RC 21576 (97320) October 19, 1999
Computer Science/Mathematics

IBM Research Report

Algorithms for Finding Maximum Matchings in Bipartite Graphs

Anshul Gupta

IBM Research Division
T.J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
anshul@watson.ibm.com

Lexing Ying

Department of Computer Science
New York University
251 Mercer Street
New York, NY 10012
lexing@cs.nyu.edu

IBM Research Division
Almaden · Austin · China · Delhi · Haifa · Tokyo · Watson · Zurich

Algorithms for Finding Maximum Matchings in Bipartite Graphs*

Anshul Gupta
IBM T.J.Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598
anshul@watson.ibm.com

Lexing Ying
Department of Computer Science
New York University
251 Mercer Street, New York, NY 10012
lexing@cs.nyu.edu

IBM Research Report RC 21576 (97320)

October 19, 1999

Abstract

In this paper we survey algorithms for finding the maximum cardinality and the maximum weight matching in a bipartite graph (V, E) with $|V|$ vertices and $|E|$ edges. We discuss the complexity of various algorithms for these problems, with an emphasis on algorithms for sparse graphs. We provide a detailed tutorial description of implementations of some promising algorithms, suggest improvements, and provide an experimental comparison of our code with other well-known codes on large sparse matrices. The improvements that we suggest significantly reduce the execution time of the original algorithm for finding the maximum weight matching in sparse bipartite graphs with the best-known strongly polynomial worst-case time complexity of $O(|V|(|E| + |V| \log |V|))$. We show that if (1) the graph can be partitioned into two parts by a node-separator of size $O(|V|^\alpha)$, where $0 \leq \alpha \leq 1$, (2) the number of nodes in the larger of the two partitions does not exceed $|V|/c$, where $c > 1$, and (3) both partitions of the graph recursively obey conditions (1) and (2), then the maximum-weight bipartite matching problem can be solved in $O(|V|^\alpha(|E| + |V| \log |V|))$ time. This is significant because a majority of graphs arising in real-world applications lend themselves to such partitioning in $O(|V| \log |V|)$ time.

Keywords: graph algorithms, bipartite matching, assignment problem, sparse matrices, linear systems.

*Last updated on January 19, 2000.

1 Introduction

Finding the maximum cardinality matching in a bipartite graph and the maximum weight matching in a weighted bipartite graph (an instance of the assignment problem) are important problems with many applications in operations research, combinatorial analysis, and solving sparse systems of linear equations. The application that motivated our study of these algorithms is the LU factorization of large sparse matrices. Numerical stability in the LU factorization is maintained through partial pivoting, which can have a negative impact on the factorization speed, significantly so on parallel computers. Row interchanges due to pivoting during factorization can unpredictably affect the nonzero structure of the factor, thus making it impossible to statically allocate and distribute data-structures among processors. Such interchanges and their side-effects can cause significant inter-processor communication and load-imbalance in a parallel implementation of LU factorization. Permuting the rows or the columns of the sparse matrix to ensure a non-zero diagonal or to maximize the product of the absolute values of the diagonal entries are among the techniques often used as preprocessing steps to LU factorization in order to reduce the number of dynamic pivoting steps [17, 16, 38, 39, 44]. Permuting to maximize the diagonal in a sparse coefficient matrix can also be a useful preprocessing step while solving a sparse linear system by an iterative method [16].

A sparse matrix can be represented by a bipartite graph. The two vertex sets of the bipartite graph correspond to the rows and columns of the matrix, respectively. If there is a nonzero value in location (x, y), then vertex x from the row set is connected by an edge to vertex y of the column set. If, for an $n \times n$ matrix, a matching of maximum cardinality (n) is found, then we have established that the matrix is structurally non-singular and we can permute the rows or the columns to place a nonzero entry at each diagonal location. In the weighted version of the problem, the edges in the bipartite graph are assigned weights equal to the absolute values of the corresponding matrix entries. Then finding the maximum cardinality maximum weight matching can lead to a permutation with not only a zero-free diagonal, but with a "heavy" diagonal, which further reduces pivoting during factorization. The algorithms for finding the maximum weight matching maximize the sum of the absolute values of the diagonal entries. However, by using the logarithm of the absolute values of the matrix entries, maximizing the product of the absolute values of the diagonal entries is straightforward and is more useful in practice in reducing partial pivoting during factorization [16, 38, 39].

In this paper we survey algorithms for finding maximum cardinality and maximum weight matching on bipartite graphs. We discuss the complexity of various algorithms, with an emphasis on algorithms for sparse graphs. We provide a detailed tutorial description of implementations of some promising algorithms, suggest improvements, and provide an experimental comparison of our code with other well-known codes on large sparse matrices. Some of the improvements suggested in this paper significantly reduce the execution time of the original algorithm for finding the maximum weight matching in a sparse bipartite graph $G = (V, E)$ with the best-known strongly polynomial worst-case time complexity of $O(|V|(|E| + |V| \log |V|))$. We show that if (1) the graph can be partitioned into two parts by a node-separator of size $O(|V|^\alpha)$, where $0 \leq \alpha \leq 1$, (2) the number of nodes in the larger of the partitions does not exceed $|V|/c$, where $c > 1$, and (3) both partitions of the graph recursively obey conditions (1) and (2), then the above problem can be solved in $O(|V|^{\alpha}(|E| + |V| \log |V|)$ time. This is significant because a majority of graphs arising in real-world problems lend themselves to such partitioning in $O(|V| \log |V|)$ time.

The paper is organized as follows. Section 2 introduces some terminology and notation used in the paper. Section 3 contains a survey of the algorithms on maximum cardinality and maximum weight matching algorithms. In Section 4, we describe in detail the Hopcroft-Karp algorithm [30] for finding maximum cardinality matching. In Section 5, we describe an algorithm due to Kuhn [36] and Munkres [43] for finding the maximum weight matching and present our implementation scheme. In Section 6, we describe another algorithm for finding a maximum weight matching that is due to Karp [33]. We provide experimental results on four different versions of the Kuhn/Munkres algorithm and compare them with those obtained on a commercial code MC64 from Harwell Subroutine Library [31] in Section 7. Section 8 contains concluding remarks.

2 Terminology and Notations

Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E. If an edge joins vertex u and v, we denote it as uv. If a path consists of edges $u_1u_2, u_2u_3, u_3u_4, \ldots, u_{k-1}u_k$ where u_i are all distinct, we denote it as $u_1u_2 \cdots u_{k-1}u_k$. A *bipartite graph* is a graph whose vertex set V can be divided into two sets X and Y and each edge joins one vertex from X and one from Y. We often denote a bipartite graph G as $((X, Y), E)$. A matching M is a set of edges such that no vertex in G is incident to more than one edge in M. A vertex is called *free* if it is not incident to any edge in M. If M is a matching, we use $w(M) = \sum_{xy \in M} w(xy)$ to denote the weight of the matching. An *alternating path* relative to a matching M is a path $P = u_1u_2 \cdots u_s$ where its edges are alternatively in $E \setminus M$ and M. An *augmenting path* relative to a matching M is an alternating path of odd length and both of its two endpoints are free. If graph G is a bipartite graph, one endpoint of any augmenting path must be in X and the other must be in Y. The symmetric difference, $A \oplus B$, of two sets A and B is defined to be $(A \setminus B) \cup (B \setminus A) = (A \cup B) - (A \cap B)$. Since $\oplus$ is associative ($(A \oplus B) \oplus C = A \oplus (B \oplus C)$), we can write $A_1 \oplus A_2 \oplus \cdots \oplus A_n$ without confusion.

3 Survey

In this section, we present a brief survey of sequential and parallel algorithms for maximum cardinality matching and maximum weight matching on bipartite graphs.

3.1 Finding a Matching with Maximum Cardinality

We usually use *maximum matching* to denote the matching with maximum cardinality. There is a theorem on maximum matching by Berge (see [4], [5]):

Theorem 3.1 *If M is not a maximum matching, there exists an augmenting path P relative to M, and $M \oplus P$ is a matching with size $|M| + 1$.*

According to this theorem, a way of finding a maximum matching in a bipartite matching is to seek augmenting paths. This algorithm is usually called *Hungarian Method* (see [5]). The following is a description of the algorithm and, at the end of it, M is the maximum matching.

```
M ← ∅;
for each vertex x ∈ X
    if x is free then
        search for an augmenting path P from x;
        M ← M ⊕ P;
    endif;
endfor;
```

Searching for an augmenting path from a free vertex $x \in X$ involves traversing the bipartite graph from x until we get a free vertex $y \in Y$. The time complexity of this algorithm is $O(|V||E|)$. In an actual implementation, we usually use depth first search rather than breadth first search for traversing because, in practice, the former visits fewer edges and turns out to be faster.

The asymptotically best result for maximum matching is accredited to Hopcroft and Karp (see [30]). Their idea was to find all augmenting paths with shortest length in two traversals of the bipartite graph. They proved that (1) shortest augmenting paths relative to a matching M are vertex disjoint, (2) if we always search for the shortest augmenting paths, then all augmenting paths found can only have $O(\sqrt{|V|})$ different lengths. Thus if we keep on searching for a maximal set of shortest augmenting paths, the time complexity is $O(\sqrt{|V|}|E|)$.

The comparison between the Hungarian method and Hopcroft-Karp algorithm is discussed in [18]. In that paper, Duff and Wiberg gave an implementation scheme and several useful heuristics. They showed that when the matrix is not large, the running time of Hungarian Method and Hopcroft-Karp algorithm is comparable; however, when matrix is large, the Hopcroft-Karp algorithm is faster. This experiment observation is consistent with the algorithmic analysis. Therefore, we choose Hopcroft-Karp algorithm for our implementation. A detailed description of Hopcroft-Karp algorithm will be provided in Section 4.

The maximum cardinality matching problem for bipartite graphs can easily be formulated as a maximum flow problem [1]. Cherkassky et al. [10] have performed a detailed computational comparison of the matching algorithms based on finding augmenting paths and maximum flows. Their conclusion is that the augmenting-paths algorithms are faster my a moderate constant factor for most classes of problems. Therefore, we do not consider flow based matching algorithms in this paper for the maximum cardinality matching problem.

Rabin and Vazirani [45] give a conceptually simple algorithm for finding a maximum cardinality matching. Their randomized algorithm works on a certain matrix representation of the given graph called *Tutte* matrix and has a worst-case time complexity $O(|V|^4)$.

Parallel algorithms for bipartite matching with maximum cardinality have also been developed. Most of them use the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM) as the parallel computation model. Schieber and Moran [46] provide an algorithm for general graphs. Their algorithm runs in $O(|V| \log |V|)$ time using $|E|$ processors. On bipartite graphs, a modified version of the algorithm can achieve running time $O(|V|)$ using $|V||E|$ processors. Goldberg et al. [26] propose a bipartite matching algorithm based on interior-point method that runs in $O(\sqrt{|E|}\log^3|V|)$ time using $|V|^3$ processors. Goldberg et al. [27] provide a sublinear-time algorithm for this problem. Their algorithm runs in $O(|V|^{2/3}\log^3|V|)$ time using $O(|V|^{2.5})$ processors. A detailed survey and description of parallel PRAM algorithms for solving the maximum-cardinality matching problem can be found in [34]. In all parallel algorithms proposed for this problem to date, the processor-time product is larger than $O(\sqrt{|V|}|E|)$—the serial time complexity of Hopcroft-Karp algorithm.

3.2 Finding a Maximum Weight Matching

The first algorithm for maximum weight matching on bipartite graphs (also known as the assignment problem) was proposed independently by Kuhn [36] and Munkres [43]. A detailed description can be found in [5, 37]. The original algorithm was described in the context of complete bipartite graphs; i.e., each pair $(x, y) \in X \times Y$ is joined. A modification of their algorithm for the sparse bipartite case can be found in the report [23] by Galil. Since our matrices of interest are sparse, we will focus only on this version of the algorithm.

We first introduce a label function l for vertices, a slackness function π for edges, and an important theorem from [23].

Definition l is a label function on the vertex set of bipartite graph $G = ((X, Y), E)$, iff for each $x \in X$, $l(x) \geq 0$, each $y \in Y$, $l(y) \geq 0$ and for each edge $xy \in E$ with weight $w(xy)$, $l(x) + l(y) - w(xy) \geq 0$.

Definition π is slackness function on edges relative to l, for each $xy \in E$, we define $\pi(xy) = l(x) + l(y) - w(xy)$. We usually call $\pi(xy)$ the slackness of edge xy.

Theorem 3.2 *Let M be a matching on a bipartite graph $G = ((X, Y), E)$. M is a maximum weight matching if there exists a label function l such that*

(1) If $xy \in M$, then $\pi(xy) = 0$.

(2) If $x \in X$ is free then $l(x) = 0$; if $y \in Y$ is free then $l(y) = 0$.

Proof: Suppose $U = \{x | \exists y : xy \in M\}$ and $V = \{y | \exists x : xy \in M\}$.

From (1) and (2), we get

$$\begin{aligned} \sum_{xy \in M} w(xy) &= \sum_{x \in U} l(x) + \sum_{y \in V} l(y) \\ &= \sum_{x \in X} l(x) + \sum_{y \in Y} l(y) \quad (\text{ because } \sum_{x \in X \setminus U} l(x) = 0, \sum_{y \in Y \setminus V} l(y) = 0). \end{aligned}$$

Assume that M' is any other matching. From the definition of the label function, we have that

$$\begin{aligned} \sum_{xy \in M'} w(xy) &\leq \sum_{x \in X} l(x) + \sum_{y \in Y} l(y) \\ &= \sum_{xy \in M} w(xy). \end{aligned}$$

So M is the maximum weight matching. $\square$

The algorithm is based on Theorem 3.2 and starts with a trivial solution that violates some of constraints (2). Then the algorithm works in a way that the number of constraints being violated is reduced. At the end, when no constraint is violated, the matching has the maximum weight. Usually we starts with solution $M = \emptyset$, $l(x) = max_{xy' \in E}\{w(xy')\}$ for all $x \in X$ and $l(y) = 0$ for all $y \in Y$. The worst-case complexity of this algorithm is $O(|V||E|\log|V|)$ (Section 5.4), which can be improved, at least theoretically, to $O(|V|(|E| + |V|\log|V|))$ (Section 5.5.1).

There are already a few implementations of this algorithm. Carpaneto and Toth [7, 8, 9], Burkard and Derigs [6], and McGinnis [40] provide Fortran implementations of algorithms for solving dense and sparse assignment problems. Variations of the basic Kuhn/Munkres algorithm have also been proposed in [6, 12, 13, 32]. Olschowka and Neumaier [44] discuss the assignment problem in the context of partial pivoting for LU factorization. Duff and Koster [16] describe an implementation for sparse matrices. Their implementation (Subroutine MC64) is a part of the Harwell Subroutine Library [31]. The complexity of that implementation is $O(|V||E|\log|V|)$.

Karp [33], introduced an algorithm which runs in $O(|V|^2 \log|V|)$ expected time with the assumption that the edge costs are independent random variables and the costs of the edges incident with any given vertices are identically distributed. A detailed description is provided in Section 6. As we shall discuss in Section 6, although the asymptotic complexity of Karp's algorithm is smaller than $O(|V||E|\log|V|)$, the constants associated with this algorithm are much higher and the $O(|V||E|\log|V|)$ smallest augmenting-paths based implementations run much faster in practice.

The assignment problem can be reduced to a transportation problem, or a minimum-cost maximum-flow problem. $O(|V||E|\log|V|)$ algorithms based on flow methods can be found in [19, 20]. Gabow and Tarjan [22] proposed an $O(|E|\sqrt{|V|}\log(|V|K))$ algorithm using cost-scaling and blocking flow techniques under the assumption that the edge-weights are integers in the range $[-K, \ldots, K]$. Goldberg and Kennedy [25] improved the original Gabow-Tarjan algorithm with several heuristics and performed a detailed experimental evaluation of this algorithm. In Section 7, we compare the results of Goldberg and Kennedy's code with that of the smallest augmenting-paths based codes such as MC64 and our own code. Just like the maximum cardinality matching problem, even for the maximum weight matching problem, the augmenting-paths based methods seem to outperform the flow algorithms for graphs arising in real applications. Also, in the context of sparse LU factorization, the augmenting-paths methods generate label vectors corresponding to the rows and columns, which are needed to scale the matrix prior to factorization. It is not clear if these scaling vectors can be computed cheaply using flow algorithms to solve the maximum weight matching problem. Therefore, the augmenting-paths based methods appear to be more suitable for our application.

In [2], Avis and Lai propose a heuristic that obtains a relatively fast approximate solution to the assignment problem that is within a small constant factor of the optimal solution with a high probability.

Parallel algorithms for maximum weight matching problem have been developed along with the parallel algorithms for maximal cardinality matching. Goldberg et al. [26] propose an algorithm dealing with the bipartite graph with only integral weights. The running time is $O(\sqrt{|E|}\log^2|V|\log(|V|C))$, where $C > 1$ is the upper bound on the absolute value of the integral weights on edges. Goldberg et al. [27] present a sublinear-time algorithm which runs in $O(|V|^{2/3}\log^3|V|\log(|V|C))$ time using $O(|V|^3)$ processors. Both algorithms use CRCW PRAM as parallel model. As far as practical parallel algorithms are concerned for the sparse bipartite assignment problem, the work by Balas et al. [3] is promising; however, no complexity analysis is presented in [3].

4 The Hopcroft-Karp Algorithm

The Hopcroft-Karp algorithm works by continually searching for shortest augmenting paths and terminating when no such path can be found. Suppose we start from the matching $M_0 = \emptyset$, and we compute sequences $\{M_0, M_1, M_2, \ldots, M_i, \ldots\}$ and $\{P_0, P_1, P_2, \ldots, P_i, \ldots\}$ where P_i is the shortest augmenting path relative to M_i, and $M_{i+1} = M_i \oplus P_i$.

In [30], Hopcroft and Karp proved some important results, which we reproduce below in the form of Theorem 4.1 to Theorem 4.4:

Theorem 4.1 *Let M and N be matchings on $G = (V, E)$, If $|M| = r$, $|N| = s$ and $s > r$, then $M \oplus N$ contains at least $s - r$ vertex disjoint augmenting paths relative to M.*

Proof: Consider the graph with V as vertex set and $M \oplus N$ as edge set. Since M and N are matchings, each vertex is incident with at most one edge from $N \setminus M$ and one edge from $M \setminus N$. So each component of this graph is

1. an isolated vertex
2. a cycle of even length with edges alternatively in $M \setminus N$ and $N \setminus M$, or
3. a path whose edges are alternatively in $M \setminus N$ and $N \setminus M$.

The components in the first two categories do not contribute to the difference between the size of N and M — $(s - r)$. Among the paths in the last category, if a path has an even number of edges, it does not contribute either. If it has an odd number of edges, then there are two cases: (1) If it is an augmenting path relative to M, then it has one more edge in N than in M. (2) If it is an augmenting path relative to N, it has one more edge in M than in N. So $s - r$ is the difference between the number of augmenting paths relative to M and the number of those relative to N. Thus, the number of augmenting path relative to M is at least $s - r$. □

Theorem 4.2 *Let M be a matching, P a shortest augmenting path relative to M, and P' an augmenting path relative to $M \oplus P$. Then*

$$|P'| \geq |P| + 2|P \cap P'|.$$

Proof: Let $N = M \oplus P \oplus P'$, from Theorem 3.1. N is a matching with size $|M| + 2$. Then $M \oplus N$ contains two vertex disjoint augmenting paths (relative to M): P_1 and P_2. Since $M \oplus N = P \oplus P'$, then $|P \oplus P'| \geq |P_1| + |P_2|$. Since P is shortest augmenting path, $|P \oplus P'| \geq 2|P|$. Also $|P \oplus P'| = |P| + |P'| - 2|P \cap P'|$. Therefore, $|P'| \geq |P| + 2|P \cap P'|$. □

An immediate corollary is that $|P_i| \leq |P_{i+1}|$.

Theorem 4.3 *For all i and j such that $|P_i| = |P_j|$, then P_i and P_j are vertex-disjoint.*

Proof: We use contradiction to prove the theorem. Suppose $|P_i| = |P_j|$, $i < j$, and P_i and P_j are not vertex disjoint. Then there exist k and l such that $i \leq k < l \leq j$, P_k and P_l are not vertex disjoint, and for each m, $k < m < l$, P_m is vertex-disjoint from P_k and P_l. Then P_l is an augmenting path relative to $M_k \oplus P_k$, so $|P_l| \geq |P_k| + |P_k \cap P_l|$. But $|P_l| = |P_k|$, and $|P_k \cap P_l| = 0$. Thus P_k and P_l have no edges in common. Now suppose P_k and P_l had a vertex v in common. Since v is not free in $M_k \oplus P_k$, it cannot be the endpoint of P_l. Then v must be incident with one matched edge in P_l. But since this matched edge is also in P_k, P_k and P_l must share one common edge. Hence, P_k and P_l are vertex disjoint, and a contradiction is obtained. □

Theorem 4.4 *Let s be the cardinality of a maximum matching. The number of distinct integers in the sequence*

$$|P_0|, |P_1|, \ldots, |P_i| \ldots$$

is less than or equal to $2\lfloor\sqrt{s}\rfloor + 2$.

Proof: Let $r = \lfloor s - \sqrt{s} \rfloor$. Then $|M_r| = r$. Let N be the maximum matching. According to Theorem 4.1, $M_r \oplus N$ contains at least $s - r$ vertex disjoint augmenting paths relative to M_r. These augmenting paths totally contain at most r edges from M_r, so one of them must contain at most $\lfloor r/(s-r) \rfloor$ edges from M_r. Therefore there are at most $2\lfloor r/(s-r)\rfloor + 1$ edges in this path. And

$$|P_r| \leq \frac{2\lfloor s - \sqrt{s}\rfloor}{(s - \lfloor s - \sqrt{s}\rfloor)} + 1 \leq 2\lfloor\sqrt{s}\rfloor + 1.$$

For each $i \leq r$, $|P_i|$ is one of the $\lfloor\sqrt{s}\rfloor + 1$ positive odd integers less than or equal to $2\lfloor\sqrt{s}\rfloor + 1$. Also $|P_{r+1}|, \ldots, |P_s|$ contribute at most $s - r = \lceil\sqrt{s}\rceil$ distinct integers, and the total number of distinct integers is less than or equal to $\lfloor\sqrt{s}\rfloor + 1 + \lceil\sqrt{s}\rceil \leq 2\lfloor\sqrt{s}\rfloor + 2$. □

From these theorems, we can break the computation of maximum matching into at most $2\lfloor\sqrt{s}\rfloor + 2$ stages. In each stage, we search for a maximal set of augmenting paths that are vertex disjoint and of shortest length. Also, since they are vertex disjoint, they are all relative to the matching with which the stage is begun. The algorithm is as follows:

$M \leftarrow \emptyset$;
while true
    Get length l of shortest augmenting path;
    if no augmenting path exists *then*
        return;
    endif

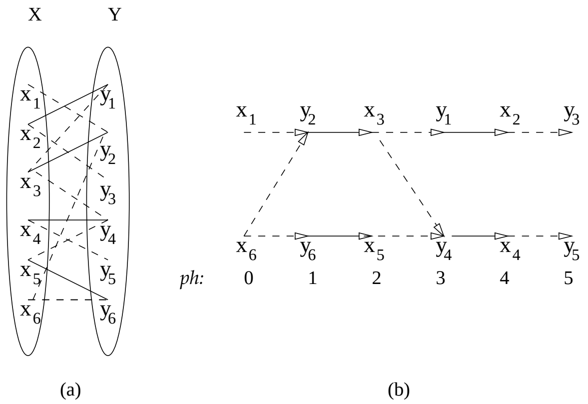


Figure 1: (a) A example bipartite graph with a matching (shown by solid edges), (b) The visited vertices are divided into phases and the edges join vertices in adjacent phases, the search is from $ph_0 = \{x_1, x_6\}$, y_3 and y_5 are free vertices in ph_5.

    Find a maximal set of paths $\{Q_1, \ldots, Q_t\}$ of length l and
        vertex disjoint;
    $M \leftarrow M \oplus Q_1 \oplus Q_2 \oplus \cdots \oplus Q_t$;
endwhile

In order to get the shortest length of augmenting path relative to existed matching M, we use breadth first search. The algorithm is described as follows:

$ph_0 \leftarrow \{x \in X | x \text{ is free}\}$;
$i \leftarrow 0$;
while true
    $ph_{2i+1} \leftarrow \{y \in Y | y \notin \bigcup_{0 \le k \le 2i} ph_k \wedge \exists x \in ph_{2i} : xy \in E \setminus M\}$;
    if some $y \in ph_{2i+1}$ is free *then*
        break;
    else (every $y \in ph_{2i+1}$ is not free)
        $ph_{2i+2} \leftarrow \{x \in X | x \notin \bigcup_{0 \le k \le 2i+1} ph_k \wedge \exists y \in ph_{2i+1} : xy \in M\}$;
        $i \leftarrow i + 1$;
    endif
endwhile

This search is performed in phases. Since the graph is bipartite, vertices of ph_i belong to X when i is even, and belong to Y when i is odd. In phase 0, we put all the free vertices of X into ph_0. Since the edges from vertices in ph_{2k} to vertices in ph_{2k+1} are not in matching M and those from vertices in ph_{2k+1} to vertices in ph_{2k+2} are in matching M, then for each i the distance of shortest alternating path from the vertices in ph_i to any free vertex in X (i.e., any vertex in ph_0) is just i. We stop after the phase in which at least one free vertex in Y is reached. The index of this phase is the length of shortest augmenting path. An example is shown in Figure 1. We begin from vertices $ph_0 = \{x_1, x_6\}$, and at phase 5, we have free vertices y_3 and y_5 from Y in $ph_5 = \{y_3, y_5\}$.

The next task in the *while* loop is to find a maximal set of augmenting paths. Let $s = min\{i | \exists \text{ free vertex } y \in ph_i\}$, i.e., s is the length of shortest augmenting path. Then consider the graph G' with vertex set $V' = \{v \in \bigcup_{0 \leq i \leq s} ph_i\}$ and edge set $E' = \{xy \in E \setminus M | x \in ph_{2i} \text{ and } y \in ph_{2i+1}\} \cup \{xy \in M | y \in ph_{2i+1} \text{ and } x \in ph_{2i+2}\}$. In the example of Figure 1, G' is figure (b). Since vertices are divided into phases and edges join vertices in phase i and $i+1$ for $0 \leq i < s$, G' is layered graph. Every shortest augmenting path relative to M is now a path from ph_0 to ph_s in G'. Finding a maximal set of shortest augmenting paths is equivalent to finding a maximal set of paths from ph_0 to ph_s. We only need to use one pass of depth first search from vertices in ph_0 to achieve that because of the following fact: Each edge processed in depth first search either becomes part of the path constructed or will never be used in any augmenting path in the maximal set. In either case each edge needs to be visited only once.

The proof is easy: suppose edge $v_k v_{k+1}$ is an edge from ph_k to ph_{k+1} in G'. In the depth first search, suppose when we reach $v_k v_{k+1}$, the path we kept in stack is $v_0 v_1 v_2 \cdots v_k$ where $v_i \in ph_i$. Suppose $v_k v_{k+1}$ cannot be an edge in any path with $v_0 v_1 v_2 \cdots v_k$ as prefix but it can be an edge in another path $v'_0 v'_1 \cdots v'_s$ where $v'_i \in ph_i$ for each i and $v'_k = v_k$ $v'_{k+1} = v_{k+1}$. However, $v_0 v_1 \cdots v_k (v'_k) v_{k+1} (v'_{k+1}) v'_{k+2} \cdots v'_s$ is also a path from ph_0 to ph_s in G' and this is contradictory to the assumption that $v_0 v_1 v_2 \cdots v_k$ cannot be the prefix of any path. Considering the example of Figure 1, the maximal sets of shortest augmenting paths can be $\{x_1 y_2 x_3 y_1 x_2 y_3, x_6 y_6 x_5 y_4 x_4 y_5\}$, $\{x_1 y_2 x_3 y_4 x_4 y_5\}$, $\{x_6 y_2 x_3 y_4 x_4 y_5\}$, or $\{x_1 y_2 x_3 y_1 x_2 y_3\}$.

In the depth-first search based implementation of the Hungarian method, a step called *cheap assignment* is widely used. Before we begin depth first search from one free vertex $x \in X$, we first check whether there is a free vertex $y \in Y$ joined with x. If so, we get a augmenting path of length 1 immediately. Otherwise, we do depth first search beginning from x. It is essentially a combination of a 1-step breadth first search and a depth first search in order to get the advantage of both methods. In Hopcroft-Karp algorithm, since in the first iteration we can get a maximal set of augmenting paths of length 1 if they exist, and in latter iterations all augmenting paths have lengths greater than 1, cheap assignment is not necessary.

5 The Kuhn/Munkres Algorithm

In this section, we present the details of Kuhn/Munkres algorithm and our implementation scheme. The method we described in Section 3.2 aims to find the maximum weight matching. However, what we actually need for LU factorization is the maximum weight matching among the matching with maximum cardinality. We solve this problem as follows: Let $\mu = \sum_{x \in X} \max_{xy \in E} \{w(xy)\}$. Then μ is a upper bound of the weight of any matching. We introduce a new weight function w', where for each xy, $w'(xy) = w(xy) + \mu$. Then we use Kuhn/Munkres algorithm on this weight function.

5.1 Algorithm Description

Recall that in Section 3.2, we introduced the label function, the slackness function, and Theorem 3.2. The theorem said that a matching M is the maximum weight matching of bipartite graph $G = ((X, Y), E)$ iff there exists a label function l on that graph G satisfying the following two constraints:

(1) if $xy \in M$, then $\pi(xy) = 0$,

(2) if $x \in X$ is free $l(x) = 0$, if $y \in Y$ is free $l(y) = 0$.

We also pointed out that the Kuhn/Munkres algorithm begins with a trivial label function that initially violates some constraints, and during the iterations of the algorithm, we decrease the number of violations. Let $X = \{x_1, x_2, \ldots, x_i, \ldots\}$ and $Y = \{y_1, y_2, \ldots, y_i, \ldots\}$. For any $S \subset X$, we define $N(S) = \{y \in Y | \exists x \in S \; \exists xy \in E \; : \pi(xy) = 0\}$. The detailed algorithm is as follows:

00 $M \leftarrow \emptyset$;
01 $l(x) \leftarrow max_{xy' \in E}\{w(xy')\}$ for each $x \in X$;
02 $l(y) \leftarrow 0$ for each $y \in Y$;
03 *for* $i = 1, \ldots, |X|$
04   *if* x_i is free *then*
05         $S \leftarrow \{x_i\}$, $T \leftarrow \emptyset$;
06   *while true*
07   *if* $N(S) \supset T$ *then*
08   pick $y \in N(S) \setminus T$;
09   *if* y is free *then*
10                     $P \leftarrow$ augmenting path from x_i to y;
11                     $M \leftarrow M \oplus P$, *break*;
12   *else* (y is not free and $\exists z : yz \in M$)
13                     $S \leftarrow S \cup \{z\}$ and $T \leftarrow T \cup \{y\}$, go to *line 06*;
14   *endif*
15   *else* ($N(S) = T$)
16   Let $\delta^1 = min_{x \in S}\{l(x)\}$
17                         $\delta^2 = min_{x \in S, y \in Y \setminus T}\{\pi(xy)\}$
18                         $\delta = min(\delta^1, \delta^2)$;
19                 $l(x) \leftarrow l(x) - \delta$ for $x \in S$;
20                 $l(y) \leftarrow l(y) + \delta$ for $y \in T$;
21   *if* $\delta = \delta^1 = l(x')$ for some $x' \in S$ *then*
22                     $P \leftarrow$ alternating path from x_i to x';
23                     $M \leftarrow M \oplus P$, break;
24   *else* ($\delta = \delta^2 = \pi(x'y')$ for some edge $x'y'$)
25   *if* y' is free *then*
26                         $P \leftarrow$ augmenting path from x_i to y';
27                         $M \leftarrow M \oplus P$, break;
28   *else* (y' is not free and $\exists z' : y'z' \in M$)
29                         $S \leftarrow S \cup \{z'\}$ and $T \leftarrow T \cup \{y'\}$, go to *line 06*;

```
                endif
              endif
            endif
        endwhile
    endif
endfor
```

In Figure 2, we provide an example of Kuhn/Munkres algorithm on a bipartite graph. We use a *stage* to denote a particular iteration of the outermost *for* loop. At the beginning of the algorithm, we initialize $M_0 = \emptyset$, $l(x) = max_{xy' \in E}\{w(xy')\}$ for all $x \in X$ and $l(y) = 0$ for all $y \in Y$. Then the algorithm works in $|X|$ stages. In stage i, if x_i is free then the algorithm begins with x_i and searches for an augmenting path composed of edges with $\pi(xy) = 0$. We denote by S all vertices visited in X by S and all vertices visited in Y by T during the search of augmenting path. At the beginning of the searching for augmenting path, $S = \{x_i\}$ and $T = \emptyset$. We also introduce $N(S)$ to be the vertices in Y that are joined to vertices in S by edges with slackness 0. In our implementation, we use breadth first search for augmenting path because it can keep the path from root to any visited vertex and thus make easy the work of changing matching. During the search of the augmenting path, until the path is found,

1. If we find $N(S) \supset T$ (which means that there are still vertices in Y linked to vertices in S by edges with slackness 0 not yet visited), then we pick an arbitrary y from $N(S) \setminus T$. Now,
   (a) If y is free, then we have found an augmenting path P from x_i to y. The next step is to change M to $M \oplus P$ and go to next stage. This is what happened in stage 1 of the example (corresponding to the transition between Figure 2(b) and Figure 2(c)) where $M = \emptyset$ and $P = x_1y_1$. In Figure 2(b), $S = \{x_1\}$, $T = \emptyset$ and $N(S) = \{y_1\}$.
   (b) Otherwise, for some $z \in Z$, suppose $yz \in M$. Then put z into S, put y into T, and keep on the searching for augmenting path. At the beginning of stage 2 (in Figure 2(c)), $S = \{x_2\}$, $N(S) = \{y_1\}$ and $T = \emptyset$. We choose y_1 from $N(S) \setminus T$, then put y_1 into T and x_1, which is joined to y_1 by an edge in the matching, into S.

2. If we find $N(S) = T$ (which means that all the vertices in Y which are joined to vertices in S by edges with slackness 0 have already been visited), then we define $\delta^1 = min_{x \in S}\{l(x)\}$, $\delta^2 = min_{x \in S, y \in Y \setminus T}\{\pi(xy)\}$ and $\delta = min(\delta^1, \delta^2)$. Then we change the label function by decreasing the label value of each vertex in S by δ and increasing the label value of each vertex in T by δ. Now there are two possibilities:
   (a) Either $\delta = \delta^1 = l(x')$ for some $x' \in S$. Then the path P from x_i to x' is an alternating (not augmenting) path. If this is the case, we let M be $M \oplus P$ and go to next stage. In stage 4 (the transition between (e) and (f) in Figure 2), when $S = \{x_1, x_2, x_3, x_4\}$, $N(S) = T = \{y_1, y_2, y_3\}$, we have $\delta = \delta^1 = l(x_3) = 3$. After we change the label value, $l(x_3) = 0$. Here P is $x_4y_3x_1y_2x_3$ and x_3 is free relative to the new matching.

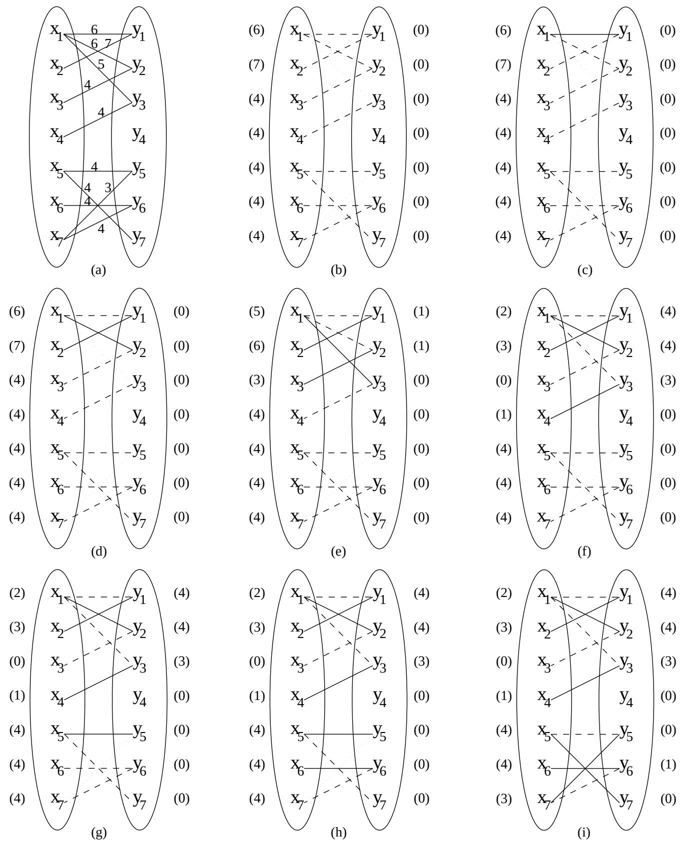


Figure 2: (a) is an example bipartite graph with weight value on edge. From (b) to (i), edges in matching are solid lines, edge with $\pi(xy) = 0$ are dashed line and all other edges are dotted line. We also denote the label value besides the vertices. (b) The initial label value. (c) The label value and matching after stage 1. In stage 1 an augmenting path x_1y_1 was found. (d) The label value and matching after stage 2. In stage 2 an augmenting path $x_2y_1x_1y_2$ was found. (e) The changed label value and matching after stage 3. In stage 3 augmenting path $x_3y_2x_1y_3$ was found. After the change on label function $\pi(x_1y_3) = 0$ and thus this edge can be used in the augmenting path. (f) The changed label value and matching after stage 4. In stage 4 an alternating path $x_4y_3x_1y_2x_3$ is found. Now $l(x_3) = 0$ and it is a free vertex. (g) The label value and matching after stage 5. In stage 5 an augmenting path x_5y_5 was found. (h) The label value and matching after stage 6. In stage 6 an augmenting path x_6y_6 was found. (i) The label value and matching after stage 7. In stage 7, an augmenting path $x_7y_5x_5y_7$ is found. After the change of label function, $\pi(x_7y_5) = 0$ and thus this edge can be reached and added into augmenting path. The label value and matching in (i) meet the constraints in Theorem 2.2, so matching in (i) is the maximum weight matching.

(b) Or $\delta = \delta^2 = \pi(x'y')$ for some edge $x'y'$. If y' is free, the path P from x_i to y' is an augmenting path. We let M be $M \oplus P$ and go on next stage. This case appears in stage 3 (the transition between (d) and (e) in Figure 2). At that point, $S = \{x_1, x_2, x_3\}$, $N(S) = T = \{y_1, y_2\}$, and we have $\delta = \delta^2 = \pi(x_1 y_3) = 1$. After we change the label value, $\pi(x_1 y_3) = 0$ and $x_3 y_2 x_1 y_3$ is an augmenting path P.

If y' is not free, then for some $z' \in X, y'z' \in M$. Now we just insert z' into S, insert y' into T and keep on the searching for augmenting path. This case appears in stage 7 (the transition between (h) and (i) in Figure 2). At that point, we have $S = \{x_6, x_7\}$ and $N(S) = T = \{y_6\}$. Since $\delta = \delta^2 = \pi(x_7 y_5) = 1$ and y_5 is not free, we add y_5 into T and add x_5, the vertex joined with y_5 by an edge in matching, into S. It turns out that through this edge we find the augmenting path $x_7 y_5 x_5 y_7$.

After all the $|X|$ stages are finished, the matching M is the maximum weight matching. We now prove the following theorem which states the correctness of the algorithm above.

Theorem 5.1 *Let $X_i = \{x_1, x_2, \ldots, x_i\}$, $G_i = ((X_i, Y), E \cap X_i \times Y)$ and M_i be the matching after stage i. After stage i, the matching M_i is the maximum weight matching of bipartite graph G_i.*

Proof: Because of Theorem 3.2, we only need to prove that after stage i for each i, function l still remains a label function and meets the constraints (1) and (2) in Theorem 3.2 relative to G_i and M_i.

Now let us consider the label function l first, we change label function only when $N(S) = T$. We claim that l remains a label function after the change of label value, because

1. Only the label value of vertex in S has decreased, but since the decreasing amount is bounded by $\delta^1 = min_{x \in S}\{l(x)\}$ from below. Therefore, after changing, the label value is also greater than 0.

2. (a) If an edge xy is in $S \times T$, then after changing, $\pi(xy)$ is still 0.
   (b) If xy is in $S \times (Y \setminus T)$, since the decreasing value is bound by $\delta^2 = min_{x \in S, y \in Y \setminus T}\{\pi(xy)\}$, $\pi(xy)$ is still greater than or equal to 0.
   (c) If xy in $(X_i \setminus S) \times T$, then $\pi(xy)$ increases.
   (d) If xy in $(X_i \setminus S) \times (Y \setminus T)$, $\pi(xy)$ does not change.

   Therefore, for any edge xy, $\pi(xy)$ is still greater than or equal to 0 after the changing of label value.

Since in the initial matching $l(y) = 0$ for each $y \in Y$ and whenever any $y \in Y$ is matched it cannot be free in the future, all free $y \in Y$ have $l(y) = 0$.

Now we only need to prove $l(x) = 0$ for all free $x \in X_i$ and if $xy \in M_i, \pi(xy) = 0$. We use induction to prove this result.

1. **Base Step**: When $i = 0$, the result is trivial.

2. **Induction Assumption**: Suppose for each $i < k$, M_i is the maximum weight matching for G_i.

3. **Induction Step**: If in M_k, vertex x_k is free, then before this stage stops, $\delta = \delta^1 = l(x_k)$, and $l(x_k)$ is set to 0. If x_k is not free, there are two cases. If we find an augmenting path P, since the edges of P is in the breadth first search tree, then for each edge $x'y' \in P$, we have $\pi(x'y') = 0$, so constraint (1) is met. Since no vertex in X_k turns to be free, so constraint (2) is also met according to induction assumption. The other case: for some vertex $x' \in X_k$, $\delta = \delta^1 = l(x')$, then since the path P from x_i to x' is in the breadth first search tree, for each edge $x'y' \in P$, we have $\pi(x'y') = 0$, so constraint (1) is met. After the change of label function, x' is free now and $l(x')$ turns to be 0, so constraint (2) is also met.

So for each i, M_i is the maximum weight matching for G_i. And when $i = |X|$, $G_i = G$ so $M_{|X|}$ is actually the maximum weight matching for G. □

5.2 Heap and Offset

The two costly operations in the algorithm described are calculating δ and changing the label function l according to δ. These operations, if implemented carefully [37, 23, 24], can be performed fairly efficiently.

We maintain the edges with $\pi > 0$ in a heap in order to find δ. Elements in heap are either (1) the slackness of an edge which is greater than 0, or (2) the label value of $x \in S$. Whenever we visit a vertex $x \in X$, i.e., $S \leftarrow S \cup \{x\}$, we put $\{\pi(xy) | y \in Y \setminus T\}$ and $l(x)$ into the heap. When we need to calculate δ, we just delete the minimum element from the heap. Since set S and T evolve along with the search of augmenting path, sometimes the edge with value δ extracted from heap may turn out to be in $S \times T$, so we need to check the element we get from the heap first. If the edge we got is in $S \times T$, we just discard it and delete the minimum element again.

According to the algorithm, after we calculate δ, we need to change the label function. Since we have introduced the heap, we need to change the value of elements in it as well. In order to make these changes efficiently, we introduce a variable, *offset*, which keeps the sum of δs by which we alter the label values. Suppose, in one stage, the sequence of δs by which we change the label values is $\{\delta_1, \delta_2, \ldots, \delta_q\}$. Let offset_1, offset_2, …, offset_q be the past values *offset*, where $\mathit{offset}_i = \sum_{1 \le k \le i} \delta_k$—the value of *offset* after the ith change. We also keep the following invariance: *The value of element in heap is its actual value plus the current value of offset.*

Suppose vertex $x \in X$ is inserted into S before the ith change, then at the end of this stage, the change of $l(x)$ is $\sum_{i \le k \le q} \delta_k$ which is equal to $\mathit{offset}_q - \mathit{offset}_{i-1}$. Now all we need to do is to add offset_{i-1} to $l(x)$ at the time x is inserted into S and subtract offset_q from $l(x)$ when this stage finished.

If vertex $y \in Y$ is inserted into T just before the ith change, then at the end of this stage, the total change of $l(y)$ is $\sum_{i \le k \le q} \delta_k$ which is also equal to $\mathit{offset}_q - \mathit{offset}_{i-1}$. Therefore, we subtract offset_{i-1} from $l(y)$ at the time y is inserted into T and add offset_q to $l(y)$ when the stage finishes.

Before we make the ith change on the label function, we must delete the entry with the least value from the heap. But from the invariance, this value is not δ_i, it is actually $\mathit{offset}_{i-1} + \delta_i$

(because before change i, the current *offset* is $offset_{i-1}$). However, this is exactly equal to $offset_i$, the value that we need to keep in *offset* after the ith change. When we make the ith change on the label value of $x \in S$ and $y \in T$, the slackness values of edges in $S \times (Y \setminus T)$ decrease by δ_i. So the values in heap should decrease by δ_i. However, since *offset* increases by δ_i, then the invariance allows us to keep the values in the heap unchanged.

5.3 Similarity with the Single-Source Shortest Path Problem

Once the label function is implemented the way described in Section 5.2, the i-th stage of the algorithm is essentially reduced to finding a shortest path from vertex x_i in X (if x_i is free) to any free vertex in Y in a directed graph $\tilde{G} = (V, \tilde{E})$, where $\langle \tilde{x}\tilde{y} \rangle \in \tilde{E}$ if $xy \notin M_{i-1}$ (M_{i-1} is the matching found after stage $i-1$) and $\langle \tilde{y}\tilde{x} \rangle \in \tilde{E}$ if $xy \in M_{i-1}$. The weight of each edge in $\tilde{G}$ is the slackness of that edge in G. Now each stage can be implemented by appropriately modifying Dijkstra's single source shortest path algorithm [14]. A best-first search is performed starting at x_i. alternating levels of the search tree have nodes from sets X and Y, respectively. The out-degree of each node at level with nodes from set Y in this tree is one because each outbound edge from such a level must be a part of M_{i-1}. The process is terminated as soon as the current shortest path includes a free node from Y. The unique path in the search tree to this node from x_i is the desired augmenting path for stage i.

5.4 Complexity Analysis

Now let us discuss the time complexity of our implementation. In each stage,

1. We visit each edge at most once, so each edge is inserted or deleted at most once. Therefore, the total number of heap operation is $O(|E|)$. Since each heap operation is logarithmic to the size of heap, which is $O(|E|)$, the heap operations cost time $O(|E|) \times O(\log |E|) = O(|E| \log |E|) = O(|E| \log |V|)$.

2. The operations for changing the label value of vertices is $O(|V|)$. For each vertex $x \in X$, if it is inserted into S, we decrease its label value by the *offset* at the point of insertion. And at the end of each stage, we increase the label value of each vertex in S by *offset* at that time. So the label value of any vertex in X is changed at most twice and the number of operations for changing label value of vertices in X is $O(|V|)$. This is also true for Y for the same reason.

3. The number of operations on slackness of edges is $O(|E|)$, because we only calculate slackness value of any edge when it is inserted into heap.

Thus, the time complexity of each stage is $O(|E| \log |V|)$. Considering there are a total of $|V|$ stages, the overall complexity is $O(|V||E| \log |V|)$.

5.5 Improvements for a Faster Implementation

While implementing the Kuhn/Munkres, several techniques can be used to significantly reduce the running time of the algorithm for most matrices. Many of these improvements have been proposed in the literature earlier. Although most of these do not reduce its worst-case asymptotic complexity for general bipartite graphs, one of them reduces the worst-case complexity for a special, but practically significant, class of graphs (Section 5.5.8).

In Section 7, we will present experimental data showing the effect of some of these techniques.

5.5.1 Using advanced data structures

As described in Section 5.3, each stage of the Kuhn/Munkres algorithm can be reduced to the application of Dijkstra's single source shortest path algorithm. This is a very well studied algorithm and new data structures have been proposed to reduce the complexity of this algorithm. Notable among these are Fibonacci Heaps [11, 21] and Relaxed Heaps [15]. The use of these data structures would reduce the complexity of each stage to $O(|V||E| + |V|\log|V|)$. Although asymptotically smaller that the worst-case of using a simple heap, the actual cost of implementing these complex data structures could easily outweight the advantages in an implementation. The main reason for this is that the actual number of steps performed while running the Kuhn/Munkres algorithm on real-life problems is much smaller than that predicted by the worst-case bound of $O(|V||E|\log|V|)$ because the search for most augmenting paths involves only a small fraction of the total number of edges. Therefore, in our implementations, we use the conventional heap or its variations.

5.5.2 Using a combination of a heap and a queue

This is a fairly straightforward idea that has been proposed earlier [13]. While computing the shortest augmenting path, the smallest slackness values and the corresponding edges are stored in a queue and only the edges whose slackness values are greater than the smallest slackness are stored in the heap. Edges from the queue are selected (at unit cost each) until the queue is empty, only when a delete operation is required on the heap. The insertions are performed in the queue and the heap depending on the slackness value of the edge being inserted. Thus, the number of operations on the heap are reduced, especially for graphs containing many edges with the same weight, and hence with many augmenting paths of the same length.

5.5.3 Finding an initial matching

After initializing the matching and label function, we can get a graph $G^* = ((X, Y), E^*)$ where $E^* = \{xy|l(x) + l(y) = w(xy)\} = \{xy|l(x) = w(xy)\}$ because we set $l(y) = 0$ during initialization. We can then run the Kuhn/Munkres algorithm on this graph G^* and get a maximum weight matching M_0 on G^*. After this step, we reinitialize the label function and run the algorithm on the entire graph G to expand the matching to include the remaining vertices.

Let $X_0 = \{x|x \text{ is not free relative to } M_0\}$ and $G_0 = ((X_0, Y), E \cap X_0 \times Y)$. M_0 is a maximum weight matching for G_0. With this M_0 and G_0 as base step, we can use induction (as in Theorem 5.1) to prove the correctness of using the preprocessing step.

Not surprisingly, in our implementation, we found this preprocessing step to be of limited effectiveness in reducing the overall run time. Just like cheap-assignment in the case of maximum-cardinality matching, this is probably because the edges that the initial matching finds are also cheaply discovered by the main algorithm. Therefore, the cost of making two passes over the graph and finding initial matching in the first pass compensates for the small advantage gained by the initial matching.

5.5.4 Minimizing the number of heap insertions

This optimization takes advantage of the fact that in each stage, we seek the shortest path to only one free vertex in Y, not to all the vertices in the graph as in the general single source shortest path problem. During the search for the augmenting path, if π_{min} is the smallest slackness value of any edge xy in the heap such that $y \in Y$ is free, then any other edge $x'y'$ in the heap such that $y' \in Y$ is not free and $\pi(x'y') \geq \pi_{min}$ will never be selected from the heap. The reason is that an augmenting path to free Y node with the smaller π value in the heap will be found first. To utilize this fact, we use a variable π_{min} to keep track of the minimum π value among the edges inserted in the heap with a free Y vertex and do not insert any edge xy in the heap if $\pi(xy)$ is greater than or equal to the current π_{min}.

5.5.5 Limiting the size of the heap

For a given $y \in Y$, only one edge xy can be used because y can be visited only once. Therefore, we can limit the size of the heap to $O(|Y|)$ instead of $O(|E|)$ by keeping at most one edge xy for each $y \in Y$. For each $y \in Y$, we keep a pointer to the location in the heap of the edge xy in $P(y)$. If there is no such edge in the heap, then $P(y) = -1$. Before inserting an edge xy in the heap, we check if another edge $x'y$ is already present in the heap. If $P(y)$ is -1, we insert xy in the heap. If $x'y$ is present in the heap then if $\pi(xy) \geq \pi(x'y)$, we discard xy, else, we replace x' by x and $\pi(x'y)$ by $\pi(xy)$ in the corresponding fields of the record in the heap at the location stored in $P(y)$ and adjust the position of the record in the heap with respect to its new π value. Since the π value can only reduce, this record can only move closer to the root of the heap. Thus the cost of each insertion and deletion can be bounded by $\log|Y|$ rather than $\log|E|$. Another important advantage of this technique is that it saves space used by the heap.

Note that in the context of Dijkstra's single source shortest path algorithm, it is a standard practice to use a heap of size $|V|$ as described above. However, the way the Kuhn/Munkres algorithm is described in some texts (e.g., [5]), it is not clear that the size of the heap can be limited to $|Y|$ when the algorithm prescribes additions to sets S and T (Lines 13 and 29).

5.5.6 Using a heap of heaps

This technique can be applied in lieu of the one described in Section 5.5.5 and attempts to take advantage of the fact that there are many more insertions in the heap than deletions. This technique doubles the cost of each deletion, but asymptotically reduces the expected cost of insertions.

Note that all the edges xy in the heap corresponding to a given $x \in X$ are inserted together. This happens when a free x_i is chosen as the starting point of a potential augmenting path at the beginning of the loop starting at Line 3 and the outgoing edges from x_i may be inserted into the heap, or when a node $y \in Y$ is visited such that $zy \in M$ and the outgoing edges from z may be inserted in the heap. We can exploit this observation and maintain a hierarchical heap structure. We maintain a heap of size at most $|X|$. An entry corresponding to an $x \in X$ in this top level heap is a pointer to a second level heap which only stores the outgoing edges from x. The value field of an entry in the top level heap is equal to the value of the root of the second level heap it points to.

At the time of insertion, a heap is constructed from all qualifying outgoing edges from a given x and this heap is inserted in the heap of heaps. At the time of deletion, an edge is deleted from the second level heap pointed at by the root of the top level heap. The top level heap is then adjusted because the value of the root changes according to the value of the new root of the corresponding second level heap.

It is well known that heapifying n elements, or constructing a heap of size n given all n elements takes $O(n)$ time [11]. Therefore, total cost of all insertions in a given step of the loop starting on Line 3 is $O(|E| + |X| \log |X|)$ and the total cost associated with heap insertions during the entire algorithm is $O(|X||E| + |X|^2 \log |X|)$. The cost of a deletion is $O(\log |X| + \log |Y|) = O(\log |V|)$. In [33], Karp shows that if the edge weights are independent random variables and for each fixed source $x \in X$, $w(xy)$ for all $y \in Y$, are drawn independently from a common distribution, then the expected value of the total number of deletions during the entire course of the algorithm is $O(|X|^2)$. Therefore, under the randomness assumptions mentioned above, the expected cost of the algorithm is $(|X||E| + |X|^2 \log |V|)$ when using the heap-of-heaps technique.

5.5.7 Column scaling

Let w_{max} be the maximum weight of any edge in the bipartite graph. Let w_j be the maximum weight of any entry in the j-th column of the sparse matrix; i.e., $w_j = \max_{i=1}^{|X|} w(x_i y_j)$. Before running the Kuhn/Munkres algorithm, we add $w_{max} - w_j$ to each edge xy_j. Now there are at least $|Y|$ edges in the graph with weight w_{max}. Without affecting the final outcome of the algorithm[1], this tends to increase the number of edges for which the initial slackness is zero. As the search for an augmenting path proceeds, this allows more edges to be put in $N(S)$ and fewer in the heap. Also, more edges are picked out of the queue $N(S)$ in the segment of the pseudocode in Lines 8-14 in Section 5.1 than from the heap. The net results is that the number of insert and delete operations on the heap is reduced. If the improvement suggested in Section 5.5.3 is implemented, then this technique tends to increase the size of the initial matching, thus leaving fewer nodes to be matched while working on the full graph.

As discussed in Section 1, if the algorithm is being used to maximize the absolute diagonal product of the sparse matrix, then the technique suggested in this section is equivalent to scaling the the columns (assuming that the set X represents rows and set Y represents columns) so that they all have at least one entry equal in magnitude to the largest entry in the matrix. This scaling tends to reduce the condition number of the matrix by eliminating the contribution of poor scaling

[1]The final value of the labels is often used to scale the sparse matrix so that each diagonal entry has a value of 1 and all non-diagonal entries are smaller than or equal to 1 [44, 16]. In this case, we subtract $w_{max} - w_j$ from the label of each $y_j \in Y$ after the algorithm to obtain the label values that would have been generated by running the algorithm on the original matrix.

to the ill-conditioning of the matrix. In Section 7, we show that without this scaling, the run time of the Kuhn/Munkres algorithm increases with the condition number of the matrix and that scaling significantly reduces the impact of initial ill-conditioning on the run time of the algorithm.

5.5.8 Using graph-partitioning to limit the scope of augmenting-path searches

In this section, we present the most important result of this paper. We show that if a bipartite graph can be partitioned into two disconnected parts by a vertex-separator of size $O(|V|^\alpha)$, where $0 \leq \alpha \leq 1$ such that the number of vertices in the larger partition does not exceed $|V|/c$, where $c > 1$, and both partitions of the graph can be similarly partitioned recursively, then the assignment problem can be solved in $O(|V|^\alpha |E| \log |V|)$ time. Miller et al. [42, 41] describe a class of graphs called *k-overlap graphs* that satisfy the above property. All graphs arising in finite-difference and finite-element computations are included in this class. Moreover, the partitioning described above can be computed in $O(|V| \log |V|)$ time [41], which is smaller than the worst-case time requirement of the matching algorithm.

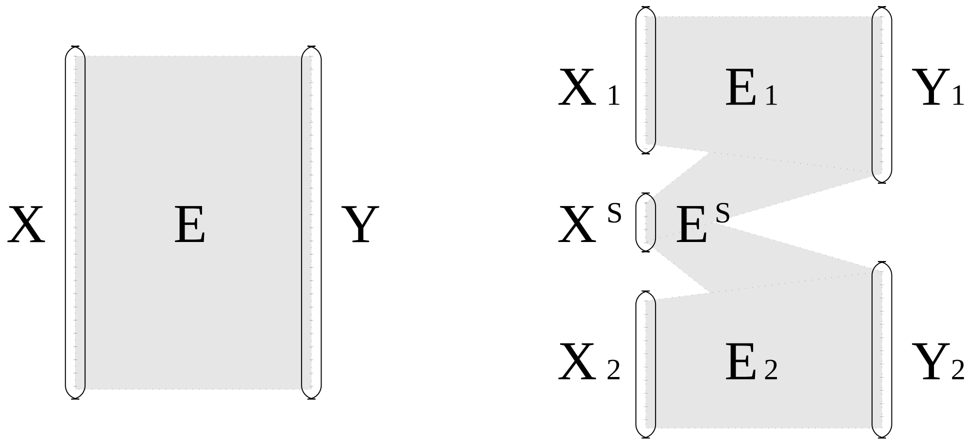
We present a theorem before arriving at the main result.

Theorem 5.2 *If an $n \times n$ sparse matrix A is the adjacency matrix of a d-dimensional overlap graph as defined in [42, 41] and $G_A = ((X_A, Y_A), E_A)$ is the bipartite graph corresponding to A, then G_A has a vertex separator $X^S \subset X_A$ of size $O(n^{(d-1)/d})$ that can be found in $O(n)$ time and partitions G_A into two bipartite graphs $G_{A1}((X_{A1}, Y_{A1}), E_{A1})$ and $G_{A2}((X_{A2}, Y_{A2}), E_{A2})$. Moreover, $\max(|X_{A1}|, |X_{A2}|) \leq n/c$, where $c > 1$.*

Proof: A is the (possibly unsymmetric) sparse adjacency matrix of some (possibly directed) graph $G_D = (V_D, E_D)$, where $\langle ij \rangle \in E_D$ iff $a_{ij} \neq 0$. Let G_U be the undirected counterpart of G_D. Then the structure of the symmetric matrix $B = A + A^T$ is the adjacency structure of $G_U = (V_U, E_U)$, where G_U is a d-dimensional overlap graph. As shown in [41], a set of edges $S \subset E_U$ can be found in $O(n)$ time that partitions V_U into two disjoint subsets V_{U1} and V_{U2} such that there is no edge in $E_U - S$ connecting a vertex $i \in V_{U1}$ and a vertex $j \in V_{U2}$; $|S| = O(n^{(d-1)/d})$; and $\max(|V_{U1}|, |V_{U2}|) < n/c$ for some $c > 1$.

Now consider the bipartite graph $G_B = ((X_B, Y_B), E_B)$ corresponding to matrix B. In G_B, $X_B = \{x_1, x_2, \ldots, x_n\}$; $Y_B = \{y_1, y_2, \ldots, y_n\}$; and $x_i y_j \in E_B$ iff $ij \in E_U$. Let $S_B = \{x_i y_j : ij \in S \text{ or } ji \in S\}$. It is easy to see that S_B is an edge-separator for the bipartite graph G_B. Let X^S be a vertex cover of S_B in X_B; i.e., $X^S = \{x_i : x_i y_j \in S_B \text{ for some } y_j \in Y_B\}$. Then the set of vertices in X^S is a vertex-separator for the graph G_B. $|X^S| \leq |S_B| = 2|S| = O(n^{(d-1)/d})$. Consider the bipartite graph $G_A = ((X_A, Y_A), E_A)$ corresponding to matrix A. Since $X_A = X_B, Y_A = Y_B$, and $E_A \subseteq E_B$, X^S is also a vertex separator for G_A.

Let X^S separate G_A in two subgraphs $G_{A1} = ((X_{A1}, Y_{A1}), E_{A1})$ and $G_{A2} = ((X_{A2}, Y_{A2}), E_{A2})$. Then $|Y_{A1}| = |V_{U1}|$ and $|Y_{A2}| = |V_{U2}|$ by construction of S_B from S (note that X^S is a vertex cover of S^B). Also, $|X_{A1}| \leq |V_{U1}|$ and $|X_{A2}| \leq |V_{U2}|$ because some vertices from each X_i contribute to the vertex cover (strict equality would have held without this contribution). Therefore, $\max(|X_{A1}|, |X_{A2}|) \leq \max(|V_{U1}|, |V_{U2}|) < n/c$, where $c > 1$.



(a) A bipartite graph (b) The graph after partitioning X

Figure 3: A bipartite graph $G = ((X, Y), E)$ before and after partitioning by the set $X^S \subset X$ of separator nodes. $X_1 \cup X_2 \cup X^S = X$, $E_1 \cup E_2 \cup E^S = E$, and $Y_1 \cup Y_2 = Y$.

This completes the proof. □

We now use the result of Theorem 5.2 to derive a worst-case complexity bound on the run time of the Kuhn/Munkres algorithm on graphs that meet the requirements of the theorem. Here we assume that the input matrix always has a perfect matching. Figure 3 shows the partition of the set X on a bipartite graph $G = ((X, Y), E)$. X^S is the subset of X whose removal partitions $G = ((X, Y), E)$ into two disjoint parts $G_1 = ((X_1, Y_1), E_1)$ and $G_2 = ((X_2, Y_2), E_2)$. We can restrict the scope of augmenting path searches originating from any vertex in X_1 or X_2 as follows. We would run the Kuhn/Munkres algorithm on G in such a way that we first choose the free vertices from X_1 as origins of augmenting path searches, followed by the vertices from X_2, and finally from X^S. As long as none of the vertices in X^S are matched, any augmenting path originating in X_1 (X_2) can contain edges from only E_1 (E_2). Only after all vertices in X_1 and X_2 are matched, finding an augmenting path originating from a vertex in X^S may require exploring all E edges. Both the partitioning and the order in which the vertices of X are considered in the algorithm can be recursively defined on each partition of the graph.

Let $(d-1)/d = \alpha$ and let $T(G)$ be the running time of the Kuhn/Munkres algorithm on a bipartite graph G whose matrix is the adjacency matrix of an overlap graph as described earlier. Then $|X^S| = \beta n^\alpha$.

$$
\begin{aligned}
T(G) &= |X^S||E|\log|X| + T(G_1) + T(G_2) \\
&= \beta n^\alpha |E| \log n + \beta |X_1|^\alpha |E_1| \log|X_1| + \beta |X_2|^\alpha |E_2| \log|X_2| \\
&\quad + T(G_{11}) + T(G_{12}) + T(G_{21}) + T(G_{22}),
\end{aligned}
$$

where G_{i1} and G_{i2} are the partitions of G_i ($i = 1, 2$). Since the separator- and partition-size properties hold recursively, we continue with the derivation of $T(G)$ as follows:

$$
\begin{aligned}
T(G) &\leq \beta(n^\alpha|E|\log n + (n/c)^\alpha|E_1|\log(n/c) + (n/c)^\alpha|E_2|\log(n/c)) \\
&\quad + T(G_{11}) + T(G_{12}) + T(G_{21}) + T(G_{22}) \\
&\leq \beta(n^\alpha|E|\log n + (n/c)^\alpha(|E_1| + |E_2|)\log n) + T(G_{11}) + T(G_{12}) + T(G_{21}) + T(G_{22}) \\
&\leq \beta|E|(n^\alpha + (n/c)^\alpha)\log n + T(G_{11}) + T(G_{12}) + T(G_{21}) + T(G_{22}) \\
&\leq \beta n^\alpha|E|(1 + c^{-\alpha} + c^{-2\alpha} + \cdots)\log n \\
&= O(n^\alpha|E|\log n).
\end{aligned}
$$

Therefore, for sparse matrices arising in two- and three-dimensional finite-element problems, the worst-case complexity for permuting the rows to maximize the product of the absolute values of the diagonal entries is $O(\sqrt{|V|}|E|\log|V|)$ and $O(|V|^{2/3}|E|\log|V|)$, respectively. The above analysis assumes the use of a standard heap data-structure while searching for augmenting paths. By using more advanced priority queues discussed in Section 5.5.1, a worst-case time bound of $O(|V|^\alpha(|E|+|V|\log|V|))$ can be obtained for overlap graphs. This is better than the best previously known strongly polynomial worst-case bound of $O(|V|(|E|+|V|\log|V|))$.

A prerequisite for achieving these lower bounds is to compute a nested-partioning based order in which vertices from the set X are chosen as starting points of augmenting path searches. This order can be computed in $O(|V|\log|V|)$ time. Although this is asymptotically smaller than the cost of matching, in practice, the cost of computing the ordering may outweigh the advantage due to a large constant factor. However, in the context of solving sparse systems of linear equations, this ordering needs to be computed only once for all matrices with the same structure but different values. Typical applications do require factoring sparse matrices of the same structure but different values multiple times.

Another advantage of using a partitioning based ordering of the vertices of X for the Kuhn/Munkres algorithm is that it can lead to efficient parallel formulations of the algorithm for the class of matrices for which sublinear size separators exist. The reason is that the augmenting path search in the two partitions can proceed independently and this parallelism can be applied recursively.

6 Karp's Algorithm

In [33], Karp proposed a new algorithm to solve the maximum weight bipartite matching problem. Let G be a bipartite graph $((X,Y),E)$ with weight function w on edges in E. Without loss of generality, we assume that $|X| \leq |Y|$. We call X the source set and Y the destination set. Karp's algorithm runs in expected time $O(|X||Y|\log|Y|)$ under the assumption that the edge weights are independent random variables and the weights of the edges incident with any given vertices in $|X|$ are identically distributed. This algorithm finds a matching of minimum weights. If we replace each weight $w(xy)$ by $w_{max} - w(xy)$, where w_{max} is the maximum of all edge weights, then it is equivalent to the original problem of finding the maximum weight matching.

6.1 Background

The *weight* of an augmenting path P, denoted as $w(P)$, is $\sum_{e\in P\setminus M} w(e) - \sum_{e\in P\cap M} w(e)$. The following theorem is well-known [20]:

Theorem 6.1 *If M is of minimum weight among matchings of cardinality k and P is of minimum weight among augmenting paths relative to M, then $M \oplus P$ is of minimum weight among matchings of cardinality $k+1$.*

From this theorem, we can easily define the following algorithm for solving the assignment problem:

$M \leftarrow \emptyset$;
while $|M| < |X|$
    Let P be the minimum-weight augmenting path relative to M;
    $M \leftarrow M \oplus P$;
endwhile

The main processing step in this algorithm is to find a minimum-weight augmenting path relative to the current matching M in each step of the outer loop. The augmenting paths relative to a matching M can be determined using a directed graph $\tilde{G} = (V, \tilde{E})$ where directed edge set $\tilde{E}$ consists of the following directed edges (here we represent directed edges from x to y as $\langle xy \rangle$):

1. $\langle xy \rangle$ is an edge with weight $w(x, y)$ if $x \in X$, $y \in Y$ and edge $xy \notin M$,

2. $\langle yx \rangle$ is an edge with weight $-w(x, y)$ if $x \in X$, $y \in Y$ and edge $xy \in M$.

Let $\tilde{P}$ be a minimum-weight directed path in $\tilde{G}$ from a free vertex in X to a free vertex in Y, then P — the minimum-weight augmenting path relative to M — is the undirected path associated with $\tilde{P}$.

It is well-known that a minimum-weight path in a directed graph from one given vertex set to another given vertex set can be computed quickly provided all edges are non-negative. In [19], Edmonds and Karp propose a technique to modify the weights of the edges to be non-negative while keeping the identity of the identity of the minimum-weight directed path (or paths) from a free vertex in X to a free vertex in Y unchanged. This is achieved by associating with each vertex v a "potential" $\alpha(v)$ which affects the weights of the edges of $\tilde{G}$ incident with v. We use $\tilde{w}$ to denote the non-negative weight in the following algorithm. This weight function changes according to the potential function α.

The revised algorithm [33] that incorporates the potential function are given below. Figure 4 illustrates this algorithm on an example bipartite graph.

$M \leftarrow \emptyset$;
for $v \in V$
    $\alpha(v) \leftarrow 0$;

03 *endfor*
04 *while* $|M| < |X|$
05 Generate the associated graph $\tilde{G}$;
06 *for* each $xy \in E \setminus M$ assign directed edge $\langle xy \rangle$ weight
07 $\tilde{w}(x, y) \leftarrow w(x, y) + \alpha(x) - \alpha(y)$;
08 *endfor*
09 *for* each $xy \in M$ assign directed edge $\langle yx \rangle$ weight
10 $\tilde{w}(x, y) \leftarrow 0$;
11 *endfor*
12 *for* all $v \in V$
13 Let $\gamma(v)$ be the minimum weight of a directed path in $\tilde{G}$ which
14 is from a free vertex in X to either v or a free vertex in Y
15 $\alpha(v) \leftarrow \alpha(v) + \gamma(v)$;
16 *endfor*
17 Let $\tilde{P}$ be a minimum-weight directed path in $\tilde{G}$ from a free vertex
18 in X to a free vertex in Y;
19 Let P be the augmenting path in G associated with $\tilde{P}$;
20 $M \leftarrow M \oplus P$;
21 *endwhile*

Lines 17–18 in the above algorithm are equivalent to finding a single source shortest path in the graph $\tilde{G}$ from a hypothetical vertex connected to all free vertices in X to any free vertex in Y. Dijkstra's single source shortest path algorithms can be easily modified for this purpose (see [33] for details). The correctness of the algorithm depends on the following properties holding at the beginning of each iteration of the *while* loop. These properties can be proven inductively.

1. for each free vertex $x \in X$, $\alpha(x) = 0$;

2. for each free vertex $y \in Y$, $\alpha(y) = \alpha^* = max_{v \in V}\{\alpha(v)\}$;

3. for each edge $xy \in E$, $\tilde{w}(x, y) = w(x, y) + \alpha(x) - \alpha(y) \geq 0$;

4. for each edge $xy \in M$, $\tilde{w}(x, y) = w(x, y) + \alpha(x) - \alpha(y) = 0$;

5. if $\tilde{P}$ is a directed path in $\tilde{G}$ from a free vertex in X to a free vertex in Y and P is the associated path in G, then $w(P) - \tilde{w}(\tilde{P}) = \alpha^* = max_{v \in V}\{\alpha(v)\}$.

6.2 Modification

While finding the shortest augmenting path using Dijkstra's algorithm, a priority queue or a heap of edges must be maintained [11]. Typically, the number of insertions in the heap is much higher than the number of deletions. Karp [33] describes a method that can significantly reduce the number of insertions with some increase in the number of deletions. It is based on the use of "surrogate"

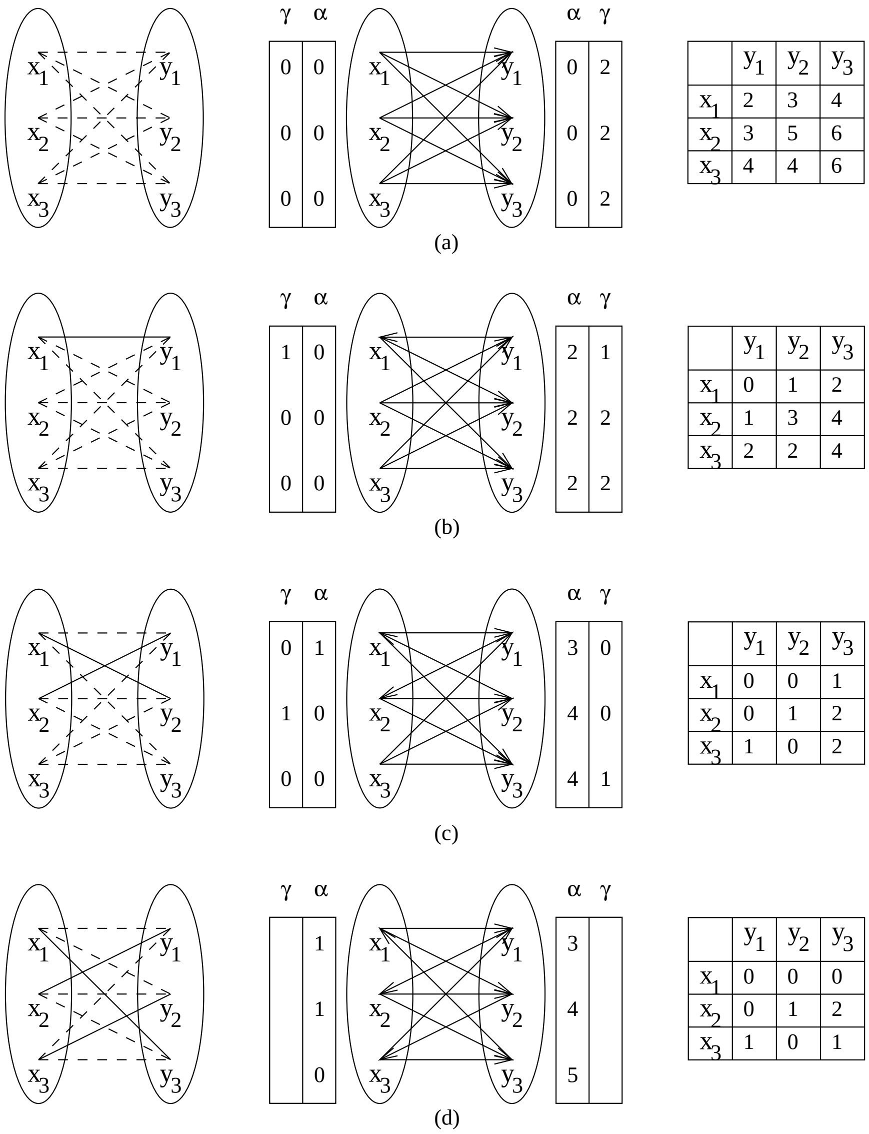


Figure 4: In each figure from (a) to (c), on the left is G with matching M denoted by solid line; in the middle is the directed graph $\tilde{G}$, α and γ; on the right is edge weight $\tilde{w}$. α is the value at the beginning of the correspondent iteration *while* loop; γ is the value calculated during that iteration and the table of $\tilde{w}$ is the value at the beginning of iteration. (d) is the α and $\tilde{w}$ after the algorithm. In the first iteration, we find the minimum-weight directed path $P = \{x_1y_1\}$. In the second iteration, the path found is $P = \{x_2y_1x_1y_2\}$. In the third iteration, the minimum-weight path is $P = \{x_3y_2x_1y_3\}$. The final matching is $\{x_1y_3, x_2y_1, x_3y_2\}$.

items, where the insertion of one surrogate item will take the place of a large number of insertions of ordinary items.

Consider a point in the execution of assignment algorithm when a matching M has just been determined and the node potentials $\alpha(v)$ have been readjusted. Let

$$\alpha^* = max_v\{\alpha(v)\}.$$

We have already defined that

$$\tilde{w}(x,y) = w(x,y) + \alpha(x) - \alpha(y).$$

Define $w^*(x,y)$ by

$$w^*(x,y) = w(x,y) + \alpha(x) - \alpha^*.$$

Then

$$\begin{aligned} w^*(x,y) &\leq \tilde{w}(x,y), \\ \gamma(x) + w^*(x,y) &\leq \gamma(x) + \tilde{w}(x,y), \end{aligned}$$

with equality if y is a free vertex in Y. Also $w^*(x,y) \leq w^*(x,y')$ if and only if $w(x,y) \leq w(x,y')$.

The modified algorithm uses a preprocessing phase in which, for each source x, a list $LIST(x)$ is formed consisting of all ordered pairs $\langle xy \rangle$ such that y is in Y, sorted in increasing order of the weight $w(x,y)$.

In the execution of the assignment algorithm, the priority queue Q contains two types of items: *regular items* of the form $\langle\langle xz \rangle, \gamma(x) + \tilde{w}(x,z)\rangle$ and *special items* of the form $\langle\langle xy \rangle, \gamma(x) + w^*(x,y)\rangle$. Such a special item is called a *surrogate* for the regular item $\langle\langle xz \rangle, \gamma(x) + \tilde{w}(x,z)\rangle$ if $w(x,y) \leq w(x,z)$ (or equivalently, $\gamma(x) + w^*(x,y) \leq \gamma(x) + w^*(x,z) \leq \gamma(x) + \tilde{w}(x,z)$).

The following operations on Q are required during the execution of the algorithm for finding the shortest augmenting path:

1. $Q \leftarrow \emptyset$;
2. Test if $Q = \emptyset$;
3. $Q \leftarrow Q \cup OUT(x)$;
4. Choose $\langle xy \rangle$ such that $\gamma(x) + \tilde{w}(x,y) = min_{\langle uv \rangle \in Q}\{\gamma(u) + \tilde{w}(u,v)\}$.

The respective implementations [33] of these four operations on Q using the surrogate items are as follows:

1. $Q \leftarrow \emptyset$;
2. Test if $Q = \emptyset$;
3. $\langle xy \rangle \leftarrow$ first element of $LIST(x)$;
   insert into Q the special entry $\langle\langle xy \rangle, \gamma(x) + w^*(x,y)\rangle$;

4. *Procedure Select*

do

  $q \leftarrow$ the item of least value in Q;
  delete q from Q;
  $\langle\langle xy\rangle \leftarrow$ the edge to which q corresponds;
  if q is special *then*
    if $\langle xy\rangle$ is not the last element of $LIST(x)$ *then*
      $\langle xw\rangle \leftarrow$ the successor of $\langle xy\rangle$ in $LIST(x)$;
      insert into Q the special item $\langle\langle xw\rangle, \gamma(x) + w^*(x,w)\rangle$;
    endif
    insert into Q the regular item $\langle\langle xy\rangle, \gamma(x) + \tilde{w}(x,y)\rangle$;
  endif

until regular item $\langle\langle xy\rangle, \gamma(x) + \tilde{w}(x,y)\rangle$ has been selected s.t. $(x,y) \notin M$

The following is the final version of Karp's algorithm [33] for the assignment problem after incorporating all the modifications:

for $x \in X$
  $LIST(x) \leftarrow$ an array containing the set of elements $\{\langle xy\rangle | y \in Y\}$
    in increasing order of $w(x,y)$;
endfor
$M \leftarrow \emptyset$;
for $v \in V$
  $\alpha(v) \leftarrow 0$;
endfor
while $|M| < |X|$
  $PATHSET \leftarrow \emptyset$;
  $Q \leftarrow \emptyset$;
  $R \leftarrow$ {free vertices in X};
  for $x \in R$
    $\gamma(x) \leftarrow 0$;
    $\langle xy\rangle \leftarrow$ first element of $LIST(X)$;
    Insert into Q the special item $\langle\langle xy\rangle, \gamma(x) + w^*(x,y)\rangle$;
  endfor
  while $R \cap$ {free vertices in Y} $= \emptyset$
    Execute *Procedure Select* (suppose the selected edge is $\langle xy\rangle$);
    if $y \notin R$ *then*
      $PATHSET \leftarrow PATHSET \cup \{\langle xy\rangle\}$;
      $R \leftarrow R \cup \{y\}$;
      $\gamma(y) \leftarrow \gamma(x) + \tilde{w}(x,y)$;
      if y is not free *then*
        $(y,v) \leftarrow$ the edge of M incident with y;

$PATHSET \leftarrow PATHSET \cup \{\langle yv \rangle\}$;
$R \leftarrow R \cup \{v\}$;
$\gamma(v) \leftarrow \gamma(y)$;
$\langle vl \rangle \leftarrow$ first element of $LIST(v)$;
insert into Q the special item $\langle \langle vl \rangle, \gamma(v) + w^*(v,l) \rangle$;
endif
endif
endwhile
for $v \notin R$
$\gamma(v) \leftarrow \gamma(y)$;
endfor
for $v \in V$
$\alpha(v) \leftarrow \alpha(v) + \gamma(v)$;
endfor
Let $\tilde{P}$ be the unique directed path from a free vertex in X to y
whose edges are all in $PATHSET$;
Let P be the set of edges in G corresponding to directed edges in $\tilde{P}$;
$M \leftarrow M \oplus P$;
endwhile

6.3 Analysis

First, the preprocessing sorting operation which is involved in forming the list $LIST(x)$ for each $x \in X$ requires $O(|X||Y|\log|Y|)$ steps.

Under the randomness assumptions state earlier in this section, Karp [33] shows that the expected value of the number of edges deleted from the heap during the entire course of the algorithm is $O(|X|^2)$. Each edge-deletion may require the selection of one special and one regular item from Q corresponding to that edge. So each deletion costs $O(\log|E|)$ time. Thus, the expected cost of the entire algorithm is $O(|X|^2 \log|E| + |X||Y|\log|Y|) = O(|X||Y|\log|Y|)$.

6.4 Practicality Issues

Although the expected asymptotic complexity of Karp's algorithm is smaller than that of the basic Kuhn/Munkres algorithm, it is much slower in practice. Karp's algorithm seeks the minimum among all matchings of cardinality k in the k-th stage. The augmenting path that Karp's algorithm seeks in the k-th stage can start from any of the free vertices of set X. On the other hand, the Kuhn/Munkres algorithm chooses a vertex from the set X as the starting point of the augmenting path. At the end of stage k, it is just assured of finding the maximum/minimum matching containing the starting vertex selected in stage k and the previously matched vertices, not an overall maximum/minimum matching of size k. Although, at the end of $|X|$ stages, both algorithms find the minimum/maximum matching, the goal that Karp's algorithm seeks at each stage is much more difficult and costly. Each stage starts with putting the all outgoing edges from all free vertices in X

Domain	Matrices
Linear Programming	allgrade, comp1, kk6
Fluid dynamics	af23560, e40r0000, e40r1000, e40r5000, rma10, shyy161
Material science	cry10000
Thermodynamics	epb3
Chemical engineering	lhr34, lhr34c, lhr71, lhr71c
Probability application	rw5151
Nuclear physics	utm5940
Structural engineering	s3dkt3m2, s3dkq4m2
Electrical engineering	pre2, twotone
Unstructured 2-D Euler solver	venkat01, venkat50
Other FEM problems	av41092, fidapm11

Table 1: Test matrices and their application domains.

into the heap. In Kuhn/Munkres algorithm, each stage starts by putting the outgoing edges from just one selected free vertex in X in the heap.

We did implement Karp's algorithm, and as expected, it was much slower than the basic Kuhn/Munkres algorithm, that did not even incorporate any of the improvements of Section 5.5. For instance, for the matrix *utm5940*, Karp's algorithm took about 7 seconds, which is slower by about a factor of 70 than Kuhn/Munkres. This factor was observed to be even higher for larger matrices.

7 Experimental Results

In this section, we present the results of our implementation of the Kuhn/Munkres algorithm and the impact of the various improvements suggested in Section 5.5. We compare the timings with a commercial code MC64 from the Harwell Subroutine Library [31]. Both MC64 and our code use similar algorithms that find the maximum weight matching by searching for shortest augmenting paths. We also present timing results of a code based on an algorithm that uses an alternate strategy to find the maximum weight matching. We have chosen a test suite containing several large sparse matrices from a variety of applications[2] to conduct our experiments on. The application domains of these matrices are listed in Table 1.

All experiments were conducted on a 200 MHz Power3 RS6000 workstation with 64Kb level 1 cache and 4Mb level 2 cache. All our codes as well as the CSA code (Section 7.3) are written in C, while MC64 is written in Fortran. The highest level of optimization (-O4 option) was used to compile all programs. In most case, the AIX Fortran compiler XLF tends to generate much faster code than the C compiler XLC. The comparison with MC64 must be interpreted in the light of this

[2]The linear programming examples are derived by choosing random bases from the constraint matrices.

Matrix	n	nnz	HK time (s)	KM-1 time (s)	KM-2 time (s)	KM-3 time (s)	KM-4 time (s)	*MC64* time (s)
af23560	23560	484256	.111	0.49	0.46	0.52	0.45	0.45
allgrade	21699	470822	.053	15.7	14.9	14.9	5.34	24.6
av41092	41092	1683902	3.99	57.0	51.7	56.1	41.2	41.7
comp1	16783	578665	.179	12.1	13.7	5.13	3.40	6.39
cry10000	10000	49699	.009	0.05	0.05	0.05	0.05	0.05
e40r0000	17281	553956	.122	1.57	1.44	0.43	0.66	0.40
e40r1000	17281	553956	.123	3.21	2.97	1.47	1.55	0.94
e40r5000	17281	553956	.123	4.53	4.26	2.93	2.77	3.21
epb3	84617	463625	.108	0.42	0.42	0.44	0.44	0.44
fidapm11	22294	623554	.082	10.2	10.2	3.14	3.07	4.14
kk6	62065	259871	.068	3.80	3.63	6.39	2.92	4.99
lhr34	35152	764014	.370	3.89	3.72	1.54	1.87	1.21
lhr34c	35152	764014	.372	4.20	3.98	3.46	3.06	2.98
lhr71	70304	1528092	.721	8.65	8.74	3.50	4.17	2.73
lhr71c	70304	1528092	.720	8.77	8.60	7.50	6.69	6.64
pre2	659033	5959282	4.29	10.4	10.3	23.0	16.5	8.03
rma10	46835	374001	.256	2.35	2.48	1.92	1.84	2.70
rw5151	5151	20199	.005	0.49	0.57	0.56	0.34	0.37
s3dkq4m2	90449	2455670	.375	2.38	2.42	1.98	1.98	1.97
s3dkt3m2	90449	1921955	.360	2.15	2.17	1.67	1.67	1.71
shyy161	76480	329762	.085	3.65	4.25	2.51	2.77	1.15
twotone	120750	1224224	.436	2.82	2.88	1.53	1.23	1.75
utm5940	5940	83842	.012	0.10	.094	0.10	.097	0.10
venkat01	62424	1717792	.449	1.25	1.36	1.27	1.26	1.17
venkat50	62424	1717792	.449	1.30	1.42	1.27	1.25	1.16

Table 2: The running time of the Hopcroft-Karp algorithm, various versions of the Kuhn/Munkres algorithm, and MC64. HK refers to the Hopcroft-Karp algorithm for maximum cardinality matching. KM-1 refers to the basic Kuhn/Munkres algorithm while minimizing the number of heap insertions and limiting the heap size as suggested in Sections 5.5.4 and 5.5.5, respectively. KM-2 is similar to KM-1 except that it uses a heap of heaps (Section 5.5.6) instead of limiting the heap size. KM-3 uses the scaling technique described in Section 5.5.7 on top of KM-1. KM-4 is the same same algorithm as KM-3 except that the vertices in X are chosen in an order determined by a recursive partitioning of X as suggested in Section 5.5.8.

fact. In all our experiments, we have sought to maximize the product of the absolute values of the diagonal entries; hence, the original maximum weight algorithms are executed on bipartite graphs whose edge-weights are derived by removing all explicit zeros from the original matrices and then taking the logarithm of the entries. The time for this preprocessing is included in the running times reported for all algorithms.

As mentioned in Section 5.5.3, finding an initial matching by a greedy algorithm did not result in a noticeable improvement in run times on an average. However, maintaining a heap and queue combination (Sections 5.5.2 and reducing the number of heap insertions by keeping track of the current shortest path to a free vertex in Y (Section 5.5.4) speeded up the basic Kuhn/Munkres algorithm significantly for all matrices. We use that implementation, as shown in the column titled KM-1 in Table 2, as the basis for comparing the effect of the other suggested improvements. Recall from Section 5.5 that limiting the size of the heap to $O(|X|)$ and using the heap-of-heaps technique are mutually exclusive. In the column titled KM-2 in Table 2, we show the timing results of using the heap-of-heaps technique in lieu of reducing the number of heap insertions. The relative performance of the algorithms in columns KM-1 and KM-2 seems to be somewhat matrix dependent and there doesn't appear to be a clear winner in terms of run time. We chose to use the algorithm in KM-1 to implement other optimizations because it uses less memory (Section 5.5.5).

Column KM-3 shows the effect of column-scaling as suggested in Section 5.5.7 on the KM-1 algorithm. Column KM-4 shows the result of prepermuting the matrix rows based on a recursive partitioning of X as suggested in Section 5.5.8. The partitioning time is not included because in typical applications, a matrix with the same structure but different values must be factored multiple number of times. The nested-partioning based row ordering needs to be computed only once for a given nonzero sparse pattern and therefore its cost is amortized over several instances of the maximum-weight matching computation. A number of software packages are available to compute good partionings very efficiently [29, 35, 28]. We used an existing sparse matrix ordering software [28] to generate a recursive row-partitioning. Columns KM-3 and KM-4 clearly show the benefits of column-scaling and row-partitioning on the Kuhn/Munkres algorithm. We present more experimental evidence in support of these two techniques in Sections 7.1 and 7.2, respectively.

Finally, the last column in Table 2 shows the run time of MC64, which is a commercial Fortran code for finding the maximum/minimum weight matchings in bipartite graphs, and includes many of the improvements suggested in Section 5.5. Algorithmically, the MC64 code is equivalent to that corresponding to column KM-3. MC64 converts the maximum weight matching problem to a minimum weight matching problem by replacing each weight $w(xy)$ by $w_{max}(x) - w(xy)$, where $w_{max}(x) = \max_{y_i \in Y} w(xy_i)$. This is equivalent to a scaling of rows, similar to the scaling of columns that we have implemented.

7.1 The Effect of Column-Scaling

Intrigued by the effect of scaling on the run time of the Kuhn/Munkres algorithm, we tried to empirically establish some connection between the condition number of a matrix and the run time of the Kuhn/Munkres algorithm. In Table 3, we present the run times of KM-3, KM-4, and MC64

Matrix	KM-1 (s)	KM-3(s)	MC64 (s)
rnd25kR	12.0	9.17	7.82
rnd25k06	46.5	11.1	16.5
rnd25k12	68.8	10.4	22.3
rnd50kR	45.1	37.0	26.3
rnd50k06	178.	36.1	56.1
rnd50k12	268.	38.2	95.5

Table 3: The effect of scaling on the run time of the Kuhn/Munkres algorithm. Algorithm KM-3 is the same as KM-1, except that the columns of the matrix are scaled in KM-3.

on different matrices of the same size and nonzero pattern. The matrix rndxxkR is an $xx000 \times xx000$ random sparse matrix generated by MATLAB. The matrix rndxxkyy is an $xx000 \times xx000$ random sparse matrix with condition number equal to 10^{yy}, again generated by MATLAB. The number of nonzeros (edges) in all matrices (graphs) is approximately 625000. The table shows that while the run times of KM-1 and MC64 increase with the condition number of the matrix, KM-3 is reasonably immune to changes in the condition number for the Matlab generated matrices. However, we would like to draw the reader's attention to the run times in columns KM-1 and KM-3 in Table 2 for matrices *e40r0000, e40r1000,* and *e40r5000.* These three matrices have identical nonzero patterns and are picked up from different time steps in a CFD simulation with a progressively worsening condition number. Although KM-3 is much faster than KM-1 in all three cases, both KM-1 and KM-3 run times increases with an increasing condition number. Thus, it appears that the scaling technique can improve the run time of the Kuhn/Munkres algorithm by neutralizing the ill-conditioning due to poor scaling, but the run time still depends on the intrinsic ill-conditioning present in the matrix.

7.2 The Effect of Row-Partitioning

Column KM-4 in Table 2 shows that partitioning the rows as discussed in Section 5.5.8 does improve the run time of the algorithm corresponding to column KM-3. However, the speedup due to partitioning is not as dramatic as the reduction in the worst-case complexity. This is primarily because of two reasons. First, the worst-case bound of $O(|V||E|\log|V|)$ is quite loose. This bound is based on assigning a cost of $O(|E|\log|V|)$ to each augmenting-path search. Even without row-partitioning, the search for an augmenting path usually involves only a small fraction of the total number of edges in the graph, especially with some of the optimizations suggested in Section 5.5. Secondly, all graphs in Table 2 come from real problems and the rows and columns in most of them are numbered in some natural way corresponding to the physical domain that the problem is modeling. The intrinsic locality of this natural ordering encompasses some of the features of a row-partitioning based ordering suggested in Section 5.5.8 and prevents a large number of augmenting path searches from spanning large portions of the graph. In Table 4 we reproduce the run times of algorithms KM-3, KM-4, and MC64, except that this time we permuted the input matrix randomly

Matrix	KM-3 time (s)	KM-4 time (s)	*MC64* time (s)
af23560	0.50	0.45	0.56
allgrade	16.7	4.76	17.9
av41092	100.	44.0	118.
comp1	6.76	3.89	6.44
cry10000	0.05	0.05	0.06
e40r0000	0.47	0.47	0.40
e40r1000	1.79	1.46	1.20
e40r5000	3.24	2.65	3.57
epb3	0.50	0.44	0.49
fidapm11	5.41	3.54	7.51
kk6	6.31	2.57	3.23
lhr34	3.06	1.72	3.94
lhr34c	4.87	2.91	4.69
lhr71	8.29	3.66	12.7
lhr71c	11.3	6.56	13.0
pre2	47.7	17.4	14.4
rma10	2.02	1.85	3.35
rw5151	0.57	0.43	0.52
s3dkq4m2	2.10	1.87	1.96
s3dkt3m2	1.75	1.55	1.81
shyy161	8.82	3.30	1.33
twotone	1.82	1.23	2.44
utm5940	.099	.093	0.10
venkat01	1.36	1.24	1.08
venkat50	1.35	1.24	1.10

Table 4: Effect of row-partitioning on the run time of the Kuhn/Munkres algorithm. The three columns in this table represent the same algorithms as the last three columns of Table 2, except that a randomly permuted version of the original matrix is supplied to the algorithms.

before starting. A comparison of the corresponding columns of Tables 2 and 4 reveals that the KM-4 run time is largely unaffected by the initial ordering; however, the KM-3 and MC64 run times are quite sensitive to the initial numbering of rows and columns in the sparse matrix. In Table 4, the KM-4 run times are significantly better than those of KM-3 and MC64. Thus, Table 4 captures the real impact of the recursive row-partitioning on the time complexity of the Kuhn/Munkres algorithm on sparse bipartite graphs.

7.3 Comparison With a Cost Scaling Push-Relabel Code

As mentioned in Section 1, there are two classes of algorithms for solving the maximum matching problems—augmenting path algorithms and flow algorithms. In this section, we present timing results of the best known implementation, namely CSA, [25] of the asymptotically fastest flow algorithm [22] and compare them with the run times of the augmenting path algorithms given in Table 2. The asymptotic complexity of the CSA algorithm [22, 25] is $O(|E|\sqrt{|V|}\log(|V|K))$, where the edge-weights are integers in the range $[-K, \ldots, K]$. Since the CSA algorithm admits only integer edge-weights, we had to make an adaptation to use it for our application, in which, the edge-weights are real, in general. Additionally, we are interested in maximizing the product of the absolute values of the diagonal entries of a matrix. Therefore, if $\omega(ij)$ is a nonzero entry in the i-th row and the j-th column of the original sparse matrix, then we assign a weight $w(ij) = PREC \times (\log(|(\omega(ij)|) - \log(\omega_{min}) + 1) - PREC + 1$ to the corresponding edge of the bipartite graph. In the preceding expression for $w(ij)$, ω_{min} is the nonzero element of the matrix with the smallest absolute value and $PREC \geq 1$ is a measure of the precision that is preserved while converting the real values into integers to input to the CSA code. After taking the logarithms, first the values are normalized so that the smallest edge-weight is 1. Now, if $PREC = 1$, then effectively all real edge-weight values are truncated to their integer parts. If $PREC = 10^t$, then t decimal digits from the fractional part are taken into account by the CSA code. The smallest edge-weight is maintained at 1, but the the largest integer edge-weight input to the CSA code increases with t (and hence, with $PREC$).

In [25], Goldberg and Kennedy describe a few different variations of their CSA code. According to the authors, CSA_Q_QM is the most promising version of their code, although they report some classes of problems on which CSA_S_QM performs better. In the first two columns of Table 5, we present the timing results of CSA_S_QM and CSA_Q_QM on our suite of test matrices for $PREC = 1$. Although CSA_S_QM is the faster of the two on more matrices than CSA_Q_QM, the latter appears to be more robust. For instance, on matrices *s3dkq4m2* and *s3dkt3m2*, CSA_S_QM is extremely slow. For some higher values of $PREC$, CSA_S_QM gave abnormally slow timings for *kk6* and the four *lhr* matrices too. Therefore, we consider CSA_Q_QM as the algorithm of choice and give detailed results for higher values of $PREC$ (the 3rd, 4th and 5th columns in Table 5) only for CSA_Q_QM. Upon comparing the results of Table 5 with those of Tables 2 and 3, we find that irrespective of the value of $PREC$, CSA is significantly slower than the Kuhn/Munkres algorithm (i.e., algorithms KM-3, MC64), even without a graph-partioning based ordering of matrix rows (KM-4). Exceptions to this are some of the MATLAB generated random matrices on which CSA

Matrix	CSA_S_QM PREC = 1	CSA_Q_QM PREC = 1	CSA_Q_QM PREC = 10	CSA_Q_QM PREC = 10^2	CSA_Q_QM PREC = 10^3
af23560	3.12	3.62	1.57	1.66	1.79
allgrade	9.81	11.5	12.6	13.6	14.8
av41092	80.6	87.0	101.	104.	85.7
comp1	9.50	9.46	9.76	15.0	14.7
cry10000	0.16	0.41	0.13	0.13	0.14
e40r0000	3.31	2.30	2.81	3.04	3.60
e40r1000	3.68	3.63	3.40	3.76	3.67
e40r5000	16.4	20.1	5.95	4.11	3.64
epb3	1.56	1.62	1.28	1.38	1.51
fidapm11	20.8	25.3	8.55	7.35	9.69
kk6	8.84	9.45	12.0	11.4	6.88
lhr34	6.09	6.06	6.43	6.38	6.34
lhr34c	15.4	16.1	15.9	15.6	15.8
lhr71	16.2	18.3	18.0	18.0	18.7
lhr71c	31.5	34.8	37.3	33.2	32.7
pre2	81.8	132.	115.	145.	133.
rma10	46.7	45.7	7.39	8.49	8.79
rw5151	0.51	0.59	0.56	0.58	0.56
s3dkq4m2	67.8	4.27	4.82	5.26	5.44
s3dkt3m2	55.0	4.61	5.10	5.48	5.73
shyy161	1.17	0.78	4.73	5.84	6.71
twotone	10.3	16.5	12.5	13.2	13.3
utm5940	0.44	0.46	0.32	0.34	0.34
venkat01	5.04	5.17	4.94	5.38	6.58
venkat50	5.73	6.44	6.31	6.78	7.13
rnd25kR	15.7	15.6	3.69	3.10	3.16
rnd25k06	9.19	9.39	7.45	6.55	8.37
rnd25k12	20.5	21.3	11.7	11.8	12.0
rnd50kR	38.4	42.0	7.95	7.48	8.07
rnd50k06	27.0	30.9	20.7	32.3	24.4
rnd50k12	31.1	35.2	39.6	41.6	35.8

Table 5: Run times of the two most promising versions of the cost scaling push-relabel algorithm for finding the maximum weight matchings with different precisions.

seems to outperform the Kuhn/Munkres algorithm if the condition number of the matrix is not too high. Just like KM-1 and MC64, the running time of CSA increases with the condition number of the random matrix, whereas that of KM-3 is almost stable. In conclusion, our tests indicate that for sparse matrices arising in a variety of real problems, a good implementation of the Kuhn/Munkres algorithm is much faster than that of the CSA algorithm.

8 Conclusions and Future Work

In this paper, we have surveyed algorithms for finding maximum cardinality and maximum weight matching on bipartite graphs. We have suggest several improvements to the basic Kuhn/Munkres algorithm for finding a bipartite matching with maximum weight and present experimental results to show the effect of these improvements on the running time of the algorithm. We present a technique that reduces the worst-case complexity of the maximum weight bipartite matching algorithm to $O(|V|^{\alpha}(|E| + |V|\log|V|))$ $(\alpha < 1)$ for a large class of graphs from the currently best known strongly polynomial bound of $O(|V|(|E|||V|\log|V|))$. Our experimental results show that two of the techniques suggested in this paper, namely column-scaling and row-partitioning can significantly reduce the run time of this algorithm. This is important in the context of the application of the algorithm to solving sparse linear systems because a typical application may involve solving the maximum weight bipartite matching problem hundreds or thousands of times.

We also present a comparison between augmenting-paths based and flow based algorithms for finding maximum weight matchings. We found that for all matrices in our test suite that come from real problems, the augmenting-paths based algorithms are significantly faster. Moreover, since the fast flow-based algorithms work with integer weights, they may not produce the absolutely optimal solution for graphs with real weights, unless the latter are converted to integer weights with a high degree of precision (which, in turn, can further slow down the flow based code). Further, it is not clear if a suitable matrix scaling, which is obtained free of cost from an augmenting-paths based primal-dual algorithm, can be obtained inexpensively through a flow based code. Therefore, we conclude that the augmenting-paths based Kuhn/Munkres algorithm is more suitable for our application of maximizing the absolute diagonal products of sparse matrices prior to LU factorization.

There are several issues pertaining to the maximum weight bipartite matching problem that need further investigation. The run time of the sparse Kuhn/Munkres algorithm is highly sensitive to the weights of the edges in the given graph. Exploring the possibility of formally expressing the complexity of the algorithm in terms a parameter based on matrix values (in addition to $|V|$ and $|E|$) could be one direction of future work in this area. We already present strong experimental evidence suggesting the dependence of the run time on the condition number of the sparse matrix corresponding to the graph.

Another important direction of future work in this area would be to explore efficient parallel algorithms for finding maximum cardinality and maximum bipartite matchings. To date, no parallel algorithms for these problems have been proposed whose processor-time product does not asymptotically exceed the serial complexity of the algorithm. A nested partition based row ordering has the potential to lead to an efficient parallel formulation of the sparse assignment problem for the class of bipartite graphs $((X, Y), E)$ for which a sublinear size separator of X exists. The augmenting path search in the two partitions can proceed independently and this parallelism can be applied recursively.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, 1993.

[2] David Avis and C. W. Lai. The probabilistic analysis of a heuristic for the assignment problem. *SIAM Journal on Computing*, 17(4):732–741, 1988.

[3] Egon Balas, Donald Miller, Joseph Pekny, and Paolo Toth. A parallel shortest augmenting path algorithm for the assignment problem. *Journal of the ACM*, 38(4):985–1004, October 1991.

[4] Claude Berge. Two theorems in graph theory. In *Proceedings of National Academy of Science, USA*, pages 842–844, 1957.

[5] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* Elsevier Science, 1976.

[6] R. E. Burkard and U. Derigs. Assignment and matching problems: Solution methods with FORTRAN programs. In *Lecture Notes in Economics and Mathematical Systems*, volume 184. Springer Verlag, 1980.

[7] Giorgio Carpaneto and Paolo Toth. Algorithm 548: Solution for the assignment problem. *ACM Transaction on Mathematical Software*, 6(1):104–111, 1980.

[8] Giorgio Carpaneto and Paolo Toth. Algorithm for the solution of the assignment problem for sparse matrices. *Computer*, 31:83–94, 1983.

[9] Giorgio Carpaneto and Paolo Toth. Primal-dual algorithms for the assignment problem. *Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 18:137–153, 1987.

[10] Boris V. Cherkassky, Andrew V. Goldberg, Joao C. Setubal, and Jorge Stolfi. Augment or push? A computational study of bipartite matching and unit capacity flow algorithms. *ACM Journal of Experimental Algorithmics (electronic journal)*, 3(*http://www.jea.acm.org/volume3.html*), 1998.

[11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, McGraw-Hill, New York, NY, 1990.

[12] U. Derigs. The shortest augmenting path method for solving assignment problems—motivation and computational experience. *Annals of Operations Research*, 4:57–102, 1985/6.

[13] U. Derigs and A. Metz. An efficient labeling technique for solving sparse assignment problems. *Computing*, 36:301–311, 1986.

[14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibbonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[16] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. Technical Report RAL-TR-1999-030, Rutherford Appleton Laboratory, April 19, 1999.

[17] Iain S. Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. Technical Report RAL-TR-97-059, Rutherford Appleton Laboratory, November 1997.

[18] Iain S. Duff and Torbjorn Wiberg. Remarks on implementations of $O(n^{1/2}\tau)$ assignment algorithms. *ACM Transactions on Mathematical Software*, 14:267–287, 1988.

[19] J. Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM*, 19:248–264, 1972.

[20] L. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton NJ, 1962.

[21] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[22] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18:1013–1036, 1989.

[23] Zvi Galil. Efficient algorithms for finding maximal matchings in graphs. Technical report, Department of Computer Science, Columbia University, 1983.

[24] Zvi Galil, Silvio Micali, and Harold N. Gabow. An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM Journal on Computing*, 15(1):120–130, February 1986.

[25] Andrew V. Goldberg and Robert Kennedy. An efficient cost scaling algorithm for the assignment problem. *Mathematical Programming*, 71:153–178, 1995.

[26] Andrew V. Goldberg, Serge A. Plotkin, David B. Shmoys, and Eva Tardos. Using interior-point methods for fast parallel algorithms for bipartite matchings and related problems. *SIAM Journal on Computing*, 21(1):140–150, 1992.

[27] Andrew V. Goldberg, Serge A. Plotkin, and Pravin M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. *Jounal of Algorithms*, 14:180–213, 1993.

[28] Anshul Gupta. WGPP: Watson graph partitioning (and sparse matrix ordering) package: Users manual. Technical Report RC 20453 (90427), IBM T. J. Watson Research Center, Yorktown Heights, NY, May 6, 1996.

[29] Bruce Hendrickson and Robert Leland. Chaco user's guide. Technical Report SAND93-2339, Sandia National Laboratories, Albuquerque, NM, October 1993.

[30] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[31] HSL. Harwell Ssubroutine Library, a catalogue of subroutines (Release 12). Technical report, AEA Technology, Harwell Laboratory, Oxfordshire, England, 1996.

[32] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987.

[33] Richard M. Karp. An algorithm to solve the $m \times n$ assignment problem in expected time $O(mn \log n)$. *Networks*, 10:143–152, 1980.

[34] Marek Karpinski and Wojciech Rytter. *Fast Parallel Algorithms for Graph Matching Problems.* Oxford University Press, 1998.

[35] George Karypis and Vipin Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1995.

[36] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[37] Eugene L. Lawler. *Combinatorial Optimization.* Holt, Rinehart and Winston, New York, NY, 1976.

[38] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Supercomputing '98 Proceedings*, 1998.

[39] Xiaoye S. Li and James W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[40] L. F. McGinnis. Implementation and testing of a primal-dual algorithm for the assignment problem. *Operations Research*, 31:277–291, 1983.

[41] Gary L. Miller, Shang-Hua Teng, William Thurston, and Stephen A. Vavasis. Geometric separators for finite-element meshed. *SIAM Journal on Scientific Computing*, 19(2):364–386, 1998.

[42] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of 31st Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[43] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of SIAM*, 5:32–38, 1957.

[44] Markus Olschowka and Arnold Neumaier. A new pivoting strategy for Gaussian elimination. *Linear Algebra and its Applications*, 240:131–151, 1996.

[45] Michael O. Rabin and Vijay V. Vazirani. Maximum matchings in general graphs through randomization. *Jounal of Algorithms*, 10:557–567, 1989.

[46] Baruch Schieber and Shlomo Moran. Slowing sequential algorithms for obtaining fast distributed and parallel algorithms: Maximum matchings. In *Proceedings, 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 282–292, 1986.