Using Statistical Measures and Machine Learning for Graph Reduction to Solve Maximum Weight Clique Problems

Yuan Sun, Xiaodong Li, and Andreas Ernst

Abstract—In this paper, we investigate problem reduction techniques using stochastic sampling and machine learning to tackle large-scale optimization problems. These techniques heuristically remove decision variables from the problem instance, that are not expected to be part of an optimal solution. First we investigate the use of statistical measures computed from stochastic sampling of feasible solutions compared with features computed directly from the instance data. Two measures are particularly useful for this: 1) a ranking-based measure, favoring decision variables that frequently appear in high-quality solutions; and 2) a correlation-based measure, favoring decision variables that are highly correlated with the objective values. To take this further we develop a machine learning approach, called Machine Learning for Problem Reduction (MLPR), that trains a supervised learning model on easy problem instances for which the optimal solution is known. This gives us a combination of features enabling us to better predict the decision variables that belong to the optimal solution for a given hard problem. We evaluate our approaches using a typical optimization problem on graphs – the maximum weight clique problem. The experimental results show our problem reduction techniques are very effective and can be used to boost the performance of existing solution methods.

Index Terms—Combinatorial optimization, machine learning, data mining, statistics, problem reduction.

1 Introduction

Large-scale combinatorial optimization problems are ubiquitous in the real world, e.g., open pit mining [1, 2], scheduling medical resident training [3] and social network analysis [4]. These problems are challenging to solve, partially due to the large search space and NP-hardness. Exact solvers in many cases cannot handle the large problem size; and heuristic methods may easily get stuck in a local optimum, resulting in a poor objective value.

A logical way to tackle large-scale optimization problems is using problem reduction; that is to reduce the size of an original problem by removing decision variables and/or constraints that are irrelevant to the optimal solution. Indeed, the optimal solution to many combinatorial optimization problems is determined by a relatively small number of decision variables. For example the maximum clique of a large graph typically consists of a small proportion of vertices [5]. The other vertices are redundant variables, that mainly slow down the optimization process. By removing some of these redundant variables, the original large search space can be significantly reduced to a size that is manageable by existing solution methods.

The existing problem reduction techniques for combinatorial optimization can be roughly classified into two categories: *exact* and *greedy* methods. An exact method removes decision variables that can not be part of the optimal solution based on analytical reasoning (e.g., a definition of objective bound) [6–10]. A greedy method, on the other hand, removes decision variables that are unlikely to be part of the optimal solution based on a measure of fitness or quality [10–13]. An exact method can guarantee that the reduced problem always captures the original optimal solution; but in many cases it can not effectively reduce the problem size. Thus, we will focus on the greedy approach in this paper.

Machine learning techniques have been successfully used to boost the performance of branch-and-bound (B&B) algorithms [14–17] and heuristic methods [18–20]; and have also been used to automatically design solution algorithms [21–25]. However, despite of its popularity machine learning has not been extensively applied for problem reduction. Although a high-level idea of applying data mining and machine learning to problem reduction has been described in [12], no detailed method description or experimental evaluation is given. More recently a machine learning model has been trained to estimate the probability of a decision variable being a part of an optimal solution [25], however this estimated probability has not been explicitly used to reduce problem size as a preprocessing step.

In this paper we propose to use statistical measures and machine learning for problem reduction, by greedily removing decision variables from a problem instance that are unlikely to be part of the optimal solution. First we use statistical measures computed from stochastic sampling of feasible solutions to evaluate the "quality" of each decision variable. Particularly we describe two measures for this: 1) a ranking-based measure, favoring decision variables that frequently appear in high-quality solutions; and 2) a correlation-based measure, favoring decision variables that are highly correlated with the objective values. We then use

- *Y. Sun and X. Li are with School of Science, RMIT University, Melbourne, 3001, Victoria, Australia. E-mail: yuan.sun@rmit.edu.au; xiaodong.li@rmit.edu.au*
- *A. Ernst is with School of Mathematical Sciences, Monash University, Clayton, 3800, Victoria, Australia. E-mail: andreas.ernst@monash.edu*

Manuscript received May 3, 2019; revised September 30, 2019.

these measures to guide the problem reduction process.

To take this further we develop a machine learning approach for problem reduction that we call Machine Learning for Problem Reduction (MLPR). We model problem reduction as a binary classification problem and use an off-the-shelf supervised learning algorithm to train the model on easy problem instances for which the optimal solution is known. This gives us a combination of features enabling us to better predict the decision variables that belong to the optimal solution for a given hard problem.

We evaluate our approaches using the Maximum Weight Clique (MWC) problem (see Section 2.2 for more details). The experimental results on benchmark and real-world datasets show that our proposed methods are effective and outperform the problem reduction methods directly computed from graph data. We test four existing solution methods and show they can benefit greatly from using our graph reduction methods as a preprocessing step. As a by-product of our methods we can generate a decision variable ordering that can be used to significantly improve the performance of B&B algorithms. Overall the MLPR approach is generally more robust than using a single statistical measure, but this method is more complex to implement as it requires initial training on a set of optimally solved instances.

The remainder of this paper is organized as follows. In Section 2, we describe the background and related work. In Section 3 and 4, we propose the statistical measures and MLPR receptively. In Section 5, we describe experimental methodology and present results. The last section concludes the paper and suggests future research directions.

2 Related Work

In this section, we briefly review the existing learning-based algorithms for combinatorial optimization, and solution methods for the MWC problem.

2.1 Learning-based Algorithms for Combinatorial Optimization

There is an increasing interest in using machine learning techniques for solving combinatorial optimization problems [26]. We briefly review these methods in three categories: 1) improving B&B algorithms; 2) improving heuristics and 3) designing heuristics automatically (hyper-heuristics).

Firstly, improving the performance of B&B algorithms via machine learning is a very active area of research recently. For example He *et al.* [15] used imitation learning to learn a branching variable selection policy; Khalil *et al.* [16] built a surrogate model to mimic the strong branching strategy; Di Liberto *et al.* [27] used a clustering method to determine the best time to switch the branching variable selection heuristic; Khalil *et al.* [28] used logistic regression to determine when to run a given heuristic; and Balcan *et al.* [17] trained a machine learning model to learn an optimal weighting of partitioning procedures to reduce the tree size. Here we have only briefly mentioned some representative examples. Interested readers are referred to [14] for a more comprehensive literature review.

Secondly, machine learning has also been used to enhance the search ability of heuristic methods. Boyan and Moore [18] built a machine learning model to evaluate a local search method to generate a smart restart rule. Shylo and Shams [19] trained a logistic regression model using samples generated by Tabu search to predict components of an optimal solution, which are in turn incorporated into Tabu search to boost its search ability. Martins *et al.* [20] extracted components that frequently appear in high-quality solutions generated by a hybrid heuristic, and used the extracted components to guide the construction of new solutions for a local search method to refine.

Thirdly, machine learning can also be used to automatically design heuristics, under the umbrella of hyper-heuristics [29, 30]. For example Zhang and Dietterich [21] and Khalil *et al.* [23] used reinforcement learning to learn a greedy policy that aims to find high-quality solution quickly. Vinyals *et al.* [22] designed a new neural architecture called pointer network that can be trained to solve combinatorial optimization problems. Fischetti and Fraccaro [31] built a supervised learning model to predict the value of optimal solutions for the offshore wind farm layout optimization problem. Burke *et al.* [32] and Nguyen *et al.* [33] used genetic programming to automatically design heuristics for packing and job shop scheduling problems. Recently Li *et al.* [25] trained a deep graph convolutional network to estimate the probability of a decision variable belonging to an optimal solution. The predicted probability maps over decision variables are then used to construct a number of high-quality solutions, leveraged by a tree search procedure. Some of these studies, especially [25], have *implicitly* used the problem reduction idea to reduce the probability of exploring the variables that are unlikely to be part of an optimal solution. In contrast, we will propose an *explicit* problem reduction method that removes these redundant variables from a problem a priori.

It is worth noting that there are other methods that do not fit into the three categories described above, e.g., learning combinatorial problem models [34], and selecting the best algorithm for solving a given problem [35, 36]. However, we will not review these methods here due to the page limit. Interested readers are directed to [26].

Despite the popularity and success of applying machine learning to combinatorial optimization, machine learning has not been intensively studied to reduce problem size as a preprocessing step. Although there have been some other methods that use the idea of problem reduction, e.g., merge search [37], the construct, merge, solve and adapt (CMSA) method [38], pruning heuristic [11], sketching method [39, 40], ant colony optimization [41] and estimation of distribution algorithm [42, 43], they are typically designed within the context of a specific algorithm or for a particular type of problems. Unlike these we will propose a generic approach based on machine learning that can be adopted as a preprocessing step for any existing solution method and is potentially applicable to a wide range of problems.

2.2 Solution Methods for Maximum Weight Clique Problem

In an undirected graph $G(V, E, W)$, where V denotes the set of vertices, E denotes the set of edges, and W denotes the weight of vertices, a clique C is a subset of V in which each

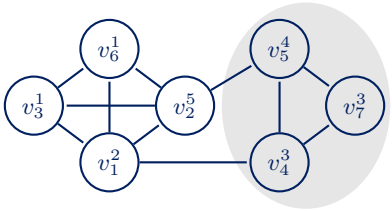


Fig. 1: An illustration of the MWC problem. The subscript of each vertex denotes the vertex index and the superscript denotes the vertex weight. The MWC of the given graph is $\{v_4^3, v_5^4, v_7^3\}$ with total weight of 10.

pair of vertices are adjacent. The MWC, as its name suggests, is a clique with the largest total weight of its vertices (see Fig. 1 for an example). Let $\bar{G}(V, \bar{E}, W)$ denote the complementary graph of $G(V, E, W)$, where the edge set $\bar{E}$ contains all the edges that are not in E: $\bar{E} = \{(i,j) \,|\, \forall (i,j) \notin E\}$. We use a binary string $\boldsymbol{x}$ to represent a clique; $x_i = 1$ means vertex v_i is in the clique; otherwise it is not. Formally the MWC problem can be defined as

$$\max_{\boldsymbol{x}} \sum_{i=1}^{|V|} w_i x_i, \tag{1}$$

$$s.t.\ \ x_i + x_j \leq 1,\ \forall (i,j) \in \bar{E}, \tag{2}$$

$$x_i \in \{0,1\},\ i = 1, 2, \cdots, |V|. \tag{3}$$

This problem has a wide range of applications, e.g., protein structure prediction [44], computer vision [45, 46] and genomics [47].

Searching for a MWC in a given graph is NP-hard. A large number of algorithms have been proposed to tackle this problem including *exact solvers* and *heuristic methods* [5]. The representatives of exact solvers include MWC based on MaxSAT reasoning [48], weighted large maximum clique [7], MWC based on two-stage MaxSAT reasoning [8]; MWC based on weight cover [49]; and conflict directed clause learning [50]. These algorithms differ mainly in the upper bound calculation and branching strategy used, and typically have difficulty in scaling up to large dense graphs.

On the other hand the heuristic methods for this problem include phased local search [51]; multi-neighbourhood Tabu search [52], fast weight clique [53], local search configuration checking with best from multiple selection [54], restart Tabu search based on a "push" operator [55], and restart and random walk based local search [56]. These methods can find a good solution quickly for small and medium-sized graphs, however unable to solve instances in very large graphs that have many local optima. Recently, a CPU-GPU local search method based on new neighborhood structures has been proposed for solving large instances [57]. In this paper we tackle large-scale graph problems from a different perspective; that is to reduce the problems to a size that is manageable by existing solution methods.

3 Statistical Measures for Problem Reduction

In this section, we first describe the random sampling method used to generate feasible solutions. We then describe the two proposed statistical measures in detail.

Algorithm 1 RANDOM_SAMPLING(V, W, B, n)

Require: vertex set V; weight W, neighbors B; number of cliques to generate n;
for i in 1 to n **do**
  Initialize the i_{th} clique $C_i \leftarrow \emptyset$;
  Initialize the objective value $y_i \leftarrow 0$;
  Initialize vertex set $V_c \leftarrow \{1, 2, \cdots, |V|\}$;
  while V_c is not $\emptyset$ **do**
    Randomly select a vertex v_s from V_c;
    Add v_s to the clique C_i;
    Accumulate the weight: $y_i \leftarrow y_i + w_s$;
    if $|C_i|$ is equal to 1 **then**
      $V_c \leftarrow B_s$ in ascending order;
    else
      $V_c \leftarrow$ INTERSECTION(V_c, B_s);
return $\mathbb{C} = \{C_1, \cdots, C_n\}$, $Y = \{y_1, \cdots, y_n\}$.

Algorithm 2 INTERSECTION(V_c, B_s)

Sort the vertices in B_s in ascending order;
Initialize $V_i \leftarrow \emptyset$; $k_1 \leftarrow 1$; $k_2 \leftarrow 1$;
while $k_1 \leq |V_c|$ and $k_2 \leq |B_s|$ **do**
  if $V_c[k_1] < B_s[k_2]$ **then**
    $k_1 \leftarrow k_1 + 1$;
  else if $V_c[k_1] > B_s[k_2]$ **then**
    $k_2 \leftarrow k_2 + 1$;
  else
    Add $V_c[k_1]$ to V_i;
    $k_1 \leftarrow k_1 + 1$; $k_2 \leftarrow k_2 + 1$;
return V_i.

3.1 Random Sampling Method

The high-level idea of the random sampling method is to construct a solution by incrementally adding a randomly selected decision variable. Here we use the MWC problem as an example; thus a feasible solution refers to a clique and a decision variable refers to a vertex. The detailed procedure of the random sampling method (Algorithm 1) that we use to generate a clique is

1) Initialize a clique C as empty; the objective value y as 0; and a candidate vertex set V_c as V;
2) Randomly select a vertex v_s from V_c; add v_s to C and accumulate the vertex weight w_s;
3) Update the candidate set V_c as the intersection of v_s's neighbors (B_s) and V_c;
4) Repeat Step 2) to 3) until V_c is empty, and a clique C is generated.

This process can be repeated multiple times (n) in order to generate multiple solutions. It is worth noting that while this sampling method is not explicitly greedy, it is biased towards generating larger cliques. That is because vertices from large cliques are more likely to be selected at each iteration of the algorithm, as there are more of them. Assuming that vertex weights are uncorrelated with the size of the cliques that they are part of, this leads to a bias towards higher weight cliques. Also only maximal cliques are generated in this manner.

In Step 3) of the sampling method, the vertices that are not adjacent to the newly added vertex (v_s) should be removed from the candidate set V_c, in the sense they cannot form a clique with v_s. This removing step (or equivalently computing the intersection between V_c and v_s's neighbors

B_s) is crucial, as it determines the time complexity of the sampling process. A naïve approach that performs pairwise comparison between the vertices in V_c and B_s costs $\Theta(|V_c||B_s|)$. In Algorithm 2 we describe a more efficient approach for computing the intersection, that only requires a linear scan through V_c and B_s. To achieve this we need to sort the vertices in B_s by their indices in ascending order. To avoid any redundant sorting, we can keep track of whether the neighbors of a vertex have been sorted before. Note that the vertices in V_c are already in sorted order.

We loop through V_c and B_s using two auxiliary variables k_1 and k_2 to denote the current working index of V_c and B_s, both initialized to 1. We compare the $k_{1\text{th}}$ vertex in V_c ($V_c[k_1]$) with the $k_{2\text{th}}$ vertex in B_s ($B_s[k_2]$). If $V_c[k_1] < B_s[k_2]$ we increment k_1 by 1, in the sense that $V_c[k_1]$ can not be in B_s, as $V_c[k_1] < B_s[k_2] < B_s[k]$ for any $k = k_2 + 1, \cdots, |B_s|$. For a similar reason we increment k_2 by 1 if $V_c[k_1] > B_s[k_2]$. If $V_c[k_1] = B_s[k_2]$ we find a common vertex and add this vertex to the new candidate set V_i, initialized as empty; we then increment both k_1 and k_2 by 1. This process is continued until all the vertices in V_c or B_s have been checked. The new candidate vertex set V_i is returned, with the vertices in V_i already in sorted order. Note that initially when V_c is equal to V, the intersection between V_c and B_s is simply B_s.

Lemma 1. *The time complexity of generating n cliques using the random sampling method (Algorithm 1) is $\mathcal{O}\big(n|E| + |E|\log(|V|)\big)$, where $|V|$ and $|E|$ are the number of vertices and edges in a given graph.*

Proof. First, we assume the neighbors of each vertex are sorted and show the time complexity of generating a clique C is in $\mathcal{O}(|E|)$. We will use the fact that the number of comparisons required to identify the intersection between V_c and B_s is in $\mathcal{O}(|V_c|+|B_s|)$. Let $\{v_{s_1}, v_{s_2}, \cdots, v_{s_{|C|}}\}$ denote the sequence of vertices added into C; B_{s_i} denote the neighbor set of v_{s_i}; and V_{c_i} denote the candidate vertex set before v_{s_i} is added into C, where $1 \leq i \leq |C|$. The total number of comparisons performed to compute the intersection is in the order of $\sum_{i=2}^{|C|}\big(|V_{c_i}| + |B_{s_i}|\big)$, as adding the first vertex into C takes $\mathcal{O}(1)$ time. It is true that $|V_{c_i}| \leq B_{s_{i-1}}$, for any $i = 2, \cdots, |C|$. Thus, $\sum_{i=2}^{|C|}\big(|V_{c_i}| + |B_{s_i}|\big) \leq 2\sum_{i=1}^{|C|}|B_{s_i}| \leq 2|E| \in \mathcal{O}(|E|)$. Note that the relaxation used here is very conservative, in the sense that $\sum_{i=1}^{|C|}|B_{s_i}|$ is usually much less than $|E|$ especially in a sparse graph.

Second, we consider the case where the vertex neighbors (B) have not been sorted. In the worst case, the neighbors of each vertex need to be sorted once. If using MergerSort, the number of comparisons used by sorting is in the order of $\sum_{i=1}^{|V|}|B_i|\log(|B_i|) < \sum_{i=1}^{|V|}|B_i|\log(|V|) = |E|\log(|V|)$. Thus the time complexity of sorting is $\mathcal{O}(|E|\log(|V|))$. [1]

In short, the total time complexity of generating n cliques is in $\mathcal{O}\big((n|E| + |E|\log(|V|)\big)$. Note that in our experiments $n >> \log(|V|)$, thus the time used in sorting is negligible. □

1. In our experiments, we will use QuickSort due to its simplicity, in-place property, and average time efficiency. Although the worst case time complexity of QuickSort is quadratic, it rarely happens in practice.

Algorithm 3 RANKING-BASED MEASURE($\mathbb{C}$, Y, n)

```
Sort the cliques in ℂ based on objective value Y; use r_i to denote the ranking of i_th clique C_i;
Initialize f_r(v_j) ← 0, for each v_j ∈ V;
for i from 1 to n do
    for j from 1 to |C_i| do
        v ← C_i[j];
        f_r(v) ← f_r(v) + 1/r_i;
return f_r.
```

3.2 Statistical Measures

Based on the generated sample solutions, we design two statistical measures to quantify the "quality" of each decision variable (i.e., vertex). Our measures are motivated by the observation that many optimization problems have a "backbone" structure [5]. In other words, high-quality solutions potentially share some components with the optimal solution. Our goal is to extract the shared components from high-quality solutions, and reduce the original search space such that it is manageable by existing algorithms.

3.2.1 Ranking-Based Measure

Let C_i denote the i_{th} generated clique, and y_i denote its objective value, where $i = 1, \cdots, n$. We sort the cliques based on their objective values in descending order, and use r_i to denote the ranking of C_i; smaller ranking indicates better solution quality. Let $\boldsymbol{x}_i$ denote the binary string representing C_i, where $x_{i,j} = 1$ if v_j is in C_i and $x_{i,j} = 0$ otherwise. The ranking-based measure for each vertex is defined as

$$f_r(v_j) = \sum_{i=1}^{n} \frac{x_{i,j}}{r_i}, \tag{4}$$

where $1 \leq j \leq |V|$. It basically accumulates $1/r_i$ across the cliques that include v_j. Two factors contribute to the accumulation: 1) the frequency that v_j appears in the cliques; and 2) the ranking of cliques that v_j is part of. Vertices with a large accumulated score are regarded as high-quality and are likely to be part of the optimal solution. By removing the vertices for which the score is less than a threshold ($f_r(v_j) < \epsilon_r$), the graph size can be significantly reduced.

In practice, it is not efficient to represent a clique using a binary string in terms of time and space complexity. Instead we can represent a clique by a set of vertices it includes, i.e., using set representation C. The time and space complexity for accumulating the ranking-based score can then be reduced from $\mathcal{O}(n|V|)$ to $\mathcal{O}(\sum_{i=1}^{n}|C_i|)$, that makes a significant difference for large sparse graphs. In Algorithm 3, we show how to calculate the ranking-based measure using the set representation C. Basically, we iterate through each vertex in each clique, and accumulate the measure $f_r(v_j)$ by $1/r_i$ if vertex v_j is in clique C_i.

3.2.2 Correlation-Based Measure

As before, we use C_i and y_i to denote the i_{th} generated clique and its objective value, where $i = 1, \cdots, n$. We use a binary string $\boldsymbol{x}_i$ to represent C_i, where $x_{i,j} = 1$ if v_j is in C_i and $x_{i,j} = 0$ otherwise. The correlation-based measure

calculates the Pearson correlation coefficient between each vertex and the objective value across the generated cliques:

$$f_c(v_j) = \frac{\sum_{i=1}^{n}(x_{i,j} - \bar{x}_j)(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_{i,j} - \bar{x}_j)^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}, \quad (5)$$

where $\bar{x}_j = \sum_{i=1}^{n} x_{i,j}/n$, and $\bar{y} = \sum_{i=1}^{n} y_i/n$. As our objective is maximization, vertices that are highly positively correlated with the objective values are likely to be in the optimal solution. On the other hand the vertices for which the correlation score is less than a given threshold ($f_c(v_j) < \epsilon_c$) may be removed from a graph, without sacrificing much of the solution quality.

Calculating our correlation-based measure directly using binary string representation costs $\mathcal{O}(n|V|)$. To improve the time efficiency, we simplify the calculation of Peason correlation coefficient as follows.

Lemma 2. *For binary variable $x_{i,j}$, the following equality holds:*

$$\sum_{i=1}^{n}(x_{i,j} - \bar{x}_j)^2 = \bar{x}_j(1 - \bar{x}_j)n, \quad (6)$$

where $\bar{x}_j = \sum_{i=1}^{n} x_{i,j}/n$.

Proof. For simplicity, we denote $\sum_{i=1}^{n}(x_{i,j} - \bar{x}_j)^2$ as σ_{x_j}. As $x_{i,j}$ is binary variable,

$$\sigma_{x_j} = n_1(1 - \bar{x}_j)^2 + n_0(0 - \bar{x}_j)^2, \quad (7)$$

where n_1 and n_0 count the number of 1s and 0s in binary variables $x_{1,j}, x_{2,j}, \cdots, x_{n,j}$; that is $n_1 = \sum_{i=1}^{n} x_{i,j} = n\bar{x}_j$ and $n_0 = n - n_1$. Thus,

$$\sigma_{x_j} = n\bar{x}_j(1-\bar{x}_j)^2 + (n - n\bar{x}_j)(0-\bar{x}_j)^2 = \bar{x}_j(1-\bar{x}_j)n. \quad (8)$$

□

Lemma 3. *For binary variable $x_{i,j}$ and variable y_i, the following equality holds:*

$$\sum_{i=1}^{n}(x_{i,j} - \bar{x}_j)(y_i - \bar{y}) = (1 - \bar{x}_j)s_{j,1} - \bar{x}_j s_{j,0}, \quad (9)$$

where $\bar{x}_j = \sum_{i=1}^{n} x_{i,j}/n$, $\bar{y} = \sum_{i=1}^{n} y_i/n$; and

$$s_{j,1} = \sum_{\substack{1 \le i \le n \\ x_{i,j}=1}} (y_i - \bar{y}),\ s_{j,0} = \sum_{\substack{1 \le i \le n \\ x_{i,j}=0}} (y_i - \bar{y}). \quad (10)$$

Proof. For simplicity, we use σ_{c_j} to denote $\sum_{i=1}^{n}(x_{i,j} - \bar{x}_j)(y_i - \bar{y})$. Using the fact that $x_{i,j}$ is a binary variable,

$$\begin{aligned}
\sigma_{c_j} &= \sum_{\substack{i=1 \\ x_{i,j}=1}}^{n} (1 - \bar{x}_j)(y_i - \bar{y}) + \sum_{\substack{i=1 \\ x_{i,j}=0}}^{n} (0 - \bar{x}_j)(y_i - \bar{y}), \\
&= (1 - \bar{x}_j) \sum_{\substack{i=1 \\ x_{i,j}=1}}^{n} (y_i - \bar{y}) + (0 - \bar{x}_j) \sum_{\substack{i=1 \\ x_{i,j}=0}}^{n} (y_i - \bar{y}), \\
&= (1 - \bar{x}_j)s_{j,1} - \bar{x}_j s_{j,0}.
\end{aligned} \quad (11)$$

□

Having these simplifications, we present in Algorithm 4 a method to calculate the correlation-based measure using set representation. The key step is to iterate through each vertex in each clique to calculate $\bar{x}_j$ and $s_{j,1}$ (line 6 to 10). Then σ_{x_j}, σ_{c_j} and thus our correlation-based measure can be easily computed. The total time complexity is $\mathcal{O}(\sum_{i=1}^{n}|C_i|)$.

Algorithm 4 CORRELATION-BASED MEASURE($\mathbb{C}$, Y, n)

1: Calculate the mean objective value: $\bar{y} \leftarrow \sum_{i=1}^{n} y_i/n$;
2: Calculate the objective difference: $y_d \leftarrow \sum_{i=1}^{n}(y_i - \bar{y})$;
3: Calculate the objective "variance": $\sigma_y \leftarrow \sum_{i=1}^{n}(y_i - \bar{y})^2$;
4: Initialize the mean $\bar{x}_j \leftarrow 0$, for each $v_j \in V$;
5: Initialize $s_{j,1} \leftarrow 0$ for each $v_j \in V$;
6: **for** i from 1 to n **do**
7: | **for** k from 1 to $|C_i|$ **do**
8: | | $j \leftarrow$ the index of vertex $C_i[k]$;
9: | | $\bar{x}_j \leftarrow \bar{x}_j + 1/n$;
10: | | $s_{j,1} = s_{j,1} + (y_i - \bar{y})$;
11: **for** j from 1 to $|V|$ **do**
12: | $\sigma_{c_j} = (1 - \bar{x}_j)s_{j,1} - \bar{x}_j(y_d - s_{j,1})$;
13: | $\sigma_{x_j} = \bar{x}_j(1 - \bar{x}_j)n$;
14: | $f_c(v_j) = \sigma_{c_j}/\sqrt{\sigma_{x_j}\sigma_y}$.
15: **return** f_c.

4 MACHINE LEARNING FOR PROBLEM REDUCTION

In this section, we describe our machine learning approach for problem reduction MLPR. We model problem reduction as a typical classification problem, and use an off-the-shelf machine learning algorithm to train the model, taking our statistical measures as features.

4.1 Modelling

We use easy graphs that we know the optimal solution (i.e., MWC) as the training dataset. We treat each vertex in a graph as a training instance, and assign a class label 1 to vertices that belong to the optimal solution and -1 to those who do not. We use our proposed statistical measures as features combined with those directly computed from graph data (see Section 4.2 for details). This becomes a typical binary classification problem. We will use support vector machine (SVM) [58, 59] to train the model though any supervised learning algorithm fits here. For a given hard graph where we do not know the optimal solution, the trained model can be used to predict a class label for vertices in this graph. The vertices with a predicted label -1 will be removed from the graph, so that the graph size can be significantly reduced. The main steps of our MLPR method are summarized as follows:

1) Solve the MWC problem to optimality for selected easy graphs using an exact solution method.
2) Extract features and assign a class label for each vertex in the easy graphs and construct a training dataset.
3) Train a classification model using a machine learning algorithm.
4) Predict a class label for each vertex in a given hard graph using the trained model, and remove vertices from the graph that are predicted to be -1.

We can then use an existing solution method to solve the MWC problem in the reduced graph.

4.2 Feature Extraction

Apart from the two statistical measures described in Section 3, we compute four other features directly from graph data to characterize a vertex (training instance). These six

features will be used as inputs to the machine learning model:

1) Vertex weight: $f_w(v_i) = w_i$. The vertex weight is an important feature as our optimization goal is to find a clique with maximum weights of its vertices.
2) Vertex degree: $f_d(v_i) = |B_i|$, where $B_i := \{\forall\, v_j \,|\, v_j \in V \wedge (v_i, v_j) \in E\}$. The degree of a vertex is the number of its neighbours. Vertices with a high degree are more likely to form a large clique.
3) Upper bound: $f_b(v_i) = w_i + \sum_{j \in B_i} w_j$. The value $f_b(v_i)$ defines an upper bound on the weight of cliques that include v_i. Vertices with an upper bound value smaller than the best known objective value can be removed from the graph.
4) Graph density: $f_{den}(v_i) = 2|E|/\big(|V|(|V|-1)\big)$. Graph density is an important feature, in the sense that the percentage of vertices that form a MWC in a dense graph is usually larger than that in a sparse graph. Note that the value of f_{den} is identical for vertices in the same graph.
5) The ranking-based measure described in Section 3.2.1: f_r.
6) The correlation-based measure described in Section 3.2.2: f_c.

The six features we derived are not in the same scale, which can have a great impact on the performance of classification algorithms. Thus, we normalize the features to a similar range to avoid the dominance of large-valued features. Specifically for each graph, we normalize the features by their maximum value in the graph except for graph density. For example we normalize the vertex weight in a graph by $f_w(v_i) = w_i/w_m$, where $w_m = \max_{1 \le i \le |V|} w_i$ denoting the maximum vertex weight in the graph.

4.3 Support Vector Machine Classification

After a training dataset is constructed, we use SVM to train a supervised learning model.

4.3.1 General Formulation

Given training dataset: feature vectors $\boldsymbol{f}_i \in \mathrm{R}^n$, and class label $c_i \in \{-1, 1\}$, $i = 1, \cdots, m$, the SVM classification algorithm solves the following primal optimization problem:

$$\min_{\boldsymbol{a}, b, \boldsymbol{\xi}} \quad \frac{1}{2}\boldsymbol{a}^T\boldsymbol{a} + r\sum_{i=1}^{m} g(\xi_i), \tag{12}$$

$$s.t. \quad c_i\big(\boldsymbol{a}^T\phi(\boldsymbol{f}_i) + b\big) \ge 1 - \xi_i,\ i = 1, \cdots m, \tag{13}$$

$$\xi_i \ge 0,\ i = 1, \cdots m, \tag{14}$$

where $\phi(\boldsymbol{f}_i)$ maps the feature vector $\boldsymbol{f}_i$ into a higher-dimensional space; $r > 0$ is the regularization parameter; ξ_i, $i = 1, \cdots, m$ are slack variables and $g(\cdot)$ is a loss function.[2] The first and second order loss functions, $g(\xi_i) := \xi_i$ and $g(\xi_i) := \xi_i^2$, are widely used. We will denote SVM with first order loss function as L1-SVM and the other as L2-SVM.

2. In machine learning literature, the commonly used mathematical notations for SVM formulations are: feature vector $\boldsymbol{x}$, class label vector $\boldsymbol{y}$, weight vector $\boldsymbol{w}$ and regularization parameter C. We use different notations here as these symbols have been used in Section 3.

4.3.2 Handling Unbalanced Data

In our training dataset, the number of positive training instances (with class label 1) is typically much less than the number of negative training instances. The traditional SVM algorithm tends to classify the negative instances better than the positive instances. However in our application misclassifying a positive instance is much harmful than misclassifying a negative instance. If a positive instance is misclassified, the reduced optimization problem no longer captures the original optimal solution. On the other hand misclassifying a negative instance only results in a slight increase of the reduced problem size. In this sense, we will penalize misclassification of positive instances more by using a larger regularization parameter r^+, in contrast to that of negative instances r^-. The refined primal problem becomes

$$\min_{\boldsymbol{a}, b, \boldsymbol{\xi}} \quad \frac{1}{2}\boldsymbol{a}^T\boldsymbol{a} + r^+\sum_{c_i=1} g(\xi_i) + r^-\sum_{c_i=-1} g(\xi_i), \tag{15}$$

subject to Eq. (13) and (14).

4.3.3 Training Medium-Sized Data

In our experiments, we will consider medium-sized dataset with thousands of instances and large-sized dataset with millions of instances. To train the medium-sized dataset, we map the feature space into a higher-dimensional space using a non-linear mapping $\phi(\cdot)$ to achieve a better classification accuracy. To avoid the need to explicitly calculate the mapping function, we solve the dual problem of L1-SVM:

$$\min_{\boldsymbol{\alpha}} \quad \frac{1}{2}\boldsymbol{\alpha}^T Q\boldsymbol{\alpha} - \boldsymbol{e}^T\boldsymbol{\alpha}, \tag{16}$$

$$s.t. \quad \boldsymbol{c}^T\boldsymbol{\alpha} = 0, \tag{17}$$

$$0 \le \alpha_i \le r^+,\ \forall\, i \,|\, y_i = 1, \tag{18}$$

$$0 \le \alpha_i \le r^-,\ \forall\, i \,|\, y_i = -1, \tag{19}$$

where $\boldsymbol{e} = [1, \cdots, m]^T$ is the vector of all ones, Q is an $m \times m$ positive semidefinite matrix, and $Q_{i,j} = c_i c_j K(\boldsymbol{f}_i, \boldsymbol{f}_j)$, and $K(\boldsymbol{f}_i, \boldsymbol{f}_j) = \phi(\boldsymbol{f}_i)\phi(\boldsymbol{f}_j)$ is the kernel function. The kernel function avoids the need to compute $\phi(\cdot)$, thus is computationally efficient.

We have considered different kernel functions in our experiments, i.e., linear, polynomial, sigmoid and Radial basis function (RBF), and we observed the RBF kernel performs the best. Thus we will use the RBF kernel to train the medium dataset, which is defined as $K_{rbf}(\boldsymbol{f}_i, \boldsymbol{f}_j) = \exp(-\gamma||\boldsymbol{f}_i - \boldsymbol{f}_j||^2)$, where γ is a kernel parameter. The RBF kernel maps the feature space to an infinity dimensional space. We will use the SMO-type (Sequential Minimal Optimization) decomposition method [60] implemented in the LIBSVM library [61] to solve the dual quadratic optimization problem. The training process for the medium-sized dataset with thousands of instances takes less than 1 second. After the dual problem is solved and the optimal parameter values α_i^* and b^* are obtained, the predicted class label for a given new instance $\boldsymbol{f}$ is determined by $\mathrm{sgn}\big(\sum_{i=1}^m c_i\alpha_i^* K(\boldsymbol{f}_i, \boldsymbol{f}) + b^*\big)$.

4.3.4 Training Large-Sized Data

To train the large-sized dataset with millions of instances, solving the dual problem is very time consuming. We have

tried to solve the dual problem of L1-SVM with RBF kernel for large-sized dataset on a desktop computer and it took more than 4 days. Thus, we will instead solve the primal problem with a linear mapping $\phi(\boldsymbol{f}_i) = \boldsymbol{f}_i$ to gain computational efficiency. As the primal problem of L1-SVM is not differentiable, we will use L2-SVM that is solved by the trust region Newton method [62] implemented in the LIBLINEAR library [63]. The training time for the large-sized dataset can be significantly reduced to around 60 seconds. After the primal problem is solved and the optimal parameter values $\boldsymbol{a}_*$ and b_* are obtained, the predicted class label for a given new instance $\boldsymbol{f}$ is determined by $\text{sgn}\left(\boldsymbol{a}_*^T\boldsymbol{f} + b_*\right)$.

5 Experiments

We use simulation experiments to show the efficacy of our proposed methods for problem reduction. In Section 5.1, we investigate whether the reduced problem generated by our proposed methods can capture the optimal solution to the original optimization problem. In Section 5.2, we try to boost the performance of existing solution methods by using our problem reduction techniques as a preprocessing step. In Section 5.3, we investigate whether the vertex ordering generated by our proposed methods can be used to improve the performances of B&B algorithms.

We use 17 medium-sized synthetic graphs from DIMACS [64] (which have more than 1000 vertices) and 25 very large real-world graphs from human brain networks [65] as our datasets.[3] A brief description of these graphs is given in Table 1. The original graph is unweighted. So we assign to a vertex v_i ($i = 1, \cdots, |V|$) a weight $w_i = (i \mod 200) + 1$, following the previous works [7, 8, 51]. We divide the graphs into two sets: 1) easy graphs, for which an optimal solution can be generated by the TSM algorithm [8] within the cutoff time (1000 seconds); and 2) hard graphs, for which an optimal solution can not be generated by TSM within the cutoff time.

All the source codes are implemented in C and C++, and are compiled using GCC/7.3.0-2.30.[4] The experiments are performed on a high performance computing system.

5.1 Efficacy of Graph Reduction Techniques

5.1.1 Setup

We use easy graphs to investigate the maximum problem size one method can reduce without losing the original optimal solution. We sort the vertices in a given graph by our ranking-based measure f_r, correlation-based measure f_c or machine learning approach MLPR (denoted as ml). For MLPR we rank the vertices based on their distance to the decision boundary, i.e., $\left(\boldsymbol{a}_*^T\boldsymbol{f} + b_*\right)$. We then select the top 5%, 10%, $\cdots$, 100% vertices each as a sub-problem solved by TSM to see how many vertices are required in order to capture the optimal solution. We compare our approaches against the features computed directly from graph data, i.e., vertex weight f_w, vertex degree f_d and upper bound f_b. As our methods are based on stochastic sampling, we repeat the experiments 25 times to alleviate randomness, and the Wilcoxon rank-sum test (significance level = 0.05) with Holm p-value correction [67] is used to determine statistical significance. Note that the features computed from graph data are deterministic. We simply repeat the results generated by these graph features 25 times to conduct statistical tests.

For MLPR, we train a separate model for medium and large graphs, as discussed in Section 4. We use a "leave-one-out" strategy to construct the training dataset. For example to test on graph M_1^{tr}, we train a model using the other 7 medium graphs (M_2^{tr} to M_8^{tr}) as the training dataset. Note that in our MLPR model, a training instance refers

TABLE 1: A brief description of the datasets used in our experiments. $|V|$ is the number of vertices; $|E|$ is the number of edges; and d is the graph density. The datasets labeled as M are medium graphs from DIMACS; and those labeled as L are large real-world graphs from human brain networks. The datasets marked with tr are easy graphs for training and those marked with te are hard graphs for testing.

ID	Name	$\|V\|$	$\|E\|$	d
M_1^{tr}	p_hat1000-1	1000	122253	0.2448
M_2^{tr}	p_hat1000-2	1000	244799	0.4901
M_3^{tr}	p_hat1000-3	1000	371746	0.7442
M_4^{tr}	p_hat1500-1	1500	284923	0.2534
M_5^{tr}	p_hat1500-2	1500	568960	0.5061
M_6^{tr}	DSJC1000.5	1000	249826	0.5002
M_7^{tr}	san1000	1000	250500	0.5015
M_8^{tr}	hamming10-2	1024	518656	0.9902
M_1^{te}	p_hat1500-3	1500	847244	0.7536
M_2^{te}	C1000.9	1000	450079	0.9011
M_3^{te}	C2000.5	2000	999836	0.5002
M_4^{te}	C2000.9	2000	1799532	0.9002
M_5^{te}	C4000.5	4000	4000268	0.5002
M_6^{te}	MANN_a45	1035	533115	0.9963
M_7^{te}	MANN_a81	3321	5506380	0.9988
M_8^{te}	hamming10-4	1024	434176	0.8289
M_9^{te}	keller6	3361	4619898	0.8182
L_1^{tr}	bn$\cdots$865_session_1	734561	331832178	0.0012
L_2^{tr}	bn$\cdots$865_session_2	714808	310365050	0.0012
L_3^{tr}	bn$\cdots$867_session_1	747410	290552588	0.0010
L_4^{tr}	bn$\cdots$867_session_2	735023	309338060	0.0011
L_5^{tr}	bn$\cdots$869_session_1	690519	270077834	0.0011
L_6^{tr}	bn$\cdots$869_session_2	716150	303089830	0.0012
L_7^{tr}	bn$\cdots$870_session_1	797293	297509236	0.0009
L_8^{tr}	bn$\cdots$870_session_2	810505	333612086	0.0010
L_9^{tr}	bn$\cdots$871_session_1	747343	337358624	0.0012
L_{10}^{tr}	bn$\cdots$873_session_1	645518	299094892	0.0014
L_{11}^{tr}	bn$\cdots$873_session_2	692397	280204316	0.0012
L_{12}^{tr}	bn$\cdots$886_session_1	780185	316369494	0.0010
L_{13}^{tr}	bn$\cdots$889_session_2	742862	263853546	0.0010
L_{14}^{tr}	bn$\cdots$912_session_2	781747	295125104	0.0010
L_1^{te}	bn$\cdots$864_session_1	696338	286316678	0.0006
L_2^{te}	bn$\cdots$864_session_2	692957	267455032	0.0006
L_3^{te}	bn$\cdots$868_session_1	727487	300887106	0.0011
L_4^{te}	bn$\cdots$868_session_2	728087	317241858	0.0012
L_5^{te}	bn$\cdots$871_session_2	734729	342011384	0.0013
L_6^{te}	bn$\cdots$872_session_2	768677	295622274	0.0010
L_7^{te}	bn$\cdots$874_session_2	769392	327046802	0.0011
L_8^{te}	bn$\cdots$876_session_1	789979	280724852	0.0009
L_9^{te}	bn$\cdots$876_session_2	779330	279742484	0.0009
L_{10}^{te}	bn$\cdots$878_session_1	699697	255812256	0.0010
L_{11}^{te}	bn$\cdots$889_session_1	704694	288939700	0.0012

3. The datasets can be downloaded from Network Repository [66]: http://networkrepository.com

4. The source codes will be made publicly available if the paper gets published.

to a vertex instead of a graph. Thus the medium training dataset roughly contains thousands of instances and the large training dataset contains millions of instances.

The parameter setting used is: the number of randomly generated solutions $n = 10\sqrt{|E|}$; kernel parameter $\gamma = 1/n_f$ where n_f is the number of features; regularization parameters $r^- = 1$ and $r^+ = \epsilon_m n_{-1}/n_1$, where n_{-1} and n_1 are the number of negative and positive instances in the training set, and ϵ_m controls the penalty for misclassifying positive instances. We have tested multiple ϵ_m values and found $\epsilon_m = 1$ performs well for training medium dataset (using RBF kernel) while $\epsilon_m = 100$ is good for training large dataset (using linear feature space mapping).

5.1.2 Results

The mean ratio of vertices required by each method to capture the optimal solution is presented in Table 2. We can observe that it is possible to generate an optimal solution to the original problem by solving a reduced problem, especially for sparse graphs. Our MLPR method generally requires the least number of vertices in order to capture the optimal solution. Our statistical measures outperform the problem specific features on medium graphs and generate comparable results with the vertex degree (f_d) and vertex bound (f_b) features on large graphs. The vertex weight feature f_w is effective for some medium graphs (e.g., M_7^{tr}), however very ineffective for large graphs. It is not surprising that the performances of vertex degree f_d and vertex bound f_b features are highly correlated; they are more effective for large sparse graphs than medium dense graphs.

In Fig. 2, we plot the number of selected vertices against the best objective value found in the corresponding subproblem for some selected graphs. We can observe that the curves generated by our statistical measures and MLPR, in many cases, are on top of those generated by problem specific features. The error bar generated by MLPR generally converges faster than the statistical measures, especially on M_4^{tr} (Fig. 2b).

We have considered four kernel functions, i.e., linear, polynomial, sigmoid and RBF, when training medium-sized dataset, and the results are presented in Table 3. We can observe that the RBF kernel generally requires the least percentage of vertices to capture the original optimal solution. Thus we will use RBF kernel to train medium-sized dataset in the rest of our experiments.

The feature weights from the primal problem of SVM, i.e., vector $\boldsymbol{a}$ in Eq. (13), is an indication of how important the features are in terms of classification tasks. To investigate the feature importance in our model, we solve the primal problem of linear L2-SVM using LIBLINEAR ($\epsilon_m = 100$). The optimal weights associated with each feature in the trained models of the medium and large datasets are presented in Table 4. The features except for graph density are normalized by their maximum value in a graph, thus they have a maximum value of 1. On the medium dataset, our statistical measures f_r and f_c have the largest absolute value of the optimal weights, thus contribute the most to the classification tasks. On the large dataset, the optimal weight of f_{den} has a large absolute value, because the density of these graphs is very small (around 10^{-3}); the multiplication of graph density and its optimal weight is around 1.4, which is also less than that of our statistical measures.

TABLE 2: The average percentage of vertices required by each method in order to capture the original optimal solution in the reduced problem. The statistically best percentage for each graph is in bold. The second to last row calculates the mean percentage required by each method across 22 $(8 + 14)$ graphs and 25 independent runs. The last row denotes the p values from statistical tests when comparing ml against each of the other methods, taking the 550 (22×25) ratios generated by each method as input.

G	f_w	f_d	f_b	f_r	f_c	ml
M_1^{tr}	**0.30**	0.50	0.50	0.43	**0.30**	**0.27**
M_2^{tr}	1.00	0.25	0.25	0.21	0.24	**0.18**
M_3^{tr}	0.50	**0.35**	**0.35**	**0.35**	0.46	**0.33**
M_4^{tr}	0.60	0.50	0.45	0.38	0.35	**0.24**
M_5^{tr}	0.95	0.25	0.25	**0.20**	**0.19**	**0.17**
M_6^{tr}	0.50	0.95	0.65	0.45	**0.39**	**0.38**
M_7^{tr}	**0.15**	0.80	0.75	0.47	0.32	0.38
M_8^{tr}	1.00	1.00	1.00	0.98	1.00	0.99
L_1^{tr}	1.00	0.10	0.10	**0.06**	0.11	**0.06**
L_2^{tr}	1.00	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**
L_3^{tr}	1.00	**0.05**	**0.05**	0.06	0.11	**0.05**
L_4^{tr}	1.00	**0.05**	**0.05**	**0.05**	0.08	**0.05**
L_5^{tr}	1.00	**0.05**	**0.05**	0.07	0.09	**0.05**
L_6^{tr}	1.00	**0.05**	**0.05**	**0.05**	0.08	**0.05**
L_7^{tr}	1.00	**0.15**	**0.15**	0.22	0.22	0.17
L_8^{tr}	1.00	**0.05**	**0.05**	**0.05**	0.06	**0.05**
L_9^{tr}	1.00	0.10	0.10	**0.09**	0.22	**0.07**
L_{10}^{tr}	1.00	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**
L_{11}^{tr}	1.00	0.10	0.10	0.08	0.08	**0.05**
L_{12}^{tr}	1.00	**0.05**	**0.05**	0.23	0.20	**0.05**
L_{13}^{tr}	1.00	0.10	0.10	**0.06**	0.11	0.09
L_{14}^{tr}	1.00	0.20	0.20	**0.05**	0.06	**0.05**
mean	0.86	0.26	0.24	0.21	0.22	**0.17**
p value	2e-99	5e-08	7e-08	3e-02	2e-10	-

TABLE 3: A comparison of kernel functions in terms of the percentage of vertices required to generate the optimal solution. The statistically best percentage is in bold.

G	RBF	linear	polynomial	sigmoid
M_1^{tr}	**0.27**	0.37	0.33	0.47
M_2^{tr}	**0.18**	0.20	**0.18**	0.28
M_3^{tr}	**0.33**	0.39	0.44	0.55
M_4^{tr}	**0.24**	**0.25**	**0.25**	0.51
M_5^{tr}	**0.17**	**0.17**	**0.17**	0.29
M_6^{tr}	0.38	0.38	0.34	0.37
M_7^{tr}	**0.38**	0.51	**0.44**	0.65
M_8^{tr}	0.99	1.00	1.00	1.00

TABLE 4: The optimal weights associated with each feature in the trained SVM models for medium and large datasets.

Dataset	f_{den}	f_w	f_d	f_b	f_r	f_c
Medium	-0.40	-0.25	0.36	0.39	1.11	1.82
Large	-1346.02	-0.40	2.42	2.50	4.39	3.68

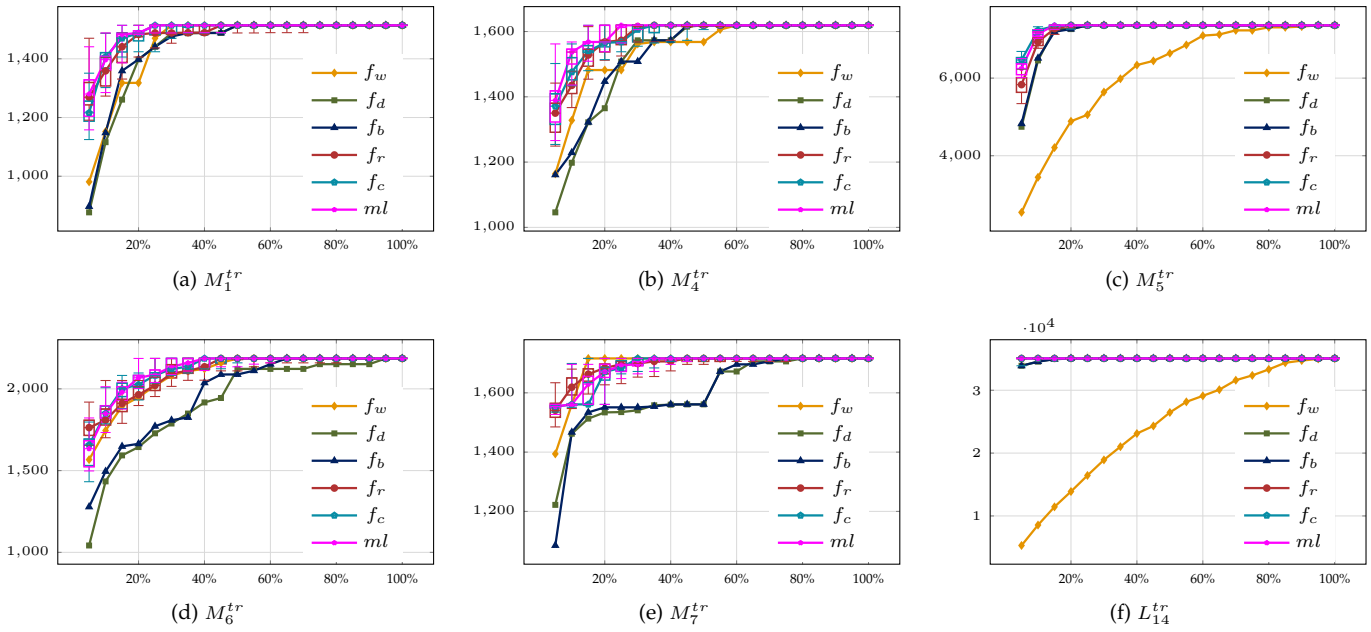


Fig. 2: A comparison between different graph reduction techniques: ml, f_c, f_r, f_b, f_d and f_w. The horizontal axis represents the percentage of vertices selected by each method; and the vertical axis represents the best objective values generated by solving the subproblem formed by the selected vertices.

5.2 Improving Existing Algorithms by Graph Reduction

5.2.1 Setup

In this section, we aim to improve the performance of existing solution methods on hard graphs using our graph reduction techniques as a preprocessing step. For our statistical measures we remove from a given graph the vertices whose score is less than a threshold, i.e ($f_r < \epsilon_r$ or $f_c < \epsilon_c$). For our MLPR method we remove the vertices that are predicted to be negative (class label -1). As discussed in Section 4, we train a separate model for medium-sized and large-sized hard graphs using the corresponding easy graphs as training dataset. We expect the optimal solutions of easy graphs are likely to be close to those of hard graphs in the feature space, thus it is possible to transfer the mapping learned from easy graphs to hard graphs.

We then use existing solution methods to solve the reduced problem to see if our problem reduction techniques are beneficial. We test two exact solvers – TSM [8] and WLMC [7] as well as two heuristic methods LSCC+BMS [54] and FastWClq [53]. We denote a solution method A with different problem reduction techniques as A-f_r, A-f_c and A-ml respectively. Although exact solvers have the optimality guarantee, the time required to generate an optimal solution may be very long. Thus we simply set a cutoff time (1000 seconds) for each algorithm, and use the best objective value obtained within the cutoff time as an indication of algorithm performance. Note that the preprocessing time used by our problem reduction techniques is counted as part of the cutoff time. Each algorithm is independently run 25 times to alleviate randomness and the test used to determine statistical significance is the same as before. We rank the algorithms on each graph and compute the average ranking across all the graphs as an indication of their overall performance.

We set the parameters $\epsilon_r = 0.01$ and $\epsilon_c = 0$, and test two values of ϵ_m (i.e., 10 and 100) to investigate the effects of different level of penalty has on the algorithm performance. The other parameter setting is the same as before.

5.2.2 Results

The results for TSM, LSCC, WLMC and FastWClq are summarized in Table 5, 6, 7 and 8 respectively. We can observe that our proposed problem reduction techniques can greatly improve the performance of existing solution methods as a preprocessing step. The best solution quality generated can be significantly improved especially for hard problem instances that an existing solution method performs poorly on. Note that the results generated by the exact solvers, TSM and WLMC, may not be exactly the same across the 25 independent runs, because the graph loading time varies slightly especially for large graphs. The percentage of vertices removed by our problem reduction methods may also slightly vary, as we re-generate a reduced graph instance in each independent run and our statistical measures are based on stochastic sampling.

Our correlation-based measure f_c tends to remove about 50% of vertices from both the medium dense and large sparse graphs when the parameter ϵ_c is set to 0. This amount of problem reduction works well on most of the medium-sized graphs, but is not enough for large sparse graphs. In this regard, our correlation-based measure is not well adaptive to graph density. However we show in the supplementary material that, by simply setting ϵ_c to 0.01 the correlation-based measure f_c can further reduce the size of large graphs to about 15%, and can further improve the performance of solution algorithms.

Our ranking-based measure f_r (with $\epsilon_r = 0.01$) is very effective in reducing the size of large sparse graphs, but

TABLE 5: The results of TSM, TSM-f_r, TSM-f_c and TSM-ml when used to solve the hard instances; $\bar{y}$ and σ_y denote the mean and standard deviation of best objective values generated in 25 independent runs within the cutoff time (1000 seconds); and $\bar{p}$ denotes the mean ratio of selected vertices. The statistically best solution quality is in bold. The last row shows the average ranking ($\bar{r}$) of each algorithm across all datasets.

G	TSM		TSM-f_r ($\epsilon_r = 0.01$)			TSM-f_c ($\epsilon_c = 0$)			TSM-ml ($\epsilon_m = 10$)			TSM-ml ($\epsilon_m = 100$)		
	$\bar{y}$	σ_y	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$
M_1^{te}	10119	0	10069	144	0.93	**10286**	82	0.41	**10294**	51	0.27	10028	182	0.49
M_2^{te}	7341	0	7479	277	1.00	**8208**	197	0.53	**8250**	142	0.55	7925	136	0.75
M_3^{te}	2407	0	2431	31	0.98	**2465**	4	0.53	**2466**	0	0.48	**2461**	12	0.69
M_4^{te}	8228	0	7805	231	1.00	**8792**	206	0.54	**8898**	161	0.52	8416	215	0.78
M_5^{te}	2402	0	2445	53	0.79	**2590**	47	0.53	**2601**	42	0.49	2539	45	0.71
M_6^{te}	34259	0	**34265**	0	1.00	33882	20	0.52	34253	6	0.98	**34265**	0	1.00
M_7^{te}	109191	4	109870	77	1.00	**110080**	35	0.51	109850	59	0.98	109890	64	0.99
M_8^{te}	4812	0	4678	92	1.00	**4899**	57	0.76	**4914**	67	0.78	4853	70	0.95
M_9^{te}	4762	0	**5306**	168	1.00	**5280**	208	0.60	**5260**	144	0.39	**5212**	161	0.80
L_1^{te}	32105	7	**32232**	57	0.02	32000	36	0.38	31958	193	0.02	32133	130	0.04
L_2^{te}	26412	0	**26890**	387	0.03	**26844**	398	0.39	**26757**	378	0.06	**26786**	342	0.11
L_3^{te}	31228	76	31298	531	0.02	**31748**	318	0.39	**31496**	376	0.06	**31650**	341	0.11
L_4^{te}	27972	0	**29511**	184	0.02	**29548**	0	0.39	**29492**	280	0.05	**29548**	0	0.09
L_5^{te}	30310	0	**32789**	231	0.02	**32731**	303	0.40	**32760**	260	0.05	**32722**	311	0.11
L_6^{te}	31371	23	**35544**	557	0.02	32572	612	0.39	32801	523	0.04	32514	617	0.08
L_7^{te}	28232	0	**30788**	198	0.03	**30734**	533	0.38	**30757**	459	0.05	**30827**	161	0.08
L_8^{te}	48716	331	49256	579	0.02	46699	2017	0.37	49630	311	0.06	**50063**	188	0.10
L_9^{te}	32658	94	33055	57	0.02	32969	44	0.37	**33085**	0	0.05	**33085**	0	0.09
L_{10}^{te}	25637	57	**27715**	302	0.02	**27775**	0	0.40	**27726**	243	0.05	**27703**	192	0.10
L_{11}^{te}	22749	206	**26459**	327	0.02	26190	0	0.40	26323	72	0.06	**26351**	37	0.12
$\bar{r}$	4.5		2.4			1.9			1.5			1.7		

TABLE 6: The results of LSCC, LSCC-f_r, LSCC-f_c and LSCC-ml when used to solve the hard instances; $\bar{y}$ and σ_y denote the mean and standard deviation of best objective values generated in 25 independent runs within the cutoff time (1000 seconds); and $\bar{p}$ denotes the mean ratio of number of vertices selected. The statistically best solution quality is in bold. The last row shows the average ranking ($\bar{r}$) of each algorithm across all datasets.

G	LSCC		LSCC-f_r ($\epsilon_r = 0.01$)			LSCC-f_c ($\epsilon_c = 0$)			LSCC-ml ($\epsilon_m = 10$)			LSCC-ml ($\epsilon_m = 100$)		
	$\bar{y}$	σ_y	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$
M_1^{te}	**10320**	5	**10318**	9	0.93	**10320**	5	0.40	10305	28	0.26	**10321**	0	0.48
M_2^{te}	**9251**	14	**9252**	13	1.00	9119	46	0.53	9135	63	0.54	9240	26	0.75
M_3^{te}	**2466**	0	**2466**	0	0.98	**2466**	1	0.53	**2466**	1	0.49	**2466**	0	0.69
M_4^{te}	10905	36	10917	33	1.00	10887	32	0.54	10883	37	0.52	**10933**	16	0.78
M_5^{te}	**2792**	0	**2791**	2	0.79	2781	13	0.53	2781	17	0.49	**2792**	0	0.70
M_6^{te}	**34230**	5	**34228**	6	1.00	33891	23	0.52	34220	13	0.98	**34231**	5	1.00
M_7^{te}	**111070**	14	111060	14	1.00	110130	42	0.51	110960	40	0.98	111060	9	0.99
M_8^{te}	**5129**	0	**5129**	0	1.00	**5129**	0	0.76	**5129**	0	0.78	**5129**	0	0.95
M_9^{te}	7778	72	7766	60	1.00	7789	70	0.60	7530	54	0.39	**7850**	87	0.80
L_1^{te}	23969	1812	30943	641	0.02	27972	1187	0.38	**31570**	283	0.02	**31443**	474	0.04
L_2^{te}	21430	1594	**26900**	460	0.03	**26385**	868	0.39	**26625**	620	0.06	**26794**	545	0.11
L_3^{te}	23923	3195	**31386**	559	0.02	**31403**	1187	0.39	**31401**	695	0.06	**31393**	769	0.11
L_4^{te}	23066	2006	**29454**	325	0.02	**29345**	730	0.39	**29500**	239	0.05	**29511**	183	0.09
L_5^{te}	25843	3039	**32797**	188	0.02	**32746**	282	0.40	**32685**	351	0.05	**32814**	103	0.11
L_6^{te}	24984	4322	**35537**	128	0.02	33061	847	0.39	35392	231	0.04	35077	515	0.08
L_7^{te}	24463	2320	**30795**	448	0.03	**30602**	1048	0.38	**30776**	454	0.05	**30776**	454	0.08
L_8^{te}	25531	4511	**49460**	561	0.02	39270	7164	0.37	**48305**	4371	0.06	**45657**	6721	0.10
L_9^{te}	24163	2875	**31395**	428	0.02	28120	785	0.37	31011	808	0.05	30418	747	0.09
L_{10}^{te}	19373	2411	**27655**	416	0.02	**27775**	0	0.40	**27694**	403	0.05	**27642**	393	0.10
L_{11}^{te}	21804	2121	**27162**	136	0.02	24824	884	0.40	26877	301	0.06	26450	419	0.12
$\bar{r}$	3.3		1.2			2.6			2.2			1.4		

TABLE 7: The results of WLMC, WLMC-f_r, WLMC-f_c and WLMC-ml when used to solve the hard instances; $\bar{y}$ and σ_y denote the mean and standard deviation of best objective values generated in 25 independent runs within the cutoff time (1000 seconds); and $\bar{p}$ denotes the mean ratio of number of vertices selected. The statistically best solution quality is in bold. The last row shows the average ranking ($\bar{r}$) of each algorithm across all datasets.

G	WLMC		WLMC-f_r ($\epsilon_r = 0.01$)			WLMC-f_c ($\epsilon_c = 0$)			WLMC-ml ($\epsilon_m = 10$)			WLMC-ml ($\epsilon_m = 100$)		
	$\bar{y}$	σ_y	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$
M_1^{te}	9846	0	9791	48	0.93	9819	48	0.41	**9960**	169	0.27	9793	32	0.48
M_2^{te}	7317	0	7182	226	1.00	**8078**	168	0.53	**8101**	185	0.54	7851	209	0.75
M_3^{te}	2360	0	2396	42	0.98	**2466**	1	0.53	**2465**	3	0.49	2454	22	0.69
M_4^{te}	7738	0	7739	221	1.00	8666	195	0.54	**8808**	202	0.52	8387	220	0.78
M_5^{te}	2383	0	2452	59	0.79	**2590**	44	0.53	**2592**	44	0.49	2448	38	0.70
M_6^{te}	**34265**	0	**34265**	0	1.00	33882	25	0.52	34255	6	0.98	**34265**	0	1.00
M_7^{te}	109789	16	109850	49	1.00	**110120**	50	0.51	110020	18	0.98	110000	8	0.99
M_8^{te}	4738	0	4386	133	1.00	4737	92	0.76	**4825**	78	0.78	4707	97	0.95
M_9^{te}	4760	0	**5214**	138	1.00	**5203**	115	0.60	**5225**	146	0.39	**5264**	158	0.80
L_1^{te}	25293	0	**31807**	341	0.02	26972	579	0.38	29533	720	0.02	27222	752	0.04
L_2^{te}	22332	0	**26331**	826	0.03	**26133**	1219	0.39	**26253**	1063	0.06	**26264**	1136	0.11
L_3^{te}	28044	0	**31446**	1065	0.02	**31438**	1077	0.39	**30907**	1381	0.06	**30799**	1454	0.11
L_4^{te}	20819	0	**29548**	0	0.02	**29453**	477	0.39	**29417**	654	0.05	**29548**	0	0.09
L_5^{te}	29398	0	**32659**	378	0.02	**32789**	231	0.40	**32797**	188	0.05	**32685**	351	0.11
L_6^{te}	26557	0	**34027**	663	0.02	32802	631	0.39	32649	527	0.04	32716	671	0.08
L_7^{te}	24560	0	**30753**	462	0.03	**30687**	621	0.38	**30757**	459	0.05	**30866**	97	0.08
L_8^{te}	34356	0	**39221**	5755	0.02	**41562**	5691	0.37	**40006**	5735	0.06	**39276**	5385	0.10
L_9^{te}	32167	0	**32347**	50	0.02	32173	29	0.37	32167	0	0.05	32224	29	0.09
L_{10}^{te}	24991	0	**27706**	343	0.02	**27775**	0	0.40	**27697**	389	0.05	**27775**	0	0.10
L_{11}^{te}	25205	0	**25309**	162	0.02	25050	98	0.40	24647	41	0.06	25205	0	0.12
$\bar{r}$	4.2		2.0			1.8			1.6			1.9		

TABLE 8: The results of FastWClq, FastWClq-f_r, FastWClq-f_c and FastWClq-ml when used to solve the hard instances; $\bar{y}$ and σ_y denote the mean and standard deviation of best objective values generated in 25 independent runs within the cutoff time (1000 seconds); and $\bar{p}$ denotes the mean ratio of number of vertices selected. The statistically best solution quality is in bold. The last row shows the average ranking ($\bar{r}$) of each algorithm across all datasets.

G	FastWClq		FastWClq-f_r ($\epsilon_r = 0.01$)			FastWClq-f_c ($\epsilon_c = 0$)			FastWClq-ml ($\epsilon_m = 10$)			FastWClq-ml ($\epsilon_m = 100$)		
	$\bar{y}$	σ_y	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$	$\bar{y}$	σ_y	$\bar{p}$
M_1^{te}	9852	40	9890	103	0.94	**10108**	57	0.41	**10120**	65	0.27	10055	62	0.49
M_2^{te}	8546	40	8516	99	1.00	**8856**	81	0.53	**8848**	109	0.55	8711	93	0.75
M_3^{te}	2400	16	2428	28	0.98	**2464**	5	0.53	**2464**	5	0.49	**2462**	5	0.69
M_4^{te}	9703	78	9594	85	1.00	**10145**	110	0.54	**10104**	85	0.53	9840	90	0.78
M_5^{te}	2437	72	2560	29	0.79	2679	44	0.53	**2724**	52	0.49	2616	59	0.70
M_6^{te}	34097	3	**34102**	6	1.00	33878	18	0.52	**34109**	15	0.98	**34103**	7	1.00
M_7^{te}	**110420**	55	110330	80	1.00	110120	27	0.51	**110480**	71	0.98	**110420**	78	0.99
M_8^{te}	4968	33	4923	31	1.00	**5013**	41	0.76	**4990**	59	0.79	4931	19	0.95
M_9^{te}	5766	78	5602	144	1.00	6108	97	0.60	**6319**	134	0.39	5884	90	0.80
L_1^{te}	30666	373	**31165**	219	0.02	30164	525	0.38	30728	386	0.02	30809	231	0.04
L_2^{te}	**27025**	386	26665	617	0.03	**27094**	319	0.39	26678	538	0.06	26778	467	0.11
L_3^{te}	**31854**	244	31280	678	0.02	**31868**	208	0.39	**31738**	349	0.06	**31659**	426	0.11
L_4^{te}	**29548**	0	**29548**	0	0.02	**29548**	0	0.39	**29548**	0	0.05	**29406**	493	0.09
L_5^{te}	32165	458	**32701**	341	0.02	**32835**	0	0.40	**32750**	283	0.05	**32798**	130	0.11
L_6^{te}	34790	120	**35098**	161	0.02	34519	247	0.39	**34975**	161	0.04	**34951**	216	0.08
L_7^{te}	**30885**	0	**30691**	602	0.03	**30880**	15	0.38	**30638**	676	0.05	**30845**	140	0.08
L_8^{te}	49912	1932	49348	519	0.02	49488	2696	0.37	50177	166	0.06	**50342**	45	0.10
L_9^{te}	**30251**	525	**30502**	514	0.02	29858	700	0.37	**30432**	390	0.05	**30317**	390	0.09
L_{10}^{te}	**27775**	0	**27775**	0	0.02	**27649**	417	0.40	**27775**	0	0.05	**27775**	0	0.10
L_{11}^{te}	26204	32	**26624**	275	0.02	25960	288	0.40	**26504**	169	0.06	**26413**	221	0.12
$\bar{r}$	2.9		2.7			2.3			1.2			1.8		

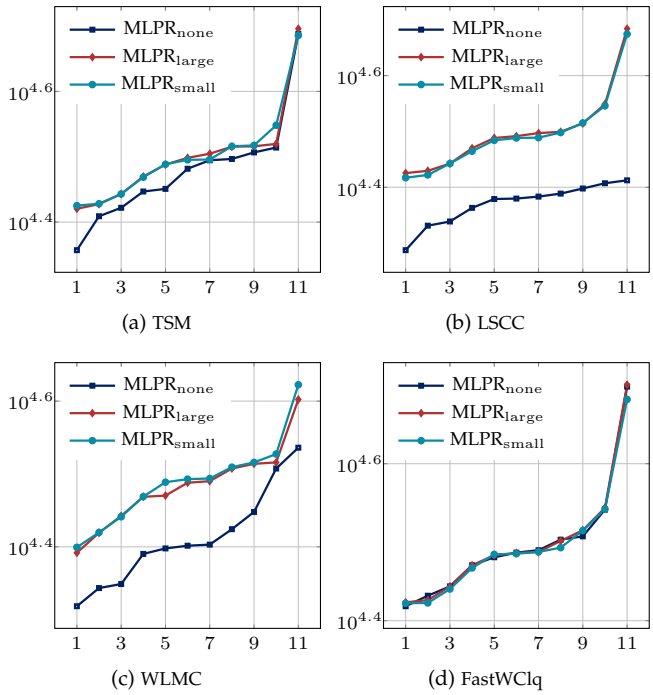


Fig. 3: A comparison between MLPR$_{\text{small}}$ (trained on small graphs), MLPR$_{\text{large}}$ (trained on large graphs) and MLPR$_{\text{none}}$ (without any problem reduction) when incorporated with the 4 algorithms to solve the 11 large hard problem instances (L^{te}). The horizontal axis represents the graph index, and the vertical axis represents the mean of best objective values generated ($\bar{y}$) within the cutoff time (1000 seconds). For each method we sort $\bar{y}$ from the 11 graphs in ascending order to generate the plots for easier visualization.

less effective for medium-sized graphs. Even when using a slightly larger parameter value ($\epsilon_r = 0.03$), our ranking-based measure is still unable to remove any vertex from some of the medium-sized graphs, partially because these graphs are very dense (see the supplementary material for detailed results). When combined with LSCC our ranking-based measure achieves the best average ranking, due to 1) LSCC is very effective for solving the medium-sized graphs, thus it can find a good solution even though f_r cannot effectively reduce the problem size; and 2) LSCC is very ineffective for solving the large-sized graphs, thus it greatly benefits from the huge size reduction by f_r for large graphs.

As expected our MLPR method is the most robust as it takes several features into account. It achieves the best average ranking when incorporated with TSM, WLMC and FastWClq, and comparable results with the ranking-based measure f_r when combined with LSCC. Furthermore the reduced problem size by MLPR is more adaptive to graph density, in the sense that it tends to remove more vertices from sparse graph but less from dense graph. Lastly we observe that when using a smaller parameter value ϵ_m, our MLPR method removes more vertices from a graph.

To test the scalability of our MLPR model, we apply the MLPR model trained on the 8 medium-sized synthetic graphs (M^{tr}) to reduce the problem size for large real-world graphs (L^{te}). However we observe that this trained model tends to remove too many vertices from these large graphs. We infer the reason is that the training instances collected from the 8 medium graphs are biased and do not cover the feature space well. We then solve this issue by including 18 more small graphs ($|V| < 1000$) from the DIMACS benchmark into the training set. A brief description of these 18 graphs can be found in the supplementary material.

As discussed in Section 4, we trained our MLPR model for medium-sized dataset by solving the dual problem of L1-SVM with RBF kernel before. However the prediction time used by this model to remove vertices from the large graphs is long (around 300 seconds). Thus we will instead train the MLPR model by solving the primal problem of linear L2-SVM to gain computational efficiency, and the prediction time can be significantly reduced to around 2 seconds. We compare this model, termed as MLPR$_{\text{small}}$, against MLPR$_{\text{large}}$ which is trained on the large easy graphs L^{tr}, as well as MLPR$_{\text{none}}$ (without any problem reduction), when incorporated with the 4 solution algorithms to solve the 11 large hard graphs L^{te}. The parameter ϵ_m is set to 10 for both of the MLPR$_{\text{small}}$ and MLPR$_{\text{large}}$ methods.

The results are shown in Fig. 3. To generate these plots, we sort the best objective values obtained by each algorithm on the 11 graphs in ascending order for better visualization. Thus the graph index in these plots may not match that of Table 1. However the detailed results and average ranking of each algorithm can be found in the supplementary material. We observe that the MLPR$_{\text{small}}$ method significantly boosts the performance of TSM, LSCC and WLMC; and achieves comparable results against MLPR$_{\text{large}}$ when used to solve the 11 large hard instances. The FastWClq algorithm only benefits slightly from our problem reduction techniques in terms of the average ranking (listed in the supplementary material), because it is already very effective in solving these hard instances. It is worth noting that the LSCC algorithm is very ineffective in solving these large instances, however by using our MLPR methods as a preprocessing step its performance can be significantly boosted to a level that is competitive with FastWClq.

In the supplementary material we have also tested the scalability of our MLPR model on other large real-world graphs that have not been considered here. The results show that our MLPR model trained on small and median graphs can significantly reduce the size of large graphs, and is still able to capture an original optimal solution (or at least a near-optimal solution) in the reduced graph.

In practice if a training dataset is available we suggest to use our MLPR method for problem reduction. Otherwise our proposed statistical measures can be used for problem reduction; specifically the correlation-based measure f_c for medium-sized graphs and the ranking-based measure f_r for large sparse graphs. Furthermore the LSCC algorithm tends to perform well on medium-size graphs, and the FastWClq and TSM algorithms are good candidates to use for solving large sparse graphs. Last we suggest the reduced problem size can be tuned by varying the parameters ϵ_r, ϵ_c and ϵ_m.

5.3 Improving B&B Algorithms by Vertex Ordering

5.3.1 Setup

Decision variable ordering defines the search tree for B&B algorithms, that has a large impact on the algorithm perfor-

TABLE 9: The results of TSM, TSM-f_r-O, TSM-f_c-O and TSM-ml-O when used to solve the hard instances; $\bar{y}$ and σ_y denote the mean and standard deviation of best objective values generated in 25 independent runs within the cutoff time (1000 seconds). The statistically best solution quality is in bold. The last row shows the average ranking ($\bar{r}$) of each algorithm across all datasets.

G	TSM		TSM-f_r-O		TSM-f_c-O		TSM-ml-O	
	$\bar{y}$	σ_y	$\bar{y}$	σ_y	$\bar{y}$	σ_y	$\bar{y}$	σ_y
M_1^{te}	10119	0	**10241**	73	**10273**	23	**10254**	52
M_2^{te}	7341	0	7735	209	**8428**	132	8184	202
M_3^{te}	2407	0	2461	13	**2465**	3	**2466**	0
M_4^{te}	8228	0	7863	218	**8705**	192	8425	245
M_5^{te}	2402	0	2600	41	**2703**	27	**2710**	21
M_6^{te}	34259	0	**34265**	0	34238	5	**34265**	0
M_7^{te}	109191	4	110200	31	**110280**	50	**110240**	150
M_8^{te}	4812	0	4896	49	**4965**	43	**4955**	40
M_9^{te}	4762	0	6175	217	**6396**	139	**6311**	197
L_1^{te}	32105	7	**32285**	22	31940	590	32193	163
L_2^{te}	26412	0	**27002**	241	**26946**	272	**27051**	193
L_3^{te}	31228	76	**31776**	267	**31650**	346	**31631**	322
L_4^{te}	27972	0	**29548**	0	**29548**	0	**29548**	0
L_5^{te}	30310	0	**32665**	356	**32685**	351	**32722**	311
L_6^{te}	31371	23	**35698**	0	**35698**	0	**35698**	0
L_7^{te}	28232	0	**30827**	288	**30866**	97	**30885**	0
L_8^{te}	48716	331	**49747**	472	48745	861	48394	2022
L_9^{te}	**32658**	94	30907	1434	29334	548	30574	1336
L_{10}^{te}	25637	57	**27775**	0	**27775**	0	**27775**	0
L_{11}^{te}	22749	206	**27492**	30	27104	590	**27456**	188
$\bar{r}$	3.6		1.8		1.6		1.2	

TABLE 10: The results of WLMC, WLMC-f_r-O, WLMC-f_c-O and WLMC-ml-O when used to solve the hard instances; $\bar{y}$ and σ_y denote the mean and standard deviation of best objective values generated in 25 independent runs within the cutoff time (1000 seconds). The statistically best result is in bold. The last row shows the average ranking ($\bar{r}$) of each algorithm across all datasets.

G	WLMC		WLMC-f_r-O		WLMC-f_c-O		WLMC-ml-O	
	$\bar{y}$	σ_y	$\bar{y}$	σ_y	$\bar{y}$	σ_y	$\bar{y}$	σ_y
M_1^{te}	9846	0	10130	115	**10244**	32	**10224**	42
M_2^{te}	7317	0	7613	278	**8350**	122	8110	150
M_3^{te}	2360	0	2459	14	**2466**	0	**2466**	0
M_4^{te}	7738	0	7726	234	**8607**	182	8237	189
M_5^{te}	2383	0	2594	43	**2698**	27	**2703**	28
M_6^{te}	**34265**	0	**34265**	0	34251	5	**34265**	0
M_7^{te}	109789	16	**110360**	43	**110370**	60	110320	63
M_8^{te}	4738	0	4806	60	**4923**	46	4881	56
M_9^{te}	4760	0	6082	146	**6268**	143	**6244**	180
L_1^{te}	25293	0	**30647**	721	27519	757	**30290**	897
L_2^{te}	22332	0	**26673**	675	**26497**	1037	**26654**	769
L_3^{te}	28044	0	**31318**	808	30722	1364	**31546**	659
L_4^{te}	20819	0	**29358**	530	**29443**	523	**29415**	465
L_5^{te}	29398	0	**32760**	260	**32601**	427	**32722**	311
L_6^{te}	26557	0	**35367**	456	**34927**	952	**34994**	825
L_7^{te}	24560	0	**30821**	240	**30866**	97	**30757**	459
L_8^{te}	34356	0	**50289**	146	47064	6664	**48041**	5601
L_9^{te}	**32167**	0	30270	1111	28950	0	29617	838
L_{10}^{te}	24991	0	**27775**	0	**27775**	0	**27775**	0
L_{11}^{te}	25205	0	**26568**	713	24764	609	**26049**	999
$\bar{r}$	3.7		1.8		1.7		1.3	

mance. In TSM and WLMC vertices are sorted by degree; that is repeatedly removing the vertex with smallest degree from the current graph. Instead, the vertex ordering generated by our methods described in Section 5.1 can be used as the branching order for TSM and WLMC. We denote an algorithm B (i.e., TSM or WLMC) using different orderings generated by our methods as B-O_{f_r}, B-O_{f_c} and B-O_{ml} respectively. Each algorithm is run 25 times on each hard graph and the cutoff time is set to 1000 seconds. The parameter setting is the same as Section 5.1.

5.3.2 Results

The results for TSM and WLMC are summarized in Table 9 and 10. We can observe that the vertex orderings produced by our problem reduction techniques can significantly improve the best solution quality found by TSM and WLMC. The MLPR method consistently achieves overall the best performance. Note that although our vertex ordering can guide the B&B algorithms towards a better solution quickly, it does not necessarily mean that the search tree generated by our vertex ordering is smaller. There has been some work that learns to minimize the size of search tree [16, 17]. However as the vertex ordering is only a by-product of our problem reduction techniques, a further investigation along this line is beyond the scope of this paper.

6 Conclusion

In this paper, we tackled large-scale combinatorial optimization problems via problem reduction. We used the maximum weight clique problem as an example, and showed how to heuristically remove vertices from a graph that are not expected to be part of the optimal solution. First we described two statistical measures computed from stochastic sampling of feasible solutions to quantify the "quality" of each vertex. We then used these measures to guide graph reduction and showed they are more useful than the features computed directly from graph data. To take this further, we proposed a machine learning approach named MLPR that combines the statistical measures with graph features, thus it enables us to better predict the vertices that belong to the optimal solution for a given graph. We used easy graphs for which the optimal solutions are known as the training dataset, and showed the knowledge learned from easy graphs is useful for reducing the problem size for a hard graph. We evaluated our problem reduction techniques using simulation experiments and showed they are effective and can boost the performance of existing solution methods.

A logical extension of this work would be to test our problem reduction techniques on other combinatorial optimization problems, e.g., traveling salesman problem and graph coloring problem. Our overarching goal is to develop a generic automated problem reduction method that simply takes a very large mixed integer program (MIP) as input. This can possibly be achieved by 1) transferring a given large MIP into a binary linear program (BLP); 2) generating sufficiently small cut-down versions of the BLP and solving them to optimality using a MIP solver; 3) constructing a training dataset; and 4) training a machine learning model to reduce the size of the original BLP. The reduced BLP can then be solved by using an existing solution method, e.g., a MIP solver.

ACKNOWLEDGMENTS

This work was supported by an ARC Discovery Grant (DP180101170) from Australian Research Council.

REFERENCES

[1] E. Jélvez, N. Morales, P. Nancel-Penard, J. Peypouquet, and P. Reyes, "Aggregation heuristic for the open-pit block scheduling problem," *European Journal of Operational Research*, vol. 249, no. 3, pp. 1169–1177, 2016.

[2] A. Kenny, X. Li, A. T. Ernst, and D. Thiruvady, "Towards solving large-scale precedence constrained production scheduling problems in mining," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 1137–1144.

[3] C.-H. Brech, A. Ernst, and R. Kolisch, "Scheduling medical residents training at university hospitals," *European Journal of Operational Research*, vol. 274, no. 1, pp. 253–266, 2019.

[4] R. A. Rossi, D. F. Gleich, A. H. Gebremedhin, and M. M. A. Patwary, "Fast maximum clique algorithms for large graphs," in *Proceedings of the 23rd International Conference on World Wide Web*. ACM, 2014, pp. 365–366.

[5] Q. Wu and J.-K. Hao, "A review on algorithms for maximum clique problems," *European Journal of Operational Research*, vol. 242, no. 3, pp. 693–709, 2015.

[6] F. V. Fomin, F. Grandoni, and D. Kratsch, "A measure & conquer approach for the analysis of exact algorithms," *Journal of the ACM (JACM)*, vol. 56, no. 5, p. 25, 2009.

[7] H. Jiang, C.-M. Li, and F. Manya, "An exact algorithm for the maximum weight clique problem in large graphs." in *AAAI*, 2017, pp. 830–838.

[8] H. Jiang, C.-M. Li, Y. Liu, and F. Manya, "A two-stage maxsat reasoning approach for the maximum weight clique problem." in *AAAI*, 2018.

[9] T. Akiba and Y. Iwata, "Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover," *Theoretical Computer Science*, vol. 609, pp. 211–225, 2016.

[10] S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck, "Finding near-optimal independent sets at scale," *Journal of Heuristics*, vol. 23, no. 4, pp. 207–229, 2017.

[11] R. Ruiz-Torrubiano and A. Suárez, "Hybrid approaches and dimensionality reduction for portfolio selection with cardinality constraints," *IEEE Computational Intelligence Magazine*, vol. 5, no. 2, pp. 92–107, 2010.

[12] R. Liu, A. Agrawal, W.-k. Liao, and A. Choudhary, "Search space preprocessing in solving complex optimization problems," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 1–5.

[13] H. A. Tayalı and S. Tolun, "Dimension reduction in mean-variance portfolio optimization," *Expert Systems with Applications*, vol. 92, pp. 161–169, 2018.

[14] A. Lodi and G. Zarpellon, "On learning and branching: a survey," *Top*, vol. 25, no. 2, pp. 207–236, 2017.

[15] H. He, H. Daume III, and J. M. Eisner, "Learning to search in branch and bound algorithms," in *Advances in neural information processing systems*, 2014, pp. 3293–3301.

[16] E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[17] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik, "Learning to branch," in *Proceedings of the 35th International Conference on Machine Learning*, vol. 80. PMLR, 10–15 Jul 2018, pp. 344–353.

[18] J. Boyan and A. W. Moore, "Learning evaluation functions to improve optimization by local search," *Journal of Machine Learning Research*, vol. 1, no. Nov, pp. 77–112, 2000.

[19] O. V. Shylo and H. Shams, "Boosting binary optimization via binary classification: A case study of job shop scheduling," *arXiv preprint arXiv:1808.10813*, 2018.

[20] D. Martins, G. M. Vianna, I. Rosseti, S. L. Martins, and A. Plastino, "Making a state-of-the-art heuristic faster with data mining," *Annals of Operations Research*, vol. 263, no. 1-2, pp. 141–162, 2018.

[21] W. Zhang and T. G. Dietterich, "Solving combinatorial optimization tasks by reinforcement learning: A general methodology applied to resource-constrained scheduling," *Journal of Artificial Intelligence Reseach*, vol. 1, pp. 1–38, 2000.

[22] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2692–2700.

[23] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 6348–6358.

[24] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," in *Advances in Neural Information Processing Systems*, 2018, pp. 9861–9871.

[25] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Advances in Neural Information Processing Systems*, 2018, pp. 537–546.

[26] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *arXiv preprint arXiv:1811.06128*, 2018.

[27] G. Di Liberto, S. Kadioglu, K. Leo, and Y. Malitsky, "DASH: Dynamic approach for switching heuristics," *European Journal of Operational Research*, vol. 248, no. 3, pp. 943–953, 2016.

[28] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao, "Learning to run heuristics in tree search," in *IJCAI*, 2017, pp. 659–666.

[29] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, 2013.

[30] J. Branke, S. Nguyen, C. W. Pickardt, and M. Zhang, "Automated design of production scheduling heuristics: A review," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 110–124, 2016.

[31] M. Fischetti and M. Fraccaro, "Machine learning meets mathematical optimization to predict the optimal production of offshore wind parks," *Computers & Operations Research*, vol. 106, pp. 289 – 297, 2019.

[32] E. K. Burke, M. R. Hyde, G. Kendall, and J. Woodward, "Automating the packing heuristic design process with genetic programming," *Evolutionary Computation*, vol. 20, no. 1, pp. 63–89, 2012.

[33] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "A computational study of representations in genetic programming to evolve dispatching rules for the job shop scheduling problem," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 5, pp. 621–639, 2013.

[34] M. Lombardi and M. Milano, "Boosting combinatorial problem modeling with machine learning," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI Press, 2018, pp. 5472–5478.

[35] K. A. Smith-Miles, "Cross-disciplinary perspectives on meta-learning for algorithm selection," *ACM Computing Surveys (CSUR)*, vol. 41, no. 1, p. 6, 2009.

[36] B. Bischl, P. Kerschke, L. Kotthoff, M. Lindauer, Y. Malitsky, A. Frchette, H. Hoos, F. Hutter, K. Leyton-Brown, K. Tierney, and J. Vanschoren, "Aslib: A benchmark library for algorithm selection," *Artificial Intelligence*, vol. 237, pp. 41 – 58, 2016.

[37] A. Kenny, X. Li, and A. T. Ernst, "A merge search algorithm and its application to the constrained pit problem in mining," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2018, pp. 316–323.

[38] C. Blum, P. Pinacho, M. López-Ibáñez, and J. A. Lozano, "Construct, merge, solve & adapt a new general algorithm for combinatorial optimization," *Computers & Operations Research*, vol. 68, pp. 75–88, 2016.

[39] M. Bateni, H. Esfandiari, and V. Mirrokni, "Optimal distributed submodular optimization via sketching," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2018, pp. 1138–1147.

[40] E. Lindgren, S. Wu, and A. G. Dimakis, "Leveraging sparsity for efficient submodular data summarization," in *Advances in Neural Information Processing Systems*, 2016, pp. 3414–3422.

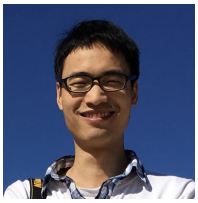
[41] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, 1997.

[42] J. S. De Bonet, C. L. Isbell Jr, and P. A. Viola, "MIMIC: Finding optima by estimating probability densities," in *Advances in Neural Information Processing Systems*, 1997, pp. 424–430.

[43] G. R. Harik, F. G. Lobo, and D. E. Goldberg, "The compact genetic algorithm," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 287–297, 1999.

[44] F. Mascia, E. Cilia, M. Brunato, and A. Passerini, "Predicting structural and functional sites in proteins by searching for maximum-weight cliques." in *AAAI*, 2010.

[45] W. Brendel and S. Todorovic, "Segmentation as maximum-weight independent set," in *Advances in neural information processing systems*, 2010, pp. 307–315.
[46] W. Brendel, M. Amer, and S. Todorovic, "Multiobject tracking as maximum weight independent set," in *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE, 2011, pp. 1273–1280.
[47] S. Butenko and W. E. Wilhelm, "Clique-detection models in computational biochemistry and genomics," *European Journal of Operational Research*, vol. 173, no. 1, pp. 1–17, 2006.
[48] Z. Fang, C.-M. Li, and K. Xu, "An exact algorithm based on maxsat reasoning for the maximum weight clique problem," *Journal of Artificial Intelligence Research*, vol. 55, pp. 799–833, 2016.
[49] C.-M. Li, Y. Liu, H. Jiang, F. Manyà, and Y. Li, "A new upper bound for the maximum weight clique problem," *European Journal of Operational Research*, vol. 270, no. 1, pp. 66–77, 2018.
[50] E. Hebrard and G. Katsirelos, "Conflict directed clause learning for the maximum weighted clique problem," in *37th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 2018, pp. 1316–1323.
[51] W. Pullan, "Approximating the maximum vertex/edge weighted clique using local search," *Journal of Heuristics*, vol. 14, no. 2, pp. 117–134, Apr 2008.
[52] Q. Wu, J.-K. Hao, and F. Glover, "Multi-neighborhood tabu search for the maximum weight clique problem," *Annals of Operations Research*, vol. 196, no. 1, pp. 611–634, 2012.
[53] S. Cai and J. Lin, "Fast solving maximum weight clique problem in massive graphs." in *IJCAI*, 2016, pp. 568–574.
[54] Y. Wang, S. Cai, and M. Yin, "Two efficient local search algorithms for maximum weight clique problem." in *AAAI*, 2016, pp. 805–811.
[55] Y. Zhou, J.-K. Hao, and A. Goëffon, "Push: A generalized operator for the maximum vertex weight clique problem," *European Journal of Operational Research*, vol. 257, no. 1, pp. 41–54, 2017.
[56] Y. Fan, N. Li, C. Li, Z. Ma, L. J. Latecki, and K. Su, "Restart and random walk in local search for maximum vertex weight cliques with evaluations in clustering aggregation," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 622–630.
[57] B. Nogueira and R. G. Pinheiro, "A CPU-GPU local search heuristic for the maximum weight clique problem on massive graphs," *Computers & Operations Research*, vol. 90, pp. 232–248, 2018.
[58] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. ACM, 1992, pp. 144–152.
[59] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.
[60] R.-E. Fan, P.-H. Chen, and C.-J. Lin, "Working set selection using second order information for training support vector machines," *Journal of Machine Learning Research*, vol. 6, no. Dec, pp. 1889–1918, 2005.
[61] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.
[62] C.-J. Lin, R. C. Weng, and S. S. Keerthi, "Trust region newton method for logistic regression," *Journal of Machine Learning Research*, vol. 9, no. Jun, pp. 627–650, 2008.
[63] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *Journal of machine learning research*, vol. 9, no. Aug, pp. 1871–1874, 2008.
[64] D. S. Johnson and M. A. Trick, *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*. American Mathematical Society, 1996, vol. 26.
[65] K. Amunts, C. Lepage, L. Borgeat, H. Mohlberg, T. Dickscheid, M.-É. Rousseau, S. Bludau, P.-L. Bazin, L. B. Lewis, A.-M. Oros-Peusquens et al., "Bigbrain: an ultrahigh-resolution 3d human brain model," *Science*, vol. 340, no. 6139, pp. 1472–1475, 2013.
[66] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
[67] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. CRC Press, 2003.



Yuan Sun received his PhD degree in optimisation and machine learning from The University of Melbourne, Australia, in 2018; and his BSc degree in theoretical and applied mechanics from Peking University, China, in 2013. He is currently a Postdoctoral research fellow at RMIT University, working on an ARC (Australian Research Council) Discovery Project, using hybrid methods (a combination of traditional and machine learning techniques) to solve large-scale optimization problems. His research interests lie in the intersection between machine learning and optimisation.

Xiaodong Li (M03-SM07) received his B.Sc. degree from Xidian University, Xi'an, China, and Ph.D. degree in information science from University of Otago, Dunedin, New Zealand, respectively. He is a Professor with the School of Science (Computer Science and Software Engineering), RMIT University, Melbourne, Australia. His research interests include machine learning, evolutionary computation, neural networks, data analytics, multiobjective optimization, multimodal optimization, and swarm intelligence. He serves as an Associate Editor of the IEEE Transactions on Evolutionary Computation, Swarm Intelligence (Springer), and International Journal of Swarm Intelligence Research. He is a founding member of IEEE CIS Task Force on Swarm Intelligence, a vice-chair of IEEE Task Force on Multi-modal Optimization, and a former chair of IEEE CIS Task Force on Large Scale Global Optimization. He is the recipient of 2013 ACM SIGEVO Impact Award and 2017 IEEE CIS "IEEE Transactions on Evolutionary Computation Outstanding Paper Award".

Andreas Ernst completed his PhD in network optimisation at the The University of Western Australia in 1995. He spent 20 years working at CSIRO Australia on research into optimisation methods and applying operations research to a wide variety of industry problems, ranging from mining supply chains to rostering ant recreational vehicle scheduling. Andreas Ernst has been a Professor of Operations Research in the School of Mathematical Science at Monash University since 1995 and is the Director of MAXIMA, the Monash Academy for Interdisciplinary Mathematical Applications. He has published extensively on hub location problems, and on matheuristics that combine meta-heuristic optimisation methods with integer programming techniques. His current research interests also include decomposition methods and high performance parallel combinatorial optimisation algorithms.