Targeted Branching for the Maximum Independent Set Problem Using Graph Neural Networks

Kenneth Langedal[1] ✉ 🄪
University of Bergen, Department of Informatics, Norway

Demian Hespe ✉ 🄪
Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Germany

Peter Sanders ✉ 🄪
Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Germany

—— **Abstract** ——

Identifying a maximum independent set is a fundamental NP-hard problem. This problem has several real-world applications and requires finding the largest possible set of vertices not adjacent to each other in an undirected graph. Over the past few years, branch-and-bound and branch-and-reduce algorithms have emerged as some of the most effective methods for solving the problem exactly. Specifically, the branch-and-reduce approach, which combines branch-and-bound principles with reduction rules, has proven particularly successful in tackling previously unmanageable real-world instances. This progress was largely made possible by the development of more effective reduction rules. Nevertheless, other key components that can impact the efficiency of these algorithms have not received the same level of interest. Among these is the branching strategy, which determines which vertex to branch on next. Until recently, the most widely used strategy was to choose the vertex of the highest degree. In this work, we present a graph neural network approach for selecting the next branching vertex. The intricate nature of current branch-and-bound solvers makes supervised and reinforcement learning difficult. Therefore, we use a population-based genetic algorithm to evolve the model's parameters instead. Our proposed approach results in a speedup on 73% of the benchmark instances with a median speedup of 24%.

---

[1] Corresponding author

1 Introduction

An independent set of an undirected graph $G = (V, E)$ with vertex set V and edge set E, is a subset $I \subset V$ such that the vertices in I are pairwise non-adjacent, i.e. $\forall u, v \in I : \{u, v\} \notin E$. The problem of finding an independent set of maximum cardinality is a fundamental NP-hard problem [15]. Its applications span computer graphics [44], network analysis [41], route planning [24], and computational biology [3, 7]. More applications use its complementary problems minimum vertex cover and maximum clique.

Branch-and-reduce is one of the most successful techniques in both theory [6, 54] and practice [1, 18, 19, 40] to solve the maximum independent set problem (MIS). Branch-and-reduce algorithms combine classical branch-and-bound algorithms with repeated application of so-called *reduction rules* after every branching step. Reduction rules can remove known parts of the graph that are provably in or out of some maximum independent set or such that a solution for the reduced graph can be lifted to a solution for the original graph. These reduction rules can often drastically reduce the size of the graphs, leading to faster processing times.

In the past, research has mainly been focused on developing reduction rules [2, 5, 8, 20, 49]. Recently, other aspects like search space pruning [40] and branching vertex selection [18] have been shown to have considerable impact on the performance of branch-and-reduce solvers. Some first results using machine learning have also shown to be effective for heuristic (inexact) solvers [26, 32], where vertices that are likely to be part of a solution are predicted.

2 Related Work

In this section, we cover the existing work on branching techniques and introduce graph neural networks. We also give a brief overview of the different learning paradigms used in this area.

2.1 Branching Vertex Selection

The most commonly used strategy for selection a branching vertex for maximum independent set and minimum vertex cover is to select a vertex of maximum degree [1, 14, 19, 40]. Akiba and Iwata [1] compare this strategy to branching on a vertex of minimum degree or a random vertex and show that both of these perform substantially worse. Wang et al. [53] also compare maximum degree branching to other simple heuristics for the weighted version of minimum vertex cover and find maximum degree branching to perform best. Hespe et al. [18] introduce several new branching vertex selection strategies and show that targeting vertices whose removal enables new reductions consistently leads to improved or equal performance compared to branching on the highest degree vertex. They show this result on benchmark instances from the 2019 PACE challenge on vertex cover [12] and the 1993 DIMACS challenge on maximum clique [21].

Most algorithms for maximum clique use a greedy algorithm to compute a vertex coloring and branch on vertices with a high coloring number [47, 51]. Other algorithms use the so-called *degeneracy order* of the graph [4] or iterative maximum independent set computations [29] to determine the branching order. Some algorithms use MaxSAT encodings to prune the set of branching vertices [29, 30, 31].

2.2 Graph Neural Networks

Graph Neural Networks (GNNs) are machine learning models that process and analyze data structured as graphs [45]. Traditional neural networks are not well-suited for processing graph-structured data since they operate on fixed-sized input vectors. GNNs address these limitations by introducing specialized architectures that handle graph data effectively. One popular GNN architecture is the Graph Convolutional Network (GCN) introduced by Kipf and Welling [25]. At each layer in a GCN, each node aggregates information from its immediate neighbors and combines it with its own data. After this, the information stored in each node is passed through a layer-specific neural network to create the node information for the next layer. This architecture is the same one we use to build our GCN models, and it is defined as follows: Let A be the adjacency matrix for an undirected graph with added self-edges and D be the diagonal degree matrix, where $D_{ii} = \sum_j A_{ij}$. Each vertex has a feature representation that changes at each layer in the model. The length d of the feature representation at layer l is denoted by $d^{(l)}$. Stacking all the feature vectors at the l'th layer gives the matrix $H^{(l)} \in \mathbb{R}^{|V| \times d^{(l)}}$. Finally, as in traditional neural networks, we have the trainable parameters $W^{(l)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ and bias $b^{(l)} \in \mathbb{R}^{1 \times d^{(l+1)}}$, in addition to a non-linear activation function σ, such as $\text{ReLU}(x) := \max(0, x)$. With this, we use the following layer-wise propagation rule.

$$H^{(l+1)} = \sigma(D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{(l)} W^{(l)} + b^{(l)})$$

In our case, there are no features associated with the vertices. The only input to the problem is the graph itself. Therefore, we set the input dimension $d^{(0)} = 1$ and initialize $H^{(0)}$ with all ones.

To clarify the propagation rule further, the first part of the equation $D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{(l)}$ is a message-passing step where each vertex aggregates information from its neighbors. The inclusion of the two $D^{-\frac{1}{2}}$ matrices scales each message with both the sender and receiver's degree. The rest, $\sigma(H^{(l)} W^{(l)} + b^{(l)})$, is like any traditional neural network. Both H and W are dense matrices where each row in H corresponds to the feature vector of one vertex. The bias term b is a vector added to each row of the output at this layer. See Figure 1 for an illustration of one GCN layer.

Supervised Learning

A widely used technique to train GNNs for combinatorial optimization tasks is supervised learning. Examples of this include the traveling salesperson problem [22], coloring [27], maximum cut [46], and maximum independent set [26, 32]. Given a set of training examples with associated labels or targets, supervised learning aims to adapt the model's parameters based on these examples and targets. More specifically, using a loss function that measures how well the current parameters align with the target outputs, the problem is to minimize this loss value. Assuming the loss function is continuous and differentiable, we can compute gradients for each parameter and perform gradient descent. For a more in-depth introduction to supervised learning, see, for instance, Goodfellow et al. [16].

Reinforcement Learning

Unlike supervised learning, reinforcement learning does not rely on training examples. Instead, an agent—in this case, a GNN model – explores an environment by taking actions. After choosing an action, the new state is evaluated into a reward and passed back along with the

Input   Message-Passing   Neural Network

$$T = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{(0)}$$ $$H^{(1)} = \sigma(TW^{(0)} + b^{(0)})$$

$$H^{(0)} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} |V| \qquad T = \begin{bmatrix} 1.18 \\ 0.85 \\ 0.96 \\ 0.96 \end{bmatrix} \begin{matrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{matrix}$$
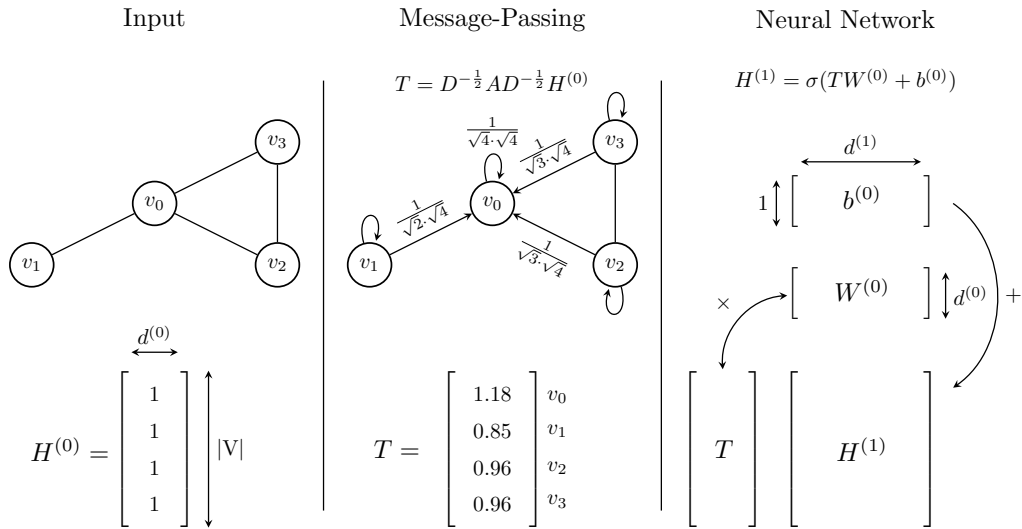
Figure 1 Illustration showing the layer-wise propagation rule introduced by Kipf and Welling [25]. The first column shows the input graph and input matrix $H^{(0)}$. Since the input is only the graph itself, the feature matrix $H^{(0)}$ is initialized with a single 1 for each vertex. The second column shows the message-passing step. The matrix T is only used in this figure as the temporary result between the message-passing and the dense neural network part. Since the message-passing step uses $D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$, the output for each vertex depends on both its own degree and the degree of each vertex in its neighborhood. The last column shows the dense neural network part. Here we do a dense matrix multiplication between T and the trainable parameters at this level $W^{(0)}$ to get $H^{(1)}$. Finally, the bias vector $b^{(0)}$ is also added to each row of $H^{(1)}$. The activation function is not shown in the illustration but is applied to each element of $H^{(1)}$ at the very end.

state to the agent. The goal is to maximize this cumulative reward. This learning paradigm saw a surge in popularity after the success of AlphaGo for the game of Go [48]. It has since also been successfully applied to combinatorial optimization problems with GNNs. Examples of this include the traveling salesperson problem [10, 23], vehicle routing [39], subgraph matching [52], and minimum vertex cover [23].

Genetic Algorithms

A far less common technique is to use genetic algorithms to train the model's parameters. Genetic algorithms make use of biology-inspired mechanisms to evolve a solution over time. These mechanisms include crossover, mutation, and selection [33]. Historically, some contention existed between supervised learning using backpropagation and genetic algorithms [36]. However, the former turned out to be far more successful, especially in light of computer vision. Still, recent results have shown that genetic algorithms can be a competitive alternative to reinforcement learning [13, 43, 50]. In these comparisons, the problem was Atari games or similar benchmarks. To our knowledge, training GNNs using genetic algorithms has yet to be applied to combinatorial optimization tasks.

3   GCN Guided Branching

We now briefly motivate our GCN-based branching strategy before describing how we train and use the models. For motivation, several good arguments exist for using machine learning for branch selection. First, the default strategy used in most solvers – branching on the

highest degree vertex – is already a heuristic. Regardless of the strategy used, the correctness of the algorithm is not changed, and the asymptotic running time remains the same. However, we know from previous studies that the choice of branching strategy can have a large impact on overall performance. The benchmark instances typically used are not arbitrary graphs either. For example, the instances from the 2019 PACE challenge include transit graphs, road networks, and social network graphs [12]. It is reasonable to think that there are patterns in this data that a machine-learning model can pick up and learn new branching strategies from.

At a high level, our strategy is to let a GCN model decide the next vertex to branch on. Before each branch, we pass the remaining graph as input to a pre-trained GCN model. The output from the model is a single value for each vertex ($d = 1$ at the last layer). The solver then uses the vertex with the highest value as the next branching vertex.

Training a GCN model to pick the next vertex to branch on is problematic from a machine-learning perspective. Ideally, we would like to do supervised learning with labeled data. However, it is not obvious how to generate such data. Several factors could make one branch better than another. For instance, after branching, the resulting graph could:

- Break apart into multiple connected components
- Yield better lower/upper bounds
- Shrink further due to reduction rules

Furthermore, these benefits may not manifest immediately. Instead, the best strategy may be to perform several seemingly bad branches that eventually lead to early termination. This makes it difficult to find optimal branching sequences, even for small instances. Previous results show that it can be done [38], but we argue that two main issues still remain.

The first is non-unique solutions. As with the independent set problem itself, an optimal solution is not necessarily unique. Also, the intersection between optimal solutions could be empty, meaning a GCN could output an optimal solution while being completely wrong measured against the known target. In their supervised learning approach to the MIS problem, Li et al. [32] addressed this issue by having the GCN model output multiple solutions and using the best one to compute gradients, using hindsight loss. While this does address the problem, it's not a solution.

The second problem is that following a branching sequence close to the optimal could result in a vastly different execution time compared to the optimal. For example, consider the case where the optimal sequence splits the graph into two connected components. If only one vertex appeared later in the sequence, the graph could remain connected, and the execution time could be worse than the maximum degree branching. The point is that we would be training on a proxy problem, and a model that achieves low loss on the training data would not necessarily perform well in practice. This also distinguishes this problem from the broader branch of prediction algorithms [34], which should be learnable with few examples. For these reasons, we do not attempt supervised learning.

The next logical step would be to use some form of reinforcement learning. The problem of picking the next branch fits nicely into the agent-action-reward cycle. However, the main problem with this approach is the intricate implementations of the existing solvers. Moving between states has the added cost of checking reduction rules, testing connectivity, and searching for lower bounds. It would not make sense to turn off these parts, as it's precisely this interaction between the branching and other elements of the solver we wish to learn. Reinforcement learning is certainly a promising approach, but we also leave this for future work.

Having excluded supervised and reinforcement learning, we move on to evolutionary-based approaches. The main advantage of this approach is that we can train directly on reducing execution times. This means we avoid the problems mentioned for supervised learning because any output resulting in the same running time reduction is rewarded equally. Using a similar genetic algorithm as Such et al. [50], we start from a population of N randomly initialized GCN models. At every iteration, we assign each model a fitness score (see Section 3.2). The genetic algorithm performs truncation selection, where the top T best-performing models become parents for the next generation. More specifically, when moving from one generation to the next, the best performer is copied directly to the next generation. The rest are mutated or crossovers from models among the top T performers in the previous population.

3.1   Genetic Operators for Neural Networks

Genetic algorithms are not commonly used directly on the parameters of a model. However, there are some previous studies that have introduced genetic operators specifically for neural networks. In the following, we present the operators that we use in our training procedure.

Mutate-Weights

The simplest form of generating a new model is by simply copying the parent's weights and adding Gaussian noise to them. It was the only operator used by Such et al. [50]. Montana and Davis also tested this in an earlier study [36]. However, they showed that it performs poorly compared to slightly more complicated operators.

Mutate-Nodes

The nodes here do not refer to the input graph's vertices but rather the nodes of the neural network. Typically, neural networks are depicted as directed graphs, with the parameters being edge weights. Based on this picture, the *mutate-nodes* operator selects a few random non-input nodes in the neural network and adds Gaussian noise to the incoming edge weights. As for the GNN architecture presented in this paper, this operator selects random columns from the W matrices to mutate, including the associated bias value. Montana and Davis [36] introduced this operator and showed it to outperform *mutate-weights*.

Crossover-Directed

This operator takes two parents and creates one offspring. The idea is to use the most important parameters from each parent and fill in the rest with random values [13]. More specifically, a new model is created as follows:

1. Remove a fraction ζ of the parents' weights closest to zero.
2. Set the child's parameters to the larger of the parents' values.
3. Add random weights to make up for the lost values.

Faycal and Zito [13] introduced this operator and argued that the largest weights contribute the most to a model's success and should therefore be the ones passed on to the offspring. They base this argument on another result by Mocanu et al. [35] demonstrating that neural networks with sparse connections can achieve the same accuracy as fully connected ones.

Crossover-Nodes

As in *mutate-nodes*, *crossover-nodes* refer to the nodes in the neural network, not the input graph. This operator also starts from two parents and produces one offspring as follows: For each node in the neural network – columns in the H matrix – flip a coin and copy the incoming edge weights from the first or second parent [36]. The intuition for this operator is that the edge weights going into one node form a natural subgroup of all the parameters in the model. This is because these weights will determine one feature for the next layer. When combining two parents to make a new model, it is not likely that a random mix of their parameters will lead to a better-performing model. Using this crossover operator instead, learned features from each parent could combine into a better-performing offspring.

We use all the operators mentioned so far to produce the next generation. As stated earlier, the best-performing model is copied unmodified to the next generation. The rest of the population is generated by choosing a random operator and then picking one or two parents based on the selected operator. The parents are chosen uniformly at random from the top T best performers from the previous generation.

3.2 Fitness Function

Deciding how to define fitness depends on what results one aims for. There are sensible aggregated measures, such as the total time to solve a set of benchmark instances or the average speedup compared to max. degree branching. The types of models one gets from training with different fitness functions can vary greatly. For example, in the case of total time, models that perform well on the longest-running instances will score the best. This model could be preferable as it may lead to more solved instances. But on the other hand, it could also perform significantly worse on easier instances. Average speedup or geometric mean of speedup are both options that treat fast and slow running instances equally. In this case, the aim is to have a model that generally makes the solver run faster. The three options mentioned so far, total, average, and geometric mean, are all reasonable. Table 1 and Figure 2 illustrates how these different fitness functions could impact the final performance. We use the geometric mean of speedup as our fitness function since it is the most robust in Table 1 and Figure 2, and it treats all instances equally, which is reasonable when aiming for a model that performs well on unknown inputs.

Table 1 Table of aggregated total speedup, average speedup, and the geometric mean of speedup for each of the respective fitness functions. The columns are models trained on each fitness function, and the value at each row is how the model performed according to a different aggregated measure. For this example, the baseline is max degree branching, and the solver is *BnR*. The source of these aggregated values can be seen in Figure 2.

↓ eval. │ trained on→	total	average	geom. mean
total	**1.19**	1.01	1.16
average	1.17	**1.22**	1.20
geom. mean	1.14	1.17	**1.19**
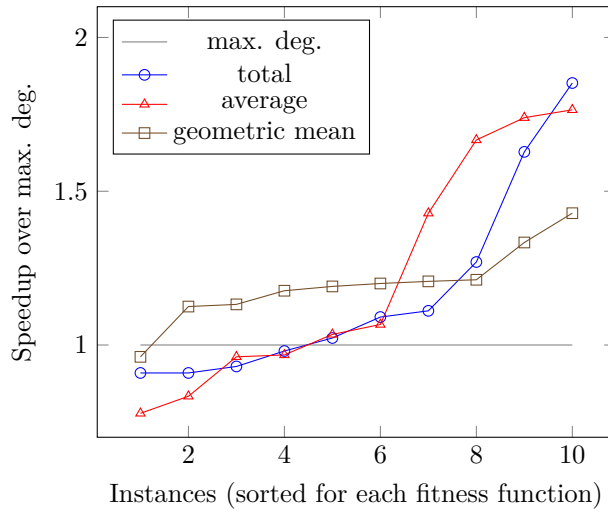
Figure 2 Illustration of how different fitness functions can lead to different results. The plot shows the speedup compared to max. degree branching for models trained with different fitness functions. Each of the models is best in their respective aggregated measure. This experiment used ten randomly selected graphs from the training data and the *BnR* solver. It only shows the difference fitness function can make and is otherwise unrelated to the later results.

4   Experiments

In the following, we present the computation platform, benchmark instances, details on how the GCN models were trained, and results from the experimental evaluation. When running the solvers, we record two measures: the execution time and the number of branches the solver checks. The number of branches can give an indication of how much overhead the GCN computation adds. The plots show speedup over max. degree branching, where speedup is computed by dividing the execution time of max. degree branching by the execution time of the respective alternative. Similarly, in the case of branches, the speedup is the number of branches with max. degree branching divided by the number of branches with the respective alternative. For a complete overview of the results, see the appendix.

4.1   Experimental Environment

We insert the GCN-based branching strategy into two pre-existing solvers. The first is the branch-and-reduce solver by Akiba and Iwata [1], also used by Hespe et al. [18] to find better branching strategies. And the second is the SAT-and-reduce solver by Plachetta and van der Grinten [40]. These two solvers will be referred to as *BnR* and *SnR*, respectively. Both solvers and the GCN implementation are written in C++ and compiled with g++ version 9.4.0 using the -O3 flag. The code for both of the modified solvers is publicly available on GitHub[23]. The machine used to execute all the experiments has an AMD EPYC 7551P 32-core processor with 256GB of DDR4 ECC memory running Ubuntu 20.04.4 with Linux Kernel 5.4.0-124. We run one instance per NUMA node to speed up the experiments, so 4 instances are run concurrently. To reduce noise and maintain fairness, only a single solver runs simultaneously, and the instances are started in the same order for each solver.

---

[2] `https://github.com/KennethLangedal/CutBranching-GNN`
[3] `https://github.com/KennethLangedal/vc-satreduce-gnn`

4.2   Instances

Table 2 List of the instances used for testing. Along with the number of vertices and edges for each graph.

Graph	$|V|$	$|E|$
as-skitter	1,696,415	11,095,298
baidu-relatedpages	415,641	2,374,044
bay	321,270	397,415
col	435,666	521,200
fla	1,070,376	1,343,951
hudong-internallink	1,984,484	14,428,382
in-2004	1,382,870	13,591,473
libimseti	220,970	17,233,144
musae-twitch_DE	9,498	153,138
musae-twitch_FR	6,549	112,666
petster-fs-dog	426,820	8,543,549
soc-LiveJournal1	4,847,571	42,851,237
web-BerkStan	685,230	6,649,470
web-Google	875,713	4,322,051
web-NotreDame	325,730	1,090,108
web-Stanford	281,903	1,992,636
PACE, 200 graphs	153 - 138,141	625 - 227,241
DIMACS, 80 graphs	28 - 4,000	72 - 3,997,732

We evaluate our GCN-based approach on the same instances the authors of *BnR* and *SnR* used. Both of these solvers were evaluated on instances from the PACE 2019 Challenge on Minimum Vertex Cover [11]. *BnR* further evaluated the branching techniques on instances from the second DIMACS Implementation Challenge [21], the Stanford Network Analysis Project (SNAP) [28], the 9th DIMACS Implementation Challenge on Shortest Paths [9], and the Network Data Repository [42]. Table 2 shows more information about the instances. Individually for each solver, the instances with less than 10 branches or less than 0.1 seconds running time are excluded for that solver. Furthermore, instances are also excluded if no branching strategy solves them within 30 minutes. When computing aggregated speedups, if one strategy times out and another finishes within 30 minutes, the strategy that times out receives the 30 minutes as execution time and the same number of branches as the strategy that finished. A time limit of 30 minutes is commonly used for evaluating exact solvers. It was used by *SnR* and is also the standard time limit for the PACE challenge [40, 12]. Due to the exponential running time for these solvers, it is unlikely that longer execution times would drastically change our results. *BnR* included results with a 10-hour time limit, and on the PACE instances, only two additional instances were solved [18].

For training the GCN models, we need to use a different dataset. The PACE 2019 Challenge drew the final 200 instances from a larger collection of over 12 thousand instances. Filtering out the 200 instances that ended up in the challenge and those with less than 0.1 seconds or more than 1 minute running time, we are left with roughly 1700 instances to use for training and validation.

4.3   GCN Training
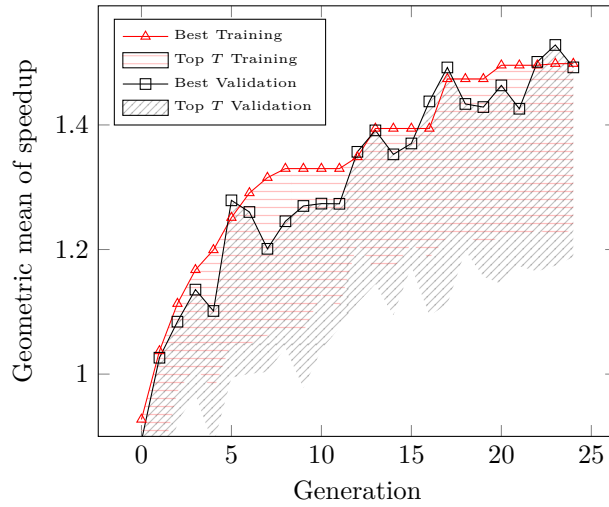
■ **Figure 3** Results from 25 generations using the *BnR* solver. The shaded red area shows the range of performance from the top T models on the training data. Similarly, the gray area shows the same for the validation data.

We use a population size of $N = 100$ and $T = 10$ parents for the next generation to train the GCN models. Each mutation operator is equally likely to be used, and we use $\zeta = 0.3$ for the *crossover-directed* operator. The models are initialized using weights drawn uniformly at random in the real interval $[-1, 1]$. The architecture of the models consists of three GCN layers with 32 features in the hidden layers. When adding noise to the weights in *mutate-weights* and *mutate-nodes*, we use a normal distribution with a mean of 0 and a standard deviation of 0.01. Finally, we remove the two highest speedups when computing the fitness. The reason for doing this is that, in some instances, the running time can be reduced to virtually zero by picking the correct vertex to branch on. When this happens, the population gets stuck since any change in that particular graph leads to a very large drop in fitness. The goal is to train a model that performs well in general, not optimally on a select few graphs.

The results from 25 generations using the *BnR* solver can be seen in Figure 3. The red line in the figure shows the fitness of the best model on the training data. Since we copy the best model unmodified, the fitness for this model will never decrease. However, the fitness for the rest of the models in the top T can vary from generation to generation. The shaded red area in the figure illustrates this, showing the range of fitness for the top T models. The black line in the figure shows the best fitness on the validation data. And similarly, the shaded black area shows the range of fitness for the top T models on the validation data. Note that the best performer on the validation set is not necessarily the best on the training data.

Population-based training of neural networks like this is highly parallelizable. Evaluating the fitness of each model can be done in parallel with no communication or synchronization. However, using execution time in the fitness function made this more challenging. Modern CPUs have dynamic clock speeds that fluctuate depending on the current workload and power/temperature limitations. Loading up all the cores will typically reduce the clock speeds significantly compared to a sequential program. Both cache and memory bandwidth are also shared resources that can impact the execution time. To address these issues, we

lowered the machine's clock speed until we got stable execution times while running several instances in parallel. Doing this introduces a new problem since it changes the cost of memory access compared to higher clock speeds. Still, the difference between running the training sequentially and in parallel is at the order of weeks compared to days, so we opt for the lowered clock speed. In the end, executing one iteration took roughly 30 minutes, with some uncertainty based on the overall quality of the particular generation. The whole training procedure took two days to finish.

4.4 Experimental Results for *BnR*
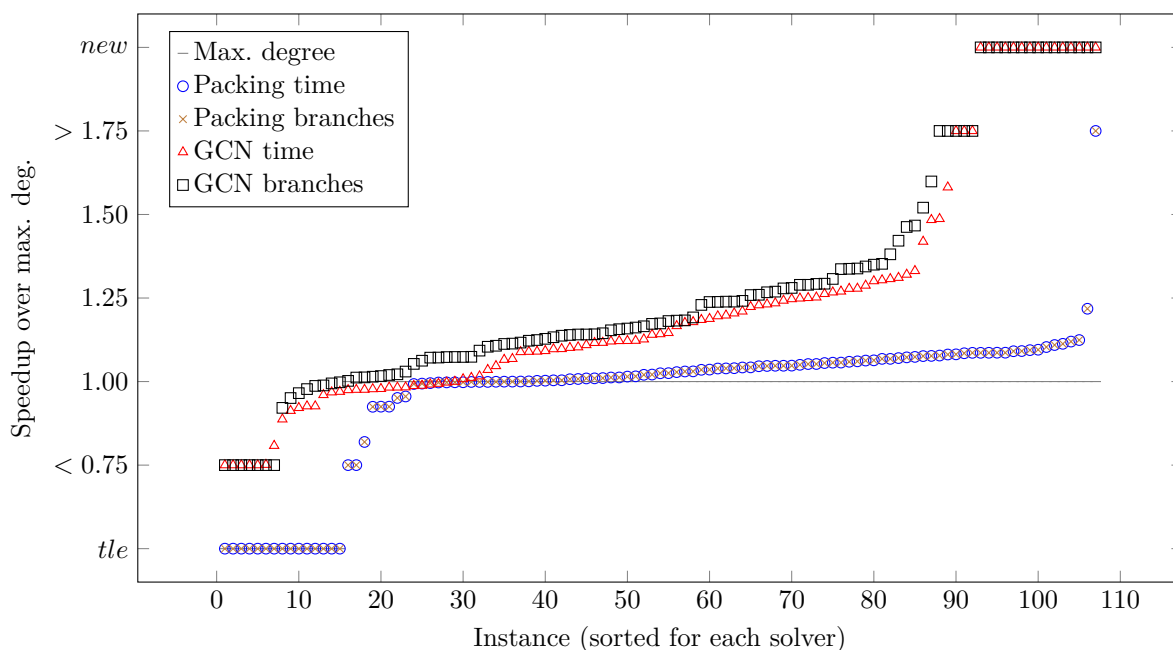
Figure 4 Cactus plot showing the speedup of our GCN-based branching and targeting the packing reduction rule in the *BnR* solver. The basis for computing speedup is the max. degree branching strategy. Instances are sorted by the speedup over max. degree branching, individually for each configuration. Instances that timed out are ranked the lowest and marked with *tle*, instances that were unsolved by max. degree branching but solved by another configuration are ranked the highest and marked with *new*. This figure shows results for the whole set of benchmark instances (excluding those solved by no configuration and those solved quickly or with very few branches), including the speedup comparing execution time and number of branches.

We start by showing the results of our GCN-based branching strategy on the whole dataset using the *BnR* solver. The results are shown in Figure 4, along with the best strategy by Hespe et al. [18] targeting the packing reduction rule. The GCN-based strategy clearly outperforms both the max. degree and packing strategies. In addition to being the fastest on the majority of instances, it also solves 15 more instances that timed out with both the max. degree and packing strategies. For this solver, the speedup in terms of branches closely resembles the speedup in execution time. However, there is a more noticeable gap between the two for the GCN strategy.

The 296 graphs used for testing can be categorized as follows:

- Unsolved by any configuration: 147
- Solved with less than 10 branches or in less than 0.1 seconds: 42
- Remaining instances shown in Figure 4: 107

Regarding aggregated measures, the result is a total speedup of 2.37 over maximum degree branching, the geometric mean of the speedups is 1.94, and the average speedup is 547.04. Note that outliers heavily influence these measures. For instance, the speedup sometimes reaches the thousands on graphs where max. degree and packing timed out. Besides looking at the tables in the appendix or Figure 4, we can observe that our GCN-based branching is fastest on 78 out of 107 instances. Among the graphs with a speedup, the median speedup is 24%. In terms of checking the fewest branches, the number increases to 92 out of 107, with a median speedup of 24%.
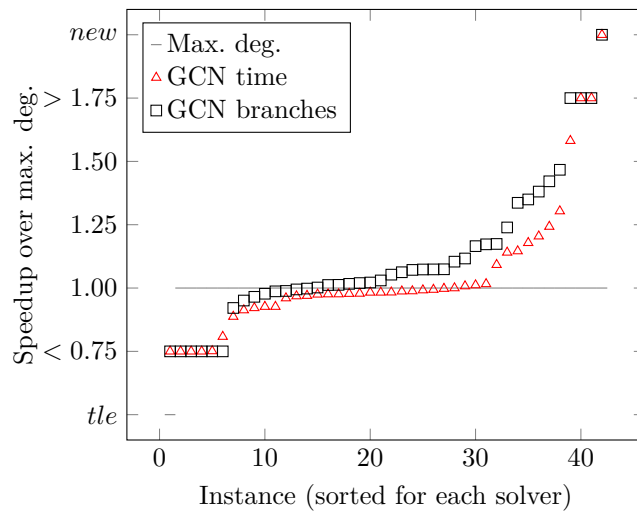
Figure 5 Speedup of our GCN-based branching compared to max. degree branching for the *BnR* solver on the DIMACS and the other sparse instances. The basis for computing speedup is the max. degree branching strategy. Instances are sorted by the speedup over max. degree branching.

The results are substantially better on the PACE instances than on the DIMACS and the other sparse instances. In fact, there is only one graph from the PACE challenge where our GCN-based strategy performs worse than max. degree branching. This is not surprising since the training data is more closely related to the PACE instances. In Figure 5, we show the speedup on only the DIMACS and sparse instances. It is clear that the results in terms of the number of branches are better than the execution time. This could indicate that the GCN model is choosing better branches than simply going for the highest degree, but the added overhead of invoking the GCN model outweighs the benefit. The GCN-based strategy also solves one more instance here compared to the other strategies.

4.5   Experimental Results for *SnR*

To the best of our knowledge, *SnR* is the current stat-of-the-art for exactly solving the vertex cover and independent set problems. In its default configuration, the SAT solver takes most of the execution time. With this setting, the solver looks at far fewer branches and, in extreme cases, times out while having checked virtually no branches. Since choosing these
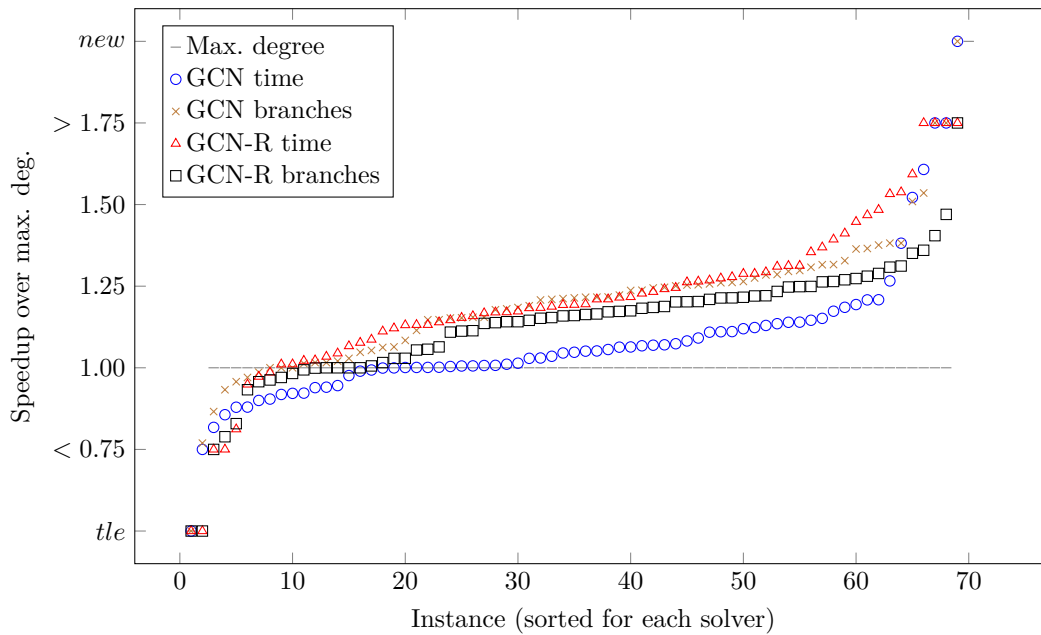
Figure 6 Cactus plot showing the speedup of our GCN-based branching in the *SnR* solver. The basis for computing speedup is the max. degree branching strategy. Instances are sorted by the speedup over max. degree branching, individually for execution time and the number of branches. Instances that timed out are ranked the lowest and marked with *tle*. Instances that were unsolved using one strategy but solved with the other are ranked the highest and marked with *new*. This figure shows results for the whole set of benchmark instances (excluding those solved by no configuration and those solved quickly or with very few branches), including the speedup comparing execution time and number of branches.

branches is the only way for our GCN-based approach to make a difference, it limits the potential impact of our GCN-based branching strategy. Therefore, we also evaluate the *SnR* solver with reduced time for the SAT solver, later called *GCN-R*. In this configuration, the SAT recourse limit is halved from its default value of 8192 down to 4096.

In a similar classification as the *BnR* solver, the 296 graphs used for testing are distributed as follows:

- Unsolved by any configuration: 94
- Solved with less than 10 branches or in less than 0.1 seconds: 133
- Remaining instances shown in Figure 6: 69

SnR is a far stronger solver than *BnR*. With the default branching strategy, it solves 52 more instances than *BnR* with our GCN-based branching. Furthermore, the number of instances solved before any branching takes place is also significantly higher, 133 compared to 25 for *BnR*, highlighting the strength of the pruning techniques used by *SnR*.

Figure 6 shows the results of our GCN-based branching in the *SnR* solver compared to the default max. degree branching. Our GCN-based branching also improves this solver, but to a lesser extent. With this solver, the number of solved instances is the same for both strategies. However, they both time out on one instance where the other finished within the time limit. Regarding aggregated numbers, the average speedup is 1.08, the geometric mean of speedup is 1.06, and the total speedup is 1.03. The ratio of improved instances stays roughly the same as in *BnR*, with 50 out of 69 graphs having a speedup with regard to the execution time. Among the graphs with a speedup, the median speedup is 6%.

When reducing the SAT resource limit, the speedups improve. The average speedup is now 1.22, the geometric mean of speedup is 1.19, and the total speedup is 1.12. The ratio of improved instances is now much higher, at 62 out of 69 instances. Among the graphs with a speedup, the median speedup is now 21%.

5   Conclusion and Future Work

We have introduced a GCN-based branching strategy and demonstrated its effectiveness on established benchmark instances. We trained and tested our technique in two different solvers. First, we used a branch-and-reduce solver that had previously been used to find better branching strategies than simply using the highest-degree vertex. Second, we used the current state-of-the-art solver for the vertex cover and independent set problems. Our GCN-based approach gave the best branching strategy on most test instances in both solvers. For the branch-and-reduce solver, we also solved 15 more test instances.

To the best of our knowledge, this work is also the first application of evolutionary-based training of graph neural networks. We combined specialized mutation operators for neural networks to make up our training procedure, and the models were trained on unused data from the 2019 PACE Challenge. This training paradigm could open up additional avenues for machine learning on combinatorial problems. Especially where supervised or reinforcement learning is difficult to use. This is because evolutionary-based training only requires a fitness function to train the model.

As is usually the case with machine learning, more data and longer training would likely push our results further. Given more time and effort, it is likely that the other training paradigms could also be made to work for this problem – especially reinforcement learning, given its success on similar problems. There are also more types of GNN architectures to try. We have only considered the GCN model in this work. Still, other options like GIN [55], k-GNNs [37], or GraphSAGE [17] would be natural continuations.

Finally, we have only considered the independent set problem and two solvers. There are other applicable problems where we could use the technique presented in this work. For example, there were many competitive solvers that used branch-and-bound type algorithms for the directed feedback vertex set problem in the 2022 PACE challenge.

───── **References** ─────

1   Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theoretical Computer Science*, 609:211–225, 2016. `doi:10.1016/j.tcs.2015.09.023`.

2   Maram Alsahafy and Lijun Chang. Computing maximum independent sets over large sparse graphs. In *International Conference on Web Information Systems Engineering*, pages 711–727. Springer, 2020.

3   Sergiy Butenko and Wilbert E. Wilhelm. Clique-detection models in computational biochemistry and genomics. *Eur. J. Oper. Res.*, 173(1):1–17, 2006. `doi:10.1016/j.ejor.2005.05.026`.

4   Randy Carraghan and Panos M. Pardalos. An exact algorithm for the maximum clique problem. *Oper. Res. Lett.*, 9(6):375–382, 1990. `doi:10.1016/0167-6377(90)90057-C`.

5   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. ACM, 2017. `doi:10.1145/3035918.3035939`.

6   Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40-42):3736–3756, 2010. `doi:10.1016/j.tcs.2010.06.026`.

7 Tammy M. K. Cheng, Yu-En Lu, Michele Vendruscolo, Pietro Liò, and Tom L. Blundell. Prediction by graph theoretic measures of structural effects in proteins arising from non-synonymous single nucleotide polymorphisms. *PLoS Computational Biology*, 4(7), 2008. `doi:10.1371/journal.pcbi.1000135`.

8 Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F Werneck. Accelerating local search for the maximum independent set problem. In *International symposium on experimental algorithms*, pages 118–133. Springer, 2016.

9 Camil Demetrescu, Andrew V Goldberg, and David S Johnson. *The shortest path problem: Ninth DIMACS implementation challenge*, volume 74. American Mathematical Soc., 2009.

10 Iddo Drori, Anant Kharkar, William R Sickinger, Brandon Kates, Qiang Ma, Suwen Ge, Eden Dolev, Brenda Dietrich, David P Williamson, and Madeleine Udell. Learning to solve combinatorial optimization problems on real-world graphs in linear time. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 19–24. IEEE, 2020.

11 M Ayaz Dzulfikar, Johannes K Fichte, and Markus Hecher. The pace 2019 parameterized algorithms and computational experiments challenge: the fourth iteration. In *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

12 M. Ayaz Dzulfikar, Johannes K. Fichte, and Markus Hecher. The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration (Invited Paper). In *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*, volume 148 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:23. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. `doi:10.4230/LIPIcs.IPEC.2019.25`.

13 Tarek Faycal and Claudio Zito. Direct mutation and crossover in genetic algorithms applied to reinforcement learning tasks. *arXiv preprint arXiv:2201.04815*, 2022.

14 Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *J. ACM*, 56(5):25:1–25:32, 2009. `doi:10.1145/1552285.1552286`.

15 M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the 6th ACM Symposium on Theory of Computing*, STOC '74, pages 47–63. ACM, 1974.

16 Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: `http://www.deeplearningbook.org`.

17 William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018. `arXiv:1706.02216`.

18 Demian Hespe, Sebastian Lamm, and Christian Schorr. Targeted branching for the maximum independent set problem. In *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*, volume 190 of *LIPIcs*, pages 17:1–17:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.SEA.2021.17`.

19 Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. WeGotYouCovered: The winning solver from the PACE 2019 challenge, vertex cover track. In *Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, pages 1–11. SIAM, 2020. `doi:10.1137/1.9781611976229.1`.

20 Demian Hespe, Christian Schulz, and Darren Strash. Scalable kernelization for maximum independent sets. *Journal of Experimental Algorithmics (JEA)*, 24(1):1–22, 2019.

21 David S Johnson. Cliques, coloring, and satisfiability: second dimacs implementation challenge. *DIMACS series in discrete mathematics and theoretical computer science*, 26:11–13, 1993.

22 Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

23 Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

24   Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed time-dependent contraction hierarchies. In *Experimental Algorithms, 9th International Symposium*, volume 6049, pages 83–93. Springer, 2010. `doi:10.1007/978-3-642-13193-6_8`.

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

26   Kenneth Langedal, Johannes Langguth, Fredrik Manne, and Daniel Thilo Schroeder. Efficient minimum weight vertex cover heuristics using graph neural networks. In *20th International Symposium on Experimental Algorithms (SEA 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

27   Henrique Lemos, Marcelo Prates, Pedro Avelar, and Luis Lamb. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 879–885. IEEE, 2019.

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

29   Chu-Min Li, Zhiwen Fang, and Ke Xu. Combining maxsat reasoning and incremental upper bound for the maximum clique problem. In *25th IEEE International Conference on Tools with Artificial Intelligence*, pages 939–946. IEEE Computer Society, 2013. `doi:10.1109/ICTAI.2013.143`.

30   Chu-Min Li, Hua Jiang, and Felip Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Comput. Oper. Res.*, 84:1–15, 2017. `doi:10.1016/j.cor.2017.02.017`.

31   Chu Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI Press, 2010. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1611`.

32   Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in neural information processing systems*, 31, 2018.

33   Melanie Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

34   Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 65(7):33–35, 2022.

35   Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):2383, 2018.

36   David J Montana, Lawrence Davis, et al. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

37   Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 4602–4609, 2019.

38   Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid Von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

39   Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.

40   Rick Plachetta and Alexander van der Grinten. SAT-and-reduce for vertex cover: Accelerating branch-and-reduce by SAT solving. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 169–180. SIAM, 2021. `doi:10.1137/1.9781611976472.13`.

41 Deepak Puthal, Surya Nepal, Cécile Paris, Rajiv Ranjan, and Jinjun Chen. Efficient algorithms for social network coverage and reach. In *IEEE International Congress on Big Data*, pages 467–474. IEEE Computer Society, 2015. `doi:10.1109/BigDataCongress.2015.75`.

42 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: `https://networkrepository.com`.

43 Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

44 Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. Graph.*, 27(5):144, 2008. `doi:10.1145/1409060.1409097`.

45 Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph geural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

46 Martin JA Schuetz, J Kyle Brubaker, and Helmut G Katzgraber. Combinatorial optimization with physics-inspired graph neural networks. *Nature Machine Intelligence*, 4(4):367–377, 2022.

47 Pablo San Segundo and Cristóbal Tapia. Relaxed approximate coloring in exact maximum clique search. *Comput. Oper. Res.*, 44:185–192, 2014. `doi:10.1016/j.cor.2013.10.018`.

48 David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

49 Darren Strash. On the power of simple reductions for the maximum independent set problem. In *Computing and Combinatorics - 22nd International Conference, COCOON 2016, Ho Chi Minh City, Vietnam, August 2-4, 2016, Proceedings*, volume 9797 of *Lecture Notes in Computer Science*, pages 345–356. Springer, 2016. `doi:10.1007/978-3-319-42634-1_28`.

50 Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

51 Etsuji Tomita, Yoichi Sutani, Takanori Higashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique with computational experiments. *IEICE Trans. Inf. Syst.*, 96-D(6):1286–1298, 2013. `doi:10.1587/transinf.E96.D.1286`.

52 Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. Reinforcement learning based query vertex ordering model for subgraph matching. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 245–258. IEEE, 2022.

53 Luzhi Wang, Chu-Min Li, Junping Zhou, Bo Jin, and Minghao Yin. An exact algorithm for minimum weight vertex cover problem in large graphs. *arXiv preprint arXiv:1903.05948*, 2019.

54 Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. *Inf. Comput.*, 255:126–146, 2017. `doi:10.1016/j.ic.2017.06.001`.

55 Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019. `arXiv:1810.00826`.

A   Results for *BnR*

Table 3 Detailed results for the *BnR* solver on the SPARSE instances. The s column shows speedup compared to the max. degree. The lowest execution time and number of branches appear in bold. Notice that one graph timed out with the max. degree and packing strategies. This instance receives the 30-minute running time and the same number of branches as the GCN-based strategy for the aggregated measures.

| | Execution Time | | | | | Number of Branches | | | | |
	Max degree	Packing	s	GCN	s	Max degree	Packing	s	GCN	s
as-Skitter	>1,800.00	>1,800.00	1.00	**1,619.45**	>1.11	>1,574,639	>1,574,639	1.00	**1,574,639**	>1.00
bay	**2.19**	3.68	0.60	3.05	0.72	**1,479**	2,539	0.58	2,572	0.58
fla	156.13	224.05	0.70	193.22	0.81	74,693	107,189	0.70	104,658	0.71
in-2004	5.68	5.68	1.00	**5.61**	1.01	769	754	1.02	**655**	1.17
libimseti	**1,185.60**	1,188.06	1.00	1,188.57	1.00	483	483	1.00	**482**	1.00
musae-twitch_FR	41.93	**41.54**	1.01	45.51	0.92	11,015	**10,858**	1.01	11,143	0.99
petster-friendships-dog	7.91	7.94	1.00	**7.91**	1.00	91	91	1.00	**90**	1.01
soc-LiveJournal1	11.26	11.26	1.00	**11.17**	1.01	134	129	1.04	**120**	1.12
web-BerkStan	212.77	**209.50**	1.02	229.68	0.93	54,590	53,450	1.02	**46,841**	1.17
web-Google	**2.43**	2.44	1.00	2.43	1.00	**47**	47	1.00	48	0.98
web-NotreDame	11.26	12.17	0.92	**4.86**	2.32	4,824	5,354	0.90	**1,565**	3.08
zhishi-baidu-relatedpages	**0.86**	0.87	0.99	1.35	0.64	**80**	80	1.00	80	1.00
zhishi-hudong-internallink	**1.37**	1.38	0.99	1.41	0.97	**12**	12	1.00	13	0.92
total	3,439.39	3,508.57	0.98	**3,314.22**	1.04	**1,722,856**	1,755,625	0.98	1,742,906	0.99
average			0.94		1.03			0.94		1.13
geom. mean			0.93		0.98			0.93		1.04

Table 4 Detailed results for the *BnR* solver on the DIMACS instances. The s column shows speedup compared to the max. degree. The lowest execution time and number of branches appear in bold.

| | Execution Time | | | | | Number of Branches | | | | |
	Max degree	Packing	s	GCN	s	Max degree	Packing	s	GCN	s
brock200_1	131.31	129.73	1.01	**129.23**	1.02	314,172	299,394	1.05	**284,536**	1.10
brock200_2	4.14	**4.14**	1.00	4.27	0.97	5,308	5,258	1.01	**5,245**	1.01
brock200_3	20.86	20.78	1.00	**18.29**	1.14	42,295	41,167	1.03	**31,335**	1.35
brock200_4	**25.53**	25.56	1.00	27.97	0.91	43,928	**43,323**	1.01	45,506	0.97
c-fat500-1	1.12	**1.12**	1.00	1.20	0.93	78	78	1.00	78	1.00
c-fat500-2	**0.56**	0.56	1.00	0.59	0.95	**38**	38	1.00	38	1.00
c-fat500-5	0.35	**0.35**	1.00	0.37	0.94	**14**	14	1.00	14	1.00
C125.9	0.98	0.93	1.00	**0.85**	1.15	3,424	3,237	1.06	**2,920**	1.17
DSJC500_5	**1,333.99**	1,334.34	1.00	1,357.59	0.98	1,018,216	1,014,808	1.00	**998,587**	1.02
gen200_p0.9_44	**145.70**	157.59	0.92	151.75	0.96	**315,541**	337,284	0.94	322,804	0.98
gen200_p0.9_55	126.96	104.25	1.22	**42.51**	2.99	298,937	231,273	1.29	**82,544**	3.62
hamming8-4	**17.83**	17.83	1.00	19.25	0.93	14,693	**14,686**	1.00	15,455	0.95
johnson16-2-4	45.03	**43.94**	1.02	59.95	0.75	308,173	**296,707**	1.04	449,724	0.69
keller4	**2.49**	2.50	1.00	2.55	0.98	4,021	**3,885**	1.04	4,074	0.99
MANN_a27	0.49	**0.49**	1.01	1.00	0.49	1,371	**1,358**	1.01	1,966	0.70
MANN_a45	74.64	**73.55**	1.01	261.54	0.29	123,729	**122,681**	1.01	228,736	0.54
p_hat1000-1	**827.20**	831.05	1.00	841.37	0.98	198,063	196,671	1.01	**192,302**	1.03
p_hat300-1	**3.87**	3.88	1.00	3.89	0.99	2,863	2,819	1.02	**2,802**	1.02
p_hat300-2	5.25	**5.23**	1.00	5.39	0.97	4,115	4,006	1.03	**3,834**	1.07
p_hat300-3	170.14	**164.26**	1.04	170.18	1.00	172,392	161,944	1.06	**160,874**	1.07
p_hat500-1	**39.42**	39.46	1.00	40.29	0.98	18,546	**18,507**	1.00	18,586	1.00
p_hat500-2	84.93	**84.34**	1.01	86.04	0.99	46,135	45,157	1.02	**42,972**	1.07
p_hat700-1	156.66	**156.32**	1.00	158.58	0.99	35,482	35,456	1.00	**35,041**	1.01
p_hat700-2	802.38	**786.44**	1.02	816.72	0.98	342,951	327,476	1.05	**322,804**	1.06
san1000	875.86	866.85	1.01	**802.24**	1.09	145,523	134,899	1.08	**99,207**	1.47
san200_0.7_1	9.76	9.83	0.99	**8.10**	1.20	14,004	13,740	1.02	**10,478**	1.34
san200_0.7_2	0.26	**0.26**	1.01	0.30	0.89	281	281	1.00	305	0.92
san200_0.9_1	12.04	**11.14**	1.08	12.15	0.99	17,277	**15,689**	1.10	16,085	1.07
san200_0.9_2	31.17	**15.37**	2.03	253.64	0.12	71,784	**28,644**	2.51	540,166	0.13
san200_0.9_3	1,068.29	1,042.05	1.03	**906.57**	1.18	2,866,949	2,646,993	1.08	**2,313,584**	1.24
san400_0.5_1	9.10	**9.07**	1.00	9.38	0.97	3,099	**2,993**	1.04	3,116	0.99
san400_0.7_1	896.18	901.66	0.99	**721.25**	1.24	544,733	542,205	1.00	**383,222**	1.42
sanr200_0.7	**44.34**	44.44	1.00	45.36	0.98	87,555	86,366	1.01	**83,157**	1.05
sanr200_0.9	646.97	583.20	1.11	**496.29**	1.30	1,734,428	1,510,574	1.15	**1,256,081**	1.38
sanr400_0.5	331.11	**330.89**	1.00	339.26	0.98	348,818	347,425	1.00	**343,145**	1.02
total	7,946.91	7,803.38	1.02	**7,795.90**	1.02	9,148,936	8,537,036	1.07	**8,301,323**	1.10
average			1.04		1.01			1.08		1.10
geom. mean			1.04		0.92			1.06		1.01

■ **Table 5** Detailed results for the *BnR* solver on the 2019 PACE instances. The s column shows speedup compared to the max. degree. The lowest execution time and number of branches appear in bold. Notice that some graphs timed out with the max. degree and packing strategies. These instances receives the 30-minute running time and the same number of branches as the GCN-based strategy for the aggregated measures.

	Execution Time					Number of Branches				
	Max degree	Packing	s	GCN	s	Max degree	Packing	s	GCN	s
vc-exact_005	1.93	1.87	1.04	**1.81**	1.07	4,942	4,689	1.05	**4,442**	1.11
vc-exact_006	0.89	0.85	1.04	**0.81**	1.10	2,732	2,635	1.04	**2,412**	1.13
vc-exact_010	2.22	2.18	1.02	**2.02**	1.10	6,157	6,010	1.02	**5,414**	1.14
vc-exact_019	3.07	2.95	1.04	**2.59**	1.19	7,032	6,636	1.06	**5,720**	1.23
vc-exact_031	77.97	71.89	1.08	**54.95**	1.42	222,904	201,316	1.11	**146,606**	1.52
vc-exact_033	0.99	1.03	0.96	**0.78**	1.26	3,081	3,192	0.97	**2,659**	1.16
vc-exact_035	7.64	7.35	1.04	**6.81**	1.12	23,591	22,275	1.06	**20,667**	1.14
vc-exact_036	**1.90**	2.32	0.82	14.26	0.13	**3,004**	3,678	0.82	26,646	0.11
vc-exact_037	10.43	9.85	1.06	**9.31**	1.12	30,311	27,905	1.09	**26,601**	1.14
vc-exact_038	12.10	11.99	1.01	**11.11**	1.09	10,838	10,641	1.02	**4,785**	2.26
vc-exact_039	93.29	100.86	0.92	**3.47**	26.85	241,090	257,121	0.94	**9,115**	26.45
vc-exact_041	48.44	45.67	1.06	**36.69**	1.32	130,627	120,246	1.09	**97,165**	1.34
vc-exact_042	36.59	33.69	1.09	**29.34**	1.25	89,789	81,377	1.10	**70,749**	1.27
vc-exact_043	181.42	173.34	1.05	**139.48**	1.30	485,761	450,065	1.08	**371,633**	1.31
vc-exact_044	91.88	85.57	1.07	**73.57**	1.25	247,265	224,352	1.10	**196,418**	1.26
vc-exact_045	24.55	23.24	1.06	**21.88**	1.12	61,960	57,378	1.08	**54,936**	1.13
vc-exact_046	112.91	104.91	1.08	**75.94**	1.49	294,775	267,548	1.10	**201,541**	1.46
vc-exact_047	57.05	52.27	1.09	**48.51**	1.18	151,632	135,866	1.12	**128,166**	1.18
vc-exact_048	25.16	22.99	1.09	**22.67**	1.11	66,630	**59,255**	1.12	59,263	1.12
vc-exact_049	18.03	**16.20**	1.11	16.54	1.09	49,831	**43,610**	1.14	45,608	1.09
vc-exact_050	50.43	47.12	1.07	**43.23**	1.17	139,040	126,211	1.10	**119,690**	1.16
vc-exact_051	55.43	51.48	1.08	**46.35**	1.20	139,558	127,253	1.10	**118,036**	1.18
vc-exact_052	23.18	**20.62**	1.12	20.77	1.12	69,106	**59,590**	1.16	59,743	1.16
vc-exact_053	59.36	**56.66**	1.05	57.34	1.04	166,485	**154,733**	1.08	164,098	1.01
vc-exact_054	32.32	30.78	1.05	**26.25**	1.23	92,747	85,916	1.08	**74,889**	1.24
vc-exact_055	7.00	**6.45**	1.09	6.69	1.05	19,686	**17,675**	1.11	18,366	1.07
vc-exact_056	95.92	86.94	1.10	**72.06**	1.33	241,359	212,937	1.13	**180,337**	1.34
vc-exact_057	64.98	61.75	1.05	**49.72**	1.31	175,755	162,101	1.08	**136,008**	1.29
vc-exact_058	48.02	45.18	1.06	**42.04**	1.14	131,639	120,494	1.09	**114,129**	1.15
vc-exact_059	13.90	13.52	1.03	**12.66**	1.10	40,884	38,320	1.07	**35,835**	1.14
vc-exact_060	77.39	73.33	1.06	**65.32**	1.18	198,644	183,543	1.08	**166,652**	1.19
vc-exact_061	22.38	21.54	1.04	**17.90**	1.25	66,667	63,137	1.06	**52,073**	1.28
vc-exact_062	67.69	63.65	1.06	**52.58**	1.29	183,366	167,497	1.09	**135,594**	1.35
vc-exact_063	68.42	63.48	1.08	**46.11**	1.48	185,107	168,191	1.10	**115,779**	1.60
vc-exact_064	28.98	26.81	1.08	**25.84**	1.12	79,071	71,632	1.10	**71,004**	1.11
vc-exact_065	36.82	32.88	1.12	**28.09**	1.31	88,525	77,805	1.14	**68,470**	1.29
vc-exact_066	8.41	8.04	1.05	**7.47**	1.13	26,832	25,102	1.07	**23,443**	1.14
vc-exact_067	120.54	110.98	1.09	**100.61**	1.20	281,249	251,171	1.12	**238,119**	1.18
vc-exact_068	8.81	**8.08**	1.09	8.25	1.07	22,323	**19,847**	1.12	20,791	1.07
vc-exact_069	43.52	41.35	1.05	**34.04**	1.28	119,349	110,681	1.08	**93,329**	1.28
vc-exact_070	11.97	11.62	1.03	**9.75**	1.23	33,671	31,962	1.05	**26,571**	1.27
vc-exact_071	34.85	31.81	1.10	**27.44**	1.27	81,516	72,726	1.12	**63,213**	1.29
vc-exact_072	44.41	42.70	1.04	**35.04**	1.27	110,122	103,759	1.06	**82,337**	1.34
vc-exact_073	42.53	39.16	1.09	**35.14**	1.21	117,587	106,432	1.10	**93,342**	1.26
vc-exact_074	7.46	6.98	1.07	**6.85**	1.09	22,981	20,823	1.10	**20,756**	1.11
vc-exact_075	>1,800.00	>1,800.00	1.00	**472.02**	>3.81	>729,491	>729,491	1.00	**729,491**	>1.00
vc-exact_077	13.90	13.49	1.03	**12.67**	1.10	40,884	38,320	1.07	**35,835**	1.14
vc-exact_081	134.31	125.76	1.07	**108.79**	1.23	350,329	317,725	1.10	**282,692**	1.24
vc-exact_082	64.11	61.22	1.05	**50.17**	1.28	196,274	182,337	1.08	**152,207**	1.29
vc-exact_083	180.28	165.94	1.09	**143.98**	1.25	503,993	448,958	1.12	**405,946**	1.24
vc-exact_088	>1,800.00	>1,800.00	1.00	**128.57**	>14.00	>298,310	>298,310	1.00	**298,310**	>1.00
vc-exact_091	237.21	221.19	1.07	**193.85**	1.22	704,763	631,255	1.12	**569,242**	1.24
vc-exact_093	106.14	100.44	1.06	**95.12**	1.12	302,210	276,323	1.09	**269,310**	1.12
vc-exact_099	>1,800.00	>1,800.00	1.00	**1.27**	>1,419.97	>2,198	>2,198	1.00	**2,198**	>1.00
vc-exact_100	>1,800.00	>1,800.00	1.00	**90.76**	>19.83	>222,987	>222,987	1.00	**222,987**	>1.00
vc-exact_101	>1,800.00	>1,800.00	1.00	**45.46**	>39.59	>107,949	>107,949	1.00	**107,949**	>1.00
vc-exact_102	>1,800.00	>1,800.00	1.00	**409.29**	>4.40	>757,151	>757,151	1.00	**757,151**	>1.00
vc-exact_104	>1,800.00	>1,800.00	1.00	**60.70**	>29.65	>93,769	>93,769	1.00	**93,769**	>1.00
vc-exact_122	>1,800.00	>1,800.00	1.00	**599.67**	>3.00	>975,834	>975,834	1.00	**975,834**	>1.00
vc-exact_123	>1,800.00	>1,800.00	1.00	**1,011.17**	>1.78	>1,603,466	>1,603,466	1.00	**1,603,466**	>1.00
vc-exact_125	>1,800.00	>1,800.00	1.00	**0.36**	>4,969.05	>240	>240	1.00	**240**	>1.00
vc-exact_134	>1,800.00	>1,800.00	1.00	**523.57**	>3.44	>834,495	>834,495	1.00	**834,495**	>1.00
vc-exact_144	>1,800.00	>1,800.00	1.00	**0.44**	>4,127.31	>444	>444	1.00	**444**	>1.00
vc-exact_149	>1,800.00	>1,800.00	1.00	**0.08**	>23,488.58	>55	>55	1.00	**55**	>1.00
vc-exact_186	>1,800.00	>1,800.00	1.00	**0.07**	>24,279.37	>63	>63	1.00	**63**	>1.00
total	27,819.17	27,661.96	1.01	**5,398.07**	5.15	12,692,126	12,086,703	1.05	**11,114,834**	1.14
average			1.04		899.84			1.06		1.56
geom. mean			1.04		3.06			1.06		1.19

B   Results for *SnR*

Table 6 Detailed results for the *SnR* solver on the DIMACS and sparse instances. The s column shows speedup compared to the max. degree. The lowest execution time and number of branches appear in bold. Notice that some graphs timed out. These instances receives the 30-minute running time and the same number of branches as the other strategy for the aggregated measures.

	Execution time					Number of Branches				
	Max degree	GCN	s	GCN-R	s	Max degree	GCN	s	GCN-R	s
brock200_2	**5.87**	6.25	0.94	6.19	0.95	**135**	141	0.96	141	0.96
brock200_3	45.25	44.75	1.01	**41.64**	1.09	127	**125**	1.02	**125**	1.02
brock200_4	**153.50**	153.62	1.00	408.02	0.38	127	**125**	1.02	161	0.79
C125.9	6.66	5.26	1.27	**4.60**	1.45	569	**283**	2.01	491	1.16
gen200_p0.9_55	9.13	4.99	1.83	**4.76**	1.92	**119**	**119**	1.00	**119**	1.00
johnson16-2-4	**3.51**	3.55	0.99	3.61	0.97	**111**	119	0.93	119	0.93
keller4	2.96	2.52	1.17	**2.00**	1.48	183	**172**	1.06	**172**	1.06
MANN_a27	3.67	2.28	1.61	**1.82**	2.01	952	**698**	1.36	700	1.36
MANN_a45	**240.48**	354.66	0.68	381.75	0.63	**59,764**	77,684	0.77	133,865	0.45
p_hat300-1	10.78	10.74	1.00	10.55	1.02	155	153	1.01	**147**	1.05
p_hat300-2	98.05	**97.88**	1.00	99.47	0.99	143	**139**	1.03	**139**	1.03
p_hat500-1	111.63	121.56	0.92	**110.44**	1.01	**147**	**147**	1.00	**147**	1.00
p_hat700-1	1,283.38	**1,176.11**	1.09	1,254.23	1.02	**149**	**149**	1.00	**149**	1.00
san1000	**1,033.95**	1,041.04	0.99	1,273.51	0.81	**583**	591	0.99	893	0.65
san200_0.7_2	3.12	3.32	0.94	**2.90**	1.08	155	**143**	1.08	161	0.96
san400_0.5_1	28.82	**27.84**	1.04	28.50	1.01	317	**301**	1.05	319	0.99
san400_0.7_1	573.00	569.68	1.01	**480.19**	1.19	131	135	0.97	135	0.97
sanr200_0.7	1,311.31	861.72	1.52	**745.57**	1.76	**155**	179	0.87	187	0.83
san400_0.7_2	>1,800.00	**1,514.90**	>1.19	>1,800.00	1.00	>**131**	131	>1.00	>**131**	1.00
musae-twitch_FR	760.18	678.89	1.12	**517.91**	1.47	63,655	**59,923**	1.06	60,233	1.06
total	7,485.23	**6,681.57**	1.12	7,177.68	1.04	**127,808**	141,457	0.90	198,534	0.64
average			1.12		1.16			1.06		0.96
geom. mean			1.09		1.09			1.04		0.94

■ **Table 7** Detailed results for the *SnR* solver on the 2019 PACE instances. The s column shows speedup compared to the max. degree. The lowest execution time and number of branches appear in bold. Notice that some graphs timed out. These instances receives the 30-minute running time and the same number of branches as the other strategy for the aggregated measures.

	Execution time					Number of Branches				
	Max degree	GCN	s	GCN-R	s	Max degree	GCN	s	GCN-R	s
vc-exact_005	14.24	10.31	1.38	**9.26**	1.54	21,373	**16,341**	1.31	16,337	1.31
vc-exact_006	4.67	4.66	1.00	**3.99**	1.17	10,321	**8,743**	1.18	8,897	1.16
vc-exact_010	11.26	9.43	1.19	**8.71**	1.29	24,805	**21,525**	1.15	21,791	1.14
vc-exact_019	12.81	12.13	1.06	**11.52**	1.11	25,573	**21,141**	1.21	**21,141**	1.21
vc-exact_031	262.48	236.28	1.11	**228.99**	1.15	751,333	**618,845**	1.21	624,627	1.20
vc-exact_035	47.16	46.91	1.01	**40.29**	1.17	103,415	**92,773**	1.11	92,965	1.11
vc-exact_037	65.20	57.22	1.14	**49.67**	1.31	139,951	**115,079**	1.22	118,101	1.19
vc-exact_038	536.75	504.55	1.06	**336.93**	1.59	47,564	**13,571**	3.50	15,224	3.12
vc-exact_041	283.35	249.62	1.14	**228.26**	1.24	560,171	**446,575**	1.25	448,417	1.25
vc-exact_042	209.96	200.84	1.05	**176.79**	1.19	412,043	**318,093**	1.30	323,581	1.27
vc-exact_043	682.69	**575.96**	1.19	608.90	1.12	1,987,935	**1,510,951**	1.32	1,706,531	1.16
vc-exact_044	830.47	747.88	1.11	**633.91**	1.31	1,169,595	**927,837**	1.26	958,159	1.22
vc-exact_045	141.62	130.87	1.08	**117.05**	1.21	223,855	**184,035**	1.22	184,411	1.21
vc-exact_046	479.52	420.71	1.14	**402.01**	1.19	1,385,345	**1,078,477**	1.28	1,095,625	1.26
vc-exact_047	432.25	403.83	1.07	**338.20**	1.28	700,511	**532,225**	1.32	554,569	1.26
vc-exact_048	172.49	187.00	0.92	**161.74**	1.07	277,717	**265,105**	1.05	269,873	1.03
vc-exact_049	143.05	158.20	0.90	**119.71**	1.19	265,311	**225,279**	1.18	231,625	1.15
vc-exact_050	289.93	240.01	1.21	**214.05**	1.35	524,847	**411,621**	1.28	420,361	1.25
vc-exact_051	368.18	351.44	1.05	**299.89**	1.23	605,749	**479,097**	1.26	490,917	1.23
vc-exact_052	112.94	131.90	0.86	**99.80**	1.13	222,787	**179,035**	1.24	187,557	1.19
vc-exact_053	367.94	367.61	1.00	**325.34**	1.13	742,883	**590,957**	1.26	608,933	1.22
vc-exact_054	203.35	181.03	1.12	**160.81**	1.26	396,235	**314,093**	1.26	317,659	1.25
vc-exact_055	42.68	52.20	0.82	**41.26**	1.03	77,763	**67,797**	1.15	69,833	1.11
vc-exact_056	484.93	481.31	1.01	**400.77**	1.21	605,493	**443,437**	1.37	461,611	1.31
vc-exact_057	533.33	564.06	0.95	**382.75**	1.39	369,193	**267,187**	1.38	312,279	1.18
vc-exact_058	296.89	296.55	1.00	**233.02**	1.27	389,189	**328,549**	1.18	350,827	1.11
vc-exact_059	94.05	89.48	1.05	**73.01**	1.29	153,767	**133,153**	1.15	134,747	1.14
vc-exact_060	441.34	441.46	1.00	**378.06**	1.17	750,693	**545,615**	1.38	555,759	1.35
vc-exact_061	106.38	103.28	1.03	**92.24**	1.15	237,715	**207,215**	1.15	209,261	1.14
vc-exact_062	339.35	377.05	0.90	**324.93**	1.04	580,635	**477,137**	1.22	495,085	1.17
vc-exact_063	364.76	354.49	1.03	**308.13**	1.18	539,377	**446,631**	1.21	539,917	1.00
vc-exact_064	203.36	190.20	1.07	**164.95**	1.23	379,593	**318,951**	1.19	323,193	1.17
vc-exact_065	214.95	201.38	1.07	**176.85**	1.22	330,061	**263,297**	1.25	274,535	1.20
vc-exact_066	65.16	56.59	1.15	**46.15**	1.41	129,915	**105,033**	1.24	106,829	1.22
vc-exact_067	746.68	695.39	1.07	**599.71**	1.25	718,307	**580,471**	1.24	622,477	1.15
vc-exact_068	83.35	**39.53**	2.11	43.90	1.90	74,777	**61,695**	1.21	76,097	0.98
vc-exact_069	244.19	213.16	1.15	**186.07**	1.31	431,177	**332,151**	1.30	339,629	1.27
vc-exact_070	74.72	66.14	1.13	**54.57**	1.37	141,555	**122,765**	1.15	122,913	1.15
vc-exact_071	199.15	164.80	1.21	**129.95**	1.53	297,773	**193,943**	1.54	202,583	1.47
vc-exact_072	307.34	349.41	0.88	**262.03**	1.17	383,985	**289,101**	1.33	299,845	1.28
vc-exact_073	287.55	285.56	1.01	**254.10**	1.13	526,765	**421,175**	1.25	433,831	1.21
vc-exact_074	48.24	47.57	1.01	**38.21**	1.26	85,133	**83,435**	1.02	84,653	1.01
vc-exact_077	94.11	89.49	1.05	**73.04**	1.29	153,767	**133,153**	1.15	134,747	1.14
vc-exact_081	910.11	855.98	1.06	**717.73**	1.27	1,129,433	**817,385**	1.38	876,193	1.29
vc-exact_082	356.12	321.09	1.11	**292.62**	1.22	868,967	**710,671**	1.22	722,971	1.20
vc-exact_083	1,114.70	1,267.89	0.88	**962.21**	1.16	1,692,351	**1,121,237**	1.51	1,204,803	1.40
vc-exact_091	1,156.70	1,184.84	0.98	**977.77**	1.18	1,959,583	**1,523,743**	1.29	1,684,477	1.16
vc-exact_093	684.89	743.10	0.92	**600.95**	1.14	1,035,161	**830,441**	1.25	883,663	1.17
vc-exact_200	**1,384.23**	>1,800.00	<0.77	>1,800.00	<0.77	**163**	>163	<1.00	>163	<1.00
total	16,551.54	16,560.38	1.00	**14,190.79**	1.17	24,641,610	**19,196,929**	1.28	20,240,222	1.22
average			1.07		1.24			1.29		1.23
geom. mean			1.05		1.23			1.26		1.21