# New Approximation Algorithms for Minimum Weighted Edge Cover

S M Ferdous[*] Alex Pothen[†] Arif Khan[‡]

**Abstract**

We describe two new 3/2-approximation algorithms and a new 2-approximation algorithm for the minimum weight edge cover problem in graphs. We show that one of the 3/2-approximation algorithms, the Dual Cover algorithm, computes the lowest weight edge cover relative to previously known algorithms as well as the new algorithms reported here. The Dual Cover algorithm can also be implemented to be faster than the other 3/2-approximation algorithms on serial computers. Many of these algorithms can be extended to solve the $b$-Edge Cover problem as well. We show the relation of these algorithms to the $K$-Nearest Neighbor graph construction in semi-supervised learning and other applications.

## 1 Introduction

An Edge Cover in a graph is a subgraph such that every vertex has at least one edge incident on it in the subgraph. We consider the problem of computing an Edge Cover of minimum weight in edge-weighted graphs, and design two new 3/2-approximation algorithms and a new 2-approximation algorithm for it. One of the 3/2-approximation algorithms, the Dual Cover algorithm is obtained from a primal-dual linear programming formulation of the problem. The other 3/2-approximation algorithm is derived from a lazy implementation of the Greedy algorithm for this problem. The new 2-approximation algorithm is related to the widely-used $K$-Nearest Neighbor graph construction used in semi-supervised machine learning and other applications. Here we show that the $K$-Nearest Neighbor graph construction process leads to a 2-approximation algorithm for the $b$-Edge Cover problem, which is a generalization of the Edge Cover problem. (These problems are formally defined in the next Section.)

The Edge Cover problem is applied to covering problems such as sensor placement, while the $b$-Edge Cover problem is used when redundancy is necessary for reliability. The $b$-Edge Cover problem has been applied in communication networks [17] and in adaptive anonymity problems [15].

The $K$-Nearest Neighbor graph is used to sparsify data sets, which is an important step in graph-based semi-supervised machine learning. Here one has a few labeled items, many unlabeled items, and a measure of similarity between pairs of items; we are required to label the remaining items. A popular approach for classification is to generate a similarity graph between the items to represent both the labeled and unlabeled data, and then to use a label propagation algorithm to classify the unlabeled items [23]. In this approach one builds a complete graph out of the dataset and then sparsifies this graph by computing a $K$-Nearest Neighbor graph [22]. This sparsification leads to efficient algorithms, but also helps remove noise which can affect label propagation [11]. In this paper, we show that the well-known Nearest Neighbor graph construction computes an approximate minimum-weight Edge Cover with approximation ratio 2. We also show that the $K$-Nearest Neighbor graph may have a relatively large number of redundant edges which could be removed to reduce the weight. This graph is also known to have skewed degree distributions [11], which could be avoided by other algorithms for $b$-Edge Covers. Since the approximation ratio of $K$-Nearest Neighbor algorithm is 2, a better choice for sparsification could be other edge cover algorithms with an approximation ratio of 3/2; algorithms that lead to more equitable degree distributions could also lead to better classification results. We will explore this idea in future work.

Our contributions in this paper are as follows:

- We improve the performance of the Greedy algorithm for minimum weight edge cover problem by lazy evaluation, as in the Lazy Greedy algorithm.
- We develop a novel primal-dual algorithm for the minimum weight edge cover problem that has approximation ratio 3/2.
- We show that the $K$-Nearest Neighbor approach for edge cover is a 2-approximation algorithm for the edge weight. We also show that practically the weight of the edge cover could be reduced by removing redundant edges. We are surprised that these observations have not been made earlier

[*] Computer Science Department, Purdue University. West Lafayette IN 47907 USA. sferdou@purdue.edu

[†] Computer Science Department, Purdue University. West Lafayette IN 47907. apothen@purdue.edu

[‡] Data Sciences, Pacific Northwest National Lab. Richland WA 99352 USA. ariful.khan@pnnl.gov

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

given the widespread use of this graph construction in Machine Learning, but could not find these results in a literature search.

- We also conducted experiments on eleven different graphs with varying sizes, and found that the primal-dual method is the best performing among all the 3/2 edge cover algorithms.

The rest of the paper is organized as follows. We provide the necessary background on edge covers in Section 2. We discuss several 3/2-approximation algorithms including the new Dual Cover algorithm in Section 3. In Section 4, we discuss the Nearest Neighbor approach in detail along with two earlier algorithms. We discuss the issues of redundant edges in Section 5. In Section 6, we experiment and compare the performance of the new algorithms and earlier approximation algorithms. We summarize the state of affairs for Edge Cover and $b$-Edge Cover problems in Section 7.

## 2 Background

Throughout this paper, we denote by $G(V, E, W)$ a graph $G$ with vertex set $V$, edge set $E$, and edge weights $W$. An Edge Cover in a graph is a subgraph such that every vertex has at least one edge incident on it in the subgraph. If the edges are weighted, then an edge cover that minimizes the sum of weights of its edges is a minimum weight edge cover. We can extend these definitions to $b$-Edge Cover, where each vertex $v$ must be the endpoint of at least $b(v)$ edges in the cover, where the values of $b(v)$ are given.

The minimum weighted edge cover is related to the better-known maximum weighted matching problem, where the objective is to maximize the sum of weights of a subset of edges $M$ such that no two edges in $M$ share a common endpoint. (Such edges are said to be independent.) The minimum weight edge cover problem can be transformed to a maximum weighted perfect matching, as has been described by Schrijver [21]. Here one makes two copies of the graph, and then joins corresponding vertices in the two graphs with linking edges. Each linking edge is given twice the weight of edge of minimum weight edge incident on that vertex in the original graph. The complexity of the best known [6] algorithm for computing a minimum weight perfect matching with real weights is $O(|V||E| + |V|^2 log|E|))$, which is due to Gabow [8]. As Schrijver's transformation does not asymptotically increase the number of edges or vertices, the best known complexity of the optimal edge cover is the same. The minimum weighted $b$-Edge Cover problem can be obtained as the complement of a $b'$-matching of maximum weight, where $b'(v) = \deg(v) - b(v)$ [21]. Here $\deg(v)$ is the degree of the vertex $v$. The complement can be computed in $O(|E|)$ time. For exact $b'$-matching the best known algorithm is due to Anstee, with time complexity $\min\{O(|V|^2|E| + |V|log\beta\ (|E| + |V|log|V)), O(|V|^2 log|V(|E| + |V|log|V|))\}$ [1,21].

In the set cover problem we are given a collection of subsets of a set (universe), and the goal is to choose a sub-collection of the subsets to cover every element in the set. If there is a weight associated with each subset, the problem is to find a sub-collection such that the sum of the weights of the sub-collection is minimum. This problem is NP-hard [13]. There are two well known approximation solutions for solving set cover. One is to repeatedly choose a subset with the minimum cost and cover ratio, and then delete the elements of the chosen set from the universe. This Greedy algorithm is due to Johnson and Chvatal [4,12], and it has approximation ratio $H_k$, the $k$-th harmonic number, where $k$ is the largest size of a subset. The other algorithm is a primal-dual algorithm due to Hochbaum [9], and provides $f-$approximation, where $f$ is the maximum frequency of an element in the subsets. The latter algorithm is important because it gives a constant 2-approximation algorithm for the vertex cover problem. An edge cover is a specific case of a set cover where each subset has exactly two elements ($k = 2$). The Greedy algorithm of Chvatal achieves the approximation ratio of 3/2 for this problem, and we will discuss it in detail in Section 3. The primal-dual algorithm of Hochbaum is a $\Delta$-approximation algorithm for edge cover, where $\Delta$ is the maximum degree of the graph.

Recently, a number of approximation algorithms have been developed for the minimum weighted $b$-Edge Cover. Khan and Pothen [14] have described a Locally Subdominant Edge algorithm (LSE). In [16], the current authors have described two different 2-approximation algorithms for the problem, static LSE (S-LSE) and Matching Complement Edge cover (MCE). We will discuss these algorithms in Section 4. In [10], Huang and Pettie developed a $(1 + \epsilon)$-approximation algorithm for the weighted b-edge cover, for any $\epsilon > 0$. The complexity of the algorithm is $O(m\epsilon^{-1} \log W)$, where $W$ is the maximum weight of any edge. The authors showed a technique to convert the runtime into $O(m\epsilon^{-1} \log(\epsilon^{-1}))$. This scaling algorithm requires blossom manipulation and dual weights adjustment. We have implemented $(1-\epsilon)$-approximation algorithms based on scaling ideas for vertex weighted matching, but they are slower and practically obtain worse approximations than a 2/3-approximation algorithm [5]. Since the edge cover algorithms are also based on the scaling idea, it is not clear how beneficial it would be to implement this algorithm. On the other hand, our

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

2- and 3/2- approximation algorithms are easily implemented, since no blossoms need to be processed, and also provide near-optimum edge weights. This is why we did not implement the $(1+\epsilon)$-approximation algorithm.

## 3 3/2-Approximation Algorithms

In this section we discuss four 3/2-approximation algorithms for the minimum weighted Edge Cover problem. Two of these algorithms are the classical Greedy algorithm, and a variant called the Locally Subdominant Edge algorithm, LSE, which we have described in earlier work. The other two algorithms, the Lazy Greedy algorithm and a primal-dual algorithm, Dual Cover, are new.

Let us first describe the primal and dual LP formulations of the minimum weighted Edge Cover problem. Consider the graph $G(V, E, W)$, and define a binary variable $x_e$ for each $e \in E$. Denote the weight of an edge $e$ by $w_e$, and the set of edges adjacent to a vertex $v$ by $\delta(v)$. The integer linear program (ILP) of the minimum weighted edge cover problem is as follows.

$$\min \sum_{e\in E} w_e x_e, \quad \text{subject to} \sum_{e\in\delta(v)} x_e \geq 1, \forall v \in V,$$
$$x_e \in \{0,1\}, \forall e \in E. \tag{3.1}$$

If the variable $x_e$ is relaxed to $0 \leq x_e \leq 1$, the resulting formulation is the LP relaxation of the original ILP. Let $OPT$ denote the optimum value of minimum weighted edge cover defined by the ILP, and $OPT_{LP}$ be the optimum attained by the LP relaxation; then $OPT_{LP} \leq OPT$ since the the feasible region of the LP contains that of the ILP. We now consider the dual problem of the LP. We define a dual variable $y_v$ for each constraint on a vertex $v$ in the LP.

$$\max \sum_{v\in V} y_v, \quad \text{subject to } y_i + y_j \leq w_e, \forall e(i,j) \in E,$$
$$y_v \geq 0, \ \forall v \in V. \tag{3.2}$$

From the duality theory of LPs, any feasible solution of the dual problem provides a lower bound for the original LP. Hence $FEAS_{dual} \leq OPT_{LP} \leq OPT_{ILP}$, where $FEAS_{dual}$ denotes the objective value of any feasible solution of the dual problem.

**3.1 The Greedy Algorithm.** Since an Edge Cover is a special case of the set cover, we can apply the Greedy set cover algorithm [4] to compute an Edge Cover. We define the *effective weight* of an edge as the weight of the edge divided by the number of its uncovered endpoints. The Greedy algorithm for minimum weighted edge cover works as follows. Initially, no vertices are covered, and the effective weights of all the edges are half of the edge weights. In each iteration, there are three possibilities for each edge: i) none of its endpoints is covered, and there is no change in its effective weight, ii) one of the endpoints is covered, and its effective weight doubles, or iii) both endpoints are covered, its effective weight becomes infinite, and the edge is marked as deleted. After the effective weights of all edges are updated, we choose an edge with minimum effective weight, add that edge to the cover, and mark it as deleted. The algorithm iterates until all vertices are covered. This produces an edge cover whose weight is at most 3/2 of the minimum weight. The worst case time complexity of the Greedy algorithm is $O(|E|log|E|)$.

Using the primal dual LP formulation stated in Equations 3.1 and 3.2, we will prove the 3/2-approximation ratio for the Greedy algorithm. This proof is important because it lays the foundation for the analysis of the Dual Cover algorithm that we will see later.

Lemma 3.1. *The approximation ratio of the* Greedy *algorithm is* 3/2.

*Proof.* We define a variable, price, at each vertex of the graph. When the Greedy algorithm chooses an edge in the cover we can consider that it assigns prices on the two end-points of the vertex. The value of price should be set such that the prices of the endpoints pay for the weight of the edges in the cover. When an edge $(i, j)$ is added to the cover in the Greedy algorithm, we could have two cases: i) The edge covers both of its endpoints. In this case, the price of each end-point is the effective weight of the edge (i.e., half of the actual weight). Or ii) only one endpoint of $(i, j)$, say $i$, was covered earlier; then the price of $i$ was set in a previous iteration. Since we have selected the edge $(i, j)$ to add to the cover, we assign the weight of the edge to be the price of $j$. If we assign the price of each vertex in this way, then the sum of weights of the edges in the cover computed by the Greedy algorithm would be equal to the sum of the price of the vertices.

The pricing mechanism assigns a value on each vertex, but can we derive $y_v$ values feasible for the dual LP from them? Let us consider the constraints assuming $y_v$ = price$(v)$. First consider the edges which are in the cover. Again we have two cases to consider: i) The edge $(i, j)$ covers two endpoints. In this case, price$(i)$ = price$(j)$ = $w_{(i,j)}/2$, resulting in price$(i)$ + price$(j)$ = $w_{(i,j)}$. So for these edges the constraints are satisfied, and price$(v)$ is equal to $y_v$. ii) Now consider those edges $(i, j)$ that cover only one endpoint, say $i$. From the assignment of the price

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

we know that $price(j) = w_{(i,j)}$. Since all the prices are positive, this tells us that the constraint of $(i,j)$ is violated. We now show that $price(i) \leq w_{(i,j)}/2$. When $i$ was covered by some edge other than $(i,j)$, the effective weight of $(i,j)$ was $w_{(i,j)}/2$. So the selected edge must have effective weight $w_{(i,j)}/2$, which implies that $\text{price}(i) + \text{price}(j) \leq 3/2 * w_{(i,j)}$.

Now consider an edge $(i,j)$ which is not included in the GREEDY edge cover. Suppose vertex $i$ is covered before vertex $j$. When $i$ is covered, the effective weight of the edge $(i,j)$ is $w_{(i,j)}/2$ since both vertices $i$ and $j$ were uncovered prior to that step. As the vertex $i$ is being covered by some edge $e'$ other than $(i,j)$, and the greedy algorithm chooses an edge of least effective weight, this weight is less than or equal to $w_{(i,j)}/2$. Hence $\text{price}(i)$ is less than or equal to this value. Now when the vertex $j$ is covered, the effective weight of the edge $(i,j)$ is $w_{(i,j)}$. Following the argument as for vertex $i$, we find that $\text{price}(j) \leq w_{(i,j)}$. Hence we have that $\text{price}(i) + \text{price}(j) \leq 3/2 * w_{(i,j)}$.

Now if we set $y_v = 2/3 * \text{price}(v)$, then the dual problem is feasible. We say that 3/2 is a shrinking factor. We can write

$$\begin{aligned} OPT_{\text{GREEDY}} &= \sum_v \text{price}(v) = 3/2 * \sum_v y_v \\ &\leq 3/2 * OPT_{LP} \leq 3/2 * OPT_{ILP}. \end{aligned}$$

■

### 3.2 The LAZY GREEDY Algorithm.

The effective weight of an edge can only increase during the GREEDY algorithm, and we exploit this observation to design a faster variant. The idea is to delay the updating of effective weights of most edges, which is the most expensive step in the algorithm, until it is needed. If the edges are maintained in non-increasing order of weights in a heap, then we update the effective weight of only the top edge; if its effective weight is no larger than the effective weight of the next edge in the heap, then we could add the top edge to the cover as well. A similar property of greedy algorithms has been exploited in submodular optimization, where this algorithm is known as the LAZY GREEDY algorithm [18].

The pseudocode of the LAZY GREEDY algorithm is presented in Algorithm 1. The LAZY GREEDY algorithm maintains a minimum priority queue of the edges prioritized by their effective weights. The algorithm works as follows. Initially all the vertices are uncovered. We create a priority queue of the edges ordered by their effective weights, $PrQ$. An edge data structure in the priority queue has three fields: the endpoints of the edge, $u$ and $v$, and its effective weight $w$. The priority queue has four operations. The makeHeap(Edges) operation creates a priority Queue in time linear in the number of edges. The deQueue() operation deletes and returns an edge with the minimum effective weight in time logarithmic in the size of queue. The enQueue(Edge e) operation inserts an edge e into the priority queue according to its effective weight. The front() operation returns the current top element in constant time without popping the element itself.

At each iteration, the algorithm dequeues the top element, *top*, from the queue, and updates its effective weight to *top.w*. Let the new top element in $PrQ$ be *newTop*, with effective weight (not necessarily updated) *newTop.w*. If *top.w* is less than or equal to *newTop.w*, then we can add *top* to the edge cover, and increment the covered edge counter for its endpoints. Otherwise, if *top.w* is not infinite, we enQueue(*top*) to the priority queue. Finally, if *top.w* is infinite, we delete the edge. We continue iterating until all the vertices are covered. The cover output by this algorithm may have some redundant edges which could be removed to reduce the weight. We will discuss the algorithm for removing redundant edges in Section 5.

**Algorithm 1** LAZY GREEDY$(G(V,E,W))$

```
$C = \emptyset$;                                   ▷ the edge cover
$c$ = Array of size $|V|$ initialized to 0;  ▷ indicates if
  a vertex is covered
PrQ = makeHeap(E)  ▷ Create a min heap from $E$
while there exists an uncovered vertex do
    top = PrQ.deQueue()
    Update effective weight of top edge,
      assign to top.w
    if top.w $< \infty$ then
        newTop = PrQ.front()
        if top.w $\leq$ newTop.w then
            $C = C \cup$ top
            Increment $c(u)$ and $c(v)$ by 1
        else
            PrQ.enQueue(top)
$C$ =REMOVE_REDUNDANT_EDGE(C)
return $C$
```

Next we compute the approximation ratio of the algorithm.

LEMMA 3.2. *The approximation ratio of the* LAZY GREEDY *algorithm is* 3/2.

*Proof.* The invariant in the GREEDY algorithm is that at every iteration we select an edge which has minimum effective weight over all edges. Now consider an edge $x$ chosen by the LAZY GREEDY algorithm in some

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

iteration. According to the algorithm the updated effective weight of $x$, denoted by $x.w$, is less than or equal to the effective weight of the current top element of the priority queue. Since, the effective weight of an edge can only increase, then $x$ has the minimum effective weight over all edges in the queue. So the invariant in the Greedy algorithm is satisfied in the Lazy Greedy algorithm, resulting in the 3/2-approximation ratio. ■

The runtime for Lazy Greedy is also $O(|E|log|E|)$, because over the course of the algorithm, each edge will incur at most two deQueue() operations and one enQueue() operation, and each such operation costs $O(log|E|)$. The efficiency of the Lazy Greedy algorithm comes from the fact that in each iteration we do not need to update effective weights of the edges adjacent to the selected edge. But the price we pay is the logarithmic-cost enQueue() and deQueue() operations. We will see in Section 6 that the average number of queue accesses in the Lazy Greedy algorithm is low resulting in a faster algorithm over the Greedy algorithm.

**3.3 The** LSE **Algorithm.** This algorithm [14] finds a set of locally subdominant edges and adds them to the cover at each iteration. An edge is *locally subdominant* if its effective weight is smaller than the effective weights of its neighboring edges (i.e., other edges with which it shares an endpoint). It can be easily shown that the Greedy and Lazy Greedy algorithms add locally subdominant edges w.r.t the effective weights at each step. The approximation ratio of LSE is 3/2.

**3.4 The** Dual Cover **Algorithm.** The proof of the approximation ratio of the Greedy algorithm presented in Section 3.1 provides an algorithm for the edge cover problem. The algorithm works iteratively, and each iteration consists of two phases: the dual weight assignment phase and the primal covering phase. At the start of each iteration we initialize the price of each uncovered vertex to $\infty$. In the assignment phase, the effective weight of each edge is computed. Each edge updates the price of its uncovered end-points, to be the minimum of its effective weight and the current price of that vertex. After this phase, each uncovered vertex holds the minimum effective weight of its incident edges. The algorithm for the assignment phase is presented in 2.

The second phase is the covering phase. In this phase, we scan through all the edges and add the edges in the output that satisfy any of the two conditions.

i The edge covers both of its endpoints. The prices on the two endpoints are equal and they sum up to the weight of the edge.

ii The edge covers only one endpoint. The price of the uncovered endpoint is the weight of the edge, and the two prices sum to at most 3/2 times the original weight of the edge.

**Algorithm 2** Dual Assignment($G(V, E, W)$, price)

```
for each $v \in V$ do
    if $v$ is uncovered then price(v) = $\infty$
for each $(u, v) \in E$ do
    if ($u$ and $v$ are both uncovered) then
        price($u$) = $MIN$(price($u$), $W(u, v)/2$)
        price($v$) = $MIN$(price($v$), $W(u, v)/2$)
    else if (only $u$ is uncovered) then
        price($u$) = $MIN$(price($u$), $W(u, v)$)
    else if (only $v$ is uncovered) then
        price($v$) = $MIN$(price($v$), $W(u, v)$)
```

The algorithm for the primal covering phase is presented in Algorithm 3. The overall algorithm is described in pseudocode in Algorithm 4.

**Algorithm 3** Primal Cover($G(V, E, W)$,price,$C$,$c$)

```
for each $(u, v) \in E$ do
    if $u$ and $v$ are both uncovered and condition (i)
      is satisfied then
        $C = C \cup (u, v)$
        Increment $c(u)$ and $c(v)$ by 1
    else if only $u$ or $v$ is uncovered and condition
      (ii) is satisfied then
        $C = C \cup (u, v)$
        Increment $c(u)$ and $c(v)$ by 1
    else if $u$ and $v$ are both covered then
        Mark $(u, v)$ as deleted
```

**Algorithm 4** Dual Cover($G(V, E, W)$)

```
$C = \emptyset$
$c$ = Array of size $|V|$ initialized 0
price = array of size $|V|$
while there exists an uncovered vertex do
    Call Dual Assignment($G(V, E, W)$, price)
    Call Primal Cover ($G(V, E, W)$, price, $C$, $c$)
$C$ = Remove_Redundant_Edge(C)
return $C$
```

Now we prove the correctness and approximation ratio of the Dual Cover algorithm.

Lemma 3.3. *The* Dual Cover *algorithm terminates.*

*Proof.* Suppose the algorithm does not terminate. Then during some iteration of the algorithm, it fails to

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

cover any uncovered vertices. We assume without loss of generality that the graph is connected. Let the uncovered vertices be $L$. We create a subgraph $G_L$ induced by the edges that are adjacent to at least one vertex in $L$. Now let $e_l = (u_l, v_l)$ be an edge with the lowest effective weight in $G_L$. If $e_l$ covers both of its endpoints, then in the Dual Assignment phase, the prices of $u_l$ and $v_l$ must be $price(u_l) = price(v_l) = weight(e_l)/2$. So this edge fulfills condition (i). If $e_l$ covers only one endpoint, say $v_l$, then $v_l \notin L$. Now price($v_l$) $\leq weight(e_l)/2$, since when $v_l$ was covered the two endpoints of edge $e_l$ were available to be added to the cover. Despite this the assignment phase did not assign $weight(e_l)/2$ to price($v_l$). So price($v_l$) $\leq weight(e_l)/2$. Now the assignment phase would have assigned price($u_l$) $= weight(e_l)$ to satisfy condition (ii), and the vertex $u_l$ would have been added to the cover. This contradiction completes the proof. ■

Another way of looking at the Dual Cover algorithm is in terms of locally sub-dominant edges. The edges chosen at every iteration are locally sub-dominant. Many edges could become sub-dominant at an iteration, and the assignment phase sets up the price to detect locally sub-dominant edges in the covering phase. The efficiency of this algorithm comes from the fraction of vertices covered through the sub-dominant edges at every iteration. As we will show in the experimental section the rate of convergence to full edge cover is fast, although the worst-case complexity of this algorithm could be $O(|C||E|)$, where $|C|$ is the number of edges in the cover.

Lemma 3.4. *The approximation ratio of the* Dual Cover *algorithm is* $3/2$.

*Proof.* First note that the weight of the edge cover is fully paid by the price of each vertex, which means that the sum of the prices equals the sum of the weights of the selected edges. Also note that for the edges in the cover the shrinking factor is at most $3/2$. Now we consider the edges that are not in the edge cover. Let $(u, v)$ be such an edge, and let $u$ be covered before $v$. When $u$ was covered both endpoints of $(u, v)$ were available. Hence the price($u$) $\leq w_{(u,v)}/2$. Now when $v$ was covered by some edge other than $(u, v)$, price($v$) $\leq w_{(u,v)}$. This implies that for the edges that are not in the cover, the shrinking factor is also $3/2$. Now let the cover be denoted by $C$. We have

$$\sum_{e\in C} w_e = \sum_{v\in V} \text{price}(v) \leq 3/2 * \sum_{v\in V} y_v$$
$$\leq 3/2 * OPT_{LP} \ \leq \ 3/2 * OPT_{ILP}.$$

■

**3.5 Extension to** $b$-**Edge Cover.** In the $b$-Edge Cover problem each vertex $v$ needs to be covered by at least $b_v$ edges. The Greedy, the LSE and the Lazy Greedy algorithms can be extended to handle this constraint. To incorporate the $b_v$ constraint, we extend the definition of covering/saturation of a vertex, $v$. A vertex is covered/saturated when it is covered by at least $b_v$ edges. It is not difficult to show that the extended algorithms also match the approximation ratio of $3/2$. In recent work, we have extended the Dual Cover algorithm to the $b$-Edge Cover problem, and we will report on this in our future work.

## 4 2-Approximation Algorithms

We know of two different 2-approximation algorithms, S-LSE and MCE, that have been discussed previously for the minimum weighted edge cover problem [16]. In this section we show that the widely-used $k$-nearest neighbor algorithm is also a 2-approximation algorithm, and then briefly discuss the two earlier algorithms.

**4.1** Nearest Neighbor **Algorithm.** The nearest neighbor of a vertex $v$ in a graph is the edge of minimum weight adjacent to it. A simple approach to obtain an edge cover is the following: For each vertex $v$, insert the edge that $v$ forms with its nearest neighbor into the cover. (We also call this a lightest edge incident on $v$.)

The worst-case runtime of the Nearest Neighbor algorithm is $O(|E|)$. This algorithm has many redundant edges that it includes in the cover, and in a practical algorithm such edges would need to be removed. Nevertheless, even without the removal of such edges, we prove that the Nearest Neighbor algorithm produces an edge cover whose total weight is at most twice that of the minimum weight.

Lemma 4.1. *The approximation ratio of the* Nearest Neighbor *algorithm is 2.*

*Proof.* Let the optimal edge cover be denoted by $OPT$. Let $o_i = (u, v)$ be an edge in the optimal cover. Suppose that the edge $o_i$ is not included in the cover computed by the Nearest Neighbor algorithm. Let a lightest edge incident on $u$ ($v$) be denoted by $e_u$ ($e_v$). If $e_u$ and $e_v$ are distinct, then both these edges (or two edges of equal weight) are included in the Nearest Neighbor edge cover. Since the edge $o_i$ is not included in the Nearest Neighbor cover, we have $w(e_u) \leq w(o_i)$, and $w(e_v) \leq w(o_i)$. So, in the worst case, for each edge in the optimal cover, we may have two edges in the Nearest Neighbor cover, whose weights are at most the weight of the edge in the optimal cover. ■

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

### 4.2 Extension to $b$-Edge Cover.

To extend the Nearest Neighbor algorithm to the $b$-edge cover, instead of choosing a nearest neighbor, we will add $b(v)$ nearest neighbors of a vertex $v$ into the cover. The proof that this is a 2-approximation algorithm can be obtained by the same argument as given above.

There are multiple ways of implementing the $b$-Nearest Neighbor algorithm, of which we mention two ways. The first is to sort all the edges incident on each vertex $v$, and then to add the lightest $b(v)$ edges to the cover. The complexity of this approach is $O(|E| \log \Delta)$, where $\Delta$ is the maximum degree of a vertex. The second approach maintains a min-heap for each vertex. The heap for a vertex $v$ contains the edges adjacent to it, with the edge weight as key. The complexity of creating a heap for a vertex $v$ is $O(\deg(v))$. Then for each vertex $v$, we query the heap $b(v)$ times to get that many lightest edges. This implementation has runtime $O(|V|\beta \log \Delta + |E|)$, where $\beta = \max_v b(v)$. The second version is asymptotically faster than the first version as long as $|E| = \Omega(|V|\beta)$. We have used the second approach in our implementation.

### 4.3 S-LSE Algorithm.

The S-LSE algorithm is described in [16], and it is a modification of the LSE algorithm in which the algorithm works with static edge weights instead of dynamically updating effective weights. At each step, the algorithm identifies a set of edges whose weights are minimum among their neighboring edges. Such edges are added to the cover and then marked as deleted from the graph, and the $b(.)$ values of their endpoints are updated. Edges with both endpoints satisfying their $b(.)$ constraints are also deleted. The algorithm then iterates until the $b$-edge cover is computed, or the graph becomes empty. The approximation ratio of S-LSE is 2.

### 4.4 MCE Algorithm.

The MCE algorithm described in [16] also achieves an approximation ratio of 2. This algorithm computes a $b$-Edge Cover by first computing a 1/2-approximate maximum weight $b'$-matching, with $b'(v) = \deg(v) - b(v)$. The $b$-Edge Cover is the complement of the edges in a $b'$-matching. If the latter is computed using an algorithm that matches in each iteration locally dominant edges (such as the Greedy or locally dominant edge or $b$-Suitor algorithms), then the MCE algorithm obtains a 2-approximation to the $b$-Edge Cover problem. The MCE algorithm produces an edge cover without any redundant edges, unlike the algorithms that we have considered.

## 5 Removing Redundant Edges

All the approximation algorithms (except the MCE) discussed in this paper may produce redundant edges in the edge cover. To see why, consider a path graph with six vertices as shown in Subfigure (a) of Figure 1. All the algorithms except MCE could report the graph as a possible edge cover. Although the approximation ratios of these algorithms are not changed by these redundant edges, practically these could lead to higher weights.

We discuss how to remove redundant edges optimally from the cover. A vertex is over-saturated if more than one covered edge is incident on it. (Or more than $b(v)$ edges are incident on $v$ for a $b$-Edge Cover.)

We denote by $G_T = (V_T, E_T, W_T)$ the subgraph of $G$ induced by over-saturated vertices. For each vertex $v$, let $c(v)$ denote the number of cover edges incident on $v$. Then $c(v_T)$ is the degree of a vertex $v_T \in G_T$. We let $b' = c(v_T) - b(v_T)$ for each vertex, $v_T \in V_T$. We have shown in earlier work [16] that we could find a maximum weighted $b'$-matching in $G_T$ and delete them from the edge cover to remove the largest weight possible from the edge cover. But since it is expensive to compute a maximum weighted $b'$-matching, we deploy a $b$-Suitor algorithm (1/2-approximation) to compute the $b'$-matching.

In Figure 1, two examples are shown of the removal process. All algorithms except MCE could produce the same graph as cover for both of the examples in Figure 1. For each example, the graph in the middle shows the over-saturated subgraph of the original graph. The label under the vertices represent the values of $c(v_T) - b(v_T)$). In Subfigure (a) we generate a sub-optimal matching (shown in dotted line), but in Subfigure (b) a maximum matching was found by the edge removal algorithm (the dotted line).

## 6 Experiments and Results

All the experiments were conducted on a Purdue Community cluster computer called *Snyder*, consisting of an Intel Xeon E5-2660 v3 processor with 2.60 GHz clock, 32 KB L1 data and instruction caches, 256 KB L2-cache, and 25 MB L3 cache.

Our testbed consists of both real-world and synthetic graphs. We generated two classes of RMAT graphs: (a) G500 representing graphs with skewed degree distribution from the Graph 500 benchmark [19], and (b) SSCA from HPCS Scalable Synthetic Compact Applications graph analysis (SSCA#2) benchmark. We used the following parameter settings: (a) $a = 0.57$, $b = c = 0.19$, and $d = 0.05$ for G500, and (b) $a = 0.6$, and $b = c = d = 0.4/3$ for SSCA. Additionally we consider seven datasets taken from the University of Florida Matrix collection covering application areas such as

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

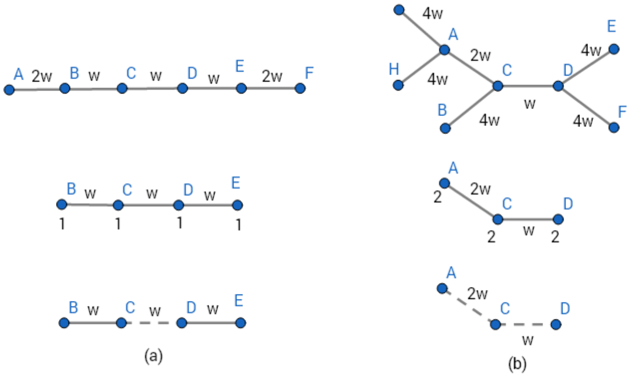

Figure 1: Removing redundant edges in two graphs. The top row of each column shows the original graph, the middle row shows the graph induced by the over-saturated vertices, and the bottom row shows edges in a matching indicated by dotted lines, which can be removed from the edge cover. In (a) we have a sub-optimal edge cover, but in (b) we find the optimal edge cover.

| **Problems** | $\lvert V\rvert$ | $\lvert E\rvert$ | **Avg. Deg.** |
|---|---:|---:|---:|
| **Fault_639** | 638,802 | 13,987,881 | 44 |
| **mouse_gene** | 45,101 | 14,461,095 | 641 |
| **Serena** | 1,391,349 | 31,570,176 | 45 |
| **bone010** | 986,703 | 35,339,811 | 72 |
| **dielFilterV3real** | 1,102,824 | 44,101,598 | 80 |
| **Flan_1565** | 1,564,794 | 57,920,625 | 74 |
| **kron_g500-logn21** | 2,097,152 | 91,040,932 | 87 |
| **hollywood-2011** | 2,180,759 | 114,492,816 | 105 |
| **G500_21** | 2,097,150 | 118,595,868 | 113 |
| **SSA21** | 2,097,152 | 123,579,331 | 118 |
| **eu-2015** | 11,264,052 | 264,535,097 | 47 |

Table 1: The structural properties of our testbed, sorted in ascending order of edges.

medical science, structural engineering, and sensor data. We also have a large web-crawl graph(*e*u-2015) [2] and a movie-interaction network(*h*ollywood-2011) [3]. Table 1 shows the sizes of our testbed. There are two groups of problems in terms of sizes: six smaller problems with fewer than 90 million edges, five problems with 90 million edges or more. Most problems in the collection have weights on their edges. The *eu-2015* and *hollywood-2011* are unit weighted graphs, and for *G500* and *SSA21* we chose random weights from a uniform distribution. All weights and runtimes reported are after removing redundant edges in the cover unless stated otherwise.

**6.1 Effects of Redundant Edge Removal.** All algorithms except the MCE algorithm have redundant edges in their covers. We remove the redundant edges by a Greedy matching algorithm discussed in Section 5. The effect of removing redundant edges is reported in Table 2. The second (fourth) column reports the weight obtained before applying the reduction algorithm, and the third (fifth) column is the percent reduction of weight due to the reduction algorithm for Lazy Greedy (Nearest Neighbor). The reduction is higher for Nearest Neighbor than for Lazy Greedy as the geometric mean for percent of reduction are 2.67 and 5.75 respectively. The Lazy Greedy algorithm obtains edge covers with lower weights relative to the Nearest Neighbor algorithm.

Table 2: Reduction in weight obtained by removing redundant edges for $b = 5$.

| **Problems** | **Init. Wt.** | **%Redn** | **Init. Wt.** | **%Redn** |
|---|---:|---:|---:|---:|
| | Lazy | Lazy | Nearest | Nearest |
| | Greedy | Greedy | Neighbor | Neighbor |
| **Fault_639** | 1.02E+16 | 4.02 | 1.09E+16 | 8.90 |
| **mouse_gene** | 3096.94 | 6.41 | 3489.92 | 11.82 |
| **serena** | 7.46E+15 | 4.92 | 7.84E+15 | 8.00 |
| **bone010** | 8.68E+08 | 1.99 | 1.02E+09 | 15.46 |
| **dielFilterV3** | 262.608 | 1.36 | 261.327 | 0.58 |
| **Flan_1565** | 5.57E+09 | 1.38 | 5.97E+09 | 3.69 |
| **kron_g500** | 4.58E+06 | 2.52 | 5.28E+06 | 8.55 |
| **hollywood** | 5.29E+06 | 2.78 | 7.63E+06 | 16.45 |
| **G500** | 1.37E+06 | 1.28 | 1.36E+06 | 0.95 |
| **SSA21** | 1.83E+12 | 7.43 | 1.87E+12 | 7.63 |
| **eu-2015** | 2.95E+07 | 1.60 | 3.31E+07 | 8.04 |
| **Geo. Mean.** | | **2.67** | | **5.75** |

**6.2 Quality Comparisons of the Algorithms.** The LSE, and the new Lazy Greedy and Dual Cover algorithms have approximation ratio 3/2. The MCE and Nearest Neighbor algo-

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

rithms are 2-approximation algorithms. But how do their weights compare in practice? We compare the weights of the covers from these algorithms with a lower bound on the minimum weight edge cover. We compute a lower bound by the Lagrangian relaxation technique [7] which is as follows. From the LP formulation we compute the Lagrangian dual problem. It turns out to be an unconstrained maximization problem with an objective function with a discontinuous derivative. We use sub-gradient methods to optimize this objective function. The dual objective value is always a lower bound on the original problem, resulting in a lower bound on the optimum. We also parallelize the Lagrangian relaxation algorithm. All the reported bounds are found within 1 hour using 20 threads of an Intel Xeon.

Table 3 shows the weights of the edge cover computed by the algorithms for $b = 1$. We report results here only for $b = 1$, due to space constraints and the observation that increasing $b$ improves the nearness to optimality. The second column reports the lower bound obtained from the Lagrangian relaxation algorithm. The rest of the columns are the percent of increase in weights w.r.t to the Lagrangian bound for different algorithms. The third through the fifth columns list the 3/2-approximation algorithms, and the last two columns list the 2-approximation algorithms. The lower the increase the better the quality; however, the lower bound itself might be lower than the minimum weight of an edge cover. So a small increase in weight over the lower bound shows that the edge cover has near-minimum weight, but if all algorithms show a large increase over the lower bound, we cannot conclude much about the minimum weight cover. The Dual Cover algorithm finds the lowest weight among all the algorithms for our test problems. Between MCE and Nearest Neighbor MCE produces lower weight covers except for the *hollywood-2011*, *eu-2015*, *kron_g500-logn21* and *bone010* graphs. Note that the 3/2-approximation algorithms always produce lower weight covers relative to the 2-approximation algorithms. The difference in weights is high for *bone010*, *kron_g500*, *eu-2015* and *hollywood-2011* graphs. The last two are unit-weighted problems, and the *kron_g500* problem has a narrow weight distribution (most of the weights are 1 or 2). On the other hand, all the algorithms produce near-minimum weights for the uniform random weighted graphs, *G500* and *SSA21*.

**6.3 Lazy Greedy and Dual Cover Performance.** The two earlier 3/2-approximation algorithms from the literature are the Greedy and the LSE [16]. Among them LSE is the better performing algorithm [14]. Hence we compare the Lazy Greedy and Dual Cover algorithms with the LSE algorithm. Table 4 compares the run-times of these three algorithms for $b = 1$ and 5. We report the runtimes (seconds) for the LSE algorithm. The Rel. Perf. columns for Lazy Greedy and Dual Cover report the ratio of the LSE runtime to the runtime of each algorithm. (The higher the ratio, the faster the algorithm). There were some problems for which the LSE algorithm did not complete within 4 hours, and for such problems we report the run-times of the Lazy Greedy and the Dual Cover algorithms.

It is apparent from the Table 4 that both Lazy Greedy and Dual Cover algorithms are faster than LSE. Among the three, the Dual Cover is the fastest algorithm. As we have discussed in Section 3, the efficiency of Lazy Greedy depends on the average number of queue accesses. In Figure 2, we show the average number of queue accesses for the test problems. The average number of queue accesses is computed as the ratio of total queue accesses (number of invocations of deQueue() and enQueue()) and the size of the edge cover. In the worst case it could be $O(|E|)$, but our experiments show that the average number of queue accesses is low. For the smaller problems, except for the *mouse_gene* graph, which is a dense graph, the average number of queue accesses is below 30, while for *mouse_gene*, it is about 600. For the larger problems, this number is below 200.

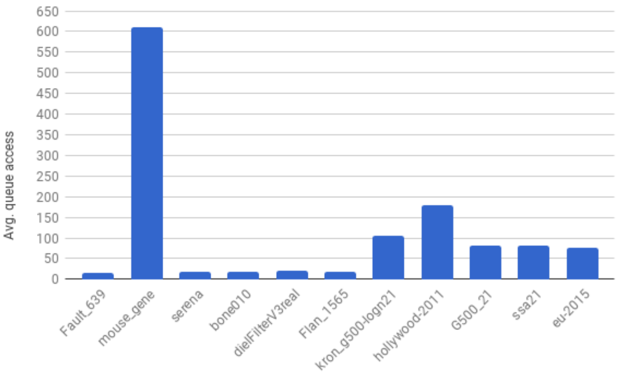

Figure 2: Average number of queue accesses per edge in the cover of Lazy Greedy algorithm.

Next we turn to the Dual Cover algorithm. As explained in Section 3, it is an iterative algorithm, and each iteration consists of two phases. The efficiency of the algorithm depends on the number of iterations it needs to compute the cover. In Figure 3, we show the number of iterations needed by the Dual Cover

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

Table 3: Edge cover weights computed by different algorithms, reported as increase over a Lagrangian lower bound, for $b = 1$. The lowest percentage increase is indicated in bold font.

| **Problems** | Lagrange bound | %Increase<br>LSE | <br>**LG** | <br>**DUALC** | <br>MCE | <br>**NN** |
|---|---|---|---|---|---|---|
| **Fault_639** | 7.80E+14 | **3.89** | **3.89** | **3.89** | 5.13 | 5.96 |
| **mouse_gene** | 520.479 | 22.29 | 22.29 | **22.26** | 36.16 | 36.55 |
| **serena** | 5.29E+14 | **2.44** | **2.44** | **2.44** | 3.61 | 4.42 |
| **bone010** | 1.52E+08 | **2.49** | 5.67 | **2.49** | 30.09 | 29.68 |
| **dielFilterV3real** | 14.0486 | **3.58** | **3.58** | **3.58** | 3.62 | 3.65 |
| **Flan_1565** | 1.62E+07 | **12.87** | **12.87** | **12.87** | **12.87** | **12.87** |
| **kron_g500-logn21** | 1.06E+06 | **5.68** | 8.52 | **5.68** | 26.27 | 22.96 |
| **G500** | 957392 | **0.07** | **0.07** | **0.07** | 0.11 | 0.13 |
| **SSA21** | 251586 | **1.13** | **1.13** | **1.13** | 1.87 | 3.15 |
| **hollywood-2011** | 1.62E+11 | N/A | 9.80 | **5.70** | 84.31 | 65.18 |
| **eu-2015** | 7.71E+06 | N/A | 4.28 | **3.19** | 21.01 | 16.52 |
| **Geo. Mean** | | 2.80 | 3.21 | 2.80 | 5.57 | 6.14 |

Table 4: Runtime comparison of the LSE, Lazy Greedy, and Dual Cover Algorithms. Values in bold font indicate the fastest performance for a problem.

| | **b=1** | | | **b=5** | |
|---|---|---|---|---|---|
| **Problems** | **Runtime** | **Rel. Perf./Run Time** | | **Runtime** | **Rel. Perf./ Run Time** |
| | LSE | **LG** | **DUALC** | LSE | **LG** |
| **Fault_639** | 3.02 | 1.32 | **3.57** | 8.93 | 3.23 |
| **mouse_gene** | 28.72 | 4.56 | **19.06** | 34.94 | 5.28 |
| **serena** | 7.56 | 1.10 | **6.32** | 16.11 | 2.00 |
| **bone010** | 70.26 | 63.48 | **259.1** | 162.2 | 109.13 |
| **dielFilterV3real** | 18.50 | 1.72 | **6.82** | 49.18 | 3.66 |
| **Flan_1565** | 9.53 | 1.26 | **7.06** | 26.76 | 2.47 |
| **kron_g500-logn21** | 1566 | 112.4 | **275.8** | 3786 | 234.6 |
| **SSA21** | 144.6 | 1.67 | **6.42** | 211.3 | 2.32 |
| **G500** | 4555 | 54.71 | **237.6** | >4 hrs | (NA, 88.17) |
| **hollywood-2011** | >4 hrs | (NA, 20.33) | (NA, 3.19) | >4 hrs | (NA, 22.41) |
| **eu-2015** | >4 hrs | (NA, 70.86) | (NA, 7.48) | >4 hrs | (NA, 74.45) |
| **Geo. Mean** | | 5.95 | 23.58 | | 8.09 |

algorithm. The maximum number of iterations is 20 for the *Fault_639* graph, while for most graphs, it converges within 10 iterations. Note that *Fault_639* is the smallest graph of all our test instances, although it is the hardest instance for Dual Cover algorithm. Note also that the hardest instance for Lazy Greedy was *mouse_gene* graph according to the average number of queue accesses.

**6.4** Nearest Neighbor **Performance.** The fastest 2-approximation algorithm in the literature is the MCE algorithm [16]. We compare the Nearest Neighbor algorithm with MCE algorithm for $b = 1$ in Table 5, and $b = 5$ in Table 6.The second and third columns show the runtime for MCE and relative performance of Nearest Neighbor w.r.t MCE. The next two columns report the weight found by MCE and percent of difference in weights computed by the Nearest Neighbor algorithm; a positive value indicates that the MCE weight is lower, and a negative value indicates the opposite. The Nearest Neighbor algorithm is faster than MCE.

The Nearest Neighbor algorithm is faster than the MCE algorithm. For $b = 1$ the geometric mean of the relative performance of the Nearest Neighbor algorithm is 1.97, while for $b = 5$ it is 4.10. There are some problems for which the Nearest Neighbor also computes a lower weight edge cover (the reported weight is the weight after removing redundant edges). For the test graphs we used, the Nearest Neighbor algorithm performs better than the MCE algorithm.

**6.5** Nearest Neighbor **and** Dual Cover **Comparison.** From the discussion so far, the best 3/2-serial algorithm for approximate minimum weighted edge cover is the Dual Cover algorithm. The Dual Cover algorithm computes near-minimum weight edge covers fast. We now compare the

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

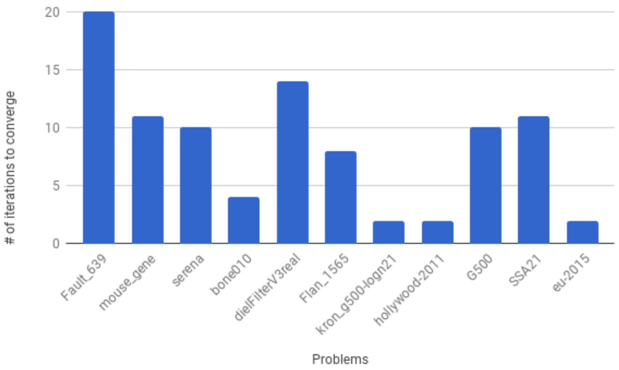

Figure 3: Number of iterations taken by the Dual Cover algorithm to compute an approximate minimum weight edge cover.

Table 5: Runtime performance and difference in weight of Nearest Neighbor w.r.t the MCE algorithm, with $b = 1$.

| Problems | Runtime | Perf. | Wt. | %Wt. |
|---|---|---|---|---|
| | | | | Incr. |
| | MCE | NN | MCE | NN |
| **Fault_639** | 2.42 | 0.31 | 8.20E+14 | 0.80% |
| **mouse_gene** | 6.79 | 0.58 | 708.697 | 0.28% |
| **serena** | 6.02 | 0.72 | 5.49E+14 | 0.78% |
| **bone010** | 3.72 | 0.27 | 1.97E+08 | -0.32% |
| **dielFilter** | 9.72 | 1.02 | 14.5565 | 0.04% |
| **Flan_1565** | 9.77 | 0.85 | 1.83E+07 | 0.00% |
| **kron_g500** | 45.92 | 8.75 | 1.34E+06 | -2.62% |
| **hollywood-2011** | 33.89 | 5.63 | 1.76E+06 | -10.38% |
| **G500** | 66.98 | 3.18 | 251869 | 0.02% |
| **SSA21** | 94.93 | 27.21 | 1.65E+11 | 1.26% |
| **eu-2015** | 82.31 | 13.33 | 9.32E+06 | -3.71% |

Dual Cover algorithm with Nearest Neighbor for b=1.

Table 7 shows the comparison between these two algorithms. The Nearest Neighbor algorithm is faster than Dual Cover but Dual Cover computes lower weight edge covers. The geometric mean of relative performance is 0.70%. For all the problems in our testbed, the Dual Cover algorithm computes a lower weight edge cover. The geometric mean of the reduction in weight is 2.87%, while it can be as large as 36%.

## 7 Conclusions

We summarize the state of affairs for approximation algorithms for the Edge Cover problem in Table 7. Nine algorithms are listed, and for each we indicate the approximation ratio; if it is a reduction from some form of matching; if there are redundant edges in the cover that could be removed to practically decrease the weight of the cover; and if the algorithm is concurrent. These algorithms can be extended to compute $b$-Edge Covers. We have implemented the MCE and S-LSE algorithms on parallel computers earlier [16], and will implement the Dual Cover algorithm on parallel machines in future work.

It seems surprising that the simple Nearest Neighbor algorithm is better in quality and runtime amongst other 2-approximation algorithms. But keep in mind that the Nearest Neighbor algorithm produces a number of redundant edges, and that the number of redundant edges increases with $b$. Also, the subgraph produced by Nearest Neighbor has irregular degree distribution that results in high degree nodes called hubs. These can be deterimental in applications such as semi-supervised learning [20]. Alternative algorithms have been proposed for machine learning, such as minimum weighted b-matching by

Table 6: Runtime performance and difference in weight of Nearest Neighbor w.r.t the MCE algorithm, with $b = 5$.

| Problems | Runtime | Rel. Perf. | Wt | %Wt. |
|---|---|---|---|---|
| | | | | Incr. |
| | MCE | NN | MCE | NN |
| **Fault_639** | 2.31 | 4.32 | 9.89E+15 | 0.09 |
| **mouse_gene** | 6.61 | 9.49 | 3087.81 | -0.34 |
| **serena** | 5.73 | 4.27 | 7.20E+15 | 0.20 |
| **bone010** | 3.65 | 5.02 | 8.43E+08 | 2.09 |
| **dielFilter** | 9.37 | 5.45 | 259.326 | 0.19 |
| **Flan_1565** | 9.18 | 6.71 | 5.74E+09 | 0.25 |
| **kron_g500** | 44.58 | 1.67 | 4.96E+06 | -2.54 |
| **hollywood-2011** | 32.80 | 5.88 | 7.10E+06 | -10.12 |
| **G500** | 66.06 | 1.77 | 1.35E+06 | 0.00 |
| **SSA21** | 92.01 | 9.81 | 1.71E+12 | 0.55 |
| **eu-2015** | 78.71 | 1.01 | 3.15E+07 | -3.57 |

Table 7: The runtime and the edge cover weights of the Nearest Neighbor and Dual Cover algorithms for $b = 1$. The third column reports the ratio of runtimes(NN/DUALC); the fifth column reports the reduction in weight achieved by the Dual Cover algorithm.

| Problems | Time | Perf. | Weight | %Wt. Impr. |
|---|---|---|---|---|
| | NN | DUALC | NN | DUALC |
| **Fault_639** | 0.31 | 0.37 | 8.26E+14 | 1.96 |
| **mouse_gene** | 0.58 | 0.38 | 710.711 | 10.46 |
| **serena** | 0.72 | 0.60 | 5.53E+14 | 1.89 |
| **bone010** | 0.27 | 1.00 | 1.97E+08 | 20.97 |
| **dielFilterV3real** | 1.02 | 0.37 | 14.5616 | 0.07 |
| **Flan_1565** | 0.85 | 0.63 | 1.83E+07 | 0.00 |
| **kron_g500-logn21** | 8.75 | 1.54 | 1.31E+06 | 14.06 |
| **hollywood-2011** | 3.18 | 1.00 | 1.58E+06 | 36.01 |
| **G500** | 13.33 | 0.70 | 251907 | 0.06 |
| **SSA21** | 5.63 | 0.25 | 1.67E+11 | 1.96 |
| **eu-2015** | 27.21 | 3.64 | 8.98E+06 | 11.44 |
| **Geo. Mean** | | 0.70 | | 2.87 |

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**

Jebara et al. [11] or Mutual $K$-Nearest Neighbor by Ozaka et al. [20]. We will explore the use of $b$-Edge Cover algorithms for this graph construction.

## Acknowledgements

We are grateful to all referees for their constructive comments, and especially to one reviewer who provided a lengthy and insightful review.

## References

[1] R. P. Anstee, *A polynomial algorithm for b-matchings: An alternative approach*, Inf. Process. Lett., 24 (1987), pp. 153–157.

[2] P. Boldi, A. Marino, M. Santini, and S. Vigna, *BUbiNG: Massive crawling for the masses*, in Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web, 2014, pp. 227–228.

[3] P. Boldi and S. Vigna, *The WebGraph framework I: Compression techniques*, in WWW 2004, ACM Press, 2004, pp. 595–601.

[4] V. Chvatal, *A greedy heuristic for the set-covering problem*, Mathematics of Operations Research, 4 (1979), pp. 233–235.

[5] F. Dobrian, M. Halappanavar, A. Pothen, and A. Al-Herz, *A 2/3-approximation algorithm for vertex-weighted matching in bipartite graphs.* Preprint, submitted for publication, 2017.

[6] R. Duan, S. Pettie, and H.-H. Su, *Scaling algorithms for weighted matching in general graphs*, in Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '17, Philadelphia, PA, USA, 2017, Society for Industrial and Applied Mathematics, pp. 781–800.

[7] M. L. Fisher, *The Lagrangian relaxation method for solving integer programming problems*, Management Science, 50 (2004), pp. 1861–1871.

[8] H. N. Gabow, *Data structures for weighted matching and nearest common ancestors with linking*, in Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '90, Philadelphia, PA, USA, 1990, Society for Industrial and Applied Mathematics, pp. 434–443.

[9] D. S. Hochbaum, *Approximation algorithms for the set covering and vertex cover problems*, SIAM Journal on Computing, 11 (1982), pp. 555–556.

[10] D. Huang and S. Pettie, *Approximate generalized matching: $f$-factors and $f$-edge covers*, CoRR, abs/1706.05761 (2017).

[11] T. Jebara, J. Wang, and S.-F. Chang, *Graph construction and b-matching for semi-supervised learning*, in Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09, New York, NY, USA, 2009, ACM, pp. 441–448.

[12] D. S. Johnson, *Approximation algorithms for combinatorial problems*, Journal of Computer and System Sciences, 9 (1974), pp. 256–278.

[13] R. M. Karp, *Reducibility among Combinatorial Problems*, Springer US, Boston, MA, 1972, pp. 85–103.

[14] A. Khan and A. Pothen, *A new 3/2-approximation algorithm for the b-edge cover problem*, in Proceedings of the SIAM Workshop on Combinatorial Scientific Computing, 2016, pp. 52–61.

[15] A. Khan, A. Pothen, S M Ferdous, M. Halappanavar, and A. Tumeo, *Adaptive anonymization of data using b-edge cover.* Preprint, submitted for publication, 2018.

[16] A. Khan, A. Pothen, and SM Ferdous, *Parallel algorithms through approximation: b-edge cover*, in Proceedings of IPDPS, 2018. Accepted for publication.

[17] G. Kortsarz, V. Mirrokni, Z. Nutov, and E. Tsanko, *Approximating minimum-power network design problems*, in 8th Latin American Theoretical Informatics (LATIN), 2008.

[18] M. Minoux, *Accelerated greedy algorithms for maximizing submodular set functions*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1978, pp. 234–243.

[19] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, *Introducing the Graph 500*, Cray User's Group, (2010).

[20] K. Ozaki, M. Shimbo, M. Komachi, and Y. Matsumoto, *Using the mutual k-nearest neighbor graphs for semi-supervised classification of natural language data*, in Proceedings of the Fifteenth Conference on Computational Natural Language Learning, CoNLL '11, Stroudsburg, PA, USA, 2011, Association for Computational Linguistics, pp. 154–162.

[21] A. Schrijver, *Combinatorial Optimization - Polyhedra and Efficiency. Volume A: Paths, Flows, Matchings*, Springer, 2003.

[22] A. Subramanya and P. P. Talukdar, *Graph-Based Semi-Supervised Learning*, vol. 29 of Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool, San Rafael, CA, 2014.

[23] X. Zhu, *Semi-supervised Learning with Graphs*, PhD thesis, Pittsburgh, PA, USA, 2005. AAI3179046.

| Algorithm | Appx. Ratio | Matching based | Red. Edges | Conc. |
|---|---|---|---|---|
| Greedy | 3/2 | N | Y | N |
| Hochbaum | $\Delta$ | Y | Y | N |
| Lazy Greedy | 3/2 | N | Y | N |
| LSE | 3/2 | N | Y | Y |
| Dual Cover | 3/2 | N | Y | Y |
| NN | 2 | N | Y | Y |
| S-LSE | 2 | N | Y | Y |
| MCE | 2 | Y | N | Y |
| Huang & Pettie | $1 + \epsilon$ | Y | N | ? |

**Copyright © 2017 by SIAM**
**Unauthorized reproduction of this article is prohibited**