INFORMS Journal on Computing
Vol. 11, No. 2, Spring 1999

0899-1499/99/1102-0138 $05.00

Computing Minimum-Weight Perfect Matchings

WILLIAM COOK / *Computational and Applied Mathematics, Rice University, 6100 Main Street, Houston, TX 77005-1892, Email: bico@caam.rice.edu*

ANDRÉ ROHE / *Forschungsinstitut für Diskrete Mathematik, Universität Bonn, Lennéstr. 2, 53113 Bonn, Germany, Email: rohe@or.uni-bonn.de*

(Received: October 1997; revised: November 1997; accepted: September 1998)

We make several observations on the implementation of Edmonds' *blossom* algorithm for solving minimum-weight perfect-matching problems and we present computational results for geometric problem instances ranging in size from 1,000 nodes up to 5,000,000 nodes. A key feature in our implementation is the use of multiple search trees with an individual dual-change ϵ for each tree. As a benchmark of the algorithm's performance, solving a 100,000-node geometric instance on a 200 Mhz Pentium-Pro computer takes approximately 3 minutes.

A *perfect matching* in a graph G is a subset of edges such that each node in G is met by exactly one edge in the subset. Given a real weight c_e for each edge e of G, the *minimum-weight perfect-matching problem* is to find a perfect matching M of minimum weight $\Sigma(c_e : e \in M)$. One of the fundamental results in combinatorial optimization is the polynomial-time *blossom* algorithm for computing minimum-weight perfect matchings by Edmonds.[22, 23] This algorithm serves as a primary model for the development of methods for attacking combinatorial integer-programming problems. Moreover, efficient implementations of the algorithm permit the solution of large instances of matching problems that arise in practical situations.

A classic application of minimum-weight matchings is that of minimizing the "up" motion of a pen plotter, as described in Reingold and Tarjan[56] and in Iri, Murota, and Matsui.[40] Other applications include scheduling crews and vehicles in mass transit systems (Ball, Bodin, and Dial[7]), creating pairings in chess tournaments (Ólafsson[53]), selecting control groups in evaluations of experimental drugs (Clyde Monma [personal communication]), ordering arithmetic operations (Brandon Dixon and Arjen Lenstra [personal communication]), vehicle routing with time constraints (Derigs and Metz[20]), scheduling training sessions in the NASA space shuttle (Bell[9]), transmitting images over a network (Riskin, Ladner, Wang, and Atlas[58]), and capacitated vehicle routing (Miller and Pekny[52]).

Edmonds' matching algorithm has been studied by a great number of researchers. The efficiency of the algorithm, as measured by bounds on its worst-case running time, has been steadily improved over the past 30 years. The interest in efficient implementations is motivated to a large degree simply by the beauty of the algorithm itself, but it is also due to the role played by matchings in solution techniques for applied problems, such as those listed above.

A straightforward implementation of Edmonds' original description of the algorithm can easily be seen to run in time bounded by $O(n^2m)$, where n is the number of nodes in the graph and m is the number of edges. This was improved by Lawler[47] and by Gabow[27] to $O(n^3)$, and later to $O(nm \log n)$ by Galil, Micali, and Gabow.[33] A further improvement was made by Gabow, Galil and Spencer,[30] lowering the bound to $O(n(m \log\log \log_{\max\{m/n,2\}} n + n \log n))$. The log log log term was then removed by Gabow,[29] resulting in a bound of $O(n(m + n \log n))$. Some of the techniques that are used to established these results are surveyed in Ball and Derigs[8] and Galil.[32]

Gabow's bound is currently the best known result in terms of n and m, but other bounds are possible when the edge weights are integers. In this case, Edmonds' algorithm can be combined with scaling techniques to produce bounds that depend not only on n and m, but also on N, the largest magnitude of an edge weight. A result of this type was described by Gabow,[28] who obtained an $O(n^{3/4}m \log N)$ time bound. Gabow and Tarjan[31] later used a sophisticated approach to obtain a bound of $O(m \log(nN)\sqrt{n\alpha(n, m) \log n})$, where $\alpha(n, m)$ is Tarjan's[61] "inverse" of Ackerman's function.

A summary of these complexity results is given in Table I. One of the practical outcomes of this line of research has been a steady stream of ideas that can be incorporated into computer implementations of Edmonds' algorithm. This has helped to spur a parallel line of research aimed at creating robust computer codes for solving perfect-matching problems. A list of some of the studies in this area is given in Table II. With the exception of the cutting-plane methods employed by Grötschel and Holland[37] and Trick,[62] each of the papers listed in Table II presents an implementation of Edmonds' algorithm. A number of the papers report computational results on problem instances having more than 1,000 nodes, and Applegate and Cook[3] include results for instances with up to 131,072 nodes.

Despite this work on good implementations of Edmonds' algorithm, there has been a large body of research on heuristic methods for finding good, although perhaps not opti-

Subject classifications: Computational algorithms.
Other key words: Perfect matchings, networks/graphs, minimum-weight.

Table I. History of Worst-Case Bounds

Year	Authors	Running Time
1965	Edmonds[22, 23]	$O(n^2m)$
1973	Lawler[47]	$O(n^3)$
1974	Gabow[27]	$O(n^3)$
1985	Gabow[28]	$O(n^{3/4}m \log N)$
1986	Galil, Micali, and Gabow[33]	$O(nm \log n)$
1989	Gabow, Galil, and Spencer[30]	$O(n(m \log\log \log_{\max\{m/n,2\}} n + n \log n))$
1990	Gabow[29]	$O(n(m + n \log n))$
1991	Gabow and Tarjan[31]	$O(m \log(nN)\sqrt{n\alpha(n, m)} \log n)$

Table II. History of Computer Implementations

Year	Authors
1969	Edmonds, Johnson, and Lockhart[24]
1973	Pulleyblank[55]
1978	Cunningham and Marsh[14]
1980	Burkard and Derigs[12]
1980	Kazakidis[44]
1981	Derigs[15]
1982	Havel[38]
1983	Minoux[51]
1985	Grötschel and Holland[37]
1986	Derigs[16]
1986	Derigs and Metz[18]
1987	Trick[62]
1988	Derigs[17]
1989	Lessard, Rousseau, and Minoux[48]
1991	Derigs and Metz[19]
1991	Gerngross[35]
1993	Applegate and Cook[3]
1993	Atamtürk[5]
1995	Miller and Pekny[52]

mal, perfect matchings. Recent work in this area includes Bentley,[11] Imielinska and Kalantari,[39] Jünger and Pulleyblank,[43] and Williamson and Goemans,[63] surveys of earlier work can be found in Avis[6] and Gerards.[34] The research into heuristic algorithms is motivated, in part, by the fact that in many scenarios the time required to compute an optimal matching may not be justified. The main contribution of the present article is a new implementation of Edmonds' algorithm that hopefully will extend the range of instances where optimal solutions can be obtained.

Our implementation combines a number of techniques used in earlier efforts together with several new observations. The code appears to be significantly faster than previous implementations and it permits the solution of instances that are larger than those reported in earlier studies. Our test set includes geometric instances (complete graphs, described by points in the plane) generated randomly, as well as structured instances from Reinelt's[57] TSPLIB library of traveling salesman problem instances, and from VLSI design. The largest of the instances in our test set has 5,000,000 nodes. As a benchmark of the code's performance, solving a 100,000-node geometric instance on a 200 Mhz Pentium-Pro computer takes approximately 3 minutes.

This article is organized as follows. In Section 1 we present a short outline of Edmonds' algorithm and in Section 2 we discuss an idea that can be used to improve its practical performance on large-scale problem instances. In Section 3 we present a method for handling dense graphs, improving on the techniques developed by Ball and Derigs[8] and Applegate and Cook.[3] In Section 4 we report the results of our computational tests.

We will assume that the reader is familiar with basic results in matchings. Excellent general references are Gerards[34] and Lovász and Plummer.[50]

Our computer implementation is available for research purposes. The code is written in the C programming language (Kernighan and Ritchie[45]) and it can be obtained over the internet at http://www.or.uni-bonn.de/home/rohe/matching.html

1. Edmonds' Algorithm

Edmonds' algorithm is based on a linear-programming formulation of the minimum-weight perfect-matching problem. Linear-programming duality provides a stopping rule used by the algorithm to verify the optimality of a proposed solution.

Let G be a graph with node set V and edge set E. To describe the linear-programming formulation, let $\mathbb{O}$ denote the set of all odd subsets of V containing at least 3 nodes, and for each $S \subseteq V$, let $\delta(S)$ denote the set of edges that meet exactly one node in S. For a vector $(x_e : e \in E)$ and a set $H \subseteq E$, let $x(H)$ denote the sum $\Sigma(x_e : e \in H)$. The incidence vector of any perfect matching in a graph G satisfies the linear system

$$x(\delta(\{v\})) = 1 \quad \text{for all } v \in V, \tag{1}$$

$$x_e \geq 0 \quad \text{for all } e \in E, \tag{2}$$

$$x(\delta(S)) \geq 1 \quad \text{for all } S \in \mathbb{O}. \tag{3}$$

So the minimum weight of a perfect matching is at least as large as the value of

$$\min (wx : x \text{ satisfies (1), (2), and (3)}). \tag{4}$$

The dual to this linear-programming problem is

$$\max \sum(y_v : v \in V) + \sum(Y_S : S \in \mathbb{O}) \quad (5)$$

subject to

$$y_u + y_v + \sum(Y_S : S \in \mathbb{O},\ e \in \delta(S)) \geqslant w_e \quad \text{for all } e = uv \in E, \quad (6)$$

$$Y_S \geqslant 0 \quad \text{for all } S \in \mathbb{O}. \quad (7)$$

Given a dual solution $(\bar{y}, \bar{Y})$, the *reduced cost* of an edge $e = uv$, denoted by $slack(e)$, is

$$slack(e) := w_e - \bar{y}_u - \bar{y}_v - \sum(\bar{Y}_S : S \in \mathbb{O},\ e \in \delta(S)),$$

that is, the slack in the corresponding constraint (6). An edge is called *tight*, with respect to $(\bar{y}, \bar{Y})$, if its reduced cost is 0. Similarly, a set $S \in \mathbb{O}$ is called *full*, with respect to a (partial) matching $\bar{x}$, if $\bar{x}(\delta(S)) = 1$. With these definitions, the complementary slackness conditions for a primal-dual pair of solutions can be stated as: for all edges $e \in E$, if $\bar{x}_e > 0$, then e is tight, and for all sets $S \in \mathbb{O}$, if $\bar{Y}_S > 0$ then S is full. So we can prove that a specified perfect matching is optimal by providing a dual solution such that these conditions are satisfied. The remarkable result of Edmonds[22] is that such a proof of optimality always exists—indeed, it is constructed by the blossom algorithm.

At each step, Edmonds' algorithm has a matching and a dual solution that together satisfy the complementary slackness conditions. (As proposed by Derigs and Metz,[18] we can initialize these solutions by solving a linear programming relaxation of the matching problem.) The matching is grown via augmenting paths until we reach a perfect matching. To ensure that the complementary slackness conditions hold after an augmentation is carried out, the algorithm only considers augmenting paths made up entirely of tight edges. The heart of the algorithm is thus a search engine for finding such augmenting paths. We will not describe the algorithm, but we do need to indicate several of its components in order to present the new features in our implementation.

A key notion is that of *shrinking* a set $S \in \mathbb{O}$ into a single *pseudonode*. The intuition is that if $\bar{Y}_S > 0$ then the complementary slackness condition $\bar{x}(\delta(S)) = 1$ is the same as the constraint $\bar{x}(\delta(\{v\})) = 1$ for individual nodes v.

Given a matching $\bar{x}$ and a dual solution $(\bar{y}, \bar{Y})$, the algorithm searches for an augmenting path of tight edges in a graph that may possibly have some pseudonodes. (We will use "node" to refer to both original nodes and to pseudonodes.) To carry out the search, we choose an *unmatched* node r (that is, $\bar{x}(\delta(\{r\})) = 0$) and grow a tree T rooted at r having the following properties: each edge in T is tight and for each node v in T, the unique path in T from v to r alternates between matched edges ($\bar{x}_e = 1$) and unmatched edges ($\bar{x}_e = 0$). Such a tree T is called an *alternating tree*. The nodes of T are labeled "+" and "−" according to the parity of the number of edges in the path back to the root r, that is, node r and all nodes of even distance from r receive the label "+" and all nodes of odd distance receive the label "−". We *grow* T by appending matched edges that meet "−" nodes or tight unmatched edges that join "+" nodes to nodes not yet in T. If we reach an unmatched node v in T (other than r), then $\bar{x}$ can be *augmented* along the path from v to r, by replacing $\bar{x}_e$ by $1 - \bar{x}_e$ for each edge e in the path.

If the tree T has not reached an unmatched node and we cannot grow T any further, we attempt to alter the dual solution in order to create new tight edges, while keeping each edge in T tight. The form of the dual change is to add a nonnegative value ϵ to $\bar{y}_v$ for each "+" node v and to subtract ϵ from $\bar{y}_v$ for each "−" node v. We choose ϵ as large as possible, subject to the condition that after the dual change the complementary slackness conditions remain satisfied. The constraints on ϵ are therefore

$$\epsilon \leqslant slack(e) \quad \text{for each edge } e \text{ joining a "+" node to a node not in } T, \quad (8)$$

$$\epsilon \leqslant slack(e)/2 \quad \text{for each edge } e \text{ joining two "+" nodes,} \quad (9)$$

$$\epsilon \leqslant \bar{Y}_S \quad \text{for each set } S \in \mathbb{O} \text{ corresponding to a "−" pseudonode in } T. \quad (10)$$

If the bound on ϵ is determined by a constraint in (8), then after the dual change e is a new tight edge and we can grow T. If, on the other hand, ϵ is determined by a constraint in (9), then adding e to the tree T creates a unique circuit C. Notice that C must contain an odd number of nodes and pseudonodes, and thus determines a set $S \in \mathbb{O}$. The circuit C is then shrunk into a new pseudonode, and we again try to grow T. Finally, if a condition (10) bounds ϵ, then we *expand* the previously shrunk circuit corresponding to the pseudonode, adjust T to obtain a new alternating tree, and once again try to grow T.

This rough outline will suffice for our purposes. Detailed descriptions of the blossom algorithm can be found in Pulleyblank,[55] Ball and Derigs,[8] Gerards,[34] Cook, Cunningham, Pulleyblank, and Schrijver,[13] and elsewhere.

2. Variable Dual Changes

One of the fundamental decisions that must be made in an implementation of Edmonds' algorithm is whether to grow a single tree T from an unmatched node r or to simultaneously grow trees $T_1, T_2, \ldots, T_k$ from each of the unmatched nodes $r_1, r_2, \ldots, r_k$. It is easy to work out the details of the algorithm for either version, but it is not so easy to predict how the variants will behave in practice. Applegate and Cook[3] used a single tree in order to minimize the amount of overhead in the search procedure, but this sometimes forced the code to find long augmenting paths in cases where short paths were available (but starting from nodes other than r). The question is whether the added complexity of the multiple-tree version is worth the potential savings obtained by carrying out global searches for augmenting paths. Gerngross[35] made an extensive study of this issue and proposed a two-phased implementation, where the single-tree variant is used to match the first 95% of the nodes and the multiple-tree variant is used to match the remaining

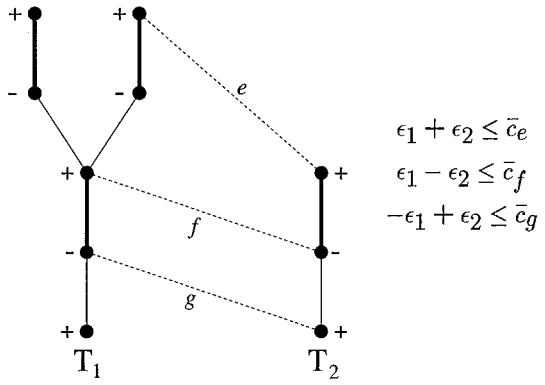

Figure 1. Constraints on dual change values.

5% of the nodes. Experimenting with Gerngross's approach, we were led to the improved procedure that we describe below.

The major drawback of the multiple-tree variant is that to compute the value of ϵ in a dual change we need to examine the edges meeting the "+" nodes in each of the (possibly many) trees, but typically the dual change will create new tight edges meeting only a very small number of these trees. We therefore perform a great deal of computation in order to make relatively little progress. A simple idea to overcome this difficulty is to allow each of the trees T_i to have their own dual change value $\epsilon_i \geq 0$. The values $(\epsilon_1, \ldots, \epsilon_k)$ will be constrained by (8), (9), and (10) for the corresponding trees T_i, together with the following constraints involving pairs of trees T_i and T_j, for $i, j \in \{1, \ldots, k\}$:

$$\epsilon_i + \epsilon_j \leq slack(e)$$

for each e joining a "+" in T_i to a "+" in T_j, (11)

$$\epsilon_i - \epsilon_j \leq slack(e)$$

for each e joining a "+" in T_i to a "−" in T_j. (12)

These additional restrictions on the dual change values are illustrated in Figure 1.

To make as much progress as possible in the dual objective function, we would like to choose $(\epsilon_1, \ldots, \epsilon_k)$ so as to maximize $\Sigma(\epsilon_i : i = 1, \ldots, k)$, subject to the constraints (8), (9), (10), (11), and (12). Computing such values $(\epsilon_1, \ldots, \epsilon_k)$ is a linear-programming problem and can thus be solved in polynomial time. (In fact, as observed by W.H. Cunningham [personal communication], this is the linear programming dual of a network optimization problem on a mixed graph.) It would be interesting to see how the use of an optimal solution to this linear-programming problem would impact the performance of Edmonds' algorithm, but this approach is unlikely to be of practical value due to the time required to solve the linear-programming problems.

An alternative idea is to employ a heuristic algorithm aimed at obtaining good, but perhaps not optimal, values $(\epsilon_1, \ldots, \epsilon_k)$. A first attempt would be to order the trees, $T_1, \ldots, T_k$, and then, for $i = 1, \ldots k$, greedily make ϵ_i as large possible. A difficulty with this approach is that the constraints on $(\epsilon_1, \ldots, \epsilon_k)$ may require that $\epsilon_i = \epsilon_j$ for some i and j and the greedy algorithm will therefore set both values to 0. To handle this, we can form a directed graph D on nodes $t_1, \ldots, t_k$, with a directed edge from t_i to t_j if and only if there is a tight edge $e \in E$ that joins a "+" node in T_i to a "−" node in T_j (such an edge constrains ϵ_i to be at most ϵ_j). The strongly connected components of D impose an equivalence relationship on the trees $T_1, \ldots, T_k$. If two trees T_i and T_j are in the same equivalence class, then we must have $\epsilon_i = \epsilon_j$ in any set of dual change values. We can therefore modify the greedy algorithm to first order the equivalence classes, and then, for each equivalence class in turn, greedily make the common value of ϵ_i for the trees T_i in the equivalence class as large as possible. In our computer code, we employ a simplified version of this idea, where we define the equivalence classes of the trees as the connected components of D. This requires slightly less overhead than computing the strongly connected components, and appears to work adequately in practice.

Table III. Sparse Graphs (Pentium-Pro, seconds)

Nodes	Edges	Single Tree	Multiple Trees	Variable ϵ
1,002	2,972	0.08	0.06	0.03
5,934	17,770	7.78	5.11	0.80
15,112	45,310	49.77	5.94	2.37
85,900	257,604	171.15	230.34	54.18

Table IV. Percentage Time for Final Augmentations

Nodes	Edges	Final Augmentation	Final 10 Augmentations
1,002	2,972	0.0%	50.0%
5,934	17,770	18.0%	58.4%
15,112	45,310	8.0%	38.3%
85,900	257,604	1.1%	78.7%

A computational comparison of the single-tree, multiple-tree, and the variable-ϵ methods is given in Table III. The problem instances are sparse graphs derived from geometric data sets that are described in Section 4 (the graphs are approximations of the Delaunay triangulation of the point set). The tests were run on a 200 Mhz Pentium-Pro computer (see Section 4 for more details on the computing platform).

Although the running times indicate that the variable-ϵ approach is worthwhile, there is no doubt room for improvement. One sign is that finding the last several augmentations in our code often takes more than half of the total computation time (see Table IV). This suggests that some other strategy (such as a priority queue-based implementation) may be preferable for the final stages of the algorithm.

3. Price and Repair

To solve large problem instances, we adopt the strategy of first computing the optimal matching over a sparse subset of

edges and then using linear-programming duality to guide us towards a solution over the entire edge set. This is a standard technique in combinatorial optimization—in this context it was first proposed by Derigs and Metz.[19]

There are many choices for the initial set of edges. Derigs and Metz[19] used the k-nearest graph, consisting of the k least costly edges meeting each node. Miller and Pekny[52] used the k-quad-nearest graph, defined as the k least costly edges in each of the four geometric quadrants around each node. In Applegate and Cook,[3] the "fractional k-nearest graph" was used; this graph is obtained by first solving a linear-programming relaxation of the matching problem and then choosing the k edges of least reduced cost meeting each node. In our study, we generally use the edges of an approximate Delaunay triangulation of the set of points. This edge set has the nice property that it is not too dense, but still captures well the structure of the point set and it contains a perfect matching (see Akl[2] and Dillencourt[21]).

Once we have solved the minimum-weight perfect-matching problem over the initial set of edges, we compute the reduced costs of the remaining edges. This procedure is called *pricing*. If all of the reduced costs are nonnegative, then our dual solution $(\bar{y}, \bar{Y})$ is a feasible solution to the full dual linear-programming problem and we can therefore conclude that the matching is optimal over the entire set of edges. If, on the other hand, some of the reduced costs are negative, then we cannot be sure that the matching we computed is optimal over the entire edge set. In this latter case, we can add some (or all) of the edges having negative reduced cost to our initial edge set, resolve the matching problem, and repeat the pricing procedure. This process can be iterated until we obtain an optimal matching over the entire edge set.

For this procedure to be successful on large problem instances, we need, first of all, an efficient pricing mechanism. In our code, we follow the ideas described in Applegate and Cook.[3] Namely, we use the least-common-ancestor algorithm of Aho, Hopcropt, and Ullman[1] to compute the reduced costs for sets of edges, taking advantage of the nested structure of the sets $\{S \in \mathbb{O} : \bar{Y}_S > 0\}$. Moreover, we use Applegate and Cook's underestimate of the reduced cost to avoid pricing the entire set of edges. This is accomplished by computing, for each node v, the value

$$sum(v) := \bar{y}_v + \sum\{\bar{Y}_S : S \in \mathbb{O},\ v \in S\}.$$

Then the reduced cost of an edge $e = uv$ is at least

$$\gamma(e) := c_e - sum(u) - sum(v),$$

so we need only to price those edge for which $\gamma(e) < 0$. Applegate and Cook describe a technique to avoid explicitly computing $\gamma(e)$ for every edge in the complete graphs determined by geometric problem instances, taking advantage of the distance function to rule out the possibility that certain edges have $\gamma(e) < 0$. We do not adopt this aspect of the Applegate-Cook process, using instead a *kd*-tree (see Bentley[11]), treating $sum(\cdot)$ as an extra geometric coordinate and using $\gamma(\cdot)$ as our distance function. This allows us to locate edges having $\gamma(e) < 0$ using the standard nearest-neighbor search algorithms for *kd*-trees. This idea is similar to the technique used by Johnson, McGeoch, and Rothberg[42] for computing spanning trees in the Held-Karp procedure for the traveling salesman problem.

If a pricing phase produces many edges having negative reduced cost, then it may make sense to resolve the new perfect-matching problem from scratch, since the addition of many edges will most likely cause both the perfect matching and the dual solution to change considerably. If, however, we have relatively few edges of negative reduced cost, then it may be quite wasteful to simply throw away our matching and dual solution. Instead, we would like to *repair* the matching by inserting the new edges into our existing solutions. Ball and Derigs,[8] building on earlier work of Weber,[64] described an elegant method for accomplishing this. Suppose we wish to add the edge $e = uv$ to the initial edge set. Their method begins by carrying out a sequence of dual changes and pseudonode expansions (preserving the complementary slackness conditions) so that one end of e, say v, is no longer contained in any pseudonode. Let f be the matching edge meeting v, and set $\bar{x}_f = 0$. Now we can decrease the value of $\bar{y}_v$ so that the reduced cost of edge e becomes 0. We then have a matching $\bar{x}$ and a dual solution $(\bar{y}, \bar{Y})$ that satisfy the complementary slackness conditions (except that we may have an unmatched pseudonode), so we can search for an augmenting path to restore $\bar{x}$ to an optimal perfect matching.

The Ball-Derigs method is very clean, but it does have some practical drawbacks, as does the slightly improved procedure used by Applegate and Cook.[3] Firstly, the dual steps and the primal steps often fight one another: the dual steps expand pseudonodes to uncover node v, and the primal steps shrink v back into a chain of pseudonodes in order to find an augmenting path. If we have more than one edge to add to our initial set, then it is advantageous to delay the primal steps until each of the edges has been added. This will result in a graph having a number of unmatched nodes and pseudonodes that can be matched with the variable-ϵ approach.

Secondly, the Ball-Derigs method requires a great deal of computational effort to ensure that the complementary slackness conditions continue to hold for the matching $\bar{x}$ and dual solution $(\bar{y}, \bar{Y})$. If we give up this requirement, we can simply expand each pseudonode containing v, setting $\bar{x}_f = 0$ for the matching edges f that meet the pseudonodes. This will result in a graph having a greater number of unmatched nodes and pseudonodes, but we can once again apply the variable-ϵ approach to restore $\bar{x}$ to an optimal perfect matching. We call this simplified procedure *careless repairs*, since we have dropped most of the constraints that guide the Ball-Derigs procedure.

In Table V, we compare careless repairs, the Ball-Derigs procedure, and the procedure of simply resolving the matching problems from scratch after each pricing iteration. The times reported are for solving the matching problem over the complete graph specified by the geometric problem instances, starting with the Delaunay edge set. It is interesting to note the good performance of the resolve method. This can be explained by the results we presented in Table

Table V. Comparison of Repair Routines (Pentium-Pro, seconds)

Nodes	Ball-Derigs	Resolve	Careless
1,002	0.20	0.27	0.15
5,934	9.55	6.63	4.19
15,112	833.27	15.16	11.73
85,900	5557.79	236.77	155.84

Table VI. Test Instances

Name	Nodes	Source
pr1002	1,002	TSPLIB
pcb3038	3,038	TSPLIB
rl5934	5,934	TSPLIB
usa13509	13,508	TSPLIB
d15112	15,112	TSPLIB
kanto	20,726	Map of Tokyo
pla85900	85,900	TSPLIB
p626628	626,628	VLSI (Bonn)
p2184278	2,184,278	VLSI (Bonn)

IV, showing that, with the variable-ϵ approach, completing a nearly-perfect matching to a perfect matching is sometimes close to being as difficult as computing the perfect matching from scratch. We take advantage of this fact to make our code more robust when dealing with initial edge sets that do not provide a good representation of the full graph: when we find greater than $n/32$ edges having negative reduced cost (in an instance having n nodes), then we simply resolve the matching problem from scratch, rather than calling our repair routine.

4. Computational Results

We refer to our computer code as *Blossom IV*, following the names "Blossom I" through "Blossom III" used by Edmonds, Johnson, and Lockhart,[24] Pulleyblank,[55] and Cunningham and Marsh,[14] respectively.

A crucial component in any efficient implementation of Edmonds' algorithm is the choice of the data structures for maintaining the search tree and the blossom family. In Blossom IV we employ the framework described in Pulleyblank[55] (see also Applegate and Cook[3]).

We tested Blossom IV on a variety of geometric problem instances, both structured and randomly generated, as well as on some sparse graphs obtained from these instances. The structured instances are listed in Table VI. Most of these examples can be found in the TSPLIB library, maintained by Gerd Reinelt.[57] We worked with a selection of the TSPLIB instances having between 1,002 and 85,900 nodes. The original data set for "usa13509" contains an odd number of points; for this instance, we follow the practice of Applegate and Cook[3] and drop the last point after sorting the x, y coordinates. The "kanto" instance is described in Asano, Edahiro, Imai, Iri, and Murota.[4] The two large VLSI instances were obtained from the VLSI design project at the Research Institute for Discrete Mathematics at the University of Bonn. For all instances other than "kanto," the edge weights are defined as the Euclidean distance (rounded to the nearest integer) between the points corresponding to the end nodes of the edges (this is the distance function specified in TSPLIB). The edge weights for "kanto" are defined similarly, but with the L_∞ norm used instead of the L_2 norm.

Table VII. Blossom IV Running Times (seconds)

Name	IBM 590	200 Mhz Pentium-Pro	AlphaServer 4100
pr1002	0.42	0.25	0.12
pcb3038	1.42	0.96	0.45
rl5934	7.79	5.76	2.56
usa13509	16.39	13.36	5.42
d15112	17.19	12.81	5.98
kanto	86.27	63.33	24.73
pla85900	210.84	165.19	76.76
p626628	36634.15	31215.27	12931.03
p2184278	Not Run	Not Run	279143.68

The randomly generated examples that we consider have integer coordinates drawn uniformly from the N by N square, when N is the number of nodes in the instance. We use the "lprand" generator that is available as part of the DIMACS Challenge that was organized by Johnson and McGeoch.[41] The generator is described in Bentley[10] and is based on Algorithm A in Section 3.2.2 of Knuth.[46] It has the nice property that on most machine types it will produce the identical sequence of integers for a given seed. We use the rounded Euclidean distance to define the edge weights in these random problem instances.

Our computational tests were carried out on three different computing platforms: an IBM RS6000, Model 590 running IBM's AIX operating system and using the IBM xlc compiler with the options "−O2 −Q=20"; a Hewlet Packard Vectra XU 6200 with a 200 Mhz Pentium-Pro processor and 256k cache, running Sun Solaris and compiled with the GNU gcc compiler using optimization level −O3; and a Digital AlphaServer 4100 (400 Mhz processor) running Digital Unix and compiled with Digital's cc compiler with the options "-tune host -O4". In Table VII, we report the running times on these three machines for the complete set of structured instances (the two VLSI instances were not run on all three machines due to memory and time limitations). In these tests we used the Delaunay graph as the initial edge set, as computed by the "sweep2" code of Fortune,[25, 26] with the exception of the L_∞-norm instance "kanto" (sweep2 requires L_2-norm instances), where we use the union of the 1-quad-nearest edge set and the edge set of a nearest-neighbor traveling salesman tour for the set of points. The running times given in the table include all phases of the algorithm: initial edge set generation, matching the initial set, and price-repair. The times for the Pentium-Pro are roughly 1.3

Table VIII. Applegate-Cook (AlphaServer 4100, seconds)

Name	Applegate-Cook	Blossom IV	Speedup
pr1002	0.57	0.12	4.8
pcb3038	2.05	0.45	4.6
rl5934	26.55	2.56	10.4
usa13509	474.81	5.42	87.6
d15112	75.16	5.98	12.6
kanto	1001.74	24.73	40.5
pla85900	559.65	76.76	5.9

Table IX. Applegate-Cook on Random Instances (AlphaServer 4100, seconds)

Nodes	Applegate-Cook	Blossom IV	Speedup
10,000	106.69	3.09	34.5
100,000	40063.40	92.15	434.7
250,000	726347.22	272.70	2663.5

Table X. Applegate-Cook on Fractional Nearest 10 (AlphaServer 4100, seconds)

Nodes	Edges	Applegate-Cook	Blossom IV	Speedup
10,000	58,583	67.11	1.60	41.9
100,000	585,072	23813.34	28.35	840.0
250,000	1,462,226	334009.88	114.10	2927.3

times faster than the IBM 590, and the times for the AlphaServer 4100 are roughly 3.0 times faster than the IBM 590 and 2.3 times faster than the Pentium-Pro. (We note that the relatively poor running times for p626628 and p2184278 are due to the time required in the pricing phase of the algorithm. For these large, structured instances, the running times can be greatly reduced by working with a more dense initial edge set.)

It is a difficult task to properly compare the performance of Blossom IV with earlier implementations, due both to the unavailability of the earlier codes and to the fact that the early codes were written for greatly different computing platforms. We limit ourselves to a comparison with the code of Applegate and Cook,[3] which appears to be the best performing code of the earlier implementations—a comparison with the code of Derigs[17] is given in Applegate and Cook.[3] In Table VIII, we report the times for Applegate-Cook on our test bed. For these instances, Blossom IV ranged from 4 to 87 times faster than the earlier code. It should be noted, however, that Applegate-Cook is known to perform well over structured instances such as those in our test set. Indeed, in tests carried out by Williamson and Goemans,[63] they found that Applegate-Cook (for computing optimal matchings) was usually faster than their own implementation of the Goemans and Williamson[36] matching heuristic

Table XI. Random Geometric Problems (IBM 590, seconds)

Nodes	Trials	Mean Time	Max Time	Min Time
1,000	100	0.36	0.57	0.23
2,000	100	0.89	1.46	0.54
5,000	100	2.99	4.84	2.07
10,000	100	7.59	11.81	5.59
20,000	100	18.34	29.43	13.63
50,000	100	69.87	165.06	45.03
100,000	100	189.06	538.13	110.13
200,000	100	581.16	2970.35	265.20
500,000	100	2317.63	12609.14	1045.71
1,000,000	33	11819.44	83621.12	2843.38
2,000,000	11	61864.41	207752.32	8297.41

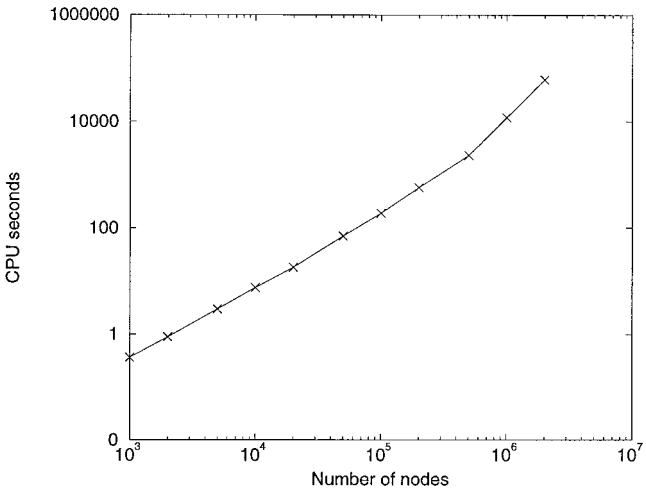

Figure 2. Log-log plot of random instances (IBM 590).

Table XII. 5,000,000 Node Random Instances (10 trials, IBM 590, seconds)

Initial Edge Set	Mean Time	Max Time	Min Time
LK10 + Nearest 2	150978.66	453890.36	57364.17

on TSPLIB problems, whereas for large randomly generated problems the heuristic was up to 2 times faster than Applegate-Cook. It is therefore not surprising that Blossom IV obtains a greater speedup on large random instances, as indicated in Table IX.

Blossom IV has an advantage over the Applegate-Cook code in that its superior price-repair routine permits it to work efficiently with a smaller initial edge set. The initial set used by Applegate-Cook is the "fractional 10-nearest"; this is a denser graph than the Delaunay graph used by Blossom IV. Moreover, the computation time needed to construct the fractional 10-nearest edge set is considerably more than that

Table XIII. Computational Estimates on β_M

Year	Authors	Estimate	Largest Test Instance
1977	Papadimitriou[54]	$\beta_M \approx 0.35$	200 nodes
1983	Iri, Murota, and Matsui[40]	$0.32 \leqslant \beta_M \leqslant 0.33$	250 nodes
1986	Weber and Liebling[65]	$\beta_M \approx 0.3189$	1,000 nodes

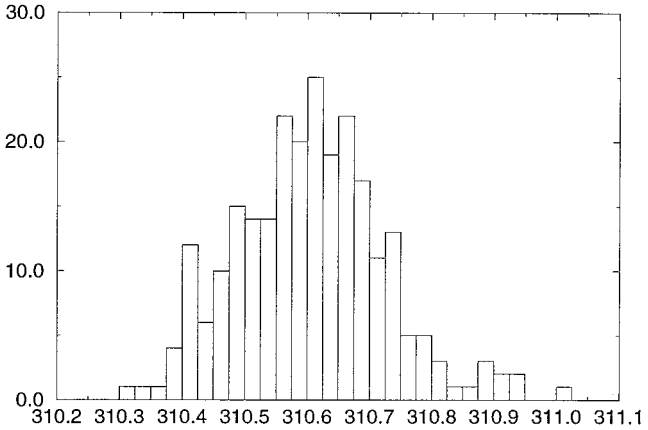

Figure 3. Lengths of 1,000,000 node random instances.

Table XIV. Delaunay Matching (Pentium-Pro, seconds)

Name	Time	Cost	Optimal Cost	% Gap	Speedup
pr1002	0.07	112723	112630	0.083	3.6
pcb3038	0.31	64489	64487	0.003	3.1
rl5934	1.09	246887	246834	0.021	5.3
usa13509	3.25	8839441	8838275	0.013	4.1
d15112	3.09	720699	720617	0.011	4.1
pla85900	61.13	67648209	67647278	0.001	2.7

Table XV. Delaunay Matching on Random Instances (IBM 590, seconds)

Nodes	Trials	Mean Time	Mean % Gap	Max % Gap	Speedup
1,000	100	0.04	0.043	0.257	9.73
10,000	100	1.24	0.036	0.070	6.11
100,000	100	40.33	0.035	0.045	4.69
500,000	100	416.30	0.035	0.040	5.57
1,000,000	33	2018.34	0.035	0.037	5.86
2,000,000	11	3834.78	0.035	0.036	16.13

Table XVI. Delaunay Graphs with (0–9,999) Edge Weights (IBM 590, seconds)

Nodes	Trials	Mean Time	Max Time	Min Time
1,000	100	0.08	0.23	0.03
2,000	100	0.23	0.55	0.09
5,000	100	1.10	3.75	0.24
10,000	100	3.04	9.52	1.07
20,000	100	9.89	35.51	3.16
50,000	100	41.68	187.01	11.71
100,000	100	105.31	268.88	43.30
200,000	100	281.54	1109.93	68.83
500,000	10	1230.13	4861.29	399.23
1,000,000	10	2346.22	5650.23	1158.77

needed by the fast Delaunay graph codes of Fortune,[25] Shewchuk,[60] and others. We remark, however, that even just solving over the fractional 10-nearest, Blossom IV is significantly faster than Applegate-Cook, as indicated in Table X.

To give an indication of the growth in running time for Blossom IV as the problem size increases, we present in Table XI results for our code over a range of random geometric instances. In these instances, the Delaunay graph was computed using the "triangle" code of Shewchuk.[60] (We found "triangle" to be more robust than "sweep2" when dealing with large point sets.) A log-log plot of these values is given in Figure 2. The running times are mean values over a number of independent instances for each problem size, as indicated in the table. It should be noted that the running times increase if the points are distributed in a larger square, for example, working in a $100N$ by $100N$ square increases the mean time by about a factor of 2 for instances on 50,000 nodes. (This is due to the increase in the number of dual changes brought on by the greater precision in the integer edge lengths.)

The plot in Figure 2 indicates that the growth in running time is modest enough to consider solving even larger problem instances. A difficulty that we encounter, however, is that the memory required by the computer code is also growing with the number of nodes.

One of the main contributors to the memory usage in Blossom IV is the storage for the edges. It is natural, therefore, to consider using an initial edge set that is less dense than the Delaunay graph. There are many possibilities for such an edge set; one that we tested consists of the union of 10 matchings produced by a Lin-Kernighan heuristic for perfect matchings, similar to the well known heuristic of Lin and Kernighan[49] for the traveling salesman problem. (For details of this matching heuristic see Rohe.[59]) For small

Table XVII. 100,000 Node Delaunay Graphs with Random Edge Weights (IBM 590, seconds)

Edge Weights	Trials	Mean Time	Max Time	Min Time
0–9	100	18.35	131.71	5.52
0–99	100	21.09	145.50	11.71
0–999	100	45.06	107.75	27.03
0–9,999	100	105.31	268.88	43.30
0–99,999	100	150.33	501.84	45.96
0–999,999	100	158.44	581.54	47.10
0–9,999,999	100	159.06	586.32	46.10
0–99,999,999	100	158.45	591.21	46.19

instances, this edge set is not practical, since it requires as much time to compute as the total computation time for Blossom IV starting with the Delaunay graph. At 100,000 nodes, computing the Lin-Kernighan edge set requires roughly one-third of the time needed by Blossom IV with the Delaunay graph, and the total running time is about 1.5 times the Delaunay version. At 1,000,000 nodes, the time needed to compute the edge set is about one-tenth of the time for Blossom IV with Delaunay, and the total running time is only about 25% slower than using the Delaunay version. The additional slowdown for the Lin-Kernighan version of the code (over the time needed to generate the edge set) results from the fact that the very sparse initial set can lead to a larger number of rounds of price-repair. To partially offset this effect, we add the nearest-2 edge set to the union of the 10 matchings when we consider large instances. To test this approach, we ran Blossom IV on 10 instances having 5,000,000 nodes. The running times for this test (on an IBM RS6000, 590) are given in Table XII.

The fact that Blossom IV can solve large random problem instances suggests that it may be a useful tool in pursuing the study of the asymptotic behavior of the length of the optimum matchings. Papadimitriou[54] has shown that there exists a constant β_M, such that if (x_1, y_1), (x_2, y_2), . . . is an infinite sequence of independent, uniformly distributed points in the unit square, and M_n denotes the length of the minimum-weight perfect matching on the points $(x_1, y_1), \ldots, (x_{2n}, y_{2n})$, then $M_n/\sqrt{n}$ converges almost surely to β_M. In Table XIII we list estimates on β_M that have been obtained by a number of researchers via computational experiments. Williamson and Goemans[63] argue that, rather than simply using $\beta_M\sqrt{n}$, the matching length M_n can be more accurately predicted with an estimator of the form

$$\beta_M \sqrt{n} + \alpha_M.$$

Using the computer code of Applegate and Cook[3] to solve a range of problem instances (including 4 instances having 131,072 nodes), they estimated that $\beta_M \approx 0.3103$ and $\alpha_M \approx 0.2357$. Thus, for 1,000,000 node instances the Williamson-Goemans estimate is $M_n = 310.5357$. We used Blossom IV to compute optimal matchings for 250 random instances having 1,000,000 nodes (using the seeds 1 through 250 with "lprand"). The histogram of the lengths of the matchings is given in Figure 3. The mean of the 250 lengths is 310.6052, and thus fits reasonably well with the Williamson-Goemans estimate.

Although the goal of Blossom IV is the exact solution of large scale instances, it should be noted that the code can also be used as a heuristic algorithm by only solving over the initial edge set, skipping the price-repair phase of the code. Indeed, as we indicate in Table XIV, very good quality matchings can be obtained by optimizing just over the Delaunay graph. In each of our test instances, the cost of the optimal matching in the Delaunay graph is within one-tenth of one percent of the cost of the optimal matching over the complete graph. The times reported in Table XIV include the time used by the "sweep2" code of Fortune[26] to compute the Delaunay graph. The speedup over the time to solve the complete graph was 2.7 or better in all cases. Similar results

Table XVIII. Running Times for pla85900 (seconds)

Machine	Compiler	Time	Speedup
Sun Sparc 10, Model 41	gcc -O3	522.27	1.0
IBM RS6000, Model 550	gcc -O3	516.08	1.0
IBM RS6000, Model 43p (133 Mhz)	xlc -O2 -Q=20	258.85	2.0
SGI Indigo 2, Impact (250 Mhz, R4400)	cc -O2	215.42	2.1
IBM RS6000, Model 590	xlc -O2 -Q=20	210.84	2.5
Digital Alpha XL 266	gcc -O3	198.61	2.6
Sun Ultra 1, Model 140	gcc -O3	172.44	3.0
HP Vectra XU 6200	gcc -O3	165.19	3.2
SGI Indigo 2, Impact 10000	cc -O2	156.82	3.3
IBM RS6000, Model 595	xlc -O2 -Q=20	139.06	3.8
Sun Ultra 2, Model 200	gcc -O3	122.42	4.3
Digital Alpha XL 366	gcc -O3	111.46	4.7
DCG EV56 (500 Mhz Alpha, 2 Mbyte cache)	gcc -O3	83.85	6.2
Digital AlphaServer 4100 (400 Mhz)	gcc -O3	78.77	6.6

hold for our tests on random graphs, using "triangle" to compute the Delaunay graphs, as reported in Table XV.

Up to this point, each of the instances we have considered have edge weights determined by some geometric norm. To give a comparison with non-geometric instances, in Table XVI we report times on random Delanuay graphs where the integer edge weights are chosen at random (uniformly) from the interval 0–9,999. In Table XVII, we give an indication of the growth in the running time as the spread of the random weights is increased. Notice that for these 100,000-node instances, the running time appears to level off after we reach the point where most of the edges receive distinct weights.

Finally, we report in Table XVIII the solution time for pla85900 across a number of different computing platforms. This gives a rough comparison of the various machines for this type of combinatorial computing.

Acknowledgments

The authors would like to thank David Applegate for his help on implementing the blossom algorithm, W.H. Cunningham, W.R. Pulleyblank, and F.B. Shepherd for their suggestions on the variable dual change criterion, and M. Goemans for his remarks about perfect matchings in Delaunay triangulations.

References

1. A.V. AHO, J.E. HOPCROFT, and J.D. ULLMAN, 1976. On Finding Lowest Common Ancestors in Trees, *SIAM Journal of Computing 5*, 115–132.
2. S.G. AKL, 1983. A Note on Euclidean Matchings, Triangulations and Spanning Trees, *Journal of Combinatorics, Information and System Sciences 8*, 175–180.
3. D. APPLEGATE and W. COOK, 1993. Solving Large-Scale Matching Problems, in *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch (eds.), American Mathematical Society, Providence, RI, 557–576.
4. T. ASANO, M. EDAHIRO, H. IMAI, M. IRI, and K. MUROTA, 1985. Practical Use of Bucketing Techniques in Computational Geometry, in *Computational Geometry*, G.T. Toussaint (ed.), North Holland, Amsterdam, 153–195.
5. A. ATAMTÜRK, 1993. Efficient Algorithms for the Minimum Cost Perfect Matching Problem on General Graphs, Master of Science Thesis, Bilkent University, Turkey.
6. D. AVIS, 1983. A Survey of Heuristics for the Weighted Matching Problem, *Networks 13*, 475–493.
7. M.O. BALL, L.D. BODIN, and R. DIAL, 1983. A Matching Based Heuristic for Scheduling Mass Transit Crews and Vehicles, *Transportation Science 17*, 4–31.
8. M.O. BALL and U. DERIGS, 1983. An Analysis of Alternative Strategies for Implementing Matching Algorithms, *Networks 13*, 517–549.
9. C.E. BELL, 1994. Weighted Matching with Vertex Weights: An Application to Scheduling Training Sessions in NASA Space Shuttle Cockpit Simulators, *European Journal of Operational Research 73*, 443–449.
10. J. BENTLEY, 1992. Software Exploratorium: Some Random Thoughts, *Unix Review 10*, 71–77. (Computer code available via anonymous ftp: ftp://dimacs.rutgers.edu/pub/netflow/generators/universal.c)
11. J.L. BENTLEY, 1992. Fast Algorithms for Geometric Traveling Salesman Problems, *ORSA Journal on Computing 4*, 387–411.
12. R.E. BURKARD and U. DERIGS, 1980. *Assignment and Matching Problems: Solution Methods with FORTRAN-Programs*, Springer Lecture Notes in Mathematical Systems 184.
13. W.J. COOK, W.H. CUNNINGHAM, W.R. PULLEYBLANK, and A. SCHRIJVER, 1998. *Combinatorial Optimization*, Wiley, New York.
14. W.H. CUNNINGHAM and A.B. MARSH, III, 1978. A Primal Algorithm for Optimum Matching, *Mathematical Programming Study 8*, 50–72.
15. U. DERIGS, 1981. A Shortest Augmenting Path Method for Solving Minimal Perfect Matching Problems, *Networks 11*, 379–390.
16. U. DERIGS, 1986. Solving Large-Scale Matching Problems Efficiently: A New Primal Matching Approach, *Networks 16*, 1–16.
17. U. DERIGS, 1988. Solving Non-Bipartite Matching Problems via Shortest Path Techniques, *Annals of Operations Research 13*, 225–261.
18. U. DERIGS and A. METZ, 1986. On the Use of Optimal Fractional Matchings for Solving the (Integer) Matching Problem, *Computing 36*, 263–270.
19. U. DERIGS and A. METZ, 1991. Solving (Large Scale) Matching Problems Combinatorially, *Mathematical Programming 50*, 113–122.
20. U. DERIGS and A. METZ, 1992. A Matching-Based Approach for Solving a Delivery/Pick-Up Vehicle Routing Problem with Time Constraints, *Operations Research Spektrum 14*, 91–106.
21. M.B. DILLENCOURT, 1990. Toughness and Delaunay Triangulations, *Discrete and Computational Geometry 5*, 575–601.
22. J. EDMONDS, 1965. Maximum Matching and a Polyhedron with 0,1-Vertices, *Journal of Research of the National Bureau of Standards 69B*, 125–130.
23. J. EDMONDS, 1965. Paths, Trees and Flowers, *Canadian Journal of Mathematics 17*, 449–467.
24. J. EDMONDS, E.L. JOHNSON, and S.C. LOCKHART, 1969. Blossom I: A Computer Code for the Matching Problem, Unpublished Report, IBM T.J. Watson Research Center, Yorktown Heights, New York.
25. S.J. FORTUNE, 1987. A Sweepline Algorithm for Voronoi Diagrams, *Algorithmica 2*, 153–174.
26. S.J. FORTUNE. "sweep2", computer code available via anonymous ftp from netlib.att.com
27. H.N. GABOW, 1974. *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs*, Ph.D. Thesis, Stanford University.
28. H.N. GABOW, 1985. A Scaling Algorithm for Weighted Matching on General Graphs, in *Proceedings 26th Annual Symposium of the Foundations of Computer Science*, IEEE Computer Society, Los Angeles, 90–100.
29. H.N. GABOW, 1990. Data Structures for Weighted Matching and Nearest Common Ancestors with Linking, in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, Association for Computing Machinery, New York, 434–443.
30. H.N. GABOW, Z. GALIL, and T.H. SPENCER, 1989. Efficient Implementation of Graph Algorithms using Contraction, *Journal of the ACM 36*, 540–572.
31. H.N. GABOW and R.E. TARJAN, 1991. Faster Scaling Algorithms for General Graph Matching Problems, *Journal of the ACM 38*, 815–853.
32. Z. GALIL, 1986. Efficient Algorithms for Finding Maximum Matchings in Graphs, *ACM Computing Surveys 18*, 23–38.
33. Z. GALIL, S. MICALI, and H.N. GABOW, 1986. An $O(EV \log V)$ Algorithm for Finding a Maximal Weighted Matching in General Graphs, *SIAM Journal of Computing 15*, 120–130.
34. A.M.H. GERARDS, 1995. Matching, in *Network Models*, M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser (eds.), North Holland, Amsterdam.
35. P. GERNGROSS, 1991. Zur Implementation von Edmonds' Match-

ing Algorithmus: Datenstrukturen und verschiedene Varianten, Diplomarbeit, Institut für Mathematik, Universität Augsburg.
36. M.X. GOEMANS and D.P. WILLIAMSON, 1995. A General Approximation Technique for Constrained Forest Problems, *SIAM Journal on Computing 24*, 296–317.
37. M. GRÖTSCHEL and O. HOLLAND, 1985. Solving Matching Problems with Linear Programming, *Mathematical Programming 33*, 243–259.
38. T.F. HAVEL, 1982. *The Combinatorial Distance Geometry Approach to the Calculation of Molecular Conformation*, Ph.D. Thesis, Biophysics Program, University of California, Berkeley.
39. C. IMIELINSKA and B. KALANTARI, 1993. A Generalized Hypergreedy Algorithm for Weighted Perfect Matching, *BIT 33*, 178–189.
40. M. IRI, K. MUROTA, and S. MATSUI, 1983. Heuristics for Planar Minimum-Weight Perfect Matchings, *Networks 13*, 67–92.
41. D.S. JOHNSON and C.C. MCGEOCH, 1993. *Network Flows and Matching—First DIMACS Implementation Challenge*, American Mathematical Society, Providence, RI.
42. D.S. JOHNSON, L.A. MCGEOCH, and E.E. ROTHBERG, 1996. Asymptotic Experimental Analysis for the Held-Karp Traveling Salesman Bound, in *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, Association for Computing Machinery, New York, 341–350.
43. M. JÜNGER and W. PULLEYBLANK, 1995. New Primal and Dual Matching Heuristics, *Algorithmica 13*, 357–380.
44. G. KAZAKIDIS, 1980. Die Lösung minimaler perfekter Matchingprobleme mitte ls kürzester erweiternder Pfade," Diplomarbeit, Mathematiches Institut der Universität Köln.
45. B.W. KERNIGHAN and D.M. RITCHIE, 1978. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey.
46. D.E. KNUTH, 1969. *Seminumerical Algorithms—The Art of Computer Programming, Vol. 2*, Addison-Wesley, Reading.
47. E.L. LAWLER, 1976. *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York.
48. R. LESSARD, J.-M. ROUSSEAU, and M. MINOUX, 1989. A New Algorithm for General Matching Problems using Network Flow Subproblems, *Networks 19*, 459–479.
49. S. LIN and B.W. KERNIGHAN, 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem, *Operations Research 21*, 498–516.
50. L. LOVÁSZ and M.D. PLUMMER, 1986. *Matching Theory*, Akadémia i Kiadoó, Budapest.
51. M. MINOUX, 1983. A New Polynomial Cutting-Plane Algorithm for Maximum Weight Matchings in General Graphs, Publication Number 302, Centre de recherche sur les transports, Université de Montréal.
52. D.L. MILLER and J.F. PEKNY, 1995. A Staged Primal-Dual Algorithm for Perfect *b*-Matching with Edge Capacities, *ORSA Journal on Computing 7*, 298–320.
53. S. ÓLAFSSON, 1990. Weighted Matching in Chess Tournaments, *Journal of the Operational Research Society 41*, 17–24.
54. C.H. PAPADIMITRIOU, 1977. The Probabilistic Analysis of Matching Heuristics, in *Proceedings of the 15th Annual Allerton Conference on Communication, Control, and Computing*, 368–378.
55. W.R. PULLEYBLANK, 1973. *Faces of Matching Polyhedra*, Ph.D. Thesis, University of Waterloo, Waterloo, Ontario.
56. E.M. REINGOLD and R.E. TARJAN, 1981. On a Greedy Heuristic for Complete Matching, *SIAM Journal of Computing 10*, 676–681.
57. G. REINELT, 1995. TSPLIB95, *Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR)*, Heidelberg. (Manuscript and problem instances available at http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html)
58. E.A. RISKIN, R. LADNER, R.-Y. WANG, and L.E. ATLAS, 1994. Index Assignment for Progressive Transmission of Full-Search Vector Quantization, *IEEE Transactions on Image Processing 3*, 307–312.
59. A. ROHE, 1997. Parallele Heuristiken für sehr große Traveling Salesman Probleme, Diplomarbeit, Research Institute for Discrete Mathematics, Universität Bonn.
60. J.R. SHEWCHUK, 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, in *First Workshop on Applied Computational Geometry*, Association for Computing Machinery, New York, 124–133. (Computer code available at http://www.cs.cmu.edu/quake/triangle.html)
61. R.E. TARJAN, 1983. *Data Structures and Network Algorithms*, SIAM, Philadelphia.
62. M. TRICK, 1987. *Networks with Additional Structured Constraints*, Ph.D. Thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia.
63. D.P. WILLIAMSON and M.X. GOEMANS, 1996. Computational Experience with an Approximation Algorithm on Large-Scale Euclidean Matching Instances, *INFORMS Journal on Computing 8*, 29–40.
64. G.M. WEBER, 1981. Sensitivity Analysis of Optimal Matchings, *Networks 11*, 41–56.
65. M. WEBER and T.M. LIEBLING, 1986. Euclidean Matching Problems and the Metropolis Algorithm, *Zeitschrift für Operations Research 30*, A85–A110.