Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Learning Graph Configuration Spaces in Engineering Domains

Michael Mittermaier

A dissertation submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy (PhD)
at the School of Computer Science and Statistics in June 2025

Supervisor
Dr. Goetz Botterweck

Examiners
Dr. Gilles Perrouin
Dr. David Gregg

Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the Library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

I consent to the examiner retaining a copy of the thesis beyond the examining period, should they so wish (EU GDPR May 2018).

Signed:  Date: 12 June 2025

Abstract

Background: The configuration process for feature-oriented product lines is well-studied for Boolean and numerical feature configurations. However, many engineering domains face the challenge of optimising *graph* structures to represent product configurations effectively.

Research Objective: Optimising complex graph structures to meet multiple objectives under various constraints demands a thorough understanding of the graph configuration space and the properties linked to each configured product. This thesis investigates two machine learning-based strategies to facilitate efficient exploration of graph configurations.

Method: Using two engineering case studies (design of HVAC systems and traffic networks), this thesis first evaluates the applicability of established methods from feature-oriented software product line engineering to graph configuration problems. After initial promising results in learning graph configuration spaces, experiments were conducted with metaheuristics —including local search, simulated annealing, and genetic algorithms— to explore and optimise within these spaces.

Results: The results demonstrate that graph embedding-based approaches lead to data-efficient learning. Additionally, learning-supported metaheuristics enable efficient exploration of graph configuration spaces and optimisation within those spaces, significantly reducing the need for costly evaluations of every graph configuration.

Conclusion: This work demonstrates that, with appropriate adaptations, strategies from software product line engineering can effectively support metaheuristic graph configuration space exploration. Future work is necessary to (i) fully leverage the potential of learning graph configuration spaces by improving sampling, learning, and evaluation strategies for enhanced training efficiency, prediction accuracy, and interpretability, and (ii) extend these methods to tasks such as dynamic configuration, constraint mining, and evolution.

Acknowledgements

First of all, I would like to thank Goetz Botterweck who offered me already in undergrad a first glimpse into research and afterwards accompanied me throughout my PhD journey as supervisor, mentor, and friend. I am also incredibly grateful for Takfarinas Saber, who was always available for great advice and directions whenever I felt lost in my research.

Ivana Dusparic and David Gregg have been diligent members of my thesis committee, I am thankful for them keeping track of my progress and making sure I was steering towards a successful outcome. I also want to thank Gilles Perrouin for examining my work and his valuable suggestions.

During this program, I have received continuous support whenever I needed it from the institutions around me, namely the School of Computer Science and Statistics here at Trinity College, but also before at the Lero research centre in Limerick. I am glad to be part of the Lero community.

I was blessed to have made great friends in my time in Ireland who always lent an ear whenever I needed help, many thanks in particular to Anila Mjeda, Farshad Toosi, and Alexander Schieweck for all the valuable comments in the office, covid-safe meet-ups during the pandemic, and occasional pints in pubs or Thomond Park.

I owe nothing but gratitude to all my friends and family back home. To my parents Ute and Ernst, my sister Sarah, to my grandfather Karl-Heinz, who never got the chance to read this, but would have been proud, and, of course, to my wife Kate, who unconditionally supported me throughout this journey, and will continue to do so for many more.

Ar deireadh, ba mhaith liom buíochas a ghabháil le muintir na hÉireann, a chuir céad míle fáilte romham. Táim an-bhuíoch as an tír seo as ucht gach deis a tugadh dom anseo.

Go raibh míle maith agaibh!

Contents

List of Figures

List of Tables

List of Acronyms

AC	Air Conditioning
CART	Classification and Regression Technique
DLR	German Aerospace Center
FGSD	Family of Graph Spectral Distances
GA	Genetic Algorithm
GAT	Graph Attention Network
GCN	Graph Convolutional Neural Network
GE	Graph Embedding
GL2Vec	Graph and Line Graph to Vector
GNN	Graph Neural Network
Graph2Vec	Graph to Vector
GraphSAGE	Graph Sample and Aggregate
HC	Hill Climbing
HGB	Histogram-based Gradient Boosting
HVAC	Heating, Ventilation, and Air Conditioning
IHC	Iterative Hill Climbing
JSON	JavaScript Object Notation
kNN	k Nearest Neighbours
LDP	Local Degree Profile-Based embedding
LEGCS	Learning Graph Configuration Spaces
MAE	Mean Absolute Error
MATLAB	Matrix Laboratory
ML	Machine Learning
MLP	Multi-Layer Perceptron
MOO	Multi-Objective Optimisation
MSE	Mean Squared Error
NetLSD	Network Laplacian Spectral Descriptor
NFP	Non-functional Property
NSGA-II	Non-Dominated Sorting Genetic Algorithm II
r^2	Coefficient of Determination
RF	Random Forest
RMSE	Rooted Mean Squared Error
RQ	Research Question
SA	Simulated Annealing
SF	Spectral Features
SOO	Single-Objective Optimisation
SPLE	Software Product Line Engineering
SUMO	Simulation of Urban Mobility
SVR	Support Vector Regression
TS	Tabu Search

Chapter 1

Introduction

1.1 Motivation

In various engineering fields, we encounter the problem of configuring a system. Configurations define systems and indirectly determine their properties. Engineers explore large spaces of possible configurations while simultaneously considering various constraints and numerous optimisation objectives to find a suitable configuration. Configuring a Linux kernel [4], optimising a building ventilation system [5], planning a public transport network [6]; all these problems involve the configuration of a system with an incomprehensibly large configuration space. Choosing the wrong configurations can lead to products with undesirable properties or products that break the system's constraints. A common goal is not to identify the best configuration as fast as possible, but rather to find a useful set of very good configurations in a given time for further analysis and elaboration by human engineers. Thus, there is a need for techniques (and corresponding tool support) that help users handle the complexity of the configuration process and narrow it down to humanly comprehensible sets of solutions to explore in more detail.

A common approach in software product line engineering (SPLE) is to represent configuration choices and variability among products within a family of software-intensive systems. In the configuration process, we make, for instance, *Boolean* or *numerical* configuration decisions within the options and constraints defined in the feature model to obtain a product configuration describing a particular product. Pereira et al. conducted a systematic literature review on learning configuration spaces and provide a taxonomy of state-of-the-art strategies and methods applied to software configuration spaces [1]. The considered techniques interpret configurations as a set of Boolean decisions and numerical parameters, which can be represented as a Euclidean vector.

However, in several engineering fields, it is more amenable to the particular domain to use configurations (in the wider sense) in *structural* or *topological* form. Examples of such problems are arranging the components of a software architecture to improve non-functional properties of the software system, sequencing the processing steps in a manufacturing plant such that costs and production time are minimised, or arranging rooms in a house to minimise energy consumption. In these graph configuration problems, we have to find a graph that represents (the configuration of) the product with the best resulting

properties while simultaneously considering various constraints. This potentially requires systematic exploration of 10,000s of graphs to narrow down the configuration space of the product to a humanly comprehensible number of candidates. I envision a usage scenario where such an identified small set of "best" candidates can be input to a review by human experts, followed by further configuration or design steps.

In industrial practice and when dealing with problems of realistic scale, the optimisation of graph configurations is often done in an iterative and heuristic fashion, thus only exploring a small part of the configuration space that is only as good as the knowledge of the expert conducting the search. This is due to the complexity and arbitrary structure of graphs that raise a variety of challenges when it comes to exploration, including scalability [7], feature engineering [8], heterogeneity [9], and interpretability [10]. Hence, there is a need for advanced exploration tools to efficiently explore large graph configuration spaces.

Furthermore, solely focusing on finding the optimal solution is often short-sighted as (i) not all data/simulations are error-free, (ii) many models are often simplified versions of the real problem which leave out details that are difficult to express, and (iii) some preferences might not have been elucidated by the decision-maker – thus an optimal solution might prove inferior or impractical [11]. Therefore, I do not only want to find the best solution. Instead, I need to search for a selection of k best (i.e., optimal or near-optimal) solutions. These k best solutions create a smaller configuration space that is humanly assessable for further constraints, requirements, or optimisation goals.

1.2 Scope

Within this thesis, I define and explore the approach of learning graph configuration spaces with applications in graph configuration exploration and optimisation. In these applications, graphs and the graph space are defined by:

1. nodes and edges with their attributes and weights,
2. domain-specific constraints limiting the valid graph space, and
3. the fitness of each graph that is computationally expensive to assess.

That results in two major challenges for my approach: (1) predicting graph fitnesses that are otherwise expensive to obtain, and (2) use these assessments to explore and optimise in graph configuration spaces while considering the domain-specific constraints.

Below, I further explain details of both challenges, and briefly introduce related problems and approaches to the one addressed in this thesis.

Graph Fitness Prediction

For clarification, the fitness of a graph in context of this thesis does not refer to purely topological properties such as connectivity, the number of cycles, or shortest paths. For these problems, the literature already provides concrete algorithms (e.g., depth-first search [12] and Dijkstra [13]) that determine those properties in specific complexities. The fitness also does not refer to local properties within the graph that are used for node classification or link prediction. Instead, the graph fitness in this thesis addresses complex, global, and domain-specific properties that require expensive evaluations of graph structures, for instance the runtime of an air conditioning in a house ventilation system in Case Study I and the average travelling time of a car in a road network in Case Study II (both case studies are presented in Section 3.5).

These are just two examples of a graph fitness that indicates how well a graph configuration performs solving a domain-specific problem (such as connecting rooms in an energy-efficient ventilation system or towns in an efficient road network). Both examples are interesting for the research presented in this work because assessing these fitnesses requires computationally expensive simulations that the approach of learning graph configuration

spaces aims to replace with predictions.

Graph Exploration and Optimisation

The problem of finding good graph structures itself is combinatorial. However, in this approach, we can also have a continuous component: the node and edge attributes. This means that the approach presented in this thesis addresses both combinatorial and mixed problems.

Related Problems and Approaches

In the travelling salesman problem (TSP), we need to find the best path between a set of cities so that our salesman travels the shortest distance while visiting each city exactly once before returning to their origin city. At the first glance this problem seems relevant for this thesis: we aim to (1) identify one graph (the salesman's path) out of (2) a space of possible graphs (every path in the complete graph between these cities where we visit each city once before returning to the origin city), while we (3) optimise a global graph property (the overall distance travelled). However, obtaining the graph fitness in this problem is so trivial that heuristic algorithms focused on combinatorial problems are more suitable for this class of problems [14].

Similarly, deterministic optimisation approaches such as integer linear programming (ILP) do not apply for problems included in this work due to the fitness assessments. Although we can solve some graph tasks using ILP [15], and we could model graph constraints in context of this work as linear, I explore problem domains where we cannot calculate the graph properties using linear functions, thus these optimisation strategies do not apply.

1.3 Structure

This section gives an overview of how I structured the body of this thesis. I start with Figure 1.1 visualising the chapters in this thesis. The black boxes in this figure show the chapter numbers, chapters in red are focused on learning, exploration in blue, and optimisation in green. Clouds represent concepts and techniques on which I base my work. Below, I explain this structure based on research questions (RQ) addressed in this thesis, and I close this section with the list of submitted or published work that I conducted during the period of registration as a PhD candidate at Trinity College in Table 1.1.

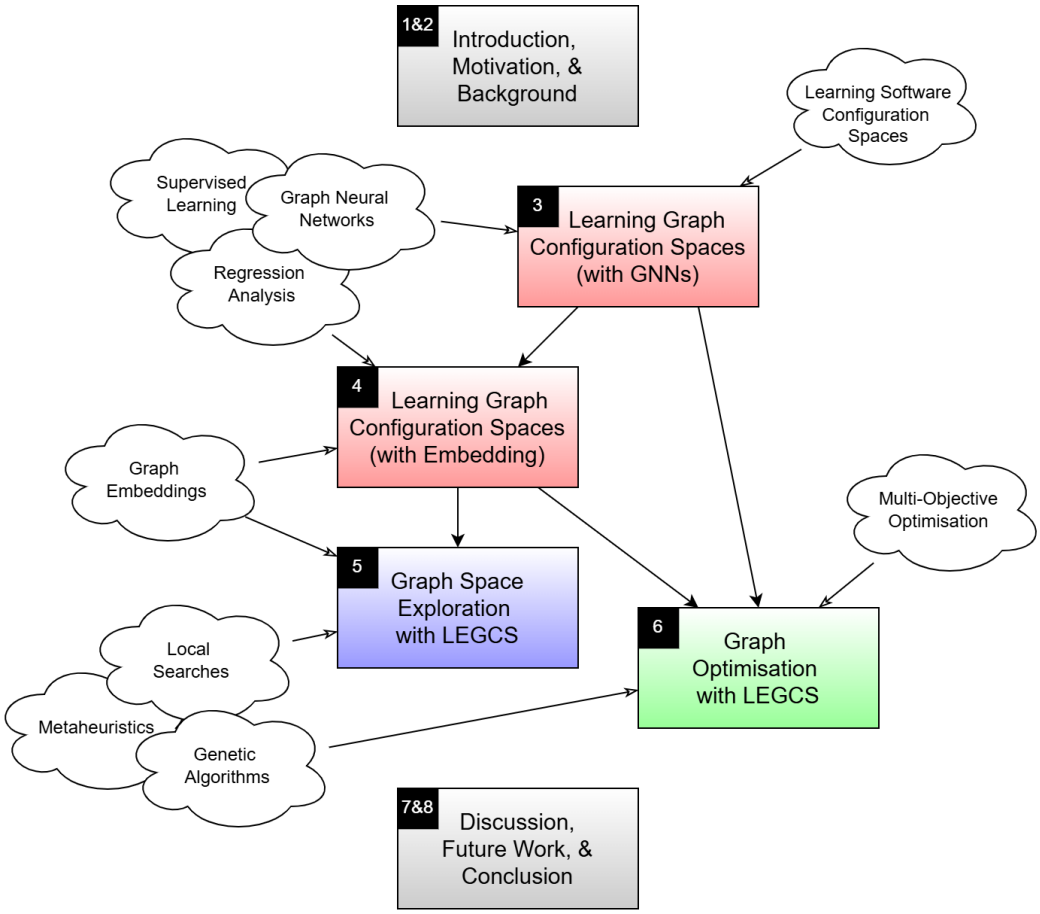


Figure 1.1: Structure of the chapters in this thesis.

Chapter 2: Background and Related Work

After this introductory chapter, I start with the background and related work where I introduce the concepts and techniques shown in the clouds in Figure 1.1.

These concepts include topics of software product line engineering and learning con-

figuration spaces, regression analysis on graphs, metaheuristics and multi-objective optimisation, and related work on search-based graph optimisation, metaheuristics on graphs, machine learning for metaheuristics, and road network optimisation.

Chapter 3: Learning Graph Configuration Spaces

In Chapter 3, I define and explore the problem of learning *graph* configuration spaces (LEGCS), transforming this learning problem from Euclidean structures to graphs. This leads to the feasibility question:

(RQ 3.1) How can we apply configuration space learning methods established for Euclidean configuration spaces to graph configuration spaces?

In other words, I aim to adapt the framework proposed in [1] and "translate" it to graph configurations to identify a combination of approaches that demonstrates feasibility of this adaption. After proposing a pipeline, I examine the efficiency by applying this pipeline in two case studies to answer the question:

(RQ 3.2) How does tool-supported learning of graph configuration spaces compare to a random search approach with respect to time consumption and quality of determined graph configurations?

Chapter 4: Embedding-Based LEGCS

In Chapter 4, I propose a variation of LEGCS that replaces the prediction model based on graph neural networks (GNN) with a two-step model based on graph embeddings and regression analysis. Replacing deep learning could help us make training data-efficient predictions. First, I investigate the efficiency and limitations of this approach by observing the influence of various embedding and learning strategies on the quality of the predictions in the context of LEGCS to answer:

(RQ 4.1) What is the impact of graph embedding and regression algorithms on learning-based graph space exploration in terms of prediction accuracy?

Subsequently, I compare how an embedding-based approach compares to the GNN approach I introduced in the previous chapter to answer:

(RQ 4.2) How does learning based on GNNs compare to regression techniques based on graph embeddings with respect to accuracy of the predictions and the size of the required training set in the

context of learning graph configuration spaces?

Chapter 5: Graph Space Exploration with LEGCS

Chapter 5 is concerned with graph space exploration in HVAC systems (Case Study I) to find the k graphs with the best properties while staying within a simulation budget. First, I apply metaheuristics on graph embeddings to answer:

(RQ 5.1) How can we apply metaheuristics to find as many of the k best graphs as possible in a configuration space using only a limited number of measurements?

Then, I apply LEGCS to the same problem with the insights gained in the previous chapter:

(RQ 5.2) How efficiently can we identify k best graphs with LEGCS using a limited number of measurements?

Eventually, I tie metaheuristics and LEGCS together to determine whether this method is more efficient in terms of identifying the k best graphs while staying within the simulation budget:

(RQ 5.3) How does (1) a learning-supported metaheuristic search for the k best graphs in a graph configuration space compare with (2) a metaheuristic search on embeddings and (3) a LEGCS search on a limited number of measurements in terms of identified graphs?

Chapter 6: Graph Optimisation with LEGCS

In Chapter 6, I optimise traffic networks (Case Study II) with a genetic algorithm. In a controlled experiment, I answer whether I can improve the optimisation outcome by applying LEGCS to support a genetic algorithm:

(RQ 6) How does a LEGCS-supported genetic algorithm compare to a plain genetic algorithm (without LEGCS support) when applied to a multi-objective traffic network optimisation problem with respect to the obtained Pareto fronts?

Chapters 7 and 8: Discussion, Future Work, and Conclusions

The thesis closes with threats to validity and future work in Chapter 7 and a conclusion in Chapter 8.

Paper	Reference
Michael Mittermaier, Takfarinas Saber, and Goetz Botterweck: *Applying Graph Neural Networks to Learn Graph Configuration Spaces.* The 19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS), 2025. DOI: 10.1145/3715340.3715444	[16]
Michael Mittermaier, Takfarinas Saber, and Goetz Botterweck: *Learning Graph Configuration Spaces with Graph Embedding in Engineering Domains.* The 9th International Conference on Machine Learning, Optimization, and Data Science (LOD), 2023. DOI: 10.1007/978-3-031-53966-4_25	[17]
Michael Mittermaier, Takfarinas Saber, and Goetz Botterweck: *Learning Graph Configuration Spaces to Support Road Network Design Optimisation.* The Genetic and Evolutionary Computation Conference (GECCO), 2025. DOI: 10.1145/3712256.3726447	[18]

Table 1.1: Published or submitted work that I conducted during the period of registration as a PhD candidate at Trinity College.

1.4 Contributions

In this section, I summarise the contributions of my thesis, organised by chapter:

Chapter 3: Learning Graph Configuration Spaces

1. I propose a pipeline of adapted techniques from learning software configuration spaces to demonstrate the feasibility / applicability of these techniques for learning *graph* configuration spaces (answering *RQ 3.1*).

2. I report on controlled experiments that explore (1) a dataset of graph configurations describing house ventilation systems by AC consumption with as few MATLAB Simulink simulations as possible and (2) a dataset of graph configurations describing traffic networks by average travel time with as few SUMO simulations as possible. In these experiments, I compare the pipeline proposed to answer *RQ 3.1* against a random search approach. Thus, providing further evidence of the feasibility of the approach (answering *RQ 3.2*).

Chapter 4: Embedding-Based LEGCS

3. I propose a variation of LEGCS based on graph embeddings that applies regression analysis on vectors representing graph configurations to facilitate a training-efficient prediction model.

4. I provide an in-depth study on the impact of graph embeddings and regression techniques on the prediction accuracy within learning graph configuration spaces (answering *RQ 4.1*).

5. I report on a controlled experiment that explores the efficiency of regression analysis with (a) the GNN-based approach and (b) the embedding-based approach with respect to accuracy and training data size (answering *RQ 4.2*).

Chapter 5: Graph Space Exploration with LEGCS

6. I propose leveraging graph embeddings to enable the use of local search for efficient graph space exploration (answering *RQ 5.1*).

7. I investigate the application of LEGCS from prediction in the previous chapter now to exploration within a simulation budget (answering *RQ 5.2*).

8. I devise a graph space exploration approach that combines the search power of meta-heuristics with the predictive efficiency of machine learning in a case study (answering *RQ 5.3*).

Chapter 6: Graph Optimisation with LEGCS

9. I propose a pipeline of supporting genetic algorithms with LEGCS to apply in a traffic network design problem.

10. I report on a controlled experiment that optimises a traffic network to minimise travelling time as well as the overall length of roads. In this experiment, I compare a genetic algorithm with and without LEGCS support (answering *RQ 6*).

As a general practice,

11. I make all generated and measured data publicly available in an online repository to encourage follow-up work and replicability (see Section 3.5).

Chapter 2

Background and Related Work

2.1 Software Product Line Engineering

In this section, I first outline a conceptual framework for feature-oriented software product line engineering (SPLE). After that, I describe the challenge of learning a model representing the whole configuration space (capturing which product properties result from feature configurations).

Corresponding to this, I introduce a graph-oriented framework for product line engineering, which has a similar logical structure to its feature-oriented counterpart but uses graphs to represent configurations. Then, analogously, in the following chapter, I describe the challenge of learning a model of the *graph*-based configuration space (capturing which product properties result from given graphs).

2.1.1 Feature-Oriented Product Line Engineering

I am mainly interested in the general principle of representing configuration choices in SPLE and then configuring and deriving products [19, 20]. For the sake of the discussion here, I focus on feature-oriented approaches, but I acknowledge that there are other related approaches, e.g. decision modelling [21] or other forms of variability modelling.

To structure the discussion, I summarise main concepts of feature-oriented (software) product line engineering in the framework shown in Figure 2.1a. It covers two levels (domain engineering and application engineering) and three aspects (configuration, implementation, and properties). This is based on common SPLE approaches [19, 20], which often focus on the configuration and implementation aspects, plus extensions that consider resulting product properties [1, 22, 23].

Configuration. In a product line, we retrieve a feature model from product descriptions to represent all configuration choices [24]. We use this model in the ❶ configuration process to retrieve a feature configuration (or similar artefact) representing one product [25].

Implementation. Since the feature model is mapped to implementation assets, we can use this mapping to ❷ derive the implementation of a particular product from a feature configuration. There are many different ways that we can represent the effect of configuration decisions in the implementation. Ideally, this product derivation is automated or at least tool-supported.

Product Properties. Every implementation of a product has measurable properties. Once we have a meaningful number of implemented products and associated properties, we can aim to ❸ learn a model that captures knowledge about product properties for all products. This model can support the configuration process by estimating product properties prior to often costly implementations and measurements [22, 23].

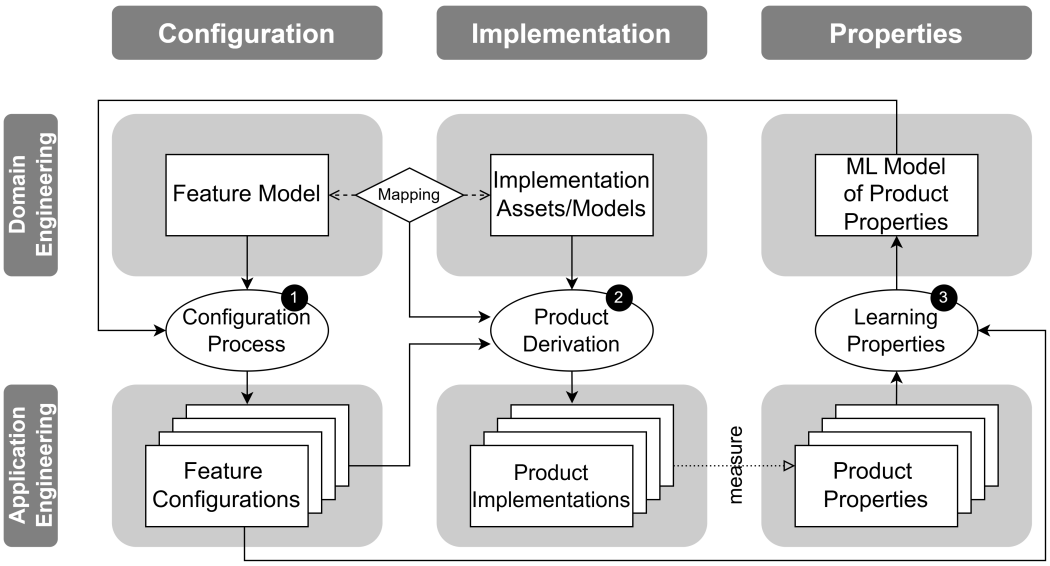
2.1.2 Graph-Oriented Product Line Engineering

In my adaptation to graph-oriented product lines, I keep the framework structure with a domain engineering level describing the whole product line, and the application engineering level describing individual products, as well as the three aspects of configuration, implementation, and properties (see Figure 2.1b).

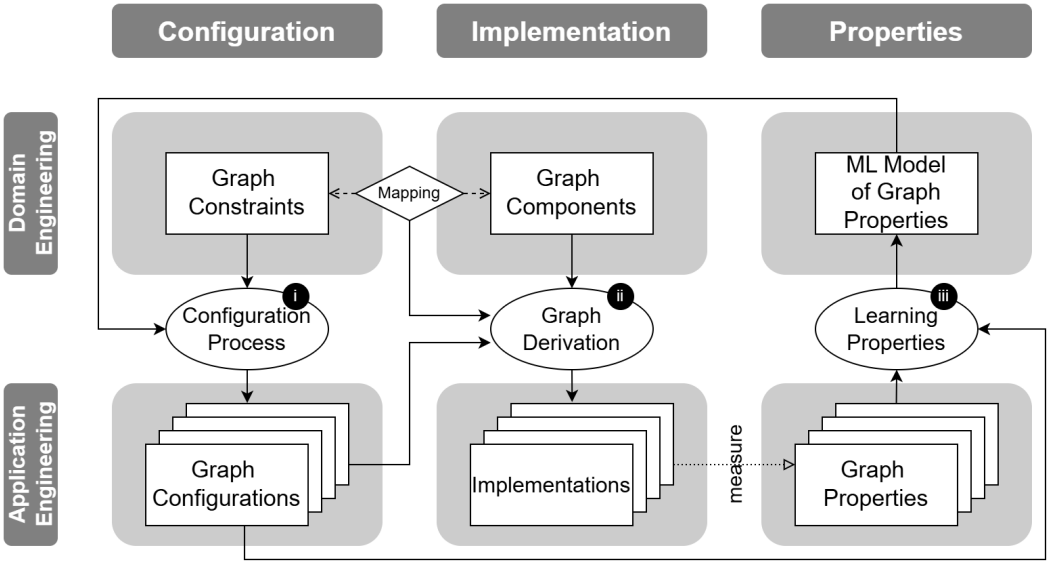
Configuration. Instead of a feature model representing the configuration choices, we now have a set of graph constraints directly mapped to the domain-specific graph components limiting the number of valid graph configurations in the graph space. We use these constraints to ❶ retrieve a graph configuration describing one product.

Implementation. Similarly to the feature-oriented approach, we can now ❷ derive a product implementation from the graph configuration and the graph components and measure the product properties.

Product Properties. We can then use these product properties to ❸ train a graph-based learning model to predict product properties for all valid graph configurations in order to support the configuration process.



(a) Learning-supported product configuration in software product line engineering.



(b) Adaptation of software product line configuration to learning-supported graph configuration.

Figure 2.1: Learning-supported product configuration processes.

2.2 Learning Software Configuration Spaces

Configuring large systems requires understanding the space of possible configurations, such that we can optimise the system towards several goals while simultaneously fulfilling multiple constraints. For instance, configuring the Linux kernel with more than 15,000 configuration options leads to at least 2^{15000} possible variants of the kernel [4], more candidates than there are atoms in the known universe. Although this example is extreme and the number of valid configurations would be reduced by constraints, it shows how the configuration space can become incomprehensibly large. Hence, for systems of realistic size and complexity, there is a need to systematically *learn* the configuration space, e.g., to make predictions on properties of a previously-unseen configuration. Below, I give an introduction to common practices for learning product properties from product configurations (❸ in Figure 2.1a) to aid the configuration process.

There are established practices in the process of learning software configuration spaces [1]. These practices consist of four phases to explore the software configuration space: sampling, measuring, learning, and evaluating (sometimes also referred to as the validation phase) as shown in Figure 2.2. Below is a short description and Table 2.1 presents strategies for each phase.

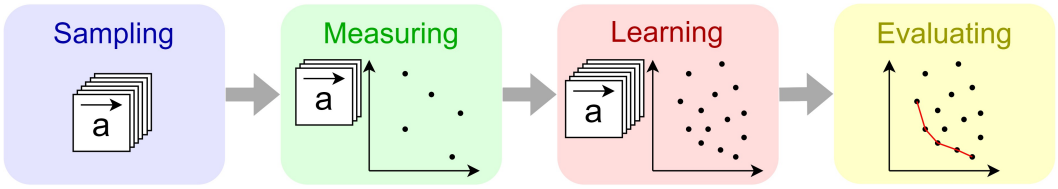


Figure 2.2: Phases of learning software configuration spaces.

1. *Sampling* – The sampling phase includes multiple challenges: (1) selecting a number of valid configurations (satisfiability problem/constraint satisfaction problem [26]), and (2) labelling a representative sample for the configuration space, so that a learning system in the third phase can generalise.

2. *Measuring* – In the measuring phase, we determine non-functional properties (NFP) of the labelled samples. There are different definitions of what a NFP actually is [27, 28]. In this work, I refer to attributes that describe the performance of a configuration (some of them are not directly derivable from the configuration). These attributes are

domain-dependent; for example, in the case study introduced in Section 3.5.1, I assess the time of air condition (AC) usage per month in a house ventilation system, and in Case Study II, I measure the average travel time for cars in the network.

3. *Learning* – Measuring all software configurations sampled in the first phase is technically possible but cost-intensive. Hence, a machine learning system is trained in this phase with the results of the measuring phase to predict measurements for the not-assessed configurations in the set.

4. *Evaluating/Validation* – Validating the results of the learning phase means evaluating the quality of the predictions. Ideally, the prediction error and measurement effort are low. The results of this final phase depend on the application of the exploration process. These applications may be pure prediction, interpretability, optimisation, mining constraints, or evolution.

Strategy	Description
	Sampling
Random sampling	Using a random sample to train the machine learning model. There are various strategies and rules of thumb to determine the number of configurations depending on the number of features.
Heuristics	The general motivation to use heuristics is to cover features and their interactions, thus capturing the essence of the configuration space with a smaller sampling size.
Transfer learning	This strategy progressively learns interesting regions in the configuration space by exploiting common similarities.
Arbitrarily chosen sampling	In some instances, domain experts choose a sample from currently available resources.
No sampling	For smaller configuration spaces, it is a valid strategy to explore the whole space and not select a sample in the space.
	Measuring
Execution	Configure the system accordingly, execute the system, and measure the NFPs.
Simulation	Configure the system accordingly, and measure the NFPs during a simulation.
Static analysis	Determine the NFPs by examining code, models, or documentation.
User feedback	Gather NFPs by collecting user feedback.
Synthetic analysis	Combine multiple metrics and measuring methods to assess various features.
	Learning
CART regression	The classification and regression technique partitions a sample into smaller clusters and uses regression techniques on these clusters.
Performance-influence models	This combination of sampling heuristics and step-wise linear regression selects relevant features and determines their influence on the configuration performance.
Other learning algorithms for regression	These strategies use a set of samples to train a regression model until the accuracy is satisfactory.
Learning to rank	In some instances, it is useful to rank configurations instead of predicting exact numerical values.
Transfer learning	Instead of training a machine learning model from scratch, this technique reuses already available knowledge from relevant sources.
	Evaluating
Accuracy metrics	There are plenty of accuracy metrics (e.g., the mean absolute error) which compare the predicted values to exact measurements.
Interpretability metrics	When we use transfer learning techniques, interpretability metrics are a tool to better understand the configuration space.
Sampling cost	One optimisation goal is to keep the sample size low because reducing measuring efforts saves resources.

Table 2.1: Strategies used in the four phases of learning software configuration spaces, according to Pereira et al. [1].

2.3 Regression Analysis on Graphs

In this research, I understand a whole graph as the configuration of a system and a graph set as design options for this system. The main challenge for adapting strategies of learning software configurations to graphs is finding a suitable learning model making efficient and accurate predictions of NFPs.

There are plenty of graph-oriented ML approaches that focus on *sub*graphs and elements of a graph. Makarov et al. calls node classification, link prediction, clustering, and graph visualisation the "four classic machine learning problems on graphs" [29]. Zhang et al. list more domain-specific applications of ML on graphs including modelling influence in large social networks, traffic network forecasting, and natural language processing [30]. In all of these applications, we use approaches that model graphs to make predictions for nodes, links, or subgraphs. Such approaches are not relevant for my research questions.

Instead, I require approaches that consider a *whole graph*, can learn and use models of *sets* of graphs, and can deal with properties resulting from a graph's structure. The regression methods that are relevant to my particular kind of problem follow one of these two paths [31, 32]:

1. Graph neural networks: Interpret the graph itself as a neural network, let the nodes learn their neighbourhoods for a number of layers, and use global pooling to determine the result [33].

2. Network embedding: Map graphs into low-dimensional vectors (graph embedding) and use regression analysis techniques.

In this section, I give a brief introduction to these two paths, that I further explore in case studies in Chapters 3 and 4.

2.3.1 Graph Neural Networks

GNNs are used for various tasks in several fields of science and engineering. Those tasks include classifying nodes [34], subgraphs [35], or graphs [36], link predictions [37], colour graphs [38], or in case of this work predicting non-functional properties of graphs. Zhou et al. [33] present an overview over current practices and applications of GNNs. While I focus

on predicting properties of graphs in an engineering context, other applications for predicting non-functional properties include: predict chemical reaction products (with molecules as graphs) in organic chemistry [39], predict protein interfaces (with proteins as graphs) in biology [40], predict software defects in software engineering [41], or predict traffic states (with street networks as graphs) in applied mathematics [42]. Even predicting connections, e.g., in social networks [43] can also be interpreted as GNNs predicting non-functional properties.

The fundamental concept of GNNs is treating a graph itself as a neural network which allows direct processing and manipulation of graph-structured data. This way, GNNs can capture intricate relationships and dependencies among graph elements, and enable expressive and context-aware learning models.

Depending on the graphs we are operating on and the task that the GNN should perform, we need to make various decisions during the design process. You et al. describe important design decisions for building GNNs in [44]:

1. *Message passing*, where we determine which characteristics of each node we pass on to its neighbours.

2. *Aggregation*, where we calculate the feature vector of a node in the next GNN layer out of its current feature vector and the neighbour's messages.

3. *Layer connectivity*, where we define how GNN layers are interconnected. Usually, we sequentially calculate the new feature vector of each node in every GNN layer. That could lead to over-smoothing where the feature vectors of all nodes become too similar. Solutions here include (1) a low number of layers and expressive aggregation, for instance, by aggregating with a deep neural network, or (2) add layers without message passing (pre- and post-processing layers).

4. *Graph manipulation*, where we improve learning results by modifying the input graph to a computational graph. Reasons for modifications include: a lack of features in the input graph, a sparse input graph (inefficient message passing), a dense input graph (costly message passing), and a large input graph (high computational complexity). Solutions include feature augmentation, adding virtual nodes and edges, sample neighbours while passing messages, and sampling subgraphs to compute em-

beddings.

5. *Learning objective*, where we decide whether the objective is on the node, edge, or whole-graph level.

In terms of regression, the standard method to go from here, especially for small graphs, is global pooling [45]. Here, we pool the feature vectors of all nodes that were calculated during the learning process of the GNN together, and calculate the regression result by, for instance, picking the maximum/minimum/mean vector or summing these vectors together [46].

2.3.2 Embedding-Based Regression

This method involves two phases. In the first phase, we embed topological information of graphs into vectors and, in the second, we can apply traditional regression models on those vectors representing graphs. Below, I describe both phases individually.

Graph Embedding

In graph embedding, we map graph (or subgraph-) structures to Euclidean data structures (vectors or matrices) that preserve structural information as well as possible [47]. Graph embedding is applied in various graph analysis tasks including node classification, link prediction, or clustering, and requires different kinds of embeddings for these tasks. Embedding single nodes or edges is especially helpful for operations on large graphs, e.g., we can classify nodes in a large graph by mapping nodes with similar neighbours to vectors with close Euclidean distances, and use these node embeddings for further analysis.

In the context of this thesis, however, I mainly use graph embeddings as similarity measurements between whole graphs (and not their parts), either as input for regression models in the context of ML (in Chapter 4), or as a basis for metaheuristic local searches to have a distance metric between graphs (in Chapter 5). Hence, I focus on whole-graph embedding rather than node, edge, or substructure embedding.

Cai et al. [2] point out the key challenge of whole-graph embedding: choosing between the expressiveness of the embedding (preserve information) and the efficiency (embedding time/low dimensionality). In the same work, they also provide a taxonomy of graph embedding techniques that I briefly describe in Table 2.2.

Family	Description
Matrix factorisation	A structure-preserving dimensionality reduction that factorises a high-dimension matrix representing graph properties such as pairwise node similarity into node embeddings that places nodes in Euclidean space at a distance according to their pairwise similarity. Examples include graph Laplacian eigenmaps and node proximity matrix factorisation.
Deep learning	There are two types of deep learning based embedding techniques. In the first one, the deep learning model is fed with random walks over the graph, and in the second type, the deep learning model is trained by the whole graph, with various strategies of breaking down the graph structure (autoencoders, mixture model networks, graph neural networks, etc.).
Edge reconstruction	This embedding technique aims to optimise one of the following three objective functions: (1) maximise the edge reconstruction probability, (2) minimise the edge reconstruction distance-based loss, or (3) minimise the edge reconstruction margin-based ranking loss.
Graph kernels	These calculate the inner product of two graphs to measure their similarity. First applied in chemistry, graph kernels now find applications in biology, neuroscience, and natural language processing. Kriege et al. [48] classified graph kernel techniques into approaches based on: neighbourhood aggregation, assignment and matching, subgraph pattern, walks and paths, or others.
Generative model	This approach embeds nodes as vectors of latent variables, thus viewing the observed graph as model-generated. Generally, this method is used for node and/or edge embedding for heterogeneous graphs.
Hybrid Techniques and Others	In this category are either techniques that use ideas from multiple other categories or use other graph characteristics, e.g., assessing the distance to a prototype graph, set landmark nodes and measure distances to these nodes, or approximating higher-order proximity matrices.

Table 2.2: Families of graph embedding techniques for whole-graph embedding [2].

Regression Analysis

Regression analysis refers to a family of statistical practices that allows us to evaluate and quantify the relationship between a dependent variable and one or more independent variables [49]. In supervised machine learing, we use regression analysis to model and predict continuous numerical values. Linear regression, for instance, is one of the most common and perhaps most comprehensive techniques in supervised machine learning. In Table 2.3, I give an overview of families of regression techniques according to the taxonomy from [3].

Family	Description
Linear regression techniques	For *linear regression*, we assume that there is a linear connection between the input vector $\vec{x}$, and the prediction value y [49]. The goal is to find parameters for β so that the prediction error for $y = \beta_0 + \sum \beta_i x_i$ is minimal (similarly polynomial regression [50] exists for a polynomial connection between input vector and prediction value $y = \beta_0 + \sum \beta_i x_i^j$). Including more input values into the calculation can lead to problems with an overfitting function or correlated input values. Popular approaches to simplify the prediction function (and thus reduce errors) are *ridge regression* [51], which penalises high parameters β, or the *lasso regression* [52] that takes unnecessary variables out of the equation by setting some parameters β to zero. *Generalised linear regression* techniques use a mixture of techniques like ridge or lasso that use penalisation on top of a linear regression approach.
Graph-based regression	*Regression trees* such as decision trees [53] are an approach to approximate values based on graphs. This technique breaks the input data up into smaller data sets. A decision node within the decision tree links to at least two more nodes, each link representing one value in the data set. Each leaf node represents one prediction result. To avoid overfitting, the technique of *random forests* [54] uses the same input for multiple decision trees and calculates the prediction result out of the results of the individual decision trees. Another way to improve predictions from trees, is translating the tree into a *regression rules* set, and tuning hyperparameters, or include other regression approaches into some rules.
Deep learning-based regression	*Neural networks* [55] prediction uses graphs of "neurons". These neurons are organised in layers, one input layer, one output layer, and a number of hidden layers between them. Each neuron is linked to each neuron in the next layer. Each node has its linear regression model making predictions on input data with its weights, bias, and output. This output influences the input of the neurons in the next layer. This mimicking of the human brain leads to precise results that are hard to interpret. Next to classical neural networks, there are other *deep learning* strategies used for regression, such as recurrent neural networks or deep belief networks.
Ensemble learning	Ensemble learning techniques are popular ways to improve the performance of regression models. *Bagging* reduces variance by applying multiple regression models on different subsets of the training data, and *boosting* reduces bias by sequentially training regression models while emphasising training data that were misclassified in the previous iteration. *Additive models* use the sum of smooth functions estimated from the data using penalised regression, generalised additive models, or smoothing splines.
Other regression techniques	There is a large variety for regression models. Here, I list more popular methods that are not part of the families mentioned above. In *least squares*, we minimise the sum of squares of the offsets of the points from the learning function curve. *Projection methods* project data points on a subspace to find the best-fitting regression model. *Bayesian models* use Bayes rule to update probabilities. With more observations (training data), the probabilities for predictions are updated to be more precise. *Quantile regression* is more robust to outliers than linear approaches because it trains models in different quantiles (areas within the dataset) instead of the whole dataset. The *nearest neighbours* technique [56] assumes that input data within short distances have similar outputs. This algorithm takes the k nearest neighbours of input data in the training data and makes the prediction based on averages of these neighbour outputs. If there is not much training data available, the *Gaussian process* [57] offers a method based on the Gaussian distribution between the points in the training data. A support vector machine introduces a support vector to find a hyperplane that separates classes in another dimension. This separator can be mapped back to the original space, but unlike linear regression, it does not have to be linear. This idea can also be used for *support vector regression* [58] by finding a hyperplane that includes the maximum number of points in the training data.

Table 2.3: Families of regression analysis techniques [3].

2.4 Metaheuristics

Exact exploration and optimization algorithms become impractical in scenarios with high-dimensional search spaces. When the search space grows exponentially with the problem size, there is a need for metaheuristic algorithms [59]. Metaheuristic algorithms solve a vast number of optimisation problems without the need to adapt to each problem. Instead, these algorithms operate on a "higher level" as opposed to problem-specific heuristics.

We differentiate between single-solution-based algorithms such as simulated annealing, Tabu search, and variable neighbourhood search and population-based algorithms that are based on evolutionary concepts or swarm intelligence [60]. In this section, I present the single solution-based techniques that I use for exploration tasks on solution sets (in Chapter 5), and one population-based technique that I use for multi-objective optimisation (in Chapter 6).

2.4.1 Single Solution-Based Metaheuristics

This class of metaheuristics are also called trajectory methods. The main idea here is evaluating single solutions and moving away towards better solutions in the optimisation process. Boussaïd et al. [60] identify the main algorithms for this class of metaheuristics, I introduce a selection of those below:

Hill Climbing (HC)

This straightforward algorithm starts with an arbitrary solution for our problem. We modify this solution until we find a better-performing solution, and then we start operating on this new, better solution. We repeat this until we cannot find a better-performing solution with these modifications anymore. Since, in the context of the use case in Chapter 5, I work within the bounds of a simulation budget, but the hill climbing algorithm could finish before the budget is used up, I apply *iterative* hill climbing (IHC), where I restart the hill climbing again with a new arbitrary solution.

Simulated Annealing (SA)

Similarly to hill climbing, we start with an arbitrary solution for our problem and "climb" by searching for better solutions within the current solution's neighbourhood. We intro-

duce, however, a temperature parameter T that decreases during the search process. While exploring the neighbourhood of the current solution, we can also accept a neighbour with a worse performance with a probability depending on T. Early in the exploration process (when T is higher), we are more likely to accept worse solutions for the coming iteration, the probability decreases with every step.

Tabu Search (TS)

This algorithm is a local search algorithm (similar to hill climbing) that keeps, however, a memory structure (Tabu list) of already visited solutions. During the search process, we do not explore solutions in this list, and accept worse solutions if there is no better solution available. These rules prevent us from getting stuck in a plateau or local optima.

2.4.2 Population-Based Metaheuristics

Among metaheuristic algorithms, population-based algorithms are useful for exploring the search space globally. Beheshti and Shamsuddin discuss in their survey popular population-based metaheuristic algorithms such as ant colony optimisation, particle swarm optimisation, and artificial bee colony optimisation [59], in scope of this research, however, are only genetic algorithms (GA) due to their efficiency in multi-objective settings [61].

A genetic algorithm is an evolutionary algorithm in the field of population-based metaheuristics. The fundamental idea here is applying concepts of natural selection to find solutions (shaped like a vector that we call "genomes") with desirable properties. Below, I first describe a setup of a classical GA, but since I focus in this work on applying a GA to graphs, we also present adaptations to interpret graphs as genomes.

Classical Genetic Algorithms

The classic genetic algorithm consists of multiple phases with several iterations of evaluation, selection, crossover, and mutation:

1. *Initialisation.* First of all, we generate a number of genomes to have an initial population to operate on in the first generation.
2. *Evaluation.* We define a fitness function that determines the quality of each genome in the current population.

3. *Selection.* According to the fitness of each genome, we select the best genomes of the generation for crossover.

4. *Crossover.* We use selected genomes to build a new population by mixing parts of these genomes.

5. *Mutation.* Since after a number of iterations, the genomes within a generation become very similar, we randomly introduce small changes according to a mutation rate to find genomes that we cannot create using crossover.

6. *Termination.* After creating a new population, we can start another iteration in the evaluation phase. We repeat this process until we reach a point to terminate the GA when we assessed a sufficient number of generations or when the best identified genome has not changed over a number of iterations.

Katoch et al. provide a review in [62] on strategies and parameters for each phase, and applications for those strategies.

Genetic Algorithms on Graphs

While the phases of GAs provide a generic framework for a metaheuristic search in a solution space, this space needs to be encoded in vectors that we can interpret as genomes. The solution space in this work, however, is a *graph* configuration space. Many popular graph embedding techniques that we apply for using ML on graphs do not apply here since they only offer transforming graphs into vectors, but not decoding them back to graphs. In context of GAs, that would mean we could not perform meaningful crossover and mutation operations to create a new population of graph solutions. Instead, we need to encode the graphs. Popular methods of representing graphs to use as genomes in GAs are listed below:

1. *Node-based encoding.* For some applications, such as the travelling salesman problem, it makes sense to encode the nodes directly by their identifiers and perform node permutations in the GA to optimise the salesman route [63]. For graph partitioning problems, however, it makes sense to assign each gene (i.e., node) a number according to one partition [64].

2. *Edge-based encoding.* Instead of encoding nodes, we can also directly encode edges, either as edge sets as used for spanning trees [65], or based on adjacency matrices to

apply GAs for community detection in social networks [66].

3. *Subgraph-based encoding.* For applications such as subgraph extraction in malware detection, we can encode edges of subgraphs according to an edge mapping [67].

4. *Other encodings.* For some domains, we can use domain-specific encodings, such as the Prüfer number for encoding trees [68], or binary encodings to map classes of paths onto single chromosomes [69].

2.5 Multi-Objective Optimisation

As the name indicates, a single-objective optimisation problem involves a single optimisation function. Whether a solution performs better than another is clearly defined by the result of the objective function being higher or lower.

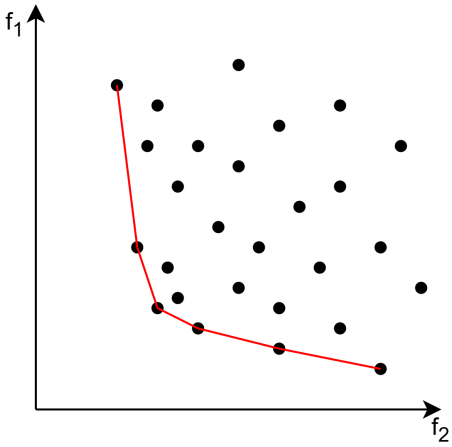
It becomes more complex once multiple goals are involved in a multi-objective optimisation (MOO) problem. For these problems, there is usually not one single solution outperforming all other solutions in every objective function; instead, there are trade-offs involved to find the optimal solution [70]. In these cases, we often speak about Pareto-optimality. A solution is Pareto-optimal if it cannot be improved without sacrificing performance in at least one other objective.

For a MOO problem with m objectives and the solution space S containing all valid solutions, we define an objective function f that determines the quality of a solution $s \in S$ by each objective: $f(s) = [f_1(s), f_2(s), ..., f_m(s)]$. A solution $x_p \in S$ is Pareto-optimal if there does not exist another $x \in S$ so that

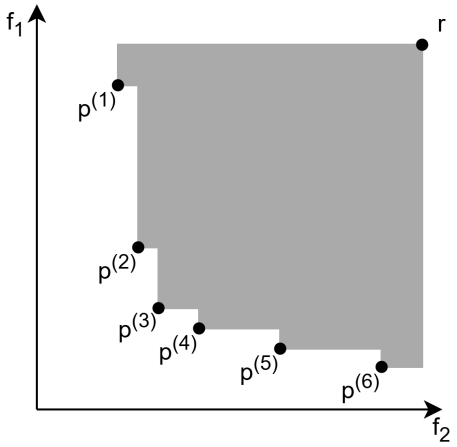
$$f_i(x) \leq f_i(x_p) \text{ for all } i \in \{1, 2, ..., m\} \text{ and } f_j(x) < f_j(x_p) \text{ for at least one } j \in \{1, 2, ..., m\}$$

We call a number of Pareto-optimal solutions a Pareto front. A common metric in evolutionary multi-objective optimisation to compare the quality of two Pareto fronts is the hypervolume indicator [71]. We calculate this parameter by measuring the area between the Pareto front and a reference point. A higher hypervolume indicates a better Pareto front.

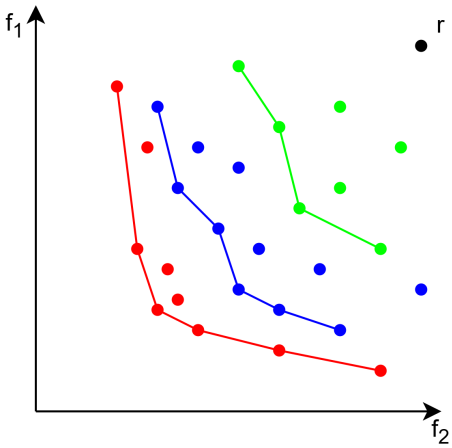
Figure 2.3 presents not only visual explanations of the Pareto front and the hypervolume indicator, but also visualises how the Pareto front moves and thus the hypervolume indicator changes over the course of a MOO run.



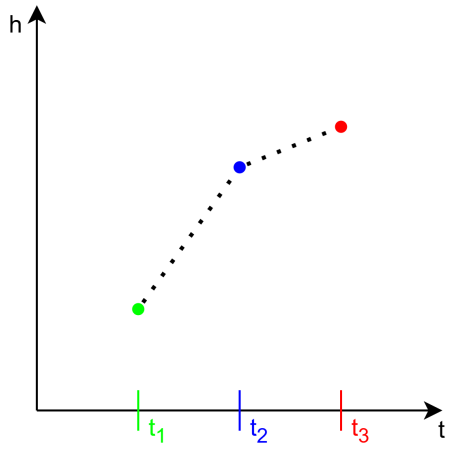
(a) A Pareto front (in red) consisting of the Pareto-optimal solutions in a MOO problem with two objective functions.



(b) The hypervolume indicator describes the grey area between Pareto-optimal solutions and a reference point *r*.



(c) The Pareto front changing over time starting from solutions early discovered (in green) to solutions later discovered in the optimisation process (in red).



(d) The hypervolume indicator changing over the same time from early stages of the optimisation algorithm (in green) to later stages (in red).

Figure 2.3: Visual explanations of the Pareto front and the hypervolume indicator during the multi-objective optimisation process.

2.6 Related Work

In this section, I give an overview of related work. First of all, on a general level, I provide a brief overview of search-based graph optimisation. Then, in connection to Case Study II (that I present in Section 3.5.2), I give an introduction to road network optimisation. I close with an overview of metaheuristics that are relevant to Chapters 5 and 6.

2.6.1 Search-Based Graph Optimisation

Finding k best solutions. Finding the k best solutions is a problem that arises in many applications in engineering and several approaches have been proposed. For instance, Dechter et al. [72] adapted well-known search heuristics such as A*, best-first, and depth-first to identify the k best solutions over graphical models (e.g., weighted constraint satisfaction problem and most probable explanation) in Bayesian networks. Comparing their approaches to a branch and bound, the authors prove that their adapted A* is complete, optimal, and efficient.

However, their graphical models are designed to represent finite domains of discrete variables with a set of real-valued cost functions which limits the search space and streamlines/ quickens the assessment of potential solutions. In contrast, our search space is not discrete, and in the absence of a mathematical function that is precise enough to assess the fitness of potential solutions without employing a time-consuming simulation the use of exhaustive search-based approaches becomes impractical.

Metaheuristics that require simulations (simheuristics). When the underlying optimisation problem is NP-hard, metaheuristics are required to solve large-scale instances in reasonable computing times. Devising metaheuristics (i.e., population and trajectory-based approaches) for problems that require the simulation of potential solutions to accurately assess their fitness is known as *simheuristics* [73] and has particularly been used in domains that deal with uncertainty such as logistics, transportation, and other supply chain areas (stochastic combinatorial optimisation problems) [74, 75]. In simheuristics, the challenge resides in finding the best solution to the problem. However, in our first case study, finding a best solution is relatively easy, but finding all (or a large portion) of the best solutions is the challenge.

2.6.2 Metaheuristics on Graphs

Metaheuristics have been used in the literature for a variety of tasks on graphs, including graph colouring [76], finding the maximum clique [77], or the minimum spanning tree [78].

With evolutionary approaches to tackle the travelling salesman problem [14, 79] and various approaches on the vehicle routing problem [80], there are two applications that are very similar to my graph configuration space exploration problem (i.e., finding one graph out of a large number of alternatives with the best non-functional properties).

The difference here is that the graph space in my work, in particular for the HVAC systems in Case Study I (see Section 3.5.1), is limited by strict constraints, and operating directly on those graphs would require constant healing to ensure that these constraints are met. While it has been shown on the network design problem that we can apply metaheuristics on graphs confirming in each step that constraints are met [81], the use case in Chapter 5 avoids this issue by generating a large set and applying the metaheuristics on graph embeddings. In Chapter 6, however, I do explore the option of applying a metaheuristic that modifies graphs in the set, and heal the graph according to the constraints during the process.

I can also find examples in the literature for applying metaheuristics on graph embeddings in applications like node clustering [82] or for combinatorial optimisation problems using graph neural networks as embedding technique [83].

2.6.3 Machine Learning for Metaheuristics

The literature presents various surveys and literature reviews on applying evolutionary approaches to improve machine learning systems [84, 85, 86]. Relevant for my research, however, is the opposite way: applying machine learning in metaheuristic algorithms.

This application of machine learning has recently gained more attention in the literature [87]. Talbi classifies the use of machine learning in metaheuristics between problem-level, low-level, and high-level in [88]. On problem-level, we use machine learning to model the optimisation problem to solve, on low-level (as I use it in this work), machine learning supports search components such as the initialisation of solutions or building neighbourhoods in local searches, and on high-level, we use machine learning to select and generate

metaheuristics to design cooperative metaheuristics.

Zhang et al. present a survey where they discuss applications of machine learning in evolutionary algorithms [89]. Among ML for population initialisation, reproduction and variation, and algorithm adaption, they also talk about the use case similar to Chapter 6: learning a fitness function.

Next to practical problems where the objective function is either not possible to express in analytical form of decision variables or it is too costly to evaluate this analytical form, they explicitly mention cases where we want to reduce the number of function evaluations. One example of this application is an approach on the minmax optimisation problem where we use a "surrogate" function (machine learning) to assess the fitness functions of solutions instead of costly calculations, and in each generation, only the best solutions get evaluated by the actual objective function [90].

In the future directions of the survey, Zhang et al. mention that there is more work to do to understand the tradeoff between computational complexity and optimisation performance, and assess the application of ML for GA on complex problems outside of numerical benchmark functions.

2.6.4 Road Network Optimisation

Farahani et al. reviewed problems and current practices of urban transportation network design problems in [91]. In this article, they differentiate between the decisions we make in network design into strategic (e.g., building new and expanding existing streets), tactical (determining the orientation of one-way streets, allocate exclusive bus lanes), and operational (e.g., scheduling traffic lights or repairs on streets). In this thesis, I focus on strategic decisions in network design.

In the same review, they list works on mixed network design problems (discrete and continuous network optimisation) by the objective (e.g., minimise total travel time), discrete decisions (e.g., constructing new streets) and continuous decisions (e.g. street capacity expansion), and solution methods (e.g., genetic algorithm, simulated annealing, scatter search, etc.). Examples include Cantarella et al., who applied various metaheuristics to minimise the total travel time through orienting sequences of streets and traffic light settings [92] and Zhang and Gao, who applied a gradient-based method with penalty function to minimise

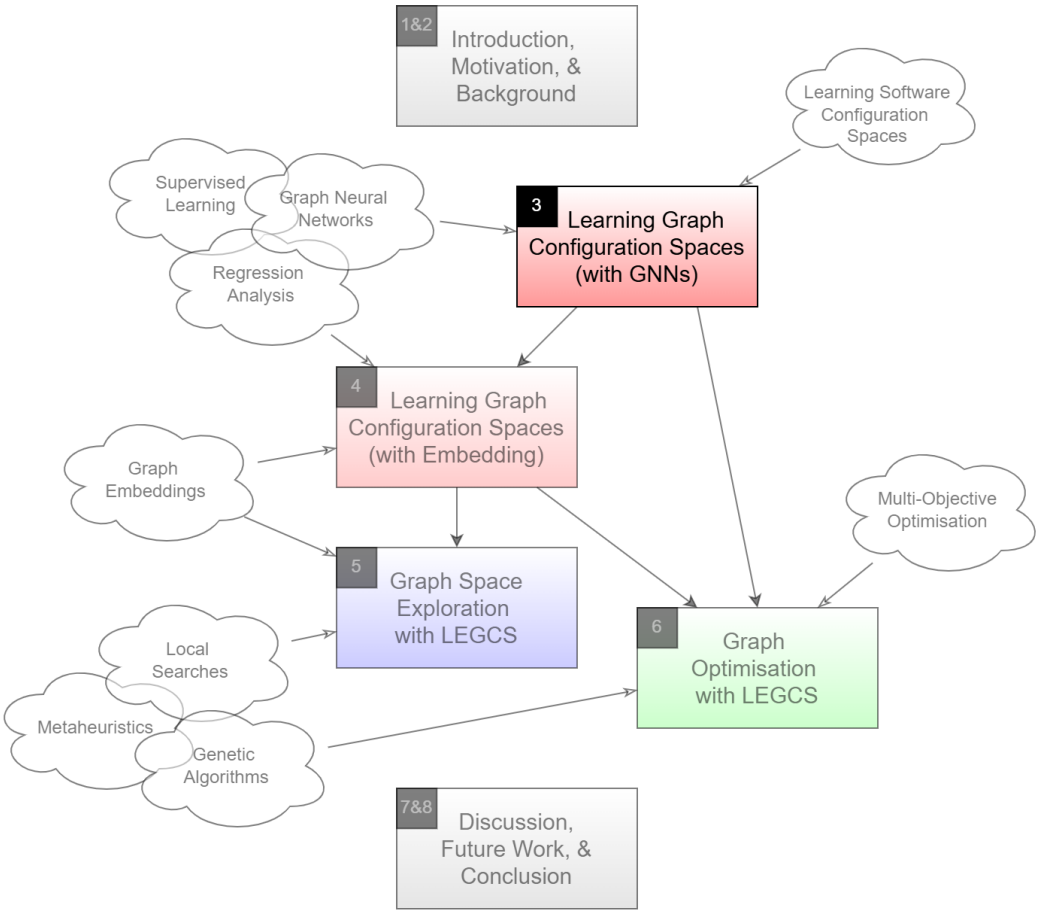
the total travel cost and construction cost by constructing new streets and street capacity expansion [93].

An elemental step to optimising road networks is traffic simulations. Aleksander and Pawel provide a review on advances in traffic optimisation where they analysed new computer-assisted technologies, techniques, and solutions [94]. In this review, they not only talk about cellular automata and multi-agent systems to optimise traffic flow, but also evolutionary approaches such as ant colony optimisation or genetic algorithms, e.g., to optimise traffic lights.

The idea of applying GNNs on traffic networks has precedent. Jiang et al. published a survey on traffic forecasting with graph neural networks [95], where they discuss predicting traffic flow, speed, and demand as well as modelling requirements for representing spatial dependency. More related problems that were not in scope of this work concern urban network traffic signal controls and traffic classification. Noaeen et al. provide a systematic literature review on applying reinforcement learning on traffic signal controls [96] while Pacheco et al. surveyed the use of machine learning for traffic classification [97]. Slowik and Kwasnicka published an article on applications of evolutionary algorithms that include but are not limited to traffic network design [98].

Chapter 3

Learning Graph Configuration Spaces



3.1 Introduction

Learning-supported product configuration has been established for configurations we can express in the Euclidean space (vectors and matrices). Some applications, however, require us to configure products in graph structures.

In this chapter, I discuss the feasibility of applying and adapting strategies of learning Euclidean configuration spaces [1] to narrow down a graph configuration space by answering the following research question:

RQ 3.1 How can we apply configuration space learning methods established for Euclidean configuration spaces to graph configuration spaces?

Here, I aim to "translate" concepts and methods proposed by [1] to learn software configuration spaces to build a pipeline that learns graph configuration spaces. Furthermore, not only do I build a solution to learn graph configuration spaces, I also investigate how efficiently this initial approach can explore graph configuration spaces by answering:

RQ 3.2 How does tool-supported learning of graph configuration spaces compare to a random search approach with respect to time consumption and quality of determined graph configurations?

Contributions. This chapter makes the following contributions:

1. I propose a pipeline of adapted techniques from learning software configuration spaces to demonstrate the feasibility and applicability of these techniques for learning *graph* configuration spaces.

2. I report on controlled experiments that explore (1) a dataset of graph configurations describing house ventilation systems by AC consumption with as few MATLAB Simulink simulations as possible and (2) a dataset of graph configurations describing traffic networks by average travel time with as few SUMO simulations as possible. In these experiments, I compare the pipeline proposed to answer *RQ 3.1* against a random search approach. Thus, providing further evidence of the feasibility of the approach.

The results show that the overall approach of learning software configuration spaces is adaptable for graph configurations and offers an alternative to iterative approaches by considering a larger space of graph configuration candidates. In terms of efficiency, first

the results show that in two case studies the learning graph configuration spaces (LEGCS) approach reliably identifies graph configurations with good performances (by AC usage in HVAC systems, and travel time in traffic networks) prior to simulations, thus, we can explore the graph configuration space much faster.

The remainder of this chapter is structured as follows: after this introduction, I provide a mathematical formalisation of the four phases of learning graph configuration spaces (Section 3.2), continued by the research design (Section 3.3) and the GNN techniques I evaluate in this chapter (Section 3.4). Afterwards, I present the case studies that I operate on (Section 3.5) and the results on feasibility (Section 3.6.1), and efficiency (Section 3.6.2). This chapter closes with a conclusion (Section 3.7).

3.2 Problem Statement

Before I discuss how I tackle the learning configuration space problem on graphs, I first formalise the problem in this section. Table 3.1 presents an overview of the variables used in this section, and below is a formal description of the four phases presented in Section 2.2 adapted to graph spaces. Parts of the formalisation are adapted from [1].

Symbol	Description
C	Domain constraints
G	Graph space
S	Graph sample
σ	Number of sampled graphs
M	Performance measurements of the sampled graphs
μ	Number of performance metrics
f	Fitness function assessing the performance of a graph
f'	Learned fitness function predicting the performance of a graph
e	Embedding function
ε	Dimensions of the embedding
f''	Learned fitness function predicting the performance of a graph based on graph embeddings
v	Validation function assessing the quality of a learning function

Table 3.1: Variables in the learning graph configuration space problem.

In the graph space exploration problem, we consider a graph space G that is limited by domain-specific constraints C.

$$C = \{c_1, c_2, \ldots, c_n\}$$

$$G = \{g \mid c_1(g) \wedge c_2(g) \wedge \ldots \wedge c_n(g)\}$$

Sampling

In the sampling phase, we sample a subset S by selecting σ graphs in G.

$$S = \{s_1, s_2, \ldots, s_\sigma\} \subseteq G$$

Measuring

In the measuring phase, we measure the performance of each graph in the sample S in each of the μ goals with a fitness function f.

$$f : G \to \mathbb{R}^{\mu} \text{ where } f_j : G \to \mathbb{R} \text{ for } 1 \leq j \leq \mu$$

$$f(g) = \begin{bmatrix} f_1(g) \\ f_2(g) \\ \vdots \\ f_{\mu}(g) \end{bmatrix}$$

$$M = \{f(g) \mid g \in S\}$$

Learning

In the learning phase, we use the graph sample S and measurements M to learn a function f' that predicts the results of the fitness function f for the whole space of graphs.

$$f' : G \to \mathbb{R}^{\mu} \text{ that optimises a validation function } v : (G, f, f') \to \mathbb{R}$$

Note that this formulation of the learned fitness function assumes that f' can operate on graphs as input directly. Such an approach can, for instance, be implemented with graph neural networks. I investigate this kind of approach here in Chapter 3. However, other learning approaches operate on *embeddings* of the graphs into a vector space. For these approaches, I define an embedding function e, and another learned fitness function f'' predicting the results of f based on the embeddings. This kind of approach is investigated in Chapter 4.

$$e : G \to \mathbb{R}^{\varepsilon} \text{ and } f'' : \mathbb{R}^{\varepsilon} \to \mathbb{R}^{\mu} \text{ so that } f'(g) = (f'' \circ e)(g) = f''(e(g))$$

Evaluation

Eventually, in the evaluating phase, we evaluate the predictions of f' with our validation function v. Below is a brief overview of commonly used loss functions [99].

1. Mean Absolute Error (MAE): This metric describes the average prediction error.

$$\text{MAE} = \frac{1}{\gamma} \sum_{g \in G} |f(g) - f'(g)|$$

2. Mean Squared Error (MSE): Here, we penalise larger prediction errors more heavily than smaller ones.

$$\text{MSE} = \frac{1}{\gamma} \sum_{g \in G} (f(g) - f'(g))^2$$

3. Rooted Mean Squared Error (RMSE): We take the root of the MSE which makes interpreting easier.

$$\text{RMSE} = \sqrt{\frac{1}{\gamma} \sum_{g \in G} (f(g) - f'(g))^2}$$

4. Coefficient of Determination (r^2): For this metric, we take the prediction error of the predicted value in relation to the prediction error if we predict the average true value every time. If the coefficient is *1*, the predicted values are a perfect fit on the true values, *0* means the predictions are as good as always predicting the average true value, and negative values mean that the predictions perform worse than always predicting the average true value.

$$r^2 = 1 - \frac{\sum (f(g) - f'(g))^2}{\sum (f(g) - a)^2} \text{ for all } g \in G \text{ where } a = \frac{1}{\gamma} \sum_{g \in G} f(g)$$

3.3 Research Design

To answer the research questions raised in the introduction, I propose a pipeline of techniques for learning configuration spaces adapted to graph configurations (*RQ 3.1*) and conduct controlled experiments using this pipeline (*RQ 3.2*). I first give details on the research methodology. In the following Section 3.4, I talk about the implementation of the applied GNN techniques and in Section 3.5 report on the case studies I apply LEGCS on.

Answering *RQ 3.1*: Proposal of a Pipeline of Adapted Techniques

A research question like *RQ 3.1* asks for a "method or means of development" (see the taxonomy suggested by Shaw [100]). I identify a pipeline of exploration techniques and suggest how they can be adapted to graph configurations. Note that the results of the experiments presented in 3.6.2 provide further evidence that the approach is feasible.

Answering *RQ 3.2*: Using a Controlled Experiment

A research question like *RQ 3.2* asks for a "design, evaluation, or analysis of a particular instance" [100]. I conduct a controlled experiment [101] and compare my approach (i.e., the pipeline suggested in RQ 3.1) to a naïve simulation approach – in terms of the quality of identified k best graph configurations and the time needed to identify them. While reporting the controlled experiment, I follow the guidelines of Jedlitschka et al. [102].

Case Studies

As the starting point of this work and to provide subjects for the controlled experiments, I chose two case studies:

1. An example problem from MATLAB Simulink [103], a simulation tool commonly used in various engineering disciplines. This civil engineering problem deals with building ventilation and HVAC (heating, ventilation, and air conditioning) systems [5]. The goal is to explore and filter the space of possible floor plans and air ventilation systems with regard to their monthly air conditioning (AC) usage. I interpret and represent these floor plans as graphs configuring a building, making this an exploration problem on graph configurations.

2. A traffic network design problem based on towns and cities in Ireland using the traffic simulation tool SUMO [104]. The goal here is to explore and filter the space of possible street networks with regard to the average travel time of cars in the network. I interpret the set of streets in one network as a graph configuration of the traffic network.

I selected these case studies, because both present engineering problems where configurations are expressed by relationships between entities that can be best expressed in graphs. Furthermore, they include non-trivial assessments of NFPs conducted by simulation tools that are commonly used in the industry. Simulations in both cases, however, use high computational resources, thus, motivating the use of surrogate assessment functions (i.e., regression analysis). Thus making these problem cases good examples for the need of learning-supported LEGCS tools. Section 3.5 gives more details on these case studies including the technical aspects.

3.4 Graph Neural Network Techniques

In Section 2.3.1, I presented GNNs as a method to apply regression on graphs, including guidelines on planning a GNN architecture. Following these guidelines about designing a GNN architecture, I made the following decisions for the learning system in context of LEGCS:

In Case Study I, I chose a five-dimensional feature vector in each node for *message passing*. The first dimension represents the room's size, the remaining ones are flags to indicate whether the room is connected to an AC (and on which end of an AC edge the room is), an outside door, or neither.

In Case Study II, I chose a two-dimensional feature vector in each node representing its geographic location (i.e., latitude and longitude). Additionally, the *message passing* is affected by the edge weight between nodes representing the distance covered by that road, as well as the number of edges between two nodes representing the number of lanes in the respective direction.

In terms of *aggregation*, *layer connectivity*, and *graph manipulation*, I have chosen to test three popular architectures that cover different families of GNN approaches [33]: GCN, GAT, and GraphSAGE.

- *GCN:* For the first GNN approach, I use a graph convolutional neural network (GCN) [105] implemented by the Pytorch Geometric framework [106]. It is a common GNN architecture [33] where we first assign a feature vector to each node. In every iteration step, we modify the feature vector of each node by aggregating it with the feature vector of the node's neighbours. Finally, we have a new set of vectors for the nodes that are learned from the neighbourhood of the nodes.

- *GAT:* Graph attention networks (GAT) [107] add an attention mechanism to the GNN. By assigning coefficients to neighbourhoods of each node, the network allows the capturing of complex relationships and dependencies within the graph.

- *GraphSAGE:* GraphSAGE [108] is another GNN architecture that samples and aggregates features from a node's local neighbourhood instead of training individual embeddings for each node to scale GNNs for large graphs.

For Case Study I, I run the GNNs with two hidden layers between input and output layers, while the GNNs in Case Study II have an additional third hidden layer. The number of channels in those layers are determined according to a parameter grid with up to 128 channels. I picked the batch size dynamically to the amount of training data, for more than 500 graphs the batch size was 128, for less training data smaller. I applied rectified linear unit activation functions, and ran the GNNs for 1000 epochs in Case Study I, and 100 epochs in Case Study II.

Since the *learning objective* is a prediction on whole-graph level (i.e., the AC consumption of the house configuration per month, and the average travel time of the cars in the traffic network), I acquire this prediction by mean pooling the feature vectors of all nodes in the GNNs [90].

3.5 Case Studies

This section provides an overview of the two case studies in this thesis. In particular, I describe the goals within the case studies and their implementations, i.e., the suggested combination of adapted techniques. I describe (1) how I generated graph configurations that span up the explored configuration space, (2) the domains-specific constraints and objectives affecting the configurations, (3) how I measured the properties resulting from the configurations, i.e., the fitness of a solution, and (4) what frameworks I used for the implementation.

3.5.1 Case Study I: House Ventilation

Goals within the Case Study

My graph space exploration example is from civil engineering dealing with HVAC systems [5]. I have a large dataset of graph configuration candidates, each configuring the setup of an HVAC system in the same house (description of this dataset below). Within my experiments I compare multiple strategies to fulfil the following goals:

- Filter the incomprehensibly large dataset of 40,000 graph configurations down to a humanly comprehensible space of desirable configuration candidates. By that, I refer to identifying the 1,344 candidates (3.36% of the whole dataset) with the lowest AC usage.
- Keep the number of costly simulations to assess the AC usage of the candidates low.

Floor Plan Generation

First, I implemented a simple floor plan generator based on the work of Lopes et al. [109] that creates JSON files, each containing one configuration. This generator divides the entire floor into a grid, places initial room positions, and then grows rooms to their maximal feasible rectangular size. Finally, some rooms grow into an L-shape to fill the remaining space. Note that in this work, I do not focus on the coverage of graph generation and sampling to the entire valid graph configuration space. Instead, my primary objective is to enhance the efficiency of the subsequent processes (i.e., learning and selection) based on a very large number of given or generated graphs. Hence, I do not attempt to consider or even enumer-

ate *all* possible graph configurations.

Graph Representation of Floor Plans

In the dataset[1], I explore a graph space consisting of 40,000 floor plans that I represent as graphs; each graph represents a different configuration of rooms within the same building. Rooms are represented as nodes in our graph model, and the airflow between neighbouring rooms as edges. One room A can be connected to another room B by (1) air exchange, (2) sucking hot air out of room B into room A's AC, or (3) both. These graphs contain between three and ten nodes. This case is based on a standard Simulink system demonstrating the modelling of buildings and their thermal properties [5] as follows:

- Each node (room) has a room size and Boolean flags indicating whether this room is connected to AC or an outside door.
- Every connected subgraph has at least one room with AC.
- Each graph has one room with an outside door.
- There are two edge types. One edge type connects two rooms for regular airflow; each node can have up to three of these edges. The other edge type connects rooms that have neither AC nor an outside door with AC. This edge pulls hot air into the AC to cool down; every AC needs one edge of this type.

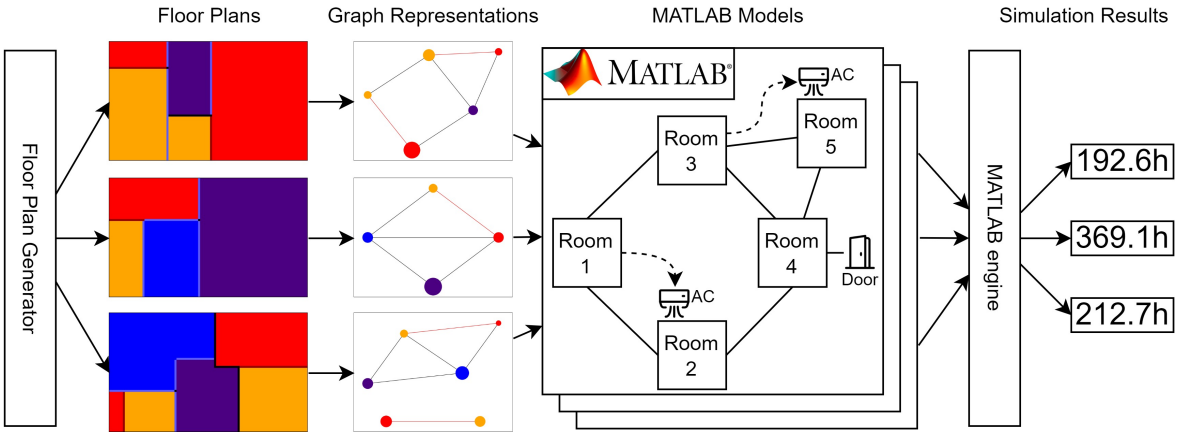


Figure 3.1: Overview of the implementation of Case Study I.

[1] available at https://github.com/mittermm/LEGCS

Graph Configuration Measurement

This exploration problem is based on a MATLAB Simulink example project [5]. Simulink is a simulation tool that is commonly used in various engineering disciplines. In order to assess the performance of each configuration candidate, I built a tool transforming our graph representations of floor plans into MATLAB models that the MATLAB engine can execute as simulations to determine the AC costs (the number of hours the AC runs within a month). Figure 3.1 gives an overview of this process.

This experiment (i.e., the simulations) was executed on a Linux Mint 21 Cinnamon machine with 16GB of RAM, and an Intel Core i7-6700 CPU. With this set-up, each simulation takes about a minute, which means a brute force simulation approach to explore a graph configuration space of 40,000 graphs takes about 28 days to complete.

3.5.2 Case Study II-A: Traffic Network Exploration

Goals within the case study

In this street network optimisation problem, I deal with networks of 32 towns and cities in Ireland; each representing one county. Within these municipalities, I aim to find connecting streets so that:

- The travel times are low (ideally, that means low travelling distances and high traffic flow). By that, I refer to the top 5.31% of the dataset (531 graphs) in terms of travelling times.
- The number of costly simulations to assess these travel times in the exploration process are low.

I optimise towards these goals by modifying the number of roads in the network as well as the orientation and number of lanes of individual streets in the network. By the definitions in [91], I make strategic decisions in a mixed network design problem.

Traffic Network Generation

I generated *10,000* valid graph configurations (meaning that the graphs representing the street network are strongly connected) with randomly selected edges and varying numbers of edges between *62* and *400*. I selected this minimum and maximum based on experience from early experiments that indicated that the ideal number of edges for low travelling times while also keeping the overall road distance low is unlikely outside this scope. One possible configuration is visualised in Figure 3.2.

Graph Representation of Traffic Networks

In this second dataset[2], I explore a graph space consisting of *10,000* street networks that I represent as graphs; each graph represents a different configuration of streets connecting the same *32* cities. One configuration in context of this case study is a graph $G = (V, E)$ representing one street network, where V is the set of 32 towns and cities, and E is a set of directed edges describing individual lanes of streets connecting them. While V stays the same for all networks, our goal consists of optimising the edges E. The only constraint for graphs in this use case is:

- The whole graph is strongly connected. (That means that every node is reachable from any node in the network.)

Graph Configuration Measurement

I analysed the configurations by running the simulation of urban mobility (SUMO) [104] created by the German Aerospace Center (DLR) (see Figure 3.2), one of the most popular simulation tools in traffic network research [94]. In these simulations, I send out traffic of *1,000* cars, each with a start and an endpoint. While those points are randomly picked, towns with higher populations are more likely to be picked as start or endpoint (proportional to the population). For each car in the set C, I asses the distance d and the average speed s during the simulations to determine the travelling time. The goal of this study is to identify street networks with minimal overall travelling time for the cars in the network, or mathematically speaking:

[2]available at https://github.com/mittermm/MOGA_LEGCS

$$\min f(G) = \sum_{c \in C} \frac{d_c}{s_c}$$

This experiment was executed on a Ubuntu 24.10 machine with 32GB of RAM, and an Intel Core i7-14700 CPU. With this set-up, one simulation takes about 20s on average, which means a brute-force approach to explore a graph configuration space of *10,000* graphs takes about 55.6h to complete.

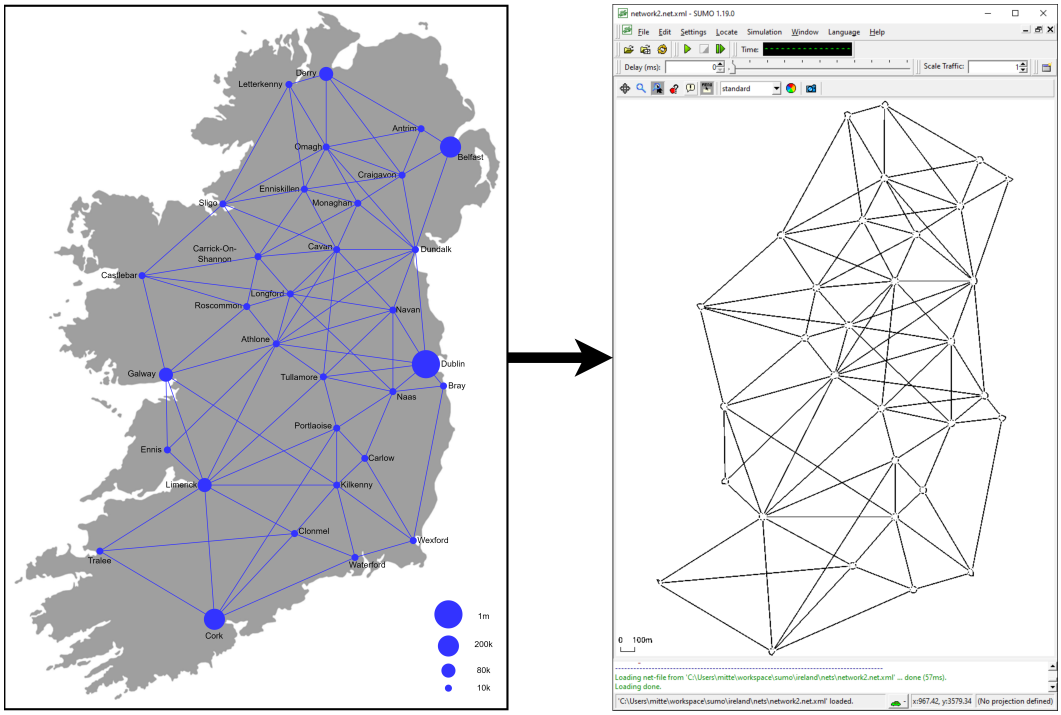


Figure 3.2: Example of a traffic network of Ireland and its equivalent SUMO network.

3.6 Results and Discussion

In this section, I present my suggested pipeline (*RQ 3.1*) and discuss the results of the controlled experiment (*RQ 3.2*).

3.6.1 Feasibility

I chose one path of strategies from learning software configuration spaces to apply to graph configuration spaces. Figure 3.3 gives an overview of the strategies I chose in each phase (highlighted in boldface) and the respective goals of these strategies. I give a rationale for each selection below.

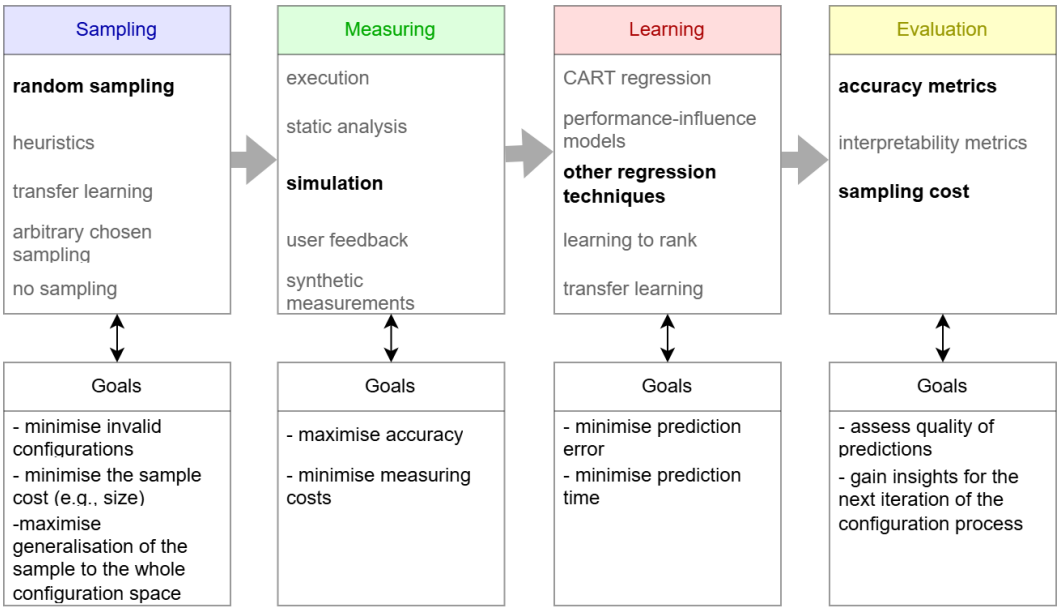


Figure 3.3: Strategies I chose in each phase of learning graph configuration spaces with their respective goals.

I used random sampling as it is a widely used baseline for sampling that needed no further adaptations for graphs. Future work will be able to answer if heuristics or transfer learning in software configurations are adaptable for graph spaces. I did not have the resources (domain experts) for arbitrarily chosen sampling, and "no sampling" is not useful due to the large size of the configuration spaces.

I relied on simulations (MATLAB Simulink for Case Study I and SUMO for Case Study II) for measuring and considered the simulation results as a "truth" regarding the properties/ fitness of the solution defined by the configuration. If even more fidelity of the performance evaluation is desired, one could actually start building houses or street networks corre-

sponding to the simulated models; however, due to the associated costs and complexity of the analysis, that is beyond the scope of my work. An additional extension could be soft qualities, like "user satisfaction" (e.g., aiming to measure satisfaction with a provided floor layout), but that is beyond the scope of my work. In practice, one would probably aim to represent such objectives with an approximation taking into account, for example, that offices with high occupancy are not desirable or that walking distances between related offices should be minimised.

In the learning phase, I use graph neural networks as an intuitive way to apply regression analysis on graph structures. However, in the following Chapter 4, I propose a variation replacing the GNN with an embedder and a regression technique in an attempt to train a sampling-efficient learning model.

I evaluated the predictions by accuracy and sampling size. Interpreting the results is challenging since it is naturally difficult for deep learning techniques [110]. Future work will discuss how interpreting could improve the learning process for graph configuration spaces. In the remainder of this thesis, I refer to the approach described in this section as the learning graph configuration space approach: LEGCS.

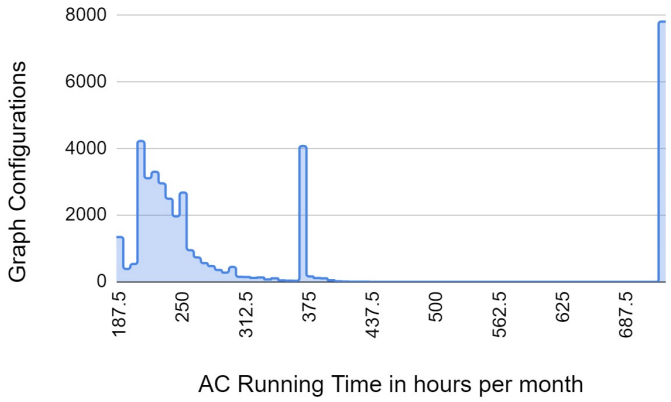
3.6.2 Efficiency

I now evaluate how efficiently we can learn the graph configuration space with respect to the accuracy and sampling costs to explore the configuration space. I implemented LEGCS according to the discussion in Section 3.6.1 and set up the experiment according to the case studies introduced in Section 3.5. In terms of reporting the controlled experiment, I follow the guidelines of Jedlitschka et al. [102] , which provide a structure for reporting controlled experiments and questions to cover, e.g., in terms of planning, analysing, and discussing the experiment.

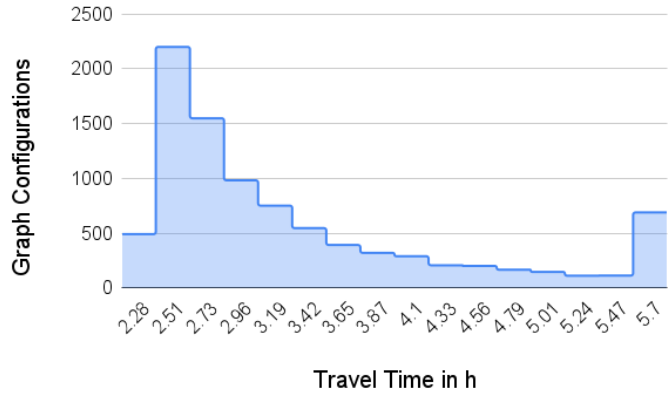
Goal of the Experiment

The overall goal of the controlled experiment is to analyse the quality of the simulated graph configurations suggested by the ranking of LEGCS predictions compared to a random selection. Instead of finding the one optimal configuration as fast as possible, I aim to narrow down the graph configuration space to provide the k best solutions (that human experts

will further explore) within a limited time. As experimental units, I operate on 40,000 graph configurations (i.e., floor plans) for Case Study I, and 10,000 graph configurations (i.e., street networks) for Case Study II. Figure 3.4 shows histograms of the graph configurations in both case studies according to the NFP that my LEGCS approach predicts.



(a) Histogram of graph configurations in the first dataset organised by AC usage in hours per month. The bin size is 25,000 seconds.



(b) Histogram of graph configurations in the second dataset organised by average travel time in hours. The bin size is 828 seconds.

Figure 3.4: Histogram of graph configurations in both case studies.

For Case Study I, the diagram shows the distribution of the 40,000 graph configurations in the dataset by AC usage per month. The peak on the right end of the spectrum shows configurations where the AC never turns off. The research presented here aims to quickly (with respect to the number of needed simulations) identify the 1,344 configurations (3.36% of the dataset) in the bin with minimal AC usage (at the left end of the spectrum).

For Case Study II, we see the distribution of the 10,000 graph configurations in the dataset by the average travel time of the cars in the traffic network. The peak on the right of the spectrum shows all networks where cars are travelling on average for 5.7h or more. I

am interested in quickly (with respect to the number of needed simulations) identifying the 531 configurations (5.31% of the dataset) in the bin with minimal travel time (at the left end of the spectrum).

The goal for the learning model from here is to accurately predict the outcomes of the simulations for these graphs within seconds (while the simulation of one configuration already takes about a minute in Case Study I and 20s in Case Study II). This way, I can explore the graph configuration space more efficiently by predicting the simulation results of most configurations and only simulating promising configurations.

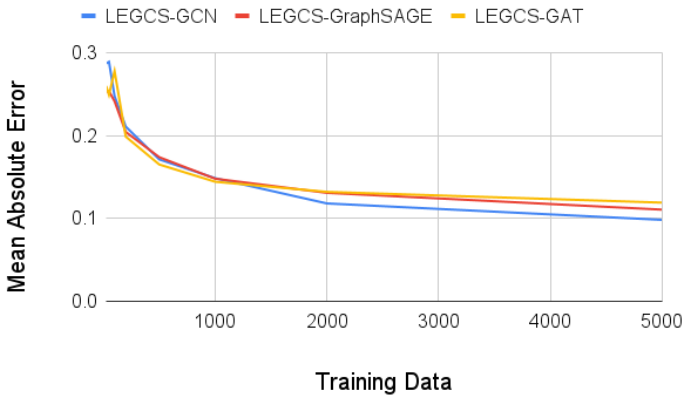
I implemented three GNNs according to Section 3.4 using the PyTorch Geometric Framework [106]. In terms of training, I have a trade-off between investing time to run *more* simulations for training the model (model available later, but gives better predictions) or *fewer* simulations (model available faster, but predictions are less precise). Below, I discuss this problem and give a rationale as to when I viewed the model as sufficiently trained.

Prediction Accuracy

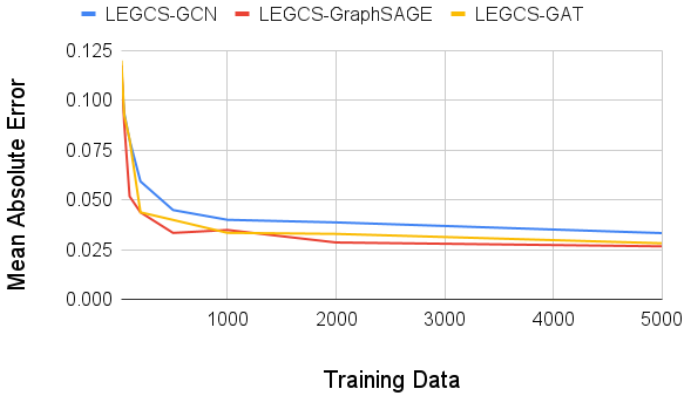
First, I assess how efficiently the GNNs can predict the NFPs with respect to the prediction accuracy and the necessary amount of training data. Figure 3.5 shows the relationship between training data and prediction accuracy in both case studies. The figures show prediction errors of learning models trained with dataset sizes between 25 and 5,000, and tested on the remaining unseen graphs (35,000 graphs for Case Study I, and 5,000 graphs for Case Study II).

We can see that the prediction error shrinks when I provide more training data to the learning models. We can see for Case Study I, the GNN architectures behave similarly, with GCN having slightly lower prediction errors for large amounts of training data. In the second case study, the prediction errors are much lower (see y-axis), and the point where adding more training data does not significantly improve the learning outcome is found much earlier. Here, GCN produces slightly higher prediction errors than the other architectures. In practice, it is unpractical (time-wise) to feed a learning model large training graph samples with their respective simulation results given the time it takes to simulate each graph (one minute per graph in Case Study I, and 20s per graph in Case Study II). Therefore, a trade-off must be made between low training samples and high accuracy of the

predictions.



(a) Learning Efficiency of LEGCS in Case Study I (HVAC systems).



(b) Learning Efficiency of LEGCS in Case Study II (Traffic Networks).

Figure 3.5: Learning efficiency of LEGCS in both case studies.

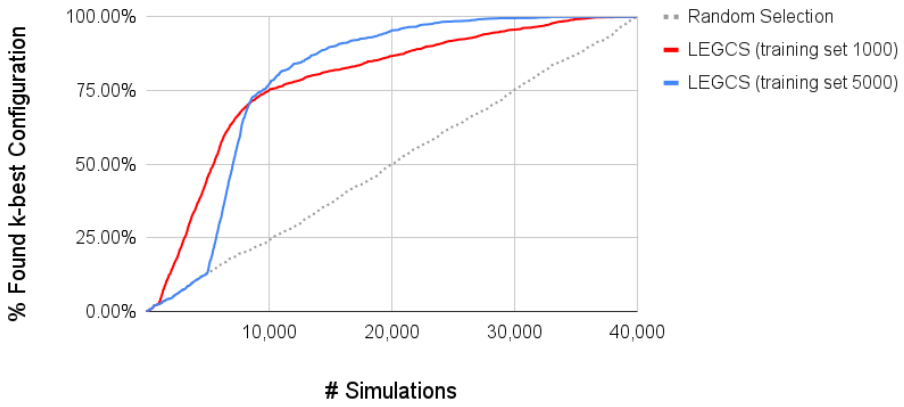
Implications for Graph Space Exploration

As described earlier in this section, I now assess how fast I can identify the best graph configurations in each dataset if I train a GNN model to guide us through the exploration process. That means that after an initial phase of random sampling and measuring to train the model, we let the model pick the next graph for simulation. Figure 3.6 shows the relationship of identified graphs with desirable properties according to the time (i.e., number of simulations) it takes to find them.

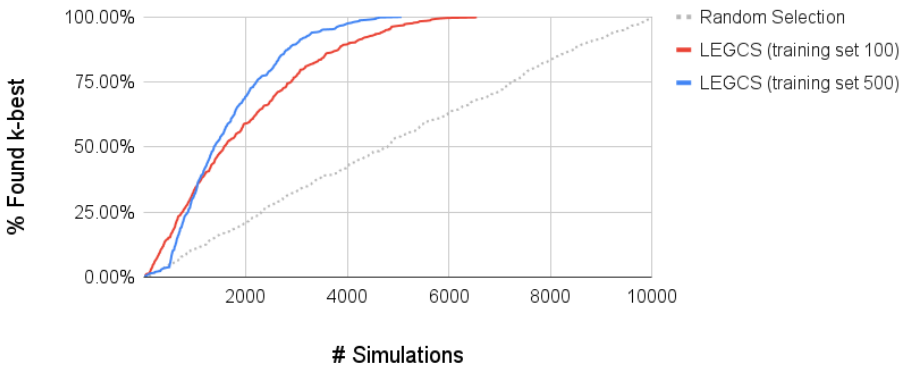
In Case Study I (HVAC systems), I chose the GCN architecture since it has the lowest prediction errors (see Fig. 3.5a). First of all, we can see a trade-off between spending time on simulations to train the model. For example, if I train the model with *1,000* graph configurations, we can earlier in the exploration process move on to the model guiding us through the

exploration process. This way, we can already identify 50% of the k best graph configurations in the whole dataset after simulating only 13.75% (i.e., *5,500* simulations including the training) of graphs in the dataset. On the other hand, if I spend *5,000* simulations to train the model, we can see that the model finds more reliably the k best graph configurations after the training phase (see the steeper curve in Figure 3.6a). This way, we can identify 75% of the k best graph configurations after simulating 23.75% (i.e. *9,500* simulations) of the dataset and over 95% after simulating half of the dataset. In the chapters 5 and 6, I explore metaheuristics that add re-training to LEGCS during the exploration and later optimisation process.

In Case Study II (traffic networks), I chose a GraphSAGE architecture because of the low prediction errors (see Fig. 3.5b). Since those prediction errors are lower for smaller sets of training data than in the previous case study, we can earlier switch from training the model to guided exploration as we can see in Figure 3.6b. After spending the first *500* simulations for model training, we can identify half of the k best graph configurations using 13.8% of the simulations (*1,380* simulations from *10,000* traffic networks), 75% of the most desirable graphs after 22.2% of the simulations, and all of the best graph configurations after simulating 50.7% of the graph dataset.



(a) Graph Space Exploration Process in Case Study I (HVAC systems).



(b) Graph Space Exploration Process in Case Study II (traffic networks).

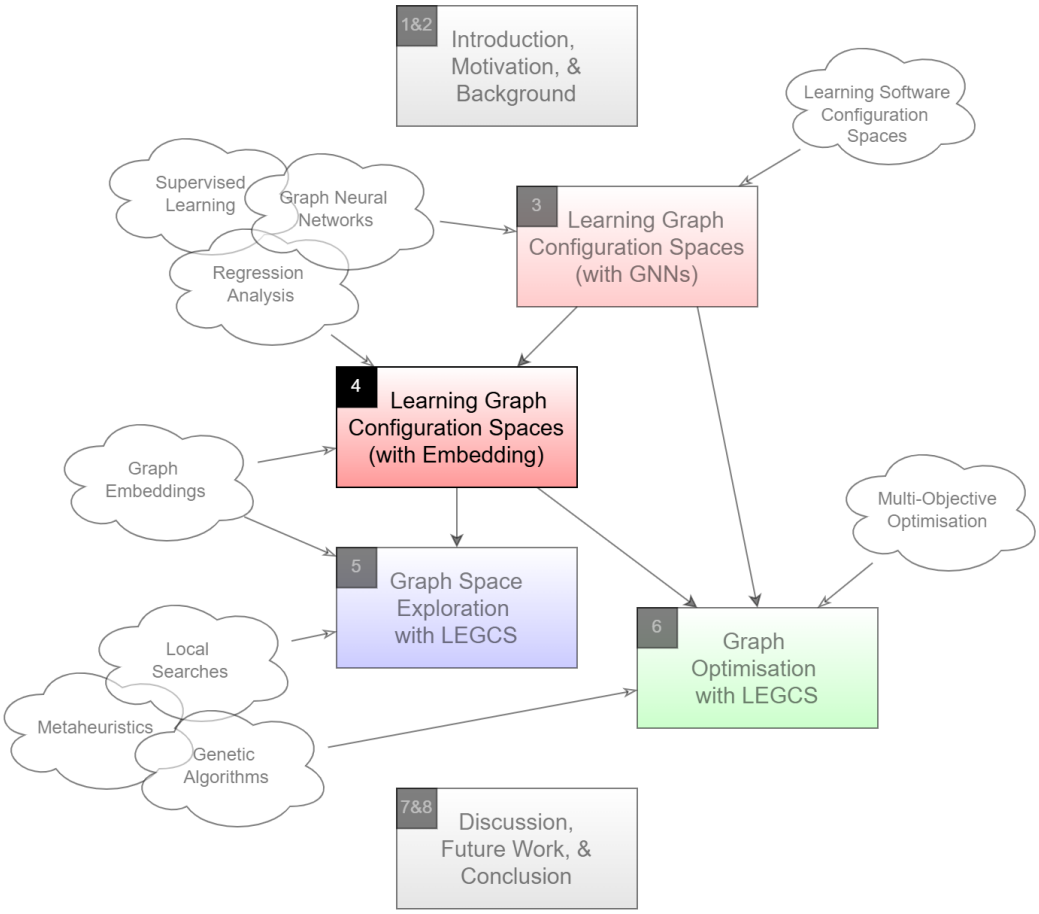
Figure 3.6: Graph space exploration process with LEGCS.

3.7 Conclusion

In this chapter, I investigated how we can adapt current practices of learning software configuration spaces originally devised for Euclidean configurations (consisting of Boolean and numerical variables) to graph configurations (containing structural information). I proposed and implemented a pipeline of strategies (that were already established for software configurations) for graphs as a proof of concept. Subsequently, I evaluated its efficiency for the exploration process. The presented approach can support engineers in exploring configurations of complex graph structures – while considering a large and diverse set of possible candidates in an efficient manner. The proposed LEGCS approach finds in two case studies on HVAC systems and traffic networks 75% of the k best graph configurations while simulating only 23.75% of the space.

Chapter 4

Embedding-Based LEGCS



4.1 Introduction

In the previous chapter, I have shown that regression analysis based on graph neural networks can support learning graph configuration spaces. The initial benefit here is that GNNs offer predictions while directly operating on graph structures without much preprocessing. However, deep learning techniques such as GNNs require large amounts of training data to generalise [111]. In a problem setting where training data is costly, this calls for an alternative learning method that learns the NFPs of the graph configurations more efficiently.

I hypothesise that it would be beneficial in terms of training efficiency to apply a preprocessing of the graph configurations that captures relevant structural information and apply traditional regression techniques on that information to predict NFPs. Capturing structural graph information in Euclidean data structures (i.e., vectors and matrices) is called graph embedding [47]. In this chapter, I investigate whether a traditional regression analysis approach on vectors in combination with a graph embedding can be used for LEGCS.

Since this approach to predicting graph properties involves two separate steps, I investigate what influence the graph embedding and the regression model have next to the training set size on the performance of the learning models. Ultimately, the learning model is limited by both: (i) the ability of the embedder to sufficiently capture graph information in a Euclidean vector, and (ii) the ability of the regression function to sufficiently depict the data points in our set. I want to observe through running various combinations of embedding and regression strategies how these two limitations influence the learning results to answer the research question:

RQ 4.1 What is the impact of graph embedding and regression algorithms on learning-based graph space exploration in terms of prediction accuracy?

Consequently, I compare how an embedding-based approach compares to the GNN approach I introduced in the previous chapter to answer:

RQ 4.2 How does learning based on GNNs compare to regression techniques based on graph embeddings with respect to accuracy of the predictions and the size of the required training set in context of learning graph configuration spaces?

Contributions. This chapter makes the following contributions:

1. I propose a variation of LEGCS based on graph embeddings that applies regression analysis on vectors representing graph configurations to facilitate a training-efficient prediction model.
2. I provide an in-depth study on the impact of graph embeddings and regression techniques on the prediction accuracy within learning graph configuration spaces.
3. I report on a controlled experiment that explores the efficiency of regression analysis with (a) the GNN-based approach and (b) the embedding-based approach with respect to accuracy and training data size.

The results show that LEGCS based on embeddings and regression analysis lead to efficient learning in terms of training data and prediction errors. The embedding technique here has to be selected carefully since it has a large influence on the learning model performance.

The remainder of the chapter is structured as follows: First, I compare both learning strategies for LEGCS (Section 4.2). Following that, I describe the concrete embedding and regression techniques I use for the experiments in this chapter (Section 4.3) and the experimental set-up (Section 4.4). Eventually, I present the results on the influence of embedding and regression on the embedding-based approach (Section 4.5.1) and the comparison to the GNN-based approach (Section 4.5.2). This chapter closes with a conclusion section (Section 4.6).

4.2 Learning Approaches in LEGCS

While I introduced LEGCS in the previous chapter, I take a deeper look into the learning phase now. In particular, I compare the current approach for predicting graph properties (i.e., GNNs) with a proposed approach based on graph embeddings. Both approaches are visualised in Figure 4.1.

4.2.1 Current Approach for Learning Phase: GNN

In the previous chapter, I addressed the current approach on LEGCS using graph neural networks, that I call from this point on $LEGCS_{GNN}$. This approach can use graph structures in the form of a neural network to learn feature vectors for each node that keep information about the node itself and its neighbourhood. Subsequently, we can use this trained GNN to predict graph properties (by pooling properties from the individual nodes). Figure 4.1a presents a visual explanation.

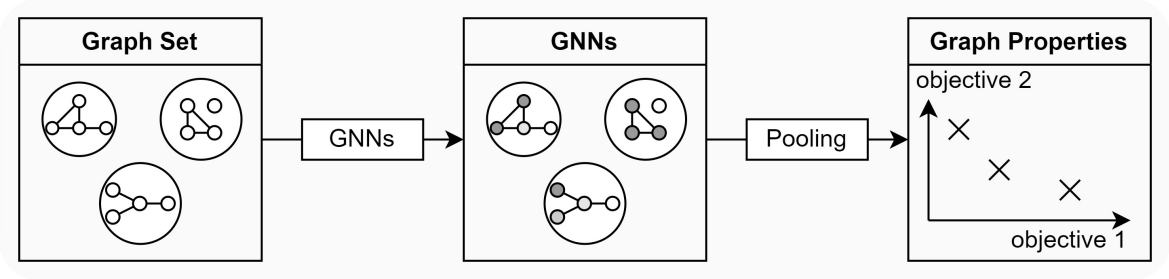
While this approach is able to assist us in the graph configuration space exploration, it takes a large amount of costly training data to get to this point. I hypothesise that a learning approach relying on graph embeddings instead of deep learning, can produce reliable predictions with less training data.

4.2.2 Proposed Approach for Learning: Emb+Reg

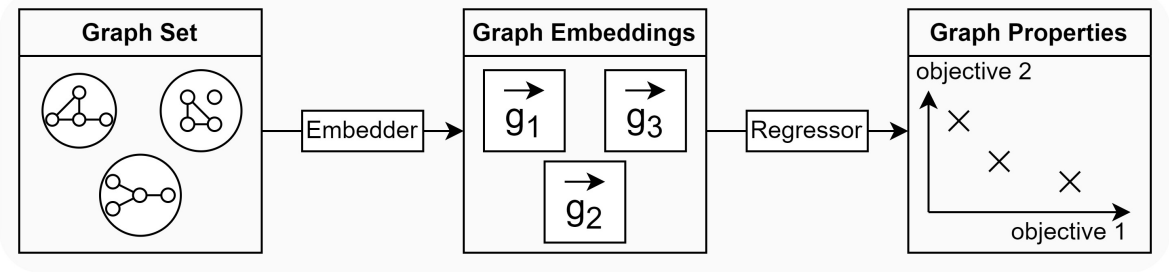
This approach proposes a two-step model for making predictions on graph structures based on the ideas in Section 2.3.2:

1. In the first step, whole-graph embedding maps every graph into a fixed-dimensional vector space. There are various methods to achieve an expressive (i.e., graph-information preserving) and efficient (low embedding time and dimensionality) embedding [2].
2. In the second step, an ML model is trained to produce predictions based on the vector representations of those graphs.

In the following section, I present various concrete embedding and regression techniques for the implementation of $LEGCS_{Emb+Reg}$. I compare various combinations of those techniques in this chapter with respect to prediction accuracy and training set size.



(a) Current Approach: Predicting graph properties based on GNNs. Trained feature vectors of the nodes represented in greyscale.



(b) Proposed Approach: Predicting graph properties based on graph embeddings.

Figure 4.1: Learning strategies to predict graph properties.

4.3 Embedding and Regression Techniques

In this section, I introduce the embedding and regression techniques that I use in context of LEGCS to observe their influence on the predictions and determine a suitable combination for my case studies.

Embedding Techniques

Below, I list the concrete embedding strategies that I use in the experiments in this chapter. In the results, I evaluate how well these strategies keep relevant information on arbitrary and complex graph structures while storing them in low-dimensional vectors in the context of our case study. For my experiment, I compare six concrete embedding techniques (implemented by the Karateclub framework [112]) that I describe below.

- *Graph to Vector (Graph2Vec)* [113] – This method first extracts rooted subgraphs from every node in the graph, and then use a skipgram model to learn an embedding from these subgraphs.
- *Graph and Line Graph to Vector (GL2Vec)* [114] – This method expands Graph2Vec utilising a line graph to (1) handle edge labels, and (2) better preserve structural information in the embedding as opposed to bundling structural and node label information in extracted subgraphs.
- *Spectral Features (SF)* [115] – This method embeds the k lowest eigenvalues of the graph's normalised Laplacian matrix.
- *Family of Graph Spectral Distances (FGSD)* [116] – This technique operates again on the normalised Laplacian matrix of the graph. This method, however, uses the histogram of the Moore-Penrose spectrum of the Laplacian matrix as embedding.
- *Network Laplacian Spectral Descriptor (NetLSD)* [117] – Similar to SF and FGSD, this embedding captures spectral information. It is, however, optimised to embed large graphs efficiently in terms of embedding space and computation time.
- *Local Degree Profile (LDP)* [118] – This method obtains an embedding capturing the degrees of all nodes and their neighbourhoods.

The Karateclub framework offers even more embedding techniques, that I do not con-

sider due to application constraints in the experiment. Embeddings based on diffusion wavelets [119] and random walk weights [120] led to high embedding dimensions with too high computational complexity for the experiments. Invariant graph embedding (IGE) [121] was not applicable due to sparse graphs in my application. Geometric scattering [122] was not applicable as well due to not-connected graphs in the dataset of Case Study I.

Regression Techniques

In this chapter, I compare the accuracy of six commonly used learning models (implemented by the Scikit-Learn library [123]) that I describe below, each model representing a different class of learning models:

- *Ridge* – This linear regression model-based model avoids over-fitting by penalising single large regression coefficients.
- *k-Nearest Neighbour (kNN)* – This proximity-based model obtains predictions by averaging the output variables of the k closest data points in the training data.
- *Random Forest (RF)* – This tree-based model trains a number of decision trees and averages the predictions over these trees to avoid over-fitting.
- *Support Vector (SVR)* – This model introduces a support vector to find a hyperplane that separates classes in another dimension.
- *Multi-Layer Perceptron (MLP)* – This model makes predictions based on a neural network with at least three layers (input, hidden, and output layer) with a non-linear activation function.
- *Histogram-based Gradient Boosting (HGB)* – This ensemble learning model sequentially fits a regression tree on the negative gradient of the given loss function to reduce the prediction error from the previous iteration. The histogram is used for the computationally efficient construction of the regression trees.

4.4 Experimental Set-Up

After describing the existing learning approach for LEGCS using GNNs, and my proposed alternative approach based on graph embeddings (Section 4.2), I report in this section on the experiments to answer the research questions in the introduction of this chapter.

Methodology

Both research questions raised in this chapter ask for a "design, evaluation, or analysis of a particular instance" [100]. I conduct controlled experiments [101] to

1. investigate the influence both steps in $LEGCS_{Emb+Reg}$ have on the prediction accuracy (to answer *RQ 4.1*) in Case Study I, and
2. compare both learning approaches for LEGCS (to answer *RQ 4.2*) with respect to the accuracy of the predictions and the size of the training set (to answer *RQ 4.2*) in both case studies.

Again, I follow the reporting guidelines of Jedlitschka et al. [102], which provide a structure for reporting controlled experiments and questions to cover, e.g., in terms of planning, analysing, and discussing the experiment.

Comparative Study on Embedding Techniques and Regression Models

I aim to answer *RQ 4.1* by conducting a controlled experiment where I apply different combinations of the embedding techniques and regression models introduced in the previous Section 4.3 on the graph configurations from Case Study I (HVAC systems). The goal of these experiments is an efficient prediction of the simulation results in the case study. This efficiency is described with the sampling costs (i.e., in this experiment, the number of necessary training data), and the prediction accuracy. I apply various metrics to assess the prediction quality of regressors that I introduced at the end of Section 3.2.

The overall goal here is to accurately predict the outcome of the simulation for each graph within seconds (while the simulation of a single configuration takes about a minute in Case Study I and 20sec in Case Study II). This way, I can explore the graph configuration space more efficiently by predicting the simulation results of most configurations and only simulating promising configurations.

Comparative Study on Learning Approaches for LEGCS

To answer *RQ 4.2*, I use the insights I gained by answering the previous research question to implement LEGCS based on graph embeddings. In controlled experiments on both case studies I compare this $LEGCS_{Emb+Reg}$ approach to $LEGCS_{GNN}$ from the previous chapter. After comparing the prediction accuracies of both approaches, I also determine the learning efficiency by comparing the number of identified k best graph configurations in a LEGCS-guided graph space exploration.

4.5 Results and Discussion

In this section, I discuss the results to answer *RQ 4.1* by analysing the influence of embedders and regressors, and *RQ 4.2* by comparing the GNN-based and the embedding-based learning approaches in context of LEGCS.

4.5.1 Influence of Embedders and Regressors

First, I analyse the influence that embedders and learning models have on the prediction accuracy with respect to the training set size in context of Case Study I. I observe how the learning models introduced in Section 4.3 perform using different embedding strategies and various numbers of training set sizes.

Influence of Embeddings

Figure 4.2 shows the prediction errors of the learning models in multiple runs using different graph embedding techniques. In each run, all models were trained with the same *500* training graphs.

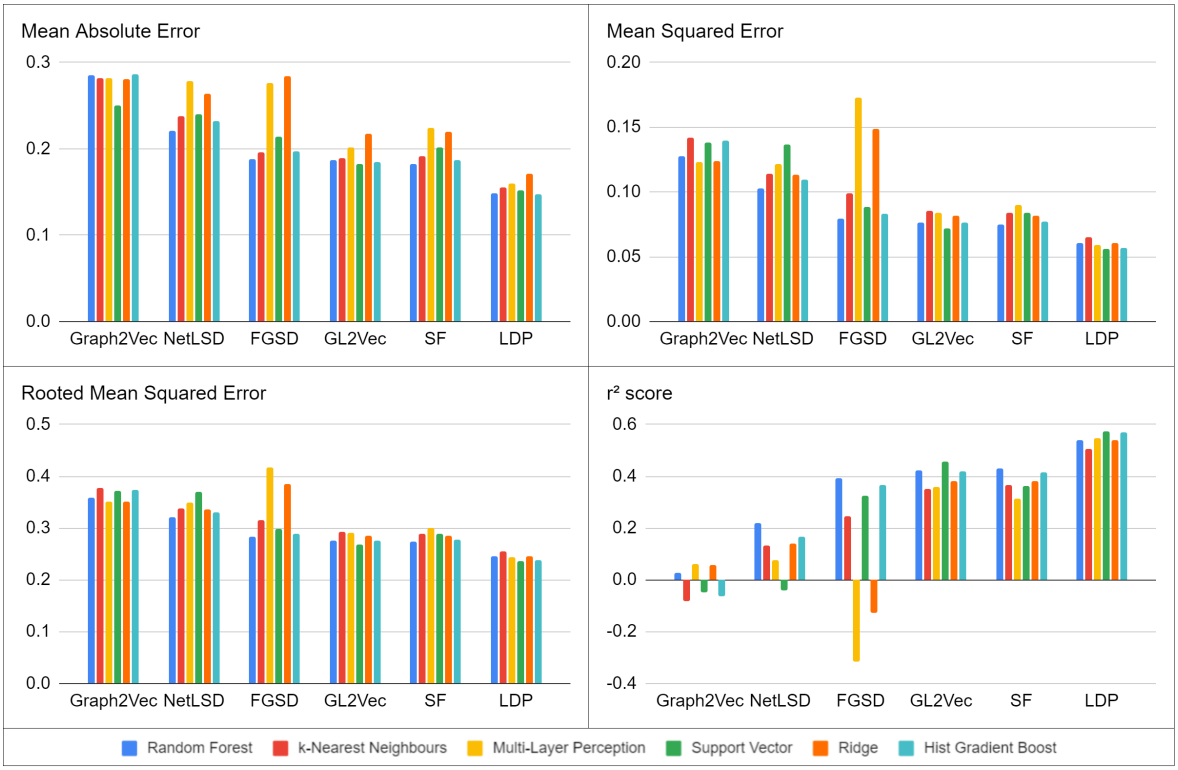


Figure 4.2: Prediction error of learning models trained with the same training graphs embedded through various techniques.

While I can identify Ridge and MLP models as outliers with higher prediction errors (especially using a FGSD graph embedding), the results indicate that (1) the same learning models perform much better using LDP embeddings than with Graph2Vec or NetLSD, and (2) different learning models lead to similar prediction errors when they operate on the same embeddings. Also, my results show that embedding methods that only consider topological data without node or edge labels (such as Graph2Vec) lead to higher prediction errors as learning models operating on embeddings where the algorithm preserves this crucial information. This confirms suggestions from [124] that optimising the data used by a machine learning model can improve the performance more than choosing algorithms or tweaking the model parameters.

Influence of Training Set Size and Learning Model

Table 4.1 and its visualisation in Figure 4.3 show how the learning models perform in multiple runs with a rising number of training graphs. These models operated on LDP embeddings, and the evaluation metric is the mean absolute error. I chose this embedding because, across all embedding strategies in this study, the prediction errors were the lowest for LDP. The results for all evaluation metrics and embedding strategies can be found in the online appendix.

Training Set	**50**	**100**	**200**	**500**	**1000**	**2000**	**5000**
Ridge	0.209	0.190	0.181	0.172	0.169	0.168	0.167
Multi-Layer Perceptron	0.213	0.191	0.179	0.160	0.160	0.153	0.143
Support Vector	**0.193**	0.171	0.164	0.152	0.147	0.144	0.140
k-Nearest Neighbours	0.209	0.182	0.167	0.155	0.151	0.141	0.137
Random Forest	0.199	**0.169**	**0.162**	0.149	0.143	0.137	0.132
Histogram-Based Grad. Boost.	0.258	0.188	0.171	**0.148**	**0.142**	**0.135**	**0.128**

Table 4.1: Mean absolute error of multiple learning models operating on LDP embedding over multiple runs of increasing training set sizes.

We can see how the prediction error decreases in the beginning, relatively quickly. However, it appears that after *500* training graphs we reach a point where the prediction error only decreases minimally over a larger number of training data. While the ridge regression model is outperformed by the other models, overall we can see that all models behave rather similarly in terms of the prediction error, but also the point where more training data does not lead to significantly lower prediction errors.

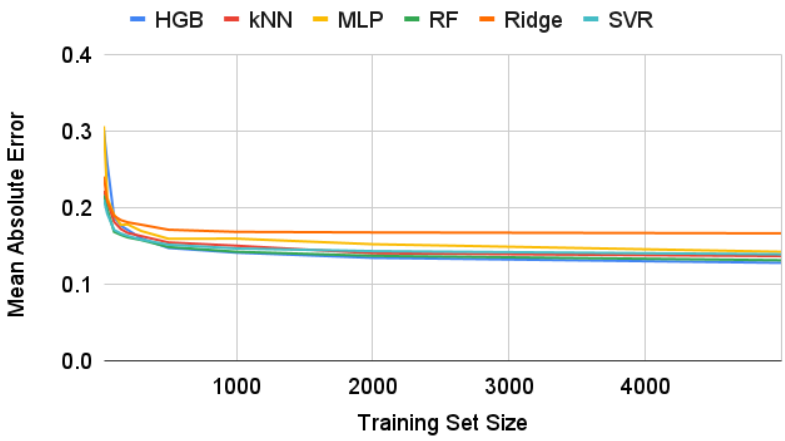


Figure 4.3: Visualisation of Table 4.1.

4.5.2 Comparison of $LEGCS_{GNN}$ and $LEGCS_{Emb+Reg}$

After the insights gained from answering *RQ 4.1*, I choose the following implementation of $LEGCS_{Emb+Reg}$ going into this controlled experiment:

LDP [118] as embedding technique implemented by the Karateclub framework [112] transforms graph configurations into a *160*-dimensional vector space as input for a random forest regressor [125] corresponding to the default implementation by Scikit-learn [123], i.e., *100* trees in the forest, no limit on the depth of the random forest, using the mean absolute error as learning criterion. Additionally, according to the techniques introduced in Section 2.2, I apply a classification and regression technique (CART) with a random forest classifier for the exploration process.

Goal of the Experiment

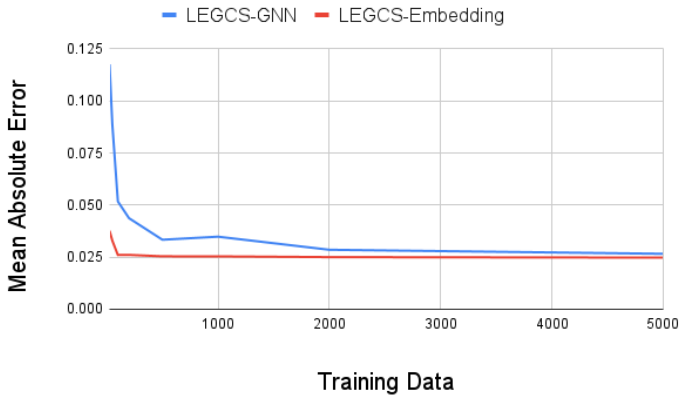
Similarly to the experiment in Section 3.6.2, which focused on GNNs, I aim to identify as many of the k best graph configurations in the datasets with the least amount of simulations as possible. I repeat this experiment for both case studies, first assessing the prediction accuracy of both learning approaches, followed by comparing the guided graph configuration space exploration.

Prediction Accuracy

First, I assess the capability of the algorithms to learn and predict the HVAC performance or average travel time, respectively, based on a sample of graph configurations with mea-

surements. Figure 4.4 shows the prediction accuracy (assessed by the mean absolute error) for both learning strategies over various sizes of training sets. All models were trained with training sets of varying sizes (between *25* and *5000*) and tested on the remaining unseen graphs (*35,000* graphs for Case Study I, and *5,000* graphs for Case Study II).

(a) Learning Efficiency of LEGCS in Case Study I (HVAC systems).



(b) Learning Efficiency of LEGCS in Case Study II (Traffic Networks).

Figure 4.4: Comparison of $LEGCS_{GNN}$ and $LEGCS_{Emb+Reg}$ by learning efficiency.

In Case Study I, the embedding-based approach using a random forest regressor operating on LDP graph embeddings delivers accurate results for low amounts of training data. I did find, however, that adding more and more training data did not significantly close the gap between simulation and prediction results. This is in contrast to the GNN approach, where we can see that we need more training data to get equally accurate predictions, and, eventually, it outperforms $LEGCS_{Emb+Reg}$ in terms of accuracy.

Similarly, in Case Study II, the embedding-based approach exploits the information from the training data more efficiently. Here, however, we cannot find the GNN-based approach outperforming the proposed approach in terms of accuracy with enough training data.

Implications for Graph Space Exploration

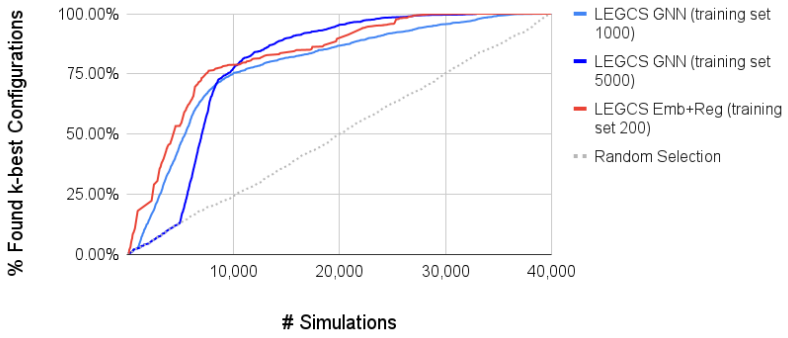
After assessing the prediction quality of $LEGCS_{Emb+Reg}$, I now apply it to guide us through a graph configuration space exploration. The results compared to the first approach are shown in Fig 4.5.

In Case Study I, we can see how the embedding-based approach outperforms the GNN-based approach in the early stages of the graph space exploration, because we can move earlier (after only 200 simulations) from simulating configurations to train the learning model to simulating configurations with promising ML predictions. We can now identify 50% of the k best graph configurations by simulating only 11.4% of the dataset (4,560 simulations, including training data), and 75% of the k best graphs with 18.8% of the simulations (7,520 simulations). At later stages of the exploration process, the GNN approach outperforms the embedding-based approach.

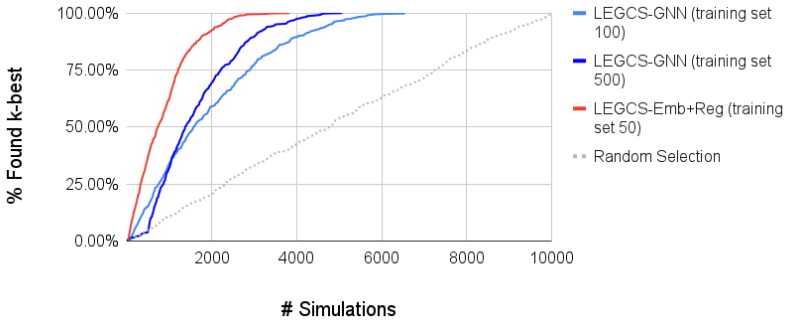
Figure 4.5c is a further visualisation of the $LEGCS_{Emb+Reg}$'s efficiency. Blue shows 200 randomly selected graphs in the dataset as we use them to train the learning model, while red shows the next 200 simulations suggested by the learning model.

In Case Study II, we can also see an improvement by applying $LEGCS_{Emb+Reg}$ (see Fig. 4.5b). We can now identify 50% of the k best graph configurations after simulating 7.4% of the dataset, 75% after 12.1% of simulations, and all of the graph configurations with the most desirable properties after simulating 38.3% of the dataset. Thus, it is also outperforming the GNN-based approach in this case study.

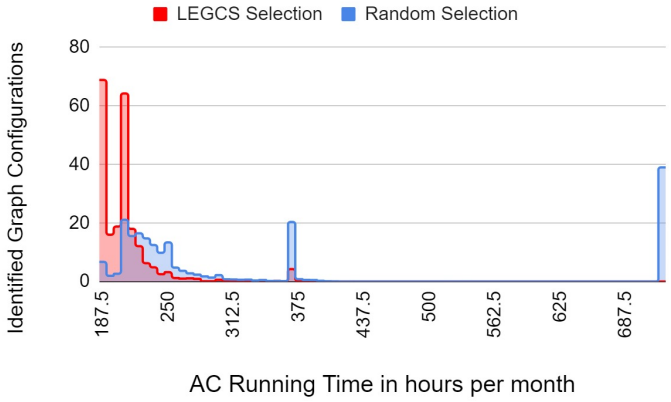
Table 4.2 puts these numbers into context by presenting the average simulation times to identify the k best graph configurations in both case studies using either selection strategy.



(a) Graph space exploration process in Case Study I (HVAC systems).



(b) Graph space exploration process in Case Study II (traffic networks).



(c) The graph configuration space for 200 graphs suggested by $LEGCS_{Emb+Reg}$ compared to a random selection.

Figure 4.5: Graph space exploration process with $LEGCS_{GNN}$ and $LEGCS_{Emb+Reg}$.

		sim time to identify 50% of the k best graph configurations	**sim time to identify 75% of the k best graph configurations**
Case Study I	Random Selection	13d 21h	20d 20h
	$LEGCS_{GNN}$ Selection	3d 20h	6d 14h
	$LEGCS_{Emb+Reg}$ Selection	3d 3h	5d 5h
Case Study II	Random Selection	27h 47min	41h 40min
	$LEGCS_{GNN}$ Selection	7h 40min	12h 20min
	$LEGCS_{Emb+Reg}$ Selection	4h 7min	6h 43min

Table 4.2: Simulation time needed in the exploration process to identify 50% and 75% of the k best graph configurations in both case studies using multiple selection strategies.

4.6 Conclusion

Exploring a space of complex graph structures requires regression analysis on graphs to narrow down the number of possible candidates through learning graph configuration spaces. Chapter 3 proposed LEGCS based on graph neural networks, a deep learning technique that requires large amounts of training data.

In this chapter, I have proposed a new approach to learning graph configuration spaces that embeds crucial structural information of graphs into vectors and applies regression analysis on those vectors.

First, I analysed the influence that embedding and regression have on the performance of the learning model. Since the regressors operate on the graph space embedded into the Euclidean space, the choice of embedding algorithm fundamentally affects the performance of the exploration approach. I observed how the same learning models perform significantly differently for different embedders while different methods may perform similarly for the same embedders. One explanation that seems to be plausible is that the algorithms can only exploit the information the embeddings can preserve. Since Euclidean data structures cannot fully capture complex graph structures, every method we build after graph embedding in a pipeline is limited by the information the embedding is able to capture.

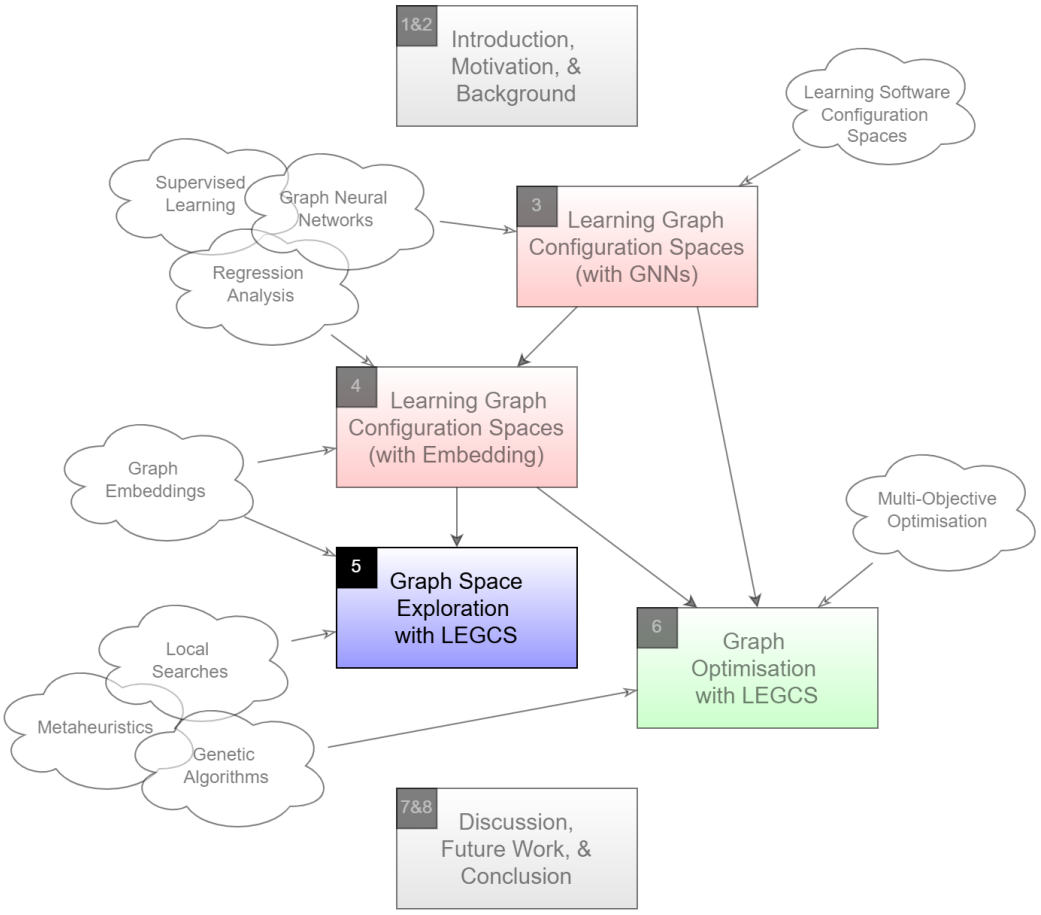
Complex tree-based regression models performed best in terms of accurately predicting the AC usage of graph configurations in our HVAC application. However, the differences in accuracy and training efficiency seem only marginal for different parameters within one tree-based regression algorithm or even for different regression techniques compared to the impact of the underlying embedding method.

In my proposed $LEGCS_{Emb+Reg}$, I have replaced GNNs with this two-step model of embedding and regression. I have shown in two case studies on HVAC systems and traffic networks that this new approach provides more accurate predictions for low amounts of training data and similar prediction errors for large amounts. For a LEGCS-guided graph space exploration in two case studies, $LEGCS_{Emb+Reg}$ improves our search from identifying 75% of the k best graph configurations in the first 24% of simulations to only 18.8% of simulations in the first and 12.1% of simulations in the second case study.

Now that we have the basic techniques to efficiently learn graph configuration spaces, I apply them in a bigger context. In the following chapters, I explore the option of supporting metaheuristics with LEGCS as (1) a more sophisticated exploration technique for low simulation budgets and (2) an optimisation technique to identify graph configurations with desirable NFPs outside of the initial datasets.

Chapter 5

Graph Space Exploration with LEGCS



5.1 Introduction

The previous chapters focused on efficient learning (i.e., using only low amounts of training data and achieving low prediction errors) of graph configurations. Now, I apply the insights gained in these chapters to sophisticated exploration strategies for low simulation budgets to Case Study I in this chapter, and optimisation strategies to Case Study II in the following chapter.

First of all, I investigate applying single solution-based metaheuristics, i.e., local search techniques, with the help of graph embedding to make efficient use of costly simulations to identify well-performing candidates (which I call the *k* best graphs) in a graph configuration space to answer:

RQ 5.1 How can we apply metaheuristics to find as many of the k best graphs as possible in a configuration space using only a limited number of measurements?

In Chapter 3, I have shown (using the same problem case) that regression analysis based on GNNs can support learning graph configuration spaces. Following that, in Chapter 4, I proposed a variation based on embeddings to find this approach leading to more efficient training of the learning model. For simplicity, I shorten $LEGCS_{Emb+Reg}$ to LEGCS in this chapter. Now, I study how efficiently I can apply this learning to filter the space to the *k* best graphs *within a simulation budget*. This leads to the research question:

RQ 5.2 How efficiently can we identify k best graphs with LEGCS using a limited number of measurements?

Eventually, I consider combining metaheuristic search strategies with learning-based exploration. I investigate whether an ML-supported metaheuristic search of *k* best graphs in a graph configuration space with a limited number of measurements can outperform approaches based solely on metaheuristics or solely on LEGCS.

RQ 5.3 How does (1) a learning-supported metaheuristic search for the k best graphs in a graph configuration space compare with (2) a metaheuristic search on embeddings and (3) a LEGCS search on a limited number of measurements, in terms of identified graphs?

Contributions. This chapter makes the following contributions:

1. I propose leveraging graph embeddings to enable the use of local search for efficient graph space exploration.
2. I investigate the application of LEGCS from prediction in the previous chapter now to exploration within a simulation budget.
3. I devise a graph space exploration approach that combines the search power of metaheuristics with the predictive efficiency of machine learning in a case study.

The results of this chapter show that a LEGCS-supported metaheuristic search can efficiently use the simulation budget to spend over-proportionally many simulations of a limited budget on the k best graph configurations.

The remainder of the chapter is structured as follows: I explain the exploration approaches I compare in this chapter (Section 5.2). I then present the research design, in particular the experimental setup to answer the research questions (Section 5.3). I answer one research question in each of the following three sections (Sections 5.4, 5.5, and 5.6), discuss the results (Section 5.7), and close with a conclusion (Section 5.8).

5.2 Proposed Approach

In this section, I give a brief overview of the different approaches I investigate and evaluate in this chapter to answer the three research questions raised in Section 5.1. This overview also includes a quick summary in Table 5.1, and a visual explanation in Figure 5.1. In particular, I describe the new proposed approach combining (i) a metaheuristic local search with (ii) ML on graphs to answer *RQ 5.3*.

	Meta-heuristic	**Search Approach**	**Machine Learning**	**Training Strategy**
RQ 5.1	✓	Locality in the embedded graph space	-	-
RQ 5.2	-	-	✓	Once
RQ 5.3	✓	Locality in the solution space by ML predictions	✓	Iteratively

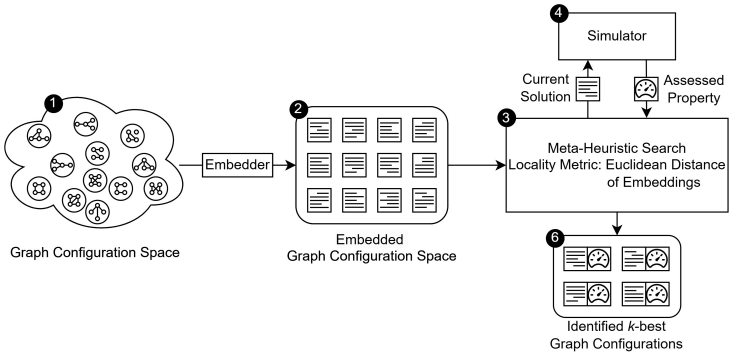
Table 5.1: Search strategies for graph configuration space exploration.

In Section 5.4, I apply a number of local search algorithms on graph embeddings to answer *RQ 5.1* (see Figure 5.1a). That means that I embed the graphs into vectors so that I have an intuitive locality metric (Euclidean distance between the embeddings), and I can apply algorithms such as hill climbing to search for the k best graph candidates.

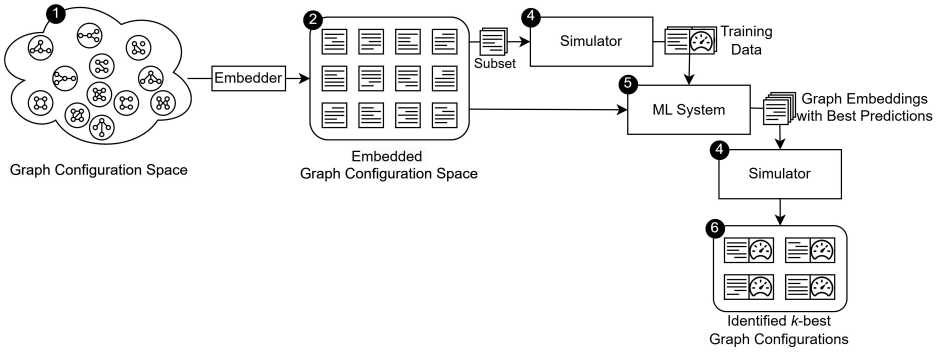
In Section 5.5, I investigate the LEGCS approach introduced in Chapter 4 that uses regression analysis on graph embeddings (see Figure 5.1b). After training the ML model, I can use predictions to search for the k best graph candidates.

Training these models to accurately predict graph properties, however, requires high computational costs. I propose reducing this computational effort by re-training the prediction model during the exploration process in the context of a metaheuristic search to answer *RQ 5.3* in Section 5.6.

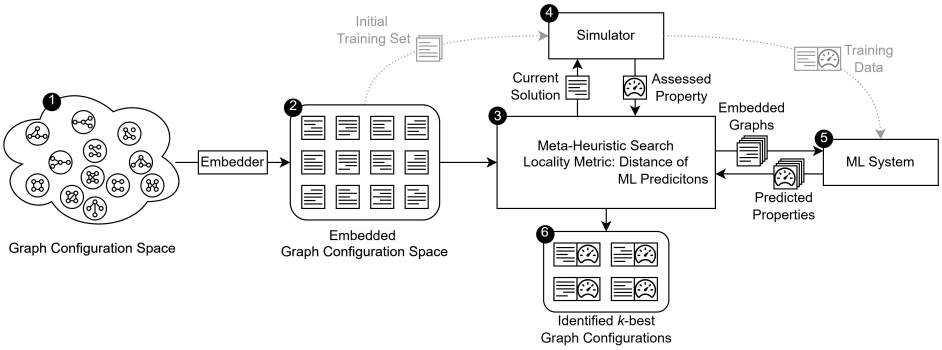
Figure 5.1c presents an overview of my proposed approach. First, I map all graphs in our ❶ graph configuration space to vectors in an ❷ embedded space. From here, I run simulations to create a (small) initial training set for the learning model (indicated in grey colour and dotted lines). Now, I conduct a ❸ local search on the graphs. For this local search, I need to determine the locality in the graph space. While in Section 5.4, I explore the possibility of using the Euclidean distance between the graph embeddings as a measurement of locality, I analyse in Section 5.6, whether ML predictions are a suitable criterion. That means that the



(a) *RQ1*: Metaheuristic search on graph embeddings.



(b) *RQ2*: Machine Learning-based search.



(c) Machine Learning-supported metaheuristic search.

Figure 5.1: Search strategies to find the k best graph configurations with a limited simulation budget.

performance of the current solution candidate (i.e., graph) is assessed using measurements from a ❹ simulator, and then I use predictions of graph properties from a ❺ machine learning model to select a new graph to become our new pivot solution. I iteratively ran this process until I reached the simulation budget. At this point, I have identified ❻ a subset of k graphs with the best properties according to the optimisation goal. Using this approach, I aim to spend less computational resources preparing the exploration and instead improve the ML model (and thus the efficiency of the search) during the exploration process.

5.3 Research Design

In this section, I talk about the methodology in this chapter. This includes the methodology and the experimental setup including the technical aspects.

Methodology

The research questions formulated in Section 5.1 ask for a "design, evaluation, or analysis of a particular instance" [100]. Within a case study, I conduct controlled experiments [101] on a graph space exploration application that:

(i) compare multiple embedding strategies and metaheuristic algorithms (Section 5.4),

(ii) use results from Chapter 4 to use LEGCS (Section 5.5), and

(iii) combine metaheuristic algorithms with regression analysis on graphs (Section 5.6).

The goal of my experiments is to efficiently identify the k best configurations in the graph space while using a limited (i.e., low) number of expensive evaluations (i.e., through time-intensive simulations).

And, as in the previous chapters, I the reporting guidelines of Jedlitschka et al. [102] for the controlled experiment.

Experimental Setup

In this chapter, I again use the same dataset, measurements, and overall goal as in Case Study I introduced in Section 3.5.1. For the embedding-supported metaheuristic, I use the techniques that I already introduced in Section 4.3. Within the case study, I compare the number of simulations within the budget where the assessed AC usage is minimal. I run each strategy *30* times to determine the average performance of each method – each run with a unique random seed, and using the same random seeds over different strategies. Since averaging the results can be distorted by outliers, I additionally perform a Mann–Whitney U test [126] to assess the quality of the exploration methods over the various runs.

This test first ranks the data points in two samples and counts for each point in the samples the number of lower data points in the other sample to calculate the U statistic of both samples (by summing up these counts). Now, the p-value represents the probability of

observing a U statistic as extreme as (or even more extreme than) the one calculated from our samples. We compare this p-value against a significance level α (usually around 0.05). If the p-value is below α, we can say that the difference of the medians in both samples is statistically significant. This test allows us to determine if the number of identified k best graph configurations of one exploration method performs not only better on average, but also statistically significantly better than another exploration method.

5.4 Metaheuristic Graph Space Exploration on Graph Embeddings

In this section, my aim is to answer the first research question raised in the introduction by applying multiple local search strategies on graph embeddings to efficiently identify the k best graph configurations in the dataset.

Locality Metric

In order to apply the local search techniques that I introduced in Section 2.4.1 to our set of *40,000* graph configurations, I first need a metric of locality within our complex graph space. One straightforward approach to creating locality in a graph space is transforming it into an Euclidean space by embedding the graphs, i.e., mapping each graph to a vector. This way, I can use the Euclidean distance between two vectors as similarity measurement between two graphs. For the experiments in this section, I apply the embedding strategies introduced in Section 4.3. Starting from a random solution, I iteratively define a neighbourhood of the current solution as the closest n solutions (in terms of Euclidean distance between embeddings). I determined the best value for n in a pre-study using a parameter grid.

I could have instead performed modifications on graphs during the local search, but that would have required constant healing to ensure the domain constraints were still met. The use case in this chapter avoids this issue by only operating on a graph set where each graph fulfils the constraints, and we achieve locality through embeddings in the Euclidean space. In the following chapter, however, I do make modifications on the dataset.

Results

In my first experiment, I apply iterative hill climbing, simulated annealing, and Tabu search on my graph configuration space (embedded by various embedding strategies), optimising the number of identified k best graphs with simulation budgets of *100*, *500*, and *1,000* simulations. To put it in perspective, I compare the results with an approach picking simulations from the space at random to identify the top 3.36% graphs in the dataset in terms of the lowest AC usage per month. I summarise the results of this experiment in Table 5.2.

We can see how the performance for the same algorithms varies a lot depending on the underlying graph embedding the metaheuristic is operating on. Interestingly, I observe

that iterative hill climbing outperforms the more sophisticated search heuristics (i.e., SA and TS). For SA and TS, I apply mechanisms to get out of local optima, which can be cost-intensive in terms of using up the simulation budget (i.e., they "burn through" simulations on the budget), while IHC takes the approach of "just" starting again from a new arbitrary graph configuration in the solution space. Note, that even though the number of simulations might appear to be a somewhat artificial metric, it directly corresponds to considerable time "wasted" if the simulations are not chosen carefully.

Meta-Heuristic	**Embedding**	**100 Sims**	**500 Sims**	**1,000 Sims**
Random Selection	None	2.90	17.60	35.73
Iterative Hill Climbing	Graph2Vec	4.57	26.47	52.27
	GL2Vec	5.57	43.20	**102.77**
	SF	6.37	41.97	71.03
	FGSD	5.37	33.40	56.07
	NetLSD	4.20	18.37	38.50
	LDP	**7.20**	**46.23**	87.33
Simulated Annealing	Graph2Vec	5.33	24.60	46.27
	GL2Vec	4.67	29.87	**72.30**
	SF	3.37	**34.97**	70.23
	FGSD	**5.77**	20.23	29.27
	NetLSD	3.30	17.47	40.47
	LDP	3.20	21.27	41.87
Tabu Search	Graph2Vec	6.47	26.7	57.43
	GL2Vec	**6.73**	41.10	**95.23**
	SF	4.23	**41.20**	85.17
	FGSD	5.90	25.93	56.27
	NetLSD	4.57	23.63	45.60
	LDP	5.47	33.77	59.40

Table 5.2: Identified k best graphs by multiple metaheuristic searches on various graph embeddings within different simulation budgets.

5.5 Machine Learning-Based Graph Space Exploration

This section focuses on answering the second research question on graph space exploration using regression analysis by comparing the approach introduced in Chapter 3 to the benchmark set in the previous Section 5.4.

Learning-Based Graph Space Exploration

I compare LEGCS with the metaheuristics in the previous section in terms of identifying the k best graphs in a graph space exploration problem on a limited simulation budget. I had multiple runs of learning-based graph space exploration aiming to identify as many k best graph configurations as possible over the same simulation budgets as in the previous section. This process includes a trade-off to use as much training data as necessary to train the learning model satisfactorily while at the same time leaving enough of the simulation budget to identify the k best graph configurations. I identify them by simulating graphs with promising predictions from the learning model.

Following the results of Section 4.5.1, I apply the regression analysis on the LDP embedding. In terms of regression, I experiment with a random forest regressor and a histogram-based gradient boosting regressor. I have shown in Table 4.1 that random forest performed best for small training set sizes while histogram-based gradient boosting has the lowest prediction error for larger training sets. I have summarised the performance of these variants in terms of graph space exploration in Table 5.3.

Results

We can see in Table 5.3 that learning-based graph space exploration is more efficient than exploration based on metaheuristics on embeddings, approximately by a factor of 2. For small simulation budgets, the random forest regressor proves to perform best, while for larger simulation budgets, the histogram-based gradient boosting regressor outperforms all other approaches since I can use more of the budget to train the learning model and achieve better predictions to use for the search of the k best graph configurations.

Method	**Training Set**	**100 Sims**	**500 Sims**	**1,000 Sims**
Random Selection	-	2.90	17.60	35.73
Iterative Hill Climbing (LDP)	-	7.20	46.23	87.33
Iterative Hill Climbing (GL2Vec)	-	5.57	43.20	102.77
Learning-Supported Graph Space Exploration (LDP-RF)	50	**14.00**	89.37	151.93
	100	-	**100.87**	178.77
	200	-	99.20	220.03
	350	-	61.47	198.40
	500	-	-	179.90
Learning-Supported Graph Space Exploration (LDP-HGB)	50	3.5	28.53	51.57
	100	-	53.37	122.07
	200	-	98.97	**226.67**
	350	-	63.03	205.93
	500	-	-	184.40

Table 5.3: Identified k best graphs by learning models using various numbers of training set sizes within different simulation budgets.

5.6 ML-Supported Metaheuristic Graph Space Exploration

I answer the third and last research question in this chapter by implementing the proposed approach introduced in Section 5.2.

Comparative Study on Learning-Supported Metaheuristics

For the third and last experiment, I build a hybrid approach using ideas from local searches and techniques of learning graph configuration spaces. This hybrid uses a learning model that I only train with a small amount of training data. Then, I evaluate "neighbours" as we do in local searches, but I define these neighbours as the graphs in the space with the closest ML prediction to the best current solution in the search. I then use the measurements from these neighbours to re-train the learning model. I iteratively predict, pick promising neighbours, measure, and re-train the model until I reached the simulation budget. Eventually, I count how many k best graph configurations I have identified during this search and compare this result to the methods in the previous sections using either metaheuristics or graph learning.

Results

I applied the hybrid approach, and following the results from the previous experiment, I use LDP for embedding and the random forest regressor. I analysed multiple variants using different numbers of initial training sets that I gathered through random sampling, and different "neighbour" sizes (i.e., the number of most promising graph configurations, I explore in each iteration). I have summarised the results of these experiments in Table 5.4.

In these results, we can see how the learning-supported hill climbing algorithm outperforms the other approaches in terms of identified k best graphs in the configuration space over multiple simulation budgets. I can see how starting with a small training set to leave most of the simulation budget for exploration and then go for many iterations of simulating small sets, and re-training the learning model often, leads to the best results.

To put that into perspective, in a dataset of *40,000* graph configurations where I consider only *1,344* of them as k best graphs in terms of AC usage (that makes 3.36% of the whole dataset), I developed and implemented a technique based on the metaheuristic idea of hill

climbing and regression analysis to identify 316.40 of those k best within the first *1,000* costly simulations (that makes 31.64% of the simulations).

Method	Training	Neighbours	100 Sims	500 Sims	1,000 Sims
Random Selection	-	-	2.90	17.60	35.73
Iterative Hill Climbing (LDP)	-	-	7.20	46.23	87.33
Iterative Hill Climbing (GL2Vec)	-	-	5.57	43.20	102.77
Learning-Based Exploration (LDP-RF)	-	-	14.00	100.87	220.03
Learning-Based Exploration (LDP-HGB)	-	-	3.50	98.97	226.67
Learning-Supported Hill Climbing	50	10	16.20	**157.47**	**316.40**
		20	**19.13**	152.10	307.47
		50	13.83	134.87	301.17
		100	-	138.43	296.83
	100	10	-	139.90	313.33
		20	-	137.07	309.13
		50	-	134.67	287.43
		100	-	108.27	272.97
	200	10	-	112.87	287.63
		20	-	107.63	282.57
		50	-	102.00	278.40
		100	-	96.70	256.77
	350	10	-	63.93	232.50
		20	-	66.07	231.33
		50	-	61.83	211.43
		100	-	76.90	215.47
	500	10	-	-	190.50
		20	-	-	184.57
		50	-	-	173.47
		100	-	-	165.83

Table 5.4: Identified k best graphs by learning-supported hill climbing searches with various training set and neighbour set sizes within different simulation budgets.

Statistical Significance

After comparing the averages of thirty runs in the previous sections, I now have a look at the statistical significance of these results. I do that by comparing the p-value of the Mann-Whitney U test of the performances of all approaches presented in this chapter. I have summarised the results in Table 5.5. For each entry in the table, I took the configuration with the best average results, i.e., for learning-based exploration, I took random forest regression for the columns *100* simulations and *500* simulations, but histogram-based gradient boosting regression for the column *1,000* simulations.

	Random	**IHC**	**Learning**	**Learning-HC**
		100 Simulations		
Random Selection	1.0	0.63	$2.60 \cdot 10^{-5}$	$5.46 \cdot 10^{-9}$
Iterative Hill Climbing		1.0	$4.56 \cdot 10^{-3}$	$2.10 \cdot 10^{-5}$
Learning-Based Exploration			1.0	0.06
Learning-Supported HC				1.0
		500 Simulations		
Random Selection	1.0	0.31	$7.90 \cdot 10^{-9}$	$2.92 \cdot 10^{-11}$
Iterative Hill Climbing		1.0	$1.88 \cdot 10^{-4}$	$7.02 \cdot 10^{-9}$
Learning-Based Exploration			1.0	$1.83 \cdot 10^{-4}$
Learning-Supported HC				1.0
		1000 Simulations		
Random Selection	1.0	$2.07 \cdot 10^{-9}$	$6.57 \cdot 10^{-10}$	$2.93 \cdot 10^{-11}$
Iterative Hill Climbing		1.0	$4.11 \cdot 10^{-6}$	$1.20 \cdot 10^{-10}$
Learning-Based Exploration			1.0	$9.88 \cdot 10^{-7}$
Learning-Supported HC				1.0

Table 5.5: P-values of Mann-Whitney U tests performed on the graph space exploration methods in the previous sections.

As common, I interpret the p-values by comparing them to a significance level of *0.05*. For the Mann-Whitney U test conducted on the performances of random selection and iterative hill climbing in case of a small simulation budget (i.e., *100* and *500* Simulations), I can see high p-values, meaning that here a small number of IHC runs performed so well that the average seems much better than the performance for random selection while actually, most runs performed similarly. I can make the same observation on a smaller scale for a simulation budget of *100* between learning-supported graph space exploration and learning-supported hill climbing. In all other cases, I have p-values below the significance

level, meaning that here there are no exceptional runs in the experiments distorting the average performances.

5.7 Discussion

In this chapter, I analysed product design exploration problems where the solution space consists of complex graph structures representing the product configurations, and measurements of individual product properties are computationally expensive (thus limiting the exploration time). I explored multiple ways of identifying products with desirable properties by analysing graph configurations in the context of an HVAC application with limited numbers of measurements:

1. embed the graph space, and apply metaheuristics,
2. embed the graph space, train a learning model, and measure graphs according to the learning model's ranking, and
3. embed the graph space, train a learning model, and apply local search techniques to progressively select graphs for measurements, and re-train the learning model.

I observed how sophisticated metaheuristics (i.e., simulated annealing and Tabu search) were not able to outperform an iterative hill-climbing approach. Since the simulation budget (i.e., the number of measurements available) is limited, I observed it to be more successful to hill climb from multiple starting points than spending simulations on getting out of local optima. I even found this iterative approach so successful that when I applied a hill climb with regression analysis (as the criterion of closeness in terms of performance of the graph configuration), it outperformed a sequential approach of training, predicting, and exploring in that exact order.

5.8 Conclusion

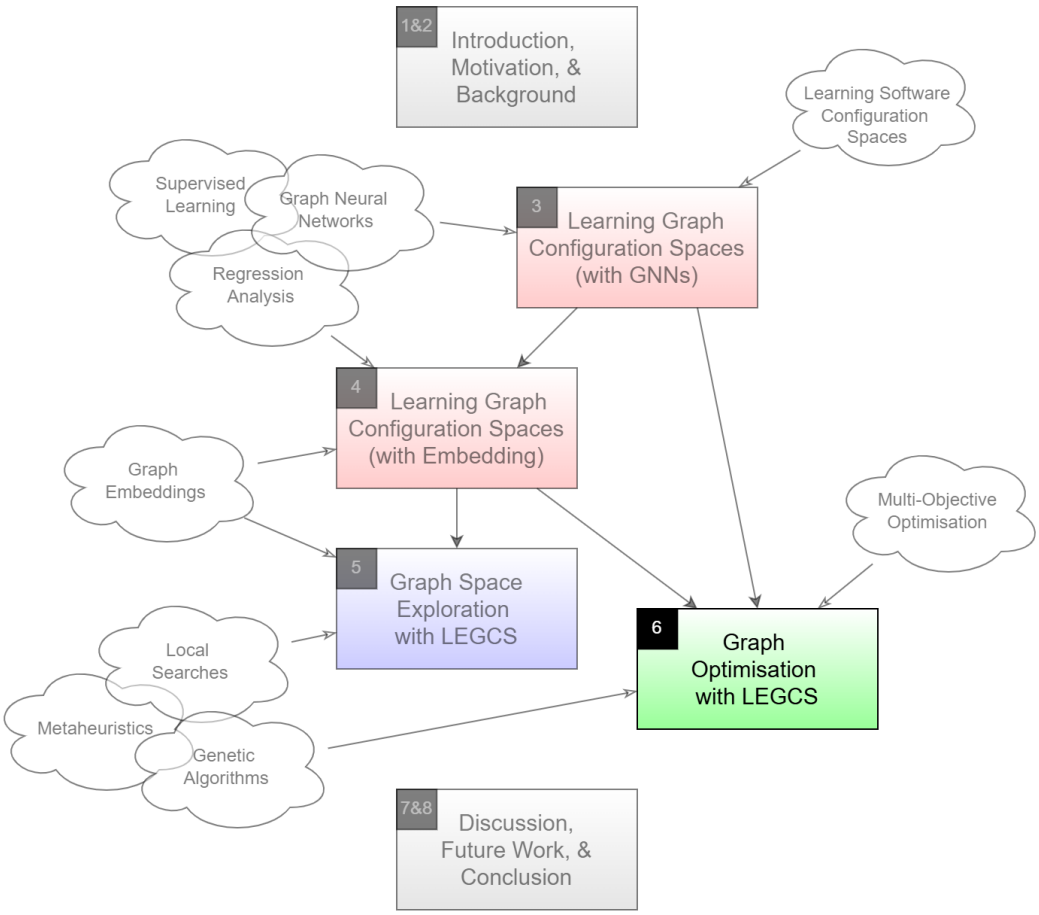
The work presented in this chapter tackles the problem of exploring graph configuration spaces. I aim to find (a set of) graph candidates representing configurations of a system that exhibit desirable properties. I assess these properties with computationally expensive simulations. I presented three approaches to finding these graph configurations in a large space (*40,000* candidates) with different simulation budgets (*100*, *500*, and *1,000* simulations): (*RQ 5.1*) a metaheuristic search, (*RQ 5.2*) a machine-learning-based search, and (*RQ 5.3*) a learning-supported metaheuristic search.

I found that an important component to efficiently explore the graph space is an expressive embedding, and I can claim that for my use case, iterative approaches performed better than sequential approaches, i.e., when a hill-climbing finds a local optimum, it is more efficient to start at a new random solution than using up the simulation budget to get out of the optimum. Even for the learning-based approaches, I found an iterative approach of starting with a small training set, and re-training the system during the exploration is more efficient than a sequential approach of sampling, measuring, training, and exploring.

Future work is needed to investigate whether these approaches generalise to larger graphs, as well as graph configuration problems in other domains.

Chapter 6

Multi-Objective Graph Optimisation with LEGCS



6.1 Introduction

So far, I have established the problem of learning graph configuration spaces, implemented and analysed two learning approaches, and used it in context of one application, i.e. graph space exploration for single objectives in large generated sets of configurations. I have not yet, however, applied LEGCS in an optimisation setting, where I use the LEGCS predictions to generate and evaluate new graph configurations. In this section, I modify Case Study II based on traffic networks that I introduced in Section 3.5.2 to a multi-objective optimisation problem that I aim to solve with a LEGCS-supported genetic algorithm.

Genetic algorithms are a widely used population-based metaheuristic for optimisations in a variety of application domains [62]. These include facility layout, scheduling, inventory control, network design, etc. As soon as we can describe single solutions as "genomes", and formulate a fitness function to assess the quality of these genomes, we can apply genetic algorithms.

It becomes more difficult, however, when (1) single solutions are described as graphs instead of straightforward Euclidean structures (i.e., vectors), (2) assessing the fitness includes multiple objectives instead of one clear goal to optimise towards, and (3) some of these objectives require computationally expensive simulations to estimate the fitness. In this chapter, I present a use case that tackles all three of these complications: a traffic network exploration problem, where we connect 32 towns with a road network while minimising the overall road length and simultaneously minimising the average travel time that we assess with resource-intensive computer simulations.

Since frequent genome assessments are necessary for genetic algorithms, I use predictions of a learning model to replace the majority of the simulations in the optimisation process, and investigate whether this loss of accuracy in terms of fitness assessment is tolerable when it allows a more successful optimisation process on the same computation budget for simulations.

In this chapter, I apply LEGCS to evaluate large numbers of similar graphs during a genetic algorithm to answer the research question:

RQ 6 How does a LEGCS-supported genetic algorithm compare to a plain genetic algorithm (without LEGCS support) when applied to a multi-objective traffic network optimisation problem with respect to the obtained Pareto fronts?

Contributions. This chapter makes the following contributions:

1. I propose a pipeline of supporting genetic algorithms with LEGCS to apply to a traffic network design problem.
2. I report on a controlled experiment that optimises a traffic network to minimise travelling time as well as the overall length of roads. In this experiment, I compare a genetic algorithm with and without LEGCS support.

The results show that replacing a majority of fitness assessments in a genetic algorithm and spend costly simulations only on genomes with promising ML predictions, leads to a loss of accuracy in the fitness assessments that is outweighed by the higher number of generations we can run on the same simulation budget. Giving the LEGCS-supported algorithm more time to evolve as opposed to the GA without LEGCS the genomes led to an increase of the hypervolume indicator in the Pareto front of identified solutions.

The remainder of this chapter is structured as follows: In Section 6.2, I present my proposed approach. After that, I present a refined case study in Section 6.3 that I use for this chapter. In Section 6.4, I present the research design to answer *RQ 6*, and discuss the results of my experiment in Section 6.5. This chapter closes with a conclusion in Section 6.6.

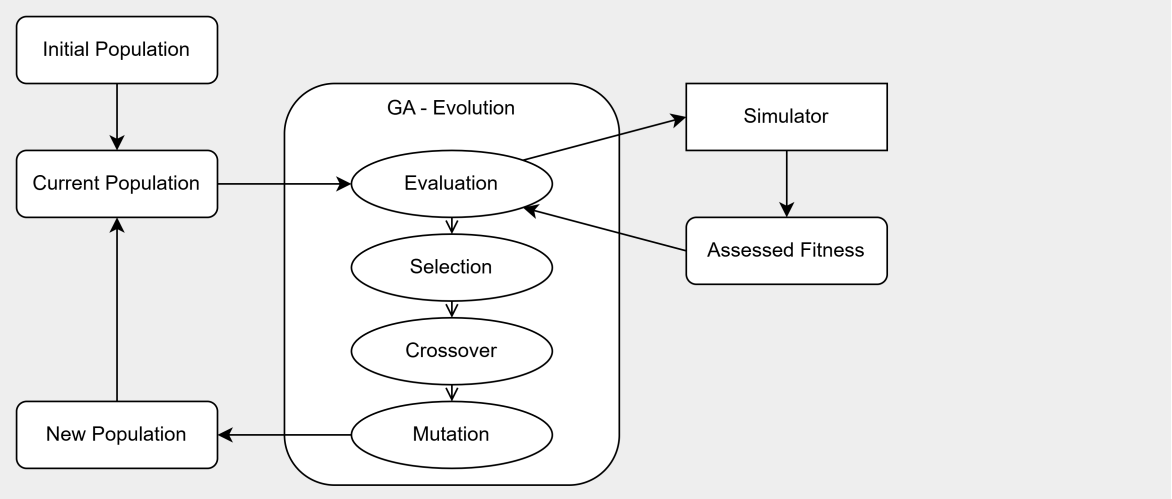
6.2 Proposed Approach: LEGCS-Supported GA

In this section, I show how I aim to accelerate the evaluation process within the GA by replacing the majority of the computationally expensive simulations to assess the fitness of the genomes in each population with predictions from a learning model.

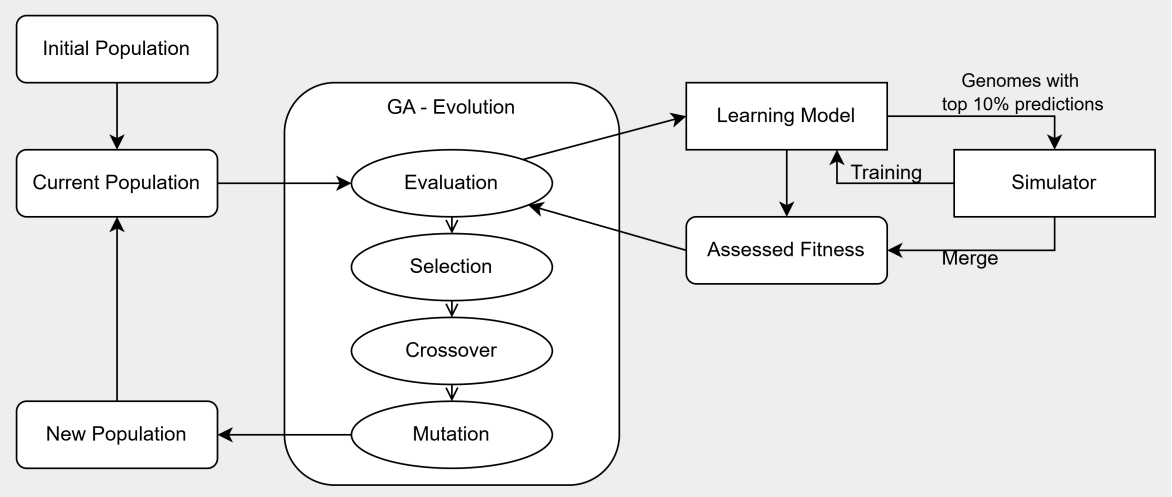
Figure 6.1a shows a first approach to implementing a GA that uses computationally expensive simulations to assess the fitness of individual genomes. The termination of this GA is directly dependent on a simulation budget, i.e., the maximum number of resources I am able to spend on the optimisation process. This motivates my proposed approach.

In Figure 6.1b, I present the proposed approach using LEGCS to limit the number of simulations to 10% of genomes in each population. With this modification, I can manage to increase the iterations of the genetic algorithm by factor ten on the same simulation budget at the cost of less accurate property assessment during the optimisation process.

In an experiment, I observe whether this loss of accuracy in the assessment is tolerable because the increased number of generations would lead the algorithm to find solutions with better properties. I conduct this experiment in which I compare the performance of the GA with and without LEGCS in a MOO setting.



(a) Genetic algorithm using computationally expensive simulations to assess the fitness of genomes.



(b) Proposed approach: replace the majority of simulations with learning model predictions.

Figure 6.1: Genetic algorithms we compare within this study.

6.3 Case Study II-B: Multi-Objective Traffic Network Optimisation

In this section, I describe how I expand from the problem of exploring a set number of traffic networks on a simulation budget that I presented in Section 3.5.2 to a traffic network optimisation problem in which I apply a genetic algorithm. I start by phrasing the goals within the case study, the assessment of configuration properties, and how I apply a GA to this graph optimisation problem. I close this section with how I evaluate my proposed algorithm of LEGCS-supported GA.

Goals within the case study

Similarly to the case study in Section 3.5.2, I work on traffic networks connecting *32* towns and cities in Ireland, and I aim to find connecting streets so that:

- the travel times are low (ideally, that means low travelling distances and high traffic flow), and
- the number of costly simulations to assess these travel times are low.

While these goals work well for an exploration problem on a given set of traffic networks, an optimisation algorithm modifying the dataset, I work on, would search towards a complete graph. In order to make this problem more challenging, I introduce another goal that conflicts with minimising the travel times of cars in the network. I want to find traffic networks where:

- the overall distance of all streets is low (ideally leading to low costs to build and maintain streets).

I apply multi-objective optimisation to find a Pareto front of solutions with desirable properties that satisfy both conflicting goals.

Traffic Network Generation

For the initial populations, I use subsets of the graph configuration, I generated for the case study in Section 3.5.2. In the process of evolutionary computation, the algorithm generates new traffic networks based on crossover and mutation. To ensure that the constraint of this

use case (i.e., strongly connected graphs) is met by those new networks, I apply healing to all graph configurations before evaluation by adding the minimum number of edges to the graph configuration that makes it strongly connected.

Configuration Assessment

Similarly to the case study on traffic network exploration, I assess the distance d and the average speed s of each car c in the network to determine the overall travel time. Additionally, I now assess the overall road length in the street network by adding up the edge weights w. Mathematically speaking, the two minimisation functions of this optimisation problem are:

$$\min f_1(G) = \sum_{c \in C} \frac{d_c}{s_c}$$

$$\min f_2(G) = \sum_{e \in E} w_e$$

I apply multi-objective optimisation by keeping the objectives separate and optimising towards both objective functions:

$$f_{MOO}(G) = \begin{bmatrix} f_1(G) \\ f_2(G) \end{bmatrix}$$

6.4 Research Design

The research question raised in the introduction asks for a "Design, evaluation, or analysis of a particular instance" [100]. After I presented my proposed approach to improve the performance of a genetic algorithm that is limited by the number of fitness assessments in Section 6.2 and refined the case study I operate on in Section 6.3, I discuss in this section the technical details of the controlled experiment I conduct to answer *RQ 6* following the guidelines of reporting on a controlled experiment [102]. I start with how I apply a genetic algorithm to the multi-objective traffic network optimisation problem, and how I evaluate the performance of the genetic algorithm both with and without LEGCS support.

Application of a genetic algorithm to traffic network optimisation

For the refined case study, I apply a GA to a graph configuration problem, in this case: a traffic network optimisation. Since the configurations (i.e., street networks) only differ in their edges, and the nodes of *32* towns stay the same, I decided to apply an edge encoding. Since my configuration goal is connecting the same *32* nodes, we can look at this configuration as a combinatorial problem. In my encoding, I give every possible edge in the graph configuration a unique index. The graph of one street network is now encoded as a list of all possible edges sorted by their individual indices, where *0* represents edges that are not included in this particular graph, and any natural number represents the number of lanes on this specific street.

For the multi-objective genetic algorithm, I use the NSGA-II algorithm [127] – implemented by JMetalPy [128] with normalised objective functions. NSGA-II is one of the most popular multi-objective optimisation algorithms as it is commonly used as a base algorithm for comparing new algorithms [129]. My simulation budget is set to *5,000* simulations (i.e., the number of simulation I can run within 24h), so in a GA without LEGCS, I work with *10* iterations on populations of *500* genomes, while in the LEGCS-supported GA, I work with *100* iterations and the same population size, where only *50* genomes are simulated in each iteration. I use a non-dominated ranking in the LEGCS-supported GA to determine these *50* genomes, and only consider simulated genomes in the Pareto fronts in the results section. I use simulated binary crossover and polynomial mutation. If a solution is not strongly connected (i.e., one cannot reach every node from any node in the network), a healing algorithm

adds the minimum number of edges to fulfil this requirement.

Algorithm evaluation

I include LEGCS in this optimisation process to assess more graph candidates without using more computationally expensive simulations. As benchmark, I compare this approach against a genetic algorithm relying on the same amount of simulations without LEGCS. In terms of multi-objective optimisation, I take the hypervolume indicator of the identified Pareto fronts as a performance measurement of the genetic algorithm. I include nine runs of both approaches on the same datasets to rule out an imbalance of the performance due to random decisions made in crossover and mutation operations.

6.5 Results and Discussion

In this section, I present and discuss the results of my controlled experiment to answer *RQ 6* in context of Case Study II-B.

Learning Accuracy

The idea of replacing simulations with ML predictions stands and falls with the reliability of these predictions. We need to make decisions with a trade-off here: keep the number of simulations low so that we can run the GA over more generations on the same simulation budget while also sufficiently re-training the learning model to keep the prediction error low and, thus, have the GA optimise efficiently on accurate predictions, even when the graphs in each generation shift away from the initial population that we trained the learning models on.

In Section 4.5.2, I compared the initial prediction errors of $LEGCS_{GNN}$ and $LEGCS_{Emb+Reg}$, and with my setup of training the models on the initial population (*500* graphs), and re-training them with *50* graphs per generation every *10* generations, the learning models kept the MAE consistently under *0.05* and the coefficient of determination above *0.9*.

Improvement in Traffic Network Optimisation

In this section, I compare the performance of the GA in a MOO with and without the support of LEGCS. This performance is assessed by the average hypervolume indicator over nine runs of the algorithm.

As we can see in Table 6.1, the GA with $LEGCS_{GNN}$ outperforms the original GA after 50% of the simulations according to the hypervolume indicator, GA with $LEGCS_{Emb+Reg}$ already after 30% of the simulations.

In Figure 6.2, we can see the Pareto fronts of the identified solutions of the algorithm during the optimisation process. The shown run is the one with the median hypervolume indicator improvement to ensure that I have not picked a run whose behaviour is an outlier. We can see that while the overall number of non-dominated solutions in each iteration is lower in the LEGCS-supported approach (this might be due to only including solutions with simulated fitness assessments), the distance between the Pareto fronts, and thus, the

performance according to the hypervolume indicator, is much higher. This distance between the Pareto fronts describes the progress of the optimisation during the same number of simulations.

Hypervolume indicator	**Genetic Algorithm**	**GA with** $LEGCS_{GNN}$	**GA with** $LEGCS_{Emb+Reg}$
initial training	0.607	0.607	0.607
after 500 simulations	0.606	0.621	0.642
after 1,000 simulations	0.618	0.634	0.652
after 1,500 simulations	0.625	0.640	0.661
after 2,000 simulations	0.633	0.650	0.661
after 2,500 simulations	0.638	0.663	0.666
after 3,000 simulations	0.643	0.666	0.671
after 3,500 simulations	0.646	0.670	0.678
after 4,000 simulations	0.650	0.676	0.676
after 4,500 simulations	0.654	0.676	0.682
after 5,000 simulations	0.656	0.683	0.686

Table 6.1: Improvements on the hypervolume indicator over the course of the GA. The values are averages over nine runs.

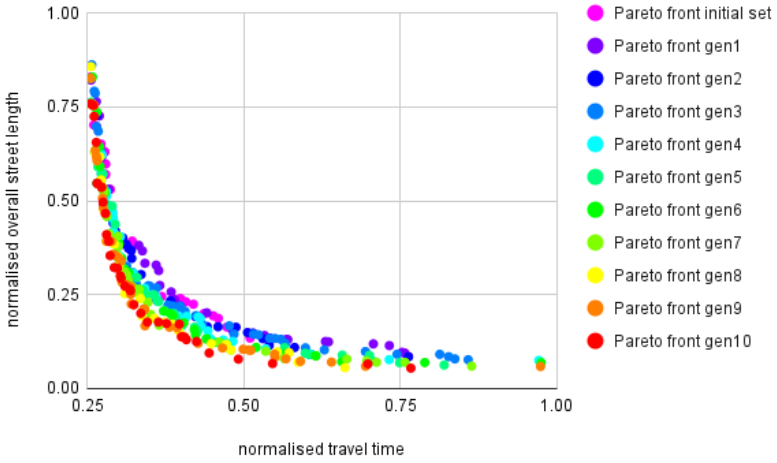
Statistical Significance

After comparing the average runs above and having a closer look at the run with the median improvement on the hypervolume indicator, I now take a look at the statistical significance of these results. Similarly to the previous chapter, I compare the p-value of the Mann-Whitney U test on the results to make sure exceptional runs did not distort the results. As input, I take the hypervolume indicator of all runs after *5,000* simulations.

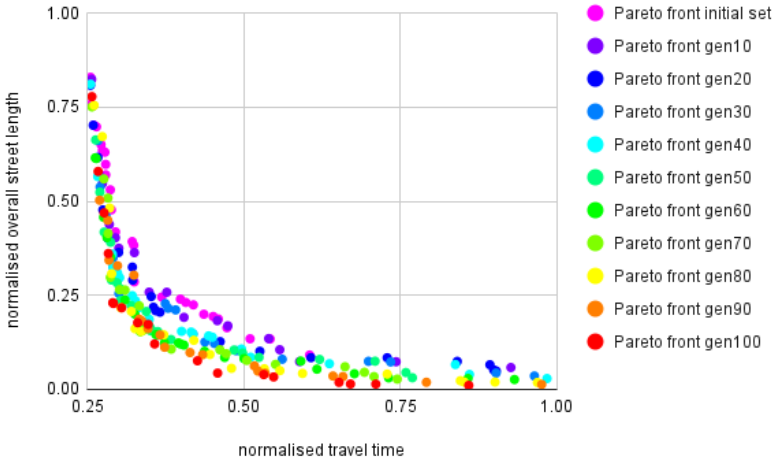
	Genetic Algorithm	**GA with** $LEGCS_{GNN}$	**GA with** $LEGCS_{Emb+Reg}$
Genetic Algorithm	1.0	$8.23 \cdot 10^{-5}$	$4.12 \cdot 10^{-4}$
Genetic Algorithm with $LEGCS_{GNN}$		1.0	0.54
Genetic Algorithm with $LEGCS_{Emb+Reg}$			1.0

Table 6.2: P-values of Mann-Whitney U tests performed on the results of the optimisation strategies in this chapter.

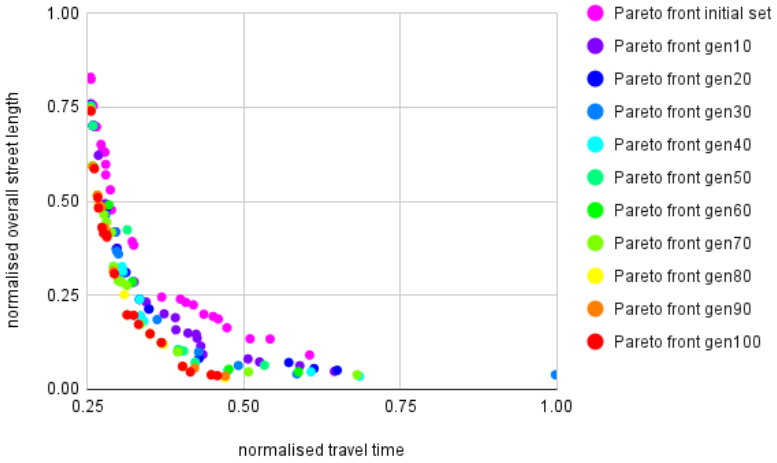
By comparing the p-values in Table 6.2 against a significance value *0.05*, we can see that a LEGCS-supported GA has a significantly better performance than GAs without LEGCS. The choice of learning model (GNN or embedding with regression) makes no significant difference in terms of the optimisation outcome.



(a) Pareto-fronts of GA without LEGCS.



(b) Pareto-fronts of a $LEGCS_{GNN}$-supported GA.



(c) Pareto-fronts of a $LEGCS_{Emb+Reg}$-supported GA.

Figure 6.2: Performance of the multi-objective GA with and without LEGCS support.

6.6 Conclusion

In this chapter, I demonstrated the use of LEGCS in a multi-objective optimisation setting by applying it as a cost-efficient fitness function in a genetic algorithm that optimises traffic networks. I have shown that there is a trade-off between the accuracy and the cost of fitness assessments. In an experiment where accurate fitness assessments are costly, I relied on a prediction model for the majority of assessments to achieve more iterations for the GA to explore traffic network candidates. This led to a more efficient use of the simulation budget.

I found that in our experiment, the learning-supported GA outperformed the GA without LEGCS and a simulation budget of *5,000* already after *1,500* simulations. A further study on the statistical significance shows that a genetic algorithm with LEGCS support leads to Pareto fronts of graph configurations with significantly better performances (lower average travel time and overall street length). It does not make a significant difference in the improvement here whether we choose a GNN-based learning model or an embedding-based learning model.

Chapter 7

Limitations and Future Directions

7.1 Threats to Validity

At this point, I reflect on the experiments and results in this thesis and acknowledge factors that may make my findings susceptible to biases or limitations, potentially impacting the validity and generalisability of my conclusions.

Construct validity. The first case study in this work is based on a MATLAB house ventilation example [5]. I modified the configuration of the HVAC system within the domain constraints to generate a valid configuration set to operate on. The second case study includes SUMO street networks of towns in Ireland. I can claim validity in terms of searching in a set of valid MATLAB house models and respectively optimising valid street networks according to the SUMO engine, but I cannot make claims on the implications these models would have in real-world settings.

Internal validity. The LEGCS approaches in the case studies of this thesis are based on MATLAB Simulink and SUMO simulations. So every attempt to explore and optimise a real-world system can only be as good as the simulation, which represents the corresponding reality. All simulations performed to assess the properties of each graph configuration candidate were performed by a MATLAB engine on the same machine or a SUMO engine on the same machine. These simulations are (although complex) deterministic, so I can confidently say that there are no factors influencing the performances of the graph candidates other than the graphs themselves.

External validity. Since I have only conducted experiments in the context of an HVAC system with comparatively small graphs (up to *10* nodes) and in context of street networks on graphs up to *32* nodes, I cannot claim generalisability for larger graphs or domains outside of engineering.

Conclusion validity. I aimed for a general understanding of the performance of all tested approaches by running every experiment numerous times and compare average or median values. In addition, in Chapters 5 and 6, I also conducted a Mann-Whitney U test to identify cases where outliers distorted the average performances.

7.2 Directions for Optimising LEGCS

In this section, I capture future work arising from the undertaken research and approaches to explore based on the findings of this thesis. First, I explore directions based on each phase of LEGCS, and in the following section, I discuss more applications for LEGCS.

Sampling

In the explorative parts of this thesis, I operated on datasets where I generated a large number of valid configurations. Not in scope were sophisticated graph generation techniques that verifiably cover the whole graph configuration space. Thus, one future direction is to integrate more sophisticated graph generators; this would preempt that well-performing and valid graph configurations are not found because they were not generated in the space we operate on.

But even if the whole valid graph configuration space is generated, so far, I have only looked into one sampling technique (random sampling) to gather training data for the learning model. In Chapter 4, I demonstrated that applying graph neural networks in Case Study I leads to a slightly higher prediction accuracy since the model can fully exploit the graph data instead of operating on vector representation. In context of this work, however, using GNNs has been proven to be impractical for explorative tasks due to the significantly higher sampling costs. One direction here is applying more sophisticated sampling strategies to see if we can achieve these low prediction errors with a comparable number of training data. These strategies could include: *stratified sampling* – divide the graphs into classes, and select them so that all classes are represented in the training set, *cluster sampling* – divide the graphs into clusters, and select one whole cluster as training set, *importance sampling* – Analyse the graphs to detect important-to-learn graphs to include into the training set, or *active learning with uncertainty sampling* – Start training the learning model with a small random set, and then select more graphs to include in the training set according to the uncertainty of the model.

Measuring

Due to the low costs of simulations, I have only considered them to gather measurements for the learning models. Ultimately, I have considered the outcome of these simulations as

"truth". However, following this logic, the predictions from the learning models can only get as close to reality as the simulator is able to depict reality. One direction here would be applying LEGCS using measurements from the real world by executing and measuring systems, using static analysis, or user feedback to assess how well LEGCS can predict non-functional properties in even more complex scenarios.

Learning

In Chapters 3 and 4, I presented and analysed two strategies of applying regression analysis on a set of graphs. While I gathered a good understanding of why they work and the dimensions of how well they work, there is more work to do to fully optimise the learning results using techniques such as hyperparameter tuning. These techniques would surely be able to improve the prediction accuracy by a few percentages. Those improvements might further facilitate the exploration and optimisation process in the graph configuration spaces.

Evaluating

In this work, I focused on "only" identifying as many k best graph configurations as possible with a limited number of simulations in the first case study and optimising towards multiple goals on a simulation budget in the second case study. So far, I have not attempted to understand which substructures in those graphs have the largest influence on desirable properties – and which ones are less relevant. Mechanisms and behaviours in the prediction process of deep learning approaches such as GNNs are already difficult to comprehend [110], but even in the embedding-based approach, we need two interpretations: (1) how graph structures are captured in the embedding, and (2) what parts of the embedding lead to good predictions.

Additionally, I have not analysed how diverse the identified set is, i.e., whether LEGCS finds a large number of similar graphs and there is a whole different cluster of k best graph configurations still in the dataset or if the set we identify is a fair representation of all k best graphs in the configuration space. In the envisioned use case (providing a set of "best" designs for review by human experts or further processing), I can expect that some diversity of the found solutions would be beneficial, e.g., to identify design alternatives.

7.3 Directions for Applying LEGCS

After presenting directions to optimise LEGCS in the previous section, I now discuss future work for *applying* LEGCS.

Objectives

In their systematic literature review [1], Pereira et al. propose six application objectives for learning software configuration spaces: pure prediction, interpretability, optimisation, dynamic configuration, mining constraints, and evolution. While one application can have multiple of these objectives (such as my graph space exploration aims for accurate predictions and optimisation), there is future work to do to evaluate the application of LEGCS for *interpretability* – explain why graph configurations perform well or not, *dynamic configuration* – monitor environment changes during runtime to adapt, *constraint mining* – identify combinations of features that make graph configurations invalid, and *evolution* – how LEGCS can adapt to real-world changes in external conditions.

Generalisation and Scalability

In this work, I have demonstrated the applicability of LEGCS for exploration and optimisation in context of two engineering domains. In those domains, I evaluated graphs with a complexity of up to *32* nodes. More work is needed, however, to (1) determine the performance of LEGCS for settings with large graphs and (2) assess the applicability of LEGCS for domains outside of engineering, e.g., in chemistry to replace expensive measurements of chemical reactions with graph-based ML predictions.

Chapter 8

Conclusion

In this thesis, I have adapted solutions from learning Euclidean configuration spaces to complex graph configuration spaces. In Chapter 3, I proposed a pipeline to learn graph configuration spaces (LEGCS) based on graph neural networks that I applied to two case studies (HVAC systems and traffic networks). With LEGCS, I was able to identify half of the HVAC systems with the minimal AC usage while only simulating 13.75% of the dataset.

Afterwards, I explored a variation of this pipeline in Chapter 4 , where I replaced the GNN with a graph embedding and a regression model operating on the vectors representing graph configurations. Furthermore, I conducted an in-depth study on the various combinations of embedding and learning techniques for the learning phase of LEGCS where I have shown that embedding strategies have a greater influence on the prediction errors than the choice of (sophisticated) learning model. Learning models are not capable of exploiting information in the graph dataset that is not preserved by the embedding. This variation allows us a more data-efficient training to achieve accurate predictions. In the first case study, we can now already identify half of the k best graphs with 11.4% of simulations, in the second case study already after 7.4% of the simulations.

After mostly focusing on learning in the context of LEGCS in these two chapters, I presented two applications of LEGCS in the following two chapters: graph space exploration in HVAC systems and graph optimisation on traffic networks.

In Chapter 5, I applied LEGCS, metaheuristic local searches, and LEGCS-supported metaheuristic local searches to explore the HVAC dataset in the first case study on three different simulation budgets. The results show that a LEGCS-supported iterative hill-climbing approach spends 31.6% of its simulation budget identifying the top 3.4% of the dataset.

Eventually, in Chapter 6, I used a population-based metaheuristic, a genetic algorithm, to investigate whether a LEGCS-supported genetic algorithm can improve the optimisation of traffic networks. In a multi-objective optimisation, I found LEGCS to improve the optimisation results significantly. The hypervolume indicator of the identified graph configurations of the GA without ML was already outperformed by the LEGCS-supported GA after using up only 30% of the simulation budget.

The work presented in this thesis lays the groundwork for learning graph configuration spaces to support metaheuristic exploration and optimisation in engineering domains. Future directions include fully leveraging the potential of LEGCS by optimising sampling, learning, and evaluating strategies as well as investigating the potential of LEGCS for more tasks such as dynamic configuration, constraint mining, and evolution.

Bibliography

[1] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. Learning software configuration spaces: A systematic literature review. *J. Syst. Softw.*, 182:111044, 2021. doi: 10.1016/j.jss.2021.111044. URL `https://doi.org/10.1016/j.jss.2021.111044`.

[2] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Trans. Knowl. Data Eng.*, 30(9):1616–1637, 2018. doi: 10.1109/TKDE.2018.2807452. URL `https://doi.org/10.1109/TKDE.2018.2807452`.

[3] Manuel Fernández Delgado, Manisha Sanjay Sirsat, Eva Cernadas, Sadi Alawadi, Senén Barro, and Manuel Febrero-Bande. An extensive experimental survey of regression methods. *Neural Networks*, 111:11–34, 2019. doi: 10.1016/j.neunet.2018.12.010.

[4] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Luc Lesoil, and Olivier Barais. Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes. *Inria Rennes-Bretagne Atlantique. hal-02314830*, 2019.

[5] Building Ventilation - MATLAB Simulink. `https://www.mathworks.com/help/simscape/ug/building-ventilation.html`, . [Accessed 28-February-2025].

[6] Reza Zanjirani Farahani, Elnaz Miandoabchi, Wai Yuen Szeto, and Hannaneh Rashidi-Bajgan. A review of urban transportation network design problems. *Eur. J. Oper. Res.*, 229(2):281–302, 2013. doi: 10.1016/j.ejor.2013.01.001. URL `https://doi.org/10.1016/j.ejor.2013.01.001`.

[7] Robert S. Pienta, James Abello, Minsuk Kahng, and Duen Horng Chau. Scalable Graph Exploration and Visualization: Sensemaking Challenges and Opportunities. In *International Conference on Big Data and Smart Computing (BIGCOMP)*, pages 271–278. IEEE Computer Society, 2015. doi: 10.1109/35021BIGCOMP.2015.7072812. URL https://doi.org/10.1109/35021BIGCOMP.2015.7072812.

[8] Ilya Makarov, Dmitrii Kiselev, Nikita Nikitinsky, and Lovro Subelj. Survey on graph embeddings and their applications to machine learning problems on graphs. *PeerJ Computer Science*, 7:e357, 2021.

[9] Xiao Wang, Deyu Bo, Chuan Shi, Shaohua Fan, Yanfang Ye, and S Yu Philip. A Survey on Heterogeneous Graph Embedding: Methods, Techniques, Applications and Sources. *IEEE Transactions on Big Data*, 2022.

[10] Siqi Miao, Mia Liu, and Pan Li. Interpretable and Generalizable Graph Learning via Stochastic Attention Mechanism. In *International Conference on Machine Learning*, pages 15524–15543. PMLR, 2022.

[11] Richard L. Church and Carlos A. Baez. Generating optimal and near-optimal solutions to facility location problems. *Environment and Planning B: Urban Analytics and City Science*, 47(6):1014–1030, 2020.

[12] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi: 10.1137/0201010. URL https://doi.org/10.1137/0201010.

[13] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi: 10.1007/BF01386390. URL https://doi.org/10.1007/BF01386390.

[14] A Hanif Halim and IJAoCMiE Ismail. Combinatorial optimization: comparison of heuristic algorithms in travelling salesman problem. *Archives of Computational Methods in Engineering*, 26:367–380, 2019. URL https://doi.org/10.1007/s11831-017-9247-y.

[15] Krasimira Genova and Vassil Guliashki. Linear integer programming methods and approaches–a survey. *Journal of Cybernetics and Information Technologies*, 11(1), 2011.

[16] Michael Mittermaier, Takfarinas Saber, and Goetz Botterweck. Applying Graph Neural Networks to Learn Graph Configuration Spaces. In *The 19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2025. doi: 10.1145/3715340.3715444.

[17] Michael Mittermaier, Takfarinas Saber, and Goetz Botterweck. Learning graph configuration spaces with graph embedding in engineering domains. In *Proc. of LOD 2023*, pages 334–348, 2024. doi: 10.1007/978-3-031-53966-4_25.

[18] Michael Mittermaier, Takfarinas Saber, and Goetz Botterweck. Learning Graph Configuration Spaces to Support Road Network Design Optimisation. In *The Genetic and Evolutionary Computation Conference GECCO*, 2025. doi: 10.1145/3712256.3726447.

[19] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proc. of VAMOS 2012*, pages 173–182, 2012. doi: 10.1145/2110147.2110167.

[20] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. CASE tool support for variability management in software product lines. *ACM Comput. Surv.*, 50(1):14:1–14:45, 2017. doi: 10.1145/3034827. URL `https://doi.org/10.1145/3034827`.

[21] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Autom. Softw. Eng.*, 18 (1):77–114, 2011. doi: 10.1007/S10515-010-0076-6. URL `https://doi.org/10.1007/s10515-010-0076-6`.

[22] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL conqueror: Toward optimization of non-functional properties in software product lines. *Softw. Qual. J.*, 20(3-4):487–517, 2012. doi: 10.1007/S11219-011-9152-9. URL `https://doi.org/10.1007/s11219-011-9152-9`.

[23] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Inf. Softw. Technol.*, 55(3): 491–507, 2013. doi: 10.1016/J.INFSOF.2012.07.020. URL `https://doi.org/10.1016/j.infsof.2012.07.020`.

[24] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. On extracting feature models from product descriptions. In Ulrich W. Eisenecker, Sven Apel, and Stefania Gnesi, editors, *Sixth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2012)*, pages 45–54. ACM, 2012. doi: 10.1145/2110147.2110153. URL https://doi.org/10.1145/2110147.2110153.

[25] José Angel Galindo, Deepak Dhungana, Rick Rabiser, David Benavides, Goetz Botterweck, and Paul Grünbacher. Supporting distributed product configuration by integrating heterogeneous variability modeling approaches. *Inf. Softw. Technol.*, 62:78–100, 2015. doi: 10.1016/J.INFSOF.2015.02.002. URL https://doi.org/10.1016/j.infsof.2015.02.002.

[26] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *Eur. J. Oper. Res.*, 119(3):557–581, 1999. doi: 10.1016/S0377-2217(98)00364-6. URL https://doi.org/10.1016/S0377-2217(98)00364-6.

[27] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines. In *SPLC*, 2011. doi: 10.1109/SPLC.2011.20. URL https://doi.org/10.1109/SPLC.2011.20.

[28] Martin Glinz. On Non-Functional Requirements. In *International Requirements Engineering Conference (RE)*, pages 21–26, 2007. doi: 10.1109/RE.2007.45. URL https://doi.org/10.1109/RE.2007.45.

[29] Ilya Makarov, Dmitrii Kiselev, Nikita Nikitinsky, and Lovro Subelj. Survey on graph embeddings and their applications to machine learning problems on graphs. *PeerJ Comput. Sci.*, 7:e357, 2021. doi: 10.7717/PEERJ-CS.357. URL https://doi.org/10.7717/peerj-cs.357.

[30] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Trans. Knowl. Data Eng.*, 34(1):249–270, 2022. doi: 10.1109/TKDE.2020.2981333. URL https://doi.org/10.1109/TKDE.2020.2981333.

[31] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine Learning on Graphs: A Model and Comprehensive Taxonomy. *J. Mach. Learn. Res.*, 23:89:1–89:64, 2022. URL http://jmlr.org/papers/v23/20-852.html.

[32] Yun Peng, Byron Choi, and Jianliang Xu. Graph Learning for Combinatorial Optimization: A Survey of State-of-the-Art. *Data Sci. Eng.*, 6(2):119–141, 2021. doi: 10.1007/s41019-021-00155-3. URL https://doi.org/10.1007/s41019-021-00155-3.

[33] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. doi: 10.1016/j.aiopen.2021.01.001. URL https://doi.org/10.1016/j.aiopen.2021.01.001.

[34] Shunxin Xiao, Shiping Wang, Yuanfei Dai, and Wenzhong Guo. Graph neural networks in node classification: survey and evaluation. *Mach. Vis. Appl.*, 33(1):4, 2022. doi: 10.1007/S00138-021-01251-0.

[35] Emily Alsentzer, Samuel G. Finlayson, Michelle M. Li, and Marinka Zitnik. Subgraph neural networks. In *Proc. of NeurIPS 2020*, 2020.

[36] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. In *Proc. of ICLR 2020*, 2020. URL https://openreview.net/forum?id=HygDF6NFPB.

[37] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Proc. of NeurIPS 2018*, pages 5171–5181, 2018.

[38] Henrique Lemos, Marcelo O. R. Prates, Pedro H. C. Avelar, and Luís C. Lamb. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*, pages 879–885. IEEE, 2019. doi: 10.1109/ICTAI.2019.00125. URL https://doi.org/10.1109/ICTAI.2019.00125.

[39] Kien Do, Truyen Tran, and Svetha Venkatesh. Graph transformation policy network for chemical reaction prediction. In *KDD*, pages 750–760. ACM, 2019. doi: 10.1145/3292500.3330958. URL https://doi.org/10.1145/3292500.3330958.

[40] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, pages 6530–6539, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/f507783927f2ec2737ba40afbd17efb5-Abstract.html.

[41] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. Convolutional neural networks over control flow graphs for software defect prediction. In *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, pages 45–52. IEEE Computer Society, 2017. doi: 10.1109/ICTAI.2017.00019. URL https://doi.org/10.1109/ICTAI.2017.00019.

[42] Chuanpan Zheng, Xiaoliang Fan, Cheng Wang, and Jianzhong Qi. GMAN: A graph multi-attention network for traffic prediction. In *Proc. of Conference on Artificial Intelligence (AAAI)*, pages 1234–1241. AAAI Press, 2020. URL https://ojs.aaai.org/index.php/AAAI/article/view/5477.

[43] David Liben-Nowell and Jon M. Kleinberg. The link-prediction problem for social networks. *J. Assoc. Inf. Sci. Technol.*, 58(7):1019–1031, 2007. doi: 10.1002/asi.20591. URL https://doi.org/10.1002/asi.20591.

[44] Jiaxuan You, Zhitao Ying, and Jure Leskovec. Design space for graph neural networks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Proc. of Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[45] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Proc. of NeurIPS*, pages 4805–4815, 2018.

[46] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *Proc. of International Conference on Learning Representations (ICLR)*. OpenReview.net, 2019. URL https://openreview.net/forum?id=ryGs6iA5Km.

[47] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowl. Based Syst.*, 151:78–94, 2018. doi: 10.1016/j.knosys.2018.03.022. URL https://doi.org/10.1016/j.knosys.2018.03.022.

[48] Nils M. Kriege, Fredrik D. Johansson, and Christopher Morris. A survey on graph kernels. *Appl. Netw. Sci.*, 5(1):6, 2020. doi: 10.1007/s41109-019-0195-3. URL https://doi.org/10.1007/s41109-019-0195-3.

[49] Dastan Maulud and Adnan M Abdulazeez. A review on linear regression comprehensive in machine learning. *Journal of Applied Science and Technology Trends*, 1(2):140–147, 2020.

[50] James A Stimson, Edward G Carmines, and Richard A Zeller. Interpreting polynomial regression. *Sociological Methods & Research*, 6(4):515–524, 1978.

[51] Gary C McDonald. Ridge regression. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(1):93–100, 2009.

[52] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[53] Min Xu, Pakorn Watanachaturaporn, Pramod K Varshney, and Manoj K Arora. Decision tree regression for soft classification of remote sensing data. *Remote Sensing of Environment*, 97(3):322–336, 2005.

[54] Mark R Segal. Machine learning benchmarks and random forest regression. 2004.

[55] Donald F. Specht. A general regression neural network. *IEEE Trans. Neural Networks*, 2 (6):568–576, 1991. doi: 10.1109/72.97934. URL https://doi.org/10.1109/72.97934.

[56] Yunsheng Song, Jiye Liang, Jing Lu, and Xingwang Zhao. An efficient instance selection algorithm for k nearest neighbor regression. *Neurocomputing*, 251:26–34, 2017. doi: 10.1016/j.neucom.2017.04.018. URL https://doi.org/10.1016/j.neucom.2017.04.018.

[57] Jie Wang. An intuitive tutorial to gaussian processes regression. *CoRR*, abs/2009.10862, 2020. URL https://arxiv.org/abs/2009.10862.

[58] Harris Drucker, Christopher J. C. Burges, Linda Kaufman, Alexander J. Smola, and Vladimir Vapnik. Support vector regression machines. In Michael Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems 9, NIPS, Denver, CO, USA, December 2-5, 1996*, pages 155–161. MIT Press, 1996. URL http://papers.nips.cc/paper/1238-support-vector-regression-machines.

[59] Zahra Beheshti and Siti Mariyam Hj Shamsuddin. A review of population-based meta-heuristic algorithms. *Int. j. adv. soft comput. appl*, 5(1):1–35, 2013.

[60] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Inf. Sci.*, 237:82–117, 2013. doi: 10.1016/J.INS.2013.02.041. URL https://doi.org/10.1016/j.ins.2013.02.041.

[61] Shubhkirti Sharma and Vijay Kumar. A comprehensive review on multi-objective optimization techniques: Past, present and future. *Archives of Computational Methods in Engineering*, 29(7):5605–5633, 2022.

[62] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multim. Tools Appl.*, 80(5):8091–8126, 2021. doi: 10.1007/S11042-020-10139-6. URL https://doi.org/10.1007/s11042-020-10139-6.

[63] Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, Iñaki Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artif. Intell. Rev.*, 13(2):129–170, 1999. doi: 10.1023/A:1006529012972. URL https://doi.org/10.1023/A:1006529012972.

[64] Jin Kim, Inwook Hwang, Yong-Hyuk Kim, and Byung Ro Moon. Genetic approaches for graph partitioning: a survey. In Natalio Krasnogor and Pier Luca Lanzi, editors, *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12-16, 2011*, pages 473–480. ACM, 2011. doi: 10.1145/2001576.2001642. URL https://doi.org/10.1145/2001576.2001642.

[65] Günther R. Raidl and Bryant A. Julstrom. Edge sets: an effective evolutionary coding of spanning trees. *IEEE Trans. Evol. Comput.*, 7(3):225–239, 2003. doi: 10.1109/TEVC.2002.807275. URL https://doi.org/10.1109/TEVC.2002.807275.

[66] Clara Pizzuti. Ga-net: A genetic algorithm for community detection in social networks. In Günter Rudolph, Thomas Jansen, Simon M. Lucas, Carlo Poloni, and Nicola Beume, editors, *Parallel Problem Solving from Nature - PPSN X, 10th International Conference Dortmund, Germany, September 13-17, 2008, Proceedings*, volume 5199 of *Lecture Notes in Computer Science*, pages 1081–1090. Springer, 2008. doi: 10.1007/978-3-540-87700-4_107. URL https://doi.org/10.1007/978-3-540-87700-4_107.

[67] Mohd Saqib, Benjamin C. M. Fung, Philippe Charland, and Andrew Walenstein. GAGE: genetic algorithm-based graph explainer for malware analysis. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pages 2258–2270. IEEE, 2024. doi: 10.1109/ICDE60146.2024.00179. URL https://doi.org/10.1109/ICDE60146.2024.00179.

[68] Charles C. Palmer and Aaron Kershenbaum. Representing trees in genetic algorithms. In *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, June 27-29, 1994*, pages 379–384. IEEE, 1994. doi: 10.1109/ICEC.1994.349921. URL https://doi.org/10.1109/ICEC.1994.349921.

[69] Vincenzo Maniscalco, Silvana Greco Polito, and Antonio Intagliata. Tree-based genetic algorithm with binary encoding for qos routing. In Leonard Barolli, Ilsun You, Fatos Xhafa, Fang-Yie Leu, and Hsing-Chung Chen, editors, *Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2013, Taichung, Taiwan, July 3-5, 2013*, pages 101–107. IEEE Computer Society, 2013. doi: 10.1109/IMIS.2013.26. URL https://doi.org/10.1109/IMIS.2013.26.

[70] Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. Multi-objective optimization. In *Decision sciences*, pages 161–200. CRC Press, 2016.

[71] Ke Shang, Hisao Ishibuchi, Linjun He, and Lie Meng Pang. A survey on the hypervolume indicator in evolutionary multiobjective optimization. *IEEE Trans. Evol. Comput.*, 25(1):1–20, 2021. doi: 10.1109/TEVC.2020.3013290. URL https://doi.org/10.1109/TEVC.2020.3013290.

[72] David Eppstein and Denis Kurz. *k*-Best Solutions of MSO Problems on Tree-Decomposable Graphs. In *12th International Symposium on Parameterized and Exact Computation (IPEC)*, 2017.

[73] Manuel Chica, Angel A. Juan, Christopher Bayliss, Oscar Cordon, and David Kelton. Why simheuristics? Benefits, limitations, and best practices when combining metaheuristics with simulation. *Statistics and Operations Research Transactions*, 2020.

[74] Angel A Juan, Javier Faulin, Scott E Grasman, Markus Rabe, and Gonçalo Figueira. A

review of simheuristics: Extending metaheuristics to deal with stochastic combinatorial optimization problems. *Operations Research Perspectives*, 2:62–72, 2015.

[75] Angel A. Juan, W. David Kelton, Christine S.M. Currie, and Javier Faulin. Simheuristics applications: dealing with uncertainty in logistics, transportation, and other supply chain areas. In *WSC*, pages 3048–3059. IEEE, 2018.

[76] Taha Mostafaie, Farzin Modarres Khiyabani, and Nima Jafari Navimipour. A systematic study on meta-heuristic approaches for solving the graph coloring problem. *Comput. Oper. Res.*, 120:104850, 2020. doi: 10.1016/J.COR.2019.104850. URL https://doi.org/10.1016/j.cor.2019.104850.

[77] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *Eur. J. Oper. Res.*, 242(3):693–709, 2015. doi: 10.1016/J.EJOR.2014.09.064. URL https://doi.org/10.1016/j.ejor.2014.09.064.

[78] Frank Neumann and Carsten Witt. Ant colony optimization and the minimum spanning tree problem. *Theor. Comput. Sci.*, 411(25):2406–2413, 2010. doi: 10.1016/J.TCS.2010.02.012. URL https://doi.org/10.1016/j.tcs.2010.02.012.

[79] Nicolas Jozefowiez, Fred W. Glover, and Manuel Laguna. Multi-objective metaheuristics for the traveling salesman problem with profits. *J. Math. Model. Algorithms*, 7(2):177–195, 2008. doi: 10.1007/S10852-008-9080-2. URL https://doi.org/10.1007/s10852-008-9080-2.

[80] Raafat Elshaer and Hadeer Awad. A taxonomic review of metaheuristic algorithms for solving the vehicle routing problem and its variants. *Comput. Ind. Eng.*, 140:106242, 2020. doi: 10.1016/J.CIE.2019.106242. URL https://doi.org/10.1016/j.cie.2019.106242.

[81] Hossain Poorzahedy and Omid M. Rouhani. Hybrid meta-heuristic algorithms for solving network design problem. *Eur. J. Oper. Res.*, 182(2):578–596, 2007. doi: 10.1016/J.EJOR.2006.07.038. URL https://doi.org/10.1016/j.ejor.2006.07.038.

[82] Adis Alihodzic, Malek Chahin, and Fikret Cunjalo. New clustering techniques of node embeddings based on metaheuristic optimization algorithms. In Ivan Lirkov and Svetozar Margenov, editors, *Large-Scale Scientific Computing - 13th International*

Conference, LSSC 2021, Sozopol, Bulgaria, June 7-11, 2021, Revised Selected Papers, volume 13127 of *Lecture Notes in Computer Science*, pages 201–208. Springer, 2021. doi: 10.1007/978-3-030-97549-4_23. URL https://doi.org/10.1007/978-3-030-97549-4_23.

[83] Defeng Liu, Vincent Perreault, Alain Hertz, and Andrea Lodi. A machine learning framework for neighbor generation in metaheuristic search. *Frontiers Appl. Math. Stat.*, 9, 2023. doi: 10.3389/FAMS.2023.1128181. URL https://doi.org/10.3389/fams.2023.1128181.

[84] Akbar Telikani, Amirhessam Tahmassebi, Wolfgang Banzhaf, and Amir H. Gandomi. Evolutionary machine learning: A survey. *ACM Comput. Surv.*, 54(8):161:1–161:35, 2022. doi: 10.1145/3467477. URL https://doi.org/10.1145/3467477.

[85] Khader Hamdia, Xiaoying Zhuang, and Timon Rabczuk. An efficient optimization approach for designing machine learning models based on genetic algorithm. *Neural Comput. Appl.*, 33(6):1923–1933, 2021. doi: 10.1007/S00521-020-05035-X. URL https://doi.org/10.1007/s00521-020-05035-x.

[86] Harsh Bhasin and Surbhi Bhatia. Application of genetic algorithms in machine learning. *IJCSIT*, 2(5):2412–2415, 2011.

[87] Angel A. Juan, Peter B. Keenan, Rafael Martí, Seán McGarraghy, Javier Panadero, Paula Carroll, and Diego Oliva. A review of the role of heuristics in stochastic optimisation: from metaheuristics to learnheuristics. *Ann. Oper. Res.*, 320(2):831–861, 2023. doi: 10.1007/S10479-021-04142-9. URL https://doi.org/10.1007/s10479-021-04142-9.

[88] El-Ghazali Talbi. Machine learning into metaheuristics: A survey and taxonomy. *ACM Comput. Surv.*, 54(6):129:1–129:32, 2022. doi: 10.1145/3459664. URL https://doi.org/10.1145/3459664.

[89] Jun Zhang, Zhi-hui Zhan, Ying Lin, Ni Chen, Yue-jiao Gong, Jing-Hui Zhong, Henry Shu-Hung Chung, Yun Li, and Yu-hui Shi. Evolutionary computation meets machine learning: A survey. *IEEE Comput. Intell. Mag.*, 6(4):68–75, 2011. doi: 10.1109/MCI.2011.942584. URL https://doi.org/10.1109/MCI.2011.942584.

[90] Aimin Zhou and Qingfu Zhang. A surrogate-assisted evolutionary algorithm for minimax optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*, pages 1–7. IEEE, 2010. doi: 10.1109/CEC.2010.5586122. URL https://doi.org/10.1109/CEC.2010.5586122.

[91] Reza Zanjirani Farahani, Elnaz Miandoabchi, Wai Yuen Szeto, and Hannaneh Rashidi. A review of urban transportation network design problems. *European journal of operational research*, 229(2):281–302, 2013.

[92] Giulio Erberto Cantarella, Giuseppe Pavone, and Antonino Vitetta. Heuristics for urban road network design: Lane layout and signal settings. *Eur. J. Oper. Res.*, 175(3): 1682–1695, 2006. doi: 10.1016/J.EJOR.2005.02.034. URL https://doi.org/10.1016/j.ejor.2005.02.034.

[93] Haozhi Zhang and Ziyou Gao. Bilevel programming model and solution method for mixed transportation network design problem. *J. Syst. Sci. Complex.*, 22(3): 446–459, 2009. doi: 10.1007/S11424-009-9177-3. URL https://doi.org/10.1007/s11424-009-9177-3.

[94] Rydzewski Aleksander and Czarnul Paweł. Recent advances in traffic optimisation: systematic literature review of modern models, methods and algorithms. *IET Intelligent Transport Systems*, 14(13):1740–1758, 2020.

[95] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *Expert Syst. Appl.*, 207:117921, 2022. doi: 10.1016/J.ESWA.2022.117921. URL https://doi.org/10.1016/j.eswa.2022.117921.

[96] Mohammad Noaeen, Atharva Naik, Liana Goodman, Jared Crebo, Taimoor Abrar, Zahra Shakeri Hossein Abad, Ana L. C. Bazzan, and Behrouz H. Far. Reinforcement learning in urban network traffic signal control: A systematic literature review. *Expert Syst. Appl.*, 199:116830, 2022. doi: 10.1016/J.ESWA.2022.116830. URL https://doi.org/10.1016/j.eswa.2022.116830.

[97] Fannia Pacheco, Ernesto Exposito, Mathieu Gineste, Cédric Baudoin, and José Aguilar. Towards the deployment of machine learning solutions in network traffic classification: A systematic survey. *IEEE Commun. Surv. Tutorials*, 21(2):1988–2014,

2019. doi: 10.1109/COMST.2018.2883147. URL https://doi.org/10.1109/COMST.2018.2883147.

[98] Adam Slowik and Halina Kwasnicka. Evolutionary algorithms and their applications to engineering problems. *Neural Comput. Appl.*, 32(16):12363–12379, 2020. doi: 10.1007/S00521-020-04832-8. URL https://doi.org/10.1007/s00521-020-04832-8.

[99] Davide Chicco, Matthijs J. Warrens, and Giuseppe Jurman. The coefficient of determination r-squared is more informative than smape, mae, mape, MSE and RMSE in regression analysis evaluation. *PeerJ Comput. Sci.*, 7:e623, 2021. doi: 10.7717/PEERJ-CS.623. URL https://doi.org/10.7717/peerj-cs.623.

[100] Mary Shaw. Writing Good Software Engineering Research Paper. In *Proceedings of the 25th International Conference on Software Engineering*, pages 726–737. IEEE Computer Society, 2003. doi: 10.1109/ICSE.2003.1201262. URL https://doi.org/10.1109/ICSE.2003.1201262.

[101] Steve Easterbrook, Janice Singer, Margaret-Anne D. Storey, and Daniela E. Damian. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008. doi: 10.1007/978-1-84800-044-5_11. URL https://doi.org/10.1007/978-1-84800-044-5_11.

[102] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008. doi: 10.1007/978-1-84800-044-5_8. URL https://doi.org/10.1007/978-1-84800-044-5_8.

[103] MATLAB Simulink - Simulation and Model-Based Design. https://uk.mathworks.com/products/simulink.html,. [Accessed 28-February-2025].

[104] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018. URL https://elib.dlr.de/124092/.

[105] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proc. of ICLR 2017*. OpenReview.net, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

[106] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019. URL http://arxiv.org/abs/1903.02428.

[107] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *CoRR*, abs/1710.10903, 2017. URL http://arxiv.org/abs/1710.10903.

[108] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proc. of NIPS 2017*, pages 1024–1034, 2017.

[109] Ricardo Lopes, Tim Tutenel, Ruben M Smelik, Klaas Jan De Kraker, and Rafael Bidarra. A constrained growth method for procedural floor plan generation. In *Proc. Int. Conf. Intell. Games Simul*, pages 13–20, 2010.

[110] Feng-Lei Fan, Jinjun Xiong, Mengzhou Li, and Ge Wang. On interpretability of artificial neural networks: A survey. *IEEE Transactions on Radiation and Plasma Medical Sciences*, 5(6):741–760, 2021.

[111] Gaurav Menghani. Efficient deep learning: A survey on making deep learning models smaller, faster, and better. *ACM Comput. Surv.*, 55(12):259:1–259:37, 2023. doi: 10.1145/3578938. URL https://doi.org/10.1145/3578938.

[112] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs. In *International Conference on Information and Knowledge Management (CIKM)*, page 3125–3132. ACM, 2020.

[113] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005, 2017. URL http://arxiv.org/abs/1707.05005.

[114] Hong Chen and Hisashi Koga. Gl2vec: Graph embedding enriched by line graphs with edge features. In Tom Gedeon, Kok Wai Wong, and Minho Lee, editors, *Neu-*

ral Information Processing - 26th International Conference, ICONIP 2019, Sydney, NSW, Australia, December 12-15, 2019, Proceedings, Part III, volume 11955 of *Lecture Notes in Computer Science*, pages 3–14. Springer, 2019. doi: 10.1007/978-3-030-36718-3_1. URL https://doi.org/10.1007/978-3-030-36718-3_1.

[115] Nathan de Lara and Edouard Pineau. A simple baseline algorithm for graph classification. *CoRR*, abs/1810.09155, 2018. URL http://arxiv.org/abs/1810.09155.

[116] Saurabh Verma and Zhi-Li Zhang. Hunt for the unique, stable, sparse and fast feature learning on graphs. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 88–98, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/d2ddea18f00665ce8623e36bd4e3c7c5-Abstract.html.

[117] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, Alexander M. Bronstein, and Emmanuel Müller. Netlsd: Hearing the shape of a graph. In Yike Guo and Faisal Farooq, editors, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 2347–2356. ACM, 2018. doi: 10.1145/3219819.3219991. URL https://doi.org/10.1145/3219819.3219991.

[118] Chen Cai and Yusu Wang. A simple yet effective baseline for non-attributed graph classification. *arXiv:1811.03508*, 2022.

[119] Lili Wang, Chenghan Huang, Weicheng Ma, Xinyuan Cao, and Soroush Vosoughi. Graph embedding via diffusion-wavelets-based node feature distribution characterization. In Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong, editors, *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, pages 3478–3482. ACM, 2021. doi: 10.1145/3459637.3482115. URL https://doi.org/10.1145/3459637.3482115.

[120] Benedek Rozemberczki and Rik Sarkar. Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models. In Mathieu d'Aquin, Stefan

Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux, editors, *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, pages 1325–1334. ACM, 2020. doi: 10.1145/3340531.3411866. URL https://doi.org/10.1145/3340531.3411866.

[121] Alexis Galland and Marc Lelarge. Invariant embedding for graph classification. In *ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data*, 2019.

[122] Feng Gao, Guy Wolf, and Matthew J. Hirn. Geometric scattering for graph data analysis. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2122–2131. PMLR, 2019. URL http://proceedings.mlr.press/v97/gao19e.html.

[123] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[124] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 843–852. IEEE Computer Society, 2017. doi: 10.1109/ICCV.2017.97. URL https://doi.org/10.1109/ICCV.2017.97.

[125] Andy Liaw and Matthew Wiener. Classification and Regression by randomForest. *R news*, 2(3):18–22, 2002.

[126] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[127] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197, 2002. doi: 10.1109/4235.996017. URL https://doi.org/10.1109/4235.996017.

[128] Antonio Benítez-Hidalgo, Antonio J. Nebro, José García-Nieto, Izaskun Oregi, and Javier Del Ser. jmetalpy: a python framework for multi-objective optimization with metaheuristics. *CoRR*, abs/1903.02915, 2019. URL http://arxiv.org/abs/1903.02915.

[129] Haiping Ma, Yajing Zhang, Shengyi Sun, Ting Liu, and Yu Shan. A comprehensive survey on NSGA-II for multi-objective optimization and applications. *Artif. Intell. Rev.*, 56(12):15217–15270, 2023. doi: 10.1007/S10462-023-10526-Z. URL https://doi.org/10.1007/s10462-023-10526-z.