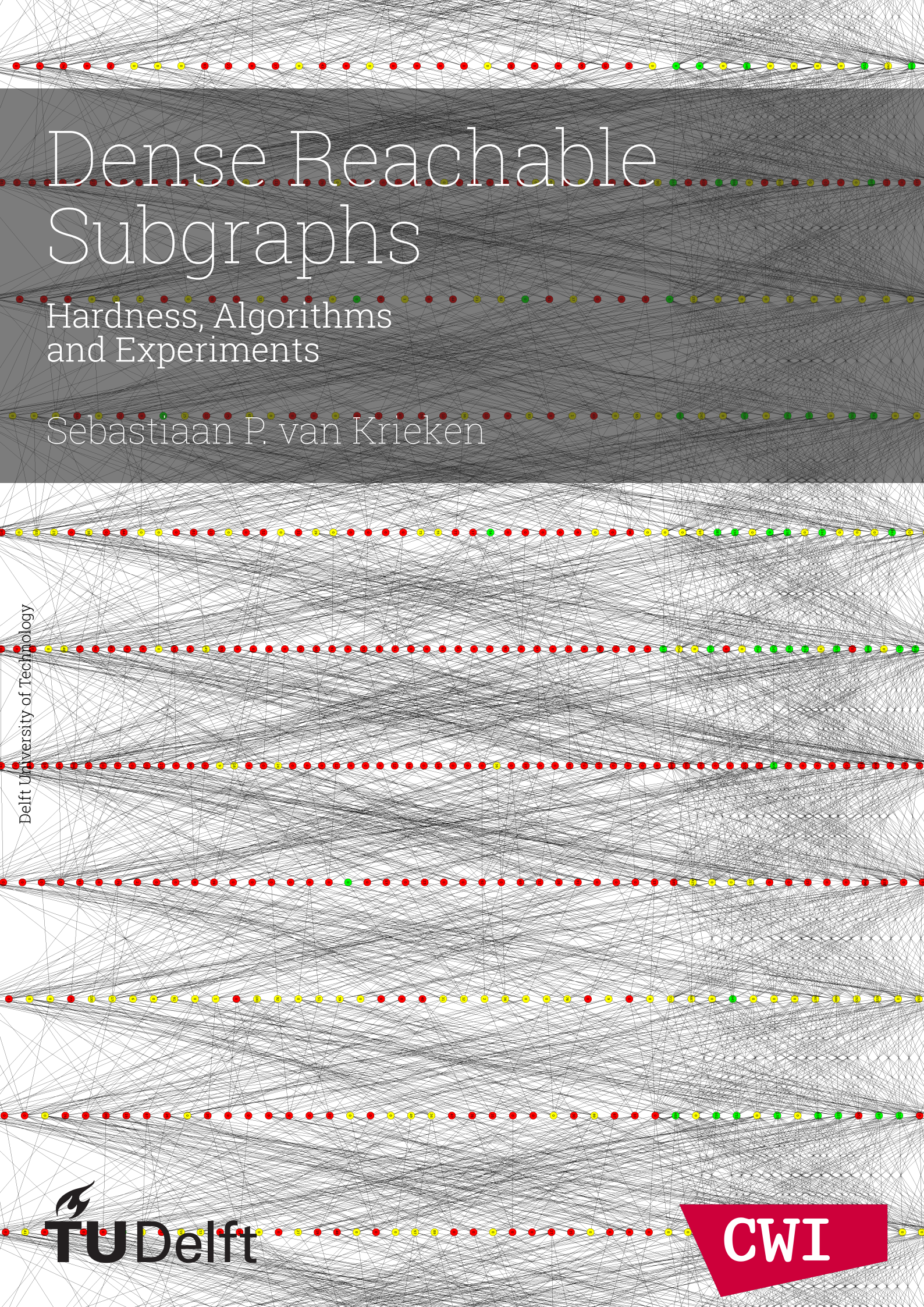


Dense Reachable Subgraphs

Hardness, Algorithms and Experiments

Sebastiaan P. van Krieken

Delft University of Technology

TU Delft

CWI

Dense Reachable Subgraphs

Hardness, Algorithms and Experiments

by

Sebastiaan P. van Krieken

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 17, 2025 at 13:30.

Student number:	5876494	
Project duration:	June, 2024 – July, 2025	
Thesis committee:	Dr. ir. L. J. J. van Iersel,	TU Delft, supervisor
	Dr. S. P. Pissis,	CWI & VU, supervisor
	H. Verbeek (MSc),	CWI & VU, supervisor
	Dr. J. L. A. Dubbeldam,	TU Delft, committee member

This thesis was conducted externally at the Centrum Wiskunde & Informatica (CWI), the National Research Institute for Mathematics and Computer Science.

Cover: An example of a solution to the Predecessor Dense Reachable Subgraph problem. Rendered using the SFML C++ library.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

We introduce two novel optimization problems on vertex-weighted multilayer graphs: the Predecessor Dense Reachable Subgraph (PDRS) problem and the Neighborhood Dense Reachable Subgraph (NDRS) problem. In PDRS, the aim is to find a subset of vertices that maximizes the ratio of the total vertex weight to the size of its predecessor neighborhood, under reachability constraints. A subset is considered reachable when it is a union of paths from the first layer to the last. The NDRS variant is equivalent to the PDRS problem, but the size of the entire neighborhood is used for the denominator of the objective function. The motivation for these problems comes from money laundering detection. We prove NP-hardness by reduction from the Vertex Cover problem and formulate MILPs that give exact solutions to the problems. Additionally, we show that the single path variant, where the solution space is restricted to single paths from the first layer to the last, can be solved in polynomial time using dynamic programming. We introduce a heuristic algorithm called Greedy Single Paths, which starts with the empty graph and uses an exact algorithm for the single path variant to iteratively grow the solution. We also introduce a heuristic algorithm called Greedy Peeling. This algorithm starts with the whole graph and iteratively peels vertices from the allowed neighbor set. The algorithms are implemented and tested on synthetic data sets to assess their performance in terms of running time and solution quality.

Contents

1 Introduction

1.1. Background and Motivation

The field of Combinatorial Optimization concerns itself with finding efficient algorithms to problems with discrete solutions. It has many applications, from classic examples such as finding the shortest path (Dijkstra 1959), optimal job assignment (Kuhn 1955) and efficient routing (Applegate et al. 2006), to more recent examples such as the Ocean Cleanup project (Hertog et al. 2025) and designing sustainable electricity grids (Aslam, Altaweel, and Nassif 2023). Many real-world problems can be modeled as mathematical optimization or decision problems. Once such a model has been created, the next key step is to design algorithms that find optimal or approximate solutions. These algorithms often differ in terms of computational speed, memory usage and output quality. Therefore, such optimization problems are analyzed from many different angles and multiple algorithms are developed, each with their own advantages and disadvantages. Research in this field is not only focused on designing algorithms, it also occupies itself with theoretical results, such as proving that no exact algorithm whose computation time scales polynomially exists or proving that a certain algorithm can always find a solution within a certain approximation ratio of the optimal solution.

In this thesis, we will introduce a novel optimization problem on networks. The motivation for the problem comes from the issue of money laundering detection in bank networks. Banks have automatic systems that will detect when large sums of money are transferred from one bank account to another. Criminals often elude these automatic systems by spreading their money over many bank accounts. The money flow then is convoluted, but eventually ends up in the target bank accounts. This is a process called ‘smurfing’, where the extra bank accounts used for transferring the smaller amounts are known as ‘smurfs’. A characteristic of such a ‘smurf’ network is that the relevant bank accounts have a high degree of transfers among themselves, yet a small degree of transfers with ordinary bank accounts. Furthermore, these accounts often display abnormal behaviors, such as a high transfer stream of money, but little money remains in the bank account at any time.

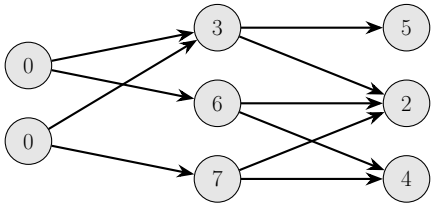
We have modeled this money laundering detection problem as a combinatorial optimization problem on multilayer graphs. The input is a directed multilayer graph in which the nodes have weights. The nodes of the graph represent the bank accounts and the edges indicate the existence of money transfers between two accounts. The weights of the nodes can be interpreted as a measure of how suspicious the bank account is.

The goal is to find a subset of nodes with high suspiciousness and little interaction with nodes outside of the chosen subset. More specifically, we try to maximize the suspiciousness divided by the size of the subset and the bank accounts that transferred money to the subset. Mathematically speaking, we aim to maximize the ratio between the sum of the weights and the size of the union of the subset and its predecessor neighborhood.

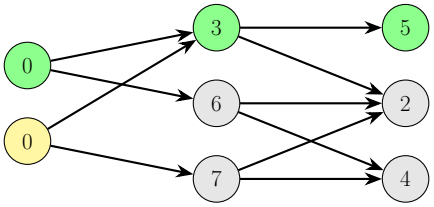
We also have a variant where we consider the bank accounts that received money from the accounts in our subset as well. In this case, we take the full neighborhood size as the denominator.

Lastly, we want such a subset to have paths from the first layer to the last layer for every node. This extra constraint is added to model the transfer of money from the first layer to the last layer. One interpretation for using a graph with layers instead of an ordinary directed graph is that banks can generate such a multilayer graph by taking all the paths of a certain length in their bank account network. Although the money laundering detection gave the motivation for the optimization problem, the thesis is focused on theoretical results and algorithms for the optimization problem in its own right, and not just for this specific application, since the optimization problem may have other applications.

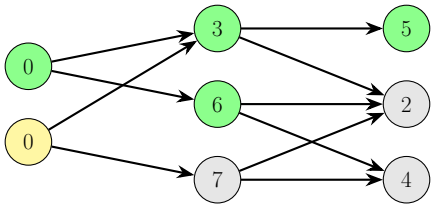
Example Here we give an example of our optimization problem. As *neighborhood* of the selected set, we consider only the predecessors of the set, i.e. the nodes that transferred money to the set. The nodes in the selected set are colored green and if a node is yellow, it is a predecessor of a node in the selected set, but not itself part of the selected set. The weights of the nodes are displayed inside the circles.



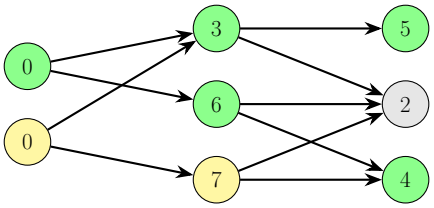
(a) The initial graph.



(b) The sum of the weights of the selected set is 8. The total number of nodes in the selected set plus its predecessor neighborhood is 4. The value of this set is thus $8/4 = 2$.



(c) The selected set is not accepted, because the node with weight 6 does not have a path from the first layer to the last contained in the selected set.



(d) The sum of the weights of the selected set is 18. The total number of nodes in the selected set plus its predecessor neighborhood is 7. The value of this set is thus $18/7$.

Figure 1.1: An example showing the types of subnetworks considered in our optimization problem, along with their objective values.

1.2. Related Work

Though we are dealing with a novel optimization problem, there are problems that show similarities to this problem and overlap in certain aspects.

The densest subgraph problem in general is a related area of research that contains various different algorithmic approaches. The most standard definition for density of the subgraphs is the average degree. Charikar (2000) presents a linear programming relaxation that finds the exact solution and a greedy peeling algorithm that gives a 2-approximation. Lee et al. (2010) present a survey of algorithmic techniques for finding dense subgraphs, where several definitions of density are used. Exact enumeration, heuristic enumeration and bounded approximation algorithms are showcased for the different density definitions. Lanciano et al. (2024) give a more recent overview of papers on finding dense subgraphs for many interpretations of density, though none define it in the same way as in this thesis.

Some researched problems resemble our problem more closely, either in objective function or in the graph structure. Miyauchi and Kakimura (2018) introduce an optimization problem that seeks to find dense graphs that are sparsely connected to the rest of the graph by using a density definition where a constant times the average number of edges that are leaving the subset is subtracted from the normal density. Boob et al. (2020) propose an iterative greedy peeling algorithm for the regular densest subgraph problem, for which Chekuri, Quanrud, and Torres (2020) prove that it converges to optimality. In fact, they prove that any problem whose goal is to find a subset maximizing the ratio of a supermodular

function to the cardinality of the set, converges with this scheme. Xu et al. (2021) present an algorithm for detecting dense graphs specifically for multilayer graphs, although their definition of multilayer graphs differs slightly from ours. Chechik et al. (2017) present algorithms for secluded connectivity problems, such as finding paths and Steiner trees with minimum neighborhood sizes.

The main work that this thesis builds upon is in (L. Li et al. 2024). In this paper, the **Heavy Nodes in a Small Neighborhood** problem is introduced, which is highly similar to the 2-layer case of our problem. The problem is as follows: Given a bipartite graph $\mathcal{G}(U, V, E)$, where the vertices in V have weights, find a set $S \subseteq V$ that maximizes the ratio of the sum of the weight of the vertices in S to the size of the neighborhood of S in U. The difference between this problem and the two layer version of our problem, is that the nodes in the first layer can not have weights and that the elements in the second layer do not count towards the size of the denominator of the objective function. Several different algorithmic techniques are applied to this problem. It contains a linear program that gives the exact solution in polynomial time. It contains a near-linear time heuristic greedy peeling algorithm that peels on the set U by removing the element in u that decreases the total weight of the remaining elements in V the least. This gives an approximation algorithm, dependent on the maximum degree of V. This algorithm can be repeated for arbitrary precision. Furthermore, the paper contains another near-linear time heuristic algorithm that peels on the set V. The integer linear program and the greedy peeling algorithm presented in this thesis are generalizations of two algorithms from this paper. Despite the similarities, the multilayer structure and reachability constraint make our problem significantly harder from a computability perspective.

Several different approaches have been considered for detecting money-laundering in bank networks. X. Li et al. (2020) propose the use of a multipartite graph to model the flow of money. They introduce a scalable algorithm, Flowscope, which is a greedy peeling algorithm where the priority of the vertices depends on the difference between the sum of the in-edges and the sum of the out-edges in the subgraph currently considered. The value of the edges are the total amounts of money transferred from one bank account to the next. The algorithm keeps the vertices whose inflow and outflow of money differs very little within the 'suspicious' subgraph. The difference with our approach, is that in our approach the vertices have weights as initial input, the edges do not have weights and we penalize large neighborhoods.

Furthermore, other standard approaches for money laundering are rule based methods (Rajput et al. 2014; Khanuja and Adane 2014), which have the disadvantage of possibly being circumvented by criminals, and machine learning based methods, of which many examples can be found in the surveys (Chen et al. 2018; Bakhshinejad et al. 2022). Some of the disadvantages of machine learning methods are that they are often hard to interpret and that they can rely on having large labeled data sets for training.

1.3. Contributions and Thesis Organization

In Chapter 2, the preliminary definitions and notations are introduced and the novel optimization problems that are the focus of this thesis are formally defined: the Predecessor Dense Reachable Subgraph problem (PDRS) and the Neighborhood Dense Reachable Subgraph problem (NDRS).

In Chapter 3, we prove the NP-hardness of PDRS and NDRS. This is done by reduction from the NP-hard Minimum Vertex Cover problem. Two different but analogous proofs are given for the two problems. The PDRS proof uses a 3-layer auxiliary graph and the NDRS proof uses a 4-layer auxiliary graph. [1] The NP-hardness gives a motivation for using integer linear programming for an exact algorithm, for looking for polynomial variants and for designing heuristic algorithms.

In Chapter 4, we first give a simple mixed integer linear-fractional program that gives an exact solution to PDRS and NDRS. We then show how we can derive a mixed integer linear program that also gives an exact solution. Lastly, we derive a mixed integer linear program with substantially fewer variables.

In Chapter 5, we define a pair of new problems by adding an extra restriction to the solution set. Namely, that the selected set can have only one node per layer. For the new variant of the problems, we prove

[1] The appendix contains a section explaining how both these proofs can be done using 2-layer graphs. This was discovered in a late stage of the thesis.

that we can find an exact solution in polynomial time by designing two dynamic programming algorithms.

In Chapter 6, we introduce two heuristic algorithms for both PDRS and NDRS. In the first algorithm, we start with the empty set and use the dynamic programming algorithm for the single paths case to iteratively add the nodes of these paths to our solution. Once our solution contains the complete set of nodes, we take the best intermediate solution that was found thus far. In the second algorithm, we instead start with the full set of nodes as an allowed neighborhood and we look what the maximum weight is we can achieve, whilst restricting the neighborhood of the selected set to the currently allowed neighborhood. We then remove nodes from this neighborhood one by one, by removing the neighbor that decreases the allowed weight the least. To calculate the impact of removing the neighbor on the allowed weight, we developed an algorithm that takes the reachability constraint into consideration. These heuristic algorithms have improved runtime performances compared to the exact algorithm, though the quality of the solutions is generally lower.

We ran experiments using implementations of all the algorithms from the previous chapters. In Chapter 7, we first discuss how we generated the individual graphs and the data sets that were used for testing. Afterwards, we show the results of the different algorithm implementations in terms of running time and solution quality. We plot the effect of different graph parameters on the implementations. These parameters are the number of nodes, the number of edges and the number of layers. Some basic observations based on the plots are given.

In the final chapter, Chapter 8, we give a conclusion of the theoretical and experimental results and there is a reflection and future work section. In the theoretical section, the results from Chapter 3 to Chapter 6 are summarized. In the experimental results section, we discuss which conclusions can be drawn from the experiments. Lastly, we give an overall discussion of the thesis and give a list of possible subjects and amendments for future work.

2

Preliminaries

In this chapter we will give an overview of the mathematical definitions and notations that will be used ubiquitously throughout this thesis. The first section contains these preliminary definitions and the second section contains the formal definitions of both variants of the optimization problem that is the focus of this thesis.

2.1. Graph Definitions and Notations

We will start with fundamental graph theoretic definitions and notations. Towards the end, we give niche definitions that are more specific to this thesis.

A **graph** is a 2-tuple of sets $\mathcal{G} = (V, E)$, where V is a set of **vertices** or **nodes** and E is a set of **edges**.

These edges are of the form $\{x, y\}$ with $x, y \in V$ and $x \neq y$.

For a given edge $\{x, y\}$, the vertices x and y are called the **endpoints** of this edge.

For any $x, y \in V$, the vertices are considered **adjacent** if $\{x, y\} \in E$, this is denoted with the notation $x \sim y$.

The **neighborhood** of a vertex x is defined as $N(x) := \{y \in V \mid x \sim y\}$.

For a subset of vertices $S \subseteq V$, the **neighborhood** is defined as the union of the neighborhoods of the elements: $N(S) := \bigcup_{x \in S} N(x)$.

The **degree** of a vertex x is the number of adjacent vertices: $d(x) := |N(x)|$.

A **path** $P \subseteq V$ is an ordered sequence of vertices $P_1, \ldots, P_{|P|} \in V$ such that $P_i \sim P_{i+1}$ $\forall 1 \leq i \leq |P|$. We use the following notation for the first and last vertex: $P_{start} := P_1$ and $P_{end} := P_{|P|}$. $P \oplus x$ is the notation used for a path that consists of a path P with a vertex x added to the end.

Analogously, we also have the definition of a directed graph with its own versions of the definitions:

A **directed graph** is a 2-tuple of sets $\mathcal{G} = (V, E)$, where V is a set of **vertices** or **nodes** and E is a set of **arcs** or **directed edges**.

These arcs are of the form (x, y) with $x, y \in V$ and $x \neq y$ (Note the difference with edges in an undirected graph).

For a given arc (x, y), the vertices x and y are again called the **endpoints** of the arc.

An arc (x, y) is called an **outgoing edge** of x and an **ingoing edge** of y.

For a vertex x, the **predecessor neighborhood** is defined as $N^-(x) := \{y \in V \mid (y, x) \in E\}$ and the **successor neighborhood** is defined as $N^+(x) := \{y \in V \mid (x, y) \in E\}$. The (full) **neighborhood** is defined as $N(x) := N^-(x) \cup N^+(x)$.

Likewise, for a subset $S \subseteq V$, the **predecessor neighborhood** and **successor neighborhood** are defined as $N^-(S) := \bigcup_{x \in S} N^-(x)$ and $N^+(S) := \bigcup_{x \in S} N^+(x)$, respectively. The (full) **neighorhood** is defined as $N(S) := N^-(S) \cup N^+(S)$.

The **in-degree** of a vertex x is defined as $d_{in} := |N^-(x)|$ and the **out-degree** as $d_{out} := |N^+(x)|$.

A **path** $P \subseteq V$, is an ordered sequence of vertices $P_1, \ldots, P_{|P|} \in V$ such that $(P_i, P_{i+1}) \in E$ $\forall 1 \leq i \leq |P|$. We use the following notation for the first and last vertex: $P_{start} := P_1$ and $P_{end} := P_{|P|}$. $P \oplus x$ is the notation used for a path that consists of a path P with a vertex x added to the end.

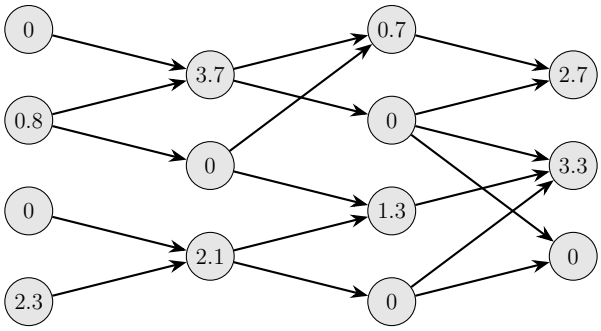


Figure 2.1: An example of a multilayer graph with 4 layers. The vertices have weights.

The following definition introduces the k-layer graph, which is the graph structure on which our optimization problem is defined.

Definition 1. *A **multilayer graph** $\mathcal{G}(V, E)$ on k layers (or k**-layer graph**) is a directed graph whose vertex set is partitioned into k disjoint ordered parts: $V = V_0 \cup \ldots \cup V_{k-1}$ and $V_i \cap V_j = \emptyset$ for all distinct $0 \leq i, j < k$. Furthermore, $\forall (x, y) \in E$: $\exists 0 \leq i < k - 1$ such that $x \in V_i$ and $y \in V_{i+1}$.*

Note that throughout this thesis, two different notations are used for vertices of a k-layer graph. Either simply as $x \in V$ or as the 2-tuple $(i, x) \in V$. In the latter, i indicates the layer that the vertex belongs to and x the element in the layer. If we have a function on the vertices, f, we often write $f(i, x)$ instead of $f((i, x))$.

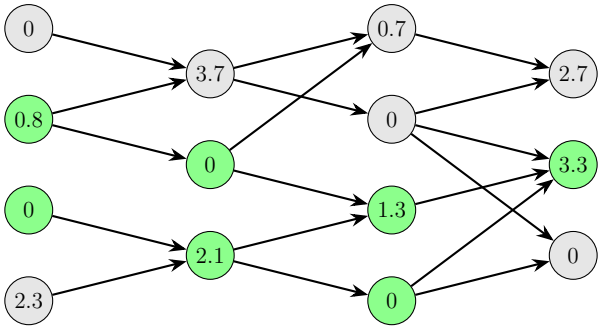


Figure 2.2: A reachable subset of vertices in green: Each vertex in the subset has a path from the first layer to the last through the green nodes.

We now define the notion of *reachability*.

Definition 2. *Let $\mathcal{G}(V, E)$ be a k-layer graph. A vertex $x \in V$ is called **reachable** with respect to a subset $S \subseteq V$ if there exists a path $P \subseteq S$, such that $x \in P$ and $P_{start} \in V_0$ and $P_{end} \in V_{k-1}$. A subset $S \subseteq V$ is called **reachable** if all its elements are reachable with respect to S.*

Alternatively, a subset of vertices is reachable when it is a union of paths from the first layer to the last.

2.2. Problem Statement

Here we give the formal definition of the first variant of the optimization problem studied in this thesis, the PDRS problem.

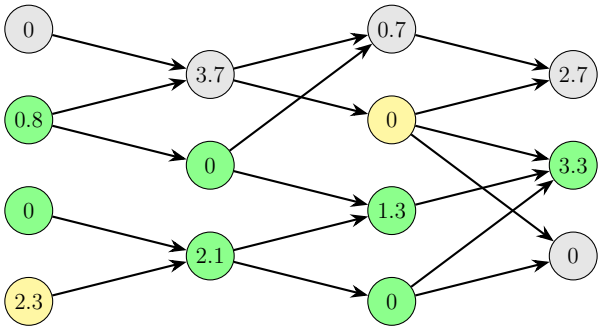


Figure 2.3: A reachable subset of vertices in green with its predecessor neighborhood in yellow. The PDRS value of this solution is the sum of the green weights divided by the number of green or yellow vertices: $(0.8 + 2.1 + 1.3 + 3.3)/9$.

Problem 1. *Let $\mathcal{G}(V, E)$ be a k-layer graph with a non-negative weight function on the vertices $w : V \to \mathbb{R}_{\geq 0}$. The **Predecessor Dense Reachable Subgraph** (**PDRS**) problem consists of finding*

$$\mathrm{PDRS}(\mathcal{G}, w) = \max_{S \subseteq \mathcal{V}(\mathcal{G})} \frac{w(S)}{|N^-(S) \cup S|}$$

where $\mathcal{V}(\mathcal{G})$ are the subsets of $S \subseteq V$ that are reachable.

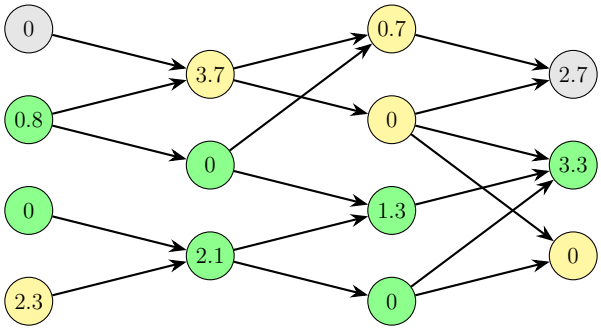


Figure 2.4: A reachable subset of vertices in green with its full neighborhood in yellow. The NDRS value of this solution is the sum of the green weights divided by the number of green or yellow vertices: $(0.8 + 2.1 + 1.3 + 3.3)/12$.

We now define the second variant of the problem, the NDRS problem.

Problem 2. *Let $\mathcal{G}(V, E)$ be a k-layer graph with a non-negative weight function on the vertices $w : V \to \mathbb{R}_{\geq 0}$. The **Neighborhood Dense Reachable Subgraph** (**NDRS**) problem consists of finding*

$$\mathrm{NDRS}(\mathcal{G}, w) = \max_{S \subseteq \mathcal{V}(\mathcal{G})} \frac{w(S)}{|N(S)|}$$

where $\mathcal{V}(\mathcal{G})$ are the subsets of $S \subseteq V$ that are reachable.

3

NP-hardness

In this chapter, we prove the NP-hardness of both the Predecessor Dense Reachable Subgraph (PDRS) problem and the Successor Dense Reachable Subgraph (NDRS) problem. Both proofs are done by reduction from the NP-complete Minimum Vertex Cover problem.

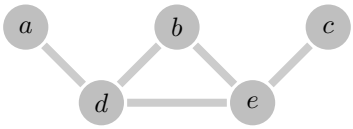
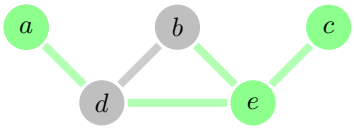


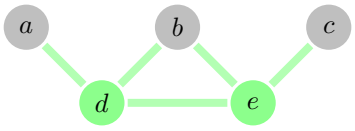
Figure 3.1: An undirected graph.

Vertex Cover Minimum Vertex Cover is a well-known NP-complete problem. As a matter of fact, it was one of Karp's original 21 NP-complete problems (Karp 1972). It involves finding the smallest subset of the vertices of a graph that contains at least one endpoint of each edge. Formally:
Problem 3. *Let $\mathcal{G}(V, E)$ be an undirected graph.*

- *In the **Minimum Vertex Cover** problem, we want to find the minimum $k \in \mathbb{N}$ for which there exists a subset of vertices $S \subseteq V$ such that $\forall e \in E\colon\ e \cap S \neq \emptyset$ and $|S| = k$.*
- *In the decision version of the **Vertex Cover** problem, the variable $k \in \mathbb{N}$ is given as input and we ask the following yes or no question: Does there exist a subset $S \subseteq V$ such that $\forall e \in E\colon\ e \cap S \neq \emptyset$ and $|S| \leq k$?*



(a) The vertices a,c,e do not form a vertex cover, because the edge bd is not covered.



(b) The vertices d and e form a (minimum) vertex cover.

Figure 3.2: Two examples of subsets of vertices: one that is not a vertex cover, and one that is.

In the following theorem, we show the NP-hardness of PDRS using a reduction from Vertex Cover. The proof uses a PDRS problem instance with three layers. [1]
Theorem 1. *The **Predecessor Dense Reachable Subgraph** (PDRS) problem is NP-hard.*

Proof. We will prove this theorem by showing that the decision version is NP-hard. This will be done by giving a polynomial-time reduction from the decision version of Vertex Cover (Problem 3) to the

[1] Unexpectedly, the reduction can also be done using a 2-layer graph. Since this was found in a late stage of the thesis, the original proof is here and we show in Appendix A how the same arguments can be applied to a 2-layer graph.

decision version of our problem.

Given an undirected graph $\mathcal{G}(V, E)$ and a number $k \in \mathbb{N}$, we want to see whether there exists a subset of V that covers all edges and has $\leq k$ elements. To that end, we construct an auxiliary graph $\mathcal{H}(V', E')$, a weight function $w : V' \to \mathbb{R}$ and an objective $\alpha \in \mathbb{R}$, such that there exists a vertex cover of G of size $\leq k$ if and only if there exists a subset $S' \subseteq V'$, such that $\frac{w(S')}{|N^-(S) \cup S|} \geq \alpha$ and S' is reachable.

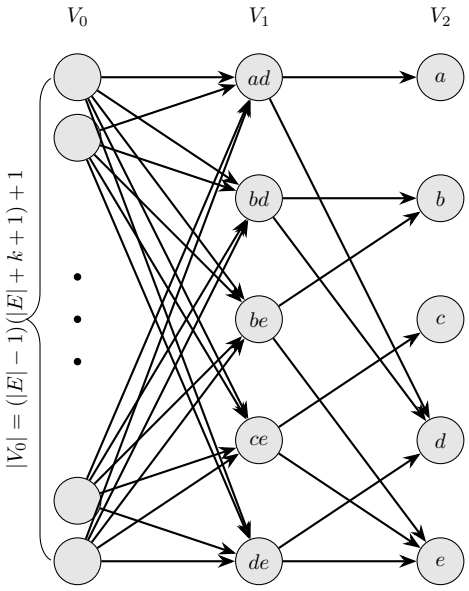


Figure 3.3: The auxiliary graph $\mathcal{H}(V', E')$ based on the vertex cover problem in Figure 3.1. The nodes in column V_1 have weight 1 and the other nodes have weight 0.

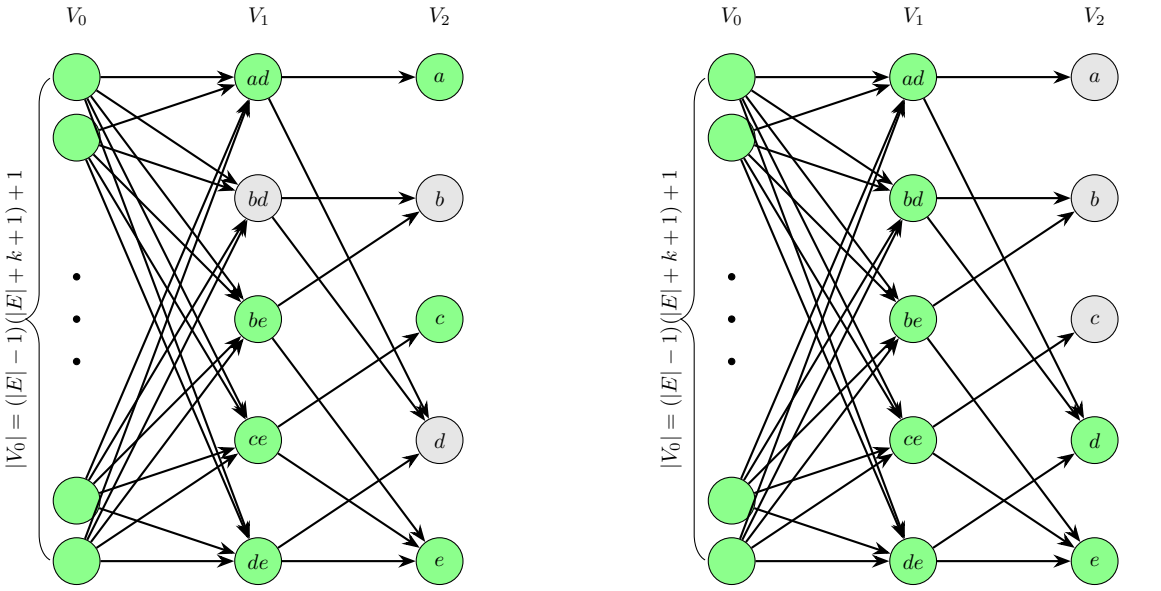
$\mathcal{H}(V', E')$ is a 3-layer graph, i.e. $V' = V_0 \cup V_1 \cup V_2$ with $V_i \cap V_j = \emptyset$ if $i \neq j$, see Figure 3.3. We define these sets as follows: $V_1 = \{x_e \mid e \in E\}$ and $V_2 = \{x_v \mid v \in V\}$ and $|V_0| = (|E| - 1)(|E| + k + 1) + 1$. So V_1 contains an element for each edge in E, V_2 an element for each vertex in V and V_0 is a set whose elements do not correspond to elements of $\mathcal{G}$, but the size of V_0 is relevant.

We now add edges: For each pair $y \in V_0, x_e \in V_1$, we add an edge (y, x_e) and for each pair $x_e \in V_1, x_v \in V_2$ we have an edge between them if and only if v is an endpoint of e in the original graph $\mathcal{G}$. As required, all the edges go from V_0 to V_1 or from V_1 to V_2.

We also define a weight function $w : V' \to \mathbb{R}_{\geq 0}$

$$w(x) = \begin{cases} 1, & \text{if } x \in V_1 \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

and we define an objective $\alpha = 1/(|E| + k + 1)$.



(a) This solution coincides with Figure 3.2a. It will not give an adequate value.

(b) This solution coincides with Figure 3.2b, which is a vertex cover of size 2. It will give an adequate value.

Figure 3.4: Two examples showing how the auxiliary graph relates to the vertex cover problem.

We will now prove that there exists a vertex cover of $\mathcal{G}$ of size $k \iff$ there exists a reachable subset S' of vertices of $\mathcal{H}$ with $\frac{w(S')}{|N^-(S) \cup S|} \geq \alpha$.

"$\Rightarrow$" Assume there exists a vertex cover S of $\mathcal{G}$ with $|S| \leq k$. Then we take $S' = V_0 \cup V_1 \cup \{x_v \mid v \in S\} \subseteq V'$.

For each element $x_e \in S' \cap V_1 = V_1$, we know that it has a path from the first layer within S', because $V_0 \subseteq S'$. Since S is a vertex cover, we know that at least one endpoint of e is in S, say v, and then $x_v \subseteq S'$ and by construction, $x_e \sim x_v$. So there is a path within S' from x_e to the last layer.

The elements of $S' \cap V_0$ and $S' \cap V_2$ are then trivially reachable since S' contains all the elements of V_1. Therefore, the set S' is reachable.

We will now show that this set S' has a satisfactory objective value:

$$
\begin{aligned}
\frac{w(S')}{|N^-(S') \cup S'|} &= \frac{|S' \cap V_1| \cdot 1 + |S' \cap (V_0 \cup V_2)| \cdot 0}{|V_0 \cup V_1 \cup (S' \cap V_2)|} \\
&= \frac{|E|}{|V_0| + |V_1| + |S|} \\
&= \frac{|E|}{(|E| - 1)(|E| + k + 1) + 1 + |E| + |S|} \\
&= \frac{|E|}{|E|\,(|E| + k + 1) - |E| - k - 1 + 1 + |E| + |S|} \\
&= \frac{|E|}{|E|\,(|E| + k + 1) - k + |S|} \\
&\geq \frac{|E|}{|E|\,(|E| + k + 1) - k + k} \qquad \text{since } |S| \leq k \\
&= \frac{|E|}{|E|\,(|E| + k + 1)} \\
&= \frac{1}{|E| + k + 1} \\
&= \alpha
\end{aligned}
$$

With that we have shown that the auxiliary graph $\mathcal{H}$ has a solution to the PDRS problem with objective value $\geq \alpha$.

"$\Leftarrow$" We will prove this by contrapositive: showing that if no size $\leq k$ vertex cover of $\mathcal{G}$ exists, then all reachable subsets of V' have an objective value $< \alpha$. Therefore, assume no such vertex cover exists. Take an arbitrary non-empty reachable subset $T \subseteq V'$. By how we constructed the edges, we know that for every $x_e \in T \cap V_1$, in order to be reachable, it must be that $x_v \in T$ or $x_w \in T$, where v and w are the endpoints of e. This means that if $|T \cap V_1| = |E|$, then $|T \cap V_2| \geq k + 1$, because otherwise $\{v \in V \mid x_v \in T\}$ would be a vertex cover of $\mathcal{G}$ of size $\leq k$.

We now look at the case where $|T \cap V_1| < |E|$, then

$$\begin{aligned}
\frac{w(T)}{|N^-(T) \cup T|} &= \frac{|T \cap V_1| \cdot 1}{|N^-(T) \cup T|} \\
&\leq \frac{|E| - 1}{|N^-(T) \cup T|} \\
&\leq \frac{|E| - 1}{|N^-(T)|} \\
&\leq \frac{|E| - 1}{|N^-(T) \cap V_0|} \\
&= \frac{|E| - 1}{|V_0|} \\
&= \frac{|E| - 1}{(|E| - 1)(|E| + k + 1) + 1} \\
&= \frac{1}{|E| + k + 1 + (|E| - 1)^{-1}} \\
&< \frac{1}{|E| + k + 1} \\
&= \alpha.
\end{aligned}$$

Now we look at the case where $|T \cap V_1| = |E|$ and $|T \cap V_2| \geq k + 1$, then

$$\begin{aligned}
\frac{w(T)}{|N^-(T) \cup T|} &= \frac{|T \cap V_1| \cdot 1}{|N^-(T) \cup T|} \\
&= \frac{|E|}{|N^-(T) \cup T|} \\
&= \frac{|E|}{|V_0| + |V_1| + |T \cap V_2|} \qquad \text{since } |T \cap V_1| = |E| \Rightarrow |T \cap V_1| = V_1 \\
&\leq \frac{|E|}{|V_0| + |V_1| + k + 1} \\
&= \frac{|E|}{(|E| - 1)(|E| + k + 1) + 1 + |E| + k + 1} \\
&= \frac{|E|}{|E|\,(|E| + k + 1) + 1} \\
&= \frac{1}{|E| + k + 1 + |E|^{-1}} \\
&< \frac{1}{|E| + k + 1} \\
&= \alpha.
\end{aligned}$$

So we see that if no size $\leq k$ vertex cover exists for $\mathcal{G}$, then no reachable subset exists for $\mathcal{H}$ with objective value $\geq \alpha$.

Thus we have shown that the two problems are equivalent.

Furthermore, the reduction is polynomial, because we only need to loop through the graph $\mathcal{G}$ once to get all the information for constructing $\mathcal{H}$ and $\mathcal{H}$ has $|V'| = |E|\,(|E| - k - 1) + |E| + |V|$ vertices and $|E'| = |E|\,(|E| - k - 1) \cdot |E| + |E| \cdot 2$ edges, which are both polynomial in the size of $\mathcal{G}$.

With that we have shown the NP-hardness of the decision version of PDRS. In fact, it is actually also NP-complete, because it is clearly in NP, since a yes-certificate would simply be a subset of vertices that satisfies the objective and calculating the objective value of this subset can be done in polynomial time.

If we have an instance of the decision version, we can simply run the optimization version to check if the optimal value is higher than the objective of the decision version. Therefore, the optimization problem must also be NP-hard. □

In this theorem, we show the NP-hardness of NDRS by again using a reduction from Vertex Cover. The arguments are analogous to the proof of Theorem 1, but the auxiliary graph instance is slightly different and it uses four layers. [2]

Theorem 2. *The **Neighborhood Dense Reachable Subgraph** (*NDRS*) problem is NP-hard.*

Proof. We again first prove that the decision version is NP-hard. The proof is highly analogous to the proof of the NP-hardness of PDRS, where we once again use a polynomial-time reduction of Vertex Cover to NDRS.

We start with a vertex cover instance with a graph $\mathcal{G}(V, E)$ and an objective $k \in \mathbb{N}$. If we do the same construction as before, it fails, because all the vertices in column V_2 will always be part of the neighborhood set if we select all the vertices in V_1, since we now take the right neighbors as well for the denominator of the objective function. Therefore, we add an extra layer of vertices, V_3, which is essentially a copy of column V_2. This layer V_3 has a vertex $\hat{x}_v$ for each vertex $v \in V$. The edges between V_2 and V_3 are exactly the pairs that correspond to the same vertex in V, i.e $x_v \in V_2$ and $\hat{x}_w \in V_3$ are adjacent if and only if $v = w$.

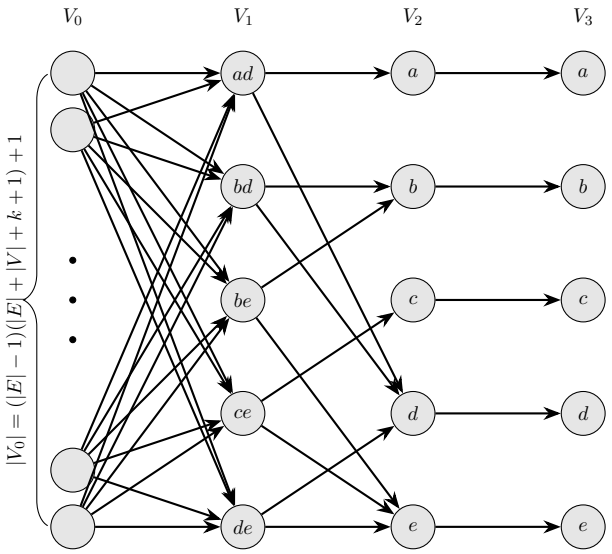


Figure 3.5: The alternative auxiliary graph where we consider the neighbors to the right as well.

Everything else is the same except $|V_0| = (|E| - 1)(|E| + |V| + k + 1) + 1$ and $\alpha = 1/(|E| + |V| + k + 1)$.

[2] We also found a 2-layer proof for this theorem, which was even more surprising. Again, there was no time leftover to change the entire proof, so Appendix A contains a section where we outline the new structure of the auxiliary graph. The rest of the arguments are then equivalent.

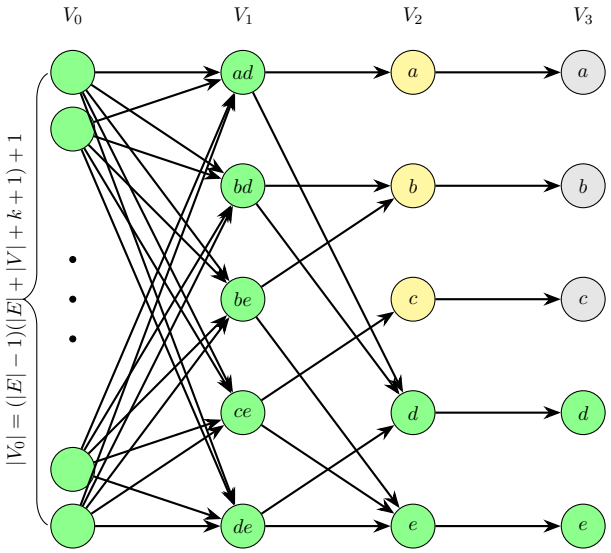


Figure 3.6: A vertex cover of size 2. The nodes a, b and c in column V_2 are part of the neighbor set.

We then see that if $|T \cap V_1| < |E|$,

$$\begin{aligned}
\frac{w(T)}{|N(T) \cup T|} &\leq \frac{|E|-1}{|N(T) \cup T|} \\
&\leq \frac{|E|-1}{|N(T)|} \\
&= \frac{|E|-1}{|V_0|} \\
&= \frac{|E|-1}{(|E|-1)(|E|+|V|+k+1)+1} \\
&= \frac{1}{|E|+|V|+k+1+(|E|-1)^{-1}} \\
&< \frac{1}{|E|+k+1} \\
&= \alpha.
\end{aligned}$$

And if $|T \cap V_1| = |E|$:

$$\begin{aligned}
\frac{w(T)}{|N(T) \cup T|} &\leq \frac{|E|}{|N(T) \cup T|} \\
&\leq \frac{|E|}{|V_0|+|V_1|+|V_2|+|T \cap V_3|} \qquad \text{since } |T \cap V_1| = |E| \Rightarrow V_1 \subseteq T \\
&= \frac{|E|}{(|E|-1)(|E|+|V|+k+1)+1+|E|+|V|+|T \cap V_3|} \\
&= \frac{|E|}{|E|\,(|E|+|V|+k+1)+|T \cap V_3|-k} \\
&= \frac{1}{|E|+|V|+k+1+(|T \cap V_3|-k)/\,|E|}
\end{aligned}$$

So we get in the case where $|T \cap V_1| = |E|$ that the objective value is $\geq \alpha = 1/(|E|+|V|+k+1)$ if and only if $|T \cap V_3| \leq k$.

The rest of the proof is the same as in the proof of Theorem 1

□

4

Mixed Integer Linear Programs

In this chapter, we discuss how the mixed integer linear programming optimization technique can be applied to the Predecessor Dense Reachable Subgraph (PDRS) and the Neighborhood Dense Reachable Subgraph (NDRS) problems.

4.1. Linear Programming Background

Linear programming is an optimization method that can find an optimal solution in polynomial time for any problem, as long as this problem can be modeled as a "linear program" (LP). A linear program requires:

- the objective function to be a linear function.
- the constraints to be modeled as linear inequalities.
- all the variables to be continuous real values.

Before it was formalized by George Dantzig in 1947 (Dantzig 1951), researchers such as Leonid Kantorovich (Kantorovich 1939) and Tjalling Koopmans (Koopmans 1951) already used systems of linear inequalities to optimize logistic processes. George Dantzig introduced the *simplex method*, an efficient method for finding optimal solutions to linear programs. Despite its fast running time in practice, the simplex method does not have a polynomial worst-case complexity. A proof that linear programs can be solved in polynomial time was given in 1979 by using the ellipsoid method (Khachiyan 1980).

Many problems can not be modeled as linear programs with only continuous variables. A "mixed integer linear program"(MILP) is a generalization of linear programs, where some variables can be forced to be integers. This greatly increases the set of problems that can be modeled as a program. However, optimizing such a program is NP-complete (Karp 1972).

Despite this, finding efficient (heuristic) solutions to MILPs has been researched extensively and sophisticated solvers have been developed for MILPs and LPs. Therefore, modeling an (NP-hard) problem as an MILP and inserting it into such a solver is a highly efficient way to find exact solutions or good approximations for your problem.

Linear Programming Basics

There are some variations of how to define linear programs, which are all equivalent. The multiplication of the matrix with a vector can be understood as a system of linear inequalities.

Definition 3. *Let $m, n \in \mathbb{N}$. Given are $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{A} \in \mathbb{R}^{m \times n}$. **Linear Programming** (LP) consists of finding a vector $\mathbf{x} \in \mathbb{R}^n$ that maximizes*

$$\mathbf{c}^T \mathbf{x}$$

and satisfies

$$\begin{aligned}\mathbf{Ax} &\leq \mathbf{b}\\ \mathbf{x} &\geq 0.\end{aligned}$$

Geometrically, this can be interpreted as finding an n-dimensional non-negative vector $\mathbf{x}$, that maximizes its projection onto the vector $\mathbf{c}$, whilst remaining in the polytope defined by $\mathbf{Ax} \leq \mathbf{b}$. Each row of $\mathbf{Ax} \leq \mathbf{b}$, cuts the n-dimensional space in half with a $(n-1)$-dimensional hyperplane. An in practice incredibly fast method of finding such a vector $\mathbf{x}$, is the *simplex method*, which methodically traverses the vertices of the polytope.

Mixed Integer Linear Programming Basics

As stated before, an MILP is a generalization of LPs, where we can require some of the variables to be integers:

Definition 4. *Let $m, n_1, n_2 \in \mathbb{N}$. Given are $\mathbf{c} \in \mathbb{R}^{n_1}$,$\mathbf{d} \in \mathbb{R}^{n_2}$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{A} \in \mathbb{R}^{m \times n_1}$ and $\mathbf{B} \in \mathbb{R}^{m \times n_2}$. **Mixed Integer Linear Programming** (MILP) consists of finding the vectors $\mathbf{x} \in \mathbb{R}^{n_1}$ and $\mathbf{y} \in \mathbb{Z}^{n_2}$ that maximize*

$$\mathbf{c}^T\mathbf{x} + \mathbf{d}^T\mathbf{y}$$

and satisfy

$$\begin{aligned}\mathbf{Ax} + \mathbf{By} &\leq \mathbf{b}\\ \mathbf{x} &\geq 0\\ \mathbf{y} &\geq 0.\end{aligned}$$

In this case, $\mathbf{y}$ contains integers. Geometrically, this means our feasible space is a grid inside of a polytope. This gives the optimization a combinatorial aspect that greatly increases the complexity of the problem. Solving an MILP is NP-hard (Karp 1972).

Gurobi Solver Strategies

The solver that was used in this thesis is the Gurobi Optimizer (Academic License). This solver was chosen, because of its reputation as one of the fastest available solvers and its implementation of state-of-the-art techniques for solving MILPs. We will provide a rudimentary overview of its approach to solving MILPs.

The main algorithm is called Branch-and-Bound: In Branch-and-Bround we traverse a search tree where we branch on the possible values for the integer variables. We can use a global lower bound, given by the best solution found so far. Then we use a “relaxation” as an upper bound for a specific choice of values for some of the integer variables. If such an upper bound is worse than the global lower bound, we know that we will never find a better solution in the subtree given by that choice of variable values.

A relaxation consists of temporarily lifting the integer constraints of the variables, in order to get an upper bound on the optimal solution. If we remove constraints, we can never get an optimal solution that is worse than the actual optimum. Since we no longer have the integer constraints, we can apply standard LP algorithms which are very fast.

Furthermore, Gurobi uses some other techniques:

- Preprocessing: During preprocessing, many factors can be optimized. For example, for some integer variables it can already be seen which value they should have, because any other value would make the problem infeasible. This variable can then be removed and replaced by that value everywhere.
- Adding cuts: Gurobi can add extra constraints that are tighter than the original constraints, but will not decrease the feasible space. This rests on the fact that it knows some variables have to be integers. This will tighten the relaxation bound.

- Heuristics: Using fast suboptimal solutions to the problem will give one a lower bound for the best solution. This allows one to prune branches in the search tree much faster. Instead of only going down the search tree until all the integer variables have a value, we can use other heuristic algorithms for global lower bounds.

4.2. Formulations

In this section, we will show how the PDRS and NDRS problems can be formulated as MILPs. Initially, we will use alternatives to MILPs and show how they can be adjusted to get MILP formulations.

4.2.1. Mixed Integer Linear-Fractional Program

A mixed integer linear-fractional program is an MILP, where the objective function is a ratio of two linear functions. Given an instance of PDRS of NDRS with graph $\mathcal{G}(V, E)$ and weight function w, we formulate the following mixed integer linear-fractional program:

$$\text{maximize} \quad \left(\sum_{v \in V} w_v x_v\right) / \left(\sum_{v \in V} y_v\right) \tag{4.1a}$$

$$\text{subject to} \quad \sum_{v \in V} y_v > 0 \tag{4.1b}$$

$$y_u \geq x_v \qquad \forall (u, v) \in \hat{E} \tag{4.1c}$$

$$y_v \geq x_v \qquad \forall v \in V \tag{4.1d}$$

$$\sum_{\substack{u \in V \\ (u,v) \in E}} x_u \geq x_v \qquad \forall v \in V \setminus V_0 \tag{4.1e}$$

$$\sum_{\substack{v \in V \\ (u,v) \in E}} x_v \geq x_u \qquad \forall u \in V \setminus V_{k-1} \tag{4.1f}$$

$$x_v \in \{0, 1\} \qquad \forall v \in V \tag{4.1g}$$

$$y_v \in \{0, 1\} \qquad \forall v \in V \tag{4.1h}$$

Here $\hat{E}$ is defined differently for the PDRS and NDRS: $\hat{E} = E$ and $\hat{E} = E \cup \{(u, v) \mid (v, u) \in E\}$, respectively.

The idea of the program is that we have, for each node of the graph, a binary variable x_v where $x_v = 1$ indicates that $v \in S$. Furthermore, for each node we have a binary value y_v, where $y_v = 1$ means that v is a (predecessor) neighbor of a node that we selected or is itself a node that we selected.

In (4.1b), we require the sum of ys to be greater than zero to prevent division by 0 in the objective function.

In (4.1c) and (4.1d) we set the constraints for enforcing that the neighbors of the selected set will be added to the denominator.

We also need to enforce the reachability of each v in the selected subset. This is done by letting the sum of the x's of the predecessors of v be greater or equal to x_v. If $x_v = 1$, then at least one of its predecessors must be selected as well. If $x_v = 0$, the inequality is meaningless. The same applies to the successors. These inequalities are on lines (4.1e) and (4.1f).

Lastly, the cost function consists of the sum of the weights of the selected nodes, divided by the number of nodes that neighbor at least one of the selected nodes, where we use the $\hat{E}$ definition of neighboring.

Lemma 3. *The program in Program (4.1) yields the optimal solution to the PDRS and NDRS problems, where $\hat{E}$ is defined as $\hat{E} = E$ for the former and as $\hat{E} = E \cup \{(u, v) \mid (v, u) \in E\}$ for the latter.*

Proof. We will only consider the PDRS variant, because the proof for NDRS is mostly identical. Take an arbitrary reachable set $S \subseteq V$. Let

$$x_v = \begin{cases} 1 \text{ if } v \in S \\ 0 \text{ otherwise} \end{cases}$$

and let

$$y_v = \begin{cases} 1 \text{ if } v \in N^-(S) \cup S \\ 0 \text{ otherwise} \end{cases}$$

By construction, it satisfies the constraints of Program (4.1). Furthermore, $\left(\sum_{v\in V} w_v x_v\right) / \left(\sum_{v\in V} y_v\right)$ is exactly equal to $w(S)/\left|S \cup N^-(S)\right|$. Since we chose S arbitrarily, we know that $\mathrm{PDRS}(\mathcal{G})$ is less than or equal to the optimal solution to the program.

On the other hand, if we take an arbitrary feasible non-zero solution $\mathbf{x}, \mathbf{y}$ to our program, we can define a set $T \subseteq V$ by $v \in T \iff x_v = 1$. By (4.1e) and (4.1f), we know that T is reachable. Let $W = \{v \in V \mid y_v = 1\}$. Then by (4.1c) and (4.1d), we know that $T \cup N^-(T) \subseteq W$. We see that:

$$\left(\sum_{v\in V} w_v x_v\right) / \left(\sum_{v\in V} y_v\right) = w(T)/\left|W\right| \leq w(T)/\left|T \cup N^-(T)\right|.$$

Which implies that the optimal solution to the Program (4.1) is less than or equal to the optimal solution of the PDRS problem. Thus we have shown that they are equal. □

4.2.2. Mixed Integer Quadratic Program

A quadratic program is equivalent to a linear program, except for the fact that it is allowed to contain quadratic functions as well as linear functions for its inequalities and objective function. In this section, we describe how we can rewrite the linear-fractional program to a quadratic program by introducing a scaling factor t.

$$\begin{aligned}
\text{maximize} \quad & t \cdot \left(\sum_{v\in V} w_v x_v\right) && && \text{(4.2a)} \\
\text{subject to} \quad & t \cdot \sum_{v\in V} y_v = 1 && && \text{(4.2b)} \\
& y_u \geq x_v && \forall (u,v) \in \hat{E} && \text{(4.2c)} \\
& y_v \geq x_v && \forall v \in V && \text{(4.2d)} \\
& \sum_{\substack{u\in V \\ (u,v)\in E}} x_u \geq x_v && \forall v \in V \setminus V_0 && \text{(4.2e)} \\
& \sum_{\substack{v\in V \\ (u,v)\in E}} x_v \geq x_u && \forall u \in V \setminus V_{k-1} && \text{(4.2f)} \\
& x_v \in \{0,1\} && \forall v \in V && \text{(4.2g)} \\
& y_v \in \{0,1\} && \forall v \in V && \text{(4.2h)} \\
& t \geq 0 && && \text{(4.2i)}
\end{aligned}$$

This is similar to the MILFP, but by letting $t \cdot \sum y_v = 1$ and thus $t = 1/(\sum y_v)$, we do not have fractions, but only products in our program.

Lemma 4. *The program in Program (4.2) yields the optimal solution to the PDRS and NDRS problems, where $\hat{E}$ is defined as $\hat{E} = E$ for the former and as $\hat{E} = E \cup \{(u,v) \mid (v,u) \in E\}$ for the latter.*

Proof. The proof is done by showing that the Mixed Integer Quadratic Program (4.2) is equivalent to the Mixed Integer Linear-Fractional Program (4.1). First, observe that the constraints (4.1c)-(4.1h) and (4.2c)-(4.2i) are identical and (4.2h) implies that $\sum_{v\in V} y_v > 0$. Therefore, the feasible spaces on the integer variables are the same.

Moreover, $t \cdot \sum_{v\in V} y_v$, so $t = 1/\left(\sum_{v\in V} y_v\right)$. Then we get that $t \cdot \left(\sum_{v\in V} w_v x_v\right) = \left(\sum_{v\in V} w_v x_v\right) / \left(\sum_{v\in V} y_v\right)$ and the objective values are equivalent as well. By Lemma 3, we then know that the Program 4.2 gives an optimal solution to the PDRS and NDRS problems. □

4.2.3. Mixed Integer Linear Program

In this program, we get rid of products of variables altogether.

$$\begin{aligned}
&\text{maximize} && \sum_{v\in V} w_v \hat{x}_v && && (4.3a)\\
&\text{subject to} && y_u \geq x_v && \forall (u,v) \in \hat{E} && (4.3b)\\
& && y_v \geq x_v && \forall v \in V && (4.3c)\\
& && \sum_{\substack{u\in V\\(u,v)\in E}} x_u \geq x_v && \forall v \in V \setminus V_0 && (4.3d)\\
& && \sum_{\substack{v\in V\\(u,v)\in E}} x_v \geq x_u && \forall u \in V \setminus V_{k-1} && (4.3e)\\
& && 0 \leq \hat{x}_v \leq x_v && \forall v \in V && (4.3f)\\
& && 0 \leq s - \hat{x}_v \leq 1 - x_v && \forall v \in V && (4.3g)\\
& && 0 \leq \hat{y}_v \leq y_v && \forall v \in V && (4.3h)\\
& && 0 \leq s - \hat{y}_v \leq 1 - y_v && \forall v \in V && (4.3i)\\
& && \sum_{v\in V} \hat{y}_v = 1 && && (4.3j)\\
& && x_v \in \{0,1\}, \hat{x}_v \geq 0 && \forall v \in V && (4.3k)\\
& && y_v \in \{0,1\}, \hat{y}_v \geq 0 && \forall v \in V && (4.3l)\\
& && s \geq 0 && && (4.3m)
\end{aligned}$$

As can be seen, we removed the products of variables. This is done by having an extra scaled variable for each binary x and y variable, the continuous $\hat{x}$ and $\hat{y}$, where $\hat{x_v} = s \cdot x_v$ and $\hat{y_v} = s \cdot y_v$ for all v. This s is then set to be equal to $s = 1/\sum y_v$ and thus we get that $\sum_{v\in V} w_v \hat{x}_v$ is equal to $\left(\sum_{v\in V} w_v x_v\right) / \left(\sum_{v\in V} y_v\right)$, as desired.

Theorem 5. *The program in Program (4.3) yields an optimal solution to the PDRS and NDRS problems, where $\hat{E}$ is defined as $\hat{E} = E$ for the former and as $\hat{E} = E \cup \{(u,v) \mid (v,u) \in E\}$ for the latter.*

Proof. The proof is done by showing that the Mixed Integer Linear Program (4.3) is equivalent to the Mixed Integer Quadratic Program (4.2). We have shown that the quadratic program finds an optimal solution in Lemma 4.

The constraints for the integer variables are equivalent for the two programs, except that in the quadratic program, it is explicitly stated that the sum of the y-variables should be greater than zero and in the linear program, this is implied by constraints (4.3h) and (4.3j).

Now we look at the constraints (4.3f)-(4.3i). If $x_v = 0$, we have the equations $0 \leq \hat{x}_v \leq 0$ and $0 \leq s - \hat{x}_v \leq 1$, in which case we have that $\hat{x} = 0$ and if $x_v = 1$, we have $0 \leq \hat{x}_v \leq 1$ and $0 \leq s - \hat{x}_v \leq 1 - 1 = 0$, so $\hat{x}_v = s$. The same applies to the y-variables.

In other words, we have that:

$$\hat{x}_v = \begin{cases} 0 \text{ if } x_v = 0 \\ s \text{ if } x_v = 1 \end{cases}$$

and

$$\hat{y}_v = \begin{cases} 0 \text{ if } y_v = 0 \\ s \text{ if } y_v = 1. \end{cases}$$

Since $\sum_{v\in V} \hat{y}_v = 1$

$$1 = \sum_{v\in v} \hat{y}_v = \sum_{v\in V} s \cdot y_v = s \cdot \sum_{v\in V} y_v,$$

we have that $s = 1/(\sum_{v \in V} y_v)$.

Then we have that

$$\sum_{v \in V} w_v \hat{x}_v = \sum_{v \in V} w_v s x_v = s \cdot \sum_{v \in V} w_v x_v = \left(\sum_{v \in V} w_v x_v \right) / \left(\sum_{v \in V} y_v \right) = t \cdot \sum_{v \in V} w_v x_v.$$

So we see that the objective functions of Equation (4.2) and Equation (4.3), rewritten to where they only depend on the integer variables, coincide. We have also seen that the feasible space for the integer variables are equivalent. Therefore, we can conclude that the two programs are equivalent. □

4.2.4. Mixed Integer Linear Program Alternative

We implemented an alternative MILP that requires fewer variables and fewer constraints than the MILP in the previous section, but it is slightly less intuitive.

$$\begin{aligned}
\text{maximize} \quad & \sum_{v \in V} w_v \hat{x}_v && && \text{(4.4a)} \\
\text{subject to} \quad & \hat{y}_u \geq \hat{x}_v && \forall (u,v) \in \hat{E} && \text{(4.4b)} \\
& \hat{y}_v \geq \hat{x}_v && \forall v \in V && \text{(4.4c)} \\
& \sum_{\substack{u \in V \\ (u,v) \in E}} x_u \geq x_v && \forall v \in V \setminus V_0 && \text{(4.4d)} \\
& \sum_{\substack{u \in V \\ (v,u) \in E}} x_u \geq x_v && \forall v \in V \setminus V_{k-1} && \text{(4.4e)} \\
& \hat{x}_v \leq x_v && \forall v \in V && \text{(4.4f)} \\
& 0 \leq s - \hat{x}_v \leq 1 - x_v && \forall v \in V && \text{(4.4g)} \\
& \sum_{v \in V} \hat{y}_v = 1 && && \text{(4.4h)} \\
& x_v \in \{0,1\}, \hat{x}_v \geq 0 && \forall v \in V && \text{(4.4i)} \\
& \hat{y}_v \geq 0 && \forall v \in V && \text{(4.4j)} \\
& s \geq 0 && && \text{(4.4k)}
\end{aligned}$$

We did away with the binary neighbor variables entirely. Instead we only use the scaled $\hat{y}$-variables.

Theorem 6. *Program (4.4) yields the optimal solution to the PDRS and NDRS problems, where $\hat{E}$ is defined as $\hat{E} = E$ for the former and as $\hat{E} = E \cup \{(u,v) \mid (v,u) \in E\}$ for the latter.*

Proof. The proof is done by showing that the Program (4.4) finds the same optimal solution as Program (4.3).

The constraints of the type $y_u \geq x_v$ have been replaced by $\hat{y}_u \geq \hat{x}_v$ in Program (4.4). Scaled variables in Equation (4.3) are equal to s if and only if the corresponding binary variable is 1 Therefore, given a feasible solution to Program (4.3), we can simply ignore the y-variables and get a feasible solution to Program (4.4). Thus, the optimal solution to the Program (4.4) is at least as good as the optimal solution to the Program (4.3).

It remains to show that the alternative program can not have a better optimal solution. Start with an arbitrary feasible partial assignment where we only consider the values of the integer variables $\mathbf{x}$. We know that

$$\hat{x}_v = \begin{cases} 0 \text{ if } x_v = 0 \\ s \text{ if } x_v = 1 \end{cases}$$

for all $v \in V$ for some s.

The objective value is $\sum_{v \in V} w_v \hat{x}_v = s \cdot \sum_{v \in V} w_v x_v$. So given this partial assignment, the objective function can be optimized by maximizing s.

We know that $\hat{y}_u \geq s$ if $\exists v \in V$ such that $(u, v) \in \hat{E}$ and $x_v = 1$ and there is no restriction on the value of $\hat{y}_u$ otherwise, except being non-negative. Since we also know that the $\sum_{v \in V} \hat{y}_v = 1$, the maximum value for s can be achieved by setting all non-zero $\hat{y}$ to the same value and thus $s = 1/\left|\{u \in V \mid \exists v \in S \text{ s.t. } (u, v) \in \hat{E} \text{ and } x_v = 1\}\right|$, which is exactly the optimal value of s that we get in Program (4.3) for this partial assignment of the x's.

Therefore, the two programs give the same optimal value. Then, by Theorem 5, Equation (4.4) also finds the optimal values to the PDRS and NDRS problems. □

5

Polynomial-Time Algorithms for the Single Path Variant

We have seen in Chapter 3 that the Predecessor Dense Reachable Subgraph (PDRS) and Neighborhood Dense Reachable Subgraph (NDRS) problems are NP-hard. An interesting question that arises then is under which circumstances these problems become solvable in polynomial time. This led to a variant of the problems where the solutions of PDRS and NDRS may only consist of a single path going from the first layer to the last. In this chapter, we show that this subproblem can be solved in polynomial time by introducing two algorithms. These algorithms apply the algorithmic technique called dynamic programming.

5.1. Dynamic Programming on Number of Neighbors

Before we explain the dynamic programming algorithms, we first give the formal definitions of the single path variants. The first definition is of the Predecessor Dense Path problem, which is the single path equivalent of the PDRS problem.

Problem 4. *Let $\mathcal{G}(V, E)$ be a k-layer graph with a non-negative weight function $w : V \to \mathbb{R}_{\geq 0}$. The **Predecessor Dense Path** problem (PDP) consists of finding*

$$\mathrm{PDP}(\mathcal{G}, w) = \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{start} \in V_0 \\ P_{end} \in V_{k-1}}} \frac{w(P)}{|N^-(P)| + 1},$$

where $\mathcal{P}(\mathcal{G})$ is the set of all paths in the graph $\mathcal{G}$.

We now introduce the Neighborhood Dense Path problem, the full neighborhood version of PDP.

Problem 5. *Let $\mathcal{G}(V, E)$ be a k-layer graph with a non-negative weight function $w : V \to \mathbb{R}_{\geq 0}$. The **Neighborhood Dense Path** problem (NDP) consists of finding*

$$\mathrm{NDP}(\mathcal{G}, w) = \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{start} \in V_0 \\ P_{end} \in V_{k-1}}} \frac{w(P)}{|N(P)|},$$

where $\mathcal{P}(\mathcal{G})$ is the set of all paths in the graph $\mathcal{G}$.

We will introduce one exact algorithm each for finding an optimal path of the PDP problem and the NDP problem. Both algorithms utilize dynamic programming techniques. Furthermore, the worst-case complexity of the algorithm for PDP is $O(|V|\,\Delta^2 k)$ and for NDP it is $O(|V|\,\Delta^4 k)$, where Δ is the maximum in- or out-degree of the graph.

Intuition The intuition behind the algorithms is that for each column in the multilayer graph and each node in the column we picked, we want to find the optimal paths from the leftmost column to the current

element we are considering. The crux is that we take the optimal path for each possible number of neighbors of a path we can have up to that element. This is necessary to find the optimal solution for the entire graph, since if we only consider the optimal ratios of the weights to the number of neighbors of the paths, we will ignore suboptimal subpaths that will give a better solution in later columns. This is elaborated in the following example:

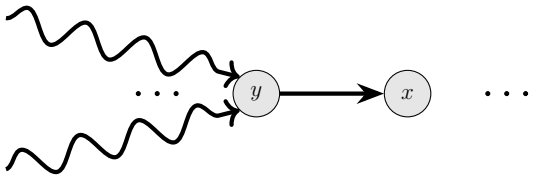


Figure 5.1: A slice of a multilayer graph. There are two paths ending at node y, indicated by the squiggly lines.

Neighbor Requirement Example Take a node x with a predecessor y, see Figure 5.1. Assume the optimal path to y has sum of weights $s_0 = 5$ and total number of predecessor neighbors $n_0 = 5$. However, there is also a path to y that has sum of weights $s_1 = 1$ and $n_1 = 2$. Clearly, the first path has a better weight function: $s_0/n_0 = 5/5 > s_1/n_1 = 1/2$. Say the weight of the node x is 10 and it only has one predecessor, then $(s_0 + 10)/(n_0 + 1) = 15/6 < (s_1 + 10)/(n_1 + 1) = 11/3$. We see that the suboptimal path is better, because the larger number of neighbors diminished the influence of the high extra weight.

Therefore, it is always important to actually consider both the sum of the weights so far and the number of neighbors so far, as opposed to only considering their ratio. On the other hand, if we have two subpaths with the same number of neighbors, then the one with the higher sum of weights will always be better for the total path. There is a slight caveat, since in the problem where we consider the neighbors from both directions, the node that is two columns in front of the current column we are considering, will influence the overlap of neighbors in the second to last column. This will be clarified in the next sections.

Difference Between Versions Like in the original variant, we have two versions of the problem: one where we only consider the predecessor neighbors and one where we consider all neighbors. For the predecessor neighborhood case, the memoization table takes three variables: the column, the node in the column and the total number of neighbors. The value will be the highest sum of weights of a path ending at that node that has exactly that number of neighbors. We then move from the leftmost column to the rightmost column to determine all the values in this table in a bottom-up fashion. Finally, we loop through the values and numbers of neighbors in the last column to find the path with the best objective function.

In the full neighborhood case, the algorithm is quite similar, but we need an extra parameter for the memoization table, since not only the number of neighbors matters, but we also need to know the second to last node in the path. The reason for this is that a node in the $(i-2)$th column and a node in the ith column may have overlapping neighbors in the $(i-1)$th column and we need to make sure that we are not counting neighbors double.

5.2. Exact Polynomial-Time Algorithm for PDP

As discussed in the previous section, our algorithm is based on the observation that, in order to find an optimal path, we must consider both the weight and the number of neighbors of the preceding subpath and not just the objective value, i.e. their ratio. Also, if there are two subpaths starting in the leftmost column and ending in the same node with different weights, but the same number of neighbors, no optimal full path will ever make use of the subpath with a lower weight, since we can always replace that subpath by the subpath with the higher weight, without decreasing the value of the full path.

Recurrence relation

In this section we will formulate a recurrence relation that is applicable to our problem. First, we consider a function that is defined as the highest weight path ending at a specific node with a specific number

of neighbors. One can then get the optimal PDP value from this function by looping through all the nodes in the last layer and all the number of neighbors for which a path exists ending in that node. The maximum ratio between the weight and the number of neighbors then gives a solution to PDP.

Definition 5. *Given an instance of PDP with multilayer graph $\mathcal{G}(V, E)$ and weight function w, we define the function OPT : $V \times \mathbb{N} \to \mathbb{R}$ as follows:*

$$\textit{OPT}((i,x),a) = \max_{\substack{P\in\mathcal{P}(\mathcal{G}) \\ P_{end}=(i,x) \\ |N^-(P)|+1=a \\ P_{start}\in V_0}} w(P), \tag{5.1}$$

where $\mathcal{P}(\mathcal{G})$ denotes the set of all paths in the graph $\mathcal{G}$. If such a path does not exist, the $\max$ is understood to be minus infinity.

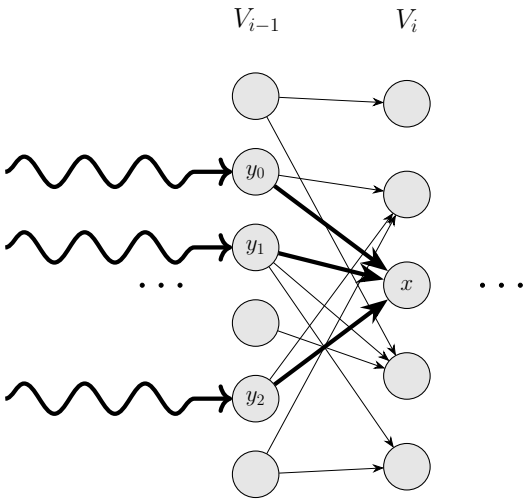


Figure 5.2: A slice of a multilayer graph. The squiggly lines indicate paths. We find the optimal path ending at node x by looking at the optimal paths ending in the predecessors of x. See Lemma 7.

Of course, we don't need to try every single path, because we can use the optimal paths for the previous layers to calculate the optimal paths for the current layer. For a specific node and a specific number of layers, we can try all the neighbors that precede the current node and see what the best path is for that amount of neighbors. The recurrence relation we then get is as follows:

Lemma 7. *Let $\mathcal{G}(V, E)$ be a k-layer graph with non-negative weight function on the vertices $w : V \to \mathbb{R}_{\geq 0}$ We have the following recurrence relation for the function OPT defined in Equation* (5.1)*.*

$$\textit{OPT}((i,x),a) = \begin{cases} w(i,x) + \max_{y\in N^-(i,x)} \textit{OPT}(y, a - |N^-(i,x)|) & \textit{if } i > 0 \\ w(i,x) & \textit{if } i = 0 \textit{ and } a = 1 \\ -\infty & \textit{otherwise} \end{cases} \tag{5.2}$$

Proof. We first look at the case where $i = 0$:
Take arbitrary $x \in V_0$. Since $\mathcal{G}$ is a multilayer graph where all edges go towards the succeeding layer, any path starting with a node in V_0 and ending at $(0, x)$ contains only node $(0, x)$. For this path P, $N^-(P) = \emptyset$, so $|N^-(P)| + 1 = 0 + 1 = 1$. Therefore, OPT$((0, x), a) = -\infty \; \forall a \neq 1$ and OPT$((0, x), 1) = w(0, x)$.

Now we look at the case where $i > 0$:
Take arbitrary $0 < i < k, x \in V_i, a \in \mathbb{N}$.

Consider the case where no path exists that ends in (i, x) with a neighbors. In that case OPT$((i, x), a) = -\infty$. If $w(i, x) + \max_{y\in N^-(i,x)}$ OPT$(y, a - |N^-(i, x)|) \neq -\infty$, then there must be some $y \in N^-(i, x)$ and a path P such that $P_{end} = y$,$P_{start} \in V_0$ and $|N^-(P)|+1 = a-|N^-(i, x)|$. However, then $P' = P \oplus (i, x)$ is also a valid path in $\mathcal{G}$, since $(v, (i, x)) \in E$ and furthermore, $|N^-(P')| + 1 = |N^-(P \cup (i, x))| + 1 =$

$|N^-(P)| + |N^-(i,x)| + 1 = a - |N^-(i,x)| + |N^-(i,x)| = a$, where we use that $N^-(P) \cap N^-(i,x) = \emptyset$, since we are considering a multilayer graph. This leads to a contradiction, since we assumed that such a path did not exist, therefore $w(i,x) + \max_{y \in N^-(i,x)} \mathsf{OPT}(y, a - |N^-(i,x)|) = -\infty = \mathsf{OPT}((i,x), a)$.

Now we consider the case where a path that ends in (i,x) with a neighbors does exist. First, we will show that the left-hand side is less than or equal to the right-hand side of Equation (5.2). Take such a path Q arbitrarily. Let y be the second to last vertex of this path and Q' the path that we get if we remove (i,x). $N^-(Q') + 1 = N^-(Q) + 1 - |N^-(i,x)| = a - |N^-(i,x)|$, so $w(Q') \leq \mathsf{OPT}((i-1,y), a - |N^-(i,x)|)$ and $y \in N^-(i,x)$, so also $w(Q') \leq \max_{y \in N^-(i,x)} \mathsf{OPT}(y, a - |N^-(i,x)|)$. Then $w(Q) = w(i,x) + w(Q') \leq w(i,x) + \max_{y \in N^-(i,x)} \mathsf{OPT}(y, a - |N^-(i,x)|)$. Q was an arbitrary path so $\max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{end} = (i,x) \\ |N^-(P)|+1 = a \\ P_{start} \in V_0}} w(P) \leq w(i,x) + \max_{y \in N^-(i,x)} \mathsf{OPT}(y, a - |N^-(i,x)|)$.

Now we show that the right-hand side is less than or equal to the left-hand side of Equation (5.2). If we take an arbitrary predecessor $y \in N^-(i,x)$ and path R ending at y with $a - |N^-(i,x)|$ neighbors, then $R \oplus (i,x)$ is a path ending at (i,x) with a neighbors so $w(R \oplus (i,x)) = w(i,x) + w(R) \leq \mathsf{OPT}((i,x), a)$ and again it was arbitrary, so $w(i,x) + \max_{y \in N^-(i,x)} \mathsf{OPT}(y, a - |N^-(i,x)|) \leq \mathsf{OPT}((i,x), a)$

We then get that indeed $\mathsf{OPT}((i,x), a) = w(i,x) + \max_{y \in N^-(i,x)} \mathsf{OPT}(y, a - |N^-(i,x)|)$.

With that we have shown that the relation holds for all possible cases. □

Pseudocode

We present the pseudocode of the dynamic programming algorithm that solves the PDP problem. A description and explanation of the algorithm follow the pseudocode.

Algorithm 1 PDP Algorithm ($\mathcal{G}$, w)

Preprocessing Graph: store edges as adjacency lists
▷ Constructing the dynamic programming table
Initialize the memoization tables `dp` and `last_node` as two-dimensional arrays of maps
for each x in first layer **do**
    $\texttt{dp}[0][x](1) \leftarrow w(0, x)$
    $\texttt{last\_node}[0][x](1) \leftarrow -1$
for $i \leftarrow 1$ to $k - 1$ **do**
    for each node x in layer i **do**
        for each y in $N^-(i, x)$ **do**
            for each key `nr_nbs` in $\texttt{dp}[i-1][y]$ **do**
                $\texttt{new\_nbs} \leftarrow \texttt{nr\_nbs} + |N^-(i, x)|$
                $\texttt{new\_sum\_weights} \leftarrow \texttt{dp}[i-1][y](\texttt{nr\_nbs}) + w(i, x)$
                if `new_nbs` not in $\texttt{dp}[i][x]$ **or** $\texttt{dp}[i][x](\texttt{new\_nbs}) < \texttt{new\_sum\_weights}$ **then**
                    $\texttt{dp}[i][x](\texttt{new\_nbs}) \leftarrow \texttt{new\_sum\_weights}$
                    $\texttt{last\_node}[i][x](\texttt{new\_nbs}) \leftarrow y$
▷ Finding the value and last node of the best path
$\texttt{best\_sum\_weights} \leftarrow 0, \texttt{best\_nr\_nbs} \leftarrow 1, \texttt{best\_node} \leftarrow -1$
for each x in last layer **do**
    for each key `nr_nbs` in $\texttt{dp}[k-1][x]$ **do**
        $\texttt{sum\_weights} \leftarrow \texttt{dp}[k-1][x](\texttt{nr\_nbs})$
        if $\texttt{sum\_weights}/\texttt{nr\_nbs} > \texttt{best\_sum\_weights}/\texttt{best\_nr\_nbs}$ **then**
            $\texttt{best\_sum\_weights} \leftarrow \texttt{sum\_weights}, \texttt{best\_nr\_nbs} \leftarrow \texttt{nr\_nbs}, \texttt{best\_node} \leftarrow x$
▷ Reconstructing the best path and building the neighbor set
$S \leftarrow \{(k-1, \texttt{best\_node})\}$
`current_elt` $\leftarrow$ `best_node`, `current_nr_nbs` $\leftarrow$ `best_nr_nbs`
for $i \leftarrow k-1$ to 1 **do**
    $\texttt{previous\_node} \leftarrow \texttt{last\_node}[i][\texttt{current\_elt}](\texttt{current\_nr\_nbs})$
    $\texttt{current\_nr\_nbs} \leftarrow \texttt{current\_nr\_nbs} - |N^-(i, \texttt{current\_elt})|$
    $\texttt{current\_elt} \leftarrow \texttt{previous\_node}$
    Add $(i-1, \texttt{current\_elt})$ to S
return S, $\texttt{best\_sum\_weights}/\texttt{best\_nr\_nbs}$

Algorithm Description

Initialization For this algorithm to run efficiently, the edges need to be saved as adjacency lists of the ingoing edges for all the nodes. The weights are stored in a (two-dimensional) array for constant lookup time.

Creating the Dynamic Programming Table In this part, the `dp` dynamic programming table is created. For each node (i, x), we store the maximum weight for each possible number of neighbors of a path plus 1 reaching this node. The plus 1 is there to remain consistent with the $|N^-(S) \cup S|$ denominator definition in PDRS and PDP. Furthermore, we want a data structure for which we can quickly see that a path with a certain number of neighbors already exists, we want to be able to add new elements quickly and we want to be able to loop through the numbers of possible neighbors efficiently. Our `dp` table is thus a two-dimensional array of maps.

The dynamic programming table has a counterpart, `last_node`, which contains the second to last node of the corresponding path. This table is necessary to reconstruct the entire path. It is also stored as a two-dimensional array of maps.

Filling the Dynamic Programming Table We now start the process of filling the dynamic programming

table. We have multiple for loops:

- First we iterate through the layers, starting at the second column: $i \leftarrow 1$ to $k-1$
- Then we iterate through the elements of the ith layer: $x \in V_i$
- We iterate through the predecessors of x: $y \in N^-(i,x)$
- Lastly, we iterate through the possible numbers of neighbors for all paths ending at $(i-1,y)$: `nr_nbs` $\in \texttt{dp}[i-1][y]$

Having a partial solution stored in `dp[i-1][y]`, means that there exists a path to (i,x). We add the weight of (i,x) and the number of neighbors to get the weight and number of neighbors of the path going through $(i-1,y)$. If this path gives a better sum, we store it in the `dp`.

Finding the Last Node of Optimal Path In the following part, we use the `dp` table to get the last node of the optimal path. This is done by iterating through the nodes of the last column of the graph and iterating through all possible numbers of neighbors of the paths ending in that node. Then we find the maximum value, which is given by the weight of the path divided by the number of neighbors.

Reproducing the Path Lastly, we need to reproduce the path. We do this by moving from right to left, and at each column we keep track of the number of neighbors of the remaining path by subtracting the number of predecessors of the current node. Then we know the node of the path in the preceding column, since we save that in the `last_node` table. During this process we fill the set S with the nodes in the path.

The algorithm builds the dynamic programming in a slightly different way than how the recurrent relation is defined. In the recurrent relation, we give the value of OPT$((i,x),a)$ for a specific number of neighbors a, whereas in the algorithm, we loop through the predecessors of (i,x) and possibly update the value `dp[i][x](a)` whenever a path through a predecessor would lead to a path to (i,x) with $a-1$ predecessor neighbors. In this lemma we formally show, using Lemma 7 that the algorithm does indeed give the optimal solution.

Lemma 8. *Algorithm 1 finds the optimal value to the Predecessor Dense Path problem.*

Proof. We will first show by induction that $\texttt{dp}[k-1][x](a)$ will be equal to OPT$((k-1,x),a)$ if OPT$((k-1,x),a) \neq -\infty$ and that $\texttt{dp}[k-1][x]$ will not contain a key a if OPT$((k-1,x),a) = -\infty$.

Base case: For each $x \in V_0$, we set $\texttt{dp}[0][x](a)$ to $w(0,x)$ for $a=1$, and we give no key for other a. This coincides with equation 5.2, since this equation is only unequal to $-\infty$ for $i=0$, when $a \neq 1$.

Take $i > 0$.

Induction hypothesis: For all $x \in V_{i-1}$ and $a \in \mathbb{N}$, $\texttt{dp}[i-1][x](a) = \text{OPT}((i-1,x),a)$ if OPT$((i-1,x),a) \neq -\infty$ and $\texttt{dp}[i-1][x]$ has no key for a otherwise.

Induction step: For each $x \in V_i$, we loop through $y \in N^-(i,x)$ and then loop through $\texttt{dp}[i-1][y]$ and update $\texttt{dp}[i][x]$ if the path through y with that specific number of neighbors gives an upgrade. Consider arbitrary $x \in V_i$ and $a \in \mathbb{N}$, by our algorithm, we have

$$\texttt{dp}[i][x](a) = w(i,x) + \max_{\substack{y \in N^-(i,x) \\ a - \left|N^-(i,x)\right| \in \texttt{dp}[i-1][y]}} \texttt{dp}[i-1][y]\left(a - \left|N^-(i,x)\right|\right),$$

which is equal to

$$\texttt{dp}[i][x](a) = w(i,x) + \max_{\substack{y \in N^-(i,x) \\ a - \left|N^-(i,x)\right| \in \texttt{dp}[i-1][y]}} \text{OPT}\left((i-1,y), a - \left|N^-(i,x)\right|\right)$$

by the induction hypothesis. Since OPT$((i-1,y), a-|N^-(i,x)|) = -\infty$ if $a - |N^-(i,x)| \notin \texttt{dp}[i-1][y]$,

$$\texttt{dp}[i][x](a) = w(i,x) + \max_{y \in N^-(i,x)} \text{OPT}\left((i-1,y), a - \left|N^-(i,x)\right|\right),$$

and this is exactly $\texttt{dp}[i][x](a) = \text{OPT}((i,x),a)$, by equation 5.2.

Finding the max value: The maximum value is given by

$$\begin{aligned}
\mathsf{PDP}(\mathcal{G}, w) &:= \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{end} \in V_{k-1}}} \frac{w(P)}{(|N^-(P)| + 1)} \\
&= \max_{x \in V_{k-1}} \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{end} = (k-1, x)}} \frac{w(P)}{(|N^-(P)| + 1)} \\
&= \max_{x \in V_{k-1}} \max_{a \in \mathbb{N}} \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{end} = (k-1, x) \\ |N^-(P)| + 1 = a}} \frac{w(P)}{a} \\
&= \max_{x \in V_{k-1}} \max_{a \in \mathbb{N}} \frac{\mathsf{OPT}((i, x), a)}{a} \\
&= \max_{x \in V_{k-1}} \max_{\substack{a \in \mathbb{N} \\ \mathsf{OPT}((i,x),a) \neq -\infty}} \frac{\mathsf{OPT}((i, x), a)}{a} \\
&= \max_{x \in V_{k-1}} \max_{a \in \mathtt{dp}[k-1][x]} \frac{\mathtt{dp}[k-1][x](a)}{a}
\end{aligned}$$

So we see that in order to find the optimal value to our problem, it suffices to loop through nodes of the last layer and to loop through the keys in the maps corresponding to the nodes, which is what we do in our algorithm. □

Complexity Analysis

Let $\mathcal{G}(V, E)$ be a k-layer graph. Δ is defined as the maximum in- or out-degree of the graph.

Complexity of Filling the Dynamic Programming Table We will first look at the complexity of filling the `dp` table. The maximum number of predecessors of a path is $\Delta(k-1)$, since each element of the path has at most Δ predecessors. As a matter of fact, this applies to each subpath as well. So a path going from the first layer to layer i, can have at most $\Delta(i-1)$ predecessors, so each map has $O(\Delta k)$ keys in the worst-case.

Filling the `dp` table consists of:

- Looping through all nodes, $O(|V|)$ iterations
- Looping through the predecessors of the node, $O(\Delta)$ iterations
- Looping through the elements of the map, $O(\Delta k)$ iterations
- Checking the value of the map, $O(1)$ time
- Possibly updating the value of the map: $O(1)$ time

If we combine all this, we get that filling the `dp` table takes $O(|V|\,\Delta^2 k)$ time.

Complexity of Finding the Best Ending Node In this part we only need to find the last node of the optimal path and its predecessor. We do this by looping through the nodes $x \in V_{k-1}$ of the last layer and looping through the key-value pairs of $\mathtt{dp}[k-1][x]$ and comparing the values with the best value so far. This takes in the worst-case $O(|V|) \times O(\Delta k) = O(|V|\,\Delta k)$ time.

Reconstructing the Path After finding the last node of the best path, we reconstruct the path from right to left. This takes $O(k)$ time.

Final Complexity We now combine all these complexities, so we end up with: $O(|V|\,\Delta^2 k + |V|\,\Delta k + k) = O(|V|\,\Delta^2 k)$.

Theorem 9. *Algorithm 1 finds the optimal solution to the Predecessor Dense Path problem in $O(|V|\,\Delta^2 k)$ time.*

5.3. Exact Polynomial-Time Algorithm for NDP

The exact dynamic programming algorithm for the NDP problem is similar to the algorithm from the previous section. However, there is an important distinction. If we only consider the highest weight path ending at a certain node with a specific number of neighbors, we do not have all the information required to calculate the best paths in the next neighbor. This is because the second to last vertex of the path could have overlapping neighbors with the vertex that we are adding to the path. The dynamic programming table thus requires an extra parameter: the second to last element.

Recurrence Relation

In this section, we will formulate a recurrence relation that will be used by the algorithm for the NDP problem. First, we will define a function that outputs the highest weight among paths ending at a specific node, with a specific number of neighbors and a given second-to-last node:

Definition 6. *Given an instance of NDP with multilayer graph $\mathcal{G}(V, E)$ and weight function w, we define the function OPT : $V \times V \times \mathbb{N} \to \mathbb{R}$ as follows:*

$$\mathit{OPT}(y, z, a) = \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{end-1}=y \\ P_{end}=z \\ |N^-(P) \cup N^+(P_{0,end-2})|=a \\ P_{start} \in V_0}} w(P), \tag{5.3}$$

where $\mathcal{P}(\mathcal{G})$ denotes the set of all paths in the graph $\mathcal{G}$. If such a path does not exist, $\max$ is understood to be minus infinity. Note that a is the number of neighbors of P that are in the columns before the last column that P traverses.

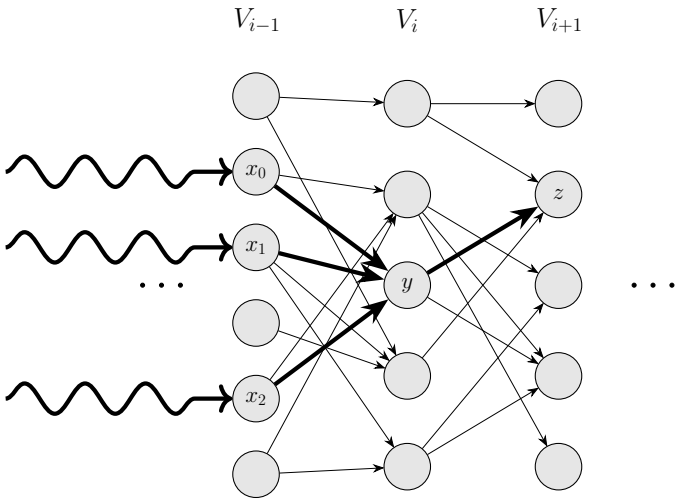


Figure 5.3: A slice of a multilayer graph. The squiggly lines indicate paths. We find the optimal path ending at nodes y and z by looking at the optimal paths ending in a predecessor of y and y itself. We need to consider the last two nodes of a path, because the vertices in V_{i-1} and V_{i+1} can have overlapping neighbors. See Lemma 10.

We will again use the optimal path values of the previous layer to get a recurrence relation:

Lemma 10. *Let $\mathcal{G}(V, E)$ be a k-layer graph with non-negative weight function on the vertices $w : V \to \mathbb{R}_{\geq 0}$ We have the following recurrence relation for the function OPT defined in Equation* (5.3)*.*

$$\mathit{OPT}(y, z, a) = \begin{cases} w(y) + w(z) & \text{if } z \in V_1 \text{ and } y \in N^-(z) \\ & \quad \text{and } a = |N^-(z)| \\ w(z) + \max_{x \in N^-(y)} \mathit{OPT}(x, y, a - |N^+(x) \cup N^-(z)|) & \text{if } z \in V_i \text{ for some } i > 1 \\ & \quad \text{and } y \in N^-(z) \\ -\infty & \text{otherwise} \end{cases} \tag{5.4}$$

Proof. For convenience, let us call the function given by the right hand side of Equation (5.4) $\phi(y, z, a)$. We will show that ϕ and OPT are equivalent.

Take arbitrary $y, z \in V, a \in \mathbb{N}$.

First, we look at the case where no path $P \in \mathcal{P}(\mathcal{G})$ exists such that $P_{end-1} = y$, $P_{end} = z$, $|N^-(P) \cup N^+(P_{0,end-2})| = a$ and $P_{start} \in V_0$. Then $\mathsf{OPT}(y, z, a) = -\infty$. Assume $\phi(y, z, a) \neq -\infty$, then we have two cases:

- $z \in V_1$ and $y \in N^-(z)$ and $a = |N^-(z)|$
- $z \in V_i$ for some $i > 1$ and $y \in N^-(z)$ and $w(z) + \max_{x \in N^-(y)} \mathsf{OPT}(x, y, a - |N^+(x) \cup N^-(z)|) \neq -\infty$.

In the first case, we have a contradiction, because we can simply take the path consisting of just y and z and it will suffice. In the second case, there must be some $x \in N^-(y)$ such that $\mathsf{OPT}(x, y, a - |N^+(x) \cup N^-(z)|) \neq -\infty$. That means that there is some path $P' \in \mathcal{P}(\mathcal{G})$ starting at the first column and ending at y such that $\left|N^-(P') \cup N^+(P'_{0,end-2})\right| = a - |N^+(x) \cup N^-(z)|$. But then $P = P' \oplus z$ is a path such that $P_{end-1} = y$, $P_{end} = z$, $|N^-(P) \cup N^+(P_{0,end-2})| = a$ and $P_{start} \in V_0$, leading again to a contradiction. So $\phi(y, z, a) = -\infty = \mathsf{OPT}(y, z, a)$.

Now we look at the case where a path does exist that ends with y, z and has a neighbors in the layers before the last layer of the path.

If $z \in V_1$, there is only one such path and a must be $a = |N^-(z)|$ and thus $\phi(y, z, a) = w(y) + w(z) = \mathsf{OPT}(y, z, a)$.

If $z \notin V_1$, it takes bit more work.

First, we show that $\mathsf{OPT}(y, z, a) \leq \phi(y, z, a)$. Take an arbitrary path P such that $P_{end-1} = y$, $P_{end} = z$, $|N^-(P) \cup N^+(P_{0,end-2})| = a$ and $P_{start} \in V_0$. Since $z \notin V_1$, P must have more than two elements: let us call $x = P_{end-2}$. Then $Q = P_{0,end-1}$ is a path ending at x and y with $|N^-(Q) \cup N^+(Q_{0,end-2})| = a - |N^+(x) \cup N^-(z)|$.

$$\begin{aligned} w(P) &= w(z) + w(P_{0,end-1}) \\ &\leq w(z) + \mathsf{OPT}(x, y, a - \left|N^+(x) \cup N^-(z)\right|) \\ &\leq w(z) + \max_{x \in N^-(y)} \mathsf{OPT}(x, y, a - |N^+(x) \cup N^-(z)|) \\ &= \phi(y, z, a) \end{aligned}$$

P was arbitrary, so $\mathsf{OPT}(y, z, a) \leq \phi(y, z, a)$.

Now we show that $\mathsf{OPT}(y, z, a) \geq \phi(y, z, a)$. Assume $\phi(y, z, a) \neq -\infty$, because otherwise we are done. Then there is, by definition, a path Q and a vertex x' such that $w(Q) = \max_{x \in N^-(y)} \mathsf{OPT}(x, y, a - |N^+(x) \cup N^-(z)|)$ and $Q_{end} = y$, $Q_{end-1} = x'$, $Q_{start} \in V_0$ and $|N^-(Q) \cup N^+(Q_{0,end-2})| = a - |N^+(x) \cup N^-(z)|$. But then $P := Q \oplus z$ is a path such that $P_{end-1} = y$, $P_{end} = z$, $|N^-(P) \cup N^+(P_{0,end-2})| = a$ and $P_{start} \in V_0$ and furthermore $w(P) = w(z) + w(Q) = w(z) + \max_{x \in N^-(y)} \mathsf{OPT}(x, y, a - |N^+(x) \cup N^-(z)|) = \phi(y, z, a)$. Thus $\phi(y, z, a) \leq \mathsf{OPT}(y, z, a)$, since the OPT function takes the max over all such paths.

With that we have shown that the functions ϕ and OPT are equivalent for each possible case. □

Pseudocode

We present the pseudocode of the dynamic programming algorithm that solves the NDP problem. A description and explanation of the algorithm follow the pseudocode.

Algorithm 2 NDP Algorithm($\mathcal{G}$, w)

Preprocessing Graph: store edges as adjacency lists
▷ Constructing the dynamic programming table
Initialize memoization tables `dp` and `last_nodes` as three-dimensional arrays of maps
for each node z in second layer **do**
    for each y in $N^-(z)$ **do**
        $\texttt{dp}[0][y][z](|N^-(z)|) \leftarrow w(0, y) + w(1, z)$
        $\texttt{last\_node}[0][y][z](|N^-(z)|) \leftarrow -1$
for $i \leftarrow 1$ to $k - 2$ **do**
    for each node x in V_{i-1} **do**
        for each z in $N^+(N^+(x))$ **do**
            $\texttt{shared\_nbs} \leftarrow N^+(x) \cap N^-(z)$
            for each y in `shared_nbs` **do**
                for each key `nr_nbs` in $\texttt{dp}[i-1][x][y]$ **do**
                    $\texttt{new\_total\_nbs} \leftarrow \texttt{nr\_nbs} + |N^+(x)| + |N^-(z)| - |\texttt{shared\_nbs}|$
                    $\texttt{new\_sum\_weights} \leftarrow \texttt{dp}[i-1][x][y](\texttt{nr\_nbs}) + w(i+1, z)$
                    if $\texttt{new\_total\_nbs} \notin \texttt{dp}[i][y][z]$ **or** $\texttt{dp}[i][y][z](\texttt{new\_total\_nbs}) < \texttt{new\_sum\_weights}$ **then**
                        $\texttt{dp}[i][y][z](\texttt{new\_total\_nbs}) \leftarrow \texttt{new\_sum\_weights}$
                        $\texttt{last\_node}[i][y][z](\texttt{new\_total\_nbs}) \leftarrow x$
▷ Finding the value and last nodes of the best path
$\texttt{highest\_path\_value} \leftarrow 0, \texttt{remaining\_nbs} \leftarrow -1$
$\texttt{best\_x}, \texttt{best\_y}, \texttt{best\_z} \leftarrow -1$
for each node y in second to last layer **do**
    for each z in $N^+(y)$ **do**
        for each key `nr_nbs` in $\texttt{dp}[k-2][y][z]$ **do**
            $\texttt{sum\_weights} \leftarrow \texttt{dp}[k-2][y][z](\texttt{nr\_nbs})$
            $\texttt{total\_nr\_nbs} \leftarrow \texttt{nr\_nbs} + |N^+(y)|$
            if $\texttt{sum\_weights}/\texttt{total\_nr\_nbs} > \texttt{highest\_path\_value}$ **then**
                $\texttt{highest\_path\_value} \leftarrow \texttt{sum\_weights}/\texttt{total\_nr\_nbs}, \texttt{remaining\_nbs} \leftarrow \texttt{nr\_nbs}$
                $\texttt{best\_x} \leftarrow \texttt{last\_node}[k-2][y][z](\texttt{nr\_nbs})$
                $\texttt{best\_y} \leftarrow y, \texttt{best\_z} \leftarrow z$
▷ Reconstructing the best path
$\texttt{current\_x} \leftarrow \texttt{best\_x}, \texttt{current\_y} \leftarrow \texttt{best\_y}, \texttt{current\_z} \leftarrow \texttt{best\_z}$
$S \leftarrow \{\texttt{current\_x}, \texttt{current\_y}, \texttt{current\_z}\}$
for $\texttt{layer} \leftarrow k - 3$ to 1 **do**
    $\texttt{nr\_mid\_nbs} \leftarrow |N^+(x)| + |N^-(z)| - |N^+(x) \cap N^-(z)|$
    $\texttt{remaining\_nbs} \leftarrow \texttt{remaining\_nbs} - \texttt{nr\_mid\_nbs}$
    $\texttt{new\_elt} \leftarrow \texttt{last\_node}[\texttt{layer}][\texttt{current\_x}][\texttt{current\_y}](\texttt{remaining\_nbs})$
    Add `new_elt` to S
    $\texttt{current\_z} \leftarrow \texttt{current\_y}, \texttt{current\_y} \leftarrow \texttt{current\_x}, \texttt{current\_x} \leftarrow \texttt{new\_elt}$
return S, `highest_path_value`

Algorithm Description

Most aspects of Algorithm 2 are the same as for Algorithm 1.

Initialization Again, we save the graph as adjacency lists. In our code these are indicated as N^+ and N^- for the successor and predecessor nodes, respectively. The weights are stored as an array.

Creating the Dynamic Programming Table The memoization tables are three-dimensional arrays of maps. $\texttt{dp}[i][x][y](\texttt{nr\_nbs})$ is the highest sum of weights of any path with second to last node $x \in V_i$

and last node $y \in V_{i+1}$ with exactly `nr_nbs` neighbors. Here we only consider the neighbors of the path that are in the ith layer or earlier. `last_node`$[i][x][y](\texttt{nr\_nbs})$ is the last node before x and y of such a max weight path.

Filling the Dynamic Programming Table We now start the process of building the memoization table. First, we define the base case like in Equation (5.4) for all $z \in V_1$ and $y \in N^-(z)$.

Then we have multiple for loops:

- First we iterate through the layers, starting at the second column: $i \leftarrow 1$ to $k-2$
- Then we iterate through the elements of the $i-1$th layer: $x \in V_{i-1}$
- We iterate through the forward neighbors of the forward neighbors of x: $z \in N^+(N^+(x))$.
- We look at the intersection of $N^+(x)$ and $N^-(z)$. Since the adjacency lists are sorted, this can be done in linear time in the size of the adjacency lists, using two pointers.
- We iterate through each y that is in this intersection.
- Lastly, we iterate through the possible numbers of neighbors for all paths ending with (x, y): each key `nr_nbs` in `dp`$[i-1][x][y]$

This means that there exists a path ending with x, y, z. The weight of z is added to get the weight of the path and the number of neighbors in layer i are added to get the number of neighbors of the path, `total_nr_nbs` (Excluding the neighbors in the last column of the path). If this path gives a better sum, we store the weight in `dp`$[i][y][z](\texttt{total\_nr\_nbs})$ and the node x in `last_node`$[i][y][z](\texttt{total\_nr\_nbs})$.

Finding the Last Node of the Optimal Path Next, we use the `dp` table to get the last two nodes of the optimal path. This is done by iterating through the nodes of the second to last column and its successors and iterating through all possible numbers of neighbors of the paths ending in those nodes. Then we find the maximum value, which is given by the weight of the path divided by the number of neighbors. We have to add the size of the successor neighborhood of the second to last vertex though, because the key of the map does not count the neighbors in the last column.

Reproducing the Path Lastly, we need to reproduce the path. We do this by moving from right to left, and at each column we keep track of the number of neighbors of the remaining path by subtracting the number of neighbors of the current node. To subtract the correct number, we need to check the intersection of `current_x` and `current_z`. Then we know the node of the path in the preceding column, since we save that in the memoization table `last_node`. During this process we fill the set S with the nodes of the path.

We will now use the recurrent relation from Lemma 10 to show that the algorithm gives a correct solution to NDP.

Lemma 11. *Algorithm 2 finds the optimal value to the Neighborhood Dense Path problem.*

Proof. The proof is similar to the proof of Lemma 8.

We will again use induction to show that $\forall i \in [0, k-2]$, $y, z, a \in \mathbb{N}$: `dp`$[i][y][z](a)$ will be equal to OPT$((i, y), (i+1, z), a)$ if $(i, y) \in V$ and $(i+1, z) \in V$ and OPT$((i, y), (i+1, z), a) \neq -\infty$ and that `dp`$[i][y][z]$ will not contain a key a otherwise, where OPT is defined as in Equation (5.3).

Base case: $i = 0$:

We see in the code that for each $z \in V_1$ and $y \in N^-((1, z))$, we set `dp`$[0][y][z](a)$ to $w(0, y) + w(1, z)$ for $a = |N^-((1, z))|$. In all other cases where $i = 0$, `dp` contains no key. This coincides exactly with equation 5.4.

Take $i > 0$.

Induction hypothesis: For all $x, y, a \in \mathbb{N}$, either `dp`$[i-1][x][y](a) =$ OPT$((i-1, x), (i, y), a)$ if $(i-1, x) \in V$ and $(i, y) \in V$ and OPT$((i-1, x), (i, y), a) \neq -\infty$ or `dp`$[i-1][y][z]$ has no key a otherwise.

Induction step: In our algorithm, for each $x \in V_{i-1}$, we loop through $z \in N^+(N^+(i-1, x))$ and then loop through $y \in N^+(i-1, x) \cap N^-(i+1, z)$. We then loop through the keys $a \in$ `dp`$[i-1][x][y]$, and

update $\texttt{dp}[i][y][z](a + |N^+(i-1,x) \cup N^-(i+1,z)|)$ if the value of $\texttt{dp}[i-1][x][y](a) + w(i+1,z)]$ gives an improvement. This is equivalent to:

$$\texttt{dp}[i][y][z](a) = (i+1,z) + \max_{\substack{x \in N^-(i,y) \\ a - \left|N^+(i-1,x) \cup N^-(i+1,z)\right| \\ \in \texttt{dp}[i-1][x][y]}} \texttt{dp}[i-1][x][y](a - \left|N^+(i,x) \cup N^-(i+1,z)\right|)$$

for arbitrary $(i,y),(i+1,z) \in V$ if $a \in \texttt{dp}[i][y][z]$. By the induction hypothesis, this is equivalent to:

$$\begin{aligned}
\texttt{dp}[i][y][z](a) &= w(i+1,z) + \max_{\substack{x \in N^-(i,y) \\ a-q \in \texttt{dp}[i-1][x][y]}} \mathsf{OPT}((i-1,x),(i,y),a-q) \\
&= w(i+1,z) + \max_{\substack{x \in N^-(i,y) \\ \mathsf{OPT}((i-1,x),(i,y),a-q) \neq -\infty}} \mathsf{OPT}((i-1,x),(i,y),a-q) \\
&= w(i+1,z) + \max_{x \in N^-(i,y)} \mathsf{OPT}((i-1,x),(i,y),a-q) \\
&= \mathsf{OPT}((i,y),(i+1,z),a),
\end{aligned}$$

where $q := |N^+(i,x) \cup N^-(i+2,z)|$. On the other hand, if $a \notin \texttt{dp}[i][y][z]$, then $a - |N^+(i-1,x) \cup N^-(i+2,z)| \notin \texttt{dp}[i-1][x][y]\ \forall (i-1,x) \in N^-(i,y)$, which means by the induction hypothesis and Equation (5.4) that $\mathsf{OPT}((i,y),(i+1,z),a) = -\infty$.

Finding the max value: The maximum value of the NDP problem is given by:

$$\begin{aligned}
\mathrm{NDP}(\mathcal{G},w) &:= \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{end} \in V_{k-1}}} \frac{w(P)}{|N(P)|} \\
&= \max_{(k-2,y),(k-1,z) \in V} \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{end}=(k-1,z) \\ P_{end-1}=(k-2,y)}} \frac{w(P)}{|N(P)|} \\
&= \max_{(k-2,y),(k-1,z) \in V} \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{end}=(k-1,z) \\ P_{end-1}=(k-2,y)}} \frac{w(P)}{|N^-(P) \cup N^+(P_{0,end-2})| + |N^+(k-2,y)|} \\
&= \max_{(k-2,y),(k-1,z) \in V} \max_{a \in \mathbb{N}} \max_{\substack{P \in \mathcal{P}(\mathcal{G}) \\ P_{end}=(k-1,z) \\ P_{end-1}=(k-2,y) \\ |N^-(P) \cup N^+(P_{0,end-2})|=a}} \frac{w(P)}{a + |N^+(k-2,y)|} \\
&= \max_{\substack{(k-2,y),(k-1,z) \in V \\ (k-2,y) \sim (k-1,z)}} \max_{a \in \mathbb{N}} \frac{\mathsf{OPT}((k-2,y),(k-1,z),a)}{a + |N^+(k-2,y)|} \\
&= \max_{\substack{(k-2,y),(k-1,z) \in V \\ (k-2,y) \sim (k-1,z)}} \max_{a \in \mathbb{N}} \frac{\mathsf{OPT}((k-2,y),(k-1,z),a)}{a + |N^+(k-2,y)|} \\
&= \max_{\substack{(k-2,y),(k-1,z) \in V \\ (k-2,y) \sim (k-1,z)}} \max_{\substack{a \in \mathbb{N} \\ \mathsf{OPT}((k-2,y),(k-1,z),a) \neq -\infty}} \frac{\mathsf{OPT}((k-2,y),(k-1,z),a)}{a + |N^+(k-2,y)|} \\
&= \max_{\substack{(k-2,y),(k-1,z) \in V \\ (k-2,y) \sim (k-1,z)}} \max_{a \in \texttt{dp}[k-2][y][z]} \frac{\texttt{dp}[k-2][y][z](a)}{a + |N^+(k-2,y)|}
\end{aligned}$$

So we see that in order to find the optimal value to our problem, it suffices to loop through the adjacent nodes of the last two layers and to loop through the keys in the maps corresponding to the nodes, which is what we do in our algorithm. □

Complexity analysis

Let $\mathcal{G}(V, E)$ be a k-layer graph. Δ is defined as the maximum in- or out-degree of the graph.

Filling the Dynamic Programming Table We will first look at the complexity of filling the `dp` table. The maximum number of neighbors of a path is $O(\Delta k)$, since in each layer we can have at most $2 \cdot \Delta$ neighbors. Filling the `dp` table consists of:

- Looping through all nodes, $O(|V|)$ iterations
- Looping through the successors of the successors of the nodes, $O(\Delta^2)$ iterations
- Calculating and looping through the intersection, $O(\Delta)$ iterations
- Looping through the keys of the map, $O(\Delta k)$ iterations
- Checking the value of the map, $O(1)$ time

If we combine all this, we get that filling the `dp` table takes $O(n\Delta^4 k)$ time.

Finding the Best Ending Nodes In this part we only need to find the last node of the optimal path and its predecessor. We do this by looping through the nodes $z \in V_{k-1}$ of the last layer and $y \in N^-(z)$ and looping through the keys of $\texttt{dp}[k-2][y][z]$ and comparing the values with the best value so far. This takes $O(|V|) \times O(\Delta) \times O(\Delta k) = O(|V|\,\Delta^2 k)$ time.

Reconstructing the Path After finding the last node of the best path, we reconstruct the path from right to left. This takes $O(k)$ time.

Final Complexity We now combine all these complexities, so we end up with: $O(|V|\,\Delta^4 k) + O(|V|\,\Delta^2 k) + O(k) = O(|V|\,\Delta^4 k)$.

Theorem 12. *Algorithm 2 finds the optimal solution to the Neighborhood Dense Path problem in $O(|V|\,\Delta^4 k)$ time.*

6

Heuristic algorithms

In this chapter, we will introduce a pair of different heuristic algorithms for both the the Predecessor Dense Reachable Subgraph problem and the Neighborhood Dense Reachable Subgraph problem: An algorithm called Greedy Single Paths and an algorithm called Greedy Peeling. Unlike the mixed integer linear programming algorithms, these algorithms do not always guarantee an optimal solution. However, they run for a substantially shorter time than the exact algorithms. They both consist of iterative processes and after termination, the best intermediate solution is selected. There are termination criteria, but the algorithm can be stopped prematurely to get the best intermediate solution so far instead.

The first algorithm, Greedy Peeling, utilizes the exact dynamic programming algorithm for single paths found in Chapter 5. Initially, the solution consists of an empty set. Then every iteration the exact dynamic path algorithm is used to find the single path that increases the objective value the most. The graph is updated to prevent double counting of weight and neighbors. This process continues until no weights can be added to the current solution.

On the other hand, the second algorithm, Greedy Peeling, peels on the allowed neighbor set. For such a neighbor set, we look at the largest reachable subset whose (predecessor) neighborhood is contained in this allowed neighbor set. Every iteration, we remove the neighbor that decreases the sum of the weights of this subset the least. The reachability aspect is taken into consideration for calculating this decrease. This process continues until all the remaining weights are zero. This algorithm was inspired by the Greedy Peeling algorithm on the HNSN problem from (L. Li et al. 2024). However, the multilayer and reachability aspects make the algorithm more complex.

6.1. Greedy Paths Algorithm

Intuition The algorithm described in this section, uses the exact dynamic programming algorithms for single paths described in Chapter 5. Initially, we start with an empty solution. Every iteration the exact dynamic path algorithm is called to construct a memoization table. This memoization table is then used to find the path that will increase the objective function of the current solution the most. The nodes of this solution are added to the current solution. We do not want the neighbors of the path to contribute to the objective function multiple times if we add a path with the same neighbors at a later iteration, so the graph is updated to prevent this. The weights of the selected nodes are set to zero, because the weights of the nodes should only be added once, but we do want to allow new paths to run through them. The optimal single path is found on this new graph and the process repeats until no more non-zero weight nodes can be added.

Pseudocode

We present the pseudocode of the Greedy Single Paths algorithm. A description and explanation of the algorithm follow the pseudocode.

Algorithm 3 Greedy Single Paths Algorithm($\mathcal{G}$, w)

Preprocessing Graph:
Initialize `adj_paths` $\leftarrow E$, `weights` $\leftarrow w$
Initialize `current_sum_weights` $\leftarrow 0$, `current_picked_nodes` $\leftarrow \emptyset$, `current_nbs` $\leftarrow \emptyset$
Initialize `best_picked_nodes` $\leftarrow \emptyset$, `best_value` $\leftarrow 0$
`adj_node_to_nbs` is a function that maps a node to all the neighbors that would be added to the denominator if the node is selected.
`adj_nb_to_nodes` is a function that maps a neighbor to all the nodes that would add it to the denominator when selected.
while `true` **do**
    `dp` $\leftarrow$ `single_path_dp(graph_vars)`
    (`path_nodes`, `path_nbs`) $\leftarrow$ `best_path(dp, current_sum_weights, |current_nbs|)`
    if |`path_nodes`| $= 0$ **then**
        break
    for each `node` in `path_nodes` **do**
        `current_sum_weights` $\leftarrow$ `current_sum_weights` $+$ `weights`[`node`]
        add `node` to `current_picked_nodes`
        `weights`[`node`] $\leftarrow 0$
    for each `nb` in `path_nbs` **do**
        add `nb` to `current_nbs`
        for each x in `adj_nb_to_nodes`(`nb`) **do**
            remove `nb` from `adj_node_to_nbs`(x)
    if |`current_nbs`| > 0 AND `current_sum_weights`/ |`current_nbs`| $>$ `best_value` **then**
        `best_value` $\leftarrow$ `current_sum_weights`/ |`current_nbs`|
        `best_picked_nodes` $\leftarrow$ `current_picked_nodes`
return `best_picked_nodes`, `best_value`

Algorithm Description

Initialization First, we have to store the graph in a suitable way. In our algorithm, the possible paths should remain, but we do not want to count the neighbors multiple times for the denominator of the objective function. Therefore, we have multiple types of adjacency lists: we have adjacency lists to determine the possible paths and we have adjacency lists that determine which nodes will be added to the neighborhood set. The reason we have these two types, is because the latter will be altered during runtime, whilst the former needs to remain the same throughout. In practice, both of these lists are split into ingoing edges and outgoing edges. The outgoing neighbor edges are only relevant in the full neighborhood version of the problem. The other relevant data that are saved are the number of layers, the elements per layer and the weights of the nodes. Furthermore, we have variables that keep track of the state of our current solution. They track the nodes and neighbors that were selected so far as well as the best intermediate solution.

Iterations The algorithm consists of a while loop that adds a subset of the remaining nodes and neighbors to the set that we currently have. The nodes that are added to our solution are the best path that can be found on the current graph. To this end, we apply the exact dynamic programming algorithm, Algorithm 1 or Algorithm 2, where we use the predecessor version or the full neighborhood version depending on the problem. The best path is decided by taking the new value after adding the weight of the paths and the number of neighbors to our current solution. The termination criterion of the algorithm is if no paths with non-zero weight can be found.

Next, we add these nodes and neighbors to our current solution. This current solution is compared to the best solution, to see whether we should update the best solution.

We also modify the graph as to not count weights or neighbors twice. We set the weight of the nodes we

selected to zero and we remove the 'neighbor' connections of the nodes to the neighbors we selected. It should be noted that we do not remove the node or the edges, since we might require some of the nodes we already selected for future paths.

In the end, we return the best solution found over all iterations.

Complexity Analysis

Let $\mathcal{G}(V, E)$ be a k-layer graph. Δ is defined as the maximum in- or out-degree of the graph.

Each iteration we add at least one node to the set of selected nodes. This means that the number of iterations of the while loop is bounded by the number of elements in the graph. In practice, we will generally add more nodes to the graph per iteration, with a maximum of the number of layers.

Each iteration we run the dynamic programming algorithms from Chapter 5. These have complexities of $O(|V|\,\Delta^2 k)$ for the predecessor neighborhood case and $O(|V|\,\Delta^4 k)$ for the full neighborhood case.

Each iteration, we also need to loop through the nodes in the path and the neighbors of the path. The number of nodes is bounded by k and the number of neighbors is bounded by $2k\Delta$. For the nodes we have to do a constant time operation, but for the neighbors we also have to loop through all their adjacent nodes, which is bounded by 2Δ. This gives us complexities of $O(k)$ and $O(k\Delta^2)$ per iteration.

All in all, we then get a total complexity of $O(|V| \cdot (|V|\,\Delta^2 k + k + k\Delta^2)) = O(|V|^2\,\Delta^2 k)$ for PDRS and $O(|V| \cdot (|V|\,\Delta^4 k + k + k\Delta^2)) = O(|V|^2\,\Delta^4 k)$ for NDRS.

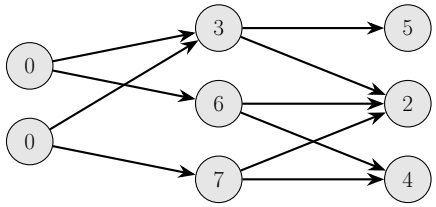
6.2. Greedy Peeling Algorithm

Intuition As opposed to the previous algorithm, this algorithm greedily peels the current subgraph. That is, we start with the entire graph as a solution and remove nodes. The algorithm is inspired by the Greedy Peeling algorithm from (L. Li et al. 2024). In our version, there is a set that functions as the allowed neighborhood. We look at the largest reachable subset whose (predecessor) neighborhood is contained in the allowed neighborhood. This largest reachable subset is the union of all reachable subsets whose (predecessor) neighborhood satisfies this constraint. For each node, we need to track whether it is a part of the allowed neighborhood, a part of the reachable subset or none of these. This is done by initially having, for each node, an extra shadow neighbor version. These shadow neighbors make up the allowed neighborhood. We then iteratively remove the shadow neighbor that decreases the total weight of the allowed reachable subset the least. When we look at the maximum total weight we consider that the allowed nodes need to be reachable, so when picking a shadow neighbor to remove, we also consider the implications for the reachability: if we remove a subset of nodes, we must also remove the nodes that no longer have a path going through them from the first layer to the last. This is taken into account when calculating the influence on the total weight of deleting a shadow neighbor.

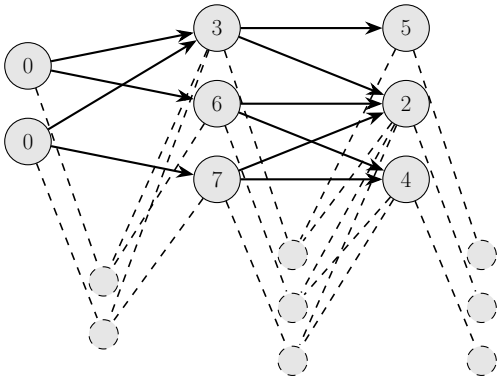
Since we have for each vertex of the original graph the normal and a shadow neighbor version, we are essentially back to the situation of the bipartite case from (L. Li et al. 2024), where the node versions are the set V and the shadow neighbor versions are the set U. Except here V has an extra topological constraint, namely the reachability. Furthermore, this peeling algorithm does not exploit the multilayered quality of the graph, so it could be applied to any directed acyclic graph and, as a matter of fact, it would work for any arbitrary definition of the 'neighborhood' of the selected subgraph.

Example

In this section, we will give a small example of the algorithm.



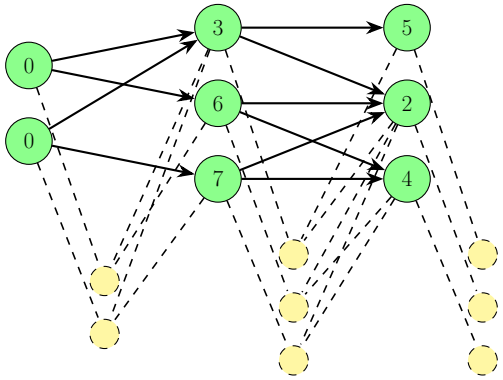
(a) The initial graph.



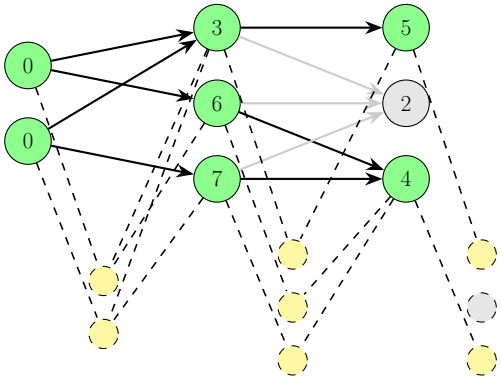
(b) For each node, we create an extra shadow neighbor node. The dashed edges show neighbor adjacencies. When we select a node from the top section, the dashed edges indicate which shadow neighbors will be activated.

Figure 6.1

We start with the initial instance given in Figure 6.1a and consider the PDRS problem. As can be seen in Figure 6.1b, we split each node and create an extra shadow neighbor node at the bottom. When we pick a subset of vertices of the original graph, the dashed edges then indicate which shadow neighbors will be activated. The objective value of a subset is then the ratio of the sum of the weights of the selected vertices to the number of shadow neighbors who are adjacent by a dashed edge to the subset.



(a) The yellow nodes indicate the allowed neighborhood. The green nodes are the current solution set.

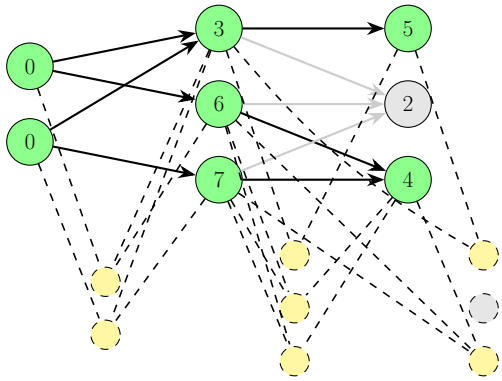


(b) A shadow neighbor and its adjacent node have been removed. The neighbor connections need to be updated.

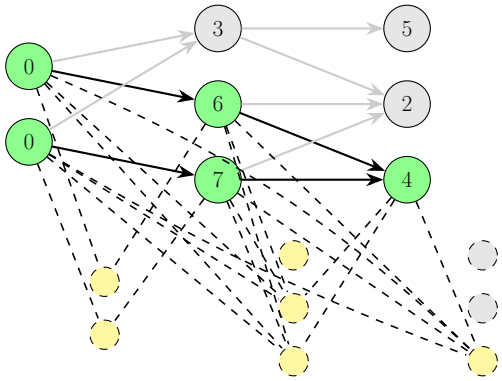
Figure 6.2

In Figure 6.2a, the neighbor set (indicated by yellow) consists of all the shadow neighbors. Therefore, we can pick all nodes in V to be part of the selected set, which gives the maximum weight solution. When deciding which shadow neighbor to remove from the allowed neighbor set, we consider the sum of the weights of all nodes that should be removed from the solution when that shadow neighbor is removed from the allowed neighbor set. In this case, the shadow neighbor connected to the node with weight 2 is removed and consequently the node with weight 2 itself.

As the nodes are removed, so are the possible paths. This means that removing a shadow neighbor might mean more nodes will have to be removed from the solution, since removing the original adjacent nodes might block off some paths which would make some of the nodes no longer reachable. Thus, the graph will have to be updated.



(a) The updated neighbor connections.



(b) The updated graph and neighbor connections after removing the nodes with weight 3 and 5.

Figure 6.3

In Figure 6.3a, we can see that if we remove the node with weight 4, we will have to remove the node with weight 6 and 7 as well. Therefore, the shadow neighbor that was previously only connected to the node with weight 4 has updated its edges. The next shadow neighbor that is removed, is the one that is connected to the 3 and 5 nodes, since the sum of the adjacent nodes is only 8.

This process continues until the allowed set of nodes is empty. Every iteration:

- We remove the shadow neighbor with the lowest sum of weights of its adjacent nodes.
- Delete the adjacent nodes.
- Update the neighbor-to-node connections based on the reachability constraint.

Pseudocode

We present the pseudocode of the Greedy Peeling algorithm. A description and explanation of the algorithm follow the pseudocode. All algorithms have access to the global variables defined in Algorithm 4.

Algorithm 4 Greedy Peeling Algorithm($\mathcal{G}$, w)

Algorithm variables:
`forward_arcs` is a function that maps the nodes to the endpoints of the outgoing edges. This coincides with N^+ in the input graph.
`backward_arcs` is a function that maps the begin points of the ingoing edges for each node. This coincides with N^- in the input graph.
`node_to_nbs` is a function that maps a node to all the shadow neighbors that would activate if the node is selected.
`nb_to_nodes` is a function that maps a shadow neighbor to all the nodes that would activate it when selected.
`nb_values` is a function that maps each shadow neighbor y to the sum of the weights of the nodes in `nb_to_nodes`(y).
`current_nbs` $\leftarrow V$
`current_nodes` $\leftarrow V$
`total_sum` $\leftarrow w(V)$
procedure Main($\mathcal{G}$, w)
    initialize_variables( )
    remove_unreachables( )
    update_nb_sets( )
    while $|$`current_nbs`$| > 1$ **do**
        `next_nb` $\leftarrow \underset{y \in \texttt{current\_nbs}}{\arg\min}$ `nb_values`(y)
        remove `next_nb` from `current_nbs`
        for each x in `nb_to_nodes` **do**
            remove_node(x)
        update_nb_sets( )
        if `total_sum`$/|$`current_nbs`$| >$ `best_value` **then**
            `best_state` $\leftarrow$ `current_state`
    return `best_state`

Algorithm 5 Auxiliary Procedures Greedy Peeling Algorithm 1

▷ Removes a node and updates the total weight and adjacency lists.
procedure remove_node(x)
    remove x from `current_nodes`
    `total_sum` $\leftarrow$ `total_sum` $- w(x)$
    for y in `node_to_nbs` **do**
        remove x from `nb_to_nodes`(y)
        `nb_values` $\leftarrow$ `nb_values` $- w(x)$
    for u in `forward_arcs` **do**
        remove x from `backward_arcs`(u)
    for u in `backward_arcs` **do**
        remove x from `forward_arcs`(u)

Algorithm 6 Auxiliary Procedures Greedy Peeling Algorithm 2

▷ Removes the nodes that can not be part of any reachable subset.
procedure remove_unreachables( )
    `forward_reached` $\leftarrow \emptyset$
    for each x in first layer **do**
        forward_dfs(x,`forward_reached`)
    `backward_reached` $\leftarrow \emptyset$
    for each x in last layer **do**
        backward_dfs(x,`backward_reached`)
    for each x in graph **do**
        if x not in `forward_reached` **or** x not in `backward_reached` **then**
            remove_node(x)
▷ This function calculates for each neighbor which nodes would have to be removed from the selected set if that neighbor were to be peeled.
procedure update_nb_set( )
    for `nb` in `current_nbs` **do**
        `new_nodes` $\leftarrow$ implication_dfs(`nb`)
        add `new_nodes` to `nb_to_nodes`(nb)
        for each x in `new_nodes` **do**
            add `nb` to `node_to_nbs`(x)
            `nb_values`(`nb`) $\leftarrow$ `nb_values`(`nb`) $+ w(x)$

Algorithm 7 Implication Depth First Search

`forward_vis_count` a function that tracks the number of times a node has been reached by an ingoing edge
`backward_vis_count` a funtion that tracks the number of times a node has been reached by an outgoing edge
`discovered_nodes` $\leftarrow \emptyset$
`starting_nodes`
procedure implication_dfs_it(x)
    for each y in `forward_arcs`(x) **do**
        if y not in `starting_nodes` and y not in `discovered_nodes` **then**
            `forward_vis_count`(y) $\leftarrow$ `forward_vis_count`(y) $+ 1$
            if `forward_vis_count`(y) $==$ |`backward_arcs`(y)| **then**
                add y to `discovered_nodes`
                implication_dfs_it(y)
    for each y in `backward_arcs`(x) **do**
        if y not in `starting_nodes` and y not in `discovered_nodes` **then**
            `backward_vis_count`(y) $\leftarrow$ `backward_vis_count`(y) $+ 1$
            if `backward_vis_count`(y) $==$ |`forward_arcs`(y)| **then**
                add y to `discovered_nodes`
                implication_dfs_it(y)
procedure implication_dfs(`nb`)
    initialize global variables
    `starting_nodes` $\leftarrow$ `nb_to_nodes`[`nb`]
    for each x in `nb_to_nodes` **do**
        implication_dfs_it(x)
    return `discovered_nodes`

Algorithm Description

Initialization During initialization, we prepare all the data structures. This is important, because the graph needs to be stored quite differently from how the multilayer graph is typically stored. As previously stated, we effectively split each vertex of the original graph into a node version and a shadow neighbor version. The `forward_arcs` and `backward_arcs` are there for the reachability constraint of the chosen subset. The `nb_to_nodes` and `node_to_nbs` are there to keep track of which nodes are no longer allowed in the solution after we remove a shadow neighbor.

Then `remove_unreachables()` is called to remove nodes that can never be a part of a reachable solution. We do a depth first search from the first layer and again from the last layer. All the nodes that are not in the intersection of the nodes reached by the two searches, are consequently removed from the `current_nodes`. This works, because if a node is in the intersection, then there is a path from the first layer to the last through the node and all other nodes on this path will be part of the intersection as well. We need to be careful, because a removed node can still be a neighbor of the selected nodes though. Afterwards, the shadow neighbor adjacency lists are updated by calling `update_nb_sets()`.

Main While Loop In the main while loop, we first take the shadow neighbor whose sum of weights of adjacent nodes is the lowest. This is done in $O(\log n)$ time, by storing the shadow neighbor and the sum of the weights as a pair in a priority queue, where they are sorted by sum of the weights. Next, the shadow neighbor and the adjacent nodes are removed from the entire graph with the `remove_node(x)` procedure. Afterwards, the shadow neighbor's adjacent nodes are updated again with `update_nb_set()`. Lastly, it is checked whether the solution of the current state is better than the previous best solution, and if so, the `best_state` is updated.

Updating the Neighbor Sets In `update_nb_sets()`, we loop through all the shadow neighbors, and for each one, we update the nodes that are adjacent to it. This is done by checking which nodes are not reachable anymore when the adjacent nodes of the shadow neighbor are removed. To that end, the procedure `implication_dfs(nb)` is called. Afterwards, all the relevant data structures are updated with the new nodes.

Implication Depth First Search The goal of the implication depth first search, is to find all the nodes that can never be part of a reachable set after a subset of nodes is removed. This subset of nodes is the starting set of the procedure. The `implication_dfs` procedure is similar to normal depth first search. There are two major differences though. Firstly, even though it is a directed graph, the search can move forwards and backwards along the arcs. Secondly, a new node is not explored unless either all the ingoing edges have been visited or all the outgoing edges have been visited. This coincides with all the predecessors or all the successors of a node having been visited in this search. Since visiting a node in the search means that it will become unreachable, the new node would become unreachable as well. Thus, the deletion of all the original adjacent nodes of the shadow neighbor and the already discovered nodes, would result in the new node becoming unreachable. The `discovered_nodes` is then exactly the set of nodes that will become unreachable when all the nodes in `starting_nodes` are removed.

Complexity

Let $\mathcal{G}(V, E)$ be a k-layer graph. Δ is defined as the maximum in- or out-degree of the graph.

The number of iterations of the while loop is at most $|V|$, since at least one shadow neighbor is removed each iteration. In each iteration, we:

- Take and remove the shadow neighbor with lowest value: $O(\log |V|)$ time.
- Remove the adjacent nodes: In theory, there could be $O(|V|)$ adjacent nodes after some updates and each one could be adjacent to $O(|V|)$ shadow neighbor nodes, so we could have in the worst case $O(|V|^2)$ time.
- Each `implication_dfs` can take $O(|V| + |E|)$ time and we have to run it for each neighbor, so we get: $O(|V|\,(|V| + |E|))$ time.

This means that we get a worst-case complexity of $O(|V|^2\,(|V| + |E|))$. However in practice, the efficiency is often much better as can be seen in Chapter 7.

7

Experimental Results

In this chapter, we show the experimental results of the implementations of the algorithms described in the previous chapters. It will be shown how the running times and solution qualities depend on the different parameters of the input graphs, such as the number of nodes, the number of layers and the number of edges. Since we are dealing with a novel problem, no readily available data sets were found. We therefore chose to generate the data sets ourselves.

The reason for implementing the algorithms and running them on data sets, is not just to observe how they scale with input size, but also to observe how the solution quality behaves. Furthermore, it allows one to get a grasp of the different advantages and disadvantages of the individual algorithms. On top of that it gives the option to compare to the theoretical worst-case complexities.

We will begin this chapter with an explanation of how the individual graphs are generated and then how the data sets containing the graphs instances are generated. Afterwards, a description of the setup of the experiments is given. The last two sections show the plots displaying the running times and the solution quality of the algorithms, respectively. These plots will be accompanied by observations. The interpretations and conclusions drawn from the experiments can be found in Section 8.2.

7.1. Data sets

The first part of this section explains the methods for generating graphs given the graph parameters. The second part shows which methods the data sets used for generating the graph parameters.

7.1.1. Generating Graphs

In this section, we give an informal overview of the different strategies for generating graphs to test the algorithms on. When generating a graph, we take in several different parameters as input. Which parameters are required, depends on the specific graph generation method.

Nodes and Layers The first two parameters for any graph, are the *number of nodes*, $|V|$, and the *number of layers*, k, of the multilayer graph. For determining the number of elements per layer, the elements are either split evenly over the layers or a partition is taken uniformly from all different ways that one can spread the nodes over the layers. Since this could lead to very unbalanced layers, we have an optional extra parameter, the *minimum layer ratio*, $r_{minLayer}$. This ensures that every layer gets at least this portion of the expected elements per layer. The remaining elements are then again spread over the layers.

Edges For generating the edges of the graphs, several different methods were used, each with their own specific input parameters. The most straightforward method takes as the input parameter *number of edges*, $|E|$. It randomly picks $|E|$ distinct edges from all possible edges.

An alternative method uses as input parameter *edge ratio*, r_{edge}. This can be implemented in two ways: We either randomly take a r_{edge} portion of all possible edges, or we have a Bernoulli distribution with probability r_{edge} for each possible edge.

Lastly, we can also generate edges by taking as input parameter *expected degree*, d_{exp}. What is meant exactly by *expected degree* (since it could mean both the in- or out-degree), will be cleared up by the following: The edges are generated with a Bernoulli distribution, but the probability depends on which layers the edge candidate is in. If a candidate edge is between two layers with a and b nodes, respectively, the number of pairs of elements is equal to ab. Then the probability of a candidate edge becoming an edge in the graph is given by $(a + b) \cdot d_{exp}/(2ab)$, in which case the expected number of edges between these two layers will become $(a+b) \cdot d_{exp}/2$. This number of edges will give the desired average degree d_{exp}.

Weights The initial weight function considered was the uniform distribution on an interval starting at zero. However, this led to a large portion of the graphs having an optimal solution consisting of picking all the nodes. Therefore, an extra Bernoulli distribution was introduced: if the Bernoulli distribution returned a 1, the node is assigned a random value from the uniform distribution as weight. Otherwise, the node is assigned value 0.

Double Graph Since the motivation for this network problem came from finding suspicious bank account networks, it makes sense to split the graphs into a suspicious part and a normal part. Thus, the nodes are split into two sets: S_1 and S_2. For this we have parameter p_{set1}. Either we do a Bernoulli distribution with probability p_{set1} for each node to decide the set or we let the first p_{set1} fraction of nodes of each layer belong to the first set and the remaining to the other. In the latter way, both sets' nodes are spread evenly over the layers.

Each of the two sets then has their own weight function, w_1 and w_2. Furthermore, the odds that a candidate edge belongs to the generated graph, depends on whether the endpoints both belong to S_1, both belong to S_2 or are in different sets. Therefore, we would need three *edgeRatio* parameters or three d_{exp} parameters.

For the data sets of the results displayed in this thesis, two specific graph generation methods were used:

- The **Double Uniform** method: This method takes as parameters: $|V|$, k, $|E|$, $r_{minLayer}$, p_{set1}, w_1 and w_2 plus a seed for generation. For deciding the node sets, we use the fraction per layer method as explained previously.
- The **Double Degree** method: This method takes as parameters: $|V|$, k, d^1_{exp},d^2_{exp},$d^{\times}_{exp}$, $r_{minLayer}$, p_{set1}, w_1 and w_2 plus a seed for generation. For deciding the node sets, we use the fraction per layer method as explained previously. For deciding whether a candidate edge is in the graph, we use d^1_{exp} for the probability if both endpoints are in set 1, d^2_{exp} if both endpoints are in set 2 and $d^{\times}_{exp}$ otherwise.

7.1.2. Data Set Generation

We will give a description of each of the datasets that were used for the plots in this chapter. There is an extra parameter, *graphs per random parameter*, which is how many graphs were generated for each graph parameter that was randomly selected. The data sets can be found on the Github repo: https://github.com/Sebastiaanvk/DRS-Public.

varAll_wDUni_edges: This data set contains 1640 graphs. The graphs were generated by using the **Double Uniform** method. Most variables were taken uniformly from a range. This data set contains graphs that are small enough to find the exact solution with the MILP implementation within reasonable time.

The parameters were as follows:

- $|V| \in \{100, 101, \ldots, 300\}$
- $k \in \{3, 4, \ldots, 10\}$
- $|E| \in [200, 1500]$
- $p_{set1} \in [0.15, 0.25]$
- $w_1(x) = X_1 \cdot Y_1$, where $X_1 \sim \text{Bernoulli}(0.5)$, $Y_1 \sim \text{Uniform}(0, 1000)$
- $w_2(x) = X_2 \cdot Y_2$, where $X_2 \sim \text{Bernoulli}(0.5)$, $Y_2 \sim \text{Uniform}(0, 100)$

- $r_{minLayer} = 0.80$
- Graphs per random parameter: 5

varAll_wDUni_dDeg_2: This data set contains 2020 graphs. The graphs were generated by using the **Double Degree** method. Most variables were taken uniformly from a range. This data set contains graphs that are too large for the exact algorithm to run sufficiently fast. Therefore, only heuristic algorithms were tested on it.

The parameters were as follows:

- $|V| \in \{100, 101, \ldots, 1800\}$
- $k \in \{3, 4, \ldots, 10\}$
- $p_{set1} \in [0.30, 0.50]$
- $d^1_{exp} \in [1, 10]$
- $d^2_{exp} \in [1, 6]$
- $d^{\times}_{exp} \in [0.6, 3.5]$
- $w_1(x) = X_1 \cdot Y_1$, where $X_1 \sim \text{Bernoulli}(0.5)$, $Y_1 \sim \text{Uniform}(0, 1000)$
- $w_2(x) = X_2 \cdot Y_2$, where $X_2 \sim \text{Bernoulli}(0.5)$, $Y_2 \sim \text{Uniform}(0, 100)$
- $r_{minLayer} = 1$
- Graphs per random parameter: 5

varNodes_wDUni_edges: This data set contains 1990 graphs. The graphs are generated by using the **Double Uniform** method. Since it fixes all other parameters, it gives a good impression of the influence of the number of nodes on the running time and solution quality. Only the heuristic algorithms were tested on it.

The parameters were as follows:

- $|V| \in \{200, 201, \ldots, 2000\}$
- $k = 5$
- $|E| = 2500$
- $p_{set1} = 0.25$
- $w_1(x) = X_1 \cdot Y_1$, where $X_1 \sim \text{Bernoulli}(0.5)$, $Y_1 \sim \text{Uniform}(0, 1000)$
- $w_2(x) = X_2 \cdot Y_2$, where $X_2 \sim \text{Bernoulli}(0.5)$, $Y_2 \sim \text{Uniform}(0, 100)$
- $r_{minLayer} = 0.80$
- Graphs per random parameter: 5

varEdges_wDUni_edges: This data sets contains 2765 graphs. The graphs are generated by using the **Double Uniform** method. This data set was generated to measure the influence of the number of edges/edge density on the running time and solution quality. Only the heuristic algorithms were tested on it.

The parameters were as follows:

- $|V| = 600$
- $k = 5$
- $|E| \in \{500, 501, \ldots, 6000\}$
- $p_{set1} = 0.25$
- $w_1(x) = X_1 \cdot Y_1$, where $X_1 \sim \text{Bernoulli}(0.5)$, $Y_1 \sim \text{Uniform}(0, 1000)$
- $w_2(x) = X_2 \cdot Y_2$, where $X_2 \sim \text{Bernoulli}(0.5)$, $Y_2 \sim \text{Uniform}(0, 100)$
- $r_{minLayer} = 0.80$
- Graphs per random parameter: 5

varLayers_wDUni_edges: This data sets contains 1780 graphs. The graphs are generated by using the **Double Uniform** method. In this graph, only the number of layers varies. Once again, the graph sizes were too big to run the exact MILP algorithms.

The parameters were as follows:

- $|V| = 450$
- $k \in \{3, 4, \ldots, 25\}$
- $|E| = 3000$
- $p_{set1} = 0.25$
- $w_1(x) = X_1 \cdot Y_1$, where $X_1 \sim \text{Bernoulli}(0.5)$, $Y_1 \sim \text{Uniform}(0, 1000)$
- $w_2(x) = X_2 \cdot Y_2$, where $X_2 \sim \text{Bernoulli}(0.5)$, $Y_2 \sim \text{Uniform}(0, 100)$
- $r_{minLayer} = 0.80$
- Graphs per random parameter: 5

7.2. Setup

This section contains an explanation of the setup of the experiments and the technical specifications. The code can be found at: https://github.com/Sebastiaanvk/DRS-Public.

The experiments were done in the following way: First, the graph parameter space is defined as seen in the previous section. Then an indefinite while loop is started. Every iteration the graph parameters are uniformly selected. A number of distinct random graphs are generated using these parameters and stored. Subsequently, the algorithm implementations are run on the newly generated graph. The implementations used depend on the experiment, because some data sets contain instances that are too large for all algorithms to finish in reasonable time. In the situation that an algorithm implementation takes too long, it is aborted after a certain timeout limit. The objective value, the running time and whether the algorithm completed successfully are then stored for each algorithm implementation. This loop continues until manually terminated. The experiment can be continued again at a later point.

Other than picking uniformly from the parameter space, experiments were also done by doing a grid search over the graph parameter space. This strategy has the advantage of having the parameters evenly distributed over the instance graphs, but it has the drawback of not being able to predict the termination time of the full experiment. Since only a limited number of results could be displayed in this thesis, these experiments did not make the cut.

The experiments were run on a laptop using Windows 11 with a 3.20GHz AMD Ryzen 7 5800H processor and 16GB of RAM memory. The algorithms were implemented in C++ and the code was compiled and executed within a Windows Subsystems for Linux 2 (WSL2) environment running on Ubuntu 20-04. The WSL2 environment had access to 8GB of RAM memory. The C++ code was compiled with the g++ compiler from the GNU Compiler Collection and used the C++14 standard. It was compiled without any -O optimization flags. For the MILP, the Gurobi Optimizer version 11.0.2 was used.

7.3. Algorithm Implementations

The following algorithm implementations were used for the experiments. All of them were used for both the PDRS and NDRS problems. We have two algorithm implementations that serve as baselines. The baselines help assess the quality of the heuristic solutions, especially when the exact algorithm was not run on a data set.

MILPAlt: This is an implementation of the mixed integer linear program from Section 4.2.4 using the MILP solver from Gurobi. It finds the optimal objective value for the instances, which is useful for comparing with the heuristic algorithms. Running times are generally time-consuming, so it can only be run on small instances.

Greedy Peeling: This is an implementation of the Greedy Peeling algorithm from Section 6.2.

Greedy Single Paths: This is an implementation of the Greedy Paths algorithm from Section 6.1.

Dynamic Path: This is an implementation of the exact Dynamic Single Path algorithm from Chapter 5. This implementation was used to observe how the running time depends on the instance graph parameters. Additionally, it served as a fast algorithm to compare solution qualities against the other heuristic algorithm implementations.

Pick Everything: This is another baseline algorithm implementation. It is not discussed in other sections of the thesis. The algorithm implementation picks the maximal reachable subset of nodes in the graph. It does this by running one depth first search from the first layer and one from the last layer and taking the intersection of the explored nodes. This algorithm works, because if a node is in the intersection, then a path exists from the first layer to the last through this node, and any node on this path will be part of the intersection of the two depth first search runs as well.

7.4. Running Times

In this section, plots are displayed to show how the running times of the implementations depend on the input parameters $|V|$, $|E|$ and k. The scales of the running times can be quite diverse. For instance, the running times of the baselines are orders of magnitudes smaller than the other implementations. Nevertheless, they are still left in the plots for completeness. Moreover, linear scaling is used for the y-axis even though log-scaling would differentiate the running times of the implementations better. This is because linear scaling gives a better image of the scaling behavior of the implementations. For some results where the running time trend of specific implementations can not be seen, separate plots with solely that implementation are displayed. The shaded areas around the lines indicate a 95% confidence interval for the mean values. Since the running times were so dependent on the test data generation method, we refrain from mentioning quantitative results, instead focusing on qualitative behavior observed in the plots.

This section only contains observations of the results. Further interpretations and conclusions can be found in Section 8.2.

Running Times vs the Number of Nodes

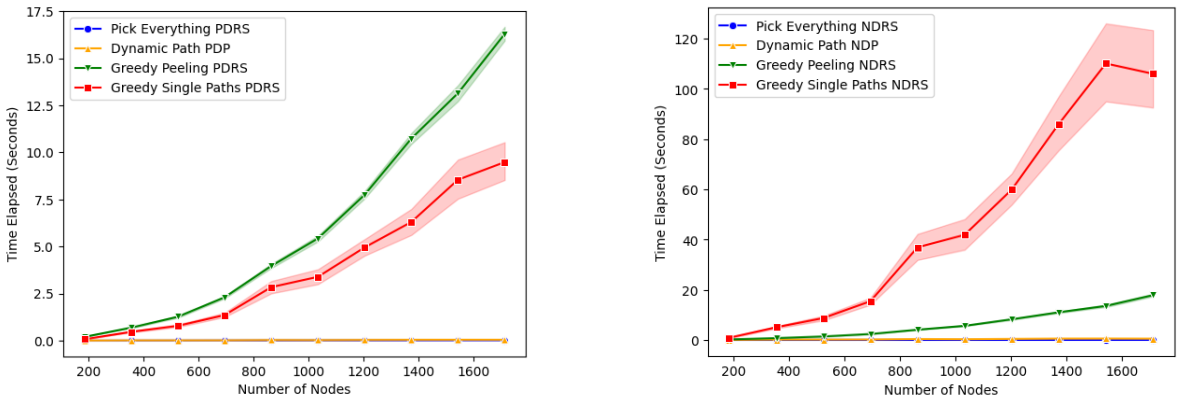


Figure 7.1: Running times as a function of the number of nodes for the heuristic algorithm implementations on data set **varAll_wDUni_dDeg_2** for PDRS (left) and NDRS (right). In this data set, all the important graph parameters vary. *Note* that the graphs were generated with expected degrees, so the number of nodes is directly related to the number of edges.

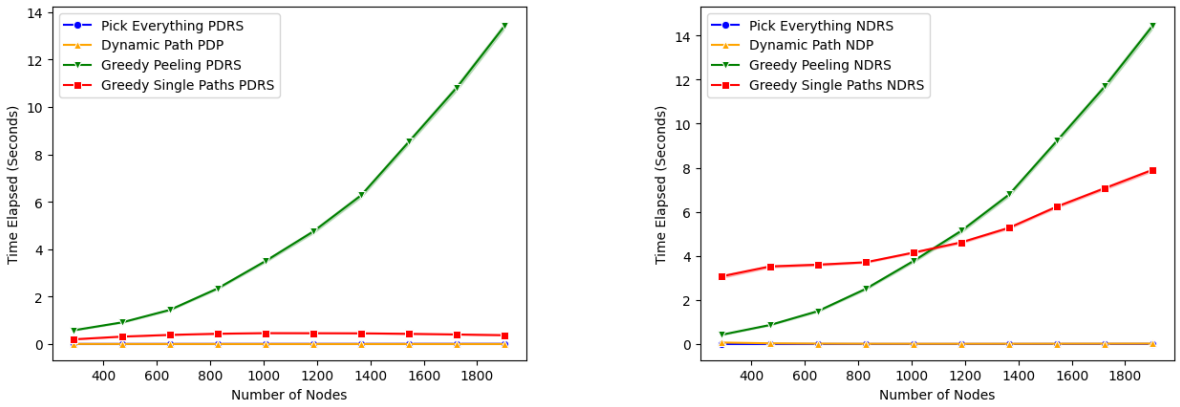


Figure 7.2: Running times as a function of the number of nodes for the heuristic algorithm implementations on data set **varNodes_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all the graph parameters are fixed except for the number of nodes.

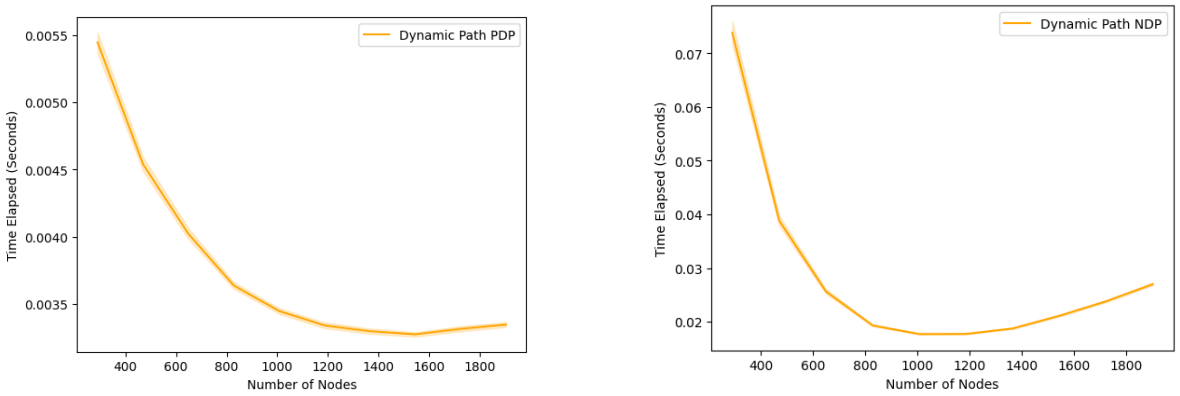


Figure 7.3: Running times as a function of the number of nodes for the **Dynamic Path** implementation on data set **varNodes_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all the graph parameters are fixed except for the number of nodes.

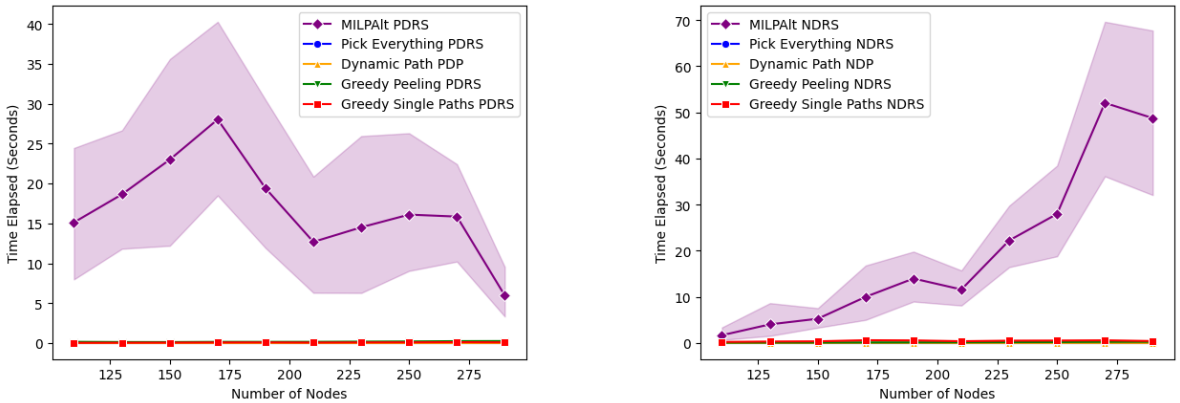


Figure 7.4: Running times as a function of the number of nodes for the heuristic algorithm implementations and the **MILPAlt** implementation on data set **varAll_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all graph parameters vary. The **MILPAlt** implementation did not terminate within 600 seconds for 28 and 37 graphs with the PDRS and the NDRS variants, respectively. Consequently, these graphs were omitted from the plots.

We observe in Figure 7.1 and Figure 7.2 that the running times of the **Greedy Peeling** implementation grow superlinearly for both data sets and both the PDRS and NDRS problems.

For the **Greedy Single Paths** implementation, we observe differences between the the data sets and PDRS and NDRS. In Figure 7.1, we observe that the running times grow superlinearly and especially for NDRS, they grow rapidly as the number of nodes increases. In Figure 7.2 the running times stay constant as the number of nodes increases for PDRS and they appear to grow linearly with a slight bend for NDRS.

In Figure 7.3, we observe that the running times of the **Dynamic Path** implementation initially decrease sharply as the number of nodes decreases, but start rising again towards the upper end.

In Figure 7.4, we observe that the running times of the **MILPAlt** implementation are quite erratic for the PDRS variant, but for the NDRS problem, the running times increase superlinearly.

Running Times vs the Number of Edges

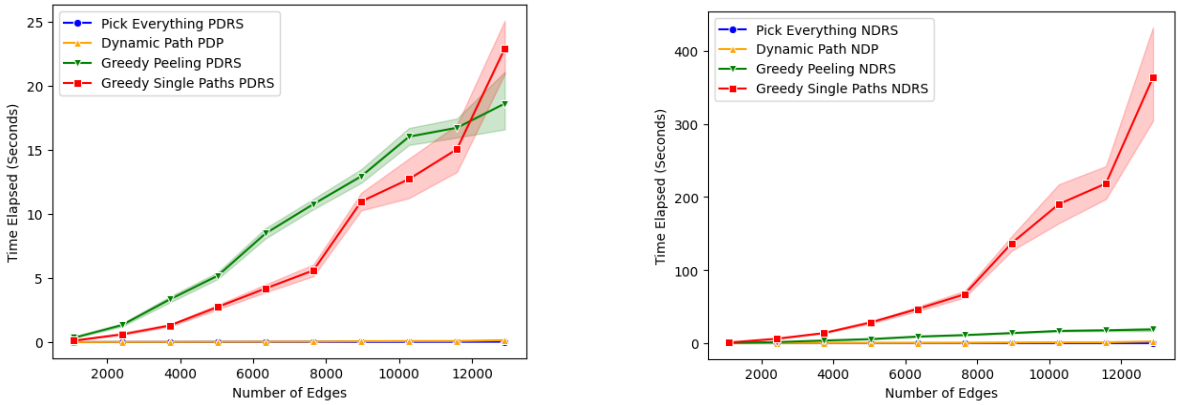


Figure 7.5: Running times as a function of the number of edges for the heuristic algorithm implementations on data set **varAll_wDUni_dDeg_2** for PDRS (left) and NDRS (right). In this data set, all the important graph parameters vary. **Note** that the graphs were generated with expected degrees, so the number of edges is directly related to the number of nodes.

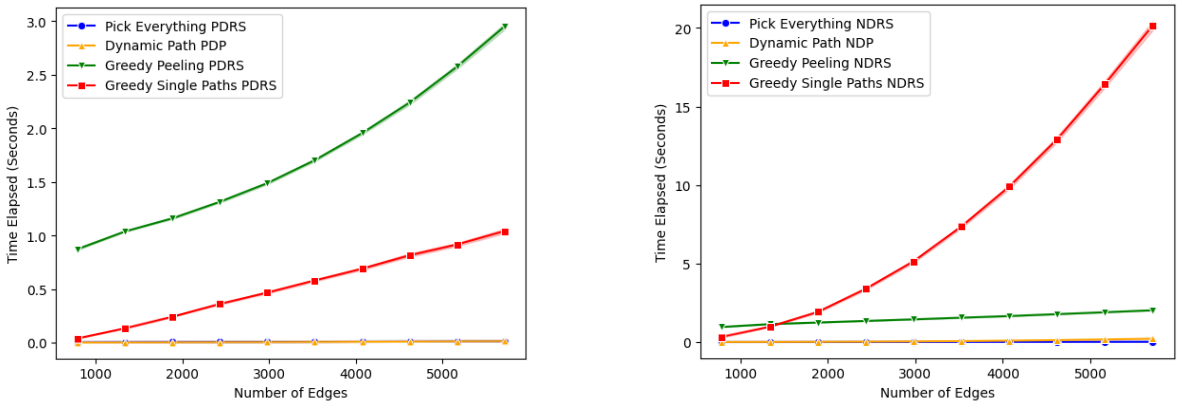


Figure 7.6: Running times as a function of the number of edges for the heuristic algorithm implementations on data set **varNodes_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all the graph parameters are fixed except for the number of edges.

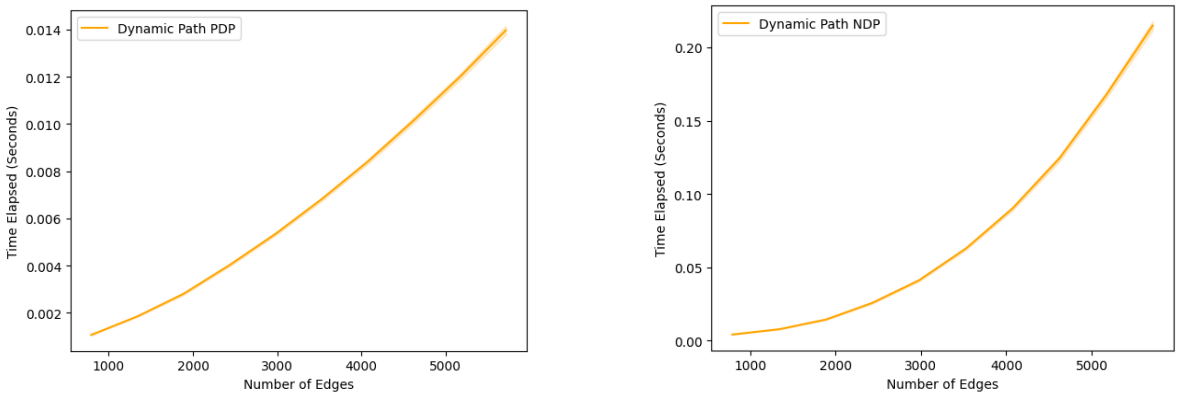


Figure 7.7: Running times as a function of the number of edges for the **Dynamic Path** implementation on data set **varEdges_wDUni_edges** for PDRS (left) and NDRS (right). In this data, set all the graph parameters are fixed except for the number of edges.

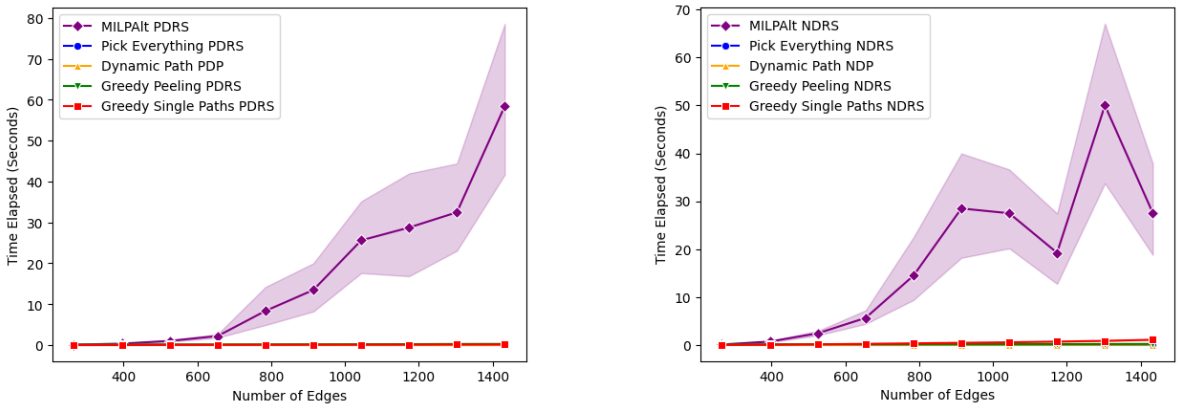


Figure 7.8: Running times as a function of the number of edges for the heuristic algorithm implementations and the **MILPAlt** implementation on data set **varAll_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all graph parameters vary. The **MILPAlt** implementation did not terminate within 600 seconds for 28 and 37 graphs with the PDRS and the NDRS variants, respectively. Consequently, these graphs were omitted from the plots.

We observe in Figure 7.5 and Figure 7.6 that the running times of the **Greedy Peeling** implementation appear to grow linearly for both data sets and both the PDRS and NDRS problems. Although the PDRS plot in Figure 7.7 shows a slight convexity.

For the **Greedy Single Paths** implementation, we observe differences between the the data sets and PDRS and NDRS. In Figure 7.5, we observe that the running times grow superlinearly and especially for NDRS, they grow very rapidly as the number of edges increases. In Figure 7.6 the running times increase linearly as the number of edges increases for PDRS and they grow superlinearly for NDRS.

In Figure 7.7, we observe that the running times of the **Dynamic Path** implementation increase superlinearly as the number of edges increase. The NDRS variant running times grow more steeply than the PDRS variant.

In Figure 7.8, we observe that the running times of the **MILPAlt** implementation grow superlinearly for PDRS and NDRS as the number of edges increases.

Running Times vs the Number of Layers

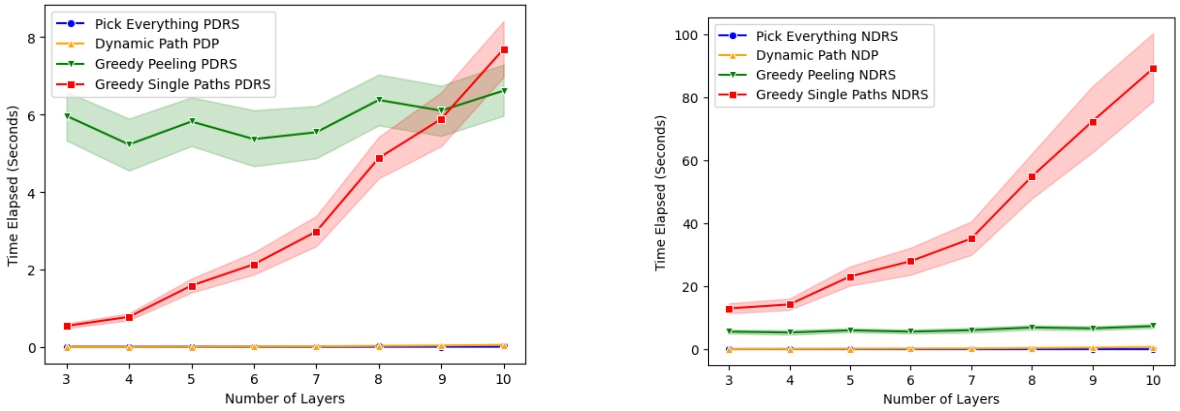


Figure 7.9: Running times as a function of the number of layers for the heuristic algorithm implementations on data set **varAll_wDUni_dDeg_2** for PDRS (left) and NDRS (right). In this data set, all the important graph parameters vary.

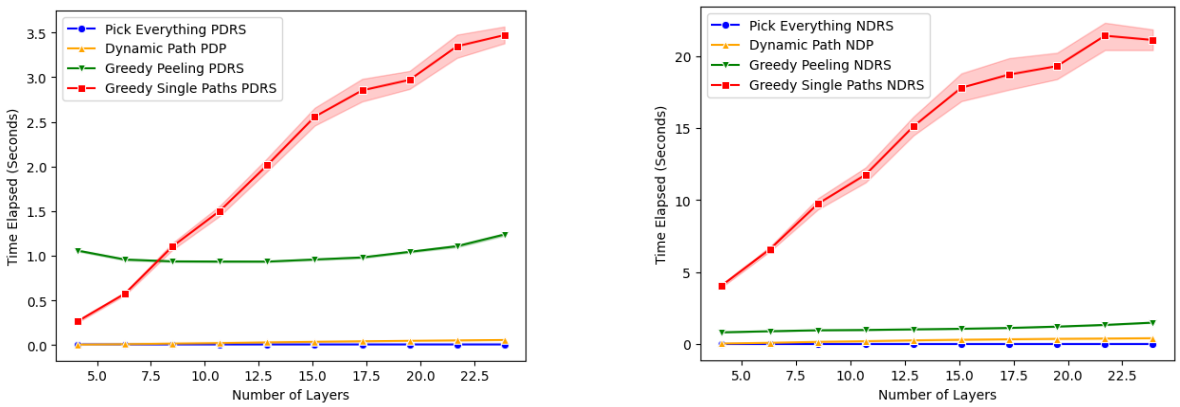


Figure 7.10: Running times as a function of the number of layers for the heuristic algorithm implementations on data set **varNodes_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all the graph parameters are fixed except for the number of layers.

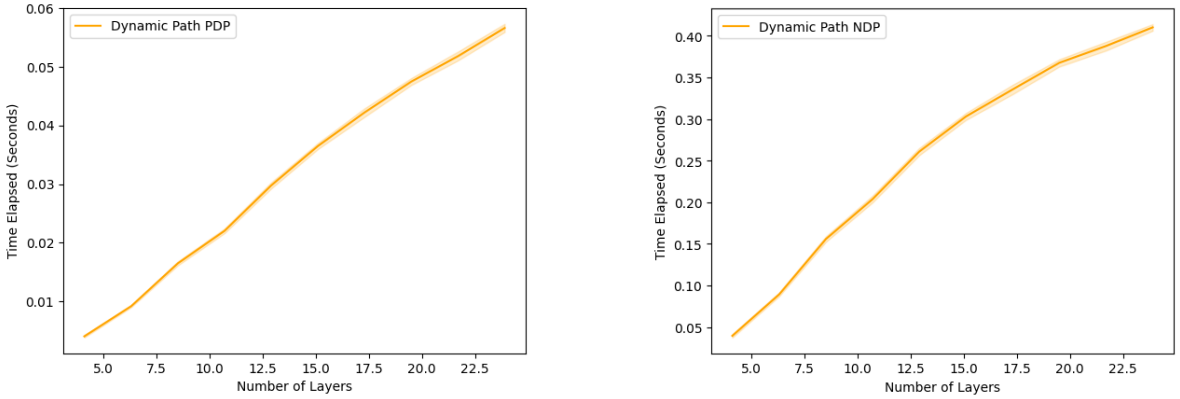


Figure 7.11: Running times as a function of the number of layers for the **Dynamic Path** implementation on data set **varNodes_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all the graph parameters are fixed except for the number of layers.

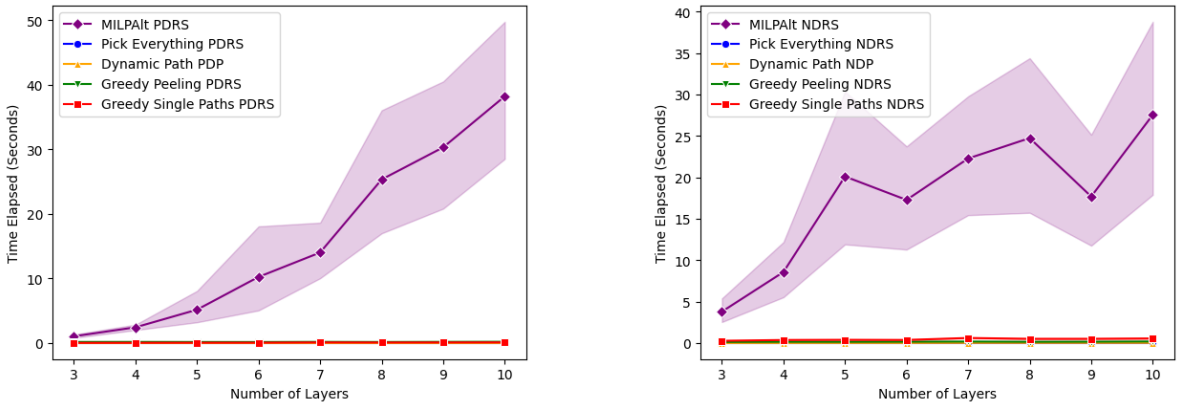


Figure 7.12: Running times as a function of the number of layers for the heuristic algorithm implementations and the **MILPAlt** implementation on data set **varAll_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all graph parameters vary. The **MILPAlt** implementation did not terminate within 600 seconds for 28 and 37 graphs with the PDRS and the NDRS variants, respectively. Consequently, these graphs were omitted from the plots.

We observe in Figure 7.9 and Figure 7.10 that the running times of the **Greedy Peeling** implementation remain constant for both data sets and both the PDRS and NDRS problems. However, the plot of NDRS on **varLayers_wDUni_edges** increases linearly with a very minor incline.

The running times of the **Greedy Single Paths** implementation grow superlinearly on the Figure 7.9 data set. However, on the Figure 7.10 the trend is unclear. Initially, it rises linearly, but towards the higher number of layers, the slope decreases. In this region, the plot also becomes more erratic and the confidence interval widens.

We observe in Figure 7.11, that the **Dynamic Path** implementation running times grow sublinearly for both PDRS and NDRS as the number of layers increases.

In Figure 7.12, we observe that the **MILPAlt** implementation running times appear to grow linearly for PDRS and sublinearly for NDRS. However, the confidence interval is quite wide, so the trends should be interpreted with caution.

7.5. Solution Quality

In this section, we show the quality of the solutions as a function of the main graph instance parameters: $|V|$, $|E|$ and k. Per parameter, there is one plot with graph instances small enough to run the exact algorithm, one plot with graph instances where all the parameters vary and one plot with graph instances where only the measured parameter varies. The shaded areas around the line indicate a 95% confidence interval. The solution quality is highly dependent on the test data generation method, so we once again focus on the qualitative behavior of the plots as opposed to mentioning the quantitative results.

For interpretations and conclusions, the reader is again referred to Section 8.1.

Solution Quality vs the Number of Nodes

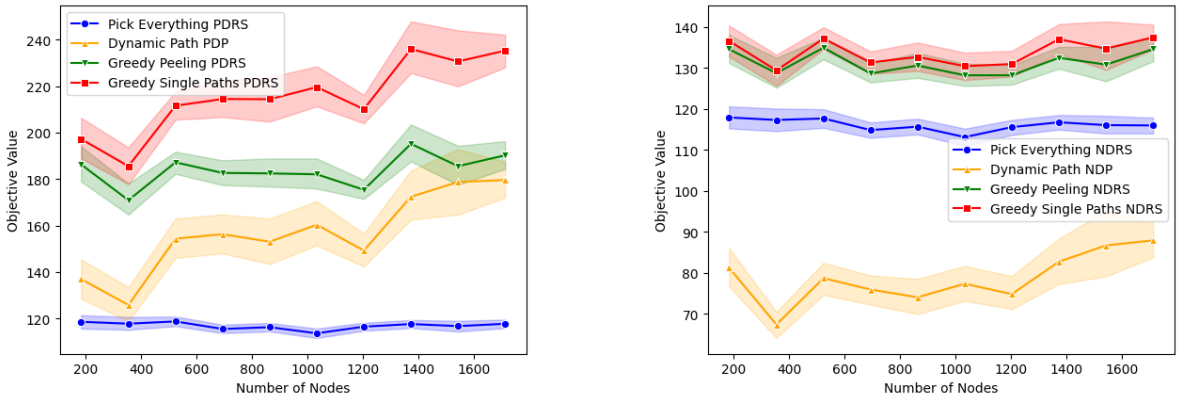


Figure 7.13: Solution quality as a function of the number of nodes for the heuristic algorithm implementations on data set **varAll_wDUni_dDeg_2** for PDRS (left) and NDRS (right). In this data set, all the important graph parameters vary. *Note* that the graphs were generated with expected degrees, so the number of nodes is directly related to the number of edges.

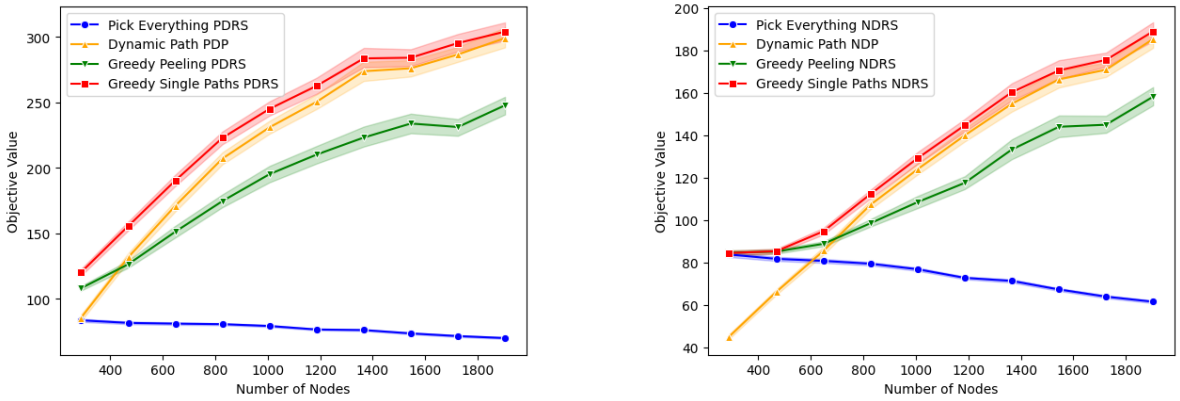


Figure 7.14: Solution quality as a function of the number of nodes for the heuristic algorithm implementations on data set **varNodes_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all the graph parameters are fixed except for the number of nodes.

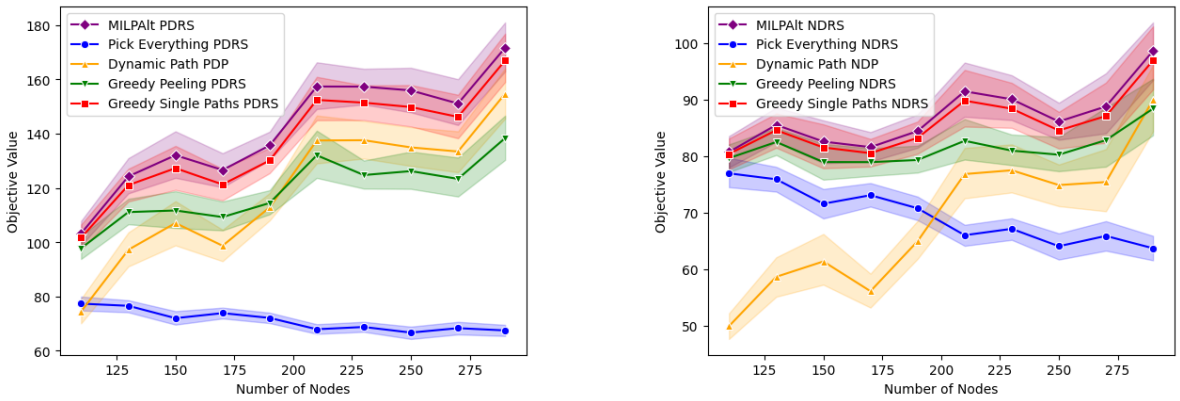


Figure 7.15: Solution quality as a function of the number of nodes for the heuristic algorithm implementations and the **MILPAlt** implementation on data set **varAll_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all graph parameters vary. The **MILPAlt** implementation did not terminate within 600 seconds for 28 and 37 graphs with the PDRS and the NDRS variants, respectively. The best solutions found before timeout were used.

We observe in Figure 7.13, that the objective values stay relatively consistent in the data set where the graphs are generated with the expected degrees method.

However, we observe in Figure 7.14 and Figure 7.15, that as the number of nodes increases, the solution quality of the **Dynamic Path** implementation actually overtakes the **Greedy Peeling** implementation on the graphs where the number of edges is fixed. The objective value gets very close to the **Greedy Single Paths** implementation's value. When the density is high, the **Dynamic Path** implementation gives substantially results.

In all figures we see that the **Greedy Single Paths** implementation outperforms the **Greedy Peeling** implementation. In Figure 7.15, we observe that the **Greedy Single Paths** implementation gets very close to the exact value on this data set.

Solution Quality vs the Number of Edges

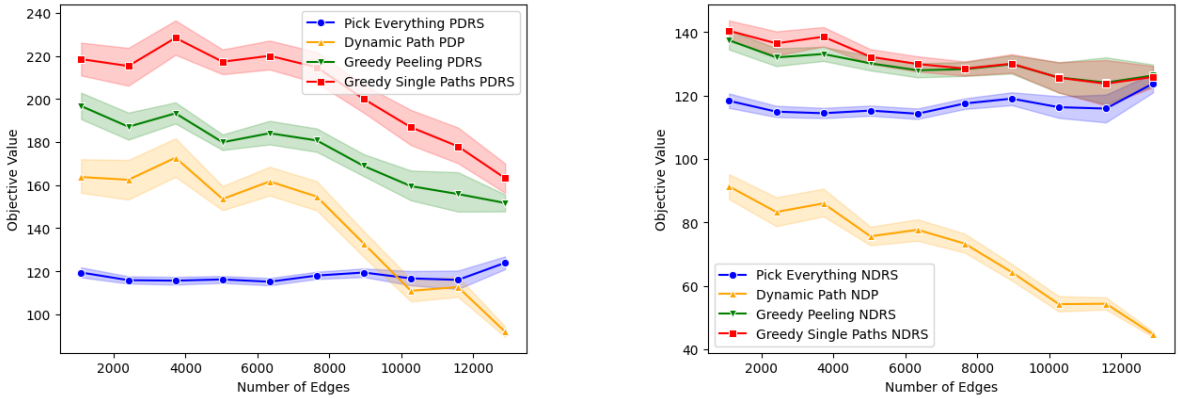


Figure 7.16: Solution quality as a function of the number of edges for the heuristic algorithm implementations on data set **varAll_wDUni_dDeg_2** for PDRS (left) and NDRS (right). In this data set, all the important graph parameters vary. **Note** that the graphs were generated with expected degrees, so the number of edges is directly related to the number of nodes.

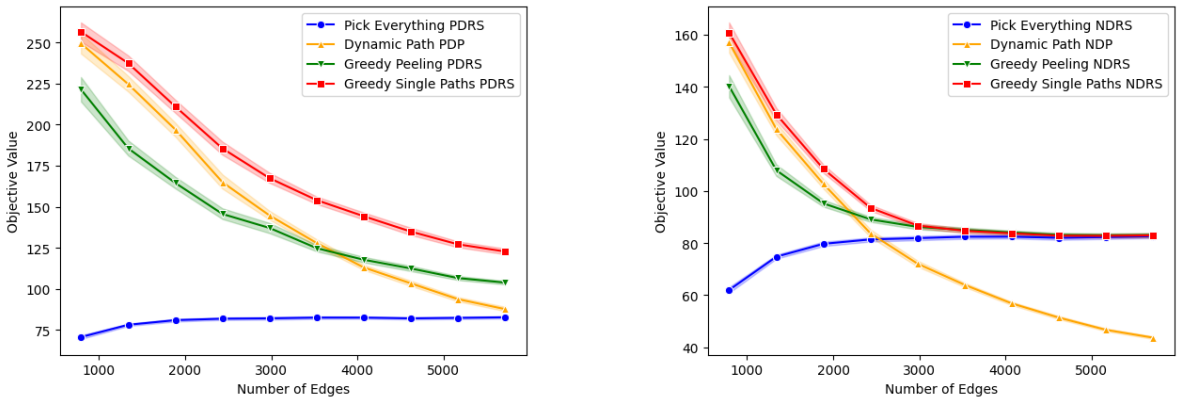


Figure 7.17: Solution quality as a function of the number of edges for the heuristic algorithm implementations on data set **varNodes_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all the graph parameters are fixed except for the number of edges.

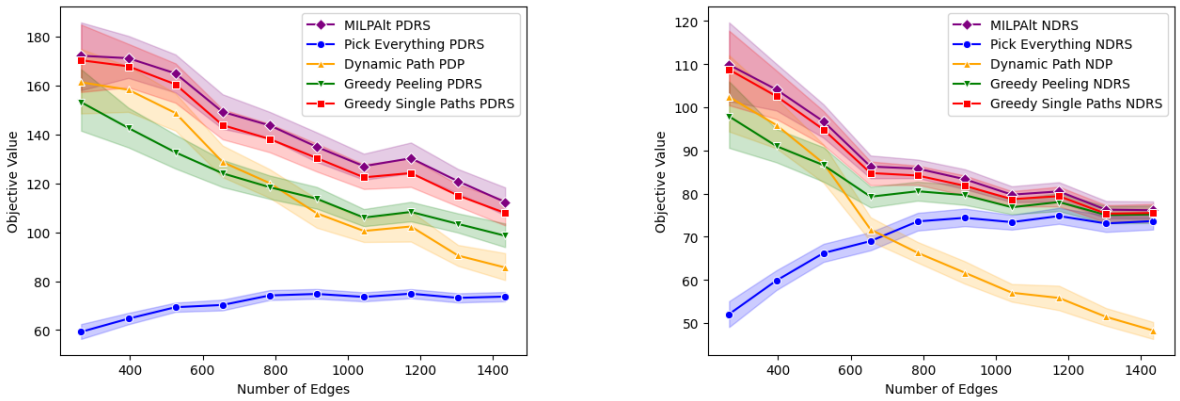


Figure 7.18: Solution quality as a function of the number of edges for the heuristic algorithm implementations and the **MILPAlt** implementation on data set **varAll_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all graph parameters vary. The **MILPAlt** implementation did not terminate within 600 seconds for 28 and 37 graphs with the PDRS and the NDRS variants, respectively. The best solutions found before timeout were used.

In Figure 7.16, Figure 7.17 and Figure 7.18, we observe the same pattern as before. The **Greedy Single Paths** implementation consistently gives the highest objective value and for low density graphs, the **Dynamic Path** baseline gives relatively high objective values. As the density increases, **Greedy Single Paths**, **Greedy Peeling** and **Pick Everything** converge to similar values.

Solution Quality vs the Number of Layers

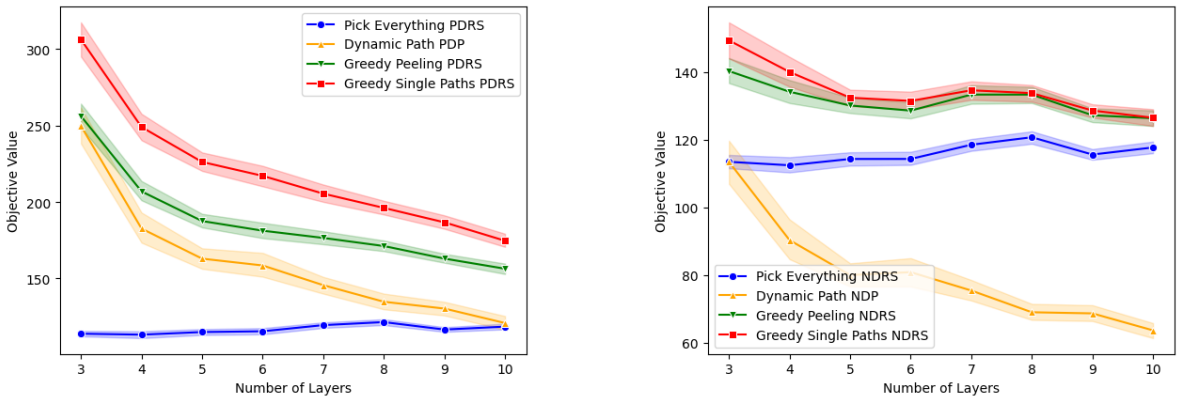


Figure 7.19: Solution quality as a function of the number of layers for the heuristic algorithm implementations on data set **varAll_wDUni_dDeg_2** for PDRS (left) and NDRS (right). In this data set, all the important graph parameters vary.

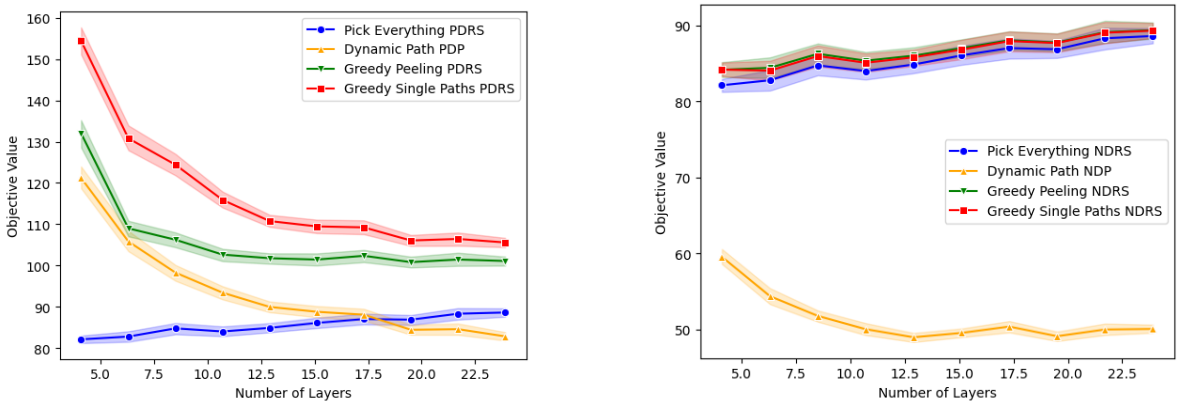


Figure 7.20: Solution quality as a function of the number of layers for the heuristic algorithm implementations on data set **varNodes_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all the graph parameters are fixed except for the number of layers.

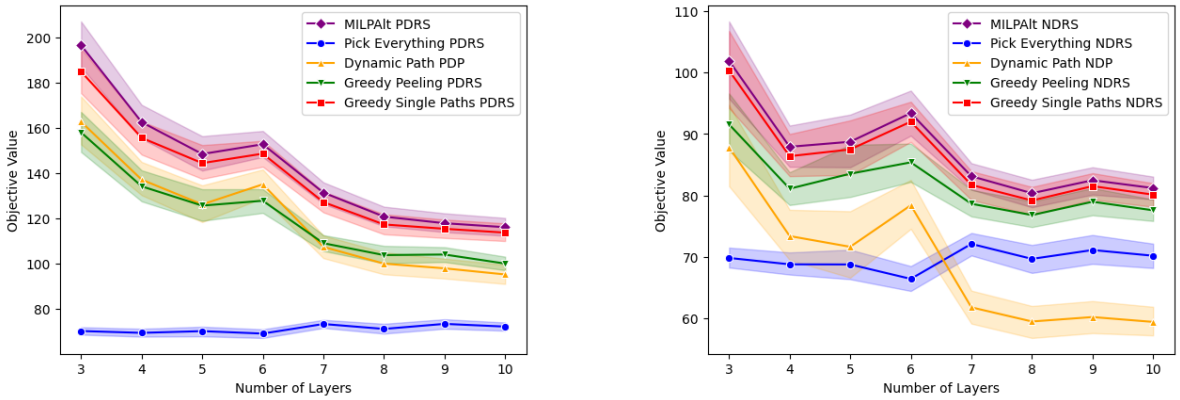


Figure 7.21: Solution quality as a function of the number of layers for the heuristic algorithm implementations and the **MILPAlt** implementation on data set **varAll_wDUni_edges** for PDRS (left) and NDRS (right). In this data set, all graph parameters vary. The **MILPAlt** implementation did not terminate within 600 seconds for 28 and 37 graphs with the PDRS and the NDRS variants, respectively. The best solutions found before timeout were used.

In the Figure 7.19 and Figure 7.20 NDRS plots, we observe that the **Greedy Single Paths** and **Greedy Peeling** algorithm implementations yield similar objective values and the objective values of the **Pick Everything** implementation get quite close, especially as the number of layers increases. However, the **Dynamic Path** implementation fares quite poorly.

In the Figure 7.19 and Figure 7.20 PDRS plots, the **Greedy Single Paths** implementation yields higher objective values than the **Greedy Peeling** implementation and the **Dynamic Path** implementation is slightly lower for low numbers of layers, but the difference increases as the number of layers increases. The **Pick Everything** implementation starts rather poorly, but gets closer to the other algorithm implementations as the number of layers increases and this implementation eventually overtakes the **Dynamic Path** implementation.

In Figure 7.21, we observe that the **Greedy Single Paths** implementation gets very close to the optimal value for this data set. The **Greedy Peeling** implementation gives slightly lower objective values. The **Dynamic Path** implementation does relatively well for smaller numbers of layers.

8

Conclusion

This chapter provides an overview of the main conclusions drawn from the research in this thesis. It is divided into three parts. The first part summarizes the theoretical results from their respective chapters. The second part concerns the experimental results. Whereas Chapter 7 only contains observations based on the experiments, the section in this chapter goes more in depth on what can be concluded from the experiments and how these behaviors can be explained. Lastly, the third section contains a reflection of how aspects of this thesis can be approached in a different way in hindsight and gives suggestions for future work.

8.1. Discussion of Theoretical Results

In this thesis, we have introduced and defined two novel optimization problems on k-layer graphs: the Predecessor Dense Reachable Subgraph (PDRS) problem and the Neighborhood Dense Reachable Subgraph (NDRS) problem.

Both problems were shown to be NP-hard by reductions from the Minimum Vertex Cover Problem. These reductions show that the problems can not be solved in polynomial time even for a fixed number of layers k. Knowing the problem is NP-hard justifies using the exponential-time MILP algorithms and developing heuristic algorithms.

Afterwards, different integer linear programming formulations were presented that give exact solutions to the problems. For this, initially a simple integer linear-fractional program was defined. Then through multiple steps and proofs, a mixed integer linear program was derived.

The variant of the original problems where we restrict ourselves to picking only a single path in the graph, was shown to be polynomial-time solvable. The algorithm used a recurrence relation that depended on the maximum path that ends in a specific node and has a specific number of neighbors. For the NDRS problem, it also depended on the second to last element of the path. The correctness of the recurrence relation and the algorithm itself were formally proven. The algorithms have complexities of $O(|V|\,\Delta^2 k)$ and $O(|V|\,\Delta^4 k)$ for PDRS and NDRS, respectively, where Δ is the maximum in- or out-degree of the graph.

In addition to the exact linear programming formulation, we designed multiple heuristic algorithms. The first algorithm uses the exact algorithms for the single path variant to add new nodes to its solution set. This is done for multiple iterations until no new nodes with non-zero weight can be added. It then returns the best intermediate solution among all iterations. This algorithm has complexity $O(|V|^2\,\Delta^2 k)$ for PDRS and complexity $O(|V|^2\,\Delta^4 k)$ for NDRS. The second algorithm is a peeling algorithm that starts with an allowed neighborhood and looks at the maximum weight that can be achieved if the (predecessor) neighborhood of the selected set needs to be contained to the allowed neighborhood. Every iteration, the allowed neighborhood removes the node that decreases the remaining weight the least. In this step, the reachability constraint on the picked set is considered when removing the neighbor. Once again, the algorithm returns the best intermediate solution out of all iterations. This algorithm has

a worst-case complexity of $O(|V|^2 (|V| + |E|))$ for both PDRS and NDRS.

8.2. Discussion of Experimental Results

In this section, we discuss the conclusions that can be drawn from the experimental results. The graph instances and data sets that the implementations were tested one, were fully synthetically generated. This comes with severe limitations, because the method of generating the data sets can influence the performance of the algorithm implementations, both in running time and solution quality. Many data set generation methods were considered and the results shown in this thesis are a selection that attempt to showcase the trends of the implementations in a relatively clear and unbiased manner. One limitation, however, is the interdependency of the graph parameters, for example if we generate a data set with a fixed number of edges, the number of nodes will influence the average degree and if we generate the graphs using expected degrees, the number of nodes is directly proportional to the number of edges. It was therefore decided to focus on qualitative conclusions, e.g. does the shape of the plot coincide with the complexities of the algorithms, as opposed to more rigid quantitative conclusions, e.g. polynomial regression models, which would give a false idea of exactness.

Running Times of Implementations

Dynamic Path: When the number of nodes increases but the other parameters are fixed, the running time initially decreases rapidly but eventually starts increasing again. This is quite interesting, but it is explainable by the complexities: Since we have the $|V|$ factor in the complexities, the expectation would be that the running time grows linearly. However, the increase in nodes makes the average degree decrease so the number of different sizes of neighborhoods of the paths decreases. Eventually, the degree gets so low, that $|V|$ starts dominating again, since we still need to loop through all the nodes when building the dynamic programming table, so the running time starts increasing as the number of nodes increases.

When the number of edges increases, we see a superlinear running time increase. The NDRS implementation's curve is steeper. This seems to align with the Δ^2 factor for the PDRP complexity and Δ^4 factor for NDRP.

When the number of layers increases and the number of edges is fixed, the running time appears to grow linearly initially, which coincides with the k factor in the complexities, but the growth actually appears to flatten. This can be partly explained by the following effect: If we have $|k|$ layers, then the average out-degree is $(|E|/(k-1))/(|V|/k)$, since the edges are split over $k-1$ locations between the layers. The average out-degree then scales with $k/(k-1)$, because $|E|$ and $|V|$ are fixed for this data set. So the average degree decreases as the number of layers increases and is especially high for the lower number of layers. This also explains why the effect is stronger for the NDRS implementations, since that complexity scales with Δ^4 as opposed to Δ^2.

Greedy Single Paths: In the data set where the number of nodes increases, but the graphs are created with the expected degree parameters, the behavior is to be expected, since we have a $|V|^2$ in the complexities for both PDRS and NDRS and the plot seems to indeed grow quadratically. In the data set where the number of nodes increases, but the number of edges is fixed, we have interesting behavior. The **Greedy Single Paths** implementation depends on **Dynamic Path**, for which the behavior is quite non-standard, since the running time drops very steeply initially but ends up increasing again. The complexity of the **Greedy Single Paths** algorithm has a $|V|^2$ factor which largely compensates for this effect. Therefore it appears that we have constant and linear behavior for PDRS and NDRS respectively, although with a slight bend, but it could be that the function behaves very differently for higher or lower numbers of nodes. This would require data sets with larger instances.

We see in the plot based on the data set where the graphs are generated with the expected degree parameter, that the running time seems to increase quadratically or faster with the number of edges. The number of edges in these plots is dependent on both the number of nodes and the expected degree. The complexity for the PDRS algorithm has factor $|V|^2 \cdot \Delta^2$, which is directly proportional to $|E|^2$. In the NDRS algorithm we have the factor $|V|^2 \Delta^4$, so the worst-case theoretical running time grows even faster than $|E|^2$. The running times for NDRS are indeed much higher.

In the data set, where the number of nodes is a graph instance parameter and the nodes and layers are

fixed, the implementation appears to grow linearly for PDRS and superlinearly for NDRS. The former result is quite surprising, because the **Dynamic Path** running time grows superlinearly. This could be explained by the fact that if there is a larger number of edges, then the odds of adding multiple new nodes per iteration are higher, so the implementation needs fewer iterations. Consequently, the fewer number of iterations could compensate for the longer running time it takes to calculate the next best single path. In the NDRS case, however, this does not apply, because the **Dynamic Path** running time grows with a stronger curve for NDRS.

In **varAll_wDUni_dDeg_2**, the running time grows superlinearly as the number of layers increases, even though the complexity has just the k factor. This can be explained by the fact that the graphs are generated with the expected degree method: If between the last layer and the first layer, edges would be generated in the same way, then the number edges would not depend on the number of layers. However, we do not generate those edges, so the number of edges in the graph scales with $(k-1)/k$, since the edges between the last layer and the first would have been the $(1/k)$th portion of the total number of edges.

In **varLayers_wDUni_edges**, the running time scales proportionally to the running times of the **Dynamic Path** implementation, which is to be expected.

Greedy Peeling: When looking at the relationship between the running time and the number of nodes, we observe that it scales closer to quadratically than cubically as the worst-case complexity would suggest. This does not give a contradiction, because the implication depth first search algorithm will generally not traverse all the nodes. As a matter of fact, how long it takes before the implication dfs terminates, is much more dependent on the average degree, which is taken uniformly in **varAll_wDUni_dDeg_2** and actually is inversely proportional to the number of nodes in **varNodes_wDUni_edges**.

The relationship between the running time and the number of edges is less clear from the plots. In the **varAll_wDUni_dDeg_2**, the relationship seems linear or superlinear. It is hard to draw conclusions from this, since the number of edges is dependent on the other graph parameters. In **varEdges_wDUni_edges**, the relationship is linear with a slight convexity. This convexity could be explained by the fact that when the number of edges is low we see that paths get cut off earlier in the implementation. This means that new nodes get discovered in the implication depth first search procedure in earlier iterations, because with fewer edges, it is easier to block off all in-going or out-going edges of nodes.

MILPAlt: The running time of the Gurobi implementation is very unpredictable and erratic. The running time on two graphs with the same $|V|$, $|E|$ and k parameters, can differ by $1000\times$. However, we do observe that for all the three parameters, the running time increases steeply. Only for the PDRS implementation in **varAll_wDUni_edges**, we see that the relationship between the running time and the number of nodes is not that clear. This is unexpected, because the number of nodes decides the number of integer variables in the MILP. This could be caused by a couple of instances that have exceptionally high running times. The higher number of edges causes the constraints to depend on larger sets of nodes, possibly causing the running times to increase. The higher number of layers may cause the reachability constraint to become more difficult to take into account for the MILP.

Solution Quality of Implementations

As highlighted before, the solution quality of the algorithm implementations is highly dependent on the data set and graph generation methods used. Therefore, any conclusions on the quality should be interpreted with caution. Nevertheless, there are some patterns and consistencies that are worth mentioning.

We see that in each data set, the **Greedy Single Paths** implementation gives the highest objective values out of all the heuristics. On **varAll_wDUni_edges**, the objective values found by **Greedy Single Paths** are very close to the optimal value. The **Greedy Peeling** implementation is generally quite close to the **Greedy Single Paths** value, where in the worst case, it is around 10-20 percent lower.

The lower the density of edges, the higher the objective value, relatively speaking, found by the **Dynamic Path** baseline. For low densities, it even overtakes the **Greedy Peeling** implementation. The **Pick Everything** baseline finds relatively better objective values when the density is higher.

Overall Performance Comparison

For our data sets, the **Greedy Single Paths** implementation consistently finds the highest objective value solutions out of all the heuristic algorithms. However, its running time generally scales much faster than the **Greedy Peeling** implementation as the input size increases, especially for the NDRS problem. The **Greedy Peeling** implementation finds worse objective values in general, but there are examples where **Greedy Peeling**'s running time is much lower than **Greedy Single Paths**'s running time. The **Greedy Peeling** implementation finds relatively good solutions for high density graphs and as the density drops it starts giving worse solutions than even the **Dynamic Path** baseline. Further research would have to be done to see how the algorithms compare on realistic or ground truth data sets.

8.3. Reflection and Future Work

This section contains a reflection and suggestions for future work. The purpose of this section is to reflect on the approaches taken in this thesis and highlight alternative strategies that could be considered. It is therefore highly related to suggestions for future work. However, not all suggestions are of this form, some are just ideas for different problems or different algorithms altogether.

Algorithm Implementations

First of all, since the implementations were written in the programming language C++, one straightforward way to speed up the algorithm implementations, would have been to compile using the -o3 optimization flag. Leaving out the optimization flag still allows one to observe how the graph instance parameters affect the running times and relative solution quality. However, having faster run times would allow one to run the algorithms on larger data sets or data sets with larger graph instances in the same time span. The same applies to running the implementations on stronger hardware. For example, the experiments could have been run on a remote server with a faster CPU.

In the **Dynamic Path** implementation, ordered maps were used. Using unordered maps (hash tables) could potentially decrease the running times. The faster lookup times might compensate for the extra overhead and memory needed for using a hash table.

In the **Greedy Single Paths** implementation, a new dynamic programming table gets created and calculated every iteration . An angle that could be invested is whether a method exists to maintain this dynamic programming table and update it efficiently whenever a new path gets selected. This would speed up the algorithm significantly, while maintaining the solution quality.

Likewise, in the **Greedy Peeling** implementation, the implication depth first search algorithm gets called every iteration for every remaining neighbor in the graph. If there was a way to reuse the information from the previous iteration, it could greatly speed up the running time of the implementation. Furthermore, the iterative peeling scheme for ratios of submodular functions (Chekuri, Quanrud, and Torres 2020; L. Li et al. 2024) could potentially be applied to the **Greedy Peeling** algorithm. One could prove that the algorithm can be repeated to attain arbitrary precision, but even without such a proof, it would be interesting to see how the solutions improve after multiple iterations.

Data Sets

One significant shortcoming of this thesis is the lack of ground truth data sets and the lack of existing data sets for the PDRS and NDRS problems in general. Since we are dealing with a novel problem, it is to be expected that no readily available data sets were found. We therefore had to develop our own method for generating instance graphs. Many different methods were considered and tested, each with their own advantages and disadvantages. For instance, creating the data set where the portion of graphs with a near-optimal trivial solution is low, already takes finetuning effort. One also does not want to deliberately or accidentally create a data set for which the new algorithms fare exceptionally well. In general, these synthetic data sets are useful for showing the influence of the parameters on the running times, but for verifying the quality of the solutions of the algorithms there are many caveats to consider. One could say that the quality of the solutions say just as much about the data set as the algorithm implementations themselves. An approach that one could take to circumvent this problem, is to first define the data sets that the algorithms will be tested on, before designing the algorithms.

Another strategy for generating data sets is to create problem instances from existing data sets. There are many methods that can be utilized to create such problem instances. For instance, one could look for problems defined on similar types of graphs. Also, one could use real data sets of, for example, bank transfers. These data sets typically only contain weighted edges, so a vertex weight function to quantify the 'suspiciousness' of each bank account would still need to be defined. On top of that, these instances are generally much too large to run most algorithm implementations on, so it would be required to design a method that samples parts of the graph. Like fully synthetic data, these methods require arbitrary parameters that can influence the quality of the algorithm implementations.

The generation of synthetic data sets for money laundering detection is an area that has been researched in its own right. Experiments could be run on data sets generated with methods proposed in works such as (Altman et al. 2024).

Alternative Problem Statements

The matter of money laundering detection could be modeled into many alternative optimization problems. Some of the problems that could be looked into are:

- One could use different objective functions. For example, one could take the same types of instance graphs and instead of $|N^-(S) \cup S|$ in the denominator we could have simply $|N^-(S)|$ or $|S|$.
- One could use weighted edges instead of vertices. This would be quite natural, because in a regular bank transaction network, the edges are the amount of money that gets transferred from one bank account to the other.
- One could use directed graphs instead of k-layer graphs and define the same problem using a source and a sink for the reachability constraint. As a matter of fact, the implementations of the algorithms in this paper could be rewritten with minimal effort to work on acyclic directed graphs.
- One could have the reachability constraint as a penalty on our objective function instead of as a hard requirement. This could potentially make the problem significantly easier
- One could define the weight of the vertices dynamically. For instance, in the FlowScope paper (X. Li et al. 2020), the weights of the paths depend on the difference between the in- and out-going edges within the selected subset. Alternatively, we could also let the weight of the vertex depend on the ratio of weights of the incident edges within the currently selected set to the total weight of the incident edges. This coincides with a natural intuition for 'cliquey' behavior of a group of vertices.
- One could broaden the definition of the PDRS and NDRS problems to allow negative vertex weights. The algorithms in this thesis do not rely on the fact that the weights are non-negative, so they could still be applied to this new problem. The broader definition might yield other interesting results though.

Alternative Algorithms

This thesis only has a small section of all the different algorithm strategies that were considered. Implementing and researching them all would be infeasible time-wise. Algorithms that run much faster are especially interesting, even if the quality of their solutions is worse, since these could be implemented for larger graph instances than the ones seen in this thesis. Here are some other ideas for developing algorithms that could potentially be adequate.

- A greedy single paths algorithm that uses a greedy algorithm for choosing the next path to add instead of the exact path algorithm. This would greatly speed up the running time and allow the implementation to run on much larger graphs.
- Algorithms that utilize the exact polynomial-time linear programming algorithm for the bipartite case from (L. Li et al. 2024). Using these locally optimal solutions could give a good heuristic for building a solution for the whole graph by going from layer to layer.
- Algorithms that use local search methods, for instance with simulated annealing. Local search methods, where we iteratively alter the current solution by removing or adding nodes, have been shown to work well for other combinatorial optimization problems.

- Algorithms based on machine learning. This is a technique that has been applied to many other optimization problems and often yields satisfactory results.
- The greedy peeling algorithm where we do not calculate all the implications at each iteration. This gives another heuristic algorithm where we vastly lower the running time at the expense of precision.

References

Altman, Erik et al. (2024). *Realistic Synthetic Financial Transactions for Anti-Money Laundering Models*. arXiv: 2306.16424 [cs.AI]. url: https://arxiv.org/abs/2306.16424.

Applegate, David et al. (Jan. 2006). "The Traveling Salesman Problem: A Computational Study". In: *The Traveling Salesman Problem: A Computational Study*.

Aslam, Sidra, Ala Altaweel, and Ali Bou Nassif (2023). *Optimization Algorithms in Smart Grids: A Systematic Literature Review*. arXiv: 2301.07512 [cs.NE]. url: https://arxiv.org/abs/2301.07512.

Bakhshinejad, Nazanin et al. (Oct. 2022). "A Survey of Machine Learning Based Anti-Money Laundering Solutions". In.

Boob, Digvijay et al. (2020). "Flowless: Extracting Densest Subgraphs Without Flow Computations". In: *Proceedings of The Web Conference 2020*. WWW '20. Taipei, Taiwan: Association for Computing Machinery, pp. 573–583. isbn: 9781450370233. doi: 10.1145/3366423.3380140. url: https://doi.org/10.1145/3366423.3380140.

Charikar, Moses (2000). "Greedy Approximation Algorithms for Finding Dense Components in a Graph". In: *Approximation Algorithms for Combinatorial Optimization*. Ed. by Klaus Jansen and Samir Khuller. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 84–95. isbn: 978-3-540-44436-7.

Chechik, Shiri et al. (Nov. 2017). "Secluded Connectivity Problems". In: *Algorithmica* 79.3, pp. 708–741. issn: 1432-0541. doi: 10.1007/s00453-016-0222-z. url: https://doi.org/10.1007/s00453-016-0222-z.

Chekuri, Chandra, Kent Quanrud, and Manuel R. Torres (2020). "Densest Subgraph: Supermodularity, Iterative Peeling, and Flow". In: *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1531–1555. doi: 10.1137/1.9781611977073.64. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611977073.64. url: https://epubs.siam.org/doi/abs/10.1137/1.9781611977073.64.

Chen, Zhiyuan et al. (Nov. 2018). "Machine learning techniques for anti-money laundering (AML) solutions in suspicious transaction detection: a review". In: *Knowl. Inf. Syst.* 57.2, pp. 245–285. issn: 0219-1377. doi: 10.1007/s10115-017-1144-z. url: https://doi.org/10.1007/s10115-017-1144-z.

Dantzig, George B. (1951). "Maximization of a Linear Function of Variables Subject to Linear Inequalities". In: *Activity Analysis of Production and Allocation*. Ed. by T. C. Koopmans. Cowles Commission for Research in Economics, Monograph No. 13. New York: John Wiley & Sons, pp. 339–347.

Dijkstra, Edsger W (1959). "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1, pp. 269–271.

Hertog, Dick den et al. (May 2025). "Optimizing the Path Towards Plastic-Free Oceans". en. In: *Oper. Res.* 73.3, pp. 1165–1183.

Kantorovich, L. V. (1939). *Mathematical Methods in the Organization and Planning of Production*. In Russian. English translation reprinted in Management Science, 1960, 6(4), 366–422. Leningrad.

Karp, Richard M. (1972). "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, pp. 85–103. isbn: 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9. url: https://doi.org/10.1007/978-1-4684-2001-2_9.

Khachiyan, L.G. (1980). "Polynomial algorithms in linear programming". In: *USSR Computational Mathematics and Mathematical Physics* 20.1, pp. 53–72. issn: 0041-5553. doi: https://doi.org/10.1016/0041-5553(80)90061-0. url: https://www.sciencedirect.com/science/article/pii/0041555380900610.

Khanuja, Harmeet Kaur and Dattatraya S. Adane (2014). “Forensic Analysis for Monitoring Database Transactions”. In: *Security in Computing and Communications*. Ed. by Jaime Lloret Mauri et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 201–210. isbn: 978-3-662-44966-0.

Koopmans, T. C., ed. (1951). *Activity Analysis of Production and Allocation*. Cowles Commission Monograph No. 13. New York: John Wiley & Sons.

Kuhn, H W (Mar. 1955). “The Hungarian method for the assignment problem”. en. In: *Nav. Res. Logist. Q.* 2.1-2, pp. 83–97.

Lanciano, Tommaso et al. (Apr. 2024). “A Survey on the Densest Subgraph Problem and its Variants”. In: *ACM Computing Surveys* 56.8, pp. 1–40. issn: 1557-7341. doi: 10.1145/3653298. url: http://dx.doi.org/10.1145/3653298.

Lee, Victor E. et al. (2010). “A Survey of Algorithms for Dense Subgraph Discovery”. In: *Managing and Mining Graph Data*. Ed. by Charu C. Aggarwal and Haixun Wang. Boston, MA: Springer US, pp. 303–336. isbn: 978-1-4419-6045-0. doi: 10.1007/978-1-4419-6045-0_10. url: https://doi.org/10.1007/978-1-4419-6045-0_10.

Li, Ling et al. (Dec. 2024). “Heavy Nodes in a Small Neighborhood: Exact and Peeling Algorithms With Applications”. In: *IEEE Transactions on Knowledge and Data Engineering* PP. doi: 10.1109/TKDE.2024.3515875.

Li, Xiangfeng et al. (Apr. 2020). “FlowScope: Spotting Money Laundering Based on Graphs”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04, pp. 4731–4738. doi: 10.1609/aaai.v34i04.5906. url: https://ojs.aaai.org/index.php/AAAI/article/view/5906.

Miyauchi, Atsushi and Naonori Kakimura (2018). “Finding a Dense Subgraph with Sparse Cut”. In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. CIKM ’18. Torino, Italy: Association for Computing Machinery, pp. 547–556. isbn: 9781450360142. doi: 10.1145/3269206.3271720. url: https://doi.org/10.1145/3269206.3271720.

Rajput, Quratulain et al. (Jan. 2014). “Ontology Based Expert-System for Suspicious Transactions Detection”. In: *Computer and Information Science* 7, pp. 103–103. doi: 10.5539/cis.v7n1p103.

Xu, Zhe et al. (2021). “DESTINE: Dense Subgraph Detection on Multi-Layered Networks”. In: *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. CIKM ’21. Virtual Event, Queensland, Australia: Association for Computing Machinery, pp. 3558–3562. isbn: 9781450384469. doi: 10.1145/3459637.3482083. url: https://doi.org/10.1145/3459637.3482083.

A

Appendix A: Alternative NP-hardness Proofs

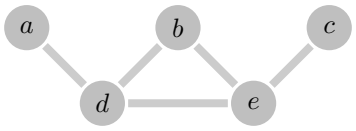


Figure A.1: An undirected graph.

In this appendix, we show how the proofs from Chapter 3 can be augmented in a way where the proof uses 2-layer auxiliary graphs. This shows that the PDRS and NDRS are NP-hard even in the case with two layers. This result is surprising, because it was previously assumed that the 2-layer version could be solvable in polynomial time. The examples in this appendix are based on the Vertex Cover problem in Figure A.1 with a decision variable k.

A.1. 2-layer NP-hardness Proof for PDRS

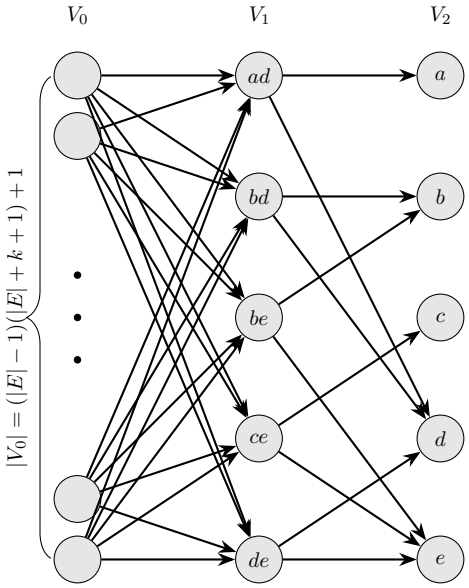


Figure A.2: The original 3-layer auxiliary graph from the proof of Theorem 1. This is the PDRS auxiliary graph based on the Vertex Cover problem in Figure A.1.

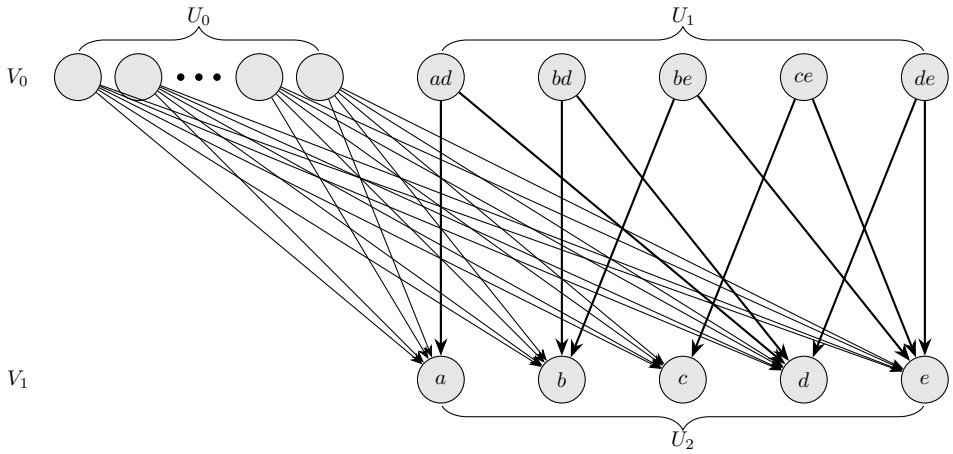


Figure A.3: An example of a 2-layer auxiliary graph that can also be used to prove the NP-hardness of PDRS. The vertex sets U_0, U_1 and U_2 coincide with the layers V_0, V_1 and V_2 of the graph in Figure A.2, respectively.

In this section, we will illustrate how the proof of Theorem 1 could have been done with a 2-layer auxiliary graph. As seen in the proof, given a Vertex Cover instance $\mathcal{G}(V, E)$ with decision variable k, we want to construct an instance of the PDRS problem, $\mathcal{H}(V', E')$, and an objective, α, such that the graph $\mathcal{G}$ has a vertex cover of size $\leq k$ if and only if the PDRS instance has a solution with objective value $\geq \alpha$. In the proof, we show how to construct such an instance. We will now explain how one can construct an equivalent 2-layer instance. We again have three sets of vertices: U_0, U_1 and U_2. These sets correspond to the three layers of the 3-layer instance. However, in the 2-layer version, U_0 and U_1 are both in the first layer and U_2 is in the second layer. The new weight function is then $w(v) = 1$ if $v \in U_1$ and $w(v) = 0$ otherwise. Furthermore, between U_0 and U_2 we have a complete bipartite graph and between U_1 and U_2 we have an arc if the element in U_2 corresponds to an endpoint of the edge that the element in U_1 corresponds to. So the entire construction, weight function and objective α are exactly the same as in the original construction, except for the fact that the 'dummy' nodes U_0 are predecessors of U_2 instead of U_1. However, this does not change the most important aspect, that for each edge $v, w \in E$, x_{vw} can only be part of the selected subset if x_v or x_w are also part of the selected subset. All the original arguments are then the same.

A.2. 2-layer NP-hardness Proof for NDRS

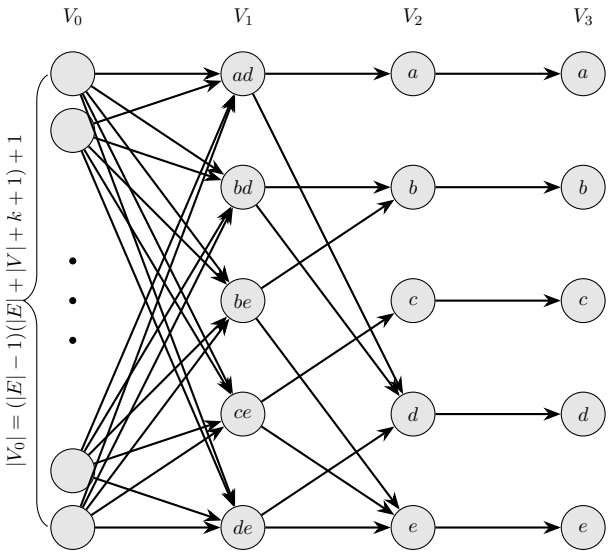


Figure A.4: The original 4-layer auxiliary graph used in the proof of Theorem 2.

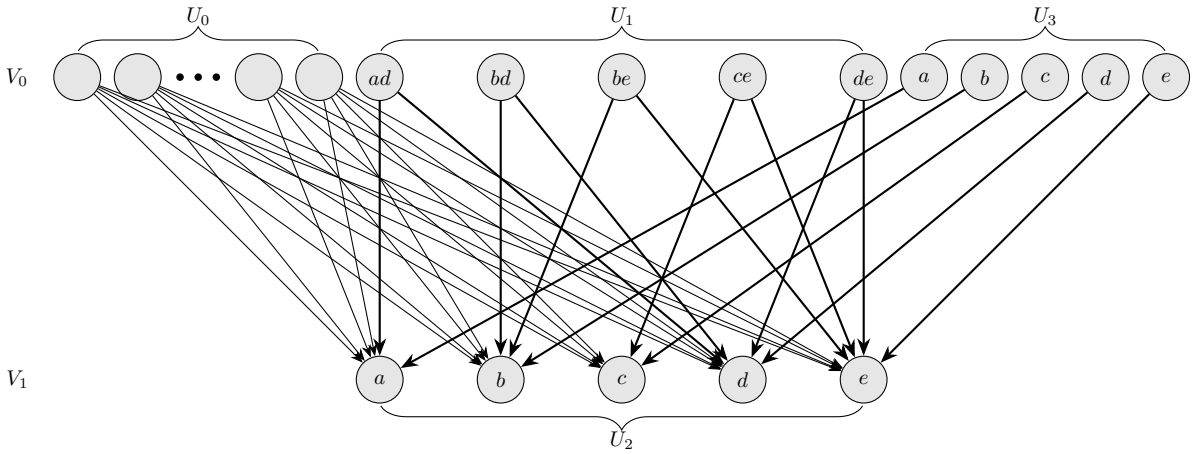


Figure A.5: An example of a 2-layer auxiliary graph that can also be used to prove the NP-hardness of NDRS. The vertex sets U_0, U_1, U_2 and U_3 coincide with the layers V_0, V_1, V_2 and V_3 of the graph in Figure A.4, respectively.

In this section, we will illustrate how the proof of Theorem 2 could have been done with a 2-layer auxiliary graph as well. The new construction is the same as in the previous section, except that we have an extra vertex set U_3 whose elements have zero weight. This vertex set corresponds with the vertices that are in V_3 in the auxiliary graph from the original proof of Theorem 2. If we do the construction from the previous section, we run into the problem of all the vertices in U_2 always being part of the neighborhood of the solution. Thus for the same reason that we add the fourth column in the original proof, we now add the U_3 to the first column. A vertex in U_3 only has an edge to the element in U_2 that corresponds to the same vertex in the Vertex Cover instance. The vertices in U_3 then only have to be part of the neighborhood if the corresponding node in U_2 actually is part of the selected solution. The proof using the 2-layer graph is then equivalent to the original proof.