PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

COMPLEXITY, LOWER BOUNDS, AND ALGORITHMS FOR SEARCHING INFECTED NODES IN UNCERTAIN TREES

JOSÉ BABOUN LARACH

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JOSÉ VERSCHAE

Santiago de Chile, September 2023

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

COMPLEXITY, LOWER BOUNDS, AND ALGORITHMS FOR SEARCHING INFECTED NODES IN UNCERTAIN TREES

JOSÉ BABOUN LARACH

Members of the Committee:
JOSÉ VERSCHAE
PABLO BARCELÓ
JANNIK MATUSCHKE
IGNACIO VARGAS Ignacio Vargas

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, September 2023

To my parents, brothers, and Cata
for their inconditional support

ACKNOWLEDGEMENTS

I first want to acknowledge my advisor, José Verschae. He showed me how to face the difficulties of the scientific process with hope and perseverance. Creating new knowledge is hard and slow. However, José always made it enjoyable and gave me confidence when I doubted myself. I am also grateful for his patience, teaching spirit, and all the time he devoted to our research. I have not yet met any other Master's or Ph.D. students with an advisor so invested in his work. Without him, this work would not have been possible. However, what stands out the most about him is his human quality.

I want to thank the people who have been with me on a personal level all this time. First, to my family and especially to my parents Carola and Andrés. They have supported me in every decision and I will always be grateful for their confidence in me. Second, to Cata. Thank you for your unconditional affection, kindness and for always being there for me. Lastly, to my grandfather Moncho. With joy, wisdom, and without ever losing his unique essence, he has been an inspiration to me in the fields of science and culture.

Finally, I want to thank all the people who have accompanied me on this journey. My dear friends from Chache, those who have given me infinite joy and laughter. To all the teachers I have encountered for their passion, inspiration, and wisdom. To all the people from the Institute of Mathematical Engineering, for creating a magical place in the university. Finally, to the Pontifical Catholic University of Chile, which gave me a space where I could grow as a person in every aspect.

This thesis was partially funded by ANID - Fondecyt Nr. 1221460 and ANID - Millennium Science Initiative Program - MIDAS - NCN17_059

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This work addresses the challenge of identifying potential new COVID-19 outbreaks through city wastewater networks. In practical situations, we may have impartial information from the network. In particular, a number of unidentified pipelines appearing in the data do not carry flow in reality. We consider the problem of finding new outbreaks by performing a sequence of adaptive queries on the nodes of a rooted directed acyclic graph, which models the network. A query determines whether the infected node is sending contaminated water through the queried node, assuming the water flows through an unknown directed in-tree. Our model is robust to any realization of the tree. We show that the problem is NP-hard when trying to minimize the worst-case number of queries.

We then present some characteristics of the optimal solution, including properties and lower bounds for the average and worst case scenarios. We show that the treewidth of the graph is not a lower bound of the optimal search strategy height. We introduce three heuristic algorithms: Entropy, Path Separation, and Greedy. Our algorithms perform well in real and simulated instances. The Entropy Algorithm provides solutions that are close to the optimal solution for the case without uncertainty. However, its running time may be exponential on the number of different paths of the graph. The Path Separation Algorithm has slightly worse results, but it can be implemented in polynomial time.

Finally, we connect the problem to the Group Testing Framework. We present an optimal algorithm for the case with no restrictions on the size of the tested set, as well as a 5-approximation algorithm for the case with the size bounded by the separator number multiplied by the in-degree of the graph. Our findings have implications in the areas of Group Testing, Operations Research, Algorithm Design, and Wastewater-based Epidemiology.

Keywords: Combinatorial optimization, Search strategies, Group testing, Heuristic Algorithms, Treewidth, NP-Completeness.

RESUMEN

Abordamos el desafío de identificar nuevos brotes de COVID-19 a través de la red de aguas residuales de una ciudad. Es posible que tengamos información parcial de la red. En particular, una serie de tuberías no identificables que aparecen en los datos no transportan flujo en la realidad. Resolvemos el problema de encontrar nuevos brotes realizando una secuencia de consultas adaptativas en los nodos de un grafo acíclico dirigido, el cual modela la red. Una consulta determina si el nodo infectado envía agua contaminada a través del nodo consultado, asumiendo que el agua fluye a través de un árbol dirigido desconocido. Nuestro modelo es robusto ante cualquier realización del árbol. Mostramos que el problema es NP-difícil cuando se intenta minimizar el número de consultas en el peor caso.

Luego presentamos algunas características de la solución óptima, incluyendo propiedades y cotas inferiores para los escenarios promedio y peor caso. Demostramos que el *treewidth* del grafo no es una cota inferior de la altura de la estrategia de búsqueda óptima. Para resolver el problema computacionalmente, presentamos tres algoritmos heurísticos. El Algoritmo de Entropía proporciona soluciones efectivas cercanas a la solución óptima para el caso sin incertidumbre. Sin embargo, su tiempo de ejecución puede ser exponencial en el número de diferentes caminos del grafo. El Algoritmo de Separación de Caminos retorna resultados ligeramente peores, pero se puede implementar en tiempo polinomial.

Finalmente, conectamos con el marco de *Group Testing*. Damos un algoritmo óptimo para el caso sin restricciones en el tamaño del conjunto a testear, así como una 5-aproximación para cuando este tamaño está acotado por el número separador del grafo multiplicado por el grado de entrada. Nuestros hallazgos tienen impacto en las áreas de *Group Testing*, investigación de operaciones, diseño de algoritmos y epidemiología en aguas residuales.

Palabras Claves: Optimización Combinatorial, Estrategias de Búsqueda, Testeo Grupal, *Treewidth*, NP-Completitud.

1. INTRODUCTION

1.1. Introduction

1.1.1. Problem motivation

During the COVID-19 pandemic in 2019, governments around the world faced the problem of detecting new outbreaks of the virus quickly and reliably. In this context, new alternatives appeared, such as performing PCR tests on the sewage network of a community. The idea behind this approach was to test a group of individuals simultaneously rather than each person on its own. Assuming that we can have a reliable estimation of how many people are infected using this kind of test, this allows us to monitor the difference in the number of affected people and trace the new outbreaks to a bounded zone of the city. Given the results on the first iteration, it is then possible to apply again the same process and bound the zone of the new outbreaks iteratively.

In a more precise way of looking at the problem, we can represent the sewage system as a network. Nodes represent manholes, and edges represent the pipelines that carry the flow between manholes. The edges have directions as all flows are directed toward a Water Treatment Plant (WTP). The WTP acts as the root of the network and receives all the flow from the community.

However, multiple challenges must be faced to implement the described system. The first is that the information representing the network may be corrupted. We will model our problem by considering a network that has a number of extra pipelines that cannot be identified. As we want our model to be useful in any scenario, we present a robust model against all possible realizations of the real network. This situation was observed in the real instance that we worked on. The place is San Pedro de la Paz, located in the Bío-Bío Region in Chile.

A second difficulty is that our model has to address the problem of locating new outbreaks of infection testing on a limited number of nodes per day, which represent the PCR

tests on the manholes. The exact value of how many tests can be performed is determined by the resources available to decision makers. This means that the number of samples is an input of the problem and it is essentially a different scenario for every number of daily tests.

A third difficulty is that the placement of the tests must not only take into account the size of the network to look for the next day. A natural approach would be to try to divide the network into components of similar sizes. However, algorithms also have to consider the *complexity* of each subnetwork. This is relevant, given that networks of the same size can be completely different in terms of the number of days that we expect to find the new outbreak. As our goal is to reduce the number of days until we find the newly infected zone, we can try to minimize the expected number of days or the maximum number of days for any node. Again, the selection depends on the decision makers. We will study both problems.

In practical applications, we may have more difficulties. An example of this can be to have a reliable estimate of the number of infected people given a sample. However, in this thesis we will not study this issue and we will focus mainly on the above-mentioned.

1.1.2. Related bibliography

The method of studying the sewer network of a location to detect new virus outbreaks is not new (Kokkinos, Ziros, Mpalasopoulou, Galanis, & Vantarakis, 2011), but has recently received much attention. See, for example, the works in the context of the COVID-19 pandemic by Baldovin et al. (Baldovin et al., 2021), Prado et al. (Prado et al., 2020), La Rosa et al. (La Rosa et al., 2020), and Ahmed et al. (Ahmed et al., 2020), among many others. In these works, the problem is usually seen from an epidemic point of view.

Similar formulations of the proposed problem have also been studied by the Computer Science community. The abstract problem is very related to searching in partially ordered sets (posets). Posets can be arranged in such a way that for certain pairs of elements,

one precedes the other. However, not all pairs can be compared. All finite posets can be represented as directed acyclic graphs. The problem was first introduced by Linial and Saks (Linial & Saks, 1985). Later, Carmo et al. showed that finding an optimal search strategy for posets is NP-hard (Carmo, Donadelli, Kohayakawa, & Laber, 2004). Although related to our problem, searching in general posets is not useful for our needs. The graph to which we have access is a directed acyclic graph, which is a poset. However, we are actually searching in an unknown tree, not in a poset. This means that we do not have access to the actual arrangement of the nodes.

Closer to our requirements is the problem of binary search in trees. This is a particular case of searching in posets. Its objective is to identify a node of the tree using queries on the edges that reveal which component is the node. The model was first formulated by Ben-Asher et al. They also gave an algorithm for computing the optimal strategy for searching a tree in the worst case with a running time of $\mathcal{O}(n^4 \log^3(n))$, with n as the number of nodes of the graph (Ben-Asher, Farchi, & Newman, 1999). Later, Onak and Parys studied it as a generalization of the classical binary search and presented a $\mathcal{O}(n^3)$ algorithm to find the optimal search strategy for the worst case scenario (Onak & Parys, 2006). Note that the model we study generalizes the binary search problem for trees. Later, Dereniowski proved that the problem of querying nodes in a tree is equivalent to finding an edge ranking. This implies the existence of a linear-time algorithm to search trees (Dereniowski, 2008). For his part, Mozes et al. presented a linear-time algorithm to find the optimal strategy for trees in the worst case (Mozes, Onak, & Weimann, 2008). This means that, essentially, the search problem is solved in the worst case scenario for trees.

On the other hand, the problem aimed at minimizing the average number of queries for searching in trees has been shown to be NP-complete by Cicalese et al. (Cicalese, Jacobs, Laber, & Molinaro, 2011). They gave a complete characterization of the complexity of the problem on the basis of the graph diameter. In particular, they showed that the problem of searching in trees for the average case is NP-complete even for the class of trees with

diameter at most 4. In a following article, the same group presented a natural greedy algorithm that achieves a 1.62-approximation (Cicalese, Jacobs, Laber, & Molinaro, 2014). Each iteration of this algorithm aims to divide the number of nodes in the two induced components of size as evenly as possible. That is, it selects a node that minimizes the maximum number between the size of its subtree and the size of the graph minus the subtree. Given the simplicity of the algorithm, the result is very useful from a practical and theoretical point of view.

The problem of searching in trees and partially ordered sets is very general. Given this characteristic, it finds multiple applications. For example, it can be used to model the problem of identifying the faulty component of software (Onak & Parys, 2006), searching databases (Bentley, 1979), and searching for holes in an oil pipeline (Lipman & Abrahams, 1995), among others.

Searching for infected nodes is also related to the area of Group Testing. This framework allows us to test on sets of multiple elements and get a unique answer for all of them. The answer is positive if there is at least one element infected in the set, and negative otherwise. In this context, similar formulations have been studied. For example, Harvey et al. have studied the non-adaptive problem of minimizing the number of parallel probes to detect a failure in an optical network (Harvey, Patrascu, Wen, Yekhanin, & Chan, 2007). This work led to a series of studies in graph-constrained group testing, in which the tested sets must satisfy conditions based on a graph structure (Cheraghchi, Karbasi, Mohajer, & Saligrama, 2012; Sihag, Tajer, & Mitra, 2021; Karbasi & Zadimoghaddam, 2012). However, none of this work has considered the case with an unknown tree contained in a graph structure.

1.1.3. Contribution

In this work, we model the problem of finding a COVID-19 infection in the sewage network as finding an infected node on a directed acyclic graph (DAG). To the best of our

knowledge, the model with uncertainty on the actual underlying edges of the tree has not been investigated. We study the issue from a theoretical and experimental point of view.

We first describe the difficulty of the problem for the worst case scenario. To do that, we first show that the problem is NP-hard. Then we disprove that the treewidth and separator number are lower bounds by studying the case of the hypercube in $\mathbb{R}^d$. Both values are structural parameters of the graph that were natural candidates to be lower bounds, as they help to quantify how similar a graph is to a tree. A new parameter, the testing separator number, is introduced. This value acts as a natural lower bound.

We then give three heuristic approaches. The first algorithm tries to separate the graph into zones with a balanced number of nodes. The second divides the number of potentially infected paths as evenly as possible. The third one aims to minimize the informational entropy of the potentially infected nodes of the graph after a query. The first two methods generalize a 1.62 approximation algorithm for the average case problem without uncertainty. Running all of them in real and artificial networks gives results very close to optimal in the perfect information case. To use artificial networks, we introduce a methodology to create plausible random DAGs that mimic the structure of a sewage network based on the topology of a city.

Finally, we study the problem in light of the Group Testing framework. Given that there is a path of infected nodes and that we are only interested in the one that propagates the infection, the standard techniques are not useful. That means that we only care about what infected node is the lowest in the topological sense. We present an algorithm that is optimal when there are no restrictions on the size or structure of the set to test. We also show another algorithm for the size restricted case that gives guarantees close to the theoretical optimal.

Our modeling and methods contribute to the literature on Group Testing, Operations Research, Algorithm Design, and Wastewater-based Epidemiology.

1.1.4. Structure of the thesis

This work is organized as follows. In Chapter 2 we give all the necessary background and set the model formulation. We also formally define a search strategy and give useful properties of those that are optimal. In Chapter 3, we study the computational complexity of the associated decision problem. In Chapter 4 we show that neither the treewidth of a graph nor its separator number are lower bounds of the height of the optimal strategy. We also define the testing separator number, a natural lower bound. In Chapter 5 we propose three heuristic approaches. In Chapter 6 we present an algorithm for generating random graphs and test the heuristics in both real and synthetic networks. In Chapter 7, we present two algorithms that work in the Group Testing framework. Finally, in Chapter 8 we analyze the results obtained through this work and discuss some possible future studies.

2. PROBLEM FORMULATION

This chapter is devoted to the definition and formulation of the necessary steps for the construction of our model. We start by giving some basic notation and definitions in Section 2.1. We formulate the model in Section 2.2 . Finally, we define a search strategy and give the characteristics of those that are optimal in Section 2.2.2.

2.1. Preliminaries

We will start this chapter by giving some basic definitions that will be used constantly throughout this work.

Definition 2.1 (Path)**.** *Given a directed graph $G = (V, E)$, a v_1v_n-path is a sequence of edges $(e_1, ..., e_n)$ for which there exists a sequence of nodes such that $e_i = (v_i, v_{i+1})$ for $i = 1, ..., n-1$. For a path P, we say $v \in P$ if there exists $i \in \{1, ..., n+1\}$ such that $u = v_i$ in the sequence of nodes of P.*

Definition 2.2 (Cycle)**.** *Given a directed graph $G = (V, E)$, a directed cycle is a path $(e_1, ..., e_n)$ with associated nodes $(v_1, ..., v_n, v_1)$. That is, the first and last nodes of the path are equal.*

Definition 2.3 (Directed Acyclic Graph (DAG))**.** *A DAG G is a directed graph without directed cycles.*

Definition 2.4 (Topological order)**.** *Given a DAG $G = (V, E)$, a topological order of G is a linear ordering of the nodes in V, such that for every $(u, v) \in E$ we find that u is before v in the ordering.*

OBSERVATION 2.1. *Consider a DAG $G = (V, E)$ and a set $S \subseteq V$. If there exists a path P in G such that $s \in P, \forall s \in S$, then there is an element of S that is the topologically lowest for any topological order of G.*

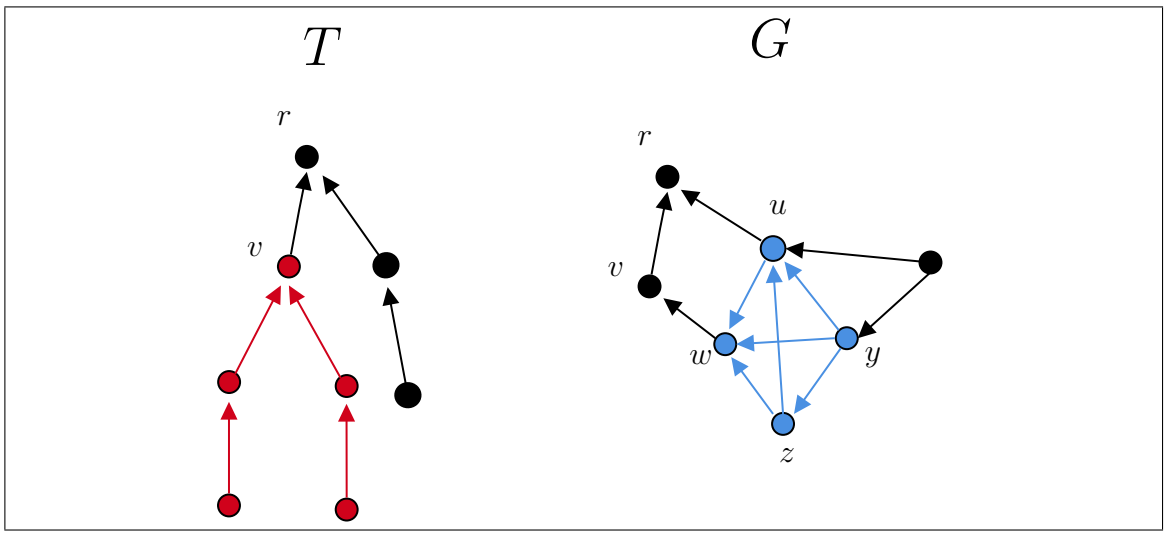


Figure 2.1. On the left an in-tree T. In red is the subtree T_v. On the right a DAG G. In blue is the induced subgraph $G[M]$ for $M = \{u, w, y, z\}$.

Definition 2.5 (Rooted DAG)**.** *A directed graph $G = (V, E)$ is a rooted DAG if it is a DAG, and there is a node r, which is called the root, to which all other nodes have a directed path. That is, for every node $v \in V \setminus \{r\}$ there exists a vr-path.*

Definition 2.6 (In-tree)**.** *A directed graph $T = (V, E)$ is an in-tree if it is a rooted DAG with root r, and for all nodes $v \in V \setminus \{r\}$ there exists exactly one vr-path.*

Definition 2.7 (Subgraph)**.** *Let $G = (V, E)$ be a directed graph and $S \subseteq V$. We define the induced subgraph $G[S]$ as follows:*

$$G[S] := (S, E'), \text{ where}$$

$$E' := \{(i, j) : (i, j) \in E \wedge i, j \in S\}.$$

Definition 2.8 (Subtree)**.** *Given an in-tree $T = (V, E)$ and a node $v \in V$, the subtree T_v represents the subgraph induced by all nodes u such that there exists a uv-path in T.*

Definition 2.9 (Predecessor)**.** *Given a rooted DAG $G = (V, E)$ and a node $v \in V$, we say that u is a predecessor of v if $(u, v) \in E$.*

Definition 2.10 (Succesor)**.** *Given a rooted DAG $G = (V, E)$ and a node $v \in V$, we say that u is a succesor of v if $(v, u) \in E$.*

2.2. Model formulation

In this section, we give all the necessary tools to formulate the model. We start by modeling the problem for a single round of queries in Section 2.2.1. In Sections 2.2.1.1 and 2.2.1.2 we discuss which nodes can be excluded based on the answers to the associated queries for the case when we are operating over a tree or a DAG.

As the sample of nodes to test in iteration j depends on the results of the queries of the samples in days $1, ..., j-1$, we may be interested in finding good strategies with respect to different metrics. For example, we may want to minimize the maximum number of days needed to find the infection in the worst case scenario with respect to the position of the infection, or the average number of days needed to find the infection over an arbitrary probability distribution over the nodes of T. We will formalize these last ideas in Section 2.2.2.

2.2.1. Discarding nodes based on a single round of queries

A wastewater network can be represented with an in-tree $T = (V, E)$ with a root r representing the Water Treatment Plant (WTP). A directed edge (u, v) represents that wastewater flow can travel from the node $u \in V$ to the node $v \in V$. By definition, every node v has a unique directed vr-path. However, reality is usually more complicated, and the company that has information on the actual network in San Pedro de la Paz had some extra edges in their records. These extra arcs are known to not be used, but it is infeasible to identify them in practice on the whole network. This means that we do not know the actual structure of T, and instead we have access to a graph $G = (V, F)$ with $E \subseteq F$. We will assume that there is exactly one infection and it can be detected on its path from the infected node v^* to the root r in the real network T.

In every iteration, we have to choose a subset of nodes $S \subseteq V$ such that $|S| = k + 1$ with $r \in S$ and $k \in \mathbb{N}$. That is, we need to select k nodes in addition to the root of the graph to test. Every node in S will be queried. For each query of a node v in S we will get a positive response if the path of infection passes through v, or a negative response in the opposite case. Note that querying nodes in the set S will always have at least one positive answer, as all infections can be detected at the root of the graph. We will denote by $q(v)$ the result of the query in node v. If $q(v) = 1$ then the test in v is positive and node v belongs to the infected path. We have $q(v) = 0$ otherwise.

Observe that, since there is an underlying tree T and an infected node v^*, there is a unique v^*r-path, which we will denote by P^*, that carries the infection. This implies that if we query any node $p \in P^*$, we will get $q(p) = 1$.

In Section 2.2.1.1 we are going to formulate the model without uncertainty. That is, we will assume that there are no extra edges, and we will search the actual network.

We are then going to generalize those ideas in Section 2.2.1.2 to consider the case with extra edges. This model will be robust against different possible realizations of the real underlying tree T.

2.2.1.1. Discarding nodes in trees

We will first assume that there are no extra edges. That is, we have complete information on the nodes and edges of $T = (V, E)$. Note that this implies that if a node v is tested and returns a negative answer (that is, $q(v) = 0$), then we can discard all nodes in T_v, as every path from them to the root passes through v by definition. On the other hand, if the node was tested positively (that is, $q(v) = 1$), then we can discard all nodes in $T - T_v$.

As we work in an in-tree, we can see that the samples in S induce a partition of T in several disjoint subtrees. More formally, let S_v be the set of samples in T_v without considering v. We define the set of nodes A_v as follows.

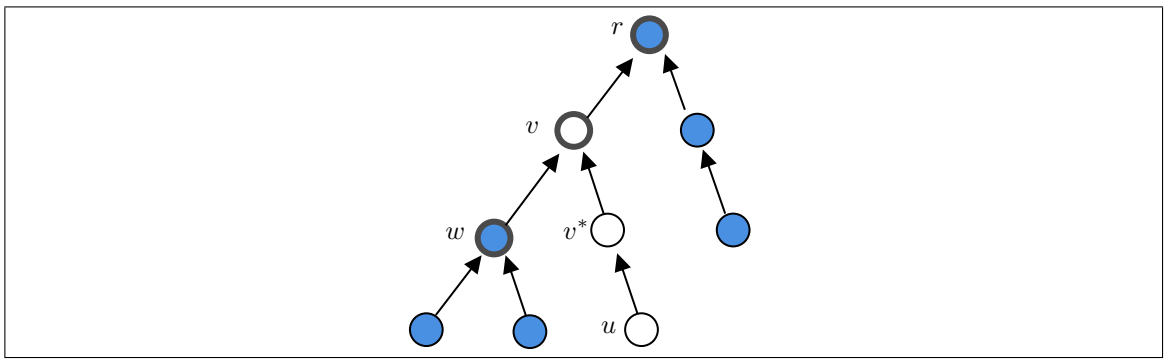


Figure 2.2. Example of an in-tree T, with an infected node v^*, and a set of sample nodes S for $k = 2$. The 3 nodes in S are in bold. The three sets in $\{A_s : s \in S\}$ are the nodes of T_w, the nodes of T_v minus the nodes of T_w, and the nodes of T_r minus the nodes of T_v. The nodes in white correspond to A_v. Notice that after sampling the set S we can be sure that the infected node belongs to A_v.

$$A_v := V(T_v) \setminus \bigcup_{u \in S_v} V(T_u)$$

with $V(T_v)$ as the set of nodes of T_v. See Figure 2.2 for an example.

We will say that $u \in S_v$ is a child of v if there is no node $w \in S_v$ such that $u \in S_w$. We denote by $C_v \subseteq S \setminus \{v\}$ the set of all children of v. With this formulation, we can then determine the number of infections in A_v, which will be denoted by i_v, in the following way,

$$i_v := q(v) - \sum_{u \in C_v} q(u).$$

As $i_v = 1$ for exactly one node $v \in S$ and $i_u = 0$, for every $u \neq v \in S$, we see that in the next iteration the search must be performed only in $T[A_v]$. This yields a problem that is analogous to the one solved in the first iteration, but now it is on a smaller tree. We will then have to select a new set S of $k + 1$ nodes, with $v \in S$. Iterating this idea for several days, we will obtain a tree with a single node and therefore recover v^*.

2.2.1.2. Discarding nodes in DAGs

Consider a DAG $G = (V, F)$ rooted in r such that there is an underlying in-tree $T = (V, E)$ with $E \subseteq F$. That is, G is the same network as T with a set of extra edges. Even if the set of edges is small, it can dramatically distort the results of the search in G, as Figure 2.3 illustrates. Given this uncertain scenario, we formulate a model that can find the infected node v^* univocally.

To be more precise about the nodes we can discard and those that we have uncertainty about whether they can or cannot be discarded, we introduce the definition of ideals and robust ideals. Informally, the ideal of a node v in G is the set of all nodes u that have at least one path to v. Similarly, the robust ideal of v in G restricted by the set S is the set of all nodes u in the ideal of v that have all their paths to v passing through S.

Definition 2.11 (Ideal)**.** *Consider a rooted DAG $G = (V, E)$. The ideal generated by v in G, is defined as*

$$I_v(G) := \{u \in V : \exists uv\text{-path in } G\}.$$

Definition 2.12 (Robust Ideal)**.** *Consider a rooted DAG $G = (V, E)$. The robust ideal generated by v in G restricted by S_0, is defined as*

$$\bar{I}_{v,S_0}(G) := \{u \in I_v(G) : \forall uv\text{-path } P \text{ in } G, \exists s \in S_0 \text{ such that } s \in P\}.$$

For simplicity and to avoid excessive notation, whenever the context allows, if we write $\bar{I}_{v,S_0}$ or I_v we will refer to the sets defined in the DAG G.

Using the above definitions, in the next lemma we will see that $I_v(G)$ constitutes an overestimation of $I_v(T) = T_v$, since every node in the subtree of T has a path to v. On the other hand, we find that $\bar{I}_{r,\{v\}}(G)$ constitutes an underestimate of $I_v(T)$.

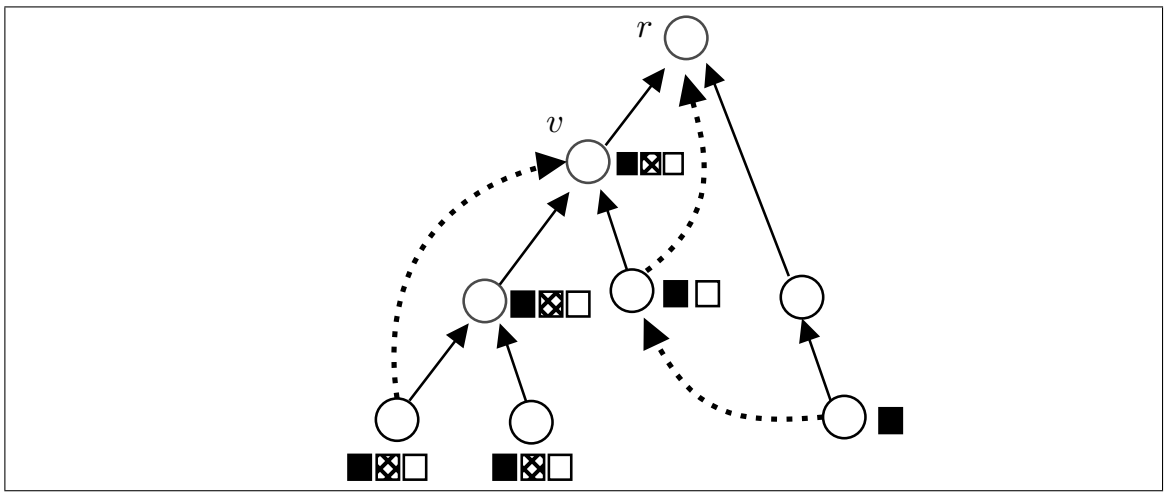


Figure 2.3. Example of a rooted DAG G with an underlying in-tree T, and sets $I_v(G)$, $I_v(T)$ and $\bar{I}_{r,\{v\}}(G)$. Edges that are not in T are shown dashed. Nodes in $I_v(G)$ are shown next to a solid-colored box, nodes in $\bar{I}_{r,\{v\}}(G)$ are shown next to a partially colored box, and nodes in $I_v(T)$ are next to a transparent box.

Lemma 2.1. *Consider a DAG $G = (V, E)$ rooted in r, then for any $v \in V$ it holds that $\bar{I}_{r,\{v\}}(G) \subseteq I_v(T) \subseteq I_v(G)$.*

PROOF. First, we prove that $\bar{I}_{r,\{v\}}(G) \subseteq I_v(T)$. By Definition 2.12 we see that every $v' \in \bar{I}_{r,\{v\}}(G)$ has all its $v'r$-paths passing through node v. Then v' is necessarily in $I_v(T)$ as the definition of the problem states that all nodes have a path to the root, including v.

To prove that $I_v(T) \subseteq I_v(G)$ is trivial, given that if $v' \in I_v(T)$, there exists a $v'v$-path. Then $v' \in I_v(G)$ by Definition 2.11. □

As in the previous section, our aim is to choose k nodes for S, which may help us to discard a set of nodes for the next iteration. Unlike searching in trees, with a negative sample, we cannot discard its entire subtree, as we do not know T_v. However, by Lemma 2.1, we can discard the robust ideal of the root restricted to the set of nodes that were tested negatively. This is valid, since every path from a node of the robust ideal to this root must have passed through this negatively tested set.

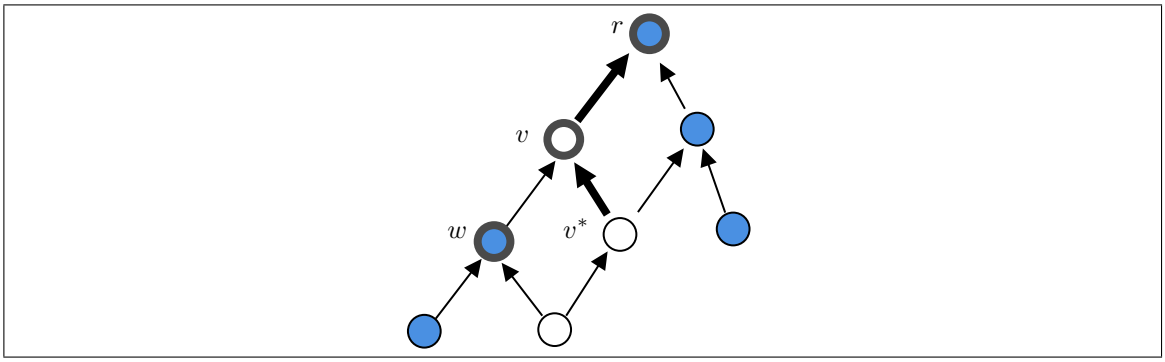


Figure 2.4. Situation for a given infected node v^* and sample set $S = \{r, v, w\}$ for the first day. The bold path denotes path P^*. The set of nodes colored white is M. That is, all blue nodes can be discarded with the information from the sample set S. Observe that the white nodes correspond to all nodes that have some path to v and do not have all paths passing through w.

Given the values of the queries in S, we aim to eliminate every node that may not be infected and construct a new set $M \subseteq V$ such that $v^* \in M$. We define S_0 as the nodes v in S such that $q(v) = 0$ and S_1 as the nodes v in S such that $q(v) = 1$. Using the fact that G is a DAG and only in the nodes of P^* can the infection be traced, as all the nodes of S_1 gave a positive response, we can always identify the lowest topological node $u^* \in S_1$ that is furthest to the root r. We then define the following set.

$$M_{S_0,S_1} := I_{u^*} \setminus \bar{I}_{u^*,S_0}.$$

Whenever there is no risk of ambiguity, we will call the set M_{S_0,S_1} just M.

Note that in $G[M]$ we find that the out-degree of u^* is 0. Then we can consider u^* as the root of the induced subgraph by M and repeat the same process on it. The following lemma will prove the correctness of the set M and allows us in the next iteration to restrain our search in $G[M]$.

Lemma 2.2. *Consider a DAG $G = (V, E)$ rooted in r, and let P^* be the infected path in G. Assume that we pick a subset $S \subseteq N$ of nodes to be sampled. Given the set $S_1 \subseteq S$ of nodes such that $q(v) = 1$ and $S_0 = S \setminus S_1$, then the infected node v^* must belong to*

$$M_{S_0,S_1} := I_{u^*} \setminus \bar{I}_{u^*,S_0}$$

where u^* is the node furthest away from r within P^* among all nodes in S_1. M is the smallest set where the infection may be. That is, for every $u \in M$ exists a ur-path P' such that $S_1 = P^* \cap S = P' \cap S$ and $S_0 = S \setminus P^* = S \setminus P'$. Furthermore, all nodes of P^* within M belong to the subpath from v^* to u^*.

PROOF. We first prove that $v^* \in M$. As $q(u^*) = 1$, then u^* must belong to P^*. Therefore, there exists a path from v^* to u^*, and thus $v^* \in I_{u^*}(G)$. Moreover, v^* cannot belong to $\bar{I}_{u^*,S_0}(G)$, as that would mean that there exists a node $u \in S_0$, such that $u \in P^*$ must contain the node u. This would imply that $q(u) = 1$, which is a contradiction. Then we have that that $v^* \in M$.

Now we will prove that M is the smallest set in which infection can occur. Take an arbitrary node $u \in M$. We will create a potentially infected ur-path P' such that $S_1 \subseteq P'$ and $S_0 \cap P' = \{\emptyset\}$. Trivially there is a u^*r-path, since by definition u^* belongs to the infected path P^*. Then we only have to prove the existence of a uu^*-path that does not intersect S_0. However, that is not hard to see, since all nodes that have every path to u^* intersected by S_0, are in $\bar{I}_{u^*,S_0}$. These nodes are not in M by the set definition. Then u has a uu^*-path that does not intersect S_0 in $G[M]$ since u belongs to I_{u^*}. Taking into account the uu^*-path and the u^*r-path we construct P'.

For the last part of the lemma, assume by contradiction that there is a node v of P^* that belongs to M and to the subpath of P^* from u^* to r. Therefore, there exists a directed path from v to u^*, but as $v \in I_{u^*}(G)$ there is also a directed path from v to u^*. This implies that G contains a directed cyclic, which contradicts that G is a DAG. □

It is not hard to see that in the case without uncertainty, that is, when $G = T$, we have $A_{u^*} = M$. This means that the method in the uncertain case generalizes the search in trees.

Note that if $k \geq 1$ we will discard at least one node per iteration. This can be easily seen as a negative answer in a single node different from the root would discard at least the tested node. A positive answer would discard at least the root. This implies that in at most $|V| - 1$ steps, we can find the infected node. The reader will notice that this bound is not useful. However, in Chapter 6 we will see that our methods work much better in practice.

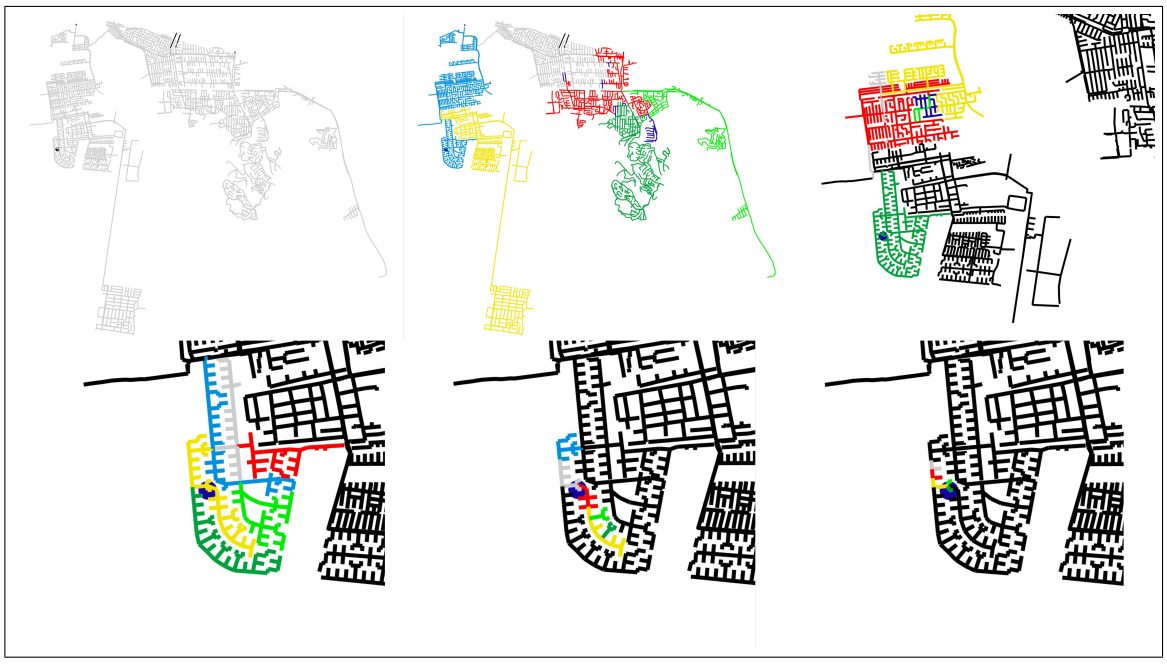


Figure 2.5. An example of a search in a graph. In the left upper picture the original graph. The dark point represents the infected node. In the next pictures are colored the induced partition for the samples and in black the discarded nodes. The narrowing of the search space is then shown.

Also, note that now the number of iterations required to find the infected node depends not only on v^*, but also on P^*. Our strategies must be robust to any choice of P^*. More

details on the search strategies will be explored in the next Section 2.2.2. An example of the search process can be seen in Figure 2.5.

Based on the result of Lemma 2.2, we give the definition of a consistent set. Informally, we say that a set M is consistent when there is a sequence of answers to queries that discard all the nodes of $V \setminus M$.

Definition 2.13 (Consistent set)**.** *Consider a DAG $G = (V, E)$ rooted in r, a set of nodes $M \subseteq V$, and k as a natural number that represents the number of samples per iteration. Let $\mathcal{P}$ be the set of directed paths from every node $v \in V$ to the root r. We say that M is consistent for the next iteration with respect to G and the path $P \in \mathcal{P}$ if there exists a set $S \subseteq V$ with $|S| \leq k$, which can be partitioned into two sets $\bar{S}_0(V)$ and $\bar{S}_1(V)$ such that $\bar{S}_1(V) \subseteq P$, $\bar{S}_0(V) \cap P = \{\emptyset\}$ and*

$$M = M_{\bar{S}_0(V), \bar{S}_1(V)} = I_{u^*} \setminus \bar{I}_{u^*, \bar{S}_0(V)},$$

with u^* as the lowest topological node of $\bar{S}_1(V)$.

We say that a set $M_n \subseteq V$ is consistent if there exist a path $P \in \mathcal{P}$ and a sequence of sets $M_0, M_1, ..., M_n$, with $M_0 = V$ such that M_i is consistent for the next iteration with respect to $G[M_{i-1}]$ and the path $P \cap M_{i-1}$ for $i = 1, ..., n$.

We define the set $S_0(M_n) := \bigcup_{i=0,...,n-1} \bar{S}_0(M_i)$. We analogously define the set $S_1(M_n)$.

Using the above definition, we may want to operate only with subsets of nodes that we can obtain after a series of queries. For that, we define the set of plausible sets as all sets that are consistent.

Definition 2.14 (Plausible sets)**.** *Given a rooted DAG $G = (V, E)$, we define the set of plausible sets $\mathcal{Q}$ in the following way,*

$$\mathcal{Q} := \{M \subseteq V : M \textit{ is a consistent set of } G\}$$

2.2.2. Search strategies

We begin this section by defining a search strategy. Informally, a search strategy is a procedure that defines which node to query based on the answers of previous tests. We will concentrate on the case with $k = 1$, as it is the most studied case in the literature. That is, we are considering one query per iteration (in addition to the root node). As every answer is positive or negative, a search strategy can be represented as a binary tree. More precisely, we define a search strategy in the following way.

Definition 2.15 (Search strategy)**.** *Given a DAG $G = (V, E)$ rooted in r and $\mathcal{P}$ as the set of directed paths from every node $v \in V$ to the root r in G, a search strategy for G is a tuple $D = (N, E', A)$, where N, E' are the vertices and edges of a binary tree and $A : N \to V$ is an assignment function. The search tree satisfies the following properties:*

- *The vertex set N is contained in the set $\{(S_0, S_1) : S_0, S_1 \subseteq V$ such that there is a path $P \in \mathcal{P}$ where $S_1 \subseteq P$ and $S_0 \cap P = \{\emptyset\}\}$. That is, each vertex of the binary tree is identified with the set of previously queried nodes, and the corresponding yes and no answers are consistent with at least one potentially infected path P.*
- *For the internal vertices $t = (S_0, S_1)$, it holds that the two children of t are $(S_0 \cup \{A(t)\}, S_1)$ and $(S_0, S_1 \cup \{A(t)\})$. In other words, a node $A(t)$ in V is queried and we either get a yes or no answer for it.*
- *For all leaves $t = (S_0, S_1)$ of the search tree, it holds that $M_{S_0,S_1} = \{v\}$. That is, a unique infected node has been identified.*
- *For all paths $P \in \mathcal{P}$ consider v as the lowest topological node of P, then there exists a leaf $t = (S_0, S_1)$ of the search tree such that $M_{S_0,S_1} = \{v\}$, $A(t) = v$, $S_1 \subseteq P$ and $S_0 \cap P = \{\emptyset\}$. That is, every path has an associated leaf that correctly identifies its corresponding infected node.*

The definition of a Search Strategy may seem a little technical. Hoping to aid the reader's intuition, we remark on some points. Each vertex of the binary tree is identified with a set of nodes that have already been tested in the original DAG, and we know which of the nodes carry or not the infection. The function A represents a mapping that assigns the vertices of the binary tree to the nodes of the original graph. Note that when the vertex represents an internal node, the assignment is a node to test on the graph. However, when the vertex of the decision tree is a leaf, the corresponding node in the graph represents the answer to the strategy for the series of queries and answers indicated by the branch.

We note that in the root of the decision tree we should not have any information about answers to queries. This means that the root should be $(\emptyset, \emptyset)$ or, equivalently, $(\emptyset, \{r\})$. Although this is not explicitly defined in Definition 2.15, it is a direct implication of the last property, as any other vertex at the root of the binary tree would discard at least one potentially infected path.

Let $\mathcal{P}$ be the set of directed paths from every node v to the root r. Let $d(D, v)$ be the function that calculates the number of arcs between a vertex v and the root of D. To calculate the cost of a search strategy in the worst case, which we will usually call its height, we only need:

$$cost(D) := \max_{v \in \text{leaves}(D)} d(D, v),$$

where leaves(D) represents the set of all leaves of the search strategy D.

Consider a probability vector $(w_P)_{P \in \mathcal{P}}$ on the paths of $\mathcal{P}$ that represents the probability that any path is the infected path, and $w(S_0, S_1)$ as the function that sums the probabilities assigned for each path $P \in \mathcal{P}$ such that $S_1 \subseteq P$ and $S_0 \cap P = \{\emptyset\}$. Then, the cost of a search strategy D in the average case is:

$$cost(D) := \sum_{t=(S_0,S_1) \in \text{leaves}(D)} d(D, v) w(S_0, S_1).$$

In this work, we will focus mainly on the worst case scenario.

Given a decision tree $D = (N, E', A)$ and an internal vertex $t = (S_0, S_1)$, we define the *left child* and the *right child* of t as $(S_0 \cup \{A(t)\}, S_1)$ and $(S_0, S_1 \cup \{A(t)\})$ respectively. Without loss of generality, when we represent a figure of a desicion tree, we assume that when the query at a node is negative, we go down to the left, and when it is positive, we go down to the right. To avoid confusion, we will use the term *node* for the input DAG (usually represented by G) and the term *vertex* for the binary tree (usually D).

Note that any search strategy must be robust to any realization of the infected path P^*. See Figure 2.6 for an example of a search strategy.

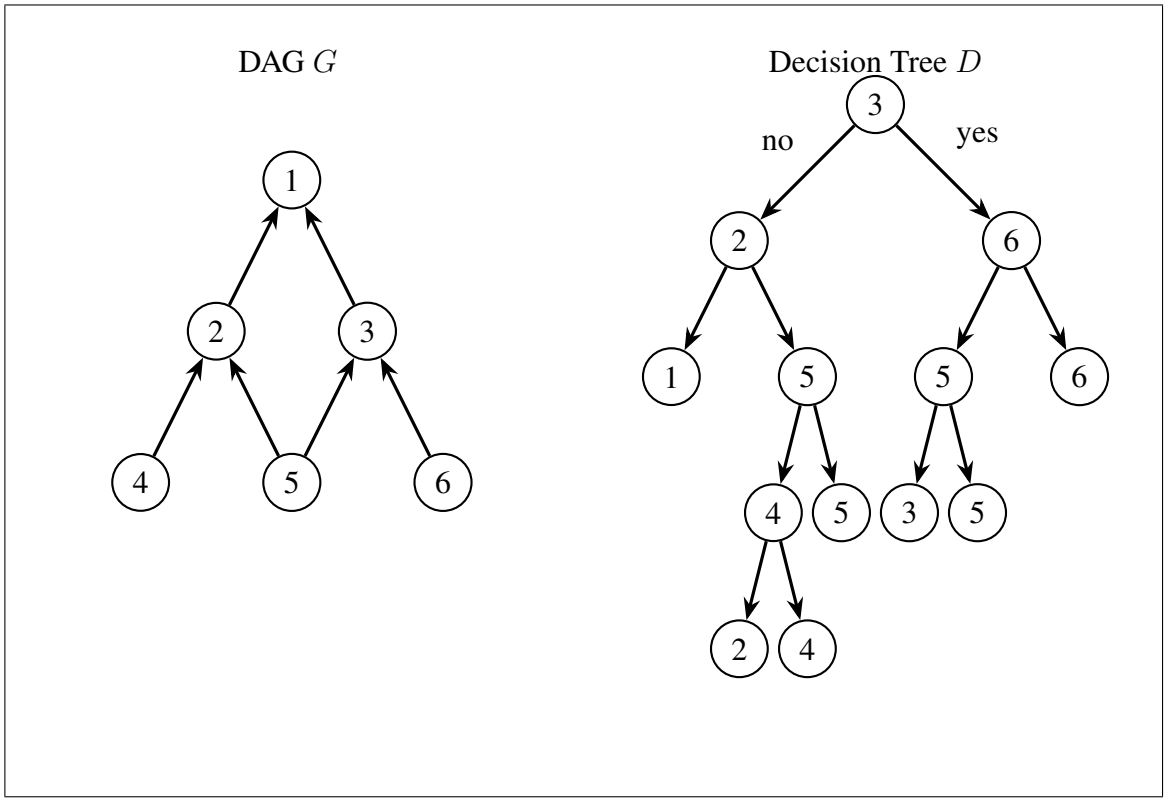


Figure 2.6. Example of a search strategy D for a DAG G.

Finally, see that in the average case, the cost of the decision tree depends on the probabilities assigned to the paths. Then, in the general case, those values may have to be in

the input. However, we may consider probabilities in implicit forms when the number of paths in $\mathcal{P}$ can be exponentially many. We define two ways to give the probabilities in an implicit form that will be used later in this work.

Definition 2.16 (Uniform distribution over paths)**.** *Consider a DAG $G = (V, E)$ rooted in r, and let $\mathcal{P}$ be the set of directed paths from every node $v \in V$ to r. We say that probabilities are uniform over paths when every path has an associated probability $\frac{1}{|\mathcal{P}|}$.*

Definition 2.17 (Uniform distribution over nodes)**.** *Consider a DAG $G = (V, E)$ rooted in r, let $\mathcal{P}$ be the set of directed paths from every node $v \in V$ to r, and $\mathcal{P}_v$ the set of all paths in $\mathcal{P}$ with node v as the lowest topological node of the path. We say that probabilities are uniform over nodes when every path that has v as the lowest topological node of the path has an associated probability $\frac{1}{|V||\mathcal{P}_v|}$.*

2.2.3. Basic properties of optimal search strategies

Using Definition 2.15, we can now list some properties of the optimal search tree. First, we make, without loss of generality, an additional assumption about the structure of optimal strategies.

ASSUMPTION 2.1 (Optimality of subtrees). *Consider a rooted DAG G and D as its optimal search strategy in the worst case scenario. We assume that for every internal vertex $t = (S_0, S_1)$ of D, the search strategy D_t is optimal in the worst case for $G[M_{S_0,S_1}]$. In other words, we assume that every subtree in the search strategy is optimal.*

This assumption, of course, does not affect the value of the desired strategy and allows us to discard search strategies that perform additional queries that give no information. Note that without the assumption, these undesirable strategies could be considered for the worst case scenario since there is only a subset of branches that define the height of the search tree.

We are now going to give two properties on the search strategies that will be useful in giving intuition to the reader.

We first show that the optimal solution of an induced subgraph is less than or equal to the optimal solution of the complete graph. For that, we first define the operations of *left deletion* and *right deletion* for a search tree D. An example of these operations is shown in Figure 2.7.

Definition 2.18 (Left (Right) deletion)**.** *Let $u \in D$, and we define left deletion of u in D as the operation that transforms D_u into D^R in D, where D^R corresponds to the right subtree of u in D. Similarly, we define right deletion.*

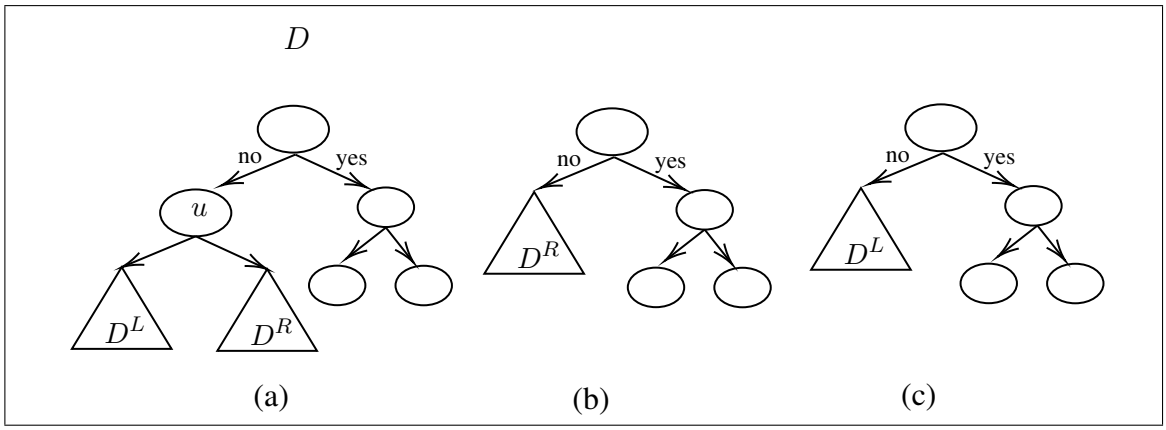


Figure 2.7. In (a) a decision tree. In (b) the decision tree after a left deletion of u. In (c) the decision tree after a right deletion of u.

For the case without uncertainty, when the studied DAG is a tree, Cicalese et al. proved that the height of the optimal decision tree for a subtree is less than or equal to the height of the strategy for the original tree (Cicalese et al., 2014). We now generalize their results to consider the case of DAGs. The argument is essentially very similar to that presented by the authors. However, we must limit ourselves to subgraphs that have been obtained through a series of queries on the graph. Informally, we start with the optimal decision tree for the entire graph. However, that binary tree has vertices assigned to nodes that are not in the induced subgraph. For these vertices, we assume an answer and replace them with the subtree associated with the assumed answer.

Lemma 2.3. *Consider a rooted DAG G and $\mathcal{Q}$ as its set of plausible sets. Then, for all $M \in \mathcal{Q}$, we have $OPT_{G[M]} \leq OPT_G$, with $OPT_{G[M]}$ as the value of the optimal decision tree, for the worst case or the average case, of the subgraph induced by the set of nodes M in G.*

PROOF. Consider an arbitrary set $M \in \mathcal{Q}$ and r as the root of G. Let u^* be the lowest topological node in $S_1(M)$, which is equivalent to being the root of $G[M]$. Consider any u^*r-path as P'. Given two paths $P_1 = (u_1, ..., u_n)$ and $P_2 = (v_1, ..., v_m)$ such that $u_n = v_1$, we define the operation of appending P_2 to P_1 as merging their corresponding sequence of nodes to obtain a new path $P_3 = (u_1, \ldots, u_n = v_1, \ldots, v_m)$.

Now we construct a new decision tree D' from the optimal decision tree D for G. We will start with D and remove the vertices that are not in M. For all the vertices assigned to $u \in V \setminus M$ in the decision tree D, we can apply a right deletion (assuming that the answer is negative) if $u \notin P'$. In the case $u \in P'$, we apply a left deletion (assuming that the answer is positive). From this we can build a search tree for $G[M]$ of equal or lower height than D. Note that for the internal vertices (S_0, S_1) of the new decision tree we may have nodes v in S_0 or S_1 such that $v \in V \setminus M$. However, we only have to consider the nodes that are in M. Then, for every vertex of the new decision tree, we will remove the nodes that are in $V \setminus M$. Formally, we map each vertex $t = (S_0, S_1)$ of the new decision tree to the vertex $t' = (S'_0, S'_1)$, with S'_0 and S'_1 as the nodes of S_0 and S_1 that belong to M, respectively. This leaves us with a new decision tree D'.

Finally, we prove that D' is a decision tree. We verify each of the four properties requested in Definition 2.15. For the first property, we note that each path P in $G[M]$ can be extended to a path in G appending P' to P. Then, since D' was constructed using operations of right and left deletion (assuming answers to queries), each vertex in D' must have at least one consistent potentially infected path.

The second property is trivially met by construction, as any vertex of D assigned to a node in $V \setminus M$ is not in D'. Then, as we removed all nodes from each set S_0 and S_1 that were not in M for each vertex, the children of each vertex met the property requested.

The third property ensures that an infected node has been identified. To verify this, suppose that for a leaf $t = (S_0, S_1)$ in D', the set M_{S_0,S_1} has a cardinality greater than 1. Then there are at least two paths P_1' and P_2' in $G[M]$ with different lowest topological nodes u and v, such that $P_i' \subseteq S_1$ and $P_i' \cap S_0 = \{\emptyset\}$ for $i = 1, 2$. P_1' and P_2' can be assigned to the paths P_1 and P_2 in G that are obtained by appending P' to them. Then u and v cannot yet be discarded as infected nodes in D at the corresponding vertex. As all deletion operations are performed in the vertex assigned to the nodes in $V \setminus M$, we see that D does not meet the property, which is a contradiction.

The last property ensures that every path has an associated leaf that correctly identifies its corresponding infected node. This can be verified directly as every path on $G[M]$ is contained in a path of G. This path can again be obtained by appending P' to each of them. Then all paths in $G[M]$ have an assosiated leaf, as D is a decision tree for G.

□

Finally, we prove that the internal vertices of the optimal decision tree are not assigned to the root of the DAG.

PROPERTY 2.1. *Let G be a DAG rooted in r, and $D = (N, E', A)$ be an optimal search strategy for G in the worst case or in the average case. Then, for any internal vertex t of D, we have $A(t) \neq r$.*

PROOF. Assume that an internal vertex v of D is assigned to r. Note that if we query in r, by the model formulation we always have $q(r) = 1$. Then we can apply a left deletion over v in D to create a new decision tree D'. This contradicts the assumption of optimality of subtrees in the search strategy. □

3. COMPUTATIONAL COMPLEXITY OF THE PROBLEM

In this chapter, we show that the problem UNCERTAIN SEARCH IN THE WORST CASE (USWC) is NP-hard. This result implies that our problem is, provably, difficult. Then we cannot solve the problem to optimality in polynomial time unless $P = NP$.

As we are trying to solve an optimization problem, to identify its difficulty we have to formulate the associated decision problem. The formal definition of the decision problem USCW is the following:

Instance: A rooted DAG G and a natural number k.

Question: Is there a decision tree of height $\leq k$ for G?

We will also use the TREE BINARY IDENTIFICATION PROBLEM (TBIP), which we formalize here. Here, we consider an in-tree, where we can query the nodes in the same way as in USWC. As before, our objective is to find a unique infected node. However, querying a node v has an associated cost $c_v \geq 0$. The objective is to find a decision tree (i.e., a search strategy) that minimizes the total cost of finding the infection in the worst case. More precisely, we consider the following decision problem.

Instance: An in-tree T, a cost value $c_v \geq 0$ for every node $v \in V(T)$, and a rational number $k \geq 0$.

Question: Is there a decision tree D for T where all the paths from the root of D to a leaf of D have a total cost $\leq k$?

In the context of TBIP, the cost of a path in the decision tree is naturally defined as the sum of the costs of all vertices queried within it. The cost of the decision tree is the maximum cost of a path from a leaf to the root, that is, the maximum cost observed in any scenario. Note that the difference with the USWC problem, where the cost, or height, of a strategy is the maximum height of a branch. That is, TBIP becomes USWC if all costs are 1.

TBIP is defined and shown to be strongly NP-complete by Cicalese et al. (Cicalese, Jacobs, Laber, & Valentim, 2012)[1]. In the original paper, the problem was defined in a different but equivalent manner. In fact, queries are taken over the edges. If a query is performed on an edge e, the answer is the connected component of $T \setminus \{e\}$ that contains the infection and a cost of $c_e \geq 0$ is incurred. However, we can simply root the tree arbitrarily over any node to give direction to the edges. Then, the query for an edge $e = (v, w)$ is equivalent to query node v in our version of the problem.

3.1. Hardness of the Problem

Theorem 3.1. UNCERTAIN SEARCH IN THE WORST CASE *is NP-hard.*

PROOF. Consider the problem TBIP when the costs are integers written in unary. Cicalese et al. (Cicalese et al., 2012) showed that this version is NP-hard. We will polynomially reduce this problem to USWC.

Consider an instance of TBIP. We will transform our in-tree T into a DAG G. For every node $v \in V(T)$ we will define in G the nodes $v_1, ..., v_{c(v)}$. That is, for a node v in T we will create as many copies of that node as the cost of v, which are polynomially many, as the costs are written in unary. Then, for every edge (u, v) in $E(T)$, we will add the edges (u_i, v_j) in G for every $i \in \{1, ..., c(u)\}, j \in \{1, ..., c(v)\}$. Note that the cost of the root $r \in V(T)$ can be assumed to be 1, since any optimal decision tree would not query the root, as shown in Chapter 2. This implies that G has a unique root. An example of the transformation can be seen in Figure 3.1. Also, note that G must be a directed acyclic graph. In fact, a directed cycle in G can be assigned to a directed cycle in T, causing a contradiction.

[1]To see that the problem is in NP we have to check a polynomial size certificate. In this case, it corresponds to the decision tree. As the graph that is searched is an in-tree, each node is queried only once in the associated decision tree. Then the decision tree has a size that is polynomial in the size of the input tree

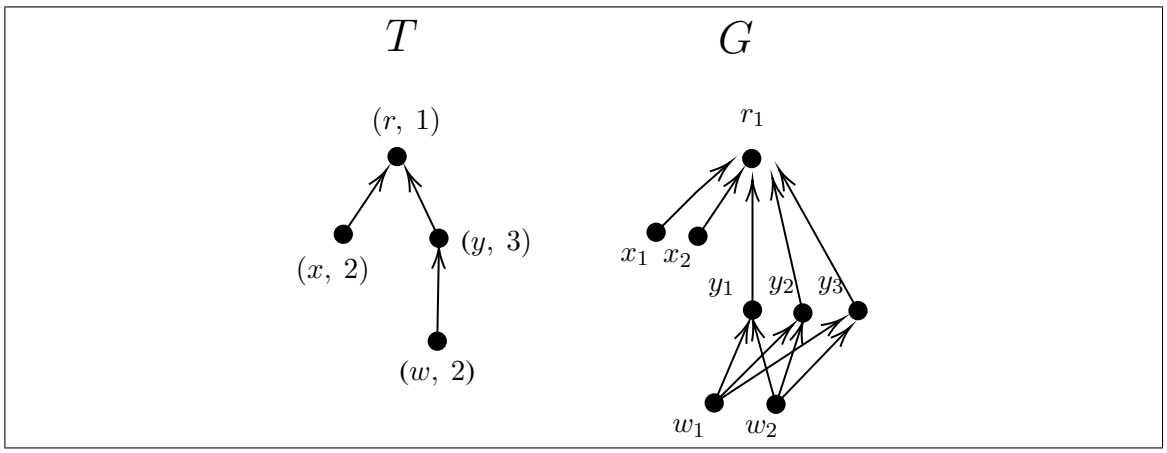


Figure 3.1. Transformation of a tree T to a DAG G: At left is the tree T. Each node has a label of the form $(v, c(v))$. At the right is the new graph G with labels on the nodes that mark its *parent*.

We will now show that the cost of an optimal decision tree for T equals the height of the optimal decision tree for G, which would imply the theorem. We split the proof into two parts.

For the first part, consider a decision tree D for T. We will transform it into a decision tree D' for G and show that the cost of D is less than or equal to the height of D'. We can take D and iteratively transform it into D', traversing it from top to bottom. Consider first that the root of D queries a vertex v, and let D_+ and D_- be the positive and negative subtrees, respectively. We replace this query with the series of queries $v_1, ..., v_{c(v)}$. To insert the list of vertices, we assume an ordering $v_1, ..., v_{c(v)}$ between them. For the first $c(v) - 1$ vertices, if the answer is positive, we will hang as a positive subtree D_+,. If it is negative, we will hang the next copy of the vertices. If all $c(v)$ queries are negative, we hang as a negative subtree D_-; see Figure 3.2. For transforming the copies of D_+ and D_- into decision trees for G, we proceed recursively as just described. Let D' be the tree obtained by this procedure. If D_+ or D_- corresponds to a single vertex, that is, we have found the infected node v, we can replace v with the unique copy of v that gave a positive response in D'. This copy exists because if in D we have ensured that v is the infected node, then we must have queried it before. In addition, we cannot have two copies that

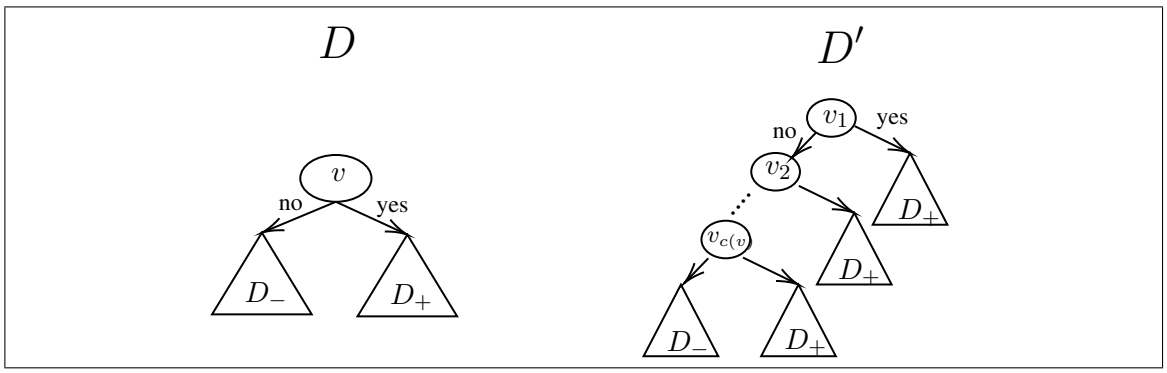


Figure 3.2. The recursive procedure that transforms D into D'

carry the infection due to the construction of G. By construction of D', a unique copy v_i of the set $v_1, \ldots, v_{c(v)}$ must have received a positive response and the answer will be assigned to that node.

Claim: D' **is a decision tree for** G. Consider any infected path $v^1_{i_1}, v^2_{i_2}, \ldots, v^\ell_{i_\ell}$, where $v^1_{i_1}$ is the infected node in G, and v^i are the corresponding nodes in T (in particular, $v^\ell = r$). Now, consider the case where v^1 is infected in T, and hence the infected path is $v^1, \ldots, v^\ell$. This infected node implies a set of queries by following the search tree D, which univocally identifies node v^1. Let us call this sequence of queried nodes $w^1, \ldots, w^k$. This implies a sequence of queries in D', where each query for w^i is replaced by a sequence of queries $w^i_1, \ldots, w^i_s$, where w^i_s is the first of these queries that is negative (otherwise $s = c(w^i)$). The construction of D' guarantees that this set of queries in D' reaches a leaf labeled $v^1_{i_1}$. Finally, note that $v^1_{i_1}$ is correctly identified in D', that is, there cannot be another infected node in G that provides consistent answers for the same set of queries. In other words, let us argue that we must have queried $v^1_{i_1}$ (and obtained a positive answer) and all its predecessors (and obtained a negative answer). Indeed, as in D we have correctly identified v^1, then within $w^1, \ldots, w^k$ we must have queried v^1 and all its predecessors, as defined in Chapter 2. By the construction of G and D', this implies that we have queried $v^1_{i_1}$ and all copies of the predecessors of v^1, which are all the predecessors of $v^1_{i_1}$. This implies our claim. Finally, to argue about the cost, consider the path of D with

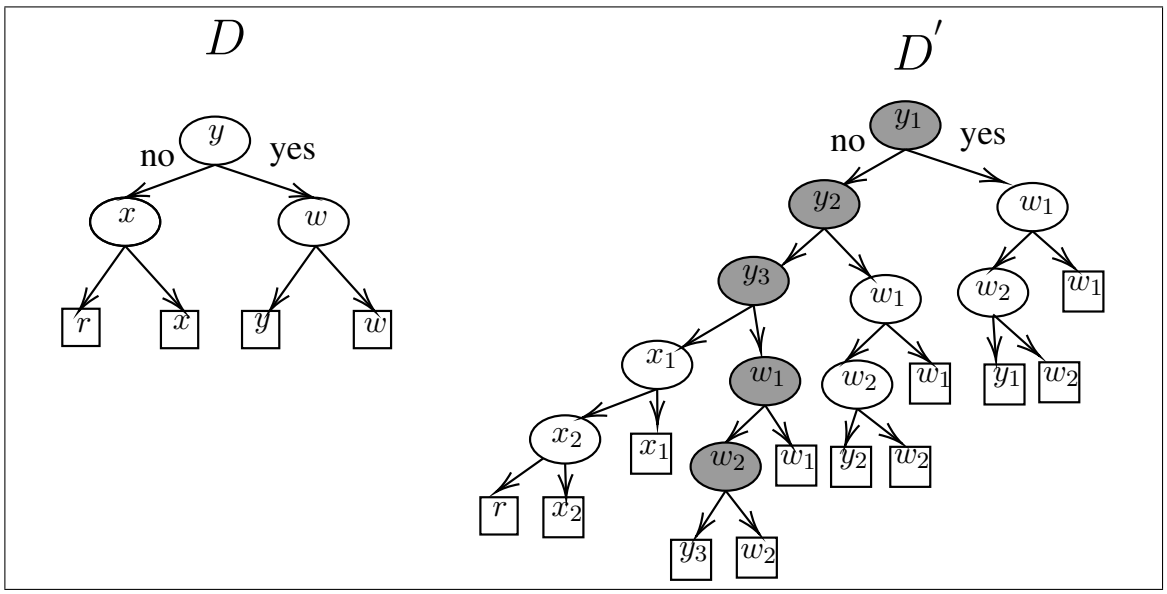


Figure 3.3. The final result of the transformation of a search strategy D for the tree T to D' for a DAG G, and vice versa if looked from right to left. In grey is the largest branch in D', which matches the value of the cost of D.

the highest cost. For any node v queried in D, we can consider the corresponding path of $c(v)$ nodes in D'. This implies that the height of D' is at least as large as the cost of D.

We now prove that a strategy in G implies a strategy in T. First, we will show that a strategy D' for G can be converted into a consistent strategy D for T. Then we will prove that the height of D' is less than or equal to the cost of D.

Consider D', an optimal search tree for G. To obtain a valid strategy for T, we again take a top-down approach. We will start at the root of D' testing the first vertex. We will use the vertices of D' to map them into nodes in T. Every vertex of the form v_i will be mapped back to the original vertex v. Now, to determine the vertex to test in case of a positive or negative response in D, we will follow the decision given by D'. That means that we will add the next mapped vertex of D' to the corresponding paths in D. If we encounter a vertex v_j after testing in v_i, we will assume a negative response in v_j and continue as indicated by D' without adding the vertex to D. This is consistent, as both v_i

and v_j map to v, which was already queried in the first encounter. This defines a search strategy D.

Let us now show that D is a valid search strategy. Considering that in G there will be at least one representative v_i from every node v, positive answers in D' will not discard any potentially infected node in T. This is true, as every node of the ideal of v_i in G is a copy of a node in the ideal of v in T. Analogously, since all nodes in the robust ideal of the root of G restricted by the set of all nodes v_i, are copies of nodes from $T - T_v$, a negative answer in D' will discard only nodes that are not potentially infected. This means that D is a consistent strategy for T. We now have to prove that when D ends a branch, that is, when it finds the infected node, the selected infected node is the only one that is consistent with the previous queries. This can be argued in a similar way to the previous claim. If we assume that there is another node s that can potentially be infected, then this would mean that there would be a copy of s in G that could not be discarded in a branch. Note that this branch in D' is the one that has the same vertices as the one we are looking for in D if we replace every positive query of a node v with v_1 and every negative query with $v_1, ..., v_{c(v)}$. This is a contradiction, as D' is an optimal search strategy in G.

To argue about the cost of both strategies, we will prove that the largest branch of D' has the same cost or less than the path with the most cumulative cost in D. As the cost of a vertex in D is paid as soon as it appears its first copy, we have to show that in the largest branch of the strategy for G, which we will denote by b, all nodes queried in T have all their respective copies in it. But this affirmation is not hard to see as a negative answer in any copy of a node can only discard exactly that same node. Within b, it is not possible to test only a fraction of copies of a node since, if that was the case, then the other copies have to be discarded with other answers in the branch. That would also discard the queried fraction. This would mean that the branch could be cut short by deleting those vertices, contradicting optimality. Then all the copies have to be queried and only the last one may have an answer that is different from a negative one. Hence, within b all copies of

a queried node must be copied. Therefore, in D, the corresponding path has a cost equal to the length of b. □

4. LOWER BOUNDS

As we have proved in Chapter 3, the problem we are studying is computationally difficult to solve to optimality. Given this, it is relevant to be able to quantify how close or far our solutions are to the optimum. The lower bounds can then act as a reference point for the quality of our approaches.

Informally, the treewidth of a graph is a structural parameter that quantifies how similar the graph is to a tree. Then, a natural conjecture was that the treewidth of a graph could act as a lower bound of the value of the optimal solution in the worst case.

In this chapter, we are going to explore how the height of an optimal solutions compares to a series of lower bounds. We start by giving some trivial bounds. Then we prove in Section 4.1 that the treewidth of a graph is not a lower bound of the optimal search strategy for the worst case scenario. Finally, in Section 4.2, we define a new parameter that acts as a natural lower bound.

CLAIM 4.1. *Consider a rooted DAG G with maximum in-degree Δ. Then Δ is a lower bound on the height of any search strategy.*

PROOF. Using the results of Lemma 2.3 we can restrict ourselves to the case of the subgraph induced by the node v_{max} with maximum in-degree of G and all its predecessors. This is valid since we may query v_{max} and obtain a positive response, and then query all predecessors of the predecessors of v_{max}, and get a negative response on all of them. Consider an adversarial player who decides the answer to the node that we are querying. Note that querying in the root does not give any information, as shown in Lemma 2.1. Then, if we test any of the predecessors of v_{max}, she can always respond with a negative answer for the first $\Delta - 1$ queries. We would then need an additional test on the Δth predecessor to determine whether the infection is in the Δth predecessor or at the root of the subgraph. □

CLAIM 4.2. *Consider a rooted DAG $G = (V, E)$. Then $\lceil \log_2(|V|) \rceil$ is a lower bound of the height of any search strategy.*

PROOF. As defined, in a search tree all nodes must have an associated leaf; then the minimum height of a binary tree with $|V|$ leaves is $\lceil \log_2(|V|) \rceil$ □

4.1. Treewidth

We begin by giving the classical definition of tree decompositions and treewidth, as stated in the original paper by Seymur and Robertson (Robertson & Seymour, 1986). These definitions are given for undirected graphs. Whenever we apply them to a directed graph, we will first remove the directions of its edges.

Definition 4.1 (Tree decomposition)**.** *A tree decomposition of an undirected graph $G = (V, E)$ is a tree $T = (I, F)$ and a family of subsets of V, $\mathcal{B} = \{B_i : i \in I\}$ such that:*

- $\bigcup_{i \in I} B_i = V$.
- $\forall (u, v) \in E$, $\exists\, B \in \mathcal{B}$ *with* $\{u, v\} \subseteq B$.
- $\forall v \in V$, *the nodes* $\{i \in I : v \in B_i\}$ *are connected in* T.

We call $\max_{i \in I}\{|B_i|\} - 1$ *the* ***width*** *of the tree decomposition.*

Definition 4.2 (Treewidth)**.** *The treewidth of a graph G is the minimum width over all tree decompositions of G.*

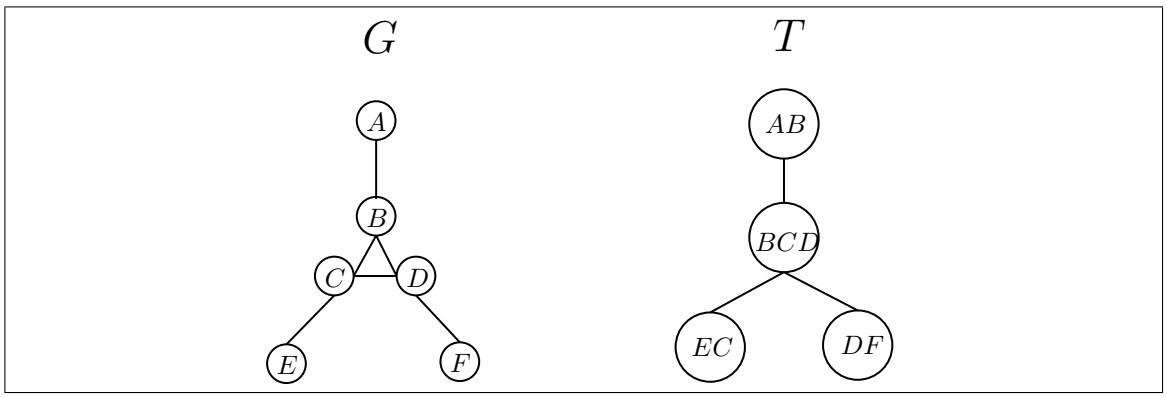


Figure 4.1. A graph G and a tree decomposition T of width 2.

We now introduce the definition of the balanced separator of a graph. This object will be important for the proof of Lemma 4.1. We also define the related value of the balanced separator number, which, along with the separator, will be useful throughout this work. Note that both the separator and treewidth are defined over undirected graphs.

Definition 4.3 (Balanced separator)**.** *Given an undirected graph $G = (V, E)$ a subset $S \subseteq V$ is called a balanced separator if after its deletion from G, all connected components have at most $\frac{1}{2}|V|$ nodes.*

Definition 4.4 (Balanced separator number)**.** *The balanced separator number of an undirected graph $G = (V, E)$, denoted by $s(G)$ (or simply s when the context allows it), is defined as the smallest integer k such that for every $Q \subseteq V$, the induced subgraph $G[Q]$ admits a balanced separator of size at most k.*

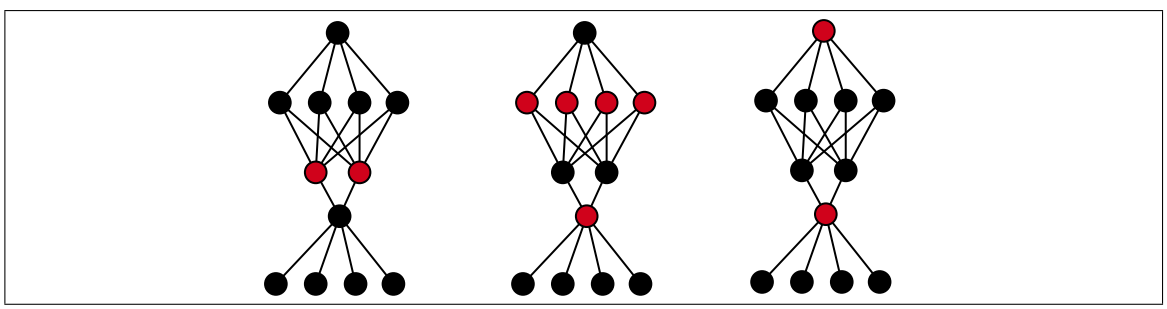


Figure 4.2. A graph with three different balanced separators marked in red.

4.1.1. A $\mathcal{O}((tw(G) + \Delta) \log_2 |V|)$ strategy

We now present a strategy that finds the infection in $\mathcal{O}((tw(G)+\Delta) \log_2(|V|))$ queries. It was presented by Rubio (Rubio, 2023). Its value lies in that it works as an upper bound for the optimal value of a search strategy in the worst case for a graph G of treewidth $tw(G)$. The idea is simple, since the algorithm finds a separator and determines which component the infection is in. We reproduce here the pseudocode.

Algorithm 1: SEPARATING(G, r)

Input: A DAG $G = (V, E)$ and its root $r \in V$

Output: The infected node v^*

$S \leftarrow$ separator of G induced by a tree decomposition ;

Query on all the nodes of S independently ;

if *All nodes in S tested negatively* **then**

  $C \leftarrow$ connected component containing r;

  return SEPARATING$(G[C], r)$

end

$u^* \leftarrow$ the topologically lowest node $s \in S$ such that $q(s) = 1$;

Test predecessor of u^*;

if *All predecessor of u^* tested negatively* **then**

  return u^*

end

$\bar{u}^* \leftarrow$ Topologically lowest predecessor u of u^* such that $q(u) = 1$;

$C \leftarrow$ connected component where is $\bar{u}^*$;

return SEPARATING$(G[C], \bar{u}^*)$

With this algorithm, we can obtain an important lemma that we will not reproduce. Details can be found in (Rubio, 2023).

Lemma 4.1. *(Rubio, 2023) Consider a rooted DAG G of treewidth $tw(G)$. There is always a search strategy of height $\mathcal{O}((tw(G) + \Delta) \log_2 |V|)$.*

4.1.2. Counterexample: The Hypercube

Motivated by Lemma 4.1, we expected that the treewidth of a graph could represent a lower bound in the height of the optimal search strategy. As we show in this section, this is not the case since a hypercube works as a counterexample of the idea. Furthermore, we will show an algorithm to find the infected node in d^2 steps for a hypercube in $\mathbb{R}^d$. The result implies that the separator number of a graph is also not a lower bound of the optimum height for a search strategy in the worst case.

First, we notice that we can represent the unit hypercube in d dimensions as a new graph that abstracts it, similar to the lattice of subsets of the set $\{1, ..., d\}$ with edges in opposite directions. We define this structure as a directed graph where each node represents the set of coordinates different from 0 within the node of the original graph. In particular, we can map each hypercube node to a lattice node using the following function $m : \{0,1\}^d \to 2^{\mathbb{N}}$ defined as $m(v) := \{i : i \in [d], v[i] = 1\}$, where $v[i]$ represents the value of the ith coordinate associated with node v. For example, the node $(0, 0, 0)$ of the unit hypercube in $\mathbb{R}^3$ is assigned to the node $\{\emptyset\}$ of the lattice, while $(0, 1, 1)$ is assigned to $\{2, 3\}$. The edges are defined by inclusion, with a difference of 1 in the cardinality of the set associated with the nodes. The direction goes from the set containing to the contained set. That is, for two nodes u and v in the unit hypercube, we have the edge (u, v) in the lattice if $m(v) \subset m(u)$ and $|m(u) \setminus m(v)| = 1$. An example of this is shown in Figure 4.3, where the hypercube in $\mathbb{R}^3$ is represented.

Lemma 4.2. *The ideal of a singleton within the lattice of a hypercube of d dimensions represents a hypercube of $d - 1$ dimensions.*

PROOF. Let H^d and H^{d-1} be hypercubes of d and $d - 1$, respectively. We will prove that there exists a one-to-one mapping between the nodes of H^{d-1} and the ideal of a singleton of H^d. Let $\{i\}$ be the singleton in H^d to consider. We denote the ideal of $\{i\}$ in H^d as $I_{\{i\}}(H^d)$. Then, we can use the function $f : I_{\{i\}}(H^d) \to V(H^{d-1})$, defined as

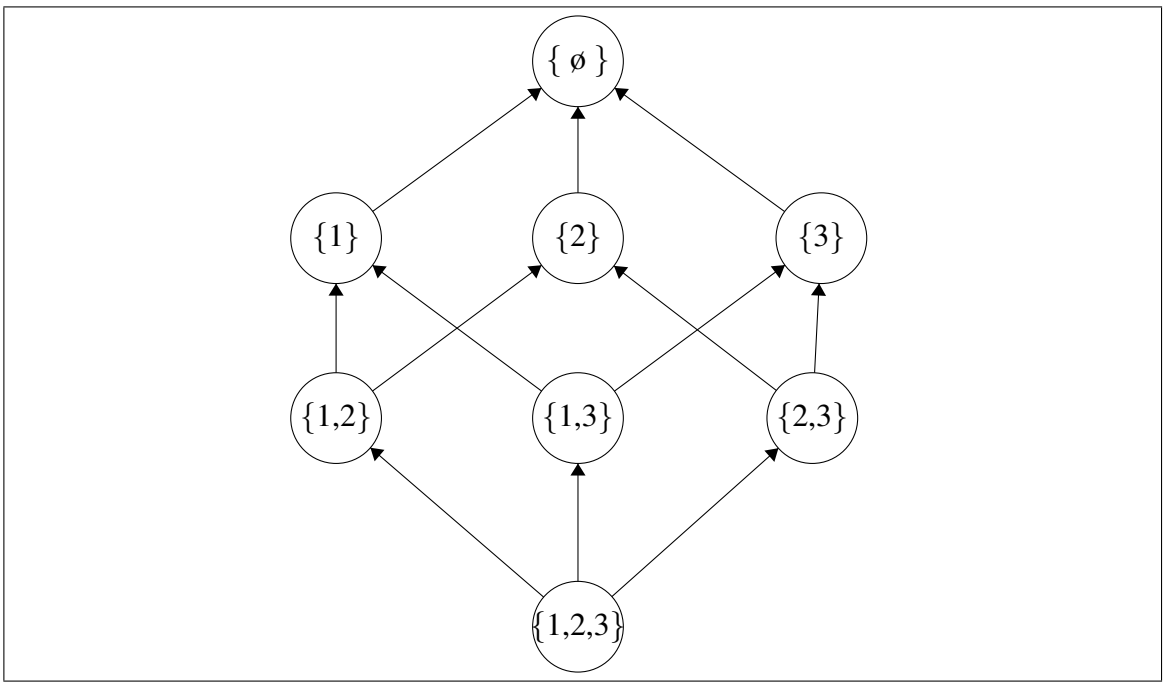


Figure 4.3. A hypercube in $\mathbb{R}^3$

$f(S_1) := S_1 \setminus \{i\}$. The function is clearly one-to-one since all nodes in the ideal of $\{i\}$ necessarily have the element i in their set. The inverse of the function can be calculated trivially by adding the node i to each of the node sets. We note that node $\{i\}$ becomes root $\{\emptyset\}$, node $\{i, j\}$ becomes node $\{j\}$, etc. Due to the construction of the lattice, we see that the same edges are maintained.

□

Using the last result, we now prove the following theorem.

Theorem 4.1. *In the hypercube of d dimensions, there exists a strategy that takes at most $\mathcal{O}(d^2)$ steps to find the infection.*

PROOF. To search in a hypercube in d dimensions, we propose the following strategy. For ease of representation, we will search on the lattice of the hypercube described above. We start by testing the first singleton. If the answer is negative, then we move on to the next. If all of them gave a negative response, then the infection is at the root. If one gives

a positive answer, then we continue testing in its ideal, which corresponds to all nodes containing the singleton. In that case, we can use the Lemma 4.2 and note that we reduce the search problem by one dimension. This is done in at most d steps. From there, we can reduce the search to a hypercube in $\mathbb{R}^{d-1}$. Therefore, we can apply the same idea. That is, this process can be done recursively, so finding the infected node would take at most $\mathcal{O}(d^2)$ steps. The theorem follows.

□

It is known that in the hypercube the minimum balanced separator size is $\Omega(\frac{2^d}{\sqrt{d}})$ (Rosenberg & Heath, 2001). From this fact we can conclude the following corollary.

Corollary 4.1. *The balanced separator number of a graph is not a lower bound for the height of the optimal search strategy in the worst case scenario.*

On the other hand, it is also known that the treewidth is greater than or equal to the balanced separator number of a graph (Gruber, 2013). As a result, we have the following corollary.

Corollary 4.2. *The treewidth is not a lower bound for the optimal search strategy.*

4.2. Testing separator number

As we have seen in the last section, the treewidth and the balanced separator number are not lower bounds for the height of the optimal search strategy in the worst case. This presents us with the challenge of finding a structural parameter of the graph that represents a natural lower bound of the optimal strategy height. As the hypercube counterexample taught us, a separator of large size can be bypassed if we test the right nodes. Inspired by this problem, we present the definition of the testing separator number. Informally, this parameter aims to be a value that represents how many queries are necessary to discard half the nodes of the graph.

We show that the testing separator number is a lower bound of the height of the optimal search strategy and leads to an algorithm that represents a logarithmic approximation.

Definition 4.5 (Testing separator number)**.** *Consider a DAG G rooted in r and the set of plausible sets $\mathcal{Q}$. For every consistent set $M \in \mathcal{Q}$, let $S_0(M)$ and $S_1(M)$ be minimal sets of query answers that fulfill $M = M_{S_0(M),S_1(M)}$. Let $\mathcal{P}$ be the set of directed paths from every node $v \in V$ to the root r, and $\mathcal{P}(M)$ be the set of paths $P \in \mathcal{P}$ such that $P \cap S_0(M) = \{\emptyset\}$ and $S_1(M) \subseteq P$.*

The testing separator number of a rooted DAG $G = (V, E)$, denoted by $ts(G)$, is defined as the smallest integer k such that for every $M \in \mathcal{Q}$, there exists a set of queries S', with $|S'| \leq k$, such that for any potentially infected path $P \in \mathcal{P}(M)$, with sets $S'_1 := \{s : s \in P \wedge s \in S'\}$ and $S'_0 := S' \setminus S'_1$, the remaining potentially infected set $M' := M_{S_0(M)\cup S'_0, S_1(M)\cup S'_1}$ satisfies $|M'| \leq |M|/2$.

In other words, for every subset obtained from a series of tests, it is possible to discard at least half of the nodes with at most k steps.

PROPOSITION 4.1. *Let $G = (V, E)$ be a rooted DAG. Then the testing separator number is a lower bound of the height of the optimal search strategy in the worst case for G.*

PROOF. Let opt be the height of the optimal search strategy of G. We can use the optimal search strategy for G to discard all nodes except the infected one. Then we have $ts(G) \leq opt$, as by Lemma 2.3 all plausible subgraphs (that is, every $M \in \mathcal{Q}$) have search trees with height not larger than opt. □

As we have seen in Theorem 4.1 and Corollary 4.2, testing a whole separator to divide the graph is not a good strategy if we want to design an approximation algorithm. Given that, we may want to use the idea behind the testing separator number. This would allow us to use the minimum number of tests to discard a constant fraction of the nodes in our graph.

Lemma 4.3. *Let $ts(G)$ be the testing separator number of a graph $G = (V, E)$. Then, there is always a strategy of height at most $ts(G)\lceil \log_2(|V|)\rceil$ to find an infected node on the graph.*

PROOF. Let S be a balanced separator of G. By definition, it is possible to discard at least half of the nodes in at most $k = ts(G)$ tests. The same process may be used on $G[M]$, where M is the set where the infection is. This idea can be applied iteratively, eliminating at least half of the nodes in each iteration with at most k steps. This implies that infection can be found in the $k\lceil \log_2(|V|)\rceil$ tests. □

Finally, we show that when the testing separator number of a graph is bounded by a constant, we can achieve a $\min\{\log(|V|), ts(G)\}$-approximation in polynomial time.

Theorem 4.2. *Consider a rooted DAG G with a constant testing separator number $ts(G)$. Then there exists a polynomial time algorithm that creates a search strategy with height at most $\mathcal{O}(ts(G)\log(|V|))$, that is, the search strategy is a $\min\{\log(|V|), ts(G)\}$-approximation.*

PROOF. Consider a rooted DAG $G = (V, E)$ with constant $ts(G)$. Using brute force, we can query all sets of at most $ts(G)$ nodes and select the first that removes half of the nodes or more. Then we can discard the fraction in at most $\mathcal{O}(|V|^{ts(G)})$ time, which is a polynomial since $ts(G)$ is constant. As the definition of the testing separator number applies to all plausible subgraphs, the idea can be applied iteratively. To verify the approximation factor of $\min\{\log(|V|), ts(G)\}$, it suffices to notice that both $ts(G)$ and $\log(|V|)$ are lower bounds, as shown in Propositions 4.1 and Claim4.2, respectively. □

Corollary 4.3. *Consider a rooted DAG G with constant treewidth and constant maximum in-degree. Then there exists a polynomial time algorithm that creates a search strategy with height at most $\mathcal{O}(ts(G)\log(|V|))$. That is, the search strategy is a $\min\{\log(|V|), ts(G)\}$-approximation.*

PROOF. Let $tw(G)$ be the treewidth of the graph, and Δ its maximum in-degree. As the treewidth and the maximum in-degree are bounded, the testing separator number is bounded by $\mathcal{O}(tw(G)) + \Delta$, which is a constant. This can be seen in the same way as Algorithm 1, we can always query all nodes in the balanced separator induced by the tree decomposition and then the predecessors of the lowest topological node that tested positively. This procedure discards at least half of the nodes of the graph. As the treewidth of a subgraph is less than or equal to the tree width of the complete graph (Bodlaender & Koster, 2011), for the subgraph in the next iteration the same conditions hold. Then by Theorem 4.2 there exists a polynomial time algorithm that creates a search strategy with height at most $\mathcal{O}(ts(G) \log(|V|))$. □

5. HEURISTIC APPROACHES

As we have seen in previous chapters, it is provably difficult to find the optimal solution for a search strategy both in the worst case and in the average case. This chapter is devoted to heuristic approaches that may help us solve the problem in real instances. We again consider the case with $k \geq 1$. We present three algorithms that aim to solve the problem using different greedy approaches. For each of them, we present the procedure for obtaining the set of nodes to test on each iteration. Once the set is selected and all nodes on it are tested, for the next iteration we can consider only the subgraph induced by all the potentially infected nodes and apply the same algorithm on it. That is, $G[M]$, with M as defined in Section 2.2.1.2.

5.1. A Natural Greedy Algorithm

Given k nodes to choose to test for finding the infection, a natural idea is to partition the graph into $k+1$ sets of elements of size as evenly as possible. This idea was explored by Cicalese et al. (Cicalese et al., 2014) for the case where $k = 1$ and with no uncertainty. For the average case scenario, consider a probability associated with each node, and w as a function that receives a subgraph and returns the sum of the probabilities associated with its nodes. They showed that in the above scenario, the algorithm that chooses to query the node v that minimizes the size of $\max\{w(T_v), w(T - T_v)\}$ in each iteration achieves a 1.62-approximation of the optimal value for the average case.

We first show that this algorithm cannot give guarantees of a constant approximation for the average case or the worst case when considering uncertainty. In this case, the natural value to minimize would be $\max\{w(G[I_v]), w(G[V \setminus \bar{I}_{r,\{v\}}])\}$ for a node $v \in V$ and with r as the root of G. In the worst case scenario we assume a uniform probability distribution over the nodes. Consider the counterexample shown in Figure 5.1 on scenario A. The Greedy Algorithm would start at the bottom and make its way up. This takes linear time and does not consider the logarithmic strategy described in the figure.

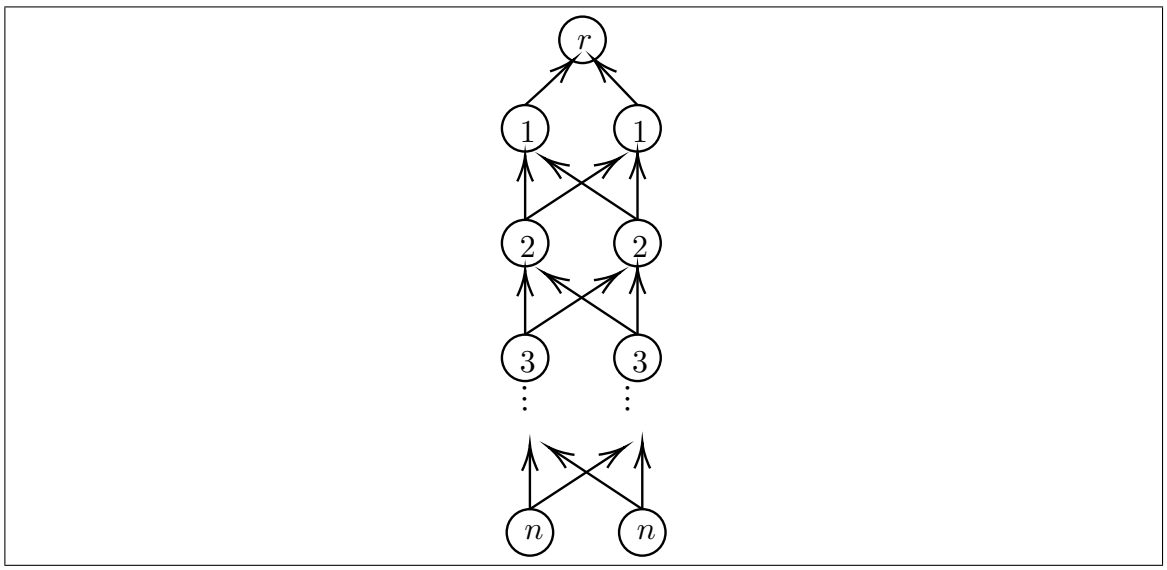


Figure 5.1. Scenario A) Each node i has a probability of being infected $1 + i\epsilon$. All paths that start at the same node have the same associated probability. Scenario B) Consider a uniform distribution on the nodes. For both scenarios, a good strategy would be to test the two nodes in the center and to perform a binary test querying if the infection is in the top or bottom half. This would take $\mathcal{O}(\log_2 |V|)$ time.

Considering that there are no guarantees on how close to the optimal solution we may be using the algorithm, we now generalize it to be able to operate over k samples and over an uncertain graph. In this case, we choose a set S of k nodes that minimizes the value $\max\{\max_{v \in S}\{w(G[I_v \setminus \bar{I}_{v,S\setminus\{v\}}])\}, w(G[V \setminus \bar{I}_{r,S}])\}$. This value represents the maximum weight of the subgraph that can be in the next iteration of the algorithm. This expression is correct since if we have more than one positive query in the set S, we only have to operate with the lowest topological node u such that $q(u) = 1$. Note that $w(G[V \setminus \bar{I}_{r,S}])$ is constant for all nodes v in S. The term represents the subgraph for the next iteration if all the samples are negative.

Although the running time is naturally $\mathcal{O}(|V|^k)$ if we enumerate all possible sets of k elements, we introduce some modifications that may help avoid that value and obtain the sample set in $\mathcal{O}(k|V|^2)$. This is done by adding a restriction that limits the maximum size

of possible partitions. For each of the elements of the sample set S, we greedily select the node that is closest to the size limit value. In the implementation, we can enumerate all possible values of the size limit $s \in \{1, \ldots, |V|\}$ of the size limit and select the one that gives the best results. We show in Appendix B that this modification in practical terms has no effect on the quality of the solutions obtained. The complete pseudocode of the procedure used can be seen in Algorithm 2.

Algorithm 2: Greedy Sampling

Input:A directed acyclic graph $G = (V, E)$ with root r, natural number k, size limit s.

Output:A set S of $k + 1$ nodes to sample with $S \ni r$.

$S \leftarrow \{r\}$;

while $|S| < k + 1$ **do**
  $aux = -1, auxv = -\infty$;
  for $v \in V \setminus S$ **do**
    $size \leftarrow w(G[I_v \setminus \bar{I}_{v,S}]$;
    if $size > auxv$ *and* $size \leq s$ **then**
      $auxv = size$;
      $aux = v$;
    end
  end
  if $aux \neq -1$ **then**
    $S \leftarrow S \cup aux$;
  end
  else
    break;
  end
end
return S;

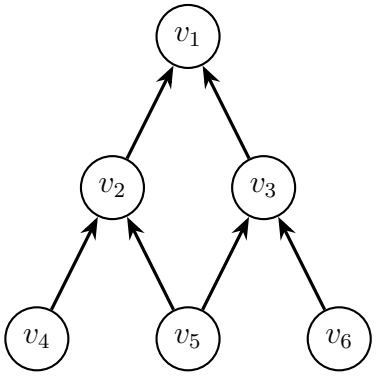
5.2. Path Separation

The next heuristic algorithm we explore tries to find more precise ways to capture uncertainty. If we consider a DAG G, we note that minimizing the number of nodes in a partition does not work well in situations where some zones of the graph are very easy to solve (for example, a path) and others are very hard (for example, a clique with directed edges). This uncertainty is better represented when we consider the number of potentially infected paths of every partition. That is, the paths that go from a node to the root of the DAG and do not pass through the nodes that were queried and their answer was negative. Given a list of paths and a set of nodes for a DAG G, we define the matrix of paths of G. There is one row per each node of G and every column is a path that is potentially infected. The (i, j) coordinate is equal to 1 if the ith node is part of the jth path and 0 otherwise. Consider the example of Figure 5.2.

We can aim to minimize the maximum number of paths that a queried tuple can induce. This is the idea that is implemented in Algorithm 3. We assume a probability distribution on the paths that represents the likelihood that any path is the infected path. That is, we receive a probability vector $(w_P)_{P\in\mathcal{P}}$, where $\mathcal{P}$ is the set of directed paths from every node v to the root r. Naturally, if we are in a setting where we do not have a natural way to define the probabilities, we assume a uniform distribution over paths, that is, $w_P = 1/|\mathcal{P}|$ for all $P \in \mathcal{P}$. For a set of paths $S \subseteq \mathcal{P}$ we denote by $w(S) = \sum_{P\in S} w_P$.

We say that an answer to a query for a set of nodes S is plausible if there is a potentially infected path that is consistent with the answers. More precisely, we denote by S_1 the set of all nodes v in S such that $q(v) = 1$. We define analogously S_0. We say that an answer to a set of nodes S is plausible if there exists a path $P \in \mathcal{P}$ with $S_1 \subseteq P$ and $S_0 \cap P = \{\emptyset\}$. Lastly, when we refer to a subgraph induced by a plausible answer, we will consider the subgraph induced by the set $I_{u^*} \setminus \bar{I}_{u^*,S_0}$, with u^* as the lowest node topologically in S_1.

A DAG G



The matrix of paths of G

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
v_1	1	1	1	1	1	1	1
v_2	1	1	0	0	0	1	0
v_3	0	0	1	1	1	0	0
v_4	1	0	0	0	0	0	0
v_5	0	1	1	0	0	0	0
v_6	0	0	0	1	0	0	0

Figure 5.2. A DAG G and its matrix of paths. The columns of the matrix M correspond to the nodes and the columns to the paths. A 1 is in the coordinate (i, j) if the node v_i is in the path p_j, and 0 otherwise.

Algorithm 3: Path Separation Algorithm

Input : A graph G and a natural number k

Output: k nodes in G to test

M $\leftarrow$ matrix of paths of G;

$S \leftarrow \emptyset$; $aux \leftarrow \infty$;

for *every* $R :=$ *subset of* k *nodes in* G **do**

  for *Every plausible answer* (R_0^i, R_1^i) *of querying* R *in* G **do**

    $u_i^* \leftarrow$ lowest topological node in R_1^i;

    $\mathcal{P}_i \leftarrow$ set of different vu_i^*-paths in subgraph induced by $I_{u_i^*} \setminus \bar{I}_{u_i^*, R_0^i}$ in M;

  end

  if $\max_i w(\mathcal{P}_i) \leq$ *aux* **then**

    $aux \leftarrow \max_i w(\mathcal{P}_i)$;

    $S \leftarrow R$;

  end

end

return S;

Consider $\mathcal{P}_v$ as the set of all paths in $\mathcal{P}$ with node v as the lowest topological node of the path. Note that in the case without uncertainty, that is, a tree T and $k = 1$, the algorithm simply tries to find a node v that minimizes the value of $\max\{w(\mathcal{P}_v), w(\mathcal{P} \setminus \mathcal{P}_v)\}$. As in a tree each path is associated with exactly one node, this implies that it generalizes the Greedy Algorithm of Cicalese et al. (Cicalese et al., 2014) and achieves a 1.62-factor approximation for the average case.

It is important to note that our problem is to find the infected node, not the infected path. In a tree, both ideas are equivalent since there is only one path to the root for every node. But when we try to separate the paths, we are solving a slightly different problem. In this case, we would actually need to assume a probability distribution on the paths consistent with that indicated by the nodes. This means that although the path separation algorithm is better suited to reduce the *complexity* of each zone of the graph given different answers, it does not directly consider the object of interest, that is, the last infected node. Due to this, it cannot give constant-factor guarantees in the average case or in the worst case. Consider again the graph in Figure 5.1, now over scenario B. The algorithm would then take a top-bottom approach. In the first iteration, a node of the first row would be queried since any answer on it would reduce the number of paths on at least one-half of the total. Then it would continue with one of the next ones, repeating the process. As described in the figure, there is a logarithmic strategy that is not used.

Another difficulty is that the number of paths may be exponential over the number of nodes. This may be a problem if we consider that the probability of infection is determined by the nodes. However, if we are in the average case scenario and we consider the model where the probabilities of infection are defined explicitly for every path, the problem becomes polynomially solvable, since this information has to be encoded in the input.

It is also relevant to mention that in the scenario with a uniform distribution over the paths, we may not have to calculate the matrix of paths. In that case, it is possible to use an algorithm that implicitly finds the number of paths that pass through each node in $\mathcal{O}(|V|^2)$

steps. Using the algorithm at most 2^k times, we can calculate all the induced partitions for every subset of k nodes. Details are presented in Algorithm 4 and an example in Figure 5.3.

Algorithm 4: Paths Hanging

Input : A graph G and root r

Output: A dictionary that indicated the number of paths hanging from each node

$dp \leftarrow \{u : 1$ for u in nodes in $G\}$;

$top \leftarrow$ topological sort of nodes in G;

for i *in top* **do**

  for j *in successors of* i *in* G **do**

    $dp[j] = dp[j] + dp[i]$

  end

end

return dp;

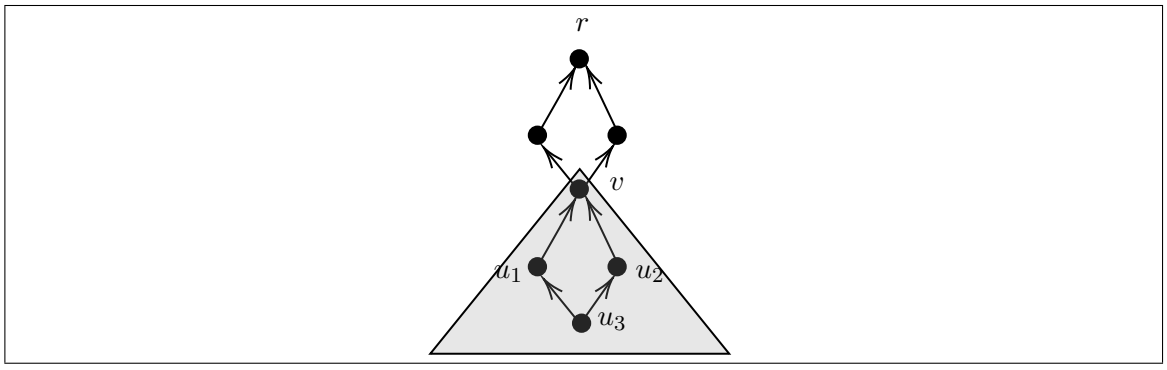


Figure 5.3. In grey the ideal of node v. Note that although the number of paths that pass through v is 10, the number of paths that hang from r is only 5. These are v, u_1v, u_2v, u_3u_2v, and u_3u_1v

5.3. Entropy minimization

Inspired by the problems of the Path Separation Algorithm, we now introduce the Entropy Minimization Algorithm. This method tries to take into account that our object

of interest is indeed the last infected node of a path, and it does not consider the rest of the infected path when comparing different alternatives to partition.

Similarly to the Path Separation Algorithm, consider a DAG G, its matrix of paths M, and $\mathcal{P}$ as the set of directed paths from every node v to the root r. We will again assume a probability vector $(w_P)_{P\in\mathcal{P}}$ on the paths of M. Now, we can partition M based on the response when querying a tuple of nodes. However, we will now not care about the complete path of infected nodes. We will only register the last infected node and its probability of infection given the associated paths. The idea of the algorithm is to reduce the complexity of the graph in the next iteration. This idea may be carried out by reducing the informational entropy of each graph and the probability that each node is infected. The intuition is that a DAG with fewer nodes or with a smaller number of paths may be easier to search.

For each response of a tuple, we can compute a list of possible infected nodes along with their probability. These probabilities can be obtained directly from the set of vectors $(w_P)_{P\in\mathcal{P}}$. We then calculate the entropy of that list. More precisely, consider a matrix of paths $\mathrm{M} = [P_1 P_2 \dots P_{|\mathcal{P}|}]$, where P_i is the ith column of the matrix (which represents a path). Let $low(P_i)$ be the function that returns the infected node on the path P_i. That is, the lowest topological node of P_i. Then we can define the entropy of a matrix of paths $\mathrm{M} = [P_1 P_2 \dots P_{|\mathcal{P}|}]$ as follows:

$$entropy(\mathrm{M}) := -\sum_{c\in V} p_c \log p_c,$$

with

$$p_c = p(c, (w_P)_{P\in\mathcal{P}}) := \sum_{P\in\mathcal{P}} w_P \mathbb{1}_{\{low(P)=c\}}.$$

The Entropy Algorithm will minimize the maximum entropy of all possible subgraphs induced by querying a set of nodes. The general procedure of the algorithm is the same

as the one of the Path Separation. The only difference is that before we were trying to minimize the maximum number of paths for the next iteration. Now, we minimize the maximum entropy for a potential graph in the next iteration. The pseudocode can be found in Algorithm 5.

Algorithm 5: Entropy Minimization Algorithm

Input : A graph G and a natural number k

Output: k nodes in G to test

M $\leftarrow$ matrix of paths of G;

$S \leftarrow \emptyset$; aux $\leftarrow \infty$;

for *every* $R :=$ *subset of* k *nodes in* G **do**

  for *Every plausible answer* (R_0^i, R_1^i) *of querying* R *in* G **do**

    $u_i^* \leftarrow$ lowest topological node in R_1^i;

    $\mathcal{P}_i \leftarrow$ set of different vu_i^*-paths in subgraph induced by $I_{u_i^*} \setminus \bar{I}_{u_i^*, R_0^i}$ in M;

  end

  if $\max_i entropy(\mathcal{P}_i) \leq$ *aux* **then**

    aux $\leftarrow \max_i entropy(\mathcal{P}_i)$;

    $S \leftarrow R$;

  end

end

return S;

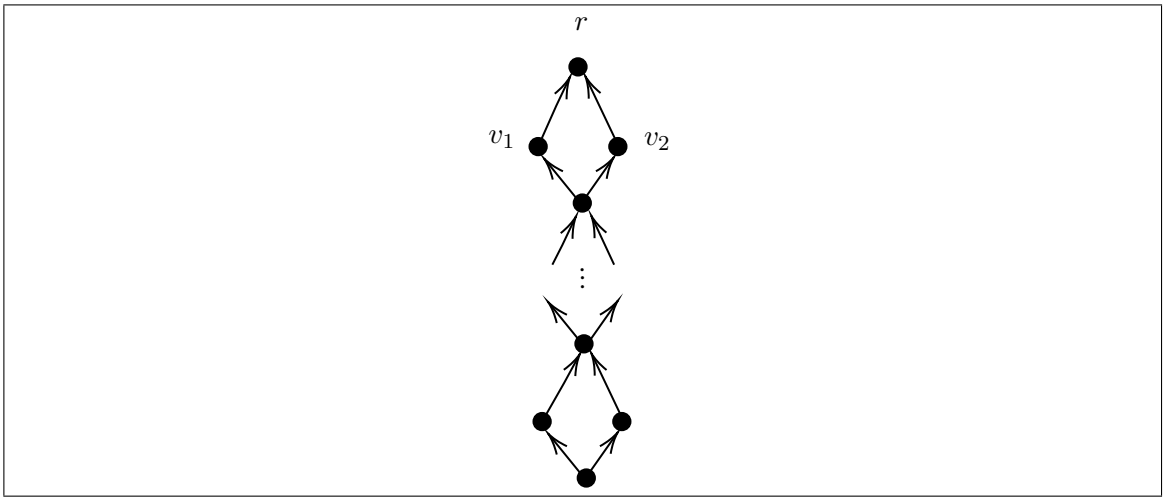


Figure 5.4. A simple instance where the Entropy Algorithm (E) achieves a $\mathcal{O}(\log_2(|V|))$ height in its decision tree and the Path Separation Algorithm (PS) achieves an $\mathcal{O}(|V|)$ height. Consider an uniform distribution over nodes (recall Definition 2.17). Then PS queries v_1 or v_2 in the first iterations and then they pass to the next level. By its part, E queries in the middle and does a binary search.

We find that when $k = 1$ and the DAG corresponds to a tree with uniform probabilities on the nodes, the algorithm performs the same queries as the one presented by Cicalese et al. (Cicalese et al., 2014). Then it achieves a 1.62 approximation guarantee in the average case for the case without uncertainty and nodes with uniform probabilities over nodes.

6. EXPERIMENTS AND RESULTS

6.1. Random graph generation

To test the performance of our method in several networks, beyond the San Pedro de la Paz instance, we develop an algorithm to generate multiple random graphs. This algorithm is designed to replicate various realistic graph properties. For example, the sewage system usually follows the topology of the road network, a property which we also observe in the graph of San Pedro de la Paz. This structure implies that the total degree of each node is typically limited to four.

We work with the data from the roadmap of an actual location to generate new graphs resembling wastewater networks. In particular, we use data from San Joaquin County (SJC) in the United States (Brinkhoff, 2002); see Figure 6.1.

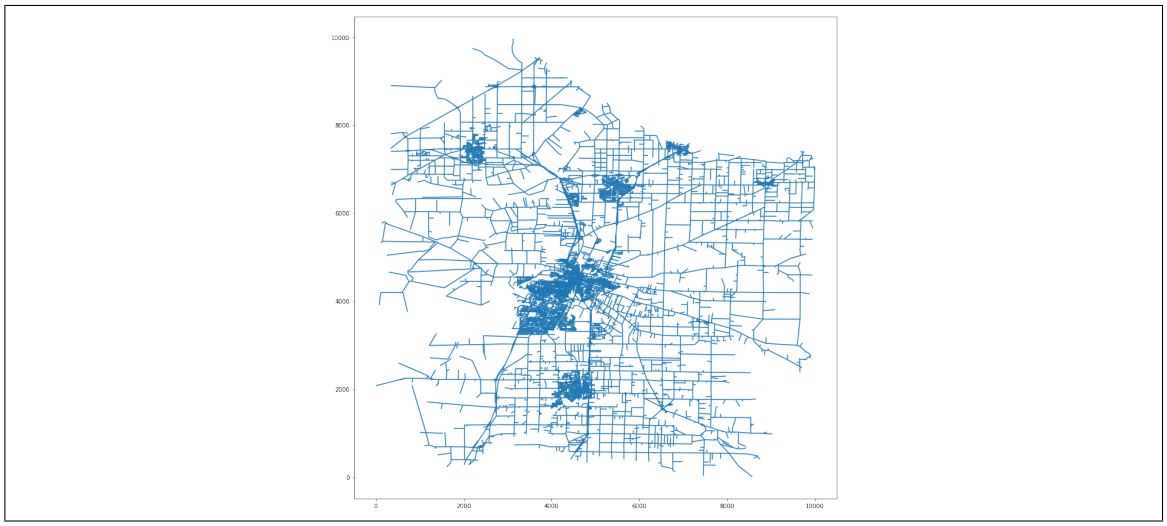


Figure 6.1. The complete road network of San Joaquin County.

Our algorithm is based on the following principles. A city has several WTPs, which are usually distributed in different sectors of the city. These WTPs induce a partition of

the sewage system. Knowing the SJC WTPs, we created random graphs to represent real sewage networks, similar to the data from San Pedro de la Paz. We propose a method consisting of two phases. In the first phase, we select one of the plants as the root of our graph. Then, we randomly choose adjacent nodes in the graph and create a directed path toward the plant. We sequentially add nodes repeating this procedure. We make sure to maintain the tree structure in this phase, which requires avoiding the creation of cycles (directed or undirected). The algorithm stops when the number of nodes in our graph reaches a pre-specified number n.

The objective of the second phase of the algorithm is to introduce noise into networks. To do so, m' random edges are added, where $m' = n \cdot C$, for some ratio of extra edges $0 \leq C \leq 1$. We add edges that do not create directed cycles. The pseudocode is presented in Algorithm 6 and examples of generated graphs can be seen in Figure 6.2.

Algorithm 6: Random Graph Generation Algorithm

Input: Graph $G = (N, E)$ with n nodes, ratio of extra edges $0 \leq C \leq 1$, edge limit e, root node r.

Output: New random graph H.

$H = (r, \emptyset)$; `// Initialize empty graph from the root with node set`

$V = \{r\}$; `// Set of visited nodes`

$Q =$ enqueue(r); `// Queue to save next nodes to visit`

while $Q \neq \emptyset$ **do**

  if *H has n nodes or more* **then**

    break;

  end

  Shuffle elements in Q randomly;

  $u = Q$.dequeue();

  for *neighbor v of u in G* **do**

    if $v \notin V$ **then**

      Add edge (u, v) to H;

      Add Q.enqueue(v);

      Add $V = V \cup \{v\}$;

    end

  end

end

$E =$ Edges remaining (in direction not generating cycles);

$E' =$ Sample of $C \cdot n$ edges from E;

for *edge $(u, v) \in E'$* **do**

  if *number of successors of node $u \in H$ in H is less than e* **then**

    Add (u, v) to H;

  end

end

return H;

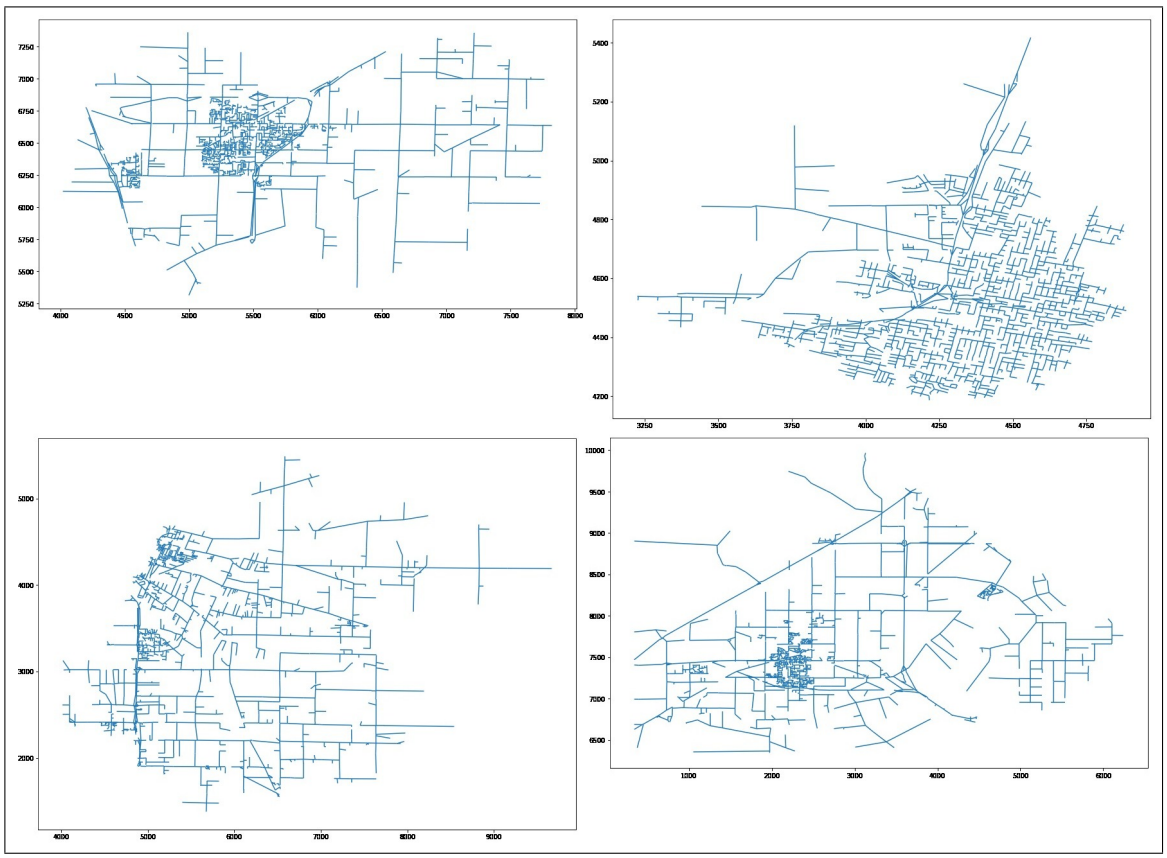


Figure 6.2. Examples of random graphs generated from the San Joaquin County network by the Algorithm 6

6.2. Simulations

In this Section, we will compare the empirical results of the search algorithms presented in Chapter 5. For all algorithms, we will consider the case where $k = 1$, as that is the case most studied in the literature and captures the difficulty of the problem. In empirical tests, when we simulated with $k > 1$ we found that the algorithms performed much better, since with more than one sample they could correct previous errors.

The instances used will be random and real. The real graphs correspond to the real data from San Pedro de la Paz, Chile. The random graphs are obtained using the method presented in Section 6.1. We will create 100 random graphs of 500 nodes each for the scenario with $0\%, 5\%, 10\%$ and 20% extra edges. For each algorithm and each scenario,

we will randomly select 100 nodes and register the mean, maximum, and 90th percentile of the number of iterations needed to find them.

The optimal value for the trees in each graph will be calculated using the optimal algorithm of Mozes et al. (Mozes et al., 2008) over a random tree contained in the graph. This value is optimal over the worst case scenario. For the Path Separation Algorithm, we will assume a uniform distribution over the paths. For the Entropy Algorithm we will assume a uniform distribution over nodes.

To register the running times, we will again create random graphs of different sizes. We will generate 10 graphs of the size $100, 200, 500, 1000$ and 2000 nodes. All of them will have 10% extra edges. For each size, we will register the mean time needed to find 100 randomly selected nodes using each algorithm.

The codes are run in Python 3 on the Networkx 3.1 and Numpy 1.24 packages. All functions and scripts can be found in the following Github repository.

6.2.1. Real-world simulation in San Pedro de la Paz, Chile

We used the real San Pedro de la Paz sewage map, located in the center-south zone of Chile. The map is presented in Figure 6.3. The associated graph has 4,606 nodes and 4,875 edges. This means that there are 270 extra edges that do not correspond to the real network and could not be identified. The maximum in-degree per node is eight, but 99.5% of them have an upper bound of four.

Figure 6.3. Sewage Network of San Pedro de la Paz, Chile.

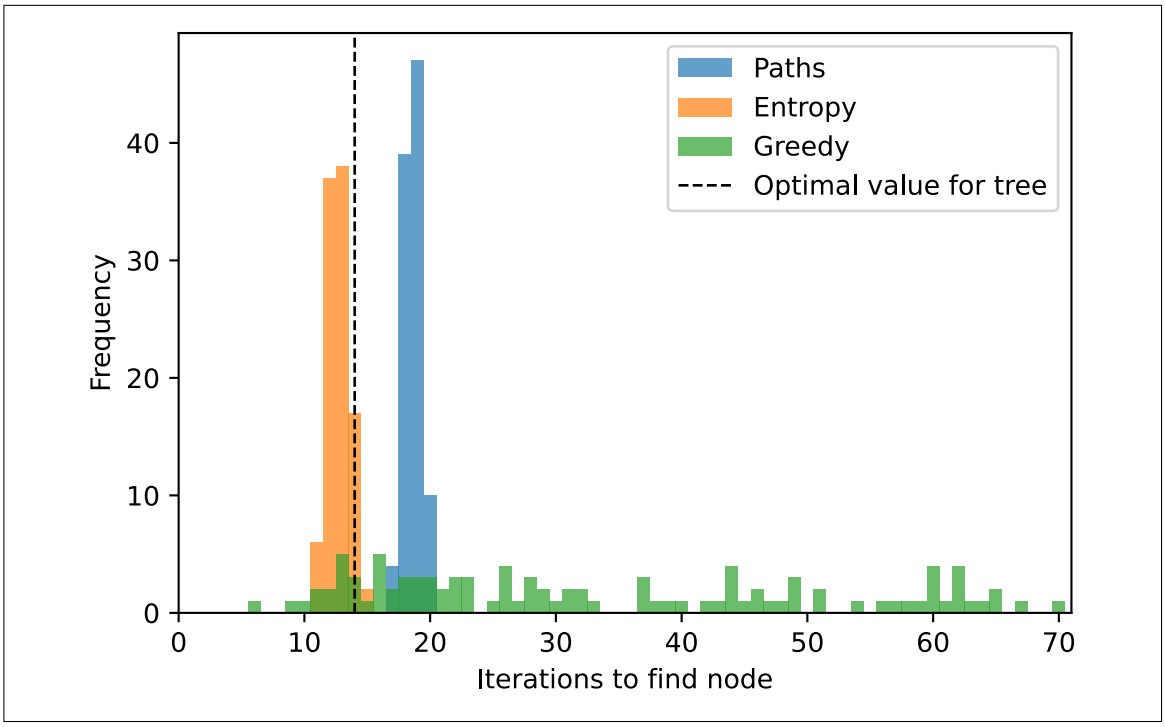


Figure 6.4. Results of simulation in San Pedro de la Paz

In Figure 6.4 it can be seen that the algorithm with the best results is Entropy Minimization. Its histogram is very compact and is better than any other algorithm studied in both the worst case and the average case. In fact, its worst case value is just one unit larger

Algorithm / Statistic	Mean	Max	90th percentile
Optimal WC Search in Tree	12.96	14	14
Greedy	33.72	70	61
Paths	18.63	20	19.1
Entropy	12.72	15	14
Treewidth	166.53	223	198

Table 6.1. Table of results of different algorithms in San Pedro de la Paz network

than the optimal value in a random subjacent tree. Also, its mean value is smaller than the mean number of iterations of the optimal algorithm in the worst case. These results are remarkable.

Similar but slightly worse results are obtained by the Path Separation Algorithm. In the worst case, its values are bounded by a factor of approximately 1.5 times the optimal value in the graph spanning tree. This is a very positive result, considering that extra edges have a great effect on the structure of the graph.

The results of the Greedy Algorithm are worse than expected, as it does not capture well the uncertainty of the graph. Its histogram is very flat and its mean value is greater than two times the optimal value for the case without uncertainty.

Lastly, the treewidth algorithm was shown to be unpractical and unreliable in the case studied. For visual purposes, its results are shown in the Appendix C. Its histogram is spread over a large area and is by far the worst algorithm compared to all simulated. We conclude that the main value of this algorithm is the upper bound that it provides from a theoretical point of view.

6.2.2. Simulation in random instances

For the simulations on random instances 100 random graphs of 500 nodes were created for each uncertainty level. For each graph, we randomly selected 100 nodes and simulated

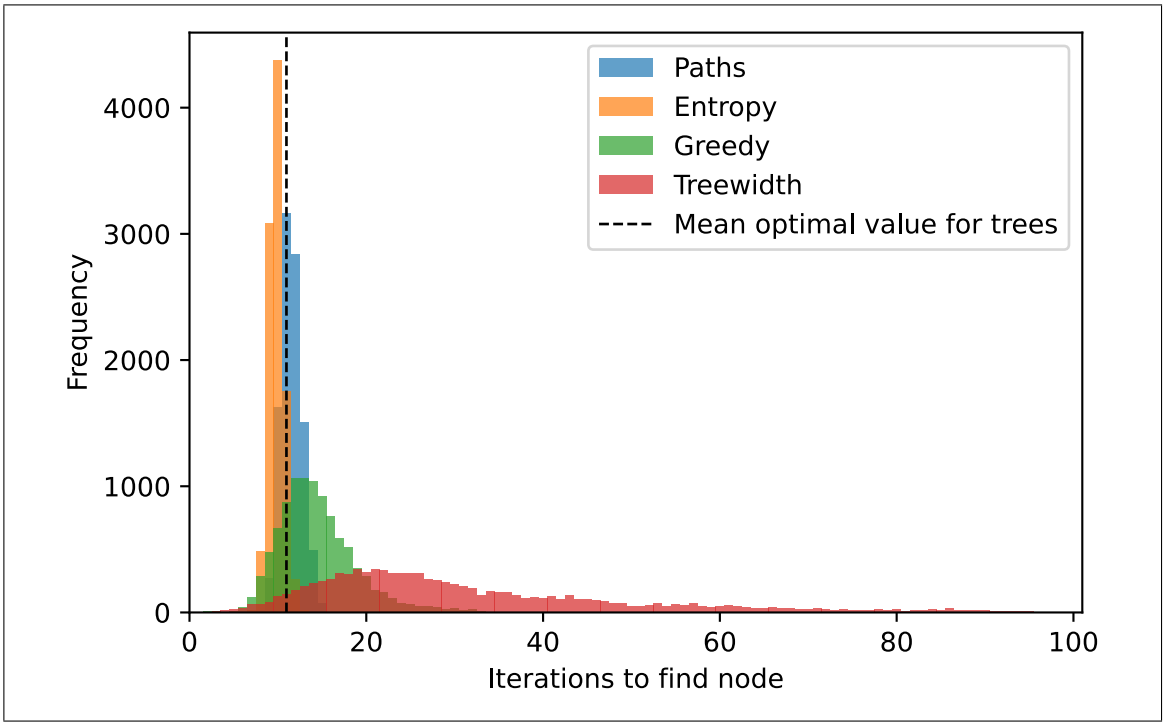


Figure 6.5. Results of simulations in 100 random graphs of 500 nodes with 20% extra edges and $k = 1$. The histograms are consistent with the results in the graph of San Pedro de la Paz. However, the Path Separation Algorithm has results very close to the Entropy Algorithm with a lower running time.

the process of finding the infection with the different algorithms. The results are presented in Table 6.2. Figure 6.5 shows the iteration histogram for the graphs with 20% extra nodes. The other histograms can be found in Appendix C.

Uncertainty	**0%**			**5%**			**10%**			**20%**		
Algorithm / Statistic	**Mean**	**Max**	**90p**	**Mean**	**Max**	**90p**	**Mean**	**Max**	**90p**	**Mean**	**Max**	**90p**
Optimal WC search	9.55	12	11	-	-	-	-	-	-	-	-	-
Greedy	11.77	20	15	12.35	28	16	12.97	37	17	14.49	46	20
Path Separation	9.13	11	10	9.64	13	10	10.26	12	11	11.54	17	13
Entropy	9.15	12	10	9.30	12	10	9.49	12	10	9.82	14	11
Treewidth	13.29	23	17	16.19	26	22	15.76	34	21	32.91	303	61

Table 6.2. Table of results of different algorithms in simulations over 100 random graphs. 90p represents the 90th percentile.

The results are excellent for the Entropy Algorithm. Even in graphs with 20% extra edges, its maximum number of iterations is only greater in two units than the maximum optimal number of iterations in the case without uncertainty. Its mean number of iterations with 10% extra edges is smaller than the mean number of iterations of the optimal algorithm in the worst case. Also, note that in the case without uncertainty and 20% extra edges, its mean number of iterations increases by approximately 0.5 units. Overall, the results of the Entropy Algorithm behave very well under different levels of uncertainty.

The Path Separation Algorithm for its parts also exhibits positive results. However, again this is slightly worse than the Entropy Algorithm. The algorithm also shows greater sensibility to changes in the uncertainty level. Note that the results with 0% uncertainty (i.e. random trees) had a mean value of 9.13 and with 20% extra edges the mean value was 11.54.

For both the Entropy Algorithm and the Path Separation Algorithm, in the case without uncertainty over the structure of the tree, their results are as good as the optimal strategy in the worst case. Also, the mean number of iterations of both algorithms is even better than the mean number of iterations of the optimal strategy in the worst case.

The Greedy Algorithm displays results that are better than in the instance of San Pedro de la Paz. The distance from its mean to the optimum in the worst case is always bounded by a factor of 1.25. Its 90th percentile behaves well and does not differ much from the mean. However, there is a tail that could not be bounded and that grows almost linearly with the level of uncertainty.

The Treewidth Algorithm also has better results in the random instances than in the real network studied of San Pedro de la Paz. In the case without uncertainty the algorithm shows similar results than the Greedy Algorithm. However, once the percentage of extra edges grows, it also increases its number of iterations in all three measured statistics. With 20% extra edges, its histogram is notorously flat and its tail very long.

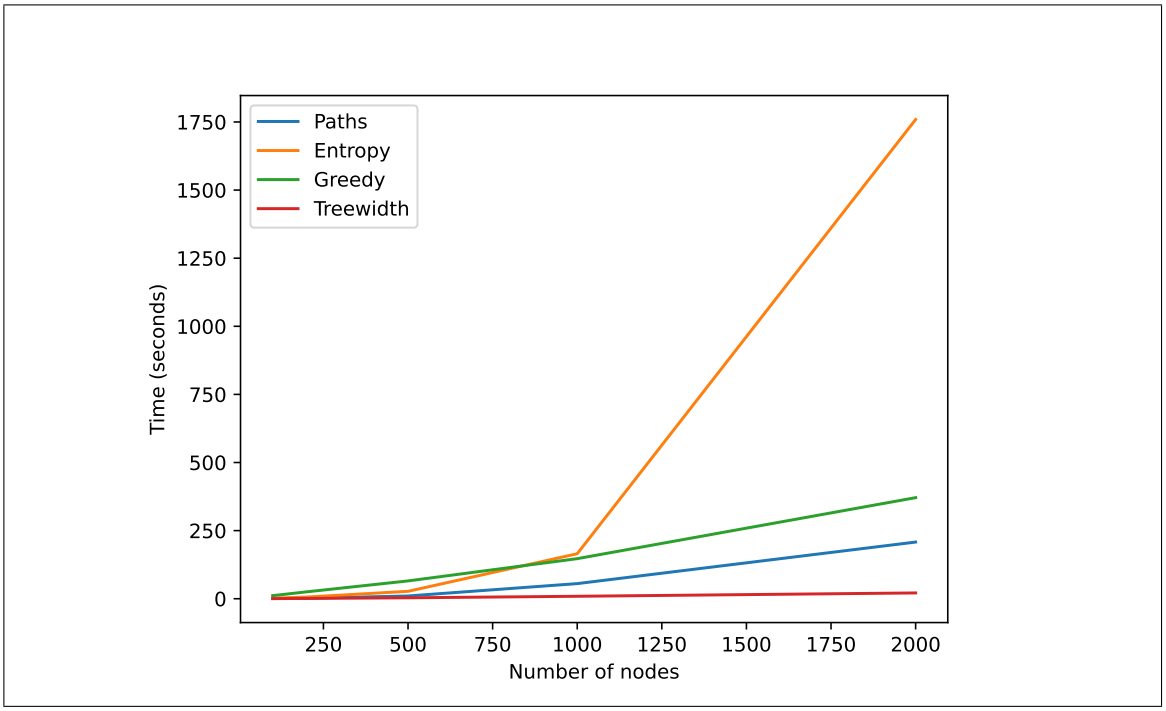


Figure 6.6. Mean registered times of executions for simulations over different graph sizes.

Finally, we analyze the execution times of the different algorithms. The Greedy, Path Separation, and Treewidth algorithms all behave well as the number of nodes of the graph increases. The three plots seem to have linear growth. For its part, the Entropy Algorithm has an exponential growth in the time needed to simulate as the number of nodes of the graph increases. The running times of all algorithms are consistent with their theoretical complexity.

7. GROUP TESTING FRAMEWORK

In this chapter, we are going to consider a slightly different scenario in which we are allowed to perform a test on multiple nodes and a series of parallel tests on each iteration. As before, the answer to the test will be a binary value. The difference is that now if the result is 0, then there are no nodes that carry the infection in the tested set. Otherwise, if the result is equal to 1, there is at least one node that may be the infected node or on its path to the root. Formally, if we test a set of nodes $S = \{s_1, s_2, ..., s_n\}$, the answer to the query is $q(S) = q(s_1) \vee q(s_2) \vee \cdots \vee q(s_n)$. This framework is known as Group Testing, which is an area of study that has a great deal of literature. Nevertheless, standard group testing techniques are not useful in this problem, as an unknown number of nodes carry the infection, and we are interested only in the last one over a topological criterion.

7.1. An optimal adaptive algorithm

This section presents an optimal algorithm for the unrestricted adaptive group testing framework. Informally, the algorithm aims to divide the graph into two groups, each of which has half of the total nodes. The first group consists of the lowest nodes topologically, and the test is applied to it on each iteration. This captures the idea that we are only interested in the last infected node of the infected path. If the test is negative, then we can discard all tested nodes, and in the next iteration we focus on the set of nodes that were not tested. With the definition of the model, we have that the infection must be located in that set. On the other hand, if the test on the set is positive, we can discard all nodes that are not on it. As we are only interested in the last infected node, we know that it has to be located on the queried set. The formalization of the idea is presented in Algorithm 7 and an example in Figure 7.1.

Algorithm 7: $GT(G, r)$

Input: $G = (V, E), r \in V$

Output: The infected node v^*

while $|V| \neq 1$ **do**

  $sort \leftarrow$ Sorted nodes of G in topological order;

  $T \leftarrow \lfloor |V|/2 \rfloor$ lowest nodes of $sort$;

  Test set T;

  if $q(T) = 0$ **then**

    $G \leftarrow G[V \setminus T]$;

  end

  else

    $G \leftarrow G[T]$;

    Add a dummy root r' and an edge from all nodes in T that do not have a successor;

  end

end

return $G = (v^*, \emptyset)$

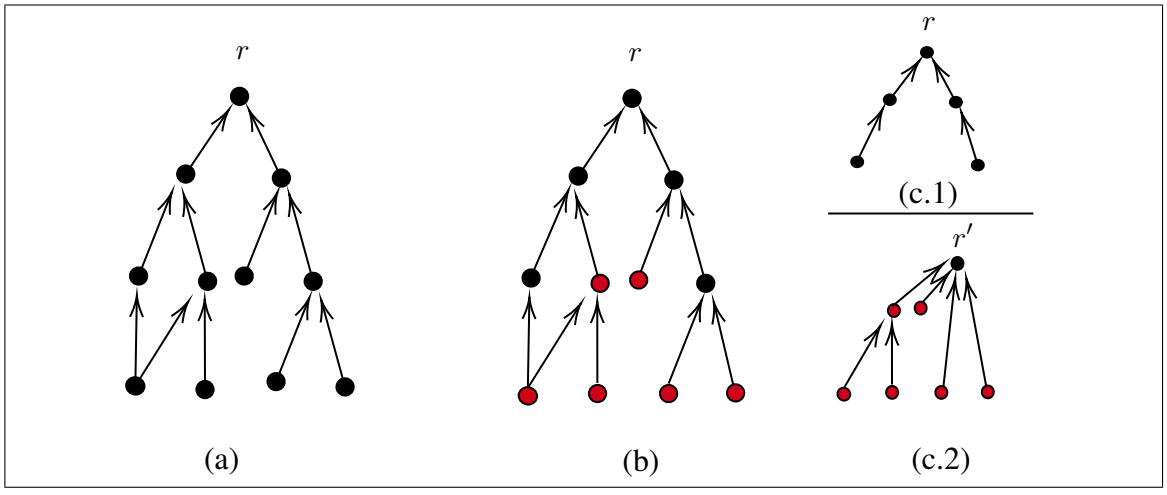


Figure 7.1. In (a) is the original graph G with its root r. On (b) is the graph with the set T to test in red. On (c.1), the graph of the next iteration if $q(T) = 0$. In (c.2), the graph for the next iteration if $q(T) = 1$ with the new *dummy* root r'.

Lemma 7.1. *Algorithm 7 is correct.*

PROOF. We will show that the algorithm does not discard any node that can be the last infected one. To do this, suppose the opposite. Then there is a node v^* that can be incorrectly removed. The node v^* cannot be in T, as it would yield $q(T) = 1$ and would remain for the next iteration. Then we must have $q(T) = 0$. But that cannot eliminate v^*, since the nodes are removed topologically for the next iteration. We have reached a contradiction. □

Lemma 7.2. *Let G be a DAG without a unique global root, and G' be a rooted version of G that is obtained by adding a dummy node r' and edges from each of the topologically greatest nodes of each set to r'. Node r' meets $q(r') = 1$. Then, querying in G' does not discard the infected node v^* in G.*

PROOF. First, consider the problem of searching for infected nodes in a graph with multiple topologically maximal nodes. Note that including the new *dummy* root r' will only modify the infected path P^* adding an additional node r' at the end of its sequence. This clearly does not modify the last infected node. Similarly, all the paths from a node v to a topologically maximal element m_i will have an extra node in their sequences.

Since we have for the dummy root r' it is always true that $q(r') = 1$, querying in r' cannot discard any path. Having a negative response in a node v can only discard nodes that have all paths intersecting v. There is clearly a one-to-one mapping between the paths in G' and the paths in G. Then an infeasible path in G' represents an infeasible path in G. The reasoning for a positive response is analogous.

□

Theorem 7.1. *The algorithm 7 is optimal and finds the infected node of $G = (V, E)$ in $\lceil \log_2(|V|) \rceil$ iterations with one test per iteration.*

PROOF. In each iteration, the algorithm reduces the number of nodes by half. The *dummy* root added does not affect the search according to Lemma 7.2. The optimality can be obtained by noting that the number of iterations matches the informational lower bound. □

7.2. An almost optimal adaptive algorithm for the size-restricted case

Now, we will consider the case where we have a restriction on the size of the set to test. In particular, we will study the case where the size of the set is bounded by the separator number of the graph multiplied by the maximum in-degree. As we have seen, the separator number is itself bounded by the treewidth, which is a more intuitive form of looking at the scenario. We will present an algorithm to find an infection on a graph $G = (V, E)$ in this framework in $5\lceil \log_2 |V| \rceil$ steps, which gives a 5-approximation algorithm. Informally, the procedure looks for a set of nodes that can act as a balanced separator of the graph. It then performs a test on the set and separately on at most three sets of nodes that are the predecessors of the nodes in the separator. There are three possible scenarios. The first is that the result of the separator is 0. Then we will discard all components except the one that contains the graph root. The second is if the answer is 1 in the separator and 0 in the rest of the predecessors. Then we will only look at the set of separator nodes in the next iterations and perform a binary search on them. Otherwise, the third scenario is if there is a set of predecessors with an answer equal to 1. We will focus on only that section of the graph for the next iteration. In the first and last cases, we would have reduced the number of nodes by at least half of the initials by the definition of the separator. In the second case, we would have to deal with the size of the separator, which should be small in real-world cases. This number is bounded above the treewidth of a graph as shown by Gruber (Gruber, 2013). Let $q(U)$ be the result of the test in U. The pseudocode is presented in the Algorithm 8.

Algorithm 8: $GT(G, r)$

Input: $G = (V, E), r \in V$

Output: The infected node v^*

while $|V| \neq 1$ **do**

Select S as a separator of G

$C_1, C_2, C_3 \leftarrow$ Partition of subsets of predecessors using the algorithm of Lemma 7.3

Test independently on C_1, C_2, C_3, S

if $t(S) = 0$ **then**

$G \leftarrow$ Subgraph induced by the partition that contains r

end

else if $q(C_1) \vee q(C_2) \vee q(C_3)$ **then**

$C \leftarrow$ Topologically lowest partition C_i such that $q(C_i) = 1$

$G \leftarrow G[C]$

if *There is more than one topologically highest node in* G **then**

Add a dummy new root r to G

Add edges from every topologically highest node in G to r

end

end

else if $q(S) = 1$ **then**

Perform a binary search on S

end

end

return $G = (v^*, \emptyset)$

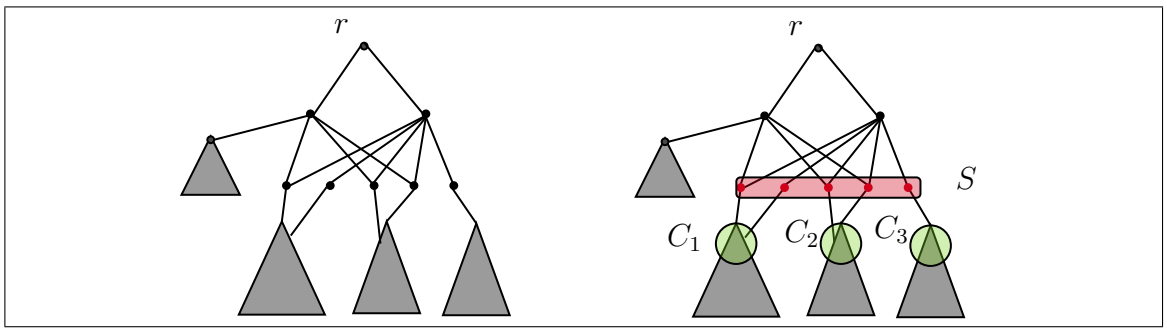


Figure 7.2. On the left is the original graph G with its root r. On the right is the graph with the separator S in red and the subset of predecessors C_1, C_2 and C_3 in green. All directions go from the lower to the upper node.

Now, we will prove a series of results that will be used to verify the correctness of the algorithm.

CLAIM 7.1. *Let S be a balanced separator of a graph and C_i be a partition of the set of predecessors of the nodes in S. If $q(S) = 0$ then $q(C_1) = q(C_2) = q(C_3) = 0$*

PROOF. Suppose not. Then there is a node v that carries the infection in C_i, for $i \in \{1, 2, 3\}$. There is a $vr-$path of nodes that carry the infection according to the definition of the problem. This path necessarily passes through S by the definition of a separator. Then a node in S must test positive. □

CLAIM 7.2. *Let S be a balanced separator of a graph, and C_i be a partition of the set of predecessors of the nodes in S. Assume that $q(S) = 1$ and $q(C_1) = q(C_2) = q(C_3) = 0$. The infection must then be located at one of the topologically greatest nodes in S, or there is no precedence relationship between the nodes of S in the infected path.*

PROOF. Similarly to the last claim, let us suppose the opposite. Then there are nodes $s_1, s_2 \in S$ such that s_1 is a predecessor of s_2 in the underlying tree and s_2 carries the infection. If s_2 is infected, then there is a s_2r- path of nodes that carry the infection that

passes through s_1. If the infected path passes through s_1, then the set of predeccesors of s_1 would have been tested positively. This contradicts the condition $q(C_1) = q(C_2) = q(C_3) = 0$.

□

Lemma 7.3. *A set of n positive real numbers $c_1, ..., c_n$ with $c_i < 0.5$ and $\sum_{i=1}^{n} c_i < 1$ can always be partitioned into three sets C_1, C_2 and C_3 such that $\sum_{i \in C_j} c_i < 0.5$ for $j = 1, 2, 3$. Furthermore, this can be done in polynomial time.*

PROOF. Consider the simple algorithm that orders the elements c_i in descendant order and then assigns each of them to the first bin C_j available for $i = 1, ..., n, j = 1, 2, 3$. The running time of this algorithm is trivially polynomial.

We assume that the algorithm is not correct. Consider the case where $n \geq 2$ since $n \leq 1$ is trivial. Then there exists an element c_k that does not fit into any of the bins. That is, $\sum_{i \in C_j} c_i + c_k > 0.5$ for each j at the moment of adding the kth element. If we sum over j, we get $\sum_{i \leq k-1} c_i + 3c_k > 1.5$. This implies $\sum_{i \leq k} c_i + 2c_k > 1.5$. Given $\sum_{i \leq n} c_i < 1$, we have $c_k > 0.5/2$. This is a contradiction because the elements are ordered, and with the first two items the condition $\sum_{i=1}^{n} c_i < 1$ would be broken.

□

Lemma 7.3 is a key part of the argument of correctness of Algorithm 8 as it is not sufficient to test all predecesors of the separator in a single test. To see this, consider a separator that induces 3 components of size $\frac{|V|-1}{3}$. Then if we test all predecesors in a single set, we could remain with $\frac{2(|V|-1)}{3}$ nodes for the next iteration.

Theorem 7.2. *Consider a rooted DAG $G = (V, E)$. Algorithm 8 finds the infection in $5\lceil \log_2 |V| \rceil$ steps. This represents a 5-approximation algorithm. Furthermore, the algorithm can be implemented in polynomial time for the class of graphs with a bounded treewidth. For the class of unbounded treewidth, the algorithm can be implemented in*

polynomial time with $\mathcal{O}(\log_2 |V| + \log_2(s\sqrt{\log s}))$ steps, with s as the balanced separator number of the graph.

PROOF. First, we will prove the correctness of the algorithm. In every next iteration, we maintain the topolylogically lowest nodes whose connected components tested positive. This means that every node discarded cannot have been the lowest being infected by the problem definition. By Lemma 7.2, we see that the addition of a *dummy* root has no impact on the search process.

We will now analyze the number of steps of the algorithm. First, we note that in every iteration we will reduce the number of nodes by half, unless $q(S) = 1$ and $q(C_1) = q(C_2) = q(C_3) = 0$. In that case, we would have at most $|S|$ nodes. Then we can do a binary search on them, finding the infected node in at most $\lceil \log_2 |S| \rceil$ more steps. This is valid since Claim 7.2 guarantees that there is no relationships of predecessors between nodes in of the infected path in S. As $q(S) = 1$, then exactly one of the nodes in the set carries the infection. In the case where $q(S) = 0$ or $q(C_1) \vee q(C_2) \vee q(C_3)$, we would remove half of the graph nodes with the 4 tests. Taking into account both scenarios, this gives us at most $4\lceil \log_2 |V| \rceil + \lceil \log_2 |S| \rceil$ queries. If we use a balanced separator with a size bounded by the balanced separator number s, we can replace $|S|$ with s. Note that since $|S| \leq |V|$ and $\lceil \log_2 |V| \rceil$ represents the informational lower bound, we find that Algorithm 8 is a 5-approximation.

Finally, to calculate the complexity of the algorithm, we can analyze each part. The loop while is bounded by $\log_2(|V|)$. The partition of subsets can be done in polynomial time using the algorithm of the Lemma 7.1. The search for a balanced separator S is not easy, as all known algorithms have an exponential time. However, using the results of Feige et al. (Feige, Hajiaghayi, & Lee, 2005) we can calculate a separator of size $\mathcal{O}(s\sqrt{\log s})$ in polynomial time. The price to pay would be that the algorithm now finds the infection in $\mathcal{O}(\log_2 |V| + \log_2(s\sqrt{\log s}))$ steps. For the class of graphs of bounded treewidth, the problem can be solved in $4\lceil \log_2 |V| \rceil + \lceil \log_2 k \rceil$, with k as the

treewidth of the graph, if we use a constant approximation algorithm such as the one presented by Boedlander (Bodlaender et al., 2016) and the balanced separator induced by tree decomposition (Robertson & Seymour, 1986). However, since any separator has a size that is not larger than the number of nodes in the graph, this scenario also represent a 5-approximation algorithm. □

Note that the result of the approximation factor is strong, in the sense that the value we are using to compare the quality of our solution is the informational lower bound. This number is calculated without restrictions of any kind. If the lower bound used can be improved, the approximation factor of the algorithm may be reduced.

8. CONCLUSIONS

8.1. Conclusions

In the present work, we have modeled the problem of finding an infection in the wastewater network as finding a node in a tree with uncertainty. We then studied the theoretical problem. First, we listed a series of properties of reasonable search strategies that helped us characterize optimal solutions. We then explored its difficulty and showed that the problem is NP-hard when we aim to minimize the height of the search tree. The average case was already proved to be NP-Complete by Cicalese et al. (Cicalese et al., 2011).

In Chapter 4 we explored possible lower bounds that help to bind the height of optimal solutions to the structural parameters of the graph. We first showed that the treewidth does not act as a lower bound. We then introduced the *testing separator number*, which is a natural lower bound for the height of the optimal strategy in the worst case. Although its definition is intuitive, in practice it is not easy to compute its value. This definition suggests a brute-force algorithm that represents a $\log_2(|V|)$ approximation. The result is relevant from a theoretical point of view, but is not very useful in real applications.

In chapters 5 and 6 we explored the empirical problem of actually finding the infection. For this, we used real and random instances. We presented 3 algorithms: the Greedy Algorithm, the Path Separation Algorithm, and the Entropy Minimization Algorithm. Although we could not give approximation guarantees for any of them, all three showed to perform very well in practice. In particular, the Entropy and Path Separation Algorithms gave results very close to the optimal solution for the worst case scenario in the spanning tree of the graph. Although both algorithms require us to calculate a very space-expensive matrix of paths, this process can be bypassed in the Path Separation Algorithm when $k = 1$ and we are looking to minimize the height of the decision tree, or the average cost over a uniform distribution over the nodes.

For its part, the Greedy Algorithm had worse results than the algorithms based on entropy and paths. However, its performance was not consistent in real-life and random instances. The results were reasonable for the average case scenario, but in the worst case, there was a tail that proved difficult to bound. The last algorithm tested was the Treewidth Algorithm. Its results were very poor. The algorithm proved to be useful in theory, but not in applied instances.

Finally, in Chapter 7, we explored the variation of the problem in light of the Group Testing framework. This is a very natural problem. Both frameworks share the same objective, but differ in limitations given by technological capabilities. In this context, we develop an optimal algorithm that can find the infection in $\lceil \log_2 |V| \rceil$ steps. We also presented a 5-approximation algorithm in the case where the size of the set to test is restricted by the separator number of the graph multiplied by the maximum in-dregree.

8.2. Open Problems

As this work explored a variation not studied of a known problem, it answered many questions and opened many more. We will try to cover the most important of them in this section. The first unanswered subject relates to the approximations guarantees that may be achieved. Although the Entropy Minimization achieved excellent results in the studied instances, it is unknown if it is possible to upper bound its response by any factor on the optimal solution. Also, it is not clear if it is even possible to achieve a constant-factor approximation for the average case or the worst case. It would also be relevant to answer if there is a case when this algorithm can be calculated in polynomial time without the paths probabilities encoded explicitly in the input.

We have shown that the test separator number is a natural lower bound of the height of the optimal strategy. But in practice, this result may not be very useful, as it depends on the separator number of a graph, which itself is NP-complete to compute. However, since there are approximation algorithms that may find the separator in polynomial time,

it could be useful to develop an algorithm to calculate this value efficiently. Another line of study can be related to developing new lower bounds to better understand the difficulty of different instances.

Other issues that may be explored in future work are variations of the studied problem. For example, optimal solutions may be achieved in polynomial time for the average case scenario if we restrict ourselves to families of probability distributions over the nodes or paths. It would also be interesting to explore the case where queries on each node have an associated cost.

REFERENCES

Ahmed, W., Angel, N., Edson, J., Bibby, K., Bivins, A., O'Brien, J. W., ... Mueller, J. F. (2020). First confirmed detection of SARS-CoV-2 in untreated wastewater in Australia: A proof of concept for the wastewater surveillance of COVID-19 in the community. *Science of The Total Environment*, *728*. doi: https://doi.org/10.1016/j.scitotenv.2020.138764

Baldovin, T., Amoruso, I., Fonzo, M., Buja, A., Baldo, V., Cocchio, S., & Bertoncello, C. (2021). SARS-CoV-2 RNA detection and persistence in wastewater samples: An experimental network for COVID-19 environmental surveillance in Padua, Veneto Region (NE Italy). *Science of The Total Environment*, *760*. doi: https://doi.org/10.1016/j.scitotenv.2020.143329

Ben-Asher, Y., Farchi, E., & Newman, I. (1999). Optimal search in trees. *SIAM Journal on Computing*, *28*(6), 2090–2102.

Bentley, J. L. (1979). Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, *SE-5*(4), 333–340.

Bodlaender, H. L., Drange, P. G., Dregi, M. S., Fomin, F. V., Lokshtanov, D., & Pilipczuk, M. (2016). A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, *45*(2), 317–378.

Bodlaender, H. L., & Koster, A. M. (2011). Treewidth computations ii. lower bounds. *Information and Computation*, *209*(7), 1103–1119.

Brinkhoff, T. (2002). A framework for generating network-based moving objects. *GeoInformatica*, *6*(2), 153–180.

Carmo, R., Donadelli, J., Kohayakawa, Y., & Laber, E. (2004). Searching in random partially ordered sets. *Theoretical Computer Science*, *321*(1), 41–57.

Cheraghchi, M., Karbasi, A., Mohajer, S., & Saligrama, V. (2012). Graph-constrained Group Testing. *IEEE Transactions on Information Theory*, *58*(1), 248–262.

Cicalese, F., Jacobs, T., Laber, E., & Molinaro, M. (2011). On the complexity of searching in trees and partially ordered structures. *Theoretical Computer Science*, *412*(50), 6879–6896.

Cicalese, F., Jacobs, T., Laber, E., & Molinaro, M. (2014). Improved Approximation Algorithms for the Average-Case Tree Searching Problem. *Algorithmica*, *68*(4), 1045–1074.

Cicalese, F., Jacobs, T., Laber, E., & Valentim, C. (2012). The binary identification problem for weighted trees. *Theoretical Computer Science*, *459*, 100–112.

Dereniowski, D. (2008). Edge ranking and searching in partial orders. *Discrete Applied Mathematics*, *156*(13), 2493–2500.

Feige, U., Hajiaghayi, M., & Lee, J. R. (2005). Improved approximation algorithms for minimum-weight vertex separators. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of Computing* (pp. 563–572).

Gruber, H. (2013). *On Balanced Separators, Treewidth, and Cycle Rank.* arXiv.

Harvey, N. J. A., Patrascu, M., Wen, Y., Yekhanin, S., & Chan, V. W. S. (2007). Non-Adaptive Fault Diagnosis for All-Optical Networks via Combinatorial Group Testing on Graphs. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications* (p. 697-705).

Karbasi, A., & Zadimoghaddam, M. (2012). Sequential group testing with graph constraints. In *Proceedings of IEEE Information Theory Workshop* (pp. 292–296).

Kokkinos, P. A., Ziros, P. G., Mpalasopoulou, A., Galanis, A., & Vantarakis, A. (2011).

Molecular detection of multiple viral targets in untreated urban sewage from Greece. *Virology Journal*, *8*, 1–7.

La Rosa, G., Iaconelli, M., Mancini, P., Bonanno Ferraro, G., Veneri, C., Bonadonna, L., ... Suffredini, E. (2020). First detection of SARS-CoV-2 in untreated wastewaters in Italy. *Science of The Total Environment*, *736*. doi: https://doi.org/10.1016/j.scitotenv.2020.139652

Linial, N., & Saks, M. (1985). Searching ordered structures. *Journal of algorithms*, *6*(1), 86–103.

Lipman, M. J., & Abrahams, J. (1995). Minimum average cost testing for partially ordered components. *IEEE Transactions on Information Theory*, *41*(1), 287–291.

Mozes, S., Onak, K., & Weimann, O. (2008). Finding an optimal tree searching strategy in linear time. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete Algorithms (SODA'08)* (Vol. 8, pp. 1096–1105).

Onak, K., & Parys, P. (2006). Generalization of binary search: Searching in trees and forest-like partial orders. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)* (pp. 379–388).

Prado, T., Fumian, T. M., Mannarino, C. F., Maranhão, A. G., Siqueira, M. M., & Miagostovich, M. P. (2020). Preliminary results of SARS-CoV-2 detection in sewerage system in Niterói municipality, Rio de Janeiro, Brazil. *Memórias do Instituto Oswaldo Cruz*, *115*. doi: 10.1590/0074-02760200196

Robertson, N., & Seymour, P. D. (1986). Graph minors. II. Algorithmic aspects of tree-width. *Journal of algorithms*, *7*(3), 309–322.

Rosenberg, A. L., & Heath, L. S. (2001). *Graph separators, with applications*. Springer Science & Business Media.

Rubio, B. (2023). *Searching for Infections: Algorithms for Multiple Sampling on Trees and Planar Graphs.* (unpublished thesis)

Sihag, S., Tajer, A., & Mitra, U. (2021). Adaptive Graph-Constrained Group Testing. *IEEE Transactions on Signal Processing*, *70*, 381–396.

APPENDIX

A. AN EXACT FORMULATION OF TREE PARTITIONING

In this section, we present an Integer Programming formulation to solve the Greedy Sampling Problem exactly on trees. In Subsection A.1 we give an exact formulation. As in practical terms, the model was too computationally expensive to be solved completely, we developed a Bender's Formulation that is shown in Subsection A.2.

A.1. An exact formulation

Sets

- $V = v_1, ..., v_n$: nodes of graph G
- $I = I_1, I_2, ..., I_i, ..., I_n$: Ideal of node i
- $P = P_{1,r}, P_{2,r}, ..., P_{i,r}, ..., P_{n,r}$: Path from i to the root r

Variables

- $x_i \in \{0, 1\}$: The sample is taken at node i
- $y_{j,i} \in \{0, 1\}$: The node j is assigned to the sample of node i

Objective function

$$\min Z$$

Restrictions

(i) Take K samples

$$\sum_{i \in V} x_i = K$$

(ii) A node can only be assigned to another if a sample is taken at the latter.

$$y_{j,i} \leq x_i, \qquad \text{for all } j \in I_i, i \in V$$

(iii) Every node i is assigned a sample that is on the path between i and the root r

$$\sum_{j \in P_{i,r}} y_{i,j} = 1, \qquad \text{for all } i \in V$$

(iv) Nodes can only be assigned to their closest parent

$$x_l \leq 1 - y_{j,i}, \qquad \text{for all } i, j \in V, l \in P_{j,i} \setminus \{i\}$$

(v) Lower bound on Z

$$\sum_{j \in I_i} y_{j,i} \leq Z, \qquad \text{for all } i \in V$$

A.2. Benders decomposition

$$\min c^T x + \theta = \theta$$

Subject to:

(i) $\sum_{i \in V} x_i = K$
(ii) $Q(x) \leq \theta$

With

$$Q(x) = \min_{y,Z} Z$$

$Q(x)$ is subject to:

(i) $y_{j,i} \leq x_i, \qquad \text{for all } i \in V, j \in I_i$
(ii) $\sum_{j \in P_{i,r}} y_{i,j} = 1, \qquad \text{for all } i \in V$
(iii) $y_{j,i} \leq 1 - x_l, \qquad \text{for all } i, j \in V, l \in P_{j,i} \setminus \{i\}$
(iv) $\sum_{j \in I_i} y_{j,i} - Z \leq 0, \qquad \text{for all } i \in V$

Feasibility cuts:

$$\sum_{i \in V} \sum_{j \in I_i} \alpha_i x_i + \sum_{i \in V} \beta_i + \sum_{i \in V} \sum_{j \in V} \sum_{l \in P_{j,i} \setminus \{i\}} \gamma_l (1 - x_l) \leq 0, \qquad \forall \alpha, \beta, \gamma \in R$$

Optimality cuts:

$$\sum_{i \in V} \sum_{j \in I_i} \alpha_i x_i + \sum_{i \in V} \beta_i + \sum_{i \in V} \sum_{j \in V} \sum_{l \in P_{j,i} \setminus \{i\}} \gamma_l (1 - x_l) \leq \theta, \qquad \forall \alpha, \beta, \gamma \in V$$

B. COMPARISSON OF EXACT AND RELAXED GREEDY APPROACH FOR ALGORITHM 2

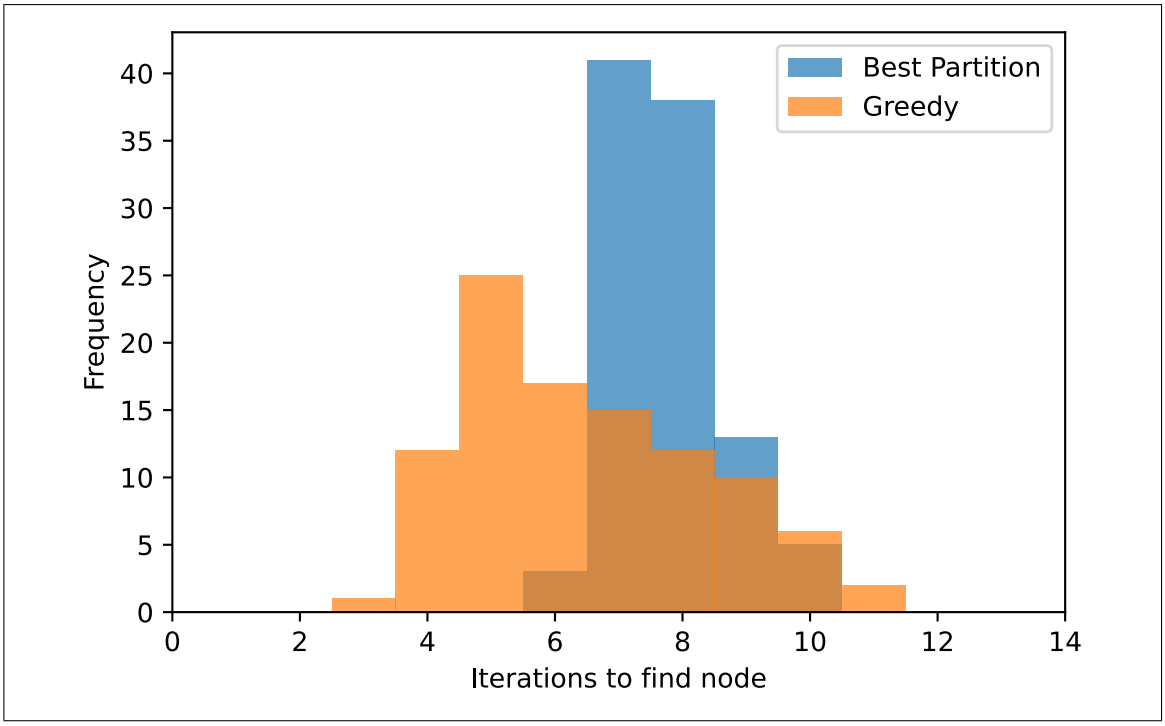


Figure B.1. Histogram of iterations for searching in a subtree of 176 nodes of the network of San Pedro de la Paz for the Greedy Algorithm (G) and the Best Partitioning Algorithm (B) for $k = 1$. Note that the mean number of iterations is lower in (G), while the maximum value is lower in (B).

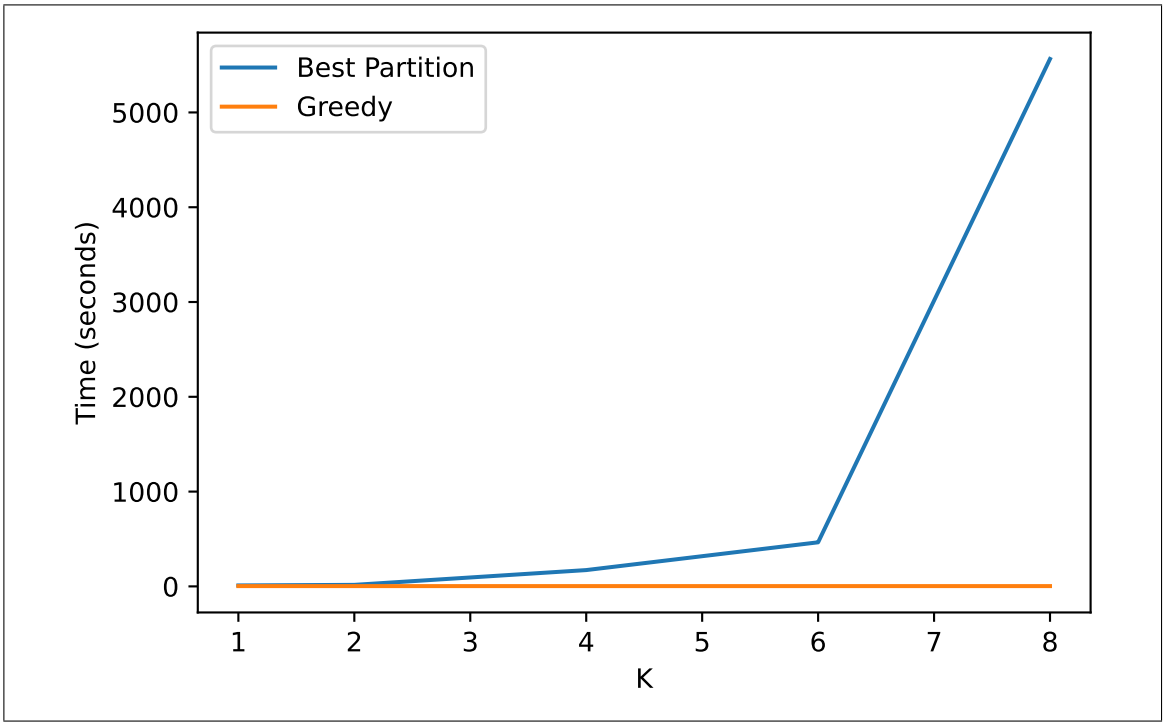


Figure B.2. The time needed to cumpute the optimal partition for the Best Partiton Algorithm grows exponentially in the value of k. On the other hand, the time needed to compute the partition for the Greedy Algorithm remains almost constant for different values of k.

C. RESULTS OF SIMULATIONS

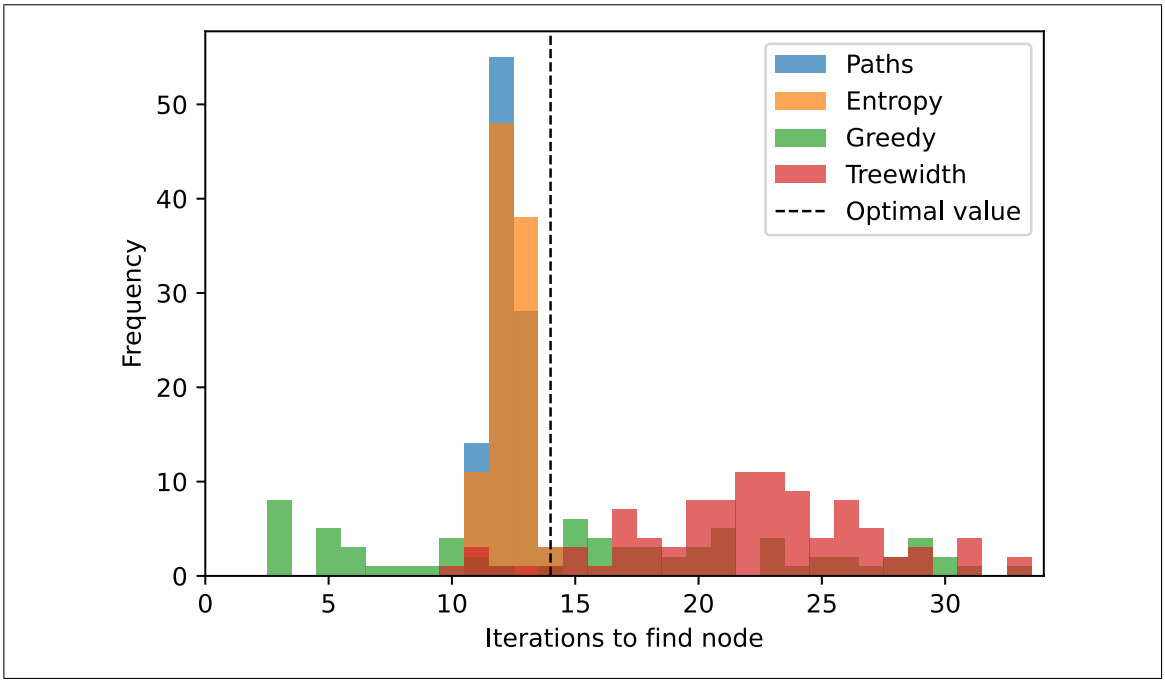


Figure C.1. Simulation results in San Pedro de la Paz for a subyacent tree with $k = 1$. Note that the Entropy and Path Separation Algorithm (that in the case without uncertainty are the same) do not have more iterations than the optimal number of iterations in the worst case.

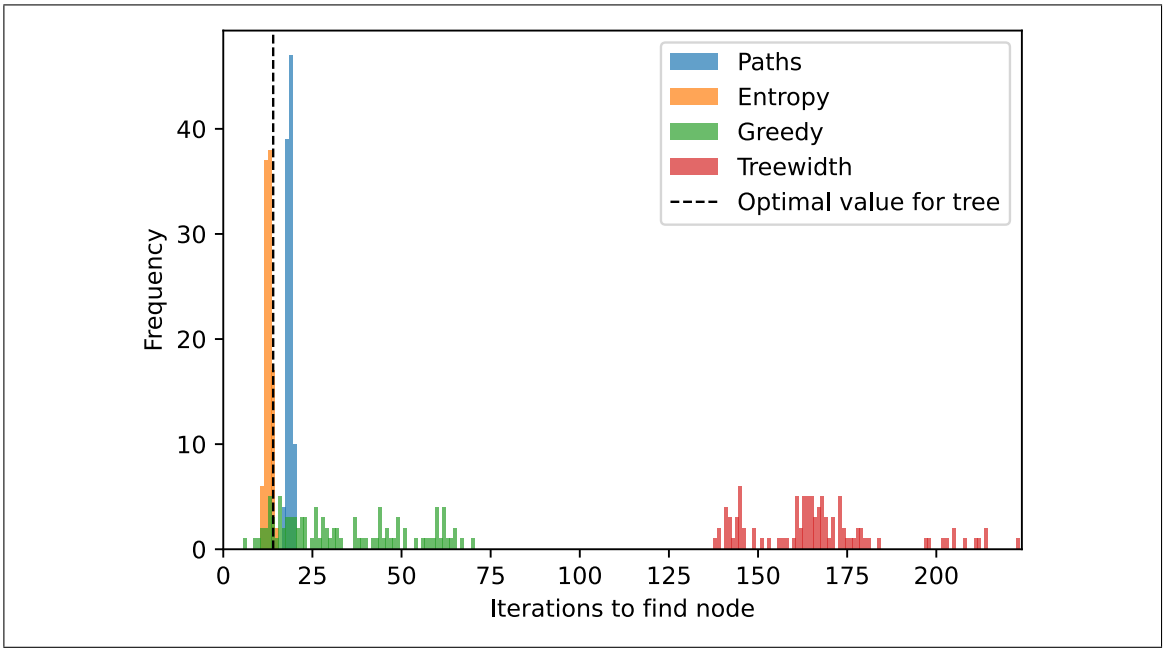


Figure C.2. Simulation results in San Pedro de la Paz for $k = 1$ including iterations of Treewidth Algorithm.

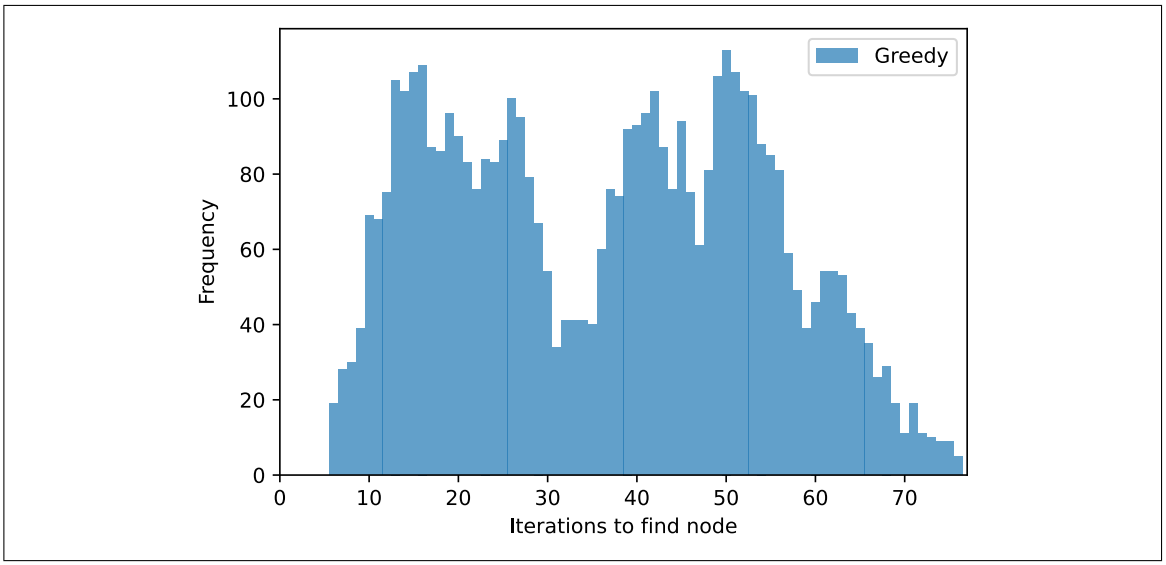


Figure C.3. Simulation results of all nodes in San Pedro de la Paz for $k = 1$ for the Greedy Algorithm.

Algorithm / Statistic	Mean	Max	90th percentile
Optimal WC Search in Tree	12.96	14	14
Greedy	23.81	55	48
Paths	12.2	14	13
Entropy	12.33	14	13
Treewidth	22.06	33	28

Table C.1. Simulation results in San Pedro de la Paz for a subyacent tree with $k = 1$.

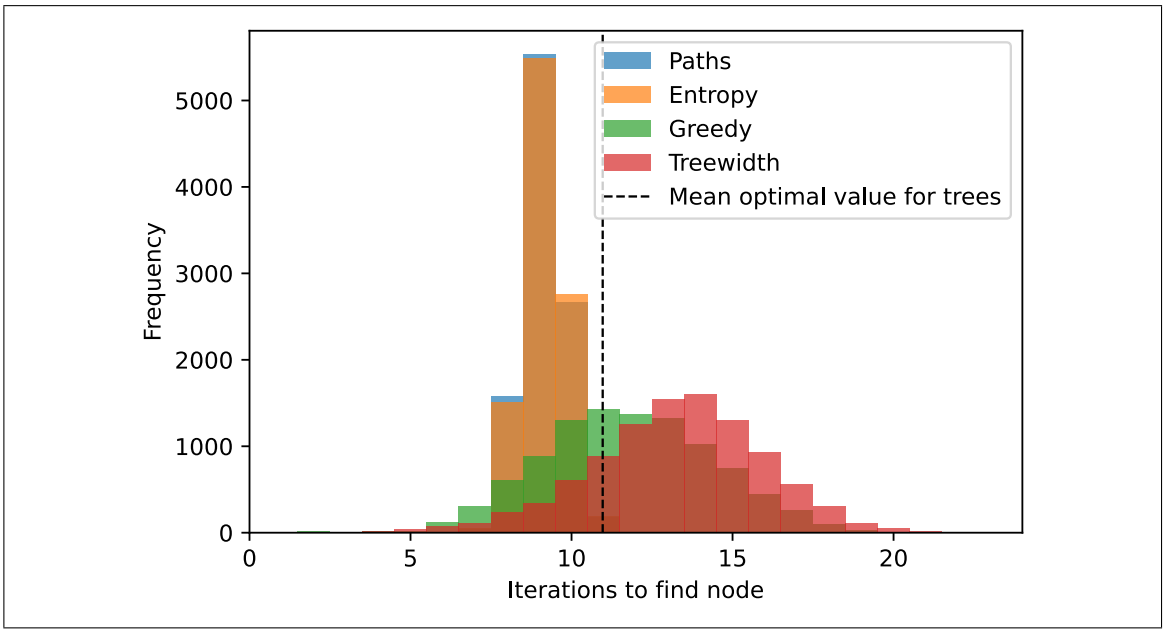


Figure C.4. Simulation results in 100 random graphs of 500 nodes, no extra edges and $k = 1$.

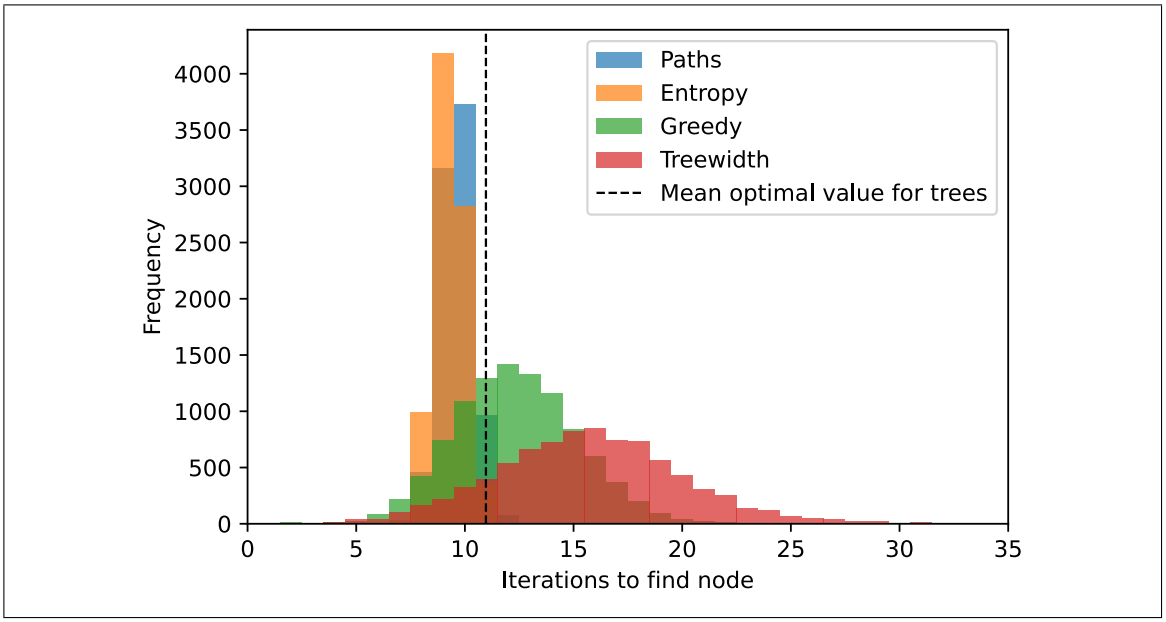


Figure C.5. Simulation results in 100 random graphs of 500 nodes, with 5% extra edges and $k = 1$.

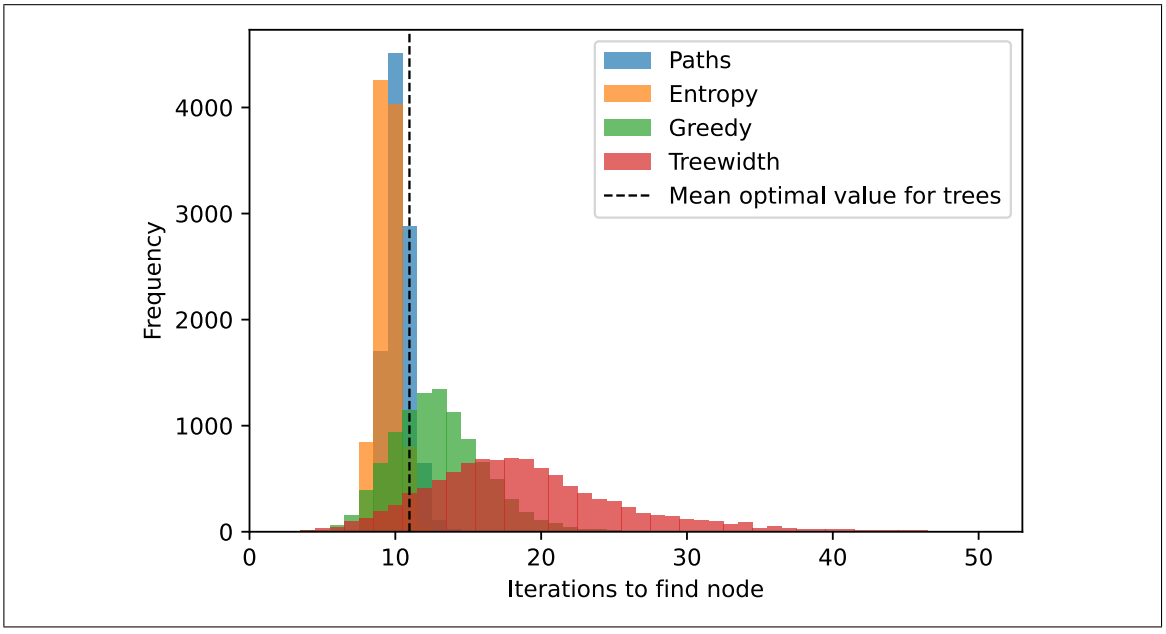


Figure C.6. Results of simulations in 100 random graphs of 500 nodes with 10% extra edges and $k = 1$. The histograms are consistent with the results in the graph of San Pedro de la Paz.