Efficient Minimum Weight Vertex Cover Heuristics Using Graph Neural Networks

Kenneth Langedal[1]
University of Bergen, Norway

Johannes Langguth
Simula Research Laboratory, Oslo, Norway

Fredrik Manne
University of Bergen, Norway

Daniel Thilo Schroeder
Simula Research Laboratory, Oslo, Norway

Abstract

Minimum weighted vertex cover is the NP-hard graph problem of choosing a subset of vertices incident to all edges such that the sum of the weights of the chosen vertices is minimum. Previous efforts for solving this in practice have typically been based on search-based iterative heuristics or exact algorithms that rely on reduction rules and branching techniques. Although exact methods have shown success in solving instances with up to millions of vertices efficiently, they are limited in practice due to the NP-hardness of the problem.

We present a new hybrid method that combines elements from exact methods, iterative search, and graph neural networks (GNNs). More specifically, we first compute a greedy solution using reduction rules whenever possible. If no such rule applies, we consult a GNN model that selects a vertex that is likely to be in or out of the solution, potentially opening up for further reductions. Finally, we use an improved local search strategy to enhance the solution further.

Extensive experiments on graphs of up to a billion edges show that the proposed GNN-based approach finds better solutions than existing heuristics. Compared to exact solvers, the method produced solutions that are, on average, 0.04% away from the optimum while taking less time than all state-of-the-art alternatives.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms; Theory of computation → Randomized local search

Keywords and phrases Minimum weighted vertex cover, Maximum weighted independent set, Graph neural networks, Reducing-peeling

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.12

Supplementary Material
Software (Source Code): `https://github.com/KennethLangedal/GNN-MWVC`
Dataset (Results): `https://github.com/KennethLangedal/MWVC-GNN-LS`

Funding The work has benefited from the Experimental Infrastructure project eX3, which is financially supported by the Research Council of Norway under contract 270053.
Kenneth Langedal: Supported by the Research Council of Norway under contract 303404.
Johannes Langguth: Supported by the Research Council of Norway under contract 303404.
Daniel Thilo Schroeder: Supported by the Research Council of Norway under contract 303404.

[1] Corresponding author

1 Introduction

Consider an undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. A vertex cover is a set $C \subseteq V$ such that $\forall \{u, v\} \in E \implies u \in C \vee v \in C$, or with words, that each edge has at least one of its endpoints in C. The minimum vertex cover problem (MVC) is to find a vertex cover where $|C|$ is minimized. The minimum weighted vertex cover problem (MWVC) includes a positive weight $w : V \to \mathbb{R}^+$ for each vertex and the problem is then to find a vertex cover where $\sum_{u \in C} w(u)$ is minimized. The decision version of MVC was one of Karp's original 21 NP-complete problems and it thus follows that MVC is NP-hard [13]. MWVC is at least as difficult as MVC since the case where the weights are all one is precisely the same as the unweighted version.

It is well known that kernelization techniques can speed up the computation of solutions to NP-hard problems. They shrink the instance by applying reduction rules so that an optimal solution for the reduced instance can be expanded to an optimal solution for the original instance. Several such reduction rules have been developed both for the MVC and the MWVC problem. Using reduction rules, Lamm et al. were able to solve large instances of both MVC and MWVC with millions of edges and vertices [8, 10, 18].

Reduction rules have also been used to speed up heuristics for computing vertex covers and solving associated problems. It is straightforward to see that such rules can be used as a preprocessing step before running an iterative search heuristic [19]. However, they can also be used in combination with heuristics that classify vertices when no reduction rules apply. This technique was introduced by Chang et al., who named it *reducing-peeling* [6]. Examples of such methods that have been used for MVC include genetic algorithms [17] and graph neural networks [21].

We present a hybrid approach that combines several strategies to create an effective heuristic for the MWVC problem. First, we use a combination of reduction rules, a graph neural network, and an exact solver to compute an initial vertex cover. The vertex cover is then further enhanced using an improved iterative search strategy. The proposed heuristic is compared to the best existing methods, yielding the following main results:

- We demonstrate the first successful application of graph neural networks on the MWVC problem.
- We give an improved local search implementation for MWVC on very large sparse graphs.
- On instances that can be solved exactly we compute solutions that are on average 0.04% heavier than the optimal ones.
- We obtain consistently better solutions than existing heuristics for the MWVC problem.
- Finally, we give results on several hundred graphs from SuiteSparse, including instances with more than 1 billion edges, which is significantly larger than those computed in previous efforts. Even at these sizes, our proposed heuristic finds vertex covers in less than one hour using a standard CPU.

In the remainder of the paper, we first introduce the main concepts along with previous work in Section 2 and present our approach in Section 3. Sections 4 and 5 present our experiments along with their results, while Section 6 concludes the paper.

2 Background and Related Work

The MWVC problem has several real-world applications, including dynamic map labeling [2], biological network alignment [1], and network engineering [26]. Furthermore, the problem of finding a minimum vertex cover is equivalent to the problem of finding a maximum independent set. For any feasible vertex cover C, the vertices not in the vertex cover $V \setminus C$ make an independent set, and if $|C|$ is a minimum vertex cover, then $V \setminus C$ is a maximum independent set. This also extends to the weighted versions of these problems. Thus, results for the maximum weighted independent set problem (MWIS) carry over to MWVC.

When constructing a cover C for a graph $G = (V, E)$, each decision whether a vertex $u \in V$ should be in C or not has immediate implications for its neighborhood $N(u)$, i.e. the set of vertices adjacent to u. The resulting graph G' is smaller and can be solved independently from previous decisions. If u is added to C, then every edge connecting u to the rest of the graph will be covered. From that point, the problem is to find an MWVC on $G' = G \setminus \{u\}$. Similarly, if u is excluded from C, it follows from the definition of a vertex cover that $N(u)$ must be in C. In that case, u and its neighborhood can be removed from the graph, yielding $G' = G \setminus N[u]$, where $N[u] = N(u) \cup \{u\}$.

This observation points to two immediate questions. First, how to decide for a vertex u whether to include u or $N(u)$, and second in which order such decisions should be made. In the following, we discuss the principal ways in which this has been done previously.

2.1 Reduction Rules

There has been extensive research on computing exact solutions to NP-hard problems using kernelization combined with various branch-and-bound (BB) techniques. BB makes temporary decisions but is able to backtrack in order to find an exact solution.

The currently best exact solver for the MWVC problem is the branch-and-reduce solver B & R by Lamm et al. [18]. B & R is actually an MWIS solver, but as stated earlier, it can be directly applied to solve the MWVC problem as well. B & R selects vertices for branching based on degree, breaking ties based on weight. Clique covers are used to find an upper bound for the optimal solution to prune the search. What extends it from branch-and-bound to branch-and-reduce is the addition of reduction rules. Before each branch, the remaining graph is checked to see if any reduction rules can be applied to the current remaining graph. It is crucial to check before each branch, not only the first, since branching on a vertex and temporarily labeling it can enable further reductions. The graph is also checked for connectivity after applying reduction rules. If there are multiple connected components, these are solved separately, combining the partial results afterward. Reduction rules are not always applicable, but they are exact in the sense that decisions taken by reduction rules never prevent an optimum solution. By using an extensive set of reduction rules, B & R can often find minimum weight vertex covers on graphs with millions of vertices in a reasonable amount of time.

Our heuristic makes use of the same reduction rules as B & R. Therefore, only an outline of the main ideas is provided here. At a high level, reduction rules come in two varieties. Rules of the first type, called *removal*, directly decide whether vertices should be added to the cover and remove vertices or neighborhoods from the graph. Rules of the second type, called *folding*, also reduce the size of the graph, but without making immediate decisions about vertices. The idea is that after solving the reduced graph, it can be unfolded to extend the solution to the original graph. The following rules are examples. Other rules used in our heuristic are described by Lamm et al. [18].

Neighborhood Removal. If the weight of a vertex is greater than or equal to the combined weight of its neighborhood, i.e, $w(u) \geq w(N(u))$, then some MWVC C includes $N(u)$ and not u. To see this assume $u \in C$. Then the cost of the solution will not increase if u is replaced by $N(u)$.

Neighborhood Folding. Let u be a vertex such that no vertices in $N(u)$ are adjacent and let $v \in N(u)$ be a lightest neighbor. If $w(u) < w(N(u))$ but $w(u) \geq w(N(u) \setminus v)$ then some MWVC includes exactly one of u and $N(u)$. This follows since if a vertex cover includes u and at least one vertex from $N(u)$, then swapping u for the remaining vertices in $N(u)$ will not increase the cost. Unlike the previous rule, there is still the possibility that an MWVC could include u and no vertices from $N(u)$. Therefore, $N(u)$ cannot directly be classified yet. Instead, $N[u]$ is folded into a new vertex u' connected to every vertex adjacent to $N(u)$. The weight of this new vertex is set to $w(N(u)) - w(u)$. The new vertex u' represents the choice between $N(u)$ and u. Choosing $N(u)$ costs more than u, but could cover more edges. To unfold the reduced graph, include $N(u)$ if u' was part of the solution and u otherwise.

Note that if there had been any edges between vertices in $N(u)$, these edges would need to be covered. Therefore, selecting u and no vertices from $N(u)$ would not be an option, and $N(u)$ should be added and its neighborhood removed as in the first rule.

Typically, different reduction rules are executed in a fixed order, checking the applicability of the rule for each vertex before moving on to the next rule. The precise order is based on the computational cost associated with each rule, and the idea is to apply the cheaper rules frequently and the more expensive ones less often. Whenever a rule successfully reduces the graph, the vertices whose neighborhoods have changed are checked again, starting from the least expensive rule.

2.2 Local Search

Since the MWVC problem is NP-hard, only heuristics and approximation algorithms are feasible for large instances. Popular heuristics include genetic algorithms [22], ant-colony approaches [12], tabu search [27], and local search [3,19,20,23]. Among these, local search heuristics are the most successful.

Consequently, local search plays an essential part in our proposed approach. These heuristics efficiently search for improvements to an existing solution, with a predictable running time for each iteration of the search. Previous studies show that local search can quickly find high-quality solutions and scale to graphs with several million vertices and edges [20].

Algorithm 1 Local search overview, outlining the core ideas used by several iterative local search procedures for the MWVC problem.

```
1:  $C \leftarrow ConstructWVC$                          ▷ Construct the initial vertex cover
2:  $C' \leftarrow C$                                   ▷ Best vertex cover found
3:  while $elapsed\_time < max\_time$ do
4:      Remove vertex $u$ with lowest $score(u)$ from $C$
5:      while $C$ is not a vertex cover do
6:          Add vertex $v$ with highest $score(v)$ to $C$
7:          Add 1 to the weight of each uncovered edge
8:      end while
9:      if $w(C) < w(C')$ then
10:         $C' \leftarrow C$
11:     end if
12: end while
```

Most of the heuristic algorithms utilize an edge weighting strategy that dynamically changes as the search proceeds [3,20,27]. The edge weights, denoted by $edge_w$, are initialized to one. We show the common strategy of these heuristics in Algorithm 1. Here

$$cost(C) = \sum_{\{u,v\} \in E | u \notin C \wedge v \notin C} edge_w(\{u,v\})$$

$$dscore(u) = \begin{cases} cost(C \setminus \{u\}) - cost(C), & u \in C \\ cost(C) - cost(C \cup \{u\}), & u \notin C \end{cases}$$

$$score(u) = \frac{dscore(u)}{w(u)}.$$

In addition to the procedure described so far, all heuristics mentioned above make use of configuration checking, which was first introduced by Cai et al. [4]. This aims at preventing a vertex that was recently removed or added to the solution from going back to its previous state in the next iteration. Only after some other change has occurred in its neighborhood will it be allowed to change state again. Incorporating configuration checking can be as simple as defining a Boolean flag for each vertex. Only vertices with set flags are eligible for selection (Line 4). When a vertex is added to C (Line 6), the flag is set to false, but its neighbors' flags are flipped to true.

Each search iteration always starts and ends with a valid vertex cover. However, the solution quality is not guaranteed to improve during every iteration. This relaxation is necessary to escape local minima since restricting the search to only make changes that improve the solution cost will cause it to get stuck quickly. To better understand how local search works, consider the initial iteration when every edge weight equals one. The first step of each iteration removes a vertex from C with the lowest score value. The score of a vertex $u \in C$ is the weight of every edge that u alone covers, divided by $w(u)$. As a sanity check, if u is redundant and could be removed without leaving any edges uncovered, then $score(u)$ would be zero. In general, vertices with high weight and a low number of covered edges will have low scores and are likely candidates for removal. The idea being that these vertices contribute less to the overall solution. After removing a vertex, the neighbors that are not part of the solution are added back in order of their score, increasing the weight of the uncovered edges along the way. The effect of increasing the edge weights is that a newly added vertex gets higher scores than it would otherwise, and is therefore less likely to be chosen for removal. This scheme effectively handles the balance between intensifying and diversifying the search and has been demonstrated to be an efficient technique in practice [3,19,20,27].

Many successful heuristics have used the technique outlined in Algorithm 1. One such heuristic, FastWVC [20], improved performance on large graphs using a new construction procedure and exchange step that removed two vertices instead of one. This was further improved in DynWVC2 [3] by using dynamic strategies for vertex selection. Another heuristic, Hybrid Iterated Local Search (Hils) [23], alternated between efficient neighborhood swaps and random permutations to balance quality and diversity. NuMWVC [19] used simple reduction rules to construct the initial solution, improved configuration checking, and dynamic vertex selection strategies. The most recent heuristic named Master-Apprentice Evolutionary Algorithm with Hybrid Tabu Search (MAE-HTS) [27] showed further improvements over NuMWVC.

2.3 Graph Neural Networks

Graph neural networks (GNNs) [25] are machine learning architectures adapted to handle graph data. There are several issues with conventional neural network architectures when working on graphs. For instance, typical convolutional neural networks (CNNs) [15] work on images of fixed size. However, graphs differ in size and cannot easily be scaled. Furthermore, results for graphs should be invariant to different vertex permutations, which CNNs do not normally support. GNNs overcome these limitations and, following the success of CNNs, neural network architectures for graphs have recently made significant progress on combinatorial optimization problems [5]. GNN models can be trained for several different tasks, including vertex labeling, edge prediction, or whole graph predictions. GNNs can learn structural information about graphs since they consider neighborhood information for each vertex.

Starting from a feature vector on each vertex, the GNN propagates information along the edges of the graph for a fixed number of rounds. When finished the GNN outputs a value for each vertex that indicates if this vertex should be part of the solution or not. Values that are being moved between two neighbors are processed using a multi-layered perceptron that has been trained on graph instances where an optimal solution is known. Like local search, decisions taken by the GNN are not exact. A previous study on the MVC problem demonstrated the effectiveness of graph neural networks in this setting, but at a significantly smaller scale [21].

3 Approach

Our proposed heuristic consists of two stages, where the first computes a vertex cover using reduction rules, output from a GNN model, and an exact solver. In the second stage, the obtained vertex cover is improved using an efficient local search procedure. In the following, we outline the stages in more detail.

3.1 Initial Computation

The first stage starts with all vertices unclassified. It then follows a greedy strategy where reduction rules are used whenever possible to classify vertices, thereby reducing the size of the remaining unsolved graph. Classified here refers to the decision made for a vertex, regardless of whether it is in the vertex cover or not. A connectivity check is performed when no reduction rule applies, and any sufficiently small component is solved exactly using a branch-and-reduce strategy. For larger components that cannot be reduced further, the vertex with the highest probability of being either in or out of the solution is classified, according to the GNN's evaluation. This procedure is repeated until a complete vertex cover for the whole graph has been obtained.

As the computation progresses, the size of the remaining unclassified graph shrinks. As a consequence, the predictions from the GNN will also change. However, computing predictions from the GNN has a cost that is linear in the size of the remaining graph, and thus, it can be expensive to apply this too often. Therefore, it is essential to decide when to update the probabilities by rerunning the GNN and when to use probabilities from the last GNN computation. The following scheme is used to decide when to recompute the probabilities from the GNN.

Each application of the GNN gives a prediction for each vertex regarding whether or not it should be part of the vertex cover. The list of predictions is ordered by decreasing assurance. When consulting this list, the top choice might already have been classified by a reduction rule. If the current configuration of that vertex and the prediction by the GNN disagree, it is an indication that the predictions should be updated. This could still occur often, so it is also required that the remaining graph's size has shrunk sufficiently since the last time the GNN output was computed.

Like the GNN computation, checking the graph's connectivity also has a cost that is linear in the size of the remaining graph and is therefore only done before each GNN computation. Small connected components are solved exactly using a simplified version of the solver presented by Lamm et al. [18]. Due to the limited input size, some aspects have been omitted, such as branch elimination based on lower and upper bounds and connectivity checking. Instead, it is more important to target the worst-case scenario, which seems to occur in very dense graphs. To this end, a degree-based branching technique is used, as described by Imamura and Iwama [11], and based on an observation by Karpinski and Zelikovsky [14]. The idea is that for some set $S \subseteq V$ with $|S| = k$, where $\forall u \in S \implies degree(u) \geq k$. Then an optimal solution C will either include the whole set $S \subseteq C$ or the entire neighborhood for one of the vertices $\exists u \in S : N(u) \subseteq C$. In a branch-and-reduce setting, this means that one can branch based on $k + 1$ options, and each of these branches will already have at least k vertices classified. This should be compared to a traditional branching on a single vertex, where one branch typically only reduces the size of the problem by one.

To recap the construction procedure, first, apply reduction rules. Then check if the size of the graph has been sufficiently reduced. If so, perform a connectivity check, solve small connected components exactly, and apply the GNN model to the rest. If the size of the graph has not decreased enough, consult the most recent GNN predictions. Finally, when the size of the remaining graph reaches zero, unfold the graph up until the first non-exact decision was made. The last unfolding only occurs after local search.

3.2 Graph Neural Network Architecture

The GNN architecture used is a combination of message-passing steps interleaved with multi-layered perceptrons (MLPs). The message-passing step is inspired by the layer-wise propagation rule presented by Kipf and Welling [16]. It is slightly modified using a few handcrafted features and a direct passthrough, similar to a residual link used in CNNs [9]. The message-passing step then looks like this:

$$H^{l+1} = AH^l|H^l|D|W|N.$$

Here, H^l is the $|V| \times f$ matrix at layer l, where f is the number of activations for each vertex at this layer and A is the graph's adjacency matrix. Then, H^l is concatenated unchanged, followed by D, W, and N, who are $|V| \times 1$ matrices corresponding to vertex degree, weight, and neighborhood weight. The output H^{l+1} then becomes a $|V| \times 2f + 3$ matrix. Initially, $f = 1$ since the only attribute of a vertex is its weight.

The MLPs used are extensions of the original perceptron model introduced by Rosenblatt [24]. Each layer is a dense matrix of trainable weights, and in between the layers are non-linear activation functions. The forward flow through one of these layers is given by:

$$H^{l+1} = \sigma(H^l W^l + b^l)$$

where σ is the activation function, W^l are the trainable weights, and b^l is the bias term at this layer. The dimensions of W^l is $f^l \times f^{l+1}$, meaning these layers can scale f as desired. The vector b^l is added to each row of $H^l W^l$. The activation functions used are *ReLU* and *sigmoid*. These are elementwise functions defined as:

$$ReLU(x) = max(0, x)$$

$$sigmoid(x) = \frac{1}{1+e^{-x}}.$$

The model used in the proposed heuristic consists of three message-passing layers interleaved with three-layer MLPs. The MLP layers consist of 32 activations with a single activation for each vertex at the output layer. ReLU is used between every MLP layer, except in the output layer, where sigmoid is used. The benefit of the sigmoid activation in the output layer is that each value can be interpreted as a probability.

3.3 Optimized Local Search

For the second stage, a local search procedure is employed, similar to that shown in Algorithm 1. However, this contains three costly operations that need to be analyzed carefully.

1. Finding the next vertex u to remove (Line 4)
2. Reconstructing the vertex cover (lines 5-7)
3. Storing the new solution if it turns out to be an improvement (Line 10)

When done naively, the first and third points will take $O(|V|)$ time and the second $O(|N(u)|^2)$, where u is the vertex from Line 4. This might not be too costly if the graph is small and the current solution is close to the optimal. However, since our aim is to find vertex covers on massive graphs with millions of vertices, a more careful implementation could make a significant difference.

To address the linear cost of finding the next vertex to remove, we store every vertex in the graph in a binary heap. Whenever the score of a vertex changes, its position in the heap is also updated. In one remove and reconstruct step, the neighborhood of u and the neighborhood of every added vertex will change their *score* value. This means that the cost of maintaining the heap during each iteration will be $\log(|V|)$ times the size of the distance-2 neighborhood of u.

The next point is how to reconstruct the solution efficiently. Since u is not allowed to enter the solution again in the same step as it was removed, the vertices with a positive *score* are precisely the neighbors of u that are currently not in the vertex cover. Furthermore, since these vertices are not part of the vertex cover, every other neighbor they might have besides u, will already be in the vertex cover. This means that the score value for each $v \in N(u)$ where $v \notin C$ will be equal to the weight of its edge incident on u divided by $w(v)$. To improve the speed of reconstructing the vertex cover, we sort the adjacency list of u based on edge weight and then add them in that order. This improves the running time for this step from $O(|N(u)|^2)$ to $O(|N(u)|\log(|N(u)|))$. Reconstructing the vertex cover this way is not equivalent to Algorithm 1. However, it is faster and based on our experiments, has negligible impact on solution quality.

Finally, the last consideration is how to keep track of the best solution found. The best solution is not used for anything during the execution of the algorithm. It is only stored to be returned at the end. If the initial solution is far from the eventual local minima, repeatedly storing the improved vertex covers could become a bottleneck. To address this issue, multiple iterations are performed without checking if the solution quality has improved. Initially, the number of iterations is high, but gradually shrinks as the search continues. This idea is not new, as NuMWVC uses a similar technique called *self-adaptive-vertex-removing* [19]. However, in NuMWVC, the authors changed the number of vertices removed during each iteration, unlike here, where only the updating is omitted. Compared with NuMWVC, our approach also varies significantly in scope. NuMWVC starts by removing three vertices and gradually moves down to one, whereas in our approach, the idea is to let the search run for thousands of iterations without updating the best solution. It is important to note that these improvements only speed up the search and does not necessarily lead to solutions that other existing heuristics would not have found given sufficient time.

4 Experimental Setup

In the following, we present the computation platform, benchmark datasets, and details on how the GNN model was trained. We refer to our heuristic as GNN with local search (GNN & LS). The exact solver is employed on connected components with ≤ 75 vertices and requires a 5% size reduction before each connectivity check and recomputation of GNN probabilities. We require a minimum of 1024 iterations during the local search before updating the best solution.

Computing Platform. All heuristics are implemented in C++ and compiled using GCC 9.3 with the O3 optimization flag. All the experiments were run on a single thread of an Intel Xeon Silver 4112 CPU with 2.60 GHz and 38 GB of memory. The machine runs Ubuntu 18.04.6 and Linux kernel version 5.4.0-109.

4.1 Benchmark Data

We use graphs from the SuiteSparse collection [7]. In order to get graphs with a wide variety of meaningful sizes and densities, a subset of the entire SuiteSparse collection was used, selected on the basis of file size. The first set of graphs, Dataset 1, contains graphs with a file size between 40 MB and 4 GB, 371 graphs in total. The smallest graphs start at roughly 500 thousand edges, while the largest exceed 180 million[2]. Most of these graphs do not initially have vertex weights or undirected edges, so the graphs are first converted to the correct format. This is done by considering each edge as undirected, removing any duplicates or self edges, and then generating vertex weights uniformly at random. There are some differences in previous studies on how weights are assigned. For instance, weights in the integer range [20, 100], [20, 120], or [1, 200] have all been used. We use weights drawn uniformly at random in the integer range [1, 200], which is similar to B & R [18] and HILS [23]. However, based on preliminary experiments, different weight ranges did not significantly impact the results.

Some of the other heuristics used for comparison contain built-in input size limitations. For instance, NuMWVC only accepts graphs with less than 9 million vertices. Therefore, Dataset 2 consist of the graphs from Dataset 1 that every heuristic accepted as input. The exact solver B & R [18], in its default configuration and with a time limit of 1200 seconds, was able to find exact solutions for 111 graphs from Dataset 1. These 111 graphs constitute Dataset 3. Lastly, six massive graphs with more than one billion edges are also included as Dataset 4.

[2] A complete list of graphs along with results and source code can be found at this repository: `https://github.com/KennethLangedal/MWVC-GNN-LS`

4.2 Training the GNN Model

When training a GNN model directly on vertex classification, one has to take into consideration that a minimum weighted vertex cover might not be unique. One idea that has already been successfully used for the MVC problem is to have multiple vertex covers as output and use a hindsight loss that only acts on the best output [21]. We propose another approach for MWVC, where one increases the weight range to lower the chance for multiple optimal solutions. Therefore, the weights assigned to the training data are drawn uniformly at random from the integer range [10, 2000] instead of [1, 200]. The input to the model is then linearly scaled down to the real interval [0, 1], to ensure that the data is in the same range when training as during inference. Roughly 1400 graphs from the SuiteSparse collection with sizes less than 40 MB were used as training data. The exact solver B & R [18] was able to find optimal solutions for 929 of these graphs, thus giving us over 8 million labeled vertices that we could then use for training.

Mean squared error (MSE) and stochastic gradient descent (SGD) with momentum were used to fit the model's parameters to the training data. The training data was divided into 90% for training and 10% for validation. Additional parameters used during the training include a 0.01 learning rate, 0.9 momentum, and a batch size of 250,000 vertices. The results from the training can be seen in Figure 1.

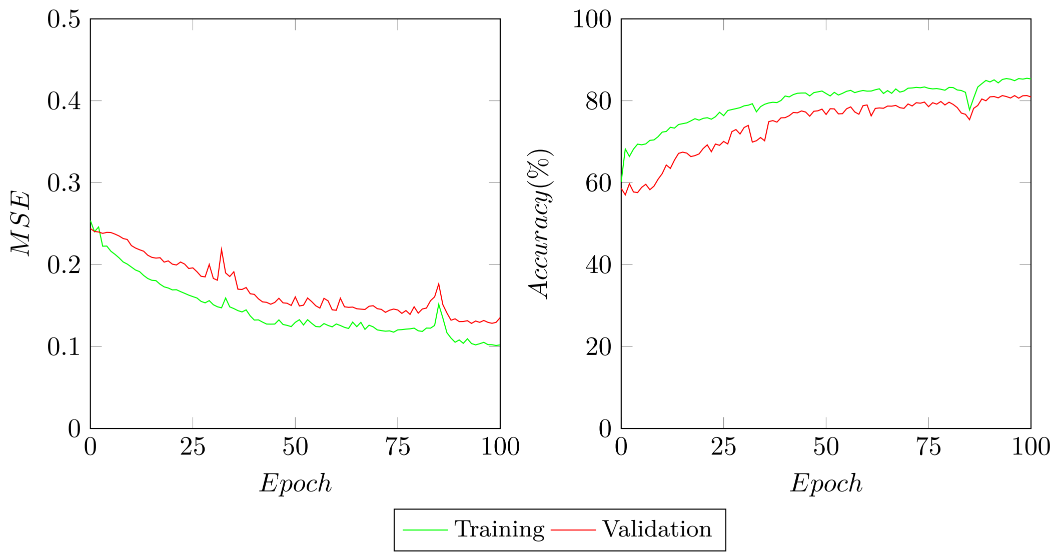

Figure 1 GNN training over 100 epochs, showing MSE and accuracy for training and validation sets.

5 Experimental Results

In this section, we report on a set of experiments to gauge the performance of our new GNN-based approach. We start with results from Dataset 2 compared to other state-of-the-art heuristics. Then, based on Dataset 1 and 3, we present a deeper analysis of the different components of our GNN-based approach. Finally, we include results on Dataset 4.

5.1 Comparison with State-of-the-art Heuristics

The first set of experiments compares GNN & LS with the following heuristics: DynWVC2, Hils, FastWVC, and NuMWVC. The source codes for all of these are available online, except NuMWVC, where the authors provided the code. There is one newer heuristic, MAE-HTS [27], but we were unable to obtain the code for it. Each heuristic ran for 1000 seconds, while recovering the best solution found. Dataset 2 was used here due to the built-in limitations on some implementations. Figure 2 shows the solution quality compared to the best solution found by any of the heuristics on each graph. The y-axis gives the percentage of graphs that a heuristic was able to solve with increasing distance to the best solution given by the x-axis. Here, the distance is measured as a percentage of the best solution. Thus the values initially show the percentage of graphs where each heuristic gave the best solution. The absolute numbers are also given in the first row of Table 1.

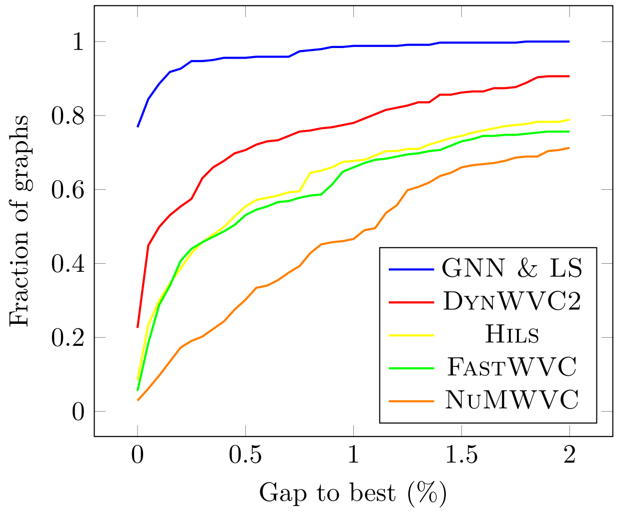

Figure 2 Solution quality on Dataset 2, based on the gap to the best solution found by one of the heuristics.

Table 1 Summary of results on Dataset 2.

	GNN & LS	DynWVC2	Hils	FastWVC	NuMWVC
Best solutions	262	77	29	19	10
Average gap to best	0.06%	0.84%	2.37%	1.44%	2.33%
Average time in sec.	480.79	735.83	907.9	826.75	669.91

As shown in Table 1, GNN & LS finds higher-quality solutions on significantly more test instances than the other heuristics. It is also, on average, closest to the best solution and uses the least time.

Table 1 gives the time it took to find the reported solution. Measuring the running time this way does not necessarily give meaningful insight into how fast the different methods are. A heuristic that finds a good solution quickly but then makes a small improvement after a long time would register as slow in this metric, while one that finds the same solution but does not make the improvement would register as fast, even though it never had a better solution than the first heuristic. In order to get a better understanding of the speed of the heuristics, we run the programs for a shorter duration and compare the results, as shown in

Figure 3. The GNN-based approach spends considerable time constructing the initial vertex cover, 27.7 seconds on average. Before that point, GNN & LS has nothing to report. Figure 3a shows the results when using the time the GNN spent constructing the initial solution as the time limit for the other programs, effectively testing the GNN construction part in isolation against the other heuristics.

Again, as can be seen from the results, GNN & LS also find higher-quality solutions with these lower time limits. Figure 3a shows that the advantage of the new heuristic is even greater when comparing the construction step alone against the other heuristics. A reason for the drop in performance from Figure 3a to 3b is due to graphs where GNN & LS did not finish constructing the initial vertex cover in the first 100 seconds.

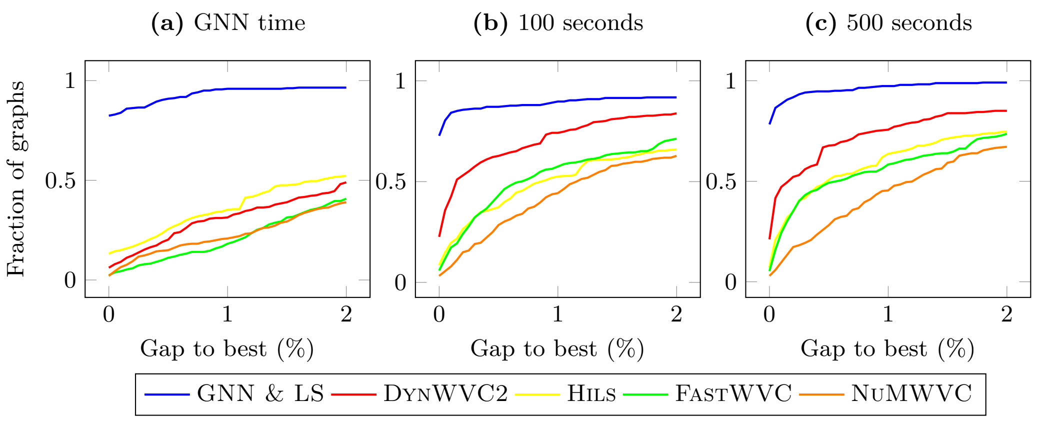

Figure 3 Solution quality on Dataset 2 using different time limits. Quality is measured based on the gap to the best solution found by one of the heuristics.

5.2 Evaluation of Different Configurations

In this section, we perform a deeper evaluation of the different components of our heuristic in isolation, with the main focus on the GNN component. The GNN's part is to label one vertex when reduction rules fail to make progress. One sensible alternative is to exclude the heaviest vertex, breaking ties based on degree, similar to how exact solvers pick a vertex to branch on. This modified version of GNN & LS will be referred to as Quick & LS. Additionally, to measure the importance of local search, both GNN and Quick without local search are also included. Lastly, a pure local search (LS) is used for comparison. These different configurations are evaluated w.r.t. running times and solution quality, including results on graphs where the optimal solution is known.

Table 2 Summary of results on Dataset 1.

	GNN	GNN & LS	Quick	Quick & LS	LS
Best solutions	65	242	57	122	77
Average gap to best	0.68%	0.01%	2.52%	0.22%	0.95%

When comparing different configurations of the proposed heuristic, GNN & LS wins on both the number of best solutions and the average gap to the best solution, as shown in Table 2. Taking away components from GNN & LS all lead to worse performance, but to different extents. Taking away the GNN, represented by Quick & LS, is the second-best

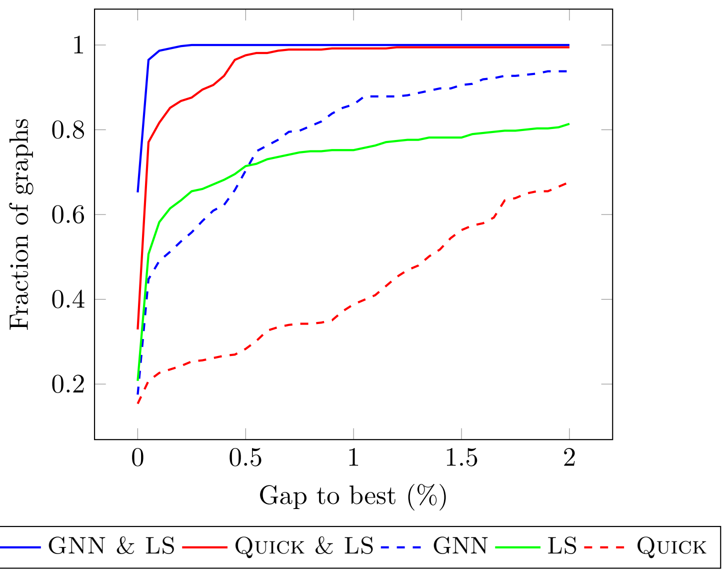

Figure 4 Solution quality on Dataset 1, based on the gap to the best solution found by one of the configurations.

configuration and shows that using reduction rules during the construction of the initial solution benefits the subsequent local search. Comparing the GNN and Quick results, both without local search, highlights the impact of the GNN model at this stage. Ultimately, the GNN, local search, and reduction rules are all responsible for significant parts of the quality of the final vertex cover.

Table 3 Summary of results on Dataset 3.

	GNN	GNN & LS	Quick	Quick & LS	LS
Optimal solutions	52	57	52	56	10
Average gap to optimal	1.12%	0.04%	3.27%	0.60%	1.77%

On Dataset 3, GNN & LS is on average closest to the optimal solutions, shown in Table 3, while the other configurations are noticeably worse. In order to find optimal vertex covers on graphs of these sizes, the graphs need to be especially amenable to reduction rules. Counting the number of vertices remaining after the initial reduction step confirms this, as 43 graphs were completely solved by reduction rules alone. On the graphs that had more than zero vertices left, the average on Dataset 3 was a reduction of 46.44%, while for Dataset 1, it was 26.7%. These numbers further demonstrate the power of reduction rules on the MWVC problem.

5.3 Running Time

So far, the focus has been on solution quality within a fixed time window. Since Dataset 1 contains several hundred graphs of various sizes, condensing this data to a single number or figure can oversimplify the results. For instance, on some graphs, the local search could quickly get stuck in a local minimum, while on others, an improved solution could be found after a long time. The interesting feature is how the solution quality changes over time. For example, the GNN-based approach takes considerable time to construct the initial solution.

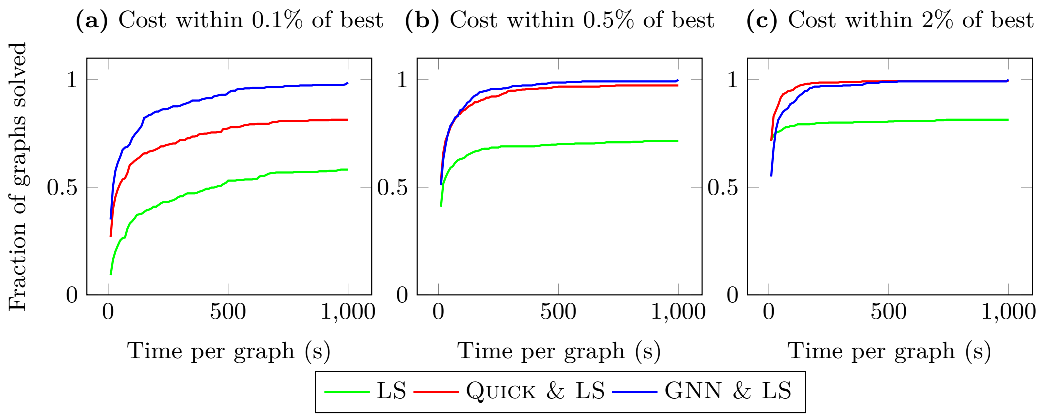

Figure 5 Performance profile on Dataset 1 showing the fraction of graphs solved over time, including different definitions of solved. Before GNN & LS and Quick & LS have solutions to report on a graph, it counts as not being solved regardless of defined threshold.

However, if that solution is better than what a pure local search could do in the same amount of time, it would likely be worthwhile. The same argument can be made for the Quick alternative as well. However, there is no guarantee that this is the case, and a worse initial solution can produce a better final solution after local search. To show how the solution quality changes over time, we ran each configuration for 1000 seconds, reporting the best solution at 10-second intervals.

The results, seen in Figure 5, show that there is a time/quality tradeoff that comes with the GNN component. When the best solution quality is desired, GNN & LS is the best choice (Figure 5a). However, since GNN & LS is slowest to produce a feasible solution, there comes the point where omitting the GNN component becomes beneficial, as shown in Figure 5c.

5.4 Large Instances

Table 4 Results from Dataset 4. Each configuration ran for 3000 seconds. The best solutions are indicated in bold.

	GNN & LS		Quick & LS		LS	
Graph	Cost	Time	Cost	Time	Cost	Time
webbase-2001	**3,440,309,297**	311.03	3,442,765,890	176.72	3,539,284,688	62.68
it-2004	**1,438,951,442**	749.59	1,439,091,400	490.23	1,480,088,254	76.98
GAP-twitter	**1,160,408,463**	724.04	1,160,512,688	712.48	1,201,023,583	30.18
twitter7	**1,160,187,362**	763.95	1,160,291,460	680.71	1,200,789,078	29.82
GAP-web	**1,861,729,481**	2,562.73	1,883,467,468	1,724.99	1,932,504,652	227.39
sk-2005	**1,861,502,940**	2,797.92	1,882,779,425	1,788.75	1,932,262,816	221.15

Dataset 4 contains six larger graphs to show that our GNN-based approach can scale beyond the problems sizes investigated in previous work. The results are shown in Table 4, with GNN & LS giving the best quality on all of these instances. The final solution cost of GNN & LS and Quick & LS is very similar. This is due to the effect the reduction rules has on these graphs, as shown in Table 5. They are particularly effective on GAP-twitter and twitter7, with less than 200,000 vertices left after the initial round of reductions. As a consequence, no matter how the remaining graphs are solved, the solution quality of the different methods is very close, but it is relevant that the GNN finds better solutions.

Table 5 Reduction rules on Dataset 4.

Graph	|V|	|E|	|V| after reduction
webbase-2001	118,142,155	854,809,760	4,936,598
it-2004	41,291,594	1,027,474,946	8,826,771
GAP-twitter	61,578,415	1,202,513,046	164,512
twitter7	41,652,230	1,202,513,046	165,489
GAP-web	50,636,151	1,810,063,329	16,803,173
sk-2005	50,636,154	1,810,063,329	16,756,003

6 Conclusion and Future Work

We have demonstrated that GNNs can boost the performance of heuristics for the MWVC problem. We have also introduced a local search implementation for large sparse graphs that avoids frequently occurring $O(|V|)$ steps. Our complete heuristic also incorporates previously established reduction rules and an exact solver. Extensive experiments on several hundred large graphs show that our heuristic significantly outperforms previous methods. We also demonstrate that every part of our strategy is needed to achieve these results and show that it can scale to larger graphs than previously considered, including graphs with more than 1 billion edges.

Despite our promising results, it is clear that each component of our strategy can be improved further based solely on existing work. For instance, the exact solver used on small connected components could also be significantly improved, evident by the success of solvers like B & R. The proposed improvements to local search are primarily focused on implementation, and there is undoubtedly room to include techniques from recent work in this area as well. Similarly, using a more sophisticated GNN architecture or increasing the amount of training data is likely to improve the performance. Naturally, it is also possible to apply the same strategy to new problems. This is something we intend to investigate in the future.

References

1 Ferhat Ay, Manolis Kellis, and Tamer Kahveci. Submap: aligning metabolic pathways with subnetwork mappings. *Journal of Computational Biology*, 18(3):219–235, 2011.

2 Ken Been, Eli Daiches, and Chee Yap. Dynamic map labeling. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):773–780, 2006.

3 Shaowei Cai, Wenying Hou, Jinkun Lin, and Yuanjie Li. Improving local search for minimum weight vertex cover by dynamic strategies. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1412–1418. International Joint Conferences on Artificial Intelligence Organization, July 2018. `doi:10.24963/ijcai.2018/196`.

4 Shaowei Cai, Kaile Su, and Abdul Sattar. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artificial Intelligence*, 175(9-10):1672–1696, 2011.

5 Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint*, 2021. `arXiv:2102.09544`.

6 Lijun Chang, Wei Li, and Wenjie Zhang. Computing a near-maximum independent set in linear time by reducing-peeling. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1181–1196, 2017.

7 Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1), December 2011. `doi:10.1145/2049662.2049663`.

8 Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdan Zavalnij. Boosting data reduction for the maximum weight independent set problem using increasing transformations. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 128–142. SIAM, 2021.

9 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

10 Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. WeGotYouCovered: The winning solver from the pace 2019 challenge, vertex cover track. In *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, pages 1–11. SIAM, 2020.

11 Tomokazu Imamura and Kazuo Iwama. Approximating vertex cover on dense graphs. In *Symposium on Discrete Algorithms: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, volume 23, pages 582–589, 2005.

12 Raka Jovanovic and Milan Tuba. An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem. *Applied Soft Computing*, 11(8):5360–5366, 2011.

13 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

14 Marek Karpinski and Alexander Zelikovsky. *Approximating dense cases of covering problems*. Citeseer, 1996.

15 Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516, December 2020. `doi:10.1007/s10462-020-09825-6`.

16 Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint*, 2016. `arXiv:1609.02907`.

17 Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F Werneck. Finding near-optimal independent sets at scale. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 138–150. SIAM, 2016.

18 Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 144–158. SIAM, 2019.

19 Ruizhi Li, Shuli Hu, Shaowei Cai, Jian Gao, Yiyuan Wang, and Minghao Yin. NuMWVC: A novel local search for minimum weighted vertex cover problem. *Journal of the Operational Research Society*, 71(9):1498–1509, 2020.

20 Yuanjie Li, Shaowei Cai, and Wenying Hou. An efficient local search algorithm for minimum weighted vertex cover on massive graphs. In *Asia-Pacific Conference on Simulated Evolution and Learning*, pages 145–157. Springer, 2017.

21 Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *arXiv preprint*, 2018. `arXiv:1810.10659`.

22 Sk Md Abu Nayeem and Madhumangal Pal. Genetic algorithmic approach to find the maximum weight independent set of a graph. *Journal of Applied Mathematics and Computing*, 25(1):217–229, 2007.

23 Bruno Nogueira, Rian G.S. Pinheiro, and Anand Subramanian. A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optimization Letters*, 12(3):567–583, 2018.

24 Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386, 1958.

25 Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

26 Changbing Tang, Ang Li, and Xiang Li. Asymmetric game: A silver bullet to weighted vertex cover of networks. *IEEE Transactions on Cybernetics*, 48(10):2994–3005, 2017.

27 Yang Wang, Zhipeng Lu, and Abraham P Punnen. A fast and robust heuristic algorithm for the minimum weight vertex cover problem. *IEEE Access*, 9:31932–31945, 2021.