line between the algorithms is remarkably similar.

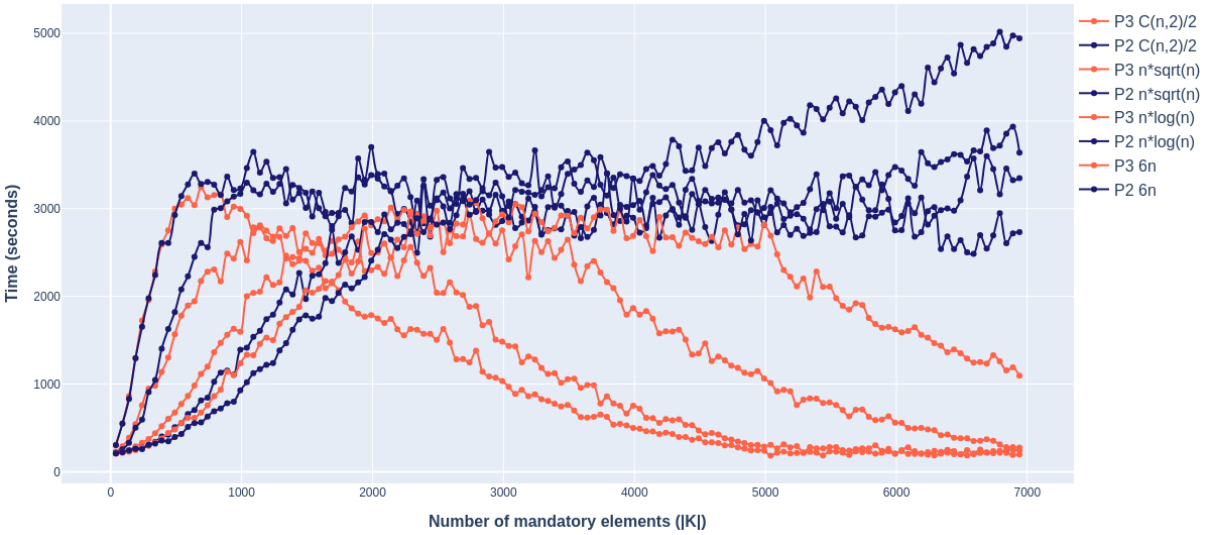


Figure 17 – Comparison of execution time versus of the size of the K for **P2** and **P3** using instances with $n = 25{,}000$

One hypothesis for this behavior is associated with the random nature of the generated instances. First, for low values of $|K|$, independently from the density, multiple solutions are more likely; thus, randomly picking a feasible solution becomes more likely without going through the entire search space. As the value of $|K|$ increases, the execution time also goes up as randomly solving the instance by "accident" becomes more unlikely, more vertices need to be explored, and more decisions are made. The denser the instance the faster this growth as more vertices are neighbors, and thus, more edges need to be explored. Finally, the breaking point that has the behavior change could be explained, again, by the random nature of the generated instances. Since they are randomly generated, the bigger the $|K|$, the more unlikely a random instance will be a yes-instance. As $|K|$ rises, it is expected that the number of no-instances will also rise, and this concentration of no-instances will also happen faster the denser the instance is. In this hypothesis, the threshold is related to the breaking point between instance difficulty and concentration of no-instances in the pool.

This hypothesis would suggest that the difference in behavior between **P2** and **P3** after the threshold is due to how each algorithm detects no-instances. **P3** would be more efficient in quickly detecting that one instance is a confirmed no-instance and dropping the execution. It is important to remember that finding a single unavoidable cycle that contains two or more elements of K leads to the confirmation of a no-instance; this is more likely to happen on very dense instances with a high value of $|K|$, explaining why the higher densities are, the faster to reach low execution time.

More experiments are discussed in the next subsections to explore this hypothesis further.

4.2.4 Yes-instance for **P2**

This subsection explores algorithm **P2** applied to the **d-random-yes** dataset.

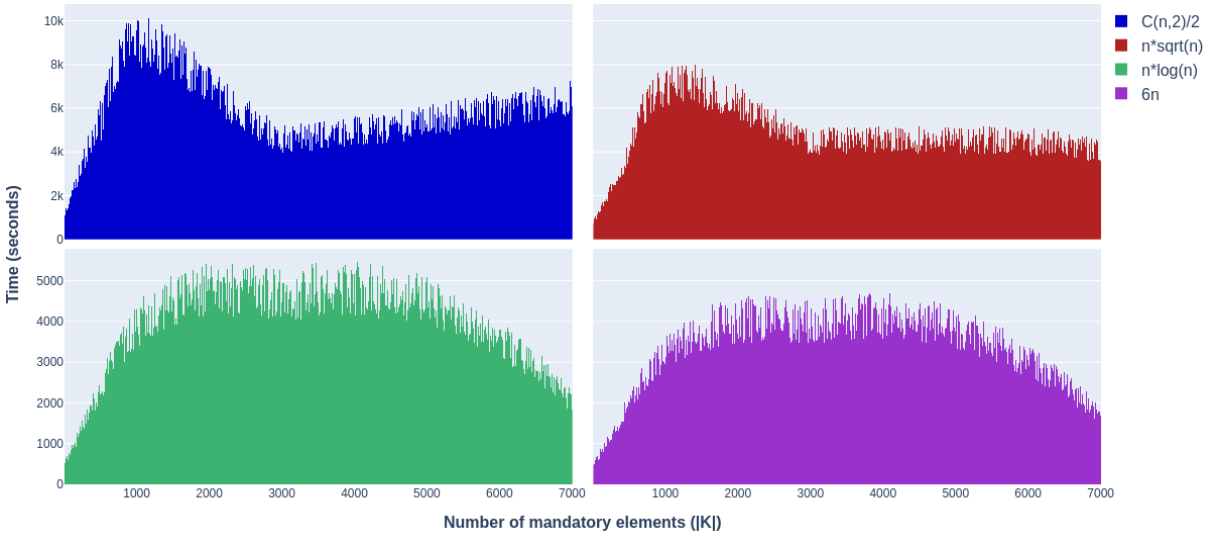


Figure 18 – Execution time versus of the size of the K for **P2** using yes-instances with $n = 25{,}000$

Figure 18 shows the data in a similar manner to Figure 12. The obvious noteworthy talking point here is how the two higher densities have a very similar behavior that seems to be exacerbated by the yes-only instances, with even the final ramp-up step appearing to be only shifted up. However, the two lower densities have very different shapes, from the initial ramp-up function to the ramp-down, to the point where, by the end, the execution times are inverted, with yes-instances being faster than the completely random.

This data highlights aspects of both the problem and the algorithm in question. Firstly, about the problem itself, based purely on this data, it is reasonable to assume that, indeed, determining that an instance is a no-instance is, on average, faster than deciding that it is a yes-instance. However, this experiment alone is not enough to conclude that this is the case, as it may also be a property of the algorithm itself. Changing the focus to the algorithm, it is

reasonable to conclude that it is capable of an early cutout when determining that there is no solution for the problem, as when this was not an option, we saw increases in the execution time.

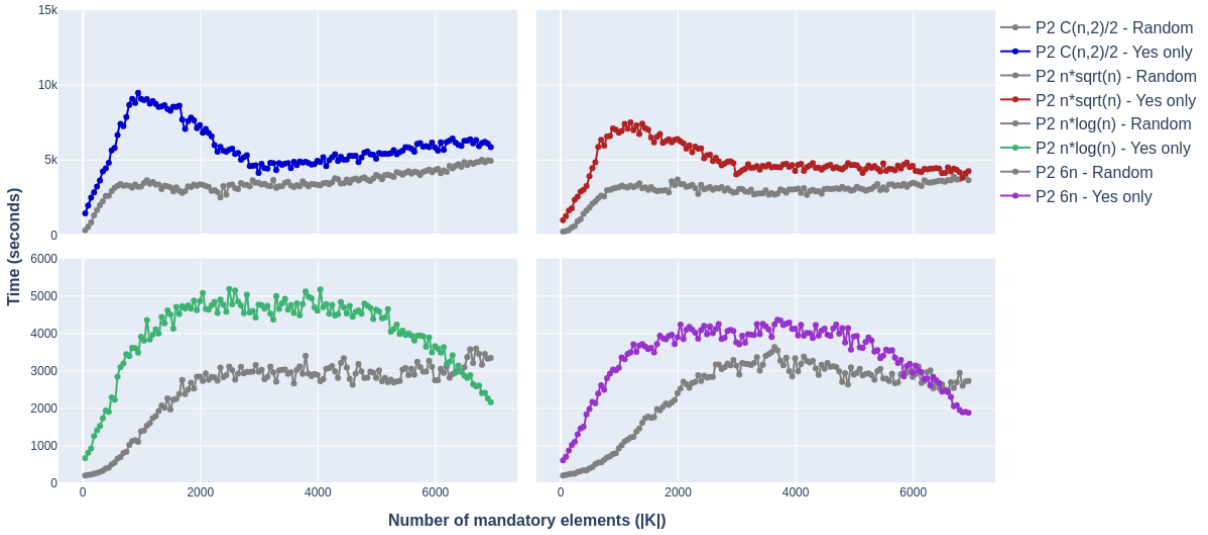


Figure 19 – Comparison of execution time versus of the size of the K using **P2** for both random and yes-only instances with $n = 25{,}000$

Figure 19 shows a comparison between **P2** applied to yes-only instances (colored lines) and **P2** applied to random instances (gray lines). This presentation shows a similar trend on the rising execution times along the x-axis, the increase ratio seems similar in both cases for all densities. The major difference is the initial accentuated growth ratio taking place in the early stages of the dataset (smaller $|K|$) on the yes-only instances. After this initial spike decays, both datasets seem to behave very similarly, separated by a constant difference in the execution time until the higher values of $|K|$. On the higher density instances, there is an indication that the execution times would invert, making the random instances slower, but there are not enough data points to confirm that this would take place however on the two lower density experiments (green and purple), the inversion is clear, around $|K| = 6{,}300$ the random instances become, on average, slower than the yes-only instances. This behavior comes from a downward trend in the execution times that is not present in the random instances.

The most reasonable hypothesis that could justify this shift is based on the same explanation on why the random instances are, on average, slower: early stop conditions. As hypothesized before, random instances should be easier as they have a higher likelihood of hitting a negative answer sooner, but since there are no trivial solutions on the generated graphs, this behavior could be flipped in certain conditions if it was easier to find one possible solution than to verify all the negative ones. This seems to be the case here, as the increase in the size of $|K|$ leads to a decrease in vertices that the algorithm can work with, a clear diminution of the search space, as we enforce no trivial solutions it seems we hit a threshold

where it is easier to find a solution as many are possible than it is to confirm a negative one. Another factor that could be influencing the results here is the aforementioned necessity not to uphold the strict density limit on the hidden solution itself, causing a drop in overall density that impacted the yes-only instances as the size of the solution became more significant.

4.2.5 Yes-instance for **P3**

Algorithm **P3** was submitted to the same **d-random-yes** dataset as **P2**. The results are available in Figure 20. Once more, the execution times are, on average, higher than the completely random instances. The data is also sectioned in intervals with roughly the same layout. An initial ramp-up with an aggressive growth rate, up until a threshold where the execution times start to ramp down. In both higher-density instances after the ramp down, there is a plateau with low execution times relative to the rest of the data. This data set has a higher variance across similar points on the x-axis, even more than the random instances, a counter-intuitive result. This chart seems to display two different patterns, one for the two higher densities and one for the two higher densities. However, it is not hard to visualize how a smooth continuous transition from the lowest density data would morph into the higher density; this would imply that the tail of the chart would reduce, and the threshold would move closer to the origin as the threshold point rose. This chart also has a larger difference between the slower execution times of each density.

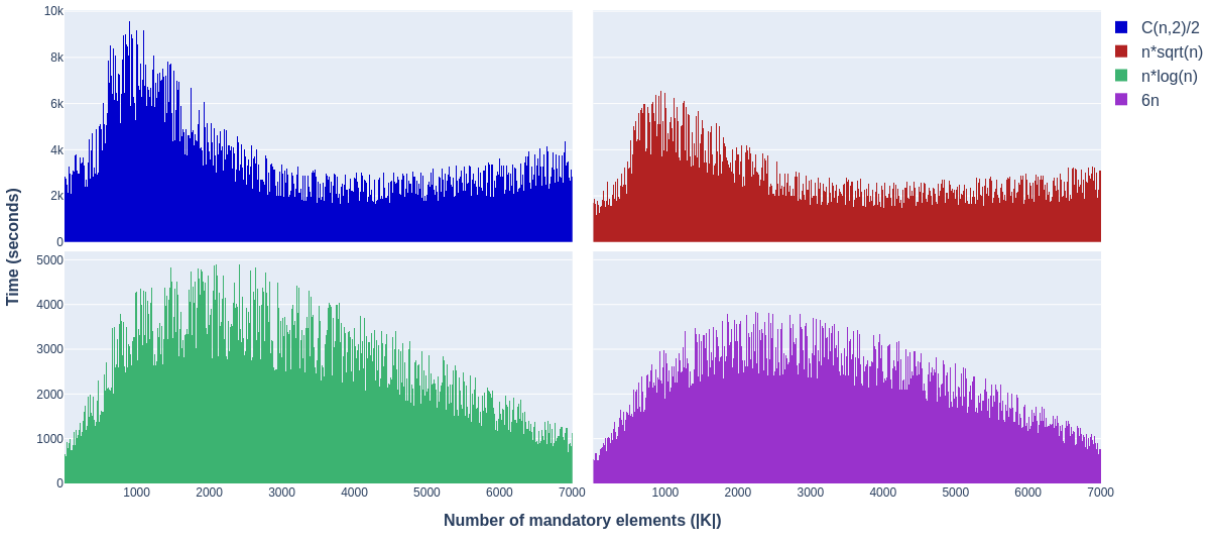


Figure 20 – Execution time versus of the size of the K for **P3** using yes-instances with $n = 25{,}000$

Figure 21 shows the comparison in execution times for **P3** between yes-only instances and random instances. From this perspective, the difference between the execution time becomes even more evident, with peaks up to double the execution time. The execution time of yes-instances overwhelms the execution time of random instances in all situations, except for the lower-density instances in which, in the second half of the experiment, the execution

times are roughly equivalent. Thus, for this algorithm, it is reasonable to assume, based on this data, that yes-instances, on average, are more complex to solve.

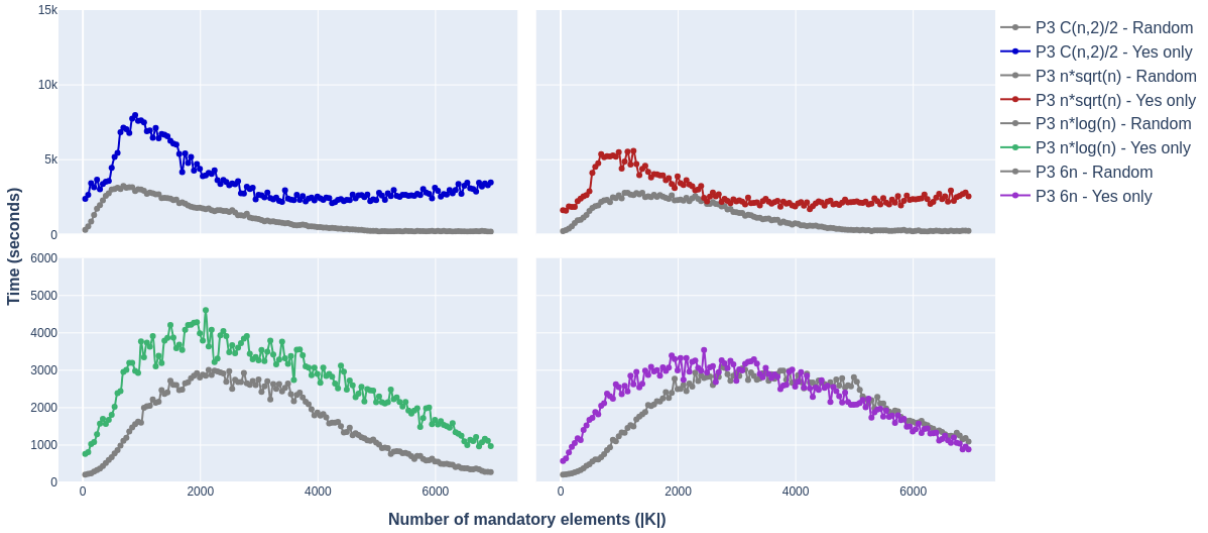


Figure 21 – Comparison of execution time versus of the size of the K using **P3** for both random and yes-only instances with $n = 25{,}000$

From this data, we can further explore the previously stated hypothesis: the low execution time plateau reached for the higher-density algorithms seems to be from easy-to-infer no-instances. Without those instances, not only those execution times are up to threefold higher, but one could argue that there is a slight upward trajectory. However, that is hard to state since that could be a consequence of the data variation and not a pattern. Since this behavior is from the end of the experiment, we don't have enough data to explore this further. Both the lower density charts, on the other hand, kept their downward tendencies in the second half of the experiment despite a shift upward in the execution time.

4.2.6 Comparing **P2** and **P3**

Figure 22 draws the comparison chart between the last two experiments by each density. The behavior of the algorithms is more similar than in the previous comparisons for the random instances. Algorithm **P3** still mostly dominates algorithm **P2** in terms of execution time, with even less victories the other way around than last time. There is a clear gap between them, especially after the threshold. But contrary to previous time the behavioral patterns are very similar, just shifted up. This suggests that, restricted to yes-instances, the impact of $|K|$ on the hardness of an instance is similar to both algorithms.

Plotting all the data in a single chart in Figure 23 reinforces the previous conclusions. There is less overlap than in the last comparison, and the clusters are evident between the algorithms; however, both seem to have the same pattern from the ramp-up rates to the threshold value to the final sections of the experiment with two downward trajectories and two upwards ones. The difference in execution time becomes obvious and we see that

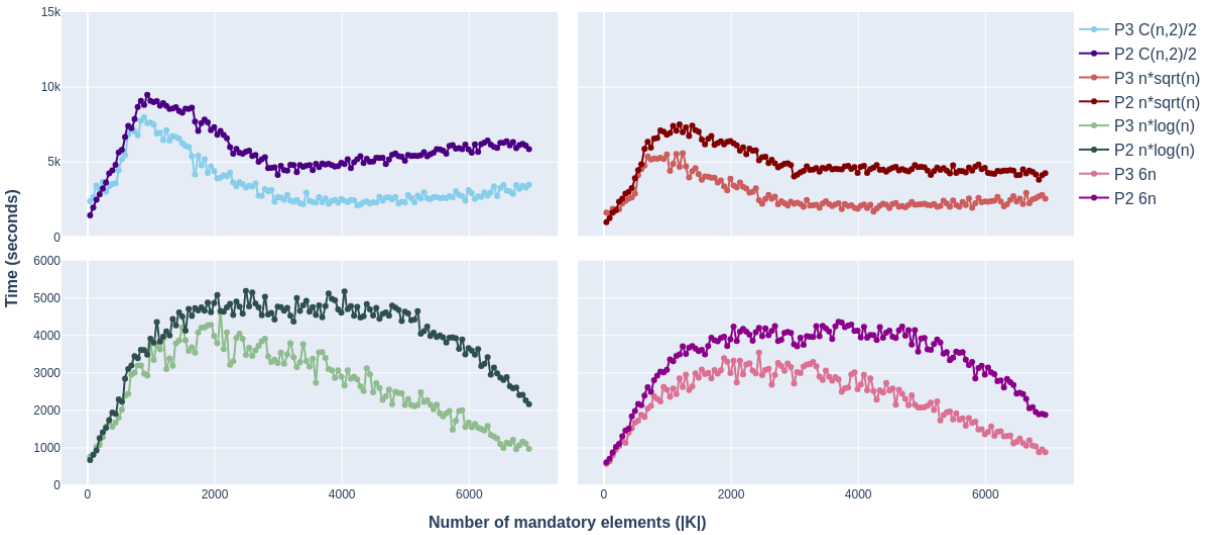


Figure 22 – Comparison of the execution time by density in the function of the size of K between **P2** and **P3** using yes-instances with $n = 25{,}000$

the position where the density where the difference in execution times between densities is minimal is shifted on the x-axis between the two data sets. For **P2** it happens when $|K| = 3{,}200$ and for **P3** it is around $|K| = 5{,}000$.

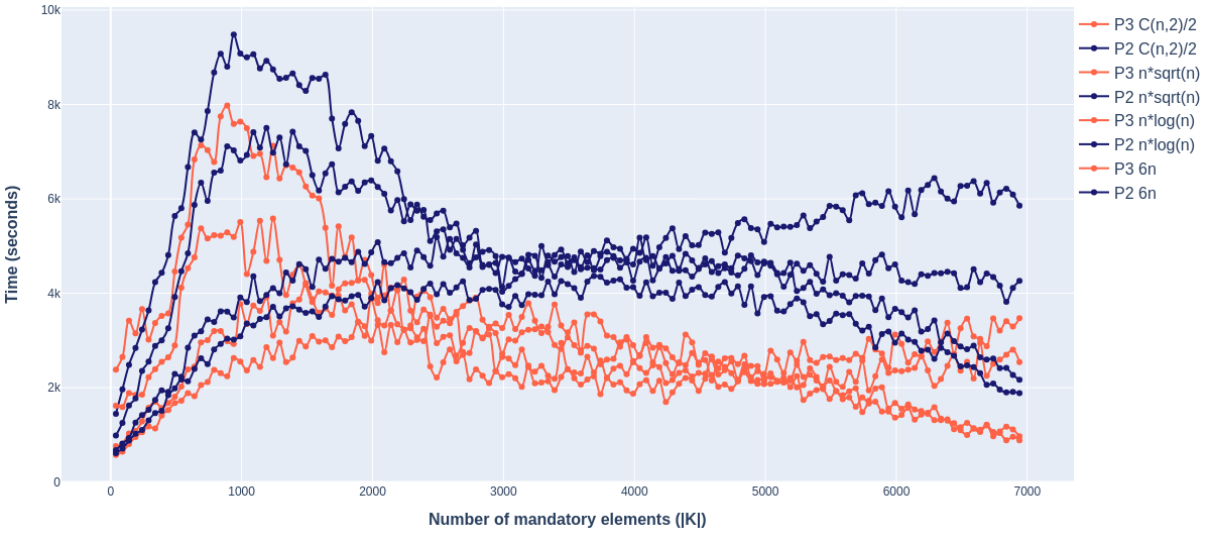


Figure 23 – Comparison of the execution time versus the size of K between **P2** and **P3** using yes-instances with $n = 25{,}000$

One possibility for this behavior is that the actual problem complexity is the same, independent of the strategy used, between the two used, explaining the similar patterns. The shift could be an implementation consequence where one algorithm takes more computational resources to perform similar evaluations. Another possible assessment is that even given the possibility of other unknown solutions in the instances they are unlikely, this would also help to explain the very similar behavior since both algorithms have little choice but to find similar answers if not equal ones. This would not be the case on entirely

random instances, as there would be several cycles that could be found that would result in a no-instance situation.

4.3 Hidden solutions experiments

In previous sections, we introduced four key objectives for this research, those being:

1. How does the size of n influence the execution time?
2. How does $|K|$ influences the execution time?
3. Are positive instances harder to solve than negative instances?
4. If the instance is positive, do any particular properties of the solution influence the execution time?

So far, we have explored the three initial points; in this section, we will shift our attention and elaborate on the last point, which has not been discussed so far. The goal is to investigate how the solution itself affects the execution time and, by proxy, try to hypothesize on the problem properties. The focus is on exploring the detection of predefined characteristic subgraphs in randomly generated graph instances.

Several instances were generated for each arbitrarily decided graph type, and both algorithms **P2** and **P3** were used to solve the same instances. To further isolate other variables for all the experiments in this section, $|K|$ is fixed in 1.000, n is $25{,}000$ as before, and the x-axis will reflect the solution size since K is set. The number of points on the x-axis will change between experiments to better conform to each solution graph as needed. It is important to note that for all the following experiments, the pre-determined solution is one possible solution, but it may not be the only solution.

Seven datasets were generated; they were collectively named Hidden solution datasets and are defined in Table 5. These datasets aim to further drill down on positive instances and detect if specific constructs inside the solution can influence the execution time. To that end, each dataset is made of positive instances only. However, those solutions were generated before the rest of the instances, and the remainder of the graph was added around this known solution. Each dataset has solutions with a different structure to assess how the implemented algorithms will behave and how different solution characteristics affect the overall execution time profile. Each dataset's special property is noted on the *Notes* column. All solution sizes needed to be bigger than that value to avoid the instances being trivially positive. Based on the restrictions presented by the individual properties, a range for the size of the hidden solution was chosen; this range is represented in the solution size column. The number of instances denoted on the *# instances* column was generated for each combination of density and solution size.

The datasets are described as follows:

d-path: All known solutions in instances of this dataset are paths. For instance, if the solution has a size of $3{,}500$, there are $1{,}000$ mandatory elements randomly distributed on a path of size $3{,}500$. The extremities of the path are guaranteed to be a part of K; otherwise, there would be a known solution smaller than intended. Since paths are simple structured

Table 5 – Hidden solution datasets

Name	edge densities	Solution size	# instances	Notes
d-path	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 1,100 End: 10,000 Step: 100 Total: 90	5 per combination 1,800 total	Every instance has at least one solution that is a path K is randomly distributed on the path
d-path-shortcut	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 2,100 End: 10,000 Step: 100 Total: 80	5 per combination 1,600 total	Every instance has at least one solution that is a path K is randomly distributed on the path Elements of K that are sufficiently separated are directly connected through a shorted path
d-btree-leaves	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 2,050 End: 9,950 Step: 100 Total: 80	5 per combination 1,600 total	Every instance has at least one solution that is a binary tree the tree is complete but not full Element of K are either on the root or on the leaves
d-btree	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 2,050 End: 9,950 Step: 100 Total: 80	5 per combination 1,600 total	Every instance has at least one solution that is a binary tree the tree is complete but not full Element of K are distributed on the entire tree
d-star-1k	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 2,001 End: 11,001 Step: 1,000 Total: 10	15 per combination 600 total	Every instance has at least one solution that is a star The star has 1,000 branches
d-star-100	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 1,101 End: 10,001 Step: 100 Total: 90	5 per combination 1,800 total	Every instance has at least one solution that is a star The star has 100 branches
d-star-20	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 1,100 End: 10,000 Step: 100 Total: 90	5 per combination 1,800 total	Every instance has at least one solution that is a star The star has 20 branches 2 of those branches have 80% of the solution vertices
d-star-no-touch	$6n$ $n \log n$ $n\sqrt{n}$ $\binom{N}{2}/2$	Start: 1,101 End: 10,001 Step: 100 Total: 90	5 per combination 1,800 total	Every instance has at least one solution that is a star The star has 100 branches Branches are not connected, except for the star center

trees, this dataset was intended to verify whether simpler structures are easier to find inside the input.

Figure 24 shows an illustrative example of an instance on the dataset. It does not conform to the densities and actual size of the graph and k, as that would be unprintable. However, it does provide a visualization to help readers better understand the intrinsic constructions and the overall objective when constructing the instances. The blue edges indicate the hidden solution inserted on the graph, while mandatory vertices are denoted by the gray color. It is important to note that there are other possible solutions beyond the hidden one in the example that are intentional, as they may happen in actual instances.

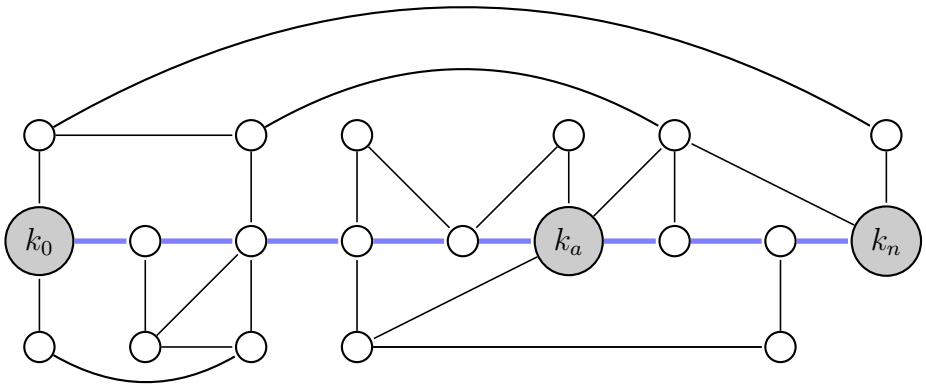


Figure 24 – Illustrative example of hiding a path inside the graph.

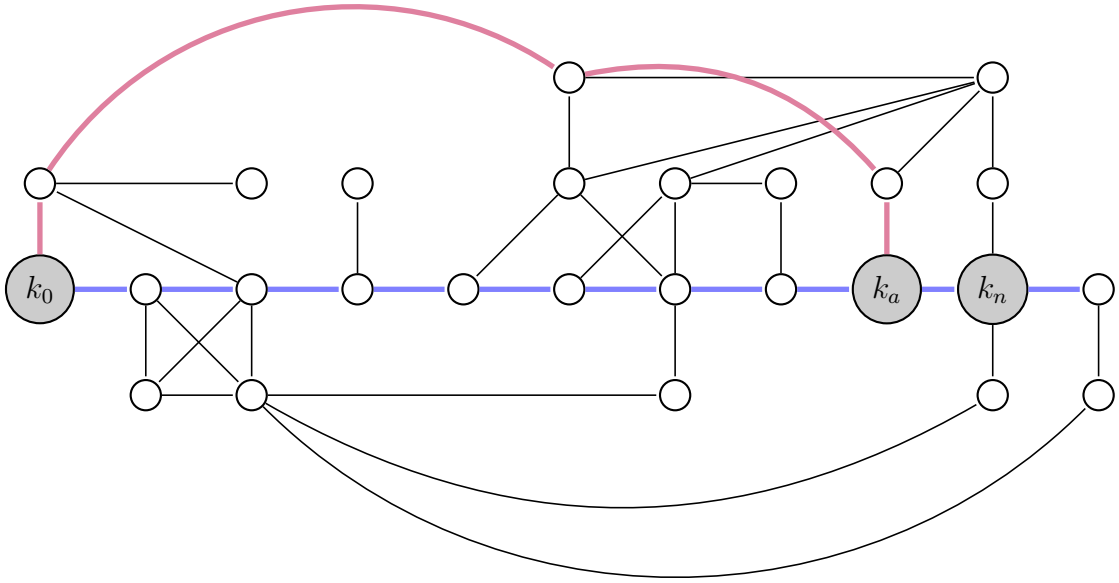


Figure 25 – Illustrative example of hiding a path with shortcuts inside the graph.

d-path-shortcut: All known solutions in instances of this dataset are paths, just like the previous datasets; however, for every pair of K that were more than 50 vertices apart, a simple path of at least two vertices was added between that pair, effectively forming shortcuts between the mandatory elements. This experiment was designed to see if the algorithms could catch up on these shortcuts and if the overall average increase in (arguably) better options would increase efficiency or distract the algorithm and increase the execution time by providing more options to explore.

Figure 25 shows an illustrative example of an instance on the dataset. The blue lines indicate the hidden path, and the purple line indicates a shortcut between k_0 and k_n. Since those two mandatory vertices were farther away than the fictitious threshold of the illustrative example a shorter path (shortcut) was introduced between them, providing an alternative solution that is significantly shorter than the original hidden path. The shortcut is not guaranteed to be the shortest path between its two endpoints, nor are they guaranteed to be the only shortcuts that exist between mandatory vertices in the graph, other similar constructions may happen randomly while adding the vertices that are not part of the solution.

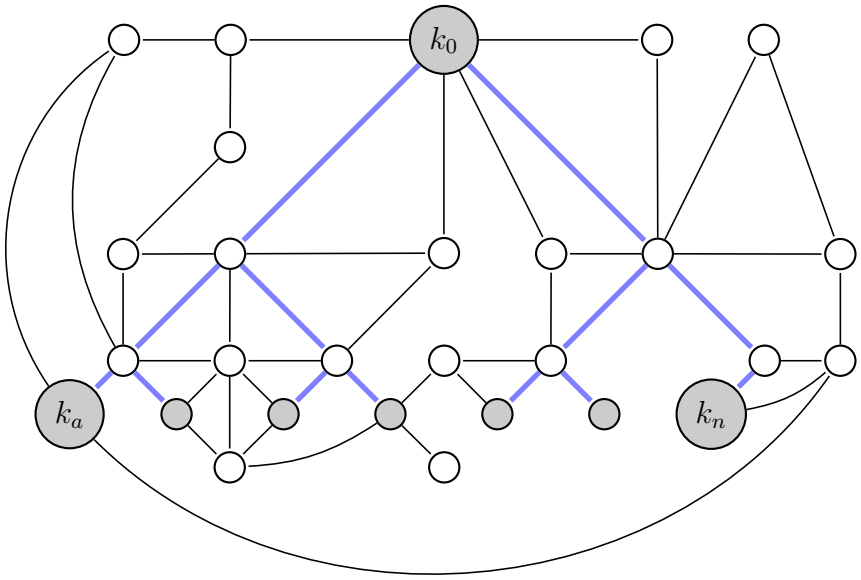


Figure 26 – Illustrative example of hiding a binary tree with k on the leaves inside a graph

d-btree-leaves: Instances where the solution is a complete binary tree (but not full) with the root node being a member of K and the other elements of this set on the tree's leaves. For this experiment, the tree size varied from $2{,}050$ to $9{,}950$ in steps of 100. Note that for all the chosen sizes, we have more than $1{,}000$ leaves; thus, K was randomly picked from the available leaves.

Figure 26 shows an illustrative example of an instance on the dataset. The hidden binary tree is denoted by the blue edges, and all the gray vertices are mandatory, including the root, k_0, and all the tree leaves. A complete binary tree is defined as a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible, which is the case both for the example as well as the actual instances. However, the trees are not necessarily full, as a full binary tree (sometimes proper binary tree or 2-tree) is one in which every node other than the leaves has two children; as seen in this example, the parent of k_n only has one child.

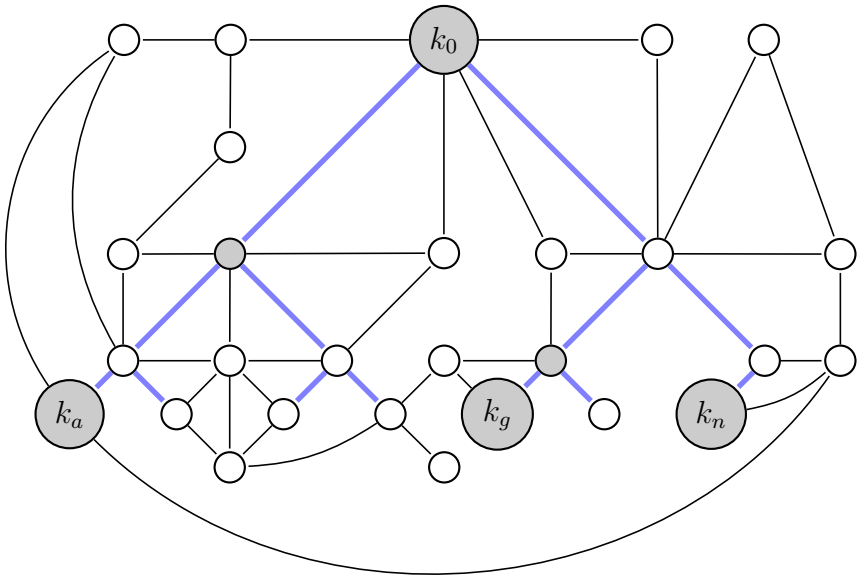


Figure 27 – Illustrative example of hiding a binary tree inside a graph

d-btree: Instances where the solution is a complete binary tree (but not full) with the root node being a member of K, two other members of K being the leftmost and rightmost leaves, eight other elements of K randomly distributed on the leaves, and all other elements are randomly picked anywhere on the tree. For this experiment, the tree size varied from $2{,}050$ to $9{,}950$ in steps of 100.

Figure 27 shows an illustrative example of an instance on the dataset. As with the previous dataset, we have a complete but not full binary tree hidden inside a graph; the hidden tree edges are highlighted in blue, and mandatory vertices are marked in gray. However, the positions of the mandatory vertices have changed from the previous example. On this one, the mandatory vertices can be located anywhere on the tree. However, to keep the tree with the intended size, both the leftmost (k_a) and rightmost (k_n) leaves are mandatory.

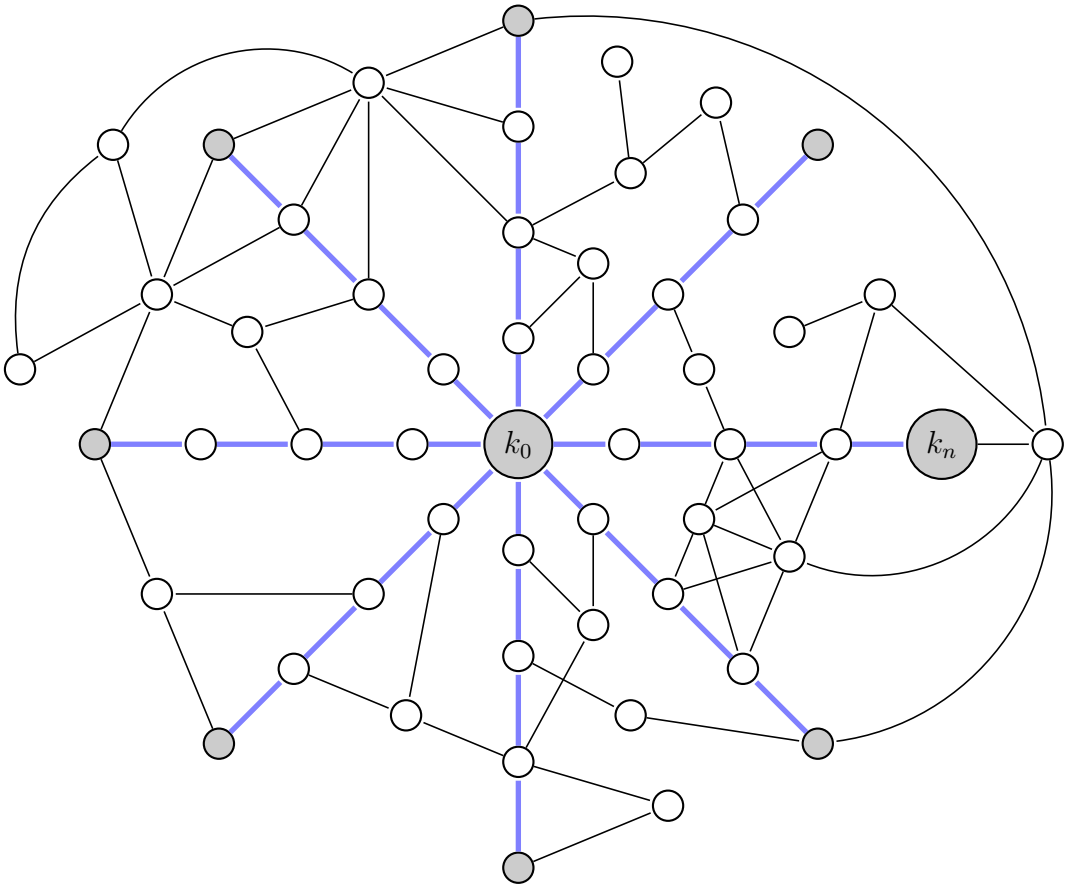


Figure 28 – Illustrative example of hiding a multi-level star with $1{,}000$ branches inside the graph.

d-star-1k: Usually a star S_i is defined tree of order i with maximum diameter 2. From hereon, we loosen that definition and refer to stars as having the same structure without the diameter restriction, a multi-level star. *I.e.* a star is defined as a collection of paths (branches) connected by a single, central vertice (root). This dataset consists of a star where the number of branches was fixed at $1{,}000$. Each branch has a mandatory element on its tip, and the root is also mandatory. Thus, we have one extra mandatory element for this experiment, making $|K| = 1,001$. The size varied from $2{,}001$ to $11{,}001$ in steps of $1{,}000$ – *i.e.*, one extra vertex per branch per step.

Figure 28 shows an illustrative example of an instance on the dataset. The hidden star edges are denoted by the blue edges; the example intentionally shows the extremities of the branches as mandatory vertices (gray) and a root element (k_0) that is also mandatory. Once again, the hidden solution is not necessarily the only or smallest. In this example, k_n could be connected to the mandatory node below it using a single extra node instead of the 3-length branch path.

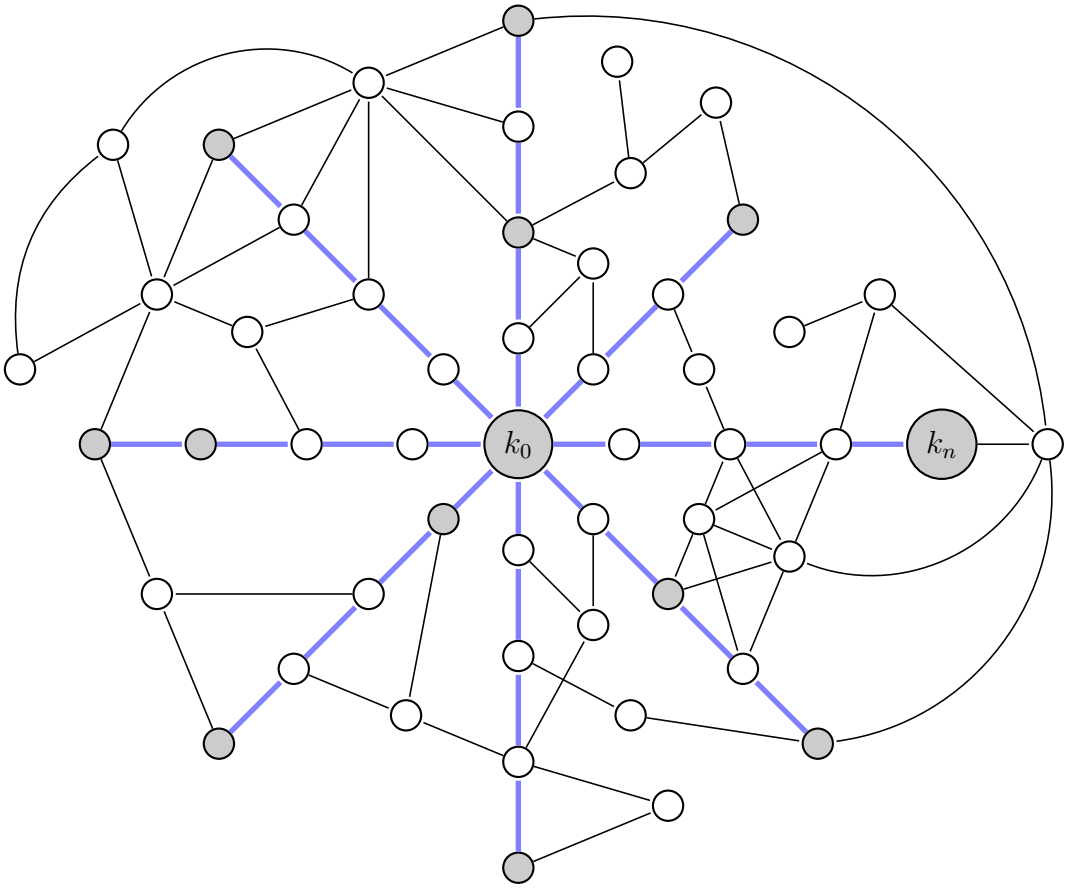


Figure 29 – Illustrative example of hiding a multi-level star with 100 branches inside the graph.

d-star-100: Consists of a star graph where the number of branches was fixed at 100. Each branch can not differ in size from any other branch by more than 1 vertex; the star's root element and all the branches' tips are members of K, and the remainder of the mandatory vertices were randomly assigned through the star-like graph. The size varied from $1{,}101$ to $10{,}001$ in steps of 100.

Figure 29 shows an illustrative example of an instance on the dataset. The difference between this example and the previous one, since the examples do not comply with the actual instance branch count, is the position of the mandatory vertices. Previously, they were all positioned at the extremities of the star; now, they are on the extremities but also in the middle of the structure. Since the number of mandatory elements does not change in the actual instance, this implies that the number of branches had to be reduced.

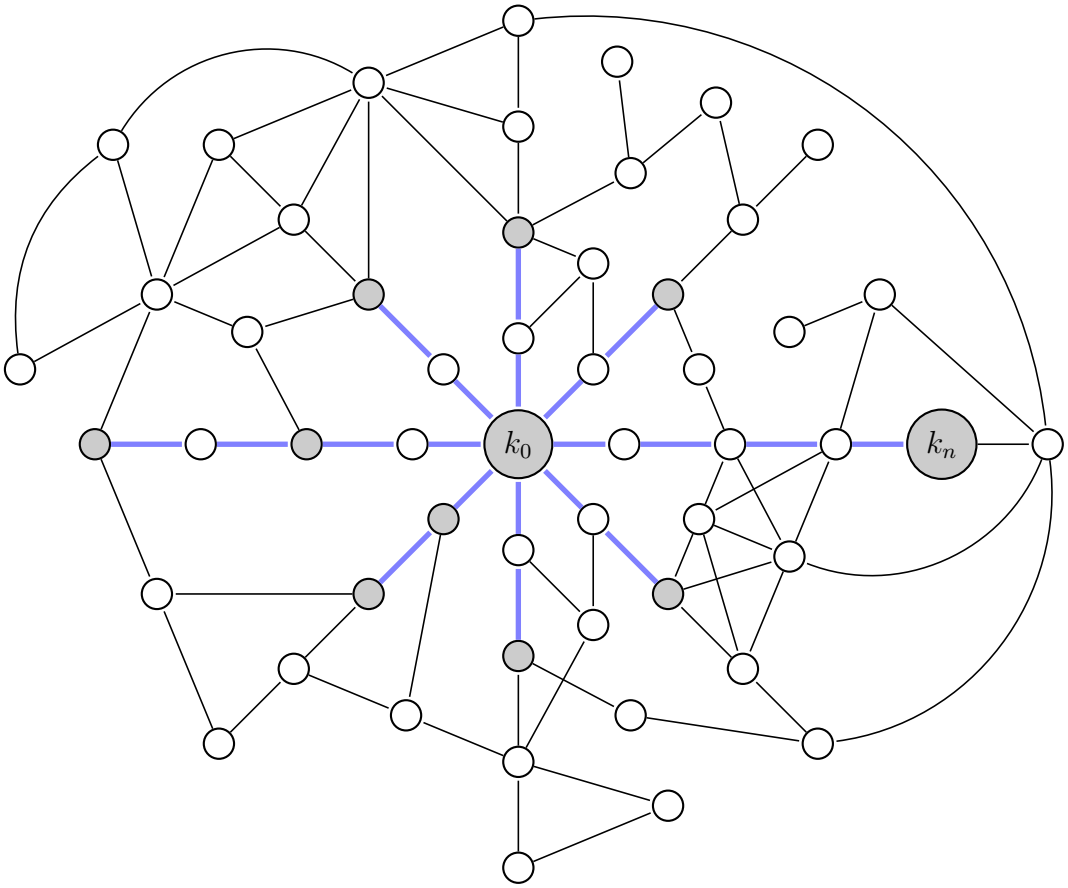


Figure 30 – Illustrative example of hiding a multi-level star with 20 branches inside the graph.

d-star-20: the stars' branches were fixed at 20, where 18 of those branches contain approximately 20% of the total vertices, diverging in size by at most 1 vertex. The remaining 2 branches will have the other 80% of the vertices and diverge by at most 1 vertex between them. This experiment was designed to be a middle term between a star and a path, and answer the following question: which is more complex when the average distance of K is the same: having a few significantly distant pairs of K or having all elements of K within a similar distance? For this experiment, the star's size varied from $1{,}101$ to $10{,}001$ in steps of 100.

Figure 30 shows an illustrative example of an instance on the dataset. Two branches are bigger than the others in this version of the hidden star. However, the mandatory vertices are still spread across the entirety of the star as well as on the extremities.

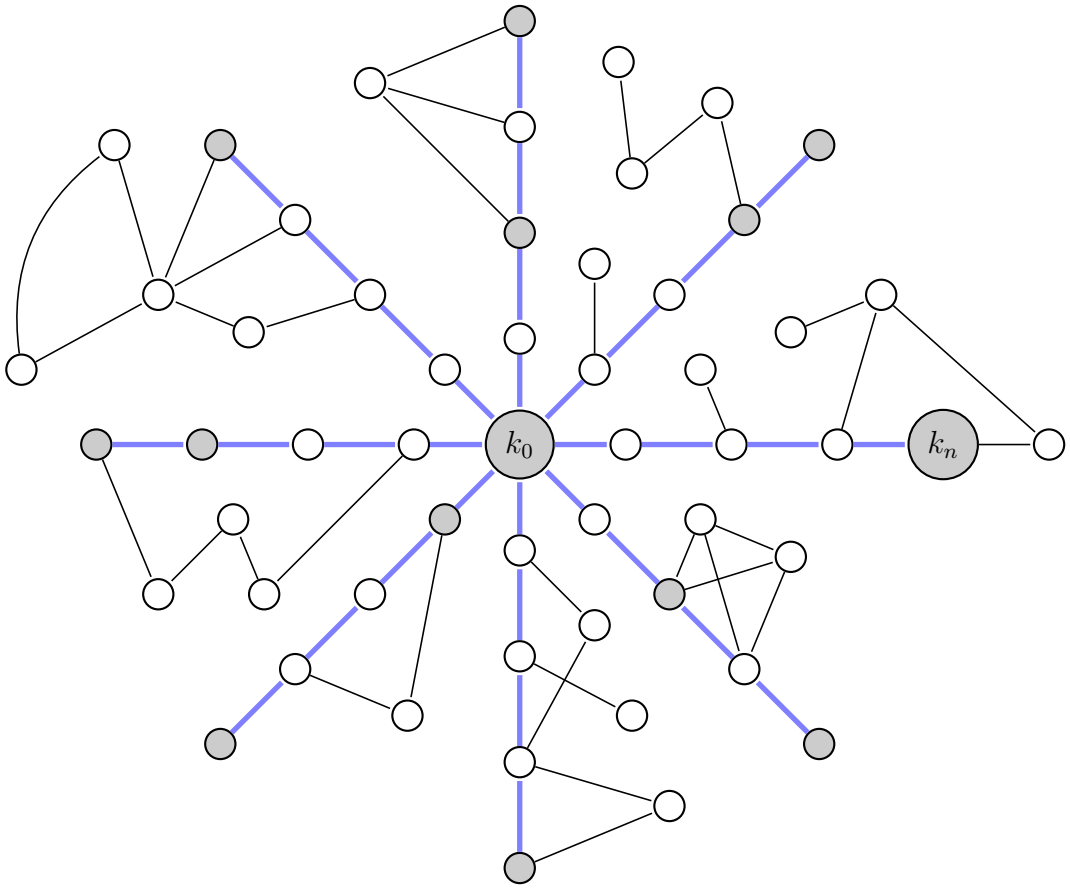


Figure 31 – Illustrative example of hiding a multi-level star graph where the branches do not touch inside the graph.

d-star-no-touch: Consists of a star graph where the number of branches was fixed at 100. Each branch can not differ in size from any other branch by more than 1 vertex; the star's root element and all the branches' tips are members of K, and the remainder of the mandatory vertices were randomly assigned through the star-like graph. Vertices assigned to a given branch could not connect to vertices assigned to another branch. One might think of these constructions as 100 graphs, each with its own known path solution hidden inside. All those similar graphs had their path solutions connected to a k_0 arbitrarily chosen. This experiment was designed to be compared to both the $d - path$ and $d - star - 100$ datasets and to provide insights on how the existence of clearly defined subproblems could influence the overall behavior of the problem's hardness. For this experiment, the star's size varied from $1{,}101$ to $10{,}001$ in steps of 100.

Figure 31 shows an illustrative example of an instance on the dataset. In this final star-related example, each vertice, not part of the solutions, is assigned and connected to a single branch. If each branch were to be separated from the others by duplicating the root element (k_0) and detaching it from the original, we would end up with one graph with a hidden path solution per branch.

This provided us with another set of tens of thousands of instances to solve, and again, both algorithms solved each dataset. In all executions, CPLEX ran at the default configurations but was limited to 16 threads. Once more, due to the number of data points generated on the results, we will present a simple average of the numbers for each data point. And, again, for the curious readers, the computational time used exceeded one year.

It is also important to reinforce that, as explained before, the process of generating yes-instances is different from random instances, and due to the compromise of generating instances on a reasonable time and ensuring that there is a valid solution with specific properties, the vertices belonging to K may have slightly different properties from the random vertices not included in the solution, because of that some properties may be unintentionally skewed from their intended values. For example, vertices on the solution may present lower average edge density than those not, causing the overall graph edge density to be slightly lower than intended; maintaining the properties to their nominal values was performed on effort cases as some instances would be prohibitively complex to generate if those properties were hard-enforced. Even when a reasonable part of the graph is on the solution, we found no evidence that these reasonably small skews affected the results and conclusions.

For each upcoming subsection, several experiments were performed; however, in the interest of effective use of the space, most of them will be only listed with the results being briefly mentioned while we focus and dive deeper only into the ones with more exciting characteristics that better represent, or conflict, the other tests. All the data that is not directly discussed is available on <https://github.com/lucas-subli/phd_thesis_data/archive/refs/tags/V0.20.zip> or, in the case of charts, in Appendix A. Finally, to better visualize the charts, there will be three yellow lines parallel to the x-axis, with the center one being a continuous line and the other two being dashed. Those lines will represent the smallest, average, and highest execution times for that same algorithm on that same density for $|K| = 1{,}000$ using yes-only instances. The objective of those lines is to give the reader an easy way to assess if, overall, that specific solution type changes the behavior when compared to random instances.

4.3.1 The solution is a path

For this category of experiments, both algorithms were used to solve instances with all four previously selected densities where the solution was known to be a path. Two datasets fall inside this path category **d-path** and **d-path-shortcut**

Figure 32 shows the result of **P3** solving **d-path**. In this figure and in others where it shows moving forward, the yellow continuous line indicates the average execution time of that same algorithm for **d-random-yes** with $|K| = 1{,}000$ the upper and lower yellow dashed lines indicate the fastest and slowest instances of that same algorithm solving **d-random-yes** with $|K| = 1{,}000$, it is a visual form of accessing if the instance is, on average, easier or

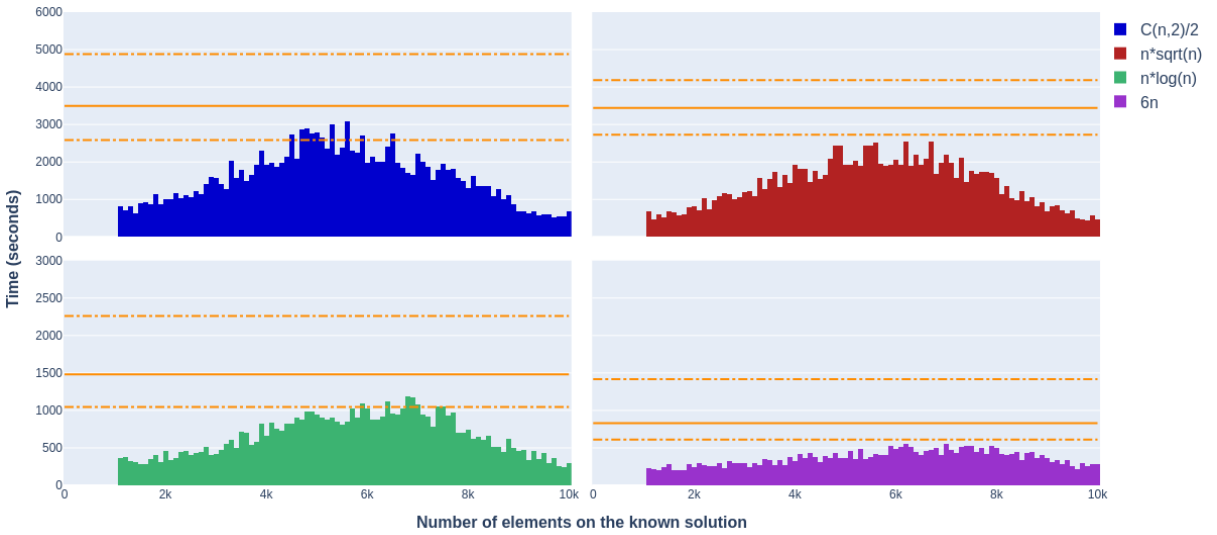


Figure 32 – Execution time versus of the size of the solution for **P3** solving **d-path**

harder to solve than a random yes-instance. The gross results are as one would naturally predict when solving instances where the solution is the most straightforward construction possible: the execution time is lower than the time for random instances across the board. We can see that most of the data is faster than the fastest instance for the same size of K compared to random instances. What can be considered unintuitive is the fact that the results also present this behavior of ramping up, plateauing, and then ramping down. It seems to be another case where the execution time starts lower and goes up as we increase the hardness (size of the solution), but it goes down again when the solution is bigger than some threshold. This effect is more accentuated on higher densities. We only show the chart for **P3**; however, algorithm **P2** performed similarly to this one, with different thresholds and execution times, but the conclusions presented here also apply to that algorithm. That being said, it still lagged behind **P3** and showed no difference in the relative behavior between the two algorithms.

Another path-related experiment was also performed. Once again, both algorithms were used to solve dataset **d-path-shortcuts**. As stated before, for this experiment, the path size varied from $2{,}000$ to $10{,}000$ in steps of 100.

Figure 33 shows the comparison between **P2** results for both **d-path** and **d-path-shortcut**. The algorithm seems capable of detecting and sticking to shorter paths to some extent, as the execution times are consistently shorter and more grouped despite the difference being minimal. Beyond that, there are no big changes between the individual algorithms, **P3** dominated **P2** for most of the experiments with some amount of overlap where both of them were equally fast, but at no moment **P2** came out ahead. As mentioned before, additional charts for **d-path-shortcut** including, but not limited to, comparisons between **P2** and **P3** are available on Appendix A; however, since they don't add significantly to the discussion they are omitted from this section.

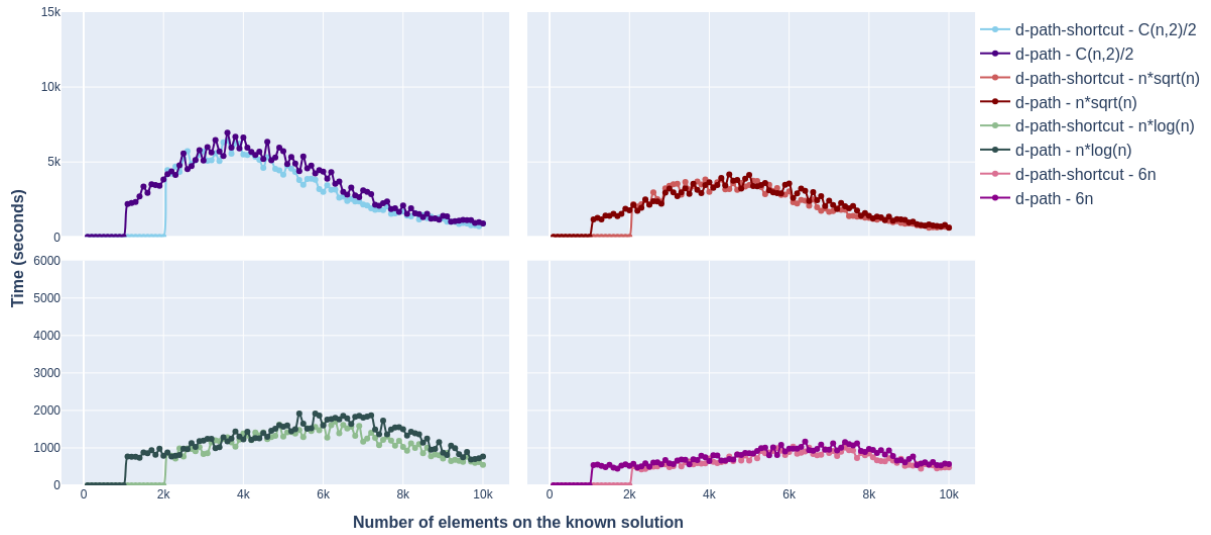


Figure 33 – Comparison of execution time versus of the size of the solution for **P2** solving **d-path** and **d-path-shortcut**

4.3.2 The solution is a binary tree

For this category of experiments, both algorithms were used to solve instances with all four of the previously selected densities within two different datasets **d-btree-leaves** and **d-btree**.

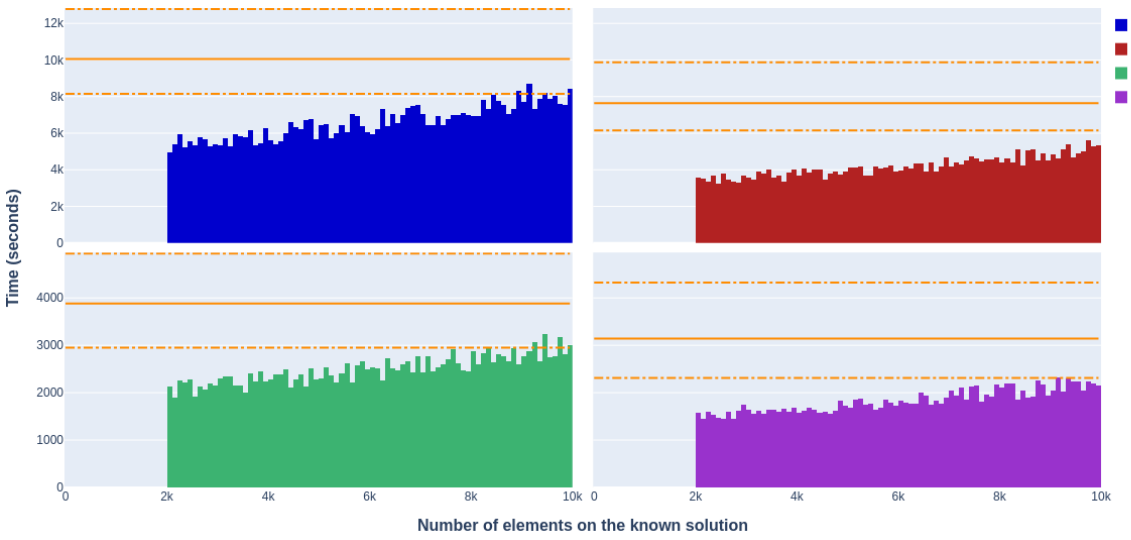


Figure 34 – Execution time versus of the size of the solution for **P2** solving **d-btree-leaves**

For **d-btree-leaves**, once again, the overall intuition was confirmed. Both algorithms presented higher execution times when compared to the path solutions but lower than their random instances experiments. Figure 34 shows the results for algorithm **P2** solving **d-btree-leaves**; the execution times start lower than the fastest executions for random instances and slowly ramp up to the minimum value by the end of the experiment. This behavior of simply ramping up, with no threshold and no changing to a ramping down behavior, is divergent

from previous experiments. Currently, it is not clear why the path and tree solutions have such different chart profiles. This profile is much easier to try to explain as it seems intuitive that more significant trees tend to have more neighbors and require further exploration of the original graph before being found. Finally, the behavior of both algorithms was similar in both cases, but with an execution time gap between them, assuring that **P3** dominated **P2** across the board thus, we will omit the data for **P3** on the same grounds of not bringing in new information.

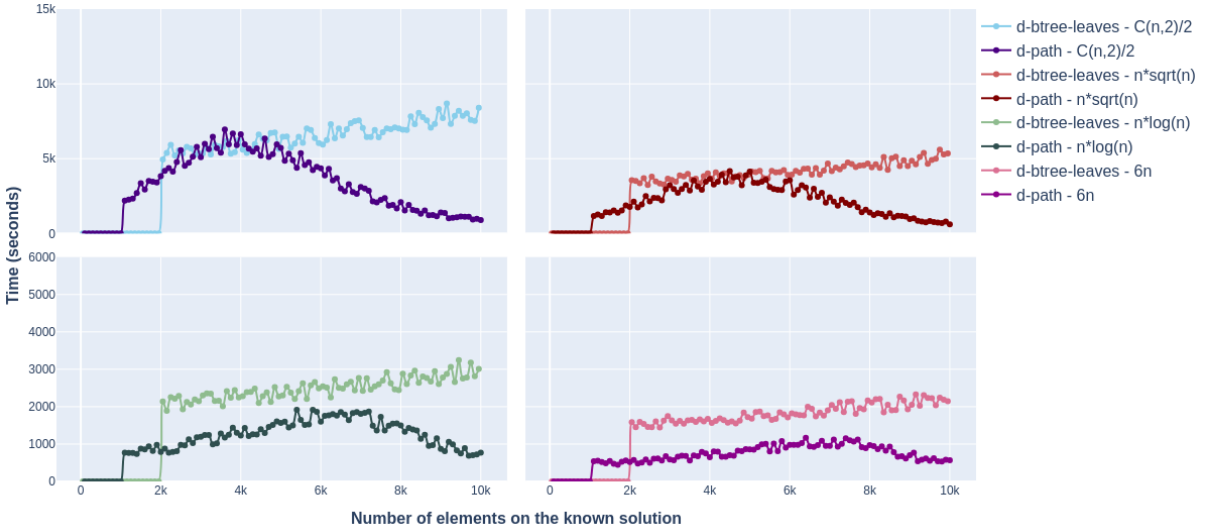


Figure 35 – Comparison of execution time versus of the size of the solution for **P2** solving **d-path** and **d-btree-leaves**

When comparing **d-btree-leaves** with **d-path** we can see two different trends across all densities in Figure 35, while **d-btree-leaves** showed a continuous increase in execution time from the start to the end of the experiment, while **d-path** showed a increase-decrease behavior. d-path started with a lower initial value in all instances and in all of them presented an initial growth rate above the one in **d-btree-leaves**. In the two higher densities, it eventually reaches similar times to the other dataset, but as soon as that intersection takes place, the execution time starts to drop, making it significantly faster than the competition. In the two other instances, despite not reaching the same execution times as **d-btree-leaves** at the peak, the behavior is the same. This confirms that, for the presented datasets, not only a path seems to be faster to solve, but it also seems to have an extrapolation that indicates that this trend continues. From this data, which has the same profile as **P3**, we can infer that **d-btree** dataset is harder to solve than **d-path**.

Figure 36 shows the results for algorithm **P3** solving **d-btree**; the execution times start lower than the fastest executions for random instances and slowly ramp up. This behavior is similar to the previous dataset but with a smaller starting point, no significant change in ramp-up rate, and more minor variance across the board. One hypothesis is that, by having more elements randomly dispersed tree, those elements help guide the algorithm

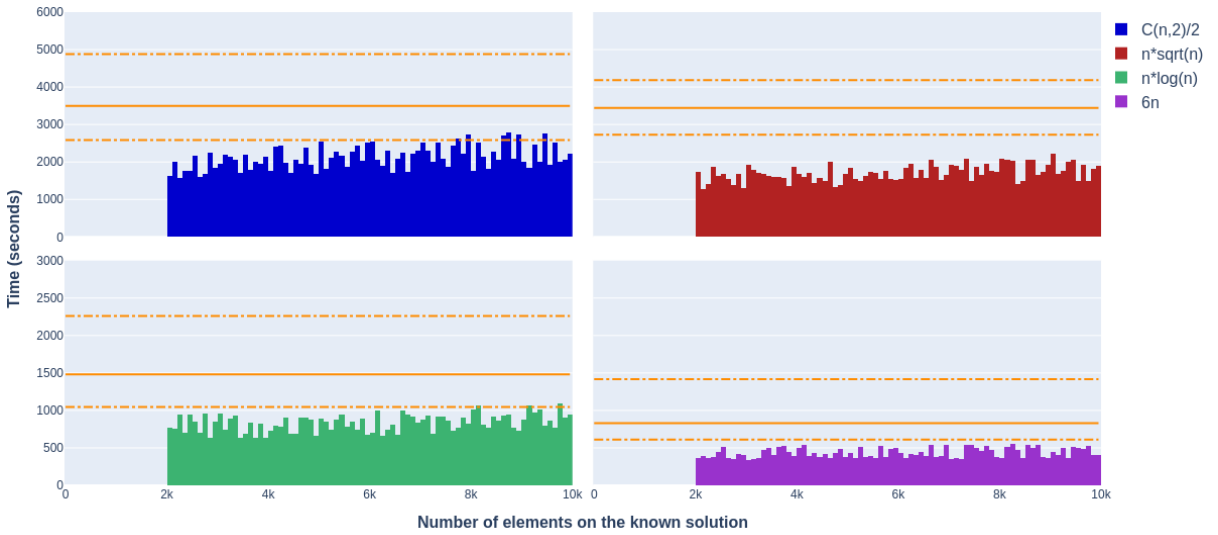


Figure 36 – Execution time versus of the size of the solution for **P3** solving **d-btree**

towards the answer, as it can have partial results much faster that will build up to the final result when compared to having the elements on the tree leaves. This would also lead to more consistent execution times, as the variability tends to be smaller as mistakes or wrong attempts are not as significant due to the smaller subtrees constructed by having the mandatory elements from the non-leaf vertices of the tree connected to the partial solution. Once again, the behavior of both algorithms was similar when comparing their executions with different datasets.

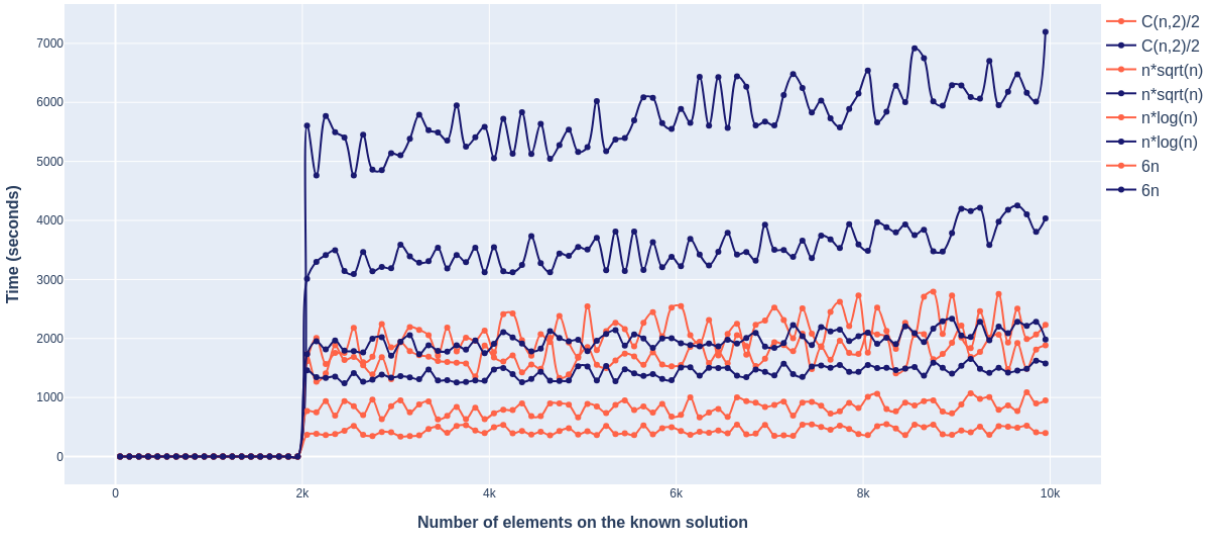


Figure 37 – Comparison of execution time versus of the size of the solution for **P2** and **P3** solving **d-btree**

Figure 37 compares the execution times for both **P2** and **P3** solving **d-btree**. The results are very similar to **d-btree-leaves**. The data shows how algorithm **P3** outperforms algorithms **P2** by a significant margin. The slower executions of **P3** (higher densities) are on par with the fastest executions of **P2** (smaller densities), despite similar execution times **P3**

sustained a higher variance in this unfair comparison, which could indicate the correlation between density and variance is not dependant on the algorithm itself, as **P2** also presented a higher variance on higher densities. We can also see that the ramp-up ratio is more accentuated on the higher densities of **P2** but present in all experiments nonetheless. It is also interesting to see that both algorithms present a clear gap between the two lower densities and the two higher ones, but while **P3** has the two higher densities close together, even with some small amount of overlapping, **P2** presents a much clearer separation between them.

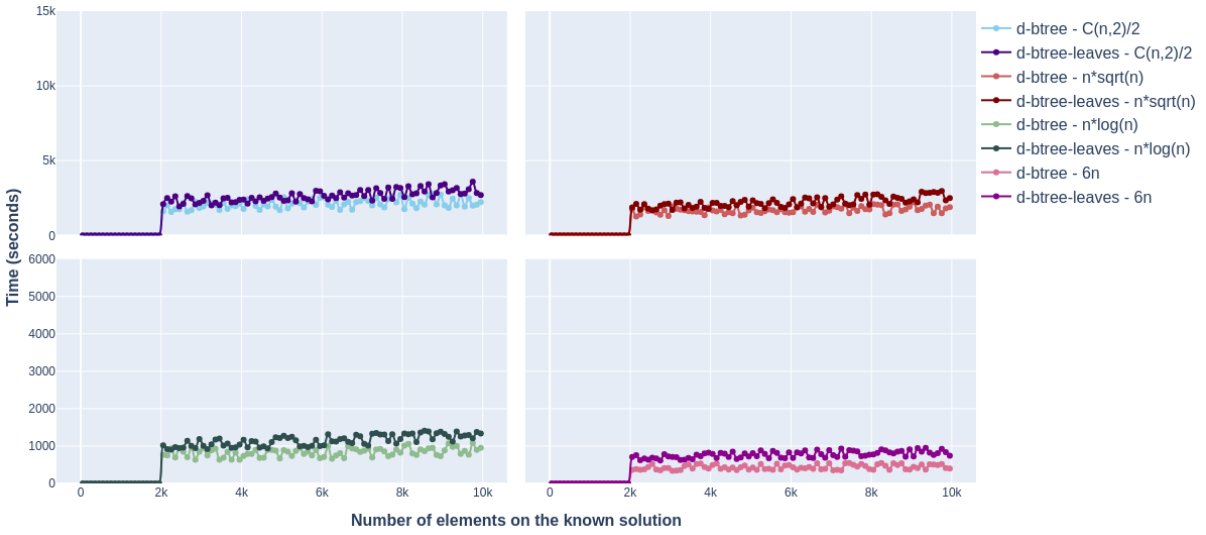


Figure 38 – Comparison of execution time versus of the size of the solution for **P3** solving **d-btree-leaves** and **d-btree**

Finally, Figure 38 shows the comparison of **P3** solving both tree instances. As expected, having the elements of K spread randomly across the tree has led to a faster execution time across the board. This can be explained as those vertices are closer to the tree's root and should be easier to get to, generating intermediate solutions by including them before reaching the leaves and guiding the algorithm to the vertices on the leaves in a certain way. Both datasets presented a curve that keeps growing as the experiment unfolds, and the gap between them seems to stay constant, which may be unexpected, as the bigger the tree, the more efficient partial solutions should make the process, however that does not seem to be the case, from the data available.

4.3.3 The solution is a star

For the final set of experiments, we will explore three different datasets where the solutions are different forms of a star. Firstly, **d-star-1k**.

Figure 39 shows the data for algorithm **P3** solving **d-star-1k**. Because of the large step size this chart has a smaller resolution than in previous experiments, but it is possible to see that the initial execution time is very low, which should be expected as the mandatory

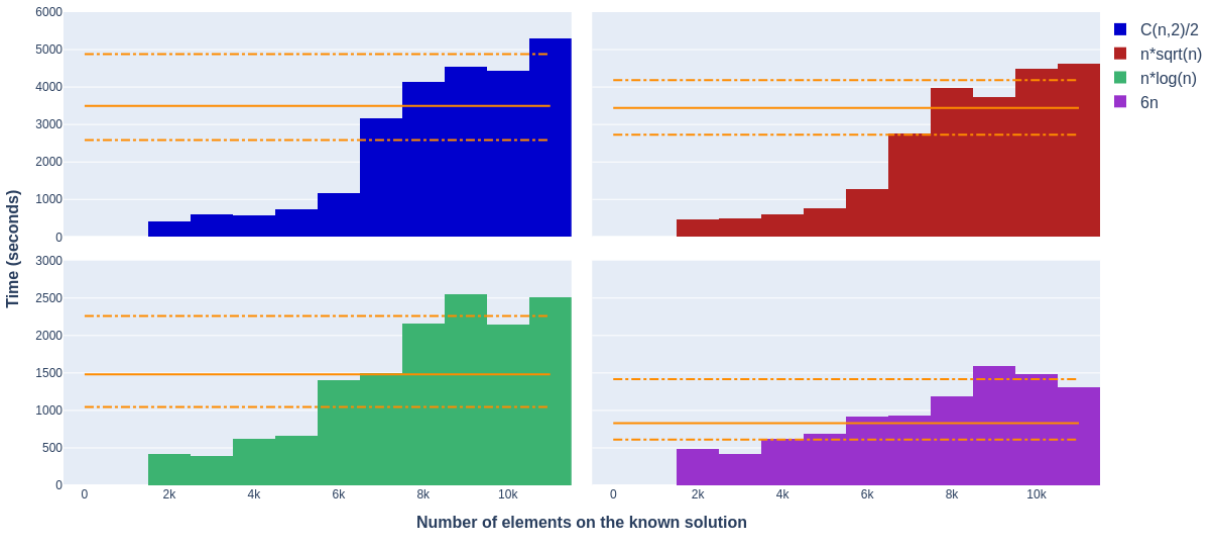


Figure 39 – Execution time versus of the size of the solution for **P3** solving **d-star-1k**

vertices are just a couple of vertices away from other mandatory vertices. As we get to branches of size 7 or 6, depending on the density, we have a spike in execution time followed by unstable increases in subsequent branch sizes, although not as steep as the previous increases; in all cases, there is the occurrence of small decreases, but there is a clear tendency on increases. The execution times were shorter than the fastest executions for the random yes-instances at first; however, at step 7 or 6, where the big increase takes place, the execution time gets closer to the average, with some later steps surpassing the average. Some data points show an average execution time greater than the worst for the random yes-instances. This behavior is also replicated in the results from **P2**, with the spike in execution time also taking place in similar steps.

The results with small stars are as expected, with very low execution times as most of K is contained on a small radius, and just exploring the second or third-degree neighbors of one arbitrary element of K would net the result. However, at higher star sizes, the boost in execution times increases exponentially; these results show that solutions similar to stars have the potential to be more challenging to solve than average random solutions. This is the first experiment that could reach and even clearly surpass the difficulty of the random yes-instances. The value on the x-axis, where there is a significant increase in execution time, is also worthy of note; it is not clear if that specific threshold is a property of the problem, the algorithm, or the implementations. It is also interesting to note that, due to the relatively large number of branches, the most distant any two elements of K could ever be was 18 vertices, which is not a high number for the instance size. However, this distance was consistent, whereas, in other experiments, we could have bigger distances between specific mandatory vertices, but on average, their distances would be shorter. The closest vertex of the solution also seems to play a part in this perceived difficulty, as the closest one would be the distance to the center of the star, 10, at the last value on the x-axis.

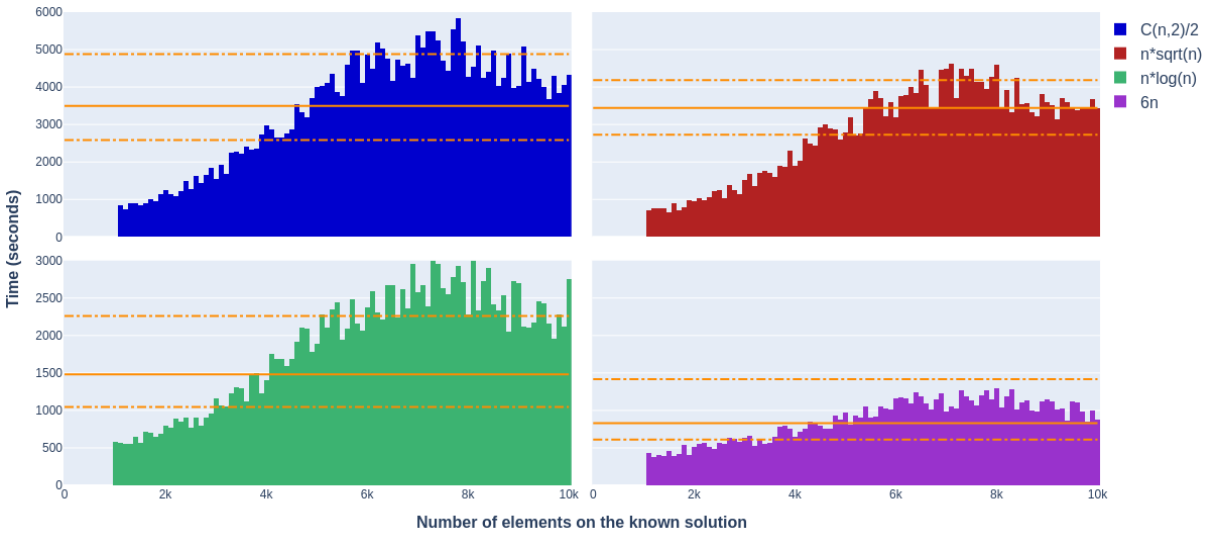


Figure 40 – Execution time versus of the size of the solution for **P3** solving **d-star-100**

For the next set of experiments, solving dataset **d-star-100**, where the number of branches was fixed at 100. The results in Figure 40 show that these instances, at larger solution sizes, are more challenging than even the harder random instances with similar $|K|$. Despite this happening on the previous dataset, the behavior is more pronounced in this result and happens at smaller solution sizes. The ramp-up behavior is still present, and there is an indication of a ramp-down despite not being fully visible. This data reinforces that stars seem to be harder to solve than paths and binary trees by a reasonable margin. However, the number of branches seems to be less influential than the overall diameter of the start. At least at the sizes tested. Since the results for **P2** are mostly similar but scaled to their own parameters, they will be omitted.

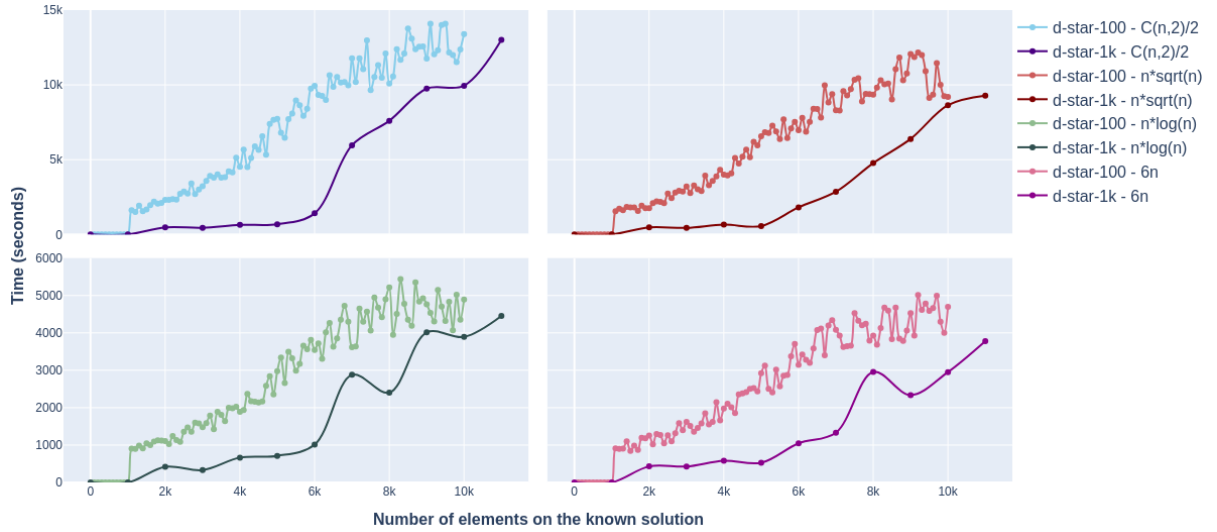


Figure 41 – Comparison of execution time versus of the size of the solution for **P2** solving **d-star-1k** and **d-star-100**

When comparing **d-star-1k** versus **d-star-100** Figure 41 shows that, overall, **d-**

star-100 seems to be harder to solve, as the execution times completely overpower the competition. That being said, **d-star-1k** seems to ramp up faster, which could lead to an overtake if we extrapolate the data with more data points on the x-axis. However, that is not an extrapolation in which we would have high confidence, as we have seen multiple times in this paper, that the data seems to reach peak execution times and engage in a ramp-down behavior eventually; it is not clear if or when that behavior could take place for either dataset. With that out the way, one possible explanation for the increased execution time on our data would be the increased radius around the root vertex between the datasets in question. By having fewer branches, the final vertice of each vertice gets further away from the initial mandatory element, which could lead to an increase in the minimum search space needed to find a solution; this explanation is in line with the one provided for the **d-star-1k** analysis.

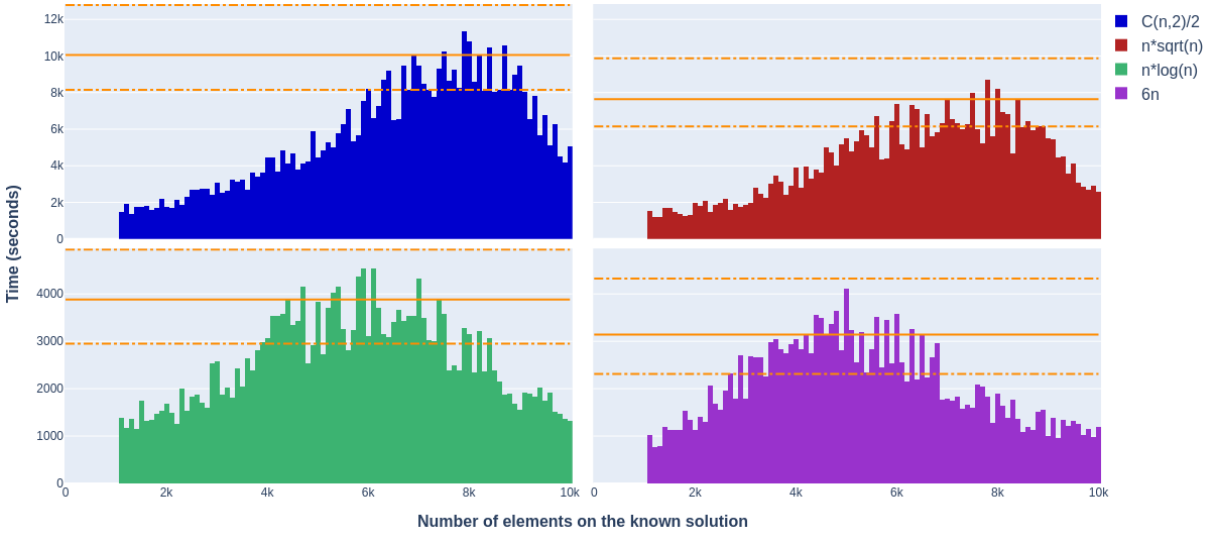


Figure 42 – Execution time versus of the size of the solution for **P2** solving **d-star-20**

For **d-star-20**, where the stars' branches were fixed at 20, and 18 of those branches will contain approximately 20% of the total number of vertices, we can see the results for **P2** in Figure 42. On this dataset, **P2** also started performing better than it did for randomly generated yes-instances and had the execution time increase as the star size increased. It also reverted to the three-step behavior of ramping up, plateauing, and ramping down, the same behavior seen in the random experiments, the yes-instance experiments, and the path-related datasets. However, on all but the less dense instance, the ramp-up seems slower than the ramp-down, which differs from the random experiments with similar profiles. The data seems to mix the difficulty of the stars with this three-step behavior last seen in the path-related experiments. In hindsight, this makes sense as having two branches significantly larger than the others; the problem can be seen as a small star appended with a significantly large path. The chart also shows the same previously observed behavior, where the peak of execution time shifts towards the right side of the x-axis as the density increases. We can also see execution times higher than the average for random yes-instances. Still,

they are fewer than previous star configurations, and we see no averages that surpass the slower yes-instance.

It is worth remembering that, by design, the experiment guarantees at least one solution, but we provide no guarantees that it is the only available solution, nor does the instance generator verify if other potential solutions abide by the properties of the guaranteed solution. These situations may have played a role in this data, as when we get more extensive paths without mandatory elements, we have a higher probability of accidentally creating other constructs that shortcut part of that long path. It is unlikely that this explanation could account for the whole effect, but it may have an effect. Another noteworthy point is that, as the solutions get bigger, up to 40% of all the vertices in this case, fewer elements are not in the solution, making it somewhat easier to solve the problem, as it is just a matter of removing a few unwanted vertices. It is easy to see that behavior on extremes; as a thought experiment, the reader may imagine a $25{,}000$ vertices graph with a solution containing $24{,}990$ of those, and it is easy to see how that would be easier to solve than a $25{,}000$ vertices graph where only $12{,}500$ are part of the solution.

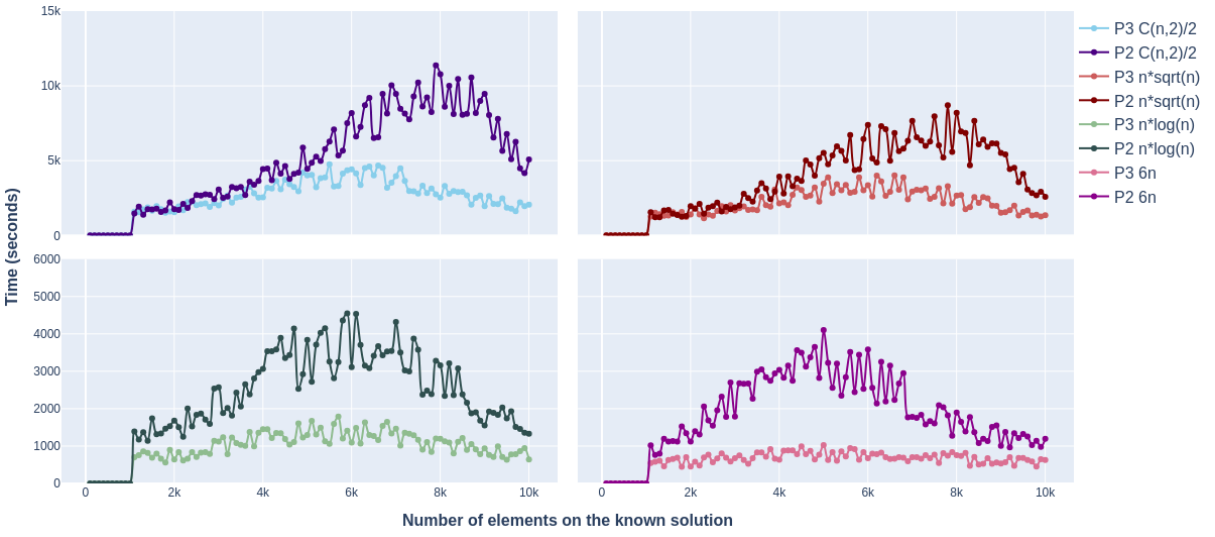


Figure 43 – Comparison of execution time versus of the size of the solution for **P2** and **P3** solving **d-star-20**

On another note, **d-star-20** was the only one where the profile of the chart significantly diverged between algorithms **P2** and **P3**. As stated before, the gap in execution times between the two approaches was always present, as well as fluctuations in their variances, with **P3** having the best showing in all experiments. However, the profile of the charts, *i.e.* the location of the peaks and valleys, the points where a ramp up or ramp down behavior would start, even the point where the execution time abruptly increased on **d-path** always had a high correlation and similar positions. This was not the case for **d-star-20**. As shown in Figure 43, **P2** had an accentuated ramp-up and ramp-down behavior, while **P3** had a very discreet variation. On the higher densities, both start fairly similar with a correlated ramping

up, but eventually **P3** plateaus and starts a very slow ramp-down, while **P2** is still going up, with a more aggressive ramp-down much later on the experiment. For the lower densities, the algorithms diverge from the start, with **P3** either staying plateaued or presenting a discreet parable, not even doubling the initial value, while **P2** has a very clear peak around midway through the experiment. Even when considering the subtle variations of **P3**, the locations where the ramp-up and ramp-down occur, the peak execution times, and the overall profiles do not seem to correlate between the algorithms. This difference in **d-star-20** could indicate that, for these characteristics, the intrinsic differences between both algorithms have an impact on the complexity of solving the problem or that we have stumbled on a difference in implementation that throttled or benefited one of the algorithms. The second hypothesis seems less likely as most of the parameters of the problem input have been constant, and we saw no other instances presenting this variation. Due to this disparity, there seems to be more at play than the natural complexity of the problem for this specific dataset.

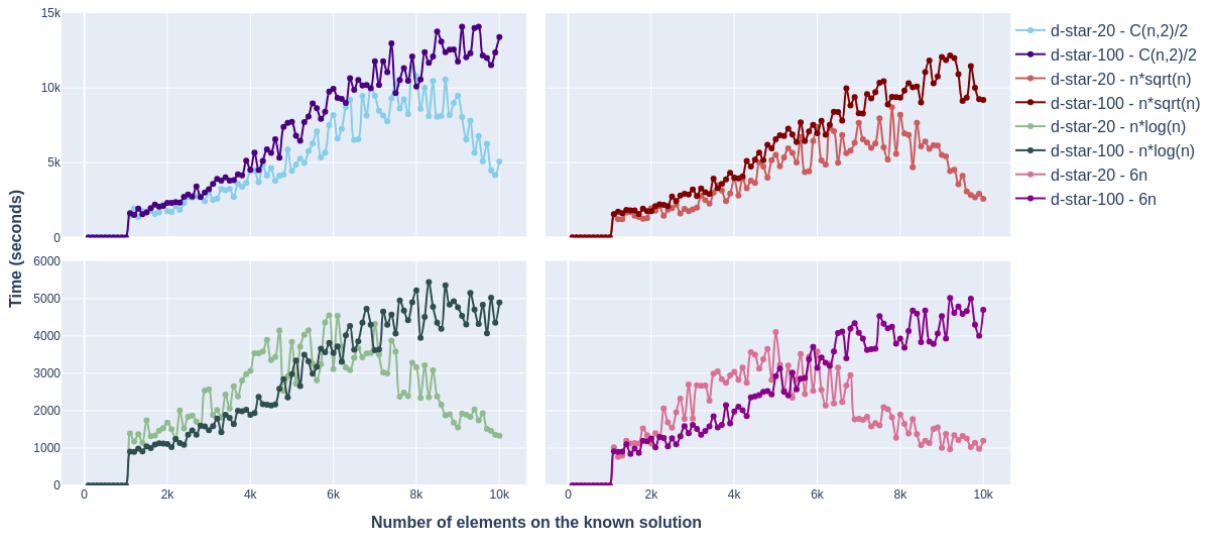


Figure 44 – Comparison of execution time versus of the size of the solution for **P2** solving **d-star-20** and **d-star-100**

Figure 44 shows the comparison between **P2** solving **d-star-20** versus **d-star-100**, on the higher densities **d-star-100** is constantly more time consuming to solve, however, by the last quarter of the experiment **d-path-20** shows a ramp-down while the other dataset does not. On the two lower densities, the behavior is similar; however, **d-star-20** starts with a higher execution time and stays like that until midway through the experiment, where it reaches its peak and starts to plateau and eventually ramps down, leading the other dataset to overtake as the slower option. This seems to indicate that **d-star-20** did indeed inherit properties of both the star and path datasets, with a high difficulty from the star but with an eventual ramp-down that, despite taking place later than the ones in **d-path**, is very visible, despite this not being something seen in the star experiments so far. This creates an interesting dynamic as the answer to whether the diameter of the solution is one of the main

drives of hardness becomes fuzzier, and the relationship between what percentage of the graph is part of the answer seems to influence that.

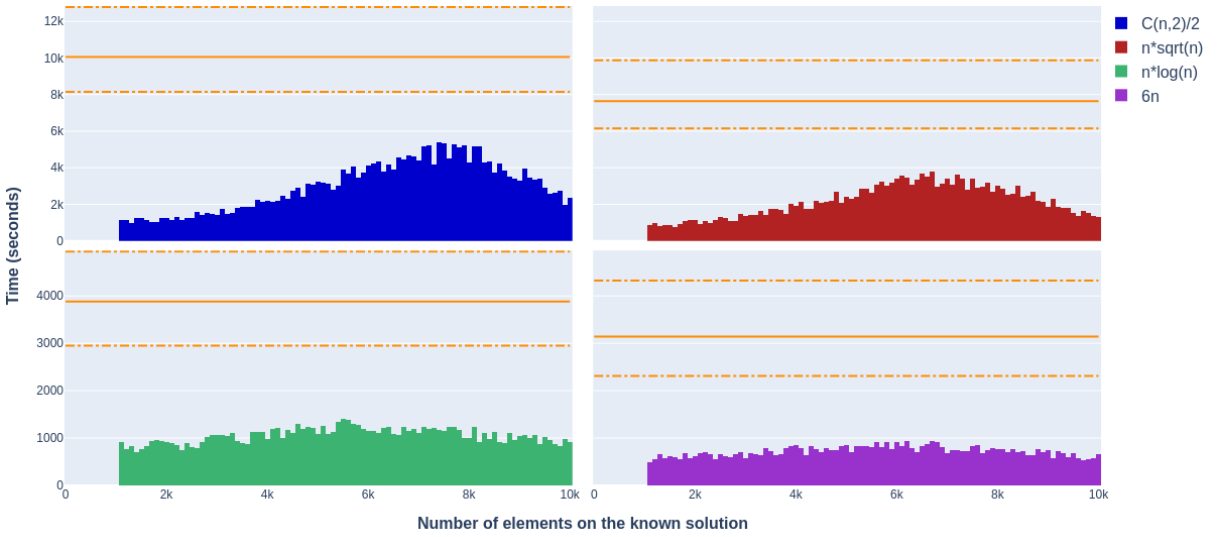


Figure 45 – Execution time versus of the size of the solution for **P3** solving **d-star-no-touch**

Finally, for **d-star-not touch**, where the stars' branches were each separated graphs connected only by the star root, we have the data shown in Figure 45. This experiment had the shortest execution times across the board, indicating that several smaller problems (the individual path solutions inside each isolated branch) are possibly easier to solve than a bigger equivalent problem. From a more practical point of view, it seems worth investing time to find possible linear complexity pre-processing strategies for the graphs if they could split the problem into smaller chunks or disconnected pieces. That note aside, the behavior seems to be closer to that of a hidden path solution than that of a hidden star solution in terms of execution time but closer to the star solution in terms of profile. For instance, the path experiments had a peak execution time close to the $5{,}000$ mark while this experiment had it near the $8{,}000$ mark. The same behavior is true for the second higher density; however, on the two sparser graphs, it is hard to determine where the peak execution time is.

When comparing the algorithms between each other on this last dataset, we see the same pattern as before, as seen in Figure 46. **P3** presented overall shorter execution times, especially near the peaks in execution times. However, it is interesting to note that on the higher density instances, the execution time starts very similar, and **P2** experiences a higher growth as the size of the solution increases until its peak, and eventually, it goes down rapidly, closing in on the execution time of **P3** once again. However, for the two smaller densities, the gap between the execution times seems to be constant over the entire experiment. In all cases **P2** also seems to have more variance in it's data points.

Figure 47 shows algorithm **P3** solving both **d-star-notouch** and **d-star-100**, this clearly shows that **d-star-notouch** is significantly easier to solve than the alternative by a large margin. Despite starting around the same execution time, they diverge quickly. It is

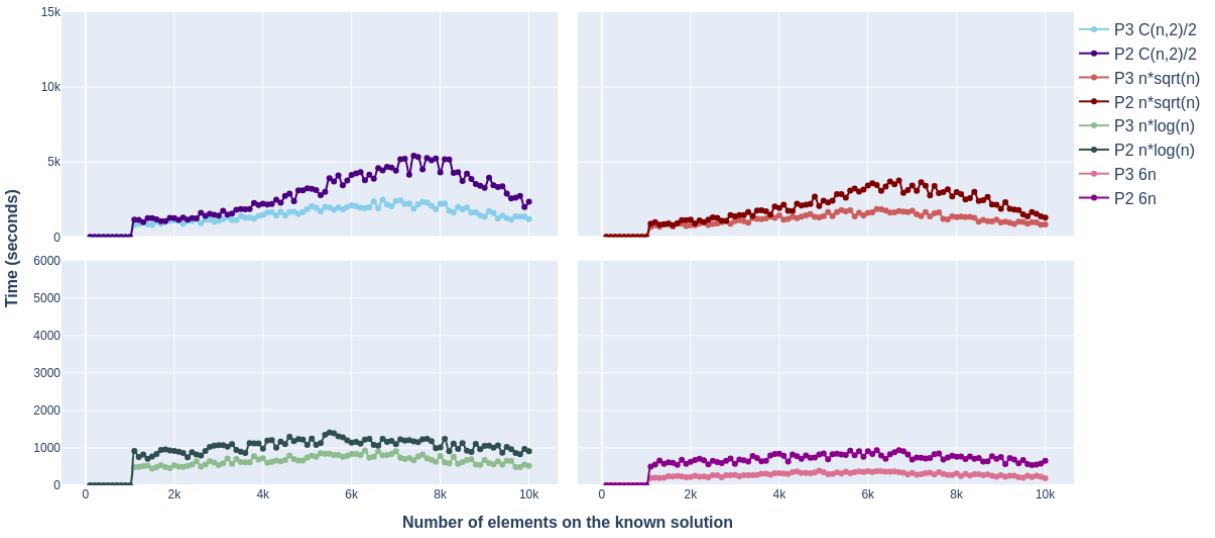


Figure 46 – Comparison of execution time versus of the size of the solution for **P2** and **P3** solving **d-star-not-touch**

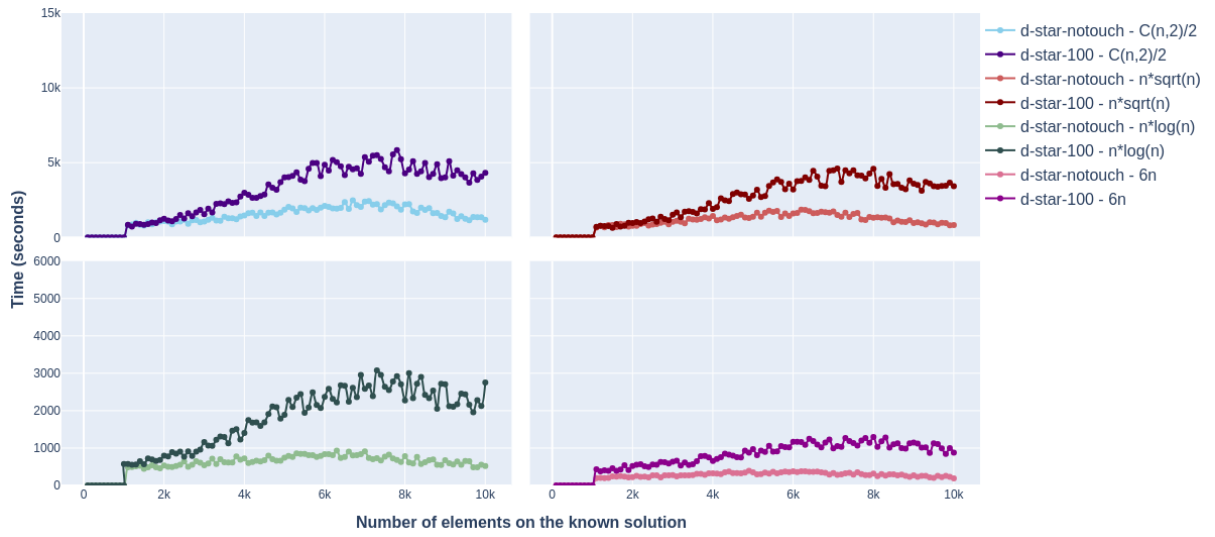


Figure 47 – Comparison of execution time versus of the size of the solution for **P3** solving **d-star-notouch** and **d-star-100**

interesting to note that the diversion takes place much sooner in the lower-density instances. This adds credibility to the suspicion that having multiple smaller, easily separable problems inside a bigger instance can help the algorithm to reach a conclusion much faster as the separation of search spaces quickly adds up as finding a solution in one of the separable problems makes it final as due to the lack of interference there will be no need to change it to conform with other possibilities to include extra mandatory vertices that were not solved yet.

This dataset is comparable to **d-path** as seen in Figure 48; the execution times are pretty similar, despite some alternate on the slower one as the dataset unfolds. In another instance where the behavior between **P2** and **P3** is pretty similar, but in different scales, Figure 49 shows the same data for algorithm **P2**, it is interesting to observe that the

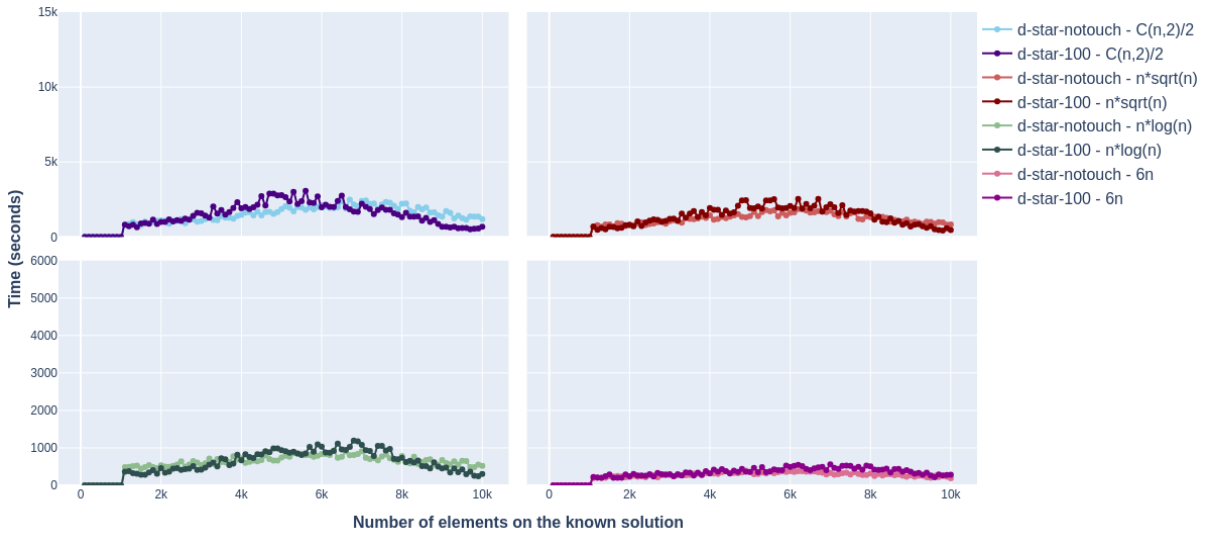


Figure 48 – Comparison of execution time versus of the size of the solution for **P3** solving **d-star-notouch** and **d-path**

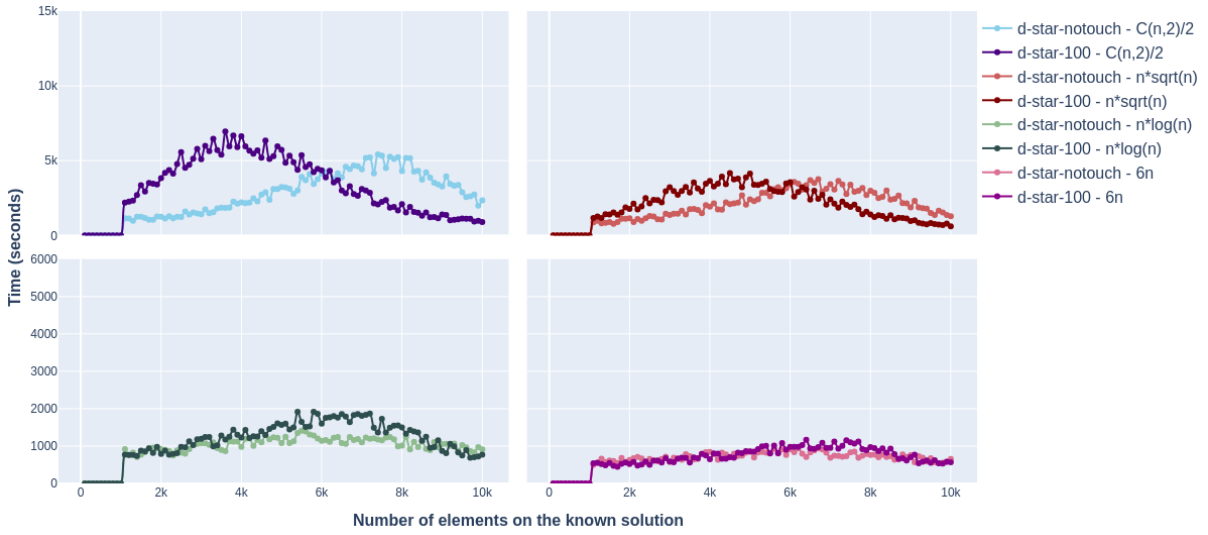


Figure 49 – Comparison of execution time versus of the size of the solution for **P2** solving **d-star-notouch** and **d-star-100**

two higher densities alternate the slower algorithm at around 60% of the experiment. Both datasets have clearly defined ramps and ramp-downs, but they are out of phase, with one taking place before the other. On the two lower densities, those behaviors are not as clear, and the data is more consulted as the situations where one is slower than the other are sparser, while for most of the experiment, there is no clearly defined gap between the results. These last figures also show that the execution profile of this dataset is closer to a path than to one of the previous stars, and despite being composed of multiple smaller path solutions, it is not clearly easier than a bigger path by itself.

5 Conclusion

In this thesis, we investigated the k-IN-A-TREE problem, a natural generalization of the well-known THREE-IN-A-TREE problem. The k-IN-A-TREE problem asks whether, given a graph and a set of k vertices, it is possible to find an induced subtree that contains all k vertices. While previous studies have focused on specific instances of this problem, such as claw-free graphs and cliques, our work addressed the problem in its broader, unbounded form.

Our initial approach to solving the k-IN-A-TREE problem involved using integer programming, with the problem modeled based on the classical Miller-Tucker-Zemlin (MTZ) formulation, commonly used in optimization problems such as the Traveling Salesman Problem (TSP). While this provided a solid foundation, we also explored two alternative approaches: one constructive and one destructive. The constructive approach enforced connectivity constraints, ensuring that all k vertices were part of a connected subtree, while the destructive approach enforced acyclicity, ensuring that no cycles were present in the solution. Despite the potentially exponential number of constraints required to maintain connectivity and acyclicity in these latter models, both approaches outperformed the MTZ-based model in practice. Demonstrating the advantage of directly enforcing problem-specific constraints as they are needed.

These models were thoroughly tested on reasonably large instances, demonstrating their practical effectiveness in solving the k-IN-A-TREE problem for graph sizes that would be intractable for similar combinatorial optimization problems, such as the Traveling Salesman Problem (TSP). Our computational experiments showed that the proposed models could solve instances with thousands of vertices and edges, even for dense graphs. We successfully tackled instances that far exceed the scale of real-world networks, such as the transportation networks of entire countries. Precisely, we handled graphs several times bigger and denser than all the cities and roads in the United States or Brazil. However, it is important to note that we did not directly compare the performance of our models to other algorithms, nor did we perform statistical inference to determine if one approach is superior. This paper's intent is exploratory, focusing on the problem itself rather than on comparing the effectiveness of different algorithms.

This discovery represents one of the key contributions of this research, as it shows that our integer programming formulations offer a feasible and scalable approach for practically solving the k-IN-A-TREE. Remarkably, this feat had not been attempted before in the literature, as much of the prior research was primarily focused on theoretical approaches applied to graphs with specific properties, such as claw-free graphs or cliques, or on theoretical

complexity results. By shifting the focus to general, large, varied graphs and demonstrating its practical solvability, even in unknown initial graph conditions, offering new avenues for research and real-world problem-solving.

Building on the success of the two most performant formulations, we took the research one step further by using these models to study the properties that influence the hardness of the problem. To achieve this, we conducted an extensive series of computational experiments that took over two years of computational time, and condensed all the results into a digestible analysis. These experiments were designed to uncover how various factors, beyond simple instance size or the value of k, contribute to the hardness of solving the problem. Our findings revealed that the hardness is not linearly nor solely correlated with the size of the instance nor with the number of vertices k that must be included in the subtree.

Instead, the complexity of the problem arises from a combination of factors that interact in intricate ways. The presence or absence of a solution is a crucial determinant of difficulty, but so are properties such as graph density, vertex connectivity, and the overall structure of the graph. For instance, denser graphs often result in more potential candidate subtrees, increasing the computational overhead required to identify or rule out solutions. However, even sparse graphs can pose significant challenges depending on how vertices and edges are distributed. In certain situations, increasing the number of vertices k that must be included in the subtree can paradoxically decrease the problem's execution time. This counterintuitive finding challenges the assumption that larger k values generally result in more complex computations. Still, the additional vertices may act as natural constraints on potential solutions, streamlining the search process rather than complicating it.

On the specific topic of the size of k, the hardness appears to exhibit threshold behavior, with computational times increasing exponentially as the number of mandatory vertices rises to a certain point. After reaching this threshold, the difficulty decreases at a slower rate until it eventually stabilizes, reaching a computational plateau. The exact number of mandatory vertices required to hit this threshold varies depending on the instance's size and the graph's edge density. Larger and denser graphs tend to reach the threshold with more mandatory vertices, while sparser or smaller graphs encounter the threshold earlier. This threshold-based behavior is crucial in understanding the problem's computational complexity and optimizing solution strategies accordingly.

This threshold behavior is not an absolute rule but rather an apparent latent trend we observed across many instances. While it seems to be a typical pattern in the instances tested, there are cases where this exponential rise followed by a slower decline and plateau either does not manifest or is less pronounced. In some cases, the data we collected may not have been sufficient to conclusively demonstrate the existence of this behavior, suggesting that additional factors or complexities might be at play. Nevertheless, this threshold pattern provides a valuable framework for understanding the computational challenges of

the problem, even if it is not universally applicable across all graphs.

The interplay of these variables makes it clear that the problem's computational hardness cannot be fully understood through simple measures like instance size or k alone. The interplay of these variables clearly shows that the computational hardness of the k-IN-A-TREE problem cannot be adequately captured by simple measures like instance size or the value of k alone. Our findings emphasize the need for a more nuanced analysis of the problem's underlying structural properties. Additionally, our demonstration that the problem can be solved without making prior assumptions about the input graph suggests that, in practical scenarios, it may be more efficient to attempt solving the unrestricted version of the problem first. This approach could be preferable to optimizing for specific cases, especially when a more sophisticated analysis is too time-consuming or impractical for real-world applications requiring rapid solutions.

Based on the performance of our models when tested on real-world graphs—often used to benchmark other complex problems—and on standard benchmark instances for similar optimization problems, we are confident that our exploration reached instances that are not only representative of real-world challenges but also potentially harder than the types of graphs typically encountered in practical applications. The proposed formulations proved to be usable and robust when compared to what is available in the literature; being able to solve instances of size superior to twenty-five thousand vertices in less than ten minutes, many real-world instances are covered within this range. Given more time and robust enough hardware, this formulation can solve even more complex instances. The ability of our formulations to handle such demanding cases indicates that the models we developed are robust and capable of addressing high-complexity scenarios, making our solutions both theoretically comprehensive and practically relevant for large-scale applications.

During the course of this research, several parts of the work were published in academic journals. Specifically, portions of the research were published in the *Simpósio Brasileiro de Pesquisa Operacional* (FERREIRA; SANTOS; WANNER, 2018), highlighting our advancements in modeling and solving the unrestricted problem using MTZ. While the publication in *International Transactions in Operational Research* (FERREIRA; SANTOS; VALLE, 2023) focused on comparing the three modeling techniques and the initial findings on the variables that influence hardness. Additionally, further developments and results have been recently submitted to other traditional journals.

5.1 Future Work

After analyzing the results obtained, many paths for the exploration of this problem have been enlightened. This paths can build up on the work still to be done in this thesis. This section will enumerate a few of them non-exhaustively.

A promising direction for future work lies in mathematically formalizing the observed threshold behavior of the k-IN-A-TREE problem. Developing a deeper theoretical understanding of this pattern and rigorously proving its existence and conditions would provide valuable insights into the problem's complexity. Additionally, enhancing the scope of comparisons in our experiments to include a broader range of variables — such as the behavior of LP bounds, the number of nodes explored during solution, and other computational metrics — would offer a more comprehensive evaluation of the models' performance. By incorporating these variables, we can better understand how they influence the problem's difficulty and solution efficiency, leading to more informed strategies for optimization and problem-solving in both theoretical and practical contexts.

Another opportunity for future work involves developing and establishing a comprehensive benchmark of instances designed explicitly for the k-IN-A-TREE problem. By analyzing and recreating particular cases that either increase or decrease the difficulty of individual instances, we can create a more targeted set of benchmarks that reflect the wide range of challenges the problem poses. This would provide a standard framework for evaluating solution approaches and help identify critical factors that contribute to instance complexity. Making this benchmark publicly available would further benefit the research community, enabling more consistent comparisons between different methodologies and fostering the development of more efficient algorithms and models for tackling the problem.

5.2 Final Considerations

The authors would like to thank professor Cristiano Arbex Valle from DCC/UFMG for making his personal optimization framework available for use in this project.

Appreciation is also due to CNPq, CAPES, FAPEMIG, CEFET-MG and UFMG for their support.

Bibliography

ALLENDER, E. et al. On the complexity of numerical analysis. **SIAM Journal on Computing**, SIAM, v. 38, n. 5, p. 1987–2006, 2009. Citado na página 15.

APPLEGATE, D. L. et al. **The traveling salesman problem: a computational study**. [S.l.]: Princeton university press, 2006. Citado na página 15.

BEKTAŞ, T.; GOUVEIA, L. Requiem for the Miller–Tucker–Zemlin subtour elimination constraints? **European Journal of Operational Research**, Elsevier, v. 236, n. 3, p. 820–832, 2014. Citado na página 22.

BIENSTOCK, D. On the complexity of testing for odd holes and induced odd paths. **Discrete Mathematics**, Elsevier, v. 90, n. 1, p. 85–92, 1991. Citado 2 vezes nas páginas 9 e 13.

BOLLOBÁS, B. **Random graphs**. [S.l.]: Springer, 1998. 215–252 p. Citado na página 33.

BONDY, J. A.; MURTY, U. S. R. et al. **Graph theory with applications**. [S.l.]: Macmillan London, 1976. v. 290. Citado na página 16.

BRUNETTA, L.; MAFFIOLI, F.; TRUBIAN, M. Solving the feedback vertex set problem on undirected graphs. **Discrete Applied Mathematics**, Elsevier, v. 101, n. 1-3, p. 37–51, 2000. Citado na página 17.

CHANG, H.-C.; LU, H.-I. A faster algorithm to recognize even-hole-free graphs. **Journal of Combinatorial Theory, Series B**, Elsevier, v. 113, p. 141–161, 2015. Citado 2 vezes nas páginas 1 e 12.

CHEN, J. et al. Improved algorithms for the feedback vertex set problems. In: SPRINGER. **Workshop on Algorithms and Data Structures**. [S.l.], 2007. p. 422–433. Citado na página 18.

CHUDNOVSKY*, M. et al. Recognizing berge graphs. **Combinatorica**, Springer, v. 25, n. 2, p. 143–186, 2005. Citado na página 12.

CHUDNOVSKY, M.; KAPADIA, R. Detecting a theta or a prism. **SIAM Journal on Discrete Mathematics**, SIAM, v. 22, n. 3, p. 1164–1186, 2008. Citado na página 11.

CHUDNOVSKY, M.; SEYMOUR, P. The three-in-a-tree problem. **Combinatorica**, Springer, v. 30, n. 4, p. 387–417, 2010. Citado 3 vezes nas páginas 1, 11 e 12.

CRESCENZI, P.; KANN, V.; HALLDÓRSSON, M. **A compendium of NP optimization problems**. 1995. Citado 2 vezes nas páginas 13 e 17.

DANTZIG, G. **Linear programming and extensions**. [S.l.]: Princeton university press, 2016. Citado na página 15.

DANTZIG, G.; FULKERSON, R.; JOHNSON, S. Solution of a large-scale traveling-salesman problem. **Journal of the operations research society of America**, INFORMS, v. 2, n. 4, p. 393–410, 1954. Citado na página 15.

DERHY, N.; PICOULEAU, C. Finding induced trees. **Discrete Applied Mathematics**, Elsevier, v. 157, n. 17, p. 3552–3557, 2009. Citado 4 vezes nas páginas 1, 12, 13 e 33.

DESROCHERS, M.; LAPORTE, G. Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints. **Operations Research Letters**, Elsevier, v. 10, n. 1, p. 27–36, 1991. Citado na página 22.

DIESTEL, R. Graph theory. 2005. **Grad. Texts in Math**, v. 101, 2005. Citado na página 8.

DIETZ, A.; ADAMS, W.; YANG, B. Tightened Miller-Tucker-Zemlin inequalities for the target visitation and related problems. 2019. Citado na página 22.

DU, D.-Z.; SMITH, J.; RUBINSTEIN, J. H. **Advances in Steiner trees**. [S.l.]: Springer Science & Business Media, 2000. v. 6. Citado na página 17.

EDMONDS, J.; KARP, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 19, n. 2, p. 248–264, 1972. Citado na página 26.

ERDŐS, P.; RÉNYI, A. et al. On the evolution of random graphs. **Publ. math. inst. hung. acad. sci**, v. 5, n. 1, p. 17–60, 1960. Citado na página 31.

EVEN, S. **Graph algorithms**. [S.l.]: Cambridge University Press, 2011. Citado 2 vezes nas páginas 7 e 9.

FERREIRA, L. S.; SANTOS, V. F. d.; VALLE, C. A. Integer programming formulations for the k-in-a-tree problem in graphs. **International Transactions in Operational Research**, Wiley Online Library, v. 31, p. 2090–3107, 2023. Citado na página 77.

FERREIRA, L. S.; SANTOS, V. Fernandes dos; WANNER, E. F. Uma abordagem exata para o problema k-in-a-tree utilizando programação linear inteira. In: **Anais do Simpósio Brasileiro de Pesquisa Operacional**. [S.l.]: Galoa, 2018. (SBPO 2018). Citado na página 77.

FESTA, P.; PARDALOS, P. M.; RESENDE, M. G. Feedback set problems. In: **Handbook of Combinatorial Optimization: Supplement Volume A**. [S.l.]: Springer, 1999. p. 209–258. Citado na página 17.

FIALA, J. et al. The k-in-a-path problem for claw-free graphs. **Algorithmica**, Springer, v. 62, n. 1-2, p. 499–519, 2012. Citado na página 13.

FORD, L. R.; FULKERSON, D. R. Maximal flow through a network. **Canadian journal of Mathematics**, Cambridge University Press, v. 8, p. 399–404, 1956. Citado 2 vezes nas páginas 26 e 27.

GILBERT, E. N.; POLLAK, H. O. Steiner minimal trees. **SIAM Journal on Applied Mathematics**, SIAM, v. 16, n. 1, p. 1–29, 1968. Citado na página 16.

GOMES, G. et al. FPT and kernelization algorithms for the k-in-a-tree problem. **arXiv preprint arXiv:2007.04468**, 2020. Citado na página 13.

GROSS, J. L.; YELLEN, J. **Handbook of graph theory**. [S.l.]: CRC press, 2004. Citado 2 vezes nas páginas 6 e 8.

HAAS, R.; HOFFMANN, M. Chordless paths through three vertices. **Theoretical Computer Science**, Elsevier, v. 351, n. 3, p. 360–371, 2006. Citado na página 13.

HASSIN, R.; RUBINSTEIN, S. Better approximations for max tsp. **Information Processing Letters**, Elsevier, v. 75, n. 4, p. 181–186, 2000. Citado na página 15.

HEINEMAN, G. T.; POLLICE, G.; SELKOW, S. **Algorithms in a nutshell: a practical guide**. [S.l.]: " O'Reilly Media, Inc.", 2016. Citado na página 27.

HWANG, F. K. On steiner minimal trees with rectilinear distance. **SIAM journal on Applied Mathematics**, SIAM, v. 30, n. 1, p. 104–114, 1976. Citado na página 17.

IVANOV, A. O.; TUZHILIN, A. A. **Minimal NetworksThe Steiner Problem and Its Generalizations**. [S.l.]: CRC press, 1994. Citado na página 17.

JANSON, S.; RUCINSKI, A.; LUCZAK, T. **Random graphs**. [S.l.]: John Wiley & Sons, 2011. Citado na página 33.

KARA, I.; LAPORTE, G.; BEKTAS, T. A note on the lifted Miller–Tucker–Zemlin subtour elimination constraints for the capacitated vehicle routing problem. **European Journal of Operational Research**, Elsevier, v. 158, n. 3, p. 793–795, 2004. Citado na página 22.

KOCH, T.; MARTIN, A.; VOSS, S. Steinlib: An updated library on steiner tree problems in graphs. In: **Steiner trees in industry**. [S.l.]: Springer, 2001. p. 285–325. Citado na página 30.

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. Citado na página 16.

LAI, K.-Y.; LU, H.-I.; THORUP, M. Three-in-a-tree in near linear time. In: **Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing**. [S.l.: s.n.], 2020. p. 1279–1292. Citado na página 11.

LANGEVIN, A.; SOUMIS, F.; DESROSIERS, J. Classification of travelling salesman problem formulations. **Operations Research Letters**, Elsevier, v. 9, n. 2, p. 127–132, 1990. Citado na página 22.

LAPORTE, G. The traveling salesman problem: An overview of exact and approximate algorithms. **European Journal of Operational Research**, Elsevier, v. 59, n. 2, p. 231–247, 1992. Citado na página 22.

LÉVÊQUE, B. et al. Detecting induced subgraphs. **Discrete Applied Mathematics**, Elsevier, v. 157, n. 17, p. 3540–3551, 2009. Citado 2 vezes nas páginas 1 e 12.

LIU, W.; TROTIGNON, N. The k-in-a-tree problem for graphs of girth at least k. **Discrete Applied Mathematics**, Elsevier, v. 158, n. 15, p. 1644–1649, 2010. Citado na página 13.

MELO, R. A.; QUEIROZ, M. F.; RIBEIRO, C. C. Compact formulations and an iterated local search-based matheuristic for the minimum weighted feedback vertex set problem. **European Journal of Operational Research**, Elsevier, v. 289, n. 1, p. 75–92, 2021. Citado na página 18.

MELO, R. A.; RIBEIRO, C. C. Maximum weighted induced forests and trees: new formulations and a computational comparative review. **International Transactions in Operational Research**, Wiley Online Library, v. 29, n. 4, p. 2263–2287, 2022. Citado na página 19.

MILLER, C. E.; TUCKER, A. W.; ZEMLIN, R. A. Integer programming formulation of traveling salesman problems. **Journal of the ACM (JACM)**, ACM, v. 7, n. 4, p. 326–329, 1960. Citado 2 vezes nas páginas 15 e 20.

PADBERG, M.; RINALDI, G. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. **Operations research letters**, Elsevier, v. 6, n. 1, p. 1–7, 1987. Citado na página 23.

PADBERG, M.; RINALDI, G. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. **SIAM review**, SIAM, v. 33, n. 1, p. 60–100, 1991. Citado na página 23.

PADBERG, M.; SUNG, T.-Y. An analytical comparison of different formulations of the travelling salesman problem. **Mathematical Programming**, Springer, v. 52, n. 1, p. 315–357, 1991. Citado na página 22.

PAOLINI, E.; STEPANOV, E. Existence and regularity results for the steiner problem. **Calculus of Variations and Partial Differential Equations**, Springer, v. 46, n. 3-4, p. 837–860, 2013. Citado na página 16.

PRIM, R. C. Shortest connection networks and some generalizations. **The Bell System Technical Journal**, Nokia Bell Labs, v. 36, n. 6, p. 1389–1401, 1957. Citado na página 16.

RAZGON, I. Exact computation of maximum induced forest. In: SPRINGER. **Algorithm Theory–SWAT 2006: 10th Scandinavian Workshop on Algorithm Theory, Riga, Latvia, July 6-8, 2006. Proceedings 10**. [S.l.], 2006. p. 160–171. Citado na página 18.

SANTOS, V. F. dos; SILVA, M. V. da; SZWARCFITER, J. L. The k-in-a-tree problem for chordal graphs. **Matemática Contemporânea**, v. 44, p. 1–10, 2015. Citado na página 13.

SHERALI, H. D.; DRISCOLL, P. J. On tightening the relaxations of Miller-Tucker-Zemlin formulations for asymmetric traveling salesman problems. **Operations Research**, INFORMS, v. 50, n. 4, p. 656–669, 2002. Citado na página 22.

TRICK, M. Graph coloring instances. **Michael Trick Operations Research Page), http://mat. gsia. cmu. edu/COLOR/instances. html (Accessed: 8 April 2024)**, 2008. Citado na página 30.

WARME, D. M.; WINTER, P.; ZACHARIASEN, M. Exact algorithms for plane steiner tree problems: A computational study. In: **Advances in Steiner trees**. [S.l.]: Springer, 2000. p. 81–116. Citado na página 17.

WEST, D. B. et al. **Introduction to graph theory**. [S.l.]: Prentice hall Upper Saddle River, NJ, 1996. v. 2. Citado na página 8.

ZHANG, P.; CHARTRAND, G. Introduction to graph theory. **Tata McGraw-Hill**, v. 2, p. 2–1, 2006. Citado na página 6.

Appendix

APPENDIX A – All experiment charts

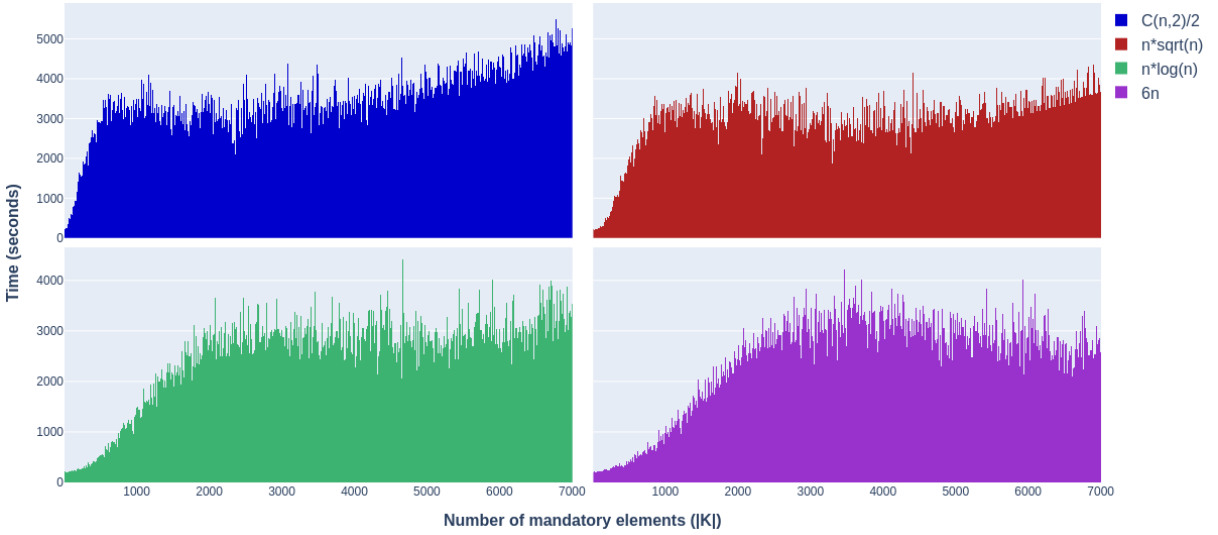


Figure 50 – Execution time versus size of K for **P2** using instances with $n = 25{,}000$, and decreasing edge density of $\binom{n}{2}/2$, $n\sqrt{n}$, $n \log n$ and $6n$ from left-to-right top-to-bottom

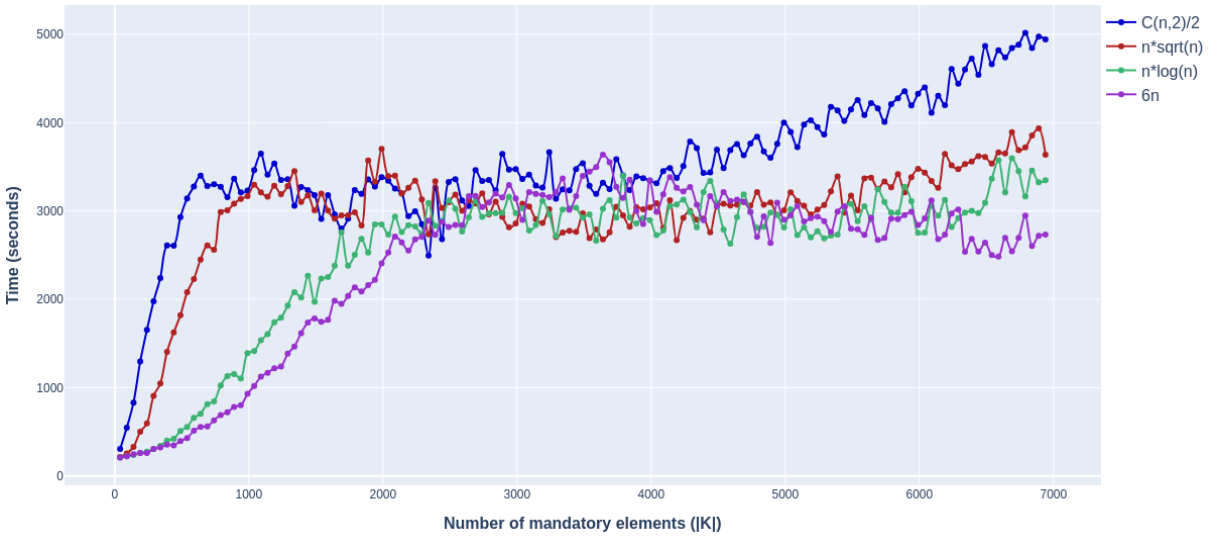


Figure 51 – Execution time versus size of K for **P2** using instances with $n = 25{,}000$

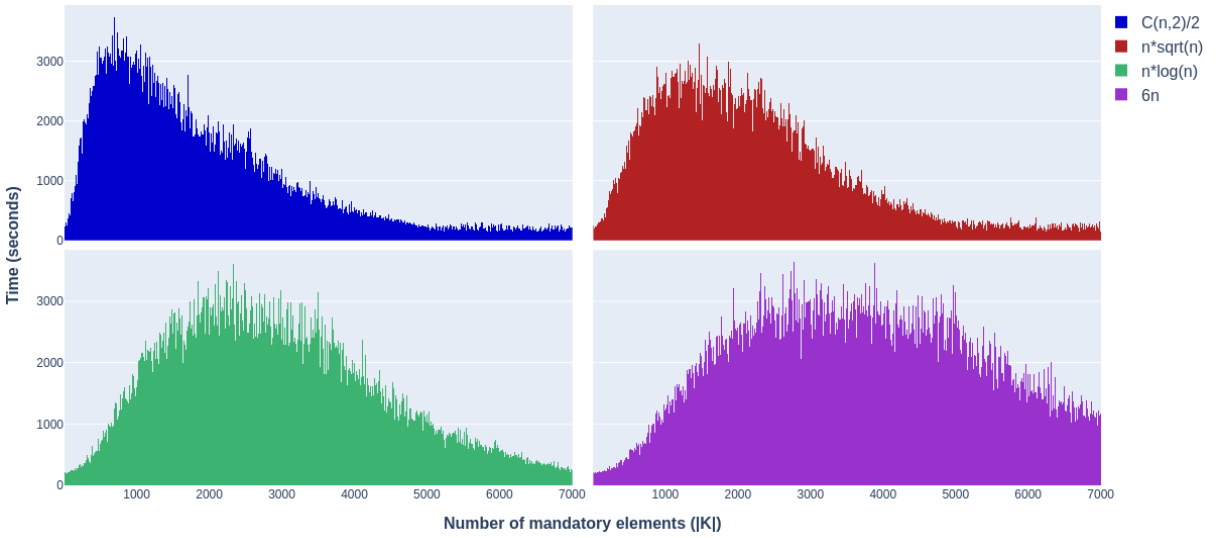


Figure 52 – Execution time versus size of K for **P3** using instances with $n = 25{,}000$, and decreasing edge density of $\binom{n}{2}/2$, $n\sqrt{n}$, $n \log n$ and $6n$ from left-to-right top-to-bottom

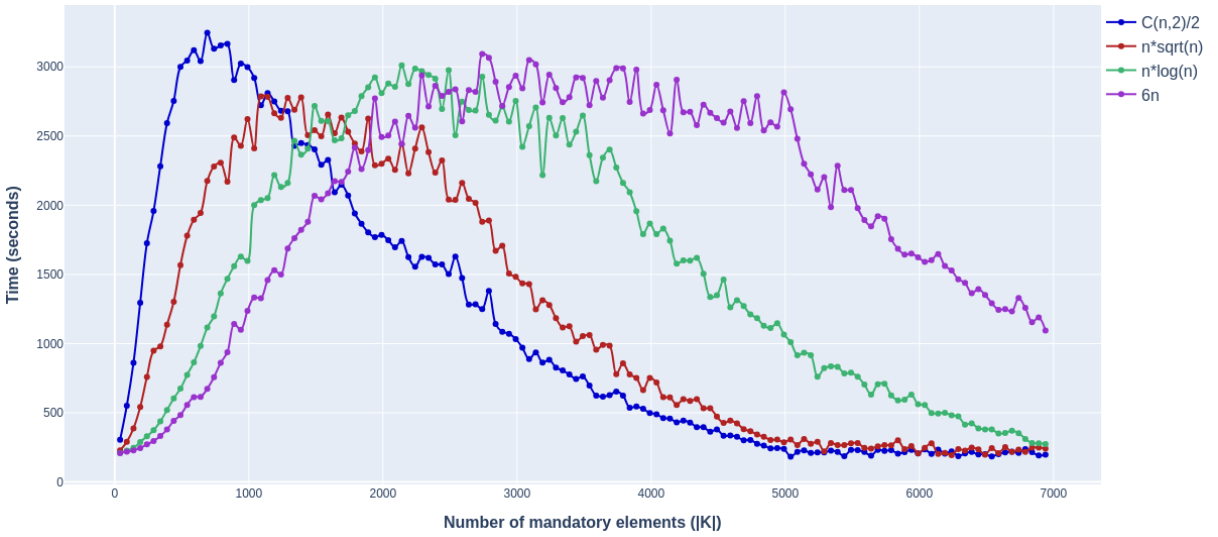


Figure 53 – Execution time versus size of K for **P3** using instances with $n = 25{,}000$

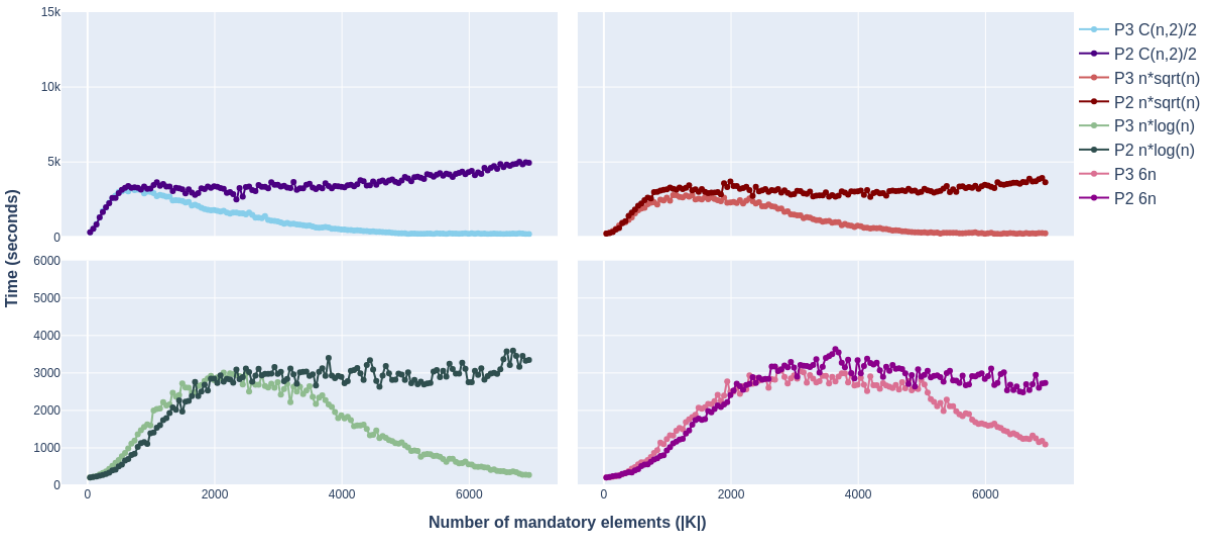


Figure 54 – Comparison of execution time versus size of K between **P2** and **P3** using instances with $n = 25{,}000$, and decreasing edge density of $\binom{n}{2}/2$, $n\sqrt{n}$, $n \log n$ and $6n$ from left-to-right top-to-bottom

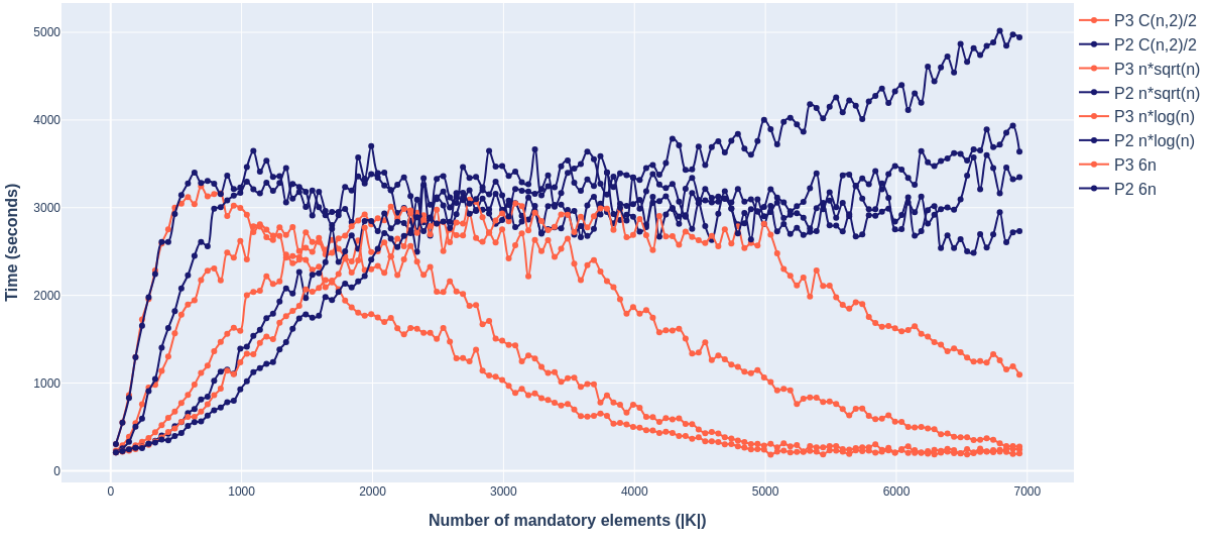


Figure 55 – Comparison of execution time versus size of K between **P2** and **P3** using instances with $n = 25{,}000$

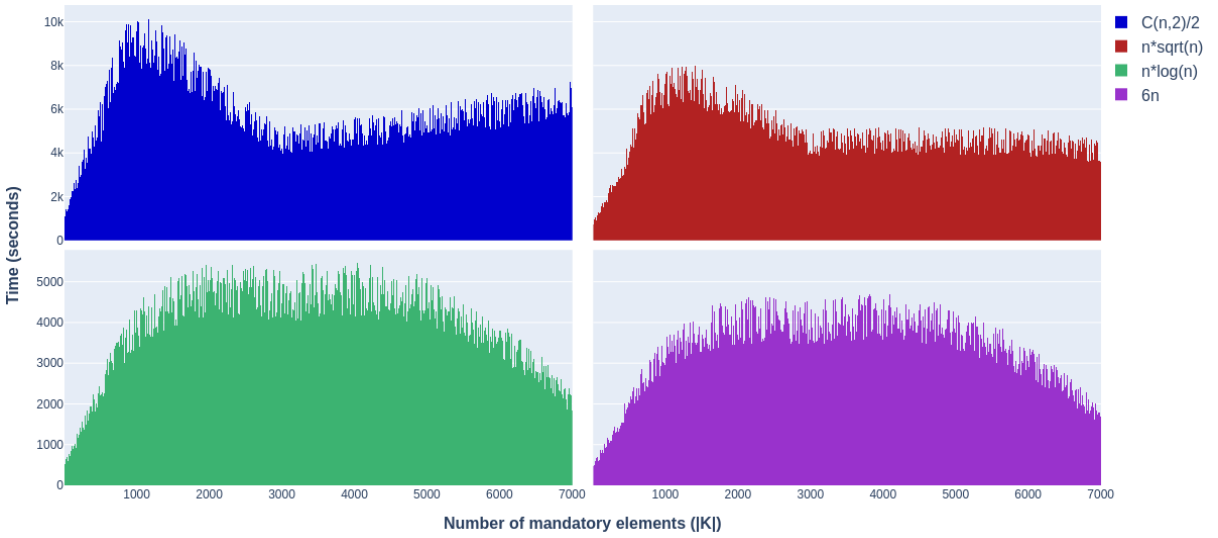


Figure 56 – Execution time versus size of K for **P2** using yes-instances with $n = 25{,}000$, and decreasing edge density of $\binom{n}{2}/2$, $n\sqrt{n}$, $n \log n$ and $6n$ from left-to-right top-to-bottom

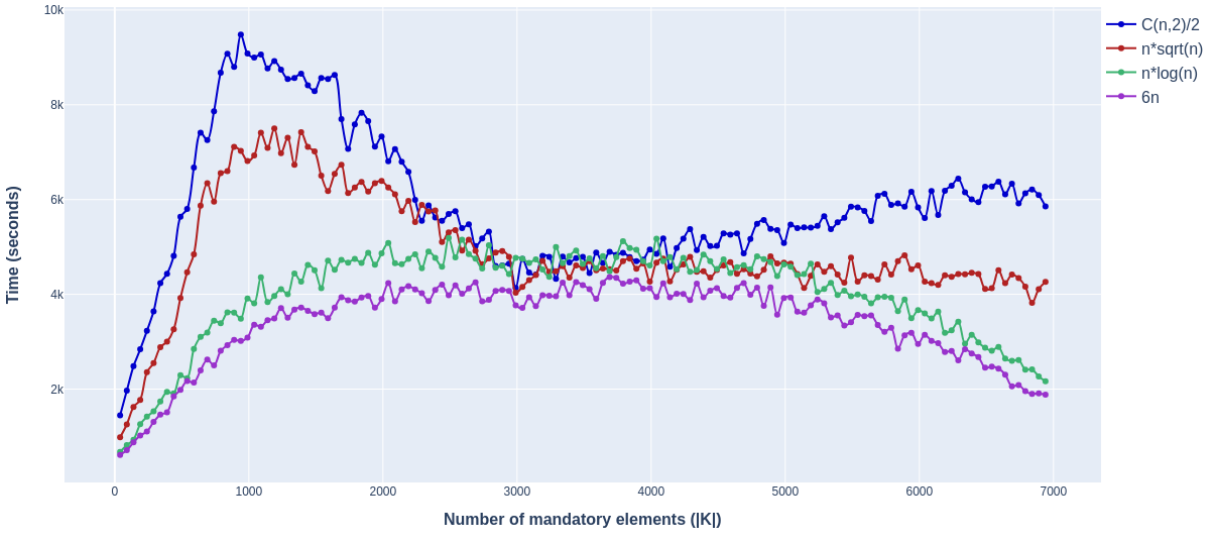


Figure 57 – Execution time versus size of K for **P2** using yes-instances with $n = 25{,}000$

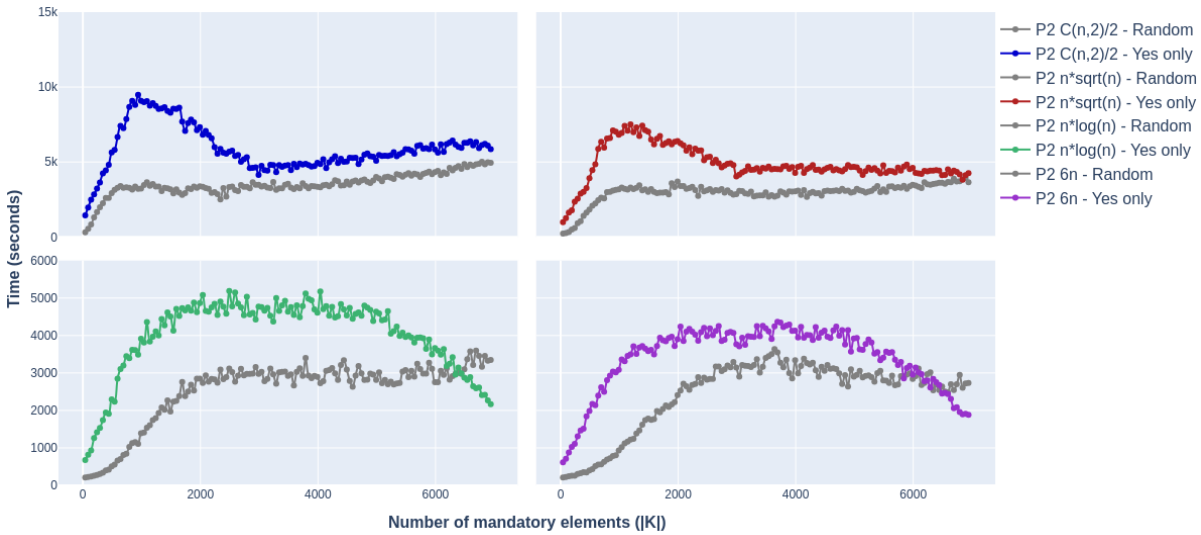


Figure 58 – Comparison of execution time versus size of K using **P2** for both random and yes-only instances with $n = 25{,}000$

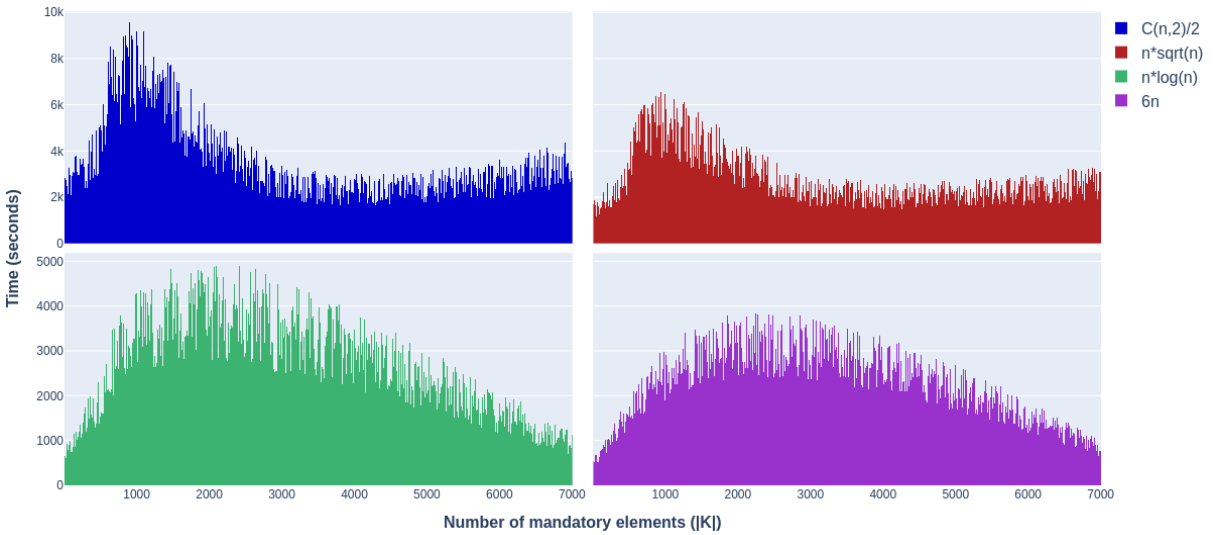


Figure 59 – Execution time versus size of K for **P2** using yes-instances with $n = 25{,}000$, and decreasing edge density of $\binom{n}{2}/2$, $n\sqrt{n}$, $n \log n$ and $6n$ from left-to-right top-to-bottom

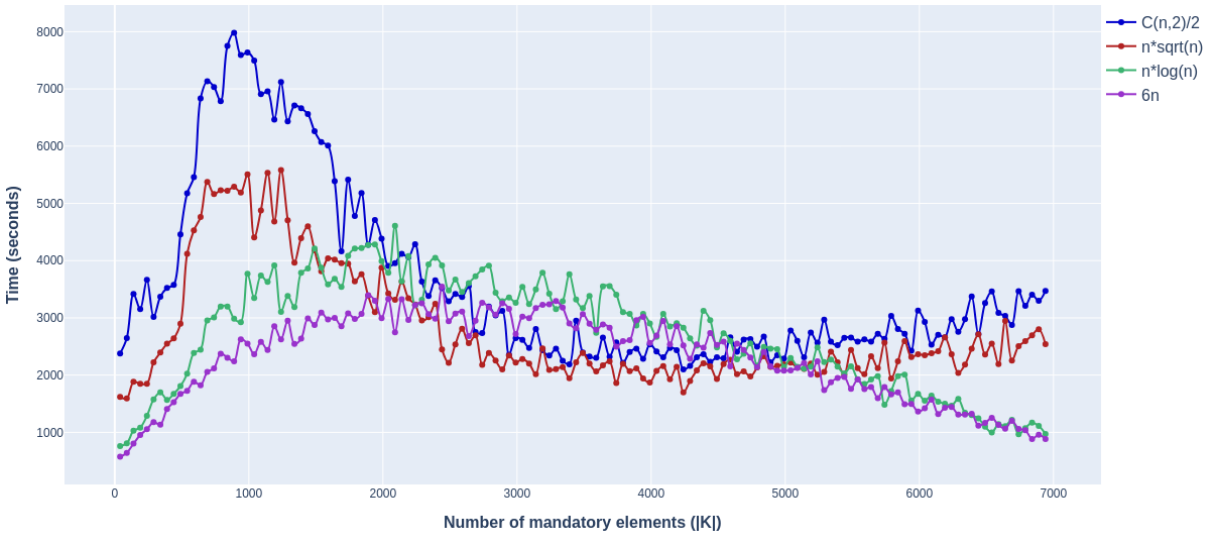


Figure 60 – Execution time versus size of K for **P2** using yes-instances with $n = 25{,}000$

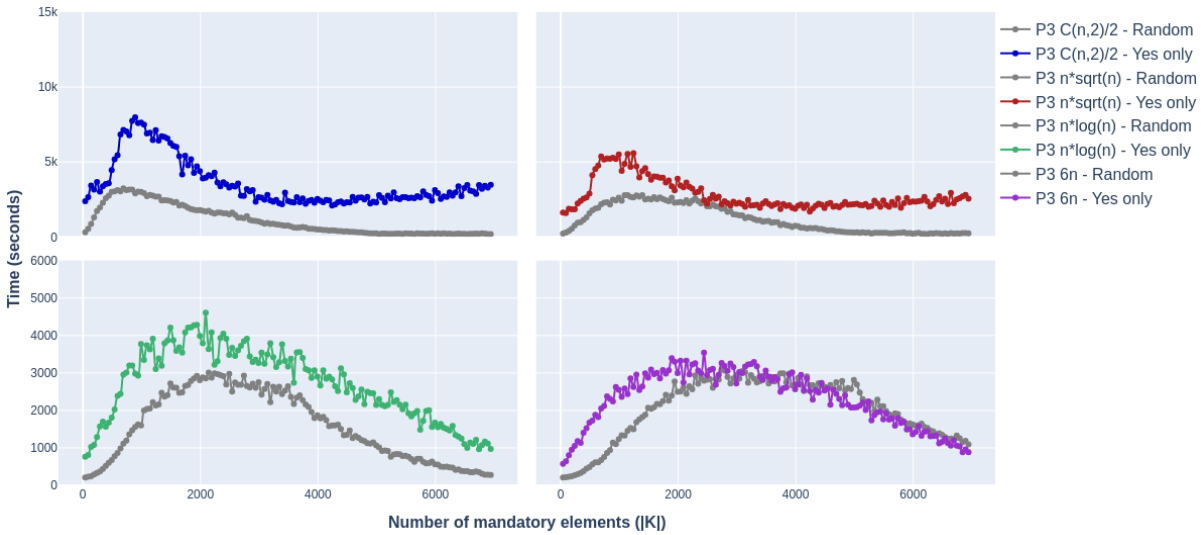


Figure 61 – Comparison of execution time versus the size of K using **P2** for both random and yes-only instances with $n = 25{,}000$

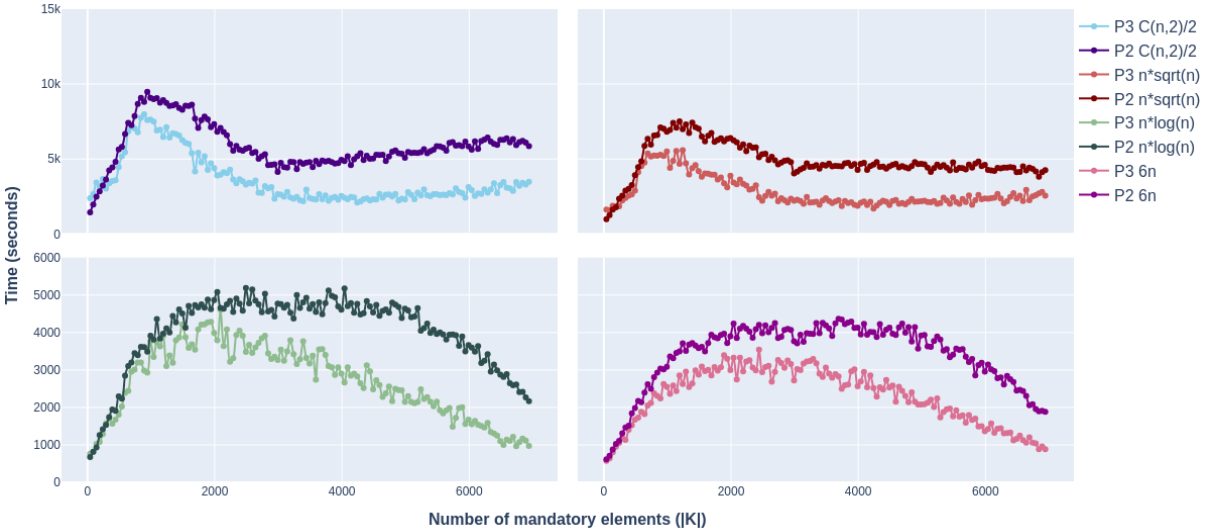


Figure 62 – Comparison of the execution time versus size of K between **P2** and **P3** using yes-instances with $n = 25{,}000$

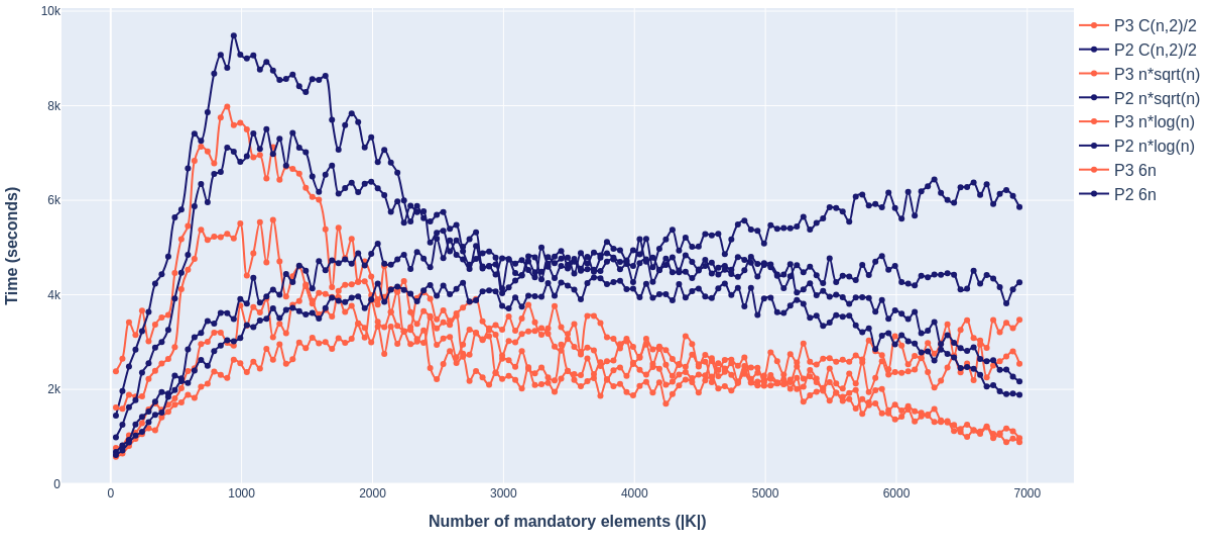


Figure 63 – Comparison of the execution time versus size of K between **P2** and **P3** using yes-instances with $n = 25{,}000$, and decreasing edge density of $\binom{n}{2}/2$, $n\sqrt{n}$, $n \log n$ and $6n$ from left-to-right top-to-bottom

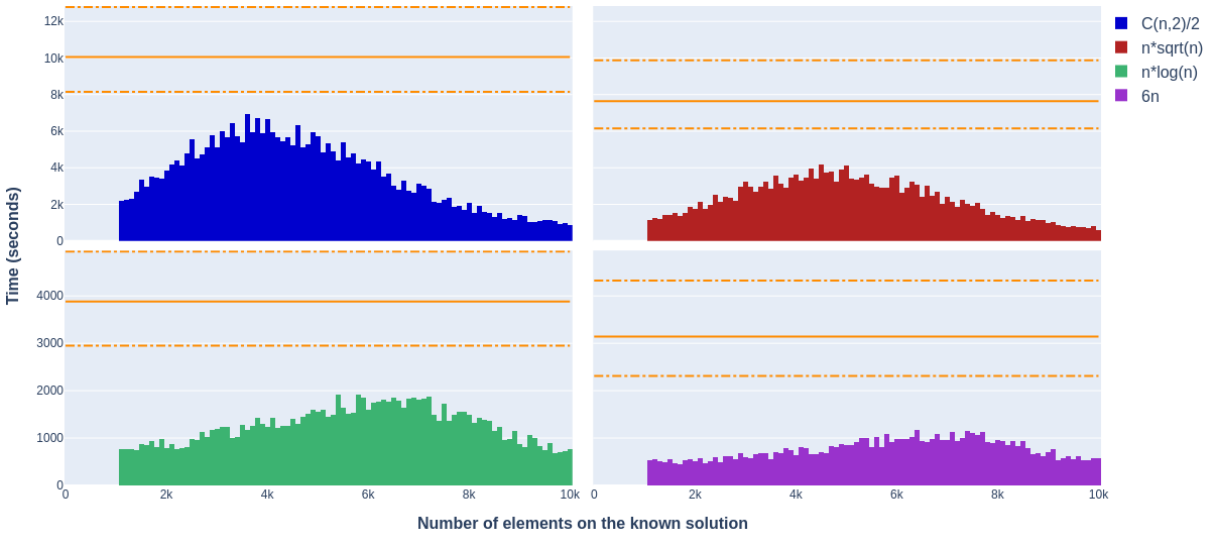


Figure 64 – Execution time versus size of the solution for **P2** solving **d-path**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

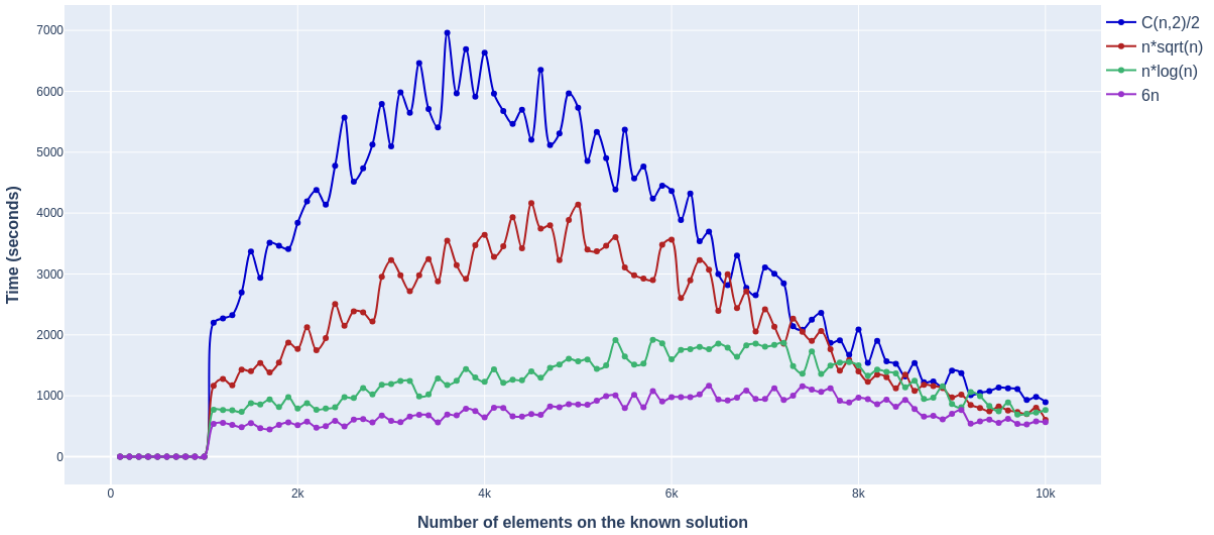


Figure 65 – Execution time versus size of the solution for **P2** solving **d-path**.

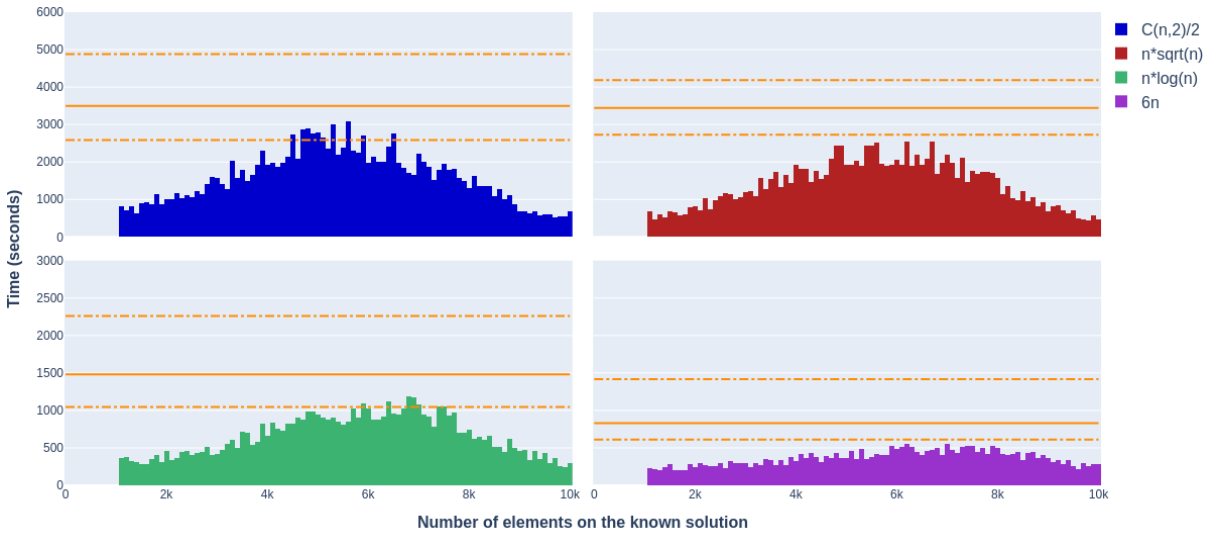


Figure 66 – Execution time versus size of the solution for **P3** solving **d-path**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

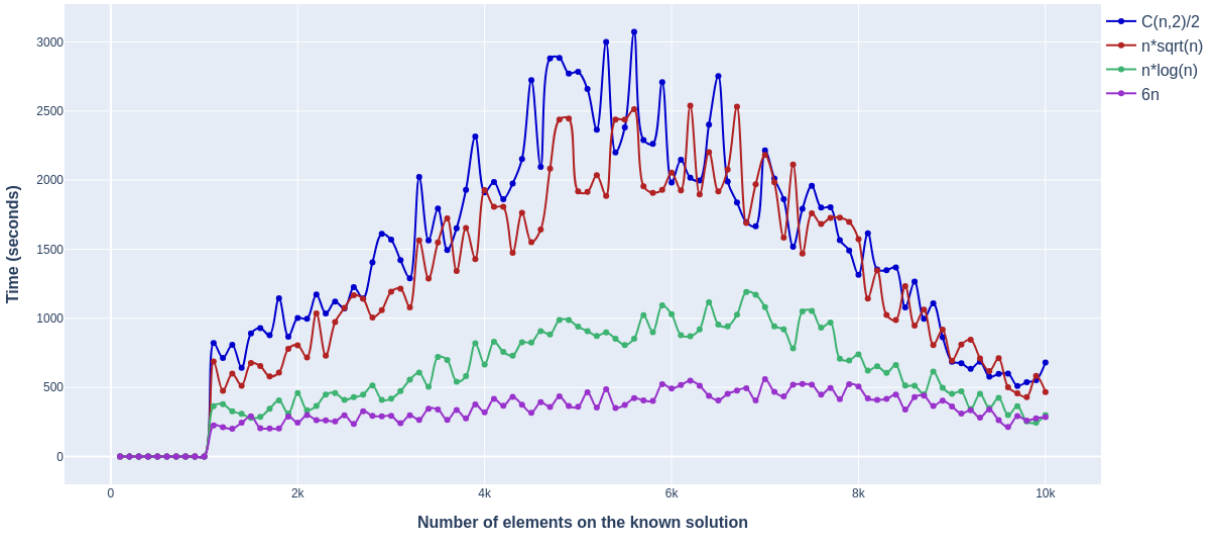


Figure 67 – Execution time versus size of the solution for **P3** solving **d-path**.

Figure 68 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-path**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

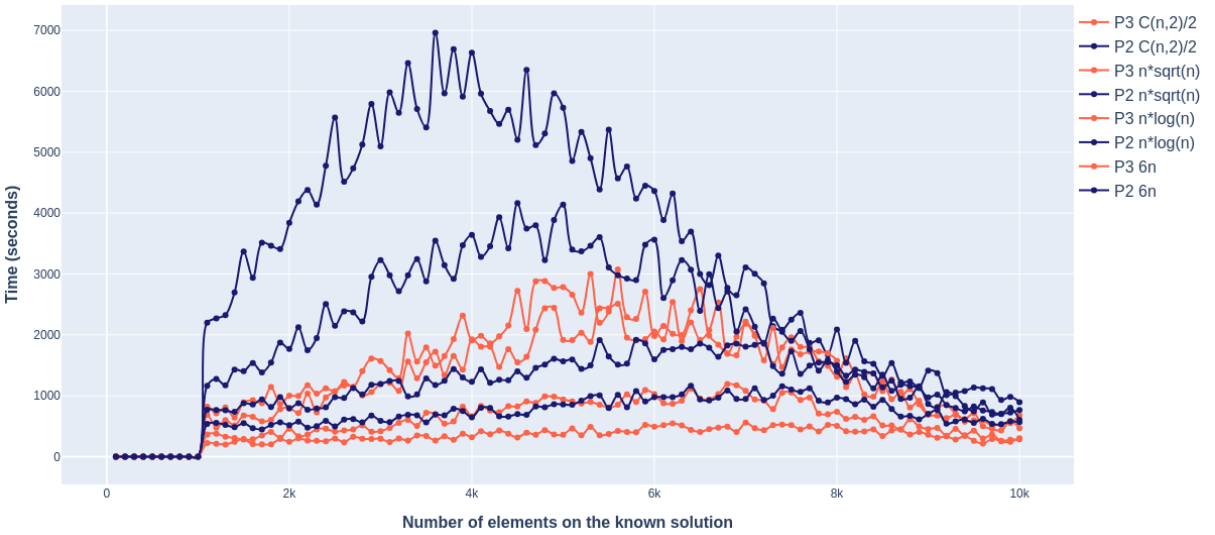


Figure 69 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-path**

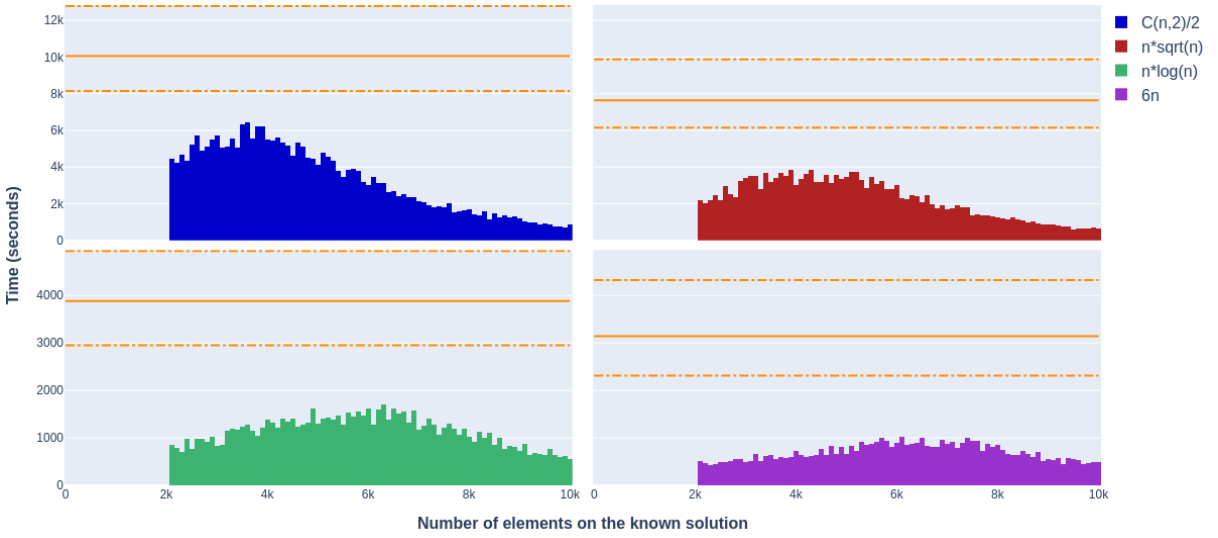


Figure 70 – Execution time versus size of the solution for **P2** solving **d-path-shortcut**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

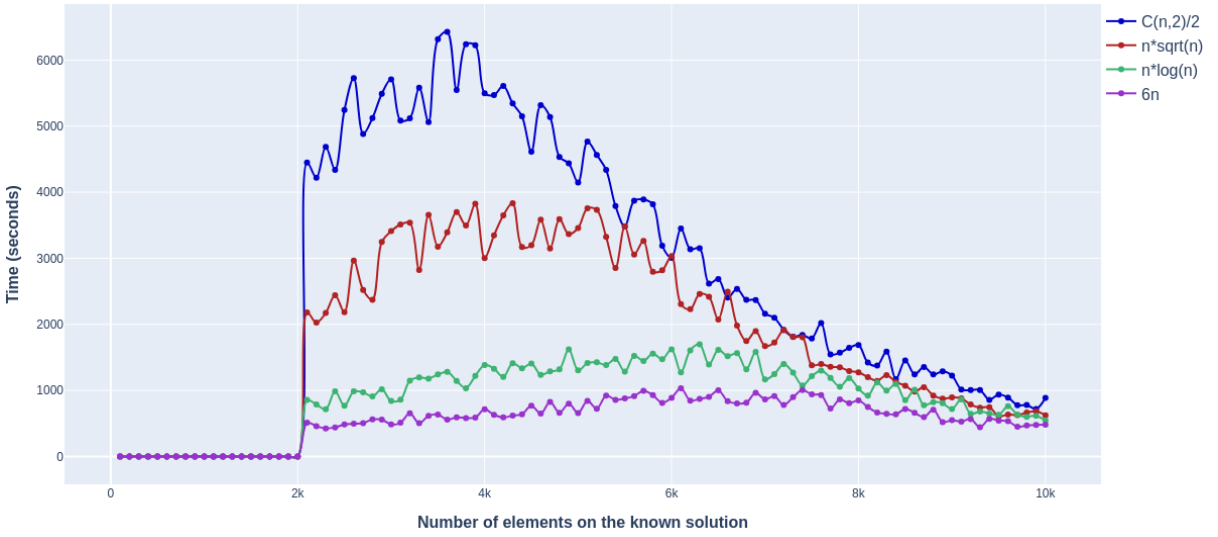


Figure 71 – Execution time versus size of the solution for **P2** solving **d-path-shortcut**

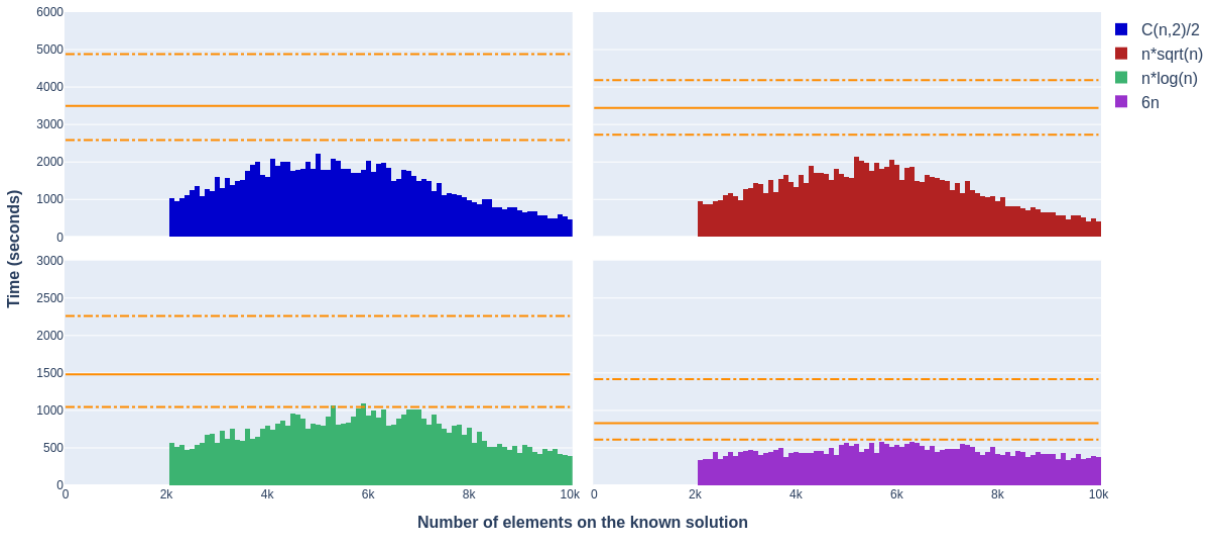


Figure 72 – Execution time versus size of the solution for **P3** solving **d-path-shortcut**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

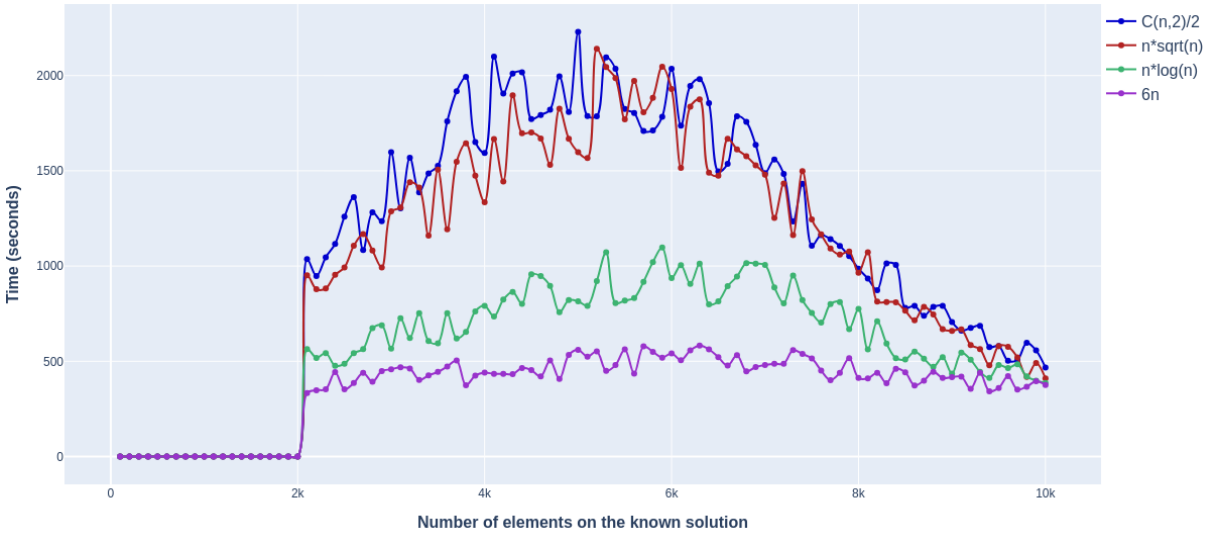


Figure 73 – Execution time versus size of the solution for **P3** solving **d-path-shortcut**

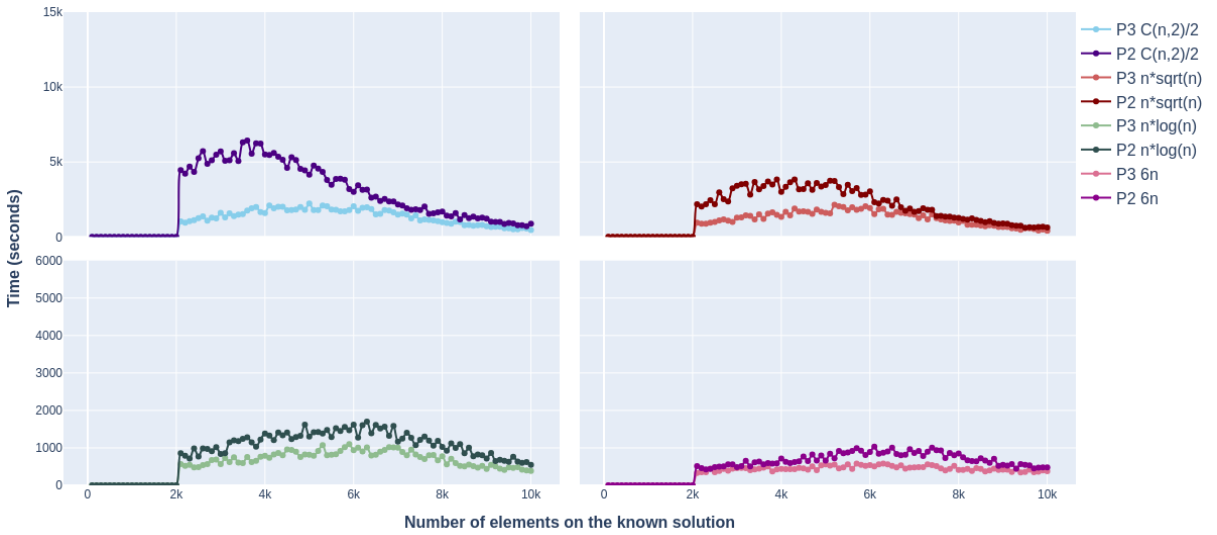


Figure 74 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-path-shortcut**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

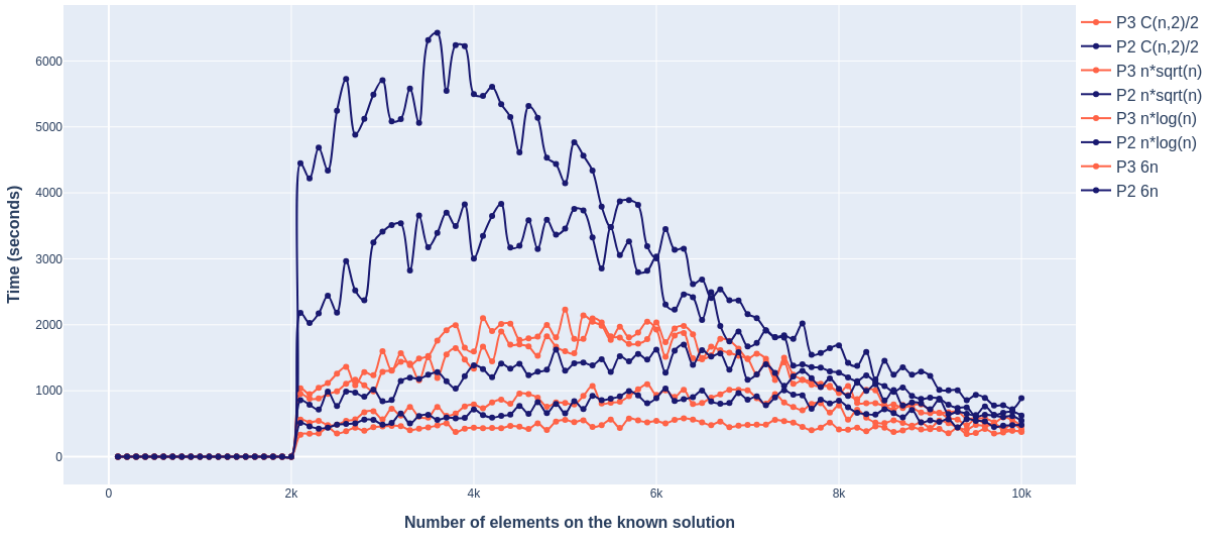


Figure 75 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-path-shortcut**

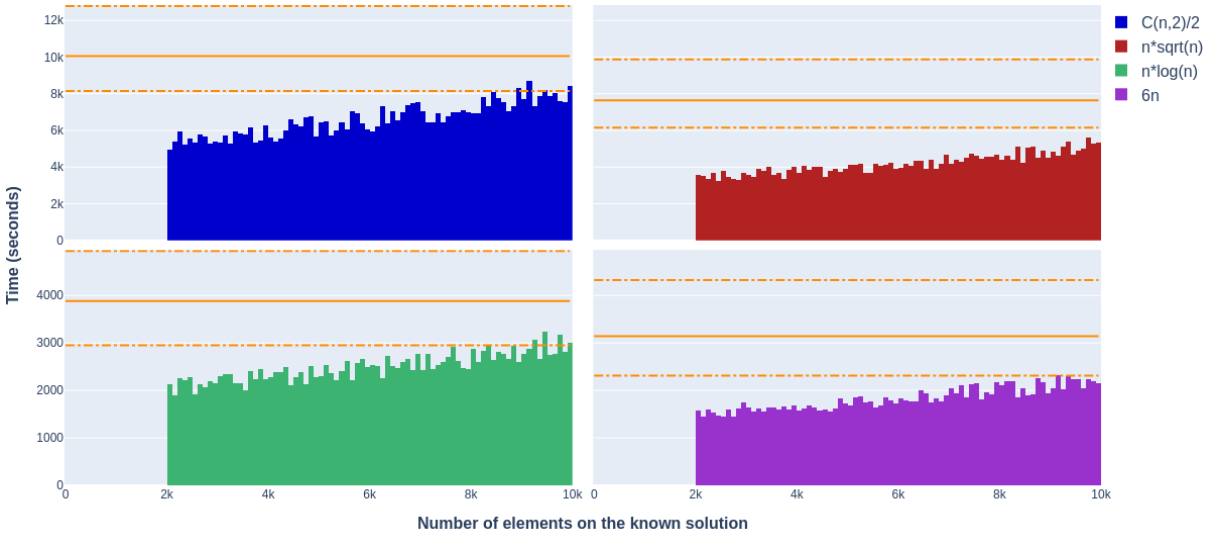


Figure 76 – Execution time versus size of the solution for **P2** solving **d-btree-leaves**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

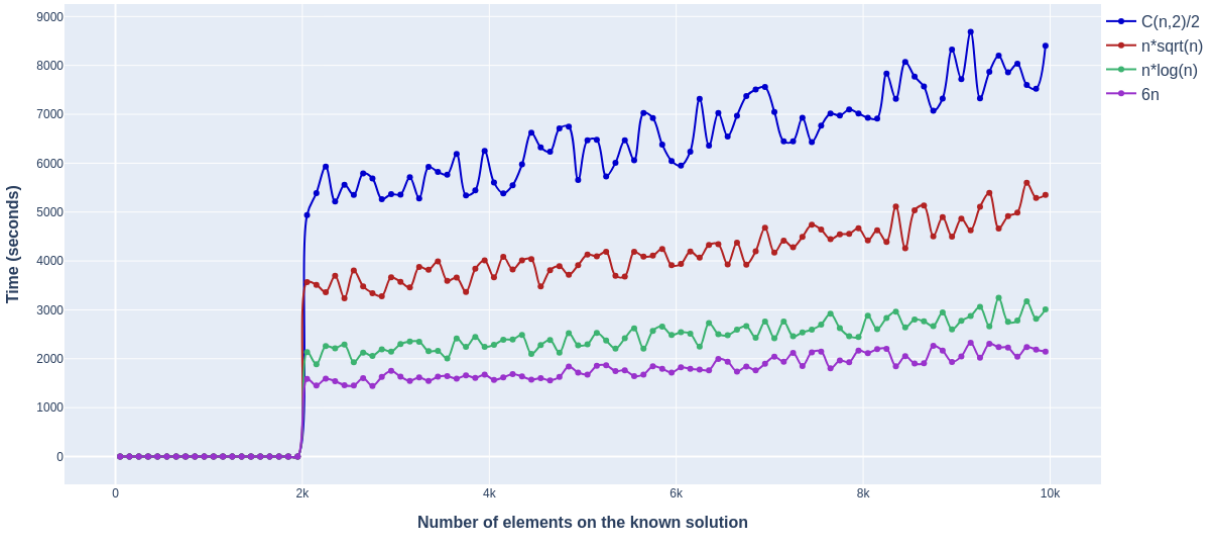


Figure 77 – Execution time versus size of the solution for **P2** solving **d-btree-leaves**

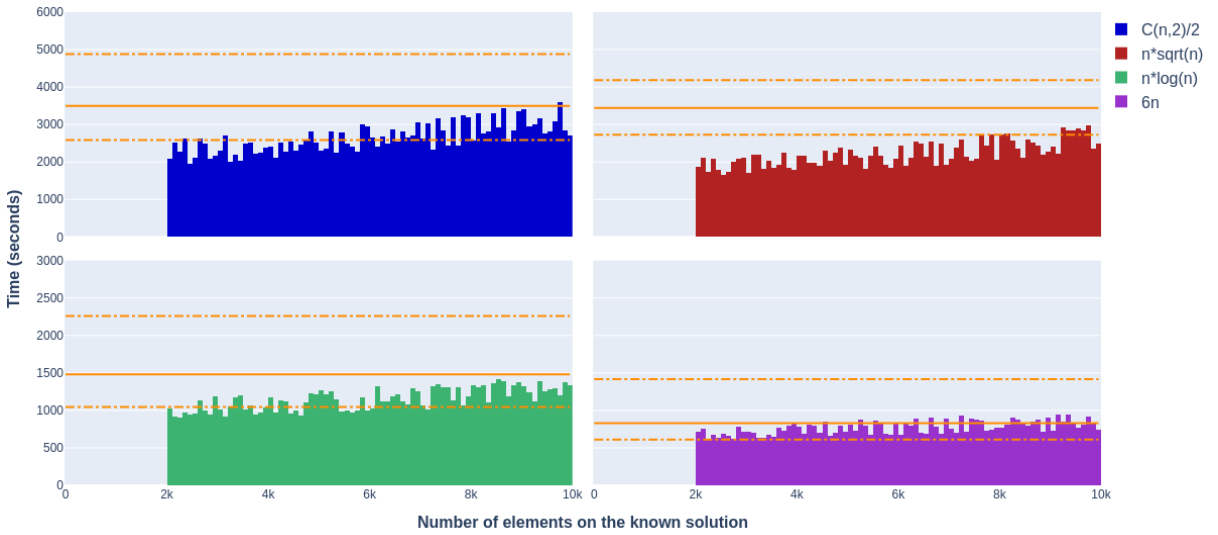


Figure 78 – Execution time versus size of the solution for **P3** solving **d-btree-leaves**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

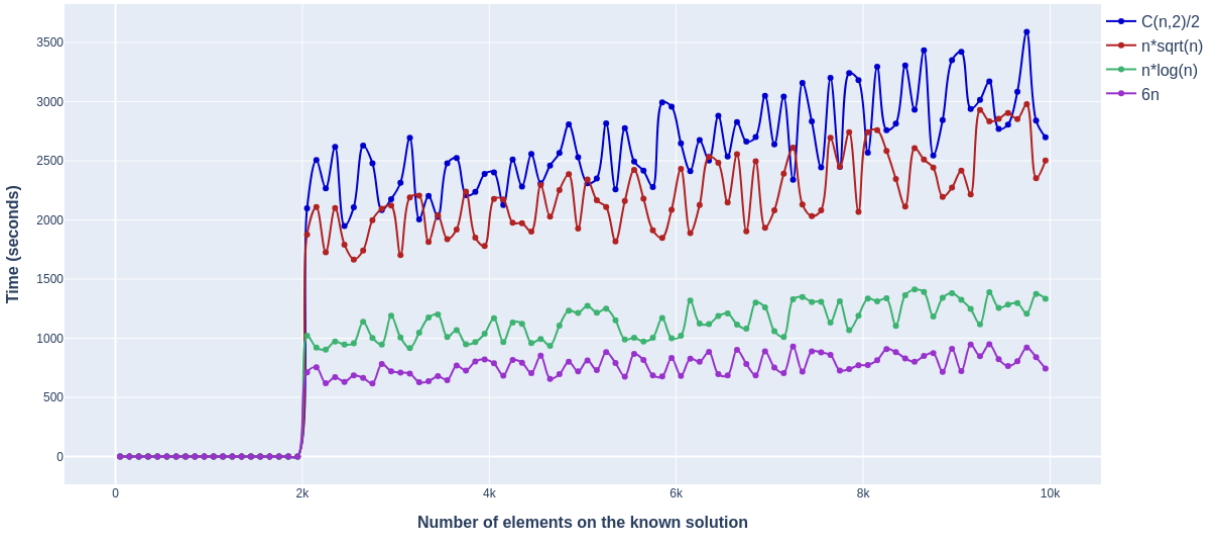


Figure 79 – Execution time versus size of the solution for **P2** solving **d-btree-leaves**

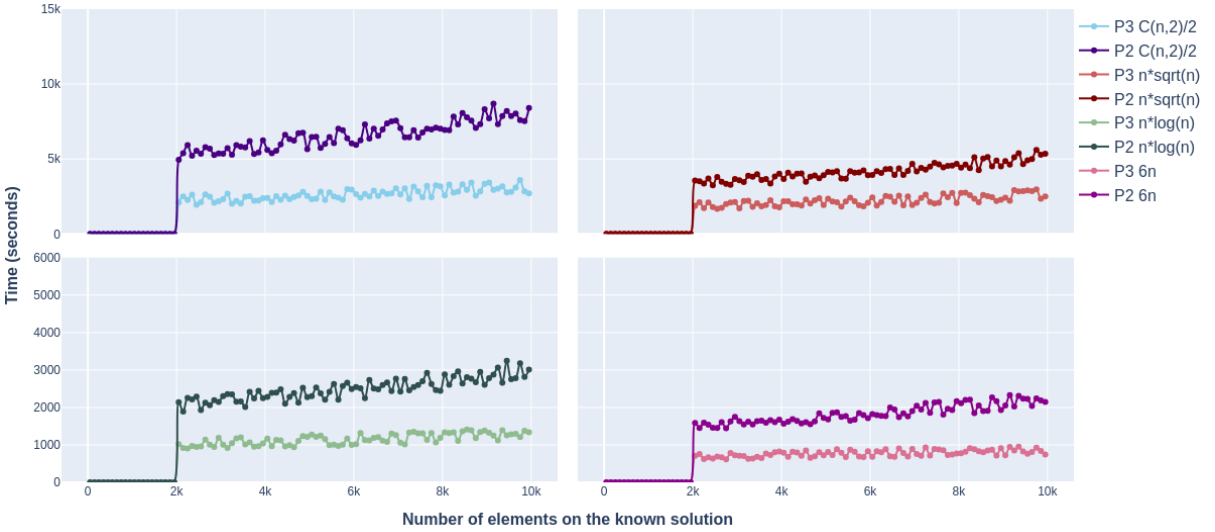


Figure 80 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-btree-leaves**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

Figure 81 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-btree-leaves**

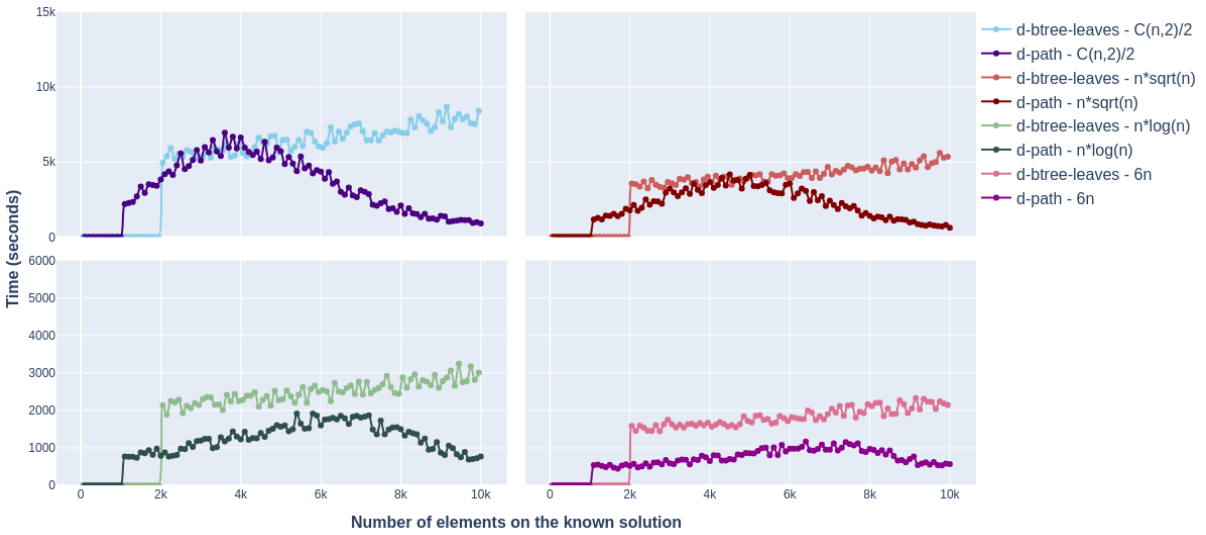


Figure 82 – Comparison of execution time versus of the size of the solution for **P2** solving **d-path** and **d-btree-leaves**

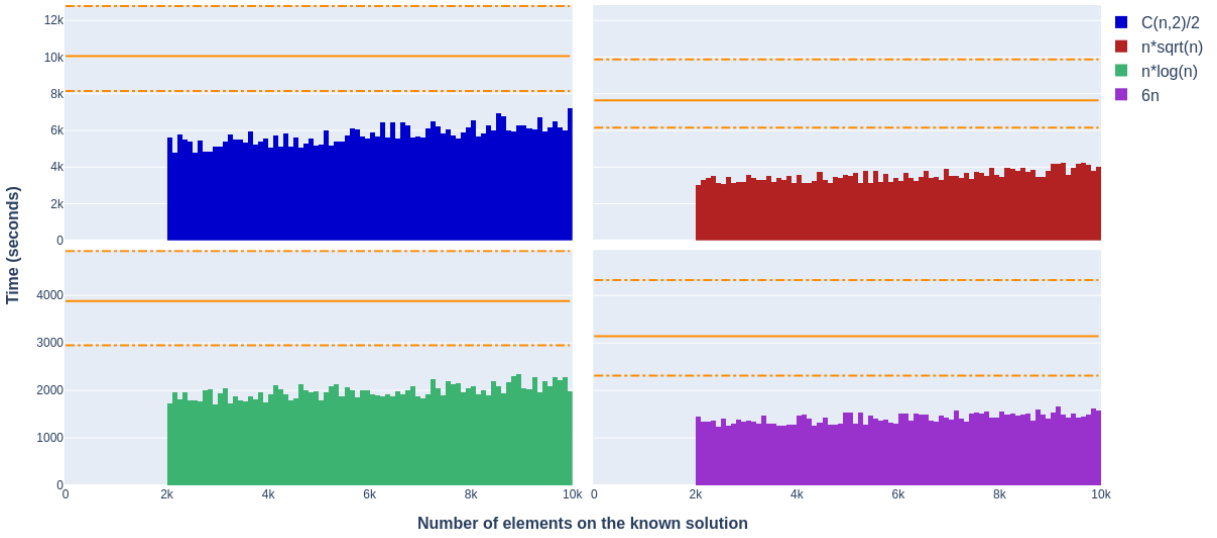


Figure 83 – Execution time versus size of the solution for **P2** solving **d-btree**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

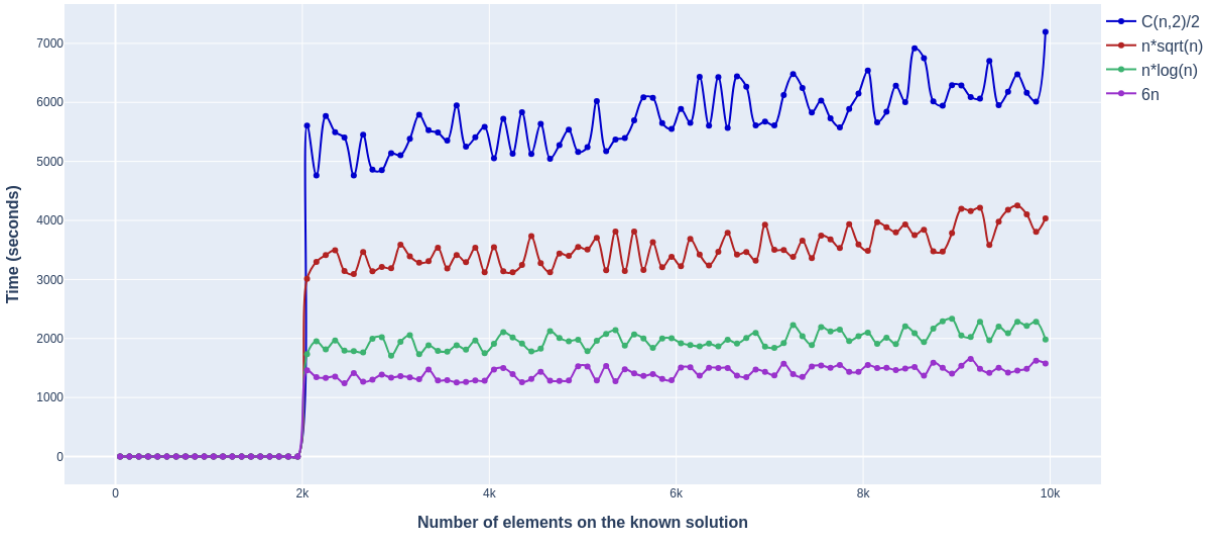


Figure 84 – Execution time versus size of the solution for **P2** solving **d-btree**

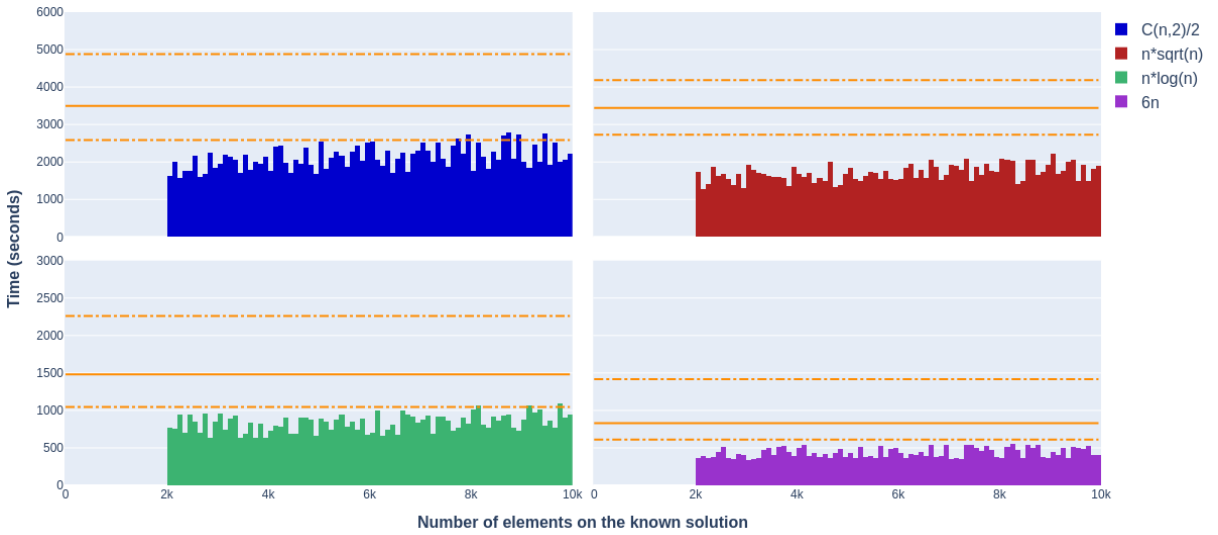


Figure 85 – Execution time versus size of the solution for **P3** solving **d-btree**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

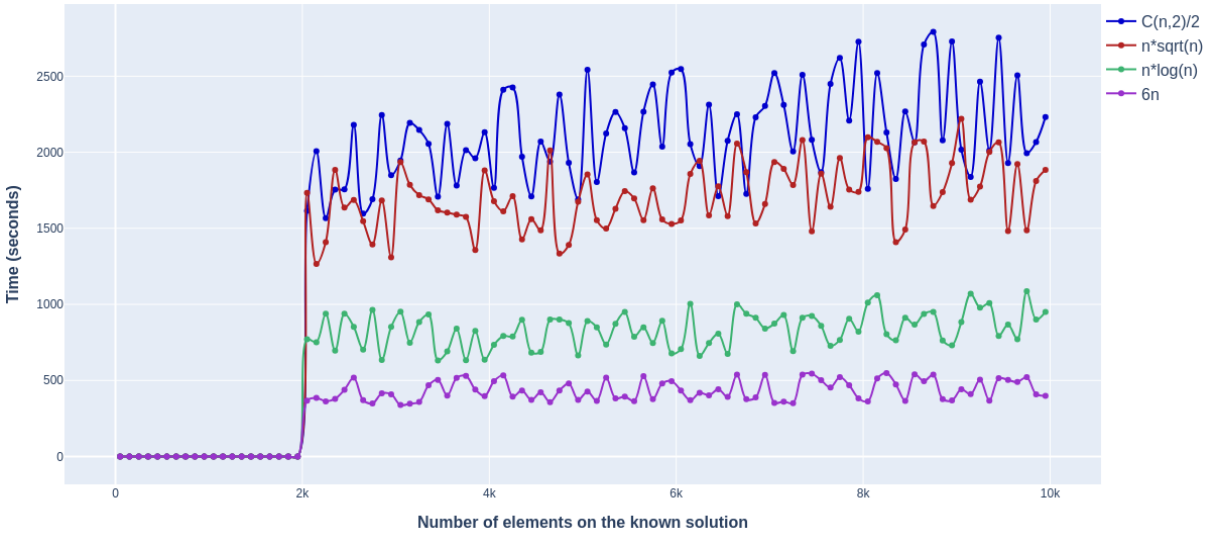


Figure 86 – Execution time versus size of the solution for **P3** solving **d-btree**

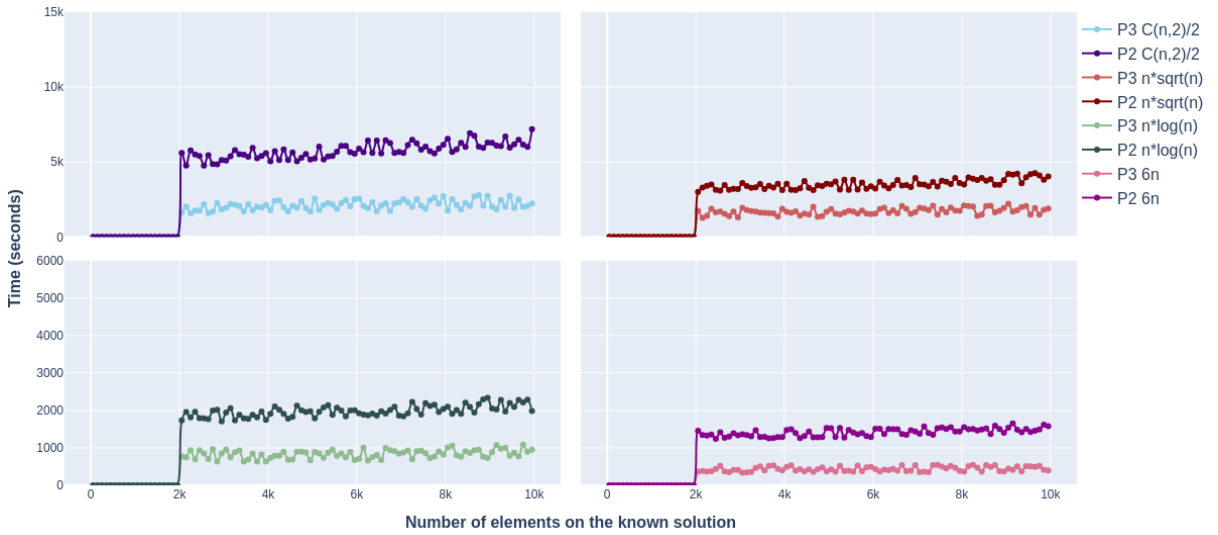


Figure 87 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-btree**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

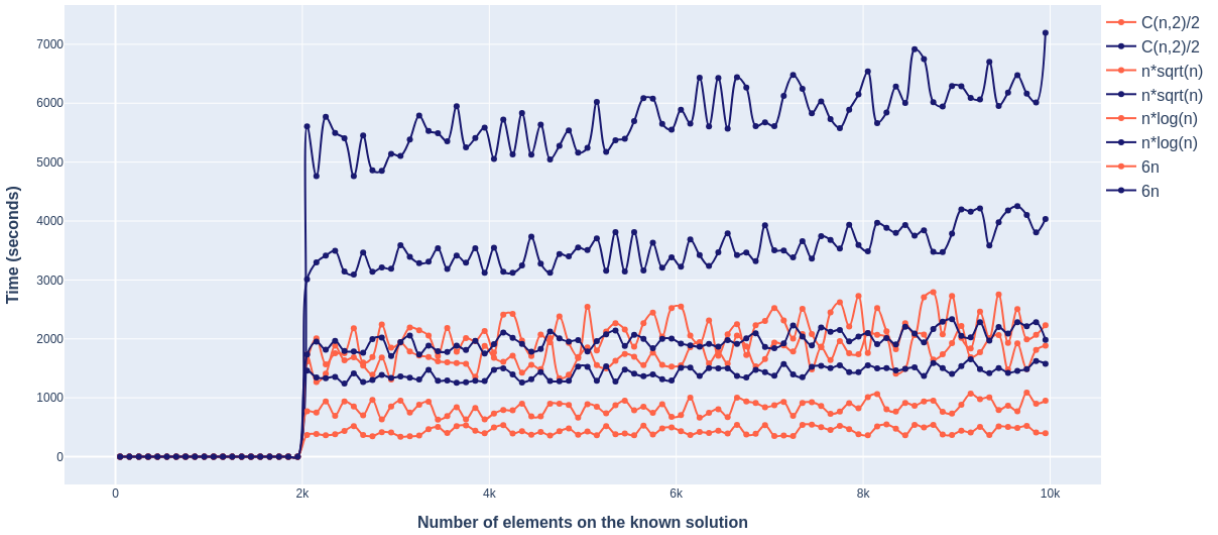


Figure 88 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-btree**

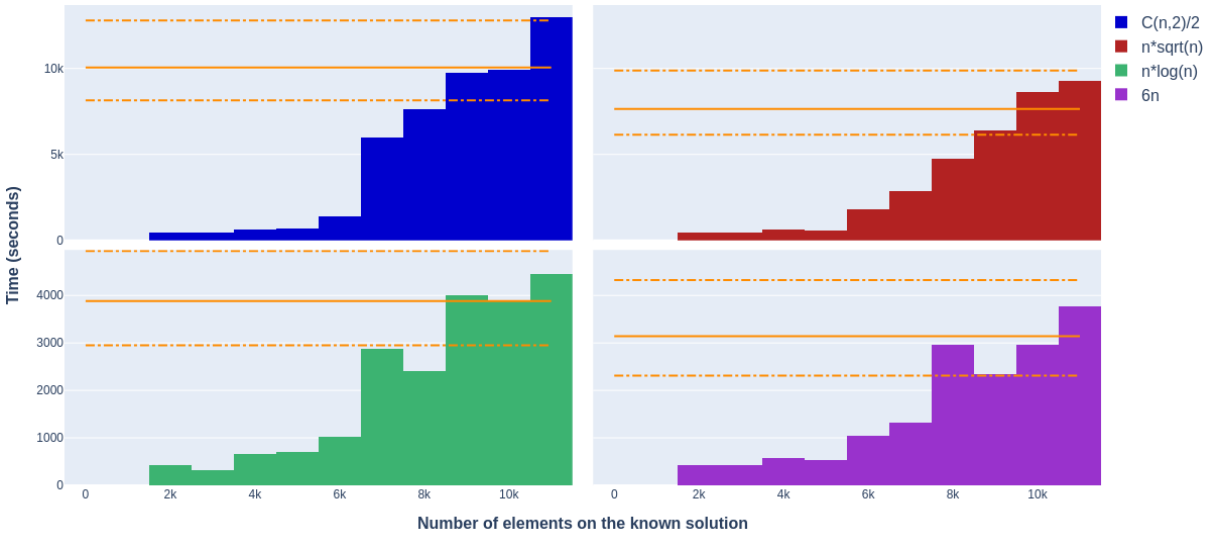


Figure 89 – Execution time versus size of the solution for **P2** solving **d-star-1k**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

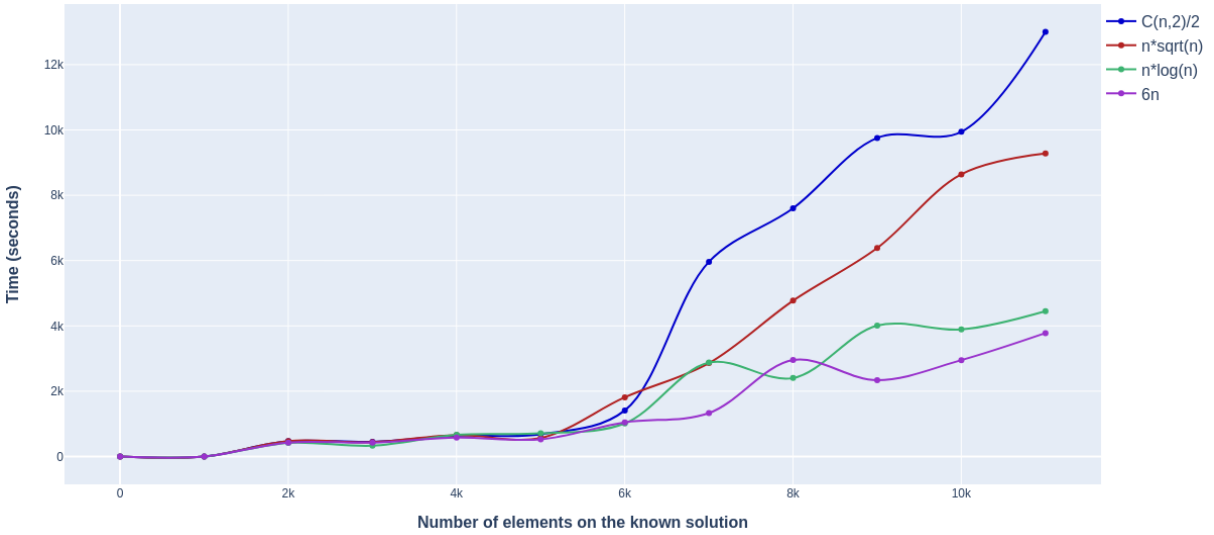


Figure 90 – Execution time versus size of the solution for **P2** solving **d-star-1k**

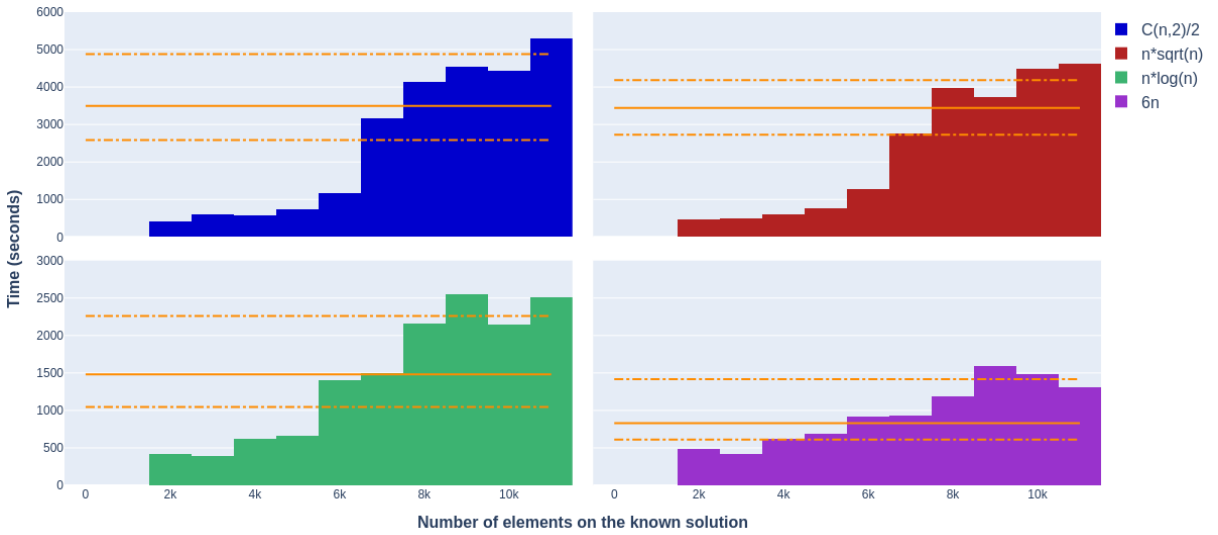


Figure 91 – Execution time versus size of the solution for **P3** solving **d-star-1k**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

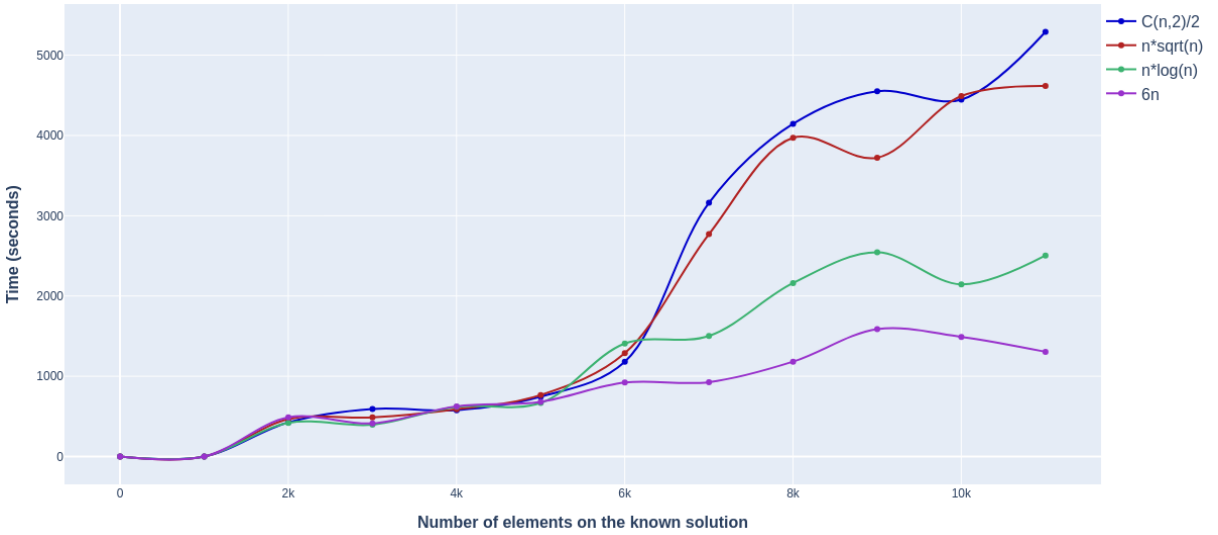


Figure 92 – Execution time versus size of the solution for **P3** solving **d-star-1k**

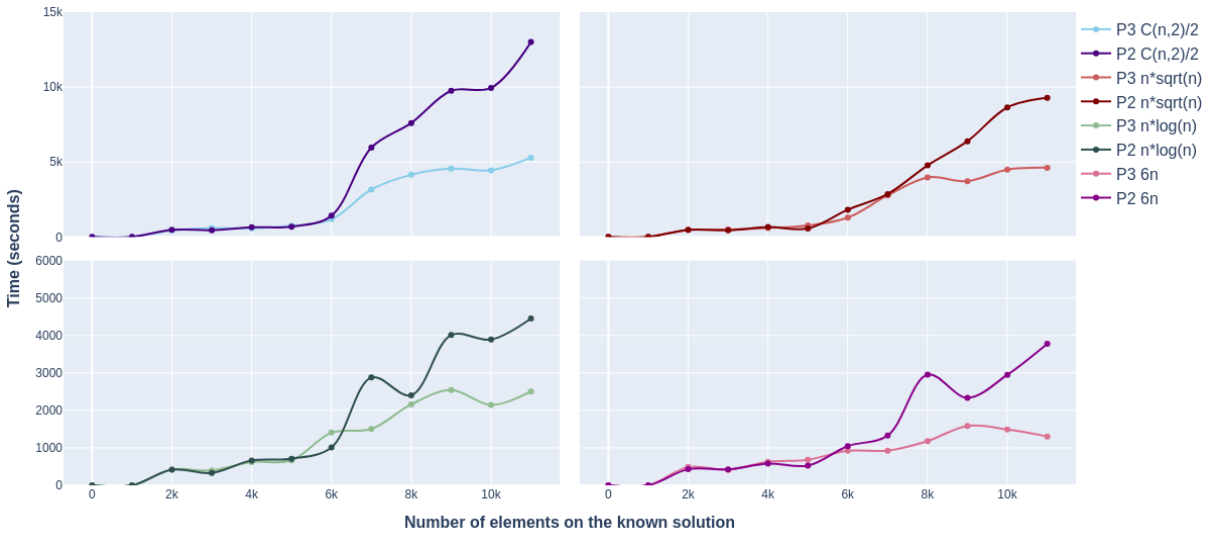


Figure 93 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-star-1k**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

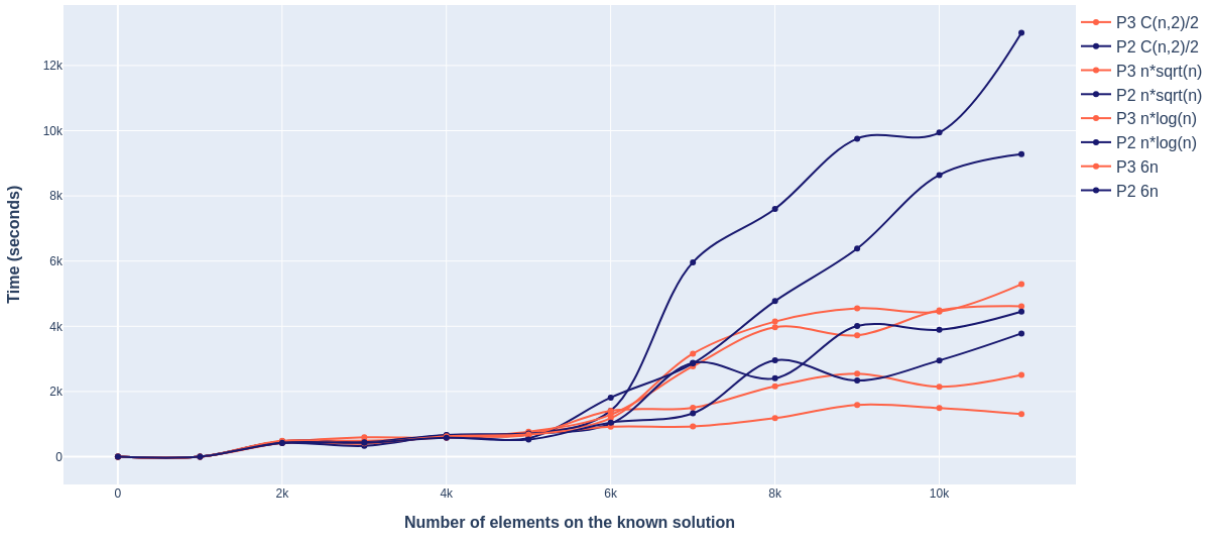


Figure 94 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-star-1k**

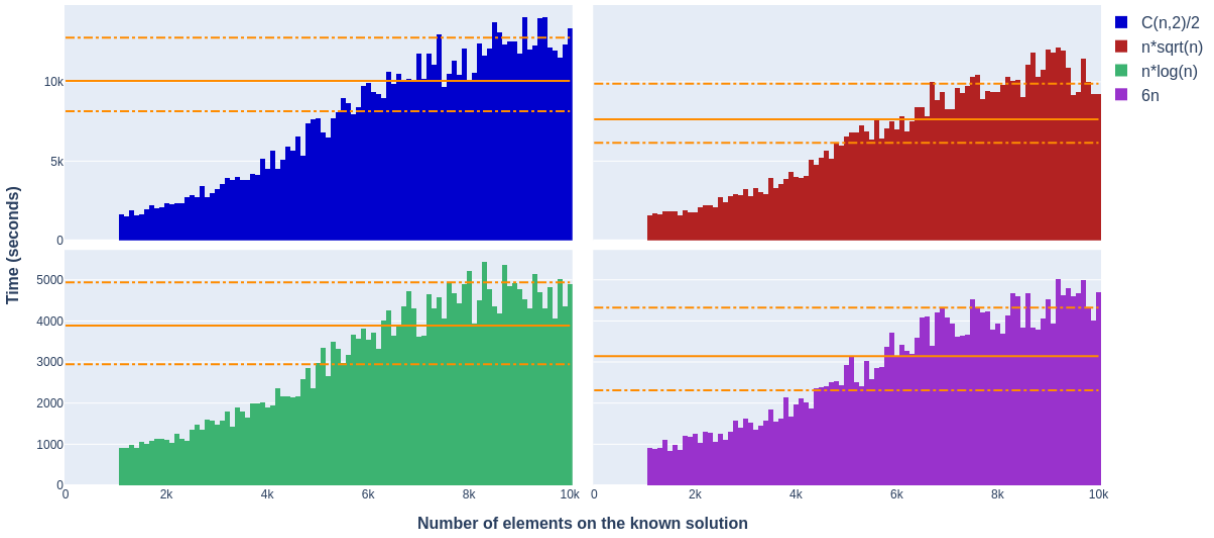


Figure 95 – Execution time versus size of the solution for **P2** solving **d-star-100**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

Figure 96 – Execution time versus size of the solution for **P2** solving **d-star-100**

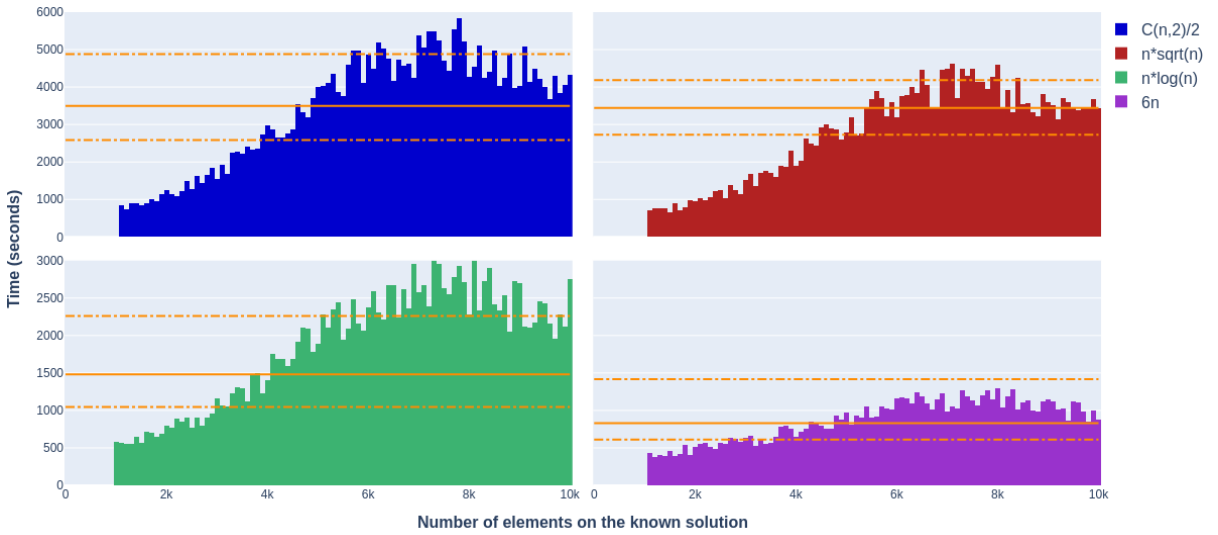


Figure 97 – Execution time versus size of the solution for **P3** solving **d-star-100**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

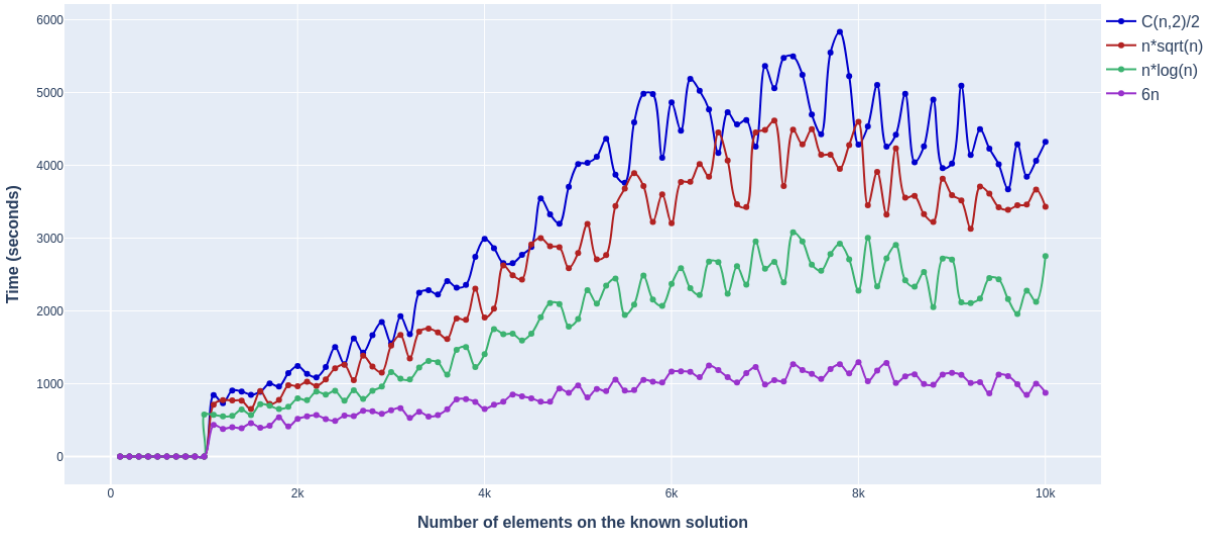


Figure 98 – Execution time versus size of the solution for **P3** solving **d-star-100**

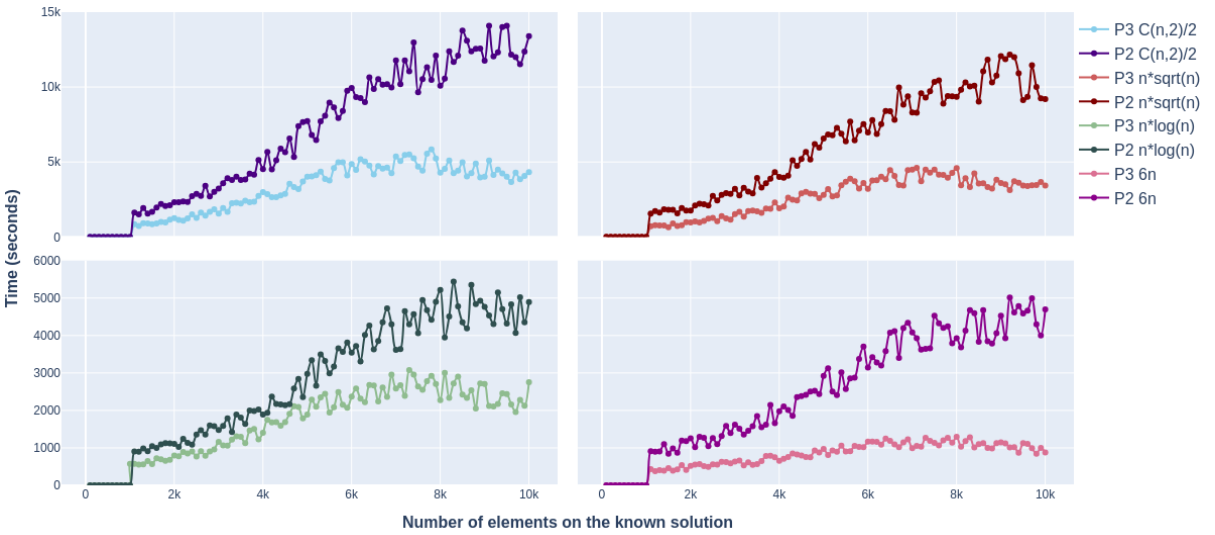


Figure 99 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-star-100**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

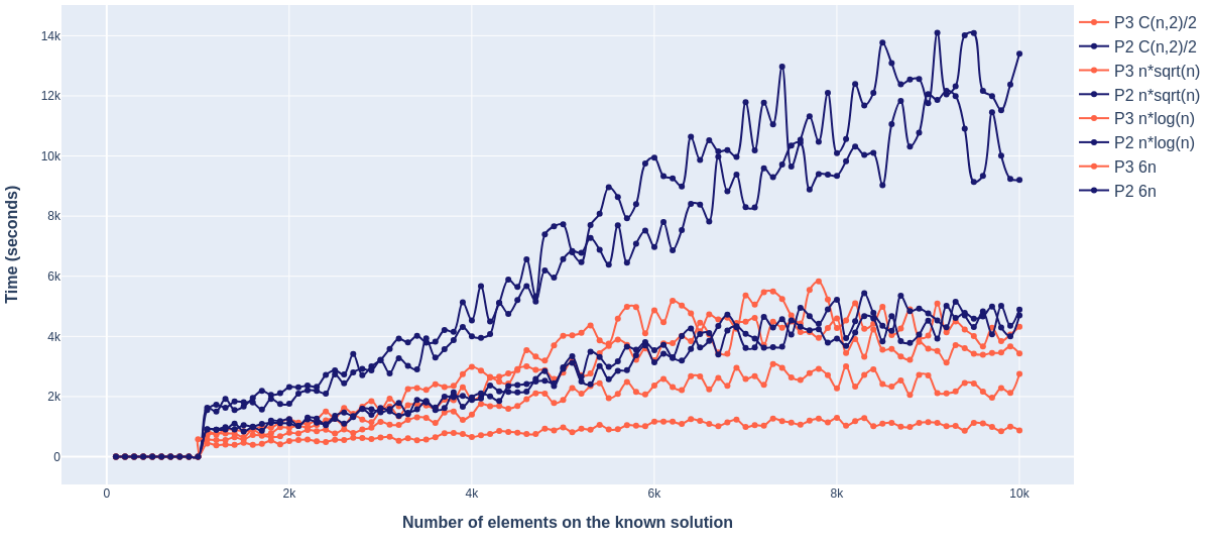


Figure 100 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-star-100**

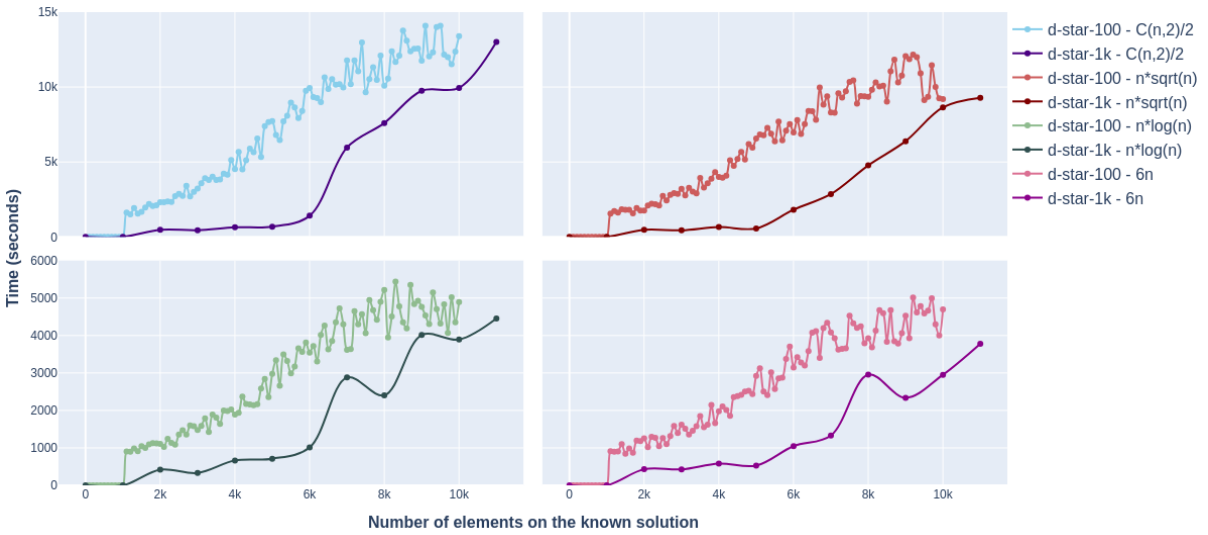


Figure 101 – Comparison of execution time versus of the size of the solution for **P2** solving **d-star-1k** and **d-star-100**

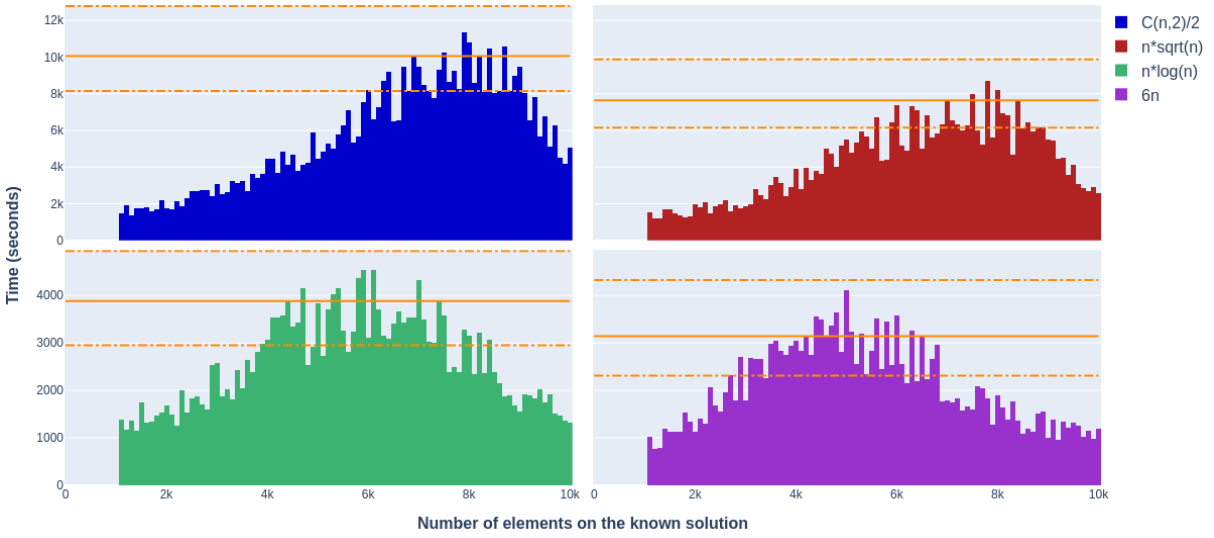


Figure 102 – Execution time versus size of the solution for **P2** solving **d-star-20**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

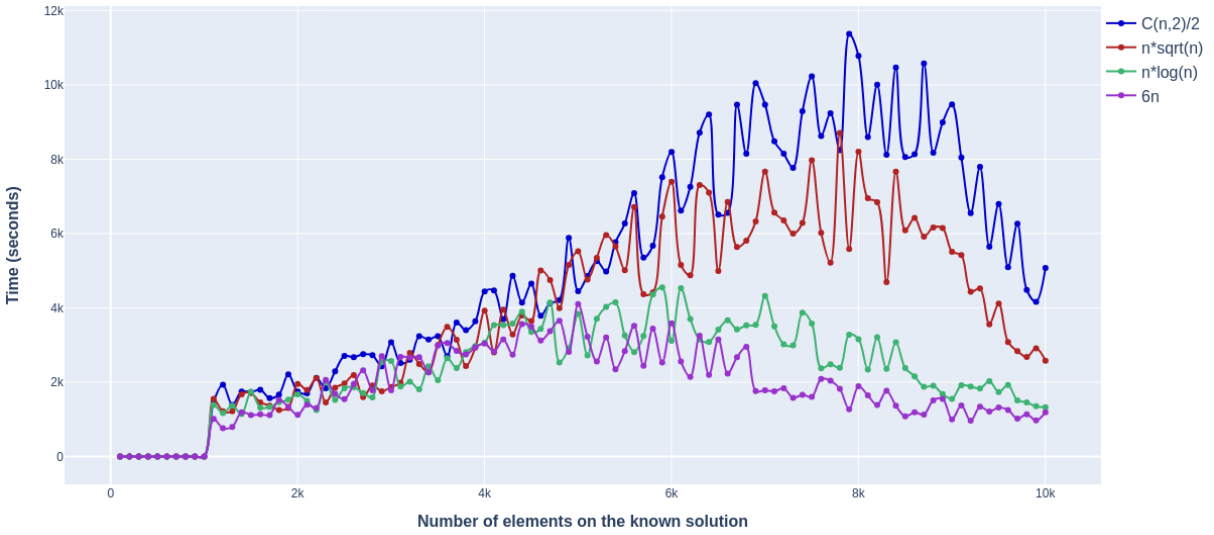


Figure 103 – Execution time versus size of the solution for **P2** solving **d-star-20**

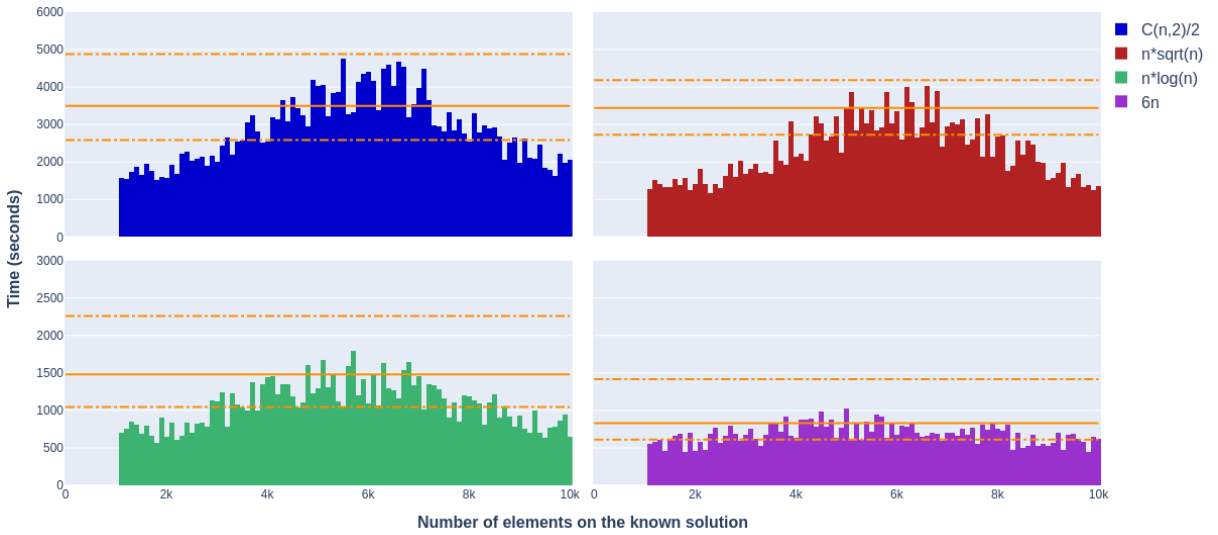


Figure 104 – Execution time versus size of the solution for **P3** solving **d-star-20**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

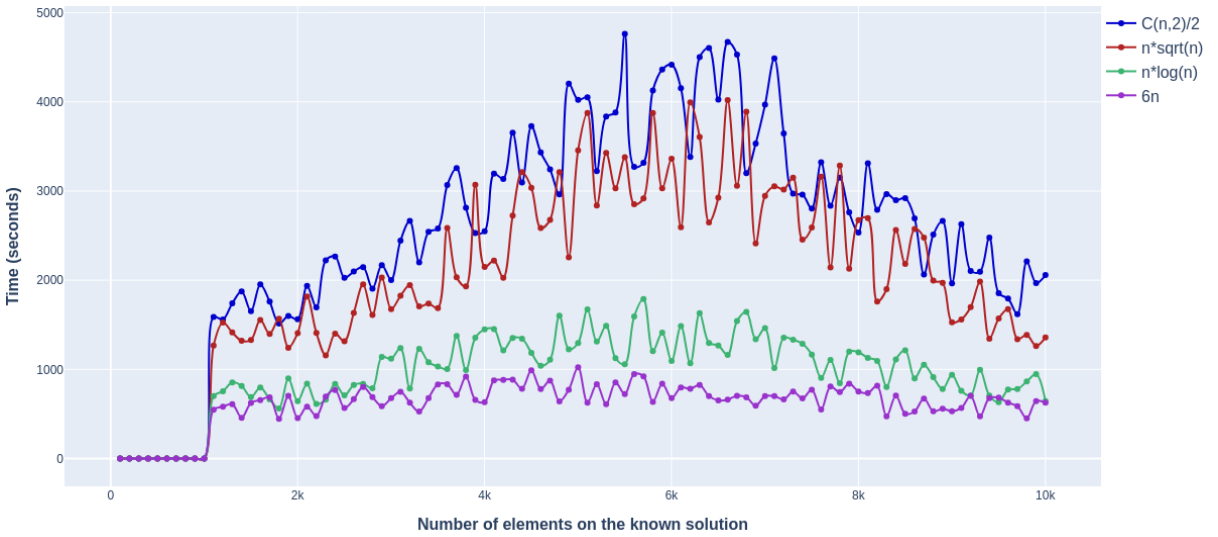


Figure 105 – Execution time versus size of the solution for **P3** solving **d-star-20**

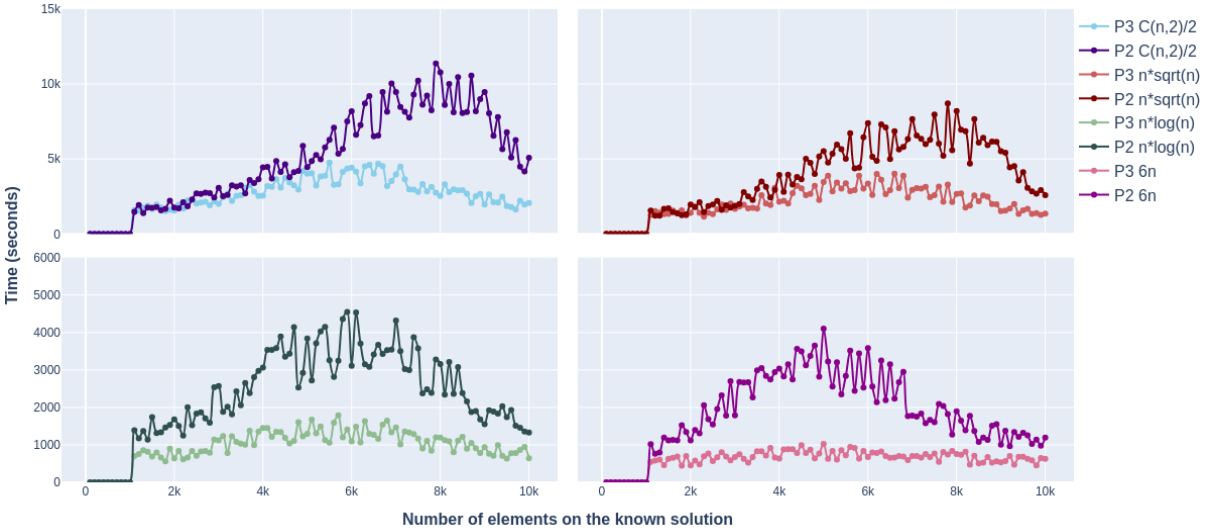


Figure 106 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-star-20**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

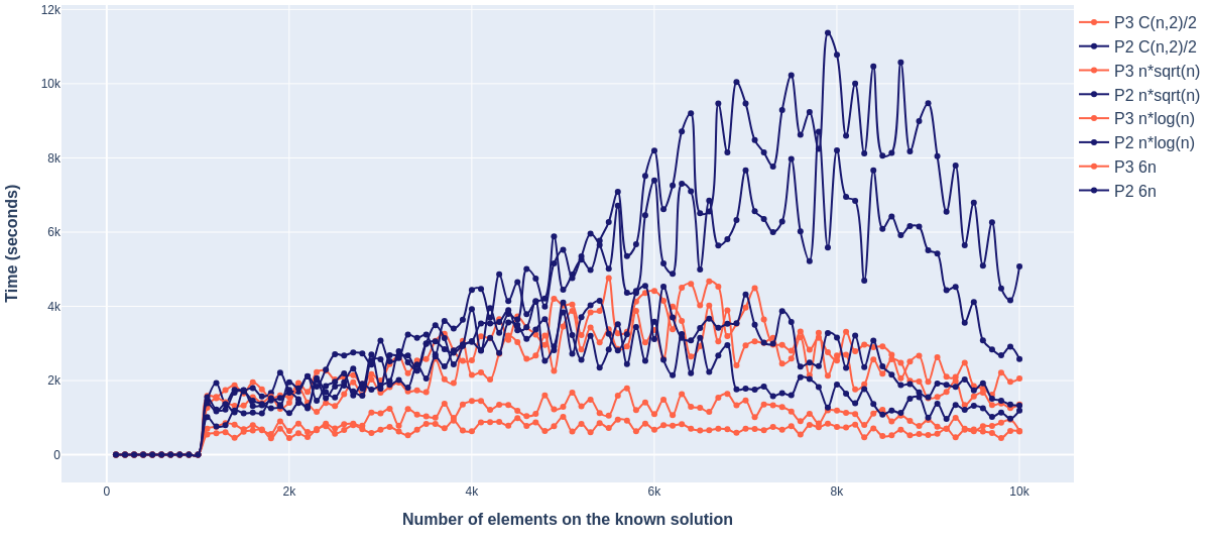


Figure 107 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-star-20**

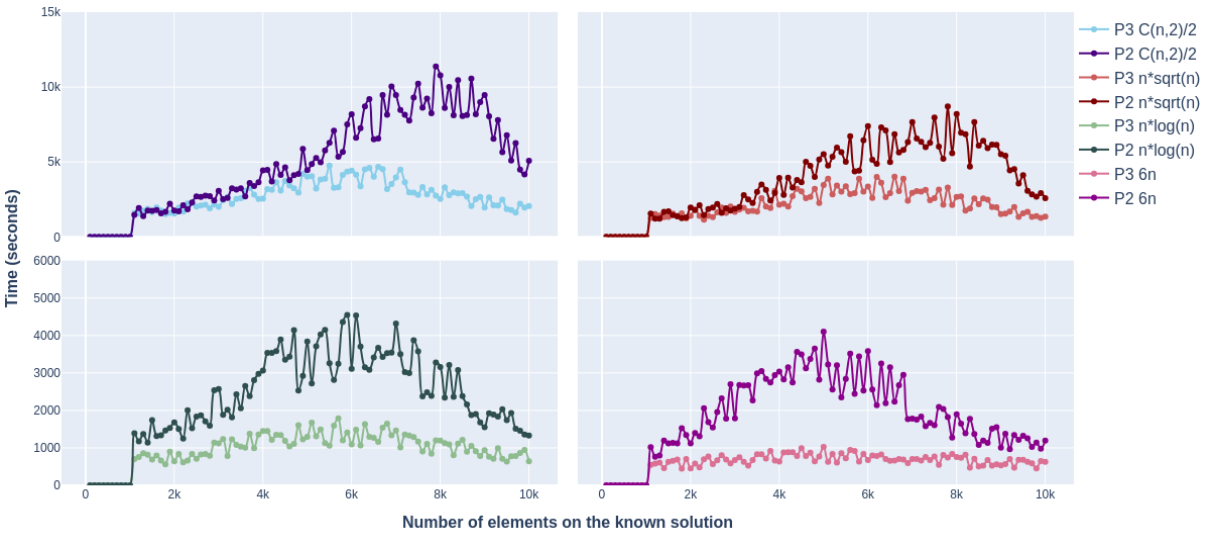


Figure 108 – Comparison of execution time versus of the size of the solution for **P2** and **P3** solving **d-star-20**

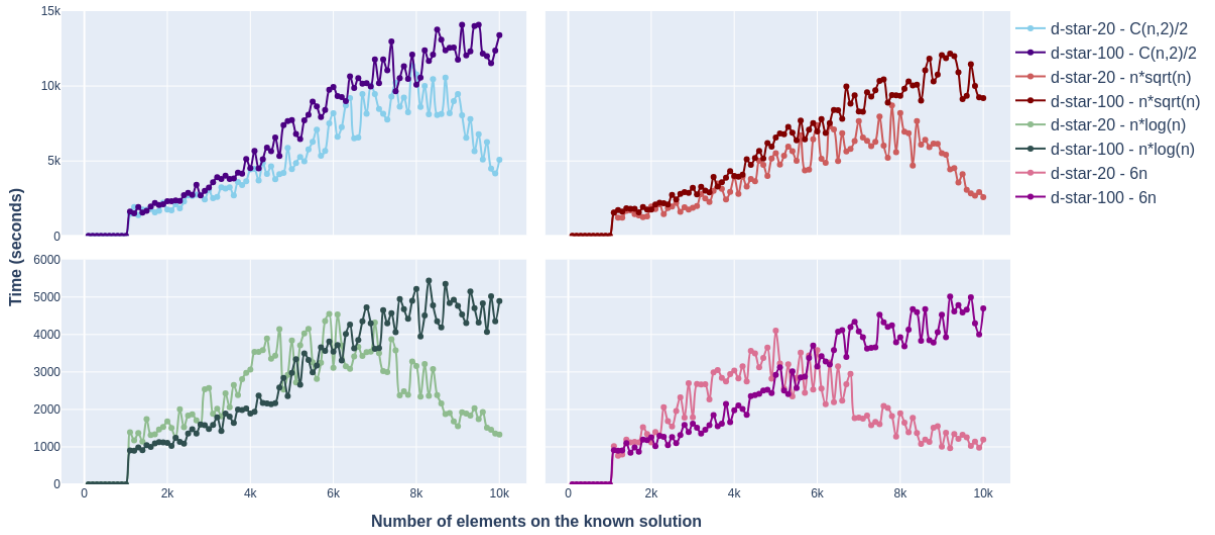


Figure 109 – Comparison of execution time versus of the size of the solution for **P2** solving **d-star-20** and **d-star-100**

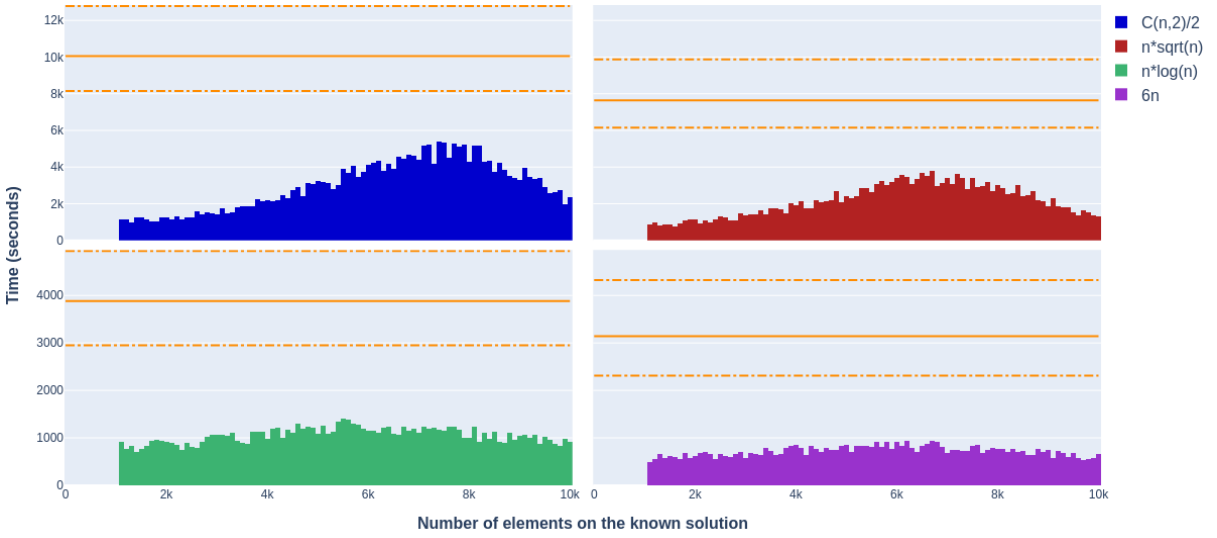


Figure 110 – Execution time versus size of the solution for **P2** solving **d-star-no-touch**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

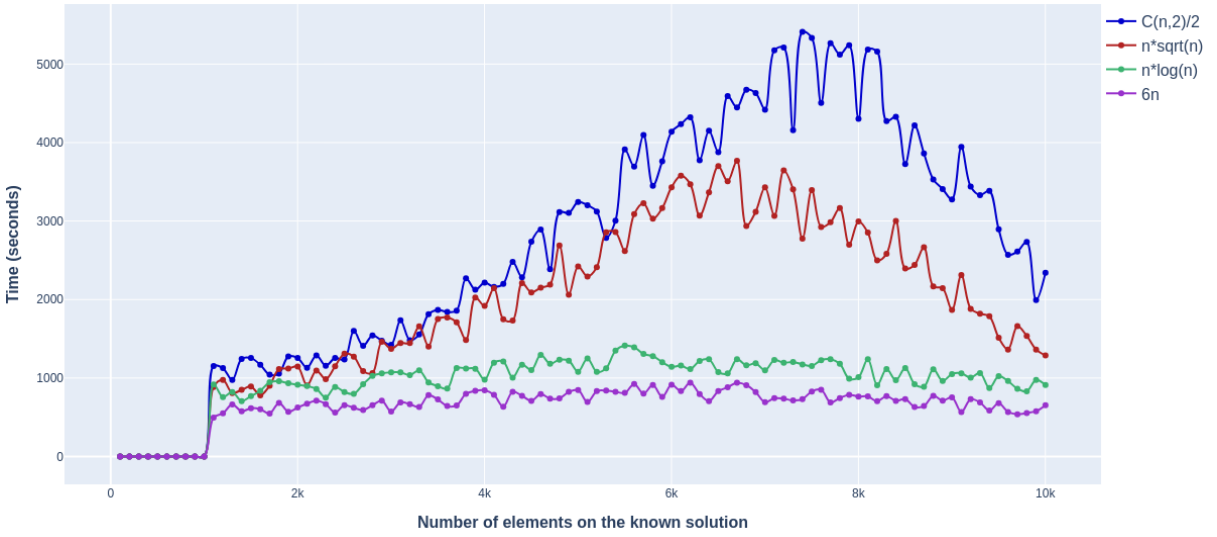


Figure 111 – Execution time versus size of the solution for **P2** solving **d-star-no-touch**

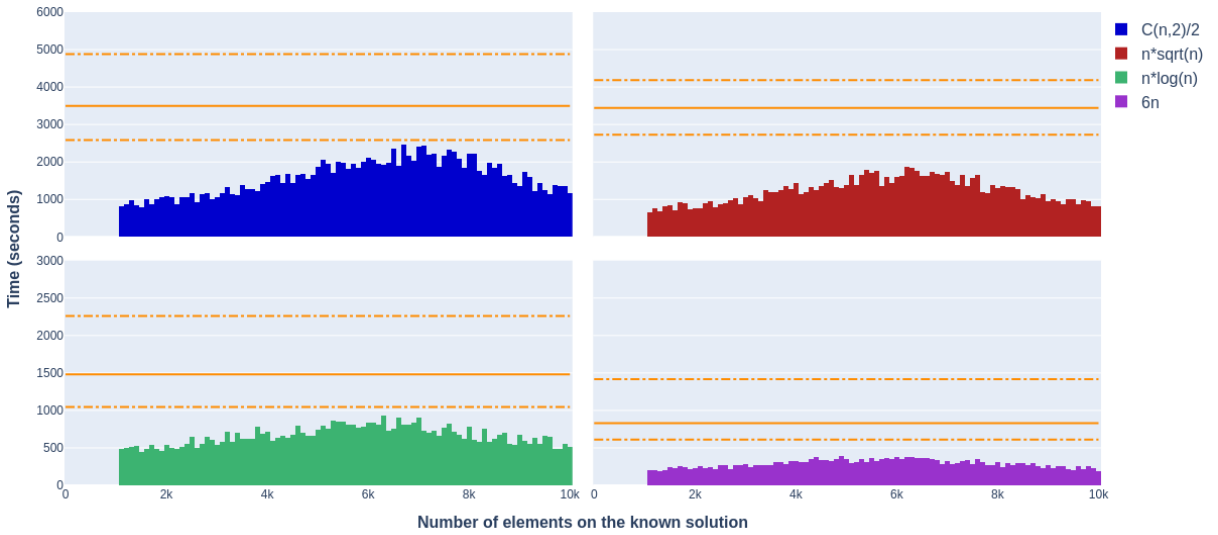


Figure 112 – Execution time versus size of the solution for **P3** solving **d-star-no-touch**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

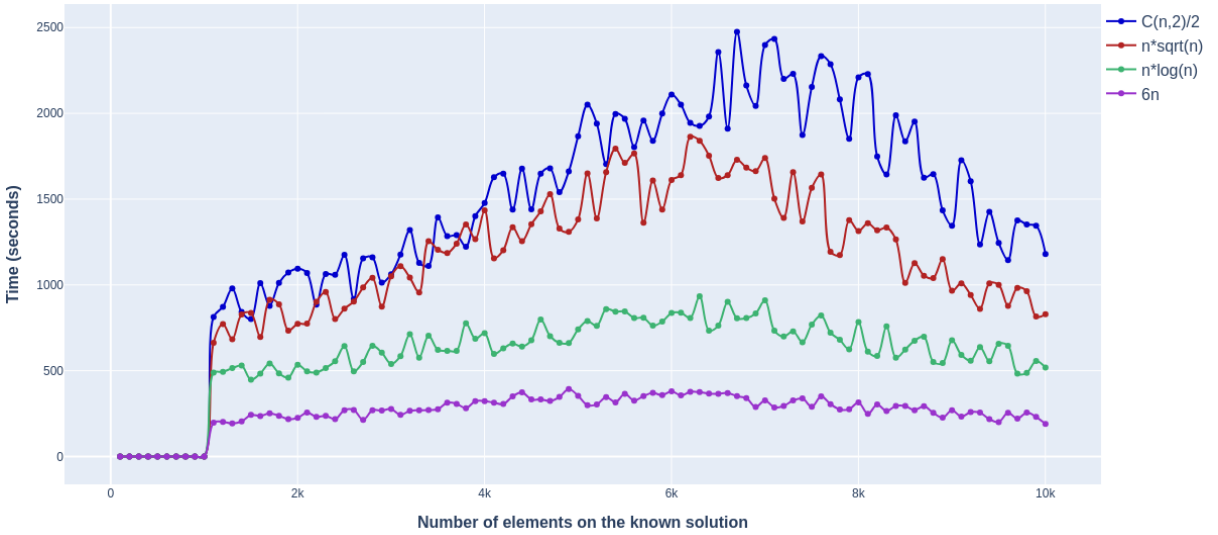


Figure 113 – Execution time versus size of the solution for **P3** solving **d-star-no-touch**

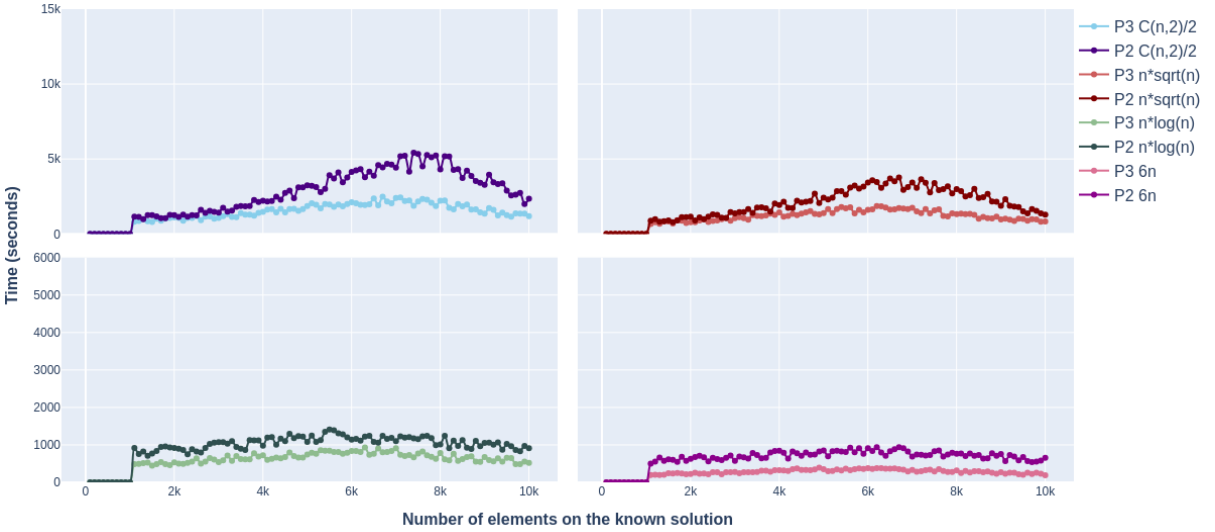


Figure 114 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-star-no-touch**. Charts are shown in decreasing edge density from left-to-right top-to-bottom

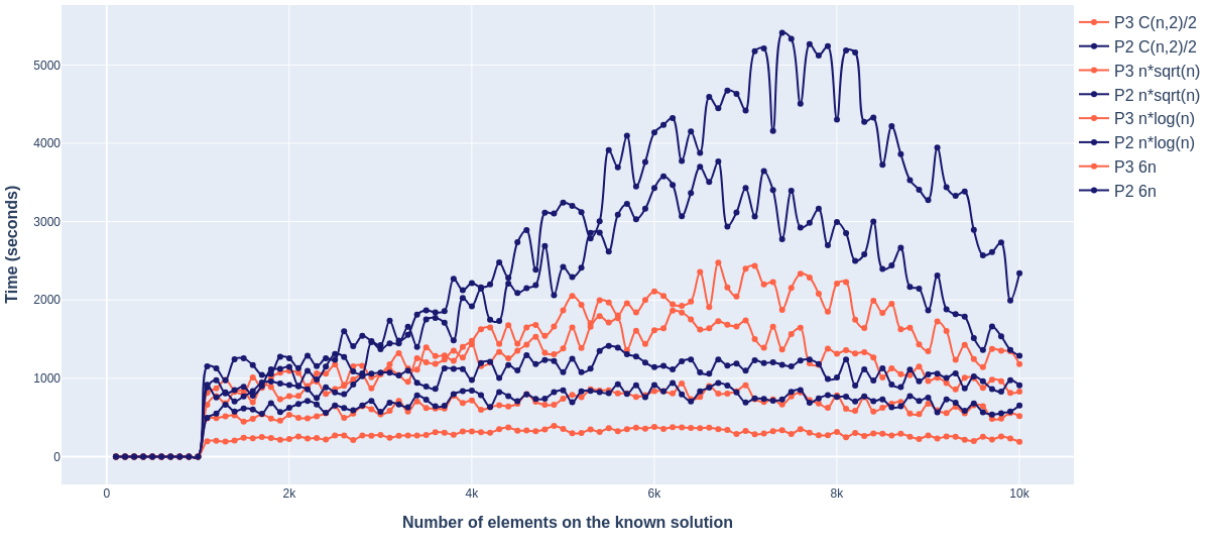


Figure 115 – Comparison of the execution time versus size of the solution between **P2** and **P3** solving **d-star-no-touch**

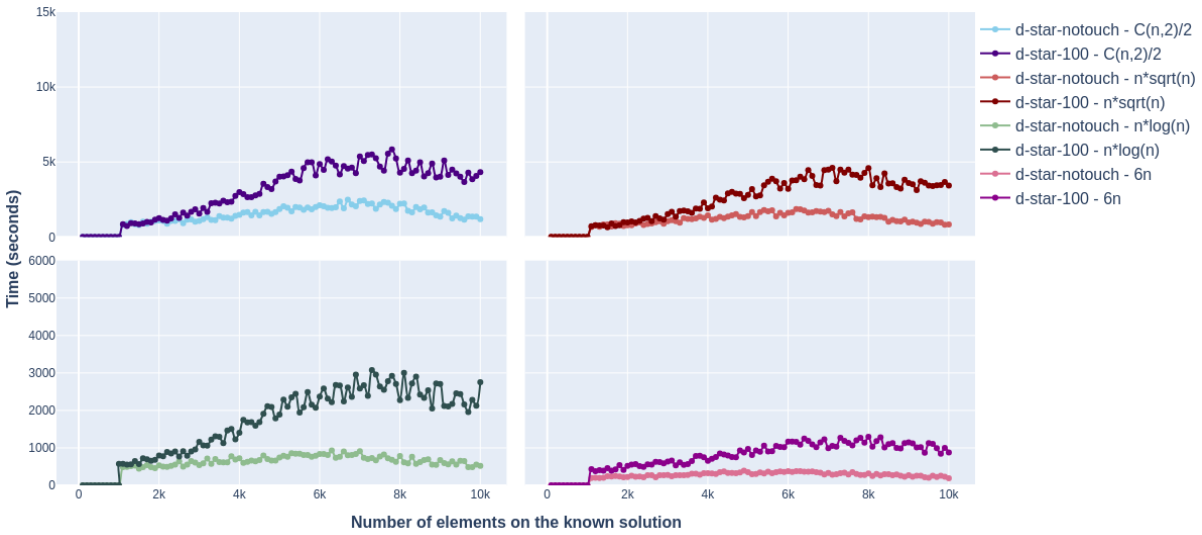


Figure 116 – Comparison of execution time versus of the size of the solution for **P3** solving **d-star-notouch** and **d-star-100**

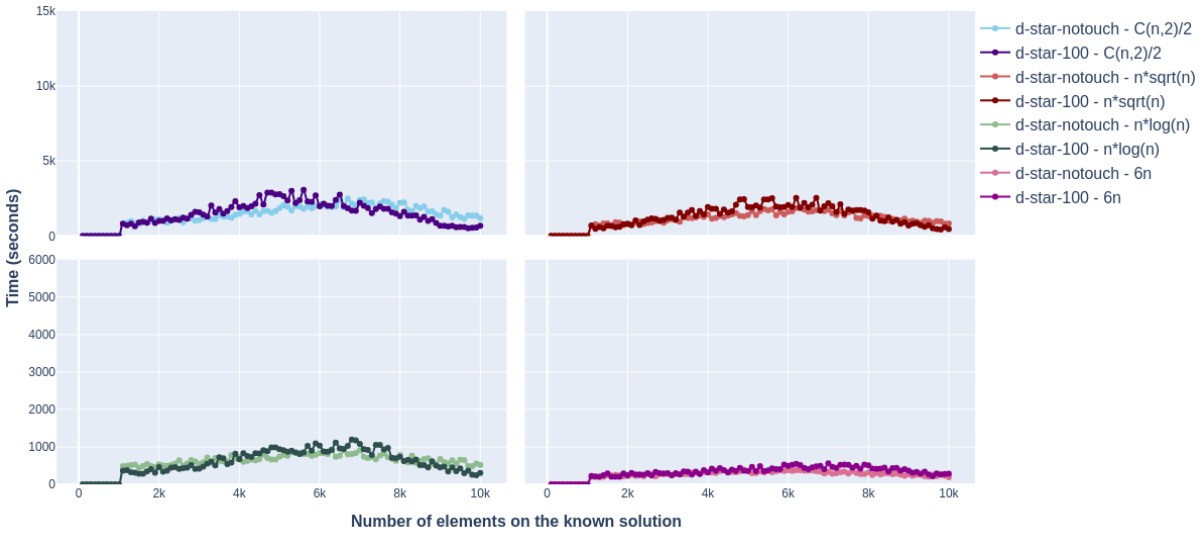


Figure 117 – Comparison of execution time versus of the size of the solution for **P3** solving **d-star-notouch** and **d-path**

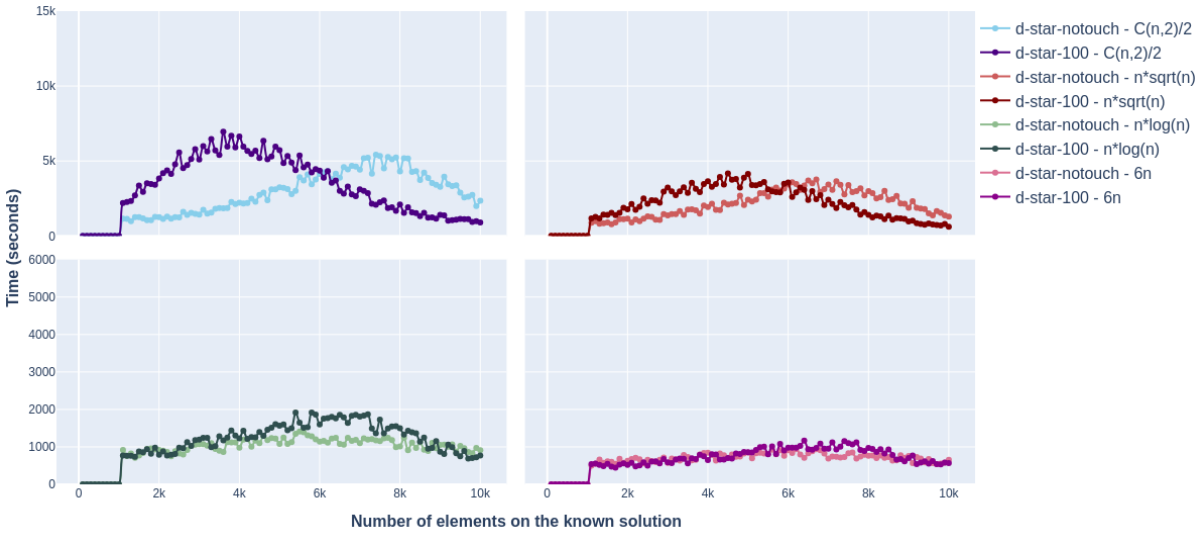


Figure 118 – Comparison of execution time versus of the size of the solution for **P2** solving **d-star-notouch** and **d-star-100**