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, May 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
POSTGRADO

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, May 2023

Gratefully to my parents and siblings

ACKNOWLEDGEMENTS

Write in a sober style your acknowledgements to those persons that contributed to the development and preparation of your thesis.

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. However, the available information could be corrupted. We found some pipelines that do not carry flow in reality. They turned out to be unidentifiable. We model the problem of finding new outbreaks as a binary search in a directed acyclic graph that contains a directed in-tree that represents the actual sewage network. Our model is robust to any realization of the tree. We show that the problem is provably difficult, as it is NP-hard in both the worst case and the average case. The average case complexity was shown previously by Cicalese et al. (Cicalese, Jacobs, Laber, & Molinaro, 2011).

We then present a 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. To solve the problem computationally, we present three heuristic algorithms. Our algorithms perform well in both real and simulated instances, with the Entropy Algorithm providing particularly effective solutions, which 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.

Finally, we also connect the problem to the Group Testing Framework and provide an optimal algorithm for the unrestricted case, as well as a 5-approximation algorithm for case with the size bounded by the separator number multiplied by the maximum outdegree of the graph. Our findings have important 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

Este trabajo aborda el desafío de identificar posibles nuevos brotes de COVID-19 a través de las redes de aguas residuales de las ciudades. La información a la que se tiene acceso podría estar corrompida. Descubrimos que existían una serie de tuberías que no transportan flujo en realidad. Identificarlas no es posible. Modelamos el problema de encontrar nuevos brotes como una búsqueda binaria en un grafo acíclico dirigido que contiene un árbol dirigido que representa la red de alcantarillado real. Nuestro modelo es robusto ante cualquier realización del árbol. Mostramos que el problema es demostrablemente difícil, ya que es NP-difícil tanto en el peor caso como en el caso promedio. El caso promedio fue demostrado previamente por Cicalese et al. (Cicalese et al., 2011).

Luego presentamos algunas caraterísticas 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. Nuestros algoritmos tienen un buen desempeño en casos reales y simulados. El algoritmo de Entropía proporciona soluciones particularmente efectivas, que se acercan a la solución óptima en el caso sin incertidumbre. Sin embargo, su tiempo de ejecución puede ser exponencial en el número de caminos del grafo.

Finalmente, conectamos el problema con el marco de *Group Testing*. Proporcionamos un algoritmo óptimo para el caso no restringido, así como una 5-aproximación para el caso con el tamaño del conjunto a testear limitado al número separador multiplicado por el grado máximo. Nuestros hallazgos tienen importantes implicaciones en las áreas de *Group Testing*, investigación de operaciones, diseño de algoritmos y epidemiología basada 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 the same process and bound the zone of the new outbreaks again.

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 must 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.

The 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 implies that the number is an input of the problem and it is essentially a different scenario for every number of daily tests.

The 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.

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 (poset). Posets can be arranged in a way that for certain pairs of elements, one precedes the other. However, not all pairs can be compared. 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 searching in an unknown tree, not in a poset. This means that we do not have access to the actual arrangement of the nodes.

More close to our requirements is the 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 at most $\mathcal{O}(n^4 \log^3(|V|))$ steps (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 (Onak & Parys, 2006). Note that the model we study generalizes the binary search problem for trees.

Dereniowski proved the existence of a linear-time algorithm for searching in trees (Dereniowski, 2008). In 2008 Mozes et al. got the result seen by Dereniowski and showed a linear-time algorithm to find the optimal strategy for trees in the worst case (Mozes, Onak, & Weimann, 2008). This work represents a breakthrough, since it dramatically improved the previous known cubic running time. 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 iterations for searching in trees has been shown to be NP-complete by Cicalese et al. (Cicalese et al., 2011). They gave a complete characterization of the complexity of the problem on the basis of the graph diameter. In particular, the 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. Later, the same group presented in a companion paper a natural greedy algorithm that achieves a 1.62-approximation (Cicalese, Jacobs, Laber, & Molinaro, 2014). This algorithm in each iteration 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. Consider examples 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 very related to the problem of Group Testing. This framework allows us to test on sets of multiple elements and get a unique answer for all of them. 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 optimal 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 SARS-CoV-2 infection in the sewage network as finding an infected node in a directed acyclic graph. 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-complete. 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 also 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 entropy of the potentially infected nodes of the graph after a query. All three methods generalize a 2 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. As there is a path of infected nodes, and 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 for the unrestricted scenario. 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, more generally, to the literature on 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 will also give a useful description of the search strategies. 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. SEARCH STRATEGIES

This chapter is devoted to the definition and formulation of the necessary steps for the construction of our model. We will start by giving some basic notation and definitions in Section 2.1. We will formulate the model in Section 2.2 and finally characterize some necessary properties of search strategies that are useful and reasonable in Subsection 2.3.1.

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 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 $i = 1, ..., n - 1$.*

Definition 2.2 (Tree)**.** *A tree is an undirected graph in which all pairs of nodes are conected by exactly one path.*

Definition 2.3 (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 2.4 (Directed Acyclic Graph (DAG))**.** *A DAG G is a directed graph without directed cycles.*

In other words, every DAG can be topologically ordered in a way that is consistent with its edges directions.

Definition 2.5 (Rooted tree)**.** *A graph $T = (V, E)$ is a rooted tree if its undirected graph is a tree, and there is a node, which is called the root, to which every other node has a directed path.*

Definition 2.6 (Subtree)**.** *Given a directed rooted tree $T = (V, E)$ and a node $v \in V$, the subtree T_v represents the subgraph induced by all nodes that have a path to v.*

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

$$G[S] := (S, E')$$

$$E' := \{i, j \in S \wedge \exists ij\text{-path in } G \text{ with all nodes in } S\}$$

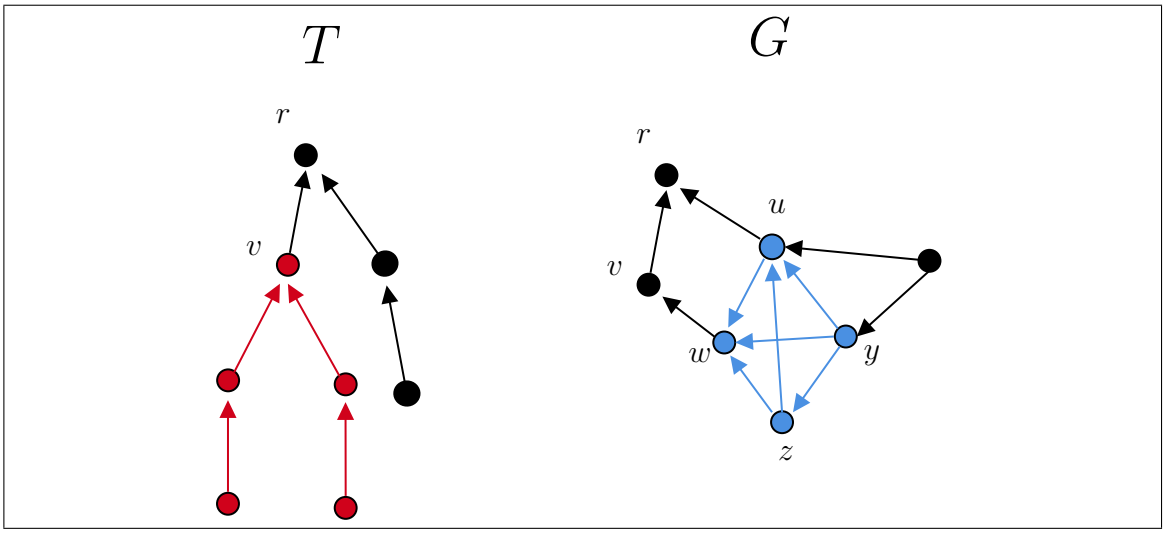


Figure 2.1. In red the subtree T_v and in blue the induced subgraph $G[S]$ for $S = \{u, w, y, z\}$.

2.2. Model formulation

A wastewater network can be represented with a rooted directed tree $T = (V, E)$ with a root r representing the Water Treatment Plant (WTP). A directed edge (u, v) represents the flow of wastewater from the node $u \in V$ to the node $v \in V$. By definition, every node v has a 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 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 $S \subseteq V$ such that $|S| = k+1$ with $r \in S$. That is, we need to select k nodes in addition to the root of the graph to test. Every node v in S will be queried and they will return a positive answer if the path of infection passes through v or a negative answer in the opposite case. Note that the query 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 path infected and $q(v) = 0$ otherwise.

In Section 2.2.1 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 2.2.2 to consider the case with extra edges. This model will have to be robust against different possible realizations of the real underlying tree T.

2.2.1. Search 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, then we can discard all nodes in T_v, as every path from them to the root passes through v by definition.

As we work in a tree, we can see that the samples in S induce a partition of T in several disjoint subtrees. In fact, let S_v be the set of samples in T_v without considering v. We can define the set of nodes.

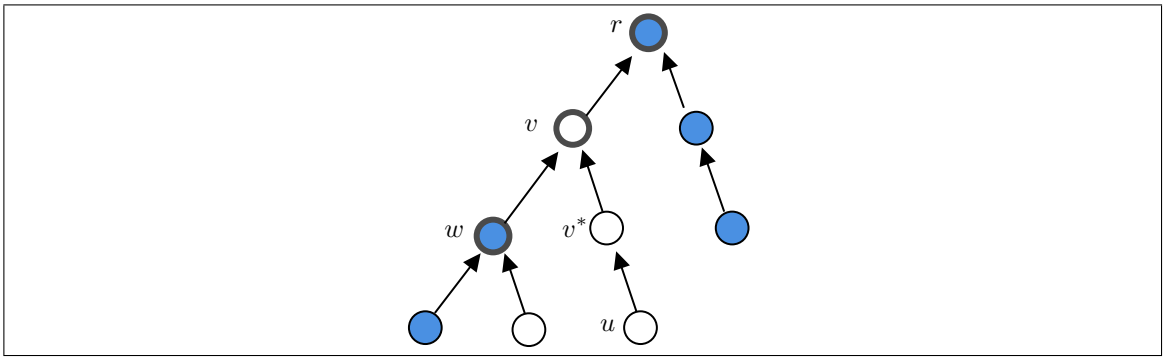


Figure 2.2. Example of an in-tree and a set of sample nodes S for $k = 2$. The 3 nodes in S are in bold. The three sets in $\{A_s : s \in S\}$ are the nodes of T_w, the nodes of T_v minus the nodes of T_w, and the nodes of T_r minus the nodes of T_v. The nodes in white correspond to A_v. Notice that 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 2.4 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 graph. 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^*. As the sample 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 that minimize the maximum days in the worst case scenario, or the average case over an arbitrary probability distribution π on the nodes of T. We will formalize these last ideas in Section 2.3.

2.2.2. Search in DAGs

Consider a DAG $G = (V, F)$ rooted in r such that there is an underlying 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 this set of arcs 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.

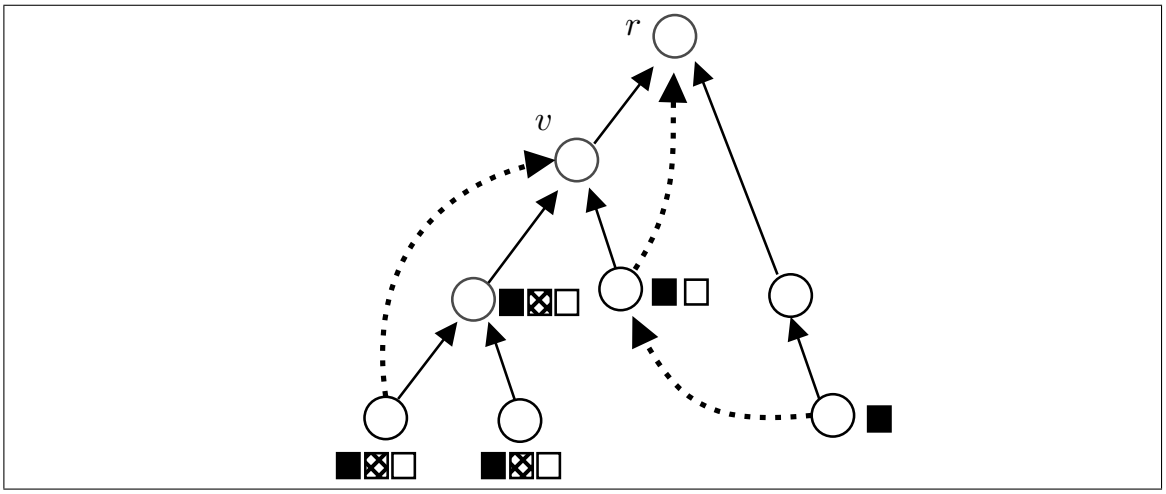


Figure 2.3. Example of sets $I_v(G)$, $N(T_v)$ and $\bar{I}_{r,\{v\}}(G)$. Nodes in $I_v(G)$ are shown next to a solid-colored box, nodes in $\bar{I}_{r,\{v\}}(G)$ are shown next to a partially colored box, and nodes in $N(T_v)$ are next to a transparent box. Edges that are not in the tree T are shown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

To be more precise about the nodes we can discard and those that we have uncertainties 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 S is the set of all nodes u that have all their paths to v passing through S.

Definition 2.8 (Ideal)**.** *The ideal generated by v in G, is defined as*

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

Definition 2.9 (Robust Ideal)**.** *The robust ideal generated by v in G restricted by S_0, is defined as*

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

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, we can 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 $\bar{I}_{v,S_0}(G)$ constitutes an underestimate of $I_v(T)$ for any $S_0 \subseteq V$ such that $\forall s \in S_0$ we have $q(s) = 0$.

CLAIM 2.1. *For any $v \in V$, and for any $S_0 \subseteq V$ such that $\forall s \in S_0$ we have $q(s) = 0$ it holds that $\bar{I}_{v,S_0}(G) \subseteq I_v(T) \subseteq I_v(G)$.*

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 $q(p) = 1$.

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 Claim 2.1, we can discard the robust ideal of the root restricted to the set of nodes that were tested negateively. This is valid, since every path from a node of the robust ideal to the root must have passed through this negatively tested set.

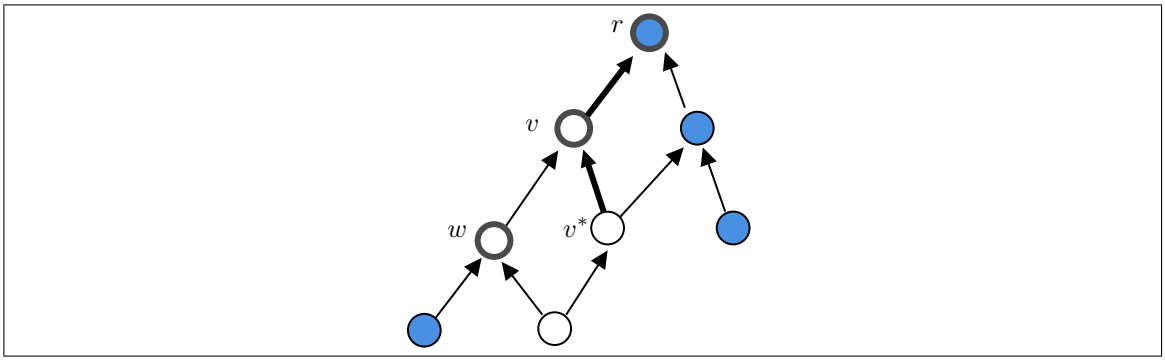


Figure 2.4. Situation for a given infected node v^* and sample set $S = r, v, w$ for the first day. The bold path denotes path P^*. The set of nodes colored white is M. That is, all blue nodes can be discarded with the information from the sample set S. Observe that the white nodes correspond to all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

Using the fact that G is a DAG and only in the nodes of P^* can the infection be traced, if there are multiple values of S that gave a positive response, we can always identify the lowest topological node u^* that is closest to the root r.

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 $i_v = 0$ and S_1 as the nodes v in S such that $i_v = 1$. We then define the following set.

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

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 2.1. *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 $i_v = 1$ and $S_0 = S \setminus S_1$, then v^* must belong to*

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

where u^* is the node furthest away from r within P^* among all nodes in S_1. Furthermore, M is the smallest set where the infection may be, and all nodes of P^* within M belong to the subpath from v^* to u^*.

PROOF. As $i_{u^*} = 1$, 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 $\bar{I}_{u,S_0}(G)$ if $u \in S_0$, as otherwise the path P^* must contain the node u which would imply that $i_u = 1$, which is a contradiction. This implies that $v^* \in M$.

Now we will prove that M is the smallest set in which infection can occur. Suppose that there exists $u \in M$ but no realization of a tree T consistent with S_0, S_1 and with u as the infected node. Since for every infected node, we know that its path to u^* is a sequence of nodes that would result in a positive response upon being tested, we conclude that there must exist a node in S_0 for every uu^*-path. Otherwise, T could be any tree consistent with S_0, S_1 and that contains a uu^*-path without nodes in S_0. This implies $u \in \bar{I}_{u^*,S_0}$. Therefore, $u \notin M$. We have reached the desired contradiction and conclude that for every node in M there exists a realization of a tree consistent with S_0, S_1, and with u as the infected node.

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. □

Using the result of Lemma 2.1, whenever we mention the set consistent with a series of answers, we refer to M.

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.

Lastly, 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 2.3. An example of the search process can be seen in Figure 2.5.

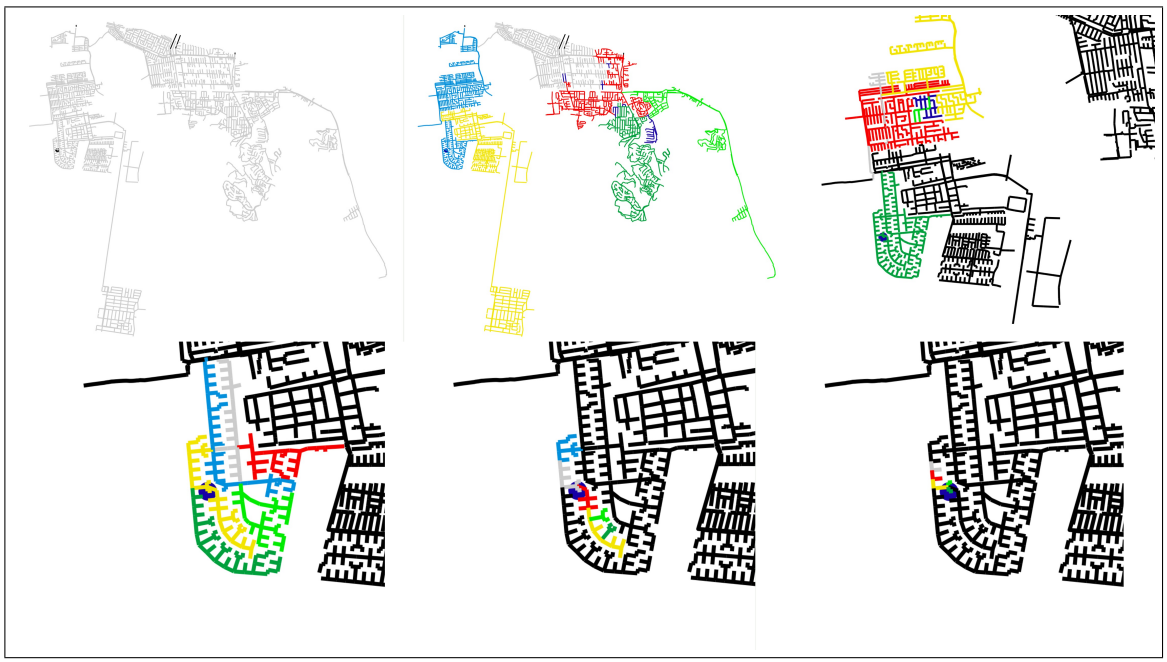


Figure 2.5. An example of a search in a graph. In the left upper picture the original graph. The dark point represents the infected node. In the next pictures are colored the induced partition for the samples and in black the discarded nodes. The narrowing of the search space is then shown.

2.3. Search strategies

We begin this section by defining a search strategy. Informally, a search strategy is a procedure that defines which node to ask 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. As every answer is positive or negative, a search strategy can be represented as a binary tree. However, all results are directly generalizable if we consider a $(k+1)$-ary tree querying k tuples instead of a binary tree querying sigle nodes. More precisely, we define a search strategy in the following way.

Definition 2.10 (Search Strategy)**.** *Given a graph $G = (V, E)$, a search tree for G is a tuple $D = (N, E', A)$, where N, E' are the nodes and edges of a binary tree and $A : N \to V$ is an assignment function. The search tree satisfies the following property:*

- *For each $v \in V$ and for each path between v and the root of G, there exists a leaf $l \in N$. This leaf l is assigned to the only node v that is consistent with the series of branch queries in N. In other words, the leaf l unequivocally denotes the 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. The function A represents a mapping that assigns the vertices of the binary tree with 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 of the strategy for the series of queries and answers indicated by the brach.

This definition is correct as the property requested in the definition guarantees that for each potentially infected path there is an assigned branch that responds correctly to the infected node.

Consider $d(D, v)$ as the function that calculates the distance from a node v to the root of D and $w(A(v))$ as a function that assigns weights to each node. To calculate the cost of a search strategy in the worst case, which we will usually call its height, we only need:

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

On the other hand, the cost of a search strategy D in the average case is:

$$cost(D) := \sum_{v \in leaves(D)} d(D, v) w(A(v))$$

In this work, we will focus mainly on the worst ca'se scenario. Without loss of generality, 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.

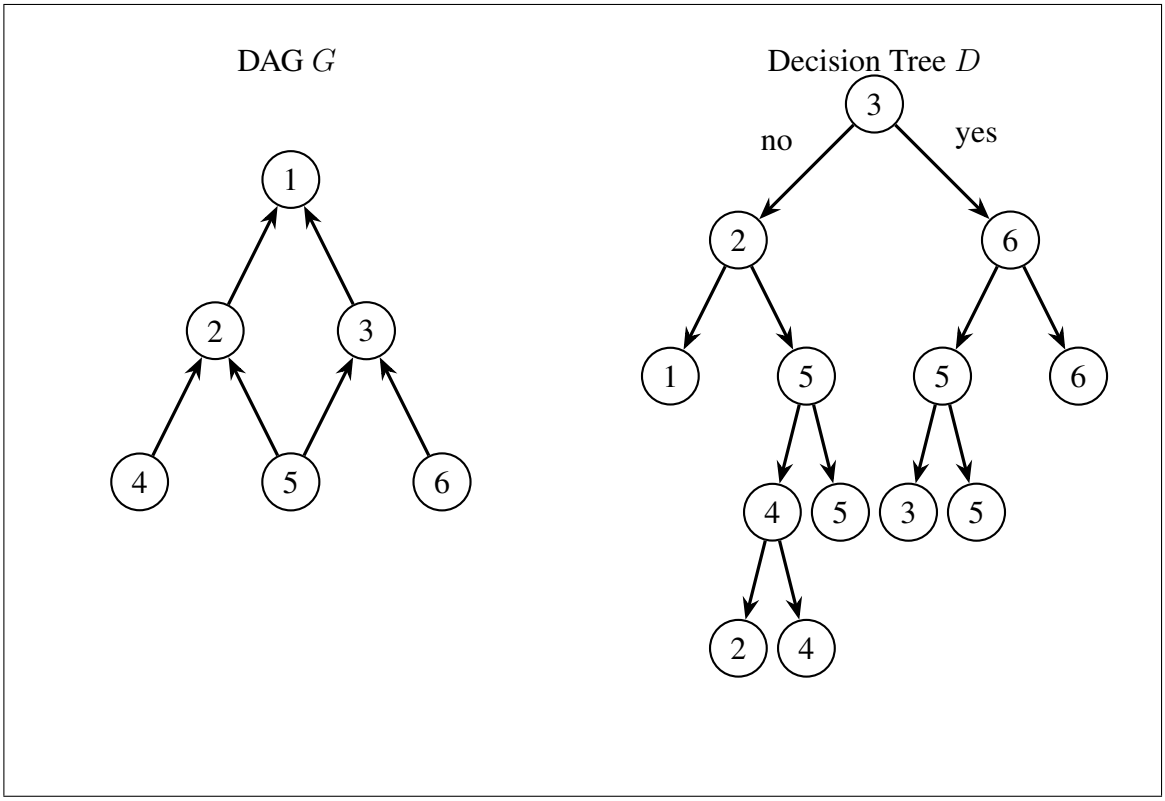


Figure 2.6. Example of a search strategy D for a DAG G.

2.3.1. Properties of reasonable search strategies

Using Definition 2.10, we can now list some properties of the optimal search tree with some properties. First, we make an additional assumption about the structure of optimal strategies. In the case of the worst case scenario, we will assume that every subtree of the search strategy is optimal. That is, every subtree of the binary tree has the minimum height possible. This, 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 this assumption this undesirables strategies could be considered 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.

PROPERTY 2.1. *Let D be an optimal search strategy for a rooted DAG G with root r. Then r is not at any internal vertex of D.*

PROOF. Assume that r is in an internal vertex of D. Note that querying in r gives no information, as by the model formulation we have $q(r) = 1$. Then we can remove the vertex of r from D, removing the branch that hangs on r in the case of a negative answer, and moving one position up the one that expected a positive answer. This contradicts the assumption of the optimality of subtrees in the search strategy. □

Finally, we 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. Intuitively, what we do is assume an answer for these queries and replace their query in D with the result of that answer. An example is shown in figure 2.7.

Definition 2.11 (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.*

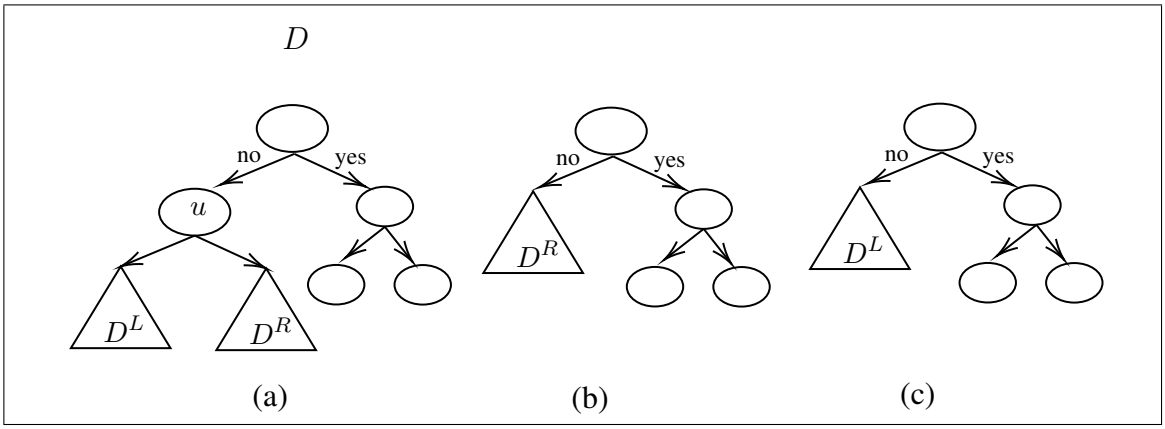


Figure 2.7. In (a) a decision tree. In (b) the decision tree after a left deletion of u. In (c) the decision tree after a right deletion of u.

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 the one presented by the authors. However, we must limit ourselves to subgraphs that have been obtained through a series of consistent queries in the graph.

Lemma 2.2. *Consider a rooted DAG G. Then, for all $S \subseteq V$ such that S can be obtained through any consistent series of answers, we have $OPT_{G[S]} \leq OPT_G$, where $OPT_{G[S]}$ represents the maximum length of a branch of the optimal decision tree of the induced subgraph by the set of nodes S in G.*

PROOF. Let D be the optimal search tree for G, and let O be a partial order of its nodes. Note that O necessarily has a unique maximal element r'. That is, r' is the root of the induced subgraph. Specifically, r' can be the root or the last sample that responded positively. We define the relation $u \geq v$ as the predence of u over v in G. For all $S \subseteq V$, we have $r' \geq s, \forall s \in S$.

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 S. For all vertex $u \in V \setminus S$ in the

decision tree D, we can apply a left deletion (assuming the answer is negative) if it is not true that $u \geq s, \forall s \in S$. Note that for every element in S, there still exists a path to r', so these nodes have not been discarded. In the case $u \geq s, \forall s \in S$, we apply a right deletion (assuming the answer is positive). From this we can build a search tree for $G[S]$ of equal or smaller size than D. This leaves us with a new decision tree D'.

□

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 $P = NP$, which is very unlikely.

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 and directed acyclic graph (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 (that is, a rooted tree with all paths directed toward the root), 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 $c_v \geq 0$ and 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: A rooted 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). In the original paper, the problem was defined in a different but equivalent manner. Indeed, queries are taken over 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 rooted 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 3. 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.

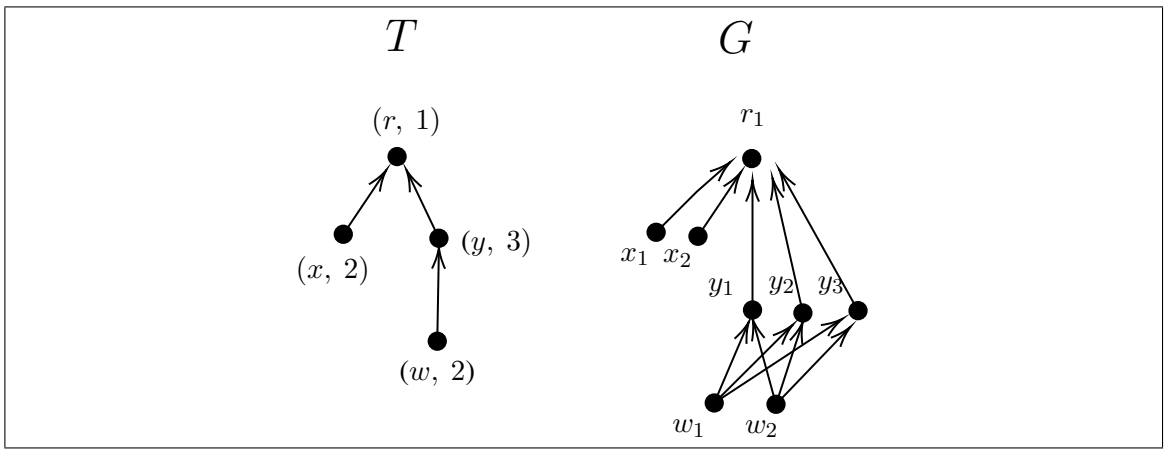


Figure 3.1. Transformation of a tree T to a DAG G: At left is the tree T. Each node has a label of the form $(v, c(v))$. At the right is the new graph G with labels on the nodes that mark its *parent*.

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 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 carry the infection due to the construction of

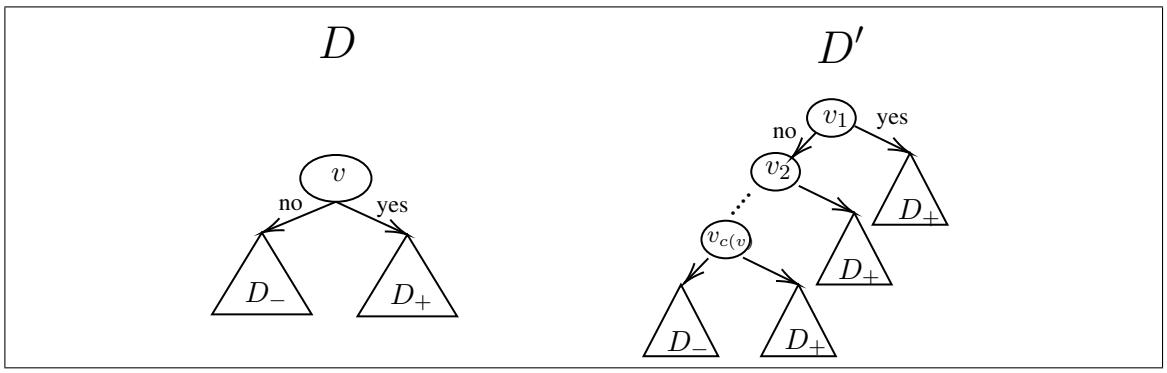


Figure 3.2. The recursive procedure that transforms D into D'

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 $w^i_1, \ldots, w^i_s$ we must have queried v^1 and all its predecessors, as shown 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 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.

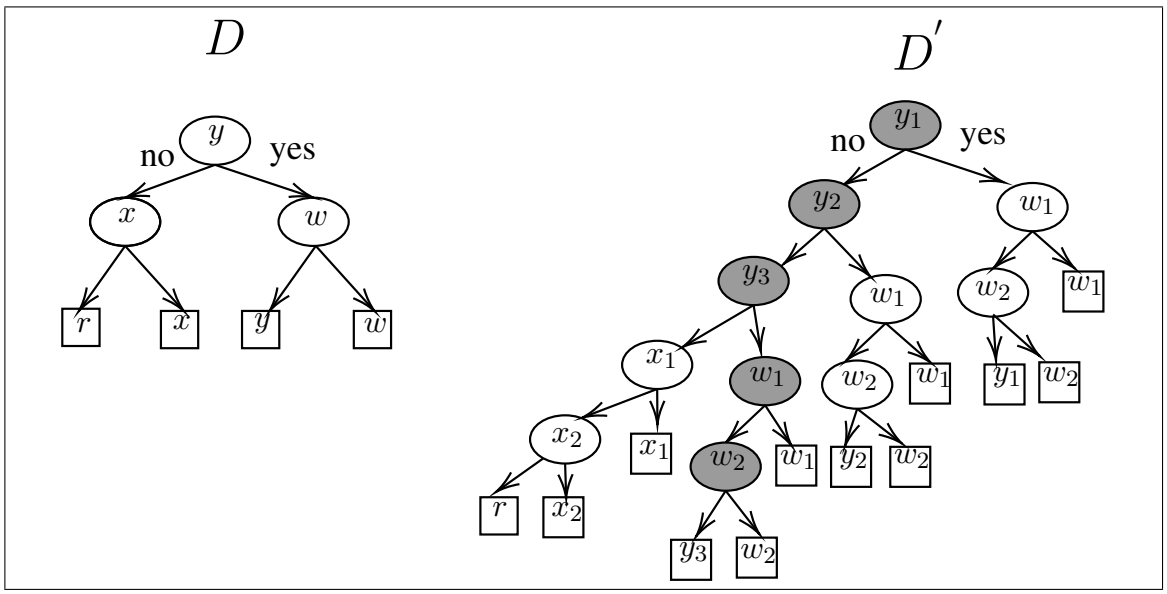


Figure 3.3. The final result of the transformation of a search strategy D for the tree T to D' for a DAG G, and vice versa if looked from right to left. In grey is the largest branch in D', which matches the value of the cost of D.

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 a node of the ideal of v_i in G is a copy of a node in the ideal of v in G. Analogously, since the robust ideal of all nodes v_i is equivalent to $T - T_v$, 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 seen 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 optimal. Then the lower values can act as a reference point for the quality of our approaches.

A natural conjecture was that the treewidth of a graph could act as a lower bound of the height of the optimal solution. This was a reasonable expectation, as the treewidth, in a sense, quantifies how close a graph is to a tree.

In this chapter, we are going to explore the height of optimal solutions based on 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. 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 2.2 we can restrict ourselves to the case of node v_{max} with a maximum in-degree of G and all its predecessors. Consider an adversarial player who decides the value of the node that we are querying. Note that querying in the root does not give any information. Then, if we test any of the predecessors, she can always respond with a negative answer for the first $\Delta - 1$ queries. We would then need an additional test on the Δ predecessor to determine if the infection is in v_{max} or in 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 a 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 papers by Seymur and Robertson (Robertson & Seymour, 1986).

Definition 4.1 (Tree decomposition)**.** *A tree decomposition of a 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.*

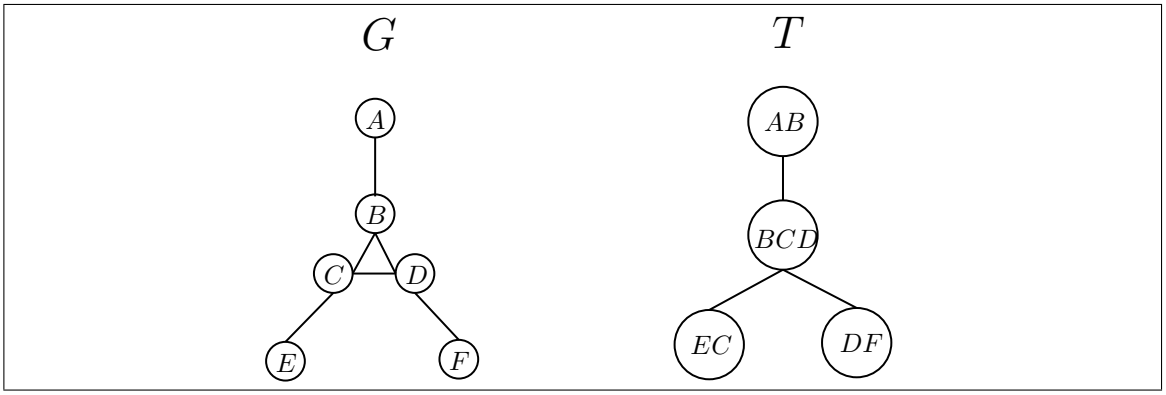


Figure 4.1. A graph G and a tree decomposition T of width 2.

We now introduce the definition of the balanced separator of a graph. This object will be very 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 a 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 a graph G, denoted by $s(G)$ (or simply s when the context allows it), is defined as the smallest integer k such that for every $Q \subseteq V$ (G), the induced subgraph $G[Q]$ admits a balanced separator of size at most k.*

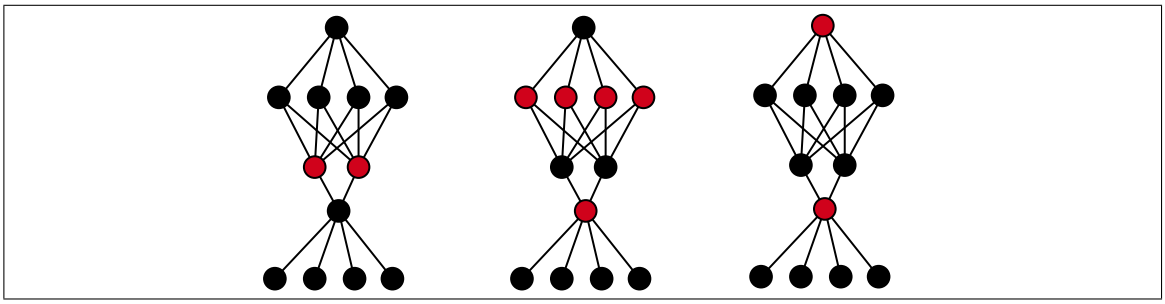


Figure 4.2. A graph with three different balanced separators marked in red.

4.1.1. A $(tw(G) + \Delta) \log_2(|V|)$ 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 (). 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, and we will 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

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

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

  $C \leftarrow$ connected component where is r

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

end

Test predecessor of u^*

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

  return u^*

end

$\bar{u}^* \leftarrow$ 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 (?,?).

Lemma 4.1. *(?, ?) Consider a rooted DAG G of treewidth $tw(G)$. There is always a height search strategy at most $(tw(G) + \Delta) \log_2(|V|)$.*

4.1.2. Counterexample: The Hypercube

Motivated by the lemma 4.1, we expected that the treewidth of a graph could represent a lower bound in the height of its 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 this structure in d^2 steps for a hypercube in $\mathbb{R}^d$. The result also shows that neither the separator number of a graph is a lower bound of its optimum value.

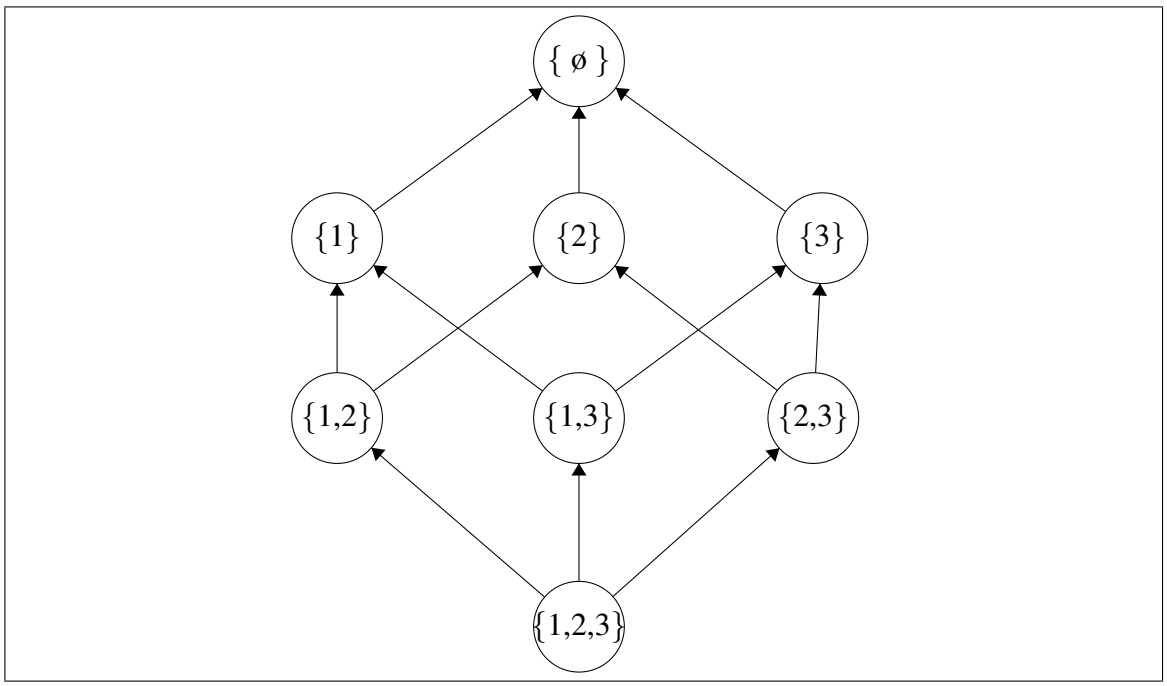


Figure 4.3. A hypercube in $\mathbb{R}^3$

First, we notice that we can represent the unit hypercube in d dimensions as a new graph that abstracts it, similar to a lattice. 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 $m : \mathbb{R}^d \to 2^{\mathbb{N}}$ 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 nodes. The direction goes from the set containing to the contained node. 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 a 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 H^{d-1} and the ideal of a singleton of H^d. Let i be the singleton to consider. We denote the ideal of $\{i\}$ in H^d as $I^{H^d}_{\{i\}}$. Then, we can use the function $f : I^{H^d}_{\{i\}} \to H^{d-1}$, defined as $f(S_1) := S_1 \setminus \{i\}$. 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 have that the same edges are maintained.

□

To search in a hypercube in d dimensions, the following strategy is proposed: 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 we would take at most $O(d^2)$ steps. With this, we conclude 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.*

It is known that in the hypercube the minimum 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 separator of a graph is not a lower bound for the optimal search strategy.*

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

Recall that for a graph $G = (V, E)$ and a series of responses based on infection, there is exactly one induced subset $Q \subseteq V$ that is consistent with being potentially infected. This subset may be retrieved by discarding every node that has no path to the lowest positive test or that its path necessarily goes through a node that tested negative. Note that Q always has a root r_Q. If there are no positive tests, then r_Q is the root of G. If Q is constructed iteratively and tests are allowed only in a potentially infected zone, then r_Q is the last positive node tested.

Definition 4.5 ((balanced) testing separator number)**.** *The (balanced) testing separator number of a graph $G = (V, E)$, denoted by $ts(G)$, is defined as the smallest integer k such that for every $Q \subseteq V$ obtained from a series of tests in G, and for every (balanced) separator $S_Q \subseteq Q$ of $G[Q]$ with $|S_Q| \leq |Q|$, $0 < c < 1$ it is possible to trace the infection to a connected component of $G[Q/S_Q]$ or S_Q in at most k steps. In other words, for every subset obtained from a series of tests and a (balanced) separator S_Q, it is possible to trace the infection to exactly one of the connected components induced by S_Q with at most k steps.*

Theorem 4.2. *The testing separator number is a lower bound of the height of the optimal search strategy for G.*

PROOF. Let opt be the height of the optimal search strategy of G and Q the subset of V obtained from a series of tests in G where $ts(G[Q])$ is the maximum. We can always use the search strategy of $G[Q]$ to trace the infection to a connected component of $G[Q/S_Q]$ or

S_Q because this strategy gives us more information than we need. Let opt_Q be the height of this search tree. Now we can use the lemma 2.2, which states that the height of the optimal strategy of a subgraph is less than or equal to that of the graph to conclude that the number of steps used is less than or equal to opt. That is, $opt_Q \leq opt$. Trivially, we have $k \leq opt_Q$, which ends the proof. □

Furthermore, we can construct an instance where $ts(G) = opt$, with opt as the height of the optimal search strategy. Consider the following graph.

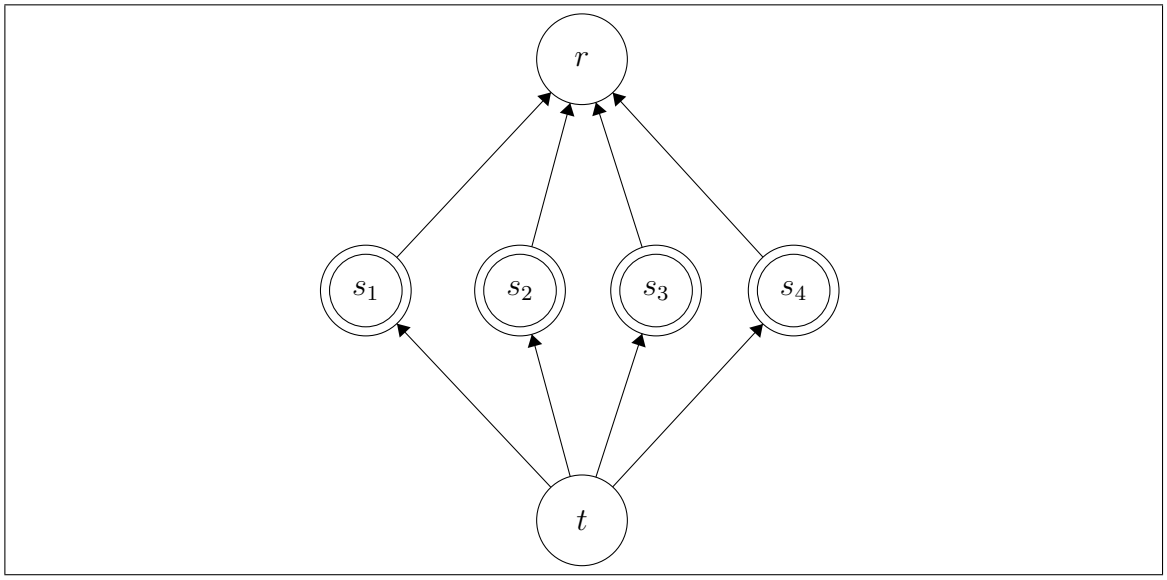


Figure 4.4. A simple instance where the bound of the testing separator number is achieved

The nodes denoted by s_i form the balanced separator. We consider the following partitions: the first is formed only by the root node r, the second is the separator and the last one has only the node t. We note that testing node r does not provide any information. As for the other nodes, an adversary can answer no for every test, but the last one, if node t is not left until the end. In that case, the adversary will answer yes only to the last node

of the separator and any answer for t. This means that the height of the optimal strategy opt and $ts(G)$ is the same value. In this case 5.

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

PROOF. Let S be a balanced separator of G. By definition, it is possible to determine the partition where the infection is in at most $k = ts(G)$ tests. The same process may be used on $G[Q]$, where Q is the ideal of the lowest known positive test that is not in the robust ideal of the nodes that were negatively tested. This idea can be applied recursively, eliminating a constant fraction in each iteration with at most k steps. This implies that infection can be found in the $k \log(|V|)$ tests. □

4.2.1. An algorithm to separate the separator

As we have seen in Theorem 4.1 and Corollary 4.3, testing a whole separator to divide the graph is not a good strategy if we want to achieve 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 half of our graph.

Lemma 4.3. *Consider a rooted DAG G with bounded treewidth and maximum in-degree Δ. Then there exists a strategy to find the infected node in at most $\mathcal{O}(ts(G) \log(|V|))$. Furthermore, the decision tree can be found in polynomial time.*

PROOF. Given a graph G and a separator S , the connected component where the infection resides can be found by brute force in at most $\mathcal{O}(2^{ts(G)}|V|^{ts(G))})$ time by exploring different search strategies to test. As the treewidth and the degree are bounded, the testing separating number is bounded by $|S| + \Delta$. Recall that, as in the algorithm **??**, we can always test the complete separator and then the predecessors of the lowest topological node.

This implies that time is dominated by $\mathcal{O}(2^{|S|+\Delta}|V|^{|S|+\Delta})$, which is a polynomial, since the separator size is bounded by the treewidth (Gruber, 2013).

To obtain the separator, we can simply use an approximation algorithm that runs in polynomial time, such as the one presented by Feige et al. (Feige, Hajiaghayi, & Lee, 2005). It is not a problem to have a separator that is larger than the separator number, since the testing separator number is defined over any (balanced) separator. □

Note that the procedure used in the proof is not practical since it uses brute force. However, it is important as it can act as a useful upper bound on the optimal height for a strategy in the worst case.

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 the real world. We again consider the case with k as a natural number. We present three algorithms that aim to solve the problem using different greedy approaches.

5.1. A Natural Greedy Algorithm

Given k nodes to choose to test to find the infection, a very 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 there is no uncertainty. They showed that in that scenario, the algorithm that chooses the node v that minimizes the size of $\max\{w(T_v), w(T - T_v)\}$ in each iteration, with w as a function that calculates the probability that each subgraph has infection, achieves a 1.62-approximation of the optimum value for the average case.

We first show that this algorithm cannot give guarantees of optimality for the average case or the worst case when considering uncertainty. In this case, the value to minimize would be $\max\{w(I_v), w(G - \bar{I}_{r,\{v\}})\}$. Consider the counterexample shown in Figure 5.1.

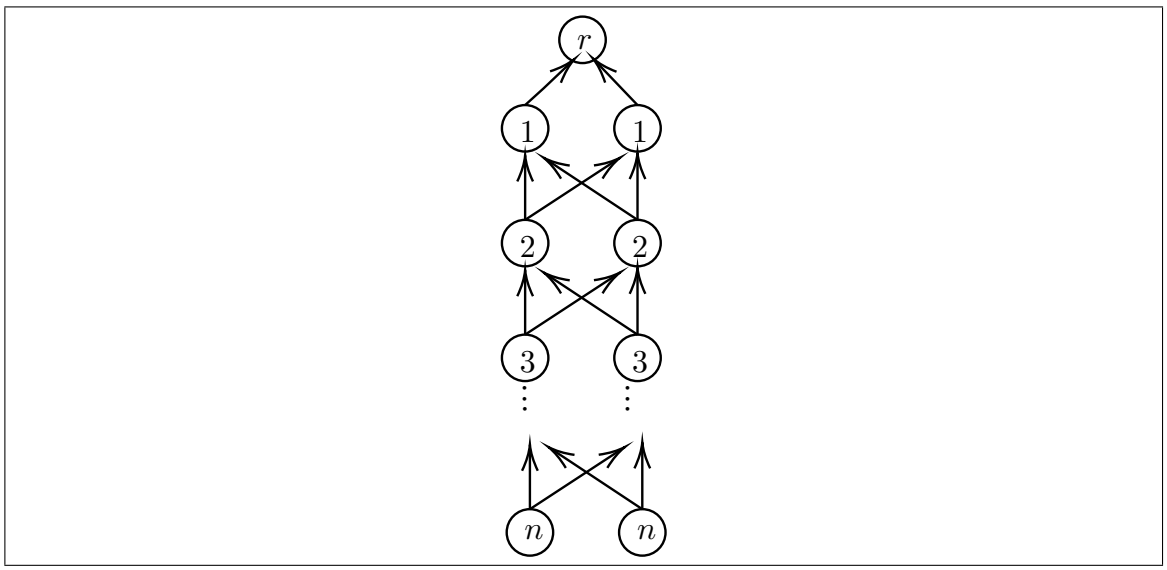


Figure 5.1. Each node i has weight $1 + i\epsilon$. Then, the Greedy Algorithm would start at the bottom and make its way up. This takes linear time. A better strategy would be to test the three elements at the center and do a binary test querying if the infection is at the top or bottom half. This would take logarithmic time.

Considering that there are no guarantees in the values that may be obtained from the algorithm, we now generalize it to be able to operate over k samples and an uncertain graph. However, while the running time is naturally $\mathcal{O}(|V|^k)$, we introduce some modifications that may help to avoid that value and get the sample set in $\mathcal{O}(k|V|^2)$. This is done by relaxing the restrictions. We show in Appendix B that this modification in practical terms has no effect on the values 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.

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

$S \leftarrow r$;

while $|S| \leq k + 1$ **do**

  aux $= -1$, auxv $= -\infty$;

  for $v \in V \setminus S$ **do**

    if $|I_u(G), :, u \in S \cup I_v(G)| >$ *auxv* **then**

      aux $= v$, auxv $= |I_u(G), :, u \in S \cup I_v(G)|$;

    end

  end

  if *aux* $\neq -1$ **then**

    $S \leftarrow S \cup aux$;

  end

  else

    break;

  end

end

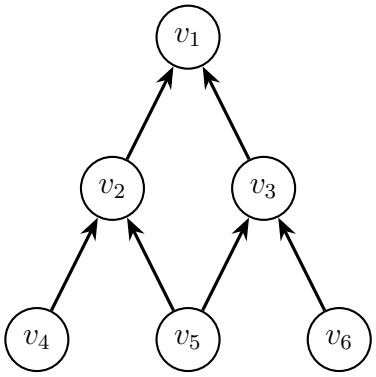
return S;

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 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 may be 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. This is the idea that is implemented in Algorithm 3. The function w calculates the sum of the weight

of the paths in a set of paths for the average case scenario. In the worst case, this function just counts the number of different paths.

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 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. 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;

$P \leftarrow \emptyset$; $aux \leftarrow \infty$;

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

  for *Every plausible answer* q_i *of querying* R *in* G **do**

    $r_i \leftarrow$ set of different paths in subgraph induced by q_i in M;

  end

  if $\max_i w(r_i) \leq$ *aux* **then**

    $aux \leftarrow \max_i w(r_i)$;

    $P \leftarrow R$;

  end

end

return P;

Note that in the case without uncertainty, that is, a tree and $k = 1$, the algorithm simply tries to find a node that minimizes the value of $\max\{w(T_v), w(T - T_v)\}$. 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 as 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. Then the algorithm would take a top-bottom approach. In the first iteration, a node of the first row would be queried, as 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 one, 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 slightly different model where the probabilities of infection are in the paths, 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 $k = 1$ and with a uniform distribution over the nodes, 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 $\mathcal{O}(n^2)$ steps. This allows the algorithm to perform in polynomial time. 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;

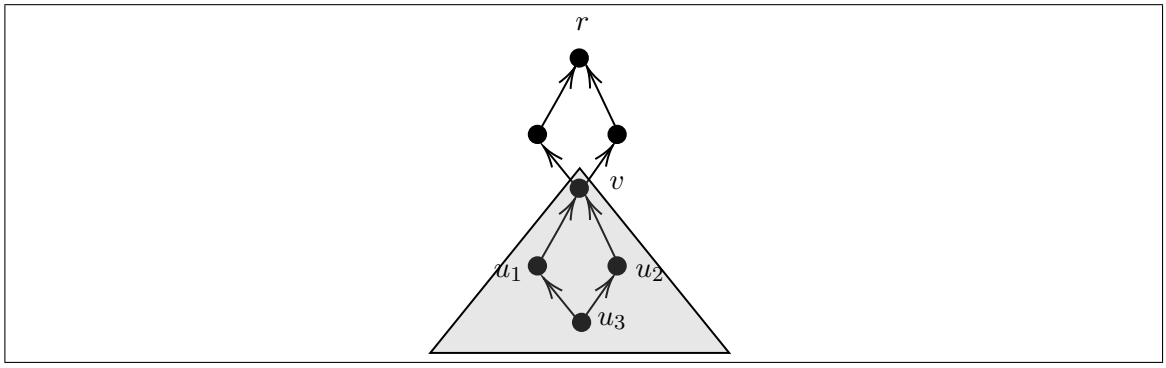


Figure 5.3. In grey the ideal of node v. Note that although the number of paths that pass through v is 10, the number of paths that hang from r is only 5. These are v, u_1v, u_2v, u_3u_2v, and u_3u_1v

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 and its path matrix M. We will again assume a probability distribution on the paths of M. Now, we can partition M based on the response when querying a node v_i. 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 find. More concretely, we see that for each response of a node v_i, we can compute a list of possible infected nodes along with their probability. We then calculate the entropy of that list. More precisely, we can define the entropy of a matrix of paths $M = C_1, C_2, \ldots, C_n$ with end nodes $E = e_1, e_2, \ldots, e_n$ in the following way:

$$entropy(M) := -\sum_{c \in E} p(c) \log p(c)$$

Then the algorithm will minimize the maximum entropy of all possible partitions induced by querying a tuple. The pseudocode can be found in Algorithm 5.

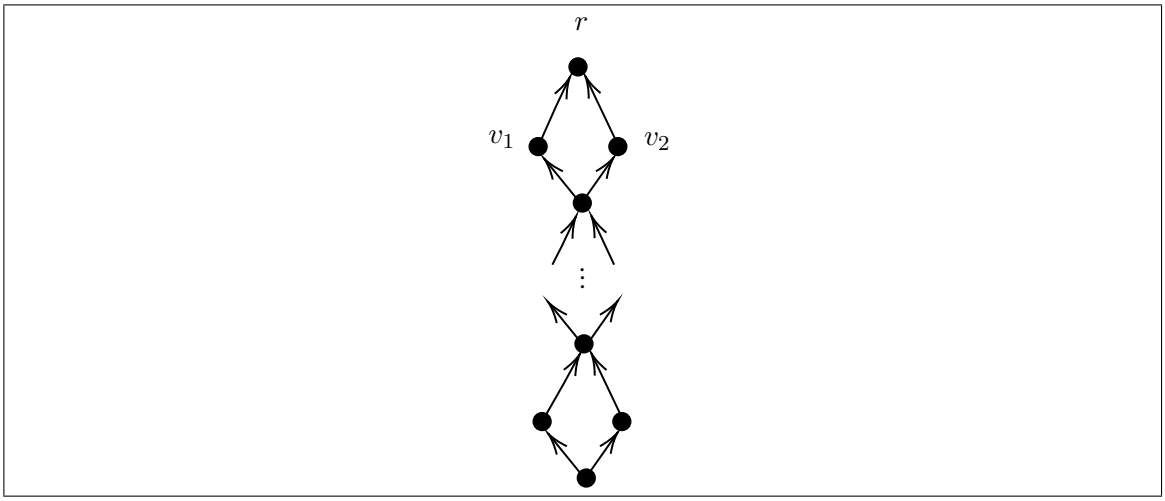


Figure 5.4. A simple instance where the Entropy Algorithm (E) achieves a $\mathcal{O}(\log_2(|V|))$ height in its decision tree and the Path Separation Algorithm (PS) achieves an $\mathcal{O}(|V|)$ height. Consider that every node has the same probability of being infected and that every path to a node has the same probability of being the infected path. 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.

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;

$P \leftarrow \emptyset$; aux $\leftarrow \infty$;

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

  for *Every plausible answer* q_i *of querying* R *in* G **do**

    $r_i \leftarrow$ set of different paths in subgraph induced by q_i in M;

  end

  if $\max_i entropy(r_i) \leq$ *aux* **then**

    aux $\leftarrow \max_i entropy(r_i)$;

    $P \leftarrow R$;

  end

end

return P;

We see again that when $k = 1$ and the DAG corresponds to a tree, the algorithm is the same 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.

6. EXPERIMENTS AND RESULTS

6.1. Random graph generation

To test the performance of our method in several graphs, beyond the instance of San Pedro de la Paz, 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 also discard the generation of random points in a plane. Instead, 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; Li, 2005); 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 $0 \leq C \leq 1$. 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 contradicting the selected tree);

$E' =$ 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;

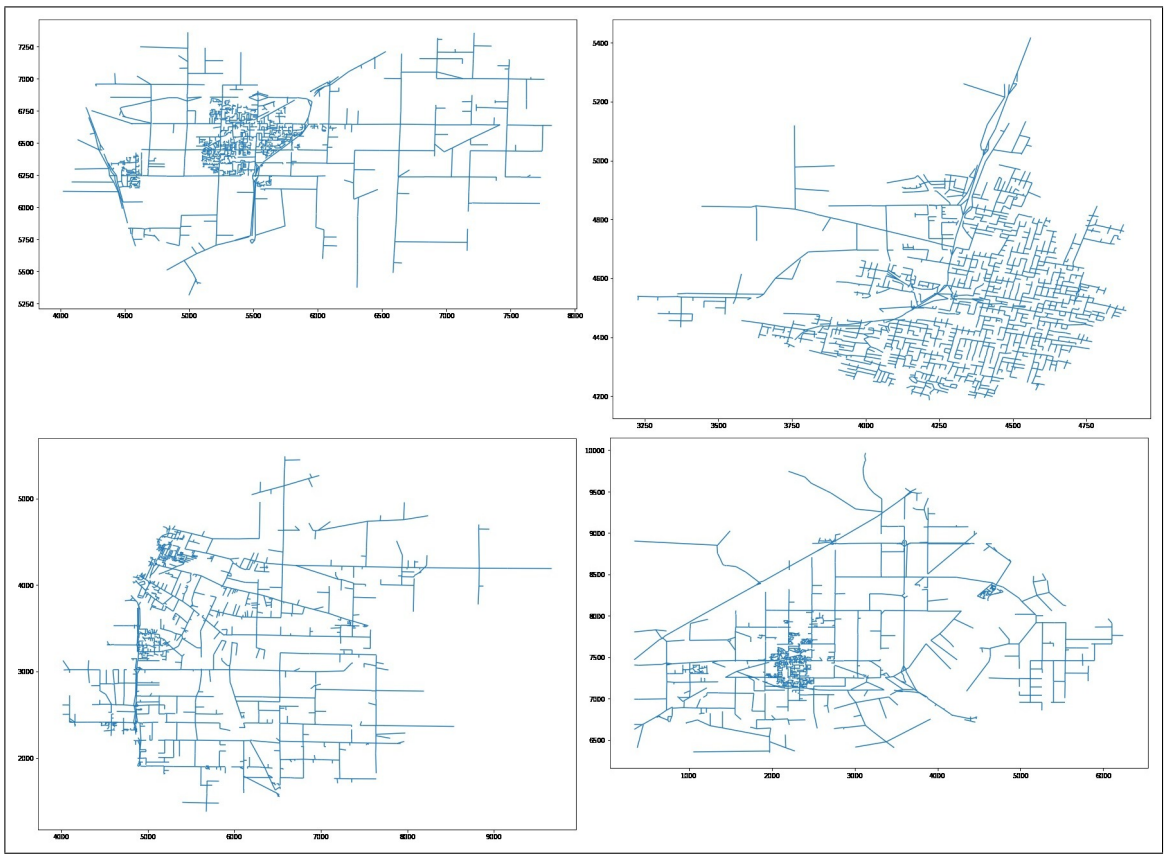


Figure 6.2. Examples of random graphs generated from the San Joaquín County network by the Algorithm 6

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. 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 for the scenario with $0\%, 5\%, 10\%$ and 20% of 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. Note that this value is optimal over the worst case scenario. The codes are run in Python 3 over 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.

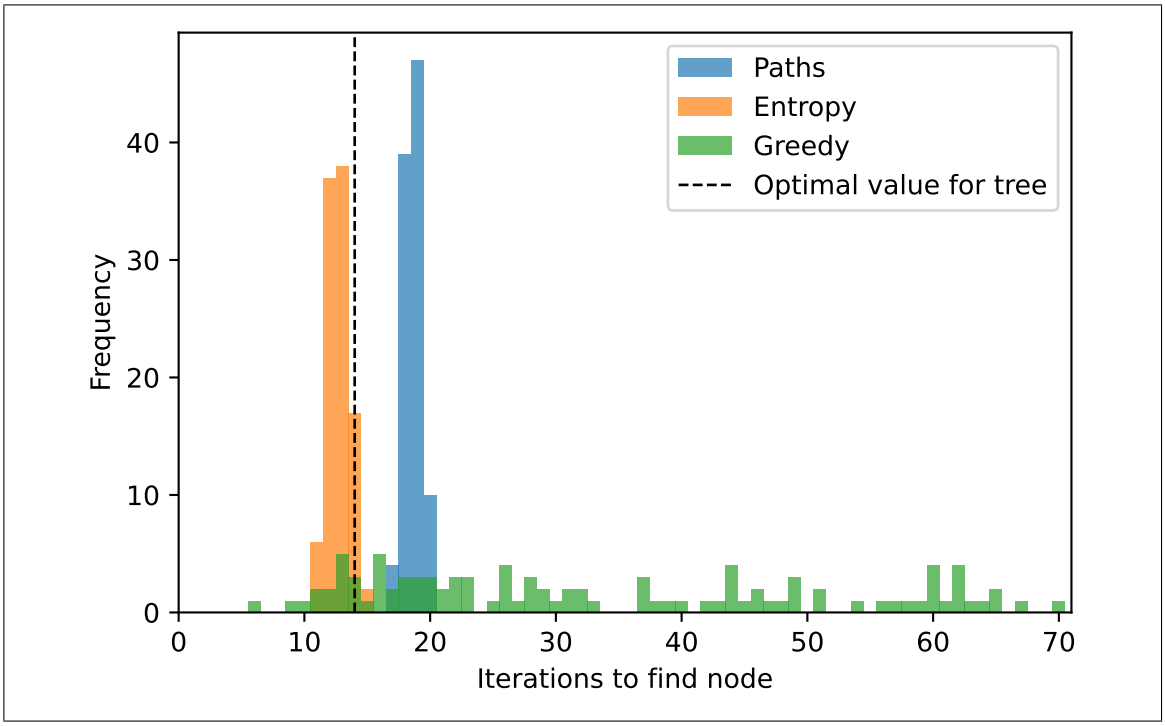


Figure 6.4. Results of simulation in San Pedro de la Paz

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 than the optimal value in a random subjacent tree. This result is 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.

Algorithm / Statistic	Mean	Max	90th percentile
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

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 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 value WC search	10.96	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.

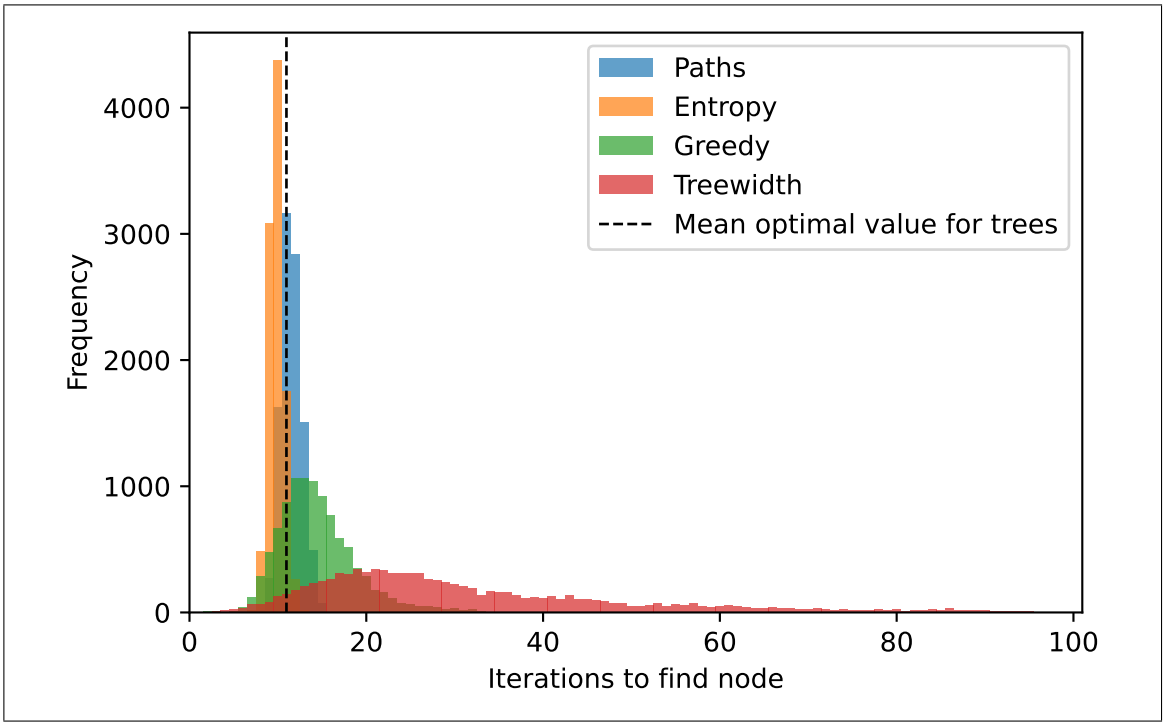


Figure 6.5. Results of simulations in 100 random graphs of 500 nodes with 20% extra edges and $k = 1$. The histograms are consistent with the results in the graph of San Pedro de la Paz. However, the Path Separation Algorithm has results very close to the Entropy Algorithm with a lower running time.

The 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. 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 Entropy Algorithm results 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.

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 1.5. 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.

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. This framework is known as Group Testing, which is an area of study that has a great deal of literature. Nevertheless, note that 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 of them 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 topologically lowest nodes, and the test on each iteration is applied to it. This captures the idea that we are only interested in the last infected node of the infected path. The formalization of the idea is presented in Algorithm 7 and an example in Figure 7.1.

Algorithm 7: $GT(G, r)$

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

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

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

$T \leftarrow \lceil |V|/2 \rceil$ lowest nodes of $sort$;

Test set T;

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

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

end

else

$G \leftarrow G[T]$;

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

end

end

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

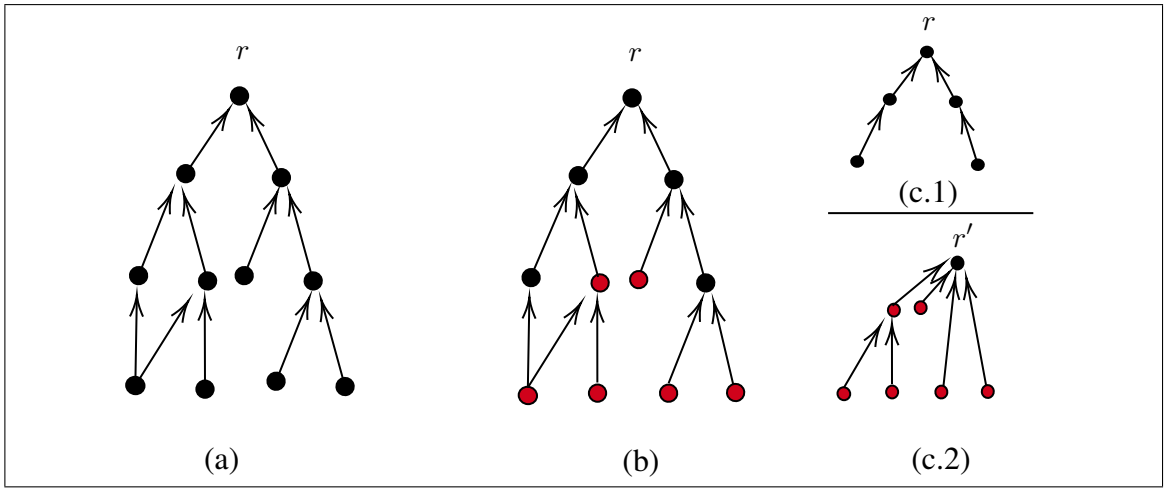


Figure 7.1. In (a) is the original graph G with its root r. On (b) is the graph with the set T to test in red. On (c.1), the graph of the next iteration if $t(T) = 0$. In (c.2), the graph for the next iteration if $t(T) = 1$ with the new *dummy* root r'.

Lemma 7.1. *The 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 may be incorrectly discarded. The node v^* cannot be in T, as it would yield $t(T) = 1$ and would remain for the next iteration. Then we must have $t(T) = 0$. But that cannot be eliminated v^*, since the nodes are removed topologically for the next iteration. We reached a contradiction. □

Lemma 7.2. *Let G be a set of DAG without a unique global root, and G' be a rooted version of G that is obtained by adding a dummy node as a root connecting the topologically greatest nodes of each set. Then, querying in G' is consistent in terms of feasibility and discarded elements for 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 does not add information: it will always answer yes and cannot be part of a balanced separator since its ideal is the complete graph.

For the consistency of the answers in G' for G, consider any answer for a set of queries. Then the only answer that would give a different result from performing the test separately in G would be if all nodes tested negative. In that case, the upper topological components would be connected by the new *dummy* root. This is consistent with the original graph, as testing the same nodes on it would yield the same result without the new root.

□

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 out-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 in a graph $G = (V, E)$ over 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 initial ones 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 $t(U)$ be the result of the test in U. The pseudocode is presented in algorithm 8.

Algorithm 8: $GT(G, r)$

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

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 First-fit-decreasing Algorithm

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 $t(C_1) \vee t(C_2) \vee t(C_3)$ **then**

$C \leftarrow$ partition that contains all the positive tested nodes

$G \leftarrow G[C]$

Add a dummy new root u to G and edges from every node of the set c_i to it

end

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

Perform a binary search on S

end

end

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

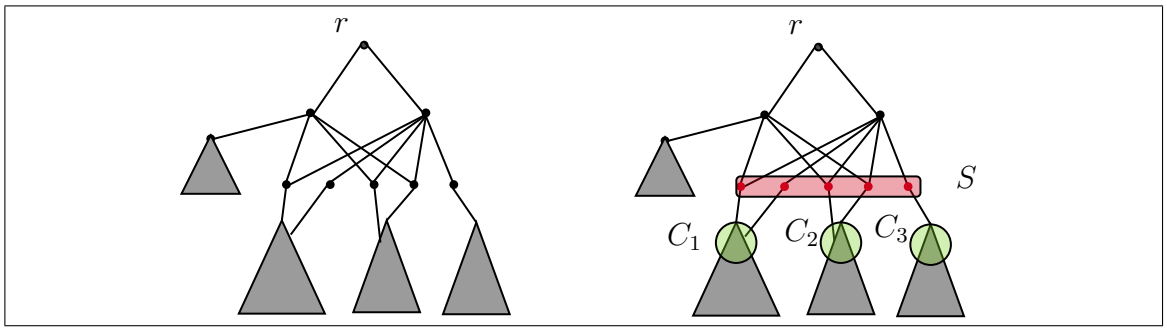


Figure 7.2. On the left is the original graph G with its root r. On the right is the graph with the separator S in red and the subset of predecessors C_1, C_2 and C_3 in green. All directions go from the lower to the upper node.

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 $t(S) = 0$ then $t(C_1) = t(C_2) = t(C_3) = 0$*

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 by the problem statement. 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. If $t(S) = 1$ and $t(C_1) = t(C_2) = t(C_3) = 0$ then the infection must be located at one of the topologically greatest nodes in S.*

PROOF. In a similar way to the last claim, let us suppose the opposite. Then there must be nodes $s_1, s_2 \in S$ such that s_1 is a predecessor of s_2. If s_2 is infected, then there would be a $vr-$path of nodes that carry the infection that passes through S. This implies $t(C_1) \vee t(C_2) \vee t(C_3) = 1$, which is a contradiction. □

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 directed rooted acyclic graph (DAG) $G = (V, E)$ and the group testing problem for searching an infected node. The algorithm 8 finds the infection in $5\lceil \log_2 |V| \rceil$ steps, with s as the balanced separator number of G. 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 in $\mathcal{O}(\log_2 |V| + \log_2(s\sqrt{\log s}))$ steps, with s as the balanced separator number of the graph.*

PROOF. First, we will prove the correctness of the algorithm. In every iteration, we discard the nodes that have every path to the root passing through a set of nodes that were all negatively tested. This means that every node discarded has no chance of 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 $t(S) = 1 \wedge t(C_i) = 0, i = 1, 2, 3$. 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 true since we can use claim 7.2 to eliminate the nodes of S that are topologically the lowest. This means that there would be no relationships of predecessors between nodes in S. As $t(S) = 1$,

then exactly one of the nodes in the set carries the infection. In the case where $t(S) = 0$ or $t(C_1) \vee t(C_2) \vee t(C_3)$, 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 see 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 the maximum of $\log_2(|V|)$ and $\log_2(s)$. 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 et al., 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 as the one presented by Boedlander (Bodlaender et al., 2016) and the balanced separator induced by tree decomposition (Robertson & Seymour, 1986). □

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 with no restrictions of any kind. If the lower bound used can be improved, the algorithm's approximation factor 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 in depth. First, we list 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 5 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 showed 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, the time needed to compute it was less, and its implementation is easier and more natural. The results are reasonable for the average case scenario, but in the worst case there is 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 an almost optimal algorithm in the case where the size of the set to test is restricted by the separator number of the graph.

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 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., ... others (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*, 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*, 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*(4), 333–340.

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

Carmo, R., Donadelli, J., Kohayakawa, Y., & Laber, E. (2004, June). Searching in random partially ordered sets. *Theoretical Computer Science*, *321*(1), 41–57. Retrieved 2021-04-22, from `https://www.sciencedirect.com/science/article/pii/S0304397504001203` (Number: 1) doi: 10.1016/j.tcs.2003.06.001

Cheraghchi, M., Karbasi, A., Mohajer, S., & Saligrama, V. (2012). Graph-constrained group testing. *IEEE Transactions on Information Theory*, *58*(1), 248-262. doi: 10.1109/TIT.2011.2169535

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, April). Improved Approximation Algorithms for the Average-Case Tree Searching Problem. *Algorithmica*, *68*(4), 1045–1074. Retrieved 2022-11-11, from http://link.springer.com/10.1007/s00453-012-9715-6 doi: 10.1007/s00453-012-9715-6

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, July). Edge ranking and searching in partial orders. *Discrete Applied Mathematics*, *156*(13), 2493–2500. Retrieved 2021-04-22, from https://www.sciencedirect.com/science/article/pii/S0166218X08001339 (Number: 13) doi: 10.1016/j.dam.2008.03.007

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, January). *On Balanced Separators, Treewidth, and Cycle Rank.* arXiv. Retrieved 2022-11-11, from http://arxiv.org/abs/1012.1344 (arXiv:1012.1344 [cs, math])

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. , 697-705. doi: 10.1109/INFCOM.2007.87

Karbasi, A., & Zadimoghaddam, M. (2012). Sequential group testing with graph constraints. In *2012 ieee information theory workshop* (p. 292-296). doi: 10.1109/ITW.2012.6404678

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., Ferraro, G. B., Veneri, C., Bonadonna, L., ... Suffredini, E. (2020). First detection of sars-cov-2 in untreated wastewaters in italy. *Science of the total environment*, *736*, 139652.

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

86–103. (Number: 1)

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 *Soda* (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*.

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.

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

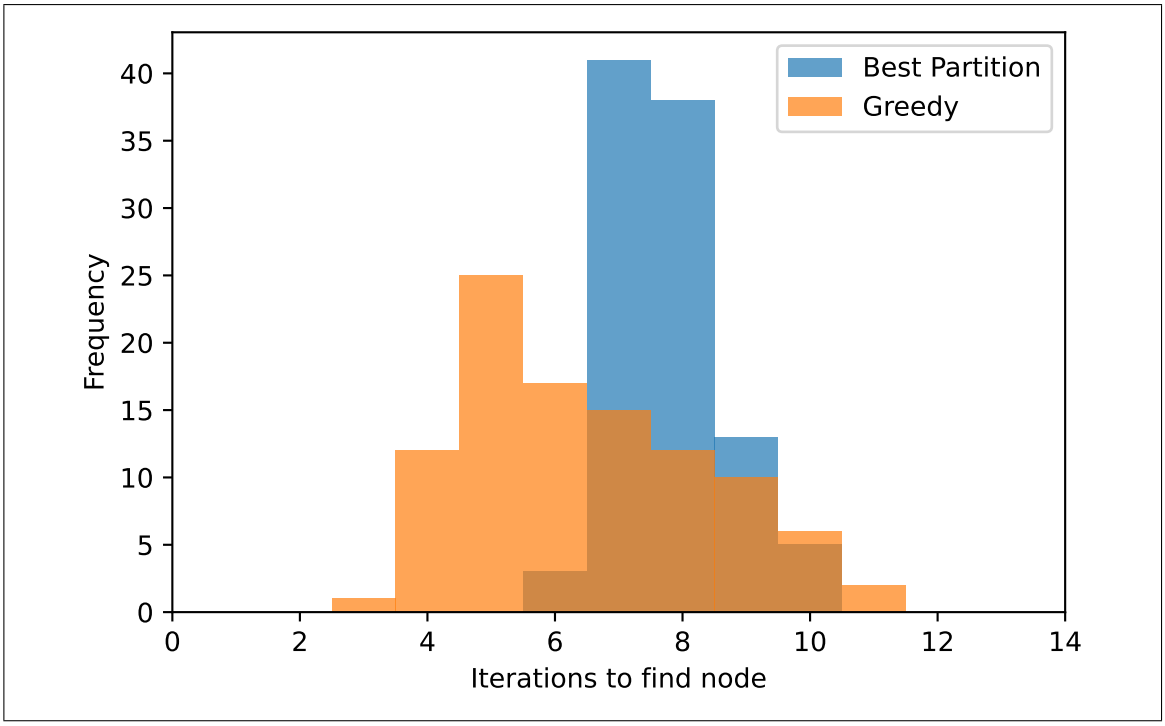


Figure B.1. Histogram of iterations for searching in a subtree of 176 nodes of the network of San Pedro de la Paz for the Greedy Algorithm (G) and the Best Partitioning Algorithm (B) for $k = 1$. Note that the mean of the number of iterations is lower in (G), while the maximum value is lower in (B).

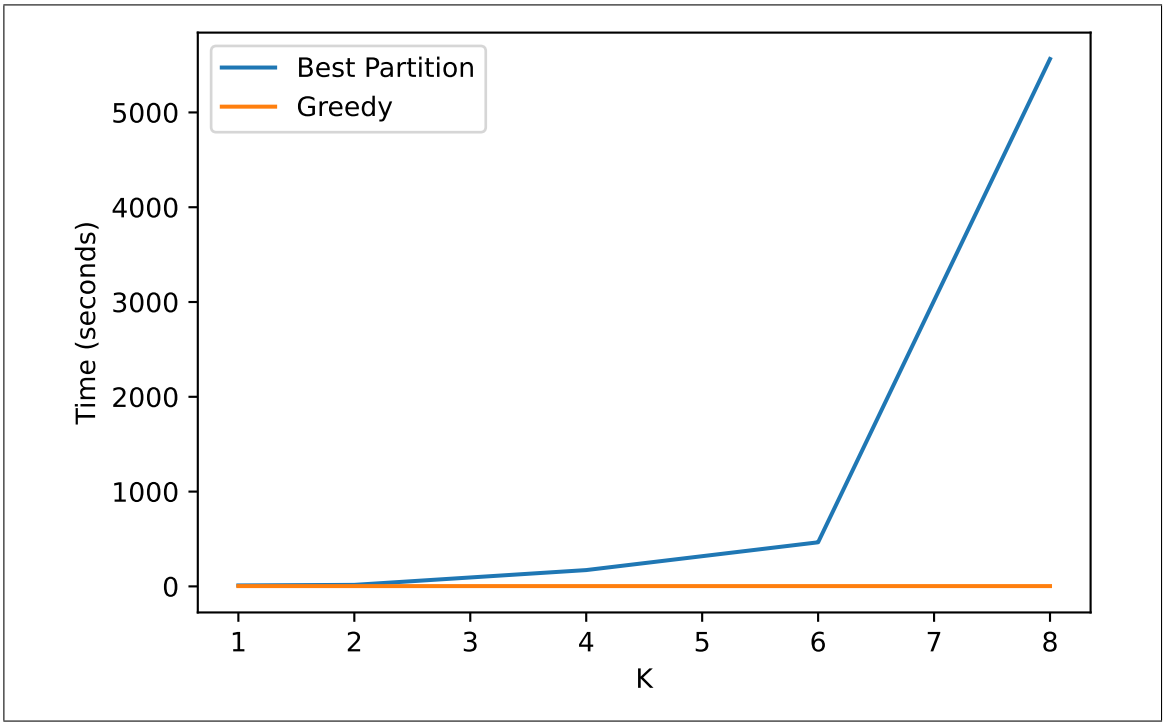


Figure B.2. The time needed to cumpute the optimal partition for the Best Partiton Algorithm grows exponentially in the value of K. On the other hand, the time needed to compute the partition for the Greedy Algorithm remains almost constant for different values of K.

C. RESULTS OF SIMULATIONS

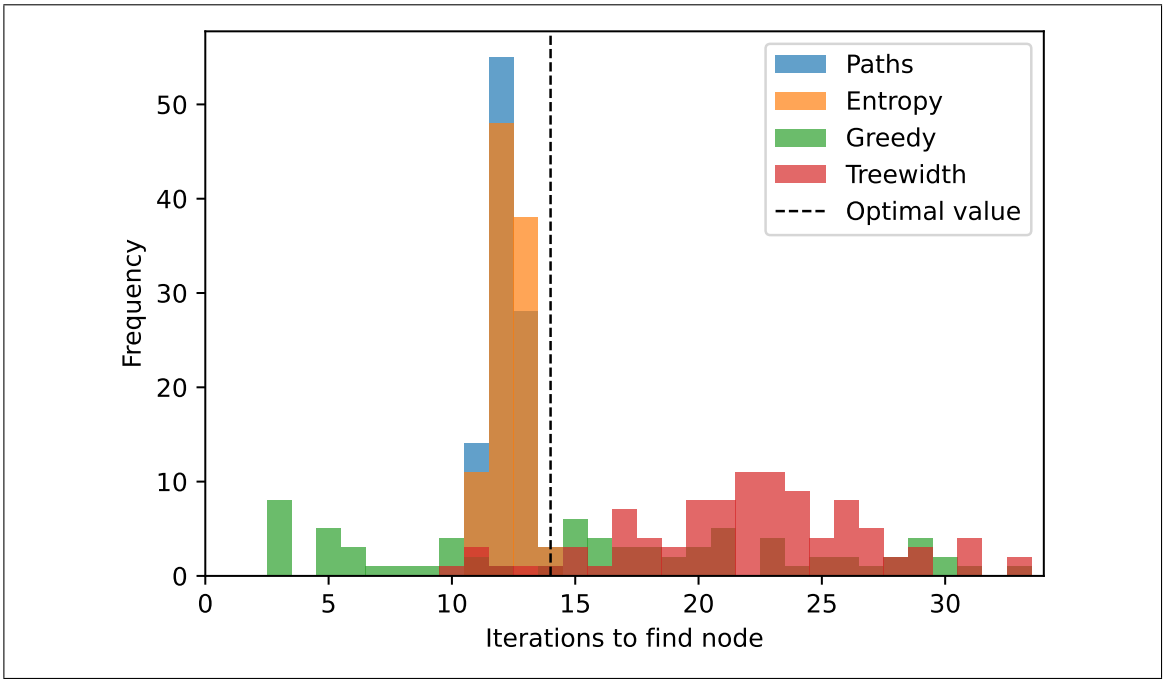


Figure C.1. Simulation results in San Pedro de la Paz for a subyacent tree with $k = 1$. Note that the Entropy and Path Separation Algorithm (that in the case without uncertainty are the same) do not have more iterations than the optimal number of iterations in the worst case.

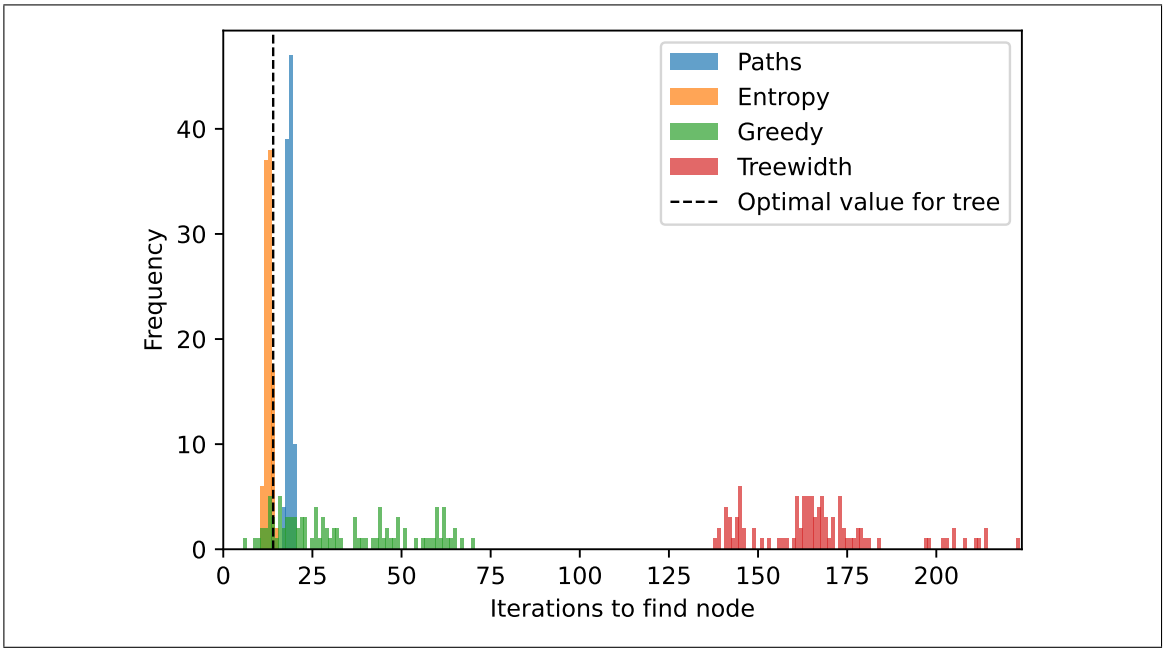


Figure C.2. Simulation results in San Pedro de la Paz for $k = 1$ including iterations of Treewidth Algorithm.

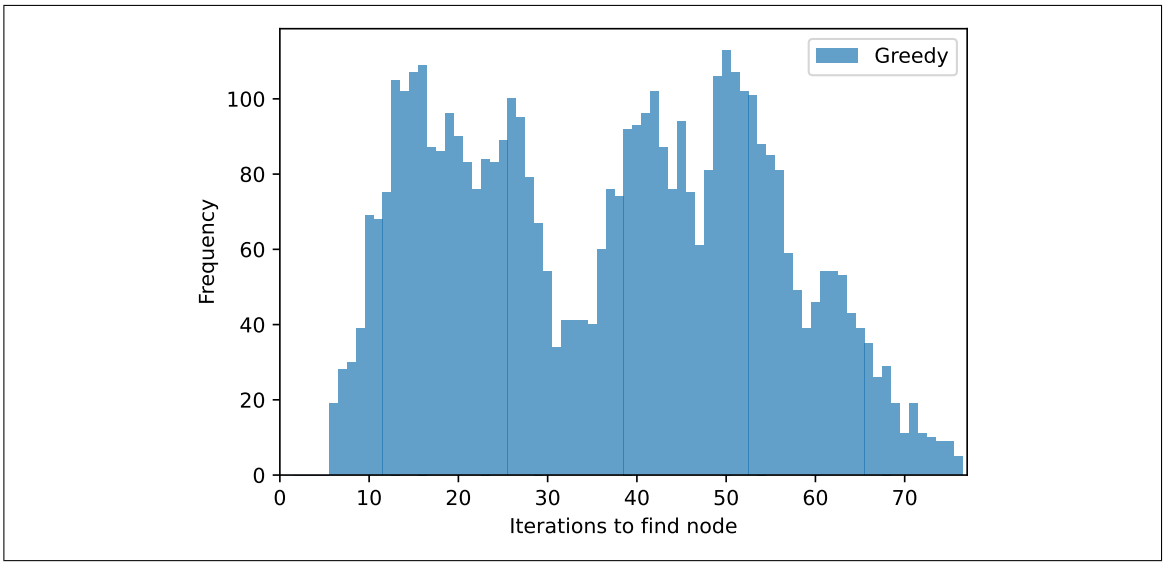


Figure C.3. Simulation results of all nodes in San Pedro de la Paz for $k = 1$ for the Greedy Algorithm.

Algorithm / Statistic	Mean	Max	90th percentile
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 $k = 1$.

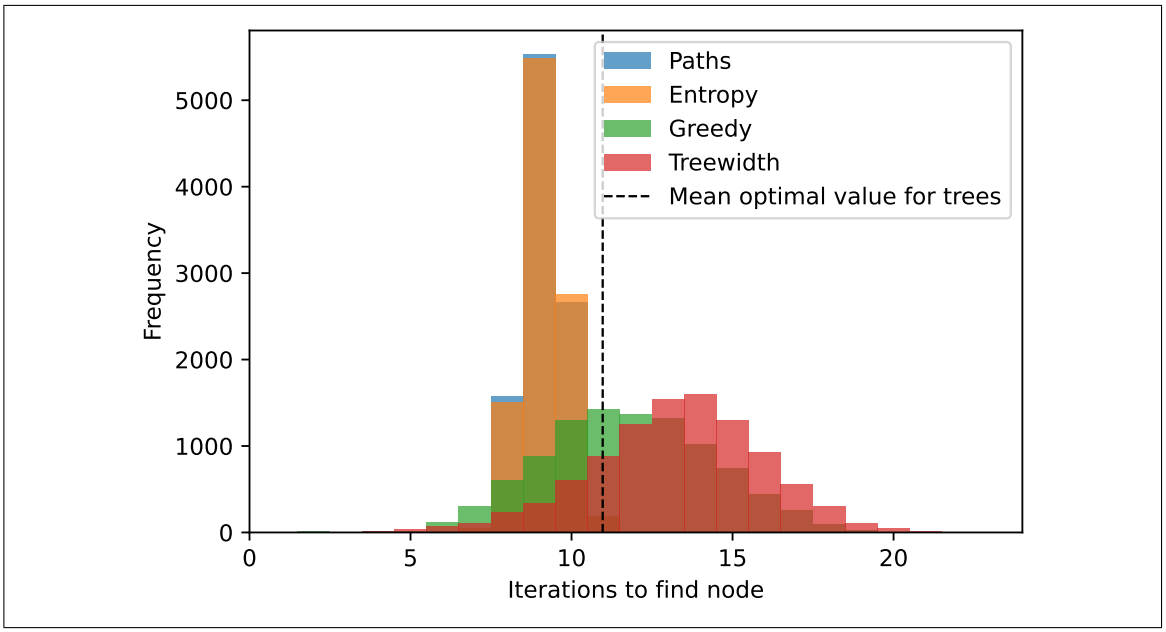


Figure C.4. Simulation results in 100 random graphs of 500 nodes, no extra edges and $k = 1$.

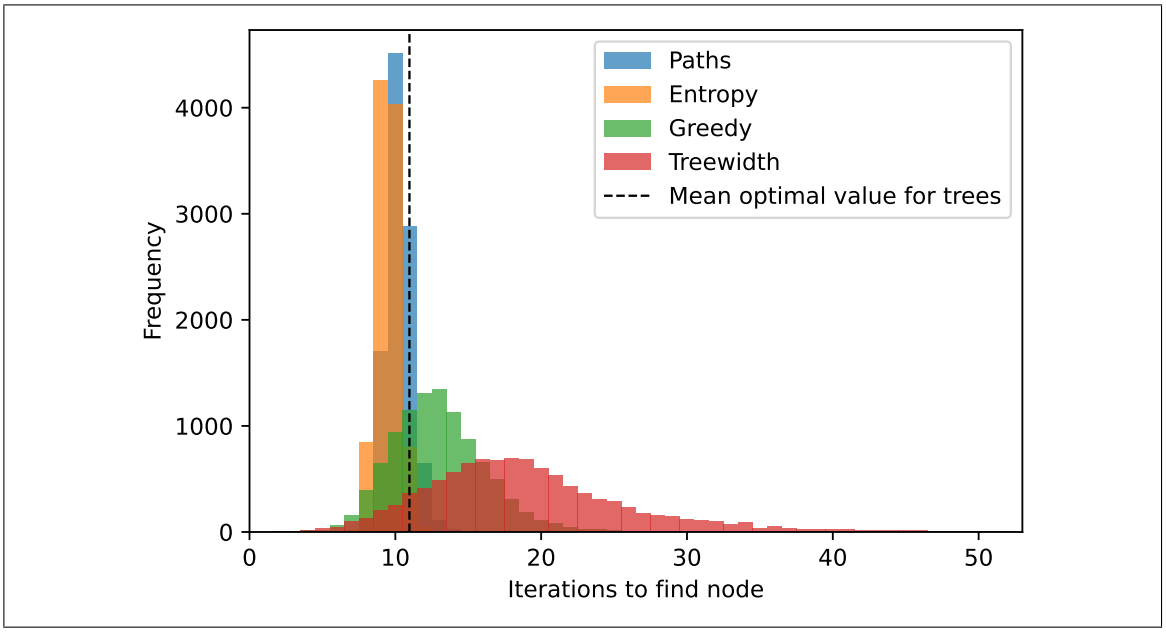


Figure C.6. Results of simulations in 100 random graphs of 500 nodes with 10% extra edges and $k = 1$. The histograms are consistent with the results in the graph of San Pedro de la Paz.

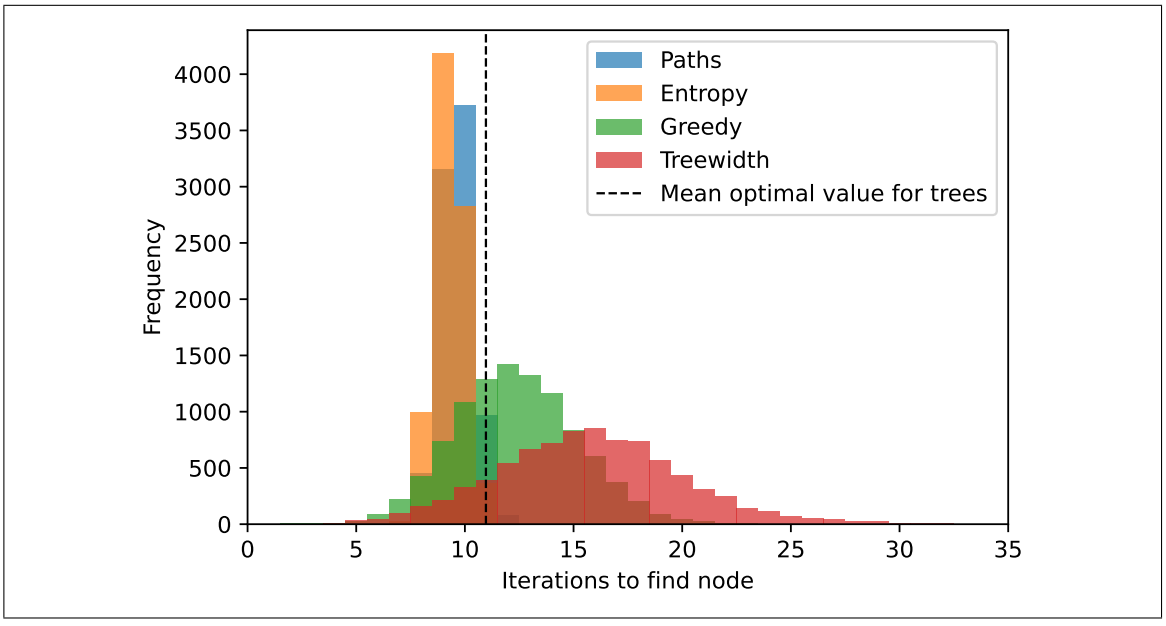


Figure C.5. Simulation results in 100 random graphs of 500 nodes, with 5% extra edges and $k = 1$.