DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

Solving Large-Scale Matching Problems

DAVID APPLEGATE AND WILLIAM COOK

ABSTRACT. We describe a new implementation of Edmonds' blossom algorithm for computing minimum weight perfect matchings. Combining this with specialized pricing techniques, we obtain a solution method for large-scale graphs. We report on the solution of a set of geometric test problems (complete graphs, described as points in the plane), the largest having 101,230 nodes.

1. Introduction

Let $G = (V, E)$ be a graph with node set V and edge set E. A *matching* $M \subseteq E$ is a subset of the edges such that each node in V is met by at most one edge in M. It is a *perfect matching* if each node is met by exactly one edge. Given a real weight w_e for each $e \in E$, the *minimum weight perfect matching problem* is to find a perfect matching M with minimum weight $w(M) = \sum(w_e : e \in M)$. Applications of minimum weight perfect matchings are discussed in Ball, Bodin, and Dial [**5**], Frederickson, Hocht, and Kim [**15**], Iri and Taguchi [**23**], and Reingold and Tarjan [**31**].

One of the cornerstones of combinatorial optimization is Edmonds' [**14**] polynomial-time *blossom* algorithm for min-weight perfect matching. This algorithm has been studied both from the viewpoint of obtaining good bounds on its asymptotic running time and from the viewpoint of developing fast computer implementations. Increasingly better time bounds have been obtained by Ball and Derigs [**6**], Gabow [**16**], Gabow [**17**], Gabow, Galil, and Spencer [**18**], Gabow and Tarjan [**19**], Galil, Micali, and Gabow [**20**], and Lawler [**25**]. The current best is $O(|V|(|E|+|V|log|E|))$ by Gabow [**17**]. (A nice overview of some of these results is given in Ball and Derigs [**6**].) Sophisticated computer implementations are described in Burkard and Derigs [**7**], Cunningham and Marsh [**8**], Derigs [**9**],

1991 *Mathematics Subject Classification*. Primary 90C27, 05C70; Secondary 90C06, 05C04.

The first author was supported in part by the National Science Foundation under contract CCR-9007602.

[**10**], [**11**], Derigs and Metz [**12**], Derigs and Metz [**13**], and Lessard, Rousseau, and Minoux [**26**].

An important class of matching problems are those described by specifying n points in the plane to match so as to minimize the distance between the pairs. In terms of graphs, this means we have the complete graph on n nodes with the weight of each edge being the distance between its end points (under some metric). Applications of this class of problems include the routing of mechanical plotters as described in [**23**] and [**31**]. Fast heuristics have been proposed for finding good solutions to these geometric problems (see Assano, Edahiro, Imai, Iri, and Murota [**3**] and Avis [**4**]) and fast exact methods have been developed for special configurations of points (see Marcotte and Suri [**27**]). Moreover, Vaidya [**34**] has shown that the blossom algorithm can be implemented with an $O(|V|^{2.5}(\log|V|)^4)$ time bound for this class.

In this paper we describe a new computer implementation of the blossom algorithm for general graphs, together with specialized methods for the solution of geometric matching problems. Like the code of Derigs and Metz [**12**], we use a fractional matching "jump start" and work with a sparse subgraph of the original graph G, using linear programming pricing to handle the remaining edges. Our pricing technique makes use of the least-common ancestor algorithm of Aho, Hopcroft, and Ullman [**2**], and we employ a sweep method to reduce the number of edges that need to be priced explicitly when solving geometric problems.

The paper is organized as follows. In Section 2 we give a brief outline of the blossom algorithm, and in Section 3 we describe our implementation. The overall strategy for solving large problems is described in Sections 4, 5, and 6. Our computational tests are reported in Section 7, including the solution of a geometric problem containing 101,230 nodes.

2. Blossom algorithm

The blossom algorithm is a primal-dual method based on Edmonds' linear programming formulation of the min-weight perfect matching problem. To describe this, let $\mathcal{O}$ denote the set of all odd subsets of V containing at least 3 nodes (we refer to these sets as *blossoms*), 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 $\sum(x_e : e \in H)$. Edmonds [**14**] used the blossom algorithm to show that the convex hull of the incidence vectors of the perfect matchings in G is precisely the solution set of the linear system

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

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

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

So the minimum weight of a perfect matching is equal to min $\{wx : x$ satisfies (1), (2), and (3) $\}$. The dual to this linear programming problem is

$$\max \sum (y_v : v \in V) + \sum (Y_S : S \in \mathcal{O}) \tag{4}$$

subject to

$$\begin{aligned} y_u + y_v + \sum (Y_S : S \in \mathcal{O}, (u,v) \in \delta(S)) &\leq w_{(u,v)} \text{ for all } (u,v) \in E && (5)\\ Y_S &\geq 0 \text{ for all } S \in \mathcal{O}. && (6) \end{aligned}$$

Given a solution $(\bar{y}, \bar{Y})$ to this dual problem, the *reduced cost* of an edge $e = (u, v)$ is

$$w_{(u,v)} - \bar{y}_u - \bar{y}_v - \sum (\bar{Y}_S : S \in \mathcal{O}, (u,v) \in \delta(S)),$$

that is, the slack in the corresponding constraint (5). An edge is called *tight*, with respect to $(\bar{y}, \bar{Y})$, if its reduced cost is 0. Similarly, a blossom $S \in \mathcal{O}$ 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 blossoms $S \in \mathcal{O}$, if $\bar{Y}_S > 0$ then S is full. So we can prove that a given perfect matching is optimal by providing a dual solution such that these conditions are satisfied. The blossom algorithm produces just such a proof. At each step, the algorithm has a partial matching and a dual solution that satisfy the complementary slackness conditions. The matching is grown via augmenting paths until we reach a perfect matching (which we know is optimal). A good description of the blossom algorithm can be found in Pulleyblank [**29**]. We will not go into all of the details of the algorithm, but we do need to describe some of its structure to present our implementation below.

We start with a simplified version of the blossom algorithm, solving the *fractional matching problem* $\min\{wx : x$ satisfies (1) and (2)$\}$. This will serve two purposes. First, it builds a framework that will be useful in our discussion of the blossom algorithm below. Secondly, fractional matchings are used at several points in the scheme for solving large matching problems described in Section 4. So let us suppose there exists a solution to (1) and (2). The fractional matching algorithm finds a $0, 1/2, 1$ valued solution $\bar{x}$ and a dual solution $\bar{y}$ such that $\bar{x}_{(u,v)} > 0$ only if (u, v) is tight, that is, $w_{(u,v)} = \bar{y}_u + \bar{y}_v$ (there are no variables Y_S). So, by complementary slackness, $\bar{x}$ will be an optimal solution to the fractional matching problem. Moreover, $\bar{x}$ will have the property that the edges e with $\bar{x}_e = 1/2$ form node disjoint odd circuits. (Notice that the fractional matching problem can also be solved by reducing it to a bipartite matching problem.)

Initially, let $\bar{x}_e = 0$ for all edges e and for each node v let $\bar{y}_v = (1/2)\min\{w_e : e \in \delta(v)\}$. At each iteration, 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, replacing $\bar{x}_e$ by $1 - \bar{x}_e$ for each edge e in the path. Such an augmentation increases the size of $\bar{x}$ by 1 (that is, $\sum(\bar{x}_e : e \in E)$ increases by 1). It is also possible to increase $\bar{x}$ if we reach a node v that meets an edge e with $\bar{x}_e = 1/2$. In this case, by the way we build $\bar{x}$, we know that v lies on an odd circuit C all of whose edges have value 1/2. Thus, the size of $\bar{x}$ can be increased by 1/2 by augmenting along the path in T from v to r and, starting with node v, alternately flipping the 1/2-valued edges in C to 0 and 1 as we traverse the circuit back to v (see Figure 1).

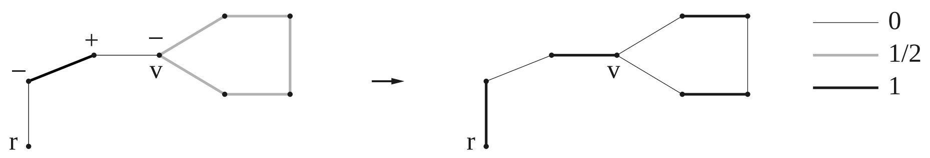

FIGURE 1. A 1/2 - augmentation

A third, and final, way to increase the size of $\bar{x}$ involves the formation of 1/2-valued odd circuits. Suppose there exists no node in T meeting a 1/2-valued edge and suppose there does exist a tight edge (u, v) joining two "+" nodes. Let P_1 denote the path in T from u to r and let P_2 denote the path from v to r. The union of P_1, P_2 and (u, v) forms an odd circuit C and a path, P, from w (the first node in P_2 we meet as we traverse P_1) to r. Again, $\bar{x}$ can be increased by 1/2 by augmenting along P (so w is now unmatched) and setting $\bar{x} = 1/2$ for all edges in the circuit C (see Figure 2). Since we have assumed that no node of T meets a 1/2-valued edge, we know that C is node disjoint from any other 1/2-valued circuits that may exist. Now suppose we cannot grow T any further and we have reached neither an unmatched node, nor a node meeting a 1/2-valued edge, nor a tight edge joining two "+" nodes. At this point we must alter the dual solution to create a new tight edge that allows us either to grow the tree or augment the fractional matching. The constraints on the dual change are that all edges in T, as well as all other edges having $\bar{x}_e > 0$, remain tight and that $\bar{y}$ remains a dual solution (that is, $\bar{y}_u + \bar{y}_v \le w_{(u,v)}$ for all edges (u, v)). These

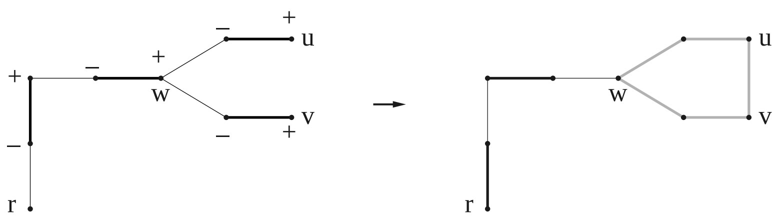

FIGURE 2. Creating a 1/2 - valued circuit

conditions can be met by adding some $\epsilon > 0$ to the $\bar{y}$-value of all "+" nodes and subtracting ϵ from all "−" nodes, making ϵ as large as possible subject to $\bar{y}$ remaining a dual solution. What stops us from making ϵ arbitrarily large is that some edge, joining a "+" node and a node not yet in T or joining two "+" nodes, becomes tight. (If no edge becomes tight, then there is no solution to (1) and (2).) In the first case we can grow T and in the second case we can create a 1/2-valued circuit. It is useful to remark that if all edge costs are integral, then the $\bar{y}$ values remain 1/2 integral throughout the procedure. This follows easily from the form of the dual changes. Indeed, suppose that at a general stage all $\bar{y}$ values (and, hence, all reduced costs) are 1/2 integral and that we are about to make an ϵ dual change. If ϵ is bounded by the reduced cost on an edge joining a "+" node and a node not yet in the tree, then it is certainly 1/2 integral. The only problem seems to be when ϵ is bounded by 1/2 of the reduced cost of an edge joining two "+" nodes. But notice that since all edges in the alternating tree are tight, we must have $\bar{y}_u \equiv \bar{y}_v$ (modulo 1) for any two nodes in the tree. It follows that the reduced cost of the edge joining the two "+" nodes is integral, and hence ϵ is again 1/2 integral.

So that is the algorithm: we repeatedly grow T, alter the dual solution, and grow T some more, until we can perform an augmentation. The full blossom algorithm works in the same way, but with two extra operations to handle the blossom variables Y_S. The key idea is that when we set $\bar{Y}_S > 0$ for some set S, we then treat S as a single node. The intuition is that the complementary slackness condition $\bar{x}(\delta(S)) = 1$ is the same as the constraint $\bar{x}(\delta(v)) = 1$ for individual nodes v. Thus, one of the new operations is to *shrink* a subset of nodes S into a *pseudonode* by contracting all edges having both ends in S. This operation will arise when we have a tight edge joining two "+" nodes (or pseudonodes) in the current alternating tree. Where in the fractional algorithm we performed a 1/2-augmentation around the odd circuit C that is formed, we instead shrink the nodes of C into a pseudonode. The converse operation is to delete a pseudonode S, bringing a subset of the nodes (and pseudonodes) we previously shrunk back into the current alternating tree T. This occurs when the $\bar{Y}$-value corresponding to S goes to 0 after a dual change. In this situation, S is a "−" node and thus is met by exactly two edges in T: a matched edge e and an unmatched edge f. We need to match the nodes of S, other than the node $w \in S$ that is already matched by e, with edges that were previously contracted. This step is straightforward, since S is spanned by an odd circuit C of tight edges (and any dual changes made after S is shrunk will not effect the reduced costs of the edges that were contracted in the shrink operation). To maintain the alternating tree T, we replace S by the even length path in C from the node $u \in S$ that meets the unmatched edge f to the node w that meets the matched edge e, labeling the nodes "$-$" and "$+$" as we move from u to w. (So the internal nodes and edges of the odd path from u to w will not be part of the tree.) This entire operation will be referred to as *expanding* the pseudonode S (see Figure 3).

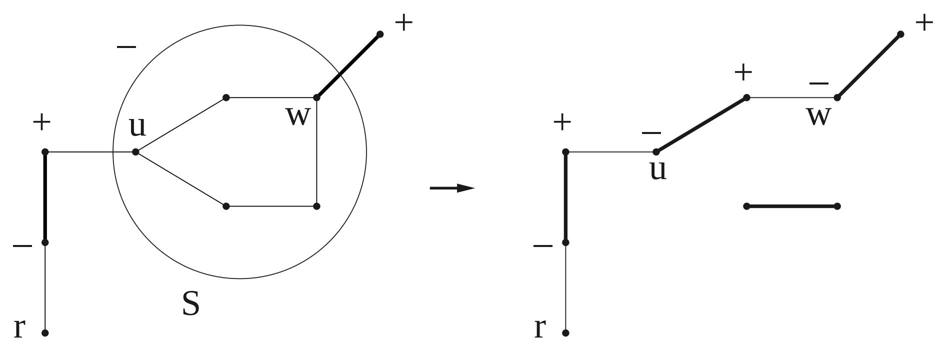

FIGURE 3. Expanding a pseudonode

At a general stage of the blossom algorithm we have carried out some number of shrinkings and expansions, resulting in a current *surface graph*. The procedure is such that a perfect matching amongst the tight edges in this graph (tight is still defined in terms of the original graph G) translates directly to an optimum matching in G. Initially, the surface graph is G itself, $\bar{x}$ and $\bar{Y}$ are 0, and $\bar{y}$ is set as in the start of the fractional algorithm. At each step we choose an unmatched node r in the surface graph and grow an alternating tree T rooted at r. If we reach an unmatched node in T, then we augment the matching in the surface graph and continue from another unmatched node r. If there exists a tight edge joining two "$+$" nodes, then we shrink the nodes of the odd circuit that is formed and continue growing the tree T. If neither of these operations are possible and we can no longer grow T, we perform a dual change, again adding as large as possible $\epsilon > 0$ to the $\bar{y}$-value of the "$+$" nodes (or the $\bar{Y}$-value of the "$+$" pseudonodes) and subtracting ϵ from the "$-$" nodes. If a new tight edge appears we can continue the algorithm as in the fractional case. But it may happen that ϵ is bounded by the $\bar{Y}$-value of a "$-$" pseudonode S (since these variables must remain nonnegative). In this case, we expand S and continue the algorithm. (If ϵ can be made arbitrarily large, then G has no perfect matching.)

When the algorithm terminates, we have a perfect matching of the final surface graph. To find the implied optimum matching of G, we just need to clean up the pseudonodes, matching their spanning circuits appropriately. Since all edges in this matching are tight and all pseudonodes correspond to full blossoms, we know that the matching is optimal. As in the fractional matching case, it again follows from the form of the dual changes that if all edge costs are integral, then $\bar{y}$ and $\bar{Y}$ remain 1/2 integral throughout the course of the algorithm.

3. Implementation

Our implementation of the blossom algorithm is straightforward. In particular, we do not make use of priority queues for handling the dual changes and determining what action should be taken while growing the alternating trees. It would be interesting to see a direct comparison of a sophisticated implementation of a priority queue approach with the implementation presented here. In our description below, we will assume that the reader is familiar with a the C (see Kernighan and Ritchie [**24**]) and with basic data structures (see Aho, Hopcroft, and Ullman [**1**] and Tarjan [**33**]).

The input to our implementation is a list of edges, giving their endpoints and costs. We assume that the costs are integral. Thus, by multiplying each of them by 2 at the start of the algorithm we may assume that the dual variables take on only integer values (using the argument that shows they are 1/2 integral on general integer input).

The graph, G, is stored internally as a list of edges and a list nodes, each node having an adjacency list consisting of a linked list of pointers to the edges it meets. (Linked lists are used to allow us to add edges to G in the procedures for large problems and to rebuild adjacency lists after a shrink or expand operation.) The adjacency lists can be implemented with an *edgeptr* structure having two fields: a pointer to an edge and a pointer to the "next" edgeptr. Since we need two such edgeptrs for each edge (the edge appears in two lists), we may as well include them directly in our edge structure. With this setup, however, the following trick can be made. We include the edgeptrs as the first two fields in our edge structure. Because the edgeptrs are now part of the edge structure, the pointer to the edge can be directly determined from a pointer to the edgeptr. Thus, we save the space for storing the pointer to the edge, and the time to look up that pointer, at the expense of some computation to convert edgeptr pointers to edge pointers. We refer to this convention as using *ugly edgeptrs* and incorporate it in our implementations.

As before, we first give the details of the fractional matching algorithm, and afterwards the changes needed to handle blossoms. The implementation of the fractional matching algorithm is simple. The alternating tree, T, is grown using depth-first search from an unmatched node r. For each node v in T, we store a pointer to the first edge in the path from v to r. Thus, it is an easy matter to trace an augmenting path from v back to r. To make 1/2-valued circuits easy to traverse, we orient the circuits and use a field in the edge structure to allow each 1/2-valued edge to point to the next edge in its circuit. We also keep a pointer from each node to the matching edge that meets it, if one exists, to allow us to skip over "−" nodes in the grow steps. Dual changes are made by traversing T in depth-first order, computing the bound on ϵ obtained from the edges meeting each "+" node and keeping a linked list of the nodes meeting edges that give the minimum bound. After making the change (by again traversing the tree) we need to grow T from each node on the linked list. And that is the implementation, except for minor details that we do not want to discuss.

Naturally, the full blossom algorithm requires some additional data structures. Our implementation follows the basic framework of the Blossom II code of Pulleyblank [**29**]. To begin with, we maintain more information about the alternating tree T, since in practice we need to traverse T more often than in the fractional algorithm. Besides the "parent edge" pointers we used above, we also keep parent, child, and sibling node pointers, that is, for each node we keep a pointer to its parent p (in the rooted tree T), a pointer to its first child, under some ordering of its children, and a pointer to the next child of p, under some ordering of p's children. We use the same system to record the current set of blossoms that have been shrunk to pseudonodes, where "parent", "sibling", and "child" refer to the nesting of the blossoms. When a pseudonode is formed from an edge (u, v) joining two "+" nodes, we call (u, v) the *blossom forming* edge. Tracing the path from u back to r, we call the first node, w, that also lies on the path from v to r, the *blossom root* (see Figure 4). Given the blossom form-

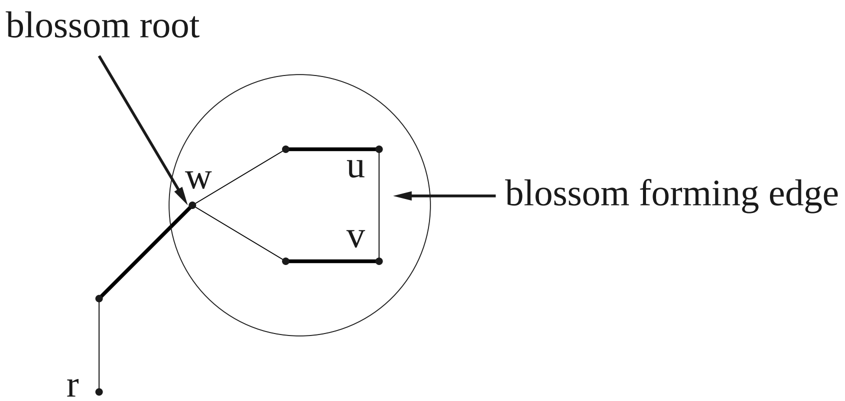

FIGURE 4. A blossom

ing edge and the blossom root, it is an easy matter to use the alternating tree information to traverse the spanning circuit of the blossom when needed during an expand operation. Thus, for each pseudonode v we let its "nest" child be the blossom root, w, of the blossom we shrunk to obtain v and we let w's "parent edge" pointer be the address of the corresponding blossom forming edge.

The surface graph is maintained by keeping a pointer from each original node to the surface node that contains it, and keeping pointers from each edge to its surface ends (if the edge appears in the surface graph). But this information is not sufficient to grow the alternating tree in a depth-first manner, since we must have access to adjacency lists for the surface nodes. This complication is avoided in some earlier codes by growing the tree (or multiple trees) by running through the edge list at each grow step, checking for tight edges joining a "+" node and an unlabeled node, or joining two "+" nodes. Instead we form explicit adjacency lists for the pseudonodes. Thus, when we shrink a set S we must gather the edgeptrs from the nodes in S, and when we expand S we must distribute them back to the individual nodes. Note that we do not make new edgeptrs, but rather reuse the existing ones. (Indeed, with ugly edgeptrs it is not possible to create new edgeptrs as we go along.)

As we mentioned above, T is grown using depth-first search from an unmatched node r. Whenever we reach a tight edge that joins a "+" node and an unlabeled node, we immediately add the edge to T and continue to grow from the unlabeled end of the edge (checking if it is matched, etc.). But when we find a tight edge e joining two "+" nodes, we do not shrink the blossom that is formed. Instead, we add e to a "shrink list" of potential pseudonodes. Only when we can no longer grow the tree do we go back and attempt the shrink operations, one at a time. After each shrink, we grow the tree from the newly created pseudonode, perhaps adding new edges to the shrink list in the process. (Notice that some of the edges on the list may have been contracted in earlier shrinks.) When the list is empty, we proceed with a dual change. The delayed shrinking is an attempt to cut down on the total number of blossoms used in the final dual solution, which is an important consideration in the procedure for large problems discussed in the next section.

A dual change is again carried out by first traversing T to compute the bound on ϵ, keeping a linked list of those nodes that meet edges giving the minimum bound, and then traversing T a second time to make the actual change. If ϵ is bounded by the $\bar{Y}$-value on some "−" pseudonodes, then we expand all such nodes and iterate the dual change procedure. Otherwise, we attempt to grow the tree from each node on the linked list we created during the first traversal of T, again delaying any shrinking until the tree can be grown no further. This grow-shrink-dual change loop continues until we find an augmenting path to match the node r. This completes the description of our blossom implementation.

Given the relative simplicity of the fractional algorithm, we should expect it to run much faster in practice than the full blossom algorithm. This is indeed the case, as is indicated in Table 1. The test problems reported in the table arise as sparse subgraphs (either the union of a set of Hamilton circuits, or one Hamilton circuit plus the 5 nearest neighbors to each node) of a set of geometric problems. (The source of the problems is given in Section 7.) The running times are reported in CPU seconds on a Dec 5000/200 workstation (which is based on a 25 MHz MIPS 3000 micropressor). The Dec 5000/200 performs the DIMACS benchmarks [**28**] wmatch test 1 in 1.9 CPU seconds and wmatch test 2 in 18.0 CPU seconds. The codes were written in the C programming language and compiled using cc with the -O optimizer provided with the Ultrix 4.2 operating system. To compare the performance of the blossom code, we have also included the running times of the SAP code of Derigs [**11**] in Table 1. The SAP code was shown in Derigs [**9**], [**10**], [**11**] to be superior to other codes in the literature for sparse problems. (For dense problems, Derigs [**10**] reports improved running times with a two-stage approach such as we describe in the next section. Also, it should be noted that Derigs and Metz [**12**], [**13**] have improved the SAP code by adding a "jump start", as we describe below.) This code is written in Fortran, and was compiled using f77 with the -O3 full optimizer, provided with Ultrix 4.2. The reported times do not include the time for reading the input file.

Nodes	*Edges*	*Fractional (CPU Seconds)*	*Blossom (CPU Seconds)*	*SAP (CPU Seconds)*
1002	3241	0.07	0.27	10.74
2392	7236	0.15	0.80	147.28
3038	5915	0.23	4.53	524.04
4040	9532	0.28	9.48	417.17
5934	10459	0.40	26.68	2072.07
7396	22148	1.50	9.90	3133.66
8064	26323	0.60	15.88	2934.30
11848	35453	0.82	78.15	10595.46
18092	56023	1.22	87.53	72246.99
20726	66872	1.27	878.20	106039.00
80864	255173	5.85	3733.87	****
101230	315677	7.37	4182.68	****

TABLE 1. Fractional, Blossom, and SAP (Derigs) Times (Dec 5000)

The times reported in Table 1 indicate that it is feasible to consider using fractional matchings to "jump start" the blossom algorithm, as in Derigs and Metz [**13**]. The idea is to solve the fractional matching problem to get good initial primal and dual solutions to begin the blossom algorithm. The dual solution is straightforward, we just let $\bar{y}$ be the optimal fractional dual solution and set $\bar{Y}_S = 0$ for all odd sets S. Thus, tight edges in the fractional problem are also tight at the start of the blossom algorithm. So any 1-valued edge in the optimal fractional matching can be set to 1 in the initial primal solution. Moreover, from each odd circuit of 1/2's in the fractional solution we can also include edges that match all but one of the nodes in the circuit in the initial primal solution. The running times using the jump start are reported in Table 2 (the jump start code of Derigs and Metz [**13**] was not available for a direct comparison). The last column in the table reports the total number of alternating trees grown in the blossom phase (so all but twice this number of nodes were matched in the initial primal solution). Again, the CPU times do not include the time for reading the input file. In 8 of the 12 cases, we see an improvement in the running time over the straight blossom implementation reported in Table 1.

Nodes	*Edges*	*CPU Seconds*	*Number of Augmentations*
1002	3241	0.20	33
2392	7236	0.70	84
3038	5915	4.08	56
4040	9532	4.97	64
5934	10459	22.05	86
7396	22148	20.65	70
8064	26323	17.00	215
11848	35453	66.37	387
18092	56023	69.20	646
20726	66872	1023.08	916
80864	255173	3491.32	3209
101230	315677	3475.58	8348

TABLE 2. Matching with fractional jump start (Dec 5000)

4. Strategy

When solving large matching problems, it quickly becomes impractical to work with the edge set of any graph, G, that is not extremely sparse. Simple storage requirements are one obvious reason, but equally important is the fact that the running time of the blossom algorithm is greatly influenced by the number of edges in G. For this reason, we follow a standard procedure in combinatorial optimization and use a two-stage approach to solve large problems. We first choose a sparse subgraph, G', of G and find the optimal matching contained in it. Next, we compute the reduced costs of the edges in G that have not been included in G' (we call this the *pricing* step). If any edge e has negative reduced cost, then we must *repair* the matching by adding e to G' and recomputing the primal and dual solutions. After repairing all negative edges, we repeat the pricing step, and again make repairs, continuing these price-repair rounds until we have no edges with negative reduced cost. At this point, the complementary slackness conditions hold for the entire graph G (setting $\bar{x}_e = 0$ for all edges not in G') and thus we have the overall optimal matching. This price-repair approach was used to solve matching problems by Grötschel and Holland [**21**] in a cutting-plane algorithm, and by Derigs and Metz [**13**] in a procedure based on the blossom algorithm. In the next two sections we discuss the price and repair phases in some detail. The remainder of this section is devoted to the choice of the sparse graph G'.

The criteria for selecting a sparse subgraph are that it should be easy to generate and it should have the property that, given an optimal dual solution, not too many of the non-included edges have negative reduced cost. An obvious choice is the *k-nearest neighbor* graph of G (for some positive integer k), where for each node v we include in the edge set of G' the k edges of minimum weight meeting v. This initial graph is used by Grötschel and Holland [**21**] (with $k = 5, 10, 15$) and Derigs and Metz [**13**] (with $k = 6, 8$), and works well, especially on problems where the distances are evenly distributed (that is, there is no clustering of the nodes) such as (uniform) randomly generated problems. A potentially better subgraph, however, can be constructed by making use of the fact that fractional matching problems can be solved very quickly in practice.

The idea is to first solve the fractional matching problem over the k'-nearest neighbor graph (for some k') and then build a new edge set consisting of the k edges of minimum reduced cost meeting each node. In other words, after finding a fractional dual solution $\bar{y}$, we select the k-nearest neighbors in G relative to the edge weights $w_{(u,v)} - \bar{y}_u - \bar{y}_v$. We refer to this sparse graph as the *fractional k-nearest neighbor* graph. In both cases, we take a greedy matching from G and add it to the sparse edge set, to avoid the complications that arise when G' does not contain a perfect matching.

The choices of sparse graphs are compared in Table 3, with $k = 5$, 10, and 15, on a set of five geometric test problems (described in Section 7). In the fractional matching graphs, the initial fractional matching was found over the 5-nearest neighbor graph (so $k' = 5$).

Problem	*Subgraph*	*Price-repair Rounds*	*Edges Added*	*Price-repair Time*	*Total Time*
7396	Nearest 5	13	651	646.2 (seconds)	707.0 (seconds)
	Nearest 10	5	53	250.2	358.3
	Nearest 15	4	15	244.8	401.5
	Fractional 5	11	613	592.1	725.2
	Fractional 10	5	35	250.1	439.5
	Fractional 15	2	9	107.2	360.9
8064	Nearest 5	13	687	495.2	524.2
	Nearest 10	6	84	170.8	200.8
	Nearest 15	5	23	92.2	158.3
	Fractional 5	9	332	239.1	281.0
	Fractional 10	5	35	102.7	128.0
	Fractional 15	2	4	66.7	112.6
11848	Nearest 5	10	1120	1237.5	1349.7
	Nearest 10	5	77	244.8	537.5
	Nearest 15	3	14	198.3	562.2
	Fractional 5	6	654	580.0	674.6
	Fractional 10	3	32	116.4	444.3
	Fractional 15	2	13	53.4	521.2
18092	Nearest 5	15	1408	1173.0	1263.8
	Nearest 10	7	73	851.2	1145.5
	Nearest 15	3	6	318.0	774.2
	Fractional 5	11	628	864.2	1062.0
	Fractional 10	2	6	260.2	585.0
	Fractional 15	1	0	56.2	472.2
20726	Nearest 5	10	3391	22431.9	23419.2
	Nearest 10	5	289	3727.55	6021.6
	Nearest 15	4	83	2427.7	7830.6
	Fractional 5	9	2029	14382.7	15396.5
	Fractional 10	4	182	3908.1	6697.4
	Fractional 15	3	49	2031.6	7504.9

TABLE 3. Choice of sparse subgraph (CPU Seconds, Dec 5000)

The tests indicate that the fractional k-nearest graph has an advantage over the corresponding k-nearest graph. Although there is not a clear winner between the fractional 10-nearest and the fractional 15-nearest, we will use the fractional 10-nearest in our procedure for geometric problems since this will allows us to keep our memory usage on very large problems at an acceptable level.

5. Making repairs

We begin our discussion with the easy case of fractional matching problems. Suppose we have optimal fractional primal and dual solutions, $\bar{x}$ and $\bar{y}$, for a graph G, and we need to introduce an edge (u,v) that is not yet contained in E. Let $t = w_{(u,v)} - \bar{y}_u - \bar{y}_v$ be the reduced cost of the new edge. If $t \geq 0$, we have no work to do. Otherwise, we alter the dual solution by replacing $\bar{y}_u$ by $\bar{y}_u - t$. So, to maintain complementary slackness, we must also alter the primal solution. If u meets an edge e having $\bar{x}_e = 1$, then we simply set $\bar{x}_e = 0$. Otherwise, u lies in a 1/2-valued circuit C, and we replace the 1/2 values on the edges of C by a matching that leaves only u unmatched. In either case, all edges (including (u,v)) will have nonnegative reduced cost and all edges e with $\bar{x}_e > 0$ will be tight, so we can use the fractional algorithm (growing an alternating tree from u) to compute new optimal primal and dual solutions.

Repairing (integral) matchings is more difficult. Again, suppose we have a graph G and optimal primal and dual solutions, $\bar{x}$ and $(\bar{y}, \bar{Y})$ (with surface graph $\bar{G}$) and we need to add the new edge (u,v). If either u or v is itself a node in $\bar{G}$ (say node u), then we can follow the procedure used above: decrease the value of $\bar{y}_u$, set $\bar{x}_e = 0$ for the matching edge meeting u, and apply the blossom algorithm to compute a new optimal pair of solutions. The difficulty arises when both u and v are contained in pseudonodes of $\bar{G}$. In this case, we cannot simply alter the value of $\bar{y}_u$ (or $\bar{y}_v$), since this will disturb the structure of $\bar{G}$, for example, the minimal pseudonode containing u will no longer be spanned by a tight odd circuit. To handle this, we can make dual changes to bring u (or v) to the surface, expanding all pseudonodes containing u. Ball and Derigs [**6**] gave an elegant procedure for this task, simplifying earlier work of Weber [**35**]. (An alternative "primal" approach to this problem is described in Cunningham and Marsh [**8**].) Their idea is to add a new node p and an edge e joining p and u (giving e reduced cost 0), and grow an alternating tree from node p. Of course, we will find no augmenting path, but during the search we will perform dual changes that allow us to expand pseudonodes. We stop the process as soon as u reaches the surface (that is, u is no longer contained in any pseudonodes). Then, deleting p and e, we proceed as above.

This repair strategy is very simple to describe and implement, but it does cause some problems in the overall price-repair strategy. The trouble is that it introduces many changes in the dual solution. Besides being time consuming, these changes affect the number of price-repair iterations we need to make. In this regard, we would like to keep the dual solution as stable as possible, to avoid introducing new negative reduced cost edges. For this reason, we employ a slightly more complicated repair procedure. The main idea is to allow matched edges to have negative reduced costs. This can be justified by either adding the redundant constraints

$$x_e \leq 1 \quad \text{for all } e \in E \tag{7}$$

to the linear system (1), (2), (3) and considering the new primal-dual pair of linear programming problems, or simply noting that the reduced cost of a matched edge e can always be raised to 0 by increasing the $\bar{y}$-value of one of its ends. (Note that this modification cannot be done during the course of making repairs, since this would disturb the structure of the surface graph, as we mentioned above.)

The added freedom of having negative reduced costs can be an advantage in making repairs. For example, if the maximal pseudonodes containing u and v are joined by an alternating path of tight edges in $\bar{G}$ (starting and ending with matched edges) then we can set $\bar{x}_e = 1 - \bar{x}_e$ for each edge in the path and insert the edge (u, v) in the graph as a matched edge with negative reduced cost (see Figure 5). The price of this freedom is the additional overhead needed

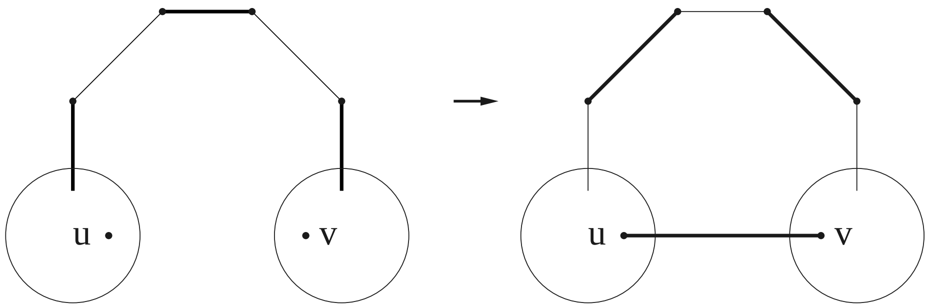

FIGURE 5. Inserting an edge

to maintain the negative edges. This manifests itself in two ways. First, we must modify the grow step of the blossom algorithm to only bring matched edges into the alternating tree when they have 0 reduced cost, since augmenting paths through the tree will flip some matched edges to unmatched edges, and these cannot be negative. Secondly, when computing the dual change bound ϵ we must take into consideration the reduced costs of the matched edges joining either two "−" nodes or a "−" node and an unlabeled node, since we do not want the reduced cost of such edges to become positive after the dual change. These complications slow down the algorithm somewhat, so in our implementation we compute the initial optimal matching and dual solution without allowing negative reduced costs, and switch to the negative version only when we begin the repair phase.

To make use of the negative reduced costs, we need a different mechanism for carrying out the repairs. To start off, we add a new node p and new edges (p, u) and (p, v), giving (p, u) reduced cost 0 and setting the reduced cost of (p, v) to

0 if $\bar{y}_u \equiv \bar{y}_v$ (modulo 2) and otherwise setting the reduced cost of (p, v) to 1. This choice of reduced costs ensures that the dual variables will continue to take on only integer values in the modified graph (recall the 1/2-integral argument and the fact that we have doubled the integral edge costs at the start of the algorithm). We grow an alternating tree from p, with the following two stopping rules. First, we stop if either u or v becomes a surface node, as in the Ball and Derigs procedure. Second, we stop whenever we find a blossom forming edge where p is the blossom root. This latter case corresponds to the situation depicted in Figure 5 (namely, we have a tight alternating path from u to v, starting and ending in matched edges) and we proceed as described above.

Nodes	*Negative (Pricing Time)*	*Negative (Rounds)*	*BD (Pricing Time)*	*BD (Rounds)*
7396	249.75 (seconds)	5	510.87 (seconds)	11
8064	102.55	5	354.40	9
11848	116.58	3	1437.33	7
18092	260.17	2	577.92	4
20726	3910.67	4	26541.10	8

TABLE 4. Price-repair Mechanism (Dec 5000)

We tested both the Ball and Derigs method and the negative reduced cost method on a series of geometric problems. The results are reported in Table 4, and show the advantage of negative reduced costs, both in total running time and in the number of price-repair rounds required.

6. Pricing

At the start of the pricing step, we have optimal primal and dual solutions, $\bar{x}$ and $(\bar{y}, \bar{Y})$, for a sparse subgraph of our original graph G. Let $\mathcal{B}$ denote the set of blossoms $S \in \mathcal{O}$ with $\bar{Y}_S > 0$, and let B be the rooted tree representing the nested structure of $\mathcal{B}$. The nodes of B consist of a root node r_V (representing the entire node set V), the original nodes V, and the pseudonodes $\mathcal{B}$. The children of each node S are the maximal members of $V \cup \mathcal{B}$ that are properly contained in S. (So each original node is a leaf of the tree, the children of each pseudonode are the nodes (and pseudonodes) that were shrunk to form it, and the children of r_V are the nodes (and pseudonodes) of the surface graph.) For each original node $v \in V$ we compute

$$sum(v) \equiv \bar{y}_v + \sum\{\bar{Y}_S : S \in \mathcal{B}, v \in S\} \tag{8}$$

by traversing the path in B from v to the root. For any pair of nodes u, v let $lca(u, v)$ denote the least common ancestor of u and v in the tree B. Then, the reduced cost of an edge (u, v) is

$$w_{(u,v)} - sum(u) - sum(v) + 2 \cdot sum(lca(u, v)). \tag{9}$$

So the work in computing the reduced costs can be reduced to least common ancestor calculations. We carry these out with an off-line version of Aho, Hopcroft, and Ullman's [**2**] algorithm, using path compression to implement the set union operations in near-linear time (Tarjan [32]). (We chose this method, over the linear-time least common ancestor algorithm of Harel and Tarjan [22], for its low overhead and programming ease.)

Although the least common ancestor algorithm is quite efficient, for large problems we need to take care not to overwhelm the method by simply pricing every edge on each round. A first step in this direction is to take advantage of the fact that we know exactly what occurs between the pricing rounds, namely a series of repair operations. Thus, if the reduced cost of an edge goes down between rounds, then we know it must go down during the dual changes in one of the operations. For this to happen, one of the ends of the edge must be contained in a "+" node in the operation's final alternating tree. (Newly formed pseudonodes are always labeled "+", so a "−" pseudonode does not hide any nodes labeled "+" duing the operation, and only "−" pseudonodes are expanded, so any nodes that leave the tree were not labeled "+" earlier in the operation.) So if we start with all nodes unmarked, and, after each repair operation, mark every node contained in a "+", then during the next pricing phase we need only price those edges that meet at least one marked node.

In the special case of geometric problems, the idea of pricing only when necessary can be carried much further. Suppose we have coordinates (v_x, v_y) for each node $v \in V$ and that the weight of an edge (u, v) is the distance from (u_x, u_y) to (v_x, v_y) under some norm $\|\cdot\|$. (Higher dimensional problems can be handled in the same manner.) Suppose further that $\|\cdot\|$ is such that the weight of edge (u, v) is at least $|u_x - v_x|$. (This holds for any L_p norm, such as the Euclidean and max norms.) Then the reduced cost of (u, v) is bounded below by

$$|u_x - v_x| - sum(u) - sum(v). \tag{10}$$

To make use of this fact we maintain two sorted lists, the first, LIST1, sorted by increasing values of $v_x - sum(v)$, and the second, LIST2, sorted by decreasing values of $v_x + sum(v)$, where each node in V appears in exactly one of the lists. With this setup, we can readily determine which edges meeting a given node u make the expression (10) nonnegative (and so need not be explicitly priced). The process is to scan from the top of LIST1, letting v be the next node in the list, and consider pricing each edge (u, v) until we reach the point when $v_x - sum(v) \geq u_x + sum(u)$ (so (10) is nonnegative). We then do the corresponding scan of LIST2, stopping when $v_x + sum(v) \leq u_x - sum(u)$. When we consider pricing an edge (u, v), we compute $w_{(u,v)}$ and only add (u, v) to a "checkout list" if $w_{(u,v)} - sum(u) - sum(v) < 0$. (This is a stronger condition than (10) being negative.) Once the checkout list gets too long (in our implementation, $100,000$ edges), we price all edges on the list with the least common ancestor algorithm, making repairs on those having negative reduced cost, and then come back and continue the pricing phase. To maintain the sorted lists, we alternate the pricing rounds with forward passes and backward passes. A forward pass begins with all nodes on LIST1, and moves each node from LIST1 to LIST2 after it is processed. A backward pass begins with all nodes on LIST2, and moves nodes from LIST2 back to LIST1. At the end of a pass, we price the edges in the checkout list, repairing those that are negative. The entire process stops when we go through a pass without finding any negative edges. Node marking can be included in the scheme by simply not processing a node if it is unmarked (just move it from one list to the other). During the procedure we need only recompute the value of $sum(v)$ when we start the processing of v. This will mean that some of the $sum()$ values will be out of date in the middle of the procedure, but whenever $sum(v)$ is less than it should be we know that v has been marked by one of the repair operations, and thus will be processed at a later point.

7. Computational tests

Our test bed of geometric problems is listed in Table 5. The majority of the problems are described in Reinelt [**30**], and are available over the Internet (see [**30**]). In two cases, the original data sets contained an odd number of points. For these instances, we sorted the x, y coordinates and deleted the last point.

Nodes	*Norm*	*Source*
1002	Euclidean	TSPLIB [**30**]
2392	Euclidean	TSPLIB [**30**]
3038	Euclidean	TSPLIB [**30**]
4042	Max	Circuit board (University of Bonn)
5934	Euclidean	TSPLIB [**30**]
7396	Euclidean	TSPLIB [**30**]
8064	Max	Circuit board (University of Bonn)
11848	Euclidean	TSPLIB [**30**]
18092	Euclidean	Locations in USA
20726	Max	Map of Tokyo [**3**]
80864	Max	VLSI (Bellcore)
101230	Max	VLSI (University of Bonn)

TABLE 5. Geometric Test Problems

The solution times for the geometric problems are given in Table 6. The times are reported in seconds on a Dec 5000/200 workstation (as are the times in the remaining tables). The second column reports the time for computing the fractional 10-nearest graph (including the input time); the third column gives the time needed to compute the fractional matching "jump start" on the fractional 10-nearest graph; the fourth column gives the time needed to compute the integral matching; the fifth column reports the time in the price-repair phase; and the last column gives the total running time of the process (including the input time).

The extremely long pricing times for the two largest examples suggests that for problems of this size we probably should spend more time selecting our initial sparse graph, perhaps using an "integral nearest k-neighbor graph" (where we first compute the optimum matching on a sparse graph, then compute the k-nearest graph with respect to the reduced costs derived from the matching's dual solution).

Nodes	*Fractional Nearest*	*Jump start*	*Matching*	*Price-Repair*	*Total*
1002	4.08	0.12	0.17	0.47	6.2
2392	15.10	0.32	2.65	2.87	24.2
3038	22.34	0.57	14.23	3.20	45.5
4040	8.21	0.47	13.47	5.25	33.1
5934	39.82	0.72	81.88	118.73	261.2
7396	123.62	2.98	47.65	249.75	438.9
8064	25.25	1.17	20.10	102.25	164.0
11848	120.08	1.68	168.02	116.9	445.1
18092	58.83	2.43	230.85	260.23	584.7
20726	77.60	2.63	2678.13	3919.33	6710.2
80864	991.30	12.52	10207.27	136166.37	147535.1
101230	1069.90	15.42	7977.35	457839.98	467544.9

TABLE 6. Solution Times of Geometric Problems (Complete Graphs)

In our final two tables, we report on the solution times for two common classes of randomly generated data. The first type are (Euclidean) geometric problems with integral coordinates drawn uniformly from the 100,000 by 100,000 square. (The generator we use is described in McGeoch [**28**] as dcube.c.) The second class are graphs with $2|V|\log|V|$ edges drawn uniformly from the set of possible edges, with each edge having a random integral edge cost drawn uniformly from (0, 100000). (This generator is described in McGeoch [**28**] as random.c.) For most of the problem sizes in each class, we made a series of 10 independent runs, and report the average, the minimum, and maximum CPU times. (The times include the time spent to generate and read the input files.) The last column in each of the tables gives the average number of blossoms that had positive value in the final dual solution. These numbers are a good indicator of the complexity of a problem instance. From this viewpoint, one can see that the sparse random graphs do not really provide a good test of a matching code.

Nodes	*Trials*	*Average Time*	*Min Time*	*Max Time*	*Average Blossoms*
2^{10}	10	11.9	7.2	15.6	299.5
$2^{10.5}$	10	21.5	14.5	28.5	392.0
2^{11}	10	43.9	33.3	63.0	575.0
$2^{11.5}$	10	83.9	39.8	128.6	747.2
2^{12}	10	156.6	92.4	212.7	1079.7
$2^{12.5}$	10	324.7	150.2	522.2	1526.1
2^{13}	10	586.6	366.8	741.7	2132.8
$2^{13.5}$	10	1867.8	1185.4	3578.7	2999.3
2^{14}	10	2883.3	2200.4	3664.9	4170.5
2^{15}	10	9407.5	5555.7	17117.9	8279.1
2^{16}	10	49060.5	26969.1	70670.4	16183.8
2^{17}	10	254955.3	160094.4	398992.2	32001.0

TABLE 7. Random Geometric Problems with Euclidean Norm

Nodes	*Trials*	*Average Time*	*Min Time*	*Max Time*	*Average Blossoms*
2^{10}	10	9.8	7.9	23.8	23.9
$2^{10.5}$	10	12.8	10.8	23.2	12.5
2^{11}	10	23.6	20.9	29.8	7.4
$2^{11.5}$	10	50.7	36.5	91.5	41.7
2^{12}	10	81.5	61.8	129.1	43.3
$2^{12.5}$	10	155.3	103.7	424.9	72.6
2^{13}	10	194.8	169.5	300.2	25.8
$2^{13.5}$	10	667.9	298.2	2631.7	276.3
2^{14}	10	884.1	500.0	2795.8	170.6
2^{15}	10	1919.2	1459.7	4004.8	164.4
2^{16}	5	6101.8	5573.2	6970.3	105.6

TABLE 8. Random Costs (0,100000) with $2|V|\log|V|$ Edges

Acknowledgements

We would like to thank Joel Gannet, Olaf Holland, Hiroshi Imai, David Johnson, Bernhard Korte, Allen McIntosh, and Gerd Reinelt for kindly providing us with test data. We would also like to thank the two referees for a number of helpful comments. A portion of this work was carried out while the authors visited the Institute for Operations Research at the University of Bonn, West Germany, with support provided by SFB (303).

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, 1974.
2. ———, *On finding lowest common ancestors in trees*, SIAM Journal of Computing **5** (1976), 115–132.
3. T. Asano, M. Edahiro, H. Imai, M. Iri, and K. Murota, *Practical use of bucketing techniques in computational geometry*, Computational Geometry (G. T. Toussaint, ed.), North Holland, 1985, pp. 153–195.
4. D. Avis, *A survey of heuristics for the weighted matching problem*, Networks **13** (1983), 475–493.
5. M. O. Ball, L. D. Bodin, and R. Dial, *A matching based heuristic for scheduling mass transit crews and vehicles*, Transportation Science **17** (1983), 4–31.
6. M. O. Ball and U. Derigs, *An analysis of alternative strategies for implementing matching algorithms*, Networks **13** (1983), 517–549.
7. R. E. Burkard and U. Derigs, *Assignment and matching problems: Solution methods with FORTRAN-programs*, Springer Lecture Notes in Mathematical Systems 184, 1980.
8. W. H. Cunningham and A. B. Marsh, III, *A primal algorithm for optimum matching*, Mathematical Programming Study **8** (1978), 50–72.
9. U. Derigs, *A shortest augmenting path method for solving minimal perfect matching problems*, Networks **11** (1981), 379–390.
10. ———, *Solving large-scale matching problems efficiently: A new primal matching approach*, Networks **16** (1986), 1–16.
11. ———, *Solving non-bipartite matching problems via shortest path techniques*, Annals of Operations Research **13** (1988), 225–261.
12. U. Derigs and A. Metz, *On the use of optimal fractional matchings for solving the (integer) matching problem*, Computing **36** (1986), 263–270.
13. ———, *Solving (large scale) matching problems combinatorially*, Mathematical Programming **50** (1991), 113–122.
14. J. Edmonds, *Maximum matching and a polyhedron with 0,1 - vertices*, Journal of Research of the National Bureau of Standards **69B** (1965), 125–130.

15. G. N. Frederickson, M. S. Hocht, and C. E. Kim, *Approximation algorithms for some routing problems*, SIAM Journal of Computing **7** (1978), 178–193.
16. H. N. Gabow, *A scaling algorithm for weighted matching on general graphs*, Proceedings 26th Annual Symposium of the Foundations of Computer Science, 1985, pp. 90–100.
17. ______, *Data structures for weighted matching and nearest common ancestors with linking*, preprint, 1990.
18. H. N. Gabow, Z. Galil, and T. H. Spencer, *Efficient implementation of graph algorithms using contraction*, Journal of the ACM **36** (1989), 540–572.
19. H. N. Gabow and R. E. Tarjan, *Faster scaling algorithms for general graph matching problems*, Journal of the ACM, to appear.
20. Z. Galil, S. Micali, and H. N. Gabow, *An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs*, SIAM Journal of Computing **15** (1986), 120–130.
21. M. Grötschel and O. Holland, *Solving matching problems with linear programming*, Mathematical Programming **33** (1985), 243–259.
22. D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM Journal of Computing **13** (1984), 338–355.
23. M. Iri and A. Taguchi, *The determination of an XY-plotter and its computational complexity*, Proceedings of the 1980 Spring Conference of the Operations Research Society of Japan, (in Japanese), pp. 204–205.
24. B. W. Kernighan and D. M. Ritchie, *The C programming language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
25. E. L. Lawler, *Combinatorial optimization: Networks and matroids*, Holt, Rinehart, and Winston, New York, 1976.
26. R. Lessard, J.-M. Rousseau, and M. Minoux, *A new algorithm for general matching problems using network flow subproblems*, Networks **19** (1989), 459–479.
27. O. Marcotte and S. Suri, *Fast matching algorithms for points on a polygon*, to appear in SIAM Journal of Computing, 1991.
28. C. McGeoch, *The first dimacs international algorithm implementations challenge: the core experiments*, draft.
29. W. R. Pulleyblank, *Faces of matching polyhedra*, Ph.D. thesis, University of Waterloo, Waterloo, Ontario, 1973.
30. G. Reinelt, *TSPLIB - A traveling salesman problem library*, to appear in ORSA Journal on Computing.
31. E. M. Reingold and R. E. Tarjan, *On a greedy heuristic for complete matching*, SIAM Journal of Computing **10** (1981), 676–681.
32. R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, Journal of the ACM **22** (1975), 215–225.
33. ______, *Data structures and network algorithms*, SIAM, Philadelphia, PA, 1983.
34. P. M. Vaidya, *Geometry helps in matching*, SIAM Journal of Computing (1989), 1202–1255.
35. G. M. Weber, *Sensitivity analysis of optimal matchings*, Networks **11** (1981), 41–56.

School of Computer Science, Carnegie Mellon University
Current address: Math Sciences Research, AT&T Bell Laboratories
E-mail address: david@research.att.com

Combinatorics and Optimization Research Group, Bell Communications Research
E-mail address: bico@bellcore.com