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 belong to 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 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&BB 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 respectively. 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&BB algorithms; 2) improving heuristics and 3) designing heuristics automatically (hyper-heuristics).

Firstly, improving the performance of B&BB 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_3^4, v_3^5, v_3^6\} with total weight of 10.

A 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 \(G(V, E, W)\) denote the complementary graph of \(G(V, E, W)\), where the edge set \(E\) contains all the edges that are not in \(E\): \(\bar{E} = \{(i, j) | \forall(i, j) \notin E\}\). We use a binary string \(x\) 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

\[
\max_{x} \sum_{i=1}^{V} w_i x_i, \tag{1}
\]

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].

Search 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.

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 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

![Algorithm 1](image1.png)

![Algorithm 2](image2.png)
Thus the time complexity of sorting is $\Theta(|V| |B_s|)$. 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 $k_{1,th}$ vertex in $V_c$ ($V_c[k_1]$) with the $k_{2,th}$ 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 $k = k_2 + 1, \ldots, |B_s|$. 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 $V_c$ 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 $O(n |E| + |E| \log(|V|))$, 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 $O(|E|)$. We will use the fact that the number of comparisons required to identify the intersection between $V_c$ and $B_s$ is in $O(|V_c| + |B_s|)$. Let $\{v_{s_1}, v_{s_2}, \ldots, v_{s_{|C|}}\}$ denote the sequence of vertices added into $C$; $B_{s_i}$ denotes the neighbor set of $v_{s_i}$; and $V_{s_i}$ denotes 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 $\sum_{i=2}^{\lfloor |C|/2 \rfloor} |V_{s_i}| + |B_{s_{i-1}}|$ as adding the first vertex into $C$ takes $O(1)$ time. It is true that $|V_{s_i}| \leq B_{s_{i-1}}$ for any $i = 2, \ldots, |C|$. Thus, $\sum_{i=2}^{\lfloor |C|/2 \rfloor} (|V_{s_i}| + |B_{s_{i-1}}|) \leq 2 \sum_{i=1}^{\lfloor |C|/2 \rfloor} |B_{s_i}| \leq 2 |E| \in O(|E|)$. Note that the relaxation used here is very conservative, in the sense that $\sum_{i=1}^{\lfloor |C|/2 \rfloor} |B_{s_i}|$ 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 in sorting is in the order of $\sum_{i=1}^{\lfloor |C|/2 \rfloor} |B_{s_i}| \log(|B_{s_i}|) < \sum_{i=1}^{\lfloor |C|/2 \rfloor} |B_{s_i}| \log(|V|) = |E| \log(|V|)$. Thus the time complexity of sorting is $O(|E| \log(|V|))$.\footnote{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.}

\[ f_r(v_j) = \sum_{i=1}^{n} \frac{x_{i,j}}{r_i}, \quad (4) \]

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 instead of $v_{s_i}$, for any $i = 2, \ldots, |C|$, which can be significantly reduced. 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 $C_i$.

**Algorithm 3** RANKING-BASED MEASURE($C$, $Y$, $n$)

1: Sort the cliques in $C$ based on objective value $Y$; use $r_i$ to denote the ranking of $i_{th}$ clique $C_i$;
2: Initialize $f_r(v_j) \leftarrow 0$, for each $v_j \in V$;
3: for $i$ from 1 to $n$ do
   4:    for $j$ from 1 to $|C_i|$ do
   5:      $v \leftarrow C_i[j]$;
   6:      $f_r(v) \leftarrow f_r(v) + 1/r_i$;
6: 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 $i = 1, \ldots, n$. 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 $x_i$ 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

\[ f_r(v_j) = \sum_{i=1}^{n} \frac{x_{i,j}}{r_i}, \quad (4) \]

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 $O(n |V|)$ to $O(\sum_{i=1}^{n} |C_i|)$, 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 $i = 1, \ldots, n$. We use a binary string $x_i$ 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:

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

(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 \( O(n|V|) \). To improve the time efficiency, we simplify the calculation of Pearson correlation coefficient as follows.

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

\[ \sum_{i=1}^{n} (x_{i,j} - \bar{x}_j)^2 = \bar{x}_j(1 - \bar{x}_j)n, \]

(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 \( \sigma_{x,j} \). As \( x_{i,j} \) is binary variable, \( \sigma_{x,j} = n_1(1 - \bar{x}_j)^2 + n_0(0 - \bar{x}_j)^2 \),

(7)

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

(8)

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

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

(9)

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

\[ s_{j,1} = \sum_{1 \leq i \leq n, x_{i,j}=1} (y_i - \bar{y}), \quad s_{j,0} = \sum_{1 \leq i \leq n, x_{i,j}=0} (y_i - \bar{y}). \]

(10)

**Proof.** For simplicity, we use \( \sigma_{c,j} \) 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,

\[ \sigma_{c,j} = \sum_{i=1}^{n} (1 - \bar{x}_j)(y_i - \bar{y}) + \sum_{i=1}^{n} (0 - \bar{x}_j)(y_i - \bar{y}), \]

\[ = (1 - \bar{x}_j) \sum_{i=1}^{n} (y_i - \bar{y}) - \bar{x}_j \sum_{i=1}^{n} (y_i - \bar{y}), \]

\[ = (1 - \bar{x}_j)s_{j,1} - \bar{x}_j s_{j,0}. \]

(11)

\[ \boxed{\sigma_{c,j} = (1 - \bar{x}_j)s_{j,1} - \bar{x}_j s_{j,0}.} \]

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 \( \sigma_{x,j}, \sigma_{c,j} \) and thus our correlation-based measure can be easily computed. The total time complexity is \( O(|V| \sum_{i=1}^{n} |C_i|) \).

**Algorithm 4** CORRELATION-BASED MEASURE \((C, Y, n)\)

1. Calculate the mean objective value: \( \bar{y} = \sum_{i=1}^{n} y_i / n \);
2. Calculate the objective difference: \( y_d = \sum_{i=1}^{n} (y_i - \bar{y}) \);
3. Calculate the objective “variance”: \( \sigma_y = \sum_{i=1}^{n} (y_i - \bar{y})^2 \);
4. Initialize the mean \( \bar{x}_j \) to 0, for each \( v_j \) \( \in V \);
5. Initialize \( s_{j,1} \) to 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. \( \bar{x}_j \leftarrow \text{the index of vertex } C_i[k] \);
9. \( \bar{x}_j \leftarrow \bar{x}_j + 1 / n \);
10. \( s_{j,1} = s_{j,1} + (y_d - s_{j,1}) \);
11. for \( j \) from 1 to \( |V| \) do
12. \( s_{j,0} = \left( \bar{x}_j \right) s_{j,1} - \bar{x}_j s_{j,0} \);
13. \( f_c(v_j) = \sigma_{c,j} / \sqrt{\sigma_x s_{j,0}} \).
14. 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: \( f_w(u_i) = w_i \). 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 \( B_i := \{ v_j \mid v_j \in V \land (v_i, v_j) \in E \} \). 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: \( f_u(v_i) = w_i + \sum_{j \in B_i} w_j \). The value \( f_u(v_i) \) defines an upper bound on the weight of vertices 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: \( f_{den}(v_i) = 2|E|/(|V|(|V| − 1)) \). 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 \( f_w(u_i) = \frac{w_i}{w_{\text{max}}} \), where \( w_{\text{max}} = \max_{1 \leq i \leq |V|} w_i \), 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 \( f_i \in \mathbb{R}^n \), and class label \( c_i \in \{-1, 1\}, i = 1, \ldots, m \), the SVM classification algorithm solves the following primal optimization problem:

\[
\begin{align*}
\min_{a,b,\xi} & \quad \frac{1}{2} a^T a + r \sum_{i=1}^{m} g(\xi_i), \\
\text{s.t.} & \quad c_i(a^T \phi(f_i) + b) \geq 1 - \xi_i, \quad \forall i = 1, \ldots, m, \\
& \quad \xi_i \geq 0, \quad i = 1, \ldots, m,
\end{align*}
\]

(12) where \( \phi(f_i) \) maps the feature vector \( f_i \) into a higher-dimensional space; \( r > 0 \) is the regularization parameter; \( \xi_i, i = 1, \ldots, m \) 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 \( x \), class label vector \( y \), weight vector \( w \) 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

\[
\min_{a,b,\xi} \frac{1}{2} a^T a + r^+ \sum_{c_i=1} g(\xi_i) + r^- \sum_{c_i=-1} g(\xi_i),
\]

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:

\[
\begin{align*}
\min_{\alpha} & \quad \frac{1}{2} \alpha^T Q \alpha - e^T \alpha, \\
\text{s.t.} & \quad c^T \alpha = 0, \\
& \quad 0 \leq \alpha_i \leq r^+, \forall i, y_i = 1, \\
& \quad 0 \leq \alpha_i \leq r^-, \forall i, y_i = -1,
\end{align*}
\]

(16) where \( e = [1, \ldots, m]^T \) is the vector of all ones, \( Q \) is an \( m \times m \) positive semidefinite matrix, and \( Q_{i,j} = c_i c_j K(f_i, f_j) \), and \( K(f_i, f_j) = \phi(f_i) \phi(f_j) \) 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 \( K_{rbf}(f_i, f_j) = \exp(-\gamma \|f_i - f_j\|^2) \), where \( \gamma \) 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 \( \alpha^*_i \) and \( b^* \) are obtained, the predicted class label for a given new instance \( f \) is determined by

\[
\text{sgn} (\sum_{i=1}^m c_i \alpha^*_i K(f_i, f) + b^*)
\]

#### 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 \( \phi(f_i) = f_i \) 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 \( a_* \) and \( b_* \) are obtained, the predicted class label for a given new instance \( f \) is determined by \( \text{sgn} (a_*^T f + b_*) \).

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. A brief description of these graphs is given in Table 1. The original graph is unweighted. So when 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. 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 \( f_e \), 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., \( (a_*^T f + b_*) \). 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 \( f_{vw} \), 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 \( M_{10} \), we train a model using the other 7 medium graphs (\( M_{11-3} \) to \( M_{17} \)) as the training dataset. Note that in our MLPR model, a training instance refers

| ID | Name | \(|V|\) | \(|E|\) | \(d\) |
|----|------|--------|--------|------|
| \(M_{1e}\) | p_hat1000-1 | 1000 | 122253 | 0.2448 |
| \(M_{1e}\) | p_hat1000-2 | 1000 | 244799 | 0.4901 |
| \(M_{1e}\) | p_hat1000-3 | 1000 | 371746 | 0.7442 |
| \(M_{1e}\) | p_hat1500-1 | 1500 | 284923 | 0.2534 |
| \(M_{1e}\) | p_hat1500-2 | 1500 | 568960 | 0.5061 |
| \(M_{1e}\) | DSJC1000.5 | 1000 | 249826 | 0.5002 |
| \(M_{1e}\) | san1000 | 1000 | 250500 | 0.5015 |
| \(M_{1e}\) | hamming10-2 | 1024 | 518656 | 0.9902 |

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

4. The C++ and C source codes are publicly available online at https://github.com/yuansuny/mwc
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 \( n = 10 \sqrt{|E_f|} \) 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 \( n_f \) are the number of negative and positive instances in the training set, and \( \epsilon_m \) controls the penalty for misclassifying positive instances. We have tested multiple \( \epsilon_m \) 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., \( M_2^r \)), 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 \( M_1^r \) (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 \( a \) 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.
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. \( f < \epsilon \) or \( f < \epsilon_r \). 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 \( A-mI \) 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 \( \epsilon_m \) (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 \( \epsilon_c \) 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 \( \epsilon_c \) 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 \(\sigma_y\) 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.

<table>
<thead>
<tr>
<th>G</th>
<th>TSM</th>
<th>TSM-(f_r) ((\epsilon_r = 0.01))</th>
<th>TSM-(f_c) ((\epsilon_c = 0))</th>
<th>TSM-(ml) ((\epsilon_m = 10))</th>
<th>TSM-(ml) ((\epsilon_m = 100))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_{11}^L)</td>
<td>10119</td>
<td>10069</td>
<td>144.99</td>
<td>10286</td>
<td>82</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>7341</td>
<td>7479</td>
<td>277.10</td>
<td>8208</td>
<td>197</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>2407</td>
<td>2431</td>
<td>31.98</td>
<td>2465</td>
<td>4</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>8228</td>
<td>7805</td>
<td>231.10</td>
<td>8792</td>
<td>206</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>2402</td>
<td>2445</td>
<td>53.79</td>
<td>2590</td>
<td>47</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>34259</td>
<td>34265</td>
<td>0</td>
<td>1</td>
<td>33882</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>109191</td>
<td>4</td>
<td>109870</td>
<td>77</td>
<td>1.00</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>4812</td>
<td>4678</td>
<td>92</td>
<td>1</td>
<td>4899</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>4762</td>
<td>5306</td>
<td>168</td>
<td>1</td>
<td>5280</td>
</tr>
</tbody>
</table>

\(\bar{r}\) \(= 4.5\) \(= 2.4\) \(= 1.9\) \(= 1.5\) \(= 1.7\)

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 \(\sigma_y\) 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.

<table>
<thead>
<tr>
<th>G</th>
<th>LSCC</th>
<th>LSCC-(f_r) ((\epsilon_r = 0.01))</th>
<th>LSCC-(f_c) ((\epsilon_c = 0))</th>
<th>LSCC-(ml) ((\epsilon_m = 10))</th>
<th>LSCC-(ml) ((\epsilon_m = 100))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_{11}^L)</td>
<td>10320</td>
<td>10318</td>
<td>9</td>
<td>0.93</td>
<td>10320</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>9251</td>
<td>9252</td>
<td>13</td>
<td>1.00</td>
<td>9119</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>2466</td>
<td>2466</td>
<td>0</td>
<td>0.98</td>
<td>2466</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>10905</td>
<td>10917</td>
<td>33</td>
<td>1.00</td>
<td>10887</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>2792</td>
<td>2791</td>
<td>2</td>
<td>0.79</td>
<td>2781</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>34230</td>
<td>34228</td>
<td>6</td>
<td>1</td>
<td>33891</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>111070</td>
<td>111060</td>
<td>14</td>
<td>1.00</td>
<td>110130</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>5129</td>
<td>5129</td>
<td>0</td>
<td>1.00</td>
<td>5129</td>
</tr>
<tr>
<td>(M_{11}^L)</td>
<td>7778</td>
<td>7766</td>
<td>60</td>
<td>1.00</td>
<td>7789</td>
</tr>
</tbody>
</table>

\(\bar{r}\) \(= 3.3\) \(= 1.2\) \(= 2.6\) \(= 2.2\) \(= 1.4\)
TABLE 7: The results of WLMC, WLMC-\( f_r \), WLMC-\( fc \) and WLMC-\( ml \) when used to solve the hard instances; \( \bar{y} \) and \( \sigma_y \) 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.

<table>
<thead>
<tr>
<th>G</th>
<th>WLMC</th>
<th>WLMC-( f_c (\epsilon_r = 0.01) )</th>
<th>WLMC-( fc (\epsilon_r = 0) )</th>
<th>WLMC-( ml (\epsilon_m = 10) )</th>
<th>WLMC-( ml (\epsilon_m = 100) )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \bar{y} )</td>
<td>( \sigma_y )</td>
<td>( \bar{y} )</td>
<td>( \sigma_y )</td>
<td>( \bar{y} )</td>
</tr>
<tr>
<td>( M_1^c )</td>
<td>9846</td>
<td>0</td>
<td>9791</td>
<td>48</td>
<td>0.93</td>
</tr>
<tr>
<td>( M_2^c )</td>
<td>7317</td>
<td>0</td>
<td>7182</td>
<td>226</td>
<td>1.00</td>
</tr>
<tr>
<td>( M_3^c )</td>
<td>2360</td>
<td>0</td>
<td>2396</td>
<td>42</td>
<td>0.98</td>
</tr>
<tr>
<td>( M_4^c )</td>
<td>7738</td>
<td>0</td>
<td>7739</td>
<td>221</td>
<td>1.00</td>
</tr>
<tr>
<td>( M_5^c )</td>
<td>2383</td>
<td>0</td>
<td>2452</td>
<td>59</td>
<td>0.79</td>
</tr>
<tr>
<td>( M_6^c )</td>
<td>34265</td>
<td>0</td>
<td>3465</td>
<td>34</td>
<td>1.00</td>
</tr>
<tr>
<td>( M_7^c )</td>
<td>109789</td>
<td>16</td>
<td>109850</td>
<td>49</td>
<td>1.00</td>
</tr>
<tr>
<td>( M_8^c )</td>
<td>4738</td>
<td>0</td>
<td>4386</td>
<td>133</td>
<td>1.00</td>
</tr>
<tr>
<td>( M_9^c )</td>
<td>4760</td>
<td>0</td>
<td>5214</td>
<td>138</td>
<td>1.00</td>
</tr>
</tbody>
</table>

TABLE 8: The results of FastWClq, FastWClq-\( f_r \), FastWClq-\( fc \) and FastWClq-\( ml \) when used to solve the hard instances; \( \bar{y} \) and \( \sigma_y \) 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.

<table>
<thead>
<tr>
<th>G</th>
<th>FastWClq</th>
<th>FastWClq-( f_r (\epsilon_r = 0.01) )</th>
<th>FastWClq-( fc (\epsilon_r = 0) )</th>
<th>FastWClq-( ml (\epsilon_m = 10) )</th>
<th>FastWClq-( ml (\epsilon_m = 100) )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \bar{y} )</td>
<td>( \sigma_y )</td>
<td>( \bar{y} )</td>
<td>( \sigma_y )</td>
<td>( \bar{y} )</td>
</tr>
<tr>
<td>( L_1^c )</td>
<td>25293</td>
<td>0</td>
<td>31807</td>
<td>341</td>
<td>0.02</td>
</tr>
<tr>
<td>( L_2^c )</td>
<td>22332</td>
<td>0</td>
<td>26331</td>
<td>826</td>
<td>0.03</td>
</tr>
<tr>
<td>( L_3^c )</td>
<td>28044</td>
<td>0</td>
<td>31446</td>
<td>1065</td>
<td>0.02</td>
</tr>
<tr>
<td>( L_4^c )</td>
<td>20819</td>
<td>0</td>
<td>29548</td>
<td>600</td>
<td>0.02</td>
</tr>
<tr>
<td>( L_5^c )</td>
<td>29398</td>
<td>0</td>
<td>32659</td>
<td>378</td>
<td>0.02</td>
</tr>
<tr>
<td>( L_6^c )</td>
<td>26557</td>
<td>0</td>
<td>34027</td>
<td>663</td>
<td>0.02</td>
</tr>
<tr>
<td>( L_7^c )</td>
<td>24650</td>
<td>0</td>
<td>30753</td>
<td>462</td>
<td>0.03</td>
</tr>
<tr>
<td>( L_8^c )</td>
<td>43356</td>
<td>0</td>
<td>39221</td>
<td>5755</td>
<td>0.02</td>
</tr>
<tr>
<td>( L_9^c )</td>
<td>23167</td>
<td>0</td>
<td>23247</td>
<td>50</td>
<td>0.02</td>
</tr>
<tr>
<td>( L_{10}^c )</td>
<td>24991</td>
<td>0</td>
<td>27706</td>
<td>343</td>
<td>0.02</td>
</tr>
<tr>
<td>( L_{11}^c )</td>
<td>25205</td>
<td>0</td>
<td>25309</td>
<td>162</td>
<td>0.02</td>
</tr>
</tbody>
</table>

\( \bar{r} \) 4.2 2.0 1.8 1.6 1.9
Fig. 3: A comparison between MLPR_{small} (trained on small graphs), MLPR_{large} (trained on large graphs) and MLPR_{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 ($\bar{g}$) within the cutoff time (1000 seconds). For each method we sort $\bar{g}$ 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 $\epsilon_m$, 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 ($M^{\text{tr}}$) 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_{small} against MLPR_{large} which is trained on the large easy graphs $L^{te}$ as well as MLPR_{none} (without any problem reduction), when incorporated with the 4 solution algorithms to solve the 11 large hard graphs $L^{te}$. The parameter $\epsilon_m$ is set to 10 for both of the MLPR_{small} and MLPR_{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.

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 medium 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 $\epsilon_r$, $\epsilon_c$ and $\epsilon_m$.

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-
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} 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 over all 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.

### Table 9: The results of TSM, TSM-f-O, TSM-f_-O and TSM-ml-O when used to solve the hard instances; \(\bar{y}\) and \(\sigma_y\) denote the mean and standard deviation of best objective values generated in 25 independent runs within the cutoff time (1000 seconds).

<table>
<thead>
<tr>
<th>G</th>
<th>TSM</th>
<th>TSM-f-O</th>
<th>TSM-f_-O</th>
<th>TSM-ml-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\bar{y})</td>
<td>10219</td>
<td>73</td>
<td>10273</td>
<td>23</td>
</tr>
<tr>
<td>(\sigma_y)</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 10: The results of WLMC, WLMC-f-O, WLMC-f_-O and WLMC-ml-O when used to solve the hard instances; \(\bar{y}\) and \(\sigma_y\) denote the mean and standard deviation of best objective values generated in 25 independent runs within the cutoff time (1000 seconds).

<table>
<thead>
<tr>
<th>G</th>
<th>WLMC</th>
<th>WLMC-f-O</th>
<th>WLMC-f_-O</th>
<th>WLMC-ml-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\bar{y})</td>
<td>9846</td>
<td>115</td>
<td>10244</td>
<td>32</td>
</tr>
<tr>
<td>(\sigma_y)</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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


