Acta Numerica (2019), pp. 541–633
doi:10.1017/S0962492919000035

Approximation algorithms in combinatorial scientific computing

Alex Pothen*
Department of Computer Science, Purdue University,
West Lafayette, IN 47907, USA
E-mail: apothen@purdue.edu

S. M. Ferdous†
Department of Computer Science, Purdue University,
West Lafayette, IN 47907, USA
E-mail: sferdou@purdue.edu

Fredrik Manne
Department of Informatics, University of Bergen,
N-5020 Bergen, Norway
E-mail: fredrikm@ii.uib.no

We survey recent work on approximation algorithms for computing degree-constrained subgraphs in graphs and their applications in combinatorial scientific computing. The problems we consider include maximization versions of cardinality matching, edge-weighted matching, vertex-weighted matching and edge-weighted b-matching, and minimization versions of weighted edge cover and b-edge cover. Exact algorithms for these problems are impractical for massive graphs with several millions of edges. For each problem we discuss theoretical foundations, the design of several linear or near-linear time approximation algorithms, their implementations on serial and parallel computers, and applications. Our focus is on practical algorithms that yield good performance on modern computer architectures with multiple threads and interconnected processors. We also include information about the software available for these problems.

* The work of the first two authors was supported in part by US NSF grant CCF-1637534; the US Department of Energy through grant DE-FG02-13ER26135; and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the DOE Office of Science and the NNSA.

CONTENTS

1. Introduction

We discuss recent progress in the design of approximation algorithms for two problems on graphs, with their applications to combinatorial scientific computing (CSC). The problems involve the computation of degree-constrained subgraphs of a graph that might represent its significant subgraphs. Computing these subgraphs reduce the computational costs and memory required of algorithms that obtain information from the graph, such as semi-supervised classification in machine learning, or the solution of sparse systems of linear equations. These subgraphs also help remove noise from the data so that machine learning algorithms perform better in classification tasks.

The first problem we consider is the classical one of computing a matching in a graph. A matching is a subset of vertex-disjoint edges; hence there is *at most* one edge of the subset incident on each vertex in the graph. Here we could seek to maximize the cardinality of a matching, or when weights are assigned to edges, maximize the sum of the weights of edges in a matching. We will also discuss a less studied variant where the weights are on the vertices instead of the edges. A generalization of matching is the b-matching problem, where we are given natural numbers $b(v)$ for each vertex v in the graph, and are required to choose at most $b(v)$ matched edges incident on v. When weights are assigned to the edges, we seek to maximize the sum of weights of the matched edges.

The second problem we consider is edge cover, where we are required to choose *at least* one edge incident on each vertex to belong to the edge cover. Here we seek to minimize the cardinality of the edges in the cover, or the sum of weights of the edges in the cover. The generalization of an edge cover leads to the b-edge cover problem, where given natural numbers $b(v)$ for each vertex v, we are required to choose at least $b(v)$ edges incident on v

to belong to the edge cover. Again, we seek to minimize the sum of weights of the edges in the cover. Our work on this problem was motivated by an application to a data privacy problem called adaptive anonymity.

Both of these problems and their variants have polynomial time algorithms to solve them; however, the asymptotic run time is larger than the product of the number of edges times the square root of the number of vertices, and this is too high to be practical for graphs with millions or billions of vertices and edges. Furthermore, exact algorithms for these problems have little concurrency. Hence we turn to the design of approximation algorithms that have near-linear time complexity in the size of the graph. We also design approximation algorithms that possess high concurrency, so that they can be implemented efficiently on parallel computers.

An *exact* algorithm for an optimization problem computes the optimum value of its objective function. An *approximation algorithm* for an optimization problem computes a value that is within some factor α (a constant or a function of the problem size) of the optimal value for all problem instances. For a maximization problem (as in matching), the ratio of the value computed by the approximation algorithm to the maximum value is at least $\alpha < 1$ for all instances; for a minimization problem (as in edge cover), the ratio of the value computed by the approximation algorithm to the optimal value is at most $\alpha > 1$, again for all instances. We say that this is an α-approximation algorithm for the problem, and that the approximation ratio of the algorithm is α. Note that this worst-case approximation ratio is obtained analytically by an *a priori* argument, and the approximation ratio for a specific instance might be much better than α. For many optimization problems, known exact algorithms might not have polynomial time complexity. Also, for many problems, for any $\epsilon > 0$ if an algorithm with approximation ratio $n^{(1-\epsilon)}$ exists then $P = NP$, which suggests that a polynomial time approximation algorithm might not exist. An algorithm for an optimization problem for which we cannot obtain an approximation ratio is called a *heuristic algorithm*. This is the situation for many problems in CSC, and we can evaluate an algorithm by empirically comparing the value of the objective function it computes with other algorithms on a collection of test problems.

There are several advantages that approximation algorithms have over exact algorithms, which we enumerate as follows.

- Approximation algorithms have lower run time complexity relative to exact algorithms, often linear or nearly linear in the size of the problem. Exact algorithms with polynomial run times can be too slow for massive graphs, but the faster approximation algorithms may be practical. Furthermore, in practice these approximation algorithms compute solutions that are nearly optimal.

- Approximation algorithms are conceptually simpler than exact algorithms, and their proofs of correctness could also be simpler.
- Approximation algorithms are easier to implement when compared to the more sophisticated exact algorithms, which is practically an important reason for their widespread use.
- Approximation algorithms can be designed to have more concurrency than exact algorithms. The simplicity of implementation of approximation algorithms is even more important for algorithms to be implemented on parallel computers. This is a major motivating factor for our work on matchings and edge covers.
- Often a matching or edge cover algorithm is used at each step of an approximation algorithm or a heuristic to solve another problem. In this case, exact matchings are not required, and its use increases the run time of the algorithm, limiting the size of the problems that could be solved. This is the case for network alignment (Khan, Gleich, Pothen and Halappanavar 2012), for adaptive anonymity (Khan *et al.* 2018*a*), k-nearest neighbour graph construction (Ferdous, Pothen and Khan 2018), ontology alignment (Kolyvakis, Kalousis, Smith and Kiritsis 2018), *etc.*

Approximation algorithms have been studied in the discrete mathematics, theoretical computer science and operations research communities. Books discussing approximation algorithms include Hochbaum (1997), Vazirani (2003), Williamson and Shmoys (2011) and Du, Ko and Hu (2012). An earlier survey on approximation algorithms for the matching problem was provided by Hougardy (2009). Our discussion of the matching problem is fairly disjoint from this survey; we have included more recent algorithms such as the Suitor and b-Suitor algorithms, and problems such as b-matching and the vertex-weighted matching problem, as well as parallel algorithms. We are not aware of an earlier survey of the edge cover and b-edge cover problems. Goemans and Williamson (1997) survey the primal–dual method for designing approximation algorithms and apply it to several network design problems, including vertex cover and edge cover.

We conclude this section with a brief discussion of the research area and community of combinatorial scientific computing (CSC). Research in CSC focuses on the design, theoretical analysis, computational evaluation and deployment of combinatorial algorithms to solve problems in computational science and engineering. The CSC community has its roots in the research areas of sparse matrix computations, algorithmic differentiation and parallel computing. This community was formally organized in the early 2000s, and biennial workshops on CSC have been held under the auspices of the Society for Industrial and Applied Mathematics

(SIAM) since 2004. Information about these meetings and proceedings published by SIAM is available at www.csc-research.org. Snapshots of recent research in CSC are included in the collection of articles edited by Naumann and Schenk (2012); the first chapter by Hendrickson and Pothen provides more detail on the historical development and emerging research areas of CSC. More recently the CSC community has joined with other computational discrete mathematicians to organize a SIAM Activity Group on Applied and Computational Discrete Algorithms. Additional information is available at www.siam.org/membership/Activity-Groups/detail/applied-and-computational-discrete-algorithms.

2. Maximum cardinality matching

2.1. Elementary definitions and concepts

We begin with notation and concepts needed for discussing matching and edge cover problems. We consider a simple, loopless, undirected graph $G = (V, E)$, where V is the set of vertices, E is the set of edges, and $|V| \equiv n$ and $|E| \equiv m$. The neighbours of a vertex u will be denoted by $\mathrm{adj}(u)$ or $N(u)$; the vertex u itself is not a member of the adjacency set. The cardinality of the adjacency set is the degree of the vertex, denoted $\deg(u)$. The maximum degree of a vertex in the graph will be denoted Δ.

An edge $e = (u, v)$ has the vertices u and v as its endpoints. We say that e is an edge incident on u (and v), and that u and v are adjacent vertices. Two edges e and f are adjacent or are neighbours if they share a common endpoint. The set of edges adjacent to $e = (u, v)$ consists of other edges in G with u or v as an endpoint.

A *path* in a graph is sequence of edges $\{(v_1, v_2), (v_2, v_3), \ldots (v_k, v_{k+1})\}$, where the vertices are distinct and consecutive edges share an endpoint. The *length* of the path is the number of edges k. A *cycle* is the concatenation of a path and an edge joining its first and last vertices, (v_1, v_{k+1}).

A matching in the graph G is a subset of vertex-disjoint edges M. Thus at most one edge in M is incident on each vertex in V. We say that an edge in M is a matched edge, and the endpoints of a matched edge are matched vertices. If an edge belongs to $E \setminus M$ then it is unmatched, and similarly for unmatched vertices.

Let M be a matching. Then a path or cycle P is *alternating* if it consists of edges drawn alternately from M and $E \setminus M$. An alternating path P is an *M-augmenting path* if it begins and ends with an unmatched edge. An augmenting path has one more unmatched edge than matched edges. By exchanging matched edges with unmatched edges along the augmenting path, *i.e.* by computing $M \oplus P$, we obtain a matching M' with one more matched edge than M. (Here $A \oplus B = (A \setminus B) \cup (B \setminus A)$.) In some situations, we consider two matchings M_1 and M_2, and consider the symmetric

difference $M_1 \oplus M_2$. The symmetric difference consists of isolated vertices (an edge belonging to both matchings), and alternating paths and cycles; here the edges belong alternately to M_1 and M_2, and thus vertices on such paths have degree 1 or 2 in the subgraph induced by the two matchings. Such paths could be used to augment the cardinality of the matching M_2 if M_1 has more edges on the path. Alternating cycles have the same number of edges from M_1 and M_2, and cannot augment the cardinality of either matching. Augmenting paths and cycles with respect to weights on edges will be discussed in Section 3.2.

A maximum cardinality matching in a graph could be obtained by an algorithm that begins with the empty matching and at each step finds an augmenting path from an unmatched vertex. Hopcroft and Karp (1973) obtained a $O(\sqrt{n}m)$-time algorithm for finding maximum cardinality matchings in bipartite graphs, and this was extended to an algorithm for finding maximum matchings in non-bipartite graphs by Micali and Vazirani (1980).

Matching algorithms are discussed in many books on graphs and graph algorithms as well as specialized books on matchings, for example Burkard, Dell'Amico and Martello (2009), Lovász and Plummer (2009) and Schrijver (2003).

2.2. Augmenting path-based approximation

We begin by proving a lemma obtained by Hopcroft and Karp (1973).

Lemma 2.1. If all augmenting paths with respect to a matching M in a graph G have length greater than or equal to $2k-1$, then the matching is a $(k-1)/k$-approximation of a maximum cardinality matching M^*.

Proof. We consider the symmetric difference of a maximum cardinality matching M^* and the given matching M. It consists of vertices of degree zero, one or two, since at most one edge in M^* and one edge in M can be incident on any given vertex. Isolated vertices and alternating cycles have the same number of edges from M and M^*, and the ratio $|M|/|M^*|$ is one. There are $|M^*| - |M|$ vertex-disjoint M-augmenting paths in the symmetric difference that account for the differences in cardinalities of the two matchings. Each augmenting path P has at least $k-1$ edges from M and k edges from M^*, and hence for the path the ratio $|M \cap P|/M^* \cap P| \geq (k-1)/k$. Since the inequality is true for every augmenting path, the approximation ratio is $|M|/|M^*| \geq (k-1)/k$. □

An important special case of the above lemma is when the augmenting path has length at least three, that is, we find a maximal matching in G. Then we have a 1/2-approximation to the maximum cardinality matching. This observation is used in many practical matching codes as an initializ-

ation step before commencing searches for longer augmenting paths. By increasing the lengths of the augmenting paths, one obtains an approximation ratio as close to one as possible; however, with higher augmenting path lengths, the algorithm more closely resembles the exact algorithm due to Micali and Vazirani (1980), which requires $O(\sqrt{n})$ phases, where each phase constructs a maximal, vertex-disjoint set of shortest augmenting paths. Bast, Mehlhorn, Schäfer and Tamaki (2006) have shown that on an Erdős–Rényi random graph on n vertices with the probability of an edge equal to $33/n$, $G(n, 33/n)$, the Micali–Vazirani algorithm requires at most $O(\log n)$ phases. A similar result was obtained earlier by Motwani (1994). Hougardy (2009) showed that an approximation ratio of $4/5$ is achievable in $O(m)$ time for the maximum cardinality matching problem by finding augmenting paths of length less than or equal to seven.

2.3. Randomized approximation algorithms

We now consider two randomized algorithms that compute approximations better than $1/2$ by first scaling the adjacency matrix of the bipartite graph to a doubly stochastic matrix, when the graph has the property that every edge belongs to some maximum cardinality matching. Such graphs are said to have *total support*. (Another way of stating this condition is that the Dulmage–Mendelsohn decomposition of the bipartite graph consists of subgraphs that have (1) a perfect matching, or (2) a row-perfect matching, or (3) a column-perfect matching, and no edge joins two distinct components to each other. The Dulmage–Mendelsohn decomposition is discussed briefly in Section 3.4.1 and in more detail in Pothen and Fan (1990).) One of the advantages of these algorithms is that they are highly concurrent, and hence can be easily implemented on parallel architectures. These algorithms were designed and implemented by Dufossé, Kaya and Uçar (2015), and we follow their discussion.

We consider a bipartite graph $G = (V_1, V_2, E)$ with the same number of vertices in the two sets V_1 and V_2, so that its adjacency matrix is a square matrix with elements belonging to $\{0, 1\}$. Every edge joins some vertex $i \in V_1$ with some vertex in $j \in V_2$, and the element (i, j) in the adjacency matrix is then 1; it would be 0 if there is no such edge. The scaling algorithm was designed by Sinkhorn and Knopp (1967), and it alternates in scaling the matrix with a diagonal matrix of the reciprocal of the column sums and another diagonal matrix with the reciprocal of the row sums. When this process is iterated on a bipartite graph that has total support, the scaled adjacency matrix converges to a doubly stochastic matrix, *i.e.* a matrix whose column sums and row sums are equal to one. The values of the matrix elements are used in a randomized algorithm to determine the probability of matching an edge.

Algorithm 1 Sinkhorn–Knopp (A, ϵ)

Input: An $n \times n$ adjacency matrix A with total support, and an error threshold ϵ.

Output: Row scaling array d_r and a column scaling array d_c.

Initialize $d_r(i) = 1$, $d_c(i) = 1$, for $i = 1, \ldots, n$.
Initialize $\text{csum}(j) = \sum_i A_{i,j}$, for $j = 1, \ldots, n$.
while $\max_j |1 - \text{csum}(j)| > \epsilon$ **do**
    $d_c(j) = \text{csum}(j)$, for $j = 1, \ldots, n$
    for $i = 1$ **to** n **do**
        $\text{rsum}(i) = \sum_j A_{i,j} \times d_c(j)$
        $d_r(i) = 1/\text{rsum}(i)$
    end for
    for $j = 1$ **to** n **do**
        $\text{csum}(j) = \sum_i A_{i,j} \times d_r(i)$
        $d_c(j) = 1/\text{csum}(j)$
    end for
end while
return d_r, d_c

Algorithm 1 describes the procedure to convert an adjacency matrix with total support to a doubly stochastic matrix. Here ϵ is an upper bound on the permissible distance from a column sum of one. Other scaling algorithms could also be used for this purpose, but the Sinkhorn–Knopp algorithm is more concurrent. Also Dufossé, Kaya and Uçar (2015) report that five to ten iterations of this scaling algorithm suffice to compute approximate matchings.

Algorithm 2 describes how a random matching is computed from the doubly stochastic matrix S derived from the bipartite graph. The variable $\text{Diag}(d_r)$ denotes a diagonal matrix obtained with the elements of the vector d_r on the diagonal. The algorithm uses the matrix element s_{ij} to determine the probability with which a column j is matched to a row i. In this algorithm a row could attempt to match to a column that is already matched to another row; in this case, one of the rows, say the last, succeeds, and the other row gets unmatched.

Dufossé *et al.* (2015) proved that this random matching will match $n(1 - 1/e)$ vertices with high probability, where e is the base of natural logarithm, the Euler number.

Theorem 2.2. Let A be an $n \times n$ adjacency matrix corresponding to a bipartite graph $G = (V_1, V_2, E)$ with total support. Then the random matching obtained by Algorithm 2 has expected cardinality $n(1 - 1/e)$ as $n \to \infty$.

Algorithm 2 $(1-1/e)$-Approximate Maximum Cardinality Matching (A, ϵ)

Input: An $n \times n$ matrix A with total support.
Output: An array cmatch(.) of the rows matched to columns.

```
(d_r, d_c) = Sinkhorn–Knopp(A, ε)
S = Diag(d_r) A Diag(d_c)
for i = 1 to n do
    Pick a random column j ∈ adj(i) with probability s_ij;
    cmatch(j) = i
end for
return cmatch
```

Proof. The probability that a column j is not matched to any of the rows in $\mathrm{adj}(j)$ is $\prod_{i\in\mathrm{adj}(j)}(1-s_{ij})$. Let d_j denote the degree of column j, *i.e.* $|\mathrm{adj}(j)|$. From the inequality that the geometric mean is less than or equal to the arithmetic mean, we have

$$\left(\prod_{i\in\mathrm{adj}(j)}(1-s_{ij})\right)^{1/d_j} \leq \frac{d_j - \sum_{i\in\mathrm{adj}(j)} s_{ij}}{d_j}.$$

Since S is doubly stochastic, the sum on the right-hand side of the inequality is one; hence after taking the d_jth power, the right-hand side simplifies to

$$(1-1/d_j)^{d_j} \leq (1/e).$$

Thus the probability that column j is matched is at least $1-1/e$, and the expected size of the matching is at least $n(1-1/e)$. □

The algorithm we have described is a $((1-1/e)\approx 0.632)$-approximation algorithm for maximum cardinality matching on bipartite graphs with total support. We mention that an online algorithm for maximum cardinality matching has the same approximation ratio. This latter algorithm was originally designed by Karp, Vazirani and Vazirani (1990), and a simpler proof of its correctness and approximation ratio was provided by Birnbaum and Mathieu (2008).

A better approximation ratio can be obtained from a variant 'two-sided' matching algorithm. In order to describe this algorithm, we need to discuss an algorithm due to Karp and Sipser (1981) for computing a random matching in a graph. The Karp–Sipser algorithm matches a vertex of degree one to its only neighbour, since it must be matched in any maximum cardinality matching. Then it deletes the endpoints of the matched edge, and the edges incident on them from the graph. This may create new vertices of degree one. The algorithm repeatedly matches vertices of degree one, until none is left. At this stage, if all vertices have been matched, then the algorithm

terminates. If not, it chooses a random vertex in the graph and matches it to one of its neighbours, and then deletes the endpoints and the edges incident on them. The algorithm iterates until all vertices are matched.

In the two-sided matching algorithm, vertices in both the vertex sets V_1 and V_2 choose at random a single neighbour, with the probability given by the matrix elements in the doubly stochastic matrix S. The subgraph induced by the chosen edges has at most $2n$ edges. It could be fewer than $2n$, if two vertices belonging to different vertex sets choose each other. Each connected component of this induced graph can have at most the same number of edges as the number of vertices, and hence it is a tree or a graph with one cycle. In this graph, we run the Karp–Sipser algorithm to compute a matching. Since each connected component is either a tree or a unicyclic graph, the Karp–Sipser algorithm computes a maximum cardinality matching in the graph. When the initial graph has total support, Dufossé, Kaya and Uçar (2015) have proved that the two-sided algorithm leads to a 0.866-approximation for maximum cardinality matching as $n \to \infty$ with high probability.

3. Edge-weighted matching

In this section we present approximation algorithms for the weighted matching problem on an undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{R}_{\geq 0}$. The objective is to find a matching M such that the sum of the weights of the edges in M is as large as possible. We denote such a maximum weight matching by M^*. The fastest exact algorithm for the weighted matching problem has run time $O(mn + n^2 \log n)$ (Gabow 2018), and the run time of computing an optimal solution can get prohibitively large even for moderate sized graphs, it is of interest to investigate fast approximation algorithms. As we will explain, some of these algorithms also lend themselves well to parallel execution.

Definitions

Let M be a matching. An alternating path or cycle P is an *augmentation* if $M \oplus P$ is also a matching. The weight of a set of edges S is $w(S) = \sum_{e \in S} w(e)$. The *gain* of an alternating path or cycle P is $g(P) = w(P \setminus M) - w(P \cap M)$.

For $k \geq 1$, a *k-augmentation* is an augmentation containing *at most* k edges not in M. Figure 3.1(a) shows all possible 1-augmentations. In the figure a solid line denotes an edge in the current matching and a dotted line an edge not in the matching. By including the paths in Figure 3.1(b) we get all possible 2-augmentations that are paths, while Figure 3.1(c) shows the only cycle possible in a 2-augmentation. Together these graphs show all possible 2-augmentations. In terms of cardinality a k-augmentation that

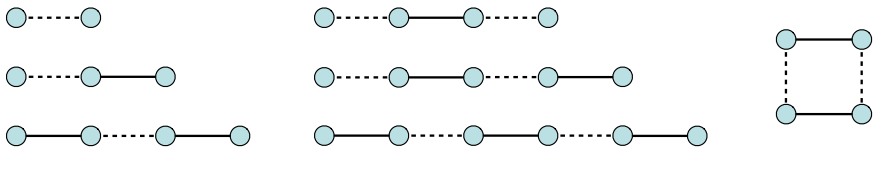

Figure 3.1. 2-augmenting paths and cycles.

has $l \leq k$ unmatched edges contains either $l-1$, l or $l+1$ matched edges. The gain of an augmentation is positive if the sum of the weights of the unmatched edges is greater than the sum of the weights of the matched edges. (Note that an alternating path with one more matched edge than unmatched edges could have positive gain, but would not be an augmenting path for the cardinality of the matching.)

3.1. Approximation lemma

It appears to be well known that if a matching does not admit positive gain $(k-1)$-augmentations then it must have a weight of at least a factor $(k-1)/k$ times the weight of a maximum matching $w(M^*)$. Still, to the best of our knowledge a proof of this does not appear to have been written down previously. We therefore include a formal proof here. This is a generalization of the proof given for Theorem 1 in Drake and Hougardy (2003*a*) which shows the result for $k = 2$.

Lemma 3.1. Let M be a matching on G and k an integer greater than 1 such that M does not admit any positive gain $(k-1)$-augmentations. Then

$$\frac{k-1}{k} w(M^*) \leq w(M),$$

where M^* is a maximum weight matching.

Proof. Let $S = M \cap M^*$, that is, S consists of the edges of G that are common to both matchings M and M^*. Further, let $R = M \setminus S$ and $R^* = M^* \setminus S$. Then $M = S \cup R$ and $M^* = S \cup R^*$. We will show that

$$w(R) \geq \frac{k-1}{k} \cdot w(R^*).$$

The result will then follow since

$$w(M) = w(S) + w(R) \geq \frac{k-1}{k} \cdot (w(S) + w(R^*)) = \frac{k-1}{k} \cdot w(M^*).$$

Since R and R^* are matchings, the subgraph induced by the edges $T = R \cup R^*$ consists of vertices of degree one or two. Thus the edges in T can

be split into a set of even length cycles C and a set of paths P. We show the result separately for each cycle in C and each path in P.

Consider a cycle $C_i \in C$. If C_i contains at most $2k-2$ edges then by the assumption of the lemma it follows that

$$w(C_i \cap M) \geq w(C_i \cap M^*) \geq \frac{k-1}{k} \cdot w(C_i \cap M^*).$$

Assume therefore that $|C_i| = 2r$ where $2r \geq 2k$. We denote by m_i edges of C_i that belong to the matching M, and by m_i^* edges that belong to M^*. Let $m_1, m_1^*, m_2, m_2^*, \ldots, m_{k-1}, m_{k-1}^*, m_k$ be a clockwise path of $2k-1$ consecutive distinct edges from C_i, where each $m_j \in M$ and each $m_j^* \in M^*$. Then by the assumption of the lemma, it follows that $\sum_{j=1}^{k} w(m_j) \geq \sum_{j=1}^{k-1} w(m_j^*)$. There are r distinct starting edges belonging to M for such a path, each one giving rise to a different inequality. In these inequalities, every $e \in C_i \cap M$ will appear once in every position m_j, where $1 \leq j \leq k$. Thus each $e \in C_i \cap M$ appears in exactly k of these inequalities. Similarly, each $f \in C_i \cap M^*$ appears in exactly $k-1$ inequalities. It follows that if we sum the left and right sides of the inequalities we get

$$k \cdot w(C_i \cap M) \geq (k-1) \cdot w(C_i \cap M^*),$$

which leads to the desired inequality for each $C_i \in C$.

Next, consider a path $P_i \in P$. Note first that $|P_i| \geq 2$ and thus contains at least one edge from each of M and M^*. If this were not the case then either M would contain an augmenting path of length one, or M^* would not be maximal. Now append two paths, each containing $2k-2$ dummy edges of weight 0, to the first and last vertex of P_i respectively. As in the case for C_i, repeatedly overlay a path $m_1, m_1^*, m_2, m_2^*, \ldots, m_{k-1}, m_{k-1}^*, m_k$ of $2k-1$ edges onto P_i such that each $e \in P_i \cap M$ appears exactly once for each m_j, where $1 \leq j \leq k$. Note that dummy vertices may be assigned to the start and the end of such paths. For each path the inequality

$$\sum_{j=1}^{k} w(m_j) \geq \sum_{j=1}^{k-1} w(m_j^*)$$

still holds true as any assigned dummy edges have weight 0. Since every $e \in P_i \cap M$ appears in exactly k inequalities, while each $f \in P_j \cap M^*$ appears in exactly $k-1$ inequalities, it follows that by taking the sum of the inequalities we get

$$w(P_i \cap M) \geq \frac{k-1}{k} \cdot w(P_i \cap M^*). \qquad \square$$

Note that the statement of Lemma 3.1 is slightly more restrictive than that of Lemma 2.1 in that the former lemma does not permit the existence of any weight-increasing path of length $2k-1$ edges with respect to the

Algorithm 3 LOCAL GREEDY($G(V, E, w)$)

```
1: M = ∅
2: Set every u ∈ V as unmarked
3: for u ∈ V do
4:     if u is unmarked then
5:         (u, v) = arg max_{x∈N(u)}{w(u, x) s.t. x is unmarked}
6:         Mark u and v
7:         M = M ∪ (u, v)
8:     end if
9: end for
10: return M
```

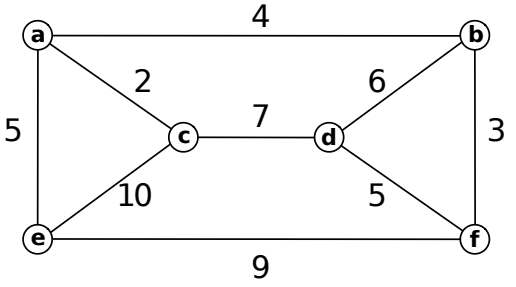

Figure 3.2. Example graph for approximate weighted matching.

matching M, where k edges belong to M and $k-1$ edges to M^*. Such an augmentation is not permissible for maximum cardinality matching as it would reduce the cardinality of the matching M.

3.2. Edge-weighted approximation algorithms

3.2.1. GREEDY *and* PATH GROWING *algorithms*

In the following we give an overview of various efforts at designing efficient linear or close to linear time algorithms for the edge-weighted matching problem.

Perhaps the simplest matching heuristic is to visit each vertex u once and unless u is already incident on a matched edge, add an edge (u, v) of maximum weight to the matching where v is not already incident on a matched edge. The heuristic is shown in Algorithm 3.

As an example consider the graph in Figure 3.2 consisting of six vertices and nine edges with integer weights. If the graph is traversed in lexicographical order then LOCAL GREEDY will compute the matching $M = \{(a, e), (b, d)\}$ of weight 11. Note that the optimal matching is given by $M^* = \{(a, b), (c, d), (e, f)\}$ of weight 20.

Algorithm 4 Greedy($G(V, E, w)$)

```
1: M = ∅
2: Set every u ∈ V as unmarked
3: for (u, v) ∈ E by non-increasing weight do
4:     if u and v are both unmarked then
5:         Mark u and v
6:         M = M ∪ (u, v)
7:     end if
8: end for
9: return M
```

While this strategy has linear run time in the size of G, it does not offer any bound on the quality of the obtained solution. To see this consider an even length (number of edges) path $v_1, v_2, \ldots, v_k$ where $w(v_i, v_{i+1}) = \epsilon$ if i is odd, $w(v_i, v_{i+1}) = \delta$ if i is even, and $\epsilon < \delta$. If the Local Greedy algorithm processes the vertices in the given order, it will compute a matching with weight $k/2 \cdot \epsilon$, while the optimal solution has weight $k/2 \cdot \delta$.

One way to obtain a quality guarantee on the computed matching is to visit the edges in a predefined order. This gives rise to the classical greedy matching algorithm as shown in Algorithm 4. Here a matching is computed by considering each edge e by non-increasing weights. Then if e is not adjacent to an edge already in the matching it is added to the matching; otherwise it is discarded.

In the graph in Figure 3.2 Greedy will compute $M = \{(c, e), (b, d)\}$ of weight 16. The following result shows that Greedy always gives a solution of weight at least half of the optimal one.

Lemma 3.2. The Greedy algorithm computes a matching M that is a 1/2-approximation to a maximum weight matching M^*.

Proof. We show that M does not admit a 1-augmentation, and the result will then follow from Lemma 3.1. For every $e \in E \setminus M$ there must be either one or two edges in M adjacent to e, otherwise e would have been added to M. Let $e' \in M$ be an edge adjacent to e of maximum weight. If $w(e) > w(e')$ then e would have been considered before any of its adjacent edges in M and then added to M. It follows that $w(e) \leq w(e')$ and that e cannot be part of a 1-augmentation with respect to M. □

It is clear that the Greedy algorithm runs in time $O(m \log n)$ due to the sorting of the edges. However, it is possible to compute a 1/2-approximation in linear time. One such is the Path Growing algorithm (Drake and Hougardy 2003*b*), shown in Algorithm 5. In line 9 of the algorithm, we concatenate the edge (u, v) to the path P.

Algorithm 5 PATH GROWING$(G(V, E, w))$

```
1: M = ∅
2: Mark each u ∈ V as unvisited
3: for u ∈ V do
4:     if u is unvisited then
5:         P = {}
6:         Mark u as visited
7:         while u has unvisited neighbours do
8:             v = arg max_{x∈N(u)}{w(u, x), x is unvisited}
9:             P = P ∘ (u, v)
10:            u = v
11:            Mark u as visited
12:        end while
13:    end if
14:    Add the heavier set of the odd or even numbered edges of P to M
15: end for
16: return M
```

In its original form the algorithm starts at an arbitrary vertex u and then grows a path P from u by traversing a heaviest edge incident on u that leads to an unvisited vertex v. The process is then continued from u and terminates when a vertex is reached such that there is no edge leading to an unvisited vertex. The result is a path $P = e_0, e_1, \ldots, e_k$. This is then repeated starting from every unvisited vertex. For each such path one selects the heavier set of the odd or even numbered edges to add to M.

In the graph in Figure 3.2 starting from vertex a PATH GROWING would follow the path a, e, c, d, b, f. The edges of this path would then be divided into sets $\{(a, e), (c, d), (b, f)\}$ (odd) of weight 15 and $\{(c, e), (b, d)\}$ (even) of weight 16. The final solution consists of the even set.

The run time of PATH GROWING is clearly $O(n + m)$ as the search for a path is only started once from each vertex and each edge is only considered twice (once from each of its endpoints). Note that the resulting matching might contain 1-augmentations. This is the case for a path of length four, e_0, e_1, e_2, e_3 with edge weights 3, 1, 2 and 3. Then the resulting matching will consist of the edges e_0 and e_2; however, e_2 and e_3 form a 1-augmentation. Thus it is not possible to use Lemma 3.1 to show the performance guarantee. Still, it is not hard to prove that the resulting matching is a 1/2-approximation.

Lemma 3.3. The PATH GROWING algorithm computes a matching M that is a 1/2-approximation to a maximum weight matching M^*.

Proof. Let $K = P_0, P_1, \ldots, P_k$ be the non-empty paths generated by Path Growing. We show that $\sum_i w(P_i) \geq w(M^*)$. The result then follows since

$$w(M) \geq \sum_i \frac{1}{2} \cdot w(P_i) \geq \frac{1}{2} \cdot w(M^*).$$

Consider an edge $(u, v) \in M^*$. Then if (u, v) is not part of some P_i at least one of u and v must be part of some path P_j, otherwise Path Growing would have started a new path from either u or v. Assume therefore that u is part of P_i and that if v is part of P_j then $i < j$. This ensures that u is visited before v irrespective of if v is part of some P_j or not. Thus when Path Growing visits u it is possible to choose the edge (u, v). Since this is not done it must choose an edge (u, x) to add to P_i such that $w(u, x) \geq w(u, v)$. Note that u can only be incident on at most one edge in M^*. It therefore follows that for every edge $e \in M^*$ either e is in some P_i or there is a unique edge $e' \in P_i$ such that $w(e) \leq w(e')$. □

The Path Growing algorithm can be improved further without increasing its asymptotic run time, by noting that one can compute an optimal matching for each P_i instead of just choosing between the even and odd numbered edges. This is done using dynamic programming as follows. Let $P = e_0, e_1, \ldots, e_k$ be the edges on a selected path and let $\mathrm{Opt}(i)$ denote the weight of an optimal matching on $e_0, e_1, \ldots, e_{i-1}$. Then the weight of an optimal matching on P can be computed in linear time using the recursion

$$\mathrm{Opt}(i) = \max\{\mathrm{Opt}(i-1), \mathrm{Opt}(i-2) + w(e_{i-1})\},$$

with $\mathrm{Opt}(0) = 0$ and $\mathrm{Opt}(1) = w(e_0)$. The actual matching can be calculated by storing a Boolean value for each i to indicate if e_{i-1} is used in the computation of $\mathrm{Opt}(i)$. We denote this algorithm as Path Growing-DP.

The first linear time 1/2-approximation algorithm for weighted matching was given by Preis (1999), and it is based on the notion of a *dominating* edge. An edge e is dominating if it is heavier than all of its incident edges. This Locally Dominant Edge algorithm repeatedly locates and adds dominant edges to the matching. When an edge is added to the matching, all adjoining edges are removed from the graph before the next dominant edge is found. The outline of this strategy is shown in Algorithm 6.

To be able to break ties we rank edges of equal weight first by the ID of the higher-numbered endpoint and then by the lower-numbered one. In practice we will only be comparing edges of equal weight incident on the same vertex and thus we always break ties by comparing the ID of the opposing vertex.

To achieve the linear time run time Algorithm 6 performs a carefully designed depth-first search (DFS) of G, locating and removing dominating edges as it progresses. Since the actual algorithm is quite involved, we

Algorithm 6 Locally Dominant Edge($G(V, E, w)$)

```
M = ∅
Mark each u ∈ V as unvisited
while E is not empty do
    Locate a dominating edge (u, v)
    M = M ∪ (u, v)
    Remove all edges incident on u and v from E
end while
return M
```

do not list the details. Moreover, a DFS search is inherently linear, thus prohibiting parallel execution. Hence we present variants of Algorithm 6 that have worse theoretical run times, but which are faster in practice and that also lend themselves better to parallel execution.

Lemma 3.4. If tie-breaking is done consistently, Algorithm 6 computes the same matching as the Greedy algorithm.

Proof. Let M and M' be the matchings computed by Algorithms 4 and 6, respectively. Let $S = e_0, e_1, \ldots, e_{m-1}$ be the edges in E by non-increasing weight. The proof is by induction on the edges in S with the induction hypothesis being that M and M' consists of exactly the same edges from $e_0, e_1, \ldots, e_k$.

For $k = 0$ we only consider the heaviest edge in G. Obviously this will be included in the matchings computed by both algorithms. Assume therefore that the induction hypothesis holds for all $l < k$, where $k \geq 1$, and consider edge e_k. If e_k is not used by Greedy then it must be adjacent to some edge e_l where $l < k$ and $e_l \in M$. By assumption $e_l \in M'$, and since it is not possible for two adjacent edges to be picked by Algorithm 6, it follows that $e_k \notin M'$.

If $e_k \in M$ then it cannot be adjacent to any edge $e_l \in M$ where $l < k$. The only way that Algorithm 6 can avoid including e_k in M' is if it is removed because some adjacent edge e_l is included in M'. For e_l to be dominant before e_k is removed requires that $w(e_l) > w(e_k)$. Since the edges are considered by non-increasing weight this implies that $l < k$, and thus by the induction hypothesis $e_l \notin M'$. It follows that $e_k \in M'$ if and only if $e_k \in M$. □

The simplest algorithm based on discovering dominating edges is the Local Max algorithm of Birn *et al.* (2013). The algorithm operates in stages. At each stage a maximal set of dominating edges are discovered and added to the matching. These edges, along with their vertices, are then removed from the graph before the process is repeated. The algorithm

terminates when the graph is empty. It is not hard to see that the run time of each stage is linear in the size of the remaining graph. It is possible that only a constant number of vertices and edges are removed at each stage, thus leading to a worst-case run time of $O(nm)$. However, it can be shown that if the edge weights are drawn from a uniform random distribution, then half of the remaining edges will be removed at each iteration (Birn *et al.* 2013). Thus in this case the expected number of rounds is $O(\log n)$ and the expected run time is $O(m)$.

3.2.2. The Suitor *algorithm*

The Suitor algorithm, designed by Manne and Halappanavar (2014), is another variant of Algorithm 6 that also computes the same matching as Greedy. The algorithm is based on considering the vertices as players that are making bids to match with one of their neighbours. The value of a bid from a vertex u to a vertex v is $w(u, v)$. We denote the highest bid a vertex u has received as its suitor value, denoted by $s(u)$.

The algorithm works by each vertex u proposing to its neighbour v such that $w(u, v)$ is maximized under the constraint that v has not already received a higher offer than $w(u, v)$. (We say that a vertex v that satisfies this property is an eligible neighbour of u.) If at a later stage in the algorithm v receives a better offer, then the bid from u to v is annulled, and u has to make a new bid to its next heaviest eligible neighbour. The algorithm terminates when every vertex u is either a suitor of one of its neighbours or when u has no more vertices to propose to. At this point the edges where $s(u) = v$ and $s(v) = u$ constitute the matching.

There is no restriction on the order in which the vertices in V are processed nor on the order in which rejected suitors are processed. This leaves considerable freedom in how the algorithm is implemented. Two of the most natural ways to handle this is to use either a queue or a stack to organize the vertices that still need to be processed. If a queue is used then all vertices are initially put in the queue and the next considered vertex is always taken as the head of the queue. Any vertex that gets dislodged as a suitor because its chosen neighbour receives a better offer will be added at the end of the queue. The algorithm terminates when the queue is empty.

When using a stack all vertices are initially put on the stack and the next vertex to process is taken from the top of the stack. Any vertex that gets dislodged as a suitor is put on the top of the stack. Note that this is equivalent to processing each vertex exactly once and then immediately starting to process any vertex that gets dislodged. This variant of the Suitor algorithm is shown in Algorithm 7. In the algorithm, if a vertex x does not have a suitor, *i.e.* $s(x) = \text{NULL}$, then $w(u, s(x)) = 0$.

As to correctness consider any initial dominating edge (u, v), for instance the heaviest edge in E. Then the first time u is processed, the value of

Algorithm 7 Suitor($G(V, E, w)$)

```
1: for each u ∈ V do
2:     s(u) = NULL
3: end for
4: for each u ∈ V do
5:     repeat
6:         v = arg max_{x∈N(u)}{w(u, x) : w(u, x) > w(x, s(x))}
7:         t = u
8:         u = s(v)
9:         if v ≠ NULL then
10:            s(v) = t
11:        end if
12:    until u == NULL
13: end for
14: return M
```

$s(v)$ will be set to u. This follows since u has no better alternative and it cannot be prevented from doing so by any other neighbour $x \in N(v)$ as $w(u, v) > w(v, x)$. Using the same argument it follows that $s(u)$ will be set to v when v is initially processed. The suitor values of u and v will remain unchanged throughout the execution for the rest of the algorithm, again because (u, v) is dominant. It follows that for the final result of the vertices in $V - \{u, v\}$, the net effect is the same as if u and v were removed initially along with their adjacent edges. The only difference is that some vertices might become temporary suitors of u or v, but when they are dislodged, as they will be, they will behave as if u and v had been removed initially. Thus Algorithm 7 follows the same pattern as Algorithm 6 in computing the same matching as Greedy.

For the time complexity note first that a vertex can only propose once to each of its neighbours. Thus it follows that there can at most be $2m$ proposals. The time complexity therefore depends on how the next vertex to propose to is determined. If a linear search is performed each time, the time complexity is $O(n + m\Delta)$. If the weights of the edges incident on each vertex are sorted initially, then the rest of the algorithm runs in $O(n + m)$ time. The run time will then be dominated by the sorting, which takes time $O(m \log \Delta)$. Other strategies include using a partial Quicksort partitioning to find the heaviest neighbours, and then only when these do not contain any more potential candidates to match with are more candidates found.

In order to say something about the expected run time of Suitor we note that there is a strong relationship between computing a greedy matching and the stable marriage problem (SM).

In the SM problem we are given two equal-sized sets $L = \{l_1, l_2, \ldots, l_n\}$ and $R = \{r_1, r_2, \ldots, r_n\}$, typically referred to as 'men' and 'women' respectively. Every man and woman has a total ranking of all the members of the opposite sex. These give the 'desirability' for each participant to match with a member of the other set. The object is to find a complete matching M (*i.e.* a pairing) between the entries in L and R such that no two $l_i \in L$ and $r_j \in R$ would both obtain a higher-ranked partner if they were to abandon their current partner in M and rematch with each other. Any such solution is *stable*.

The main differences between computing a matching in a bipartite graph and an SM instance is that for SM one only considers how attractive two partners are relative to each other and also that attractiveness might not be symmetric. Thus one can have cycles such as l_1, r_1, l_2, r_2, l_1 where each one prefers to match with the next one in the list. This does not happen in the matching problem. Moreover, the overall objective is to find any stable solution, whereas in a matching instance possible solutions can be ranked.

Gale and Shapley (1962) formulated the stable marriage problem and also proposed the first algorithm for solving it. The algorithm operates in rounds as follows. In the first round each man in L proposes to his most preferred woman in R. Each woman will then reject all proposals she has received in this round except the one that is the highest in her ranking. In subsequent rounds each man that was rejected in the previous round will again propose to the woman that he has ranked highest, but now disregarding any woman that he has already proposed to in previous rounds. Gale and Shapley showed that this process will terminate with each man in L being matched to a woman in R and that this solution is stable. The algorithm also converges to a stable solution even if each participant has only ranked a subset of the opposing participants. For a recent discussion of stable matching and generalizations such as stable room-mates, stable fixtures, *etc.*, see the book by Manlove (2013).

Manne, Naim, Lerring and Halappanavar (2016) show that a greedy matching on a graph G can be computed by reformulating the problem as an appropriately constructed instance of SM and then solving this using the Gale–Shapley algorithm. In fact, the Gale–Shapley algorithm has a strong resemblance to the Suitor algorithm when using a queue. Similarly, there is a variation of the Gale–Shapley algorithm due to McVitie and Wilson (1971) corresponding to the Suitor algorithm when using a stack.

Wilson (1972) showed that for any profile of women's preferences, if the men's preferences are random, then the expected sum of men's rankings of their mates as assigned by the Gale–Shapley algorithm is bounded above by nH_n, where H_n is the nth harmonic number. Knoblauch (2007) showed that this is also an approximate lower bound in the sense that the ratio of the expected sum of men's rankings of their assigned mates and $(n+1)H_n - n$

has limit 1 as $n \to \infty$. Thus if the men's preferences are random then this sum is $\Theta(n \ln n)$ for large n. However, it is not hard to design instances where this sum is $\Theta(n^2)$. One such case is when the men have identical preferences. These results carry over to the Suitor algorithm in that for a graph with random weights the expected number of proposals made is also bounded by $O(n \ln n)$, and if the graph is sufficiently dense by $\Theta(n \ln n)$.

The depth of a parallel algorithm is the length of the critical path in the algorithm, and the work in the algorithm is the total number of operations performed by the algorithm. Blelloch, Fineman and Shun (2012) have shown that a maximal matching in a graph can be computed with $O(m)$ work and $O(\log m \log \Delta)$ depth. Their proof technique was adapted by Khan, Pothen and Ferdous (2018*b*) to show that the parallel Suitor algorithm has $O(m)$ work and $O(\log m \log \Delta)$ depth when the edge weights are chosen uniformly at random.

Among the algorithms presented thus far, experimental evaluations have shown that the highest-weight matching is obtained by Path Growing-DP. This is due to the use of dynamic programming to select an optimal solution on the discovered paths. The Global Paths Algorithm by Maue and Sanders (2007) further expands this idea, by trying to get a heavier set of edges on which to perform the dynamic programming. The algorithm starts by sorting the edges and then considers each edge by order of non-increasing weight, similar to the Greedy algorithm. An edge is kept for further consideration in a set S if it either connects at most two paths already contained in S, or if it completes a cycle of even length contained in S. The algorithm then uses dynamic programming on the paths and cycles in S to obtain the final matching. Even though the Global Paths Algorithm in most instances gives higher weight matchings compared to Path Growing-DP, it does not improve the approximation ratio beyond half.

The idea of computing optimal matchings on restricted subgraphs was also used by Manne and Halappanavar (2014) in the M1M2 algorithm. This algorithm is based on the observation that the union of the edges from two matchings always consists of paths and even length cycles. The M1M2 algorithm first computes a greedy matching M_1 on a graph $G(V, E, w)$. It then computes a second greedy matching M_2 on $G(V, E \setminus M_1, w)$ before performing dynamic programming on $M_1 \cup M_2$ in the same way as was done in Global Paths Algorithm. This gave results of a similar quality to Global Paths Algorithm. Further use of this idea was shown in Idelberger and Manne (2014) where the dynamic programming was expanded to maximum weight spanning trees, while also performing mergers of multiple matchings following a tree-like structure.

While the algorithms presented thus far tend to perform quite well in practice, often producing optimal or close to optimal solutions, they all

have in common that they cannot guarantee a performance ratio higher than 1/2. To do so requires that one considers augmentations containing more than one unmatched edge.

Drake and Hougardy (2005) presented the first such algorithm which computes a $(2/3 - \epsilon)$-approximation in time $O(m\epsilon^{-1})$. The algorithm is based on increasing the weight of a maximal matching by repeatedly performing 2-augmentations. This algorithm was subsequently simplified by Pettie and Sanders (2004) who also improved the run time to $m \log \epsilon^{-1}$. Their algorithm is based on finding the best 2-augmentation *centred* in a node u. For a matching M and 2-augmenting path P, a centre node $u \in P$ has the property that each edge $(x, y) \in P \setminus M$ is either incident on u or on the matching partner v of u. For a particular vertex u one can then find the best 2-augmenting path centred in u in time $O(\deg(u) + \deg(v))$.

Pettie and Sanders presented both a simple randomized algorithm RAMA and a slightly more involved deterministic one. In RAMA one picks a random vertex u and then finds the best 2-augmentation centred at v and augments if this gives a positive gain. By repeating this $(1/3) \log \epsilon^{-1}$ times the algorithm has an expected run time of $O(m \log \epsilon^{-1})$ and an expected performance ratio of $2/3 - \epsilon$ (Pettie and Sanders 2004). The algorithm can either start with an empty matching or from an existing one. A variant of RAMA, named ROMA was later determined to be more effective in practice (Maue and Sanders 2007). In ROMA the algorithm iterates multiple times through all vertices in a random order and performs the same augmentation step as in RAMA. This algorithm computed a higher weight matching when initialized with a matching given by Global Paths Algorithm, albeit at the cost of higher run times. Subsequent developments have given $(3/4 - \epsilon)$-approximation algorithms running in time $O(m \log n \log \epsilon^{-1})$ (Duan and Pettie 2010, Hanke and Hougardy 2010). However, these have not been tested in practice.

Finally, Duan and Pettie (2014) presented a $(1 - \epsilon)$-approximation algorithm that runs in time $O(m\epsilon^{-1} \log \epsilon^{-1})$. This algorithm employs the scaling approach to computing weighted matchings, with the subproblem at each scale solved by a primal–dual linear programming formulation of matching. The feasibility and complementary slackness conditions are enforced approximately, with the violation of these conditions dynamically dependent on the scale of the computation.

3.3. Experiments on edge-weighted matching algorithms

The data set consists of ten graphs taken from the SuiteSparse collection (Davis and Hu 2011). Structural properties of these graphs are shown in Table 3.1.

Table 3.1. Structural properties of graphs used for weighted matching, sorted in increasing order of edges.

Problems	Vertices	Edges
af_shell10	1 508 065	25 582 130
Serena	1 391 349	31 570 176
audikw_1	943 695	38 354 076
channel-500x100x100-b050	4 802 000	42 681 372
delaunay_n24	16 777 216	50 331 601
europe_osm	50 912 018	54 054 660
Flan_1565	1 564 794	57 920 625
Cube_Coup_dt6	2 164 760	62 520 692
kron_g500-logn21	2 097 152	91 040 932
nlpkkt200	16 240 000	215 992 816

3.3.1. Comparison of exact and approximation algorithms

We begin by comparing the performance of exact algorithms for maximum edge-weighted matching with several approximation algorithms. We report on the exact algorithm implemented in LEDA (Mehlhorn and Näher 1999). We report on three variant algorithms: LEDA1 uses no initialization, LEDA2 employs a greedy 1/2-approximation matching, and LEDA3 uses a fractional matching initialization. The latter initialization computes a fractional $\{0, 1/2, 1\}$ solution to a linear programming formulation of the edge-weighted matching problem; the solution is computed by a combinatorial algorithm, not an LP solver. The fractional solution is then rounded to obtain an initial matching. The $(2/3 - \epsilon)$-approximation algorithm ROMA is initialized with the Global Paths Algorithm and the Suitor algorithms. We report results for the RAMA algorithm without any initialization since the initialized versions computed lower weights than the corresponding variants of the ROMA algorithm; however, it is faster than the latter. The Duan and Pettie $(1 - \epsilon)$-approximation algorithm for weighted matching was implemented by Al-Herz and Pothen (2019), and we report results for it as well.

For the experiments we used an Intel Xeon E5-2660 processor-based system (part of the Purdue University Community Cluster), called *Rice*.[1] The machine consists of two processors, each with ten cores running at 2.6 GHz

[1] https://www.rcac.purdue.edu/compute/rice/

(20 cores in total) with 25 MB unified L3 cache and 64 GB of memory. The operating system is Red Hat Enterprise Linux release 6.9. All code was developed using C++ and compiled using the g++ compiler (version: 4.4.7) using the –O3 flag.

Run times and relative performances are reported in Table 3.2. Note that LEDA1 does not terminate on the last problem in the set, whereas LEDA2 and LEDA3 do. For the other nine problems, there is not much difference between the uninitialized version and the greedy initialization, whereas the fractional matching initialization is more effective, about four times faster than the uninitialized version. The $(2/3 - \epsilon)$-algorithms and the $(1 - \epsilon)$-approximation algorithms are 20–30 times faster than LEDA1, while the SUITOR algorithm is about 1700 times faster, in geometric mean. For the last problem, we use LEDA2 as the baseline algorithm, the SUITOR algorithm is 2000 times faster than LEDA2, while for the other approximation algorithms the factor is in the range 35–50.

Now we compare the weights computed by the algorithms. All the exact algorithmic variants compute the same weight, and this is reported in Table 3.3. For the approximation algorithms we report the gap to optimality expressed as a percentage. This value is computed as the ratio of the difference in weight between the maximum and approximate weights and the maximum weight. The SUITOR algorithm computes more than 94% of the maximum weight in geometric mean, with the lowest value being 78%. ROMA obtains a lower gap than RAMA, and the ROMA algorithm initialized with the SUITOR algorithm computes the highest weight among the $(2/3 - \epsilon)$-approximation algorithms, but the $(1 - \epsilon)$-algorithm with $\epsilon = 1/4$ lowers the gap further to about 1.6%. Note that all these algorithms obtain much better weights than their worst-case approximation ratios.

Finally we compare the cardinalities of the maximum weight matching and approximate matchings in Table 3.4. Note that the cardinality of the exact algorithm is not necessarily equal to that of the maximum cardinality matching. The gap in cardinality is computed analogously to the gap in weights. The cardinality of the approximation algorithms is lower than that of the exact algorithm. The gap in cardinalities is about 3% for SUITOR, while it is less than 0.5% for the other approximation algorithms.

The relative performance of these algorithms has a strong dependence on the weights. When we use these algorithms to solve vertex-weighted matchings (with vertex weights chosen uniformly at random, and edge weights computed by adding the vertex weights on the endpoints of an edge), there are problems for which the exact algorithms do not terminate in hundreds of hours. For these problems the $(2/3 - \epsilon)$-algorithms tend to obtain better weights than the $(1 - \epsilon)$-approximation algorithm. We discuss these in Section 5 on vertex-weighted matchings.

Table 3.2. Comparing the run times of exact and approximation algorithms for maximum edge-weighted matching. Edge weights are from a uniform random distribution in the range [1 1000]. Run times of LEDA without any initialization are reported in LEDA1, and for all other algorithms the ratios of the run time of LEDA1 to the other algorithm are presented. For the last problem in the set, LEDA1 did not complete in four hours, and LEDA with greedy initialization (LEDA2) is the baseline. LEDA3 is the LEDA matching algorithm with fractional matching initialization.

	Time (s)	Relative performance							
	Exact	Exact	Exact	$1-\epsilon$		RAMA	ROMA	SUITOR, ROMA	SUITOR
	LEDA1	LEDA2	LEDA3	Scaling		$2/3-\epsilon$			
				$\epsilon = 1/4$	$\epsilon = 1/3$		$\epsilon = 0.01$		
af_shell10	310	1.10	1.70	18.00	24.00	24	23.00	23	1200
Serena	610	1.10	2.10	49.00	63.00	37	36.00	35	1400
audikw_1	990	1.00	2.50	120.00	150.00	62	62.00	58	2300
channel-500x100x100-b050	610	1.30	2.80	13.00	14.00	17	14.00	15	920
delaunay_n24	720	1.60	4.40	2.60	2.90	12	7.90	11	610
europe_osm	3200	3.30	12.00	4.80	6.00	31	9.20	26	1600
Flan_1565	940	1.10	4.80	49.00	80.00	35	33.00	32	1500
Cube_Coup_dt6	1400	1.20	4.70	71.00	94.00	43	38.00	38	1800
kron_g500-logn21	3500	2.00	15.00	41.00	51.00	64	36.00	53	3800
Geometric mean		1.40	4.30	24	30	32	24	29	1500
		Time (s)		Relative performance					
nlpkkt200		7100	2.30	44	50	39	40	36	1700

Table 3.3. Comparison of weights obtained by the exact and approximation algorithms for maximum edge-weighted matching. For each algorithm, we report the difference between one and the ratio of the weight computed by the approximation algorithm and the maximum weight obtained by the exact algorithm, expressed as a percentage.

	Weight	Gap to optimal weight (%)					
	Exact	$1-\epsilon$		RAMA	ROMA	Suitor, ROMA	Suitor
	LEDA	Scaling		$2/3-\epsilon$			
		$\epsilon = 1/4$	$\epsilon = 1/3$	$\epsilon = 0.01$			
af_shell10	7.20E+08	1.30	1.70	3.90	3.60	2.00	5.10
Serena	6.70E+08	1.20	1.70	4.20	3.90	2.10	4.90
audikw_1	4.60E+08	1.40	1.80	3.70	3.50	2.00	4.70
channel-500x100x100-b050	2.20E+09	1.80	2.40	4.80	4.30	2.70	6.90
delaunay_n24	6.40E+09	2.60	3.60	3.00	2.30	2.10	7.90
europe_osm	1.50E+10	1.30	1.70	1.50	0.50	0.53	3.70
Flan_1565	7.70E+08	1.10	1.40	3.50	3.20	1.60	3.60
Cube_Coup_dt6	1.10E+09	1.10	1.40	3.80	3.50	1.80	3.90
kron_g500-logn21	3.50E+08	4.70	6.10	2.80	1.80	1.70	22.00
nlpkkt200	7.60E+09	1.40	1.90	4.10	3.60	2.10	5.70
Geometric mean		1.60	2.10	3.40	2.70	1.70	5.80

Table 3.4. Comparison of cardinalities obtained by the maximum edge-weighted matching algorithm and the approximation algorithms. Note that this is not the cardinality obtained by a maximum cardinality matching. For each algorithm, we report the difference between one and the ratio of the cardinality of the matching computed by the approximation algorithm and the cardinality of the maximum edge-weighted matching, expressed as a percentage.

	Cardinality	Gap to cardinality of maximum weight matching (%)					
	Exact	$1-\epsilon$		RAMA	ROMA	Suitor, ROMA	Suitor
	LEDA	Scaling		$2/3-\epsilon$			
		$\epsilon = 1/4$	$\epsilon = 1/3$	$\epsilon = 0.01$			
af_shell10	754 032	0.16	0.24	0.18	0.16	0.19	1.80
Serena	695 674	0.08	0.14	0.09	0.08	0.09	1.80
audikw_1	471 847	0.01	0.03	0.04	0.04	0.04	1.60
channel-500x100x100-b050	2 400 980	0.76	0.99	0.48	0.42	0.41	3.60
delaunay_n24	8 217 157	4.10	4.60	2.60	2.00	2.20	8.20
europe_osm	22 996 323	3.50	3.80	1.90	0.83	1.00	5.90
Flan_1565	782 397	0.01	0.02	0.03	0.03	0.03	1.00
Cube_Coup_dt6	1 082 380	0.02	0.03	0.04	0.04	0.04	1.20
kron_g500-logn21	442 107	8.30	9.60	4.20	3.20	3.20	24.00
nlpkkt200	8 000 000	0.33	0.47	0.14	0.13	0.16	2.80
Geometric mean		0.25	0.36	0.26	0.21	0.23	3.10

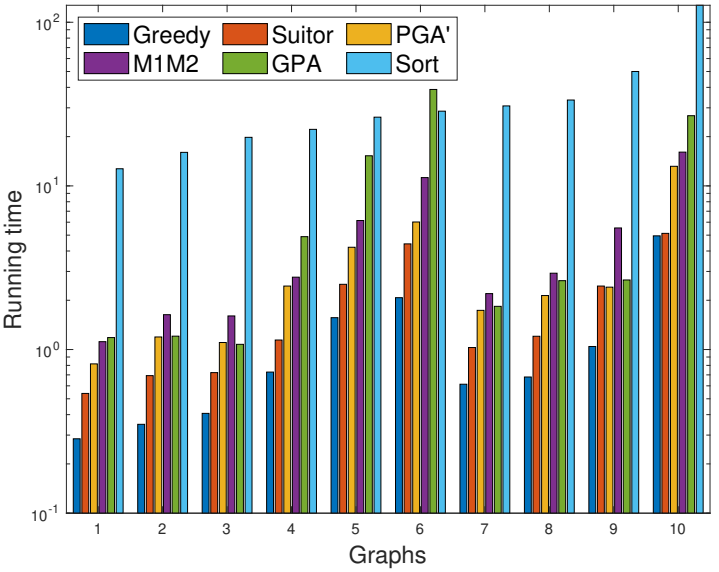

Figure 3.3. Run times for five 1/2-approximation algorithms.

3.3.2. Comprehensive comparison of approximation algorithms

Now we compare the various approximation algorithms using original edge weights from the graphs rather than the random integer weights used in the previous experiments. We first compare the 1/2-approximation algorithms. Figure 3.3 shows the run time of the GLOBAL PATHS ALGORITHM (GPA), and the GREEDY, SUITOR, PATH GROWING-DP and M1M2 algorithms for the ten graphs. For GREEDY and PATH GROWING-DP the time to sort the edges is shown separately. This is done using the standard *qsort* routine in C. Thus for an application where the edges are not already sorted, then this time must be added to get the total run time.

For these experiments we used a Dell computer running GNU/Linux equipped with four 10 core 2.00 GHz Intel Xeon E7-4850 processors with a total of 128 GB memory. Algorithms were implemented in C using OpenMP for parallelization and compiled with `gcc` using the –O3 flag.

As can be seen from the figure, for all but one graph the time to sort dominates the time taken by all other algorithms, in many cases by as much as an order of magnitude. When one excludes the time to sort, GREEDY is the fastest algorithm followed by SUITOR and PATH GROWING-DP. These again outperform M1M2 and the GLOBAL PATHS ALGORITHM. If one includes the time to sort the edges then SUITOR outperforms GREEDY while M1M2 outperforms the GLOBAL PATHS ALGORITHM.

To measure the quality of the solutions given by these algorithms we compute the average performance ratio for each algorithm compared to the

best algorithm for each graph. This shows that Greedy, Path Growing-DP, M1M2 and Path Growing algorithms have an average performance ratio of 2.44%, 1.67%, 0.21% and 0.04%, respectively, higher than the best algorithm for each graph. Thus it follows that the more time-consuming algorithms also gives better results.

Next, we compare the use of either RAMA or ROMA for post-processing a solution. We first note that in terms of run time RAMA is about 10% slower than ROMA. Each application of ROMA is on average 4.9 times slower than the corresponding edge sorting. Thus there is a fairly large cost for using this post-processing. As there is also a slight advantage of using ROMA over RAMA in terms of quality of solution, we only present results from using ROMA.

How much improvement ROMA gives depends on the starting configuration. The average improvement for using Suitor followed by ROMA was 4.9% while both M1M2 and Global Paths Algorithm were improved by 2.6%. In terms of absolute quality of the final solution Suitor followed by ROMA always gave the best solution. On average M1M2 was 0.19% worse while Global Paths Algorithm was 2.11%. Thus, in terms of both run time and quality, the preferred strategy is to use Suitor followed by ROMA.

In Figure 3.4 we show the speedup curves for Suitor, M1M2 and Local Max algorithms when run on the graph nlpkkt200. For both Suitor and M1M2 the speedups increase almost linearly as long as there are available threads. Although the CPUs can perform hyper-threading, this does not give any additional speedup when using 50 threads. The speedup for Local Max drops off earlier than Suitor. The sequential run time for Local Max is about 70% higher than that of Suitor. Combined with the lower speedup it follows that the parallel run time of Local Max is even worse.

Approximation algorithms for edge-weighted matchings have been implemented on GPUs by several authors. We list some of these papers: Cohen and Castonguay (2012), Fagginger Auer and Bisseling (2012), Halappanavar *et al.* (2012) and Naim *et al.* (2015).

3.4. Applications of matchings

3.4.1. Maximum cardinality matching

Maximum cardinality matchings in bipartite graphs have been employed in sparse matrix algorithms for several purposes. A maximum matching can be used to permute a sparse matrix to place the largest number of non-zeros on the diagonal. This number is the structural rank of the matrix, which is an upper bound on the numerical rank.

In the following we say that a matrix has a row-perfect matching to mean that its bipartite graph consisting of row vertices and column vertices has a

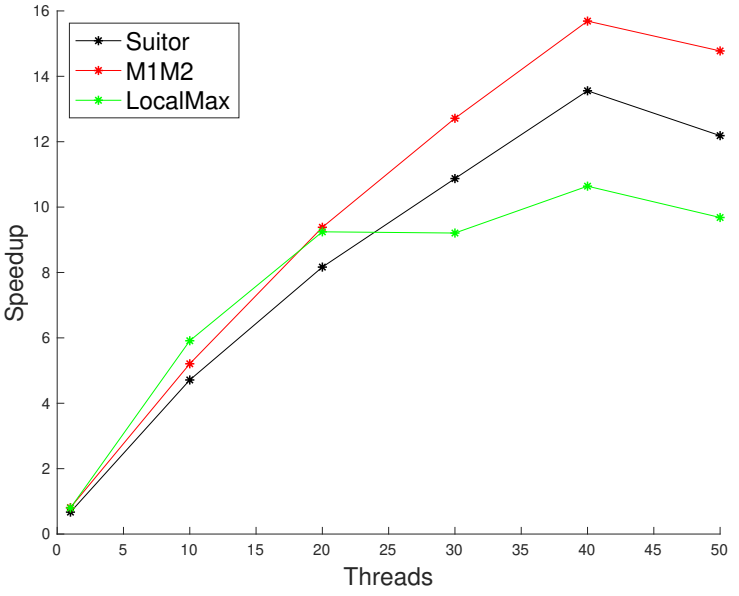

Figure 3.4. Speedup for SUITOR, M1M2 and LOCAL MAX algorithms on the nlpkkt200 graph.

matching in which all rows are matched. The concept of a column perfect matching is similar, and a perfect matching is both row-perfect and column-perfect.

A maximum matching can be used to compute the block (lower) triangular form (BTF) of a sparse matrix, which has the block matrix structure

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & A_{33} \end{pmatrix},$$

where A_{11} has fewer rows than columns and has a row-perfect matching, A_{22} is a square matrix with a perfect matching, and A_{33} has fewer columns than rows and has a column-perfect matching. The submatrices A_{12}, A_{13} and A_{23} could be zero. The submatrix A_{11} has the *strong Hall property* with respect to its rows (Coleman, Edenbrandt and Gilbert 1986, Pothen and Fan 1990). This property states that every set of k rows has non-zeros in at least $(k+1)$ columns. The submatrix A_{33} has the strong Hall property with respect to its columns, and A_{22} has the strong Hall property with respect to both rows and columns (in this last case $k < n$, where n is the dimension of A_{22}).

The BTF of a sparse matrix is derived from the Dulmage–Mendelsohn decomposition of the bipartite graph of the matrix, which could be computed from a maximum matching in the graph. The subgraph corresponding to A_{11} is obtained by following all alternating paths from unmatched columns, and including all rows and columns reached in the subgraph. The subgraph corresponding to A_{33} is obtained by following all alternating paths from unmatched rows, and including all rows and columns reached. The submatrix corresponding to A_{22} consists of all rows and columns not reached thus far, and they must be matched to each other since all unmatched columns and rows have been accounted for. (There is a finer decomposition for A_{22} which we do not describe here.) The decomposition is unique for a bipartite graph and is independent of the specific maximum matching used to induce it. Further details may be found in Pothen and Fan (1990).

The BTF can be used to reduce the work in solving sparse linear systems of equations by a factorization-based algorithm, since only the diagonal blocks in the BTF need to be factored. The BTF has been used to solve Kirchoff's equations in circuit design within the Xyce circuit simulation code (Keiter *et al.* 2011), and it is also used to solve nonlinear systems of equations in systems modelling software such as Modelica (Fritzson 2014). The strong Hall matrices in the BTF could be used to correctly predict the non-zero structures of orthogonal-triangular (QR) factors of sparse matrices (Coleman *et al.* 1986, Pothen 1993). Matchings have also been used to compute sparse bases for the null space of sparse, under-determined matrices, as also sparse bases for the column space of such matrices (Coleman and Pothen 1987, Pinar, Chow and Pothen 2006).

In applications such as the BTF, one needs to compute a maximum cardinality matching in bipartite graphs, and an approximation will not suffice. Duff, Kaya and Uçar (2011) have addressed the several choices one needs to obtain an efficient practical algorithm. The Karp–Sipser algorithm is used as an initialization, and then there are choices to be made on if the augmenting path searches should use breadth-first search (BFS) or depth-first search (DFS) or some combination of them. It is also clear from their work and other authors, that the theoretically less efficient $O(nm)$ time algorithms are faster than the Hopcroft–Karp algorithm with $O(n^{1/2}m)$ time complexity. Azad, Buluç and Pothen (2017) have designed efficient shared-memory parallel algorithms for this problem. A distributed-memory parallel algorithm scaling to hundred-thousand cores has been implemented using sparse matrix vector products using the Graph-BLAS operations by Azad and Buluç (2016).

3.4.2. Maximum edge-weighted matchings

An important application we consider is sparse Gaussian elimination (LU factorization). Here, permuting the sparse matrix to have large elements

on the diagonal before the numerical factorization makes it less likely that numerical pivoting (row permutations) will be required during the computation to prevent large element growth in the factors. Olschowka and Neumaier (1996) described a pivoting strategy using a primal–dual algorithm for the assignment problem, which was then adapted and implemented for sparse bipartite graphs by Duff and Koster (2001). A perfect matching that has the maximum product of matched elements is computed in this context, and the resulting code, MC64, is widely used. Hogg and Scott (2013) discuss several pivoting strategies, including matching-based orderings, to solve symmetric, indefinite systems of equations.

As parallel computers are able to solve systems of linear equations with millions of rows and columns, one challenge that has remained here is that the primal–dual algorithm used to compute perfect matchings does not have much concurrency. Hogg and Scott (2015) have employed the auction algorithm on shared-memory machines to compute the matching in parallel, but scaling to the large numbers of cores on distributed-memory machines remained an open problem. This raised the possibility that approximation algorithms for matching could be employed in this context. In recent work Azad *et al.* (2018) have described an approach that scales to 17 000 cores, and we consider this next.

Azad *et al.* (2018) first compute a perfect matching in the bipartite graph and then seek to increase its weight. (A perfect matching must exist if the matrix is non-singular.) The perfect matching is computed by a distributed-memory parallel algorithm that recasts the problem in terms of matrix–vector products in a suitable semi-ring using the Graph-BLAS operations (Azad and Buluç 2016). The increase in weight is accomplished by means of the $(2/3 - \epsilon)$-approximation algorithm of Pettie and Sanders (2004), which can be simplified here because the graph is bipartite and the matching is perfect. Hence they search for weight-increasing cycles with four edges (see Figure 3.1), finding for each vertex a cycle that leads to the largest increase in matching weight. From the set of cycles obtained, a maximal set of vertex-disjoint cycles is chosen using a greedy algorithm, and the matching weight is increased without affecting the cardinality of the perfect matching. The approximation guarantee of this algorithm is $(2/3 - \epsilon)$ and it is computed in $O(m \log \epsilon^{-1})$ time.

In a distributed-memory parallel implementation, it is not easy to find a maximal set of vertex-disjoint cycles without extensive communication, and hence these authors instead use only local comparisons to obtain a heavy set of vertex-disjoint cycles. Thus the parallel algorithm does not have the approximation guarantee, but it computes a heavy-weight matching that performs well in practice; it runs in $O((m/p)(1 + \beta) + \alpha p)$ time, where p is the number of processors, α is the latency and β is the inverse bandwidth. The authors report results for the algorithm on distributed-memory

machines with 17 000 cores. Their code has been incorporated into the distributed-memory SuperLU solver for sparse, unsymmetric, linear systems of equations (Li and Demmel 2003). Additional details on the applications of matchings in sparse matrix algorithms is provided in Duff and Uçar (2012).

Exact algorithms for maximum cardinality matchings and maximum edge-weighted matchings in bipartite and non-bipartite graphs are available in the LEDA library (Mehlhorn and Näher 1999). We compare the performance of approximation algorithms for vertex- and edge-weighted matchings with the LEDA codes in Section 5.

3.5. Matching software

In Table 3.5 we tabulate software libraries available for several matching problems. Software for b-matching, edge cover and b-edge cover will be discussed in their respective sections. An earlier discussion of matching algorithms and software is provided by Burkard *et al.* (2009).

The MC64 code in the Harwell Subroutine Library (HSL) computes a maximum product edge-weighted perfect matching in a bipartite graph in order to place large elements on the diagonal of a sparse, square matrix (Duff and Koster 2001). The primal–dual Hungarian algorithm is employed with a logarithmic transformation of the weights to compute a maximum product matching. Code for computing a maximum sum edge-weighted perfect matching is also included. This code is widely used in the preprocessing step in solving sparse systems of linear equations.

LEDA (Mehlhorn and Näher 1999) has codes for computing maximum edge-weighted matchings in both bipartite graphs and non-bipartite graphs. In our experience, this code is the fastest of the exact matching codes available for this problem. The algorithm uses a fractional matching initialization for non-bipartite graphs, which speeds it up over a greedy initialization. It is commercial software. The LEMON library (Dezső, Jüttner and Kovács 2011) contains an exact maximum edge-weighted matching code. BLOSSOM V is another code for minimum weight perfect matching in non-bipartite graphs by Kolmogorov (2009).

The colleagues and students of the first author have created a software library called Matchbox, which contains a number of exact and approximation algorithms for several variants of the matching problem. Included are exact algorithms for maximum cardinality matching in bipartite and non-bipartite graphs, and a 1/2-approximation algorithm for bipartite graphs. For edge-weighted matching, it has exact algorithms for maximum edge weight, perfect matching with maximum edge weight, and maximum weight semi-perfect matching, all for bipartite graphs. It also has a 1/2-approximate greedy algorithm for both bipartite and non-bipartite graphs. For vertex-weighted matchings, it has exact algorithms for both bipartite and non-

Table 3.5. Matching software libraries.

Library/Code	URL	Algorithms included
HSL/MC64	http://www.hsl.rl.ac.uk/catalogue/mc64.html	Max. product (and sum) edge-weighted perfect matching in bipartite graphs
Blossom V	http://pub.ist.ac.at/~vnk/software.html	Min. edge-weighted perfect matching
LEDA	http://www.algorithmic-solutions.com/leda	Max. edge-weighted matching in bipartite and general graphs
LEMON	http://lemon.cs.elte.hu	Max. edge-weighted matching in general graphs
Matchbox	https://github.com/CSCsw/matchbox	*Cardinality* Max. matching in general graphs; 1/2-approximate matching in bipartite graphs *Edge weight* Max. edge weight, perfect matching with max. edge weight, and max. edge-weighted semi-perfect matching, all in bipartite graphs; 1/2-approximate edge-weighted matching (greedy) for general graphs *Vertex weight* Max. matching for general graphs; 2/3-, $(1-\epsilon)$- and variants of 1/2-approximation algorithms (Greedy, Locally Dominant Edge, Path Growing-DP, Suitor) for max. matching in general graphs

Table 3.5 continued.

Library/Code	URL	Algorithms included
SUITOR	http://www.ii.uib.no/~fredrikm/matching/	Several 1/2-approximation edge-weighted matching algorithms including SUITOR
Matchmaker	http://people.sabanciuniv.edu/kaya/software.html	Ten variant bipartite max. cardinality matching algorithms
Matchmaker2	http://tda.gatech.edu/software/matchmaker2/	GPU code for bipartite max. cardinality matching
MS-BFS-GRAFT	https://bitbucket.org/azadcse/ms-bfs-graft	Multi-threaded code for 1/2-approximate bipartite max. cardinality matching
CombBLAS 2.0	https://bitbucket.org/berkeleylab/combinatorial-blas-2.0, directory CombBLAS/Applications/BipartiteMatchings/	Distributed-memory algorithms for bipartite matching (max. cardinality, 1/2-approximate cardinality, heavy-weight perfect matching) via matrix vector multiplication in a semi-ring

bipartite graphs. The collection of approximation algorithms includes 2/3-, $(1-\epsilon)$- and several variants of 1/2-approximation algorithms (Greedy, Locally Dominant Edge, Path Growing-DP, Suitor), for both bipartite and non-bipartite graphs. (The bipartite graph algorithms for vertex-weighted matching make use of the Mendelsohn–Dulmage theorem to make them more efficient.)

Codes for several 1/2-approximation algorithms for edge-weighted matching, including the Suitor algorithm, will be available from the third author's website. Included are Suitor, Local Max, Locally Dominant Edge, Path Growing, Path Growing-DP, Global Paths Algorithm and M1M2 algorithms. Multi-threaded parallel implementations of some of these algorithms are also included.

Implementations of ten variants of exact algorithms for maximum cardinality matching in bipartite graphs are provided by Kamer Kaya in a code called Matchmaker. The algorithms are based on breadth-first search, depth-first search and push–relabel techniques. MATLAB interfaces to the codes are also included. A GPU code for this problem, Matchmaker2, is also available (Deveci, Kaya, Uçar and Çatalyürek 2013).

A parallel multi-threaded code for computing 1/2-approximate maximum cardinality matchings in bipartite graphs has been provided by Azad. Distributed-memory parallel codes for exact maximum cardinality matching, 1/2-approximate cardinality matching and heavy-weight perfect matching, all in bipartite graphs, are also available. These codes employ Combinatorial BLAS (matrix operations in a suitably defined semi-ring) (Buluç and Gilbert 2011) to compute the matchings.

4. The b-matching problem

4.1. Background on b-matching

We turn to a generalization of the matching problem. Given a graph $G = (V, E)$ and a function $b(.)$ that maps each vertex to a natural number, a b-matching is a subset of edges M such that at most $b(v)$ edges in M are incident on each vertex v. We assume that $b(v) \leq \deg(v)$ for all v. If $b(v)$ is identically equal to one, then we have the matching problem. If the edges have weights w, then the weight of a b-matching is the sum of the weights of the matched edges, and we seek to compute a b-matching of maximum weight. We denote $b(V) = \sum_{v \in V} b(v)$ and $\beta = \max_{v \in V} b(v)$.

An exact algorithm for a maximum weight b-matching was first designed by Edmonds (1965), and Pulleyblank (1973) later gave a pseudo-polynomial time algorithm with complexity $O(mnb(V))$. Anstee (1987) 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. Derigs and Metz (1986) and Miller and Pekny (1995) improved the Anstee algorithm further. Padberg and Rao (1982) developed another algorithm using the branch and cut approach, and Grötschel and Holland (1985) solved the problem using the cutting plane technique. A survey of exact algorithms for b-matchings was provided by Müller-Hannemann and Schwartz (2000). More recently, Huang and Jebara (2011) 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.

We now describe work on approximate b-matching. Mestre (2006) 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 weight b-matching. We describe his proof in the next subsection. Mestre also generalized the path-growing algorithm of Drake and Hougardy (2003*b*) 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 (Khan *et al.* 2016*b*). Since the Path Growing algorithm is inherently sequential, it is not a good candidate for parallelization. Additionally, Mestre (2006) generalized a randomized algorithm for matching to obtain a $(2/3 - \epsilon)$-approximation algorithm with expected run time $O(\beta m \log(1/\epsilon))$. De Francisci Morales, Gionis and Sozio (2011) 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 (Koufogiannakis and Young 2011, Manshadi *et al.* 2013), but these methods are orders of magnitude slower than the b-Suitor algorithm. Georgiadis and Papatriantafilou (2013) have developed a distributed algorithm based on adding locally dominating edges to the b-matching, which leads to a Locally Dominant Edge algorithm. Our recent work on b-matching algorithms have focused on developing the Locally Dominant Edge and b-Suitor algorithms, and implementing them efficiently on serial computers and shared-memory and distributed-memory multiprocessors (Khan *et al.* 2016*a*, 2016*b*). We will discuss the b-Suitor algorithm later in this section.

4.2. Half-approximation algorithms for b-matching

4.2.1. The Greedy *algorithm*

Algorithm 8 describes the Greedy algorithm for computing a b-matching in a graph G.

The Greedy algorithm is a 1/2-approximation algorithm for the maximum weight b-matching problem. We prove this by an argument that employs matroid theory due to Mestre (2006).

Algorithm 8 GREEDY$(G(V, E, w, b))$

```
Sort edges in non-increasing order of weights
M = ∅
for e = (u, v) ∈ E in order do
    if M + e is a b-matching then
        M = M + e
    end if
end for
return M
```

Let E be a set of elements, and $\mathcal{I}$ a collection of subsets of E. The tuple $(E, \mathcal{I})$ is a *matroid* if

(i) for all $A \subseteq B$, if $B \in \mathcal{I}$, then $A \in \mathcal{I}$, and

(ii) for all $A, B \in \mathcal{I}$ with $|A| < |B|$, there exists an element $x \in B \setminus A$ such that $A + x \in \mathcal{I}$.

The sets in $\mathcal{I}$ are called the *independent* sets of the matroid, and by the first property, the empty set is independent. Maximal independent sets are called the bases of the matroid, and by the second property, all bases have the same cardinality. The reader unfamiliar with matroids may find two examples helpful. If E is the set of columns of a matrix, and independence corresponds to linear independence in a vector space, then we have a matric matroid. If E is the set of edges of a connected undirected graph without loops, and a subset of edges is defined to be independent if the edges do not induce a cycle, then we have a graphic matroid. A basis here corresponds to a spanning tree of the graph.

Assign every element in E a non-negative weight, define the weight of a set as the sum of the weights of its elements, and consider the problem of finding an independent set of largest weight. The GREEDY algorithm begins with the empty set, and at each step adds an element of largest weight if its addition will preserve independence, and rejects it otherwise. The GREEDY algorithm finds an independent set (a basis) of maximum weight if and only if the tuple $(E, \mathcal{I})$ is a matroid.

It is well known that a matching does not correspond to a matroid, but a matching in a bipartite graph is the intersection of two matroids. We now consider a more general system called a k-extendible system related to a matroid. A k-extendible system is a tuple $(E, \mathcal{I})$ that satisfies property (i) of a matroid, but property (ii) is replaced by

(ii′) for all $A \subseteq B \in \mathcal{I}$ and $x \in E$, if $A + x \in \mathcal{I}$ but $B + x \notin \mathcal{I}$, then there exists $Y \subseteq B \setminus A$, such that $B \setminus Y + x \in \mathcal{I}$ and $|Y| \leq k$.

In words, this means that if we can augment a smaller independent set A with a new element x and preserve its independence, then it should be possible to remove a subset of size at most k from any superset of A that is independent, add x to the new set and still preserve independence. Maximal independent sets do not necessarily have unique maximum cardinality for k-extendible systems.

As a concrete example, let us consider an undirected graph $G = (V, E, w)$ with vertex set V, edge set E and non-negative weights on its edges W. Further, we are given a set of natural numbers $b(v) \leq \deg(v)$ for each $v \in V$. Let the collection of independent sets $\mathcal{I}$ be defined as subsets of edges that consist of a b-matching in G, *i.e.* subsets of edges M such that $\deg_M(v) \leq b(v)$ for all $v \in V$. Let A be a b-matching in G and let $B \supset A$ be another b-matching. Let $x = (u, v)$ be an edge that could be added to A but not to B to obtain a larger b-matching. The reason that this edge cannot be added to B is that the values of one or both of $b(u)$ and $b(v)$ would be exceeded. By removing one edge incident on u and another edge incident on v from B, we would be able to add the edge (u, v) to B and preserve a b-matching. Thus we have shown that a b-matching is a 2-extendible system.

We now provide some concepts and prove a preliminary lemma in order to prove that the Greedy algorithm computes a 1/2-approximation algorithm. We prove a more general result for k-extendible systems. An *extension* of an independent set $A \in \mathcal{I}$ is a superset B of A with $B \in \mathcal{I}$. We denote an extension of maximum weight of a set A by $\mathrm{OPT}(A)$.

Let the Greedy algorithm choose elements $x_1, x_2, \ldots, x_l$ on a maximization problem on a k-independence system $(E, \mathcal{I})$. We denote the set of chosen elements at the end of the ith iteration of the Greedy algorithm by S_i, with $S_0 = \emptyset$ and $S_i = S_{i-1} + x_i$. Note that $\mathrm{OPT}(\emptyset)$ is the optimal solution to the maximization problem, and $\mathrm{OPT}(S_l) = S_l$, since the set S_l is maximal. Furthermore, we have

$$w(\mathrm{OPT}(S_0)) \geq w(\mathrm{OPT}(S_1)) \geq \cdots \geq w(\mathrm{OPT}(S_l)),$$

since with increasing index i, S_i has fewer extensions available than S_{i-1}.

Lemma 4.1. If $(E, \mathcal{I})$ is a k-extendible system, then the element x_i chosen by the Greedy algorithm at the ith step satisfies

$$w(\mathrm{OPT}(S_{i-1})) \leq w(\mathrm{OPT}(S_i)) + (k-1)\, w(x_i).$$

Proof. By the definition of k-extendibility, $S_{i-1} \in \mathcal{I}$ and $\mathrm{OPT}(S_{i-1}) \in \mathcal{I}$. Now since the Greedy algorithm chooses the element x_i at the ith step, it does not belong to S_{i-1}, but it could belong to $\mathrm{OPT}(S_{i-1})$. If it does, then $\mathrm{OPT}(S_{i-1}) = \mathrm{OPT}(S_i)$, and the lemma holds trivially. Hence consider the situation when the element $x_i \notin \mathrm{OPT}(S_{i-1})$. By k-extendibility, there exists $Y \in \mathrm{OPT}(S_{i-1}) \setminus S_{i-1}$ such that $\mathrm{OPT}(S_{i-1}) \setminus Y + x_i \in \mathcal{I}$, where

$|Y| \leq k$. We have the relationships

$$\begin{aligned} w(\mathrm{OPT}(S_{i-1}) &= w(\mathrm{OPT}(S_{i-1}) \setminus Y + x_i) + w(Y) - w(x_i), \\ &\leq w(\mathrm{OPT}(S_i)) + w(Y) - w(x_i). \end{aligned}$$

The first line of the equation above follows since Y belongs to $\mathrm{OPT}(S_{i-1})$ but x_i does not. The second follows because $\mathrm{OPT}(S_{i-1}) \setminus Y + x_i$ is an extension of the set $S_i = S_{i-1} + x_i$ by the choice of Y, and $\mathrm{OPT}(S_i)$ is an extension of maximum weight of the set S_i.

Now we show that any element $y \in Y$ is not heavier than the element x_i, *i.e.* $w(y) \leq w(x_i)$. Assume for the sake of contradiction that $w(y) > w(x_i)$. If y is heavier than x_i, then since $y \notin S_{i-1}$ by the choice of Y, the element y must have been considered by the Greedy algorithm before x_i was included in S_i, but was not included in the greedy solution. Thus there exists $j \leq i$ such that $S_j + y \notin \mathcal{I}$, but $S_j + y \subseteq \mathrm{OPT}(S_{i-1}) \in \mathcal{I}$. But this contradicts property (i) of a k-independence system. Thus $w(y) \leq w(x_i)$, and since all weights are positive, $w(Y) \leq k w(x_i)$. □

Theorem 4.2 (Mestre). If $(E, \mathcal{I})$ is a k-extendible system, then the Greedy algorithm is a $1/k$-approximation algorithm for computing an independent set of maximum weight.

Proof. Let $\{x_i : i = 1, 2, \ldots, l\}$ be the elements chosen by the Greedy algorithm, and the corresponding sets of chosen elements be denoted by $\{S_i : i = 1, 2, \ldots, l\}$. We apply Lemma 4.1 l times, beginning with the empty set S_0, to get

$$\begin{aligned} w(\mathrm{OPT}(S_0)) &\leq w(\mathrm{OPT}(S_1)) + (k-1)\, w(x_1) \\ &\leq w(\mathrm{OPT}(S_l)) + (k-1) \sum_{i=1}^{l} w(x_i) \\ &= w(S_l) + (k-1)\, w(S_l) = k\, w(S_l). \end{aligned}$$

The first term in last line of the equation follows from the fact that the set S_l is maximal, and the second term is obtained by the summing the weight of the elements in S_l. Since $\mathrm{OPT}(S_0)$ is an independent set of maximum weight, we obtain the $1/k$-approximation property of the Greedy algorithm. □

4.2.2. The b-Suitor algorithm

The *b*-Suitor algorithm, designed by Khan *et al.* (2016*b*), computes a 1/2-approximate *b*-matching; indeed it computes a matching identical to the one obtained by the Greedy algorithm, provided ties in weights are broken consistently in both algorithms. This algorithm is based on proposals, much like the algorithms for the stable marriage problem and its variants (here the

stable fixtures problem), and is a generalization of the Suitor algorithm. Vertices can propose to their heaviest neighbours, and these proposals may be reciprocated or annulled by other vertices. Two vertices are matched when both propose to each other. Unlike the Greedy algorithm, the b-Suitor algorithm does not need to process edges in non-increasing order of weights; instead, it can process vertices in any order, although each vertex has to make proposals to its available neighbours in non-increasing order of weights.

The pseudo-code for the b-Suitor algorithm is described in Algorithm 9. It maintains a priority queue S to track the proposals made in the algorithm. The queue $S(u)$ consists of the suitors of a vertex u, *i.e.* those neighbours of u that currently have an active proposal to u. The operation $S(u).insert(v)$ adds a vertex v to the priority queue $S(u)$, and $S(u).remove(v)$ removes it. The value $r(u)$ is the number of proposals that a vertex u currently has made to its neighbours. We keep track of the lowest weight of a proposal received by a vertex u (made by a suitor of u) in $S(u)$.last. If u has received fewer than $b(u)$ proposals, this value is NULL.

In each iteration of the outer **while** loop, the algorithm processes vertices that have their current $b(.)$ values unsatisfied, and makes the requisite number of proposals to satisfy the $b(.)$ values; these vertices are stored in a set Q. During the iteration it collects vertices whose $b(.)$ values may be decremented in a set Q' so that they could be processed in the next iteration. For each vertex $u \in Q$, while its $b(u)$ value is not satisfied and adj(u) has not been exhaustively searched, the algorithm finds eligible neighbours to propose to. A neighbour v is an *eligible neighbour* of u if it holds fewer than $b(v)$ proposals, or u can beat the lowest offer that v currently has, which is $S(v)$.last. If u cannot beat this offer, then it considers its next heaviest neighbour, and this prevents proposals being extended which at the current stage of the algorithm have no chance of success, saving work. But if v has fewer than $b(v)$ proposals, or if u can beat the lowest offer that v has, then u proposes to v, and becomes a suitor of v. In the latter case, u annuls the lowest-weight proposal of v, say from a vertex y, and y has to make another proposal in the next iteration. The value of $S(v)$.last is also updated.

Algorithm 9 shows that when a proposal made by a vertex y is annulled, it is placed into a queue to be processed in the next iteration. This corresponds to the Gale–Shapley algorithm for stable marriage which processes proposals in rounds. Instead, one could consider the vertex y making a proposal immediately, and this would correspond to the McVitie–Wilson algorithm. It is the possibility of proposals being annulled that permits vertices to be processed in any order, thus increasing the concurrency in the algorithm. We have shown that if the edge weights are chosen uniformly at random, then the total expected work in the algorithm is linear in the number of edges in the graph, so that the algorithm does not create a lot of unnecessary work

Algorithm 9 b-Suitor algorithm

Input: Graph $G = (V, E, w, b)$.
Output: A $1/2-$approximate edge-weighted b-matching M.
Data Structures: Q is the set of vertices that propose in each iteration of the outer **while** loop, and Q' is the set of vertices that need to make proposals in the next iteration. $S(u)$ is the set of suitors of u, $r(u)$ is the number of outstanding proposals that u currently has made.

```
1: procedure b-Suitor(G, b)
2:     Q = V; Q' = ∅;
3:     Initialize array S to be empty and r to zero;
4:     while Q ≠ ∅ do
5:         for vertices u ∈ Q in any order do
6:             while r(u) < b(u) and adj(u) ≠ exhausted do
7:                                                  ▷ Make a proposal from u
8:                 Let v ∈ N(u) be the heaviest eligible neighbour of u;
9:                 if v ≠ NULL then
10:                                                 ▷ Make u a suitor of v
11:                    Insert u into S(v);
12:                    r(u) = r(u) + 1;
13:                    if u annuls the proposal of a vertex y then
14:                        Remove y from S(v);
15:                        Add y to Q'; r(y) = r(y) − 1;
16:                    end if
17:                else adj(u) = exhausted;
18:                end if
19:            end while
20:        end for
21:        Q = Q'; Q' = ∅;
22:    end while
23: end procedure
```

(Khan *et al.* 2018*b*). This paper also shows that the depth (the length of the critical path) is $O(\log m \log \Delta)$.

4.3. Implementation and applications of b-matchings

The b-Suitor algorithm was implemented on serial, shared-memory and distributed-memory computers, and it is currently the fastest practical algorithm that we know of (Khan *et al.* 2016*a*, Khan *et al.* 2016*b*). It has been compared with the Greedy algorithm and the Locally Dominant Edge algorithm (LDE), and it outperforms both these algorithms with regard to run times. All of these algorithms compute the same b-matching provided

ties in weights are broken in a consistent manner. The b-Suitor algorithm has the desirable property that the parallel algorithms and the serial algorithm compute the same b-matching. This algorithm was scaled to more than 12 000 cores on a distributed-memory parallel computer. An implementation on GPUs was reported by Naim and Manne (2018).

b-matchings have been applied to a number of problems such as finite element mesh refinement (Müller-Hannemann and Schwartz 2000), median location problems (Tamir and Mitchell 1998), spectral data clustering (Jebara and Shchogolev 2006), *etc.* An important application of b-matching is in graph-based semi-supervised machine learning, where a b-matching has been used to replace the well-known k-nearest neighbour graph construction (Jebara, Wang and Chang 2009). Both of these constructions are discussed in this context by Subramanya and Talukdar (2014). We will discuss this matter in more detail when we consider applications of b-edge cover, but we state here that an approximate b-matching construction reduces the time complexity of this approach from $O(b(V)m \log n)$ to $O(m \log \beta)$, without any discernible loss in quality in the classification. Recently, Choromanski, Jebara and Tang (2013) used b-matching to solve a data privacy problem called adaptive anonymity. Again, we will discuss this problem as an application of b-edge cover.

An implementation of b-Suitor is available at https://github.com/ExaGraph. Parallel versions of this code will be included in this repository.

5. Vertex-weighted matching

We consider a variant of the matching problem that has not been well studied. We are given a graph $G = (V, E)$, and a weight function $w : V \mapsto \mathbb{R}_{\geq 0}$ that assigns non-negative real-valued weights to vertices. The weight of a matching in G is now the sum of the weights on the matched vertices. The problem is to compute a matching of maximum (vertex-) weight in the graph G (MVM).

By summing the weights on the endpoints of an edge and assigning it to the edge, we can transform the vertex-weighted matching problem into an edge-weighted matching problem. So at first blush it appears that we can solve MVM problems with edge-weighted matching algorithms. But at least for exact algorithms, this can increase the run times of the edge-weighted matching algorithms by three or four orders of magnitude (Dobrian, Halappanavar, Pothen and Al-Herz 2018). Additionally the MVM problem has a rich structure that leads to simpler algorithms than those employed for the MEM problem. We can say that the MVM problem is closer to the maximum cardinality matching problem than the maximum edge-weighted matching problem. Dobrian *et al.* (2018) have designed both exact and approximation algorithms for this problem.

An MVM can be characterized in two different ways. The first characterization is in terms of augmenting paths and weight-increasing paths. An augmenting path is defined as earlier, a path that has alternating unmatched edges and matched edges, with one more unmatched edge than matched edges. By augmenting a matching using this path, we add the weights of the unmatched endpoints of the path to the matching, and hence the cardinality of the matching increases while the weight of the augmented matching cannot decrease since vertex weights are non-negative. A *reversing path* is an alternating path with an even number of edges that begins with a matched edge and ends with an unmatched edge. A reversing path is *weight-increasing* if the matched endpoint of the path has lower weight than the unmatched endpoint. In this case, by switching the matched and unmatched edges on the path we increase the weight of the matching. Dobrian *et al.* (2018) proved the following result.

Theorem 5.1. A matching M is an MVM if and only if there is neither an augmenting path nor a weight-increasing path with respect to M.

The second characterization is in terms of weight vectors. For any matching M, consider a weight vector which lists the weights of the matched vertices in non-increasing order. Now we can compare two matchings by comparing their weight vectors lexicographically. The following result may also be found in Dobrian *et al.* (2018).

Theorem 5.2. A matching M is an MVM if and only if its weight vector is lexicographically maximum among all weight vectors of matchings.

These results lead to two different exact algorithms for the MVM problem.

One of these algorithms processes vertices in non-increasing order of their weights, and from each unmatched vertex u searches for a heaviest unmatched vertex v it can reach by augmenting paths. If the augmenting path search is successful, then the matching is augmented. If it is not successful, we discard this unmatched vertex u (we will not find an augmenting path from u in the future steps of the algorithm). In both cases, we process the next heaviest unmatched vertex, terminating when we have processed or matched all the vertices. In this algorithm by the choice of vertices matched at each step, there will be no weight-increasing path, and it suffices to search for augmenting paths.

A second algorithm would first compute a maximum cardinality matching (of arbitrary weight) in the graph. Then we search for weight-increasing paths of even length from each unmatched vertex. If we succeed, then we switch matched to unmatched edges and *vice versa* on this path, and increase the matching weight. If we do not succeed, then we process the next unmatched vertex. Note that in this case, by construction, there cannot be an augmenting path. This algorithm is attractive practically for several

reasons. The first is that this algorithm does not need to process vertices in non-increasing order of weights, and hence there is more concurrency in this algorithm, making an implementation on a parallel computer feasible. The second is that it is attractive if we compute the Gallai–Edmonds decomposition (Lovász and Plummer 2009), since this decomposition identifies a subgraph that has a perfect matching in any maximum cardinality matching. We can remove this subgraph from further consideration, since all vertices in the subgraph are matched and every maximum cardinality matching has the same weight. Hence we need to run the MVM algorithm only on the residual graph.

The first of these algorithms was designed by Dobrian *et al.* (2018) and has time complexity $O(nm)$. Spencer and Mayr (1984) have designed an exact MVM algorithm with lower $O(m\sqrt{n}\log n)$ time complexity, which employs recursion to compute the matching. As far as we know, this algorithm has not been implemented, and it is not clear if it is practical.

An important reason for designing the exact algorithm discussed in the previous paragraph is that by restricting the length of the augmenting path to at most three, we may obtain a 2/3-approximation algorithm. This result was first obtained for bipartite graphs by Dobrian *et al.* (2018). Here one can split the MVM problem into two 'one-side-weighted' subproblems. This means given a bipartite graph $G = (V_1, V_2, E)$ in the first subproblem we ignore the weights on V_2, and in the second subproblem we ignore the weights on V_1. The advantage is that now we can find any augmenting path from an unmatched vertex in V_1 (V_2) for the first (second) subproblem. The vertices still need to be processed in non-increasing order of weights as in the exact algorithm, and we limit the search for augmenting paths to length at most three. Once the two matchings M_1 and M_2 are obtained, the Mendelsohn and Dulmage (1958) theorem can be invoked to find a matching M in which all the V_1 vertices (the weighted vertices) in M_1 and the V_2 vertices (again the weighted vertices) in M_2 are matched. The new matching M has the same weight as the sum of the matchings M_1 and M_2, and it is a 2/3-approximation to the maximum vertex-weighted matching. The time complexity of this algorithm is $O(m + n\log n)$, which is linear, except for the sorting step. While this algorithm is easy to state, its proof of correctness is somewhat involved. The proof works by finding, for each vertex that is not matched in the approximation algorithm but matched by the exact algorithm (failed vertices), two matched vertices in the approximate matching that are at least as heavy as the failed vertex.

Al-Herz and Pothen (2019) have extended this result to non-bipartite graphs. Here, since there is no bipartition of the vertices, we cannot invoke the Mendelsohn–Dulmage theorem, and new concepts about augmenting paths are needed. The 2/3-approximation algorithm is described in Algorithm 10; it processes vertices in non-increasing order of weights, and

Algorithm 10 Two-Third Approximate Matching ($G = (V, E, w)$)

Input: A graph G with weights w on the vertices.
Output: A 2/3-approximate vertex-weighted matching M.

```
1: M ← ∅; Q ← V
2: while Q ≠ ∅ do
3:     Let u be a heaviest vertex in Q
4:     Let v denote a heaviest unmatched vertex reachable from u by
           an augmenting path P of length at most three
5:     if P is found then
6:         M ← M ⊕ P; delete v from Q
7:     end if
8:     Delete u from Q
9: end while
10: return M
```

matches each vertex if possible to a heaviest unmatched vertex reachable by augmenting paths of length at most three. The time complexity of this algorithm is $O(m \log \Delta + n \log n)$. The proof of the approximation ratio involves charging the weight of each failed vertex to two distinct matched vertices in the approximate matching such that they have equal or higher weight relative to the failure. These vertices are found by examining the augmenting paths in the approximation algorithm, where we distinguish between a vertex from which an augmenting path search begins (an origin), and a vertex where it ends (a terminus).

Al-Herz and Pothen (2019) have implemented the 2/3-approximation algorithm for MVM and compare it with an exact maximum edge-weighted matching algorithm in LEDA (Mehlhorn and Näher 1999), the $(2/3 - \epsilon)$-approximation algorithms of Maue and Sanders (2007), the $(1-\epsilon)$-algorithm of Duan and Pettie (2014) and a greedy 1/2-approximation algorithm for MVM. The LEDA algorithm benefits from the fractional matching initialization; it is about twelve times faster relative to the LEDA algorithm without any initialization, on problems where both terminated. There are a number of problems where LEDA without initialization did not terminate in four hours. On problems where LEDA with fractional initialization terminated in at most four hours but where the algorithm with no initialization did not, we use the former algorithm as the baseline. The exact MVM algorithm (it does not use any initialization) was slower than the LEDA algorithm by a factor less than two. The $(1 - \epsilon)$-approximation algorithm with $\epsilon = 1/3$ was faster than LEDA by a factor of seven for these problems; the $(2/3 - \epsilon)$-approximation algorithm with a Global Paths Algorithm initialization was about nine times faster; the 2/3-approximate MVM was 140 times faster than LEDA; the 1/2-approximation algorithm for MVM

was 400 times faster. There was also a problem (nlpkkt200) where none of the exact algorithms terminated in four hours, but the 2/3-approximate MVM computed a matching in under a minute. In terms of weights, the 2/3-approximate MVM and the $(2/3-\epsilon)$-approximate MEM computed more than 99.99% of the maximum weight; the $(1-\epsilon)$-approximation algorithm was about 1% off the optimal, and the 1/2-approximate algorithm was 3% off the optimal.

Vertex-weighted matchings have been applied to the design of network switches (Tabatabaee, Georgiadis and Tassiulas 2001), and scheduling of astronaut training sessions (Bell 1994). More recently, online algorithms for vertex-weighted matchings in bipartite graphs have been surveyed by Mehta (2012) due to applications in internet advertising. Our interest in this problem was spurred by its application to computing sparse null space and column space bases of sparse, under-determined matrices (Coleman and Pothen 1987, Pinar *et al.* 2006).

6. The edge cover problem

An edge cover in a graph $G = (V, E)$ is a set of edges C such that there is at least one edge belonging to C incident on each vertex in V. The set of edges E is a trivial edge cover of the graph G. We consider the problem of finding an edge cover with minimum cardinality. The following relationship between a maximum matching and a minimum edge cover is due to Gallai (1959) and Norman and Rabin (1959), and is described in Theorems 19.1 and 19.2 of Schrijver (2003).

Theorem 6.1. Every maximum cardinality matching of a graph $G = (V, E)$ 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 $O(m)$ time, and *vice versa*.

The $O(m)$ time in the last part of Theorem 6.1 comes from a simple algorithm due to Norman and Rabin (1959). Given a maximum cardinality matching, we add the matched edges to the edge cover; additionally we add one of the edges incident on each unmatched vertex to the edge cover. This results in a minimum cardinality edge cover. Further, a minimum cardinality edge cover can be computed in $O(\sqrt{n}m)$ time, the time needed for a maximum cardinality matching.

Now we consider a weighted graph $G = (V, E, w)$, where $w : E \mapsto \mathbb{R}_{\geq 0}$ is a weight function that maps each edge to a non-negative weight. Recall that the weight of a set S is the sum of all the weights of the edges in S. We seek to minimize the weight of an edge cover of G. In the graph in

Figure 3.2, there are two minimum weight edge covers of weight 15: the set of edges $\{(a, e), (b, f), (c, d)\}$ and $\{(a, c), (a, e), (b, f), (d, f)\}$.

The simplest algorithm for computing a minimum weight edge cover selects an edge of minimum weight incident on each vertex and adds it to the cover. This Nearest Neighbour algorithm (and many other algorithms that we consider) does not compute a minimal edge cover; that is, there could be redundant edges that decrease the weight of an edge cover when they are removed, while retaining the property of being an edge cover of the graph. We will show later in the context of the more general b-edge cover problem that this algorithm computes a 2-approximation to an edge cover of minimum weight, even without the removal of the redundant edges.

Next we describe an approximation-preserving reduction of the minimum weight edge cover problem to the maximum weight matching problem. The reduction is mentioned in Chapter 27 of Schrijver (2003), and Huang and Pettie (2017) proved that it is approximation-preserving. We proceed to describe this reduction next.

For each vertex $v \in V$ we define $\mu(v)$ to be the minimum weight of any edge incident on v. Now we compute a transformed weight w' for each edge $e = (u, v)$ 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 neighbouring 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 neighbouring 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 $e(v)$ denote an edge of minimum weight incident on v. We can compute an edge cover C as follows:

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

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 showed that this reduction is approximation-preserving.

Theorem 6.2. Let M be $(1-\epsilon)$-approximate matching obtained with the transformed weights w' from a graph $G = (V, E, w)$. Then the edges in M together with a set of minimum weight edges incident on the unmatched vertices constitute a $(1+\epsilon)$-approximate edge cover C of the graph G with respect to the original weights w.

Proof. Let C^* denote an optimal edge cover with respect to the weights w and let M^* denote an optimal matching with respect to the weights w'. Since we have shown that $w'(u,v) \leq w(u,v)$ for all edges (u,v), we have that

$$w'(M^*) = \sum_{(u,v)\in M^*} w'(u,v) \leq \sum_{(u,v)\in M^*} w(u,v) = w(M^*) \leq w(C^*).$$

We make use of this result in the following.

From the construction of C and the definition of M, we have

$$\begin{aligned} w(C) &= w(M) + \mu(V \setminus V(M)) \\ &= \mu(V(M)) - w'(M) + \mu(V \setminus V(M)) \\ &= \mu(V) - w'(M). \end{aligned}$$

Similarly we obtain

$$w(C^*) = \mu(V) - w'(M^*). \tag{6.1}$$

Making use of the approximation ratio of the matching algorithm that computed M, we obtain

$$\begin{aligned} w(C) &= \mu(V) - w'(M) \\ &\leq \mu(V) - (1-\epsilon)w'(M^*) \\ &= \mu(V) - w'(M^*) + \epsilon\, w'(M^*) \\ &= w(C^*) + \epsilon\, w'(M^*) \\ &\leq (1+\epsilon)\, w(C^*). \end{aligned}$$

In the third line we used (6.1), and in the fourth line we made use of the inequality from the first paragraph. This completes the proof. □

We can conclude that the time complexity of computing a minimum weight edge cover is the same as that of computing a maximum weight matching, $O(mn + n^2 \log n)$ (Gabow 2018). Furthermore, we can compute an approximate minimum weight edge cover using one of the approximate maximum weight matching algorithms.

We now turn to a reduction of an edge cover to a minimum weight perfect matching problem, described in Schrijver (2003). We make a second copy of the graph G which we call G'. We join each vertex v in G with its corresponding vertex v' in G' by an edge with weight set to twice the minimum weight edge incident on v. We call these latter edges the linking edges. By construction the doubled graph has a perfect matching. Furthermore, we can choose the matched edges in G' to correspond to copies of the matched edges in G. We replace each linking edge (v, v') in the matching by a minimum weight edge incident on the vertex v, and add it to the edge cover. To these edges we add the matched edges from G, to obtain a minimum weight edge cover of G. This reduction of an edge cover to a perfect matching was modified to compute a prize-collecting variant of the minimum weight edge cover (in which vertices might not be covered by paying a cost) by Azad *et al.* (2010) to compute a dissimilarity measure for high-dimensional proteomic data.

A third reduction to matching, also mentioned in Schrijver (2003), computes $b'(v) = \deg(v) - 1$, and then computes a b'-matching of maximum weight. A minimum weight edge cover is the set of edges complementary to the matching. We will discuss this algorithm in more detail when we consider the b-edge cover problem. We will also describe some computational results on the edge cover problem after we discuss the latter problem.

7. The b-edge cover problem

Given a graph $G = (V, E)$ and a function $b(v)$ that maps each vertex $V \in V$ to a natural number, a b-edge cover is a subset of edges C such that *at least* $b(v)$ edges in C are incident on v. We assume that $b(v) \leq \deg(v)$. If $b(v)$ is identically equal to one for all vertices v, then we have the edge cover problem. If the edges are weighted by a non-negative function $w(.)$, then the weight of a b-edge cover is the sum of weights of the edges in the cover. The problem we consider is to compute a b-edge cover of minimum weight. Unfortunately the weight transformation approach we used to compute a minimum weight edge cover from a maximum weight matching does not work for b-edge cover. An exact algorithm for the problem has polynomial time complexity, since it can be computed as the complement of a maximum weight b'-matching, where $b'(v) = \deg(v) - b(v)$ for all $v \in V$. We seek to design approximation algorithms that have near-linear time complexity and have more concurrency.

7.1. b-edge cover algorithms

The simplest algorithm that one could think of for the b-edge cover problem is for each vertex v to independently choose $b(v)$ edges to add to the cover. This is the b-Nearest Neighbour algorithm (bNN) described in Algo-

Algorithm 11 b-NEAREST NEIGHBOUR$(G = (V, E, w, b))$

1: $C = \emptyset$
2: **for** each $v \in V$ **do**
3: $\quad E_v = b(v)$ lightest edges incident on v
4: $\quad C = C \cup E_v$
5: **end for**
6: **return** C

rithm 11, and we show in Section 7.5 that this leads to a 2-approximation algorithm for the minimum weight b-edge cover problem.

Practically the weight of the edge cover could be reduced by removing redundant edges. An edge (u, v) is *redundant* if there are more than $b(u)$ edges from the cover incident on u and there are more than $b(v)$ edges from the cover incident on v. We say that such a vertex u or v is over-saturated; if there are exactly $b(u)$ edges incident on a vertex u it is saturated; and if fewer than $b(u)$ edges are incident on u (this can happen only during the computation of an edge cover since it violates the requirements of an edge cover), then it is unsaturated.

The next algorithm one could think of is a greedy algorithm, except that unlike the matching problem, here the weights have to be dynamically updated. This algorithm is inspired by a greedy algorithm for the set multicover problem, in which we are given a collection of subsets of a set, and we are required to choose subsets from the collection so that each element v in the set is covered $b(v)$ times. The GREEDY algorithm for the set cover problem is described by Chvatal (1979). The b-edge cover problem is a special case of the set multicover problem where the elements in the set correspond to vertices, and subsets to edges, which consist of pairs of vertices.

We define the *effective weight* of an edge as the weight of the edge divided by the number of its unsaturated endpoints. The GREEDY algorithm for minimum weight edge cover works as follows. Initially, no vertices are covered, and the effective weights of all the edges are half of the edge weights. 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 neighbouring 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 12.

We can see from an analysis of a PRIMAL DUAL algorithm for this problem (provided in Section 7.4) that this algorithm produces an edge cover whose

Algorithm 12 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
```

weight is at most 3/2 the minimum weight. The worst-case time complexity of the Greedy algorithm is $O(\beta m \log n)$.

The next algorithm we consider is 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 edges, which is the most expensive step in the algorithm, until they are 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 after the update, 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 (Minoux 1978). Its time complexity is $O(m \log n)$, which is better than the Greedy algorithm, and in practice it performs even better. We refer the reader to Ferdous, Pothen and Khan (2018) for more details on this algorithm.

Yet another variant of the Greedy algorithm is the Locally Subdominant Edge (LSE) algorithm. An edge is *locally subdominant* if it has minimum effective weight among all of its neighbouring edges. The LSE algorithm identifies locally subdominant edges, adds them to the cover, and updates the effective weight of its neighbouring edges. This process iterates until no unsaturated vertices remain. This algorithm was proposed by Khan *et al.* (2016*a*), and they proved that it is also a 3/2-approximation algorithm.

We briefly consider an algorithm that obtains a b-edge cover from the complement of a b'-matching. For each vertex v, define $b'(v) = \deg(v) - b(v)$, and then compute a b'-matching of maximum weight. The complement of

the matching is a b-edge cover of minimum weight. What is interesting is that if we use a 1/2-approximation algorithm that matches locally dominant edges such as the Greedy b'-matching algorithm or the b'-Suitor algorithm, then we can show that the resulting b-edge cover is a 2-approximation to a minimum weight b-edge cover. This algorithm, called the matching complement edge cover (MCE) algorithm, was designed by Khan *et al.* (2018*b*). The MCE algorithm works with static edge weights since it computes a matching, not an edge cover, and hence it has much more concurrency than the other algorithms described thus far except for the b-Nearest Neighbour algorithm. An important feature of the MCE algorithm is that it computes a maximal b'-matching, and hence the complement is a minimal b-edge cover, which implies that there are no redundant edges to remove.

There are interesting relationships among the algorithms that have been described in this section. The Greedy, Lazy Greedy and LSE algorithms compute the same b-edge cover of a graph if (1) ties in weights are broken consistently, and (2) redundant edges are removed greedily. By this we mean that we consider the subgraph induced by the over-saturated vertices, compute a maximum weight matching in this graph, and remove these edges as redundant. These results are obtained in Khan and Pothen (2016) and Ferdous, Pothen and Khan (2018). The Primal Dual algorithm we describe in the next section will also compute the same b-edge cover under the two conditions mentioned above. Since we prove that this last algorithm has the approximation ratio 3/2, we can conclude that all the algorithms mentioned in this paragraph have the same approximation ratio.

Finally we discuss the $(1+\epsilon)$-approximation algorithm for b-edge cover designed by Huang and Pettie (2017). One way to solve a minimum weight b-edge cover is to reduce it to a capacitated b-matching problem and then solve the latter problem (Schrijver 2003). Unfortunately this reduction is not approximation-preserving. However, the authors show that if a b-matching is constructed maintaining a certain approximate complementary slackness condition, then the complement graph, a b-edge cover, would also satisfy an approximate complimentary slackness condition, which results in the desired approximation preservation. They describe a linear time algorithm to find such a b-matching. This linear time scaling algorithm is a generalization of the approximation algorithm for maximum weight matching designed by Duan and Pettie. As a by-product they also obtain an $O(m\sqrt{b(V)})$-time algorithm for the cardinality version of these problems.

7.2. A linear programming formulation

Now we turn to a linear programming formulation of the b-edge cover problem to describe primal–dual algorithms for the b-edge cover problem. Using the primal–dual framework, we will describe a 3/2-approximation algorithm,

analyse the bNN algorithm and show that it is a 2-approximation algorithm, and describe a Δ-approximation algorithm.

We begin by describing the primal and dual linear programming (LP) formulations of the minimum weight b-edge cover problem. Recall that we consider a graph $G = (V, E, w, b)$, 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. Recall 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 $x(e) = 1$ 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 weight edge cover problem is as follows:

$$\min \sum_{e \in E} w(e)x(e), \text{ subject to } \sum_{e \in \delta(v)} x(e) \geq b(v), \text{ for all } v \in V,$$
$$x(e) \in \{0,1\}, \text{ for all } e \in E. \tag{7.1}$$

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

$$\min \sum_{e \in E} w(e)x(e), \text{ subject to } \sum_{e \in \delta(v)} x(e) \geq b(v), \text{ for all } v \in V,$$
$$x_e \geq 0, \ -x(e) \geq -1, \text{ for all } e \in E. \tag{7.2}$$

The final constraint, equivalent to $x(e) \leq 1$, 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:

$$\max \sum_{v \in V} b(v)y(v) - \sum_{e \in E} z(e),$$
$$\text{subject to } y(i) + y(j) - z(e) \leq w(e), \text{ for all } e = (i,j) \in E,$$
$$y(v), z(e) \geq 0, \text{ for all } v \in V \text{ and } e \in E. \tag{7.3}$$

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

For 1-edge cover, the upper bound constraints on the primal variables for the LP relaxation shown in (7.2), *i.e.* $x(e) \leq 1$ for all $e \in E$, are not necessary. If in the solution there is any $x(e) > 1$, we can change it to $x(e) = 1$, 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 $[0, 1]$. Consider the

dual problem, (7.3). The dual variable corresponding to each upper bound constraint on $x(e)$ in the primal is $z(e)$. 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 $z(e)$ variables in our analysis of the algorithm.

Let EC^* be the objective value of an *optimum* b-edge cover solution, EC_{LP} the objective value of an *optimum* solution of the relaxed LP shown in (7.2), and EC_{dual} the objective value of a *feasible* solution of the dual problem shown in (7.3). Then from linear programming and weak duality theory we have

$$EC_{dual} \leq EC_{LP} \leq EC^*. \tag{7.4}$$

7.3. 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 7.1 (paid in full). The algorithm finds these primal and approximate dual variables that maintain the equality of primal and approximate dual objective values, that is,

$$\sum_{e\in E} w(e)x(e) = \sum_{v\in V} b(v)y^a(v) - \sum_{e\in E} z^a(e). \tag{7.5}$$

Property 7.2 (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 α-approximation. Replacing $\mathbf{y^a}$ and $\mathbf{z^a}$ in (7.5), we have

$$\sum_{e\in E} w(e)x(e) = \alpha \cdot \left(\sum_{v\in V} b(v)y(e) - \sum_{e\in E} z(e)\right). \tag{7.6}$$

Since $\mathbf{y}$ and $\mathbf{z}$ are dual feasible, from (7.4) and (7.6) we have

$$\sum_{e\in E} w(e)x(e) = \alpha \cdot \left(\sum_{v\in V} b(v)y(v) - \sum_{e\in E} z(e)\right) \leq \alpha \cdot EC_{LP} \leq \alpha \cdot EC^*. \tag{7.7}$$

This proves the required α-approximation guarantee. We now show how to instantiate this hypothetical algorithm to obtain 3/2- and 2-approximation algorithms for b-edge cover.

7.4. A 3/2-approximation algorithm

Rajagopalan and Vazirani (1993) 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 Ferdous, Pothen and Khan (2018). Our aim is to formulate *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 edge 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_e to denote the endpoints that e covers, and hence $0 \leq |Q_e| \leq 2$. The set C includes the edges in the cover.
- The *effective weight* of an edge, eff_weight(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, 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, 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.
- We maintain a list $L(v)$ that consists of $r(v)$ edges of lowest effective weight incident on a vertex v. Only these edges will be considered for inclusion in the b-edge cover. If there are two or more vertices with the $r(v)$th lowest effective weight, this list removes the need to process the edges in non-decreasing order of effective weights. Recall that both the effective weights and the values of $r(v)$ are dynamically changing in the algorithm.

The output of the algorithm is a set of edges C. We can derive an integral primal solution from C by setting $x_e = 1$ for all $e \in C$ and $x_e = 0$ for all

Algorithm 13 PRIMAL DUAL($G = (V, E, w, b)$)

```
C = ∅
while there exists an unsaturated vertex do
    Call PRICE ASSIGNMENT(G(V, E, w, b),max_price)
    Call AUGMENT COVER (G(V, E, w, b), max_price, C, r)
end while
return C
```

Algorithm 14 PRICE ASSIGNMENT($G = (V, E, w, b)$, max_price)

```
for each v ∈ V do
    if v is unsaturated then
        max_price(v) = {eff_weight(e) : r(v)th lowest effective weight of
                        an edge incident on v}
        L(v) = r(v) edges of lowest effective weight incident on v
    end if
end for
```

$e \in E \setminus C$. 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

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

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 it is needed for the proof analysis. Hence we defer its definition till then.

The pseudocode of the algorithm is shown in Algorithm 13. 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_price(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.

We provide the pseudocode for the PRICE ASSIGNMENT phase in Algorithm 14.

The second phase of the algorithm, the AUGMENT COVER phase, adds vertices to the edge cover using the **max_price** information set by the first phase. The pseudo-code for the AUGMENT COVER phase is presented in Algorithm 15. 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_price(.) values of both of its endpoints, then it would be included in the cover.

Algorithm 15 Augment Cover($G = (V, E, w, b)$, max_price,price,C, r)

for each $e = (u, v) \in E$ **do**
  if u and v are both covered **then**
    Mark (u, v) as deleted
    Continue
  end if
  if u and v are both uncovered, $e \in L(u) \cap L(v)$, and
      eff_weight$(e) \leq \{$max_price(u), max_price$(v)\}$ **then**
    Set price(u, e) and price(v, e) to eff_weight(e)
    $C = C \cup (u, v)$
    Decrease $r(u)$ and $r(v)$ by 1
    Mark (u, v) as deleted
  else if only u is uncovered, $e \in L(u)$, and
      eff_weight$(e) \leq$ max_price(u) **then**
    Set price(u, e) to eff_weight(e)
    $C = C \cup (u, v)$
    Decrease $r(u)$ and $r(v)$ by 1
    Mark (u, v) as deleted
  else if only v is uncovered, $e \in L(v)$, and
      eff_weight$(e) \leq$ max_price(v) **then**
    Set price(v, e) to eff_weight(e)
    $C = C \cup (u, v)$
    Decrease $r(u)$ and $r(v)$ by 1
    Mark (u, v) as deleted
  end if
end for

Upon finding such an edge e the algorithm fixes the values of price(i, e) and price(j, e) to the value of eff_weight(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_price(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.)

Once the algorithm terminates, we have settled the price(v, e) values for each vertex and covering edge pair. We have also 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

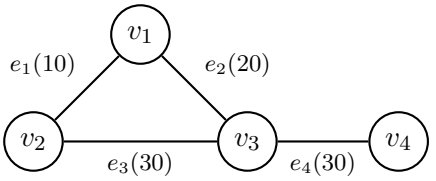

Figure 7.1. A small graph whose b-edge cover is to be computed.

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 $\text{max\_price}(v) = \max\{\text{price}(v, e) : e \in S_v\}$.

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 the algorithm, but they are needed for the proof analysis. We motivate this variable by means of a small example.

Example 7.3. Consider the graph with four vertices v_1, v_2, v_3 and v_4 in Figure 7.1, with the edge labels and weights also shown. The $b(v)$ value is 2 for each vertex v except v_4, 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 $\text{price}(v_1, e_1) = \text{price}(v_2, e_1) = 5$, $\text{price}(v_1, e_2) = \text{price}(v_3, e_2) = 10$, $\text{price}(v_2, e_3) = \text{price}(v_3, e_3) = 15$ and $\text{price}(v_3, e_4) = \text{price}(v_4, e_4) = 15$. The $\text{max\_price}(v)$ values are as follows: $\text{max\_price}(v_1) = 15$, $\text{max\_price}(v_2) = 10$, $\text{max\_price}(v_3) = 15$ and $\text{max\_price}(v_4) = 15$. In the next phase, suppose we scan through edges in the order e_1, e_2, e_3 and e_4. We select e_1 since the effective weight of this edge is less than the max_price(.) values for both endpoints. We decrease the $r(v_1)$ and $r(v_2)$ values by 1 and mark e_1 as deleted. Similarly we select e_2 and e_3. Note that v_1, v_2 and v_3 are saturated. We cannot add e_4 in this phase because the effective weight of e_4, which is now 30, is greater than $\text{max\_price}(v_4)$ which is 15. Next we start the second iteration. In the PRICE ASSIGNMENT phase, we set $\text{max\_price}(v_4)$ as 30 and in the AUGMENT COVER phase we select e_4, since the effective weight of e_4 equals $\text{max\_price}(v_4)$.

At the termination of the algorithm, the max_price values for our example are as follows: $\text{max\_price}(v_1) = 10$, $\text{max\_price}(v_2) = 15$, $\text{max\_price}(v_3) = 15$ and $\text{max\_price}(v_4) = 30$. Let us consider the dual constraints defined in (7.3). For e_1 the left side of the constraint using the approximate duals is $\text{max\_price}(v_1) + \text{max\_price}(v_2) - \text{excess}(e_1)$. But $\text{max\_price}(v_1) + \text{max\_price}(v_2) = 25$, which is much greater than the 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 $\text{excess}(e) = 0$. 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: $\text{excess}(e_1) = (10-5)+(15-5) = 15$, $\text{excess}(e_2) = (10-10)+(15-10) = 5$, $\text{excess}(e_3) = (15-15)+(15-15) = 0$ and $\text{excess}(e_4) = 30-30 = 0$. Observe that all of the dual constraints now become feasible except for e_4, where the left side of the constraint is $\text{max\_price}(v_3)+\text{max\_price}(v_4)-\text{excess}(e_4) = 15+30-0 = 45$, 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 7.4. The approximation ratio of the PRIMAL DUAL algorithm is 3/2.

Proof. First note that by construction $\text{max\_price}(v)$ is non-negative; since it is the maximum of the price values of incident edges of a vertex, $\text{excess}(e)$ is also non-negative. We need to show that by setting $\alpha = 3/2$, the paid in full and shrinking factor properties defined in Section 7.3 are maintained. Using the approximate duals, **max_price** and **excess**, we first show that the objective value of dual LP defined in (7.3) 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 $\text{excess}(e) = 0$ if e is not in the cover. Then we have

$$\begin{aligned}
\sum_{e \in C} \text{excess}(e) &= \sum_{e \in C} \sum_{q \in Q_e} (\text{max_price}(q) - \text{price}(q,e)) \\
&= \sum_{e \in C} \sum_{q \in Q_e} \text{max_price}(q) - \sum_{e \in C} \sum_{q \in Q_e} \text{price}(q,e) \\
&= \sum_{v \in V} b(v) \cdot \text{max_price}(v) - \sum_{e \in C} w(e). \qquad (7.8)
\end{aligned}$$

The second term in the last line of (7.8) 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 (7.3),

$$\begin{aligned}
&\sum_{v \in V} b(v) \cdot \text{max_price}(v) - \sum_{e \in E} \text{excess}(e) \\
&\quad = \sum_{v \in V} b(v) \cdot \text{max_price}(v) - \sum_{v \in V} b(v) \cdot \text{max_price}(v) + \sum_{e \in C} w(e) \\
&\quad = \sum_{e \in C} w(e). \qquad (7.9)
\end{aligned}$$

But we cannot substitute **max_price** for **y** and **excess** for **z** because these are not dual feasible. Define $\alpha \equiv 3/2$, and set $\mathbf{y} = \mathbf{max\_price}/\alpha$ and $\mathbf{z} = \mathbf{excess}/\alpha$. We 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 (7.3),

$$\begin{aligned}
&\frac{1}{\alpha} \cdot (\text{max_price}(i) + \text{max_price}(j) - (\text{max_price}(i) - \text{price}(i, e)) \\
&\qquad - (\text{max_price}(j) - \text{price}(j, e))) \\
&\quad = \frac{1}{\alpha}(\text{price}(i, e) + \text{price}(j, e)) \leq \frac{1}{\alpha} w(e) \leq w(e). \qquad (7.10)
\end{aligned}$$

The last line follows from the fact that during the algorithm we maintain $\text{price}(i, e) + \text{price}(j, e) = w(e)$.

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 (7.3) becomes

$$\begin{aligned}
&\frac{1}{\alpha} \cdot (\text{max_price}(i) + \text{max_price}(j) - (\text{max_price}(i) - \text{price}(i, e))) \\
&\quad = \frac{1}{\alpha}(\text{max_price}(j) + \text{price}(i, e)). \qquad (7.11)
\end{aligned}$$

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

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

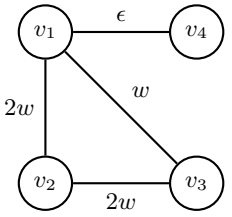

Figure 7.2. A tight example for the PRIMAL DUAL algorithm.

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$(i) \leq w(e)/2$ and max_price$(j) \leq w(e)$. We have

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

We combine the analysis as follows:

$$\begin{aligned}
&\sum_{e \in C} w(e) \\
&\quad = \sum_{v \in V} b(v) \cdot \text{max_price}(v) - \sum_{e \in E} \text{excess}(e) \quad \text{(from (7.9))} \\
&\quad = \alpha \cdot \left(\sum_{v \in V} b(v) y(v) - \sum_{e \in E} z(e) \right) \quad \text{(replacing } \textbf{max_price} \text{ and } \textbf{excess}\text{)} \\
&\quad = \alpha \cdot EC_{dual} \leq \alpha \cdot EC_{LP} \leq \alpha \cdot EC^* \quad \text{(from (7.6))}.
\end{aligned}$$

This gives the 3/2-approximation ratio. □

The approximation ratio is tight, and a tight example is the graph shown in Figure 7.2. Suppose $b = 1$ for each vertex. In the first iteration the PRIMAL DUAL algorithm will add (v_1, v_4) to the cover, and it cannot 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 (v_1, v_3), (v_1, v_2) or (v_2, v_3). Suppose it chooses the edge (v_1, v_3). Then to cover the vertex v_2, it has to choose either (v_1, v_2) or (v_2, v_3), resulting in a cover with weight $3w + \epsilon$, whereas the optimal weight is $2w + \epsilon$. 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 13, which are as follows.

- In the Augment Cover phase, when an edge is selected we mark all the neighbouring vertices of its covered endpoints as potential vertices. During the Price Assignment phase we need only update the max_price values of the potential vertices.
- In 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 7.5. The time complexity of Algorithm 13 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 $e = (i, j)$ can be marked at most $\deg(i) + \deg(j) = O(\Delta)$ times. Summing over m edges we obtain $O(\Delta m)$. Hence the time complexity is $O(\beta\,\Delta m)$. □

7.5. A 2-approximation algorithm

We have presented in Algorithm 11 the b-Nearest Neighbour algorithm for finding a b-edge cover. It is similar to the popular and well-known k-nearest neighbour 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 prove this result using the dual fitting framework developed in Section 7.3.

Lemma 7.6. The approximation ratio of the b-Nearest Neighbour algorithm is 2.

Proof. Let $\mathbf{x}$ be the primal integral solution and C a b-edge cover computed by the algorithm; hence $x(e) = 1$ if $e \in C$, and otherwise $x(e) = 0$. Let S_v denote the set of the $b(v)$ lightest edges incident on v. We define a price value for each covering vertex and edge pair, and consider an edge $e = (i, j) \in C$.

If the weight of the edge e is among the lightest $b(i)$ edges incident on the vertex i *and* the lightest $b(j)$ edges incident on j, then we set $\text{price}(i,e) = \text{price}(j,e) = w(e)/2$. In this case we say the edge e covers both of its endpoints. Otherwise if e is only among the lightest $b(i)$ ($b(j)$) edges incident on i (j), we assign $\text{price}(i,e) = w(e)$ ($\text{price}(j,e) = w(e)$). In this case the edge e covers only one of its endpoints. Next we set the approximate dual variables. We define for each vertex v, $\text{max\_price}(v) = \max_{e \in S_v} \text{price}(v,e)$. For each edge $e \in C$, let Q_e denote its covered endpoints. Note that Q_e may contain one or two vertices. We define for each $e \in C$, $\text{excess}(e) = \sum_{q \in Q_e} (\text{max\_price}(q) - \text{price}(q,e))$. If an edge e is not included in the cover then we set $\text{excess}(e) = 0$. Note that since price values of a vertex and edge pair are always non-negative, max_price is non-negative. Again, as $\text{max\_price}(v) \geq \text{price}(v,e)$ for all $e \in S_v$, the excess variable is also non-negative.

We now show that with $\alpha = 2$, the two properties mentioned in Section 7.3 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 7.4.

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

In the first scenario e belongs to the cover. Then replacing $y(v)$ and $z(e)$ on the left side of the first constraint of (7.3), we obtain

$$y(i) + y(j) - z(e) = \frac{1}{\alpha}(\text{max\_price}(i) + \text{max\_price}(j) - \sum_{q \in Q_e} (\text{max\_price}(q) - \text{price}(q,e))). \tag{7.12}$$

We have two cases to consider.

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

$$\begin{aligned} y(i) + y(j) - z(e) &= \frac{1}{\alpha}((\text{max\_price}(i) + \text{max\_price}(j)) \\ &\quad - (\text{max\_price}(i) - \text{price}(i,e)) - (\text{max\_price}(j) - \text{price}(j,e))) \\ &= \frac{1}{\alpha}(\text{price}(i,e) + \text{price}(j,e)) \\ &= \frac{1}{\alpha}(w(e)/2 + w(e)/2) \leq w(e). \end{aligned} \tag{7.13}$$

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

$$\begin{aligned} y(i) + y(j) - z(e) &= \frac{1}{\alpha}(\text{max\_price}(i) + \text{max\_price}(j) \\ &\quad - (\text{max\_price}(i) - \text{price}(i, e)) \\ &= \frac{1}{\alpha}(\text{max\_price}(j) + \text{price}(i, e)) \\ &\leq \frac{1}{\alpha}(w(e) + w(e)) \leq \frac{1}{2}(2w(e)) \leq w_e. \end{aligned} \tag{7.14}$$

The last line follows because $\text{max\_price}(j) \leq w(e)$, 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 $\text{excess}(e) = 0$, and 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 was saturated first and then j. This establishes that $\text{max\_price}(i) \leq w(e)$ and $\text{max\_price}(j) \leq w(e)$ since i and j were covered by an edge with lower weight than that of e. We have

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

We combine these analyses as follows:

$$\begin{aligned} &\sum_{e \in C} w(e) \\ &\quad = \sum_{v \in V} b_v \cdot \text{max\_price}(v) - \sum_{e \in E} \text{excess}(e) \quad \text{(from (7.9))} \\ &\quad = \alpha \cdot \left(\sum_{v \in V} b(v) y(v) - \sum_{e \in E} z(e) \right) \quad \text{(replacing } \textbf{max\_price} \text{ and } \textbf{excess}\text{)} \\ &\quad = \alpha \cdot EC_{dual} \leq \alpha \cdot EC_{LP} \leq \alpha \cdot EC^* \quad \text{(from (7.6))}. \end{aligned}$$

□

As in the 3/2-approximation algorithm we can also show the tightness of the approximation ratio of the b-NEAREST NEIGHBOUR algorithm. We generate a graph with n vertices, where n is odd, as follows. The vertex v_0 is connected to all other vertices $v_1 \ldots v_{(n-1)}$. Each vertex v_i for odd $i > 0$ is connected with $v_{(i+1)}$. All edges have the same weight x. We let $b = 1$ for every vertex. An example with $n = 9$ is shown in Figure 7.3.

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

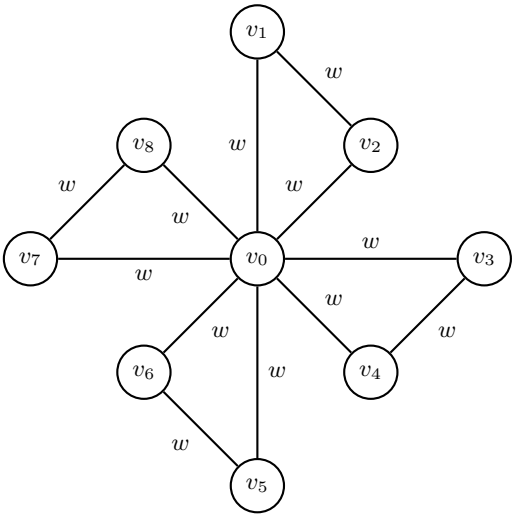

Figure 7.3. A tight example for the b-NEAREST NEIGHBOUR algorithm.

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

Lemma 7.7. The time complexity of the b-NEAREST NEIGHBOUR algorithm is $O(m \log \Delta)$.

Proof. We can sort the adjacency list of a vertex v in $O(\deg(v) \log(\deg(v)))$ time. Summing over all the vertices we obtain $O(m \log \Delta)$. □

The time complexity can be improved to $O(m)$ if we use the worst-case linear time selection algorithm as described in Cormen, Leiserson, Rivest and Stein (2009) to find the b_ith smallest element in the adjacency list of a vertex.

7.6. Δ-approximation algorithm

Here we present another algorithm based on linear programming duality for b-edge cover, with an approximation ratio of Δ, 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 Hall and Hochbaum (1986), 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 (7.3) is a lower bound for the optimum b-edge cover in (7.1). 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.

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

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

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

III

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

The proposed algorithm is as follows.

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

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

We first show that such an algorithm would provide a Δ-approximation.

Lemma 7.8. The algorithm is a Δ-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

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

We show that the value of ϵ 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 $\bar{y}(i) + \bar{y}(j) - \bar{z}(e) < w(e)$, equivalently $w(e) - (\bar{y}(i) + \bar{y}(j) - \bar{z}(e)) > 0$. We set $\bar{y}(v)' = \bar{y}(v) + \epsilon$, and $\bar{z}(e)' = \bar{z}(e) + \epsilon$, for edges $e \in \delta(v)$ and $e \in C$. 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 ϵ. This establishes that C is a feasible cover.

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

$$\sum_{v \in V} b(v)\bar{y}(v) - \sum_{e \in E} \bar{z}(e) \leq EC_{LP} \leq EC^*. \tag{7.15}$$

Algorithm 16 DUAL FEASIBLE($G = (V, E, w, b)$)

```
Initialize y_v = 0 and z_e = 0, for all v ∈ V and e ∈ E
Assign w'(e) = w(e), for all 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
```

Using property III, we obtain

$$\sum_{v \in V} \bar{y}(v) \leq EC^*. \tag{7.16}$$

From the construction of the cover we have

$$\begin{aligned} \sum_{e \in C} w(e) + \sum_{e \in C} \bar{z}(e) &= \sum_{e=(i,j) \in C} \bar{y}(i) + \bar{y}(j) \\ &= \sum_{v \in V} \sum_{e \in (\delta(v) \cap C)} \bar{y}(v) \\ &\leq \sum_{v \in V} |\delta(v)| \bar{y}(v) \\ &\leq \Delta \sum_{v \in V} \bar{y}(v) \leq \Delta \cdot EC^*. \end{aligned} \tag{7.17}$$

In the last line we used (7.16). Hence we have established the Δ-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 in Algorithm 16.

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 $w'(e)$ from $z(e)$ (thus the value of $z(e)$ increases), and sets $w'(e)$ 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 now show that the dual vectors derived are maximal and satisfy the three properties.

Claim 7.9. The variables $\mathbf{y}$ and $\mathbf{z}$ 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 $z(e)$ is updated by subtracting $w'(e)$ when its value is negative, and hence these variables are non-negative as well. □

Claim 7.10. All edges e in the cover C satisfy $y(i) + y(j) - z(e) = w(e)$.

Proof. Let i be a vertex at some iteration of the algorithm and $f = (i, j)$ 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 $y(i)$. In the **for** loop over edges, we 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, $y(i) + y(j) - z(e) = w(f)$, 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 $y(i) + y(j) - z(e) = w(e)$ is maintained by increasing the value of $z(e)$. □

Claim 7.11. The inequality $y(i)+y(j)-z(e) \leq w(e)$ holds for all $e \in E \backslash C$.

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

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

Lemma 7.12. The dual vector $(\mathbf{y}, \mathbf{z})$ of the Dual Feasible algorithm is maximal.

Proof. (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 7.9). Suppose we increase a dual variable $y(v)$ by a non-negative amount ϵ. 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 ϵ, its constraint is also violated.) To compensate for the constraint violations, we need to add ϵ 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 $(\bar{y}, \bar{z})$.

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

(III) Since $z(e) = 0$, for all $e \in E \setminus C$, it suffices to show that

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

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 show that the inequality above is true in every iteration:

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

For $t = 1$, the left side is zero since the $w'(e)$ values for all the edges are non-negative after the first iteration, and the right side is ≥ 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 kth iteration by v, and let f be the minimum weight edge incident on v with weight $w'(f)$. Then the right-hand side of the displayed equation increases by $(b(v) - 1)w'(f)$. Now this could lead to increase in some $z(e)^k$, where e is incident on v. When f is included in the cover, there are at most $(b(v) - 1)$ covering edges already incident on the vertex v. 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 $(b(v) - 1)w'(f)$. Hence the inequality is preserved at the end of this iteration. □

We can show a tight example of the Δ-approximation algorithm by considering the graph in Figure 7.3 with different weights as follows. The weights of the edges $(v_i, v_{(i+1)})$, where $i = 1 \cdots 7$, 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 $\Delta/2 * (2\epsilon/\Delta) + w = \epsilon + w$, whereas if the Dual Feasible algorithm picks

Table 7.1. 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

the first vertex to be v_0, the weight of the edge cover could be Δx. Taking the ratio we have $(\Delta w)/(\epsilon + w)$. As $\epsilon \to 0$, the ratio approaches Δ.

Lemma 7.13. 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 $\sum_{v \in V} b(v) \cdot \deg(v) = O(\beta\ m)$. □

7.7. Computational results

7.7.1. Experimental set-up

All the experiments were conducted on a Purdue community cluster computer called Rice, described in Section 3.3.

Our test set consists of both real-world and synthetic graphs shown in Table 7.1. We generated two classes of RMAT graphs: (a) G500, representing graphs with skewed degree distributions from the Graph 500 benchmark (Murphy, Wheeler, Barrett and Ang 2010), 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 SuiteSparse Matrix Collection (Davis and Hu 2011) covering application areas such as medical science, structural engineering and sensor data. We also have a

Table 7.2. Comparison of weights of edge covers computed by approximation algorithms with respect to 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-logn21	26 301 787	0.12	0.18	0.01
hollywood-2011	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
Geometric mean		1.32	2.25	0.28

large web-crawl graph (eu-2015) (Boldi, Marino, Santini and Vigna 2014) and a movie-interaction network (hollywood-2011) (Boldi and Vigna 2004).

All reported results are the average of five runs on 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].

7.7.2. 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 Section 6, the Nearest Neighbour 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.

Table 7.3. Relative performance of run times of approximation algorithms with respect to the exact algorithm for edge cover.

Problems	Time (s)	Relative performance		
	Exact algorithm	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-logn21	147.89	37.92	78.83	43.96
hollywood-2011	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
Geometric mean		36.73	74.33	42.97

In Table 7.2 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.

In Table 7.3 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. The larger the relative performance, the faster the algorithm. Note 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.

7.7.3. b-edge cover results

We do not have an exact algorithm to compare the approximation algorithms against since the weight transformation reduction does not work for b-edge cover. We compare the PRIMAL DUAL algorithm (PD), the LAZY

Table 7.4. Weight of b-edge covers computed by the PRIMAL DUAL algorithm. 'Increase' is the percentage of increase in weight of edge covers computed by the b-NEAREST NEIGHBOUR algorithm with respect to those from the PRIMAL DUAL algorithm.

Problem	PRIMAL DUAL		Increase (%)	
	Original	Remove redund.	Original	Remove redund.
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-logn21	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-2011	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
Geometric mean			4.77	0.51

GREEDY algorithm (LG) and the b-NEAREST NEIGHBOUR algorithm (bNN). Both the PD and LG algorithms are 3/2-approximate and compute the same b-edge cover, while bNN is 2-approximate. In Table 7.4 we report the weights of the b-edge covers. We include the weight computed by the original algorithm, and then the weight obtained 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 Figure 7.4 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 mean that the algorithm is faster than LG. Note that in geometric mean, the PD algorithm is twice as fast as LG, and the bNN algorithm is about six times as fast as LG.

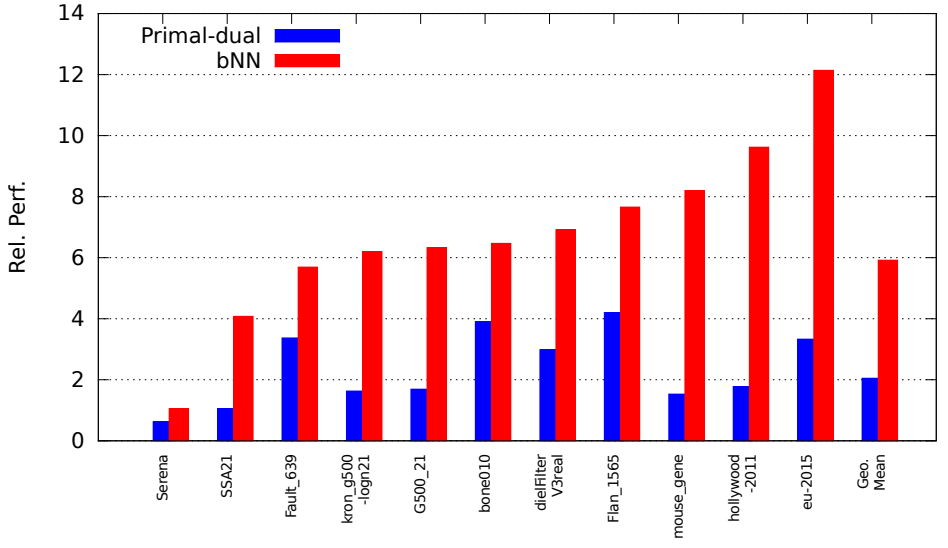

Figure 7.4. Relative performance of run times of the PRIMAL DUAL and b-NEAREST NEIGHBOUR algorithms for b-edge cover with respect to the LAZY GREEDY algorithm.

The Δ-approximation algorithm was implemented by Khan and Pothen (2016) and compared with the LSE algorithm, which has an approximation ratio of 3/2. The performance of the latter algorithm was highly sensitive to the order in which the vertices are processed, for both weights and run times. Generally the LSE algorithm computed lower weights and it was also faster.

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 (Khan *et al.* 2018*b*). 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 8192 cores of a distributed-memory parallel computer with good speedups (Khan *et al.* 2018*a*), and it has been used to solve the adaptive anonymity problem, which we discuss in the next subsection.

Software for exact (based on the weight transformation to a matching problem) and approximation algorithms for edge cover and b-edge cover (GREEDY, LAZY GREEDY, b-NEAREST NEIGHBOUR and PRIMAL DUAL algorithms) will be included at https://github.com/CSCsw/EdgeCover. Code

for the b-Suitor algorithm from which the MCE b-edge cover could be computed is available at https://github.com/Exa-Graph.

7.8. Applications of b-edge cover

A widely used application of b-edge cover is in computing b-Nearest Neighbour graphs (bNN graphs) to construct sparse graphs out of noisy data, although practitioners do not seem to know about the relationship of this construction to the b-edge cover problem. Subramanya and Talukdar (2014) provide a recent discussion of the bNN graph construction in semi-supervised machine learning, and compare it with other methods such as b-matching (Jebara *et al.* 2009). (In the literature this is called the k-nearest neighbour graph.) We have shown here that the minimum weight b-edge cover formulation leads to a more general formulation where we are not constrained to use an identical value of $b(v)$ for all vertices. This is practically important since choosing $b(v)$ adaptively, *e.g.* to be proportional to the degree of v, could lead to a graph that more faithfully represents the data. Furthermore, linking the b-nearest neighbour problem to the b-edge cover problem leads to several 3/2-approximation algorithms, while the bNN graph construction leads to a 2-approximation algorithm. We have also shown that the bNN graph construction leads to the inclusion of many redundant edges, which could be removed to reduce the weight of the edge cover, although the worst-case approximation ratio is not reduced by this technique. Use of the bNN graph construction is seen experimentally to lead to highly skewed degree distributions, since several neighbours of a vertex v could include it among its nearest neighbours, whereas the other b-edge cover algorithms provide better control of the degree distribution. This can influence the quality of the classification results obtained in semi-supervised machine learning.

Jebara *et al.* (2009) have proposed the use of b-matching to solve this problem, using an exact belief propagation-based algorithm for computing perfect b-matchings from a complete graph that represents the data. The use of an exact algorithm makes this approach quite expensive relative to using a 1/2-approximation algorithm to compute the b-matching. Furthermore, working with a complete graph of the data makes the algorithm expensive for large data, whereas with a sparse representation of the data, the b-edge cover formulation ensures that each vertex v has at least $b(v)$ neighbours in the constructed graph (the b-matching only ensures at most $b(v)$ neighbours).

We discuss a problem in data privacy, adaptive anonymity, that stimulated our work on the b-edge cover problem.

Table 7.5 shows a small example to illustrate the adaptive anonymity problem. The top matrix contains binary data measured on six features for six individuals. Each individual is willing to permit their data to be

Table 7.5. A small example illustrating adaptive anonymity. From top to bottom: original input, dissimilarity matrix (Hamming distances) and anonymized output.

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	*

published provided each one is confused with at least one other person in the database. The idea is that a '$*$' could correspond to a 0 or a 1. The problem is to mask the least number of data items with '$*$' so that the privacy requirement of every individual is met. The matrix in the middle shows the Hamming distances between pairs of individuals, which is a measure of dissimilarity that indicates in how many entries the data of the two individuals differ. Now we compute a minimum weight 1-edge cover of the complete graph corresponding to the dissimilarity matrix, which includes

the edges $(1, 2)$, $(3, 4)$ and $(5, 6)$. Finally, we pair individuals using this edge cover, and introduce '$*$'s in the columns where these pairs of individuals differ. Note that the weight of the edge cover is five, and there are ten '$*$'s in the anonymized output matrix that could be published for machine learning purposes.

More formally, in adaptive anonymity, each individual v expresses a required level of privacy, to be confused with $b(v) - 1$ other individuals. We are given a data set $X \in \mathbb{Z}^{n \times f}$, 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 individual v to the data set and consists of f discrete features. A feature might be race, age, height, weight, income bracket, *etc.*, but not unique identifiers such as social security number. A vector $\mathbf{b}$ of length n, where an element $b(v)$ is a privacy requirement of the vth individual, is also given. The value $b(v)$ specifies that the data of the vth user must be indistinguishable from that of $b(v) - 1$ other users. The output of the algorithm is an anonymized data set $Y \in (\mathbb{Z} \cup \{*\})^{n \times f}$, where the '$*$' symbol indicates that a particular feature has been masked.

The adaptive anonymity problem is NP-hard, but Choromanski *et al.* (2013) proposed an algorithm that finds a good-quality approximate solution. The approximate solution comes from the observation that if we group similar instances together (with respect to their corresponding features) then we need to hide fewer features. The algorithm is a variational optimization method 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. Thus at each iteration one needs to solve the grouping step.

Choromanski *et al.* (2013) used a perfect b-matching of minimum weight to group similar instances together. This limited the size of the instances they could solve to a few thousand individuals, since the exact b-matching algorithm has $O(b(V)\, m \log n)$ time complexity. Khan *et al.* (2018*a*) have shown that one could instead use a minimum weight b-edge cover formulation, and then use an approximation algorithm to compute the b-edge cover. This results in a 2β-approximation algorithm for minimizing the number of stars in the anonymized data, where β is the maximum value of the privacy requirements over individuals. The factor 2 comes from the choice of the algorithm used to compute the b-edge cover, and it could be reduced to $3/2$ with the Greedy or the Primal Dual algorithm. These authors computed the b-edge cover by taking the complement of a $1/2$-approximate b-matching,

using the b-Suitor algorithm. This algorithm was implemented efficiently on shared-memory and distributed-memory parallel computers, to provide the first reported parallel solutions for the adaptive anonymity problem. The work (total number of operations) in the distributed-memory parallel algorithm is $O(b(V)m)$, and its depth is $O(\log \Delta \log m)$. This algorithm also reduced the memory required to solve the problem from quadratic to linear in the number of individuals, since for each row in the dissimilarity matrix we need to store only the top $k(v)$ values (where $k(v)$ is some multiple of $b(v)$) to compute the maximum weight b-matching. If these do not suffice to obtain the b-matching, then the elements that had been processed could be discarded, and an additional set of $k(v)$ elements could be computed. The parallel algorithm solved an anonymity problem involving Medicare/Medicaid physician billing data with more than 700 000 individuals and 500 features on a distributed-memory computer with 8192 cores in four minutes. This represents an increase in problem size by three orders of magnitude over the earlier approaches.

8. Other approximation algorithms in CSC

We discuss two other classical problems in CSC where approximation algorithms have been designed or inapproximability results obtained.

8.1. Graph colouring

The graph colouring problem assigns the fewest colours to the vertices of a graph such that adjacent vertices receive different colours. Formally, we find a function $c : V \mapsto \mathbb{N}$ such that for every edge (u, v) we have $c(u) \neq c(v)$, and c uses the minimum number of colours. The *chromatic number* of a graph G is the minimum number of colours needed to colour it. Computing the chromatic number of a graph is an NP-hard problem. Furthermore, an inapproximability result due to Feige and Kilian (1998) states that for any $\epsilon > 0$ no polynomial time algorithm can approximate the chromatic number to within a factor of $n^{1-\epsilon}$ unless $P = NP$.

In CSC, several variants of graph colouring problems are of interest in computing sparse Jacobians and Hessians efficiently. Despite the pessimistic approximability result for colouring, in many contexts in CSC (such as finite element methods for solving partial differential equations) the graphs that need to be coloured are bounded in degree. Any graph can be coloured in at most $\Delta + 1$ colours, and hence the colouring problem is easy for the class of bounded degree graphs. An even tighter upper bound is the colouring number, which is obtained from an ordering computed by iteratively deleting vertices of minimum degree in the graph, updating degrees, and continuing until the graph is empty. (This is the same algorithm that computes the maximum core of a graph.) The maximum value of the minimum degree

observed during this process is the colouring number. By colouring the graph in the reverse order in which the vertices are deleted from the graph, one can colour the graph in this many colours. A more detailed discussion is available in Gebremedhin, Manne and Pothen (2005).

For Erdős–Rényi graphs with constant average degree, the chromatic number can be precisely computed, and the greedy colouring algorithm can be shown to colour the graph in at most twice this number of colours. Let $G(n, d/n)$ denote an Erdős–Rényi graph on n vertices with the probability of an edge equal to d/n. We say that a property of a random graph on n vertices A_n holds asymptotically almost surely (a.a.s.) if the probability that A_n is true satisfies $\mathbb{P}(A_n) \to 1$ as $n \to \infty$. Achlioptas and Naor (2005) proved the following result (see also Kang and McDiarmid 2015).

Theorem 8.1. Given $d > 0$, let k_d be the least integer k for which $d < 2(k-1)\ln(k-1)$. Then $\chi(G(n, d/n))$ is $k_d - 1$ or k_d a.a.s. If $d > (2k_d - 3)\ln(k_d - 1)$, then $\chi(G(n, d/n)) = k_d$ a.a.s.

A colouring algorithm that finds maximal independent sets and colours them greedily can be shown to use at most twice the number of colours as the chromatic number (McDiarmid 1984). This paper discusses why this is simultaneously both a good and a bad result.

Now we turn to what is known about approximation algorithms for some of these colouring problems.

A distance-2 colouring of a graph $G = (V, E)$ is a colouring such that a vertex receives a colour distinct from any of its neighbours at distance 2 or less. This variant arises in computing Hessians if one does not take into account the symmetry of the graph. McCormick (1983) described a simple algorithm that computes an $O(\sqrt{n})$-approximation to the distance-2 chromatic number. Two other colouring problems that occur in Hessian computation where we need to compute only one among the two elements H_{ij} or H_{ji} for $i \neq j$, due to the symmetry of the Hessian, are star colouring and acyclic colouring. In star colouring, adjacent vertices receive distinct colours, and also every path on four vertices (P_4) should receive at least three colours. Thus two-coloured subgraphs should be stars. In acyclic colouring, adjacent vertices should receive distinct colours, and every cycle should receive at least three colours. Thus two-coloured subgraphs should be forests. *ZPP* is the class of all problems solvable in zero error probabilistic time. Gebremedhin, Tarafdar, Manne and Pothen (2007) showed that the star chromatic number and the acyclic chromatic number cannot be approximated to within a factor of $n^{(1/3-\epsilon)}$ unless $NP \subseteq ZPP$.

Bicolouring problems arise when we evaluate a Jacobian matrix using both its rows and columns. In this context, we do not need to evaluate every row and every column, since each non-zero in the Jacobian could be computed from its row or its column. Hence we have a colouring problem in

which a vertex u could be assigned the colour 0 to indicate that u will not be used to compute any non-zeros in that row or column. The star bicolouring of a bipartite graph $G = (V_1, V_2, E)$ is a function $c : \{V_1 \cup V_2\} \mapsto \mathbb{N} \cup \{0\}$ such that

(1) $c(v_i) \neq c(v_j)$ if $(i, j) \in E$;

(2) the set of non-zero colours (set of 'true' colours) of V_1 is disjoint from the set of non-zero colours of V_2;

(3) two vertices v_i and v_k adjacent to a vertex v_j with $c(v_j) = 0$ receive distinct colours; and

(4) every path on four vertices receives at least three colours.

The acyclic bicolouring problem satisfies all the conditions of the star bicolouring problem, except for the last condition, which is replaced with: every cycle receives at least three colours.

Juedes and Jones (2012) have designed an approximation algorithm for the star bicolouring problem. The algorithm computes distance-2 independent sets of vertices (from either V_1 or V_2) containing a vertex of maximum degree in the current graph using a greedy algorithm, and then assigns them all one colour. The choice of independent sets from V_1 or V_2 is done based on the maximum degree and ratios of $|V_i|/\Delta$. They show that this algorithm has $O(n^{2/3})$ approximation ratio and that its time complexity is $O(m^2/n^{1/3})$. Since every star bicolouring is also an acyclic bicolouring, the approximation result holds for the latter problem as well. The authors have also implemented their algorithms, and showed on graphs with several hundred vertices and thousands of edges that the algorithm is competitive in run time with heuristic colouring algorithms that have been designed for star bicolouring.

8.2. Minimizing fill in sparse Cholesky factorization

The minimum fill problem is one of the most basic problems in the area of sparse matrix computations. Given a sparse, symmetric positive definite matrix A, we wish to permute its rows and columns symmetrically and compute the Cholesky factors of the permuted matrix

$$PAP^T = LL^T,$$

where P is a permutation matrix and L is the lower triangular Cholesky factor, such that the number of non-zeros in L are minimized. A fill element is a matrix element $L_{ij} \neq 0$ such that $A_{ij} = 0$.

In a graph model, we consider the undirected adjacency graph of A, and eliminate its vertices one by one. To eliminate a vertex, we add edges to make all of its neighbours a clique, and then delete the vertex and all

edges incident on it. The problem is to find an ordering for eliminating the vertices that minimizes the edges added in this process, *i.e.* the fill edges. This problem was proved to be NP-complete by Yannakakis (1981). It is well known that the graph of the Cholesky factor, the filled graph, is a chordal graph.

Nested dissection, proposed by George (1973) is a technique for solving this problem using the divide and conquer paradigm. It works by finding a vertex separator that divides the graph into roughly two equal-sized (in terms of vertices) subgraphs. The two subgraphs are ordered first followed by the separator. One recurses on the subgraphs to find separators in the subgraphs, and orders them with this process. For planar graphs there exist separators of size $O(n^{1/2})$, and the fill can be bounded by $O(n \log n)$; for graphs (finite element meshes) that can be embedded in three dimensions with good aspect ratios, the separator size is $O(n^{2/3})$, and the fill is $O(n^2)$. Another heuristic which is widely used is the minimum degree algorithm and its many variants.

Unfortunately we cannot prove how close to optimum the solutions from these heuristic fill reduction algorithms are, although they obtained less fill than an approximation algorithm designed by Agrawal, Klein and Ravi (1993) for many matrices. These authors obtained an algorithm with approximation ratio $O(n^{1/2} \log^{3.5} n)$ for the total number of edges in the filled graph. (Their objective function was the sum of the number of edges in the original graph and the fill edges.) Natanzon, Shamir and Sharan (2000) obtained an approximation algorithm for minimizing the fill with approximation ratio eight times the minimum fill size. A polynomial time approximation scheme (PTAS) is a polynomial time algorithm that takes a parameter $\epsilon > 0$ and produces an approximate solution for a minimization problem with approximation ratio $(1+\epsilon)$. Recently, Cao and Sandeep (2017) proved that if the problems of minimizing the fill or the total number of edges in the filled graph has a PTAS, then $P = NP$.

9. Conclusions

We have surveyed the design and implementation of approximation algorithms for several matching and edge cover problems and their applications in combinatorial scientific computing. Our interest is in algorithms that could be implemented to obtain high performance on modern processor architectures, including serial and parallel computers, both shared-memory and distributed-memory machines. We have viewed approximation as a paradigm for designing parallel algorithms (Khan *et al.* 2018*b*).

The paradigm of designing approximation algorithms for parallelism has been considered in the theoretical computer science community for vertex and set cover problems by Khuller, Vishkin and Young (1994), and for

facility location, max cut, set cover and low stretch spanning trees, by Blelloch, Peng and Tangwongsan (2011) and Tangwongsan (2011). The idea underlying many of these parallel algorithms is that a greedy algorithm chooses a most cost-effective element in each iteration, and by allowing a slack, a factor of $(1+\epsilon)$, more elements can be selected at the cost of a slightly worse approximation ratio. These algorithms have poly-logarithmic depth, and although some of them have linear work requirements, there are few parallel implementations that we know of. Blelloch *et al.* (2012) have computed a maximal cardinality matching (which would be a 1/2-approximate algorithm) in parallel.

For matching problems we have discussed new approximation algorithms for maximum cardinality matching. For edge-weighted matching, we have described 1/2-approximation algorithms that employ different paradigms: greedy, path-growing, and proposals (related to the stable matching problem). We have considered $(2/3 - \epsilon)$-approximation algorithms for this problem as well. We have discussed 1/2-approximation algorithms for the b-matching problem, including a matroid-theoretic proof that the greedy algorithm leads to 1/2-approximation. We have also designed 1/2- and 2/3-approximation algorithms for vertex-weighted matching, using techniques that differ from those used for edge-weighted matching. We summarize the matching problems, exact and approximation algorithms to solve them, and their time complexity in Table 9.1.

For the minimum weight edge cover problem we have discussed an approximation-preserving reduction to the maximum weight matching problem, leading to both exact and several approximation algorithms. The b-edge cover problem has a rich collection of approximation algorithms, from the greedy algorithm and variants, and a primal–dual algorithm; these algorithms result in 3/2-approximation. The well-known b-nearest neighbour graph construction leads to a 2-approximation for this problem, as do greedy-like algorithms that do not compute dynamic effective weights, and an algorithm that computes the complement of a suitable 1/2-approximation matching algorithm. For the b-edge cover problem, we discussed a primal–dual linear programming framework that helps establish the approximation ratios of several algorithms. We summarize the edge cover problems, exact and approximation algorithms to solve them, and their time complexity in Table 9.2.

We have considered applications of matching to the solution of sparse systems of linear equations and other matrix computations. For the b-edge cover problem, we have discussed the construction and sparsification of graphs from large, noisy data sets; we have also described the solution of a data privacy problem called adaptive anonymity.

We believe that approximation algorithms represent a fruitful area for progress in designing efficient algorithms for solving problems in CSC, data

Table 9.1. Some of the exact and approximation algorithms for matching problems and their time complexity.

Problem	Algorithm	Reference	Complexity
Maximum matching		Micali and Vazirani (1980)	$O(\sqrt{n}\ m)$
Maximum weight matching		Gabow (2018)	$O(n(m + n\ \log n))$
1/2-approximation MWM	Path Growing Suitor	Drake and Hougardy (2003*b*) Manne and Halappanavar (2014)	$O(m)$ $O(m \log \Delta)$
(2/3-ϵ)-approximation MWM		Pettie and Sanders (2004)	$O(m \log \epsilon^{-1})$
$(1 - \epsilon)$-approximation MWM		Duan and Pettie (2014)	$O(m\epsilon^{-1} \log \epsilon^{-1})$
Maximum b-matching		Gabow (1983)	$O(\sqrt{b(V)}\ m)$
Maximum weight b-matching		Gabow (1983)	$O(b(V)\min\{m \log n, n^2\})$
1/2-approx. maximum weight b-matching	b-Suitor	Khan *et al.* (2016*b*)	$O(m \log \beta)$
Maximum vertex weight matching		Spencer and Mayr (1984)	$O(\sqrt{n}\ m \log n)$
1/2-approximation MVM	Greedy		$O(m + n \log n)$
2/3-approximation MVM		Al-Herz and Pothen (2019)	$O(m \log \Delta + n \log n)$

Table 9.2. Some of the exact and approximation algorithms for edge cover problems and their time complexity.

Problem	Algorithm	Reference	Complexity
Minimum EC		Norman and Rabin (1959), Micali and Vazirani (1980)	$O(\sqrt{n}\ m)$
Minimum weight EC	Weight trans. + MWM	Schrijver (2003), Gabow (2018)	$O(n(m+n\ \log n))$
3/2-approx. minimum weight EC	Weight trans. + 1/2-MWM	Schrijver (2003), Drake and Hougardy (2003*b*)	$O(m)$
2-approx. minimum weight EC	Nearest Neighbour	Ferdous *et al.* (2018)	$O(m)$
Minimum b-EC		Huang and Pettie (2017)	$O(\sqrt{b(V)}\ m)$
Minimum weight b-EC	Complement of b'-matching	Schrijver (2003), Gabow (1983)	$O(b'(V)\min\{m\log n, n^2\})$
3/2-approx. min. weight b-EC	Primal Dual Lazy Greedy	Ferdous *et al.* (2018)	$O(\beta\ \Delta m)$ $O(m\log n)$
2-approx. minimum weight b-EC	b-Nearest Neighbour MCE	Khan *et al.* (2018*b*)	$O(m)$ $O(m\log\beta')$
$(1+\epsilon)$-approx. min. weight b-EC		Huang and Pettie (2017)	$O(m\epsilon^{-1}\log\epsilon^{-1})$

science, machine learning and other emerging application domains. We expect that the increasing sizes of these problems and the availability of parallel computing resources will demand the development of efficient and concurrent approximation algorithms. We trust that this survey will stimulate further work along these lines.

Acknowledgements

We are grateful to Bora Uçar, ENS Lyon, for his extensive comments on our manuscript. It is a pleasure to thank our colleagues who have collaborated with us on matching and edge cover problems: Arif Khan and Mahantesh Halappanavar, both of Pacific Northwest National Laboratory; Florin Dobrian, Conviva Corporation; Ariful Azad, Indiana University; Ahmed Al-Herz, Purdue University; Johannes Langguth, Simula; Aydin Buluç, Lawrence Berkeley National Laboratory; Mostofa Ali Patwary, NVIDIA; Pradeep Dubey, Intel Corporation; Rob Bisseling, Utrecht University; and Seth Pettie, University of Michigan. We also thank Peter Sanders of the Karlsruhe Institute of Technology and Jens Maue of Zürich for sharing with us their code for the $(2/3 - \epsilon)$-approximation matching algorithms.

REFERENCES[2]

D. Achlioptas and A. Naor (2005), 'The two possible values of the chromatic number of a random graph', *Ann. of Math.* **162**, 1335–1351.

A. Agrawal, P. N. Klein and R. Ravi (1993), Cutting down on fill using nested dissection: Provably good elimination orderings. In *Graph Theory and Sparse Matrix Computations* (A. George, J. R. Gilbert and J. W. H. Liu, eds), Springer, pp. 31–55.

A. Al-Herz and A. Pothen (2019), 'A 2/3-approximation algorithm for vertex-weighted matching', *Discrete Appl. Math.*, under review. arXiv:1902.05877

R. P. Anstee (1987), 'A polynomial algorithm for b-matchings: An alternative approach', *Inform. Process. Lett.* **24**, 153–157.

A. Azad and A. Buluç (2016), Distributed memory algorithms for maximum cardinality matching on bipartite graphs. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, pp. 32–42.

A. Azad, A. Buluç and A. Pothen (2017), 'Computing maximum cardinality matchings in parallel on bipartite graphs via tree grafting', *IEEE Trans. Parallel Distrib. Syst.* **28**, 44–59.

A. Azad, A. Buluç, X. S. Li, X. Wang and J. Langguth (2018), 'A distributed memory approximation algorithm for maximum weight perfect bipartite matching', *SIAM J. Sci. Comput.*, under review. arXiv:1801.09809v1

[2] The URLs cited in this work were correct at the time of going to press, but the publisher and the authors make no undertaking that the citations remain live or are accurate or appropriate.

A. Azad, J. Langguth, Y. Fang, A. Qi and A. Pothen (2010), Identifying rare cell populations in comparative flow cytometry. In *Algorithms in Bioinformatics: International Workshop on Algorithms in Bioinformatics (WABI)*, Vol. 6293 of Lecture Notes in Bioinformatics, Springer, pp. 162–175.

H. Bast, K. Mehlhorn, G. Schäfer and H. Tamaki (2006), 'Matching algorithms are fast in sparse random graphs', *Theory Comput. Syst.* **39**, 3–14.

C. E. Bell (1994), 'Weighted matching with vertex weights: An application to scheduling training sessions in NASA space shuttle cockpit simulators', *Europ. J. Oper. Res.* **73**, 443–449.

M. Birn, V. Osipov, P. Sanders, C. Schulz and N. Sitchinava (2013), Efficient parallel and external matching. In *Euro-Par 2013 Parallel Processing*, Vol. 8097 of Lecture Notes in Computer Science, Springer, pp. 659–670.

B. Birnbaum and C. Mathieu (2008), 'On-line bipartite matching made simple', *ACM SIGACT News* **39**, 80–87.

G. E. Blelloch, J. T. Fineman and J. Shun (2012), Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*, ACM, pp. 308–317.

G. E. Blelloch, R. Peng and K. Tangwongsan (2011), Linear work parallel greedy approximate set cover and variants. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*, ACM, pp. 23–32.

P. Boldi and S. Vigna (2004), The WebGraph framework I: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW 2004)*, ACM, pp. 595–601.

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

A. Buluç and J. R. Gilbert (2011), 'The Combinatorial BLAS: Design, implementation and applications', *Internat. J. High Perf. Comput. Appl.* **25**, 496–509.

R. Burkard, M. Dell'Amico and S. Martello (2009), *Assignment Problems*, SIAM.

Y. Cao and R. B. Sandeep (2017), Minimum fill-in: Inapproximability and almost tight lower bounds. In *Proceedings of the 28th Annual Symposium on Discrete Algorithms (SODA)*, SIAM, pp. 875–880.

K. M. Choromanski, T. Jebara and K. Tang (2013), Adaptive anonymity via b-matching. In *Advances in Neural Information Processing Systems (NIPS 2013)* (C. J. C. Burges *et al.*, eds), pp. 3192–3200.

V. Chvatal (1979), 'A greedy heuristic for the set-covering problem', *Math. Oper. Res.* **4**, 233–235.

J. Cohen and P. Castonguay (2012), Efficient graph matching and coloring on GPUs. Presentation available at: http://on-demand.gputechconf.com/gtc/2012/presentations/S0332-Efficient-Graph-Matching-and-Coloring-on-GPUs.pdf

T. F. Coleman and A. Pothen (1987), 'The null space problem II: Algorithms', *SIAM J. Algebraic Discrete Methods* **8**, 544–563.

T. F. Coleman, A. Edenbrandt and J. R. Gilbert (1986), 'Predicting fill for sparse orthogonal factorization', *J. Assoc. Comput. Mach.* **33**, 517–532.

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

T. Davis and Y. Hu (2011), 'The University of Florida Sparse Matrix Collection', *ACM Trans. Math. Softw.* **38**, 1:1–1:25.

G. De Francisci Morales, A. Gionis and M. Sozio (2011), 'Social content matching in MapReduce', *Proc. VLDB Endowment* **4**, 460–469.

U. Derigs and A. Metz (1986), 'On the use of optimal fractional matchings for solving the (integer) matching problem', *Computing* **36**, 263–270.

M. Deveci, K. Kaya, B. Uçar and Ü. Çatalyürek (2013), GPU accelerated maximum cardinality matching algorithms for bipartite graphs. In *Proceedings of 19th International Euro-Par Conference on Parallel Processing*, pp. 850—861.

B. Dezső, A. Jüttner and P. Kovács (2011), 'LEMON: An open source C++ graph template library', *Electron. Notes Theoret. Comput. Sci.* **264**, 23–45.

F. Dobrian, M. Halappanavar, A. Pothen and A. Al-Herz (2019), 'A 2/3-approximation algorithm for vertex-weighted matching in bipartite graphs', *SIAM J. Sci. Comput.* **41**, A566–A591.

D. Drake and S. Hougardy (2003*a*), Linear time local improvements for weighted matchings in graphs. In *Experimental and Efficient Algorithms* (K. Jansen, M. Margraf, M. Mastrolilli and J. Rolim, eds), Vol. 2647 of Lecture Notes in Computer Science, Springer, pp. 107–119.

D. E. Drake and S. Hougardy (2003*b*), 'A simple approximation algorithm for the weighted matching problem', *Inform. Process. Lett.* **85**, 211–213.

D. E. Drake and S. Hougardy (2005), 'A linear time approximation algorithm for weighted matchings in graphs', *ACM Trans. Algorithms* **1**, 107–122.

D. Du, K. Ko and X. Hu (2012), *Design and Analysis of Approximation Algorithms*, Springer.

R. Duan and S. Pettie (2010), Approximating maximum weight matching in near-linear time. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS '10)*, IEEE, pp. 673–682.

R. Duan and S. Pettie (2014), 'Linear-time approximation for maximum weight matching', *J. Assoc. Comput. Mach.* **61**, 1–23.

I. S. Duff and J. Koster (2001), 'On algorithms for permuting large entries to the diagonal of a sparse matrix', *SIAM J. Matrix Anal. Appl.* **22**, 973–996.

I. S. Duff and B. Uçar (2012), Combinatorial problems in solving linear systems. In *Combinatorial Scientific Computing* (U. Naumann and O. Schenk, eds), CRC, pp. 21–68.

I. S. Duff, K. Kaya and B. Uçar (2011), 'Design, implementation, and analysis of maximum transversal algorithms', *ACM Trans. Math. Softw.* **38**, 13:1–13:31.

F. Dufossé, K. Kaya and B. Uçar (2015), 'Two approximation algorithms for bipartite matching on multicore architectures', *J. Parallel Distrib. Comput.* **85**, 62–78.

J. Edmonds (1965), 'Maximum matching and a polyhedron with 0, 1-vertices', *J. Res. Nat. Bureau Standards* **69B**, 125–130.

B. O. Fagginger Auer and R. H. Bisseling (2012), A GPU algorithm for greedy graph matching. In *Facing the Multicore-Challenge II* (R. Keller, D. Kramer and J. Weiss, eds), Springer, pp. 108–119.

U. Feige and J. Kilian (1998), 'Zero knowledge and the chromatic number', *J. Comput. Systems Sci.* **57**, 187–199.
S. Ferdous, A. Pothen and A. Khan (2018), New approximation algorithms for minimum weighted edge cover. In *2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, SIAM, pp. 97–108.
P. Fritzson (2014), *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*, Wiley/IEEE.
H. N. Gabow (1983), An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing (STOC '83)*, ACM, pp. 448–456.
H. N. Gabow (2018), 'Data structures for weighted matching and extensions to b-matching and f-factors', *ACM Trans. Algorithms* **14**, 39:1–39:80.
D. Gale and L. S. Shapley (1962), 'College admissions and the stability of marriage', *Amer. Math. Monthly* **69**, 9–15.
T. Gallai (1959), 'Über extreme Punkt- und Kantenmengen', *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Mathematica* **2**, 133–138.
A. H. Gebremedhin, F. Manne and A. Pothen (2005), 'What color is your Jacobian? Graph coloring for computing derivatives', *SIAM Review* **47**, 629–705.
A. H. Gebremedhin, A. Tarafdar, F. Manne and A. Pothen (2007), 'New acyclic and star coloring algorithms with application to computing Hessians', *SIAM J. Sci. Comput.* **29**, 1042–1072.
A. George (1973), 'Nested dissection of a finite element mesh', *SIAM J. Numer. Anal.* **10**, 345–363.
G. Georgiadis and M. Papatriantafilou (2013), 'Overlays with preferences: Distributed, adaptive approximation algorithms for matching with preference lists', *Algorithms* **6**, 824–856.
M. X. Goemans and D. P. Williamson (1997), The primal–dual method for approximation algorithms and its application to network design problems. In *Approximation Algorithms for NP-hard Problems* (D. S. Hochbaum, ed.), PWS Publishing Co., pp. 144–191.
M. Grötschel and O. Holland (1985), 'Solving matching problems with linear programming', *Math. Program.* **33**, 243–259.
M. Halappanavar, J. Feo, O. Villa, F. Dobrian and A. Pothen (2012), 'Approximate weighted matching on emerging manycore and multithreaded architectures', *Internat. J. High Perf. Comput. Appl.* **26**, 413–430.
N. G. Hall and D. S. Hochbaum (1986), 'A fast approximation algorithm for the multicovering problem', *Discrete Appl. Math.* **15**, 35–40.
S. Hanke and S. Hougardy (2010), New approximation algorithms for the weighted matching problem. Research report 101010, Research Institute for Discrete Mathematics, University of Bonn.
D. S. Hochbaum, ed. (1997), *Approximation Algorithms for NP-hard Problems*, PWS Publishing Co.
J. Hogg and J. Scott (2015), 'On the use of suboptimal matchings for scaling and ordering sparse symmetric matrices', *Numer. Linear Algebra Appl.* **22**, 648–663.

J. Hogg and J. Scott (2013), 'Pivoting strategies for tough sparse indefinite systems', *ACM Trans. Math. Softw.* **40**, 4.

J. Hopcroft and R. Karp (1973), 'An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs', *SIAM J. Comput.* **2**, 225–231.

S. Hougardy (2009), Linear time approximation algorithms for degree constrained subgraph problems. In *Research Trends in Combinatorial Optimization* (W. J. Cook, L. Lovász and J. Vygen, eds), Springer, pp. 185–200.

B. C. Huang and T. Jebara (2011), Fast b-matching via sufficient selection belief propagation. In *Proc. 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 361–369.

D. Huang and S. Pettie (2017), Approximate generalized matching: f-factors and f-edge covers. arXiv:1706.05761

A. Idelberger and F. Manne (2014), New iterative algorithms for weighted matching. In *Norsk Informatikkonferanse 2014*. www.nik.no/publikasjoner/

T. Jebara and V. Shchogolev (2006), b-matching for spectral clustering. In *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)*, Vol. 4212 of Lecture Notes in Computer Science, Springer, pp. 679–686.

T. Jebara, J. Wang and S.-F. Chang (2009), Graph construction and b-matching for semi-supervised learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*, ACM, pp. 441–448.

D. Juedes and J. Jones (2012), 'Coloring Jacobians revisited: A new algorithm for acyclic and star bicoloring', *Optim. Methods Softw.* **27**, 295–309.

R. J. Kang and C. McDiarmid (2015), Colouring random graphs. In *Topics in Chromatic Graph Theory* (R. J. Wilson and L. W. Beineke, eds), Cambridge University Press, pp. 199–219.

R. M. Karp and M. Sipser (1981), Maximum matching in sparse random graphs. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (SFCS 1981)*, pp. 364–375.

R. M. Karp, U. Vazirani and V. Vazirani (1990), An optimal algorithm for on-line bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC '90)*, ACM, pp. 352–358.

E. R. Keiter, H. K. Thornquist, R. J. Hoekstra, T. V. Russo, R. L. Schiek and E. L. Rankin (2011), Parallel transistor-level circuit simulation. In *Advanced Simulation and Verification of Electronic and Biological Systems* (P. Li, L. M. Silveira and P. Feldmann, eds), Springer, pp. 1–21.

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

A. Khan, K. Choromanski, A. Pothen, S. Ferdous, M. Halappanavar and A. Tumeo (2018*a*), Adaptive anonymization of data using b-edge cover. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*, IEEE, pp. 59:1–59:11.

A. M. Khan, D. F. Gleich, A. Pothen and M. Halappanavar (2012), A multithreaded algorithm for network alignment via approximate matching. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC '12)*, IEEE, pp. 64:1–64:11.

A. Khan, A. Pothen and S. M. Ferdous (2018*b*), Parallel algorithms through approximation: *b*-edge cover. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, pp. 22–33.

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

A. Khan, A. Pothen, M. M. Patwary, N. Satish, N. Sundaram, F. Manne, M. Halappanavar and P. Dubey (2016*b*), 'Efficient approximation algorithms for weighted *b*-matching', *SIAM J. Sci. Comput.* **38**, S593–S619.

S. Khuller, U. Vishkin and N. Young (1994), 'A primal–dual parallel approximation technique applied to weighted set and vertex covers', *J. Algorithms* **17**, 280–289.

V. Knoblauch (2007), Marriage Matching: A conjecture of Donald Knuth. Working papers 2007-15, University of Connecticut, Department of Economics.

V. Kolmogorov (2009), 'BLOSSOM V: A new implementation of a minimum cost perfect matching algorithm', *Math. Prog. Comput.* **1**, 43–67.

P. Kolyvakis, A. Kalousis, B. Smith and D. Kiritsis (2018), 'Biomedical ontology alignment: An approach based on representation learning', *J. Biomed. Semantics* **9**, 21.

C. Koufogiannakis and N. E. Young (2011), 'Distributed algorithms for covering, packing and maximum weighted matching', *Distrib. Comput.* **24**, 45–63.

X. S. Li and J. W. Demmel (2003), 'SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems', *ACM Trans. Math. Softw.* **29**, 110–140.

L. Lovász and M. D. Plummer (2009), *Matching Theory*, AMS.

D. F. Manlove (2013), *Algorithmics of Matching Under Preferences*, World Scientific.

F. Manne and M. Halappanavar (2014), New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, pp. 519–528.

F. Manne, M. Naim, H. Lerring and M. Halappanavar (2016), On stable marriages and greedy matchings. In *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, SIAM, pp. 92–101.

F. M. Manshadi, B. Awerbuch, R. Gemulla, R. Khandekar, J. Mestre and M. Sozio (2013), 'A distributed algorithm for large-scale generalized matching', *Proc. VLDB Endowment* **6**, 613–624.

J. Maue and P. Sanders (2007), Engineering algorithms for approximate weighted matching. In *Experimental Algorithms: 6th International Workshop on Experimental and Efficient Algorithms (WEA 2007)*, Vol. 4525 of Lecture Notes in Computer Science, Springer, pp. 242–255.

S. T. McCormick (1983), 'Optimal approximation of sparse Hessians and its equivalence to a graph coloring problem', *Math. Program.* **26**, 153–171.

C. McDiarmid (1984), 'Colouring random graphs', *Ann. Oper. Res.* **1**, 183–200.

D. G. McVitie and L. B. Wilson (1971), 'The stable marriage problem', *Commun. Assoc. Comput. Mach.* **14**, 486–490.

K. Mehlhorn and S. Näher (1999), LEDA: A platform for combinatorial and geometric computing. www.algorithmic-solutions.com/leda/index.htm

A. Mehta (2012), 'Online matching and ad allocation', *Found. Trends. Theor. Comput. Sci.* **8**, 265–368.

N. S. Mendelsohn and A. L. Dulmage (1958), 'Some generalizations of the problem of distinct representatives', *Canad. J. Math.* **10**, 230–241.

J. Mestre (2006), Greedy in approximation algorithms. In *Algorithms: 14th Annual European Symposium on Algorithms (ESA 2006)*, Vol. 4168 of Lecture Notes in Computer Science, Springer, pp. 528–539.

S. Micali and V. V. Vazirani (1980), 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 (SFCS 1980)*, IEEE, pp. 17–27.

D. L. Miller and J. F. Pekny (1995), 'A staged primal–dual algorithm for perfect b-matching with edge capacities', *ORSA J. Comput.* **7**, 298–320.

M. Minoux (1978), Accelerated greedy algorithms for maximizing submodular set functions. In *Optimization Techniques: Proceedings of the 8th IFIP Conference on Optimization Techniques (IFIP 1977)* (J. Stoer, ed.), Springer, pp. 234–243.

R. Motwani (1994), 'Average-case analysis of algorithms for matchings and related problems', *J. Assoc. Comput. Mach.* **41**, 1329–1356.

M. Müller-Hannemann and A. Schwartz (2000), 'Implementing weighted b-matching algorithms: Insights from a computational study', *J. Exp. Algorithmics* **5**, 8.

R. C. Murphy, K. B. Wheeler, B. W. Barrett and J. A. Ang (2010), Introducing the Graph 500. In *Proceedings of the Cray User's Group Meeting (CUG), 2010.*

M. Naim and F. Manne (2018), Scalable b-matching on GPUs. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPS)*, pp. 637–646.

M. Naim, F. Manne, M. Halappanavar, A. Tumeo and J. Langguth (2015), Optimizing approximate weighted matching on Nvidia Kepler K40. In *IEEE 22nd International Conference on High Performance Computing (HiPC 2015)*, pp. 105–114.

A. Natanzon, R. Shamir and R. Sharan (2000), 'A polynomial approximation for the minimum fill-in problem', *SIAM J. Comput.* **30**, 1067–1079.

U. Naumann and O. Schenk, eds (2012), *Combinatorial Scientific Computing*, CRC Press.

R. Z. Norman and M. O. Rabin (1959), 'An algorithm for a minimum cover of a graph', *Proc. Amer. Math. Soc.* **10**, 315–319.

M. Olschowka and A. Neumaier (1996), 'A new pivoting strategy for Gaussian elimination', *Linear Algebra Appl.* **240** (suppl. C), 131–151.

M. W. Padberg and M. R. Rao (1982), 'Odd minimum cut-sets and b-matchings', *Math. Oper. Res.* **7**, 67–80.

S. Pettie and P. Sanders (2004), 'A simpler linear time $2/3 - \epsilon$ approximation for maximum weight matching', *Inform. Process. Lett.* **91**, 271–276.

A. Pinar, E. Chow and A. Pothen (2006), 'Combinatorial algorithms for computing column space bases that have sparse inverses', *Electron. Trans. Numer. Anal.* **22**, 122–145.

A. Pothen (1993), 'Predicting the structure of sparse orthogonal factors', *Linear Algebra Appl.* **194**, 183–203.

A. Pothen and C.-J. Fan (1990), 'Computing the block triangular form of a sparse matrix', *ACM Trans. Math. Softw.* **16**, 303–324.

R. Preis (1999), Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *1999 Proceedings of 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS 99)*, Vol. 1563 of Lecture Notes in Computer Science, Springer, pp. 259–269.

W. R. Pulleyblank (1973), Faces of matching polyhedra. PhD thesis, Faculty of Mathematics, University of Waterloo.

S. Rajagopalan and V. V. Vazirani (1993), 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 (SFCS '93)*, IEEE, pp. 322–331.

A. Schrijver (2003), *Combinatorial Optimization: Polyhedra and Efficiency*, Vol. A: *Paths, Flows, Matchings*, Springer.

R. Sinkhorn and P. Knopp (1967), 'Concerning nonnegative matrices and doubly stochastic matrices', *Pacific J. Math.* **21**, 343–348.

T. H. Spencer and E. W. Mayr (1984), Node weighted matching. In *Proceedings of the 11th Colloquium on Automata, Languages, and Programming (ICALP)*, Vol. 172 of Lecture Notes in Computer Science, Springer, pp. 454–464.

A. Subramanya and P. P. Talukdar (2014), *Graph-Based Semi-Supervised Learning*, Vol. 29 of Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool.

V. Tabatabaee, L. Georgiadis and L. Tassiulas (2001), 'QoS provisioning and tracking fluid policies in input queueing switches', *IEEE/ACM Trans. Netw.* **9**, 605–617.

A. Tamir and J. S. B. Mitchell (1998), 'A maximum b-matching problem arising from median location models with applications to the roommates problem', *Math. Program.* **80**, 171–194.

K. Tangwongsan (2011), Efficient parallel approximation algorithms. PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

V. V. Vazirani (2003), *Approximation Algorithms*, Springer.

D. P. Williamson and D. B. Shmoys (2011), *The Design of Approximation Algorithms*, Cambridge University Press.

L. B. Wilson (1972), 'An analysis of the marriage matching assignment algorithm', *BIT* **12**, 569–575.

M. Yannakakis (1981), 'Computing the minimum fill-in is NP-complete', *SIAM J. Algebraic Discrete Methods* **2**, 77–79.