# ALGORITHMS FOR DEGREE-CONSTRAINED SUBGRAPHS AND APPLICATIONS

by

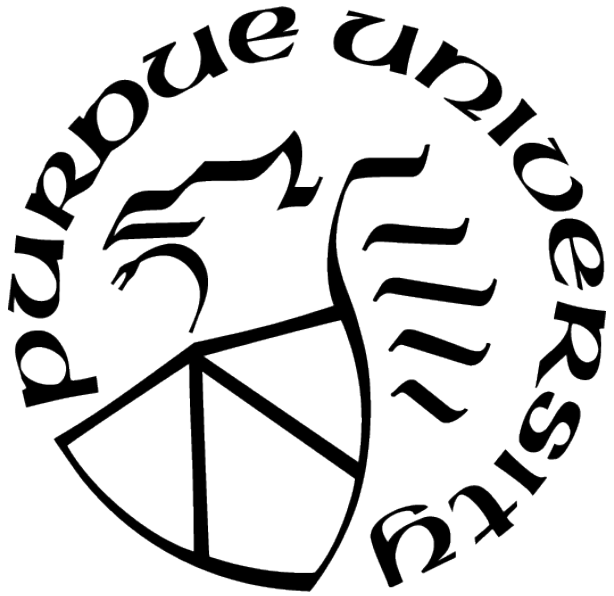
**S M Ferdous**

**A Dissertation**

*Submitted to the Faculty of Purdue University*

*In Partial Fulfillment of the Requirements for the degree of*

**Doctor of Philosophy**




Department of Computer Science

West Lafayette, Indiana

December 2021

# THE PURDUE UNIVERSITY GRADUATE SCHOOL
# STATEMENT OF COMMITTEE APPROVAL

**Dr. Alex Pothen, Chair**

Department of Computer Science

**Dr. Petros Drineas**

Department of Computer Science

**Dr. Seth Pettie**

Department of Computer Science, University of Michigan-Ann Arbor

**Dr. Hemanta Maji**

Department of Computer Science

**Dr. Pan Li**

Department of Computer Science

**Approved by:**

Dr. Kihong Park

# ACKNOWLEDGMENTS

Acknowledging is an extremely tough job. Always there are risks to miss out on many. With this in mind, here is an honest effort to list the bodies I am grateful for during the PhD journey.

I want to acknowledge the funding agencies, NSF and DOE, for providing financial support during my PhD journey. My advisor Prof. Alex Pothen maintained a proper balance of independence and guidance during my PhD. He was always prompt to advise when it was required. I can not thank you enough for the advice and support. He introduced me to the art of research, which I am still learning. Besides professional support, he also cared about my personal well-being. He understands the struggles of international students like me who leave a significant part of their families overseas and helps them to settle in. His wife, Bobbie Pothen, is always kind and welcoming to me.

I am grateful to the dissertation committee members, Prof. Petros Drineas, Prof. Seth Pettie, Prof. Hemanta Maji, and Prof. Pan Li, for their constructive suggestions about my thesis. I would also like to acknowledge my collaborators across different labs and industries for the opportunities to work and learn from them. I want to express my gratitude to Dr. Mahantesh Halappanavar and Dr. Arif Khan of Pacific Northwest National Laboratory, co-authors of many works presented in this thesis. Thanks to all my fellow group members, especially Shivaram Gopal, for their encouragement and fruitful discussion.

I want to especially acknowledge the sacrifice and support of my loving wife, Bushra Ferdousi. She encourages and supports unconditionally for my cause. I also want to express gratefulness to my parents for their continuing sacrifice for my life. My brother, sister, and sister-in-law always stood by me when needed.

I am thankful to Dr. Sohel Rahman, my master's supervisor, and Dr. Arif Dowla, Managing director of ACI Ltd. They have motivated and supported me to apply for a PhD in the first place. I thank my school and college teachers, especially Mr. Sahadat Hossain and Mr. Rafi Ahammed, for their tremendous help in my education.

I want to express my gratitude to the fellow Bangladeshi students at Purdue University for the community support. Especially, I want to mention my friends Priyam Biswas, Sagar Chowdhury, Marufa Khandaker Joyeeta, and Abdullah Al Mamun for the gracious support, long night addas (informal discussions), and fantastic foods, which immensely helped to relieve my stress. Time spent with the kids of my friends at Purdue will forever be in my memory. Among them, I want to mention Shanaya, Yusuf, and Mersiha especially.

Finally, I am ever grateful to the Almighty God for everything.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF SYMBOLS

$n$ number of vertices of a graph

$m$ number of edges of a graph

$\Delta$ maximum degree of a graph

# ABSTRACT

A degree-constrained subgraph construction (DCS) problem aims to find an optimal spanning subgraph (w.r.t an objective function) subject to certain degree constraints on the vertices. DCS generalizes many combinatorial optimization problems such as Matchings and Edge Covers and has many practical and real-world applications. This thesis focuses on DCS problems where there are only upper and lower bounds on the degrees, known as b-matching and b-edge cover problems, respectively. We explore linear and submodular functions as the objective functions of the subgraph construction.

The contributions of this thesis involve both the design of new approximation algorithms for these DCS problems, and also their applications to real-world contexts. We designed, developed, and implemented several approximation algorithms for DCS problems. Although some of these problems can be solved exactly in polynomial time, often these algorithms are expensive, tedious to implement, and have little to no concurrency. On the contrary, many of the approximation algorithms developed here run in nearly linear time, are simple to implement, and are concurrent. Using the local dominance framework, we developed the first parallel algorithm submodular b-matching. For weighted b-edge cover, we improved the classic Greedy algorithm using the lazy evaluation technique. We also propose and analyze several approximation algorithms using the primal-dual linear programming framework and reductions to matching. We evaluate the practical performance of these algorithms through extensive experimental results.

The second contribution of the thesis is to utilize the novel algorithms in real-world applications. We employ submodular b-matching to generate a balanced task assignment for processors to build Fock matrices in the NWChemEx quantum chemistry software. Our load-balanced assignment results in a four-fold speedup per iteration of the Fock matrix computation and scales to 14,000 cores of the Summit supercomputer at Oak Ridge National Laboratory. Using approximate b-edge cover, we propose the first shared-memory and distributed-memory parallel algorithms for the adaptive anonymity problem. Minimum weighted b-edge cover and maximum weight b-matching are shown to be applicable to

constructing graphs from datasets for machine learning tasks. We provide a mathematical optimization framework connecting the graph construction problem to the DCS problem.

# 1. INTRODUCTION

This thesis explores efficient approximation algorithms and real-life applications for Degree-constrained subgraph construction (DCS) problems. We construct significant subgraphs based on two criteria: 1) the subgraph has bounds on vertex degrees (to control sparsity), 2) and the subgraph optimizes a suitable objective function (to choose significant subgraphs). The DCS is a broad problem that encompasses several discrete optimization problems, and could also be viewed as a *sketching* of the data.

## 1.1 Basic Terminology

We provide a collection of minimal terminologies to get started. we will re-introduce many of these terminologies later if it is felt necessary as well as additional notations and terminologies later part of the thesis.

Almost all the algorithms in the thesis takes undirected graph as an input. A graph $G$ consists of a finite non-empty sets of elements called the vertices (or nodes) $V$, and a collection of unordered pair of vertices called the edges $E$. A major part of the thesis assumes the graph is simple i.e., there are no parallel edges or self-loops. If $\mathrm{e} = \{u, v\}$, where $u, v \in V$ is an edge in $G$, $u$ and $v$ are called endpoints of the edge. Two edges those share an endpoint are called adjacent edges. The set of adjacent edges, those share an endpoint $v \in V$, are called incident on $v$. If the graph is simple then an edge, $\{u, v\}$ is a set of cardinality exactly 2, and $E$ becomes a set of such edges. We employed $\{u, v\}$ and $(u, v)$ alternatively to represent an edge throughout the thesis. An unweighted graph is denoted by $G(V, E)$. We define a function $w : E \to \mathbb{R}_{\geq 0}$, that assigns a non-negative real valued weight to the edges. In presence of weights, we denote the graph as $G(V, E, w)$. The weight function can be extended to a subset of edges as follows. Given a set $S \subseteq E$, we define $w(S) \coloneqq \sum_{\mathrm{e} \in S} w(\mathrm{e})$. If $G$ and $H$ are two graphs and every edge of $H$ is also an edge of $G$, then $H$ is called an induced subgraph (or simply a subgraph) of $G$.

Given a set, $S \subset V$, $\delta(S)$ is defined as the *cut* edge set of $S$ i.e., the edges those have only one endpoint in $S$. So, the set of the edges incident on a vertex $v$ is $\delta(\{v\})$ (or $\delta(v)$).

The degree of a vertex is thus $|\delta(v)|$. Similarly, $\gamma(S)$ represents the set of edges those have both endpoints in $S$.

### 1.2 The degree-constrained subgraph problem

We now formally define the DCS problem and the variants that we discuss in later chapters in this thesis. Let $G(V, E, w)$ be the original graph where $V$ and $E$ are the set of vertices and edges, and $w : E \to \mathbb{R}^+$ is the weight function on the edges. We denote by $n$ and $m$ the number of vertices and edges, respectively. Let $\delta(v)$ denote the set of edges incident on a vertex $v \in V$. Our goal is to construct a subgraph of $G$ following a set of degree constraints on the subgraph. Given a binary function $x : E \to \{0, 1\}$ we define, $E' \coloneqq \{\mathrm{e} \in E : x(\mathrm{e}) = 1\}$. Formally, we are interested in the following general binary optimization problem.

$$\begin{aligned} &\text{optimize } f(E'), \\ &\qquad l(\mathrm{i}) \leq \sum_{\mathrm{e}\in\delta(\mathrm{i})} x(\mathrm{e}) \leq u(\mathrm{i}), \quad \forall \mathrm{i} \in V, \\ &\qquad x(\mathrm{e}) \in \{0, 1\} \quad \forall \mathrm{e} \in E. \end{aligned} \tag{1.1}$$

Here $E'$ consists of the edges in the subgraph, and for such edges $x(\mathrm{e}) = 1$; $l(\mathrm{i})$ and $u(\mathrm{i})$ are positive integers representing the lower bounds and upper bounds on the degree of a vertex i, and $f$ is a set function. In this thesis, we limit our focus to linear and submodular functions. When the function is linear $f$ can be expressed as $\sum_{\mathrm{e}\in E} w(\mathrm{e})x(\mathrm{e})$. For submodular objective, we do not assume any specific functional form. For concreteness, we state a submodular function that we have recently applied in [1] for load balancing parallel Fock matrix computations in quantum chemistry. In this setting, the graph is a bipartite with two parts in vertex set, say, $U$ and $V$. The objective function is

$$f = \sum_{u\in U} \left( \sum_{\mathrm{e}\in\delta(u)} w(\mathrm{e})x(\mathrm{e}) \right)^{\alpha} + \sum_{v\in V} \left( \sum_{\mathrm{e}\in\delta(v)} w(\mathrm{e})x(\mathrm{e}) \right)^{\alpha},$$

is submodular when $\alpha \in (0\ 1)$.

DCS is one of the classical combinatorial optimization problems. The general DCS for linear function was first described and solved by reduction to $b$-matching in [2]. The existence criteria for a DCS with particular lower and upper bounds are shown in [3]–[5]. Several extensions of basic DCS for linear functions have been addressed recently. If in addition to the lower bound $l(\mathrm{i})$, we also allow 0, that is the degree of a vertex can be now in the set, $\{0, l(\mathrm{i}), \ldots, u(\mathrm{i})\}$, the maximization problem becomes NP-hard even for bipartite graphs [6]. If the subgraph also needs to be connected, maximizing the linear function becomes NP-hard [7] even for $l(\mathrm{i}) = 0$, and $u(\mathrm{i}) = d$, $\forall \mathrm{i} \in V$, and $d \geq 2$. In [8], the authors provide a $(\min\{n/2, m/2\})$-approximation algorithm for any $d \geq 2$.

Depending on the constraints and objective function, several special problems can be developed. We list our problems of interest as follows.

### 1.3 $b$-matching

When $l(\mathrm{i}) = 0$ for all the vertices the formulation 1.1 reduces to a $b$-matching constrained problem.

**Definition 1.3.1.** *Given a function $b(v)$ that maps each vertex to a natural number i.e., $b : V \to \mathbb{N}$, a $b$-matching in a graph $G(V, E)$ is set of edges $M \subseteq E$ such that* at most $b(v)$ *edges in $M$ are incident on a vertex $v$.*

If the objective function $f$ is linear, the maximum weight $b$-matching problem is polynomially solvable, and we will discuss exact algorithms for $b$-matching is subsequent sections. If $b(v) = 1$, $\forall v \in V$, we call it a 1-matching or simply a matching problem. We denote for a set $S \subseteq V$, $b(S) = \sum_{v \in S} b(v)$, and $\beta = \max_{v \in V} b(v)$.

#### 1.3.1 Exact algorithms for 1-matching problem

We provide an extremely brief review of the exact algorithms for weighted and unweighted matching problems. A maximum cardinality matching in an unweighted graph has the maximum number of independent edges. If the graph is bipartite, solving the Linear Programming (LP) relaxation of the integer programming formulation provides an integral solution due

to the total unimodularity of the constraint matrix. Thus it establishes the polynomial complexity of maximum cardinality matching. The best runtime of maximum cardinality matching in general bipartite graphs is $O(\sqrt{n}m)$ due to Hopcroft and Karp [9]. For regular biparite graphs, Goel, Kapralov and Khanna in [10] developed a randomized algorithm that runs in $O(n \lg n)$ time. For general non-bipartite graphs unfortunately the LP relaxation does not guarantee an integral solution. We can generate a stronger LP by adding *odd set constraints* which is defined next. Let $S$ be a subset of $V$ of odd cardinality ($|S| \geq 3$). Any matching of $G$ cannot intersect $\gamma(S)$ in more than $(|S| - 1)/2$ edges. This gives us a set of new constraints to the LP relaxation. Edmonds in [11] showed that the new LP with the odd set constraints has integral solutions. The best runtime for maximum cardinality matching in general graph matches the runtime as in the biparitite graph i.e, $O(\sqrt{n}m)$ [12].

For weighted graphs, the primal-dual Hungarian algorithm [13] solves the bipartite maximum weighted matching problem. The best runtime ($O(nm + n^2 \log n)$) for weighted matching in a bipartite graph is achieved by using Fibonacci heaps in the Hungarian algorithm and is due to Fredman and Tarjan [14]. Similar to the cardinality version, the general maximum weight matching is complicated by the presence of odd set constraints. Edmonds in the seminal work [11], [15] shows how to avoid maintaining the exponential number of odd set constraints by introducing blossoms and the data-structures to maintain them in polynomial time, thus showing the maximum weighted matching is solvable in polynomial time. Using a more complex data structure, Gabow [16] developed an algorithm for maximum weighted matching in a general graph whose runtime matches the bipartite counterpart. We have shown an LP formulation of the 1-matching problem with the odd set constraints in Formulation A.2.

### 1.3.2 Exact algorithms for $b$-Matching

We show a strong LP relaxation of the weighted $b$-matching problem in Formulation A.5. An exact algorithm for a maximum weight $b$-Matching was first designed by Edmonds in [11]. Pulleyblank [17] later gave a polynomial time algorithm with complexity $O(mnb(V)) = O(m^2 n)$. Cunningham and Marsh [18] developed an algorithm for $b$-Matching that runs

in $O(n^2 m \log \beta)$ time, and Gabow [2] designed an efficient reduction based algorithm that requires $O(m^2 \log n \log \beta)$ time. Anstee [19] proposed a three-stage algorithm where the $b$-MATCHING problem is solved by transforming it to a Hitchcock transportation problem, rounding the solution to integer values, and finally invoking Pulleyblank's algorithm. It requires $\tilde{O}(n^2 m)$. Miller and Pekny in [20] improved the Anstee algorithm further, and developed an algorithm that requires $O(n^2 m)$ time. [21] developed another algorithm using the branch and cut approach, and [22] solved the problem using the cutting plane technique. A survey of exact algorithms for $b$-MATCHINGS was provided by [23]. More recently, [24] proposed an exact $b$-MATCHING algorithm based on belief propagation. The algorithm assumes that the solution is unique, and otherwise it does not guarantee convergence.

There are several reductions from $b$-MATCHING to 1-matching. Tutte [25] showed a reduction from $b$-MATCHING to matching that creates a new graph by splitting each node of $G$ into $b(v)$ copies, and replaces each edge $(u, v)$ by $b(u)b(v)$ edges that forms a complete bipartite subgraph joining the copies of $u$ and $v$. The new graph has $O(m)$ vertices and $O(\beta^2 m)$ edges. There is a different reduction that replaces each edge by $b(u) + b(v) + 1$ edges; it thus creates $O(\beta m)$ edges in total. Using the best weighted matching algorithm as black-box, this reduction based algorithm runs in $\tilde{O}(\beta m^2)$ time. For sparse graphs and small $\beta$, this could be better than the three stage algorithm due to Anstee. In Appendix B, we provide a detailed discussion of the second reduction.

### 1.3.3 Approximation algorithms for $b$-MATCHING

Mestre in [26] showed that a $b$-MATCHING is a relaxation of a *matroid* called a 2-*extendible* system, and hence that the Greedy algorithm gives a 1/2-approximation for a maximum weighted $b$-MATCHING. Mestre also generalized the Path-Growing algorithm of [27] to obtain an $O(\beta m)$ time 1/2-approximation algorithm. These algorithms are slower in practice than a serial $b$-SUITOR algorithm that generalizes the SUITOR algorithm [28]. Since the PATH GROWING algorithm is inherently sequential, it is not a good candidate for parallelization. Additionally, [26] generalized a randomized algorithm for MATCHING to obtain a $(2/3 - \epsilon)$-approximation algorithm with expected running time $O(\beta m \log \frac{1}{\epsilon})$. [29] have

adapted the GREEDY algorithm and an integer linear program (ILP) based algorithm to the MapReduce environment to compute $b$-MATCHINGS in bipartite graphs. $b$-MATCHING algorithms have also been developed using linear programming [30], [31], but these methods are orders of magnitude slower than the $b$-SUITOR algorithm. [32] describes a distributed algorithm based on adding locally dominating edges to the $b$-MATCHING, which leads to a LOCALLY DOMINANT EDGE algorithm.

### 1.4 $b$-edge cover

Another special form of Problem 1.1 arises when we assume $u(\mathrm{i}) \geq |\delta(\mathrm{i})|$, i.e., the subgraph of interest is lower bounded only, and this is known as $b$-edge cover.

**Definition 1.4.1.** *Given a function $b(v)$ that maps each vertex to a natural number, a $b$-edge cover in a graph is set of edges $C$ such that* at least *$b(v)$ edges in $C$ are incident on a vertex $v$.*

When the objective is to minimize a linear function, the corresponding problem is polynomially solvable, since it can be reduced to $b$-matching [33].

#### 1.4.1 Exact algorithm for edge cover

For the unweighted version of both MATCHING and EDGE COVER , we have a well established relationship due to Gallai [34] and Norman and Rabin [35]. We restate this relationship by the following theorem from Schrijver [33, Theorem 19.1 and 19.2 ].

**Theorem 1.4.1.** *Let $G = (V, E)$ be a graph with $n$ vertices and $m$ edges. Every maximum cardinality matching is contained in a minimum cardinality edge cover, and every minimum cardinality edge cover contains a maximum cardinality matching. Moreover if we have a maximum cardinality matching in $G$, we can find a minimum cardinality edge cover in time $O(m)$, and vice versa.*

The $O(m)$ time in the last part of the Theorem 1.4.1 comes from a simple algorithm due to Norman and Rabin [35]. For each unmatched vertex of a maximum cardinality matching,

we add an arbitrary incident edge. This results in a minimum cardinality cover. Thus the runtime of a minimum cardinality edge cover algorithm is $O(\sqrt(n)m)$.

There are several reductions from the minimum weighted edge cover to the weighted matching problem. We sketch a couple of reductions here.

**Reduction to Perfect Matching:**

Given a graph $G = (V, E, w)$, the reduction creates a second disjoint copy $\bar{G} = (\bar{V}, \bar{E}, \bar{w})$ of the original graph, and connects each vertex $v \in V$ with its copy $\bar{v} \in \bar{V}$. Let $G'(V', E', w')$ denote the new graph, and let $\mu(v)$ denote an edge of lowest weight incident on $v$ in $G$. The weight function $w'$ on $E'$ is defined as follows: $w'(\mathrm{e}) = w(\mathrm{e})$ for each $\mathrm{e} \in E$, $w'(\mathrm{e}') = \bar{w}(\bar{\mathrm{e}})$ for each $\bar{\mathrm{e}} \in \bar{E}$, and $w'(v, \bar{v}) = 2w(\mu(v))$. It is easy to verify that $G'$ has a perfect matching. We can compute a minimum weight perfect matching $M^*$ in $G'$; then we obtain a minimum weight edge cover of $G$ by selecting the edges $C = (M^* \cap E) \cup \{\mu(v) : (v, \bar{v}) \in M^*, \forall v \in V\}$.

**Reduction to maximum weighted matching:**

We compute a transformed weight $w'$ for each edge $\mathrm{e} = (u, v)$ as

$$w'(u, v) = \mu(u) + \mu(v) - w(u, v).$$

(Note that we write $w(u, v)$ instead of $w((u, v))$.) Consider how the weight of an edge is transformed under this mapping. There are three cases.

**Case 1** $\mu(u) = \mu(v) = w(u, v)$. We call such an edge locally subdominant, since this edge is of minimum weight among all of its neighboring edges. The reader can verify that the transformation does not change the weight of such an edge; thus $w(u, v) = w(u, v)$.

**Case 2** $\mu(u) = w((u, v)) > \mu(v)$. The reader can verify that $w(u, v) = \mu(v) < w(u, v)$.

**Case 3** $w(u, v) > \mu(v) > \mu(u)$. It is easily verified that $w(u, v) < \mu(u) < w(u, v)$.

The other cases may be obtained from the symmetry of $u$ and $v$. In all cases, we see that the transformed weight of an edge is no larger than the minimum weight among its neighboring edges.

Assume that we are given a MATCHING $M$ with the transformed weights on the edges. Define $V(M)$ to be the set of matched vertices in $M$, and let $\mathrm{e}(v)$ denote an edge of minimum weight incident on $v$. We can compute an edge cover $C$ as follows:

$$C = M \cup \{\mathrm{e}(v) : v \in V \setminus V(M).$$

Hence the EDGE COVER consists of the matched edges and a set of minimum weight edges incident on the unmatched vertices.

If $M^*$ is a maximum weight matching with respect to the weights $w$, then the resulting edge cover $C^*$ is an edge cover of minimum weight with respect to the weights $w$. Furthermore, Huang and Pettie [36] showed that this reduction is approximation-preserving.

### 1.4.2 Exact Algorithm for $b$-EDGE COVER

Similar to weighted 1-edge cover, weighted $b$-EDGE COVER admits several reductions to weighted $b$-matching. We discuss in Chapter 4.2 how to reduce a $b$-EDGE COVER to a constraint perfect $b$-matching problem.

#### Reduction to $b'$-matching:

We refer to $b'$-MATCHING instead of $b$-MATCHING to avoid ambiguity in this subsection. Given a graph $G = (V, E, b)$, a minimum weight $b$-EDGE COVER can be obtained from a maximum weight $b'$-MATCHING [33] as follows:

1. For each vertex $v$, compute $b'(v) = \deg(v) - b(v)$.
2. Compute $M_{opt}$, a maximum weight $b'$-MATCHING.
3. Compute a $b$-EDGE COVER as the complement of the matching: $C_{opt} = E \setminus M_{opt}$.

In this construction, steps 1 and 3 ensure that the computed $b$-Edge Cover is a valid cover, and the optimality of the cover depends on step 2.

**Reduction to 1-edge cover**

Recall that a $b$-matching can be reduced to 1-matching by replacing each edge with $b(u) + b(v) + 1$ edges. In a similar way a weighed $b$-edge cover can be also be reduced to a weighted 1-edge cover problem. We provide a detailed of the reduction in Appendix B.

### 1.4.3 Approximation algorithm for $b$-Edge Cover

There are several approximation algorithms for the $b$-edge cover problem. The KNN algorithm provides 2-approximation, and other 2-approximation algorithms include the Locally subdominant edge and Matching complement algorithms. 3/2-approximation algorithms include Greedy, Lazy greedy, and Primal dual.

## 1.5 Background on Submodular Optimization

In this section we describe submodular functions and their properties, formulate the submodular $b$-Matching problem, and discuss approximation algorithms for the problem.

### 1.5.1 Submodular $b$-Matching

**Definition 1.5.1** (Marginal gain)**.** *Given a ground set $X$, the marginal gain of adding an element* $\mathrm{e} \in X$ *to a set $A \subseteq X$ is*

$$\rho_{\mathrm{e}}(A) = f(A \cup \{\mathrm{e}\}) - f(A).$$

The marginal gain may be viewed as the discrete derivative of the set $A$ for the element e. Generalizing, the marginal gain of adding a subset $Q$ to another subset $A$ of the ground set $X$ is

$$\rho_Q(A) = f(A \cup Q) - f(A).$$

**Definition 1.5.2** (Submodular set function)**.** *Given a set $X$, a real-valued function $f$ defined on the subsets of $X$ is submodular if*

$$\rho_{\mathrm{e}}(A) \geq \rho_{\mathrm{e}}(B)$$

*for all subsets $A \subseteq B \subseteq X$, and elements $\mathrm{e} \in X \setminus B$. The function $f$ is monotone if for all sets $A \subseteq B \subseteq X$, we have $f(A) \leq f(B)$; it is normalized if $f(\emptyset) = 0$.*

We will assume throughout this paper that $f$ is normalized. The concept of submodularity also extends to the addition of a set. Formally, for $Q \subseteq X \setminus B$, $f$ is submodular if $\rho_Q(A) \geq \rho_Q(B)$. If $f$ is monotone then $\rho_{\mathrm{e}}(A) \geq 0$, $\forall A \subseteq X$ and $\forall \mathrm{e} \in X$.

**Proposition 1.5.1.** *Let $S = \{\mathrm{e}_1, \ldots, \mathrm{e}_k\}$, $S_{\mathrm{i}}$ be the subset of $S$ that contains the first* i *elements of $S$, and $f$ be a normalized submodular function. Then $f(S) = \sum_{\mathrm{i}=1}^{k} \rho_{\mathrm{e}_{\mathrm{i}}}(S_{\mathrm{i}-1})$.*

*Proof.* By construction, $S_0 = \emptyset$, and $S_k = S$.

$$\begin{aligned}\sum_{\mathrm{i}=1}^{k} \rho_{\mathrm{e}_{\mathrm{i}}}(S_{\mathrm{i}-1}) &= \sum_{\mathrm{i}=1}^{k} f(S_{\mathrm{i}-1} \cup \mathrm{e}_{\mathrm{i}}) - f(S_{\mathrm{i}-1}) \\ &= f(S_k) - f(S_0) = f(S).\end{aligned}$$

■

**Proposition 1.5.2.** *For sets $A \subseteq B \subseteq X$, and $\mathrm{e} \in X$, a monotone submodular function $f$ defined on $X$ satisfies $\rho_{\mathrm{e}}(A) \geq \rho_{\mathrm{e}}(B)$.*

*Proof.* There are three cases to consider. i) $\mathrm{e} \in X \setminus B$: The inequality holds by definition of a submodular function. ii) $\mathrm{e} \in A$: Then both sides of the inequality equal zero and the inequality holds again. iii) $\mathrm{e} \in B \setminus A$: Then $\rho_{\mathrm{e}}(B) = 0$, and since $f$ is monotone, $\rho_{\mathrm{e}}(A)$ is non-negative. ■

Proposition 1.5.2 extends to a set, i.e., monotonicity of $f$ implies that for every $A \subseteq B \subseteq X$, and $Q \subseteq X$, $\rho_Q(A) \geq \rho_Q(B)$. Informally Proposition 1.5.2 states that if $f$ is monotone then the diminishing marginal gain property holds for every subset of $X$.

We are interested in maximizing a monotone submodular function with $b$-MATCHING constraints. Let $G(V, E, W)$ be a simple, undirected, and edge-weighted graph, where $V, E, W$ are the set of vertices, edges, and non-negative edge weights, respectively. For each $\mathrm{e} \in E$ we define a variable $x(\mathrm{e})$ that takes values from $\{0, 1\}$. Let $M \subseteq E$ denote the set of edges for which $x(\mathrm{e})$ is equal to 1, and let $\delta(v)$ denote the set of edges incident on the vertex $v \in V$. The submodular $b$-MATCHING problem is

$$\begin{aligned} &\max f(M) \\ &\text{subject to} \\ &\sum_{\mathrm{e}\in\delta(v)} x(\mathrm{e}) \leq b(v) \quad \forall v \in V, \\ &x(\mathrm{e}) \in \{0, 1\}. \end{aligned} \tag{1.2}$$

Here $f$ is a non-negative monotone submodular set function, and $0 \leq b(v) \leq |\delta(v)|$ is the integer degree bound on $v$. Denote $\beta = \max_{v\in V} b(v)$.

We now consider the $b$-MATCHING problem on a bipartite graph with two parts in the vertex set, say, $U$ and $V$, where the objective function is a concave polynomial.

$$\begin{aligned} f = \max &\sum_{u\in U} \left( \sum_{\mathrm{e}\in\delta(u)} W(\mathrm{e}) x(\mathrm{e}) \right)^{\alpha} \\ &+ \sum_{v\in V} \left( \sum_{\mathrm{e}\in\delta(v)} W(\mathrm{e}) x(\mathrm{e}) \right)^{\alpha} \end{aligned} \tag{1.3}$$

subject to

$$\begin{aligned} &\sum_{\mathrm{e}\in\delta(u)} x(\mathrm{e}) \leq b(u) \quad \forall u \in U, \\ &\sum_{\mathrm{e}\in\delta(v)} x(\mathrm{e}) \leq b(v) \quad \forall v \in V, \\ &x(\mathrm{e}) \in \{0, 1\}. \end{aligned}$$

The objective function Problem 1.3 becomes submodular when $\alpha \in [0, 1]$. This formulation has been used for peptide identification in tandem mass spectrometry [37], [38], and word alignment in natural language processing [39].

### 1.5.2 Complexity of Submodular $b$-Matching and Approximation

A *subset system* is a pair $(X, \mathcal{I})$, where $X$ is a finite set of elements and $\mathcal{I}$ is a collection of subsets of $X$ with the property that if $A \in \mathcal{I}$ and $A \subseteq A$ then $A \in \mathcal{I}$. A *matroid* is a subset system $(X, \mathcal{I})$ which satisfies the property that $\forall A, B \in \mathcal{I}$ and $|A| < |B|$, $\exists \mathrm{e} \in B \setminus A$ such that $A \cup \{\mathrm{e}\} \in \mathcal{I}$. Here the sets in $\mathcal{I}$ are called *independent sets*. A *subset system* is *$k$-extendible* [26] if the following holds: let $A \subseteq B$, $A, B \in \mathcal{I}$ and $A \cup \{\mathrm{e}\} \in \mathcal{I}$, where $\mathrm{e} \notin A$, then there is a set $Y \subseteq B \setminus A$ such that $|Y| \leq k$ and $B \setminus Y \cup \{\mathrm{e}\} \in \mathcal{I}$.

Maximizing a monotone submodular function with constraints is NP-hard in general; indeed, it is NP-hard for the simplest constraint of cardinality for many classes of submodular functions [40], [41]. A Greedy algorithm that repeatedly chooses an element with the maximum marginal gain is known to achieve $(1 - 1/\mathrm{e})$-approximation ratio [42], and this is tight [43]. The Greedy algorithm with matroid constraints is 1/2-approximate. More generally, with $k$-matroid intersection and $k$-extendible system constraints, the approximation ratio of the Greedy algorithm becomes $1/(k + 1)$ [44].

## 1.6 Related Work on Submodular b-matching

The maximum $k$-cover problem can be reduced to submodular $b$-Matching [45]. Feige [40] showed that there is no polynomial time algorithm for approximating the max $k$-cover within a factor of $(1 - 1/\mathrm{e} + \epsilon)$ for any $\epsilon > 0$. Thus we obtain an immediate bound on the approximation ratio of submodular $b$-Matching.

New approximation techniques have been developed to expedite the greedy algorithm, especially for cardinality and matroid constraints. The continuous greedy approach for cardinality and matroid constraints has been introduced in [44]. In [46], the authors improve the continuous greedy technique to develop faster algorithms for cardinality and intersection of $p$-system and $l$-knapsack constraints. Mirzasoleiman et al. developed a randomized greedy

algorithm that is faster than lazy greedy and achieves $(1-1/\mathrm{e})$ approximation in expectation [47]. They also developed a distributed two-round algorithm for the cardinality constraints in [48]. Under certain conditions, this algorithm achieves an approximation ratio as good as the centralized greedy algorithm. But for more general constraints such as matroid or $k$-system, the approximation ratio depends on the number of distributed nodes and the maximum independent set size. [49] gave the first deterministic $1/2+\epsilon$ algorithm for submodular maximization with matroid constraints. The running time of their algorithm is $O(np^2+pT)$, where $n, p$ are the size of the ground set and rank of the matroids, respectively, and $T$ is the time required to compute a maximum weight perfect matching in a complete bipartite graph with $2p$ vertices.

Surveys on submodular function maximization under different constraints may be found in [50]–[52].

Several approximation algorithms have been proposed for maximizing monotone submodular functions with $b$-Matching constraints. If the graph is bipartite, then the $b$-Matching constraint can be represented as the intersection of two partition matroids, and the Greedy algorithm provides a 1/3-approximation ratio. But $b$-Matching forms a 2-extendible system and the Greedy algorithm yields a 1/3-approximation ratio for non-bipartite graphs. Feldman *et al.* [53] showed that $b$-Matching is also a 2-exchange system, and they provide a $1/(2+\frac{1}{p}+\epsilon)$-approximation algorithm based on local search, with time complexity $O(\beta^{p+1}(\Delta-1)^p nm\epsilon^{-1})$. (Here $p$ is a parameter to be chosen.) The continuous greedy and randomized LP rounding algorithms have been used in [54] to compute a submodular $b$-Matching algorithm that produces a $(\frac{1}{3+2\epsilon})(1-\frac{1}{\mathrm{e}})$ approximate solution in $O(m^5)$ time.

Recently Fuji [45] developed two algorithms for the problem. One of these, Find Walk, is a modified version of the Path Growing approximation algorithm [27] proposed for 1-matching with linear weights. Mestre [26] extended the idea to $b$-Matching. In [45], Fuji extended this further to the submodular objectives. They showed an approximation ratio of 1/4 with time complexity $O(\beta\, m)$. The second algorithm uses randomized local search, has an approximation ratio of $1/(2+\frac{1}{p})-\epsilon$, and runs in $O(\beta^{p+1}(\Delta-1)^{p-1}m\log\frac{1}{\epsilon}$ time in expectation. Here a vertex is chosen uniformly at random in each iteration, and the algorithm searches for a certain length alternating path with increasing weights. This

algorithm is similar to the $2/3 - \epsilon$ approximation algorithm for linear weighted matching in [55] and the corresponding $b$-MATCHING variant in [26]. We list several approximation algorithms for submodular $b$-MATCHING in Table 1.1.

**Table 1.1.** Algorithms for the submodular $b$-MATCHING problem. The last column lists if the algorithm is concurrent or not.

| Alg. | Appx. Ratio | Time | Conc.? |
|---|---|---|---|
| Greedy[42] | 1/3 | $O(\beta nm)$ | N |
| Lazy Greedy [56] | 1/3 | $O(\beta m \log m)$ assuming 1 | N |
| Local Search [53] | $1/(2 + \frac{1}{p} + \epsilon)$ | $O(\beta^{p+1}(\Delta - 1)^p nm\epsilon^{-1})$ | N |
| Cont. Grdy+ Rand. Round[54] | $(\frac{1}{3+2\epsilon})(1 - \frac{1}{\mathrm{e}})$ | $O(m^5)$ | N |
| Path Growing [45] | 1/4 | $O(\beta m)$ | N |
| Rand LS [45] | $1/3 - \epsilon$ | $O(\beta^2 m \log 1/\epsilon)$ in expectation | N |
| Local Lazy Greedy | $\frac{\epsilon}{2+\epsilon}$ | $O(\beta m \log \Delta)$ assuming 1 | Y |

## 1.7 Contribution of the thesis

This thesis makes two major contributions. The first contribution is to develop efficient, simple to implement and concurrent approximation algorithms for various DCS problems. The second contribution is to apply the DCS problem to real-life applications. We highlight these contributions as follows.

- Algorithms
  - We develop, analyze and implement several approximation algorithms for $b$-edge cover.
    - We improved the greedy algorithm using lazy evaluation (Chap. 2.1.1), and show that $k$-NN algorithm is 2-approximate (Chap. 2.2).

    - We developed a primal-dual 3/2-approximate algorithm for $b$-Edge Cover. We also analyzed existing 2- and $\Delta$-approximate algorithms using the primal dual framework (Chap. 3)
    - We employ a reduction from $b$-Edge Cover to $b$-Matching to construct a highly parallel 2-approximation algorithm. We have also shown that the algorithm has logarithmic parallel depth if the weights are chosen uniformly at random (Chap. 4.1).
    - We show a novel reduction from $b$-Edge Cover to a constrained perfect $b$-Matching and developed a 3/2-approximate algorithm using this reduction (Chap. 4.2).
  - We design and implement Local Lazy Greedy ,a novel parallel approximation algorithm for submodular $b$-Matching. This algorithm is 1/3-approximate and runs in $O(m\beta \log n$ time in serial (Chap. 5).
  - We introduce Lambda Matching that provides a trade-off between the weights and cardinality of a matching. We show theoretically and empirically the effectiveness of Lambda Matching to find a matching with large weight and high cardinality (Chap. 6).

- Applications
  - We employ submodular $b$-Matching to generate a balanced assignment of tasks to processors for building Fock matrices in quantum chemistry within the NWChemEx software. Our load-balanced assignment results in a four-fold speedup per iteration of the Fock matrix computation, and scales to $14,000$ cores of the Summit supercomputer at ORNL (Chap. 8)
  - We propose the first shared-memory as well as distributed memory parallel algorithms for the *adaptive anonymity* problem using an iterative algorithm where each iteration solves a 2-approximate $b$-Edge Cover. We are able to solve adaptive anonymity problems with hundreds of thousands of instances and hundreds of features on a supercomputer in under five minutes. (Chap. 7)

– We propose to apply DCS problem to construct graphs from dataset in machine learning problems. We provide a mathematical optimization framework supporting the applicability of DCS for the graph construction problem. Using real world data, We report preliminary experiments that confirm that graphs of high quality are constructed (Chap. 9).

# 2. GREEDY AND LOCAL ALGORITHMS FOR $b$-EDGE COVER

GREEDY is one of the most common algorithmic paradigms. In the greedy paradigm, an element is added to the partial solution set based on the importance of the element. This importance criterion could be precomputed or computed at runtime. We describe in this chapter different greedy algorithms for $b$-EDGE COVER. We will first discuss the algorithms that require global orderings of edges. Next, we will relax the global ordering to design local algorithms that need only the information from the neighborhood of an edge. These latter algorithms are advantageous in parallel environments.

## 2.1 GREEDY and LAZY GREEDY

Given a universal set $U$, and a collection of subsets $S$ of $U$, a *set cover* $C \subseteq S$ is a subcollection of the sets whose union is $U$. The set cover problem aims to find a minimum cardinality set cover. If additionally each set $s \in S$ has a non-negative weight specified, the weighted set cover problem is to find a set cover of minimum total weight. The Greedy algorithm and its analysis for the unweighted set cover is by Johnson [57], Lovász [58], and Stein [59]. The greedy algorithm for the weighted set cover problem is analyzed by Chvátal [60]. The $b$-EDGE COVER problem is a special case of the set multicover problem where the elements correspond to vertices of a graph, and subsets to its edges consisting of pairs of vertices.

Given an edge cover, we say a vertex, $v$ is saturated w.r.t to the cover if the cover contains at least $b(v)$ edges incident on $v$, otherwise we call $v$ unsaturated. The *effective weight* of an edge is defined as the weight of the edge divided by the number of its unsaturated endpoints. The GREEDY algorithm for minimum weighted edge cover works as follows. Initially, no vertices are saturated, and the effective weight of every edge is half of its edge weight. At each iteration the algorithm chooses an edge of minimum effective weight and adds it to the cover. It then decrements the $b(.)$ values of the endpoints of this edge by one and updates the effective weights of its neighboring edges. For the effective weight update, there are three possibilities for each edge: i) none of its endpoints is saturated, and there is no change in its effective weight, ii) one of the endpoints is saturated, and its effective weight doubles or iii)

both endpoints are saturated, its effective weight becomes infinite, and the edge is marked as deleted. The algorithm iterates until all vertices are saturated. The algorithm is described in Algorithm 1.

**Algorithm 1** GREEDY-$b$-EDGE COVER ($G = (V, E, w, b)$)

```
C = ∅
Compute effective weight of all edges e ∈ E
while there exists an unsaturated vertex do
    Add an edge of minimum effective weight e = (u, v) to C
    Delete e
    Decrement b(u) and b(v) by one
    for x ∈ {u, v} do
        if b(x) = 0 then update effective weights of edges incident on x
            Delete any edge (x, y) with b(y) = 0
        end if
    end for
end while
return C
```

Next, we prove the approximation ratio of the greedy $b$-EDGE COVER algorithm shown in Algorithm 1. Note that the $b$-EDGE COVERis a special case of the set multicover problem. The algorithm 1 necessarily a greedy multiset cover algorithm specialized for $b$-EDGE COVER. From [61, Thm. 3.1], it is known the greedy multiset cover is $H(a)$-approximate, where $a$ is the maximum size of a subset. In the $b$-EDGE COVER $a = 2$, resulting in 3/2-approximation. Nevertheless, we will provide a direct proof of the 3/2-approximation next and will show a different proof using the primal-dual technique in Chapter 3.

**Lemma 2.1.1.** *The* GREEDY-$b$-EDGE COVER IS 3/2-APPROXIMATE.

*Proof.* The total weight of the $b$-EDGE COVER $C$ is denoted by $w(C)$. When the greedy algorithm chooses an edge e, let us assume it charges the price per vertex that is newly covered by e to satisfy the $b(.)$ covering constraints of the vertices. This charge is the current effective weight of e. Since a vertex $v$ is charged exactly $b(v)$ times, the sum of the prices of all the vertices is the weight of the greedy edge cover. Formally, let $price(u, \mathrm{i})$ be the price of $u$ when it is charged for the ith time during the greedy algorithm. Then the total weight of

the greedy $b$-Edge Cover is $w(C) = \sum_{v \in V} \sum_{\mathrm{i}=1}^{b(v)} price(v, \mathrm{i})$. Also assume that the price values for each vertex (stored in a list $price(.)$) are sorted in descending order.

Now we scan the edges in an optimal edge cover $C_*$.During the scanning process we also maintain an array of counters (initialized to zero) of size $|V|$ to update the saturation of the vertices. For an edge $\mathrm{e}^* = \{x, y\}$ in an optimal cover $C_*$, w.l.o.g assume that the greedy algorithm saturates $x$ and then $y$. When $x$ is saturated by the greedy algorithm, both $x$ and $y$ were unsaturated. So, the greedy algorithm pays at most $w(\mathrm{e}^*)/2$ for covering one of the $b(x)$ constraints. If the greedy algorithm were to choose $\mathrm{e}^*$, one of the price values of $x$ would be $w(\mathrm{e}^*)/2$. So, there must be a unique $price(x, k)$, where $k \in [b(x)]$ that satisfies $price(x, k) \leq w(\mathrm{e}^*)/2$. To find that, we assign the first price value of the $price(x)$ list (say it is $k$) that is smaller than or equal to $w(\mathrm{e}^*)/2$, and mark this $price(x, k)$ as unavailable. After we assign the price values of $x$, we increase the counter for $x$. Once the counter reaches $b(v)$ for some $v \in V$, we make all the price values available for that vertex. At this point for any future assignment of an optimal edge e, we just find a $price(x, k)$ in the price list that is smaller than or equal to $w(\mathrm{e})/2$. Note that at this case we do not require uniqueness. For $y$ we can repeat the same argument as given above, but now for $w(\mathrm{e}^*)$. So the total charge of the vertices of $\mathrm{e}^*$ paid during the greedy algorithm through the price values is at most $\frac{3}{2}w(\mathrm{e}^*)$. The above charging mechanism ensures that each vertex in the optimal edge cover is charged at least $b(v)$ times, of which the first $b(v)$ times are unique. Summing over all $\mathrm{e}^* \in C_*$, the total amount charged by the greedy algorithm to the vertices is at most $\sum_{\mathrm{e}^* \in C_*} \frac{3}{2}w(\mathrm{e}^*)$. Now,

$$\frac{3}{2} \sum_{\mathrm{e}^* \in C_*} w(\mathrm{e}^*) = \frac{3}{2} w(C_*) \geq \sum_{v \in V} \sum_{\mathrm{i}=1}^{b(v)} price(v, \mathrm{i}) = w(C).$$

That concludes the proof. ■

### 2.1.1 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 updating 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 [56].

The pseudocode of the Lazy Greedy algorithm is presented in Algorithm 2. 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 unsaturated. 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() function 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 $newTop.w$, 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.

Next we compute the approximation ratio of the algorithm.

**Lemma 2.1.2.** *The approximation ratio of the* Lazy Greedy *algorithm is* $3/2$*, and the runtime is* $O(m \log m)$*.*

*Proof.* The invariant in the Greedy algorithm is that we select an edge that has minimum effective weight over all edges at every iteration. Now consider an edge $x$ chosen by the Lazy Greedy algorithm in some iteration. According to the algorithm, the updated

**Algorithm 2** LAZY GREEDY($G(V, E, w)$)

```
1: C = ∅;                                              ▷ the edge cover
2: c = Array of size |V| initialized to 0;     ▷ indicates the saturation level of the vertex
3: PrQ = makeHeap(E)                                   ▷ Create a min heap from E
4: while there exists an unsaturated vertex do
5:     top = PrQ.deQueue()
6:     Update effective weight of top edge,
7:      assign to top.w
8:     if top.w < ∞ then
9:         newTop = PrQ.front()
10:        if top.w ≤ newTop.w then
11:            C = C ∪ top
12:            Increment c(u) and c(v) by 1
13:        else
14:            PrQ.enQueue(top)
15:        end if
16:    end if
17: end while
18: return C
```

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 $O(m \log m)$, 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 $O(\log m)$. ■

The efficiency of the LAZY GREEDY algorithm comes from the fact that we do not need to update effective weights of the edges adjacent to the selected edge in each iteration. But the price we pay is the logarithmic-cost enQueue() and deQueue() operations.

### 2.2 $b$-NEAREST NEIGHBOR algorithm

Now we turn to greedy algorithms that works on local neighborhood. The simplest of these algorithms is a nearest neighbor algorithm described next. The $b(v)$-nearest neighbor

of a vertex $v$ in a graph is the least weighted $b(v)$ edges of adjacent to it. A simple approach to obtain an edge cover is the following: For each vertex $v$, insert the $b(v)$-nearest neighbor edges into the cover. (We also call these $b(v)$ lightest edge incident on $v$.) This is the $b$-Nearest Neighbor algorithm (bNN) described in Algorithm 3.

**Algorithm 3** $b$-Nearest Neighbor$(G = (V, E, w, b))$

```
C = ∅
for each v ∈ V do
    E_v = b(v) lightest edges incident on v
    C = C ∪ E_v
end for
return C
```

**Lemma 2.2.1.** *The approximation ratio of the $b$-Nearest Neighbor algorithm is 2.*

*Proof.* Let the optimal edge cover be $C^*$ and the set of edges incident on $v$ in $C^*$ be $E^*_v$. Let $C$ and $E_v$ similarly defined for $b$-Nearest Neighbor. Since a vertex, $v$ needs to be covered by $b(v)$ incident edges, the best case for it to be covered by the least weight $b(v)$ edges. So, if we sum the weights of the incident edges of $v$ chosen by the optimal algorithm it will always be less or equal than the sum of weights of the $b(v)$ ligtest edges, i.e., $w(E^*_v) \geq w(E_u)$. An edge can cover at most two endpoints, so an optimal edge $\mathrm{e}^*(u, v) \in C^*$ can be present in one of $E^*_u$ and $E^*_v$ or in both of them. So, $w(C^*) \geq \frac{1}{2}\sum_{v \in V} w(E^*_v) \geq \frac{1}{2}w(E_u) \geq \frac{1}{2}w(C)$. ■

**Lemma 2.2.2.** *The runtime of $b$-Nearest Neighbor algorithm is $O(m)$*

*Proof.* Given an adjacency list representation of a vertex $v$, we can use a worst-case linear time selection algorithm [62, Ch. 9] to find the $b(v)$-th lowest weight edge, say $\mathrm{e}^v_b$. To get the $b(v)$ cheapest edges, we scan through the adjacency list to find the edges that are strictly smaller than $w(\mathrm{e}^v_b)$. If the number of such edges is less than $b(v) - 1$, then we find necessary edges with weight $w(\mathrm{e}^v_b)$. This takes linear time w.r.t to the size of the adjacency list of a vertex. Summing over all the vertices, this step takes $O(m)$ time. ■

## 2.3 LSE and S-LSE

An edge is called locally subdominant at some time in the algorithm if its effective weight at this point is the minimum effective weight among all of its neighboring edges. An algorithm that chooses edges that are locally subdominant to add to the edge cover is called the LSE algorithm. Khan and Pothen [63] showed that such an algorithm is 3/2-approximate. Since effective weight is dynamic through out the runtime of the algorithm, we ask what would be the quality guarantee, if we work on the original weight instead of the effective weight. The S-LSE algorithm [64] iteratively computes a set of locally sub-dominant edges (w.r.t the original weight) to add to the edge cover. Ties are broken by prioritizing an edge with lower numbered endpoints. In each iteration locally sub-dominant edges are uniquely defined, and are independent of each other, i.e., they do not share an endpoint. The algorithm iteratively finds a set of locally sub-dominant edges, adds them to the edge cover and updates $b(v)$ values. These edges are marked as deleted from the graph, and new locally dominant edges are identified. If both endpoints of an edge have their $b(v)$ values satisfied, then it is marked as deleted from the graph. The algorithm is described in Algorithm 4.

At each iteration, we calculate the set of locally sub-dominant edges $S$ as follows. Each vertex $u$ sets a pointer to the edge of least weight incident on it. If the end-points of an edge point to each other, then the edge is locally sub-dominant. We pick each such edge, add it to the cover, remove it from further consideration, and decrement the $b(v)$ values of the end points. The time complexity of the (serial) algorithm is $O(m \log \Delta)$, where $\Delta$ is the maximum degree of a vertex.

During the algorithm, an edge $(u, v)$ covers at least one endpoint. It covers both if the both $u$ and $v$ were under saturated when $(u, v)$ was chosen, otherwise either $u$ or $v$ was under saturated.

**Lemma 2.3.1.** *The covering edges of a vertex $u$ in the solution are exactly the least weighted $b(u)$ edges.*

*Proof.* Let us assume that ties in edge weights are broken consistently. Let $c(u)$ denote the current saturation of $u$, which is initialized to zero. Consider the cases when an edge $(u, v)$ was added to the cover. There are two cases. In the first case, $(u, v)$ was added because

**Algorithm 4** S-LSE$(G(V, E, w), b)$

```
1: C = ∅
2: while b(.) constraints are not satisfied do
3:     Compute locally sub-dominant edges S of G
4:     for each (u, v) ∈ S do
5:         C = C ∪ (u, v)
6:         E = E \ (u, v)
7:         for x ∈ {u, v} do
8:             if b(x) > 0 then
9:                 b(x) = b(x) − 1
10:            end if
11:        end for
12:    end for
13: end while
14: return C
```

both $u$ and $v$ are under saturated; and in the second one of the endpoints (say $u$) is under saturated. In the first case, $(u, v)$ must be an edge that is one of the $b(u)$ and $b(v)$ least weighted edges incident on $u$ and $v$. In fact it is the $c(u) + 1$-th least weighted edge adjacent to $u$ and $c(v) + 1$-th least weighted edge adjacent to $v$, otherwise it cannot be a locally subdominant edge. We can repeat the argument when $(u, v)$ covers only one endpoint, say $u$, and show that it must be the $c(u)+1$-th least weight edge adjacent to $u$. This immediately provides us a decomposition of the cover $C$ into a set of $b(.)$ edges per vertex. Those edges must be the $b(.)$ least weighted edges incident to the vertex. ■

**Lemma 2.3.2.** S-LSE *is 2-approximate.*

*Proof.* Using the Lemma 2.3.1, we argue that the solution of the S-LSE is essentially the set of $b$ nearest neighbors. Using Lemma 2.2.1 the approximation ratio is immediate. ■

## 2.4 Improving $b$-Edge Cover empirically

All of the algorithms that we presented in this chapter may not produce minimal edge cover i.e., they may have redundant edges in the solution. To see this consider a path graph of 4 vertices with equal weights on the edges, with $b(.)$ values set to 1. If the algorithms pick the middle edge first, they are forced to choose the 2 other edges making the middle edge

redundant. So, we can improve the edge covers computed by the algorithms by removing the redundant edges. In Chapter 4, we will describe an algorithm that produces minimal covers.

We can also preprocess the original graph by spasification without sacrificing the quality of the greedy algorithm. The preprocessing works by removing edges with certain weights but it does not guarantee any improvement on the theoretical time complexity. Let $\mathrm{e}_b^v$ be a $b(v)$th lowest weight edge incident on $v$. We have the following result.

**Lemma 2.4.1.** *Any edge* $\mathrm{e}(u,v) \in E$ *with weight* $w(\mathrm{e}) > 2(w(\mathrm{e}_b^u))$ *and* $w(\mathrm{e}) > 2(w(\mathrm{e}_b^v))$ *cannot be in the solution of the greedy algorithm.*

*Proof.* For such edge e, the effective weight is *at least* $w(\mathrm{e})/2$ which is greater than both $(w(\mathrm{e}_b^u))$ and $(w(\mathrm{e}_b^v))$. If any of the endpoints of e is unsaturated, it immediately follows that e always has a neighboring available edge whose effective weight is strictly greater than e's effective weight. Hence, e will not be in the solution of the greedy edge cover. ■

It is easy to verify that the Lemma 2.4.1 also applies to the Lazy Greedy the LSE algorithm, and the primal-dual algorithm presented in Chap 3.2. In fact the Lemma 2.4.1 applies to an optimal algorithm. Since if e were to be in an optimal edge cover, we could remove e and add *at most* two of the available $b(.)$ nearest neighbors of the endpoints resulting in a better solution. This provides us the following general result concerning an optimal algorithm.

**Lemma 2.4.2.** *Any edge* $\mathrm{e}(u,v) \in E$ *with weight* $w(\mathrm{e})$ *such that* $w(\mathrm{e}) > w(\mathrm{e}_b^u) + w(\mathrm{e}_b^v)$ *can not be in an optimal edge cover.*

# 3. LP BASED ALGORITHMS FOR $b$-EDGE COVER

We describe a mathematical programming formulation of the $b$-EDGE COVER problem to develop primal-dual framework for developing approximation algorithms for the problem. Using the primal-dual framework, we will describe a 3/2-approximation algorithm, analyze the $b$-NEAREST NEIGHBOR algorithm and show a different proof of 2-approximation, and describe a $\Delta$-approximation algorithm. These results have been published in [65].

## 3.1 Linear Programming Framework

We begin by describing the primal and dual linear programming (LP) formulations of the minimum weighted $b$-EDGE COVER problem. We consider a graph $G = (V, E, w)$ and $b : V \to \mathbb{N}$, where $w(.)$ denotes the non-negative weights on the edges, and we need to choose at least $b(v)$ edges incident on each vertex $v$. We will say that a vertex $v$ is *uncovered* or *unsaturated* if a current $b$-EDGE COVER has fewer than $b(v)$ edges in it.

Define a vector $\mathbf{x} \in \{0,1\}^m$ with the intent that $x(\mathrm{e}) = 1$ if the edge is in the cover, and 0 otherwise. Denote the set of edges incident on a vertex $v$ by $\delta(v)$ (i.e., the set of edges one of whose endpoints is $v$). The integer linear program (ILP) formulation of the minimum weighted edge cover problem is as follows.

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

The linear programming relaxation of the ILP is obtained by relaxing the binary constraint $x(\mathrm{e}) \in \{0,1\}$ to $0 \leq x(\mathrm{e}) \leq 1$. The relaxation is as follows.

$$\min \sum_{\mathrm{e}\in E} w(\mathrm{e})x(\mathrm{e}), \quad \text{subject to } \sum_{\mathrm{e}\in\delta(v)} x(\mathrm{e}) \geq b(v), \forall v \in V, \\ x_{\mathrm{e}} \geq 0, \ -x(\mathrm{e}) \geq -1, \forall \mathrm{e} \in E. \tag{3.2}$$

The final constraint, equivalent to $x(\mathrm{e}) \leq 1$, indicates that we may use each edge e at most once in the cover. Any $\mathbf{x}$ satisfying the constraints of this linear program is a feasible solution.

To construct the dual program of the relaxed linear program we define a dual variable for each of the constraints. We define two sets of variables, namely $\mathbf{y} \in \mathbb{R}^n_{\geq 0}$ and $\mathbf{z} \in \mathbb{R}^m_{\geq 0}$. The dual is as follows.

$$\begin{aligned} \max \sum_{v \in V} b(v)y(v) - \sum_{\mathrm{e} \in E} z(\mathrm{e}), \\ \text{subject to } y(\mathrm{i}) + y(\mathrm{j}) - z(\mathrm{e}) \leq w(\mathrm{e}), \forall \mathrm{e} = (\mathrm{i}, \mathrm{j}) \in E, \\ y(v), z(\mathrm{e}) \geq 0, \ \forall v \in V, \ \forall \mathrm{e} \in E. \end{aligned} \tag{3.3}$$

Again, any $\mathbf{y}$ and $\mathbf{z}$ satisfying the constraints of the dual program is dual feasible.

For 1-Edge Cover, the upper bound constraints on the primal variables for the LP relaxation shown in Eqn. 3.2, i.e., $x(\mathrm{e}) \leq 1 \ \forall \mathrm{e} \in E$, are not necessary. If in the solution there is any $x(\mathrm{e}) > 1$, we can change it to $x(\mathrm{e}) = 1$, producing a feasible solution with a lower objective value. For the $b$-Edge Cover formulation, without the upper bound constraints we cannot guarantee that the relaxation produces a solution in $[0, 1]$. Now consider the dual problem, Eqn. 3.3. The dual variable corresponding to each upper bound constraint on $x(\mathrm{e})$ in the primal is $z(\mathrm{e})$. These variables serve the same purpose as its counterpart constraints in the primal, by helping the program to choose distinct edges in the cover by providing enough slack to make some of the dual constraints feasible. We will discuss the role of $z(\mathrm{e})$ variables in our analysis of the algorithm.

Let $C^*$ be the objective value of an *optimum* $b$-Edge Cover solution, $C_{LP}$ the objective value of an *optimum* solution of the relaxed LP shown in Eqn. 3.2, and $C_{dual}$ the objective value of a *feasible* solution of the dual problem shown in Eqn. 3.3. Then from linear programming and weak duality theory we have

$$C_{dual} \leq C_{LP} \leq C^*. \tag{3.4}$$

### 3.1.1 Dual-Fitting Algorithms

Now let us assume $\mathbf{x}$ is a feasible *integral* solution to the primal linear program, and let $\mathbf{y_a}$, $\mathbf{z_a}$ be the *approximate dual* solutions to the corresponding dual program. We say these are *approximate dual* variables as they may not necessarily satisfy the dual constraints.

Suppose we have a hypothetical algorithm that satisfies the following two properties:.

**Property 3.1.1** (Paid in full)**.** *The algorithm finds these primal and approximate dual variables that maintain the equality of primal and approximate dual objective values, i.e.,*

$$\sum_{e \in E} w(e)x(e) = \sum_{v \in V} b(v)y_a(v) - \sum_{e \in E} z_a(e). \tag{3.5}$$

**Property 3.1.2** (Shrinking factor)**.** *Let $\alpha > 0$ be a constant such that $\mathbf{y} = \mathbf{y_a}/\alpha$ and $\mathbf{z} = \mathbf{z_a}/\alpha$ become dual feasible variables.*

We can prove that this hypothetical algorithm guarantees an $\alpha$-approximation. Replacing $\mathbf{y_a}$ and $\mathbf{z_a}$ in Eqn. 3.5, we have

$$\sum_{e \in E} w(e)x(e) = \alpha \cdot (\sum_{v \in V} b(v)y(e) - \sum_{e \in E} z(e)). \tag{3.6}$$

Since $\mathbf{y}$ and $\mathbf{z}$ are dual feasible, from Eqns. 3.4 and 3.6 we have,

$$\sum_{e \in E} w(e)x(e) = \alpha \cdot (\sum_{v \in V} b(v)y(v) - \sum_{e \in E} z(e)) \leq \alpha \cdot C_{LP} \leq \alpha \cdot C^*. \tag{3.7}$$

This proves the required $\alpha$-approximation guarantee. We will now show how to instantiate this hypothetical algorithm to obtain 3/2- and a 2-approximation algorithms for the $b$-Edge Cover.

## 3.2 A 3/2-Approximation Algorithm

Rajagopalan and Vaziraini [66] have employed dual fitting to design an algorithm for set multicover. The primal dual algorithm that we present for 3/2-approximation of $b$-Edge Cover is motivated by this algorithm. It also generalizes a primal dual Edge Cover

algorithm discussed in [67]. Our aim is to come up with suitable *approximate dual* variables such that the two properties mentioned earlier are satisfied. We first define a few concepts and variables required to understand the algorithm and its analysis.

- An *unsaturated* vertex $v$ is *covered* by one of its incident edges e if during the execution of the algorithm e is selected to cover that vertex. This e is called a *covering edge* of the vertex $v$. Note that after covering a vertex $v$ by e it may still be *unsaturated*. Note also that a $b$-Edge Cover might include edges incident on $v$ that are not covering edges of $v$, since an edge $(u, v)$ may have been chosen as a covering edge of $u$ but not $v$. We denote by $S_v$ the set of covering edges of $v$.

- In general during the run of the algorithm an edge e is *available* if it can cover at least one of its endpoints. We define a set $Q_{\mathrm{e}}$ to denote the endpoints that e covers, and hence $0 \leq |Q_{\mathrm{e}}| \leq 2$. The set $C$ includes the edges in the cover.

- The *effective weight* of an edge, effectiveweight(e), is defined as the ratio of the weight of the edge and the number of its *unsaturated* endpoints. The effective weight of an edge can be thought of as the *price* the algorithm needs to pay to cover its unsaturated endpoints. Hence we define price$(v, \mathrm{e})$ as the effective weight of e where $v$ is an unsaturated endpoint of e. When an edge e is included in the cover, we fix the price$(v, \mathrm{e})$ value(s) of the endpoint(s) it covers.

- Let $r(v)$ be a variable defined on each vertex $v$. We call it the *dynamic requirement of saturation* since this variable will let us know whether the vertex $v$ is already saturated or not. The $r(v)$ values are initialized to the $b(v)$ values.

The output of the algorithm is a set of edges $C$. We can derive an integral primal solution from $C$ by setting $x_{\mathrm{e}} = 1 \; \forall \mathrm{e} \in C$, and $x_{\mathrm{e}} = 0 \; \forall \mathrm{e} \in E \setminus C$. We now introduce max_price$(v)$, a non-negative variable defined on each vertex $v$, which is equivalent to the approximate dual variable $y_a(v)$. During the execution of the algorithm we set

$$\text{max\_cprice}(v) = r(v)\text{-th lowest effective weight among the edges incident on } v.$$

We call this variable the current maximum price of the vertex. max_price($v$) is then defined as the maximum of all max_cprice($v$) during the execution of the algorithm. We create another non-negative variable excess(e) equivalent to $z_a$(e) for each edge e. Unlike the **max_price**, the **excess** variable is not necessary during the run of our algorithm, but its importance lies in the proof analysis. Hence we will defer its definition till then.

The pseudocode of the algorithm is shown in Algorithm 5. The algorithm iterates until all the vertices become saturated. In each iteration, there are two phases, the PRICE ASSIGNMENT phase, and the AUGMENT COVER phase. The PRICE ASSIGNMENT phase computes the max_cprice($v$) values of each unsaturated vertex. These values are used by the AUGMENT COVER phase to add as many edges to the cover as possible.

**Algorithm 5** PRIMAL DUAL($G = (V, E, w), b$)

```
C = ∅
while there exists an unsaturated vertex do
    Call PRICE ASSIGNMENT(G(V, E, w), b,max_cprice)
    for each v ∈ V, max_price(v) = max{max_cprice(v), max_price(v)} end for
    Call AUGMENT COVER (G(V, E, w), b, max_cprice, C, r)
end while
return C
```

We provide the pseudocode for the PRICE ASSIGNMENT phase in Algorithm 6.

**Algorithm 6** PRICE ASSIGNMENT($G = (V, E, w), b,$ max_cprice)

```
for each v ∈ V do
    if v is unsaturated then
        max_cprice(v) = {effectiveweight(e) : effectiveweight(e) is the
            r(v)-th lowest effective weight of an edge incident on v}
    end if
end for
```

The second phase of the algorithm, the AUGMENT COVER phase, adds edges to the edge cover using the **max_cprice** information set by the first phase. The pseudo-code for the AUGMENT COVER phase is presented in Algorithm 7. This phase scans the edges to find eligible ones to add in the cover. An edge e is selected as follows. If the edge e = (i, j) covers both of its endpoints and if its effective weight is less than or equal to the

max_cprice(.) values of both of its endpoints, then it would be included in the cover. Upon finding such an edge e the algorithm fixes the values of price(i, e) and price(j, e) to the value of effectiveweight(e). Note that the equation price(i, e) + price(j, e) = $w$(e) is then satisfied by this edge. On the other hand if the edge e covers only one endpoint $u$, to be included in the cover its effective weight must be less than or equal to the max_cprice(u) value. In this case we fix price($u$, e) to be $w$(e) so that the equation price($u$, e) = $w$(e) holds. Whenever we add an edge to the cover, we mark it as deleted and update the $r(v)$ values of its endpoints. (We update both $r(u)$ and $r(v)$ when an edge $(u, v)$ is deleted for identifying redundant edges in a post-processing.)

**Algorithm 7** AUGMENT COVER($G = (V, E, w, b)$, max_cprice, price,$C, r$)

```
1: for each e = (u, v) ∈ E do
2:     if u and v are both covered then
3:         Mark (u, v) as deleted
4:         Continue
5:     end if
6:     if u and v are both uncovered and effectiveweight(e) ≤
           {max_cprice(u), max_price(v)} then
7:         Set price(u, e) and price(v, e) to effectiveweight(e)
8:         C = C ∪ (u, v)
9:         Decrease r(u) and r(v) by 1
10:        Mark (u, v) as deleted
11:    else if only u is uncovered and effectiveweight(e) ≤ max_cprice(u) then
12:        Set price(u, e) to effectiveweight(e)
13:        C = C ∪ (u, v)
14:        Decrease r(u) and r(v) by 1
15:        Mark (u, v) as deleted
16:    else if only v is uncovered and effectiveweight(e) ≤ max_cprice(v) then
17:        Set price(v, e) to effectiveweight(e)
18:        C = C ∪ (u, v)
19:        Decrease r(u) and r(v) by 1
20:        Mark (u, v) as deleted
21:    end if
22: end for
```

Once the algorithm terminates, we have settled the price($v$, e) values for each vertex and covering edge pair. We also have updated max_price($v$) values for each vertex. For each vertex $v$ there are two kinds of edges incident on $v$. One kind is the set of *covering* edges,

which were the edges used to cover vertex $v$. The second kind would be other edges incident on $v$ that were necessary to cover the other endpoint $w$ of an edge $(v, w)$. Let $S_v$ denote the set of covering edges incident on each vertex $v$, and note that $|S_v| = b(v)$. Observe also that when the algorithm terminates, the value of max_price$(v) = \max\{\text{price}(v, \text{e}) : \text{e} \in S_v\}$.

We have not yet defined how we get the other set of approximate dual variables, i.e., **excess**. These variables are not necessary for the execution of algorithm, but needed for the proof analysis. We motivate this variable by means of a small example.

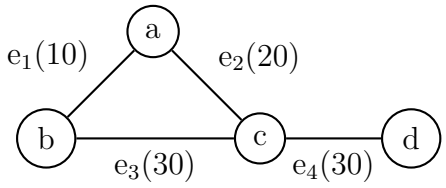



**Figure 3.1.** A small graph whose $b$-Edge Cover is to be computed.

*Example* 1. Consider the graph with four vertices $a$,$b$,$c$ and $d$ in Fig. 3.1, with the edge labels and weights also shown. The $b(v)$ value is 2 for each vertex $v$ except $d$, for which it is 1. The optimal edge cover is the graph itself. We run the primal dual algorithm on this problem. First in the Price Assignment phase, we assign the price values of each vertex and edge pair. Here price$(a, \text{e}_1)$ = price$(b, \text{e}_1) = 5$, price$(a, \text{e}_2)$ = *price*$(c, \text{e}_2) = 10$, price$(b, \text{e}_3)$ = price$(c, \text{e}_3) = 15$, and price$(c, \text{e}_4)$ = price$(d, \text{e}_4) = 15$. The max_cprice$(v)$ values are as follows: max_cprice$(a) = 15$, max_cprice$(b) = 10$, max_cprice$(c) = 15$ and max_cprice$(d) = 15$. In the next phase, suppose we scan through edges in the order $\text{e}_1, \text{e}_2, \text{e}_3$ and $\text{e}_4$. We select $\text{e}_1$ since the effective weight of this edge is less than the max_cprice(.) values for both endpoints. We decrease the $r(a)$ and $r(b)$ values by 1 and mark $\text{e}_1$ as deleted. Similarly we select $\text{e}_2$ and $\text{e}_3$. Note that now $a$, $b$ and $c$ are saturated. We cannot add $\text{e}_4$ in this phase because the effective weight of $\text{e}_4$, which is now 30, is greater than max_cprice$(d)$ which is 15. Next we start the second iteration. In the Price Assignment phase, we set max_cprice$(d)$ as 30 and in the Augment Cover phase we select $\text{e}_4$, since the effective weight of $\text{e}_4$ now equals max_cprice$(d)$.

At the termination of the algorithm, the max_price values for our example are as follows. max_price($a$) = 10, max_price($b$) = 15, max_price($c$) = 15 and max_price($d$) = 30. Let us consider the dual constraints defined in Eqn. 3.3. For $e_1$ the left side of the constraint using the approximate duals is, max_price($a$) + max_price($b$) − excess(e). But max_price($a$) + max_price($b$) = 25, which is much greater than weight of $e_1$. So we have a large excess on the summation that we need to balance. This is the purpose of the approximate dual, **excess**. If an edge e was not included in a cover we set excess(e) = 0. If an edge e = (i, j) covered both of its endpoints when e was added to the cover, we set

$$\text{excess(e)} = (\text{max\_price(i)} - \text{price(i, e)}) + (\text{max\_price(j)} - \text{price(j, e)}).$$

Otherwise if e covered only one endpoint i when it was added to the cover, then

$$\text{excess(e)} = \text{max\_price(i)} - \text{price(i, e)}.$$

We can restate this as

$$\text{excess(e)} = \sum_{q \in Q_e} (\text{max\_price}(q) - \text{price}(q, e)).$$

In the example, the **excess** values of the edges are as follows: excess($e_1$) = (10−5)+(15−5) = 15; excess($e_2$) = (10 − 10) + (15 − 10) = 5; excess($e_3$) = (15 − 15) + (15 − 15) = 0; and excess($e_4$) = 30−30 = 0. Observe that all of the dual constraints now become feasible except for $e_4$, where the left side of the constraint is max_price($c$) + max_price($d$) − excess(e) = 15 + 30 − 0 = 45, which is greater than the weight of the edge. We will show that we can scale the approximate duals in such a way that the scaled dual variables always satisfy the constraints.

**Lemma 3.2.1.** *The approximation ratio of the primal dual algorithm is 3/2.*

*Proof.* First note that by construction max_price(v) is non-negative; since it is the maximum of the price values of incident edges of a vertex, excess(e) is also non-negative. We need to show that assuming $\alpha = 3/2$, the paid in full and shrinking factor properties defined in

Section 3.1.1 are maintained. Using the approximate duals, **max_price** and **excess**, we will first show that the objective value of dual LP defined in Eqn. 3.3 equals the weight of the cover that we get from the algorithm. Let us first consider the right side of the (approximate) dual objective value i.e., $\sum_{e \in E} \text{excess(e)}$. Note that $\sum_{e \in E} \text{excess(e)} = \sum_{e \in C} \text{excess(e)}$, since excess(e) = 0 if e is not in the cover. Then we have,

$$\begin{aligned}
&\sum_{e \in C} \text{excess(e)} \\
&= \sum_{e \in C} \sum_{q \in Q_e} (\text{max_price}(q) - \text{price}(q, e)) \\
&= \sum_{e \in C} \sum_{q \in Q_e} \text{max_price}(q) - \sum_{e \in C} \sum_{q \in Q_e} \text{price}(q, e) \\
&= \sum_{v \in V} b(v) \cdot \text{max_price}(v) - \sum_{e \in C} w(e).
\end{aligned} \tag{3.8}$$

The second term in the last line of Eqn. 3.8 follows because $\sum_{q \in Q_e} \text{price}(q, e)$ is equal to $w(e)$, an invariant we maintain during the execution of Augment Cover phase. For the first term note that a particular vertex $v \in V$ will appear exactly $b(v)$ times in the sum.

Replacing $\sum_{e \in E} \text{excess(e)}$ in the objective value of the dual LP from Eqn. 3.3,

$$\begin{aligned}
&\sum_{v \in V} b(v) \cdot \text{max_price}(v) - \sum_{e \in E} \text{excess(e)} \\
&= \sum_{v \in V} b(v) \cdot \text{max_price}(v) - \sum_{v \in V} b(v) \cdot \text{max_price}(v) + \sum_{e \in C} w(e) \\
&= \sum_{e \in C} w(e).
\end{aligned} \tag{3.9}$$

But we cannot substitute **max_price** for **y** and **excess** for **z** because these are not dual feasible. Define $\alpha \equiv 3/2$. Set $\mathbf{y} = \mathbf{max\_price}/\alpha$ and $\mathbf{z} = \mathbf{excess}/\alpha$. We will show that the scaled variables **y** and **z** now become feasible. There are two scenarios to consider.

For the first scenario assume that an edge e belongs to the cover. We have two cases: Case 1. e = (i, j) covers both of its endpoints. Then replacing $y$(i), $y$(j) and $z$(e), the left side of the first constraint in Equation 3.3,

$$\begin{aligned}&\frac{1}{\alpha}\cdot(\text{max\_price(i)}+\text{max\_price(j)}-(\text{max\_price(i)}-\text{price(i,e)})\\&\quad-(\text{max\_price(j)}-\text{price(j,e)}))\\&=\frac{1}{\alpha}(\text{price(i,e)}+\text{price(j,e)})\leq\frac{1}{\alpha}w(\text{e})\leq w(\text{e}).\end{aligned}\tag{3.10}$$

The last line follows from the fact that during the algorithm we maintain

$$\text{price(i,e)}+\text{price(j,e)}=w(\text{e})$$

. Case 2. The edge e covers only one endpoint, say i. Using the definitions of $y$(i), $y$(j) and $z$(e) with e = (i, j), the left side of the constraint in Equation 3.3 becomes

$$\begin{aligned}&\frac{1}{\alpha}\cdot(\text{max\_price(i)}+\text{max\_price(j)}-(\text{max\_price(i)}-\text{price(i,e)}))\\&=\frac{1}{\alpha}(\text{max\_price(j)}+\text{price(i,e)}).\end{aligned}\tag{3.11}$$

From the algorithm, price(i, e) = $w$(e). When vertex j was saturated, vertex i was still unsaturated. We have not picked e as a *covering* edge for j. So all of the price values of the covering edge incident on j must be $\leq w(\text{e})/2$. Since max_price of a vertex is the maximum of the price values of the covering edges incident on that vertex, max_price(j) $\leq w(\text{e})/2$. So we have

$$\frac{1}{\alpha}\cdot(\text{max\_price(j)}+\text{price(i,e)})\leq\frac{1}{\alpha}\cdot\frac{3}{2}w(\text{e})\leq w(\text{e}).$$

We now consider the second scenario when e is not part of the $b$-Edge Cover and excess(e) = 0. So the left side of the constraint becomes

$$\frac{1}{\alpha}\cdot(\text{max\_price(i)}+\text{max\_price(j)}).$$

Without loss of generality assume i has become saturated first and then j. This immediately establishes that max_price(i) $\leq w(\mathrm{e})/2$ and max_price(j) $\leq w(\mathrm{e})$. Now we have

$$\frac{1}{\alpha} \cdot (\text{max\_price(i)} + \text{max\_price(j)}) \leq w(\mathrm{e}).$$

We combine the analysis as follows.

$$\begin{aligned}
&\sum_{\mathrm{e}\in C} w(\mathrm{e}) \\
&= \sum_{v\in V} b(v) \cdot \text{max_price}(v) - \sum_{\mathrm{e}\in E} \text{excess(e)} && \text{[from Eqn.3.9]} \\
&= \alpha \cdot (\sum_{v\in V} b(v)y(v) - \sum_{\mathrm{e}\in E} z(\mathrm{e})) && \text{[replacing } \textbf{max_price} \text{ and } \textbf{excess}\text{]} \\
&= \alpha \cdot C_{dual} \leq \alpha \cdot C_{LP} \leq \alpha \cdot C^*. && \text{[from Eqn. 3.6]}
\end{aligned}$$

This gives the 3/2-approximation ratio. ■

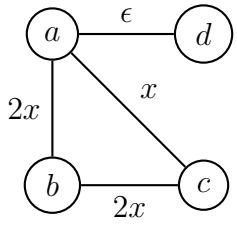



**Figure 3.2.** A tight example for primal-dual algorithm.

The approximation ratio is tight, and a tight example is the graph shown in Figure 3.2. Suppose $b = 1$ for each vertex. In the first iteration the primal-dual algorithm will add $(a, d)$ to the cover, and it can not add any other edge. In the second iteration since all the remaining edges have the same effective weight, it may add any one of the edges $(a, c)$, $(a.b)$ or $(b, c)$. Suppose it chooses the edge $(a, c)$. Then to cover the vertex $b$, it has to choose either $(a, b)$ or $(b, c)$, resulting in a cover with weight $3x + \epsilon$, whereas the optimal weight is $2x + \epsilon$. So as $\epsilon \to 0$, we get the approximation ratio 3/2.

Next we derive the time complexity of the algorithm. We can do a couple of optimizations on the general algorithm presented in Algorithm 5, which are as follows.

- During the AUGMENT COVER phase when an edge is selected we mark all the neighboring vertices of its covered endpoints as potential vertices. During the PRICE ASSIGNMENT phase need update the max_price values of only the potential vertices.

- During the PRICE ASSIGNMENT phase, when the max_price value changes, we mark all of its incident edges as potential covering edges. In AUGMENT COVER phase we can scan only these edges.

**Lemma 3.2.2.** *The time complexity of Algorithm 5 is* $O(\beta\,\Delta m)$.

*Proof.* Initially all the vertices and edges are marked as potential (see the optimization of the algorithm just mentioned). During the execution of the algorithm a vertex $v$ can be marked at most $\deg(v)$ times as potential. Each time it is marked it will have to find an edge incident on it with the $r(v)$-th minimum effective weight. One can find such an entry in $O(\beta\ \deg(v))$ time. Summing over all vertices we obtain $O(\beta\,\Delta m)$.

Similarly, during the PRICE ASSIGNMENT phase, an edge $\mathrm{e} = (\mathrm{i}, \mathrm{j})$ can be marked at most $\deg(\mathrm{i}) + \deg(\mathrm{j}) = O(\Delta)$ times. Summing over $m$ edges we obtain $O(\Delta m)$. Hence the total complexity is $O(\beta\,\Delta m)$. ■

### 3.3 A 2-Approximation Algorithm

We have presented in Algorithm 3 the $b$-NEAREST NEIGHBOR algorithm for finding a $b$-EDGE COVER. It is similar to the popular and well known $K$-NEAREST NEIGHBOR graph construction algorithm, used in many domains including machine learning and data mining to represent data by a sparse graph or to sparsify a graph. The difference between these problems is how the values of $k$ and $b$ are defined. In the former case $k$ is constant for all vertices while the latter case is more general, with the option to set user defined values of $b(v)$ for each vertex in the graph. This algorithm, like many other algorithms for the $b$-EDGE COVER problem, could have redundant edges in the cover, i.e., edges that could be removed while the residual edges form a $b$-EDGE COVER, thus resulting in an edge cover of lower weight. However, even without removing such edges, we can show that this algorithm gives us an approximate solution to the $b$-EDGE COVER problem, where the weight of the

cover is at most twice the optimal weight. In this section we will prove this result using the dual fitting framework developed in Sec. 3.1.1.

**Lemma 3.3.1.** *The approximation ratio of the b-*NEAREST NEIGHBOR *algorithm is 2.*

*Proof.* Let $\mathbf{x}$ be the primal integral solution and $C$ a $b$-EDGE COVER computed by the algorithm; hence $x(\mathrm{e}) = 1$, if $\mathrm{e} \in C$ and otherwise $x(\mathrm{e}) = 0$. Let $S_v$ denote the set of the $b(v)$ lightest edges incident on $v$. We will define a price value for each covering vertex and edge pair, and consider an edge $\mathrm{e} = (\mathrm{i}, \mathrm{j}) \in C$. If the weight of the edge e is among the lightest $b(\mathrm{i})$ edges incident on the vertex i **and** the lightest $b(\mathrm{j})$ edges incident on j, then we set $\mathrm{price}(\mathrm{i}, \mathrm{e}) = \mathrm{price}(\mathrm{j}, \mathrm{e}) = w(\mathrm{e})/2$. In this case we say the edge e covers both of its endpoints. Otherwise if e is only among the lightest $b(\mathrm{i})$ ($b(\mathrm{j})$) edges incident on i (j), we assign $\mathrm{price}(\mathrm{i}, \mathrm{e}) = w(\mathrm{e})$ ($\mathrm{price}(\mathrm{j}, \mathrm{e}) = w(\mathrm{e})$). In this case the edge e covers only one its endpoints. Next we set the approximate dual variables. We define for each vertex $v$, $\mathrm{max\_price}(v) = \max_{\mathrm{e} \in S_v} \mathrm{price}(v, \mathrm{e})$. For each edge $\mathrm{e} \in C$, let $Q_\mathrm{e}$ denote its covered endpoints. Note that $Q_\mathrm{e}$ may contain one or two vertices. We define for each $\mathrm{e} \in C$, $\mathrm{excess}(\mathrm{e}) = \sum_{q \in Q_\mathrm{e}} (\mathrm{max\_price}(q) - \mathrm{price}(q, \mathrm{e}))$. If an edge e is not included in the cover then we set $\mathrm{excess}(\mathrm{e}) = 0$. Note that since price values of a vertex and edge pair are always non-negative, max_price is non-negative. Again as $\mathrm{max\_price}(v) \geq \mathrm{price}(v, \mathrm{e}), \forall \mathrm{e} \in S_v$, the excess variable is also non-negative.

We will now show that with $\alpha = 2$, the two properties mentioned in Sec. 3.1.1 are satisfied by the approximate dual variables **max_price** and **excess**.

The first property is the equality of primal objective and approximate dual objective functions, and this follows directly from the corresponding proof in the 3/2-approximation algorithm described in Section 3.2.

For the second property, we will show that setting $y(v) = \mathrm{max\_price}(v)/\alpha$ and $z(\mathrm{e}) = \mathrm{excess}(\mathrm{e})/\alpha$ for $\alpha = 2$ make these dual feasible. We consider two scenarios for an edge $\mathrm{e} \in E$.

In the first scenario e belongs to the cover. Then replacing $y(v)$ and $z(\mathrm{e})$ on the left side of the first constraint of Eqn. 3.3, we obtain

$$\begin{aligned} y(\mathrm{i}) + y(\mathrm{j}) - z(\mathrm{e}) = \frac{1}{\alpha}(&\mathrm{max\_price}(\mathrm{i}) + \mathrm{max\_price}(\mathrm{j}) \\ &- \sum_{q \in Q_\mathrm{e}} (\mathrm{max\_price}(q) - \mathrm{price}(q, \mathrm{e}))). \end{aligned} \tag{3.12}$$

We have two cases to consider.

Case 1. The edge e covers both of the endpoints, and hence $Q_{\mathrm{e}} = \{\mathrm{i}, \mathrm{j}\}$. Recall that in this case e is among the lightest $b(\mathrm{i})$ edges incident on i, and the lightest $b(\mathrm{j})$ edges incident on j. The price values are then assigned as $\mathrm{price}(\mathrm{i}, \mathrm{e}) = \mathrm{price}(\mathrm{j}, \mathrm{e}) = w(\mathrm{e})/2$. Simplifying, we obtain

$$\begin{aligned} y(\mathrm{i})+y(\mathrm{j}) - z(\mathrm{e}) = \frac{1}{\alpha}&((\mathrm{max\_price(i)} + \mathrm{max\_price(j)}) \\ &- (\mathrm{max\_price(i)} - \mathrm{price(i,e)}) - (\mathrm{max\_price(j)} - \mathrm{price(j,e)})) \\ =&\frac{1}{\alpha}(\mathrm{price(i,e)} + \mathrm{price(j,e)}) \\ =&\frac{1}{\alpha}(w(\mathrm{e})/2 + w(\mathrm{e})/2) \leq w(\mathrm{e}). \end{aligned} \tag{3.13}$$

Case 2. e covers only one endpoint, say i. In this case we assign $\mathrm{price}(\mathrm{i}, \mathrm{e}) = w(\mathrm{e})$.

$$\begin{aligned} y(\mathrm{i}) + y(\mathrm{j}) - z(\mathrm{e}) =&\frac{1}{\alpha}(\mathrm{max\_price(i)} + \mathrm{max\_price(j)} \\ &- (\mathrm{max\_price(i)} - \mathrm{price(i,e)}) \\ =&\frac{1}{\alpha}(\mathrm{max\_price(j)} + \mathrm{price(i,e)}) \\ \leq&\frac{1}{\alpha}(w(\mathrm{e}) + w(\mathrm{e})) \ \leq \ \frac{1}{2}(2w(\mathrm{e})) \ \leq \ w_{\mathrm{e}}. \end{aligned} \tag{3.14}$$

The last line follows because $\mathrm{max\_price(j)} \leq w(\mathrm{e})$, since j is saturated and e is not a covering edge for j.

We now consider the second scenario when e is not part of the cover. In this case $\mathrm{excess(e)} = 0$, and the left side of the constraint becomes

$$\frac{1}{\alpha} \cdot (\mathrm{max\_price(i)} + \mathrm{max\_price(j)}).$$

Without loss of generality assume i was saturated first and then j. This establishes that $\mathrm{max\_price(i)} \leq w(\mathrm{e})$ and $\mathrm{max\_price(j)} \leq w(\mathrm{e})$ since i and j were covered by an edge with lower weight than that of e. We have

$$\frac{1}{\alpha} \cdot (\mathrm{max\_price(i)} + \mathrm{max\_price(j)}) \ \leq \ \frac{1}{2} \cdot 2w(\mathrm{e}) \ \leq \ w(\mathrm{e}).$$

We combine these analyses as follows.

$$\begin{aligned}\sum_{\mathrm{e}\in C} w(\mathrm{e}) &= \sum_{v\in V} b_v \cdot \mathrm{max\_price}(v) - \sum_{\mathrm{e}\in E} \mathrm{excess(e)} \quad \text{[from Eqn.3.9]} \\ &= \alpha \cdot (\sum_{v\in V} b(v)y(v) - \sum_{\mathrm{e}\in E} z(\mathrm{e})) \quad \text{[replacing } \textbf{max\_price} \text{ and } \textbf{excess}\text{]} \\ &= \alpha \cdot EC_{dual} \leq \alpha \cdot EC_{LP} \leq \alpha \cdot EC^*. \quad \text{[from Eqn. 3.6]}\end{aligned}$$

■

As in the 3/2-approximation algorithm we can also show the tightness of the approximation ratio of the $b$-Nearest Neighbor algorithm. We generate a graph with $n$ vertices, where $n$ is odd, as follows. The vertex $v_0$ is connected to all other vertices $v_1 \dots v_{(n-1)}$. Each vertex $v_\mathrm{i}$ for odd $\mathrm{i} > 0$ is connected with $v_{(\mathrm{i}+1)}$. All edges have the same weight $x$. We let $b = 1$ for every vertex. An example for $n = 9$ is shown in Figure 3.3.

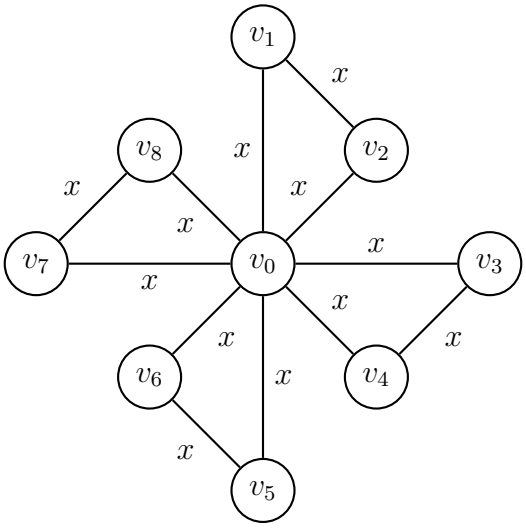



**Figure 3.3.** A tight example for the $b$-Nearest Neighbor algorithm

The optimal edge cover for this example would have weight $5x$, consisting of the four edges not incident on $v_0$ and one edge incident on $v_0$. But the $b$-Nearest Neighbor could produce an edge cover with weight $8x$ by choosing all edges incident on $v_0$. For a graph with $n$ vertices, the optimal weight would be $\frac{1}{2}(n-1) * x + x$ but the $b$-Nearest Neighbor

could produce an edge cover with weight $(n-1)*x$. Thus the approximation ratio is $\frac{2(n-1)}{n+1}$, which as $n \to \infty$ is 2.

The time complexity of the $b$-NEAREST NEIGHBOR algorithm is $O(m)$ as shown in Chpater 2.2.

### 3.4 $\Delta$-Approximation Algorithm

Here we present another algorithm based on linear programming duality for $b$-EDGE COVER, with an approximation ratio of $\Delta$, which is larger than the ratios 3/2 and 2 for the algorithms we have considered thus far. We do this for several reasons. First, the worst-case approximation ratio does not always determine how well an algorithm does practically. Second, the analysis of this algorithm enables us to present a different technique for designing a primal dual algorithm. This algorithm is derived from an algorithm for set multicover designed by [68], which leads to a better approximation ratio for vertex cover.

Recall that due to weak duality and LP relaxation, the objective value of any feasible solution to the dual problem in Eqn. 3.3 is a lower bound for the optimum $b$-EDGE COVER in Eqn. 3.1. But in general a dual feasible solution does not guarantee an approximation ratio. However, there exists a particular dual feasible solution, a *maximal dual feasible* solution, whose objective value provides a bound on the optimum value. A dual feasible solution (denoted $\bar{\mathbf{y}}$ and $\bar{\mathbf{z}}$) is *maximal* if it satisfies the following three properties.

1. There does not exist a feasible solution$(\mathbf{y}, \mathbf{z})$ with $y \geq \bar{y}$, $z \geq \bar{z}$ and

$$\sum_{v \in V} b(v)y(v) - \sum_{\mathrm{e} \in E} z(\mathrm{e}) > \sum_{v \in V} b(v)\bar{y}(v) - \sum_{\mathrm{e} \in E} \bar{z}(\mathrm{e}).$$

2. $\bar{z}(\mathrm{e}) = 0$ whenever $\bar{y}(\mathrm{i}) + \bar{y}(\mathrm{j}) < w(\mathrm{e})$.

3.

$$\sum_{v \in V} \bar{y}(v) \leq \sum_{v \in V} b(v)\bar{y}(v) - \sum_{\mathrm{e} \in E} \bar{z}(\mathrm{e}).$$

The proposed algorithm is as follows.

1. Find a *maximal* dual feasible solution ($\bar{\mathbf{y}}$,$\bar{\mathbf{z}}$).

2. Output the Cover $C = \{\mathrm{e} = (\mathrm{i}, \mathrm{j}) | \bar{y}(\mathrm{i}) + \bar{y}(\mathrm{j}) - \bar{z}(\mathrm{e}) = w(\mathrm{e})\}$.

We will first show that such an algorithm would provide a $\Delta$-approximation.

**Lemma 3.4.1.** *The algorithm is a $\Delta$-approximation algorithm for $b$-*Edge Cover.

*Proof.* We first establish that $C$ is a feasible cover. For the sake of contradiction, assume that $C$ is not a feasible cover; hence there exists a vertex $v$ that is not covered by at least $b(v)$ edges. Let

$$\epsilon = \min_{\mathrm{e} \in \delta(v)} \{\epsilon_{\mathrm{e}} = w(\mathrm{e}) - (\bar{y}(\mathrm{i}) + \bar{y}(\mathrm{j}) - \bar{z}(\mathrm{e})) \text{ and } \epsilon_{\mathrm{e}} > 0\}.$$

We show that the value of $\epsilon$ is well-defined. According to our assumption at most $b(v) - 1$ edges incident on $v$ are included in $C$. Since $\bar{y}$ and $\bar{z}$ are dual feasible, there must be at least one edge e where $\bar{y}(\mathrm{i}) + \bar{y}(\mathrm{j}) - \bar{z}(\mathrm{e}) < w(\mathrm{e})$, equivalently $w(\mathrm{e}) - (\bar{y}(\mathrm{i}) + \bar{y}(\mathrm{j}) - \bar{z}(\mathrm{e})) > 0$. We set $\bar{y}(v) = \bar{y}(v) + \epsilon$, and $\bar{z}(\mathrm{e}) = \bar{z}(\mathrm{e}) + \epsilon$, for edges $\mathrm{e} \in \delta(v)$ and $\mathrm{e} \in C$. The variables$(\bar{y}, \bar{z})$ are dual feasible. But this contradicts the maximality (property I) of $\bar{y}, \bar{z}$, since $\bar{y}$ increases the first term of the dual objective value by at least $b(v)\epsilon$, and $\bar{z}$ increases the second term by at most $(b(v) - 1)\epsilon$, with a net increase of at least $\epsilon$. This establishes that $C$ is a feasible cover.

Now since $(\bar{y}, \bar{z})$ is a dual feasible solution, the weak duality theorem applies.

$$\sum_{v \in V} b(v)\bar{y}(v) - \sum_{\mathrm{e} \in E} \bar{z}(\mathrm{e}) \leq C_{LP} \leq C^*. \tag{3.15}$$

Using Property III, we obtain

$$\sum_{v \in V} \bar{y}(v) \leq C^*. \tag{3.16}$$

From the construction of the cover we have

$$\begin{aligned}\sum_{\mathrm{e}\in C} w(\mathrm{e}) + \sum_{\mathrm{e}\in C} \bar{z}(\mathrm{e}) &= \sum_{\mathrm{e}=(\mathrm{i},\mathrm{j})\in C} \bar{y}(\mathrm{i}) + \bar{y}(\mathrm{j}) \\ &= \sum_{v\in V} \sum_{\mathrm{e}\in(\delta(v)\cap C)} \bar{y}(v) \\ &\leq \sum_{v\in V} |\delta(v)|\bar{y}(v) \\ &\leq \Delta \sum_{v\in V} \bar{y}(v) \leq \Delta \cdot C^*.\end{aligned} \tag{3.17}$$

In the last line, we have used Eqn. 3.16. Hence we have established the $\Delta$-approximation ratio for the proposed algorithm. ■

Next our goal is to design an algorithm that produces a maximal feasible dual solution. One such algorithm is shown on Algorithm 8.

**Algorithm 8** DUAL FEASIBLE($G = (V, E, w), b$)

```
Initialize y_v = 0 and z_e = 0, ∀v ∈ V and ∀e ∈ E
Assign w'(e) = w(e), ∀e ∈ E
while there exists an unsaturated vertex v ∈ V do
    f = arg min{w'(e) : e ∈ δ(v) − C}
    y(v) = y(v) + w'(f); C = C ∪ {f}
    for e ∈ δ(v) do
        w'(e) = w'(e) − w'(f)
        if w(e) < 0 then
            z(e) = z(e) − w'(e)
            w'(e) = 0
        end if
    end for
    decrease r(v) by 1
    if r(v) = 0 then
        Mark v as saturated
    end if
end while
```

The algorithm first initializes the dual variables $(\mathbf{y}, \mathbf{z})$ to zero. For each edge it maintains a variable, the *residual* weight $\mathbf{w}'$. This is initialized by the weight of the edge. In each iteration, it picks an unsaturated vertex, $v$, and then it finds an adjacent edge $f$ incident

to $v$ with minimum residual weight. Upon finding the edge $f$, it adds $f$ to the cover, adds $w'(f)$ to $y(v)$, and subtracts $w'(f)$ from the residual weights of all edges incident on $v$. Note that the iteration in line 6 of the algorithm goes over all edges incident on $v$, including $f$ and other edges that may have been added to the cover in earlier iterations. If the weight of any residual edge (say e) becomes negative, it subtracts $w'(\mathrm{e})$ from $z(\mathrm{e})$ (thus the value of $z(\mathrm{e})$ increases), and sets $w'(\mathrm{e})$ to zero. It then decreases the requirement $r(v)$ by 1, and marks $v$ as saturated if $r(v) = 0$.

The output of the algorithm is the set $C$. We will now show that the dual vectors derived are maximal and satisfy the three properties.

**Claim 1.** The variables **y** and **z** are non-negative.

*Proof.* The initial edge weights are non-negative, and the algorithm maintains the residual weight $w'$ to be non-negative. The variable $y$ is updated by adding $w'$ to it, and $z(\mathrm{e})$ is updated by subtracting $w'(\mathrm{e})$ when its value is negative, and hence these variables are non-negative as well. ■

**Claim 2.** All edges e in the cover $C$ satisfy $y(\mathrm{i}) + y(\mathrm{j}) - z(\mathrm{e}) = w(\mathrm{e})$.

*Proof.* Let i be a vertex at some iteration of the algorithm and $f = (\mathrm{i}, \mathrm{j})$ be an available edge with minimum residual weight $w'(f)$. We have $w'(f) \geq 0$ from the previous Claim, and then we add $w'(f)$ to $y(\mathrm{i})$. In the **for** loop over edges, we will now subtract $w'(f)$ from all edges incident on i, including the edge $f$. Hence $w'(f)$ is set to zero. Every time the weight of an edge is decreased by some amount in the algorithm, it is transferred to the $y(.)$-variable of one of its endpoints. Hence at this point in the algorithm, $y(\mathrm{i}) + y(\mathrm{j}) - z(\mathrm{e}) = w(f)$, since $z(f)$ is zero as long as $w'(f)$ is non-negative. In future iterations involving other available edges incident on i or j, the invariant $y(\mathrm{i}) + y(\mathrm{j}) - z(\mathrm{e}) = w(\mathrm{e})$ is maintained by increasing the value of $z(\mathrm{e})$. ■

**Claim 3.** The inequality $y(\mathrm{i}) + y(\mathrm{j}) - z(\mathrm{e}) \leq w(\mathrm{e})$ holds $\forall \mathrm{e} \in E \setminus C$.

*Proof.* Since $w'(\mathrm{e}) \geq 0$ for the edges not in the cover, the two endpoints of such edges have absorbed a weight of at most $w(\mathrm{e})$. Note that in this case $z(\mathrm{e}) = 0$. ■

So $(\mathbf{y}, \mathbf{z})$ is a dual feasible solution, but we need to show that it is also maximal.

**Lemma 3.4.2.** *The dual vector $(\mathbf{y}, \mathbf{z})$ of the* Dual Feasible *algorithm is maximal.*

*Proof.* Property I: Each vertex $v$ is covered by $b(v)$ edges for which the dual constraints $y(v)+y(u)-z((v,u) = w(v,u)$ are tight (according to Claim 1). Suppose we increase a dual variable $y(v)$ by a non-negative amount $\epsilon$. Now at least $b(v)$ constraints (those corresponding to the covering edges of $v$) are violated. (We say at least, since if there is an edge incident on $v$ that is not a covering edge with residual weight less than $\epsilon$, its constraint is also violated.) To compensate for the constraint violations, we need to add $\epsilon$ to at least $b(v)$ elements of $\mathbf{z}$. So the increase in objective function is exactly $b(v)\epsilon$ while the decrease is at least $b(v)\epsilon$. Hence the objective function value for $(y, z)$ is not greater than that of $(\bar{y}, \bar{z})$.

Property II: According to the construction, the residual weight $w'(\mathrm{e}) \geq 0$, $\forall \mathrm{e} \in E \setminus C$. That means for such edges e we have $y(\mathrm{i}) + y(\mathrm{j}) \leq w(\mathrm{e})$. Since $w'(\mathrm{e}) \geq 0$, line 8 of the algorithm will never be satisfied for e, resulting in $z(\mathrm{e}) = 0$.

Property III: Since $z(\mathrm{e}) = 0$, $\forall \mathrm{e} \in E \setminus C$, it suffices to show that

$$\sum_{\mathrm{e} \in C} z(\mathrm{e}) \leq \sum_{v \in V} (b(v) - 1)\, y(v).$$

We will prove this using induction on the number of iterations in the algorithm. Let $y^t$ and $z^t$ denote the variable $y$ and $z$ after $t$ iterations, and let the number of iterations in the algorithm be denoted by $T$. We will show that the inequality above is true in every iteration:

$$\sum_{\mathrm{e} \in C} z(\mathrm{e})^t \leq \sum_{v \in V} (b(v) - 1)\, y(v)^t, \quad t = 1, \ldots, T.$$

For $t = 1$, the left side is zero since the $w'(\mathrm{e})$ values for all the edges are non-negative after the first iteration, and the right side is $\geq 0$, since we must have identified an edge incident on a vertex $v$ with the minimum weight and added its weight to $y(v)$. Note that if $b(v)$ equals 1 the right side is zero, and otherwise it is greater than zero. We inductively assume that the inequality holds for $t = 1, \ldots, k-1$.

Denote the vertex selected at the $k$-th iteration by $p$, and let $f$ be the minimum weight edge incident on $p$ with weight $w'(f)$. Then the right hand side of the displayed equation

increases by $(b(p)-1)w'(f)$. Now this could lead to increase in some $z(\mathrm{e})^k$, where e is incident on $p$. When $f$ is included in the cover, there are at most $(b(p)-1)$ covering edges already incident on the vertex $p$. In the current iteration, only the $z(.)$ values of these edges can increase, and hence the net increase on the left side is at most $(b(p)-1)w'(f)$. Hence the inequality is preserved at the end of this iteration. ■

We can show a tight example for the $\Delta$-approximation algorithm by considering the graph in Fig. 3.3 with different weights as follows. The weights of the edges $(v_{\mathrm{i}}, v_{(\mathrm{i}+1)})$ where $\mathrm{i} = 1 \ldots 7$, are changed to $2\epsilon/\Delta$. Other weights remain the same. The maximum degree in this graph is $n-1$. Assuming $b = 1$ for every vertex, the optimal cover weight is $\Delta/2 * (2\epsilon/\Delta) + x = \epsilon + x$, whereas if the Dual Feasible algorithm picks the first vertex to be $v_0$, the weight of the edge cover could be $\Delta x$. Taking the ratio we have $\frac{\Delta x}{\epsilon + x}$. As $\epsilon \to 0$, the ratio approaches $\Delta$.

**Lemma 3.4.3.** *The time complexity of the* Dual Feasible *algorithm is* $O(\beta\, m)$.

*Proof.* A vertex $v$ can be selected at most $b(v)$ times. When it is selected it has to find the edge with minimum residual weight, which can be found in $O(deg(v))$ time. Summing over all vertices we get $\sum_{v \in V} b(v) \cdot deg(v) = O(\beta\ m)$. ■

# 4. REDUCTION TO MATCHING BASED ALGORITHMS FOR $b$-EDGE COVER

Matching is one of the fundamental and well-studied combinatorial problems. There are interesting relationships between edge cover and matching. One can exploit theses relationships to design new algorithms. In this chapter, we show two such algorithms for $b$-EDGE COVER. One of them uses the complimentary relationship between $b$-MATCHING and $b$-EDGE COVER. We call it MCE algorithm [64]. The other reduces the $b$-EDGE COVER to a constrained perfect $b$-matching.

## 4.1 $b$-EDGE COVER via compliment to $b$-MATCHING

In this section, we discuss the details of the *Matching Complement Edge cover* (MCE) algorithm. We will describe 2-approximate parallel algorithm using $b$-SUITOR.

An optimal algorithm for the minimum weight $b$-EDGE COVER problem can be obtained by computing a maximum weight $b'$-MATCHING, by the following three step procedure [33]:

1. For each vertex $v$, compute $b'(v) = \deg(v) - b(v)$, where $\deg(v)$ is the degree of $v$.
2. Compute $M_*$, a maximum weight $b'$-MATCHING.
3. A min weight $b$-EDGE COVER is the complement of the matching: $C_* = E \setminus M_*$.

In this construction, steps 1 and 3 ensure that the computed $b$-EDGE COVER is a valid cover, and the optimality of the cover depends on step 2. If we compute an approximate $b'$-MATCHING, keeping steps 1 and 3 fixed, then the solution to the $b$-EDGE COVER *may not necessarily* be an approximate solution for $b$-EDGE COVER. However, we have showed in [64] that if the $b'$-MATCHING is computed using the $b$-SUITOR algorithm then the corresponding $b$-EDGE COVER will satisfy 2-approximation bounds. We show a parallel MCE algorithm using 1/2-approximate parallel $b$-SUITOR in Algorithm 9.

Since $b$-SUITOR is an essential part of the MCE algorithm, we briefly describe a variant of it in Algorithm 10. For more details, we refer the reader to the papers [69]. The $b$-SUITOR algorithm is derived from the SUITOR algorithm for maximum weighted matching [70]. The

**Algorithm 9** Parallel MCE$(G, b)$

1: $C = \emptyset$
2: **for** $v \in V$ **in parallel do**
3: $\quad b'(v) = max\{0, \deg(v) - b(v)\}$
4: **end for**
5: $M$=Parallel_$b$-Suitor$(G, b')$
6: **for** $v \in V$ **in parallel do**
7: $\quad C = C \cup \{N(v) \setminus M(v)\}$
8: **end for**
9: **return** $b$-Edge Cover $C$

algorithm is based on vertices making proposals to each other, just as in the Stable Matching problem. Vertices can propose in any order, but each vertex must propose to its current heaviest *eligible* neighbor. A vertex $v$ is an *eligible neighbor* of a vertex $u$ if $v$ does not already have a proposal of higher weight from another neighbor of $v$. A vertex $u$ can also annul the proposal made by a vertex $w$ to a mutual neighbor $v$, if the weight of the edge $(u, v)$ is higher than the weight of $(v, w)$. In this case, $u$ proposes to $v$, and annuls the proposal $(v, w)$; now $w$ must propose to its next heaviest eligible neighbor. An edge is matched when two vertices propose to each other. Since we can annul proposals, any vertex can make proposals thus increasing the parallelism.

The parallel $b$-Suitor algorithm is shown in Algorithm 10. The algorithm maintains a queue $Q$ of vertices whose $b(v)$ values are not satisfied yet, for which it tries to find partners during the current iteration of the **while** loop; and also a queue of vertices $Q'$ whose proposals are annulled in this iteration, and will be processed again in the next iteration. (This is what "delayed" means; annulled vertices are not processed in the same iteration. "Partial" means that the adjacency lists are partially sorted to find a subset of heaviest neighbors.) The algorithm then seeks a partner for each vertex $u$ in $Q$ in parallel. It tries to find $b(u)$ proposals for $u$ to make while the adjacency list $N(u)$ has not been exhaustively searched thus far in the course of the algorithm.

Consider the situation when a vertex $u$ has $\mathrm{i} - 1 < b(u)$ outstanding proposals. The vertex $u$ can propose to a vertex $p$ in $N(u)$ if it is a heaviest eligible neighbor in the set $N(u)$

**Algorithm 10** Parallel_$b$-SUITOR$(G, b)$

```
1: Q = V; Q' = ∅;
2: S(v) = ∅, min-priority heap ∀v
3: while Q ≠ ∅ do
4:     for vertices u ∈ Q in parallel do
5:         i = 1;
6:         while i <= b(u) and N(u) ≠ exhausted do
7:             Let p ∈ N(u) be an eligible partner of u;
8:             if p ≠ NULL then
9:                 Lock S(p);
10:                if p is still eligible then
11:                    i = i + 1;
12:                    Add u to S(p);
13:                    if u annuls the proposal of v then
14:                        Add v to Q'; Update db(v);
15:                        Remove v from S(p);
16:                    end if
17:                end if
18:                Unlock S(p);
19:            else
20:                N(u) = exhausted;
21:            end if
22:        end while
23:    end for
24:    Update Q using Q'; Update b using db;
25: end while
26: return S
```

and if the weight of the edge $(u, p)$ is greater than the lowest offer that $p$ has. In this case, $p$ would accept the proposal of $u$ rather than its current lowest offer.

If the algorithm finds a partner $p$ for $u$, then the thread processing the vertex $u$ attempts to acquire the lock for the priority queue $S(p)$ so that other vertices do not concurrently become Suitors of $p$. This attempt might take some time to succeed since another thread might have the lock for $S(p)$. Once the thread processing $u$ succeeds in acquiring the lock, then it needs to check again if $p$ continues to be an eligible partner, since by this time another thread might have found another Suitor for $p$, and its lowest offer might have changed. If $p$ is still an eligible partner for $u$, then we increment the count of the number of proposals made by $u$, and make $u$ a Suitor of $p$. If in this process, we dislodge the last Suitor $x$ of $p$,

then we add $x$ to the queue of vertices $Q$ to be processed in the next iteration. Finally the thread unlocks the queue $S(p)$.

We fail to find an eligible partner $p$ for a vertex $u$ when we have exhaustively searched all neighbors of $u$ in $N(u)$, and none offers a weight greater than the lowest offer $u$ has. In this case $u$ will have fewer than $b(u)$ matched neighbors. After we have considered every vertex $u \in Q$ to be processed, we can update data structures for the next iteration. We update $Q$ to be the set of vertices in $Q$; and the vector $b$ to reflect the number of additional partners we need to find for each vertex $u$ using $db(u)$, the number of times $u$'s proposal was annulled.

### 4.1.1 Approximation Bounds

In this section, we show that MCE is a 2-approximation algorithm for $b$-Edge Cover. We will need a Lemma from [63]. The Greedy algorithm for $b$-Matching matches edges in increasing order of (static) edge weights.

**Lemma 4.1.1.** *When the* Greedy *algorithm for* $b$-Matching *matches an edge, it is a locally dominant edge in the residual graph (the graph induced by the currently unmatched edges).*

**Theorem 4.1.2.** MCE *is a* 2*-approximation algorithm for* $b$-Edge Cover.

*Proof.* Let the optimal minimum weight $b$-Edge Cover be denoted by $C_*$, the complement of an optimal maximum weight $b'$-Matching, $M_*$. Also, let the $b$-Edge Cover computed by MCE be denoted by $C$, which takes the complement of the 1/2-approximate matching $M$, obtained by $b$-Suitor.

Consider an edge $\mathrm{e}(u, v) \in C_* \setminus C$, which belongs to the optimal edge cover but not the approximate edge cover. This implies that $\mathrm{e}(u, v) \in M \setminus M_*$ since the covers are obtained by complementing the matched edges. The worst case scenario for $b'$-Matching is when $b$-Suitor matches the edge $\mathrm{e}(u, v)$, and thus cannot match two other edges that belong to $M_*$, say $\mathrm{e}(x, u) \in M_*$ and $\mathrm{e}(v, y) \in M_*$. Hence $\mathrm{e}(x, u) \notin M$ and $\mathrm{e}(v, y) \notin M$. Since the

$b$-Suitor algorithm computes the same matching as the Greedy algorithm, $\mathrm{e}(u, v)$ must be a locally dominating edge when it is matched, by Lemma 4.1.1. Thus

$$\begin{aligned} w(u,v) \geq w(x,u); \quad w(u,v) \geq w(v,y); \quad \text{hence} \\ 2w(u,v) \geq w(x,u) + w(v,y). \end{aligned} \tag{4.1}$$

Since $\mathrm{e}(x, u) \notin M$ and $\mathrm{e}(v, y) \notin M$, both of these edges belong to the approximate cover $C$. Therefore, the weight of $C$ can be bounded as follows.

$$\begin{aligned} w(C) &= w(C_*) - w(u,v) + w(x,u) + w(v,y) \\ &\leq w(C_*) - w(u,v) + 2w(u,v) \quad \text{(from Eqn 4.1)} \\ &= w(C_*) + w(u,v). \end{aligned} \tag{4.2}$$

By summing over all edges in the optimal cover that are not included in the approximate cover, $C_* \setminus C$, we obtain

$$\begin{aligned} w(C) &\leq w(C_*) + \sum_{(u,v) \in C_*} w(u,v) \\ &= \; w(C_*) + w(C_*) \quad = 2\; w(C_*). \end{aligned} \tag{4.3}$$

Thus MCE is a 2-approximation algorithm for $b$-Edge Cover. ■

**Lemma 4.1.3.** *A $b$-Edge Cover computed by the* MCE *algorithm does not have redundant edges.*

*Proof.* An approximate maximum weight $b'$-Matching $M$ of a graph computed by the $b$-Suitor algorithm cannot have two neighboring vertices $u$ and $v$, with $u$ having fewer than $b'(u)$ and $v$ having fewer than $b'(v)$ incident edges belonging to $M$. For, then we can add the edge $\mathrm{e}(u, v)$ to the $b'$-Matching without violating the matching constraints and increase the weight of the approximate matching. But this contradicts the fact that the $b$-Suitor algorithm computes a maximal matching. By considering the complement, a $b$-Edge Cover obtained by the MCE algorithm cannot have two super-saturated neighboring vertices in $C$. Hence a cover computed by the MCE algorithm does not have redundant edges. ■

### 4.1.2 Parallel Depth and Work of SUITOR and $b$-SUITOR

In this section we show that the SUITOR [70] and the $b$-SUITOR algorithms have provably low parallel depth and work. The depth is the number of time steps needed by the parallel algorithm, and the work is the total number of operations performed by the algorithm. These are the first results on the depth of the SUITOR and $b$-SUITOR algorithms that we know of. The low depth and work of $b$-SUITOR immediately establish the low depth of the parallel MCE algorithm, since the other two steps of the algorithm are embarrassingly parallel (i.e., of constant depth) with linear work.

**Theorem 4.1.4.** *The expected parallel depth of the* SUITOR *algorithm that computes a* $1/2$*-approximate* 1*-matching in a graph is* $O(\log(\Delta)\log m)$*, when the weights of the edges are chosen uniformly at random.*

*Proof.* We begin by analyzing an algorithm related to the SUITOR algorithm, the Locally Dominant Edge (LDE) algorithm. This algorithm adds an edge to the approximate matching when there are no neighboring edges of higher weight (it becomes locally dominant), and then deletes all of the neighboring edges. An algorithm of Blelloch, Fineman and Shun [71] for computing an unweighted maximal matching in parallel uses random priorities on the edges to compute the matching. Hence it is equivalent to the LDE algorithm for weighted matching with random edge weights, and an analysis of the maximal matching algorithm shows that the LDE algorithm has the stated parallel depth.

Now we turn to the SUITOR algorithm and consider its relationship to the LDE algorithm. Specifically we consider the "delayed" version of the algorithm in which a vertex with a proposal annulled is queued for further processing in the next iteration. In the LDE algorithm, an edge is matched when it becomes locally dominant, detected by its two endpoints pointing to each other. In the SUITOR algorithm, each vertex $u$ keeps track of the highest weight of the proposal it has received so far. A neighbor of $u$ could use this information, if it is already available, to propose to its next heaviest eligible neighbor without first proposing to $u$. Hence if we view the computations of these algorithms in rounds, in the SUITOR algorithm, a vertex gets matched in the same or an earlier round relative to the LDE algorithm. Hence the SUITOR algorithm also has $O(\log(\Delta)\log m)$ depth. ■

**Theorem 4.1.5.** *The expected work in the* Suitor *algorithm is* $O(m)$ *when the edge weights are chosen uniformly at random.*

*Proof.* The adjacency lists can be sorted in expected linear time using bucket sort when the weights are chosen randomly [62]. The Suitor algorithm needs to go through the sorted adjacency list of each vertex at most once. ■

Obtaining linear work for the maximal matching algorithm of Blelloch et al. [71] is more complicated, and is accomplished by working on a prefix of the graph whose size is carefully chosen, which increases the depth to $O(\log^4 m/\log\log m)$.

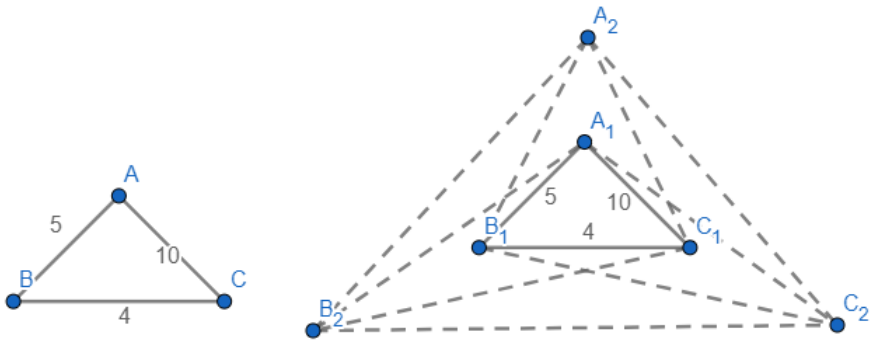



**Figure 4.1.** Reduction from a $b$-Matching to a Matching. (Left) Original graph, (Right) Reduced graph for $b = 2$.

We now show that these results can be extended to the $b$-Suitor algorithm by reducing the $b$-Matching problem to the 1-matching problem in a modified graph. This reduction is due to Tutte [25]. We replace each vertex $u$ with $b(u)$ vertices in the modified graph; each edge $(u, v)$ is replaced by a complete bipartite graph of $b(u)\, b(v)$ edges, with weights equal to the original weight of the edge $(u, v)$. We restrict only one of the edges in the bipartite subgraph to be matched, but other vertices in this subgraph could be matched to edges in other subgraphs. We show an example of the reduction in Figure 4.1. The value of $b$ is 2. We see each edge is replaced by a complete bipartite graph with the same weight. In the example graph, if we choose $(A_1, B_1)$ as a matched edge then we can not match the edge

$(A_2, B_2)$. With this restriction, a 1/2-approximate matching in the transformed graph would correspond to a 1/2- approximate $b$-MATCHING in the original graph.

The reduced graph has $O(m)$ vertices, $O(\beta^2 m)$ edges, and the maximum degree $\beta\Delta \leq \Delta^2$. Thus the parallel depth of $b$-SUITOR algorithm when the edge weights are uniformly random becomes $O(\log(\Delta^2)\ \log(\beta^2 m)) = O(\log \Delta\ \log m)$. Similarly the work becomes $O(\beta^2\ m)$. (Recall that $\beta = \max_v b(v)$, and $b(V) = \sum_v b(v)$.) We can derive similar depth and work results for the MCE algorithm too.

### 4.2 $b$-EDGE COVER via reduction to a constrained perfect $b$-MATCHING

One of the ways to construct a minimum weighted 1-edge cover is to reduce it to a perfect matching instance [33, Chapter 19]. Given a graph $G = (V, E, w)$, the reduction creates a second disjoint copy $\bar{G} = (\bar{V}, \bar{E}, \bar{w})$ of the original graph, and connects each vertex $v \in V$ with its copy $\bar{v} \in \bar{V}$. Let $G'(V', E', w')$ denote the new graph, and let $\mu(v)$ denote an edge of lowest weight incident on $v$ in $G$. The weight function $w'$ on $E'$ is defined as follows: $w'(\mathrm{e}) = w(\mathrm{e})$ for each $\mathrm{e} \in E$, $w'(\mathrm{e}') = \bar{w}(\bar{\mathrm{e}}) = w(\mathrm{e})$ for each $\bar{\mathrm{e}} \in \bar{E}$, and $w'(v, \bar{v}) = 2w(\mu(v))$. It is easy to verify that $G'$ has a perfect matching. We can compute a minimum weight perfect matching $M_*$ in $G'$; then we obtain a minimum weight edge cover of $G$ by selecting the edges $C_* = (M_* \cap E) \cup \{\mu(v) : (v, \bar{v}) \in M_*, \forall v \in V\}$.

We now modify this graph construction to compute a minimum weight $b$-edge cover. As earlier, we create a second disjoint copy $\bar{G}$ of the graph $G$. But now $\mu(v)$ is a *list* of $b(v)$ lowest weight edges incident on $v$ in $G$. $\mu(\bar{v})$ is also defined similarly for $\bar{G}$. Let the new graph be $G'(V', E', w')$, where $V' = V \cup \bar{V}$. For each vertex $v \in V$ in $G'$ we add $b(v)$ *connecting edges* from $v$ to its copy $\bar{v}$. These edges have one to one correspondence with the edges in the list $\mu(v)$. We denote the connecting edges by the indices in the $\mu(.)$ list. Thus $(v, \bar{v}, \mathrm{i})$ represents the i-th connecting edge joining $v$ and $\bar{v}$, which corresponds to the i-th edge in $\mu(v)$, denoted by $\mu(v, \mathrm{i})$. The weight function is defined as follows: $w'(\mathrm{e}) = w(\mathrm{e})$, $\forall \mathrm{e} \in E$, and $w'(\mathrm{e}) = \bar{w}(\bar{\mathrm{e}}) = w(\mathrm{e})$, $\forall \bar{\mathrm{e}} \in \bar{E}$. The weight of the connecting edges $w'((v, \bar{v}, \mathrm{i})) = 2w(\mu(v, \mathrm{i}))$. We solve the minimum weight perfect $b$-matching problem on $G'$ with an additional set of constraints. The additional constraints are necessary to avoid the

selection of a connecting edge and its corresponding original edge in the same matching. We can formulate the constrained perfect $b$-matching as follows.

$$\begin{aligned} \min \sum_{\mathrm{e}\in E'} w'(\mathrm{e})x(\mathrm{e}) \\ \text{subject to } \sum_{\mathrm{e}\in\delta(v)} x(\mathrm{e}) = b(v), \forall v \in V', \\ x_{\mu(v,\mathrm{i})} + x_{(v,\bar{v},\mathrm{i})} \leq 1, \forall v \in V, \mathrm{i} \in [b(v)], \\ x_{\mu(\bar{v},\mathrm{i})} + x_{(v,\bar{v},\mathrm{i})} \leq 1, \forall \bar{v} \in \bar{V}, \mathrm{i} \in [b(\bar{v})], \\ x(\mathrm{e}) \in \{0,1\}, \forall \mathrm{e} \in E'. \end{aligned} \tag{4.4}$$

Here $x(\mathrm{e})$ is a binary variable defined on edges of $G'$, and $\delta(v)$ is the set of edges incident on the vertices of $G'$. In Problem 4.4, the second and third set of constraints ensure that a perfect matching does not contain both an original edge and its corresponding connecting edge.

Let $M'_*$ denote the optimal solution of Problem 4.4. We can recover a $b$-edge cover from $M'_*$ as follows: $C_* = (M'_* \cap E) \cup \{\mu(v,\mathrm{i}) : v \in V, \mathrm{i} \in [b(v)] \text{ and } (v,\bar{v},\mathrm{i}) \in M'_*\}$. We will show that $C_*$ is a minimum weighted $b$-edge cover.

**Lemma 4.2.1.** *$C_*$ is an optimal $b$-edge cover*

*Proof.* Let $M$ and $\bar{M}$ be the disjoint subset of the optimal matching in $G$ and $\bar{G}$ respectively, i.e., $M = M'_* \cap E$ and $\bar{M} = M'_* \cap \bar{E}$. Also let $S$ be the set of connecting edges in $M'_*$. So, $M'_* = M \cup \bar{M} \cup S$. Since $G$ and $\bar{G}$ are copy of each others $M$ is a valid matching in $\bar{G}$ and $\bar{M}$ is a valid matching in $G$. We observe that $w'(M) = w'(\bar{M})$, otherwise we could replace the larger subset with the smaller one and achieve matching whose weight is less than $w'(M')$ contradicting the optimality of $M'_*$. Since the connecting edges are assigned weights twice their corresponding edge weights, we see that:

$$w(C_*) = w'(M) + \frac{1}{2}w'(S) = \frac{1}{2}\Big(w'(M) + w'(\bar{M}) + w'(S)\Big) = \frac{1}{2}w'(M'_*). \tag{4.5}$$

Now we will discuss a reverse construction that will take any $b$-edge cover of $G$ as input and construct some perfect $b$-matching of $G'$. We start with a feasible $b$-edge cover $C$

and an empty matching $M'$. For each edge e $\in C$, if e can be added without violating the matching constraints in $G'$, we add e and $\bar{\text{e}}$ into $M'$. If not one of the endpoints of e must be under-saturated in $G'$. If e has an available corresponding connecting edge then we insert the connecting edge in the matching, otherwise we choose a feasible (w.r.t the constraints of Problem 4.4) connecting edge with maximum weight. Note that such a connecting edge must be available since we have $b(.)$ parallel edges to choose from. The weight of the selected connecting edge is at most twice the weight of e. Hence, $w'(M') \leq 2w(C)$. Using Equation 4.5,

$$w(C_*) = \frac{1}{2}w'(M'_*) \leq \frac{1}{2}w'(M') \leq w(C).$$

The second inequality is due to the optimality of $M'_*$. ■

### 4.2.1 Approximate $b$-Edge Cover using constrained perfect matching

We now turn to approximation algorithms. We will show that a greedy (instead of an optimal) constrained perfect $b$-matching of $G'$ provides a 3/2-approximate $b$-edge cover of G. But before that, let us discuss an improvement of the graph construction technique which might be helpful in achieving practical efficiency.

We can avoid copying the graphs by creating a new graph where there are $b(v)$ loop edges for each vertex, $v \in V$. These loop edges are equivalent to the connecting edges of the earlier construction thus correspond to $b(v)$ lowest weighted edges incident on $v$. These are denoted by $(v, v, \text{i}), \forall v \in V, \text{i} \in [b(v)]$. Also $\mu(v, \text{i})$ now represents the original edge corresponding to the i-th loop edge incident on $v$. Formally the new graph is $G'(V', E', w')$, where $V' = V$, and $E' = E \cup \{(v, v, \text{i}) : v \in V, \text{i} \in [b(v)]\}$. As before the weights $w'(\text{e}) = w(\text{e})$, $\forall \text{e} \in E$, and $w'((v, v, \text{i})) = 2w(\mu(v, \text{i}))$. This construction is equivalent to the earlier one but creates fewer vertices and edges as it avoids making the second disjoint copy of the graph. Recall that $\beta = \max_{v \in V} b(v)$. The numbers of vertices and edges in $G'$ using the new transformation are $n$ and $m + \sum_{v \in V} b(v) \leq m + \beta\, n$, respectively; the earlier transformation creates $2n$ vertices and $2m + \beta\, n$ edges.

The approximation algorithm using greedy matching is shown in Algorithm 11. We maintain a saturation counter array $cV$ of size $n$ initialized to $b(v)$ for each vertex $v$ to count how many edges incident on $v$ need to be added to an edge cover to saturate the vertex. An edge from the original graph is feasible if the counter on both of its endpoints is greater than 0. A loop edge $(v, v, \mathrm{i})$ is feasible if the counter on $v$ is greater than 0; it will be added to an edge cover only if the original edge that it corresponds to has not already been included in the current edge cover.

Note that the analysis of the algorithm works with effective weights of the edges, defined as the weight of the edge divided by the number of its unsaturated endpoints. The algorithm starts with an empty cover, and hence the number of unsaturated endpoints for all edges is 2. The effective weight of each original edge is half of its actual weight.

**Algorithm 11** Constrained Greedy Prf. Match$(G', cV)$

```
1: sort edges in E' in ascending order of half of the weights.
2: C = ∅
3: for e ∈ E' do
4:     if e is a feasible edge then
5:         if e is a loop edge (v, v, i) and μ(v, i) ∉ C then
6:             C = C ∪ μ(v, i)                          ▷ Let μ(v, i) := (v, x)
7:             cV(v) = cV(v) − 1
8:             cV(x) = cV(x) − 1
9:         else
10:            C = C ∪ e                                ▷ Let e := (u, v)
11:            cV(u) = cV(u) − 1
12:            cV(v) = cV(v) − 1
13:        end if
14:    end if
15: end for
```

**Lemma 4.2.2.** *The edge cover from algorithm 11 is* 3/2*-approximate.*

*Proof.* We will show by induction that the edge has the lowest effective weight when the algorithm decides to insert an edge into $C$, the edge has the lowest effective weight. The Greedy algorithm that constructs an edge cover by adding such edges is known to be 3/2-approximate [65], and hence the result follows. Note that the Greedy algorithm (Chapter 2.1)

requires one to update the effective weight of the edges during the algorithm. In contrast, our graph construction allows us to work with the initial weights.

We say a *phase* of the algorithm begins with the search for a feasible edge to add to the cover, and ends when the algorithm succeeds in finding such an edge. The first phase ends after the first iteration of the **for** loop, since initially all the edges are feasible and the algorithm chooses one with minimum weight, which is also the minimum effective weight. This edge cannot be a loop edge since we have multiplied the weight of such edges by a factor of two.

We assume inductively that the algorithm chooses edges with the smallest effective weight for the first i phases. Let us consider the edge e chosen in the i + 1-th phase.

If e is not a loop edge then both of its endpoints are not saturated yet, and its effective weight is $w'(\mathrm{e})/2 = w(\mathrm{e})/2$. For contradiction, assume e is not the edge with a minimum effective weight, and $\mathrm{e}'$ be such an edge. There are two cases, $\mathrm{e}'$ is a regular edge or a loop edge. If $\mathrm{e}'$ is a regular edge, its effective weight is $w'(\mathrm{e}')/2 = w(\mathrm{e}')/2$. But since we sorted according to the half of weights, this edge should have been considered earlier phases which contradicts the induction hypothesis. Similarly if $\mathrm{e}'$ is a loop edge, the effective weight of the corresponding edge is $w'(\mathrm{e}')/2$. We can repeat the same argument as above and show that this is a minimum effective weighted edge.

If $\mathrm{e}(v, v, k)$ is a loop edge, then obviously one endpoint of the corresponding edge $\mu(v, k)$ is covered (i.e., this vertex is saturated or over-saturated) hence the effective weight of $\mu(v, k)$ is $w'(v, v, k)/2 = w(\mu(v, k))$. which is exactly the effective weight of $\mu(v, k)$. The same argument as for the first case shows that $\mu(v, k)$ has the lowest effective weight amongst all the feasible edges. This concludes the proof. ■

The difference between the greedy algorithm and the constrained greedy algorithm is that in the latter, by modifying the graph, we establish a linear relationship between the effective weight and the weights of the reduced graph. That allows us to get rid of the dynamic weight updates of the edges.

**Lemma 4.2.3.** *The runtime of Algorithm 11 is* $O(m \log n)$.

*Proof.* It is easy to verify that given the graph $G'$, the algorithm 11 runs in $O(m \log n)$ time. We will now describe how we can construct $G'$ in $O(m)$ time. Note that the most computationally expensive step to compute $G'$ is to find the $b(v)$-th lowest weight edge for each vertex. Given an adjacency list representation of a vertex $v$, we can use a worst-case linear time selection algorithm [62, Ch. 9] to find the $b(v)$-th lowest weight edge, say $s_b$. To get the $b(v)$ cheapest edge, we scan through the adjacency list to find the edges that are strictly smaller than $w(s_b)$. If the number of such edges is less than $b(v) - 1$, then we find necessary edges with weight $w(s_b)$. This takes linear time w.r.t to the size of the adjacency list of a vertex. Summing over all the vertices, this step takes $O(m)$ time. ■

## 4.3 Computational Results of $b$-Edge Cover algorithms

### 4.3.1 Experimental Setup

All the experiments were conducted on a Purdue Community cluster computer called *Rice*, 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, 25 MB L3 cache, and 64 GB memory per node. For compilation we have used the `g++` compiler.

**Table 4.1.** The structural properties of our graphs listed in increasing order of edges.

| Problems | Vertices | Edges | Mean Degree |
|---|---|---|---|
| Fault_639 | 616,923 | 5,715,102 | 19 |
| bone010 | 986,703 | 7,861,302 | 16 |
| Serena | 1,382,121 | 13,716,976 | 20 |
| mouse_gene | 43,126 | 14,461,095 | 671 |
| dielFilterV3real | 1,102,824 | 21,583,469 | 39 |
| Flan_1565 | 1,564,794 | 22,636,872 | 29 |
| kron_g500-logn21 | 1,544,087 | 91,040,932 | 118 |
| hollywood-2011 | 1,985,306 | 114,492,816 | 115 |
| G500_21 | 1,598,722 | 118,594,475 | 148 |
| SSA21 | 2,089,808 | 123,097,397 | 118 |
| eu-2015 | 10,972,981 | 257,659,403 | 47 |

Our testset consists of both real-world and synthetic graphs shown in Table 4.1. We generated two classes of RMAT graphs: (a) G500, representing graphs with skewed degree

distributions from the Graph 500 benchmark [72] and (b) SSCA, from the 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 problems taken from the the SuiteSparse Matrix Collection [73] covering application areas such as medical science, structural engineering, and sensor data. We also have a large web-crawl graph(*eu-2015*) [74] and a movie-interaction network(*hollywood-2011*) [75].

All reported results are the average of five runs on a graph with random edge weights. For Edge Cover the uniform random weights are in the range [1 100]. Since LEDA works only with integer weights, the real-valued weights are then rounded to their nearest integers. For the $b$-Edge Cover experiment, the uniform random weights are chosen from the range [1 1000].

### 4.3.2 Edge Cover Results

We compare four algorithms: the exact algorithm that computes the minimum weight of an edge cover using the weight transformation described in the previous chapters of this thesis; the nearest neighbor algorithm (NN); the 3/2-approximation primal dual algorithm (PD) and a 1/2-approximate maximum weight matching algorithm that with the approximation-preserving weight transformation obtains a 3/2-approximation minimum weight edge cover (Match). We use LEDA's maximum weight matching code to compute the maximum matching with transformed weights, and the Suitor algorithm to compute the approximate matching. We removed redundant edges from the edge covers computed by these algorithms.

In Table 4.2 we report the minimum weight computed by the exact algorithm, and the distance to optimality of the other algorithms, computed as $(\text{approx} - \text{opt})/\text{opt} * 100$, where opt and approx are weights from the optimal and approximation algorithms, respectively. Note that all three algorithms perform much better than their worst-case ratios (3/2 for PD and Match; 2 for the NN algorithm). The Match algorithm is the best performer, followed by PD and then NN.

**Table 4.2.** Comparison of weights of edge covers computed by approximation algorithms w.r.t. the exact algorithm.

| Problems | OPT weight | distance from optimality (%) | | |
|---|---|---|---|---|
| | | PD | NN | Match |
| Fault_639 | 2,475,175 | 3.21 | 5.75 | 1.07 |
| bone010 | 4,500,059 | 3.20 | 5.73 | 1.04 |
| serena | 5,477,688 | 3.21 | 5.62 | 1.01 |
| mouse_gene | 188,517 | 1.94 | 3.86 | 1.24 |
| dielFilterV3real | 3,007,336 | 2.80 | 4.77 | 0.82 |
| Flan_1565 | 4,362,155 | 3.17 | 5.62 | 1.15 |
| kron_g500 | 26,301,787 | 0.12 | 0.18 | 0.01 |
| hollywood_11 | 9,310,300 | 1.87 | 3.69 | 0.45 |
| G500_21 | 25,624,663 | 0.11 | 0.16 | 0.01 |
| ssa21 | 8,243,419 | 2.85 | 4.05 | 0.66 |
| eu-2015 | 218,514,387 | 0.49 | 0.89 | 0.03 |
| Geo. Mean | | 1.32 | 2.25 | 0.28 |

In Table 4.3, we compare the run times of the algorithms. We report the time taken by the exact algorithm, and report the relative performance of the approximation algorithms as the ratio of the run time of the exact to that of the approximation algorithm. Larger the relative performance, the faster the algorithm. Notice that the NN algorithm is the fastest, followed by the Match algorithm, and then the PD algorithm. But the run times of the approximation algorithms are within a factor of two of each other.

### 4.3.3 *b*-Edge Cover Results

The implementation of an optimal *b*-Edge Cover algorithm is left as a future work of this thesis. In this section we compare the approximation algorithms instead. We compare the Primal-dual algorithm (PD), the Lazy Greedy algorithm (LG), and the b-nearest neighbor algorithm (bNN). In a future work [76], we plan to report the computational results of the greedy constrained greedy perfect matching algorithm.

Both PD and the LG algorithms are 3/2-approximate and compute the same *b*-Edge Cover, while bNN is 2-approximate. In Table 4.4, we report the weights of the *b*-edge covers. We include the weight computed by the original algorithm, and then the weight obtained

**Table 4.3.** Relative performance of runtimes of approximation algorithms w.r.t. the exact algorithm for edge cover.

| Problems | Time(s) | Rel. Perf. | | |
|---|---|---|---|---|
| | Exact Alg. | PD | NN | Match |
| Fault_639 | 8.78 | 36.61 | 73.32 | 44.23 |
| bone010 | 13.37 | 37.70 | 77.23 | 43.97 |
| serena | 21.48 | 36.58 | 74.27 | 43.28 |
| mouse_gene | 14.39 | 35.74 | 69.13 | 40.79 |
| dielFilterV3real | 27.30 | 34.78 | 69.05 | 40.74 |
| Flan_1565 | 27.20 | 32.91 | 63.50 | 39.85 |
| kron_g500 | 147.89 | 37.92 | 78.83 | 43.96 |
| hollywood_11 | 122.78 | 33.88 | 65.43 | 39.84 |
| G500_21 | 182.00 | 36.95 | 77.53 | 41.99 |
| ssa21 | 233.71 | 40.62 | 87.41 | 46.00 |
| eu-2015 | 408.40 | 41.22 | 85.88 | 48.84 |
| Geo. Mean | | 36.73 | 74.33 | 42.97 |

by removing redundant edges. The last two columns show the percentage difference in weights between the bNN and PD algorithms. The results show that the PD algorithm computes smaller weights for the edge cover. The difference in weights between the two original algorithms can be large, up to 13% for these problems, with a geometric mean of about 5%. However, after removing redundant edges, this difference narrows to at most 3%. This implies that more redundant edges are removed from the bNN algorithm.

The run times of the algorithms are plotted in Fig. 4.2 as the ratio of the run time of the LG algorithm to the run time of the second algorithm. Values higher than one for an algorithm means that the algorithm is faster than LG. Note that in geometric mean, the PD algorithm is about 3 times faster than the LG, and the bNN algorithm is about 8 times faster.

The $\Delta$-approximation algorithm was implemented by [63] and compared with the LSE algorithm, which has approximation ratio of 3/2. The performance of the latter algorithm was highly sensitive to the order in which the vertices are processed, both for the weights and the runtimes. Generally the LSE algorithm computed lower weights and it was also faster.

**Table 4.4.** Weight of $b$-edge covers computed by the PRIMAL DUAL algorithms. The difference is the percentage of increase in weight using $b$-NEAREST NEIGHBOR w.r.t PRIMAL DUAL.

| Problem | Primal Dual | | difference(%) | |
|---|---|---|---|---|
| | original | remove redundant | orig. | rem. red. |
| bone010 | 3.93E+09 | 3.93E+09 | 0.00 | 0.00 |
| Fault_639 | 2.86E+09 | 2.86E+09 | 0.00 | 0.00 |
| Serena | 6.85E+09 | 6.84E+09 | 0.07 | 0.00 |
| Flan_1565 | 1.06E+10 | 1.04E+10 | 2.14 | 0.12 |
| G500_21 | 6.78E+09 | 6.62E+09 | 2.20 | 0.20 |
| kron_g500 | 6.00E+09 | 5.86E+09 | 3.02 | 0.36 |
| eu-2015 | 3.28E+10 | 3.22E+10 | 3.63 | 0.28 |
| mouse_gene | 1.11E+08 | 1.06E+08 | 5.29 | 0.45 |
| hollywood_11 | 9.80E+09 | 9.53E+09 | 7.21 | 1.65 |
| dielFilterV3real | 7.47E+09 | 7.30E+09 | 10.50 | 3.03 |
| ssa21 | 9.26E+09 | 8.59E+09 | 12.88 | 0.90 |
| Geo. Mean | | | 4.77 | 0.51 |

Several of the approximation algorithms for the $b$-EDGE COVER problem have been implemented on parallel computers. The LSE algorithm and the Matching Complement Edge cover (MCE) algorithm have been implemented on shared memory parallel machines: an IBM Power-8 with 764 cores and an Intel Xeon with 36 cores [64]. The Primal Dual 3/2-approximation algorithm has been implemented by us on multiple cores of an Intel Xeon (Ferdous and Pothen, unpublished). The MCE algorithm has also been implemented on $8,192$ cores of a distributed memory parallel computer with good speedups [77], and it has been used to solve the adaptive anonymity problem, which we discuss in the next subsection.

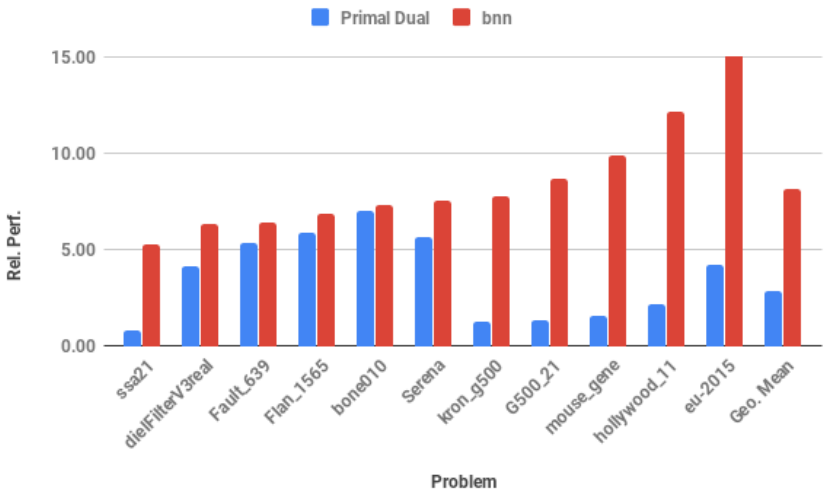



**Figure 4.2.** Relative Performance of runtimes of the Primal-Dual and bNN algorithms for $b$-Edge Cover w.r.t. the Lazy Greedy algorithm.

## 5.1 Greedy and Lazy Greedy Algorithms

A popular algorithm for maximizing submodular $b$-MATCHING is the GREEDY algorithm [42], where in each iteration, an edge with the maximum marginal gain is added to the matching. In its simplest form the GREEDY algorithm could be expensive to implement, but submodularity can be exploited to make it efficient. The efficient implementation is known as the LAZY GREEDY algorithm [56], [78]. As the maximum gain of each edge decreases monotonically in the course of the algorithm, we can employ a maximum heap to store the gains of the edges. Since the submodular function is normalized, the initial gain of each edge is just the function applied on the edge, and at each iteration we pop an edge e from the heap. If e is an *available edge*, i.e., e can be added to the current matching without violating $b$-MATCHING constraints, we update its marginal gain $g(\text{e})$. We compare $g(\text{e})$ with the next best marginal gain of an edge, available as the heap's current top. If $g(\text{e})$ is greater than or equal to the marginal gain of the current top, we add e to the matching; otherwise we push e to the heap. We iterate on the edges until the heap becomes empty. Algorithm 12 describes the LAZY GREEDY approach.

**Algorithm 12** LAZY GREEDY Algorithm ($G(V, E, W)$)

```
pq = max heap of the edges keyed by marginal gain
while pq is not empty do
    Edge e = pq.pop()
    Update marginal gain of e
    if e is available then
        if marg_gain of e ≥ marg_gain of pq.top() then
            Add e to the matching
            update b(.) values of endpoints of e
        else
            push e and its updated gain into pq
        end if
    end if
end while
```

The maximum cardinality of a $b$-MATCHING is bounded by $\beta\, n$. In every iteration of the GREEDY algorithm, an edge with maximum marginal gain can be chosen in $O(m)$ time.

Hence the time complexity of the GREEDY algorithm is $O(\beta\, n\, m)$. The worst-case running time of the LAZY GREEDY algorithm is no better than the GREEDY algorithm [56]. However, by making a reasonable assumption we can show a better time complexity bound for the LAZY GREEDY algorithm.

The adjacent edges of an edge $\mathrm{e} = (u, v)$ constitute the set $N(\mathrm{e}) = \{\mathrm{e}' : \mathrm{e}' \in \delta(u) \text{ or } \mathrm{e}' \in \delta(v)\}$. Likewise, the adjacent vertices of a vertex $u$ are defined as the set $N(u) = \{v : (u, v) \in \delta(u)\}$.

**Assumption 1.** The marginal gain of an edge e depends only on its adjacent edges.

With this assumption, when an edge is added to the matching only the marginal gains of adjacent edges change. We make this assumption only to analyze the runtime of the algorithms but not to obtain the quality of the approximation. This assumption is applicable to the objective function in Problem 1.3 that has been used in many applications, including the one considered in this thesis of load balancing Fock matrix computations

**Lemma 5.1.1.** *Under Assumption 1, the time complexity of Algorithm 12 is* $O(\beta\, m \log m)$.

*Proof.* The time complexity of Algorithm 12 depends on the number of push and pop operations in the max heap. We bound how many times an edge e is pushed into the heap. The edge e is pushed when its updated marginal gain is less than the current top's marginal gain, and thus the number of times the marginal gain of e is updated is an upper bound on the number of push operations on it. From our assumption, the update of the marginal gain of an edge e can happen at most $2\beta$ times. Hence an edge is pushed into the priority queue $\mathrm{O}(\beta)$ times, and each of these pushes can take $O(\log m)$ time. Thus the runtime for the all pushes is $O(\beta\, m \log m)$. The number of pop operations are at most the number of pushes. Thus the overall runtime of the LAZY GREEDY algorithm for $b$-MATCHING is $O(\beta\, m \log m)$. ■

## 5.2 Locally Dominant Algorithm

We introduce the concept of $\epsilon$-local dominance, use it to design an approximation algorithm for submodular $b$-MATCHING, and prove the correctness of the algorithm.

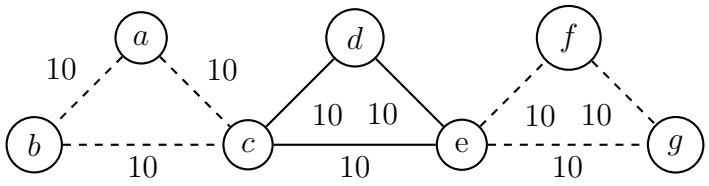



**Figure 5.1.** The original graph.

### 5.2.1 $\epsilon$-Local Dominance and Approximation Ratio

The Lazy Greedy algorithm presented in Algorithm 12 guarantees a $\frac{1}{3}$ approx. ratio [44], [79] by choosing an edge with the highest marginal gain at each iteration, and thus it is an instance of a globally dominant algorithm. We will show that it is unnecessary to select a globally best edge because the same approximation ratio could be achieved by choosing an edge that is best in its neighborhood.

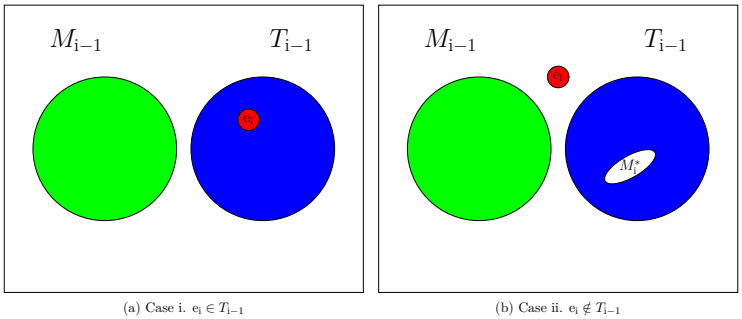
Recall that given a matching $M$, an edge e is *available w.r.t* $M$ if both of its end-points are unsaturated in $M$.

**Definition 5.2.1** (Locally dominant matching)**.** *An edge* e *is* locally dominant *if it is available w.r.t a matching* $M$*, and the marginal gain of* e *is greater than or equal to all available edges adjacent to it. Similarly, for an* $\epsilon \in (0, 1]$*, an edge* e *is* $\epsilon$- locally dominant *if its marginal gain is at least* $\epsilon$ *times the marginal gain of any of its available adjacent edges. A matching* $M$ *is* $\epsilon$-locally dominant *if every edge of* $M$ *is* $\epsilon$*-locally dominant when it is added to the matching.*

A globally dominant algorithm is also a locally dominant one. Thus our analysis of locally dominant matchings would establish the same approximation ratio for the Greedy and Lazy Greedy algorithms.

**Theorem 5.2.1.** *Any algorithm that produces an* $\epsilon$*-locally dominant* $b$-Matching *is* $\frac{\epsilon}{2+\epsilon}$*-approximate for a submodular objective function.*

*Proof.* Let $M^*$ denote an optimal matching and $M$ be a matching produced by an $\epsilon$-locally dominant algorithm. Denote $|M| = k$. We order the elements of $M$ such that when the edge




(a) Case i. $e_i \in T_{i-1}$ (b) Case ii. $e_i \notin T_{i-1}$

**Figure 5.2.** Pictorial representation of the two cases

$e_i$ is included in $M$, it is an $\epsilon$-locally dominant edge. Let $M_i$ denote the locally dominant matching after adding $e_i$ to the set, where $M_0 = \emptyset$ and $M_k = M$.

Our goal is to show that for each edge in the locally dominant algorithm, we may charge at most two distinct elements of $M^*$. At the ith iteration of the algorithm when we add $e_i$ to $M_{i-1}$, we will show that there exists a distinct subset $M^*_i \subset M^*$ with $|M^*_i| \leq 2$ such that $\rho_{e_i}(M_{i-1}) \geq \epsilon\rho_{e^*_j}(M_{i-1})$, for all $e^*_j \in M^*_i$. We will achieve this by maintaining a new sequence of sets $\{T_j\}$, where $T_{i-1}$ is the reservoir of potential edges that $e_i$ could be charged to. The initial set of this sequence of sets is $T_0$, which holds the edges in the optimal matching $M^*$. The sequence of $T$-sets shrink in every iteration by removing the elements charged in the previous iteration, so that it stores only the candidate elements that could be charged in this and future iterations. Formally, $M^* = T_0 \supseteq T_1 \supseteq \cdots \supseteq T_k = \emptyset$ such that for $1 \leq i \leq k$, the following two conditions hold.

i) $M_i \cup T_i$ is also a $b$-MATCHING and

ii) $M_i \cap T_i = \emptyset$.

The two conditions are satisfied for $M_0$ and $T_0$ because $M_0 \cup T_0 = M^*$ and $M_0 \cap T_0 = \emptyset \cap M^* = \emptyset$.

Now we will describe the charging mechanism at each iteration. We need to construct the reservoir set $T_i$ from $T_{i-1}$. Recall that $e_i$ is added at the ith step of the $\epsilon$-locally dominant matching to obtain $M_i$. There are two cases to consider:

i) If $e_i \in T_{i-1}$, the charging set $M^*_i = \{e_i\}$, $M_i = M_{i-1} \cup \{e_i\}$, and $T_i = T_{i-1} \setminus \{e_i\}$.

ii) Otherwise, let $M^*_i$ be a smallest subset of $T_{i-1}$ such that $(M_{i-1} \cup \ldots \cup \{e_i\} \cup T_{i-1}) \setminus M^*_i$ is a $b$-MATCHING. Since a $b$-matching is a 2-extendible system, we know $|M^*_i| \leq 2$. Then

$M_i = M_{i-1} \cup \{e_i\}$; and $T_i = T_{i-1} \setminus M_i^*$.

Note that the two conditions on $M_i$ and $T_i$ from the previous paragraph are satisfied after these sets are computed from $M_{i-1}$ and $T_{i-1}$. Since $M$ is a maximal matching, we have $T_k = \emptyset$; otherwise we could have added any of the available edges in $T_k$ to $M$.

Now when $e_i$ is added to $M_{i-1}$, all the elements of $M_i^*$ are available. This set $M_i^*$ must be the adjacent edges of $e_i$. Thus $\forall e_j^* \in M_i^*$, we have $\epsilon \rho_{e_j^*}(M_{i-1}) \leq \rho_{e_i}(M_{i-1})$. We can sum the inequality for each element of $e_j^* \in M_i^*$, leading to $\sum_j \rho_{e_j^*}(M_{i-1}) \leq \frac{2}{\epsilon} \rho_{e_i}(M_{i-1})$.

Rewriting the summation we have,

$$\begin{aligned}
\rho_{e_i}(M_{i-1}) \geq & \frac{\epsilon}{2} \sum_j \rho_{e_j^*}(M_{i-1}) \\
\geq & \frac{\epsilon}{2} \sum_j \rho_{e_j^*}(M_{i-1} \cup \{e_1^*, \ldots, e_{j-1}^*\}) \\
= & \frac{\epsilon}{2} \sum_j (f(M_{i-1} \cup \{e_1^*, \ldots, e_j^*\}) \\
& - f(M_{i-1} \cup \{e_1^*, \ldots, e_{j-1}^*\})) \\
= & \frac{\epsilon}{2} (f(M_{i-1} \cup \{e_1^*, \ldots, e_{|M_i^*|}^*\}) - f(M_{i-1})) \\
= & \frac{\epsilon}{2} (f(M_{i-1} \cup M_i^*) - f(M_{i-1})) \\
\geq & \frac{\epsilon}{2} (f(M \cup M_i^*) - f(M)).
\end{aligned}$$

In line 2, each of the summands is a superset of $M_{i-1}$, and the inequality follows from submodularity of $f$ (Proposition 1.5.2). Line 3 expresses the marginal gains in terms of the function $f$. The fourth equality is due to telescoping of the sums, the fifth equality replaces the set $M_i^*$ for its elements, and the last inequality follows by monotonicity of $f$ (from Proposition 1.5.2).

We now sum over all the elements in $M$ as follows.

$$\begin{aligned}
\sum_{i} \rho_{e_i}(M_{i-1}) \geq & \frac{\epsilon}{2} \sum_{i} (f(M \cup M_i^*) - f(M)), \\
f(M) \geq & \frac{\epsilon}{2} \sum_{i} (f(M \cup \{M_1^* \cup \dots M_i^*\}) \\
& - f(M \cup \{M_1^*, \dots, M_{i-1}^*\})) \\
= & \frac{\epsilon}{2} (f(M \cup M^*) - f(M)) \\
\geq & \frac{\epsilon}{2} (f(M^*) - f(M)). \\
f(M) \geq & \frac{\epsilon}{2+\epsilon} f(M^*).
\end{aligned}$$

The left side of the second line of the above equations is due to Proposition 1.5.1, while the right side comes from Proposition 1.5.2. The next equality telescopes the sum, and the fourth inequality is due to monotonicity of $f$. Finally the last line is a restatement of the inequality above it. ■

**Corollary 5.2.2.** *Any algorithm that produces an $\epsilon$-locally dominant semi-matching is $\frac{\epsilon}{1+\epsilon}$-approximate for a submodular objective function.*

*Proof.* A semi-matching (there are matching constraints on only one vertex part in a bipartite graph) forms a matroid, which is a 1-extendible system [26]. So by definition of 1-extendible system, $|M_i^*| \leq 1$. We can substitute this value in appropriate places in the proof of Lemma 5.2.1 and get the desired ratio. ■

### 5.2.2 Local Lazy Greedy Algorithm

Now we design a locally dominant edge algorithm to compute a $b$-MATCHING, outlining our approach in Algorithm 14. We say that a vertex $v$ is *available* if there is an available edge incident on it, i.e., adding the edge to the matching does not violate the $b(v)$ constraint.

For each vertex $v \in V$, we maintain a priority queue that stores the edges incident on $v$. The key value of the queue is the marginal gain of the adjacent edges. At each iteration of the algorithm we alternate between two operations: *update* and *matching*. In the *update*

step, we update a best incident edge of an unmatched vertex $v$. Similar to Lazy Greedy, we can make use of the monotonicity of the marginal gains, and the lazy evaluation process is shown in Algorithm 13. After this step, we can consider a best incident edge for each vertex as a candidate to be matched. We also maintain an array (say *pointer*) of size $|V|$ that holds the best vertex found in the *update* step. The next step is the actual *matching*. We scan over all the available vertices $v \in V$ and check whether $pointer(v)$ also points to $v$ (i.e., $pointer(pointer(v)) = v$). If this condition is true, we have identified a locally dominant edge, and we add it to the matching. We continue the two steps until no available edge remains.

**Algorithm 13** Lazy Evaluation (Max Heap pq)

```
1:  while pq is not empty do
2:      Edge e = pq.pop()
3:      Update marginal gain of e
4:      if e is available then
5:          if marg_gain of e ≥ marg_gain of pq.top() then
6:              break
7:          else
8:              push e and its updated gain into pq
9:          end if
10:     end if
11: end while
```

We omit the short proofs of the following two results.

**Lemma 5.2.3.** *The* Local Lazy Greedy *algorithm is locally dominant.*

**Corollary 5.2.4.** *For the* $b$-Matching *problem with submodular objective, the Local Lazy Greedy algorithm is* $1/3$*-approximate.*

**Lemma 5.2.5.** *Under Assumption 1, the time complexity of Algorithm 14 is* $O(\beta\, m \log \Delta)$.

*Proof.* As for the Lazy Greedy algorithm, the number of total push operations is $O(m\beta \log \Delta)$ (the argument of the logarithm is $\Delta$ instead of $m$ because the maximum size of a priority queue is $\Delta$). We maintain two arrays, say *PotentialU* and *PotentialM*, of vertices that hold the candidate vertices for iteration in the *update* and *matching* step, respectively. Initially

**Algorithm 14** Local Lazy Greedy Algorithm

```
    ▷ Initialization
 1: for v ∈ V do
 2:     pq(v) := max-heap of the incident edges keyed by marginal gain
 3:     pointer(v) = pq(v).top
 4: end for

    ▷ Main Loop
 5: while ∃ an edge with both its endpoints available do
    ▷ Updating
 6:     for v ∈ V such that u is available do
 7:         Update pq(v) using Lazy Evaluation (pq(v))
 8:         pointer(v) = pq(v).top
 9:     end for

    ▷ Matching
10:     for  u ∈ V such that u is available do
11:         v = pointer(u)
12:         if v is available and pointer(v) == u then
13:             M = M ∪ {u, v}
14:         end if
15:     end for
16: end while
```

all the vertices are in *PotentialU* and *PotentialM* is empty. The two arays are set to empty after their corresponding step. In the *update* phase, we insert the vertices for which the marginal gain changed into *PotentialM*. In the *matching* step, we iterate only over the vertices in *PotentialM* array. When an edge $(u, v)$ is matched in the *matching* step, we insert $u$, $v$ if they are unsaturated and all their available neighboring vertices into the *PotentialU*. This is the array on which in the next iteration, *update* would iterate. Since a vertex $u$ can be inserted at most $b(u) + \sum_{v \in N(u)} b(v)$ times into the array, the overall size of *PotentialU* array during the execution of the algorithm is $O(m\beta)$. The *PotentialM* is always a subset of *PotentialU*. So it is also bounded by $O(m\beta)$.Combining all these we get, an $O(\beta\, m \log \Delta)$ time complexity. ■

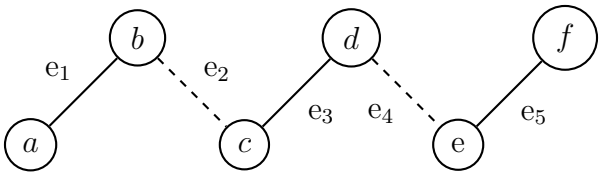



**Figure 5.3.** A tight graph.

### 5.2.3 A tight input for locally subdominant Submodular $b$-Matching

Here we demonstrate tightness of the greedy 1/3-approximate Submodular $b$-Matching. For simplicity, we will work on $b = 1$. Consider the graph in Figure 5.3. The weight of the edges are all 1. Let $x_i$'s be the binary variable that takes $\{0,1\}$. The objective function we are considering is $f = x_1 + x_2 + x_3 + x_4 + x_5 - x_2x_4$. $f$ is monotone submodular. To see this, take two sets $A = \mathrm{e}_1$, $B = \mathrm{e}_1, \mathrm{e}_2$. we add $\mathrm{e}_4$ to $A$ resulting a marginal gain of 1, But when $\mathrm{e}_4$ is added to $B$ the marginal gain is 0. It is monotone because adding new element to set does not lower the functional value. Now the greedy variants of algorithms (including the local lazy greedy) could choose $\mathrm{e}_2, \mathrm{e}_4$ as the matching output that results $f = 0+1+0+1-1 = 1$, whereas the optimal one is $\mathrm{e}_1, \mathrm{e}_3, \mathrm{e}_5$ with $f = 1 + 0 + 1 + 0 + 1 - 0 = 3$.

### 5.2.4 Parallel Implementaion of Local Lazy Greedy

Both the standard Greedy and Lazy Greedy algorithm offer little to no concurrency. The Greedy algorithm requires global ordering of the gains after each iteration, and the Lazy Greedy has to maintain a global priority queue. On the other hand, the Local Lazy Greedy algorithm is concurrent. Here local dominance is sufficient to maintain the desired approximation ratio. We present a shared memory parallel algorithm based on the serial Local Lazy Greedy in Algorithm 15.

One key difference between the parallel and the serial algorithms is on maintaining the *potentialU* and *potentialM* arrays. One option is for each of the processors to maintain individual *potentialU* and *potentialM* arrays and concatenate them after the corresponding steps. These arrays may contain duplicate vertices, but they can be handled as follows. We maintain a bit array of size of $n$ initialized to 0 in each position. This bit array would be reset

**Algorithm 15** Parallel Local Lazy Greedy

```
    ▷ Initialization
 1: for v ∈ V in parallel do
 2:     pq(v) := max-heap of the incident edges keyed by marginal gain
 3:     pointer(v) = pq(v).top
 4: end for

    ▷ Main Loop
 5: while ∃ an edge where both endpoints are available do
    ▷ Updating
 6:     for v ∈ V such that v is available in parallel do
 7:         Update pq(v) according to Lazy Evaluation (pq(v))
 8:         pointer(v) = pq(v).top
 9:     end for

    ▷ Matching
10:     for u ∈ V such that u is available in parallel do
11:         v = pointer(u)
12:         if v is available and u < v and pointer(v) == u then
13:             Mark (u, v) as a matching edge
14:         end if
15:     end for
16: end while
```

to 0 at every iteration. We only process vertices that have 0 in its corresponding position in the array. To make sure that only one processor is working on the vertex, we use an atomic `test-and-set` instruction to set the corresponding bit of the array. Thus the total work in the parallel algorithm is the same as of that the serial one i.e., $O(\beta\, m \log \Delta)$. Since the fragment inside the while loop is embarrassingly parallel, the parallel runtime depends on the number of iterations. This number depends on the weights and the edges in the graph, but in the worst case, could be $O(\beta n)$. We leave it for future work to bound the number of iterations under different weight distributions (say random) and different graph structures.

### 5.3 Experimental Results

The experiments on the serial algorithm were run on an Intel Haswell CPUs with 2.60 GHz clock speed and 512 GB memory. The parallel algorithm was executed on an Intel

Knights Landing node with a Xeon Phi processor (68 physical cores per node) with 1.4 GHz clock speed and 96 GB DDR4 memory.

### 5.3.1 Dataset

We tested our algorithm on both real-world and synthetic graphs shown in Table 5.1. (All Tables and Figures from this section are at the end of the paper.) We generated two classes of RMAT graphs: (a) G500, representing graphs with skewed degree distributions from the Graph 500 benchmark [72] and (b) SSCA, from the HPCS Scalable Synthetic Compact Applications graph analysis (SSCA#2) benchmark using 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. Moreover, we considered eight problems taken from the SuiteSparse Matrix Collection [73] covering application areas such as medical science, structural engineering, and sensor data. We also included a large web-crawl graph(*eu-2015*) [74] and a movie-interaction network(*hollywood-2011*) [75].

**Table 5.1.** The properties of the test graphs listed by increasing number of edges.

| Problems | Vertices | Edges | Mean Degree |
|---|---|---|---|
| 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 | 118,594,475 | 113 |
| SSA21 | 2,097,152 | 123,097,397 | 117 |
| eu-2015 | 11,264,052 | 257,659,403 | 46 |
| nlpkkt240 | 27,993,600 | 373,239,376 | 27 |

### 5.3.2 Serial Performance

In Table 5.2 we compare the Local Lazy Greedy algorithm with the Lazy Greedy algorithm. Each edge weight is chosen uniformly at random from the set $[1, 5]$. The submodular function employed here is the concave polynomial with $\alpha = 0.5$, and $b = 5$ for each vertex. Since both Lazy Greedy and Local Lazy Greedy algorithms have equal approximation ratios, the objective function values computed by them are equal, but the Local Lazy Greedy algorithm is faster. For the largest problem in the dataset, the Local Lazy Greedy algorithm is about five times faster than the Lazy Greedy, and it is about three times faster in geometric mean.

**Table 5.2.** The objective function values and comparison of the serial run times for the Lazy Greedy and Local Lazy Greedy algorithms.

| Problems | Weight | Time (sec.) | | Rel. Perf |
|---|---|---|---|---|
| | | LG | LLG | LG/LLG |
| Fault_639 | 3.07E+06 | 61.05 | 16.83 | 3.63 |
| mouse_gene | 1.90E+05 | 50.68 | 22.41 | 2.26 |
| Serena | 6.69E+06 | 155.81 | 40.27 | 3.87 |
| bone010 | 4.80E+06 | 177.37 | 44.15 | 4.02 |
| dielFilterV3real | 5.35E+06 | 221.92 | 62.22 | 3.57 |
| Flan_1565 | 7.63E+06 | 310.31 | 72.00 | 4.31 |
| kron_g500-logn21 | 3.69E+06 | 304.85 | 105.58 | 2.89 |
| hollywood-2011 | 8.59E+06 | 622.73 | 163.26 | 3.81 |
| G500_21 | 3.93E+06 | 344.13 | 137.06 | 2.51 |
| SSA21 | 9.46E+06 | 588.16 | 285.79 | 2.06 |
| eu-2015 | 2.40E+07 | 1098.40 | 396.16 | 2.77 |
| nlpkkt240 | 1.31E+08 | 2456.34 | 465.30 | 5.28 |
| Geo. Mean | | | | **3.29** |

### 5.3.3 Parallel Performance

Performance of the parallel implementations of the Local Lazy Greedy algorithm is shown by a scalibility plot in Figure 5.4. Figure 5.4 reports results from a machine with 68 threads, with all the cores on a single socket. We see that all problems show good speedups, and all but three problems show good scaling with high numbers of threads.

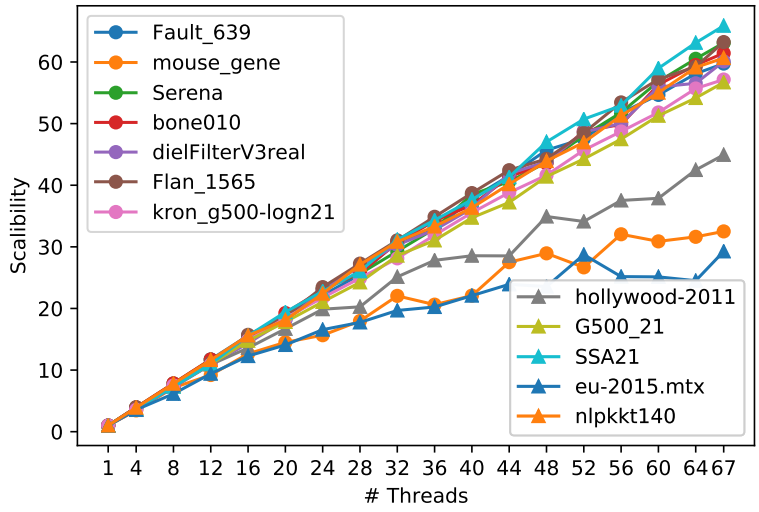



**Figure 5.4.** Scalability of the Local Lazy Greedy algorithm for submodular $b$-matching with 67 threads.

### 5.3.4 Effect of $\alpha$ in Concave Polynomial

In this experiment we vary $\alpha$ of the concave polynomial function from 0.1 to 1 with spacing of 0.1. In Figure 5.5, we plot the $\alpha$ Vs. linear weight (the left axis) and Cardinality (the right axis) of the matching. we observe how the weight function and cardinality of the matching changes with $\alpha$. The weight function here is the linear weight of the corresponding matching. We see that, for all the problems decreasing $\alpha$ increases the cardinality of the matching. We also see that for many problems decreasing $\alpha$ also increases the linear weight. It is surprising that although we are solving for the submodular maximization, the matching output is a better alternative than the greedy matching algorithm for linear objective function.

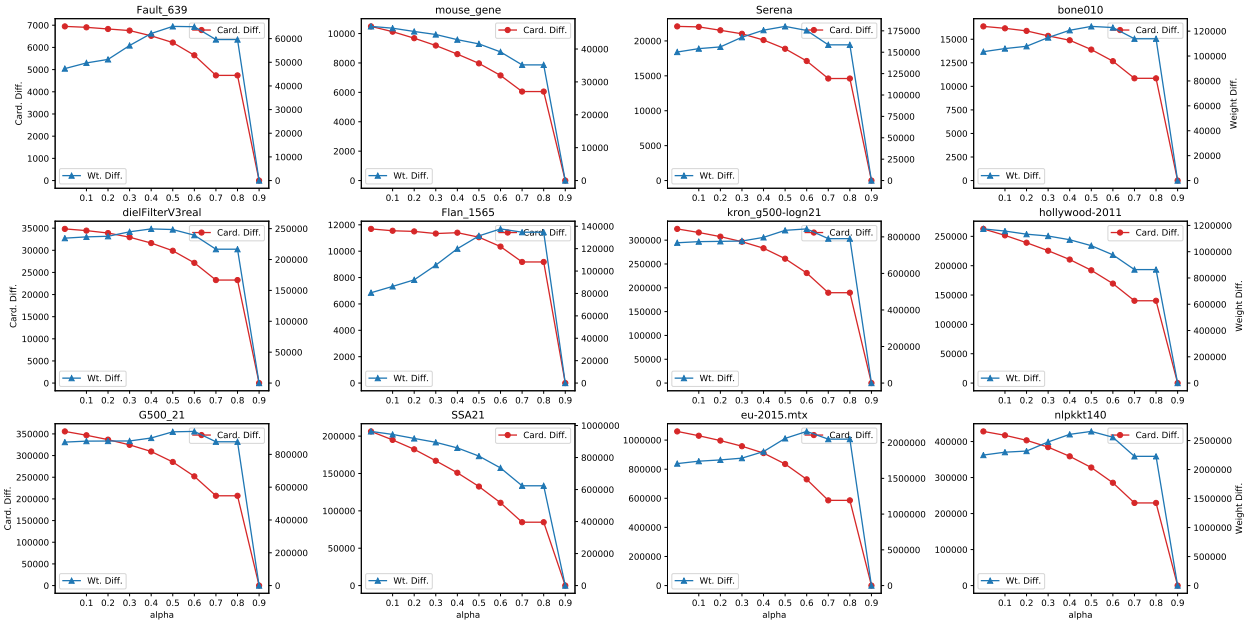



**Figure 5.5.** Cardinality and weight of matching comparison for various $\alpha$

# 6. HEAVY WEIGHT HIGH CARDINALITY MATCHING

In many scientific applications, we may require matchings with heavy weights but also high cardinalities. In this chapter, we will introduce LAMBDA MATCHING ,that provides a trade-off between the weight and cardinality of the matching. This is achieved by formulating the matching problem as a bi-objective optimization problem.

## 6.1 Cardinality sensitive matching formulation

A matching is an independent set of edges in a graph. We restate the maximum weight matching formulation described in Chapter 1.

Consider a graph $G(V, E, w)$, where $V$ and $E$ are the vertex and edge sets respectively, and $w$ is a positive weight function defined on the edges. We denote the number of vertices $|V|$ by $n$ and the number of edges $|E|$ by $m$ throughout this section. The maximum weight matching problem on the graph $G$ can be formulated as follows.

$$\begin{aligned} &\max \sum_{\mathrm{e}\in E} w(\mathrm{e})x(\mathrm{e}) \\ &\text{subject to} \quad \sum_{\mathrm{e}\in\delta(v)} x(\mathrm{e}) \leq 1, \\ &x(\mathrm{e}) \in \{0,1\}. \end{aligned} \tag{6.1}$$

Here $\delta(v)$ is the set of edges incident on vertex $v$ and $x$ is the characteristic vector of the matching.

The formulation (6.1) maximizes only the weight function, and does not consider the cardinality of the matching. We can reformulate the matching problem to account for the cardinality by introducing a non-negative parameter $\lambda$, and modifying the formulation to

$$\begin{aligned} &\max \sum_{\mathrm{e}\in E} w(\mathrm{e})x(\mathrm{e}) + \lambda \sum_{\mathrm{e}\in E} x(\mathrm{e}) \\ &\text{subject to} \quad \sum_{\mathrm{e}\in\delta(v)} x(\mathrm{e}) \leq 1, \\ &x(\mathrm{e}) \in \{0,1\}. \end{aligned} \tag{6.2}$$

The formulation (6.2), called LAMBDA MATCHING, has two terms in its objective function. The first term by itself would maximize the weight of a matching while the second term by itself would maximize its cardinality. The parameter $\lambda \geq 0$ incorporates the trade-off between these two objectives. If $\lambda = 0$, then this produces a maximum weight matching as in Problem (6.1). It is intuitive that increasing $\lambda$ would also increase the cardinality of the matching and possibly decrease the weight of the matching in terms of the original edge weights. Ultimately when the value of $\lambda$ is sufficiently large, we will obtain a maximum cardinality matching from this formulation. We note that computing a LAMBDA MATCHING has the same time complexity as computing a maximum weight matching since the modification amounts to translating the original weight of every edge by $\lambda$. We can obtain the weight of a matching in terms of the original weights of the edges by subtracting $\lambda$ from the weight of each matched edge.

### 6.1.1 Lower bound on the weight

As we increase $\lambda$ the weight of the matching (projected to the original edge weights) may decrease. In this section we will establish a lower bound on the weight of the transformed matching projected to the original edge weights. Let $M_*^\lambda$ be the optimum matching returned by the algorithm using the parameter value $\lambda$, and let $W_\lambda$ be the total weight of this matching. Then $M_*^0$ denotes the maximum weight matching of $G$ when $\lambda = 0$, i.e., this would be the maximum sum weight matching in $G$. We also have

$$\begin{aligned} W_\lambda(M_*^\lambda) &= \sum_{e \in M_*^\lambda} w(e) + \lambda |M_*^\lambda| \\ &= W_0(M_*^\lambda) + \lambda |M_*^\lambda|. \end{aligned} \tag{6.3}$$

The first term on the right-hand-side is the weight of the matching in terms of the original weights of the edges, and the second term reflects the additional weight due to the translation of the edge weights by $\lambda$. Thus $W_0(M_*^0)$ is the weight of the optimum matching with $\lambda = 0$. Let $M^c$ be a matching of maximum weight among all maximum cardinality matchings in $G$. We are interested in computing a lower bound for the weight $W_0(M_*^\lambda)$.

**Lemma 6.1.1.** $W_0(M_*^\lambda) \geq \max\{W_0(M^c) + \lambda|M^c|, W_0(M_*^0) + \lambda|M_*^0|\} - \lambda|M_*^\lambda|$.

*Proof.* Since $M_*^\lambda$ is the maximum weight matching for Lambda Matching, the weight of this matching must be at least the weight of the matching $M_*^0$ translated by $\lambda$. Using a similar argument we can show that the weight of $M_*^\lambda$ is equal to or larger than the weight of $M^c$ translated by $\lambda$. Thus we have

$$\begin{aligned} W_\lambda(M_*^\lambda) &\geq \max\{W_\lambda(M^c), W_\lambda(M_*^0)\} \\ &\geq \max\{W_0(M^c) + \lambda|M^c|, \\ &\qquad W_0(M_*^0) + \lambda|M_*^0|\}. \end{aligned}$$

We can substitute Eq. 6.3 on the left side of the inequality to obtain the lower bound we seek.

$$\begin{aligned} W_0(M_*^\lambda) + \lambda|M_*^\lambda| &\geq \max\{W_0(M^c) + \lambda|M^c|, \\ &\qquad W_0(M_*^0) + \lambda|M_*^0|\} \\ W_0(M_*^\lambda) &\geq \max\{W_0(M^c) + \lambda|M^c|, \\ &\qquad W_0(M_*^0) + \lambda|M_*^0|\} - \lambda|M_*^\lambda|. \end{aligned}$$

■

### 6.1.2 Pareto Optimality of Weight and Cardinality

To show the Pareto optimality between weight and cardinality, we have to prove:

1. The weight $W_0(M_*^\lambda)$ is largest among matchings with cardinality $|M_*^\lambda|$, and
2. The cardinality of $M_*^\lambda$ is the highest among matchings with weight $W_0(M_*^\lambda)$.

Note that if $\lambda = 0$, the second condition may not always be satisfied. For example, consider a path graph with 4 vertices $\{a, b, c, d\}$ with edge weights $(a, b) = (c, d) = 1$ and $(b, c) = 2$. This has two maximum weight matchings, namely $(a, b), (c, d)$ and $(b, c)$. For $\lambda = 0$ the Lambda Matching could return the latter matching, thus violating the second condition.

**Lemma 6.1.2.** *For $\lambda > 0$, the matching computed by solving* Lambda Matching *is Pareto optimal w.r.t weight and cardinality.*

*Proof.* (Proof of weight:) For a contradiction, suppose there exists another matching $M_1^\lambda$ with $|M_1^\lambda| = |M^\lambda|$ but $W_0(M_1^\lambda) > W_0(M^\lambda)$. Then we have $W_\lambda(M_1^\lambda) > W_\lambda(M^\lambda)$, and $M_*^\lambda$ cannot be an optimum matching.

(Proof of cardinality:) For a contradiction, suppose we have a matching $M_1^\lambda$ with weight $W_0(M_1^\lambda) = W_0(M_*^\lambda)$ but $|M_\lambda^1| > |M_*^\lambda|$. As $\lambda > 0$, we see $\lambda|M_\lambda^1| > \lambda|M_*^\lambda|$. Again $W_\lambda(M_1^\lambda) > W_\lambda(M_*^\lambda)$, which is a contradiction. ■

### 6.1.3 Choosing a suitable value of $\lambda$

We require some matching concepts for this discussion. Given a matching $M$ of a graph, a path or cycle $P$ is called *alternating* if it consists of edges chosen alternatively from $M$ and $E \setminus M$. An *augmenting path* with respect to the current matching is an alternating path that begins and ends with unmatched vertices; by switching matching edges to non-matching edges and *vice versa* we can increase the cardinality of the matching. Now we can develop some guiding principles for choosing the value of $\lambda$.

**Lemma 6.1.3.** *Let $\gamma$ be the maximum weight and $\delta$ be the minimum weight of the edges in $G$. If $\lambda = \max\{\frac{(k-1)}{2}\gamma - \frac{(k+1)}{2}\delta, 0\} + \varepsilon$, where $\varepsilon > 0$, then $M_*^\lambda$ obtained from* Lambda Matching *has the maximum weight among all matchings such that there exists no augmenting path of length $k(\geq 3)$ or less w.r.t the matching.*

*Proof.* Suppose choosing $\lambda$ as mentioned in the lemma does not lead to a maximum weight matching with the augmenting path length guarantee. So there is an augmenting path, say $P$, of length $k$ w.r.t to the optimal matching $M_*^\lambda$. In $P$, there are $\frac{k-1}{2}$ matched edges and $\frac{k+1}{2}$ unmatched edges. Let $\Delta$ be the change of weight if we augment along $P$. First assume

that $\frac{(k-1)}{2}\gamma > \frac{(k+1)}{2}\delta$. Since the weight of each matched edge in $P$ can be at most $\gamma$ and the weight of each matched edge can be at least $\delta$, we have

$$\begin{aligned}\Delta &\geq \frac{k+1}{2}\delta + \frac{k+1}{2}\lambda - \frac{(k+1)}{2}\gamma - \frac{k-1}{2}\lambda \\ &= \frac{k+1}{2}\delta + \frac{k^2-1}{4}\gamma - \frac{(k+1)^2}{4}\delta + \frac{(k+1)}{2}\varepsilon \\ &- \frac{k-1}{2}\gamma - \frac{(k-1)^2}{4}\gamma + \frac{k^2-1}{4}\delta - \frac{k-1}{2}\varepsilon \\ &= (\frac{k+1}{2} - \frac{k-1}{2})\varepsilon \quad > 0.\end{aligned}$$

We have substituted for $\lambda$ in the second line. The last line follows since $\frac{k+1}{2} > \frac{k-1}{2}$.

Now if $\frac{(k-1)}{2}\gamma \leq \frac{(k+1)}{2}\delta$, we have

$$\begin{aligned}\Delta &\geq \frac{k+1}{2}\delta + \frac{k+1}{2}\lambda - \frac{(k-1)}{2}\gamma - \frac{k-1}{2}\lambda \\ &= (\frac{k+1}{2}\delta - \frac{(k-1)}{2}\gamma) + (\frac{k+1}{2} - \frac{k-1}{2})\varepsilon \ > 0.\end{aligned}$$

The second line follows by replacing $\lambda$ by $\varepsilon$. The final inequality is due to the fact that $\frac{(k-1)}{2}\gamma \leq \frac{(k+1)}{2}\delta$, and $\frac{k+1}{2} > \frac{k-1}{2}$.

Hence in both cases the augmentation increases the weight, which is a contradiction since we began with a maximum weight matching. Thus such an augmenting path does not exist with respect to the matching $M_*^\lambda$ in the graph. ■

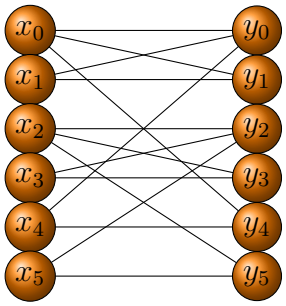
**Corollary 6.1.4.** *The* LAMBDA MATCHING *obtained by setting* $\lambda = \max\{\frac{(n-2)}{4}\gamma - \frac{(n+2)}{4}\delta, 0\} + \varepsilon$, *where* $n$ *is the number of vertices in the graph and* $\varepsilon > 0$, *is a matching of maximum weight among all maximum cardinality matchings of* $G$.

*Proof.* The maximum length of an augmenting path of a graph is $n/2$ ( or $(n-1)/2$ if $n$ is odd). We get the desired $\lambda$ using $k = n/2$ in Lemma 6.1.3. ■

# 7. ADAPTIVE ANONYMIZATION USING $b$-EDGE COVER

## 7.1 A Generalized Framework

| username | | | | |
|---|---|---|---|---|
| Anna ($x_0$) | 1 | 0 | 0 | 0 |
| Robert ($x_1$) | 0 | 0 | 0 | 0 |
| Paul ($x_2$) | 0 | 0 | 1 | 1 |
| Peter ($x_3$) | 1 | 0 | 1 | 1 |
| Carl ($x_4$) | 1 | 1 | 0 | 0 |
| Olaf ($x_5$) | 0 | 1 | 1 | 1 |




| | | | | key |
|---|---|---|---|---|
| * | * | 0 | 0 | 172 ($y_0$) |
| * | 0 | 0 | 0 | 236 ($y_1$) |
| * | * | 1 | 1 | 672 ($y_2$) |
| * | 0 | 1 | 1 | 229 ($y_3$) |
| 1 | * | 0 | 0 | 761 ($y_4$) |
| 0 | * | 1 | 1 | 298 ($y_5$) |

**Figure 7.1.** An example of an *adaptive anonymity* problem. Left: usernames and feature matrix $(x, X)$; Right: the anonymized feature matrix with keys $(Y, y)$; Center: A bipartite graph that matches each user to a set of anonymized keys compatible with the user's data. There are six users and four features, and the privacy requirements are: $k(x_0) = 3, k(x_1) = 2, k(x_2) = 3, k(x_3) = 2, k(x_4) = 2, k(x_5) = 2$. The solution using adaptive anonymity masks eight data items, while $k$-Anonymity for $k \geq 2$ would mask ten elements.

In this section we give a precise mathematical description of the problem and an algorithm that achieves *adaptive anonymity* and high utility of the shared data at the same time. We start with the definition of adaptive anonymity, which is generalized from $k$-anonymity proposed in [80].

**Definition 7.1.1.** *A release of data is said to have the adaptive-anonymity property if the information for each individual $v$ contained in the release cannot be distinguished from the information of at least $k(v) - 1$ individuals in the dataset.*

The difference between *adaptive anonymity* and $k$-anonymity is that the latter uses a uniform value $k$ for all individuals instead of a value $k(v)$ for each individual $v$. For $k$-anonymity, the value of $k$ has to be the maximum of $k(v)$ for all $v$ to satisfy the privacy requirements. If there exists a user who would like their record to be confused with all others, in the $k$-anonymity setting, the obfuscated data will have little utility.

Our model is illustrated in Figure 7.1. We are given a dataset $X \in \mathbb{Z}^{n \times f}$, where $n$ is the number of individuals and $f$ is the number of features. Each row $x_v \in \mathbb{Z}^f$ of $X$ is a

contribution of the user $v$ to the dataset and consists of $f$ discrete features. A feature might be race, age, height, weight, income bracket, etc. A vector $\mathbf{k}$ of length $n$, where an element $k(v)$ of $\mathbf{k}$ is a privacy parameter of the $v$-th user is also given. The value $k(v)$ specifies that the data of the $v$-th user must be indistinguishable from that of $k(v) - 1$ other users. The output of the algorithm is an anonymized dataset $Y \in (\mathbb{Z} \cup \{*\})^{n \times f}$, where the $*$ symbol indicates that a particular feature has been masked. Each feature vector $x_v \in X$ is associated with a username $x_v \in \mathbb{Z}$ and each row $y_u$ of $Y$ is associated with a key $y_u \in \mathbb{Z}$. Keys are output together with a matrix $Y$. The sensitive information that needs to be hidden from an *adversary* (one who is trying to discover the identity of the users from the value of their features) is the matching between usernames $x_v$ and corresponding keys $y_u$. We call this the *canonical matching*. Thus, the goal of the adversary is to reveal as many edges of the canonical matching as possible. The engine must publish data in such a way that instances with larger privacy parameter $k(v)$ have a smaller probability of being matched with the corresponding key by the adversary.

We say that feature vector $x_v \in X$ is *compatible* with vector $y_u \in Y$ if $x_v(l) = y_u(l)$ for every $1 \leq l \leq f$ such that $y_u(l) \neq *$. Hence $y_u$ can be obtained from (confused with) $x_v$ after masking some attributes of $x_v$. Thus either the feature values agree between the instances or we can match a '*' in the second vector to 0, 1, or * in the first.

In the suppression model analyzed here the goal is to mask as few attributes in $X$ as possible to produce $Y$ (to get as high utility of the published data as possible), but in such a way that each entry $x_v$ of $X$ can be confused with at least $k(v)$ rows $y_u$ in $Y$. Thus we have the following measure of the utility of the presented scheme.

**Definition 7.1.2.** *The utility of the suppression model is the ratio of the number of unmasked features and the total number of features, $nf$. The goal of the database algorithm producing obfuscated data is to minimize the number of masked features such that all privacy constraints are met.*

Given a dataset $X$, we define a function $\gamma(\mathrm{i}, \mathrm{j}, l)$ as follows,

$$\gamma(\mathrm{i}, \mathrm{j}, l) = \begin{cases} 1 & \text{if } X_{\mathrm{i}l} \neq X_{\mathrm{j}l}, \\ 0 & \text{otherwise.} \end{cases}$$

For an undirected graph $G$ with an adjacency matrix $\mathbf{G} \in \{0,1\}^{n \times n}$ and a dataset $X$, we define the *Hamming distance* $h(G)$ as

$$h(G) = \sum_{\mathrm{i}} \sum_{\mathrm{j}} \mathbf{G}_{\mathrm{ij}} \sum_{l} \gamma(\mathrm{i}, \mathrm{j}, l).$$

Here $\mathbf{G}_{\mathrm{ij}}$ is either zero or one.

Given a graph $G$, we compute an expression for the number of stars to put in the dataset. The maximum number of stars one can put is $nf$. Let us consider a node i in the graph $G$, and a column $l$ in $X$. Now we find the rows j of column $l$ which correspond to neighbors of the node i, i.e, $\mathbf{G}_{\mathrm{ij}} = 1$. If every such element $X_{\mathrm{j}l}$ is equal to $X_{\mathrm{i}l}$, then we do not have to put a star in $Y_{\mathrm{i}l}$; otherwise we need to put a star in the position $Y_{\mathrm{i}l}$. Mathematically this can be expressed as follows.

$$s(G) = nf - \sum_{\mathrm{i}} \sum_{l} \prod_{\mathrm{j}} (1 - \mathbf{G}_{\mathrm{ij}} \ \gamma(\mathrm{i}, \mathrm{j}, l)).$$

The second term in this equation counts the positions in the matrix $Y$ where no stars are needed. If $X_{\mathrm{j}l} = X_{\mathrm{i}l}$ then $\gamma(\mathrm{i}, \mathrm{j}, l) = 0$; if this is true for all neighbors j of node i, then every term in the product is 1, and then the value of $Y_{\mathrm{i}l}$ is set to $X_{\mathrm{i}l}$ and not a star.

Choromanski, Jebara and Tang proposed the following method [81] that finds a good-quality approximation of $G$. First the algorithm minimizes $h(G)$ over all graphs $G$ satisfying the privacy requirements. Then a variational upper bound [82] on $s(G)$ is iteratively minimized with the use of the weighted version of the Hamming distance. The first phase of their algorithm solved the $b$-Matching problem exactly.

It is clear from the definition that there are two goals for solving the *adaptive anonymity* problem: group instances to satisfy privacy constraints and hide as little data as possible. Since optimal solution for this problem is NP-complete [81], our approximate solution comes from the observation that if we group similar instances together (w.r.t. their corresponding

features) then we need to hide fewer features. Our proposed *adaptive anonymity* algorithm is shown in Algorithm 16.

**Algorithm 16** Adaptive Anonymity ($X \in \mathbb{Z}^{n \times f}$, $\mathbf{k} \in \mathbb{Z}^{\mathbf{n}}$, $\epsilon$)

Let $c_\epsilon = \log(\frac{\epsilon}{1+\epsilon})$
Initialize $W \in \mathbb{R}^{n \times f}$ to the all ones matrix
Initialize $Y$ to $X$
**while** not converged **do**
    Let $G$ be a weighted graph, where:
    $E_{\mathrm{ij}} = \sum_l (W_{il} + W_{\mathrm{j}l})\gamma(\mathrm{i}, \mathrm{j}, l)$ ▷ Compute Graph
    $C = b$-Edge Cover $(G, E, \mathbf{k} - \mathbf{1})$ ▷ Grouping Step
    **for all** $\mathrm{i}, l$ **do** ▷ Update W
        $W_{\mathrm{i}l} \leftarrow exp(\sum_{\mathrm{j}} C_{\mathrm{ij}}\gamma(\mathrm{i}, \mathrm{j}, l)\ c_\epsilon)$
    **end for**
**end while**
**for all** $\mathrm{i}, l$ **do**
    **if** $C_{\mathrm{ij}} = 1$ and $X_{\mathrm{j}l} \neq X_{il}$ for any j **then**
        $Y_{\mathrm{i}l} = *$
    **end if**
**end for**
$Y_{public} = MY$, where $M$ is a random row permutation of $\mathbb{B}^{n \times n}$

The algorithm is a variational optimization algorithm which iterates until some convergence criterion is met or a maximum number of iterations is reached. First, the algorithm creates a complete graph of $n$ vertices corresponding to instances and a weight multiplier matrix initialized to all ones. Within an iteration, the algorithm assigns the weight of an edge between two vertices based on some dissimilarity measure between the two instances, multiplied by the weight multiplier. Next, the algorithm performs a *grouping* step based on the current weight assignment, and then the weight multipliers are adjusted based on the grouping.

A critical part of Algorithm 16 is how the *grouping* step is done. There are two requirements for the *grouping* step: i) each instance $v$ has to be in a group with $k(v) - 1$ other instances, and ii) "similar" instances should be grouped together in order to minimize number of masked data elements. In order to achieve this goal, we use a $b$-Edge Cover formulation for the *grouping* step.

**Definition 7.1.3.** *A b-*Edge Cover *in a graph is a subgraph C such that every vertex v has* ***at least*** $b(v)$ *edges incident on it in the subgraph. If the edges are weighted, then a cover that* ***minimizes*** *the sum of weights of its edges is a minimum weight b-*Edge Cover*.*

Given the definition of the *b*-Edge Cover, we group instances together with the following three steps:

1. Create a complete graph $G$ where the instances are the vertices.
2. Calculate the edge weight between a pair of vertices using the dissimilarity measure between the corresponding instances. The dissimilarity between two instances is the number of the features in which the instances do not agree.
3. Set $b(v) = k(v) - 1$ for each vertex, and solve a *b*-Edge Cover problem with the input $(G, b)$.

Since *b*-Edge Cover is a minimization problem, it will group less dissimilar, i.e., more similar vertices together; each vertex $v$ is grouped with $k(v) - 1$ other vertices. We use a 2-approximation algorithm called the MCE algorithm for the *b*-Edge Cover problem [64], [67]. we have compared the anonymizations obtained with the 3/2-approx and 2-approx *b*-Edge Cover algorithms, LSE and MCE respectively, and found less than 1% difference in utility. The details of the MCE algorithm are explained the next section.

Now we provide more details of our *b*-Edge Cover based formulation of adaptive anonymity, in Algorithm 16. The algorithm starts by initializing the weight multiplier matrix $W \in \mathbb{R}^{n \times f}$, to all 1's. The matrix $W$ associates a weight $W_{\text{ij}} \geq 1$ to each entry of input dataset $X_{\text{ij}}$, which the algorithm updates at each iteration. The algorithm iterates until the utility measure converges or a maximum number of iterations is reached. At each iteration, we compute edge weights in the graph as the weighted sum of the product of the weight multipliers and the dissimilarity between two instances. Then we group instances using a *b*-Edge Cover $C$ in the graph and the $k(v)$ values. Finally, we update the weight multipliers based on the following rule: We proportionally increase the multiplier value associated with the feature $l$ of the instance i, $W_{il}$, based on how many times the feature $X_{il}$ differs with $X_{\text{j}l}$, the corresponding feature of other instances j grouped together with

i. We increase the weight multiplier of a feature when it has the potential to create more maskings because we compute an approximate minimum weight $b$-Edge Cover. When the algorithm converges, we mask a feature if it does not agree with other values of the same feature in the same group. Then we publish a row-permuted copy of the masked data. The time complexity of Algorithm 16 per iteration is as follows: the *Compute Graph* step has complexity of $O(n^2 f)$; the *Update W* step has complexity of $O(nkf)$, where $k$ is $max(k(v))$; and the *grouping Step* has time complexity $O(n^2 logn)$ if an exact $b$-Edge Cover algorithm is used. If a 2-approximate $b$-Edge Cover algorithm is used, the time complexity of the last step becomes $O(\beta m)$, where $\beta = max_{v \in V} b(v)$.

We discuss theoretical guarantees on the quality of the computed solution below. We will need the following lemma.

**Lemma 7.1.1.** *For any graph $G$, the Hamming distance $h(G)$ and the number of stars $s(G)$ satisfy $s(G) \leq h(G) \leq ks(G)$, where $k = max_{v \in V} k(v)$.*

*Proof.* The first inequality is immediate since we need at most one star for each difference that contributes to the Hamming distance. We consider the contributions that an instance i and a feature $l$ make to $h(G)$ and $s(G)$.

$$\begin{aligned} h(G) &= \sum_{\mathrm{i}} \sum_{\mathrm{j}} \mathbf{G}_{\mathrm{ij}} \sum_{l} \gamma(\mathrm{i}, \mathrm{j}, l) = \sum_{\mathrm{i}} \sum_{l} \mathbf{G}_{\mathrm{ij}} \sum_{\mathrm{j}} \gamma(\mathrm{i}, \mathrm{j}, l) \\ &\equiv \sum_{\mathrm{i}} \sum_{l} h_{\mathrm{i}l}(G). \\ s(G) &= nf - \sum_{\mathrm{i}} \sum_{l} \prod_{\mathrm{j}} (1 - \mathbf{G}_{\mathrm{ij}} \gamma(\mathrm{i}, \mathrm{j}, l) \\ &= \sum_{\mathrm{i}} \sum_{l} (1 - \prod_{\mathrm{j}} (1 - \mathbf{G}_{\mathrm{ij}} \gamma(\mathrm{i}, \mathrm{j}, l))) \\ &\equiv \sum_{\mathrm{i}} \sum_{l} s_{\mathrm{i}l}(G). \end{aligned}$$

Now we consider two cases. Case 1: If for every instance j such that $\mathbf{G}_{\mathrm{ij}} = 1$, we have $\gamma(\mathrm{i}, \mathrm{j}, l) = 0$, then $h_{\mathrm{i}l}(G) = 0$ and $s_{\mathrm{i}l}(G) = 0$, and hence the inequality holds.
Case 2: There is some j such that $\mathbf{G}_{\mathrm{ij}} = 1$ and $\gamma(\mathrm{i}, \mathrm{j}, l) = 1$. Then, considering the worst-case

$$h_{\mathrm{i}l}(G) = \sum_{\mathrm{j}} \mathbf{G}_{\mathrm{ij}} \gamma(\mathrm{i}, \mathrm{j}, l) \leq k.$$

Also $s_{il}(G) = 1$ since we need a star here. Hence

$$h_{il}(G) \leq k \leq k \cdot s_{il}(G).$$

Summing over all i and $l$, we obtain the lemma. ■

**Theorem 7.1.2.** *The first iteration of the while-loop in Algorithm 16 finds a* $b$-Edge Cover $C$ *such that*

$$s(C) \leq \alpha k \min_{G \in \mathbb{B}^{n \times n}} s(G),$$

*where the minimum in the expression is over all adjacency matrices satisfying privacy requirements, and* $\alpha$ *is the approximation ratio of the* $b$-Edge Cover *algorithm.*

*Proof.* Initially the variational matrix $W$ has all elements set to one, and hence in the first iteration the objective of the $b$-Edge Cover is proportional to the *Hamming distance* of $G$. Suppose $\hat{G} = \text{argmin } h(G)$; then $h(C) \leq \alpha\ h(\hat{G})$ since we have an approximation algorithm for $b$-Edge Cover. For any graph $G$, $s(G) \leq h(G)$ from the Lemma. Combining, we have $s(C) \leq h(C) \leq \alpha\ h(\hat{G})$. Now suppose $G^* = \text{argmin } s(G)$. Since $\hat{G}$ minimizes $h(G)$, we have $h(\hat{G}) \leq h(G^*)$. From the Lemma 7.1.1, we have $h(G) \leq k\ s(G)$. Combining all these, $s(C) \leq h(C) \leq \alpha\ h(\hat{G}) \leq \alpha\ h(G^*) \leq \alpha k\ s(G^*)$. ■

We illustrate the Grouping step by a $b$-Edge Cover by the example shown in Figure 7.2, with six instances and six binary features. Each user expects 2-anonymity, i.e., each user wants to be confused with at least one other user. The anonymity algorithm computes a $b$-Edge Cover with $b = 1$ for each node. Given the input data, the anonymity algorithm first constructs a dissimilarity matrix $S$. Each row of the matrix defines the dissimilarity between that instance and all other instances in the input. For example, the second entry of the first row denotes the dissimilarity between user $U_1$ and $U_2$ which is 2, because the instances disagree in features f2 and f4. This dissimilarity matrix acts as the input adjacency matrix to the $b$-Edge Cover algorithm where each entry refers to the weight of an edge. The bold-font entries are the edges included in the $b$-Edge Cover. We see that grouped pairs are: $(U_1, U_2)$, $(U_3, U_4)$ and $(U_5, U_6)$. Next the anonymity algorithm uses this grouped

| Instances | f1 | f2 | f3 | f4 | f5 | f6 |
|---|---|---|---|---|---|---|
| $U_1$ | 1 | 0 | 1 | 0 | 1 | 0 |
| $U_2$ | 1 | 1 | 1 | 1 | 1 | 0 |
| $U_3$ | 0 | 1 | 0 | 1 | 0 | 1 |
| $U_4$ | 0 | 0 | 0 | 0 | 0 | 1 |
| $U_5$ | 1 | 1 | 0 | 0 | 0 | 0 |
| $U_6$ | 1 | 1 | 0 | 0 | 0 | 1 |

| $S$ | $U_1$ | $U_2$ | $U_3$ | $U_4$ | $U_5$ | $U_6$ |
|---|---|---|---|---|---|---|
| $U_1$ | - | **2** | 6 | 4 | 3 | 4 |
| $U_2$ | | - | 4 | 6 | 3 | 4 |
| $U_3$ | | | - | **2** | 4 | 2 |
| $U_4$ | | | | - | 3 | 2 |
| $U_5$ | | | | | - | **1** |
| $U_6$ | | | | | | - |

| Instances | f1 | f2 | f3 | f4 | f5 | f6 |
|---|---|---|---|---|---|---|
| $U_1$ | 1 | * | 1 | * | 1 | 0 |
| $U_2$ | 1 | * | 1 | * | 1 | 0 |
| $U_3$ | 0 | * | 0 | * | 0 | 1 |
| $U_4$ | 0 | * | 0 | * | 0 | 1 |
| $U_5$ | 1 | 1 | 0 | 0 | 0 | * |
| $U_6$ | 1 | 1 | 0 | 0 | 0 | * |

**Figure 7.2.** An example for *adaptive anonymity*. From top to bottom: original input, dissimilarity matrix (Hamming distances) and anonymized output.

output to mask entries in the following manner: for each pair, it finds the dissimilar features and mask those features with a $*$. For example, $U_5$ and $U_6$ are grouped and the instances do not agree on feature f6, so the algorithm puts $*$ in the corresponding f6 entries. As we can see, there are $6 \times 6 = 36$ entries and after one iteration the algorithm masks 10 entries. Thus the utility at this iteration is $1 - 10/36 = 0.722$, i.e., 72%.

An important feature of our framework, specifically in the shared memory context, is that the memory requirement of Algorithm 16 is linear in the number of instances, whereas a state-of-the-art algorithm [81] requires quadratic memory. This significant reduction comes from an interesting property of the MCE algorithm for solving the $b$-Edge Cover problem.

## 7.2 Experiments and Results

We conducted our experiments on Cori, a Cray XC40 supercomputer at NERSC, Berkeley. Each node on Cori consists of two 16-core 2.3 GHz Intel E5-2698 (Haswell) processors with 128 GB RAM. Each core in a node has its own 64 KB L1 cache and 256 KB L2 cache, as well as a 40 MB shared L3 cache per socket. Cori nodes are also interconnected with the Cray Aries network using the Dragonfly topology.

We used the *Intel MPI* implementation for inter-node communication and *OpenMP* for intra-node multi-threading, and compiled the code with the built-in compiler wrapper optimized for the system, *CC-2.5.12* with the flags `-O3 -qopenmp`. Our hybrid implementation used the following MPI-openMP settings: one OpenMP process for each of the 32 cores on a node, and one MPI process per node. Hyper-threading did not improve the performance of our code.

We consider eight datasets for *adaptive anonymity* experiments in Table 7.1. We use four small datasets in our experiments to compare an adaptive anonymity algorithm by Choromanski et al [81], which groups individuals using belief propagation and exact $b$-matching algorithms, with our approximate $b$-edge cover approach. For each of these problems, these authors picked a specific privacy requirement $k$ that varies with each data item, and we use the same values to be consistent with their work. The values for $k$ ranged from 2 to 10. The earlier belief propagation algorithm is not capable of solving the larger problems in our test set. We consider three larger problems from a Machine Learning Repository at University of California, Irvine [83], and one from the Centers for Medicare and Medicaid Services [84], to demonstrate that the approximate $b$-edge cover approach can solve them. We generated $b(v)$ values for each problem as the minimum of the degree of a vertex and a uniform random integer between one and the square root of the number of vertices. For each experiment we repeat the computations three times and report the average run-time. The utility for each problem is invariant since the same $b$-edge cover is computed. Our algorithm terminates if three consecutive iterations do not show any improvement in the utility measure. For smaller problems the algorithm terminates within four iterations and for the larger problems it takes three to nine iterations. However, most of the feature masking occurs in the first iteration.

**Table 7.1.** Problem sets for *adaptive anonymity*.

| **Problem** | **Instances** | **Features** |
|---|---|---|
| Caltech36 | 768 | 101 |
| Reed98 | 962 | 139 |
| Haverford76 | 1,446 | 145 |
| Simmons81 | 1,518 | 140 |
| UCI_Adult | 32,561 | 101 |
| USCensus1990 | 158,285 | 68 |
| Poker_hands | 500,000 | 95 |
| CMS | 745,280 | 512 |

For example, the *CMS* dataset achieves a utility of 79.3% in 6 iterations, whereas the utility after the first iteration is 70.9%.

**Table 7.2.** Comparing run times of the *Belief Propagation (BP)* and MCE algorithms on a single thread of an Intel Haswell.

| **Problem** | **BP** | MCE | **Rel. Perf.** | **Utility Diff** (%) |
|---|---|---|---|---|
| Caltech36 | 13m 15s | 10s | 80 | -0.85 |
| Reed98 | 20m 47s | 22s | 57 | -0.32 |
| Haverford76 | 1h 07m | 55s | 73 | 0.23 |
| Simmons81 | 59m 29s | 45s | 79 | -0.81 |

### 7.2.1 Shared Memory Results

**Table 7.3.** Effect of working set memory on runtimes using the MCE algorithm on 32 cores of an Intel Haswell processor.

| **Problems** | on the fly | $8 * k^h n$ | $32 * k^h n$ | $128 * k^h n$ | Utility |
|---|---|---|---|---|---|
| UCI_Adult | 1m 48s | 18s | **15s** | 20s | 90.4% |
| USCensus1990 | 3h 47m | **13m** | 19m | 22m | 87.1% |
| Poker_hands | >24h | 51m | **46m** | 1h 02m | 81.3% |
| CMS | >24h | **6h 11m** | 6h 48m | 6h 16m | 79.3% |

In Table 7.2, we compare the run-times of the belief propagation algorithm (BP) [81], with the MCE based algorithm. BP algorithms are well-known in Machine Learning, and

have been used to solve a variety of problems including graph matching [85]. We observe that the MCE algorithm is 55 to 80 times faster than the BP algorithm. The last column of Table 7.2 shows that $b$-Edge Cover also achieves this improvement without compromising the utility, since the differences are less than 1%. We proceed to describe results from our parallel algorithm that can solve large problems, which are not feasible for the BP algorithm.

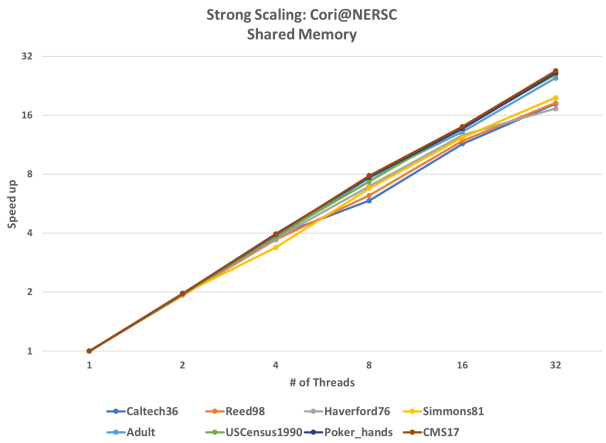



**Figure 7.3.** Strong scaling of *adaptive anonymity* problems on 32 cores of an Intel Haswell processor.

Next we show the impact of the linear memory formulation of the *adaptive anonymity* problem in the shared memory context using 32 cores on one node of Cori. We consider four larger problems to illustrate the effect of trading computation for space. The problems `Poker_hands` and `CMS` have roughly $500K$ and $750K$ instances. Assuming each entry of $E$ is a 4 byte integer, storing the full matrix $E$ would require approximately 1TB and 2TB of memory, respectively, but our machine only has 128GB of memory. Hence we cannot solve these problems with an algorithm that needs the entire dissimilarity matrix for computations. For these problems, we randomly generate $k$ values between $k^l = 5$ and $k^h = 100$ for privacy parameters. We partially generate the dissimilarity matrix using the following strategy: for each row, we generate $t{*}k^h$ entries, with $t \in \{8, 32, 128\}$, yielding a total memory requirement of $t * k^h n$ for $n$ rows in the dissimilarity matrix $E$. We summarize the results in Table 7.3. The fastest result for each problem is indicated in bold font. We observe that "*on the fly*" computation, i.e, no storage for the matrix, is significantly slower than using optimal part

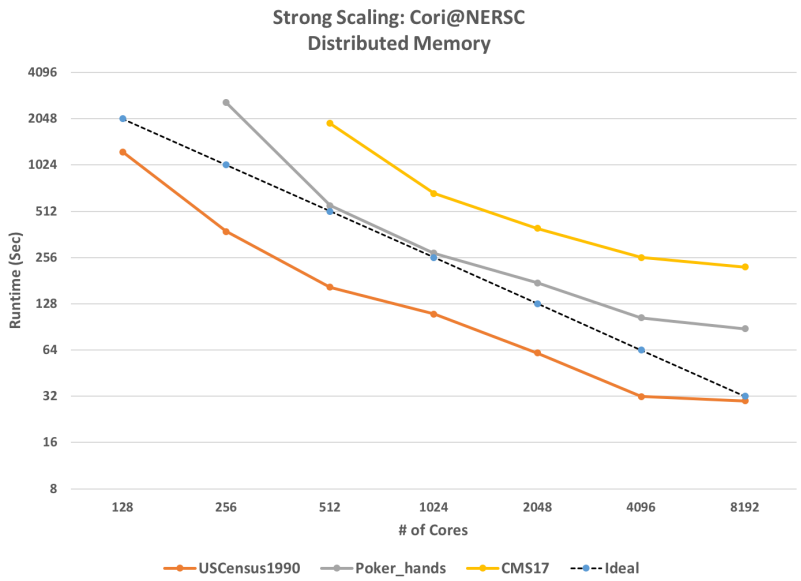



**Figure 7.4.** Strong scaling of *adaptive anonymity* algorithm on Cori using our hybrid MPI-OpenMP code.

sizes. We mask less than 25% of the data elements for these problems. Our linear memory formulation also shows the adaptability of our algorithm in terms of memory constraints. A user can easily choose a size factor $t$ dependent on the memory available, and provide it as an input to the algorithm.

Next we show the strong scaling performance of the *adaptive anonymity* algorithm that uses the MCE algorithm on all eight problems in Figure 7.3. For the larger four problems, we use the best part sizes from Table 7.3. We observe that the algorithm achieves a speedup of 27× on 32 threads for the larger problems. The smaller problems do not scale well beyond 16 threads because the amount of work available per thread is small, and cannot offset the NUMA costs.

### 7.2.2 Distributed Memory Results

We report the strong and weak scaling performance of our algorithm on the three largest problems. The distributed memory implementation does not employ the linear memory formulation, since it is not memory-constrained as the shared memory implementation is.

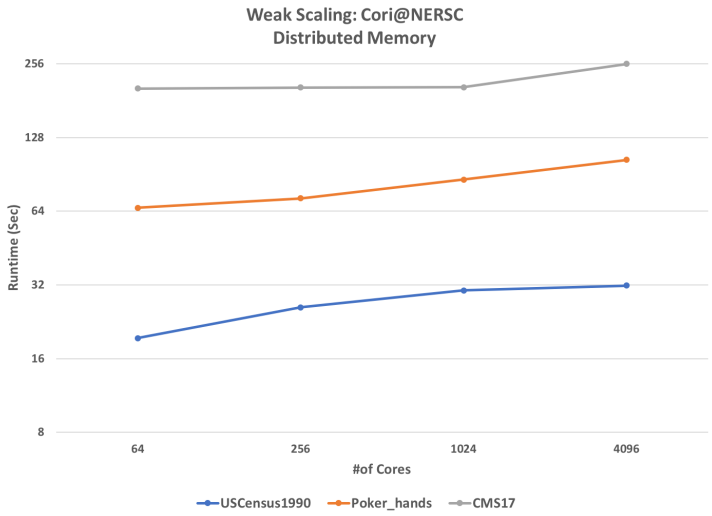



**Figure 7.5.** Weak scaling of *adaptive anonymity* algorithm on Cori.

The implementation uses a hybrid strategy, i.e., each compute node is assigned one *MPI* task for inter-node communication and each node uses 32 *OpenMP* threads for parallel computation. In Figure 7.4, we report the strong scaling performance of our algorithm. An ideal speed-up curve is plotted so that the reader could compare its slope with the observed slopes of the three problems. We observe that the algorithm scales well up to $8,192$ cores, and that initially some problems exhibit super-linear speed-up.

This is due to the smaller memory available on fewer processors. For example, `Poker_hands` roughly requires 1 TB of memory for the data and data structures, but 8 compute nodes (256 cores) have only 1 TB memory in total. Since the operating system requires some memory as well, the problem does not fit in the memory. Hence it is likely that many memory swaps occur, slowing the code for fewer processors. The `UCI_Adult` problem strongly scales to $1,024$ cores but not beyond it, due to its small size.

The *adaptive anonymity* problem has quadratic memory complexity in the number of instances,and hence for each problem we randomly pick 12.5%, 25%, 50% and 100% of all instances and run on 64, 256, 1024 and 4096 cores, respectively. We repeat the process three times for each problem and the report the average run-times in Figure 7.5. Our algorithm

exhibits reasonably good weak scaling as the curves are nearly horizontal, implying that it could potentially scale beyond $8K$ cores with larger problem sizes.

# 8. LOAD BALNCING FOCK MATRIX COMPUTATION USING $b$-MATCHING

## 8.1 Load Balancing in Quantum Chemistry

We show an application of submodular $b$-MATCHING in Self-Consistent Field (SCF) computations in computational chemistry [86].

### 8.1.1 Background

The SCF calculation is iterative, and we focus on the computationally dominant kernel that is executed in every SCF iteration: the two-electron contribution to the Fock matrix build. The algorithm executes forty to fifty iterations to converge to a predefined tolerance.

The two-electron contribution involves a $\Theta(n^4)$ calculation over $\Theta(n^2)$ data elements, where $n$ is the number of basis functions. The computation is organized as a set of $n^4$ tasks, where only a small percentage ($< 1\%$) of tasks contribute to the Fock matrix build. Before starting the main SCF iterative loop, the work required for the Fock matrix build in each iteration is computed from the number of nonzeros in the matrix, which is proportional to the work across all SCF iterations. This step is inexpensive since it only captures the execution pattern of the Fock matrix build algorithm without performing other computations. The task assignment is recorded prior to the first iteration and then reused across all SCF iterations.

The Fock matrix build itself is also iterative (written as a $\Theta(n^4)$ loop), where each iteration represents a task that computes some elements of the Fock matrix. For a given iteration, a task is only executed upon satisfying some domain constraints based on the values in two other pre-computed matrices, the Schwarz and density matrices.

The default load balancing used in NWChemEx [87] is to assign iteration indices of the outermost two loops in the Fock matrix build across MPI ranks using an atomic counter based work sharing approach. All MPI ranks atomically increment a global shared counter to identify the loop iterations to execute. This approach limits scalability of the Fock build since the work and number of tasks across MPI ranks are not guaranteed to be balanced.

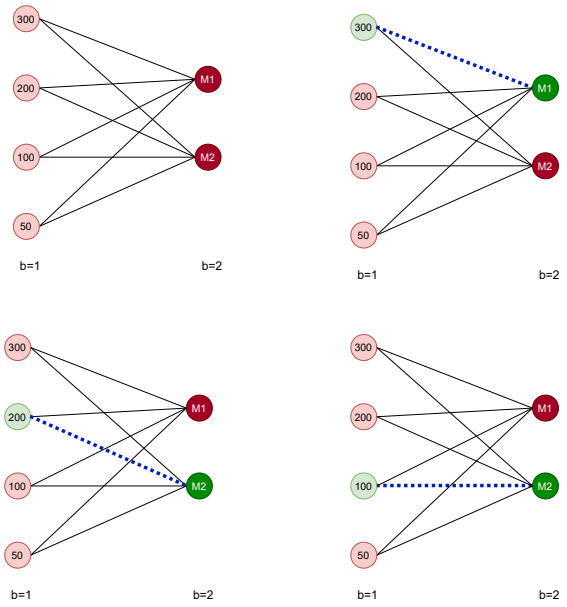



**Figure 8.1.** Assigning tasks to processors to balance the computational work using a submodular $b$-matching.

The task assignment problem here naturally corresponds to a $b$-MATCHING problem. Let $G(U, V, W)$ be a complete bipartite graph, where $U, V, W$ represent the sets of blocks of the Fock matrix, the set of machines, and the load of the (block,machine) pairs, respectively. The $b$ value for each vertex in $U$ is set to 1; for each vertex in $V$, it is set to $\lceil |U|/|V| \rceil$ in order to balance the number of MPI messages that each processor needs to send. We will show that a submodular objective with these $b$-MATCHING constraints implicitly encodes the desired load balance. To motivate this, we use the square root function ($\alpha = 0.5$) as our objective function in Eqn. (1.3).

We consider the execution of the Greedy algorithm for Submodular $b$-MATCHING on a small example consisting of four tasks with work loads of 300, 200, 100 and 50 on two machines $M1$ and $M2$. The $b$-MATCHING constraint requires each processor to be assigned two tasks. At the first iteration, we assign the first block (load 300) to machine $M1$. Note that assigning the second block to machine $M1$ would have the same marginal gain as assigning it to $M2$ if the objective function were linear. But since the square root objective function is submodular, the marginal gain of assigning the second block to the second machine is higher than assigning it to the first machine. So we will assign the second block (load 200) to machine $M2$. Then the third block of work 100 would be assigned to $M2$ rather than $M1$, due to the higher marginal gain, and finally the last block with load 50 would be assigned to $M1$ due to the $b$-MATCHING constraint. We see that modeling the objective by a submodular function implicitly provides the desired load balance, and the experimental results will confirm this.

Assigning tasks to machines is a classic scheduling problem. The most studied objective here is minimizing the makespan, i.e., the maximum total time used by any machine. The problem of makespan minimization can be generalized to a General Assignment Problem(GAP), where there is a fixed processing time and a cost associated with each task and machine pair. The goal is to assign the tasks into available machines with the assignment cost bounded by a constant $C$ and makespan at most $T$. Shmoys and Tardos [88] extended the LP relaxation and rounding approach [89] to GAP. The makespan objective can be a surrogate to the load balancing that we are seeking, but the GAP problem does not encode the

$b$-matching constraints on the machines. Computationally solving a GAP problem entails computing an LP relaxation that is expensive for large problems.

Another possible approach is to model our load balancing problem as a multiple knapsack problem (MKP). In an MKP, we are given a set of $n$ items and $m$ knapsacks such that each item i has a weight (profit) $w_{\mathrm{i}}$ and a size $s_{\mathrm{i}}$, and each knapsack j has a capacity $c_{\mathrm{j}}$. The goal here is to find a subset of items of maximum weight such that they have a feasible packing in the knapsacks. MKP is a special case of GAP [90], and like the GAP, we cannot model the $b(v)$ constraints by MKP.

Our formulation of load balancing has the most similarity with the Submodular Welfare Maximization (SWM) problem [91]. In the SWM problem, the input consists of a set of $n$ items to be assigned to one of $m$ agents. Each agent j has a submodular function $v_{\mathrm{j}}$, where $v_{\mathrm{j}}(S)$ denotes the utility obtained by this agent if the set of items $S$ is allocated to her. The goal is to partition the $n$ items into $m$ disjoint subsets $S_1, \ldots, S_m$ to maximize the total welfare, defined as $\sum_{\mathrm{j}=1}^{m} v_{\mathrm{j}}(S_{\mathrm{j}})$. The greedy algorithm achieves 1/2- approximation ratio [91]. Vondrak's $(1 - 1/\mathrm{e})$-approximation [92] is the best known algorithm for this problem. This algorithm uses continuous greedy relaxation of the submodular function and randomized rounding. Although we have modeled our objective as the sum of submodular functions, unlike the SWM, we have the same submodular function for each machine; our approach could be viewed as Submodular Welfare Maximization with $b$-matching constraints. In the original SWM problem, there are no constraints on the partition size, but in our problem we are required to set an upper bound on the individual partition sizes.

### 8.1.2 Results

As a representative bio-molecular system we chose the Ubiquitin protein to test performance, varying the basis functions used in the computation to represent molecular orbitals, and to demonstrate the capability of our implementation to handle large problem sizes. The assignment algorithm is general enough to be applied to any scenario where such computational patterns exist, and does not depend on the molecule or the basis functions used.

We visualize the load on the processors in Fig. 8.2. The standard deviation for the current assignment is $10^5$, and the coefficient of variation (Std./Avg.) is $7.5 \times 10^{-2}$; while these quantities for the submodular assignment are 436 and $3 \times 10^{-4}$, respectively. It is clear that the latter assignment achieves much better load balance than the former. The run time is plotted against the number of processors in Figure 8.4. It can be seen that the current assignment does not scale beyond 3000 processors, where as the submodular assignment scales to 8000 processors of Summit. The better load balance also leads to a four-fold speedup over the default assignment. Since the Fock matrix computation takes about fifty iterations, we reduce the total run time from 30 minutes to 8 minutes on Summit.

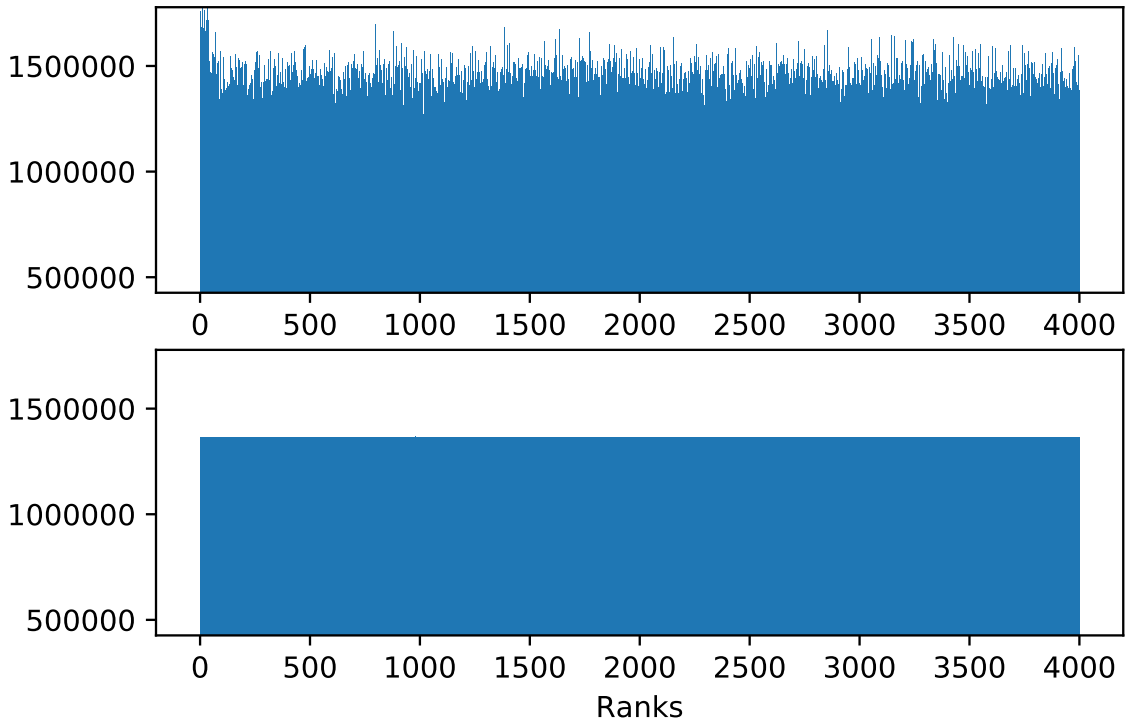



**Figure 8.2.** Visualizing the load distribution for the Fock matrix computation for the Ubiquitin protein. Results from: Top, current assignment on NWChemEx. Bottom, submodular assignment.

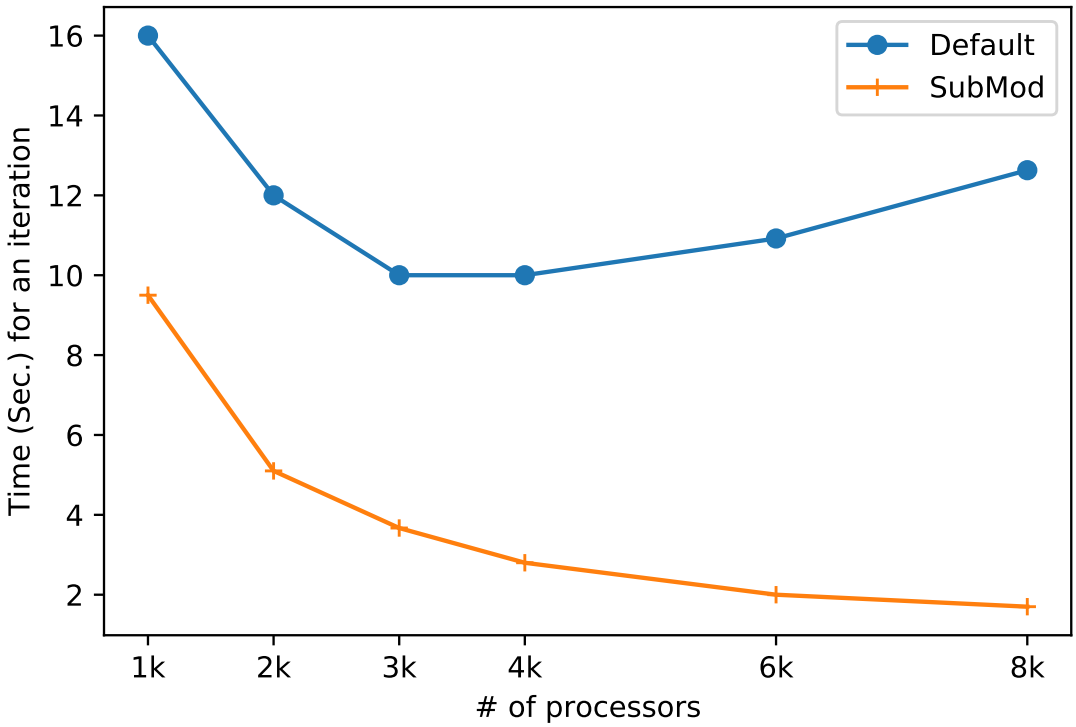



**Figure 8.3.** Runtime comparison per iteration for the default and proposed scheduling with the sto3g basis functions.

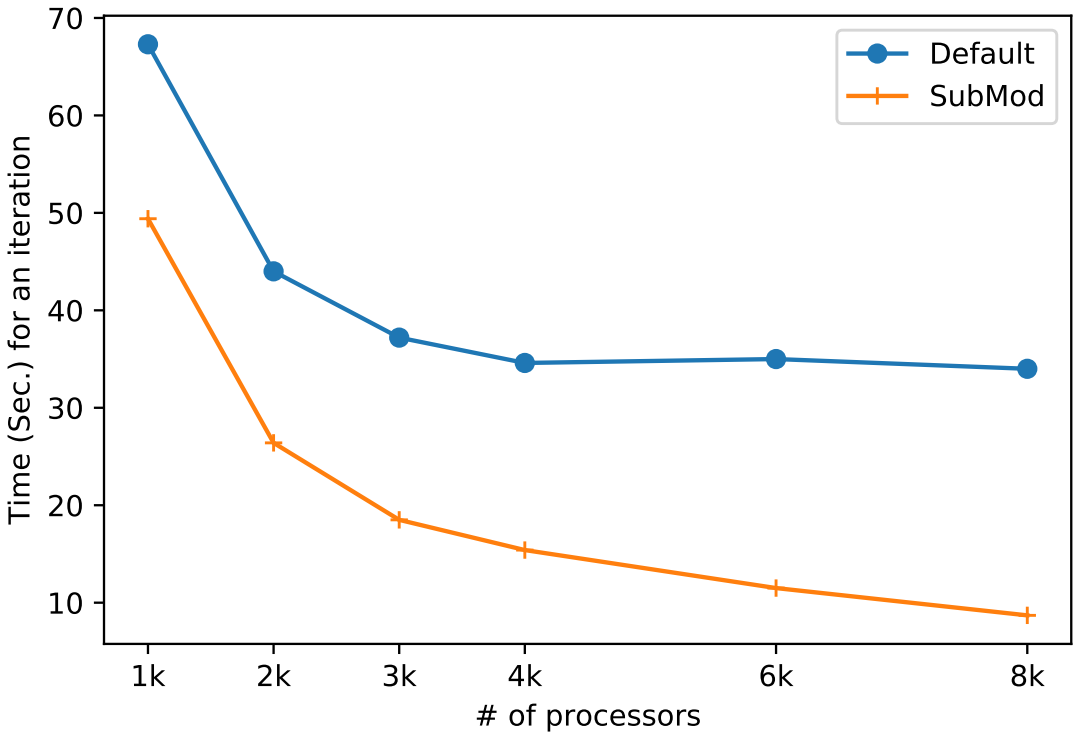



**Figure 8.4.** Runtime per iteration for the current (default) and submodular assignments with the 6-31g basis functions for the Ubiquitin protein in NWChemEx on Summit.

# 9. DEGREE-CONSTRAINED GRAPH CONSTRUCTION FOR MACHINE LEARNING

## 9.1 Background

We have $n$ samples of data and $C$ discrete classes. The data is further divided into $l$ labeled and $u$ unlabeled examples. Let $L = \{(x_1, y_1), \ldots, (x_l, y_l\}$ be the labeled samples, where $(x_\mathrm{i}, y_\mathrm{i})$ is the ith tuple of feature vector and corresponding label. Let also $U = x_{l+1}, \ldots, x_{l+u}$ be the unlabeled data. Here, $n = l + u$, usually $l \ll u$. Our goal is to infer the labels of $U$.

Semi-supervised learning (SSL) is useful in this situation because of the imbalance of the labeled samples. One can also think of SSL as halfway between supervised and unsupervised learning. Most of the SSL algorithms are *transductive* in nature. A *transductive* learning algorithm aims to provide a prediction for only the given test cases. In contrast, an *inductive* learning is more ambitious, where the goal is to learn a rule or function that can predict any test case from the entire space. One of the important assumptions in SSL is the *manifold* assumption, which states that a low dimensional manifold embeds the higher dimensional input data. Graph-based learning employs a graph to approximate this manifold. During the last decade, the graph-based approaches have gained significant attention [93]–[95]. GSSL has been successfully applied to a number of inference tasks. Text classification [96], [97], sentiment analysis [98], question answering [99], part-of-speech tagging [100], web-page classification [101], class-instance acquisition [102], [103], image colourisation [104], and detection of solar photovoltaic arrays [105] are a few examples.

Typically a GSSL consists of two sub-problems: graph construction and label inference. A graph is explicit in some applications, such as a node or link prediction in social networking. Here, we are interested in situations where the graph relationship is not apparent. One has to construct the nodes and edges out of the dataset. In GSSL, data samples become the vertices, and the edges represent the similarity or distance between the examples. Graph construction methods can further be divide into two categories, namely *task-independent* and *task-dependent* construction [94]. *Task-independent* construction does not use the labels, whereas the *task-dependent* methods do the opposite. In this chapter, we consider

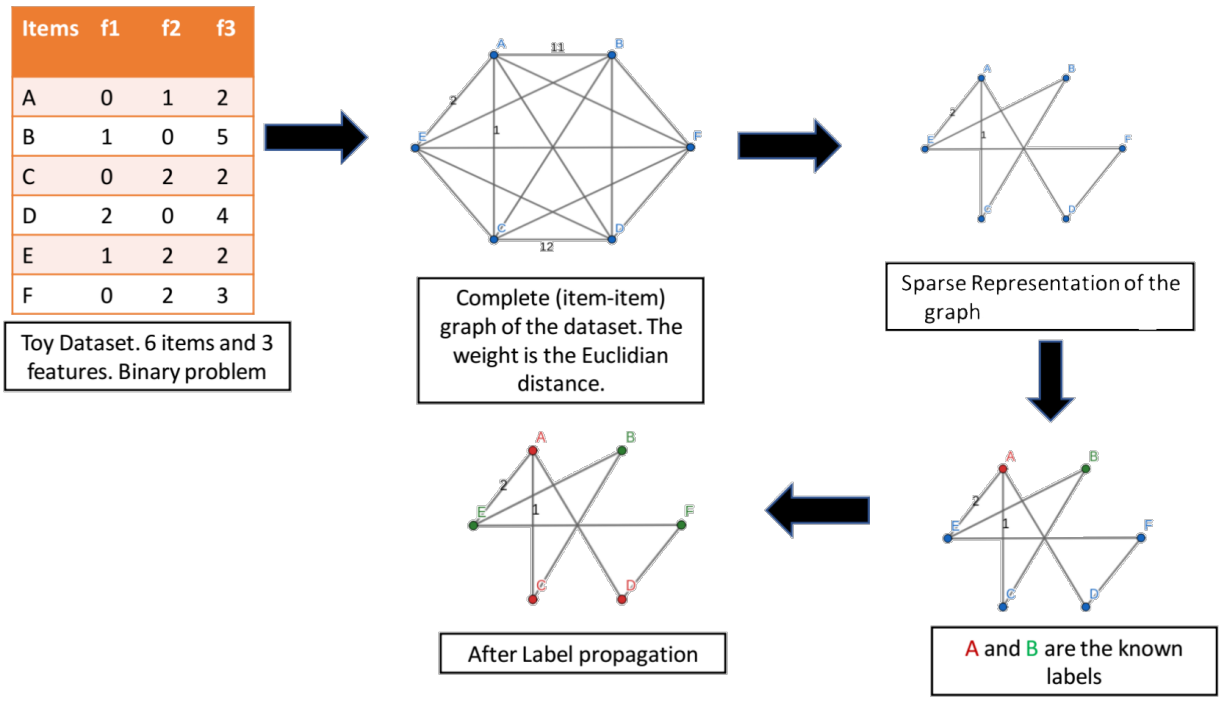



**Figure 9.1.** A typical GSSL flow

only *task-independent* construction. The most commonly practiced way here is creating the nearest neighbor graphs. A $k$-nearest neighbor chooses $k$ top similar items of a sample. A $\epsilon$- neighborhood graph uses a global threshold $\epsilon$ to select the edges.

$k$-NN is simple to construct and also preserves clustering in some sense [106]. However, it also has disadvantages. A $k$-NN graph is highly irregular; thus often has local hubs. The irregularity of $k$-nn graph has been addressed in [107]. In [107], the author showed that this irregularity affects the inference algorithm. To create a regular graph, they suggested employing perfect $b-$matching. A $b$-matching is a subgraph of the complete graph where each vertex has a degree exactly $b$. Modification of the original KNN graph construction approaches is also developed. In [108], the authors applied mutual knn to construct a graph. In mutual knn, an edge is inserted into the graph if and only if this edge is in the top $k$ neighbors on both of the endpoints. In [109], [110], a knn is constructed sequentially using a relevance measure.

The label inference follows the graph construction. In most of the settings, there are few instances with known labels, which are called *seeds*. The seed vertices are initialized with the labels. The earliest and the most popular label inference algorithm is the Gaussian Random Field (GRF) (aka Label Propagation) [111]. GRF obtains a smooth label assignment over the graph keeping the labels on the seed nodes unaltered. Suppose the constructed graph is $G(V, E, w)$, where $V$ is the set of vertices, $E$ is the set of edges, and $w$ is the weight non-negative weight function on edge. Let $S$ be the set of (seed node, label) pairs, and $Y \in 0, 1^{n \times C}$ be the indicator matrix for the labels. Here $Y$ is initialized with seed labels. At every iteration of the label propagation, for all the (node, label) pairs, $Y$ is updated as the normalized weighted sum of all the neighbors of the node. The new labels are then clamped for the seeds to maintain their original labels. It is apparent that for each node $u$, the $u$th row of $Y$ provides a probability distribution over the classes. Once this algorithm converges, we infer the most probable labels from $Y$. The algorithm is presented as pseudocode in Algorithm 17.

Other transductive label inference methods include Local and global consistency [112], Adsorption based approaches [113], [114], information regularization [115], and measure propagation [97].

### 9.2 Sparsification through constraining degree

A simple, undirected, and a weighted graph is represented by sets $V$, $E$, and function $w$ as follows.

- $V$ is the set of vertices. Assume $|V| = n$.
- $E$ is the set of edges. An edge is an unordered tuple $\mathrm{e}\{v_\mathrm{i}, v_\mathrm{j}\}$, where $v_\mathrm{i}, v_\mathrm{j} \in V$. As the graph is simple and undirected we have $v_\mathrm{i} \neq v_\mathrm{j}$ and $\mathrm{e}\{v_\mathrm{i}, v_\mathrm{j}\} = \mathrm{e}\{v_\mathrm{j}, v_\mathrm{i}\}$.
- w is a non-negative function defined on edges where $w(\mathrm{e}) > 0, \forall \mathrm{e} \in E$ and zero otherwise.

Adjacency Matrix of the graph is denoted by $A \in \{0, 1\}^{n \times n}$. Sometimes we represent the weight function as a matrix $w \in \mathbb{R}^{n \times n}$. $A(\mathrm{e}\{v_\mathrm{i}, v_\mathrm{j}\}) = A(\mathrm{e}\{v_\mathrm{j}, v_\mathrm{i}\}) = 1$ for the edges e in $E$

**Algorithm 17** LabelPropagation($G(V, E, W), S, Y$)

$t = 0$
$Y^t = \mathbf{0}$
**for** $u \in U$ **do**
  **if** $u$ is one of the seeds **then**
    $l \leftarrow$ label of $u$
    $Y^t_{ul} = 1$
  **else**
    $k \leftarrow$ random integer between 1 and $C$
    $Y^t_{uk} = 0$
  **end if**
**end for**
**while** not converged **do**
  $t = t + 1$
  **for** $u \in U$ **do**
    **for** $l \leftarrow 1 : C$ **do**
      $Y^t_{ul} = \frac{\sum_{(u,v)\in E} W_{uv} Y^{t-1}_{vl}}{\sum_{(u,v)\in E} W_{uv}}$
    **end for**
  **end for**
  **for** $(u, l) \in S$ **do**
    **for** $k \leftarrow 1 : C$ **do**
      $Y^t_{uk} = 0$
    **end for**
    $Y^t_{ul} = 1$ ▷ Clamping the know labels
  **end for**
**end while**
**for** $u \in U$ **do**
  $Label(u) = argmax_l Y^t_{ul}$
**end for**

and zero otherwise. For notation convenience we will often use $A(\mathrm{e})$ and $w(\mathrm{e})$. The set of incident edges on a vertex is denoted by $\delta(v)$. The degree of a vertex $v$ can be computed as $\sum_{\mathrm{e}\in\delta(v)} A(\mathrm{e})$.

We represent the dataset as $D \in \mathbb{R}^{n\times f}$, where $n$ is the number of entries, and $f$ is the number of features. We can compute the complete graph from this dataset as follows. The entries represent the vertices. For any two entries $\mathrm{i}, \mathrm{j}$ with $\mathrm{i} \neq \mathrm{j}$ we add an edge. The weight function on the edges is the similarity or distance measure between the entries. By the definition of our Adjacency and Weight matrix, it is clear that the $W$ obeys the two properties.

- $A(\mathrm{e}) = 0 \implies w(\mathrm{e}) = 0$
- $A(\mathrm{e}) = 1 \implies w(\mathrm{e}) > 0$

More formally these two properties ensure that $\sum_{\mathrm{e}\in E} w(\mathrm{e})A(\mathrm{e}) = \sum_{\mathrm{e}\in E} w(\mathrm{e})$. We define, $\mathbf{W} = \sum_{\mathrm{e}\in E} w(\mathrm{e})$.

### 9.2.1 The Sparsification Problem

Let $X$ be the adjacency matrix of the sparsified graph. We use the weighted Frobenius norm of a matrix as the objective function to the sparsification problem. This norm weighted low-rank approximation of matrices [116]. Given two matrices, $M$ and $T$ of dimension $p, q$ where each entry of $T$ is positive, the weighted Frobenius norm is defined as $||M||_T \coloneqq \sqrt{\sum_{\mathrm{i,j}}^{p,q} T(\mathrm{i},\mathrm{j})M(\mathrm{e})^2}$. For our purpose, we would adopt the corresponding distance measure namely,

$$||A - X||_w^2$$

Following the definition of the norm, we define the weighted Frobenius distance as

$$||A - X||_w^2 = \sum_{\mathrm{e}\in E} w(\mathrm{e})(A(\mathrm{e}) - X(\mathrm{e}))^2$$

With the degree bound, we can now formalize our optimization problem as follows.

$$\text{optimize} \sum_{\mathrm{e} \in E} w(\mathrm{e})(A(\mathrm{e}) - X(\mathrm{e}))^2$$

$$\text{subject to}$$

$$\sum_{\mathrm{e} \in \delta(v)} X(\mathrm{e}) \text{ is bounded} \tag{9.1}$$

We can expand the objective function as follows.

$$\begin{aligned}
||A - X||_W^2 &= \sum_{\mathrm{e} \in E} w(\mathrm{e})(A(\mathrm{e}) - X(\mathrm{e}))^2 \\
&= \sum_{\mathrm{e} \in E} w(\mathrm{e})(A(\mathrm{e})^2 + X(\mathrm{e})^2 - 2A(\mathrm{e})X(\mathrm{e})) \\
&= \sum_{\mathrm{e} \in E} w(\mathrm{e})(A(\mathrm{e}) + X(\mathrm{e}) - 2A(\mathrm{e})X(\mathrm{e})) \\
&= \sum_{\mathrm{e} \in E} (w(\mathrm{e}) + w(\mathrm{e})X(\mathrm{e}) - 2w(\mathrm{e})X(\mathrm{e})) \\
&= \sum_{\mathrm{e} \in E} (w(\mathrm{e}) - w(\mathrm{e})X(\mathrm{e})) \\
&= \sum_{\mathrm{e} \in E} w(\mathrm{e}) - \sum_{\mathrm{e} \in E} w(\mathrm{e})X(\mathrm{e}) \\
&= \mathbf{W} - \sum_{\mathrm{e} \in E} w(\mathrm{e})X(\mathrm{e}) \qquad (9.2)
\end{aligned}$$

The third line follows because both $A$ and $X$ are binary matrices. The fourth line is due to the properties of the weight matrix $W$. The first part of the equation 9.2 does not depend on the variable $X$. So optimizing $||A - X||_W^2$ amounts to optimizing $-\sum_{\mathrm{e} \in E} W(\mathrm{e})X(\mathrm{e})$.

### 9.2.2 Choice between minimization and maximization

The unconstrained problem of optimizing $-\sum_{\mathrm{e} \in E} w(\mathrm{e})X(\mathrm{e})$ is not particularly useful as depending on minimizing or maximizing; it would compute the empty or full graph. Our notion of sparsification is through bounding the degree. Informally we want to optimize

$-\sum_{e \in E} w(e)X(e)$ with the constraint that the resulting graph has some bound on the degree of the vertices. We explore two types of constraints, namely upper and lower bounds.

### Similarity and upper bound

We assume that the user has a target degree of vertices for the sparsified graph. Let the target degrees be $b(v) \quad \forall v \in V$. We also assume that $w$ here is a similarity measure between the vertices (with an edge). In this scenario the natural objective function is to minimize the weighted Frobenius distance i.e., $||A-X||_W^2$ or maximize $\sum_{e \in E} w(e)X(e)$. The unconstrained optimization would not make any sense because this would produce the complete graph as the optimal solution. This motivates us to design the following optimization problem upper bounding the degrees.

$$\begin{aligned} &\max \sum_{e \in E} w(e)X(e) \\ &\text{subject to,} \\ &\sum_{e \in \delta(v)} X(e) \leq b(v) \forall v \in V \\ &X(e) \in \{0,1\} \end{aligned} \tag{9.3}$$

We note that this discrete optimization problem is solvable in polynomial time and known as $b-$matching. We discuss the exact and approximation algorithms for $b$-matching in the Chapter 1.3 of this thesis.

### Dis-similarity and lower bound

In contrast if our weight matrix represents distance or dis-similarity, one would be interested to maximize $||A - X||_W^2$ which amounts to minimize $\sum_{e \in E} w(e)X(e)$. Once again, it should be intuitive that the unconstrained or only the upper bound constraints on the degrees do not make sense because it would result in an empty graph. As a consequence, if our

weight matrix is a distance measure, we are interested in solving the following optimization problem.

$$\begin{aligned} &\min \sum_{\mathrm{e}\in E} w(\mathrm{e})X(\mathrm{e}) \\ &\text{subject to,} \\ &\sum_{\mathrm{e}\in\delta(v)} X(\mathrm{e}) \geq b(v) \forall v \in V \\ &X(\mathrm{e}) \in \{0,1\} \end{aligned} \tag{9.4}$$

This combinatorial problem has a polynomial solution and known as $b-$edge cover in the literature. We discuss the exact and approximation algorithms for $b$-matching in the Chapter 1.4 of this thesis.

### 9.3 Use of approximation in sparsification

In the previous section, we have established the optimality of the graph sparsification problem through equivalent problems such as optimal $b$-Matching and $b$-Edge Cover. Although these problems have polynomial time solution, often the algorithms are complex and computationally expensive. The best known optimal algorithm for solving optimally $b$-Matching or $b$-Edge Cover requires $\tilde{O}(mn^2)$ time. This is prohibitive in most of the real-life problems.

An alternative approach is to compute approximate solutions of $b$-Matching or $b$-Edge Cover. We would like to analyze the approximation guarantee of the sparsification objective given that the equivalent problems are solved approximately. Let the optimal solution of the $b$-Matching problem is $OPT_{mc}$ and the approximate solution is $APPR_{mc}$. Let also be the approximation factor $0 < \alpha < 1$ i.e., $APPR_{mc} \geq \alpha OPT_{mc}$. It is straightforward from equation 9.2 that the optimal sparsification objective is $\mathbf{W} - OPT_{mc}$. Then we have the following guarantee.

**Lemma 9.3.1.** $\mathbf{W} - APPR_{mc} \leq \mathbf{W} - OPT_{mc} + (1-\alpha)OPT_{mc}$

*Proof.* From the definition of the approximation algorithm,

$$\begin{aligned} APPR_{mc} &\geq \alpha OPT_{mc} \\ \mathbf{W} - APPR_{mc} &\leq \mathbf{W} - \alpha OPT_{mc} \\ &\leq \mathbf{W} - OPT_{mc} + OPT_{mc} - \alpha OPT_{mc} \\ &\leq \mathbf{W} - OPT_{mc} + (1-\alpha) OPT_{mc} \end{aligned}$$

■

For the corresponding lower bound problem a similar result holds. Assuming the optimal and approximate solution of the $b-$edge cover problem respectively as $OPT_{\text{ec}}$ and $APPR_{\text{ec}}$. Also let $\beta > 1$ be the approximation factor i.e., $APPR_{\text{ec}} \leq \beta OPT_{\text{ec}}$, then the following results hold for the sparsification problem.

**Lemma 9.3.2.** $\mathbf{W} - APPR_{\text{ec}} \geq \mathbf{W} - OPT_{\text{ec}} - (\beta - 1) OPT_{\text{ec}}$

The proof is very similar to the lemma 9.3.1.

## 9.4 Preliminary Experiments and Results

In this section, we will report results of our preliminary experiments. Our target application here is text classification.

### 9.4.1 Dataset

For the preliminary experiment, we chose two popular text classification dataset. We describe the dataset next.

#### Reuters-21578

Reuters-21578 [117] corpus consists of 21578 documents in 135 categories. These documents appeared on the Reuters newswire in 1987. We used here the ModApte version. The documents with multiple category labels are discarded. After that, the total number of documents is 8293, with 65 categories.

**20Newsgroup**

The *20Newsgroup* is a popular data set for experiments in text applications of machine learning techniques. It consists of approximately 20,000 messages on newsgroup, and was originally distributed by Lang [118]. The dataset has 20 labels (newsgroups) with (nearly) even distribution across the documents.

We conduct standard pre-processing steps on the dataset. We remove the stop words and generate a bag of words representation of it. Cosine distance and cosine similarity is used for the distance or similarity measure between two examples. For the graph construction, we apply the matching and edge cover construction on a large $k$-nn graph. For example, if we want to create a $b$-edge cover graph, where $b = 10$, we first generate a 30-NN graph, and run our $b$-edge cover algorithm using $b = 10$.

### 9.4.2 Experiment results

We plot in Fig. 9.2, and Fig. 9.3 the quality of the graphs generated by the 3/2-approximate $b$-Edge Cover algorithm and the $b$-Nearest Neighbor algorithms for different values of b for the reuters dataset. We can see that compared to the $b$-Nearest Neighbor graphs, the edge cover graphs are more regular in degree distribution.

We apply the Label propagation algorithm describe in Algorithm 17 in the generated graphs, and calculate the weighted F1 score. We plot the accuracies for Reuters and 20Newsgroup dataset in Fig. 9.4, and Fig. 9.5. As $b$ increases all the graph construction algorithms accuracy drop. But compared to $b$-Nearest Neighbor the approximate $b$-Matching and $b$-Edge Cover graphs provide more stable score.

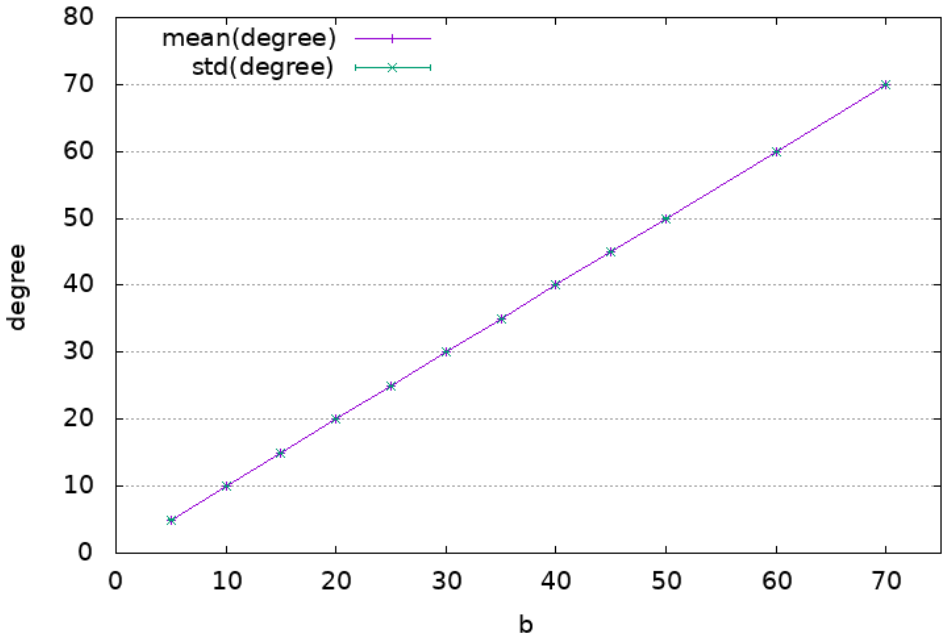



**Figure 9.2.** Degree distribution of the generated graph using 3/2-approx edge cover algorithm

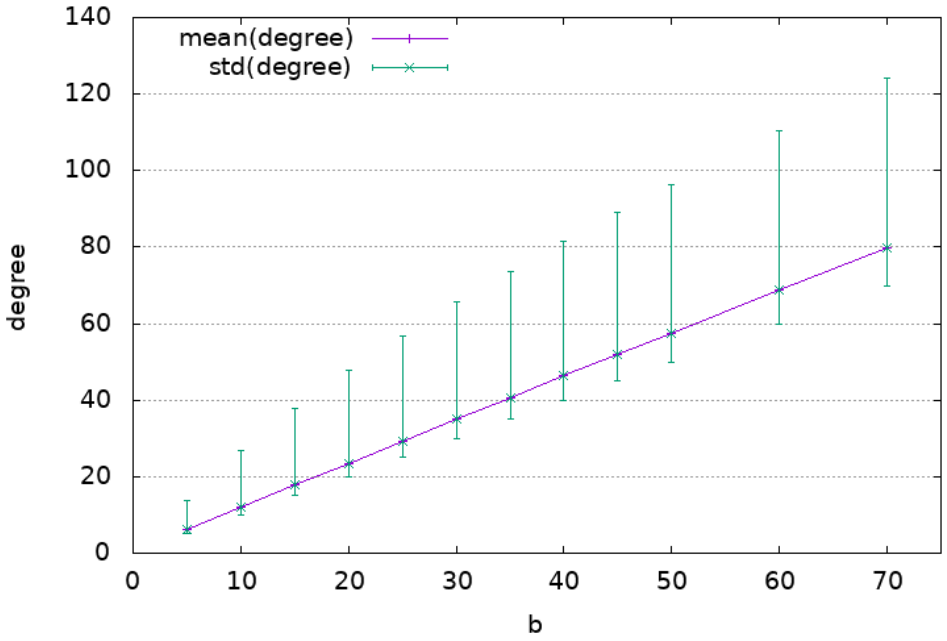



**Figure 9.3.** Degree distribution of the generated graph using 2-approx $k$-NN algorithm

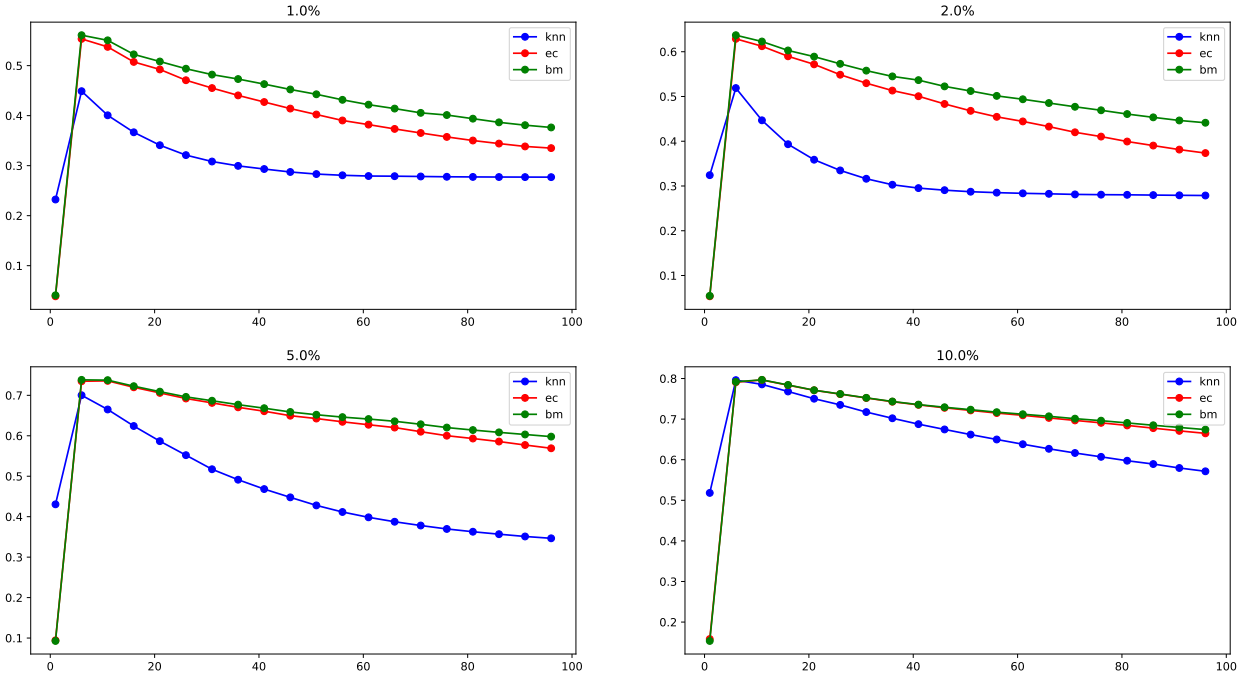



**Figure 9.4.** Weighted F1 of different percentage of labels and different $b$-values for Reuters Dataset

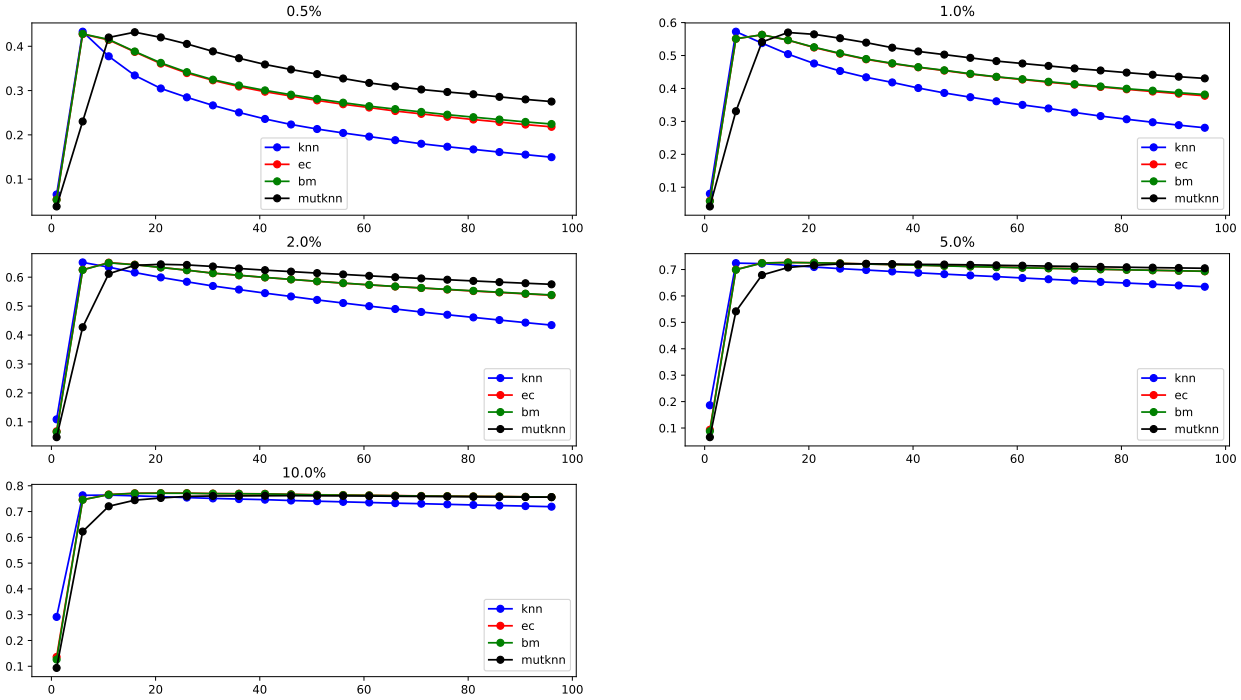



**Figure 9.5.** Weighted F1 of different percentage of labels and different $b$-values for newsgoup Dataset

# 10. CONCLUSION AND FUTURE WORK

## 10.1 Summary

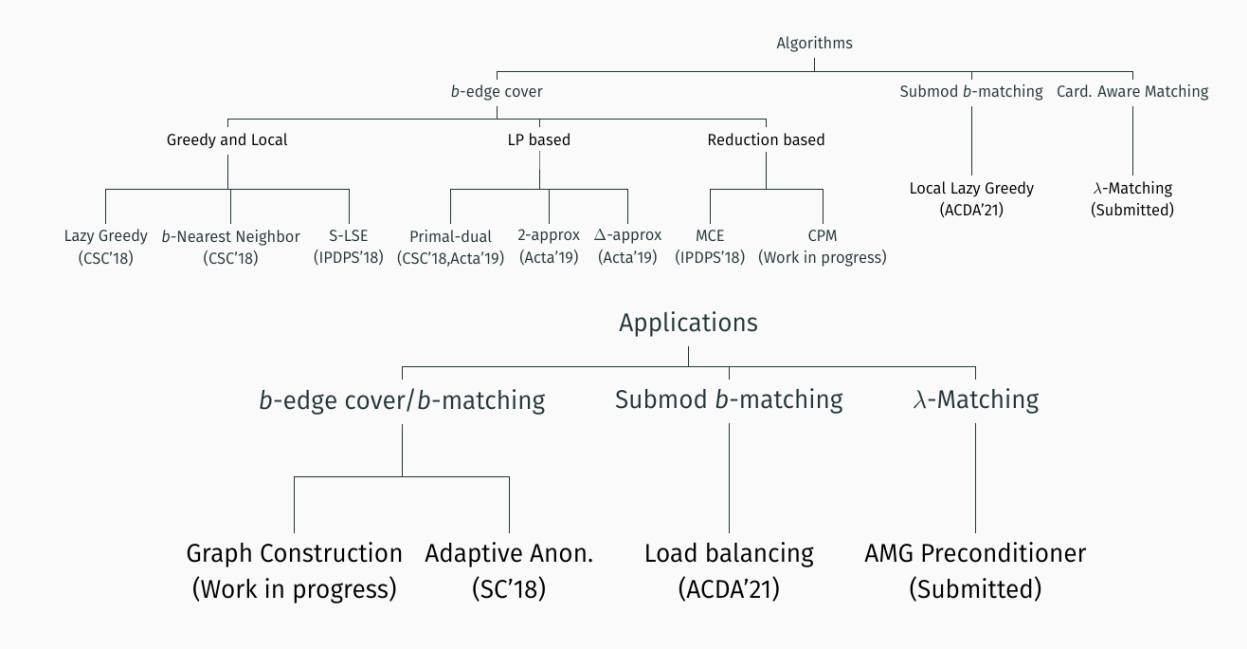



**Figure 10.1.** Contributions of the thesis

We highlight the contributions of the thesis in Figure 10.1. We have investigated several algorithmic paradigms to design approximate $b$-edge cover and $b$-matching. We improved the traditional greedy algorithm for $b$-edge cover by introducing lazy evaluation. We also analyzed the $b$-nearest neighbor algorithm in the edge cover framework and provided its approximation guarantees. Linear Programming based framework is used to develop a primal-dual 3/2-approximate algorithm. We also analyzed several existing algorithms using different Primal-dual algorithmic techniques. Matchings and edge covers are related problems. We use existing and novel reduction techniques to design approximation algorithms for edge cover problems. We developed a 2-approximate highly parallel MCE algorithm using a complimentary relation, and we propose a novel constrained perfect matching based reduction. For the submodular $b$-matching, we introduced the local dominance framework and developed parallel algorithms that combine the local dominance and the lazy greedy algorithm. In many real-life applications, one needs to find matchings that have high cardinalities and

large weights. We augment the traditional matching formulation to account for cardinalities and weights. We show that a simple weight translation achieves Pareto optimality of weights and cardinality.

We also applied the algorithms to aolve various real-life problems. We provide an optimization framework that connects graph construction from data to degree-constrained subgraphs. We also use the edge coves to solve an adaptive anonymity problem. Using a 2-approximate highly parallel MCE algorithm that scales up to 8,000 cores, we were able to anonymize datasets in minutes that took hours previously. We employed submodular b-matchings to generate a balanced assignment of tasks to processors for building Fock matrices in the NWChemEx Chemistry software. This assignment results in a four-fold speedup per iteration using 14,000 processors of the Summit Supercomputer at Oak Ridge National Laboratory.

Next, we discuss few future work related to the thesis.

### 10.2 Practical streaming and online algorithms for graph problem

We are observing an outstanding growth in data generation. The advances in data collections and simulations of scientific experiments have made data generation exceptionally fast relative to its analysis. The enormous amount of data is impossible to analyze as a whole. Even constructing a graph between the observed entities of the dataset often requires more memory than is available. There are also situations where the complete data is not observable, and one needs to analyze it as a stream.

The computational approaches that model this situation correspond to streaming and online algorithms. These algorithms view data as dynamic entities. Streaming algorithms process the input as a sequence of items, which may be examined only a limited number of times (passes), where a pass is complete when we observe the whole data. The length of the sequence is often limited. For example, let $G$ be a graph with $n$ vertices and $m$ edges. In a streaming model, we could only observe $o(n)$ edges of $G$ at a time. The number of passes is often constant (ideally just one). The online algorithms assume an extreme setting where the number of passes is just one, and the input sequence length is $O(1)$.

For graph problems, a relaxed semi-streaming setting [119] is often used. In a semi-streaming algorithm, $O(n\text{polylog}(n))$ edges is observed. The semi-streaming model for graph problems is introduced because most of the problems become provably intractable if the available space is sublinear in $n$, the number of vertices [120].

There exist Primal-dual algorithmic frameworks for both online and streaming algorithms. Buchbinder and Naor [121] proposed a primal-dual approach for designing online algorithms. Using this framework, they designed algorithms for combinatorial problems such as online set cover, generalized caching, maximizing ad-auction revenue, etc. We are interested in utilizing this framework for computing degree-constrained subgraph problems in the online setting.

Problems related to matchings have been recently considered in the semi-streaming setting. A breakthrough in streaming maximum weight MATCHING came in 2018 when Paz and Schwartzman developed a $(1/(2+\epsilon))$ algorithm in a single-pass [122]. The authors utilize a classic offline local-ratio based 2-approximate algorithm [123] due to Bar-Yehuda and Even to design the streaming one. For the submodular MATCHING, the first results are a 1/7.75-approximate MATCHING in a single pass and $1/(3+\epsilon)$-approximation when using $O(\epsilon^{-3})$ passes [124]. Using a randomized primal dual technique, Levin and Wajc [125] developed the best known 1/5.828-approximate submodular $b$-MATCHING and $1/(3+\epsilon)$-approximate linear weight $b$-MATCHING in a single pass. They extended the local-ratio algorithm of Paz and Schwartzman [122] by reinterpreting them in the primal-dual framework. The submodular function was extended to a real function using *concave closure* to formulate a linear program (LP). The algorithm maintains a stack of edges and vertex potentials (the duals). When an edge arrives, the marginal gain of the edge is compared to a function of the corresponding vertex potentials, and the algorithm probabilistically decides on adding the edge to the stack. Finally, at the end of the stream, the stack is unwound greedily in reverse order. One future work could extend this local ratio-based algorithm to covering problems such as edge cover and b-edge covers.

### 10.3 Continuous optimization approaches to combinatorial problems

A significant part of algorithm design is concerned with problems optimizing discrete objects such as graphs or set systems. When optimizing over combinatorial objects, an obvious algorithm exists, i.e., to enumerate all possible solutions and pick the right one. Although this algorithm runs in finite time, the complexity increases exponentially to the size of the problem. Much of the efforts in combinatorial algorithms is to avoid such exponential difficulty by designing algorithmic techniques whose complexity increases algebraically (or polynomially) to the problem size. Traditionally these algorithms are discrete and often leverage the rich theory developed from operation research such as duality and integrality.

An alternative continuous approach to combinatorial algorithms exists. These approaches consist of three steps at a high level: the combinatorial problem of interest is modeled as a continuous optimization formulation. The continuous problem is solved. Finally, the continuous solution is projected back to a discrete one. We mention two early examples of this model.

Weighted matching in a bipartite graph is simpler than the non-bipartite graph as the constraint matrix has a special structure known as total unimodularity. Due to this, one can model the matching problem as a linear program relaxing the integrality. The total unimodularity ensures that the solution from LP relaxation is integral, and from weak duality, this solution is also optimal. Note that here the third step is not necessary since the solution is already discrete.

Computing a minimum weighted vertex cover in a non-bipartite graph is NP-hard. One of the well-known 2-approximation algorithms relaxes the ILP formulation to an LP, and solves the LP. Unlike bipartite matching, we do not have an integrality guarantee. But rounding the real solution to $\{0, 1\}$ leads to a 2-approximation guarantee.

For both problems, solving an LP is the most demanding step. Although used extensively in practice, the classical simplex method to solve an LP is shown to be exponential for some worst-case input. Polynomial-time algorithms such as ellipsoid and interior-point methods exist for solving LPs. The interior-point method is more efficient than the ellipsoid method with runtime depending on the current matrix multiplication complexity, which is $O(n^{2.37})$

at present [126]. It establishes the complexity of LP-based maximum weighted bipartite matching as $O(m^{2.37})$ and vertex cover as $O(n^{2.37})$ time. But the best alternative combinatorial algorithms for both these problems are theoretically and practically faster than the continuous methods.

A breakthrough discovery in applying continuous approaches to combinatorial problems happened when Spielman and Teng [127] showed that the Laplacian system of equations could be solved in nearly-linear time. Using this as a black-box, a continuous method to compute max-flow is developed and shown to improve the runtime of decade-old combinatorial problems significantly [128]. Often the continuous formulation is solved by a gradient-based iterative algorithm. Typically the steps inside an iteration are highly concurrent and should benefit from the modern heterogeneous architecture. Although there is recent work on developing continuous algorithms for matching related problems, there is no known work for the edge covers. One future work is to investigate the continuous approaches to the matching problems and extend them to the edge covers. Another future direction is to implement the algorithms on serial, parallel, and distributed architectures.

## 10.4 Other Future work

There are several short-term future directions that follow from the work in the thesis.

### 10.4.1 Algorithms and implementation of the optimal $b$-matchings and $b$-edge covers

For optimal $b$-matching, direct, flow-based and reduction-based algorithms exist. For optimal $b$-matching, direct, flow-based, and reduction-based algorithms exist. The direct algorithm uses the blossom techniques of Edmond; the flow-based one transforms b-matching to min-cost flow, and the reduction-based approach reduces the $b$-matching problem to the 1-matching problem. One future work could be to implement algorithms using these three paradigms and compare them.

### 10.4.2 Data-locality Sensitive Load-balancing

In NWChemEx, the input matrices for computing the Fock matrix are replicated at each node. This replication allowed us to model the block-processor computation graph as a complete bipartite graph where a block's adjacent edges have uniform weights. For large problems, this replication might be prohibitive. Instead, one could partition the input to the available processors. In most modern supercomputers, processor-to-processor communication is not homogeneous. We propose to incorporate data locality in our submodular assignment. Specifically for Fock matrix computation, our algorithm takes a given partition of input matrices, a network topology (i.e., the communication latency of the processors), and the nonzero pattern of the final Fock matrix. With this information, we form a block-processor bipartite graph. In one partition, we have the blocks of the Fock matrix (the computational units), and the other partition has the available processors. Each computation unit, i.e., each block, has a non-uniform communication cost across the processors. The edge cost could combine the computational cost discovered from the nonzero pattern and the communication cost of moving the block to a processor. In future, We plan to apply our submodular b-matching approach on this block-processor graph to find a load-balanced assignment.

### 10.4.3 Graph Construction from geometric data

Knowing the characteristics of data may also help to devise better algorithms. A special case is when the data items are points in the space. In many machine learning applications, the data is transformed to a feature matrix, where the rows embed the entry in hyper-dimensional space. Due to the metric property that ensures triangle inequalities, the graph construction can be simpler and more efficient than the general case. One future work is to explore such geometric aspects of data for graph construction.

### 10.4.4 Further applications

Combinatorial optimization is exceptionally rich in algorithms and analysis. Future work can bring some of this beautiful theoretical work to life, i.e., connect them with appropriate scientific applications.

One such application is community detection. Community detection techniques are extensively used in many scientific data analyses. But unfortunately, many of these techniques are heuristic and do not provide any provable guarantee on the objective. In particular one can use *Network design* methods, which is a class of combinatorial algorithms that provide clusters of vertices with provable communities.

Another interesting application is when the graphs are from multiple sources, such as protein-protein interaction networks from different tissues or molecule-molecule networks. One mandatory step to analyze these graphs is by graph or network alignment. It is a method to uncover the correspondence among nodes across different networks. These multiple networks are often combined to form a large graph, and one can use the degree-constrained subgraph approach to construct this graph.

# REFERENCES


[1] S. Ferdous, A. Pothen, A. Khan, A. Panyala, and M. Halappanavar, "A parallel approximation algorithm for submodular $b$-matching," in *Proceedings of the First SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*, SIAM, 2021. DOI: 10.1137/1.9781611976830.5.

[2] H. N. Gabow, "An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems," in *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, 1983, pp. 448–456.

[3] L. Addario-Berry, K. Dalal, and B. A. Reed, "Degree constrained subgraphs," *Discrete Applied Mathematics*, vol. 156, no. 7, pp. 1168–1174, 2008.

[4] K. Heinrich, P. Hell, D. G. Kirkpatrick, and G. Liu, "A simple existence criterion for (g< f)-factors," *Discrete Mathematics*, vol. 85, no. 3, pp. 313–317, 1990.

[5] L. Lovász, "The factorization of graphs. II," *Acta Mathematica Academiae Scientiarum Hungarica*, vol. 23, no. 1-2, pp. 223–246, 1972.

[6] A. Arulselvan, Á. Cseh, M. Groß, D. F. Manlove, and J. Matuschke, "Matchings with lower quotas: Algorithms and complexity," *Algorithmica*, vol. 80, no. 1, pp. 185–208, 2018.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1979, ISBN: 0716710447.

[8] O. Amini, D. Peleg, S. Pérennes, I. Sau, and S. Saurabh, "Degree-constrained subgraph problems: Hardness and approximation results," in *International Workshop on Approximation and Online Algorithms*, Springer, 2008, pp. 29–42.

[9] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comput.*, vol. 2, no. 4, pp. 225–231, 1973. DOI: 10.1137/0202019. [Online]. Available: https://doi.org/10.1137/0202019.

[10] A. Goel, M. Kapralov, and S. Khanna, "Perfect matchings in o($n$ log $n$) time in regular bipartite graphs," in *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, L. J. Schulman, Ed., ACM, 2010, pp. 39–46. DOI: 10.1145/1806689.1806697. [Online]. Available: https://doi.org/10.1145/1806689.1806697.

[11] J. Edmonds, "Maximum matching and a polyhedron with 0, 1-vertices," *Journal of research of the National Bureau of Standards B*, vol. 69, no. 125-130, pp. 55–56, 1965.

[12] S. Micali and V. V. V. Vazirani, "An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs," in *Proceedings of the 21st Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE Computer Society, 1980, pp. 17–27. DOI: 10.1109/SFCS.1980.12. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1980.12.

[13] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[14] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, 1987. DOI: 10.1145/28869.28874. [Online]. Available: https://doi.org/10.1145/28869.28874.

[15] J. Edmonds, "Paths, trees, and flowers," *Can. J. Math.*, vol. 17, pp. 449–467, 1965.

[16] 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*, San Francisco, California, USA: Society for Industrial and Applied Mathematics, 1990, pp. 434–443.

[17] W. R. Pulleyblank, "Faces of matching polyhedra," PhD thesis, Faculty of Mathematics, University of Waterloo, 1973.

[18] A. B. Marsh III, "Matching algorithms," PhD thesis, The Johns Hopkins University, 1979, p. 220, ISBN: 9798661646057. [Online]. Available: https://www.proquest.com/dissertations-theses/matching-algorithms/docview/302927413/se-2?accountid=13360.

[19] R. P. Anstee, "A polynomial algorithm for $b$-matchings: An alternative approach," *Information Processing Letters*, vol. 24, no. 3, pp. 153–157, 1987.

[20] D. L. Miller and J. F. Pekny, "A staged primal-dual algorithm for perfect $b$-matching with edge capacities," *ORSA J. of Computing*, vol. 7, pp. 298–320, 1995.

[21] M. W. Padberg and M. R. Rao, "Odd minimum cut-sets and $b$-matchings," *Mathematics of Operations Research*, vol. 7, no. 1, pp. 67–80, 1982, ISSN: 0364765X, 15265471. [Online]. Available: http://www.jstor.org/stable/3689360.

[22] M. Grötschel and O. Holland, "Solving matching problems with linear programming," *Mathematical Programming*, vol. 33, no. 3, pp. 243–259, Dec. 1985, ISSN: 1436-4646. DOI: 10.1007/BF01584376. [Online]. Available: https://doi.org/10.1007/BF01584376.

[23] M. Müller-Hannemann and A. Schwartz, "Implementing weighted b-matching algorithms: Towards a flexible software design," *ACM J. Exp. Algorithmics*, vol. 4, p. 7, 1999. DOI: 10.1145/347792.347815. [Online]. Available: https://doi.org/10.1145/347792.347815.

[24] B. Huang and T. Jebara, "Fast $b$-matching via sufficient selection belief propagation," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 361–369.

[25] W. T. Tutte, "A short proof of the factor theorem for finite graphs," *Canadian Journal of Mathematics*, vol. 6, pp. 347–352, 1954.

[26] J. Mestre, "Greedy in approximation algorithms," in *Proceedings of the 14th European Symposium on Algorithms*, Springer, 2006, pp. 528–539.

[27] D. E. Drake and S. Hougardy, "A simple approximation algorithm for the weighted matching problem," *Information Processing Letters*, vol. 85, no. 4, pp. 211–213, 2003.

[28] A. Khan, A. Pothen, M. M. Patwary, N. Satish, N. Sundaram, F. Manne, M. Halappanavar, and P. Dubey, "Efficient approximation algorithms for weighted $b$-matching," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, S593–S619, 2016.

[29] G. De Francisci Morales, A. Gionis, and M. Sozio, "Social content matching in Mapreduce," *Proceedings of the VLDB Endowment*, vol. 4, no. 7, pp. 460–469, 2011.

[30] C. Koufogiannakis and N. E. Young, "Distributed algorithms for covering, packing and maximum weighted matching," *Distributed Computing*, vol. 24, no. 1, pp. 45–63, 2011.

[31] F. M. Manshadi, B. Awerbuch, R. Gemulla, R. Khandekar, J. Mestre, and M. Sozio, "A distributed algorithm for large-scale generalized matching," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 613–624, 2013.

[32] G. Georgiadis and M. Papatriantafilou, "Overlays with preferences: Distributed, adaptive approximation algorithms for matching with preference lists," *Algorithms*, vol. 6, no. 4, pp. 824–856, 2013.

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

[34] T. Gallai, "Uber extreme punkt-und kantenmengen, annales universitatis scientiarum budapestinensis de rolando eotvos nominatae," *Sectio Mathematica*, vol. 2, pp. 133–138, 1959.

[35] R. Z. Norman and M. O. Rabin, "An algorithm for a minimum cover of a graph," *Proceedings of the American Mathematical Society*, vol. 10, no. 2, pp. 315–319, 1959.

[36] D. Huang and S. Pettie, "Approximate generalized matching: $f$-factors and $f$-edge covers," *CoRR*, vol. abs/1706.05761, 2017. arXiv: 1706.05761. [Online]. Available: http://arxiv.org/abs/1706.05761.

[37] W. Bai, J. Bilmes, and W. S. Noble, "Bipartite matching generalizations for peptide identification in tandem mass spectrometry," in *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, 2016, pp. 327–336.

[38] W. Bai, J. Bilmes, and W. S. Noble, "Submodular generalized matching for peptide identification in tandem mass spectrometry," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 16, no. 4, pp. 1168–1181, 2018.

[39] H. Lin and J. Bilmes, "Word alignment via submodular maximization over matroids," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, 2011, pp. 170–175.

[40] U. Feige, "A threshold of ln n for approximating set cover," *Journal of the ACM (JACM)*, vol. 45, no. 4, pp. 634–652, 1998.

[41] A. Krause and C. Guestrin, "Near-optimal nonmyopic value of information in graphical models," in *Proceedings of the Twenty-first Conference on Uncertainty in Artificial Intelligence*, 2005, pp. 324–331.

[42] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions—I," *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978.

[43] G. L. Nemhauser and L. A. Wolsey, "Best algorithms for approximating the maximum of a submodular set function," *Mathematics of Operations Research*, vol. 3, no. 3, pp. 177–188, 1978.

[44] G. Calinescu, C. Chekuri, M. Pál, and J. Vondrák, "Maximizing a monotone submodular function subject to a matroid constraint," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1740–1766, 2011.

[45] K. Fujii, "Faster approximation algorithms for maximizing a monotone submodular function subject to a $b$-matching constraint," *Information Processing Letters*, vol. 116, no. 9, pp. 578–584, 2016.

[46] A. Badanidiyuru and J. Vondrák, "Fast algorithms for maximizing submodular functions," in *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, SIAM, 2014, pp. 1497–1514.

[47] B. Mirzasoleiman, A. Badanidiyuru, A. Karbasi, J. Vondrák, and A. Krause, "Lazier than Lazy Greedy," in *Proceedings of the Twenty-ninth AAAI Conference on Artificial Intelligence*, AAAI Press, 2015, pp. 1812–1818, ISBN: 0262511290.

[48] B. Mirzasoleiman, A. Karbasi, R. Sarkar, and A. Krause, "Distributed submodular maximization," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 8330–8373, 2016.

[49] N. Buchbinder, M. Feldman, and M. Garg, "Deterministic $(1/2+\varepsilon)$-approximation for submodular maximization over a matroid," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2019, pp. 241–254.

[50] N. Buchbinder and M. Feldman, "Submodular functions maximization problems," in *Handbook of Approximation Algorithms and Metaheuristics, Second Edition, Volume 1: Methologies and Traditional Applications*, T. F. Gonzalez, Ed., Chapman and Hall/CRC, 2018, pp. 753–788. DOI: 10.1201/9781351236423-42. [Online]. Available: https://doi.org/10.1201/9781351236423-42.

[51] A. Krause and D. Golovin, "Submodular function maximization," in *Tractability: Practical Approaches to Hard Problems*, L. Bordeaux, Y. Hamadi, and P. Kohli, Eds., Cambridge University Press, 2014, pp. 71–104.

[52] E. Tohidi, R. Amiri, M. Coutino, D. Gesbert, G. Leus, and A. Karbasi, "Submodularity in action: From machine learning to signal processing applications," *IEEE Signal Processing Magazine*, vol. 37, no. 5, pp. 120–133, 2020.

[53] M. Feldman, J. S. Naor, R. Schwartz, and J. Ward, "Improved approximations for $k$-exchange systems," in *Proceedings of the European Symposium on Algorithms*, Springer, 2011, pp. 784–798.

[54] V. Chaoji, S. Ranu, R. Rastogi, and R. Bhatt, "Recommendations to boost content spread in social networks," in *Proceedings of the 21st International Conference on World Wide Web*, 2012, pp. 529–538.

[55] S. Pettie and P. Sanders, "A simpler linear time $2/3$-$\varepsilon$ approximation for maximum weight matching," *Information Processing Letters*, vol. 91, no. 6, pp. 271–276, 2004.

[56] M. Minoux, "Accelerated greedy algorithms for maximizing submodular set functions," in *Proceedings of the 8th IFIP Conference on Optimization Techniques*, Springer, 1977, pp. 234–243.

[57] D. S. Johnson, "Approximation algorithms for combinatorial problems," *Journal of Computer and System Sciences*, vol. 9, no. 3, pp. 256–278, 1974.

[58] L. Lovász, "On the ratio of optimal integral and fractional covers," *Discrete Mathematics*, vol. 13, no. 4, pp. 383–390, 1975.

[59] S. K. Stein, "Two combinatorial covering theorems," *Journal of Combinatorial Theory, Series A*, vol. 16, no. 3, pp. 391–397, 1974.

[60] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, vol. 4, no. 3, pp. 233–235, 1979.

[61] G. Dobson, "Worst-case analysis of greedy heuristics for integer programming with nonnegative data," *Mathematics of Operations Research*, vol. 7, no. 4, pp. 515–531, 1982.

[62] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[63] 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.

[64] A. Khan, A. Pothen, and S. M. Ferdous, "Parallel algorithms through approximation: $b$-edge cover," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 22–33. DOI: 10.1109/IPDPS.2018.00013.

[65] A. Pothen, S. Ferdous, and F. Manne, "Approximation algorithms in combinatorial scientific computing," *Acta Numerica*, vol. 28, pp. 541–633, 2019. DOI: 10.1017/S0962492919000035.

[66] S. Rajagopalan and V. V. Vazirani, "Primal-dual RNC approximation algorithms for (multi)-set (multi)-cover and covering integer programs," in *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, 1993, pp. 322–331. DOI: 10.1109/SFCS.1993.366855.

[67] S M Ferdous, A. Khan, and A. Pothen, "New approximation algorithms for minimum weighted edge cover," in *Proceedings of SIAM Workshop on Combinatorial Scientific Computing*, 2018, pp. 97–108.

[68] N. G. Hall and D. S. Hochbaum, "A fast approximation algorithm for the multicovering problem," *Discrete Applied Mathematics*, vol. 15, no. 1, pp. 35–40, 1986.

[69] A. Khan, A. Pothen, M. M. Patwary, M. Halappanavar, N. Satish, N. Sundaram, and P. Dubey, "Designing scalable *b*-matching algorithms on distributed memory multiprocessors by approximation," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 773–783.

[70] F. Manne and M. Halappanavar, "New effective multithreaded matching algorithms," in *Proceedings of the 28th International Parallel and Distributed Processing Symposium*, IEEE, 2014, pp. 519–528.

[71] G. E. Blelloch, J. T. Fineman, and J. Shun, "Greedy sequential maximal independent set and matching are parallel on average," in *Proceedings of 24th SPAA*, Pittsburgh, Pennsylvania, USA, 2012, pp. 308–317, ISBN: 978-1-4503-1213-4. DOI: 10.1145/2312005.2312058. [Online]. Available: http://doi.acm.org/10.1145/2312005.2312058.

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

[73] T. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, 1:1–1:25, 2011.

[74] 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.

[75] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, 2004, pp. 595–602.

[76] J. Langguth, S. Ferdous, and A. Pothen, "Faster approximation algorithms for the *b*-edge cover problem," Work in progress, 2021.

[77] A. Khan, K. Choromanski, A. Pothen, S. Ferdous, M. Halappanavar, and A. Tumeo, "Adaptive anonymization of data using *b*-edge cover," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18, Dallas, Texas: IEEE Press, 2018, 59:1–59:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291735.

[78] A. Krause, A. Singh, and C. Guestrin, "Near-optimal sensor placements in Gaussian processes: Theory, efficient algorithms and empirical studies," *Journal of Machine Learning Research*, vol. 9, no. 8, pp. 235–284, 2008. [Online]. Available: http://jmlr.org/papers/v9/krause08a.html.

[79] M. L. Fisher, G. L. Nemhauser, and L. A. Wolsey, "An analysis of approximations for maximizing submodular set functions—II," in *Polyhedral Combinatorics: Dedicated to the memory of D.R. Fulkerson*, M. L. Balinski and A. J. Hoffman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 73–87, ISBN: 978-3-642-00790-3. DOI: 10.1007/BFb0121195. [Online]. Available: https://doi.org/10.1007/BFb0121195.

[80] L. Sweeney, "K-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 5, pp. 557–570, 2002.

[81] K. Choromanski, T. Jebara, and K. Tang, "Adaptive anonymity via $b$-matching," in *27th Annual Conference on Neural Information Processing Systems (NIPS)*, 2013, pp. 3192–3200.

[82] M. I. Jordan, Z. Ghahramani, T. Jaakkola, and L. K. Saul, "An introduction to variational methods for graphical models," *Machine Learning*, vol. 37, no. 2, pp. 183–233, 1999.

[83] K. Bache and M. Lichman, *UCI Machine Learning Repository*, 2013. [Online]. Available: http://archive.ics.uci.edu/ml.

[84] *Centers for Medicare & Medicaid Services*, https://www.cms.gov/OpenPayments/About/Resources.html, Accessed: 2018-02-15.

[85] S. Sanghavi, D. Malioutov, and A. Willsky, "Belief propagation and LP relaxation for weighted matching in general graphs," *IEEE Transactions on Information Theory*, vol. 57, no. 4, pp. 2203–2212, 2011.

[86] T. P. Hamilton and H. F. Schaefer III, "New variations in two-electron integral evaluation in the context of direct SCF procedures," *Chemical Physics*, vol. 150, no. 2, pp. 163–171, 1991.

[87] K. Kowalski, R. Bair, N. P. Bauman, J. S. Boschen, E. J. Bylaska, J. Daily, W. A. de Jong, T. Dunning, N. Govind, R. J. Harrison, M. Keçeli, K. Keipert, S. Krishnamoorthy, S. Kumar, E. Mutlu, B. Palmer, A. Panyala, B. Peng, R. M. Richard, T. P. Straatsma, P. Sushko, E. F. Valeev, M. Valiev, H. J. J. van Dam, J. M. Waldrop, D. B. Williams-Young, C. Yang, M. Zalewski, and T. L. Windus, "From NWChem to NWChemEx: Evolving with the computational chemistry landscape," *Chemical Reviews*, vol. 121, no. 8, pp. 4962–4998, 2021, PMID: 33788546. DOI: 10.1021/acs.chemrev.0c00998. eprint: https://doi.org/10.1021/acs.chemrev.0c00998. [Online]. Available: https://doi.org/10.1021/acs.chemrev.0c00998.

[88] D. B. Shmoys and É. Tardos, "An approximation algorithm for the generalized assignment problem," *Mathematical Programming*, vol. 62, no. 1, pp. 461–474, 1993.

[89] J. K. Lenstra, D. B. Shmoys, and É. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical Programming*, vol. 46, no. 1, pp. 259–271, 1990.

[90] C. Chekuri and S. Khanna, "A polynomial time approximation scheme for the multiple knapsack problem," *SIAM Journal on Computing*, vol. 35, no. 3, pp. 713–728, 2005.

[91] B. Lehmann, D. Lehmann, and N. Nisan, "Combinatorial auctions with decreasing marginal utilities," *Games and Economic Behavior*, vol. 55, no. 2, pp. 270–296, 2006.

[92] J. Vondrák, "Optimal approximation for the submodular welfare problem in the value oracle model," in *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, 2008, pp. 67–74.

[93] X. J. Zhu, "Semi-supervised learning literature survey," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2005.

[94] A. Subramanya and P. P. Talukdar, "Graph-based semi-supervised learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 8, no. 4, pp. 1–125, 2014. DOI: 10.2200/S00590ED1V01Y201408AIM029. eprint: https://doi.org/10.2200/S00590ED1V01Y201408AIM029. [Online]. Available: https://doi.org/10.2200/S00590ED1V01Y201408AIM029.

[95] O. Chapelle, B. Schölkopf, and A. Zien, Eds., *Semi-Supervised Learning*. The MIT Press, 2006, ISBN: 9780262033589. DOI: 10.7551/mitpress/9780262033589.001.0001. [Online]. Available: https://doi.org/10.7551/mitpress/9780262033589.001.0001.

[96] M. Orbach and K. Crammer, "Graph-based transduction with confidence," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2012, pp. 323–338.

[97] A. Subramanya and J. Bilmes, "Semi-supervised learning with measure propagation," *Journal of Machine Learning Research*, vol. 12, no. Nov, pp. 3311–3370, 2011.

[98] A. B. Goldberg and X. Zhu, "Seeing stars when there aren't many stars: Graph-based semi-supervised learning for sentiment categorization," in *Proceedings of the first workshop on graph based methods for natural language processing*, Association for Computational Linguistics, 2006, pp. 45–52.

[99] A. Celikyilmaz, M. Thint, and Z. Huang, "A graph-based semi-supervised learning for question-answering," in *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, 2009, pp. 719–727.

[100] A. Subramanya, S. Petrov, and F. Pereira, "Efficient graph-based semi-supervised learning of structured tagging models," in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 2010, pp. 167–176.

[101] R. Liu, J. Zhou, and M. Liu, "Graph-based semi-supervised learning algorithm for web page classification," in *Sixth International Conference on Intelligent Systems Design and Applications*, IEEE, vol. 2, 2006, pp. 856–860.

[102] P. Talukdar, J. Reisinger, M. Pasca, D. Ravichandran, R. Bhagat, and F. Pereira, "Weakly-supervised acquisition of labeled class instances using graph random walks," in *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, 2008, pp. 582–590.

[103] P. P. Talukdar and F. Pereira, "Experiments in graph-based semi-supervised learning methods for class-instance acquisition," in *Proceedings of the 48th annual meeting of the association for computational linguistics*, Association for Computational Linguistics, 2010, pp. 1473–1481.

[104] B.-B. Liu and Z.-M. Lu, "Image colourisation using graph-based semi-supervised learning," *IET image processing*, vol. 3, no. 3, pp. 115–120, 2009.

[105] Y. Zhao, R. Ball, J. Mosesian, J.-F. de Palma, and B. Lehman, "Graph-based semi-supervised learning for fault detection and classification in solar photovoltaic arrays," *IEEE Transactions on Power Electronics*, vol. 30, no. 5, pp. 2848–2858, 2014.

[106] V. Satuluri, S. Parthasarathy, and Y. Ruan, "Local graph sparsification for scalable clustering," in *ACM SIGMOD*, ACM, 2011, pp. 721–732.

[107] 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*, ser. ICML '09, Montreal, Quebec, Canada: ACM, 2009, pp. 441–448, ISBN: 978-1-60558-516-1. DOI: 10.1145/1553374.1553432. [Online]. Available: http://doi.acm.org/10.1145/1553374.1553432.

[108] 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 Computational Natural Language Learning*, ACL, 2011, pp. 154–162.

[109] L. Berton, A. de Andrade Lopes, and D. A. Vega-Oliveros, "A comparison of graph construction methods for semi-supervised learning," in *2018 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2018, pp. 1–8.

[110] D. A. Vega-Oliveros, L. Berton, A. M. Eberle, A. de Andrade Lopes, and L. Zhao, "Regular graph construction for semi-supervised learning," in *Journal of physics: Conference series*, IOP Publishing, vol. 490, 2014, p. 012 022.

[111] X. Zhu, Z. Ghahramani, and J. D. Lafferty, "Semi-supervised learning using gaussian fields and harmonic functions," in *Proceedings of the 20th International conference on Machine learning (ICML-03)*, 2003, pp. 912–919.

[112] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf, "Learning with local and global consistency," in *Advances in neural information processing systems*, 2004, pp. 321–328.

[113] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly, "Video suggestion and discovery for youtube: Taking random walks through the view graph," in *Proceedings of the 17th international conference on World Wide Web*, 2008, pp. 895–904.

[114] P. P. Talukdar and K. Crammer, "New regularized algorithms for transductive learning," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2009, pp. 442–457.

[115] A. Corduneanu and T. Jaakkola, "On information regularization," in *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*, 2002, pp. 151–158.

[116] N. Srebro and T. Jaakkola, "Weighted low-rank approximations," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ser. ICML'03, Washington, DC, USA: AAAI Press, 2003, pp. 720–727, ISBN: 1-57735-189-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=3041838.3041929.

[117] D. D. Lewis, *UCI machine learning repository*. [Online]. Available: https://archive.ics.uci.edu/ml/machine-learning-databases/reuters21578-mld/.

[118] K. Lang, "Newsweeder: Learning to filter netnews," in *Proceedings of the Twelfth International Conference on Machine Learning*, 1995, pp. 331–339.

[119] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," *Theoretical Computer Science*, vol. 348, no. 2-3, pp. 207–216, 2005.

[120] A. McGregor, "Graph stream algorithms: A survey," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 9–20, 2014.

[121] N. Buchbinder and J. Naor, "The design of competitive online algorithms via a primal-dual approach," *Foundations and Trends in Theoretical Computer Science*, vol. 3, no. 2-3, pp. 93–263, 2007.

[122] A. Paz and G. Schwartzman, "A (2+ $\varepsilon$)-approximation for maximum weight matching in the semi-streaming model," *ACM Transactions on Algorithms (TALG)*, vol. 15, no. 2, pp. 1–15, 2018.

[123] R. Bar-Yehuda and S. Even, "A local-ratio theorem for approximating the weighted vertex cover problem," in *North-Holland Mathematics Studies*, vol. 109, Elsevier, 1985, pp. 27–45.

[124] A. Chakrabarti and S. Kale, "Submodular maximization meets streaming: Matchings, matroids, and more," *Mathematical Programming*, vol. 154, no. 1, pp. 225–247, 2015.

[125] R. Levin and D. Wajc, "Streaming submodular matching meets the primal-dual method," in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SIAM, 2021, pp. 1914–1933.

[126] M. B. Cohen, Y. T. Lee, and Z. Song, "Solving linear programs in the current matrix multiplication time," *Journal of the ACM (JACM)*, vol. 68, no. 1, pp. 1–39, 2021.

[127] D. A. Spielman and S.-H. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 2004, pp. 81–90.

[128] Y. T. Lee, S. Rao, and N. Srivastava, "A new approach to computing maximum flows using electrical flows," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 755–764.

# A. MATH PROGRAMMING FORMULATION OF VARIOUS DCS PROBLEMS

We review the integer and linear programming formulation for matching and $b$-matching problem.

## A.1 Edge-weighted matchings

### A.1.1 1-matching

The Integer Linear Programming (ILP) formulation of maximum weight 1-matching or matching is shown in Formulation A.1.

$$\begin{aligned} \max \ & \sum_{\mathrm{e}\in E} w(\mathrm{e})x(\mathrm{e}), \\ & \sum_{\mathrm{e}\in\delta(\mathrm{i})} x(\mathrm{e}) \leq 1, \quad \forall \mathrm{i} \in V, \\ & x(\mathrm{e}) \in \{0,1\}, \quad \forall \mathrm{e} \in E. \end{aligned} \tag{A.1}$$

Relaxing the integrality constraints to fractional one provides a linear programming relaxation of Problem A.1. But this relaxation does not yield an integral solution, and one can introduce a new set of constraints that would guarantee integrality. Such Linear programming formulation with additional odd set constraints is shown in Formulation A.2.

$$\begin{aligned} \max \ & \sum_{\mathrm{e}\in E} w(\mathrm{e})x(\mathrm{e}), \\ & \sum_{\mathrm{e}\in\delta(\mathrm{i})} x(\mathrm{e}) \leq 1, \quad \forall \mathrm{i} \in V, \\ & \sum_{\mathrm{e}\in\gamma(S)} x(\mathrm{e}) \leq (|S|-1)/2, \quad \forall S \subseteq V, |S| \geq 3, |S| \text{ odd}, \\ & x(\mathrm{e}) \geq 0, \quad \forall \mathrm{e} \in E. \end{aligned} \tag{A.2}$$

For a perfect matching the second set of constraints in Formulation A.1 becomes $\sum_{\mathrm{e}\in\delta(\mathrm{i})} x(\mathrm{e}) = 1$, $\forall \mathrm{i} \in V$. In this case, there is an alternative LP formulation that uses *odd* cut set con-

straints. In a perfect matching, there must be at least one cut edge from any odd subset of vertices of the graph. The alternative LP formulation for perfect matching is shown in Formulation A.3

$$\begin{aligned}
\max \sum_{\mathrm{e}\in E} & w(\mathrm{e})x(\mathrm{e}), \\
& \sum_{\mathrm{e}\in\delta(\mathrm{i})} x(\mathrm{e}) = 1, \quad \forall \mathrm{i} \in V, \\
& \sum_{\mathrm{e}\in\delta(S)} x(\mathrm{e}) \geq 1, \quad \forall S \subseteq V, |S| \geq 3, |S| \text{ odd}, \\
& x(\mathrm{e}) \geq 0, \quad \forall \mathrm{e} \in E.
\end{aligned} \tag{A.3}$$

### A.1.2 $b$-matching

The Integer Linear Programming (ILP) formulation of maximum weight $b$-matching is shown in Formulation A.4.

$$\begin{aligned}
\max \sum_{\mathrm{e}\in E} & w(\mathrm{e})x(\mathrm{e}), \\
& \sum_{\mathrm{e}\in\delta(\mathrm{i})} x(\mathrm{e}) \leq b(\mathrm{i}), \quad \forall \mathrm{i} \in V, \\
& x(\mathrm{e}) \in \{0,1\}, \quad \forall \mathrm{e} \in E.
\end{aligned} \tag{A.4}$$

The Linear programming relaxation of maximum weighted $b$-matching with the odd set constraints is shown in Formulation A.5.

$$
\begin{aligned}
\max \ & \sum_{\mathrm{e}\in E} w(\mathrm{e})x(\mathrm{e}), \\
& \sum_{\mathrm{e}\in\delta(\mathrm{i})} x(\mathrm{e}) \le b(\mathrm{i}), \quad \forall \mathrm{i} \in V, \\
& \sum_{\mathrm{e}\in\gamma(S)} x(\mathrm{e}) \le (b(S)-1)/2, \quad \forall S \subseteq V, |S| \ge 3, b(S) \text{ odd}, \\
& 0 \le x(\mathrm{e}) \le 1, \quad \forall \mathrm{e} \in E.
\end{aligned} \tag{A.5}
$$

Similar to the 1-matching, for a perfect $b$-MATCHING, we have an alternative LP formulation shown in Formulation A.6.

$$
\begin{aligned}
\max \ & \sum_{\mathrm{e}\in E} w(\mathrm{e})x(\mathrm{e}), \\
& \sum_{\mathrm{e}\in\delta(\mathrm{i})} x(\mathrm{e}) = b(\mathrm{i}), \quad \forall \mathrm{i} \in V, \\
& \sum_{\mathrm{e}\in\delta(S)} x(\mathrm{e}) \ge 1, \quad \forall S \subseteq V, |S| \ge 3, b(S) \text{ odd}, \\
& 0 \le x(\mathrm{e}) \le 1, \quad \forall \mathrm{e} \in E.
\end{aligned} \tag{A.6}
$$

## A.2 Vertex-weighted matchings

In a vertex weighted matching, the weight function is defined on vertices. So the graph is $G(V, E, w)$, where $w : V \to \mathbb{R}_+$.

### A.2.1 1-matching

A vertex-weighted matching can be reduced to an edge-weighted matching by assigning an edge weight that equals the sum of the weight of this edge's endpoints. This transformation allows us to apply the formulation developed for edge-weighted matchings. But in this section, we will develop a direct formulation using a new set of variables that holds the

matching information of the vertices. The ILP and a LP formulation of vertex weighted matching is shown in A.7, and A.8 respectively.

$$\begin{aligned}
\max \sum_{v \in V} & w(v)y(v), \\
& \sum_{\mathrm{e} \in \delta(\mathrm{i})} x(\mathrm{e}) \leq y(\mathrm{i}), \quad \forall \mathrm{i} \in V, \\
& x(\mathrm{e}) \in \{0,1\}, \quad \forall \mathrm{e} \in E \\
& y(v) \in \{0,1\}, \quad \forall v \in V.
\end{aligned} \tag{A.7}$$

$$\begin{aligned}
\max \sum_{v \in V} & w(v)y(v), \\
& \sum_{\mathrm{e} \in \delta(\mathrm{i})} x(\mathrm{e}) \leq y(\mathrm{i}), \quad \forall \mathrm{i} \in V, \\
& \sum_{\mathrm{e} \in \gamma(S)} x(\mathrm{e}) \leq (|S|-1)/2, \quad \forall S \subseteq V, |S| \geq 3, |S| \text{ odd}, \\
& y(v) \geq 0, \quad \forall v \in V.
\end{aligned} \tag{A.8}$$

### A.2.2 $b$-matching

We show the IP and LP formulation for vertex weighted $b$-Matching formulation in A.9 and A.10 respectively.

$$\begin{aligned}
\max \sum_{v \in V} & w(v)y(v), \\
& \sum_{\mathrm{e} \in \delta(\mathrm{i})} x(\mathrm{e}) \leq b(\mathrm{i})y(\mathrm{i}), \quad \forall \mathrm{i} \in V, \\
& x(\mathrm{e}) \in \{0,1\}, \quad \forall \mathrm{e} \in E \\
& y(v) \in \{0,1\}, \quad \forall v \in V.
\end{aligned} \tag{A.9}$$

$$
\begin{aligned}
\max \ & \sum_{\mathrm{e}\in E} w(\mathrm{e})x(\mathrm{e}), \\
& \sum_{\mathrm{e}\in\delta(\mathrm{i})} x(\mathrm{e}) \le b(\mathrm{i})y(\mathrm{i}), \quad \forall \mathrm{i} \in V, \\
& \sum_{\mathrm{e}\in\gamma(S)} x(\mathrm{e}) \le (b(S)-1)/2, \quad \forall S \subseteq V, |S| \ge 3, b(S) \text{ odd}, \\
& 0 \le x(\mathrm{e}) \le 1, \quad \forall \mathrm{e} \in E, \\
& 0 \le y(v) \le 1, \quad \forall v \in V.
\end{aligned} \tag{A.10}
$$

# B. REDUCTION FROM $b$-MATCHING TO 1-MATCHING AND $b$-EDGE COVER TO 1-EDGE COVER

In this section, we will describe a polynomial time reduction that transforms a maximum weighted $b$-MATCHING to a maximum weighted 1-matching problem, and a minimum weighted $b$-EDGE COVER to a minimum weighted 1-edge cover problem.

## B.1 The new graph construction

Given a graph $G(V, E, w)$ and $b(.)$ values defined on each vertex, we create a new graph $G'(V', E', w')$ as follows. For each vertex $v \in V$, we create $b(v)$ new copies of $v$ in $V'$. Denote the copy vertex set of $v$ by $\phi(v)$. Also for each edge $\mathrm{e}(u, v) \in E$, we insert two new vertices in $V'$. We call these new vertices $p_{\mathrm{e},u}$ and $p_{\mathrm{e},v}$. Formally,

$$V' = \{\cup_{v \in V} \phi(v)\} \cup \{\cup_{\mathrm{e}(u,v) \in E} (p_{\mathrm{e},u} \cup p_{\mathrm{e},v})\}.$$

For each vertex $v \in V$, We set $b(v') = b(v) \quad \forall v' \in \phi(v)$, and $b(p_{\mathrm{e},u}) = b(p_{\mathrm{e},v}) = 1 \quad \forall \mathrm{e} \in E$. Note that $|V'| = \sum_v b(v) + |E| = O(m)$.

Now let us describe the edge set $E'$. For each edge, $\mathrm{e} \in E$, we insert an edge $(p_{\mathrm{e},u}, p_{\mathrm{e},v})$ in $E'$. We call this edge as the middle edge of e. Again for each edge, $\mathrm{e}(u, v) \in E$, we create $b(u)$ and $b(v)$ more edges in $E'$. These connect each of the copy vertices in $\phi(u)$ with $p_{\mathrm{e},u}$, and copy vertices in $\phi(v)$ with with $p_{\mathrm{e},v}$. These edges are called outer edges of e. Formally,

$$E' = \{\cup(p_{\mathrm{e},u}, p_{\mathrm{e},v}) : \mathrm{e}(u, v) \in E\} \cup \{\cup(u', p_{\mathrm{e},u}) : u' \in \phi(u), \mathrm{e}(u, v) \in E\}\}$$

We note that $|E'| = |E| + \sum_{v \in V} d(v)b(v) = m + O(\beta m) = O(\beta m)$.

So for each edge $\mathrm{e}(u, v)$ in $G$, we have $b(u) + b(v) + 1$ corresponding edges in $G'$. The weight of all these edges are set to $w(\mathrm{e})$.

### B.2 Computing $b$-Matching

Let $M'_*$ be the maximum weighted matching in $G'$. We show how to recover a $b$-Matching $M^*_b$ from $M'_*$. During the $b$-Matching construction we will also build a set $S$, that holds the edges those are in $M'_*$ but can not be in $M^*_b$. Both $M^*_b$ and $S$ are initialized to the empty set. Since $M'_*$ is a maximal matching, we have the following cases for each edge $\mathrm{e}(u, v)$.

1. Two outer edges of e are in $M'_*$. Set $M^*_b = M^*_b \cup \mathrm{e}$.
2. Only one of the outer edges is in $M'_*$. In this case, we can replace this outer edge with the middle edge $(p_{\mathrm{e},u}, p_{\mathrm{e},v})$. This does not change the weight of the matching. We set $S = S \cup (p_{\mathrm{e},u}, p_{\mathrm{e},v})$.
3. The middle edge $(p_{\mathrm{e},u}, p_{\mathrm{e},v})$ is in $M'_*$. We set $S = S \cup (p_{\mathrm{e},u}, p_{\mathrm{e},v})$.

### B.3 Computing $b$-Edge Cover

We can repeat the graph construction for $b$-Edge Cover too. Let $C'_*$ be the minimum weighted edge cover of $G'$. For each edge $\mathrm{e} \in E$, we have the following cases.

1. Two outer edges of e are in $C'_*$. In this case $C^*_b = C^*_b \cup \mathrm{e}$.
2. An outer and a middle edge are in $C'_*$. In this case, we can replace the middle edge with another outer edge and construct a feasible cover with same weight. We set $C^*_b = C^*_b \cup \mathrm{e}$.
3. The middle edge $(p_{\mathrm{e},u}, p_{\mathrm{e},v})$ is in $C'_*$. We set $S = S \cup (p_{\mathrm{e},u}, p_{\mathrm{e},v})$.

### B.4 Analysis

We define for a set $T$ of edges in $G$ or $G'$, $W(T) = \sum_{\mathrm{e}\in T} w(\mathrm{e})$. Also let $\sum_{\mathrm{e}\in E} w(\mathrm{e}) = \mathbf{W}$.

Note that the construction of $b$-Matching or $b$-Edge Cover as discussed above does not depend on the optimality of the matching or the edge cover in $G'$. We can repeat the same construction given any maximal matching or minimal edge cover.

Likewise, we can do a reverse construction i.e., we start with a $b$-Matching or $b$-Edge Cover in $G$ and compute a maximal matching or minimal edge cover in $G'$. Next we will discuss this reverse construction.

### B.4.1 Constructing matching in $G'$

Given any $b$-MATCHING $M_b$, we can construct a maximal matching in $G'$ as follows. We start with a empty set $M'$.

- For a matching edge $e(u, v) \in M_b$, we insert the two of the unmatched outer edge in $M'$. Since $M_b$ is a valid $b$-matching i.e., it has at most $b(u)$ and $b(v)$ edges incident to $u$ and $v$, we must find two of the outer edges available to match in $G'$.
- We make $M'$ maximal by selecting necessary middle edges. These edges are inserted into the set $S$.

We can repeat the same reverse construction for any $b$-EDGE COVER $C_b$ too and create a minimal edge cover $C'$.

We define a pair $(M_b, M')$, where either $M_b$ and $M'$ is connected by either the usual or the reverse construction. Similarly $(C_b, C')$ is defined for $b$-EDGE COVER .

**Proposition B.4.1.** $W'(S) = \mathbf{W} - W(M_b)$

*Proof.* For each edge $e \in E$, either e in $M_b$ or e is in $S$. So,

$$W(M_b) + W'(S) = W(M_b) + W(S) = \mathbf{W}$$

■

**Proposition B.4.2.** $W'(S) = \mathbf{W} - W(C_b)$

*Proof.* The proof follows the same argument as in the proof of Proposition B.4.1. ■

**Proposition B.4.3.** *Given any pair* $(M_b, M')$ *we have*

$$W(M_b) = W'(M') - \mathbf{W}.$$

*Proof.* By construction,

$$\begin{aligned} W(M_b) =& \frac{1}{2}(W'(M') - W(S)) \\ =& \frac{1}{2}(W'(M') - \mathbf{W} + W(M_b)) \\ W(M_b) =& W'(M') - \mathbf{W}. \end{aligned}$$

■

The second line comes from Proposition B.4.1. Similarly we can show the following result for $b$-Edge Cover.

**Proposition B.4.4.** *Given any pair $(C_b, C')$ we have*

$$W(C_b) = W'(C') - \mathbf{W}.$$

**Lemma B.4.1.** *$M_b^*$ is a maximum weighted $b$-Matching*

*Proof.* Replace Proposition B.4.3 using $M_b^*$ and $M'_*$

$$W(M_b^*) = W'(M'_*) - \mathbf{W}.$$

Now take any $b$-Matching, $M_b$ of $G$ and perform the reverse construction. Let $M'$ be the maximal matching in $G'$ from the reverse construction. We have

$$\begin{aligned} W(M_b^*) =& W'(M'_*) - \mathbf{W} \\ \geq& W'(M') - \mathbf{W}. \end{aligned}$$

The last line is due to the fact the $M'_*$ is a maximum weighted matching in $G'$. Again using Proposition B.4.3,

$$\begin{aligned} W(M_b^*) =& W'(M'_*) - \mathbf{W} \\ \geq& W'(M') - \mathbf{W} \\ =& W(M_b). \end{aligned}$$

■

Similarly, we can show the following result for $b$-Edge Cover.

**Lemma B.4.2.** *$C_b^*$ is a minimum weighted $b$-Edge Cover.*

*Proof.* Using Proposition B.4.4 using $C_b^*$ and $C'_*$,

$$W(C_b^*) = W'(C'_*) - \mathbf{W}.$$

Now take any $b$-Edge Cover, $C_b$ of $G$ and perform the reverse construction. Let $C'$ be the minimal edge cover. We have

$$\begin{aligned} W(C_b^*) =& W'(C'_*) - \mathbf{W} \\ \leq& W'(C') - \mathbf{W}. \end{aligned}$$

The last line is due to the fact the $C'_*$ is a minimum weighted edge cover. Using Proposition B.4.4,

$$\begin{aligned} W(C_b^*) =& W'(C'_*) - \mathbf{W} \\ \leq& W'(M') - \mathbf{W} \\ =& W(C_b). \end{aligned}$$

■

# VITA

## Personal Information


*Place and Date of Birth:* Dhaka, Bangladesh | 25 December 1988

*Address:* 224 Arnold Drive, Apt 12,
West Lafayette, In-47906, USA

*Phone:* +1(765)-409-8632

*Email:* sferdou@purdue.edu; ferdous.csebuet@gmail.com

: https://smferdous1.github.io | : http://bit.ly/linkedIn-smf


## Professional Appointments

*2016 – 2021* **Graduate Research Assistant**
Department of Computer Science
School of Science
Purdue University, USA.

*2015 – 2016* **Ross Fellow**
Purdue Graduate School
Purdue University
IN,USA.

*Jun – Aug 2021* **PhD Intern**
Data Science and Machine Intelligence Group
Pacific Northwest National Lab
WA, USA.

*Jun – Aug 2021* **PhD Intern**

ENSA Group

Nokia Bell Labs

NJ, USA.

*May – Aug 2017* **PhD Intern**

Data Science and Machine Intelligence Group

Pacific Northwest National Lab

WA, USA.

*Mar – Jul 2015* **Assistant Professor**

Department of Computer Science and Engineering

Ahsanullah Univ. of Science and Technology

Dhaka, Bangladesh.

*2011 – 2015* **Lecturer**

Department of Computer Science and Engineering

Ahsanullah Univ. of Science and Technology

Dhaka, Bangladesh.

# Education

*2015 – 2021* **PhD in Computer Science**, Purdue University, West Lafayette, Indiana

*Thesis*: "Algorithms for degree-constrained subgraphs and applications"

*Advisor*: Prof. Dr. Alex Pothen | *GPA*: 3.93/4.00.

*2011 – 2014* **MSc Engg. in Computer Science and Engineering**
Bangladesh University of Engineering and Technology (BUET)
*Thesis:* "Practically Efficient Algorithms for Minimum String Cover and Minimum Common String Partition"
*Advisor:* Prof. Dr. M. Sohel Rahman | *GPA*: 3.33/4.00.

*2006 – 2011* **BSc Engg. in Computer Science and Engineering**, BUET
9/138, *Degree with Honours* | *GPA*: 3.89/4.00.

## Fellowships

*2020 – 2021* **John R. Rice Fellowship** for Scientific Computing, Department of Computer Science, Purdue University.

*2015 – 2016* **Ross Fellowship** for incoming graduate student.

*2006 – 2011* **Dean's list** and **merit scholarship** in each of the four academic years in undergrad for excellent academic results.

## Awards & Honors

*2018* **Travel grant** for attending SIAM Combinatorial Scientific Computing Workshop in Bergen, Norway.

*2017* **Third best prize** on SIAM Computationasl Science and Enginneging student poster competition at Purdue University.

*2016* **Travel and accommodation grant** for attending Week long SAMSI summer school on optimization at Research Triangle Park, NC.

*2008* **Tenth among 50 teams** in ACM Inter Collegiate Programming Contest Regional Dhaka Site.

## Courses and Projects during PhD

**Selected courses**

• Statistical Machine Learning • Algorithm Design, Analysis and Implementation • Computational Methods in Optimization • Mathematical Toolkit for Computer Science • Data Communication and Computer Networks • Parallel Computing • Quantum Computation and Information • Reinforcement Learning • Approximation Algorithm in Action.

**Selected projects**

*Spr. 2018* **Implementing Grover's search**
Grover's search is one of the most influential quantum algorithms. In Spring 2018, I completed a Quantum Computation course offered by *Prof. Sabre Kais*. As a class project, I implemented Grover's search in IBM QISKIT. I tested my implementation using 6 Qubits in IBM Quantum simulator.

*Spr. 2017* **On bounding the weight of $b$-matching problem**
In this project, I investigate Lagrangian-relaxation based upper bounds for the maximum weight $b$-matching problem. The problem is formulated as an integer program, and then the relaxed dual problem is solved using subgradient methods to compute the upper bound. Since the method may not find a feasible $b$-matching, a simple heuristic is presented to find feasible solutions from the dual optimal variables. Preliminary experiments show that the method generates bounds that are close to bounds obtained from a linear programming based relaxation, but could be faster than the latter by an order of magnitude.

*Fall 2015* **Modeling Air Travel Demand between two cities**
The goal of this team project was to model the air travel demand between any two cities, based on the socio-technical factors, using machine learning techniques. The demand was treated as a categorical value. We picked 30 major US airports and collected demand data between two airports for the last 10 years. we considered publicly available such as population of the cities, average income of the cities, distance between two airports, airport category and so on. Using SVM as learning algorithm, we were able to acheive 72% test accuracy.

# Others

## Certifications

*Jun 2012* Algorithms: Design and Analysis, Part 1, Stanford University, Coursera (earned 87.8%)

*Aug 2012* Machine Learning, Stanford University, Coursera (earned 97.3%)

*Dec 2012* Algorithms: Design and Analysis, Part 2, Stanford University, Coursera (earned 82.5%)

*Mar 2016* Approximation Algorithms Part I, Ecole normale superieure, Coursera (earned 96%)

### Review experiences

Served as a reviewer in journal **PLoS ONE** and **ACM Transactions on Parallel Computing**.

### Extra-curricular Activities

*2018 – 2019* Served as **General Secretary**, Bangladesh Students association, Purdue University.

*2017 – 2018* Served as **Web Master**, Bangladesh Students association, Purdue University.