TU WIEN Informatics

Graph Neural Networks Meet Local Search for the Weighted Total Domination Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Alexander Simunics, Bsc

Matrikelnummer 01309478

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Projektass. Dipl.-Ing. Johannes Varga, BSc BSc

Wien, 21. Oktober 2024

_____   _____
Alexander Simunics   Günther Raidl

TU WIEN Informatics

Graph Neural Networks Meet Local Search for the Weighted Total Domination Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Alexander Simunics, Bsc
Registration Number 01309478

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Projektass. Dipl.-Ing. Johannes Varga, BSc BSc

Vienna, October 21, 2024 _____   _____
                                           Alexander Simunics   Günther Raidl

Erklärung zur Verfassung der Arbeit

Alexander Simunics, Bsc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 21. Oktober 2024

_____
Alexander Simunics

Acknowledgements

Kurzfassung

In den letzten Jahren (oder eher Jahrzehnten) gewinnen neuronale Netze als zentraler Bestandteil vieler moderner Ansätze für maschinelles Lernen immer mehr an Bedeutung, nicht nur in der IT-bezogenen Forschung, sondern auch in Anwendungen aus der realen Welt. Graph Neural Networks (GNNs) sind in der Öffentlichkeit weniger bekannt, werden jedoch nicht nur in vielen modernen Anwendungsfeldern wie Natural Language Processing oder Visual Computing verwendet, sondern können auch auf klassische Graphenprobleme wie das Problem des Handelsreisenden angewandt werden.

In sogenannten End-to-end Learning Ansätzen wird ein neuronales Netz mit dem Ziel trainiert, mit nur minimalen weiteren algorithmischen Hilfestellungen Näherungslösungen für kombinatorische Optimierungsprobleme zu liefern. Trotz ausgefeilter Techniken bleibt die Qualität solcher Lösungen häufig immer noch weit hinter jenen klassischer Optimierungsalgorithmen zurück. In dieser Arbeit beschäftigen wir uns damit, die Ausgabe eines solchen GNNs zu verwenden, um die Performance eines klassischen metaheuristischen Ansatzes, der Adaptive Large Neighborhood Search (ALNS), auf dem eigentlichen Problem zu verbessern. Eine ALNS besteht aus Destroy- und Repair-Methoden, die auf einer temporären Lösung des Problems agieren. Zunächst entwickeln und testen wir traditionelle Methoden. Danach werden Destroy-Methoden entwickelt, die die Ausgabe des GNNs verwenden, und wir vergleichen die beiden Ansätze.

Dies wird anhand des Weighted Total Domination Problem (WTDP) demonstriert, ein klassisches Graphenproblem, das das bekannte (Total) Dominating Set Problem um eine Zielfunktion erweitert, die Gewichte auf Knoten und Kanten des Graphen miteinbezieht. Im Zuge der Arbeit werden für dieses Problem auch Regeln zur Vorbehandlung der Probleminstanzen, ein neuartiger Ansatz für eine "abstimmungsbasierte" ALNS-Methode, sowie die Berechnung von Features zum Trainieren des GNNs vorgestellt.

Unsere Ergebnisse zeigen, dass ein Mix aus traditionellen ALNS Methoden, zusammen mit den vom GNN unterstützten Methoden, im Allgemeinen bessere Lösungen liefern als die traditionellen Methoden alleine.

Abstract

In the last years (or rather decades), neural networks as the core of many modern-day machine learning approaches are becoming more influential, not only in computer science related research, but also in real world applications. Graph Neural Networks (GNNs), while less known in the public, are not only used in popular fields like natural language processing or visual computing, but are also applied to classical graph problems such as the Travelling Salesman Problem (TSP).

In so-called end-to-end learning approaches, a neural network is trained with the goal to find approximate solutions for combinatoric optimization problems with minimal additional algorithmic help. Despite sophisticated techniques, the quality of such solutions often falls far behind those from classical optimization algorithms. In this work, we investigate the possibilities of using the output to improve a metaheuristical approach, the Adaptive Large Neighborhood Search (ALNS). An ALNS consists of destroy- and repair-methods that act on intermediate solutions. We first develop and test traditional methods, and afterwards use a set of destroy-methods that utilize the output of the GNN for each vertex. Both approaches are experimentally compared.

This demonstrated on the Weighted Total Domination Problem (WTDP), an extension of the well-known Dominating Set Problem, which introduces an objective function based on vertex and edge weights. For this problem, we also present preprocessing rules, a novel "voting-based" ALNS method, and custom feature computations for GNN training.

Our results show that the mix of the traditional ALNS methods, together with the GNN-supported methods, are in general performing better than the traditional methods alone.

Contents

CHAPTER 1 ▪

Introduction

1.1 Overview

Graph Neural Networks (GNNs) have been on the rise of attention in recent years (see e.g. Wu et al. [WPC⁺20]), and have been used for a large variety of graph problems. Applications range from problems in microbiology [GDJ⁺21] to social network related problems [NLL⁺21], but also include well-known classical graph problems such as the Travelling Salesman Problem (TSP) [KvHW19], to which many traditional approaches exist. However, most of the time these GNNs are applied directly to a problem, such that the output is either used to directly indicate a solution set (as in [NLL⁺21]), classify nodes or subgraphs, or to sequentially build a solution (as in [KvHW19]). In this work, the focus is not on directly solving the problem with a GNN, but to extend a classical metaheuristical approach with a GNN.

This is done on a relatively new problem, the Weighted Total Domination Problem (WTDP), which was introduced by Ma et al. [MCY19]. It builds upon other well-known Dominating Set problems, and introduces an objective function using both vertex and edge weights. Practical usage is, at the moment, still limited, as Ma et al. did not provide a real-world application for that problem. However, it could have appliances in the domain of social networks, and has also been used related to that context by Kapunac et al. [KKD23].

The problem was chosen, because it is a classical graph problem that, to the best of our knowledge, was only tackled with traditional heuristic and exact methods. Many problems for which GNNs are utilized, are classification or clustering problems that are more similar to neural network classification problems with the addition of the graph aspect, see for example the well-known CORA dataset [SNB⁺08]. The WTDP still provides at least the vertex- and edge-weights as potential additional input features to the graph neural network, as opposed to the more simple variants like the Minimum Total Dominating Set problem.

Based on work on similar problems, we also derive preprocessing methods for the WTDP to simplify problem instances before actually solving them. Unfortunately, on the synthetic and generally quite dense graphs that we tested our approaches on (the instances were taken from Alvarez et al. [AMS21]), these preprocessing rules could not be applied. Nevertheless, they could be more useful on graphs with a larger number of nodes with a low degree.

The metaheuristic we first implement as a baseline, and then try to improve using a GNN, is the Adaptive Large Neighborhood Search (ALNS). The general idea is that on the current incumbent solution, first a "destroy"-method is applied, which modifies a rather large portion of the solution without respecting feasibility or optimality. Afterwards, using a "repair"-method, this is restored to a feasible and locally optimal solution, which may become the new incumbent, if its objective value is good enough.

We first implement traditional destroy- and repair-methods as part of the ALNS. Most of these methods are simple random selection of vertices to be modified, together with local search based repair methods. However, we also experiment with destroy methods based on the graph structure as well as a "voting"-based destroy method, which introduces a different kind of random selection viewing the nodes as "voters" acting based on their local neighborhood.

Afterwards, we implement similar methods, which take a nodewise GNN-output as the random weights, instead of some input feature (such as the direct effect on the objective value, or the weight of the node). The comparison between these two variants of the ALNS – with and without using GNN-outputs – is the main interest of this work. Leaving the rest of the parameters untouched, we aim to detect the impact of these methods.

For the Graph Neural Network, different structures are experimented with. As a baseline, we use a traditional neural network (with no message-passing, i.e. no information from the hidden neurons of neighboring vertices is used). For the actual GNNs, we are using existing implementations of "transformer-like" layers (as used, for example, in [KvHW19]), which combine graph attention networks with feed-forward layers, and use them in two variants.

Also, for training the models we experimented with two different loss functions. On one hand, we use the typical binary crossentropy loss, which is usually used for classification problems in neural network training, with the target values being determined by the best solutions found using the ALNS without GNN-outputs. On the other hand, we also adapt an "energy-based loss function" from Nair et al. [NBG+20] to the WTDP, which estimates the expected cost for taking a node based on a large sample of solutions, instead of taking only the best (known) solutions and taking them as the only ground truth.

The results are compared on the same instances as previously used by Alvarez et al. [AMS21], which are divided into classes based on instance size, density, and weight structure (whether there is more emphasis on vertex- or edge-weights). Although Ma et al. [MCY19] experimented with smaller instance sizes, and Kapunac et al. [KKD23]

extended this set also to larger instances after the start of our experiments, we focus on those instances with a vertex count between 75 and 125.

1.2 Outline

First, we define the WTDP and its related problems, and discuss related work with respect to the WTDP and other similar problems in Chapter 2. In Chapter 3 we present the various methods we applied to the problem, starting from the adapted preprocessing rules, the ALNS-methods, the GNN-based ALNS-methods, and the GNN structures used, together with the necessary details (such as input features and targets). Afterwards, in Chapter 4 we explain in detail how the problem instances were generated, and describe our experiments with several parameters, both for the ALNS runs and the GNN training. Finally, in Chapter 5 we present the final results of the ALNS runs, compare the different methods, and give a short interpretation.

Problem Definition and Related Work

In this chapter, we are first providing the definitions of the Weighted Total Domination Problem and the problems it was derived from, starting with the Dominating Set Problem. Afterwards, an overview of related work is given, containing both the state-of-the-art of the WTDP, and details on recent work (as well as a short theoretical introduction) related to the methods that we use to approach it.

2.1 Notation

Unless specified otherwise (for example for describing related works), we are concerned with simple, undirected graphs with weights on both vertices and edges. An undirected graph $G = (V, E)$ is a collection of vertices V together with the set of edges E where $E \subseteq \{\{u, v\} \mid u, v \in V\}$. If there is an edge between two vertices u and v, i.e. $\{u, v\} \in E$, the vertices are called adjacent or neighbors. A simple graph does not contain self-loops, meaning no edges $\{v, v\}$ of a vertex to itself, and at most one edge between each pair of vertices. As opposed to directed graphs, the edges in undirected graphs are bidirectional.

Within a graph, the distance $d(v, u)$ between two vertices v, u is the length of the shortest path from v to u. The neighborhood $N(v)$ of a vertex v consists of all vertices that are adjacent to it, i.e. $N(v) := \{u \mid \{u, v\} \in E\}$. Note that the vertex itself is not included in its own neighborhood by this definition (in a simple graph). The degree $deg(v)$ of a vertex is defined as the number of incident edges, which in the simple case is equal to the number of neighbors $|N(v)|$. Let us also denote the k-hop-neighborhood by $N_k(v)$, which includes all vertices u that can be reached in at most k steps from v, i.e. the vertices with $d(v, u) \leq k$. This means, that $N_0(v) = \{v\}$ only contains the vertex itself, and $N_1(v) = \{v\} \cup N(v)$, which is also sometimes denoted as the "closed neighborhood"

of v (for example in [KKD23]). For convenience, we can also expand the notion of the neighborhood for a set of vertices: $N(S) := \bigcup_{v \in S} N(v)$, for $S \subseteq V$.

We can define a weight-function on the vertices and the edges to obtain a *weighted* graph. Such functions are of the form $w : V \to \mathbb{R}^+$ or $c : E \to \mathbb{R}^+$ respectively, and the shorthand notations $w_v := w(v)$ and $c_e := c(e)$ or $c_{u,v} := c(\{u,v\})$ for $v \in V$ and $e = \{u,v\} \in E$ will be used for more concise definitions. Since the main focus of this work is on graphs with weights on both vertices and edges, we only denote such graphs as *weighted* graphs, although in the literature this term is often used for graphs without edge weights as well. To differentiate between these cases, graphs that have weights on only the vertices or only the edges are referred to as *vertex-weighted* graphs and *edge-weighted* graphs, respectively, in this work. The presence of weights does not have an influence on the property of a (total) dominating set.

In our figures, we usually show the vertex weights directly on the vertex, and the edge weights near the edge. In some cases, to simplify referencing the vertices, we display an identifier like u or v on the vertex itself, and the weight next to the vertex, for example in Section 3.1.

2.2 Problem Definition

Now, we can give the problem definitions, starting with the most basic variant of Dominating Set Problems, then advancing to the actual problem addressed in this work, the Weighted Total Domination Problem (WTDP).

2.2.1 Minimum (Weighted) Dominating Set problem
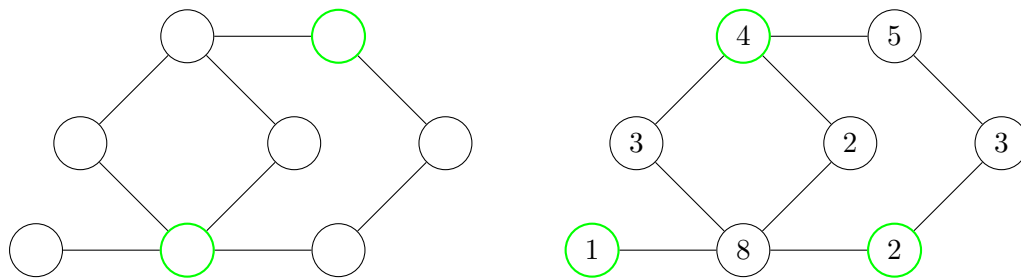
(a) An instance for the MDSP   (b) An instance for the MWDS

Figure 2.1: Two structurally identical graphs (left unweighted, right weighted), with the optimal solution for the respective problem marked in green

Given an undirected simple graph $G = \{V, E\}$, a set $D \subseteq V$ of vertices is called a *dominating set*, if every vertex $v \in V$ either is in D, or has a neighbor u in D, i.e. $N_1(v) \cap D \neq \emptyset$. The most basic problem of the dominating set problem family is the

Minimum Dominating Set Problem (MDSP): Given an undirected, simple graph G, find a minimum cardinality dominating set. A simple generalization is the Minimum Weight Dominating Set (MWDS) problem: For an undirected vertex-weighted graph G with weights w, find a dominating set D that minimizes $\sum_{v \in D} w_v$. Intuitively, each vertex has a (possibly non-uniform) cost associated to selecting it for the dominating set. The MDSP can be obtained from this by setting all vertex weights to 1. Example instances, together with optimal solutions, are given in Figure 2.1.

2.2.2 Minimum (Weighted) Total Dominating Set problem

A dominating set $D \subseteq V$ is called a *total dominating set*, if *every* vertex $v \in V$—including those in D—has a neighbor in D. Therefore, a total dominating set has to fulfill $N(v) \cap D \neq \emptyset, \quad \forall v \in V$. The Minimum Total Dominating Set problem (MTDS) and the Minimum Weight Total Domination problem can be defined exactly as above, but requiring D to be a *total* dominating set. Here, the objective stays the same – minimizing either the cardinality (MTDS) or the sum of the weights – but *every* vertex has to have a neighbor in D, including the vertices selected.

2.2.3 Weighted Total Domination Problem
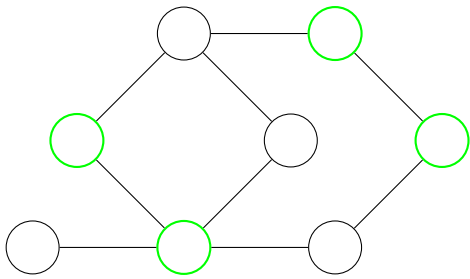
(a) An instance for the MTDS (the solution from 2.2b would also be optimal)
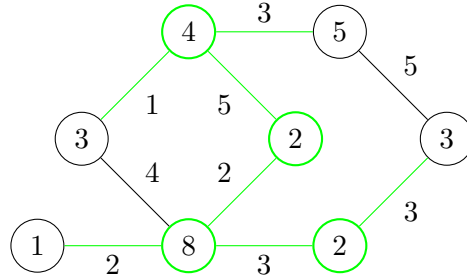
(b) An instance for the WTDP, with possible solution in green (objective value: 35)

Figure 2.2: Examples for the MTDS and the WTDP.

The problem this thesis focuses on is the Weighted Total Domination Problem (WTDP). It was first stated by Ma et al. in 2019 [MCY19]. The WTDP is defined on a weighted graph $G = (V, E)$ with vertex weights w_v and edge weights $c_e = c_{i,j} \geq 0$ for $e = \{i, j\} \in E$. The goal is to find a total dominating set $D \subseteq V$ while minimizing the objective function

$$\text{obj}(D) = \sum_{i \in D} w_i + \sum_{e \in E(D)} c_e + \sum_{i \in V \setminus D} \min_{\{i,j\} \in E \wedge j \in D} c_{i,j} \tag{2.1}$$

where $E(D) := \{\{i, j\} \in E \mid i, j \in D\}$ is the set of edges within the subgraph induced by D.

Here, the objective function consists of three parts, referenced in the literature as follows [AMS21, KKD23]:

- **vertex selection cost**: The vertex weights of each selected vertex $v \in D$.

- **internal edge cost**: The edge weights of every edge $\{i, j\}$ within D, i.e. where $i, j \in D$.

- **external edge cost**: For each **not** selected vertex $v \notin D$, the edge weight of the cheapest edge to a vertex in D.

As [MCY19] pointed out, this is an extension of other well-known domination problems, which can be obtained by setting the edge weights to 0 (yielding the classical, vertex-weighted Minimum Weight Total Domination problem), additionally setting the vertex weights to 1 (yielding the Minimum Total Dominating Set problem), and / or relaxing the constraint to accept ordinary dominating sets instead of total dominating sets. Note that already the TDS problem (in the formulation as decision-problem) without any weights is NP-complete [LPHH84], therefore the WTDP is as well [MCY19, AMS21].

Motivation

The choice of the edge-weights' contribution to the objective function is quite interesting, since a more trivial extension of the previous problems by edge weights could be accomplished without internal edge costs or by using a less complicated external edge cost function. For example, one could include edge weights by just summing the cheapest edge costs into D for *all* vertices. This definition may result in counting the costs of an edge twice, which may not be desirable. However, since the problem was defined this way by Ma et al. in [MCY19] and was already followed up by other researchers in [AMS21, KKD23], we will focus on the same problem.

Ma et al. have not given a particular application or motivation for this problem when introducing it in [MCY19], other than it being an extension to other well-known dominating set problems. However, total dominating problems in general can have applications in communication networks and committee forming [AMS21, Hen04]. A particular example from Henning [Hen04] can be extended by the weights: Suppose a group of people has to form a committee, where every person must know at least one member of the committee, including the committee members themselves. Now, we can model the weights accordingly:

- Set the weights of the vertices to represent the suitability of the person to be a committee member in general (the lower the weight, the better)

- Let the weights of the edges indicate conflicts between people (the higher the weight, the more likely a disagreement would be)

Then, by solving the WTDP, a committee is found consisting of the most competent people, while minimizing the conflicts within the committee as well as between the committee and the rest of the group (as each person would choose their "favourite" committee member as their contact).

Another application is proposed by Kapunac et al in [KKD23]. They model the spread of information within a social network, and show that by solving the WTDP, a better starting set for the spread of information is found than by random selection.

2.3 Existing Approaches to the WTDP

At the time of writing this thesis, there are three publications that we know of that focus on the Weighted Total Domination Problem. We will review their main contributions in the following.

2.3.1 Ma et al. [MCY19]

The first definition of this problem is found in "Integer linear programming models for the weighted total domination problem" by Ma et al. in 2019. They provide three integer linear programs to model the problem, which are step-by-step improvements aiming to achieve better performance on the solver. While the first formulation has a decision variable for each vertex *and* edge, and additionally a helper variable for each edge to determine whether it is about an external edge, the second formulation gets rid of the helper and the final program uses a more sophisticated approach, directly modelling the cost of each vertex using a helper variable for edge related costs instead. The main idea is that every vertex $v \in D$ in the solution contributes its vertex weight as well as *half* the costs of each edge to adjacent vertices, that are in D as well, to the objective function, while every other vertex $u \notin D$ contributes the external edge costs as usual.

These models were evaluated using randomly generated graphs (in the same way as in this work, see Section 4) with up to 100 vertices.

2.3.2 Álvarez and Sinnl [AMS21]

In 2021, Álvarez and Sinnl provided "Exact and heuristic algorithms for the weighted total domination problem", improving the integer linear programs of Ma et al. and additionally introducing a metaheuristic approach, namely a genetic algorithm, for the WTDP.

The improvements to the mixed integer linear programs are on one hand achieved through reformulating it, such that similarities to uncapacitated facility location problem (UFL) become clear, which enables strategies designed for that problem to be adapted to the WTDP (through valid inequalities and resulting optimizations within the branch-and-cut algorithm). On the other hand, the authors have made further progress by implementing a starting heuristic, which is used for an initial solution to the branch-and-cut-algorithm,

as well as a primal heuristic, that assists during the execution. Both their heuristics work on the basis of a local search algorithm: The starting heuristic begins with having all vertices in the dominating set, and greedily removing them according to the score improvement it would bring, until no vertex can be removed without leaving a vertex without a neighbor in D. The primal heuristic starts with an empty solution and adds vertices according to the value of their decision variable in the LP-relaxed version of the problem (vertices with larger values are added first, ties are broken by the degree of the vertex). Vertices are only added if they cover at least one additional vertex that the solution did not cover yet, and the process is stopped once a total dominating set is obtained. A local search trying to improve the score by adding or removing single vertices from the current solution is applied to each of the heuristics afterwards.

The genetic algorithm uses these heuristics as well to some degree. Let us first recapitulate how a genetic algorithm in general works. A genetic algorithm keeps a (usually fixed) number of solutions in memory, called the population. Multiple operators that are inspired by the biological genetics in the real world are defined and then applied to the population: a **crossover** operator that takes two solutions and merges them into one "offspring" solution, a **mutation** operator that alters a solution randomly, and a **selection** operator that decides which solutions are kept in the population are the most common ones.

Álvarez and Sinnl first generate the initial population by using their starting heuristic described before, but modified it to add a random element and not end up with the same solution everytime. In particular, the removal of a vertex is skipped with a probability of 30%. This way the procedure is turned into a GRASP-based (generalized randomized adaptive search procedure) starting heuristic, which is described in detail in [RR16]. The **crossover** operator is simply defined as the union of the two solutions, i.e. all vertices that are selected in either of the solutions, followed by the same GRASP procedure that removes vertices greedily with a random chance to skip a greedy decision, until no improvement was found in one iteration. The **mutation** is done by randomly removing 1-4 vertices from the solution, followed by a local search, and the **selection** is proportional to the objective value, but no two solutions with the same size and the same objective value are allowed to keep the population diverse. This decision may be connected to the small population size of only 40, which enables them to apply the crossover operator over each combination of solutions within the population.

Their evalutation expanded the instance set by larger instances and also including different weight structures. While Ma et al. only generated graphs with vertex weights as well as edge weights between 1 and 5, Álvarez and Sinnl introduced the instance-classes that we use in this work as well, with different weight-ranges for vertices and edges and integer weights up to 50. Also, instances with 75, 100 and 125 vertices were generated. Their MIP-based approaces outperform Ma et al., and the genetic algorithm also found the optimal (or at least best known) solution for almost all of the instances.

2.3.3   Kapunac et al. [KKD23]

The most recent work, published in 2023, completely focuses on the metaheuristic approach and provides a variable neighborhood search (VNS) algorithm, evaluating it on even larger artificial graphs with up to 1000 nodes. They also apply the same algorithm to real-world social network graphs, and compare the performance of the total dominating set to a randomly selected set of the same size with regard to a problem about information spreading in social networks.

One interesting detail is that for their local search, they use a custom fitness function that is defined as $\text{fit}(D) = \text{viol}(D) + \frac{\text{obj}(D)}{W_{\text{tot}}+1}$. Here, $viol(D)$ is the number of vertices that are not covered by any vertex in the solution D, and W_{tot} is the sum of all weights **w** and **c** of the graph. Therefore, this fitness function can even be used for infeasible solutions to compare the number of vertices that are not covered, while the objective value functions as a tiebreaker (and is the only component, if the solution is feasible).

Their local search then uses this fitness function to find better solutions - solutions that cover more vertices if they are infeasible, and solutions with a better objective value in general. This contrasts the local search algorithm by Álvarez and Sinnl, which only operates on already feasible solutions. The second difference is that Kapunac et al. iterate over the vertices in a random order, while Álvarez and Sinnl first try to add vertices not in D, then try to remove vertices from D, but without random permutation. Apart from that, a second, slower local search algorithm is used if no improvement was accomplished for too long. Even then, it is only used on every tenth iteration. This second algorithm, in addition to adding or removing single vertices, also explores solutions obtained by swapping two vertices, i.e. adding a vertex to D while removing another one at the same time.

To sum up the approach of the VNS: Given a solution, an improvement is sought by first **shaking** the solution—in this case randomly removing a number of vertices from D, where this number is varied within a range between iterations—and then employing the local search algorithm(s) explained above, to find a new local optimum. The initial solution is obtained randomly in this case.

Similar to our approach, to more efficiently compute the change of the objective value when adding or removing a vertex, information about the external edges of each vertex is stored. The similarities and differences will be discussed in Section 3.1.

Their VNS was directly compared to the genetic algorithm by Álvarez and Sinnl on the same instances they used, as well as on larger instances with 250, 500 and 1000 vertices. As they showed improvement compared to the genetic algorithm, this will be regarded as the state-of-the-art and will be compared to our own solution in Chapter 4.

Additionally, they used the same algorithm in the context of an information spreading problem. In this kind of problem, the task is usually to find a good starting set of vertices within a (social) network, such that information reaches the maximum amount of people (in the minimum amount of time). They solved a WTDP on real-life social

network graphs from the SNAP dataset [LK14], and compared the performance of the solutions (the total dominating sets) to the performance of a random set of vertices of the same size. They describe no change in the number of reached vertices, but a much faster convergence (i.e. the same number is reached with less iterations). Unfortunately, it is not explained which weights are used when using the graphs in the WTDP setting, and whether the weights have any impact in the information spreading model, although the latter seems unlikely since the model is explained in detail in the paper. Also, a comparison to another baseline would have been interesting, namely taking high-degree vertices instead of completely random ones. In conclusion, we acknowledge the search for a practical use-case of the WTDP, but in our opinion, the usefulness in this context (information spreading in social networks) still has to be explored more.

2.4   Relevant Work for Similar Problems

In this section, we discuss some recent work on other problems related to the WTDP, such that the presented techniques can in principle be adapted to be used for the WTDP as well.

2.4.1   Wang et al. on the MWDS [WCY17, WCCY18]

Wang et al. [WCY17] have employed a kind of tabu search to find a Minimum Weight Dominating Set (MWDS) on vertex-weighted graphs. They use configuration checking (CC) to avoid revisiting the same solutions, by setting a flag for neighboring, possibly impacted vertices when the local search makes a move. Wang et al. [WCCY18] later improved this approach by using three values instead of a simple flag, to additionally store information on whether a vertex could still be covered by the dominating set. This was also done on larger datasets than used by Álvarez and Sinnl in [AMS21].

The local search algorithm `FastMWDS` used in [WCCY18], from which some elements were used and adapted to the WTDP in this thesis as well, will now be explained in greater detail. First, the graph is simplified through a procedure called ConstructDS that consists of a set of rules that detect simple patterns. This includes isolated vertices, vertices (or groups of vertices) with only one neighbor who has lower weight than itself, or triangles where two vertices have higher weight and no other neighbors than the third one. As long as such vertices are found, they are removed, and the procedure repeats until no such patterns are present anymore. A more detailed description is given in Section 3.1. Afterwards, an initial feasible solution is formed by randomly selecting vertices from the remaining graph.

The search itself is based on a score function already introduced in [WCY17]. It is defined there as follows (using our notation):

$$score_f(u) = \begin{cases} \frac{1}{w_u} \cdot \sum\limits_{v \in C_1(u)} f_v & \text{for } u \notin D \\ -\frac{1}{w_u} \cdot \sum\limits_{v \in C_2(u)} f_v & \text{for } u \in D \end{cases}$$

where $C_1(u) := N(u) \setminus N(D)$ and $C_2(u) := N(u) \setminus N(D \setminus \{u\})$. The f_v value (in the original paper denoted as $freq[v]$) is an additional value that is stored and updated during the algorithm for each vertex. It intuitively indicates, how often the vertex was uncovered during the execution of the algorithm so far.

For each vertex u that would be added to D, the f_v values of the vertices that would be additionally covered are added up for the score and divided by the cost w_u of adding the vertex to D. Inversely, if the vertex u is already in D, f_v values of the vertices are taken that would become uncovered, should u be removed from D.

In each step of the algorithm, at first two vertices are removed from D according to their score (one greedily with the highest $score_f$, and in [WCCY18] an additional one with the highest $score_f$ from a randomly selected sample). Then, vertices are added until D is a total dominating set again, preferring vertices with a high score. During this phase, after a new vertex u was added to D, f_v is increased by one for all remaining vertices v that are not yet covered. The motivation behind this is that vertices, which are frequently uncovered, should have a higher priority to be covered, as they then have a greater weight in the $score_f$ function of their neighbors.

The tabu mechanism referred to as "Configuration Checking" will not be described in detail here, as such a strategy was not adapted to the WTDP in this thesis. In Section 3.1, we discuss how we adapted the other parts from this paper to the WTDP.

2.4.2 Dijkstra et al. on the MWDS [DGZ22]

On the same problem, Dijkstra et al. formulated probabilistic equations to estimate the objective value under the assumption, that the vertices of the dominating set D are randomly selected. Three theorems are built that assign those probabilities in different ways. Their algorithm to provide a dominating set is then simply to execute those probabilities – to decide for each vertex randomly, with a given probability depending on the theorem, whether to include it in the vertex or not. Since it is about "normal" domination, any non-covered vertices can simply be added to the dominating set as well, and in a local search manner, any unnecessary vertices can be removed as long as the set remains a dominating set.

This work is interesting in the context of this thesis especially due to the simlarity to the "energy-based loss function" described in Section 3.3. This shows that the goal is similar, that we aim to produce a sort of "probability" or score with a GNN, instead of with manually crafted equations based on assumptions on the problems. However, this

would not be possible for the WTDP or at least much more complicated, since the edge weights make up a crucial part of the objective function in this case.

2.4.3   Bohan Li et al. on the MWCDS [LWWC21]

The Minimum Weighted Connected Dominating Set Problem (MWCDS) is defined similarly to the MWDS, however, the solution dominating set D is required to be connected, i.e. every vertex in D has to be reachable by every other vertex in D using only edges within D. Li et al. adapted the local search of Wang et al. [WCCY18] to this problem in 2021, by imposing additional constraints when choosing vertices to add or remove (to keep D connected), and stating a search algorithm that makes use of the connectedness. Also, a restart mechanism was added, that is triggered after a configurable number of non-improving iterations. This restart mechanism does not only restart the search from a newly generated initial solution, but also smoothes down the f_v values to prevent that single vertices accumulate too much f_v over time (e.g. when stuck in a specific local optimum) and distort the scores.

2.4.4   Ruizhi Li et al. on the MWCDS [LWL$^+$21]

The same problem was tackled earlier in 2021 by Ruizhi Li et al., who did not use the frequency based score function, but instead used the weights of the uncovered vertices in the same place. Also, a restart mechanism was already used in this paper, which started the search from a new random initial solution after a certain amount of iterations. An interesting mechanism was used to select the vertices during the local search: with a certain probability, the vertices were chosen greedily according to the scoring function, otherwise the chosen vertex was determined randomly. In their evaluations, a probability of 60-80% of greedy selection yielded the best results.

2.4.5   Hu et al. on the MTDS [HLW$^+$21]

A simililar approach (local search including a tabu mechanism) was used by Hu et al. on a **total** dominating set problem without any weights, i.e. the problem to find a minimum cardinality total dominating set. They also keep track of how many iterations a vertex has been uncovered, i.e. has not had a neighbor in D, similar to the f_v value from Wang et al. in [WCY17, WCCY18]. However, since there are no weights on the vertices, this is the only factor in this case that determines the vertices to be chosen next.

Methodology

In this chapter, we will describe our approach in detail. First, the preprocessing rules presented by Wang et al. in [WCCY18] are adapted to the WTDP, i.e., the edge weights and their role in the objective function is taken into consideration. We will also discuss the data structure that we use, and the methods at the core that are used within our different Large Neighborhood Search (LNS) operators to add or remove vertices, and to compute the score-difference that such an action would bring, since this is used commonly within our operators.

Afterwards, the LNS will be described. After a short introduction about (Adaptive) Large Neighborhood Search in general, we present the different **destroy** and **repair** operators that were implemented.

Then, we summarize the functionality of Graph Neural Networks (GNNs), elaborate on our feature-selection and training process and present an alternative loss function, which is an "energy-based loss function" based on Nair et al. [NBG$^+$20]. Finally, we bring those two approaches together and present LNS-operators that are based on the results that the GNN delivers.

3.1 Preprocessing and Data Structure

The preprocessing is based on the preprocessing for the MWDS proposed in [WCCY18], although the different problem definition has some implications.

3.1.1 Preprocessing

In the original paper [WCCY18], there were four reduction rules used to simplify the instance through preprocessing (while the rest of the construction of the initial solution used the same heuristic as during the local improvement). In short these are:

- Weighted-Degree-0 Rule: isolated vertices have to be included in D

- Weighted-Degree-1 Rule-1: If there is a vertex u with only one neighbor v, who has less weight than itself, this neighbor v is fixed to be in D and u is removed from G

- Weighted-Degree-1 Rule-2: If there are multiple vertices u_i with degree one and the only common neighbor v, then v can even be fixed if its weight is only smaller than the sum of the weights of u_i

- Weighted-Degree-2 Rule: If there are u_1 and u_2 with their only neighbors being each other and a common neighbor v, and both have larger weight than v, then v can be fixed in D and u_1 and u_2 be removed from G

For the total dominating set problem (hence also for WTDP), instances with an isolated node are not solvable. Therefore, the Weighted-Degree-0 Rule has no counterpart for the WTDP. In this work, we will only consider problem instances without any isolated nodes. We verified that the instances of Álvarez et al. [AMS21] and Kapunac et al. [KKD23], which we will also use to compare our solution against their approaches, are all connected.
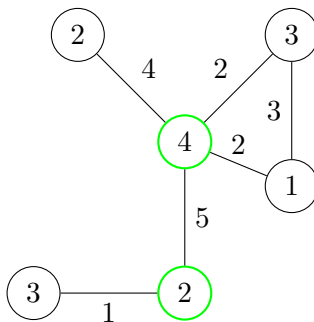

Figure 3.1: Example instance: The green nodes must be included independently of any weights

The two Weighted-Degree-1 rules from [WCCY18] are actually even simpler for WTDP: Since for total domination, *every* vertex has to have a neighbor in D, for every vertex u with only one neighbor v, we have to put v in our solution D – independently from any weights. An example can be seen in Figure 3.1. However, we cannot just remove u from the graph, since it may still be optimal to be selected to provide v with a neighbor in D.

To address this problem, one part is to look at the situation of Weighted-Degree-1 Rule-2: If there are multiple degree-1 vertices $u_i \in N(v), i = 1, ..., k$, we only need to keep the cheapest one, i.e., u_j, s.t. $w_{u_j} = \min\{w_{u_i} \mid N(u_i) = \{v\}\}$. The edge costs are irrelevant here, since they are included anyway in the objective function, regardless of whether u_i is selected or not: Either as the cheapest (because only) edge from the u_is into D, or as an edge inside D. And clearly a vertex u_i that costs more than another, when both can only serve the same one purpose (fulfilling the total domination constraint for v),

cannot be part of an optimal dominating set D. Here, the edge costs of deleted vertices can simply be added to the vertex costs of v, to retain the correct objective value.

Even for the cheapest (or only) one of such vertices, $u \in V$ with $N(u) = \{v\}$, we can still try to remove it. We can generalize the approach from above: If there is another vertex $u^* \in N(v)$, regardless of its degree, that will always be cheaper to include than u (independent of the rest of the solution), then we can safely delete u, handling its edge cost as above. To check this, we can compute the upper bound of the cost of such a u^* being in D by summing the weight of all edges incident to u^*, as well as its own weight w_{u^*}, because in the worst case all other neighbors of u^* have to be in D as well. If there is a $u^* \in N(v)$ such that $w_{u^*} + \sum_{e:u^* \in e} c_e \leq w_u$, then u can be removed. Note that this also covers the case described above, when u^* has degree one as well. One might be tempted to apply this approach in general – but remember that this simplification can only be applied, because v is a fixed member of the dominating set D, and that the only reason to select u would be to fulfill the total dominating set constraint – both of which is not the case if u has more than one neighbor.
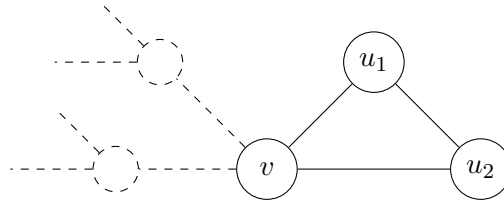

Figure 3.2: Base "triangle" constellation to check for degree 2 nodes

The Weighted-Degree-2 Rule for the MWDS problem looks at triangles $\{v, u_1, u_2\}$ with only one connecting vertex v to the rest of the graph. Here, it is harder to justify fixing v in D, since u_1 and u_2 can be perfectly valid choices to be in D. However, we can make a similar argument as before: If selecting u_1 and u_2 is never viable due to the weights, we can fix v, since even if one of u_1 and u_2 will be selected, it needs a neighbor in D, which has to be v if taking u_1 and u_2 was ruled out. Also note that selecting all three vertices is never optimal, since once v is selected, the cost of one edge and one vertex can be omitted by removing either u_1 or u_2 from D without impacting the covered-ness of any other node. Following those two observations, we are left with four possible options to cover all three nodes, which we can analyze and compare their costs to possibly rule some out:

1. u_1 and u_2 are selected – the cost is $w_{u_1} + w_{u_2} + c_{u_1,u_2} + x_D$, where x_D is the cost of the cheapest edge from v into D

2. Only v is selected from the three and it has to have another neighbor in D – the cost would be $w_v + c_{v,u_1} + c_{v,u_2} + y_D$

17

3.   a) v and u_1 are selected – the cost is $w_v + w_{u_2} + c_{v,u_2} + \min\{c_{v,u_1}, c_{u_2,u_1}\} + y_D$

       b) v and u_2 are selected – the cost is $w_v + w_{u_1} + c_{v,u_1} + \min\{c_{v,u_2}, c_{u_1,u_2}\} + y_D$

Where y_D denotes the "effective" costs from other edges (not to u_1 or u_2), that are directly or indirectly caused by selecting v. Note that

$$y_D \leq \sum_{\substack{u' \in N(v) \cap D \\ u' \notin \{u_1, u_2\}}} c_{v,u'}$$

since costs can only be added from incident edges within D. Edges to other vertices not in D may also contribute to the objective value since it may be their cheapest edge into D. However, if that is the case, selecting v did not cause the costs to increase, but to decrease (or it only made the solution feasible), hence they are not considered in the sum and the term above only constitutes an upper bound.

The first observation is that since the last term y_D is equal for the cases 3(a) and 3(b), and v's state is identical (selected and covered) for both, we can immediately rule the more expensive one out, because the other terms are independent of the candidate solution D. For the sake of simplicity, let us denote the remaining possibility as option 3 with cost $B + y_D$, where

$$B := w_v + \min \left\{ \begin{array}{l} w_{u_1} + c_{v,u_1} + \min\{c_{v,u_2}, c_{u_1,u_2}\}, \\ w_{u_2} + c_{v,u_2} + \min\{c_{v,u_1}, c_{u_2,u_1}\} \end{array} \right\}$$

Furthermore, denote the cost of the first option as $A + x_D$ for $A := w_{u_1} + w_{u_2} + c_{u_1,u_2}$. Note that $B + y_D$ is an upper bound for the costs in the case that v is selected. Now we can compare the worst possible case for when v is selected and the best case for when u_1 and u_2 are. The worst case for when v is selected, would be that every other neighbor of v also has to be in D – then we would have to count every edge incident to D. Therefore

$$y_D \leq \sum_{u' \in N(v) \setminus \{u_1, u_2\}} c_{v,u'}$$
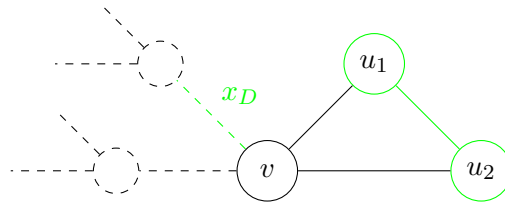
If instead, u_1 and u_2 are both selected, the best case would be that x_D is actually the cost of the cheapest edge incident to v – we get
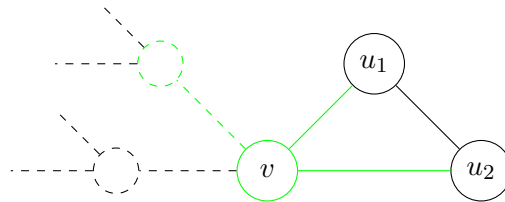
$$x_D \geq \min\{c_{u',v} \mid u' \in N(v)\}$$

With those estimates, we can check whether the following inequality always holds, independently from D:

$$A + x_D \geq A + \min\{c_{u',v} \mid u' \in N(v)\} \overset{?}{>} B + \sum_{u' \in N(v) \setminus \{u_1, u_2\}} c_{v,u'} \geq B + y_D \qquad (3.1)$$
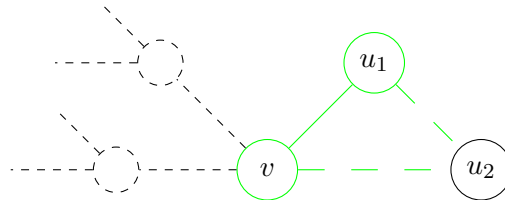
Option 1, where u_1 and u_2 are selected and v is not. x_D, the cheapest edge into D, could of course also be one of the edges to u_1 or u_2.

Option 2, where only v is selected and is covered by some other node.

Option 3, v together with the cheaper one from u_1 and u_2 is selected. Only the cheaper edge from c_{v,u_1} and c_{u_1,u_2} is counted towards the objective function.

Figure 3.3: The three options where all three nodes are covered, with the used edges colored.

If it does, then we can rule out the first option, since its cost will always be higher than the alternative, while covering less vertices than if v and all its neighbors were selected. In that case, we can fix v to be selected, and possibly u_1 or u_2 in case $B \leq w_v + c_{v,u1} + c_{v,u2}$, with the same argument that option 3 covers more vertices and is cheaper than option 2.

Otherwise, it could still be that u_1 and u_2 turn out to be the best choice. But we can nevertheless try to fix the cheaper option from u_1 and u_2, with the same argument as above.

When there are multiple such pairs $\{u_1^i, u_2^i\}$ with $i \in 1, 2, ...k$ for the same v, we can still improve the approach: *not* taking v in the dominating set would require all of them to be in D. So in this case, we can compute A in the inequality (3.1) as $A = \sum_{i=1}^{k} A_i$, where $A_i := w_{u_1^i} + w_{u_2^i} + c_{u_1^i, u_2^i}$. Also, since once v is covered, there would not be any need to select another u (with an exception explained below), we can actually use the minimal B_i, i.e. let B in (3.1) be $B := \min(B_i)$.
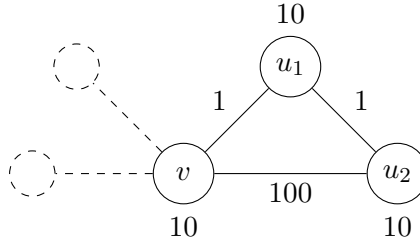

Figure 3.4: Example where the edge $\{v, u_2\}$ can be removed

There is another easy reduction for these triangles, that should even be checked beforehand. Consider the case that $c_{v,u_2} \geq c_{v,u_1} + w_{u_1} + c_{u_1,u_2}$. Then, we can delete the edge $\{v, u_2\}$ and, in consequence, fix u_1 to be in D. This can be seen when looking at the possibilities to cover u_2: if $u_1 \notin D$, then u_2 needs to be covered by v, and therefore the edge $\{v, u_2\}$ has to be included in the costs. But if covering it via u_1 is cheaper because that edge is so expensive, even if both of u_1's edges are included, then selecting u_1 is always better, since additionally v will be covered by u_1. An example is given in Figure 3.4.

After applying these reductions, that fix vertices to be in D and possibly remove vertices or edges, we can take another step: For each fixed vertex v, we can check whether they already have another neighboring fixed vertex. If that is the case, there is great potential for further improvements: First, we can actually remove any vertex that only has v as neighbor, as selecting them would only add their weight and not improve the solution, since v already has a neighbor in D.

More generally, we can also look for vertices u, where $N(u) \setminus \{v\} \subseteq N(v) \setminus \{u\}$ holds. Adding it to D cannot cover additional vertices (since they are all already covered by v), so the only reason to do so would be if it reduces the costs. Also, if adding it to D would not reduce the costs overall, then it never can (since if other neighbors are added as well, it can only become more expensive since there are more D-internal edges) – in which case it can be removed.

Summarizing the reduction rules for WTDP For the following rules, let $m_v := w_v + \sum_{e:v \in e} c_e$, so m_v is the maximal cost that taking v could cause.

- **TDOM-deg-1** For all vertices u with $N(u) = \{v\}$, fix v to be in D.

- **WTDP-deg-1-weights** From all such vertices $u_1, u_2, ..., u_k$ with $N(u_i) = \{v\}, \forall i = 1, ..., k$ for the same v, fix all vertices u_i **not** to be in D, except for u_j, where j is the index of the node with the lowest m_{u_j}. If there is another $u^* \in N(v)$ with $deg(u^*) > 1$ and $m_{u^*} \leq m_{u_j}$, or a neighbor of v that is already fixed in D, u_j can also be fixed **not** to be in D.

- **WTDP-deg-2-expensive-edge** For each triple of vertices $\{v, u_1, u_2\}$, where $N(u_1) = \{v, u_2\}$ and $N(u_2) = \{v, u_1\}$: if $c_{v,u_2} \geq c_{v,u_1} + w_{u_1} + c_{u_1,u_2} = m_{u_1}$, fix u_1 to be in D (and the edge $\{v, u_2\}$ can be removed from the graph).

- **WTDP-deg-2-triangles** For each such triple t as above, compute $A_t := w_{u_1} + w_{u_2} + c_{u_1,u_2}$, and $B_t := w_v + \min_{i=1,2}(w_{u_i} + c_{v,u_i} + \min(c_{v,u_j}, c_{u_i,u_j}))$, with j being the other index ($j = 3 - i$).
  Now, for each v (with $deg(v) > 2$), let T_v be the set of the triples it is part of, and $U_v := \bigcup_{t \in T_v} t \setminus \{v\}$. If

  $$\min_{u' \in N(v)} c_{v,u'} + \sum_{t \in T_v} A_t > \sum_{u' \in N(v) \setminus U_v} c_{v,u'} + \min_{t \in T_v} B_t$$

  then fix v to be in D. Also, if WTDP-deg-2-expensive-edge was already executed, all remaining $u \in U_v$ except for the one with minimal $w_u + c_{u,v}$ can be fixed to be **not** in D.

- **WTDP-already-fixed** If a vertex v is already fixed in D and also has a neighbor that is fixed in D, look for any $u \in N(v)$ with $N(u) \setminus \{v\} \subseteq N(v) \setminus \{u\}$: If adding u to D does not bring an improvement to the objective function, fix u **not** to be in D.

Possibilities for further improvement   After applying these rules, we can remove vertices completely from the graph if their decision is fixed and all neighboring vertices are also fixed. If such a vertex is to be included in D, we just need to make sure that the information about the vertices it covered is stored somehow else, including the information about cheapest edges for its neighbors.

Finally, if removing all vertices that are fixed *and* covered results in the graph being disconnected, we can split the instance up into multiple, smaller instances: For each such connected component, include the vertices of the component as well as the fixed and covered vertices adjacent to them. Note that if we would allow fixed but uncovered vertices, this approach would not be optimal because both subproblems (of both components that this vertex was connecting) would need to find a neighbor of that vertex to cover it, while actually one is sufficient. It is important to keep the information about which vertices are already covered. The advantage of splitting an instance can be quite big, since the two subproblems can not interfere with each other. For example, if there would be one subproblem with k un-fixed vertices, and another with l, instead of having 2^{k+l} possible

solutions, we only need to check $2^k + 2^l$, since finding an optimum for both subinstances would also be optimal for the whole instance. However, it remains to be seen how often such a cut can be made.

It may even be beneficial to try to detect such vertices, that could split up the instance if they become fixed and covered. Artificially doing so, checking the "fixed in D" state and the "fixed not in D" state separately, would still only need $2 \cdot (2^k + 2^l)$ instead of 2^{k+l} possibilities to check (if one would use an exact, naive approach) - provided that it is already covered by another fixed vertex, otherwise this covering vertex would have to be fixed similarly.

However, these approaches are a rather general topic, not restricted to the WTDP, and could be done in general on graph problems where the decision variables only have local influence. Therefore, such preprocessing steps were not performed in the course of this work, and remain open for future investigation.

3.1.2   Data structure

The data structure, especially the helper variables that we use during our computation, was chosen mostly with the following considerations:

- The algorithm (i.e., a **destroy** or **repair** operator) regularly needs to know the effect of adding or removing a single vertex on the objective value.

- Most of the time, single vertices are added or removed one after another (especially for **repair** methods)

- The contribution to the objective value only changes for vertices in $N_2(u)$, if u is added to or removed from D (see explanation later).

- In the case that multiple vertices are added or removed at the same time, the overhead should be limited (i.e. it must be possible to recompute all helpers at once after such a change)

Our data structure for a solution of an instance contains the following information for each vertex u, although for some techniques (e.g., a frequency based scoring function), additional helper variables are stored:

- x_u: The decision variable for vertex u, indicating whether $u \in D$ or not. Note that in the rest of this work, including the algorithms, we use D with set operations instead of this variable for clarity.

- fixed_u: A helper variable that is only set during preprocessing, to mark a decision as fixed with respect to the preprocessing rules explained above.

- $covered_u$: Indicates **c**overedness of a vertex, by storing the number of neighbors that are in D. Therefore, a vertex with $covered_u = 0$ is not covered and violates the total dominating set constraint, while a solution with $covered_u \geq 1$ for all $u \in V$ is valid (feasible).

- δ_u: The amount by which the objective function would change in case variable x_u changes. This can also be set to a special value NaN (Not a Number) to indicate that this value is invalid and has to be recomputed.

- e_u: The cost of the cheapest edge into D, i.e. $\min\limits_{e \in \{\{u,v\}|v \in N(u) \cap D\}} c_e$. If there is no such edge, 0 is stored instead. (Note that all weights have to be strictly positive, so 0 is an indicator that the vertex is not covered).

For any given solution, these values can be easily computed. The computation of δ_u may be done lazily, i.e. only when one of the other algorithms needs the value. The procedure is listed in detail in Algorithm 3.1. If the vertex is currently in D, it would be removed, then the following changes in the objective function:

- The vertex weight w_u is subtracted

- The internal edge costs to other vertices of D are all subtracted, except for the cheapest one

- If the u was part of the cheapest edge of another $v \in N(u)$, the new cheapest edge (if any) has to be found and the difference added.

In the opposite direction, when the vertex is currently not in D, it would be added, and the following changes:

- The vertex weight w_u is added

- The internal edge costs to all other vertices of D are added, while the cost for the cheapest edge e_u is subtracted (so it is not included twice)

- For each neighbor v, if the edge cost $c_{u,v}$ is smaller than the cost of the currently cheapest edge e_v, the difference $e_v - c_{u,v}$ is subtracted

The main methods to add or remove a vertex u from D then are quite straightforward. The possibly precomputed δ_u can be added to the current objective value, δ_u can then be set to $-\delta_u$, and the helpers $covered_v$ and e_v have to be updated for all $v \in N(u)$. Furthermore, δ_v has to be invalidated or recomputed for all $v \in N_2(u)$, as was also pointed out by Álvarez et al in [AMS21]. The reason is, that for all vertices in $N(u)$, the cheapest or second cheapest external edge could be the one to u, which could in both cases affect δ_v of their neighbors. For an example, see Figure 3.5.
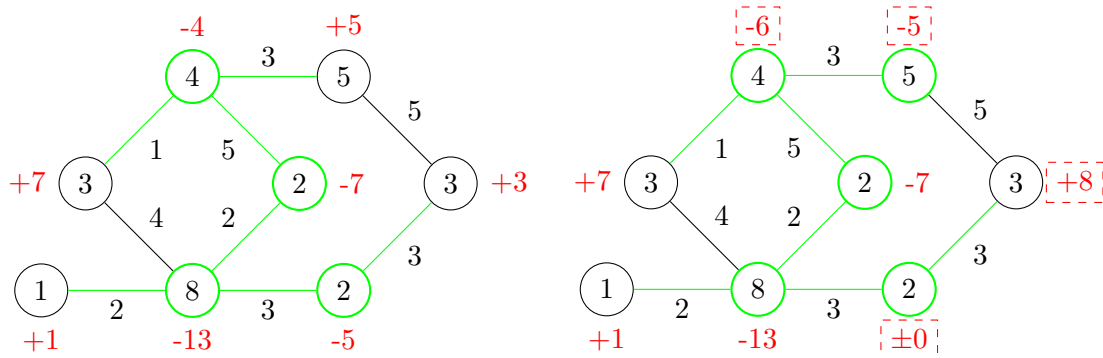
Figure 3.5: Example solutions, with δ_u in red next to each vertex. On the right, the vertex with weight 5 was added to D, and the δ_vs that changed were highlighted. Note that the vertex on the bottom right is affected, although it is not a direct neighbor to the changed vertex.

Note an interesting property of the WTDP compared to other dominating set problems: Adding a vertex to a solution D can in some cases even improve the objective value. This is possible, since the vertex might provide a better external edge to one or more neighboring vertices not in D. For an example, see Figure 3.6. It follows, that an optimal solution does not necessarily have to be as small as possible in the sense, that there can still be vertices that could be removed without violating the total dominating set constraint, but would worsen the objective value as a result of doing so.
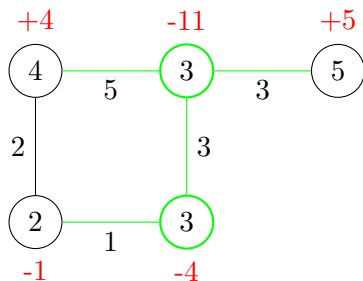

Figure 3.6: Example solution on another instance, with δ_u in red. Note that the improvement in external edge costs would outweigh the cost of taking the vertex on the bottom left, resulting in a negative δ_u.

The main difference to [KKD23] in the data structure seems to be, that we do not store all edges into D for all vertices, but only the cost of the cheapest one. This approach vastly simplifies the adding and removing of vertices compared to Kapunac

et al. However, it leads to a more complicated compute_δ_u procedure and limits the possible precomputation of δ_u, since it is invalidated for a large neighborhood on each action, which could be avoided if a list of external edges is kept. Another difference is the behaviour in case of infeasible solutions - we stick to the definition and do not account for uncovered vertices in the objective function during computation (therefore, an infeasible solution usually has a lower objective value since additional vertices have to be added for the solution to be feasible), while Kapunac et al. extend the objective function such that an infeasible solution always has a worse score than a feasible one.

Algorithm 3.1: compute_δ_u

Input: A WTDP instance $G = (V, E)$ with vertex weights w and edge weights c, current solution $D \subseteq V$, vector e with cost of the cheapest edge into D, a vertex u for which δ_u should be computed.

Output: δ_u, the change in the objective function if x_u would change.

```
 1  if u ∈ D then                                    /* vertex would be removed        */
 2  |    δ_u = -w_u
 3  |    mincost = ∞
 4  |    for v ∈ N(u) do
 5  |    |    if v ∈ D then                           /* remove internal edge            */
 6  |    |    |    δ_u = δ_u - c_{u,v}
 7  |    |    |    mincost = min(mincost, c_{u,v})
 8  |    |    else if c_{u,v} = e_u then
 9  |    |    |    e' = second cheapest edge cost from v into D, 0 if there is none
10  |    |    |    δ_u = δ_u + e' - e_u
11  |    |    end
12  |    end
13  |    if mincost < ∞ then
14  |    |    δ_u = δ_u + mincost
15  |    end
16  else                                              /* vertex would be added           */
17  |    δ_u = w_u - e_u
18  |    for v ∈ N(u) do
19  |    |    if v ∈ D then                           /* add internal edge               */
20  |    |    |    δ_u = δ_u + c_{u,v}
21  |    |    else if c_{u,v} < e_v then               /* better external edge for v       */
22  |    |    |    δ_u = δ_u + c_{u,v} - e_v
23  |    |    else if e_v = 0 then                     /* only external edge for v         */
24  |    |    |    δ_u = δ_u + c_{u,v}
25  |    |    end
26  |    end
27  end
28  return δ_u
```

3.2 Large Neighborhood Search

Shaw introduced the term "Large Neighborhood Search" (LNS) for a family of local search based methods in [Sha98]. We provide a short overview mostly based on the summaries in surveys [AEOP02] and [WNJ+22].

First, we have to understand what "Neighborhood" means in this context. In local search based metaheuristics in general, the solution space is searched by applying changes to the current solution, and then checking the objective value and feasability of the new solution – starting the next step from there, if it is a better solution. By the neighborhood of a solution we mean all other solutions, that can be reached by such a change. These neighborhoods are always problem-specific, since the solution representation can differ. While in a routing problem, a neighboring solution can be defined by a single swap within the permutation of destinations, in our case this can simply be achieved by removing or adding a single vertex to D. The LNS is characterized by the larger amount of solutions that are considered to be neighbors to a given, current solution, accomplished through changes affecting larger parts of the solution that are applied in one step.

In LNS, these changes are split up into two parts: First, a **destroy** operator is applied, which alters the solution by changing decision variables in a random way and/or guided by some heuristic. This does not necessarily pay attention to the problems constraints or objective function, but is only responsible for exploring solutions that are far away from the current solution. Therefore, after this step, the result is expected to have a considerably worse objective value or to be infeasible (so the solution was "destroyed"). The second operator is a **repair** method, that usually uses local search to guide this intermediate solution into a feasible local optimum, much like in other local search based metaheuristics. Here we have to carefully distinguish the neighborhood implicated by the terms "local optimum" and "local search" from the neighborhood in the LNS. The local optimum is not necessarily the best solution from the whole neighborhood as described before, but rather the best solution of its immediate neighbors, found by a local search with a much smaller neighborhood relationship.

3.2.1 Adaptive Large Neighborhood Search

A very common extension of the LNS also used in this work is the Adaptive Large Neighborhood Search (ALNS) introduced in [RP06]. Windras Mara et al. give an overview of recent applications of this framework in [WNJ+22]. In an ALNS, multiple different destroy and repair methods can be defined and used simultaneously. Before each iteration, a pair of one destroy and one repair method to be used is selected randomly. The algorithm usually keeps track of the performance of the individual methods and uses this information to regularly update the probabilities, with which each method is chosen.

Another addition to the usual LNS is that there is a random check, that may accept slightly worse solutions than the current one as the new starting point for the next iteration. A metropolis criterion check is applied, which is computing $e^{\frac{-|\text{obj}(D')-\text{obj}(D)|}{T}}$

and comparing it to a random number in $[0, 1]$, where D is the solution before and D' the solution after applying the methods. If the random number is lower than this expression, the new solution is accepted although being worse. The factor T is a parameter that brings a form of simulated annealing into this method, and is lowered after each iteration.

The procedure is summarized in pseudocode in Algorithm 3.2, which was adapted from [WNJ$^+$22]. The *accept* function in our case just returns $\mathrm{obj}(D') \leq \mathrm{obj}(D)$.

Algorithm 3.2: Standard structure of ALNS, adapted from [WNJ$^+$22]

    Input: A current feasible solution D, destroy operators Ω^-, repair operators Ω^+, initial temperature T_0, cooloff parameter α, probability update frequency η

    Output: The best found solution D^* before stopping criteria were met

1 $T = T_0;\ D^* = D;\ D' = D;\ i = 1$

2   Initialize probabilities p for operator selection

3   **repeat**

4   Select destroy from Ω^-, repair from Ω^+ using p

5 $D' = repair(destroy(D))$

6   **if** $accept(D', D) \vee rand(0,1) < e^{\frac{-|\mathrm{obj}(D') - \mathrm{obj}(D)|}{T}}$ **then**

7 $D = D'$

8   **end**

9   **if** $\mathrm{obj}(D') < \mathrm{obj}(D^*)$ **then**

10 $D^* = D'$

11   **end**

12   **if** $i = \eta$ **then**

13   Update probabilities p of each operator, depending on tracked performance

14 $i = 0$

15   **end**

16 $i = i + 1;\ T = \alpha T$

17   improving the integer linear

18   **until** *stopping criterion met*

19   **return** D^*

3.2.2 Destroy operators

We first introduce a categorization of destroy methods, to make further explanations easier and to be able to focus on the more subtle differences between different operators later. Then, the basic algorithm is provided that is behind most destroy operators that we use. Its behaviour differs by core procedures that make the actual decisions. Lastly, we provide a table with all our destroy methods, as well as a more detailed description for our voting-based destroy algorithm.

Categorization

Our destroy methods will be categorized in three ways: based on whether vertices are added or removed, how the vertices are selected, and whether the method adapts to the intermediate solutions or not.

If we compare this categorization with the overview given in Table 8 in [WNJ+22], we find that they greatly differ. The most apparent difference is that there are no adding destroy methods: as in the original papers for LNS and ALNS, the destroy operators only remove vertices, and the repair operators only add vertices to the solution. One reason for this is, that they mostly analyze Routing and Scheduling problems, where the solutions are most likely permutations of the vertices to be traversed, instead of a subset of all vertices to be chosen – adding vertices does not make sense in this case for a feasible solution. Although an argument could be made that adding vertices is not actually "destroying" the solution, since it remains feasible, it fits well into the framework and turned out to provide a substantial boost to the performance of our ALNS approach.

Adding / Removing vertices For simplicity, all of the destroy-methods either only add vertices, or only remove them from the dominating set. If vertices are removed, the intermediate solution is not feasible, if vertices are added then it is simply not optimal. This also limits the candidate vertices that can be selected, either to the Dominating Set D or to the vertices *not* in D. All of them take a parameter that determines how many vertices should be affected. Besides absolute values, a percentage is also accepted, which is interpreted as the percentage of the *candidate* vertices that should be added or removed. In contrast, the repair methods can do both, as they have to be able to repair any destroyed solution (as will be explained in more detail in Section 3.2.3).

Selection of vertices Most methods employ one of the following principles to select the vertices to add or remove.

- **Random**: The simplest form is to just randomly select the vertices from the candidates (uniformly, i.e. with the same probability for each vertex).

- **Weighted Random**: We can also select the vertices randomly, but with weights associated with each vertex, with the goal that promising vertices are more likely to be chosen.

- **Greedy**: The greedy methods follow some heuristic to estimate the value of a vertex to be chosen, and always select the best one.

- **Structural**: Some methods select the vertices according to structural properties, such as selecting vertices close to each other.

Adaptiveness After a vertex is added or removed, some of the methods adapt to the new intermediate solution – for example, they recompute the estimates for a greedy approach. This comes at the cost of more computation time. Other methods are oblivious to that and only compute the such scores for the initial solution, or do not adapt to the concrete solution at all (like uniformly random approaches).

Basic algorithm

As stated before, most of the approaches share the same basis. Notably, the **Structural** methods are excluded from this because there is more logic involved in choosing the vertices. However, all **Random**, **Weighted Random** and **Greedy** methods can be described with this common algorithm. Differences still emerge from the other categories (e.g., different types of adaptiveness slightly changes how this core has to work), but this will be indicated with conditional expressions. Note that the actual implementation might be optimized in some cases, but produces the same results as the algorithm described here.

At the core of each operator, there is a *score* function. Given the current instance and solution, it returns a data structure that maps each candidate vertex to a score, which in turn guides the selection of vertices: In a **Greedy** operator, the vertex with the highest score will be selected. In a **Weighted Random** algorithm, the higher the score, the higher the chance will be that the vertex is selected, but the selection will be carried out randomly. The score directly relates to the weight that is given to that vertex during random selection, meaning, a vertex with twice the score of another one will also be selected with a probability twice as high. This effectively delegates the implementation of more sophisticated random selection methods (for example, weights based on rank, or cut-off probabilities) to the score function itself. In this case, due to the usage as weights, all used scores must be positive. Finally, the **Random** operators do not compute a score at all, but select each vertex with the same probability. It can also simply be seen as the special case in which $score(v) = 1$ for all $v \in V$. Note that the *score* function has different meanings depending on whether the operator is adding or removing vertices: In an adding method, a higher score should be given to vertices that seem to be promising candidates to be in D, while in removing methods, the same vertices should have low values to avoid being removed.

The general process is as follows: First, determine the candidate set C for the operation – this is either D or $V \setminus D$, depending on whether it removes or adds vertices to D. Naturally, vertices that were fixed during preprocessing are excluded. Afterwards, compute and store the score for each candidate $v \in C$, and repeatedly add or remove a vertex chosen by the given selection method, considering $score(v)$. This is done until the desired number of vertices were added or removed. In case the operator is *adaptive* not only to the initial solution, but to each intermediate solution, the scores have to be recomputed every time D is altered. We provide a pseudocode description in Algorithm 3.3. Note that *adaptive*, *adding* and *selection* as well as *score* are constant "meta"-variables that are pre-determined by the concrete destroy method.

The desired number of vertices to add or remove to D can be given via the input parameter *par* in two ways: either the absolute value of vertices to be altered, or a percentage (given as a number between zero and one), which is interpreted as the fraction of vertices in C that should be affected. Therefore, in a first step the actual count of affected vertices is determined as follows:

$$count = \begin{cases} \lceil |C| \cdot par \rceil & \text{for } par \in (0,1) \\ \min(|C|, par) & \text{for } par \in \mathbb{N}^+ \end{cases}$$

This, together with the determination of C as written above, is hidden in the subroutine *getCandidatesAndCount* (see Algorithm A.1 in the Appendix) for better readability in Algorithm 3.3.

Algorithm 3.3: The basic destroy algorithm

> **Input:** A WTDP instance $inst = (G = (V, E), w, c)$ with vertex weights w and edge weights c, the current solution $sol = (D, h)$ with h holding helper variables as described in Section 3.1.2, $par \in (0, 1) \cup \mathbb{N}^+$ for the desired number of vertices to be altered
> **Output:** Destroyed solution D'

1 $(C, count) = getCandidatesAndCount(V, par, adding)$
2 $scores = score(C, inst, sol)$
3 $sol' = (D' = D, h' = h)$
4 **while** $count > 0$ **do**
5 $\quad$ Select c from C according to selection method and *scores*
6 $\quad$ **if** *adding* **then** $D' = D' \cup \{c\}$
7 $\quad$ **else** $D' = D' \setminus \{c\}$
8 $\quad$ $C = C \setminus \{c\}$
9 $\quad$ **if** *adaptive (not only initial)* **then**
10 $\quad\quad$ Recompute h'
11 $\quad\quad$ $scores = score(C, inst, sol')$
12 $\quad$ **end**
13 $\quad$ $count = count - 1$
14 **end**
15 **return** D'

Voting-based heuristic

As it is probably the most unusual of the proposed destroy methods and, to the best of our knowledge, is a novel method at least in the class of domination problems, we explain the scoring function of the **voting** destroy method in greater detail.

One possibility for selecting the vertices to be removed or added, is viewing the vertices as voters. Each vertex v has its neighbors as candidates for the dominating set, to be covered by it. Their preferences can then be modeled by the cost of selecting the

candidate, i.e., the weight of the edge to it, and possibly the weight of the vertex as well. In case a candidate is already in D, its vertex weight would have to be omitted since it would not be added again to the current objective function. Based on this score, different voting schemes can be used to determine a winner to be added to the dominating set - the simplest would be to just give one vote to the most preferred candidate, but this could be expanded by veto rules or other schemes. Vertex removal could then be based on the same score, to measure "popularity" – possibly enhanced by additional "voting" from the vertices in D. However, in this work we focus on the **adding** variant.

As described so far, the heuristic would be pretty static and probably very similar to just computing the δ_u-scores, since the only thing that changes with the current solution is the score of the vertices newly selected into D – preferring them also in the future. To counter this, we propose to add weight or probability to the votes. One possibility is for each voter to have a probability to skip voting, that increases with each round the current favorite is already in D. This would hopefully have the effect that after some iterations, new vertices are elected into D, where they can actually compete with the previous vertices. A similar effect could be achieved by adding weight to those voters, whose favorite currently isn't in the dominating set. Another approach would be to directly manipulate the score – i.e., by gradually adding cost to vertices already in D when calculating the preferences.

As a simple way to introduce such randomness, we decided to include a mechanism to skip voting, and apply it to each neighbor that would be a better candidate for the voter. By traversing those candidates in a random order for each voter, but making the skip more probable for vertices providing only a small improvement, we hope to get some variation in the votes while still adhering to the modeled preferences. In the current version, the voting destroy method is as follows: For each vertex $u \notin D$, check the vertices $v \in N(u)$ with $c_{u,v} < e_u$ – i.e. its neighbors in D that would provide a cheaper edge into D for that vertex (remember that e_u is the cost of the cheapest edge from u into D). In a random order, check whether v fulfills the condition and whether $\frac{c_{u,v}}{e_u}$ is smaller than a random number $rand \in [0, b]$. Finally, the first such vertex, if any, gets a vote.

The upper bound b of the random number range is a parameter that can be chosen arbitrarily, with larger numbers making a skip less likely. Larger values lead to the randomness laying more emphasis on the random permutation, and less on the improvement the candidate would bring. In our experiments, we found 1.5 to be a good value for b. As an example, suppose a vertex u currently has the cheapest edge into D with weight $e_u = 4$, but has neighbors with edge weights 2 and 3 to it, so $\frac{c_{u,v}}{e_u} = \frac{2}{4}$ resp. $\frac{3}{4}$. This means, the first neighbor has a chance of $\frac{0.5}{1.5} = 33.\overline{3}\%$ to skip the vote and go to the next candidate, while the next neighbor has a chance of $\frac{0.75}{1.5} = 50\%$ to skip the vote. Therefore, if the neighbors are traversed in this order, there would be a $66.\overline{6}\%$ chance for the first candidate to get a vote, a $16.\overline{6}\%$ for the second candidate to get a vote, and also a $16.\overline{6}\%$ chance for no vertex getting the vote from this voter. Considering all possible permutations, the chances would be 50%, $33.\overline{3}\%$ and $16.\overline{6}\%$ respectively. The lower bound of the interval could also be changed, for example, to disable skipping at

all for large improvements (if the bound is raised) or to add a chance of unconditional skipping (if choosing a negative lower bound). In this work, however, we did not change this parameter from 0.

In the end, vertices are added according to who received the most votes. Therefore, in the scheme within the methods derived from the basic destroy function as described before, it can be classified as a **Greedy** method with the *score* function assigning the number of votes to each vertex. Note, however, that it is not greedy in the classical sense since the voting itself contains randomness. Also, in terms of our **Adaptive** classification, it only computes the scores (does the voting) for the initial solution, since on one hand repeating the vote after each selection would somehow contradict the intuition of an election, and on the other hand it would incur much more computation. The runtime complexity for computing the scores alone is $\mathcal{O}(m)$, not accounting for the randomness (shuffling the neighbors of each vertex as well as the random number for the skip-check).

We describe the obtained *score* function in detail in Algorithm 3.4.

Algorithm 3.4: *score* function using the voting heuristic

Input: The candidate set C, a WTDP instance $inst = (G = (V, E), w, c)$ with vertex weights w and edge weights c, the current solution $sol = (D, h)$ with h holding helper variables as described in Section 3.1.2
Output: *votes*, the number of votes for each vertex in C

1 $votes_c = 0$ for all $c \in C$
2 **for** $u \in V$ **do**
3   **for** v *in random permutation of* $N(u) \cap C$ **do**
4        $change = \frac{c_{u,v}}{h.e_u}$
5   **if** $change < 1 \wedge change < 1.5 \cdot rand()$ **then**
6           $votes_v = votes_v + 1$
7   **break for** v
8   **end**
9   **end**
10 **end**
11 **return** *votes*

Destroy methods

In Table 3.1, we summarize all destroy methods not using GNN-scores that were implemented in the course of this thesis.

Frequency-based destroy   Inspired by [WCCY18, WCY17], we also implemented a destroy method based on a similar mechanism as described in Section 2.4.1, using an additional helper variable f_v, storing how often a vertex was not covered during the destroy phase. Then, a similar (greedy) approach to the **cost** operator was applied,

Table 3.1: Summary of destroy methods.

Name	Mode	Selection	Adaptive	Description
random remove	remove	random	no	Randomly remove vertices from D
random add	add	random	no	Randomly add vertices to D
voting	add	greedy	only initial	As described in 15: Each vertex "votes" for one neighbor that is not in D, if it has a cheaper edge than the one it has currently into D. One neighbor is checked after another, and if the conditions are met and a random check is also passed, that neighbor gets a vote. Then, vertices are added according to most votes.
cost	remove	greedy	adaptive	Greedily removes vertex after vertex, each time selecting the one lowering the objective value the most ($score_u = -\delta_u$)
weighted	remove	weighted	no	Randomly removes vertices, with the vertex weights being the weights for random selection. This does not take edge weights or the current solution into account.
pairs	remove	structural (greedy)	adaptive	Tries to select pairs of vertices in D: Vertices that have only each other as neighbors in D are preferred, if there are no such pairs, then the vertex in D with the least neighbors in D is taken. Actually equivalent to a greedy method with $score_u = -covered_u$
region	remove	structural	no	Selects a vertex in D randomly, and using BFS removes it and the nearest neighbors from D until the desired number was removed.

with a *score* function equal to the one from Wang et al., only replacing the weight of the vertex w_u by δ_u, since in the WTDP the cost of adding a vertex to D is not only determined by the weight, but also by internal and external edge costs. However, the method brought did not perform well in comparison to the other, less greedy methods, and was therefore excluded from the table above. It had similar performance to **cost**, which was kept for comparison purposes.

3.2.3 Repair operators

As mentioned before, the repair methods have to be able to restore an infeasible solution to a feasible one, as well as optimize a solution where vertices can still be removed. For all our repair methods, both is done in a two-phase-approach, since there is no real

drawback: If the solution is already feasible, it can quickly be determined and there is no need to restore it. Also, after being restored, the solution may still have to be improved by the optimization step to become an actual local optimum. In most cases, this will be the case if a vertex added during the "restoring" phase can now be removed again, because other vertices (that were added afterwards) now cover its neighbors anyway.

Restoring

To restore the destroyed solution to be feasible again, we use a greedy approach: We add one vertex after another to the solution, each time selecting the one with the least effect on the objective function (i.e., the smallest δ_u), until every vertex has a neighbor in D again. In this selection, vertices that would only cover vertices already covered, are excluded. Afterwards, if any vertices added would lower the objective function even more (negative δ_u although adding), we add them as well.

Optimizing

For the optimizing step, we have two different approaches. One is again greedy, and is similar to the greedy local search in [AMS21] (however, GRASP is not used in our approach). It simply removes the vertex which would most improve the objective value (lowest δ_u), until no vertex can be removed anymore without making the solution infeasible or increasing the objective value.

The other approach is similar, but instead of the greedy local search, the weighted random approach is applied: The vertex to be removed is randomly selected, but weighted by the cost gain, such that vertices with greater immediate impact are more likely to be removed first.

Another, quite costly approach would be an exhaustive search: We can iterate over all possible combinations of removing vertices, while the solution stays feasible. The result is the optimal solution D' of all solutions that are a subset of D. As mentioned, this is computationally not feasible since it basically is a bruteforce search, especially if there are a lot more vertices than needed. Therefore, it was excluded in this work. For future work, however, it would be interesting to adapt the approach to search a smaller space, e.g. by limiting the "depth" k: We can start with a greedy search, and then add the last k removed vertices again. Now, the expectation would be that only around k vertices can be removed without making the solution infeasible again, effectively limiting the search space. However, there is no guarantee that this is the case, so there has to be another mechanism limiting the search, ensuring that the method does not take too long. Other possibilities to improve on these repair methods would be beam-search methods.

3.3   Graph Neural Networks

In this section, a brief overview over Graph Neural Networks (GNNs) is given, followed by examples of recent applications in graph problems similar to the WTDP to some

extent. Afterwards, we explain how we use a (Graph) Neural Network for helping to solve the WTDP. In particular, the input and expected output of the neural network is explained. Then, we describe the different architectures that were implemented, as well as training details. Finally, we present how the output of the GNN is used in the ALNS.

3.3.1 Relevant literature

In this section, we will give a few examples where GNNs were used either for problems similar to the WTDP, or inspired some of the architectural choices in this work. A comprehensive overview of recent applications can be found e.g. in [WPC+20].

At the core of the GNNs we are focusing on is a concept called Message Passing, introduced by Gilmer et al. [GSR+17]. To take account of the structural information of a graph in a neural network, the idea is that for each vertex, the information of all neighboring nodes ("messages") is processed by a trainable neural network, then aggregated (since there can be a variable number of neighbors in a general graph, in contrast to classical convolutional neural networks (CNNs) as used e.g. in visual computing), and finally used to compute the next embedding ("hidden state") of the vertex.

A concrete graph problem that is somewhat similar to the WTDP is the Multi-Hop Influence Maximization problem with the goal to find good starting vertices in a (e.g. social) network to maximize information spread from the selected vertices. Ni et al. [NLL+21] utilized an Attention-based GNN in 2021 for improved results especially on larger graphs for this problem. Parallels to the WTDP can be seen in that the problem is also about selecting nodes to "dominate" the other nodes (although less concise, and the influence is not restricted to the direct neighbors). GNN layers that employ an attention mechanism, so-called graph attention networks (GATs), were introduced by Veličković et al. [VCC+18]. In short, they use an additional matrix of trainable weights that enables the network to learn which of the neighboring vertices are more important and will have greater weight during message passing.

Another example of a GAT presented by Kool et al. in [KvHW19] has largely influenced our network architecture (see Section 3.3.3) and will therefore be explained in greater detail. They implement an "encoder-decoder model" similar to the transformer model presented in [VSP+17]. Within the network, the first layers are used to produce node embeddings, which is the "encoder" part. This is then used as an input to the "decoder" layers to produce a solution, which in this case is a route in the Travelling Salesman Problem (TSP). Their encoder layers consist of two sub-layers: first, a graph convolutional layer using multi-head attention, followed by a node-wise feed-forward layer. The final encoder consists of a single dense layer, to get from the input features to a first node embedding, which are then fed into a chain of the aforementioned encoder layers. An additional "graph embedding" is computed as the average over all node embeddings.

3.3.2 Using Machine Learning for the WTDP

To be able to use the GNN within the ALNS, we first have to specify, how such a GNN can be trained on a given WTDP instance. On one hand, a target of the GNN must be defined, that can be later used by the ALNS. On the other hand, we propose methods of input preprocessing to convert the input instances, i.e. the graphs together with vertex and edge weights, to features that can be used by a neural network as an input.

Target output

The goal in the WTDP is, to choose a dominating set D with a minimal objective value $\text{obj}(D)$. In terms of decision variables, this means we have to decide for each vertex in an instance graph, whether it should be in D or not. The most straightforward target output of the GNN therefore is (in our opinion), to predict for each vertex, whether it will be in an optimal solution D. For each instance, the graph is given as an input, and the target data for training the model is simply the best solution that we have available from the ALNS with classical operators. This is a binary classification task - for each vertex, the GNN has to decide whether the vertex is likely in the optimal solution or not.

There are drawbacks to this approach – for example, for most of the instances we do not know whether the found solution is actually optimal. Furthermore, instances can have multiple different solutions with the same objective value. We could include all known optimal solutions in the training data, however, then there are samples with the same input features, but a different output class – which usually should not be the case. One possibility is to classify a node as "good", if it is part of *any* optimal solution, another would be to include a third class that indicates that the node is in *some* optimal solutions. That would also be interesting from the point of view that these could even be the more important kind of nodes, since the vertices selected in *all* optimal dominating sets could be more "obvious" and therefore already be found by the traditional heuristics. As the most intuitive solution to this problem, and the availability of quite a lot of solutions due to their demand for the energy-based loss function introduced below, we set the target of a node to true if it is part of *any* of the known solutions with the best objective value.

It was considered to include the edges in the target as well – i.e., to train a neural network to identify, which *edges* are likely to be used in the optimal solution (as an internal or external edge), and use this information as well within the ALNS. However, since the decisions are still done regarding the vertices, we found no trivial way to include such information in destroy or repair methods without aggregating them first in some way. Since such aggregation can still be done within the neural network, we decided to do that for network architectures supporting edge features, and refrain from implementing custom aggregation methods on such edge outputs as well as additional, more complex ALNS methods, in this work.

Energy-based loss function

In the setup described so far, the output is mapped to be between $[0, 1]$ using a sigmoid activation layer, and as a loss function, binary crossentropy loss (also called logarithmic loss) is used, as is usual for binary classification problems. Nair et al. proposed a different loss function in [NBG+20], that overcomes some of the issues of computing scores for each vertex based on a single best known solution. The core idea is to first use the objective function to define a probability distribution, that favors better solutions over worse ones, and to then train a neural network to approximate this distribution using a sample of known solutions.

First, we define an energy function similar to equation 8 in [NBG+20], with the only difference being that the objective function of the WTDP is used.

$$E(D; inst) = \begin{cases} \mathrm{obj}(D) & \text{if } D \text{ is feasible} \\ \infty & \text{otherwise} \end{cases}$$

The probability distribution can then be defined in the same way as in the original paper, using our notation:

$$p(D|inst) = \frac{\exp(-E(D; inst))}{Z(inst)}$$

Where $Z(inst) = \sum_D \exp(-E(D; inst))$ normalizes the distribution to a sum of 1 over all possible dominating sets D. This is the desirable distribution that our network will then aim to approximate, with higher probabilities attributed to better solutions.

As explained by Nair et al., training is then done using a sample of solutions (not necessarily optimal ones), that were collected using our ALNS algorithm with traditional destroy and repair methods. Let us describe the loss function with regard to a single instance I, where D_I are the sampled, feasible solutions for that instance and $p_\theta(D = S|I)$ is the probability, that solution S is produced under the distribution approximated with model parameters θ.

$$L_I(\theta) = -\sum_{S \in D_I} w_{S,I} \log p_\theta(D = S|I)$$

The core part, $w_{S,I}$, provides weights for the logloss function that results in preferring better solutions. Since solutions with a better objective value will improve the loss more, a higher probabilty for those solutions results in a better loss improvement than for worse solutions. Normalization is added to avoid giving more weight to instances with better objective values in general, when the loss is finally computed over mutliple instances.

$$w_{S,I} = \frac{\exp(-E(S; I))}{\sum\limits_{S' \in D_I} \exp(-E(S'; I))}$$

37

With the actual loss used being $L(\theta) = \sum_{I \in J} L_I(\theta)$, summed over all sampled instances J in the current batch, we arrive at the same loss function as presented in [NBG$^+$20] in equations 11 and 12, where we refer the reader for a more detailed discussion. The model to compute $p_\theta(D = S|I)$ from the output for each vertex is also taken from there, multiplying the individual probabilities for taking each vertex as if the choices were independent from each other, and using Bernoulli distribution for each such decision.

Note that the data collection for the training data only has to be done once per instance, and can be done "along the way" when running the ALNS for obtaining a best known solution for the classical binary crossentropy loss.

Input preprocessing

The question, which input features should be used, is not trivial for this problem. The weight and the degree of the node are naturally included. However, the most pressing question is how to include the edge weights, as for the simplest approaches, the network architecture does not necessarily support edge features, but the edge weights are a substantial part of the problem. To tackle this problem, for each node, we precompute the minimum, maximum, mean and median of the weights of the incident edges. Also, to provide these measures also in a normalized form, we compute the same measures for the edge weights rescaled to [0, 1] using the minimum and maximum of the *other* node (simply by subtracting the minimum, then dividing by the span) – see the description of Figure 3.7 for an example. This is also intuitively useful, since for this problem, the cost of an edge is especially relevant in relation to the other edges incident to that same vertex (since at least for vertices *not* in D, we want to find the cheapest such edge into D).
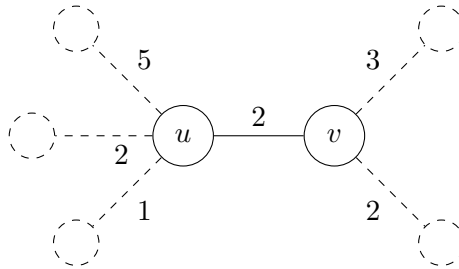
Figure 3.7: Normalisation for weights: for the left node u, the weight of the center edge is scaled to 0 since it is one of the cheapest edges incident to v, while for the right node v, it is $\frac{2-1}{5-1} = 0.25$.

The pre-computation of measures to be used as input features was also used, for example, by Almasan et al. [ASVR$^+$22], who trained a reinforcement learning agent to determine the best action on edges in a network based on a routing optimization problem, and computed the "betweenness" of a link as an input feature. Other approaches to include the edge weights would be to use or develop graph neural network layers, that simply accept the weight as an additional input during message passing. For example, Liyu Gong et al. proposed a framework to use edge features even for problems that do not have

edge features initially, to enhance the performance of GNNs [GC19]. These improvements could also be done additionally to the existing solution of aggregating the incident edges per node.

An approach that is left for future work is to include the current solution of the LNS as an input, and try to let the GNN decide not only which vertices would be good to add in general, but specifically how to possibly improve the given solution. Although it may provide better, more precise output with regard to the solution, it would also require to run a forward-pass of the network not only once, but in each iteration of the LNS. Also, in a classical supervised training setting, the training data would be much more ambiguous and hard to justify, since the problem of multiple solutions being correct applies even more when we are only looking for operations that lead to better solutions than the current one, instead of looking for vertices in one of the few optimal solutions overall. Therefore, we propose that a reinforcement learning setting would be much more fitting for this kind of task, but leave this question for future research.

3.3.3   GNN Architecture

We propose multiple architectures with slight differences that are mainly inspired by Kool et al. [KvHW19]. The basic structure is the same for all of them:

- First, a single dense layer, to "upscale" the input features to initial node embeddings to be used in the following layers.

- Then, we experiment with a number of different kinds of graph convolutional layers (GCN). In one network, we use a chain of only one kind of layer here. We compare the performance in preliminary experimentation, as is explained in detail in Section 4.2, and only use the best architectures in further experiments with the ALNS. The layers experimented with contain:

  - Classical graph convolutional layers (GCN)
  - Graph convolutional layers with attention (GAT)
  - Transformer-like layers, combining GAT-layers with feed-forward layers in between, like described in Section 3.3.1
  - Alternating layers training node embeddings with ones that train edge embeddings, as described below.
  - No convolutional layer at all as a baseline (i.e., the network has no message passing and is therefore not actually a GNN)

- In the place of the "decoder" in the work of Kool et al., since we do not have to generate a sequence but just want to output a single value per node, we simply use a chain of dense layers with ReLu activation, using a sigmoid activation to get values between 0 and 1 in the last layer.

39

Dropout layers are used after the final dense layers (with the dropout rate being a tuned parameter, but in general around 0.5), as well as skip-connections and batch-normalization (for the convolutional layers).

(a) Structure "plain" (without any GNN layers)

(b) Structure "trans"

(c) Structure "edges"
$\mathbf{h}_u$ are the node embeddings of another node u.
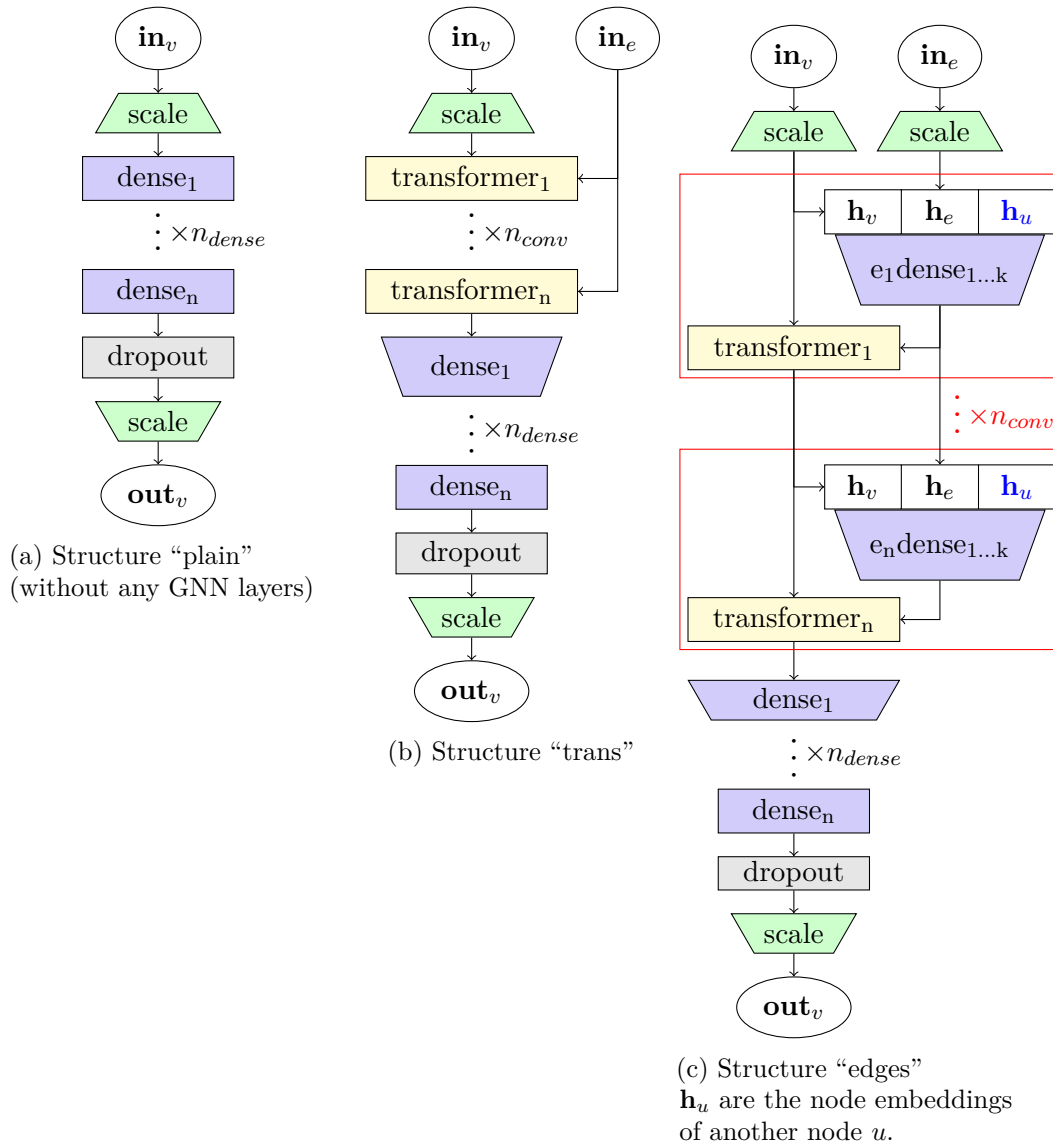
Figure 3.8: Overview of the three neural network structures which were evaluated

Edge embeddings In the WTDP, the edge weights are an important part of the problem, so only considering them as part of the input features precomputed per vertex could leave some of the potential of GNNs. Therefore, we additionally experiment with two variants of edge features to be used in the graph convolutional layers (that support it):

- The (normalized) weights of the edges

- Edge embeddings that are trained alongside the node embeddings

Training edge embeddings alongside the node embeddings is inspired by Jiang et al. [JJL19, JZLJ20]. We use a simple approach, where we first compute initial edge embeddings in a similar manner as for the node embeddings, using the input features of both endpoints of the vertex as the input for a dense layer. Then, before each "transformer" layer that supports concatenating edge features, we use another dense layer, with the input being the current edge embedding as well as the two node embeddings of the incident vertices, to receive the new edge embedding (that is used in the next step to compute the next node embeddings).

Summary of structures   Finally, after some preliminary experiments, three structures were chosen to be evaluated and compared in more detail. They are shown in Figure 3.8. As a baseline and evaluation of our input processing method, the "plain" structure does not use a graph convolutional layer and can therefore not really be classified as a GNN (as the graph structure is not considered at all during training). For the other two structures, we decided to use a transformer-like layer as the convolutional layers, using the two variants of edge features respectively. The used parameters will be explained in more detail in Section 4.2.

3.3.4   Combining the GNN with the ALNS

Since the goal of the work is to improve the existing ALNS approach using the results from a GNN, we introduce LNS operators with the same basis as presented in Section 3.2, but using the output of the GNN. The output scores from the GNNs are obtained once per instance and model before the ALNS run, and then used as an additional input.

It would also be possible to include the GNN output to introduce bias towards more promising vertices during the restoring or optimizing phases, however, we decided to focus on the destroy methods. One reason for this is that repair methods have to "react" to the intermediate solution to be able to find a local optimum, for example in a greedy local search, while the GNN will only return scores for vertices that are considered to be generally good.

GNN-based destroy methods

Using the same basic destroy algorithm outlined in Algorithm 3.3, we can very easily use the scores obtained by the GNN as the *score* function within our destroy operators. An overview is given in Table 3.2. For adding methods, the output itself can be used. However, for the removing destroy methods, we need to invert the scores to put a larger weight on vertices with a lower score.

41

Table 3.2: Summary of destroy methods based on the GNN scores

Name	Mode	Selection	Adaptive	Description
gnnremove (keep)	remove	weighted	no	Randomly select vertices in D to be kept, with the GNN scores out_v being the weights for random selection for vertex v.
gnnremove (inv)	remove	weighted	no	Randomly remove vertices, with out_v^{-1} being the weight for random selection.
gnnadd	add	weighted	no	Randomly add vertices, with out_v being the weights for random selection.

Inverting the GNN-scores for removal   We propose three different variants to accomplish the inversion of the output of the GNN for **removing** destroy methods. First, we can simply use out_v^{-1}. This works for all scores and preserves the proportionality of the scores, meaning that a vertex with half the score will have double the probability to be removed. Another possibility, since the output of our GNN is already mapped to $[0, 1]$, we can use $1 - out_v$ as the weights, interpreting the output as percentages. However, this does not have the same property, and can lead to very similar probabilities for vertices that have low, but still very different values. For example, consider two outputs from the GNN being $out_{v_1} = 0.01$ and $out_{v_2} = 0.1$. With this conversion, the weights for random selection would be 0.99 and 0.9, respectively – resulting in v_1 only having a probability of $\frac{0.99}{0.99+0.9} \approx 52.4\%$ to be removed instead of v_2, altough having a score ten times lower. This effect also gains importance due to the fact, that for our experiments, the average output for the scores was usually very small. Our third way to use the GNN output for removal of vertices from D is actually reversing the selection of vertices: instead of choosing the vertices to remove, we select the vertices to keep, based on the original scores.

Preliminary evaluation showed very similar results for the first and third methods, while the second one had little difference to completely random methods with equal weights. We attribute this to the aforementioned low average of scores from the GNN, as well as an additional averaging effect caused by the larger number of removals within one iteration. Therefore, we omit the second method in the further experiments, and will show in chapter 4.3.2 that the other two are similar enough to just keep comparing one of them to the baselines.

Experiments & Evaluation

4.1 Training and Evaluation Data

The training data was obtained by randomly generating new instances in the same way as the testing instances. The baseline approach (ALNS) was applied to these instances, and the best found solution was taken as the target for training the network. Additionally, the intermediate solutions were tracked to be used by the energy-based loss function described in Section 3.3.2 during those runs.

4.1.1 Random graph generation

The graphs used for final evaluation were taken from [AMS21] and [KKD23] as provided. For training and preliminary evaluation of different approaches, we used the same random generation as for the existing instances, the Erdős Rényi model [AMS21, KKD23, MCY19] together with random weight assignments. Given a number of nodes, and a probability p, a graph with that number of nodes and initially no edges is created. Then, each possible edge is added with a probability of p. Therefore, the parameter p affects the density of the graph [AMS21]. Note that we only accept connected graphs, and generated another one if it was not connected. Finally, for each vertex and each edge, the weight is determined uniformly randomly from the ranges $[1, w_{\max}]$ or $[1, c_{\max}]$, where $w_{\max}$ and $c_{\max}$ are parameters for the maximum weight of vertices or edges, respectively.

4.1.2 Parameters of the generated graphs

Directly taken from Alvarez et al. [AMS21], the following parameter values for the random graphs were used:

- Number of vertices n: 75, 100, 126

- Edge probability p: 20%, 50%, 80%

- Weight distribution $W(w_{\max}, c_{\max})$: $(10, 50)$, $(25, 25)$, $(50, 10)$

This amounts to a total of 27 instance classes. For each class, 1000 train + 250 validation graphs were generated for training the GNN. Note that we may use the term "class" in the further sections for the 9 classes which are indifferent regarding instance size, containing for example the class of graphs with 80% edge probability and $(25, 25)$ weight structure. In general, we denote the classes with a term in the form "$<n>$-$<p>$-$<w_{\max}>$-$<c_{\max}>$", where the edge probability p is displayed as a fraction (0.8 instead of 80%) and unfiltered parts are omitted (i.e., 0.8 as a class contains all graphs with edge probability 0.8, regardless of instance size or weight structure).

4.2   GNN Training and Parameterization

The implementation of the GNN was done in Julia using the `Flux.jl` [ISF+18, Inn18] and `GraphNeuralNetworks.jl` [Loc21] libraries, which provided the "dense" and "transformer" layers used in Section 3.3.3. In this section, we discuss how the GNNs were trained, which parameters were experimented with, and how the different GNN structures were evaluated before using their output in the ALNS.

4.2.1   Training

The models were trained on the 1000 generated training instances, with a limit of 300 epochs and an early-stopping criterion if the loss on the 250 validation instances did not improve in the last 20 epochs.

For the optimizer, ADAM [KB14] was used (`Flux.jl` implementation [ISF+18]) with a learning rate of $\eta = 0.0013$, which was determined in manual experiments and experimentation with automated tuning, see below.

Due to the large amount of data, especially for the energy loss function (since here, solutions have to be additionally stored), mini-batching was used for training, batching 50 instances, resulting in 20 updates per epoch. This also leads to faster convergence, however, the weights are never updated on the full dataset.

Still, the energy loss function introduced some other problems: models would often randomly diverge from the beginning of training, and in general the training time (per epoch) significantly increased. Adapting the learning rate manually did not bring a sufficient improvement to justify tuning the models with this method (which includes a lot of training runs). Therefore, the classical binary cross-entropy loss (logloss) was used for all experiments, and we only train an additional model with the energy-based loss function using the tuned parameters, comparing their performance directly in the ALNS. For these models, if the model diverged from the beginning, the process was simply restarted until a converging model was found.

Training the models was done with GPU support (NVIDIA GeForce RTX 3070Ti 8GB). The time to train the models was not measured exactly, but was around 1-3 minutes per model (with pre-computed features and targets for the graphs), mostly depending on the early stopping, the structure used, and the instance class (especially for the "edges" structure, with noticeable longer times on denser graphs).

4.2.2 Parameter tuning

In the beginning, extensive manual experiments were done in order to get a feeling for suitable parameters, and to test the implementation of the different structures and overcome initial problems. Afterwards, we used the hyperparameter optimization package SMAC3 [LEF+22] to try and automatically tune the parameters in question, with a one hour time limit for each structure and class. However, analyzing the resulting configurations, it became clear that some of those runs were clearly not converged, and provided suboptimal configurations for some of the classes. We attribute this to the long running time for each model in relation to the time limit, the small effect the parameters have on the target (here the PRG-AUC, see Section 4.2.3), and possibly too largely chosen search-spaces for certain parameters.

To provide a solid configuration for each model (in each class and structure), we adapted our approach to scanning a very limited search-space, using one baseline configuration which then is only adapted in a single parameter per experiment. The baseline configuration was determined from the better-performing models from SMAC and manual experiments. Some of the parameters were fixed from this data, like the learning-rate $\eta = 0.0013$ (from SMAC) and the batch size 50 (from manual experiments).

All experiments were done separately for each of the 9 instance classes, and each structure, to observe potentially different effects of the parameters under these circumstances.

Tested parameter values

The parameter values tested are presented in Table 4.1, with the middle one being the value in the baseline configuration, and one tested variant each for increasing and decreasing the value. In total, there were 10 configurations other than the baseline tested this way, two for each parameter.

Tested additional features

The baseline input features used were the weight, the degree, and the average normalized edge weight (as described in Section 3.3.2), all normalized. These were determined during preliminary experiments. However, experiments were done by adding additional input features. The following variants of the edge values were experimented with:

- **r_min**: the minimum normalized edge weight

- **r_max**: the maximum normalized edge weight

Table 4.1: Summary of the GNN parameter values tested.

Parameter	dec.	baseline	inc.	Description
nConv	2	3	4	Number of convolutional layers (not used in structure "plain")
nDense	2	3	4	Number of dense layers (*)
nhidden	4	8	16	Number of hidden neurons for the transformer layers (not used in structure "plain")
nh_dense	8	16	32	Number of hidden neurons for the dense layers (*)
dropout	0.3	0.5	0.7	Dropout rate for the dropout layer

* Dense layer parameters were also used for the dense layers to compute the edge embeddings $\mathbf{h}_e$ in structure "edges"

- **max**: the maximum edge weight (not normalized)

The last feature was added due to some promising results where the normalization was not needed, and to compare the results against its normalized counterpart. However, it did not bring any improvement, and was omitted in the result plots in order to keep them more readable.

Additionally, egonet features for 1-hop and 2-hop ($N \in \{1, 2\}$) neighborhoods for each node were available:

- **ego$N$_n**: the number of nodes in the N-hop neighborhood

- **ego$N$_m**: the number of internal edges within the N-hop neighborhood

- **ego$N$_o**: the number of outgoing edges adjacent to the N-hop neighborhood

For the trans and edges structures, the edge cost c_e was always passed as the single edge input feature for $\mathbf{in}_e$.

4.2.3 Precision Recall Gain curves

A central element in our method to compare different models is the little-known Precision-Recall-Gain area-under-curve metric (PRG-AUC). In general, area-under-curve metrics like ROC-AUC (Receiver Operator Characteristic), PR-AUC (Precision-Recall) or PRG-AUC are not dependent on a single threshold (in contrast to Accuracy or F_β-score metrics). Instead, they are defined by the area under a curve obtained by computing a point for each possible threshold. This fits our scenario well, since we are not using the output in a classifier (where one threshold would be needed), but to skew random selection in our ALNS-methods towards the better vertices. ROC-curves are in general considered a suboptimal metric for imbalanced datasets [DG06, JCDLT13, BDA13], especially when compared to PR-curves.

Flach and Kull however make a point that PR-curves have some shortcomings with respect to interpretability and may lead to worse model selection, and propose the Precision-Recall-Gain curve. Here, the precision gain is plotted against the recall gain, which are harmonically scaled equivalents of the precision and recall respectively. For example, $precG = \frac{prec - \pi}{(1-\pi)prec}$ for π being the true ratio of positive samples. [FK15]

The advantages include (also summarized from [FK15]):

- Linear interpolation of the curve can be reasoned

- The area under the curve can be put in relation to the expected F_1 score

- The comparison to the baseline is easier (since in PR-curves, the baseline performance is dependent on π, which is already factored in in PRG-curves - a classifier with baseline F_1 score ends up on the minor diagonal)

One important caveat is, that due to the nature of the curve computation, the magnitude of the outputs as well as the magnitude of the differences between two outputs does not have any influence on these curves. Only the rank of the outputs matter, with respect to their ground truth target value. For example, the following three predictions (where each output is shown underlined if the ground truth is "true") would result in the same PRG-curve:

- $[0.1, \underline{0.2}, 0.5, 0.5, \underline{0.6}]$

- $[0.01, \underline{0.02}, 0.05, 0.05, \underline{0.06}]$

- $[0.01, \underline{0.2}, 0.3, 0.3, \underline{0.99}]$

When using those outputs as weights for random selection (as we do in our ALNS-methods), the first and the second set of predictions result in the same probabilities. However, the last one would heavily favor the last vertex with a weight of 0.99. If the outputs would be directly used as probabilities (e.g. by randomly permutating the candidate vertices and then selecting each vertex v with probability $\mathbf{out}_v$, until the desired number was selected), the second (scaled-down) prediction would also have a different effect.

Under this perspective, using the PRG-AUC values to tune and select models might lead to selecting a model, whose outputs have less practical use in the adapted ALNS-methods. However, the magnitude of the outputs have shown to be similar between models, and additional experiments will be done with transforming the outputs before using them to mitigate this.

4.2.4 Method of comparison

For the final evaluations presented here, we always trained 5 models with the same parameters (to account for outliers) on the 1000 training instances for a single class (with $n = 75$ vertices), and evaluated them on the 750 validation instances of the same class (250 each for $n \in \{75, 100, 125\}$). The 250 validation instances for $n = 75$ are the same as were used for early stopping during training, but the models were not trained on them. Note that this means that the model was trained on 75000 nodes each epoch. For each model, we compute the PRG-AUC for all evaluated instances at once (as if it was a single not-connected graph).

To compare different configurations, we initially planned to determine a "median" model based on this PRG-AUC within each configuration, and compare those against each other. However, when inspecting the results we came to the assumption that the random variation between different trained models often had a larger effect than changing the parameter itself, and that 5 runs are too small of a sample size to make up for that effect. For some classes, for example, the comparison would have concluded that every tested parameter change would beat the baseline (no matter which parameter, no matter if it was increased or decreased), which was then explained by the situation that two of the baseline models were keeping up with the other models, but three were slightly worse. See the comparison in Table 4.2 and Figure 4.1.

Configuration	PRG-AUC
baseline	0.98317
nConv=2	0.98356
nConv=4	0.98360
nDense=2	0.98833
nDense=4	0.98740
nhidden=4	0.98382
nhidden=16	0.98673
nh_dense=8	0.95524
nh_dense=32	0.98663
dropout=0.3	0.98793
dropout=0.7	0.98775

Table 4.2: Results according to the previous method using the median model. Values better than the baseline are shown in green.
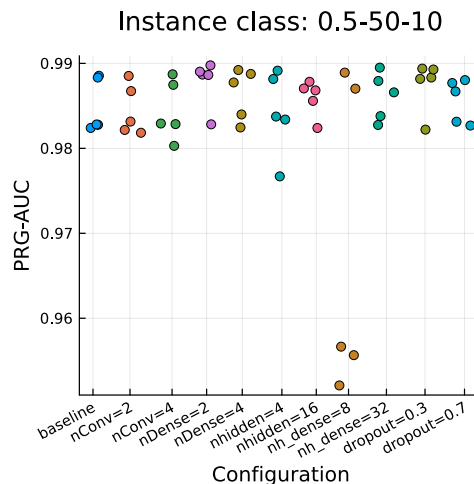
Figure 4.1: Comparison of all models of the same structure ("edges") and class, showing quite similar results

Other measures tested were the average PRG-AUC per instance, and performing a sign-test on the PRG-AUCs per instance. However, these methods still compare two *models* (and not parameter configurations), so they suffer from the same problem. Generating a merged model by using, for each vertex, the maximum of its scores across the five models,

was an attempt to mitigate the problem by generating a single model to be compared. This process introduced some randomness as well, with the model sometimes performing even better than the best model, sometimes much worse, so the problem with randomly worse baseline models would still have applied using this. For these reasons, we compare all models of a parameter configuration against each other in a dot-plot as shown in Figure 4.1, and decide on this basis where improvements seem likely to actually come from the parameter change. Since the differences are quite small in general, it seems reasonable to assume that no large improvements could be done by further parameter experimentation, and that we can select good models from those experiments to be used in the ALNS.

4.2.5 Results of parameter tuning

We see two cases where the parameters seem to make a consistent, although small difference. First, the models of the "plain" structure seem to profit from additional egonet features as well as "r_min", see Figure 4.2. And second, for the "trans" models we can see better results for more convolutional layers (better for nConv=4 and worse for nConv=2), see Figure 4.3.

The effects are more visible on the classes with edge-heavy weight structures, which seem to be more difficult in general, based on these results. For the "edges" structure, no parameter configuration seems to consistently bring a (noticable) advantage, which could mean that the more complex neural network is enough to offset some of the problems that the other structures have. We included figures for all structures and parameters in Appendix B. It also makes sense that the "plain" structure profits from the egonet features, while the other ones, utilizing graph convolutional layers, can more easily induce the role of a vertex in the graph without that information.

Final tuned models

For the models to be used in the ALNS runs, we therefore use:

- **plain**: The baseline parameters with all egonet features and the r_min feature

- **trans**: nConv=4, otherwise baseline parameters

- **edges**: unmodified baseline parameters

With these parameters, we again train 5 models per class, and choose the best for each structure according to the PRG-AUC to use their outputs in the further experiments.

4.3 Results of ALNS Runs

In this section, we present the results on the test instances, that we obtained using our ALNS methods with and without using GNN scores, presented in Sections 3.2 and 3.3.4.
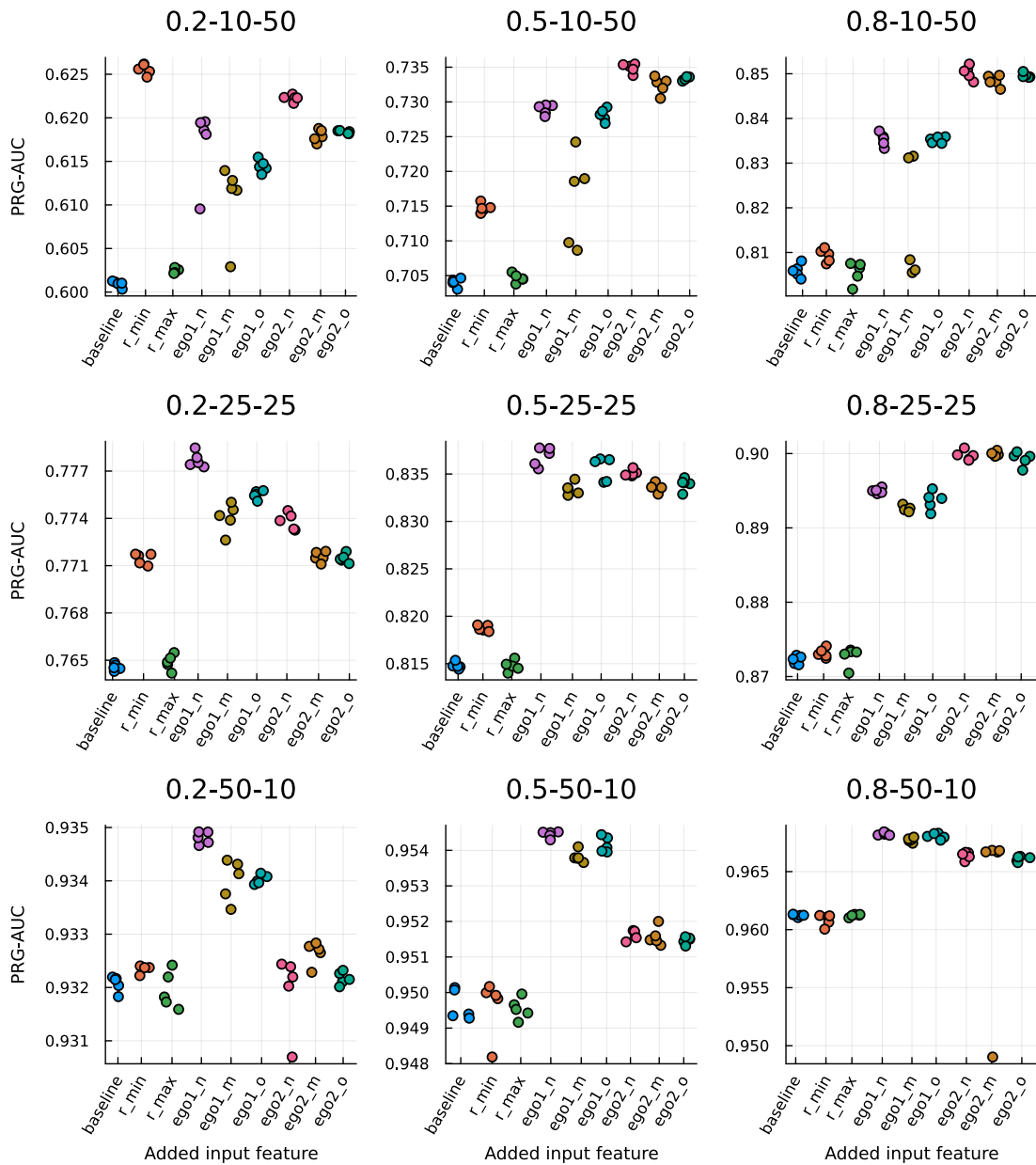
Figure 4.2: Additional feature tuning results for structure "plain", on different instance classes.

We also provide a comparison to the reported results of the genetic algorithm (GA) by Alvarez et al.[AMS21], since they were obtained on the same instances.

Both repair methods, the greedy and the weighted random optimization together with the greedy restoring phase, were used for all of the experiments. In addition, various
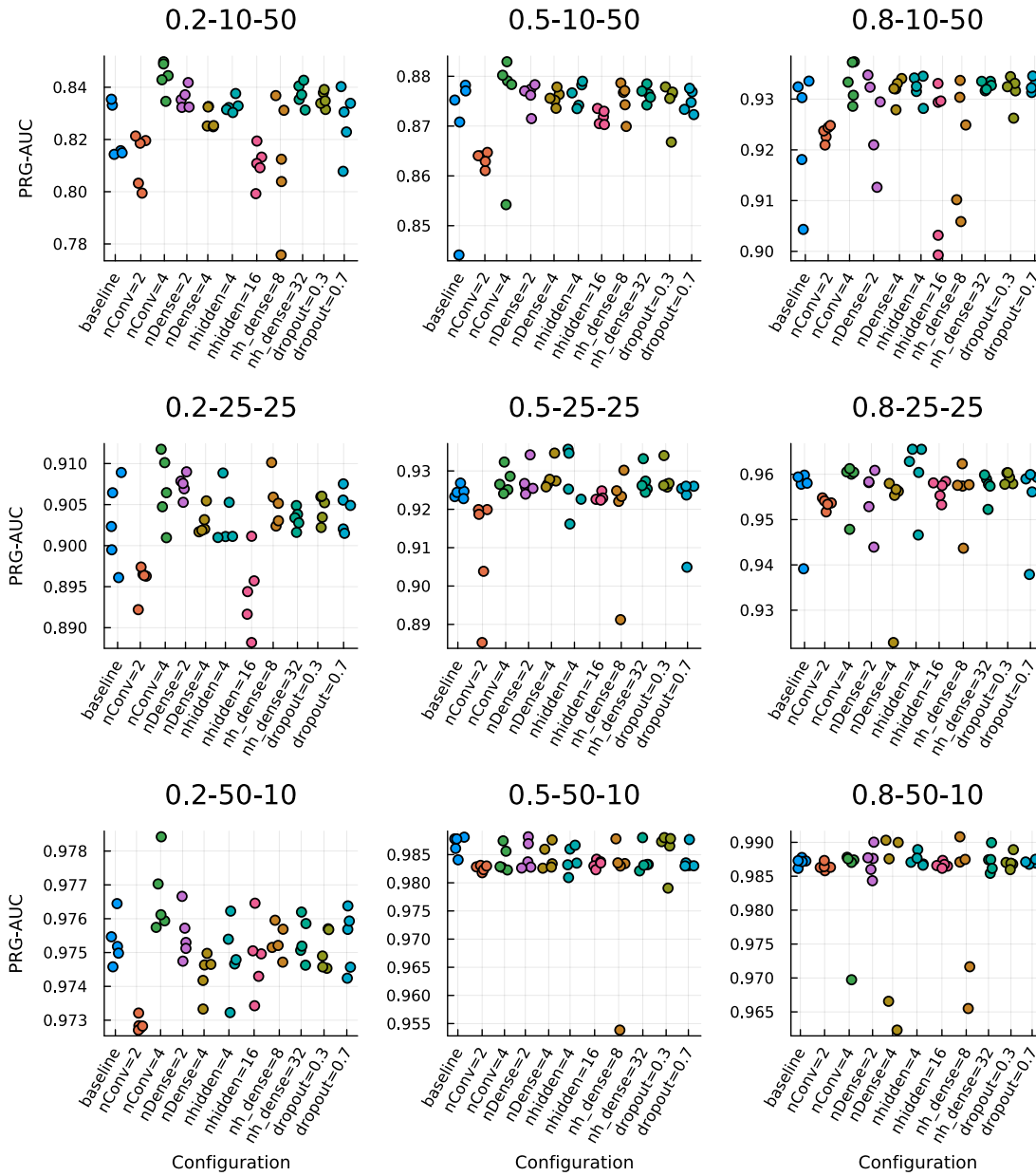
Figure 4.3: Parameter tuning results for structure "trans", on different instance classes.

combinations of the described destroy operators are used. All of the listed destroy operators for one set of results are picked with the same probability (per parameter).

For the initial solutions, we use greedy approaches. We try both, starting with an empty dominating set, as well as starting with the full vertex set, and greedily add resp. remove vertices as in the greedy repair approach.

The time reported is the time that was needed to find the presented solution. All methods were run for 90 seconds per instance on a Intel Xeon E5540, 2.53 GHz Quad Core. For the GNN-supported methods, neither the time to train the GNN nor the time to generate the scores for vertices on the test graphs is included. However, the latter is insignificant: computing the scores took about 8 seconds for all 135 test instances, implying an average of about 0.06s for scoring a single instance.

For each of the reported method combinations, 10 runs were done per instance. All of them are taken into account in the plots, unless stated otherwise. For the ablation studies, where only one of the methods were used (with different parameters), only 5 runs were done per instance.

4.3.1 ALNS with traditional operators only

To determine a baseline performance of our ALNS algorithm, we compared different combinations of the traditional destroy operators described in Section 3.2. The following parameters are used (amount of vertices to be added or removed, either absolute or relative to the number of candidate vertices).

- voting: 20%, 5

- weighted random remove 30%, 5

- random add: 30%

- cost: 5

- For runs with only one method, the parameters 20%, 40% and 5 were used.

Some of the proposed methods proved not to be very effective, for example the "pair-" or "region-" based ones. A possible explanation would be the graph structure - especially for the more dense graphs, it is probably no longer the case that the graph is most efficiently covered by pairs, and the two-step-neighborhood (or a similar region) almost makes up the whole graph, so it makes little difference to distinguish the near vertices from the far ones.

On the following figures, on the x-axis the primal gap is shown, ie. $\frac{\text{obj}-\text{opt}}{\text{opt}}$, where obj is the objective value of the solution, and opt is the best known objective value from the baseline [AMS21] (even if not proven optimal). On the y-axis we see, how many of the runs have achieved a primal gap of at most that much percent. Since this is also given as a percentage, we can compare the performance although the methods were run a different number of times (and we have only one result for each instance for the baseline), although there remains some uncertainty.

As seen in Figure 4.4, the combination of our voting heuristic, together with weighted random removal of vertices, performed best in our tests, although not by much. Adding
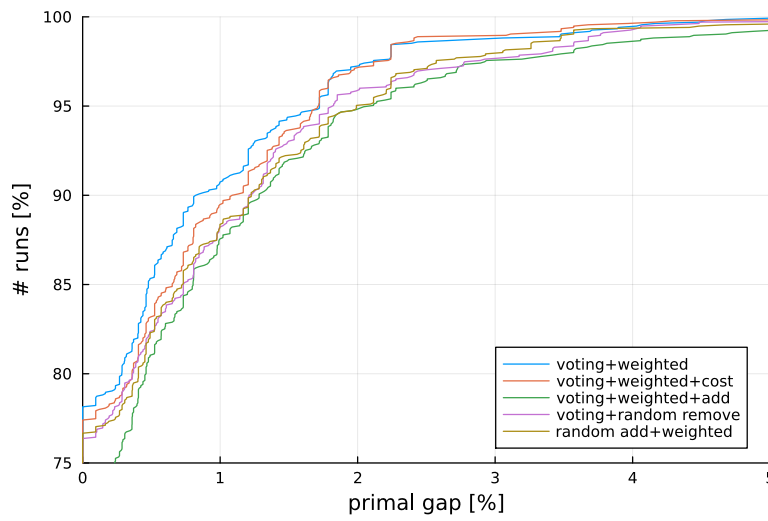
Figure 4.4: Different combinations of the destroy methods.

some of the other methods seem to even worsen the average performance – probably because they are taking computation time, or lead to non-optimal local optima too fast.

We also conducted an ablation study, to see whether one of the two destroy methods can be omitted. However, it is clear in Figure 4.5, that these methods alone have much worse results.
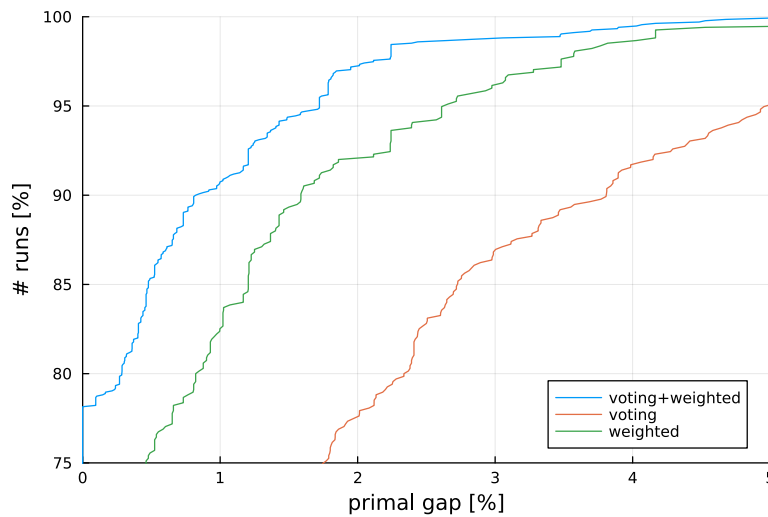
Figure 4.5: Comparison to each of the two methods alone.

4.3.2   ALNS with methods using the scores from GNNs

For the destroy operators using the scores from a GNN, there are a few variants that have to be considered before comparing the results to the baseline. In order to keep the experiments to a feasible number of combinations, one aspect is tested after another. For the first tests, we make an educated guess on which of the untested variants brings the best results, and for later runs, we use the variants that performed best previously. Note that the energy-based loss function was only evalutated on the best model because of difficulties during training: The models failed to converge more often than for logloss, and in general the training process took a lot longer due to the complicated computation of the loss (taking 1000 solutions for each instance).

Table 4.3 contains a summary of which variants were tested. The results and the details of the tested variants are described below.

Table 4.3: Summary of the variants tested.

Aspect	Tested Variants	Guess	Best	Comment
ALNS methods	**gnn only**: GNN methods only **trad+gnn**: GNN + traditional methods **gnnadd+weighted**: GNN add + trad. remove **voting+gnnremove**: GNN remove + trad. add	-	trad+gnn	
Removing submethod	**keep**: Picks vertices to be kept, weighted by out_v **inv**: Picks vertices to remove, weighted by $\frac{1}{out_v}$	inv	keep	We also compare the gnn-only methods in both variants.
Model structures	**edges**: Transformer layers with edge embeddings **trans**: Transformer layers with edge weights as the only edge input **plain**: Only dense layers	edges	edges	Pre-selection based on the PRG-AUC-Scores on validation instances
Training instance size	**75**: Models are only trained on instances with $n = 75$ vertices **100**: $n = 100$ **125**: $n = 125$ **MIX**: Individual models trained for each instance size	MIX	MIX	To study generalization capabilities
Loss function	**logloss**: Classical binary cross-entropy loss **energy**: Energy based loss function (see Section 3.3.2)	logloss	logloss	Due to problems during training, only checked for the best model.

In the following figures and tables we always compare the results of 1350 ALNS runs: 10 runs on each of the 5 test instances in each of the 27 classes.

ALNS method combinations

The first question is, which combination of destroy methods should be applied. For this, we used the scores of the "edges" models, as they showed the most promise with regard to their precision-recall-gain area-under-curve scores (on the generated validation instances), as well as the "inv" algorithm for removing vertices.

In these variants "trad" refers to all of the traditional methods from the best configuration found above (see Section 4.3.1), i.e. the voting add method and the weighted remove method. "gnn" refers to the combination of the "gnnadd weighted" and the "gnnremove weighted" methods described in Table 3.2. The other variants combine the traditional adding method (voting) with the GNN-based removal, and vice versa. All GNN-based methods are used with the same parameters as their traditional counterpart with the same role: the gnnadd method with two variants that add 20% of the candidates or 5 vertices (like for voting), and the gnnremove with two variants that remove 30% of the candidates or 5 vertices respectively.
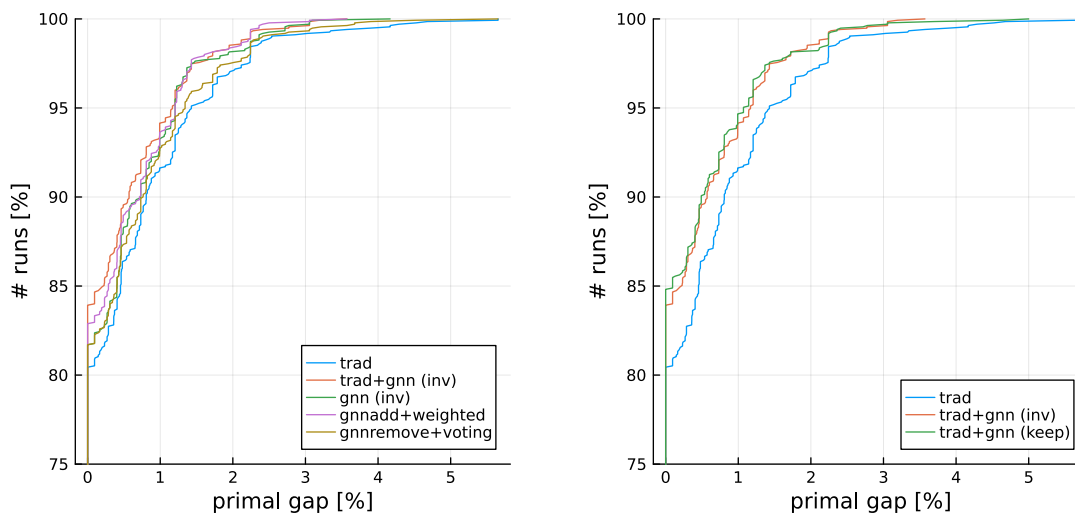
(a) Comparison of different combinations of ALNS methods

(b) Comparison of different submethods for removing vertices

Figure 4.6: Overall performance for different combinations of used gnn-assisted ALNS methods. "trad" (runs using only the traditional methods without using gnn scores) is included as a baseline.

In Figure 4.6, the results are shown for different ALNS-method combinations and removing submethod variants. It can be seen that the combination of both traditional and gnn-supported methods yields the best results in our experiments, with around 84% of runs

finding the best known solution. Also, using just gnn-based destroy methods already performs better than using only the traditional methods. The difference however is quite small. Looking at the detailed results per instance class in Table 4.4, the classes where the differences are most noticeable are the denser classes (0.5-25-25 and 0.8-10-50).

The performance of the different removing submethods is very similar, with the "keeping" algorithm performing slightly better. We therefore use the mixed variant with the voting, weighted, gnnadd and gnnremove (keep) methods for the further experiments.

Table 4.4: Detailed comparison of method combinations (including removing submethods) by instance class. opt% is the percentage of runs that found the best known solution, and $\overline{gap}$ is the average gap (w.r.t. the best known solution). For each variant and instance class, 50 runs were performed (10 runs on the 5 test instances each).

instance class	gnn (inv)		gnnadd +weighted		gnnremove +voting		trad+gnn (inv)		trad+gnn (keep)	
	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$
75-0.2-10-50	76%	**0.4%**	**80%**	0.4%	72%	0.5%	**80%**	0.4%	74%	0.5%
100-0.2-10-50	70%	0.5%	72%	0.4%	68%	0.4%	**74%**	**0.3%**	**74%**	0.4%
125-0.2-10-50	**76%**	**0.1%**	70%	0.2%	70%	0.2%	70%	0.1%	74%	0.1%
75-0.5-10-50	98%	0.0%	90%	0.0%	**100%**	**0%**	96%	0.0%	98%	0.0%
100-0.5-10-50	66%	0.4%	70%	0.2%	**74%**	0.2%	70%	**0.2%**	64%	0.3%
125-0.5-10-50	64%	0.2%	**80%**	**0.1%**	76%	0.1%	74%	0.1%	76%	0.1%
75-0.8-10-50	80%	0.2%	82%	0.2%	92%	0.2%	**100%**	**0%**	**100%**	**0%**
100-0.8-10-50	50%	**0.4%**	48%	0.6%	36%	0.6%	**52%**	0.5%	48%	0.5%
125-0.8-10-50	**100%**	**0%**	94%	0.1%	82%	0.3%	88%	0.2%	94%	0.1%
75-0.2-25-25	60%	0.4%	**66%**	0.4%	52%	0.5%	58%	0.4%	62%	**0.4%**
100-0.2-25-25	**68%**	0.2%	**68%**	0.2%	64%	0.3%	**68%**	**0.2%**	66%	0.2%
125-0.2-25-25	20%	0.8%	**36%**	0.7%	**36%**	**0.6%**	34%	0.7%	**36%**	0.8%
75-0.5-25-25	84%	0.2%	82%	0.2%	**96%**	**0.1%**	84%	0.2%	92%	0.1%
100-0.5-25-25	**100%**	**0%**	**100%**	**0%**	96%	0.0%	**100%**	**0%**	**100%**	**0%**
125-0.5-25-25	78%	0.2%	86%	0.1%	88%	0.1%	90%	0.1%	**92%**	**0.1%**
75-0.8-25-25	**100%**	**0%**	**100%**	**0%**	96%	0.0%	**100%**	**0%**	**100%**	**0%**
100-0.8-25-25	80%	0.1%	72%	0.1%	72%	0.2%	80%	0.1%	**90%**	**0.0%**
125-0.8-25-25	40%	0.6%	46%	0.5%	**62%**	0.5%	52%	0.5%	54%	**0.4%**
75-0.2-50-10	98%	0.0%	98%	0.0%	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.2-50-10	**100%**	**0%**	98%	0.0%	80%	0.3%	96%	0.0%	98%	0.0%
125-0.2-50-10	98%	0.0%	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	98%	0.0%
75-0.5-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
125-0.5-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-50-10	**100%**	**0%**	**100%**	**0%**	98%	0.0%	**100%**	**0%**	**100%**	**0%**
125-0.8-50-10	**100%**	**0%**	**100%**	**0%**	96%	0.0%	**100%**	**0%**	**100%**	**0%**

Comparison of GNN structures

Next, we compare the results when using different models to provide the scores for the vertices to our ALNS methods. First, we trained models for each of the 27 classes individually, and computed the scores on the test instances for the corresponding class that the models were trained on. This corresponds to the variants with the suffix "MIX" in the following figures.

Second, we also experimented with scores only taken from the models trained on instances with $n = 75$, $n = 100$ or $n = 125$ nodes (but still with respect to the 9 remaining classes, i.e. only the scores for the instance classes 75-0.2-10-50, 100-0.2-10-50 and 125-0.2-10-50 were provided by the same model), to investigate how well the models can generalize w.r.t. instance size. This is only done for the "edges" structure.
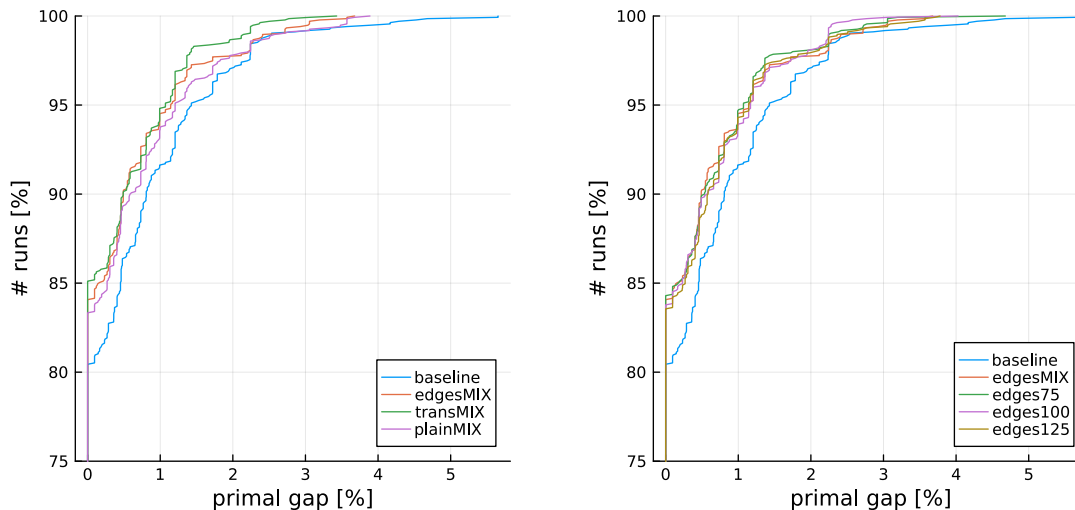
(a) Comparison of different model structures

(b) Comparison of models trained on different instance sizes

Figure 4.7: Comparisons of ALNS runs using scores from different GNN-models. "baseline" refers to the ALNS runs without any GNN-based destroy methods.

In Figure 4.7a we see somewhat surprisingly, that the "trans" structure performed slightly better than the "edges" structure in our experiments, which still outperforms the "plain" structure. We attribute this mostly to random variance of the ALNS runs, since the exact same parameters and model outputs were used as in the previous comparison ("trad+gnn (keep)" in Figure 4.6b), where the same configuration also found the best known solution in 85% of the runs. It is difficult to then interpret the results in any meaningful way, since it seems that the same configuration has a larger variance than what the effect between tested variants is. Detailed tables on the performance of different models are given in Appendix C. The difference is even smaller between models with the same structure, trained on different instance sizes. However, this can also be seen as a positive sign that it does not seem necessary to perform additional training on different instance sizes.

We also manually analyzed the outputs of those models, in order to understand what the differences e.g. between the edges125 and the edges75 models were. In the samples we took, we observed a clear correlation between the predictions, i.e. a node with a high prediction value according to one model most likely also has a higher value according to the other, although outliers exist and the absolute values of the outputs vary. See Figure 4.8 for the comparison on one (arbitrarily chosen) instance.
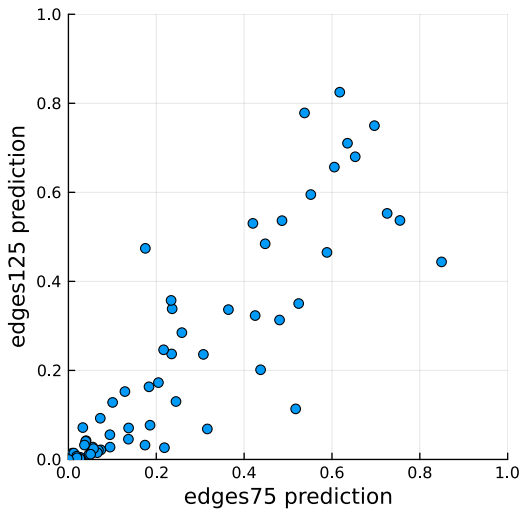
Figure 4.8: The predictions of two models plotted against each other, for all nodes of instance "NEW-75-0.2-10-50-1.wtdp"
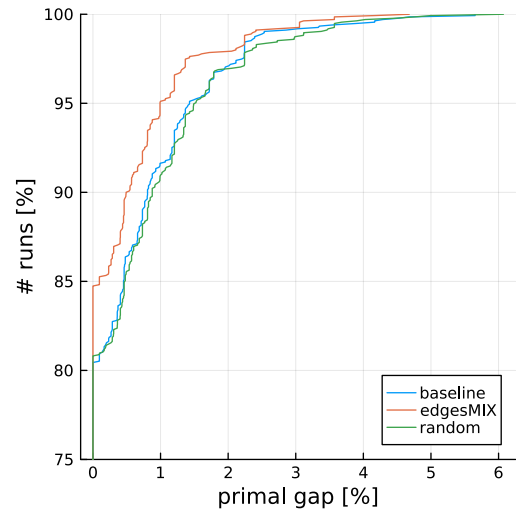
Figure 4.9: Performance of the ALNS using gnn-methods with random scores, in comparison with using the edges model, or no gnn-methods at all (baseline)

Another interesting aspect is seen when comparing different models trained with exactly the same parameters and input data on different instances. In Figure 4.10, the PRG-AUC scores of five models are shown for all validation instances of the same class. Here, we can see that within the same class, there still seem to be instances that are easier for all of the models, and some that are harder for most.
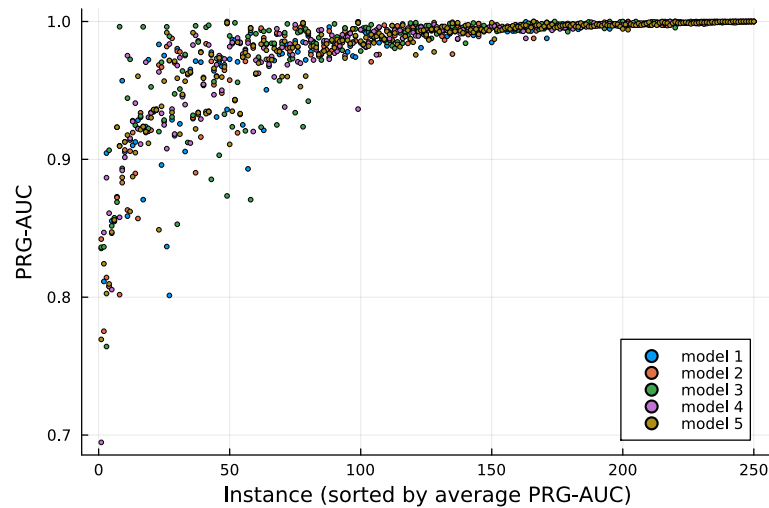
Figure 4.10: Comparison of the PRG-AUC values of five models trained on the same parameters and input data, on validation instances of the same instance class.

Due to the very similar results of different models, we also conducted an experiment where we used the GNN-based destroy methods with completely random outputs for the vertices, to investigate whether there is an influence of the concrete vertex scores at all. However, in our experiments, the results of using the random scores for the gnn-based methods basically matched the baseline (of not using them at all, and only relying on the traditional methods), see Figure 4.9. This indicates that the GNN outputs do matter, and that the cause of the similar results is more likely to be the similarity between the predictions themselves. Note how here again, using the edgesMIX scores had better results than in the previous experiments in Figure 4.7, showcasing the variance of the ALNS runs even when using the exact same inputs (apart from the random seed).

Still, this is a somewhat surprising result, since the PRG-AUC scores (and other considered measures such as validation loss and PR-AUC) varied much more between the different model structures, especially on the classes with a larger value-range on the edge weights (class 10-50). Compare for example the result value ranges on the y-axis of Figures 4.3 and Figure 4.2, where the PRG-AUC values of the individual models are plotted.

Model trained using the energy-based loss function

Finally, we also performed the runs with scores from a model that was trained using the energy-based loss function introduced in Section 3.3.2. We used the "edges" structure and the same parameters as before, but only trained on instances with $n = 75$ vertices to reduce the required number of models to be trained. As seen in Figure 4.11, there is not much of a difference to the model trained with the classical, binary cross-entropy loss (logloss).
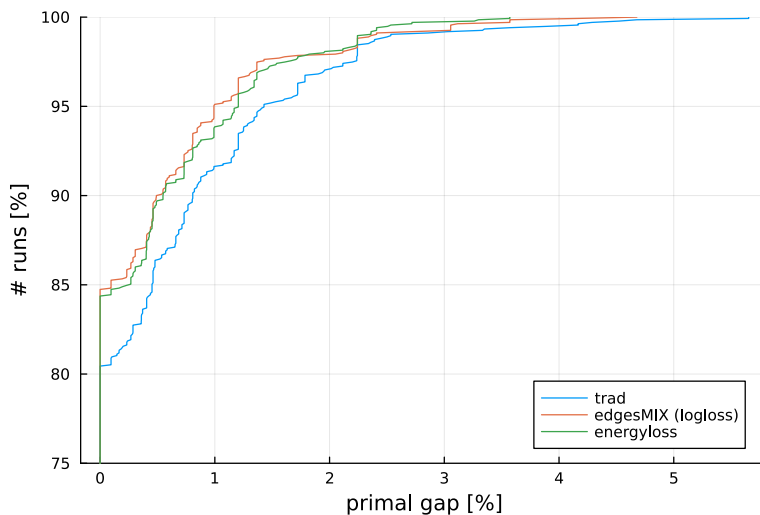
Figure 4.11: Performance of the ALNS using scores from a model trained with the energy-based loss function.

4.3.3   Capability of models to generalize

Until now, the used models were always trained on randomly generated graphs with the same properties (density and weight structures) as the target graphs. In order to investigate whether the models can be used on different classes as well (effectively reducing the required training effort), we compared their PRG-AUC scores with the results of the models trained on the target class. The goal was to find a smaller set of models, which could be expected to perform similarly on the whole dataset as the 27 individual models for each class. Then, we let the ALNS run with the outputs from just this set of models.
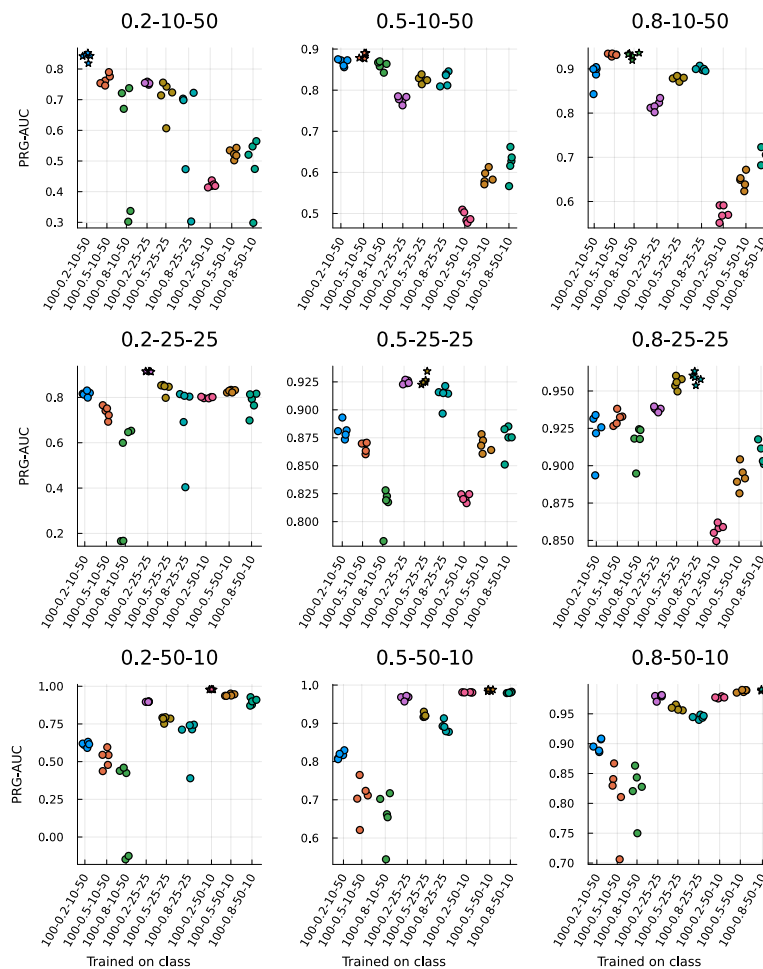
Figure 4.12: The PRG-AUC scores on validation instances, of models trained on different instance classes. The models which were trained on the same instance class are marked with a star.

In Figure 4.12 we can observe that the strongest contenders are always the models, that were at least trained on graphs with the same weight structure. Out of these, we chose to

use models trained on the "0.5"-instances, both as the middle option and the seemingly best variant on average. For a single model, the central instance class 0.5-25-25 would seem like the best compromise, but especially on the vertex-weight-heavy graphs (50-10), it looks like they perform significantly worse than their counterpart trained on 0.5-50-10. Note that the models we observed here were all trained on instances with 100 vertices, which was mainly selected as the middle option.

Results of ALNS-runs using the output of those models are shown in Figure 4.13. Although there is still some improvement in relation to the traditional methods only, the quality of the solutions was generally worse than when using models trained on same instance class as the target.
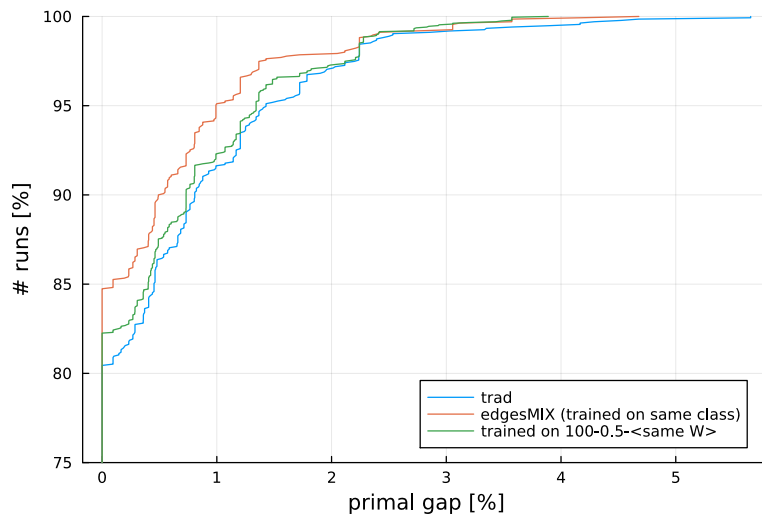
Figure 4.13: Comparison of ALNS runs using models trained on the same instance class as the target, against models that were trained on instances with only the same weight structure W.

4.3.4 Proportion of new solutions

One concern about the GNN-based destroy methods was, that based on the scores there could be a strong preference for a few vertices, which could actually hinder the discovery of new solutions and therefore of new optima. To investigate this possibility, we analyzed the variety of discovered solutions after a large number of iterations.

After training the first models, but before the final parameter tuning, the ALNS using only GNN-based methods was run on all 1250 train- and validation instances of a single instance class (100-0.8-25-25). The class was chosen for the preliminary experiments due to the comparatively bad performance of the ALNS with traditional methods, and the medium size. The parameters used were slightly different parameters than above (20% of candidates, or 5 vertices flat for both the adding and the removing variant). During

these runs, all intermediate solutions were collected, i.e. the new candidate solution after both the destroy and the repair method were applied.
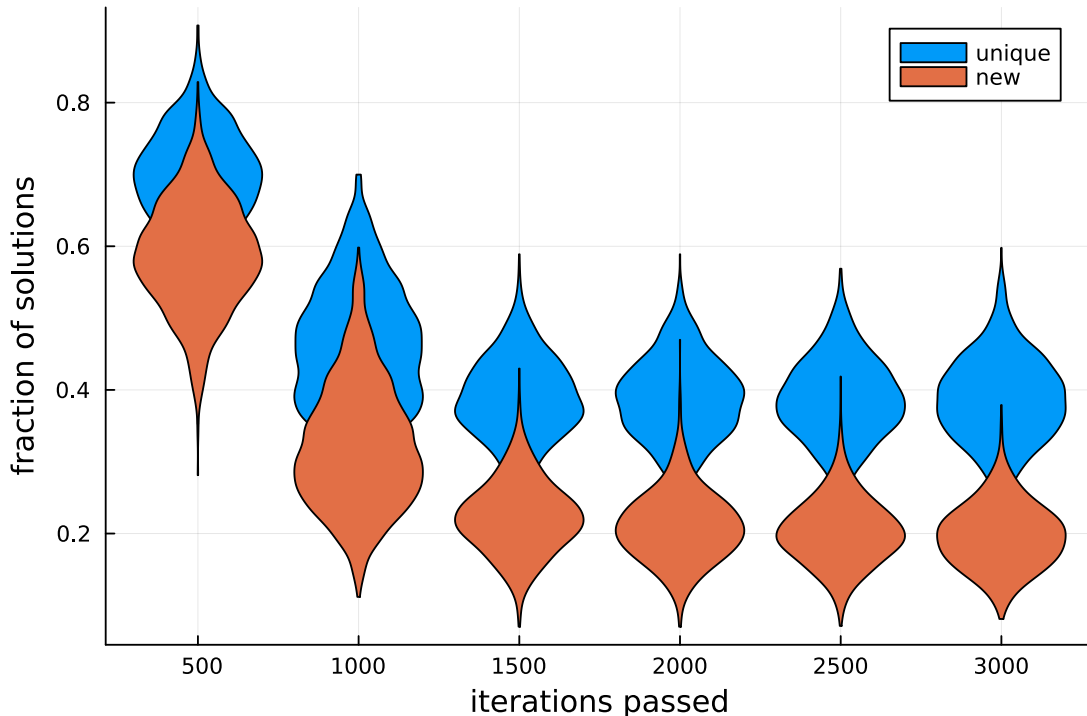
Figure 4.14: Violin plots showing the distribution (across runs) of the fraction of 1. unique and 2. new solutions among the last 100 iterations, after 500, 1000, etc. iterations.

The results are presented in Figure 4.14. It can be seen that after 1500 iterations (or even earlier, see Figure 4.16), the distribution stabilizes to be approximately normally distributed around $\mu = 0.2$ for new solutions. In other words, among the 100 solutions in iterations 1400-1500, there are on average 38 different (unique) solutions, of which on average 20 were not seen in this run before. This is clearly less than the almost 60% on average in the beginning (after 500 iterations). Nevertheless, it also means that we are nowhere near a situation where no new solutions are discovered at all.

Figure 4.15 shows how many iterations the runs had in total on average. The limit of 3000 for this analysis was chosen on this basis.

Note that those runs were done on different hardware, so this is not an indication for the number of iterations in the "real" runs (which most of the time did not reach 3000 iterations). Other factors limiting the applicability of these insights include the different model scores used (from an intermediate, only manually optimized model), and especially the single instance class for which this data is available. But still, it is at least an indication that there is no problem with being stuck on the same solutions over and over again.
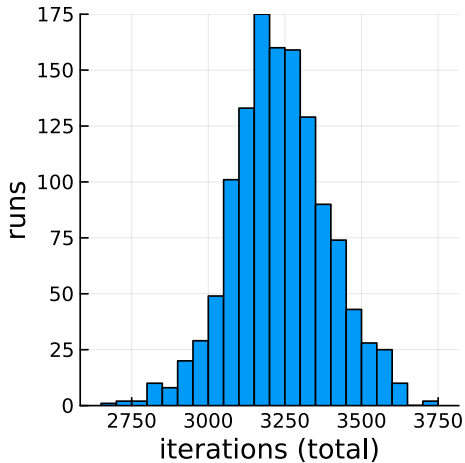
Figure 4.15: Histogram of the number of iterations on one run each on 1250 generated instances for 100-0.8-25-25.
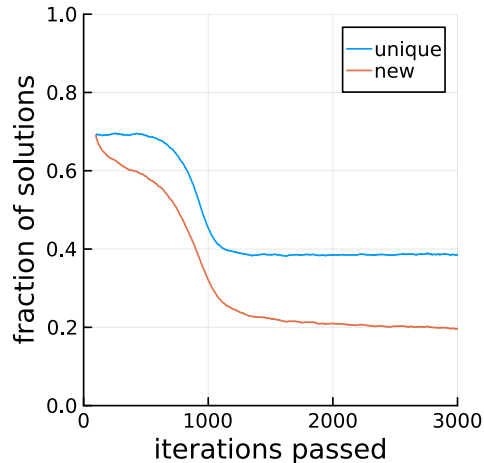
Figure 4.16: The fraction of the new and unique solutions found in the last 100 iterations, after the given number

4.3.5 Comparison to the baseline

In comparison to the genetic algorithm by Alvarez and Sinnl [AMS21], our approach with traditional methods mostly has similar results – it also reaches optimality in about 80% of the runs, and seems to have a slightly lower primal gap for the harder instance on average (see Figure 4.17). While the traditional methods only can keep up with the performance of the genetic algorithm, using the gnn-based methods in addition seems to bring a more noticeable advantage.

4.3.6 Analysis of the effect of instance size and structure

If we consider the results on the different sizes, weight structures and densities of the input graph, we can also make some interesting observations. For example, for the larger instances, it seems that the primal gap of our approach was slightly bigger, and for the smaller ones, it was the other way around (Figure 4.18).

A similar image, but a bit clearer, can be seen for the graph density. Here, the GA finds the best known solution for almost all of the most dense instances, while our approach only does so for 80% ("trad") resp. 85% ("trad+gnn") of the runs. For the less dense graphs, however, our approach seems to be a bit better, see Figure 4.19.

The most interesting pattern is seen for the different weight structure. It seems that for the cases where either vertex or edge weights have a larger maximum than the other, the average achieved primal gap is smaller with our approach, and especially when vertex weights are more important, it is significantly better. However, for the instances with the same weight range for both, this is no longer the case.
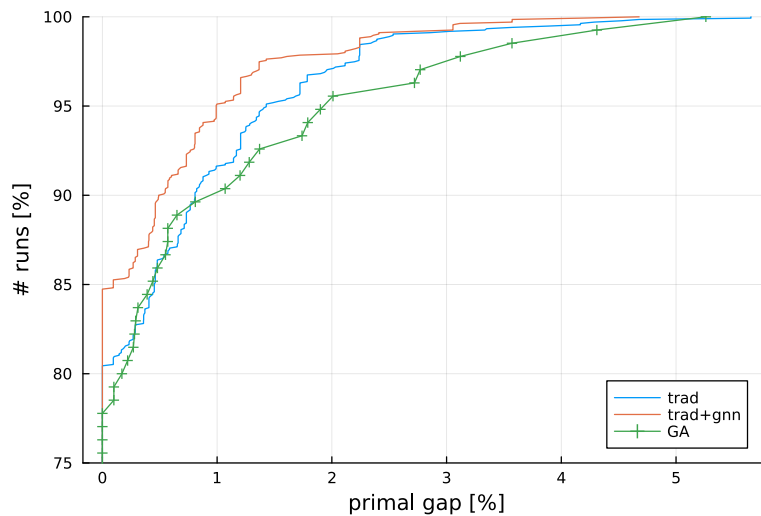
Figure 4.17: Comparison to baseline (genetic algorithm (GA), results reported by Alvarez et al.[AMS21]).
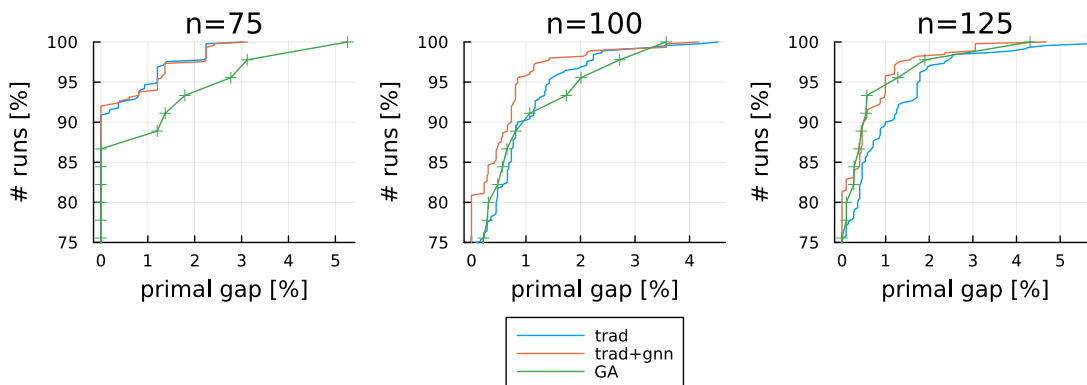
Figure 4.18: Comparison on all instances grouped by their size (number of vertices).

4.4   Discussion

After extensive experiments for training a GNN and optimizing the models backing the new ALNS methods, we can actually see a small improvement to our baseline approach. This basically is true across all different instance classes, although the exact numbers vary. The most significant improvement seems to be for the class of instances, where the vertices have a larger range of weights than the edges (50-10).

There are several caveats of these results. First and foremost, during training and tuning the models as well as for the final results of the ALNS runs, we observed a high variance of the results. Even though we did 10 runs per instance, this effect can still be easily observed between two sets of runs with the exact same inputs.
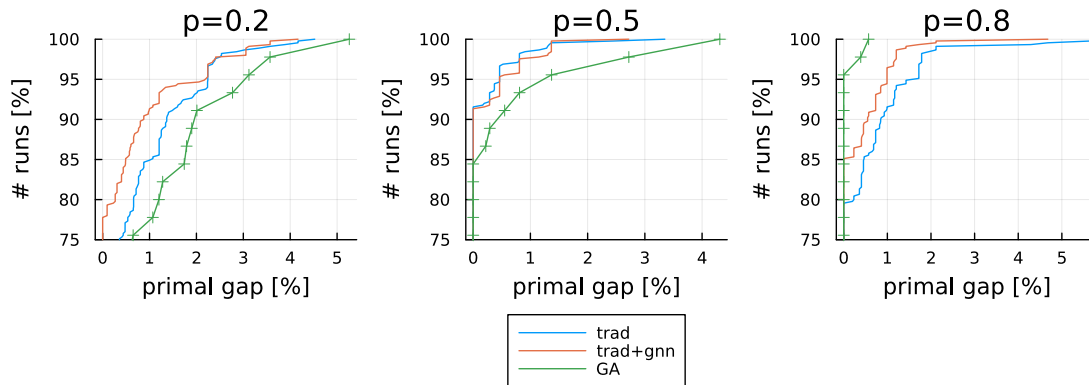
Figure 4.19: Comparison on all instances grouped by their density, where p is the edge probability in the randomly generated graphs.
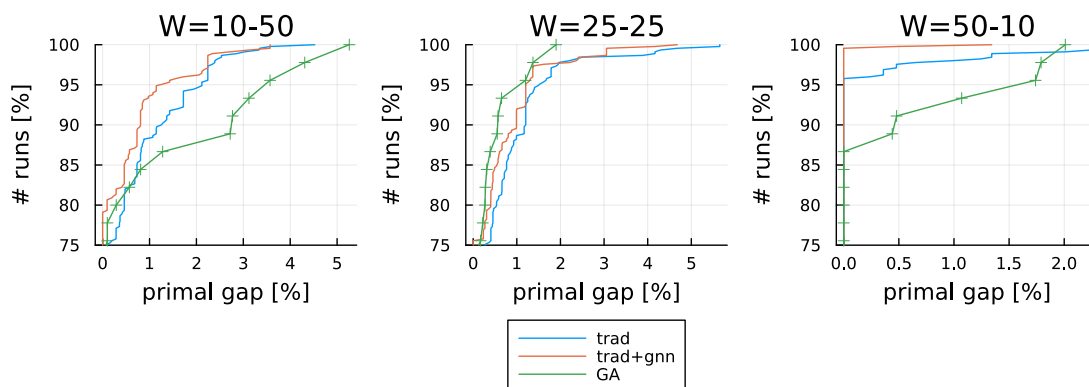
Figure 4.20: Comparison on all instances grouped by their weight structure W. The first number denotes the maximal vertex weight, and the second the maximal edge weight.

Important to note here is that only a few instances are affected at all. There are a lot of instances that are (almost) always solved optimally by all our approaches (as well as the baseline), and these instances can be found in almost all instance classes. The differences mostly arise in a few, seemingly more difficult instances, where, for example, one approach finds the optimal solution in around 20% of the cases, and the other one in around 50% of the cases.

We did not conduct any further investigation on what makes these specific instances more difficult than others. The class of the instances definitely seems to have an influence on it, but it cannot be said that all instances of one class are strictly more difficult than another. Especially in the comparison with the genetic algorithm by Alvarez and Sinnl, we can still see that the type of algorithm has an impact on which classes are more likely to be solved optimally. One possible explanation for the GA to be better on denser

65

graphs, and worse on less dense graphs, would be that the high number of incumbent solution prevents being stuck in a local optimum for longer. Even though we showed that after thousands of iterations, we still discover a new solution in around every fifth iteration for the ALNS approach, those are always searched from a single incumbent solution. A stepping stone to the optimal solution might have been discarded before, whereas in the genetic algorithm there is a higher chance that it is kept in the population.

An interesting effect to observe is also that although the different model structures had very different results on the validation graphs with respect to the PRG-AUC scores (which were used for pre-selection and tuning of models), the result of using them in the ALNS was almost the same. Two explanations for this come to mind: It could be, that the essential information for the improvements are already found with the less sophisticated models, and the further improvements of the PRG-AUC scores are caused by smaller, irrelevant changes in the outputs, that are covered up by the randomness of the vertex selection in the gnn-based ALNS methods. Another explanation would be, that there is something about the ALNS-based destroy methods themselves, that brings the improvement, and that the model outputs are not that relevant after all. However, we at least conducted the experiment with only random vertex outputs, which showed that the model outputs have at least some significance and can not be disregarded completely.

All in all, there seems to be more space for improvement with regard to the ALNS-methods using the outputs of GNN models, rather than improving the GNN models themselves. Also, investigation on the few more difficult instances might be worth looking into, however, for a purely artificial dataset such as this it may not bring any benefits. The current results are enough to observe at least a trend. Finally, it might also be worth it to improve the ALNS algorithm itself by hyperparameterization, allowing more deviation from the current incumbent, finding new ALNS-methods altogether, or trying restart approaches (since only for a very small number of instances, **none** of the runs have found the best known solution).

CHAPTER 5

Conclusion

On the example of the Weighted Total Domination Problem, we investigated the possibility to improve the performance of a traditional metaheuristic, in our case the Adaptive Large Neighborhood Search, by using the output of a Graph Neural Network. We have experimented with new approaches on different points of that problem – preprocessing, ALNS-methods, and model-training – and found small, but visible improvements.

There were five preprocessing rules introduced, which are adaptions of the ones for the MWDS by Wang et al. [WCCY18]. Although they could not be applied on the instances we ran our final experiments on, they could be useful to reduce the number of decisions on graphs with a number of nodes with a degree of only 1 or 2. Some of these decisions are probably trivial to find with other methods, but it would still effectively be an instance-size reduction in those cases.

For the ALNS-methods, we presented the "voting-based" destroy method, viewing each node as a "voter" that will vote for a node in its neighborhood to be added to the solution, in case it brings an improvement to the objective function in a local sense – that is, if the nodes external cost is reduced. Together with a random aspect that introduces some variance in this "behaviour", it is better than just adding nodes randomly, and works well together with the destroy method that removes vertices based on the vertex weight alone, being part of the best configuration for our experiments with traditional destroy operators only.

Also, the GNN-supported destroy-methods were introduced. These methods utilize the GNN output as the weight for random selection of vertices to be removed or added to the solution. The runs using those methods in addition to the traditional methods found the best known solution in approximately four percentage points more of the runs (85% instead of 81%, roughly).

Interestingly, the different model structures, parameters and training instance sizes did not seem to have a noticeable impact on the performance of the ALNS, as long as

they were reasonably chosen. This was unexpected, since when comparing the models'
performances on the validation set (using measures like the Precision Recall Gain AUC),
there was indeed a significant difference at least between the different model structures.
However, the same methods using random scores instead of the output of a model only
performed on baseline level, so our explanation is that the model outputs are just quite
similar, regardless of the model structure. Also, the random nature of the GNN methods
make such small differences even more unlikely to show up in the end.

Here, we definitely see the possibility for future work to investigate, where exactly the
model complexity and size has to be in order to make an impact. We extensively tested
parameter configurations for training the GNN models, in order to be sure that for each
of the structures, we have a good model (resp. good outputs) for a fair comparison.
Also, we trained those models on 1000 randomly generated training instances per class,
used further 250 validation instances, for which we had to find reasonably good solutions
beforehand to refer to as the "close to optimal" solutions. We suspect that this could
be drastically reduced, and furthermore that a generalized model can be learned for
multiple classes, especially with regards to instance size and weight structure. The worst
performance for a different class was measured on models that were trained on graphs
with a different density. All in all, it would be interesting to see how much these training
efforts can be reduced without impacting the quality of the model too much.

In order to further improve the performance of the ALNS, different ALNS-methods
seem to be more promising to be looked into. This work only utilizes the vertex-scores
obtained by the GNN in the most basic variants of destroy-methods. There could be
potential in using them in repair-methods as well, and in general there could be other,
more sophisticated repair-methods that could bring an improvement even without the
help of GNN-outputs. An example would be beam-search methods, or limited-depth
exhaustive searches on each repair iteration, as was outlined in Section 3.2.3.

Other interesting directions include different variants of the voting-based destroy operator,
since the voting mode as well as the modeled preferences can be done in a multitude
of alternative ways. For example, instead of a single vote with random influence, each
node could have multiple votes, perhaps with different weights attached, or a single
transferable vote could be implemented. Also, our destroy method only adds new vertices
to the solution D, but in a similar fashion it could make sense for the vertices in D
to vote for vertices to be removed. In general, it is interesting to investigate how such
methods relate to more "traditional" destroy methods. For example, imagine a system
where each node votes for a neighbor proportional to the advantage it would bring on
the objective function, and those votes were normalized in addition by the degree of the
nodes. This would at least be close to just computing δ_u, the effect of taking u into D,
and it remains to be seen whether this "voting" mechanism in some cases just provides a
different way to compute the same scores. Finally, it would be interesting to see simliar
appliances in different problems than the WTDP.

One problem that has occured on multiple occasions during our experiments was the
randomness of runs or model trainings that were performed with exactly the same

parameters. Having performed 10 runs on each instance, for some of them it seems that this is enough to show the randomness between the runs, but too little to be sure whether this randomness has a larger influence than the configurations itself. With this insight, future work should include more runs per instance or at least per instance class (with more generated test instances). It is also hard to justify any argumentation without a practical use case behind it, since the observed differences are, for any real application, quite small: A few percent of runs more find the best known solution, while the rest often only has an optimality gap of 1% or less.

A bit of research should also be done in the direction of those instances that are not always solved optimally. In the ALNS runs as well as when training the GNNs, some of the instances seem to be easier than others, accross multiple different models and ALNS runs. Although this form of randomly generating graphs for the WTDP makes sure to include graphs of different forms and structural properties, it would be interesting to find out what makes these instances difficult in particular, and to be able to extract them in a separate class.

Finally, another idea on using a GNN in combination with an ALNS (or local search based metaheuristics in general) would be to use the current solution as an input to the neural network as well, to compute possible moves instead of just a general estimate on vertices regardless of the current situation. Although this would need a forward-pass in each iteration, it could still be worth the efforts. However, this would probably already fit better into a Reinforcement Learning setting, when thinking about obtaining training data for classical neural network training: How would the target even be defined for a given instance and incumbent solution, when there are most likely multiple intermediate solutions that are valid stepping stones to a new optimum?

Additional Algorithms

Algorithm A.1: getCandidatesAndCount

Input: The vertex set V, $amount \in (0,1) \cup \mathbb{N}^+$ for the desired number of
vertices to be altered, $adding \in \{true, false\}$

Output: A tuple $(C, count)$, where C is the candidate set and count is the
number of vertices to be removed or added

1 **if** $adding$ **then** $C = V \setminus D$

2 **else** $C = D$

3 $C = C \setminus \{u | fixed_u = true\}$

4 **if** $amount \in (0,1)$ **then** $count = \lceil |C| * amount \rceil$

5 **else** $count = \min(amount, |C|)$

6 **return** $(C, count)$

APPENDIX B

Detailed Tuning Results

All results from parameter tuning for the GNN are shown here. Only graphs with more interesting effects are shown in the main work.
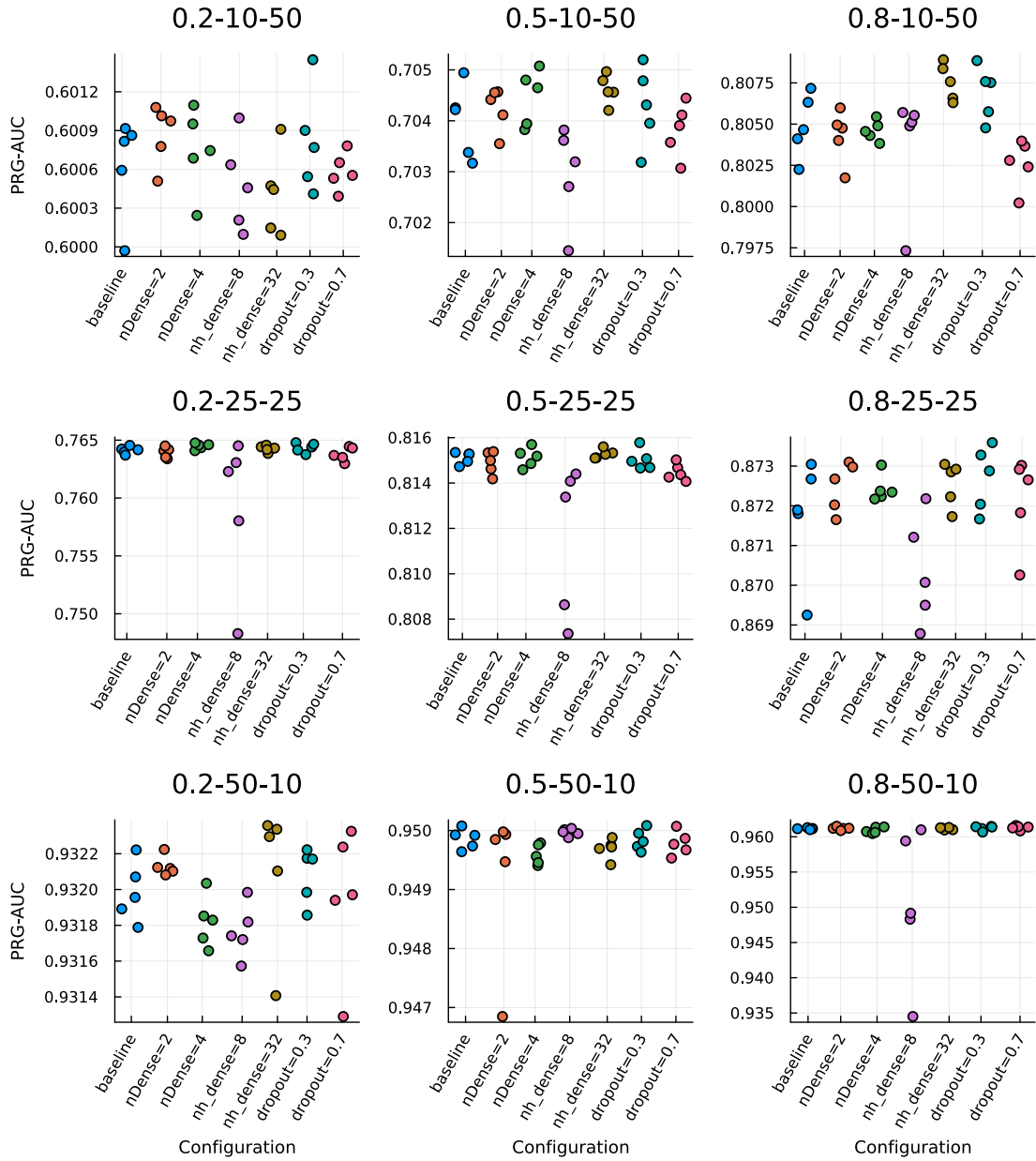
Figure B.1: Parameter tuning results for structure "plain".
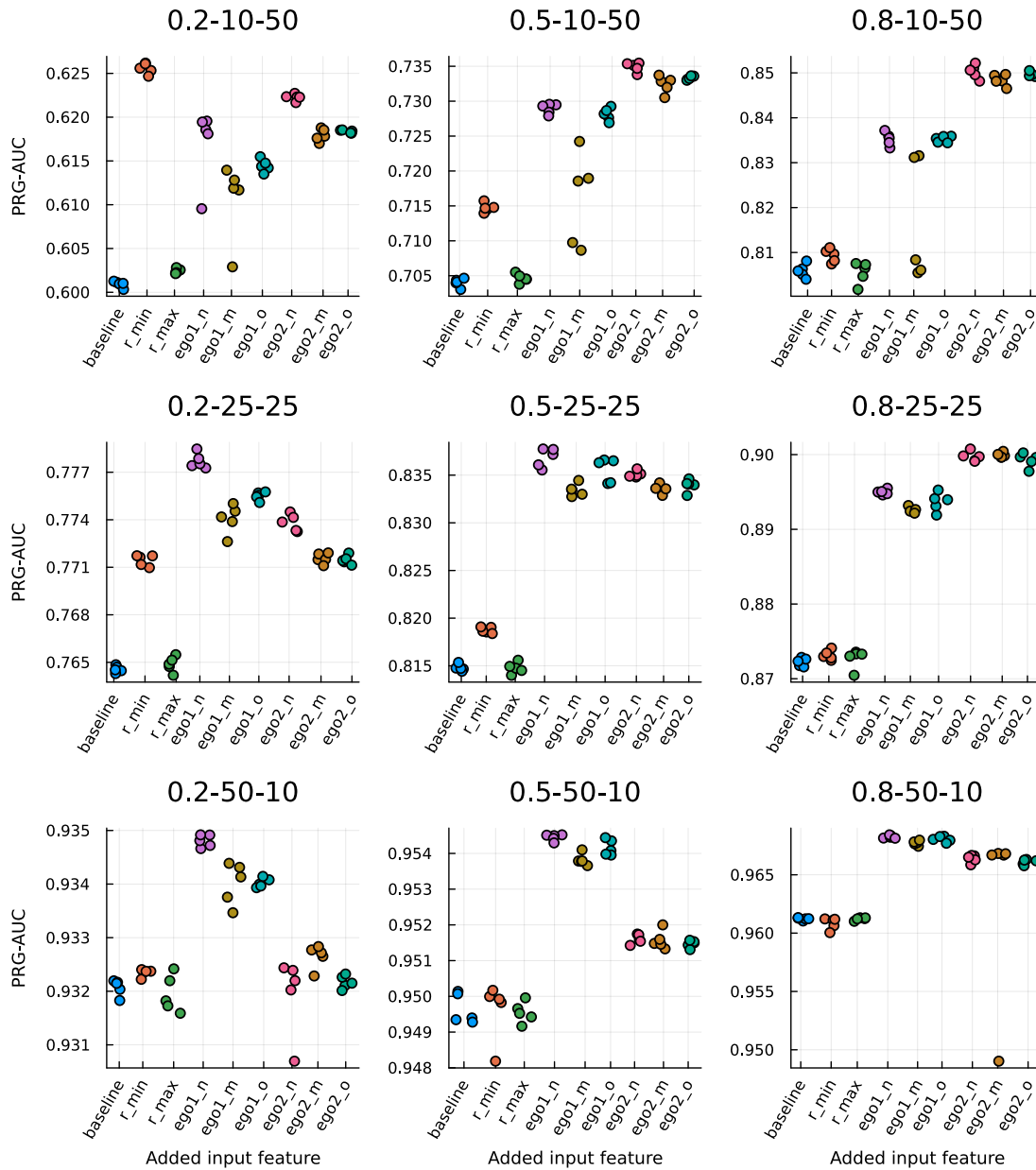
Figure B.2: Additional feature tuning results for structure "plain".
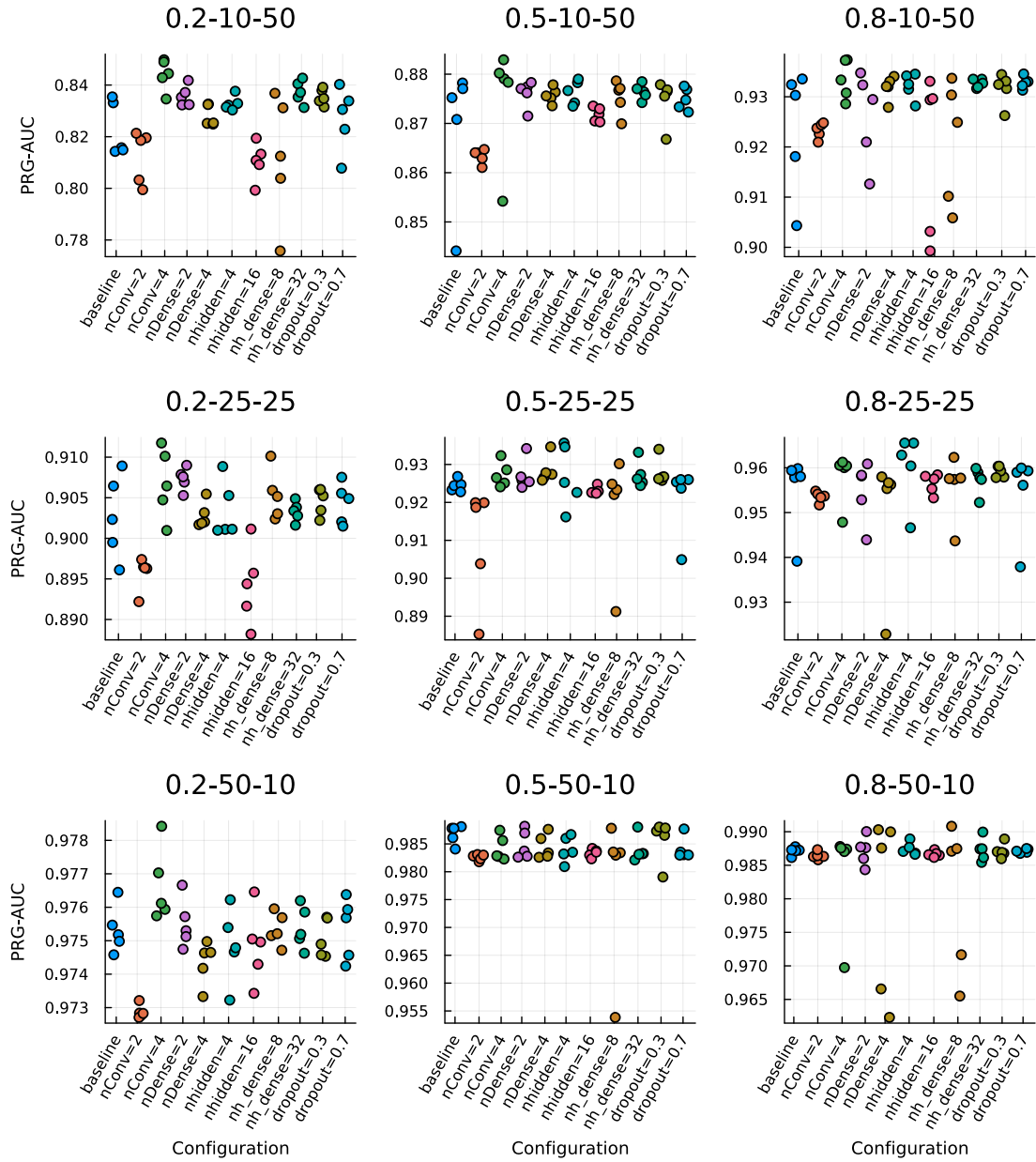
Figure B.3: Parameter tuning results for structure "trans".
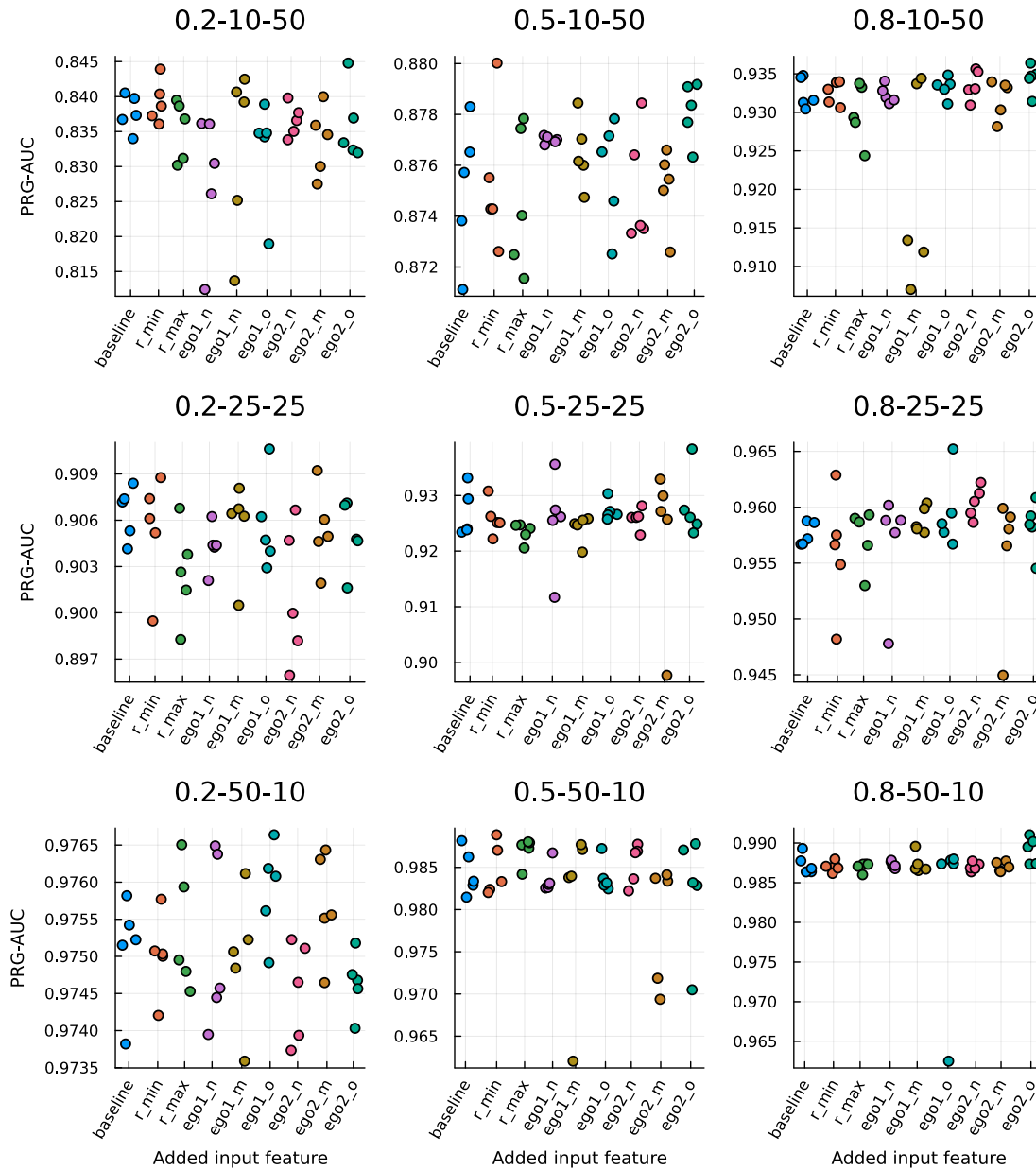
Figure B.4: Additional feature tuning results for structure "trans".

Figure B.5: Parameter tuning results for structure "edges".
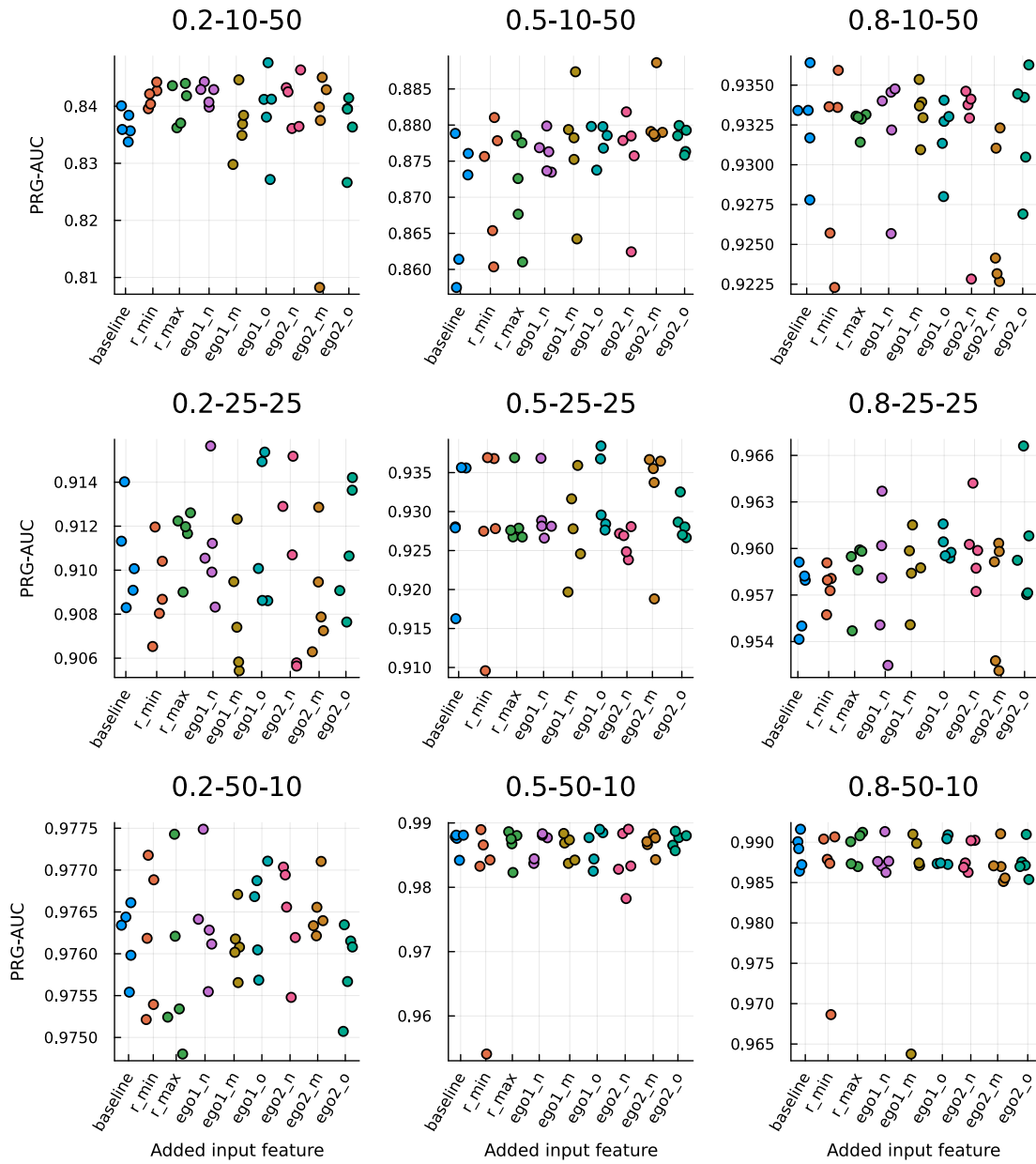
Figure B.6: Additional feature tuning results for structure "edges".

Detailed Result Tables

The following tables contain the more detailed data, averaged over all runs on instance-class resp. instance level. If not specified differently, there are 10 runs per instance, and therefore 50 runs per instance class, for each row. $\overline{gap}$ contains the average gap to the best known solution, and opt% is the percentage of runs that have a gap of 0.

Table C.1: Comparison of model structures on the different instance classes

instance class	baseline		edgesMIX		transMIX		plainMIX	
	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$
75-0.2-10-50	**80%**	0.5%	78%	**0.4%**	74%	0.6%	70%	0.6%
100-0.2-10-50	58%	0.7%	**78%**	0.4%	68%	**0.3%**	60%	0.6%
125-0.2-10-50	66%	0.3%	72%	**0.1%**	**76%**	0.1%	64%	0.3%
75-0.5-10-50	86%	0.1%	**100%**	**0%**	**100%**	**0%**	90%	0.0%
100-0.5-10-50	78%	**0.1%**	70%	0.3%	74%	0.2%	**84%**	0.2%
125-0.5-10-50	74%	0.2%	**76%**	**0.1%**	72%	0.1%	70%	0.2%
75-0.8-10-50	**100%**	**0%**	98%	0.1%	**100%**	**0%**	98%	0.1%
100-0.8-10-50	40%	0.6%	40%	0.5%	52%	0.5%	**56%**	**0.4%**
125-0.8-10-50	80%	0.3%	90%	0.2%	**94%**	**0.1%**	84%	0.3%
75-0.2-25-25	**66%**	0.4%	64%	0.5%	62%	**0.3%**	52%	0.6%
100-0.2-25-25	58%	0.4%	62%	0.3%	**70%**	**0.2%**	**70%**	0.2%
125-0.2-25-25	30%	0.7%	42%	0.6%	**50%**	**0.4%**	38%	0.7%
75-0.5-25-25	94%	**0.1%**	88%	0.2%	86%	0.2%	**96%**	0.1%
100-0.5-25-25	**100%**	**0%**	**100%**	**0%**	98%	0.0%	**100%**	**0%**
125-0.5-25-25	92%	0.1%	94%	0.0%	90%	0.1%	**98%**	**0.0%**
75-0.8-25-25	92%	0.1%	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-25-25	66%	0.2%	76%	**0.1%**	**80%**	0.2%	76%	0.2%
125-0.8-25-25	50%	0.8%	54%	0.4%	54%	0.4%	**70%**	**0.3%**
75-0.2-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	96%	0.0%
100-0.2-50-10	74%	0.4%	94%	0.1%	**98%**	**0.0%**	80%	0.2%
125-0.2-50-10	**100%**	**0%**	98%	0.0%	**100%**	**0%**	98%	0.0%
75-0.5-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
125-0.5-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-50-10	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
125-0.8-50-10	88%	0.1%	96%	0.0%	**100%**	**0%**	**100%**	**0%**

Table C.2: Comparison of model structures on the instances with 75 vertices

instance	baseline		edgesMIX		transMIX		plainMIX	
	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$
75-0.2-10-50-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.2-10-50-2.wtdp	**100%**	**0%**	90%	0.2%	90%	0.1%	**100%**	**0%**
75-0.2-10-50-3.wtdp	90%	0.3%	**100%**	**0%**	90%	0.2%	60%	0.7%
75-0.2-10-50-4.wtdp	**100%**	**0%**	80%	0.3%	90%	0.0%	90%	0.0%
75-0.2-10-50-5.wtdp	10%	2.0%	**20%**	**1.7%**	0%	2.5%	0%	2.3%
75-0.5-10-50-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-10-50-2.wtdp	80%	0.1%	**100%**	**0%**	**100%**	**0%**	90%	0.0%
75-0.5-10-50-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-10-50-4.wtdp	60%	0.1%	**100%**	**0%**	**100%**	**0%**	90%	0.0%
75-0.5-10-50-5.wtdp	90%	0.0%	**100%**	**0%**	**100%**	**0%**	70%	0.1%
75-0.8-10-50-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-10-50-2.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-10-50-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-10-50-4.wtdp	**100%**	**0%**	90%	0.4%	**100%**	**0%**	90%	0.4%
75-0.8-10-50-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.2-25-25-1.wtdp	0%	1.2%	10%	1.5%	**30%**	**0.8%**	10%	1.0%
75-0.2-25-25-2.wtdp	70%	0.3%	**90%**	0.3%	70%	**0.1%**	30%	1.3%
75-0.2-25-25-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.2-25-25-4.wtdp	**60%**	**0.4%**	20%	0.6%	10%	0.8%	20%	0.6%
75-0.2-25-25-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-25-25-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-25-25-2.wtdp	80%	0.1%	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-25-25-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-25-25-4.wtdp	**90%**	**0.1%**	40%	0.8%	30%	1.0%	80%	0.3%
75-0.5-25-25-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-25-25-1.wtdp	90%	0.1%	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-25-25-2.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-25-25-3.wtdp	70%	0.3%	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-25-25-4.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-25-25-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.2-50-10-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.2-50-10-2.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	80%	0.1%
75-0.2-50-10-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.2-50-10-4.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.2-50-10-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-50-10-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-50-10-2.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-50-10-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-50-10-4.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.5-50-10-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-50-10-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-50-10-2.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-50-10-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-50-10-4.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
75-0.8-50-10-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**

Table C.3: Comparison of model structures on the instances with 100 vertices

instance	baseline		edgesMIX		transMIX		plainMIX	
	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$
100-0.2-10-50-1.wtdp	90%	0.2%	**100%**	**0%**	90%	0.2%	**100%**	**0%**
100-0.2-10-50-2.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	90%	0.1%
100-0.2-10-50-3.wtdp	20%	1.4%	**70%**	**0.5%**	30%	1.0%	60%	0.7%
100-0.2-10-50-4.wtdp	40%	0.3%	**80%**	**0.0%**	60%	0.1%	30%	0.3%
100-0.2-10-50-5.wtdp	40%	1.6%	40%	1.3%	**60%**	**0.4%**	20%	2.1%
100-0.5-10-50-1.wtdp	**50%**	**0.4%**	20%	0.6%	0%	0.8%	**50%**	**0.4%**
100-0.5-10-50-2.wtdp	40%	0.2%	70%	0.1%	80%	0.1%	**90%**	**0.0%**
100-0.5-10-50-3.wtdp	**100%**	**0%**	60%	0.8%	90%	0.3%	80%	0.4%
100-0.5-10-50-4.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-10-50-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-10-50-1.wtdp	30%	0.5%	50%	0.3%	**100%**	**0%**	80%	0.1%
100-0.8-10-50-2.wtdp	**20%**	**0.6%**	0%	0.7%	0%	0.7%	10%	0.7%
100-0.8-10-50-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-10-50-4.wtdp	40%	0.8%	50%	0.7%	50%	0.8%	**70%**	**0.5%**
100-0.8-10-50-5.wtdp	10%	1.0%	0%	0.9%	10%	1.0%	**20%**	**0.6%**
100-0.2-25-25-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.2-25-25-2.wtdp	20%	0.8%	**40%**	**0.2%**	10%	0.5%	10%	0.4%
100-0.2-25-25-3.wtdp	80%	0.1%	20%	0.4%	70%	0.1%	**100%**	**0%**
100-0.2-25-25-4.wtdp	90%	0.4%	80%	0.5%	**100%**	**0%**	**100%**	**0%**
100-0.2-25-25-5.wtdp	0%	0.7%	**70%**	**0.1%**	**70%**	0.2%	40%	0.5%
100-0.5-25-25-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-25-25-2.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-25-25-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-25-25-4.wtdp	**100%**	**0%**	**100%**	**0%**	90%	0.1%	**100%**	**0%**
100-0.5-25-25-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-25-25-1.wtdp	30%	0.3%	30%	0.3%	**50%**	**0.2%**	20%	0.4%
100-0.8-25-25-2.wtdp	70%	0.1%	70%	0.1%	**100%**	**0%**	**100%**	**0%**
100-0.8-25-25-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-25-25-4.wtdp	30%	0.7%	**80%**	**0.2%**	50%	0.6%	60%	0.5%
100-0.8-25-25-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.2-50-10-1.wtdp	70%	0.1%	**100%**	**0%**	90%	0.0%	70%	0.2%
100-0.2-50-10-2.wtdp	20%	1.5%	70%	0.4%	**100%**	**0%**	40%	0.7%
100-0.2-50-10-3.wtdp	90%	0.0%	**100%**	**0%**	**100%**	**0%**	90%	0.0%
100-0.2-50-10-4.wtdp	90%	0.1%	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.2-50-10-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-50-10-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-50-10-2.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-50-10-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-50-10-4.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.5-50-10-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-50-10-1.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-50-10-2.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-50-10-3.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-50-10-4.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**
100-0.8-50-10-5.wtdp	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**	**100%**	**0%**

Table C.4: Comparison of model structures on the instances with 125 vertices

instance	baseline		edgesMIX		transMIX		plainMIX	
	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$	opt%	$\overline{gap}$
125-0.2-10-50-1.wtdp	50%	0.9%	**60%**	**0.4%**	50%	0.5%	30%	0.9%
125-0.2-10-50-2.wtdp	**100%**	**0%**	100%	0%	100%	0%	100%	0%
125-0.2-10-50-3.wtdp	70%	0.4%	90%	0.0%	100%	0%	90%	0.1%
125-0.2-10-50-4.wtdp	**30%**	**0.1%**	10%	0.1%	30%	0.1%	20%	0.2%
125-0.2-10-50-5.wtdp	80%	0.1%	100%	0%	100%	0%	80%	0.1%
125-0.5-10-50-1.wtdp	90%	0.2%	100%	0%	100%	0%	100%	0%
125-0.5-10-50-2.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.5-10-50-3.wtdp	90%	0.3%	100%	0%	100%	0%	100%	0%
125-0.5-10-50-4.wtdp	**90%**	**0.0%**	70%	0.1%	40%	0.3%	30%	0.5%
125-0.5-10-50-5.wtdp	0%	0.5%	10%	0.4%	**20%**	**0.4%**	**20%**	0.5%
125-0.8-10-50-1.wtdp	**100%**	**0%**	100%	0%	100%	0%	100%	0%
125-0.8-10-50-2.wtdp	100%	0%	100%	0%	100%	0%	90%	0.1%
125-0.8-10-50-3.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.8-10-50-4.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.8-10-50-5.wtdp	0%	1.6%	50%	0.8%	**70%**	**0.3%**	30%	1.2%
125-0.2-25-25-1.wtdp	20%	1.8%	30%	1.6%	**80%**	**0.5%**	30%	1.6%
125-0.2-25-25-2.wtdp	**20%**	0.3%	**20%**	0.3%	10%	0.4%	10%	**0.3%**
125-0.2-25-25-3.wtdp	70%	0.4%	**90%**	**0.1%**	80%	0.2%	70%	0.6%
125-0.2-25-25-4.wtdp	40%	0.4%	60%	0.2%	**80%**	**0.1%**	70%	0.3%
125-0.2-25-25-5.wtdp	0%	0.8%	**10%**	**0.5%**	0%	0.6%	**10%**	0.7%
125-0.5-25-25-1.wtdp	**100%**	**0%**	100%	0%	100%	0%	100%	0%
125-0.5-25-25-2.wtdp	100%	0%	100%	0%	90%	0.1%	100%	0%
125-0.5-25-25-3.wtdp	90%	0.1%	100%	0%	100%	0%	100%	0%
125-0.5-25-25-4.wtdp	80%	0.1%	80%	0.1%	80%	0.1%	**90%**	**0.0%**
125-0.5-25-25-5.wtdp	90%	0.1%	90%	0.1%	80%	0.1%	**100%**	**0%**
125-0.8-25-25-1.wtdp	**100%**	**0%**	100%	0%	100%	0%	100%	0%
125-0.8-25-25-2.wtdp	60%	0.4%	30%	0.8%	30%	0.8%	**90%**	**0.1%**
125-0.8-25-25-3.wtdp	50%	2.2%	**100%**	**0%**	90%	0.2%	100%	0%
125-0.8-25-25-4.wtdp	20%	0.3%	40%	0.2%	**50%**	**0.2%**	**50%**	**0.2%**
125-0.8-25-25-5.wtdp	**20%**	1.3%	0%	**1.0%**	0%	**1.0%**	10%	1.1%
125-0.2-50-10-1.wtdp	**100%**	**0%**	100%	0%	100%	0%	100%	0%
125-0.2-50-10-2.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.2-50-10-3.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.2-50-10-4.wtdp	100%	0%	90%	0.0%	100%	0%	100%	0%
125-0.2-50-10-5.wtdp	100%	0%	100%	0%	100%	0%	90%	0.0%
125-0.5-50-10-1.wtdp	**100%**	**0%**	100%	0%	100%	0%	100%	0%
125-0.5-50-10-2.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.5-50-10-3.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.5-50-10-4.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.5-50-10-5.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.8-50-10-1.wtdp	80%	0.2%	80%	0.2%	**100%**	**0%**	100%	0%
125-0.8-50-10-2.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.8-50-10-3.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.8-50-10-4.wtdp	100%	0%	100%	0%	100%	0%	100%	0%
125-0.8-50-10-5.wtdp	60%	0.1%	**100%**	**0%**	100%	0%	100%	0%

Overview of Generative AI Tools Used

No generative AI tools were used in writing this thesis.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AEOP02]   Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002.

[AMS21]   Eduardo Álvarez-Miranda and Markus Sinnl. Exact and heuristic algorithms for the weighted total domination problem. *Computers & Operations Research*, 127:105–157, 2021.

[ASVR+22]   Paul Almasan, José Suárez-Varela, Krzysztof Rusek, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case. *Computer Communications*, 196:184–194, 2022.

[BDA13]   Mohamed Bekkar, Hassiba Djema, and T.A. Alitouche. Evaluation measures for models assessment over imbalanced data sets. *Journal of Information Engineering and Applications*, 3:27–38, 2013.

[DG06]   Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 233–240, New York, NY, USA, 2006. Association for Computing Machinery.

[DGZ22]   Lukas Dijkstra, Andrei Gagarin, and Vadim Zverovich. Weighted domination models and randomized heuristics. *arXiv preprint arXiv:2203.00799*, 2022.

[FK15]   Peter Flach and Meelis Kull. Precision-recall-gain curves: Pr analysis done right. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[GC19]   Liyu Gong and Qiang Cheng. Exploiting edge features for graph neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[GDJ+21]   Thomas Gaudelet, Ben Day, Arian R. Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy B. R. Hayter, Richard Vickers, Charles Roberts,

Jian Tang, David Roblin, Tom L. Blundell, Michael M. Bronstein, and Jake P. Taylor-King. Utilizing graph machine learning within drug discovery and development. *Briefings in Bioinformatics*, 22(6), 2021.

[GSR+17]   Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1263–1272. JMLR.org, 2017.

[Hen04]   Michael A. Henning. Restricted total domination in graphs. *Discrete Mathematics*, 289(1):25–44, 2004.

[HLW+21]   Shuli Hu, Huan Liu, Yupan Wang, Ruizhi Li, Minghao Yin, and Nan Yang. Towards efficient local search for the minimum total dominating set problem. *Applied Intelligence*, 51(12):8753–8767, 2021.

[Inn18]   Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018.

[ISF+18]   Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux. *CoRR*, abs/1811.01457, 2018.

[JCDLT13] László A. Jeni, Jeffrey F. Cohn, and Fernando De La Torre. Facing imbalanced data–recommendations for the use of performance metrics. In *2013 Humaine Association Conference on Affective Computing and Intelligent Interaction*, pages 245–251, 2013.

[JJL19]   Xiaodong Jiang, Pengsheng Ji, and Sheng Li. Censnet: Convolution with edge-node switching in graph neural networks. In *International Joint Conference on Artificial Intelligence*, 2019.

[JZLJ20]   Xiaodong Jiang, Ronghang Zhu, Sheng Li, and Pengsheng Ji. Co-embedding of nodes and edges with graph neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP:1–1, 2020.

[KB14]   Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[KKD23]   Stefan Kapunac, Aleksandar Kartelj, and Marko Djukanović. Variable neighborhood search for weighted total domination problem and its application in social network information spreading. *Applied Soft Computing*, 143:110–387, 2023.

[KvHW19]   Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

[LEF+22]   Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

[LK14]   Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[Loc21]   Carlo Lucibello and other contributors. GraphNeuralNetworks.jl: a geometric deep learning library for the Julia programming language. `https://github.com/CarloLucibello/GraphNeuralNetworks.jl`, 2021.

[LPHH84]   Renu Laskar, John Pfaff, S. M. Hedetniemi, and S. T. Hedetniemi. On the algorithmic complexity of total domination. *SIAM Journal on Algebraic Discrete Methods*, 5(3):420–425, 1984.

[LWL+21]   Ruizhi Li, Yupan Wang, Huan Liu, Ruiting Li, Shuli Hu, and Minghao Yin. A restart local search algorithm with tabu method for the minimum weighted connected dominating set problem. *Journal of the Operational Research Society*, pages 1–14, 2021.

[LWWC21]   Bohan Li, Kai Wang, Yiyuan Wang, and Shaowei Cai. Improving Local Search for Minimum Weighted Connected Dominating Set Problem by Inner-Layer Local Search. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[MCY19]   Yuede Ma, Qingqiong Cai, and Shunyu Yao. Integer linear programming models for the weighted total domination problem. *Applied Mathematics and Computation*, 358:146–150, 2019.

[NBG+20]   Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. Solving mixed integer programs using neural networks. *ArXiv*, abs/2012.13349, 2020.

[NLL+21]   Runbo Ni, Xueyan Li, Fangqi Li, Xiaofeng Gao, and Guihai Chen. Fastcover: An unsupervised learning framework for multi-hop influence maximization in social networks. *arXiv preprint arXiv:2111.00463*, 2021.

[RP06]   Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

[RR16]   Mauricio G.C. Resende and Celso C. Ribeiro. *Optimization by GRASP*. Springer, 2016.

[Sha98]   Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431. Springer Berlin Heidelberg, 1998.

[SNB+08] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93, 2008.

[VCC+18] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010. Curran Associates Inc., 2017.

[WCCY18] Yiyuan Wang, Shaowei Cai, Jiejiang Chen, and Minghao Yin. A fast local search algorithm for minimum weight dominating set problem on massive graphs. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1514–1522. International Joint Conferences on Artificial Intelligence Organization, 2018.

[WCY17]   Yiyuan Wang, Shaowei Cai, and Minghao Yin. Local search for minimum weight dominating set with two-level configuration checking and frequency based scoring function. *Journal of Artificial Intelligence Research*, 58:267–295, 2017.

[WNJ+22] Setyo Tri Windras Mara, Rachmadi Norcahyo, Panca Jodiawan, Luluk Lusiantoro, and Achmad Pratama Rifai. A survey of adaptive large neighborhood search algorithms and applications. *Computers & Operations Research*, 146:105–903, 2022.

[WPC+20] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.