A Scaling Algorithm for Weighted Matching on General Graphs

Harold N. Gabow *

Department of Computer Science
University of Colorado
Boulder, CO 80309

Abstract.

This paper presents an algorithm for maximum matching on general graphs with integral edge weights, running in time $O(n^{3/4}m \lg N)$, where n, m and N are the number of vertices, number of edges, and largest edge weight magnitude, respectively. The best previous bound is $O(n(m \lg \lg \lg_d n + n \lg n))$ where d is the density of the graph. The algorithm finds augmenting paths in batches by scaling the weights. The algorithm extends to degree-constrained subgraphs and hence to shortest paths on undirected graphs, the Chinese postman problem and finding a maximum cut of a planar graph. It speeds up Christofides' travelling salesman approximation algorithm from $O(n^3)$ to $O(n^{2.75} \lg n)$. A list splitting problem that arises in Edmonds' matching algorithm is solved in $O(m\alpha(m,n))$ time, where m is the number of operations on a universe of n elements; the list splitting algorithm does not use set merging. Applications are given to update problems for red-green matching, the cardinality Chinese postman problem and the maximum cardinality plane cut problem; also to the all-pairs shortest paths problem on undirected graphs with lengths plus or minus one.

1. Introduction.

Matching is one of the broadest classes of linear programs with guaranteed integral optima. Practical applications of general matching include mechanical plotting [RT], routing [FHK], VLSI [IA] and others. Edmonds' gave the first polynomial-time algorithm [E], which works by augmenting paths. The most efficient algorithms for matching (and its special case, network flow) find augmenting paths in batches. Hopcroft and Karp achieved this for cardinality matching on bipartite graphs [HK]. Even and Kariv [EvK] and Micali and Vazirani [MV] extended this to general graphs. Batching was applied to weighted bipartite matching in [G83b] using a scaling technique. This paper takes the fourth step by extending batching to weighted matching on general graphs, also by scaling.

The scaling method for network problems was introduced by Edmonds and Karp [EK]. Initial computational experience with their method seemed to show that scaling worked poorly in practice [L]. Recent experiments on their algorithm [BJ] and weighted bipartite matching [B] indicate it is efficient. Applications of scaling to computational geometry are given in [GBT].

This paper gives an algorithm for maximum matching on a general graph with integral edge weights. It runs in time $O(n^{3/4}m \lg N)$; throughout this paper, n, m and N are the number of vertices, number of edges, and largest edge weight magnitude, respectively. The time bound is the same as bipartite matching [G83b]. The best previous bound for general matching is $O(n(m \lg \lg \lg_d n + n \lg n))$ where d is the density of the graph [GGS].

The scaling algorithm for general matching is more involved than the bipartite case because of blossoms. The algorithm is based on a new view of blossoms as a "shell" structure. These shells also have a combinatorial interpretation (in Section 2.1) that is independent of the algorithm.

Blossoms also introduce a data structures problem because they must be "expanded". This problem occurs in Edmonds' algorithm and hence in all its implementations [e.g., GGS]. We reduce this problem to a list splitting problem and present an $O(m\alpha(m,n))$ algorithm. Here m is the number of operations on a universe of n elements and α is Tarjan's inverse to Ackermann's function [T83]. The splitting algorithm does not use set merging.

Section 2 presents the matching algorithm. It also shows how the algorithm finds approximate optimum matchings for real-valued weights. This speeds up Christofides' travelling salesman approximation algorithm from $O(n^3)$ to $O(n^{2.75} \lg n)$. The splitting algorithm is applied to solve update problems for red-green matching in $O(m\alpha(m,n))$ time.

Section 3 extends the matching algorithm to degree-constrained subgraphs, achieving similar efficiency. This gives improved running times for shortest paths on undirected graphs, the Chinese postman problem and finding a maximum cut of a planar graph. The splitting algorithm is applied to do updates in the cardinality Chinese postman problem and the maximum cardinality plane cut problem, in $O(m\alpha(m,n))$ time. It also finds all-pairs shortest paths on undirected graphs with lengths plus or minus one, in $O(nm\alpha(m,n))$ time. Section 4 presents the list splitting algorithm.

* This work was supported in part by NSF Grant #MCS-8302648.

We close this section by reviewing some ideas from matching; more thorough treatments are in [GMG, L, T83]. A *matching* is a set of edges, no two of which share a vertex. A *free* vertex is not on any matched edge; a *complete* matching has no free vertices. An *alternating path* (*cycle*) for a matching is a simple path (cycle) whose edges are alternately matched and unmatched. An *augmenting path* is an alternating path joining two free vertices.

Let each edge ij have a *weight* w_{ij}; in this paper weights are integral unless stated otherwise. The weight of a set of edges S, denoted $w(S)$, is the sum of the individual edge weights. A *maximum* (*minimum*) *complete matching* is a complete matching of maximum (minimum) weight; a *maximum weight matching* is a matching of maximum weight.

In a graph with a matching, a *blossom* is a subgraph B defined as follows. Let k be a positive integer. The vertices of B are partitioned into sets B_i, $0 \le i \le 2k$, where each B_i either consists of a single vertex or is itself a blossom. The edges of B are e_i, $0 \le i \le 2k$, where e_i joins B_i to B_{i+1} (e_{2k} joins B_{2k} to B_0) and e_i is matched precisely when i is odd. Note that a blossom is not an induced subgraph. The *base vertex* of B is defined as B_0 or its base, depending on whether B_0 is a vertex or a blossom.

The blossom structure of a graph is represented by a *blossom tree*. Its nodes are the graph G, the blossoms of G, and all vertices included in blossoms. The root is G, whose children are the maximal blossoms. The children of a blossom B are its constituents B_i, $0 \le i \le 2k$, as above. Any vertex is a leaf. Throughout this paper *node* refers to a node in a blossom tree and *vertex* refers to a vertex in the matched graph.

For a blossom B, n_B denotes the number of vertices contained in B, that is, the number of descendant leaves of B; m_B denotes the number of edges in the subgraph induced on the vertices of B (not the number of edges in the subgraph B, which is $2k+1$). Sometimes we use the same symbol B to denote a blossom and its vertex set (i.e., its descendant leaves).

For symmetry the cardinality of a set of vertices S is sometimes denoted n_S. Set inclusion is denoted as $\subseteq$ and proper inclusion as $\subset$. Logarithms are always base two and denoted *lg n*.

2. The matching algorithm.

The basic problem is taken to be maximum complete matching, on a graph that has a complete matching and whose weights are nonnegative even integers. Section 2.1 reviews Edmonds' algorithm and introduces shells. Sections 2.2-2.5 give a top-down description of the algorithm. Section 2.6 gives the final result and applications.

2.1. Edmonds' algorithm and shells.

We review Edmonds' algorithm [E] as applied to maximum complete matching. It is a primal-dual algorithm [D]. Each vertex i has a real-valued dual variable y_i ; each set B of an odd number of vertices, $n_B \ge 3$, has a nonnegative dual variable z_B (see Figure 1). The duals are *dominating* if for every edge ij,

$$y_i + y_j + \sum_{ij \in B} z_B \ge w_{ij} \ . \qquad (1)$$

Edge ij is *tight* if equality holds in (1). A set of dual variables is denoted y, z. For duals y, z and an arbitrary set of vertices S, define

$$(y,z)\,S \;=\; \sum_{i \in S} y_i \;+\; \sum_{B \subseteq S} \lfloor n_B / 2 \rfloor z_B \;+\; \sum_{S \subseteq B} \lfloor n_S / 2 \rfloor z_B \ .$$

This dual value function plays an important role in the algorithm, although we do not explicitly appeal to duality theory. Note that the set of all vertices V has $(y,z)V = \sum_{i \in V} y_i + \sum_B \lfloor n_B / 2 \rfloor z_B$, which is the dual objective function. For a vertex i, $(y,z)i = y_i$.

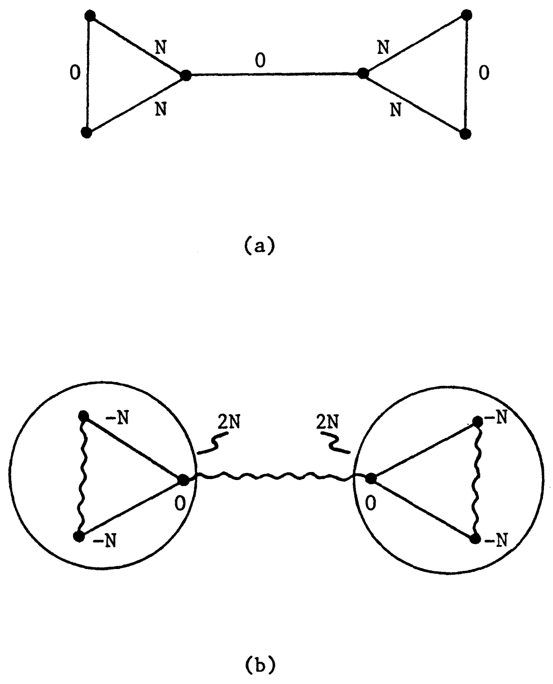

Figure 1.
(a) Graph with edge weights.
(b) Complete structured matching.

For any dominating dual variables y,z and any complete matching M,

$$(y,z)V \ge w(M). \qquad (2)$$

(This follows since M has at most $\lfloor n_B / 2 \rfloor$ edges contained in any odd set B.) If a complete matching consists of tight edges then it achieves equality in (2) and hence is a maximum complete matching.

Edmonds' algorithm works by maintaining a *structured matching*, which consists of a matching (not necessarily complete), a blossom tree (for the matching), and dual variables that are dominating and tight. In this definition the only odd sets with positive dual variables are blossoms (of the blossom tree); a *weighted blossom* is one that has a positive dual variable. Also in this definition, the duals y,z are said to be *tight* (with respect to the matching and its blossoms) if all matched edges and all edges of blossom subgraphs are tight. Clearly a complete structured matching is a maximum complete matching.

The algorithm can be viewed as having input consisting of a graph with a structured matching and output a complete structured matching. The algorithm starts with the given structured matching. It repeatedly does a "search" followed by an "augment step" until the matching is complete.

A *search* does "grow", "blossom", "expand" and "dual variable adjustment" steps until it finds a "weighted augmenting path". The first three types of steps build a search structure of tight edges: a *grow step* adds new tight edges to the structure; a *blossom step* constructs a new blossom in the structure; an *expand step* replaces an unweighted blossom by its components. These steps are repeated until the search structure is maximal.

If the resulting structure does not contain a weighted augmenting path, a *dual variable adjustment* is done. This step starts by computing a quantity δ. The duals of all free vertices are decreased by δ. Other duals y_i change by $\pm\delta$ or zero; z_B's change by $\pm 2\delta$ or zero. The adjustment keeps the duals dominating and tight. Our assumption of even edge weights ensures that all quantities computed are integers [PS, p. 267, ex. 3]. The dual adjustment decreases the dual objective $(y, z)V$ by $f\,\delta$, where f is the number of free vertices.

After the dual adjustment the search continues with grow, blossom and expand steps. Eventually the search stops when it finds a *weighted augmenting path* P. This is an augmenting path whose edges are tight. The *augment step* enlarges the matching M by one edge to $M \oplus P$.

Now we introduce the notion of a shell. In a complete structured matching if B is a blossom with a descendant C, the graph induced on $B-C$ is a *shell*.

Shell Lemma. A maximum complete matching on a shell $B-C$ exists and weighs $(y,z)B-(y,z)C$; for a child C of B this is $(y,z)(B-C)$.

Proof. Prove the first assertion as follows. First suppose that C is a vertex i. Form the graph B' from B by adding a vertex i' and an edge ii' of weight zero. The blossom structure gives a complete matching on $B-i$, which extends to B' by matching ii'. Define duals (y',z') on B' to be identical to (y,z) except that $y'_{i'} = -y_i$ and $z'_B = \sum_{B \subseteq D} z_D$. These duals are dominating and tight. Thus a maximum complete matching on B' exists and weighs $(y',z')B' = (y,z)B - y_i$. So a maximum complete matching on $B-i$ exists and weighs $(y,z)B - y_i$, as desired.

In the general case choose an arbitrary vertex $i \in C$. The blossom structure gives a complete matching on $B-i$ (as above) which is also complete on $C-i$ and $B-C$. It weighs $((y,z)B-y_i)-((y,z)C-y_i) = (y,z)B-(y,z)C$. A larger matching would give a larger matching on $B-i$, which is impossible. ∎

2.2. The scaling routine and the difficulty.

The idea of scaling is to solve the problem recursively for a graph with edge weights $\lfloor w_{ij} / 2 \rfloor$ and transform that solution, which is "close to optimum", to the desired optimum solution. To ensure even edge weights use a slightly different regime: The scaled weight function $\overline{w}$ has $\overline{w}_{ij} = 2\lfloor w_{ij} / 4 \rfloor$. In the algorithm below the input graph G has even edge weights and the output is a complete structured matching.

Procedure *scale* (w).
Step 0. If all weights w_{ij} are zero return a complete matching M, all duals $y_i = 0$, and no blossoms.
Step 1. Construct the scaled weight function $\overline{w}$ and call *scale*$(\overline{w})$ recursively to find a complete structured matching with duals y,z and blossom tree T.
Step 2. Let M be the empty matching on G. For each vertex i set $y_i^0 \leftarrow 2y_i + 1$; for each blossom B set $z_B^0 \leftarrow 2z_B$.
Step 3. Use the modified duals y^0, z^0 and blossom tree T to find the desired matching by calling a routine *match*.
∎

This algorithm has $O(lg\ N)$ levels of recursion. The extra space is $O(m)$ since the i^{th} level of recursion can compute the weight of an edge with given weight w as $2\lfloor w / 2^{i+1} \rfloor$.

The duals y^0, z^0 are dominating on G. (The expression for y^0 accounts for the bit in w_{ij} that is lost in passing to $\overline{w}_{ij}$). The duals are also close to optimum in that if M^* is a maximum complete matching on G,

$$(y^0,z^0)V \geq w(M^*) \geq (y^0,z^0)V - n \qquad (3)$$

The first inequality follows from (2); the second follows since the matching found in Step 1 achieves equality in (2). The shortcoming is that the blossom structure of Step 1 need not valid be for the duals y^0, z^0, since the edges of blossom subgraphs need not be tight. So Edmonds' algorithm, which requires a structured matching, cannot be applied directly in the *match* routine. Note that in network flow [EK] and bipartite matching [G83b], scaling up the duals presents no problem.

There are two approaches to remedy this shortcoming. Neither leads to an efficient algorithm but both are used as tools in the efficient algorithm.

The "distribution approach" gets valid duals by eliminating blossoms from the previous scale. To *distribute* δ *units of blossom* B means to decrease z_B by δ and for each vertex $i \in B$, increase y_i by $\delta/2$. (Edmonds' algorithm distributes inner blossoms.) Distribution maintains dominance. However it increases the dual objective $(y,z)V$ by $\delta/2$, moving it further from optimum. The distribution approach eliminates blossoms by distributing all z_B units of each blossom. But since the resulting objective function is no longer close to optimum and efficient scaling depends on this property, the distribution approach is inadequate. Section 2.4 does more controlled distributions.

The basic defect in the distribution approach lies in the fact that z duals are necessary. For instance in Figure 1 *any* set of dominating duals that does not use z has $(y,z)V$ at least $2N$, not close to the optimum value zero.

The "bottom-up approach" processes blossoms from the previous scale bottom-up: After all subblossoms of B have been processed, B essentially has a structured matching. So B can be processed with a variant of Edmonds' algorithm that drives z_B to zero as desired. The problem is that the bottom-up approach can be inefficient: If a postorder traversal of the blossom tree is used, an edge can be scanned n times (when the blossom tree is a path). The next section gives a more efficient bottom-up approach.

2.3. The *match* routine and compressed postorder.

We start with some terminology. The *match* routine works with two types of blossoms. It is given the blossom tree of the previous scale (see Step 3 of *scale*). An *old blossom* is a node in this tree. The *match* routine constructs a *current* matching M (initially empty in Step 2 of *scale*.) A *current blossom* is a blossom in M. Although M has an associated blossom tree it is irrelevant in the current scale. The relevant tree is the one from the previous scale and hereinafter the term "blossom tree" refers exclusively to it.

Similarly *match* works with dual variables z_B for both current and old blossoms. Hence in the definition of dominance (1) the summation is over duals z_B for both current *and old* blossoms. Similarly for tightness and the dual value function $(y,z)S$. An old blossom *dissolves* when its dual z_B becomes zero (it is no longer weighted) or it becomes a current blossom. As a special case the root of the blossom tree, which is the graph G, is considered to be a blossom that never dissolves. The object of the *match* routine is to dissolve all old blossoms except G.

The *match* routine maintains y duals so that for each vertex i,

$$y_i \geq y_i^0 . \qquad (4)$$

This ensures dominance on the edges going out of the blossom being processed, so these edges can be safely ignored.

The efficient bottom-up algorithm uses the idea of a compressed tree, introduced by Tarjan [T79]. In any tree, for a node i let n_i denote the number of leaves descending from i. A descendent j of i is a *major descendant* of i if $n_j > n_i/2$. The major descendants of i form *the major path from* i. Partition the vertices of a tree into major paths as follows: Start with the major path from the root of the tree, and then recursively partition the trees rooted at the children of this path. A *start vertex of a major path* is the first vertex of any major path in this partition.

Procedure *match*.
Partition the blossom tree into major paths. Traverse the start vertices of major paths in postorder. At each start B call a routine *path*(B) to dissolve the old blossoms of B's major path, maintaining (1) and (4), and in addition when $B = G$, to find a complete structured matching. ∎

Lemma 2.1. If the time for *path*(B) is $O(n_B{}^{3/4} m_B)$ then *match* is $O(n^{3/4} m)$.
Proof. Fix an integer i and consider all blossoms B that start major paths and satisfy $n / 2^{i-1} > n_B \geq n / 2^i$. Any vertex is in at most one of these blossoms, so any edge is in at most one of them. Thus for some constant c, the time spent on these blossoms is less than $c\, n^{3/4} m / 2^{3(i-1)/4}$. Summing over i gives the desired bound. ∎

Now we discuss *path*(B). Let P be the major path from B. The *shells of* B are the shells formed by consecutive undissolved (old) blossoms of P. As *path* executes and blossoms dissolve, the shells of B merge together until finally none remain. There are two special shells at the boundaries. The *first shell of* B, $A-A'$, is the one formed by the first two undissolved blossoms A,A' on P. ($A = B$ as long as B is weighted or if $B = G$.) The *last shell of* B, $Z-\phi$, is the last undissolved blossom Z on P, considered to be a shell. This is not really a shell as defined in Section 2.1 so its properties are slightly different. *Path*(B) repeatedly calls a routine *search*(S) to search each shell S of B and dissolve old blossoms. Before giving the *path* routine we describe *search*.

2.4. The *search* routine and dissolving blossoms.

The *search* routine is Edmonds' algorithm modified to take old blossoms into account. Consider a shell $C-D$ that has one or more free vertices. The *search* routine runs Edmonds' algorithm on $C-D$. (Recall this shell is defined as an induced subgraph.) When the duals are adjusted by a quantity δ, 2δ units of C and 2δ units of D are distributed (see Section 2.2.) There are two exceptions: For the last shell $Z-\phi$ only 2δ units of Z are distributed; when $C = G$ only 2δ units of D are distributed. A *dissolve step* is executed when C or D dissolves. This step enlarges the shell, by adding the shell adjacent to the dissolved blossom C or D (forming a new induced subgraph). Any free vertex that gets added to the shell is immediately added to the search structure of Edmonds' algorithm (as a new outer vertex). Edmonds' algorithm is modified to allow for dissolve steps when calculating the value of the next dual adjustment.

There are two ways for the shell search to stop. First, any search stops when a weighted augmenting path is found in the shell. As a special case this implies that if a dissolve step enlarges the shell by adding a new shell S', and S' has already been searched and found to contain an augmenting path, the current search stops. Second, a search of the first shell $A-A'$ stops if an augmenting path is found or alternatively if A dissolves. Note that the latter makes $A'-A''$ the first shell (for some descendant A''); a subsequent search of this shell can stop by dissolving A'. As a final special case note that in searching the last shell $Z-\phi$ it is possible that all old blossoms dissolve, and so *path* is complete.

In the following Lemma f denotes the number of free vertices in a shell $C-D$. If $C-D$ is not the last shell of B then f is even; if it is last then f is odd.

Lemma 2.2. (i) A shell search maintains dominance and tightness (1), and (4).

(ii) If $C-D$ is not the last shell of B a dual adjustment of δ decreases $(y, z)B$ by $\delta(f-2)$ and $(y, z)(C-D)$ by δf. If it is the last shell $Z-\phi$ a dual adjustment of δ decreases $(y, z)B$ and $(y, z)Z$ by $\delta(f-1)$.

Proof. (i) Edmonds' algorithm maintains dominance and tightness on the edges of the shell. The distributions of C and D ensure dominance on all edges and also property (4). Note that the distribution of D is needed for dominance on edges ij where $i \in D$, $j \in C-D$: z_C decreases by 2δ and y_j may have no net change, so y_i may need to increase by 2δ. The proof of (ii) is similar. ■

Correctness of *search* is essentially Lemma 2.2(i). For the efficiency several data structures are used. The data structure for expand steps is discussed in Section 4. Details of various lists and indicators are omitted from this abstract. The remaining data structure is the priority queue needed for Edmonds' algorithm. For any shell $C-D$ except the last, the priority queue can be implemented as an array, for the following reasons. The algorithm works only with integers. (Dissolve steps cause no problem with the integrality of δ, since a z dual is always even). A calculation similar to Lemma 2.3 below shows that the dual objective on $C-D$ decreases by at most $n_C - n_D$. So Lemma 2.2(ii) implies that an array with $n_C - n_D$ entries can be used as the priority queue.

Thus in a search of any shell but the last, the only processing that uses more than linear time is the algorithm of Section 4 for expand steps. So executing *search* once on every shell of B except the last uses time

$$O(m_B\ \alpha(m_B, n_B)). \tag{5}$$

The same priority queue can be used to a search the last shell when $f > 1$. When $f = 1$ similar reasoning can be applied, but the array priority queue is not quite as efficient: The calculation of Lemma 2.3 shows that the last shell $Z-\phi$ dissolves after dual adjustments totalling n_Z. Since there can be $\Omega(n_B)$ old blossoms this gives quadratic time for a search of a last shell that dissolves all blossoms. This suffices for our time bound when $m = \Omega(n^{5/4})$ but not sparser graphs. However as shown in Lemma 2.5 it suffices to use a method that, in a search of a last shell that dissolves all blossoms, uses time $O(n^{1/2} m\, \alpha(m,n))$. This can be achieved by another approach based on array priority queues. Alternatively the algorithm of [GGS] can be used to achieve time $O(m \lg\lg \lg_d n + n \lg n)$. The array priority queue is slower but probably preferable in practice, because of its simplicity.

2.5. The *path* routine and batching augmenting paths.

Procedure *path*(B).
Repeat the following steps until all old blossoms of B (except G) dissolve and, if $B = G$, the matching is complete.

Step 1. Construct a graph T from B by contracting every current maximal weighted blossom C, and keeping only the edges of the contracted graph that are tight and have both vertices in the same shell. Transform the current matching M to a maximum cardinality matching on T, using the Micali-Vazirani cardinality matching algorithm. Match B according to M.

Step 2. Order the shells of B that have at least one free vertex as S_i, $i = 1, \cdots, k$, so the number of free vertices in S_i is nonincreasing.

Step 3. For $i = 1, \cdots, k$, if S_i is still a shell (i.e., no shells have dissolved into it) do *search*(S_i). ■

Observe that when B is a blossom, *path* can end in two ways. Possibly all old blossoms dissolve and *path* ends with a matching on B of arbitrary cardinality. Alternatively *path* can find a maximum cardinality matching on B (with exactly one free vertex) before all blossoms dissolve. In this case k becomes one in Step 2 and Step 3 does a search that eventually dissolves all blossoms. Also observe that when $B = G$, after all old blossoms except G dissolve Steps 2-3 amount to doing a search of Edmonds' algorithm.

To show *path* is correct, note that Step 1 matches only tight edges. It modifies the current blossom tree only by removing maximal blossoms that are unweighted. So the resulting matching, together with the modified current blossom tree and the unchanged duals, is structured. This is valid input to the *search* routine, which requires a structured matching since it uses Edmonds' algorithm. Now we turn to the efficiency of *path*.

Lemma 2.3. *path*(B) decreases $(y, z)B$ by at most n_B.

Proof. Assume B is a blossom; when $B = G$ the argument is a simple special case. First note from Lemma 2.2 that the dual objective never increases. In particular when the *match* routine calls *path*(B), the previous executions of *path* did not increase the dual objective $(y,z)B$. So when *path*(B) starts, $(y,z)B$ is at most $(y^0,z^0)B$. Furthermore $(y,z)B$ never increases in *path*(B). So we need only show that any duals y,z computed by *path*(B) satisfy

$$(y,z)B \geq (y^0,z^0)B - n_B\ .$$

Choose any vertex $i \in B$. Let w be the weight of a maximum complete matching on the shell $B-i$. Then

$$w \geq (y^0,z^0)B - y_i^0 - n_B\ .$$

This is an analog of (3), and also depends on the Shell Lemma applied to the matching from the previous scale and shell $B-i$. Any duals computed in *path*(B) are dominating, and so satisfy

$$(y,z)B - y_i \geq w\ .$$

(Strictly speaking this involves a construction of duals similar to that in the proof of the Shell Lemma.) Combining these inequalities along with (4) (also maintained by *path*(B)) gives the desired result. ■

Now we analyze the number of iterations of Steps 1-3 of *path*. For a given iteration let F denote the set of vertices in B that are free immediately before Step 2 begins. Let $f = |F|$.

Lemma 2.4. For any exponent $e > 0$ the number of iterations with $f \ge n_B{}^e$ is $O(n_B{}^{1-e})$.

Proof. Assume B is a blossom; when $B = G$ the argument is a simple special case. Call a shell *small* if it has at most two free vertices and *big* otherwise. Consider an iteration with $f \ge n_B{}^e$.

The first case is when at least $f/2$ vertices of F are in big shells (immediately before Step 2). Observe that in general, Step 3 decreases the duals of at least one third of the free vertices initially in big shells by one or more. This is true because after Step 1, the matching M on T does not have an augmenting path. Hence a search of any shell must do at least one dual variable adjustment. (This follows from inspection of Edmonds' algorithm. Alternatively it follows since in any structured matching, an augmenting path that keeps the matching structured gives an augmenting path on the graph with all weighted blossoms contracted.) So any shell searched in Step 3 has its duals adjusted by at least one. A shell may not have its duals adjusted if an adjacent shell dissolves into it in the same dual adjustment that creates a weighted augmenting path. However the ordering of the searches (Step 2) guarantees that such a shell is adjacent to a shell that was adjusted and had at least as many free vertices. Thus at least one third of the free vertices in big shells are decreased, as desired.

So in the first case the big shells whose duals are adjusted contain at least $f/6$ vertices of F. Since the shells are big Lemma 2.2 implies that $(y, z)B$ decreases by at least $f/12 \ge n_B{}^e / 12$. Lemma 2.3 implies this case occurs $O(n_B{}^{1-e})$ times.

The second and remaining case is when at least $f/2$ vertices of F are in small shells. Assume the iteration is not the first. Any shell with a free vertex had its matching enlarged by at least one edge in Step 1, by the stopping condition of *search* in the previous iteration. There are at least $f/4$ such shells. So at least $f/2$ vertices were matched in Step 1. Thus Step 1 multiplied f by $2/3$ or less. So this case occurs $O(lg\ n_B)$ times. ■

Lemma 2.5. The time for *path*(B) is $O(n_B{}^{3/4}m_B)$.

Proof. For convenience, in this proof only, let n denote n_B and m denote m_B. First consider the time for Steps 2-3. These steps are executed $O(n^{1/2})$ times: The number of executions with $f \ge n^{1/2}$ is $O(n^{1/2})$ by Lemma 2.4. The number with $f \le n^{1/2}$ is $O(n^{1/2})$, since each execution of Step 1 except possibly the last matches at least one more edge, by the stopping condition of *search*. Step 2 is done in linear time with a bucket sort. From (5) one execution of Step 3 uses time $O(m\,\alpha(m,n))$ for all searches except the last shell. So the total time for Steps 2-3 excluding last shells is $O(n^{1/2}m\,\alpha(m,n))$. Over all iterations the last shell searches amount to at most one search of the entire blossom. This can be done in the above time, as indicated in Section 2.4. So Steps 2-3 use less time than the bound of the Lemma.

Next consider Step 1. The number of executions with $f \ge n^{3/4}$ is $O(n^{1/4})$ by Lemma 2.4. One execution uses $O(n^{1/2}m)$ time [MV], giving total time equal to the bound of the Lemma. The executions with $f \le n^{3/4}$ match a total of at most $n^{3/4}$ additional vertices; each execution except possibly the last matches at least one more edge, as above. An execution matching a additional edges uses $O((a+1)m)$ time (by inspection of [MV]; note that Step 1 uses the current matching as input to the cardinality matching algorithm). This gives total time equal to the bound of the Lemma. ■

2.6. The final result and applications.

Theorem 2.1. A maximum complete matching on a graph with integral edge weights can be found in $O(n^{3/4}m\ lg\ N)$ time and $O(m)$ space. The same bounds hold for maximum weight matching.

Proof. The time bound follows from Lemmas 2.5, 2.1 and the *scale* routine. The final step is to show that throughout the algorithm the duals y,z have magnitude $O(nN)$. (This justifies charging $O(1)$ time for each arithmetic operation). This is proved by bounding the change in duals in one execution of *path*, then in all executions of *path* (using the proof of Lemma 2.1), and finally in *scale* (as in [G83b]).

To justify our various assumptions, note that the base case of *scale* can check that the graph has a complete matching. To make all edge weights nonnegative even integers, if the given weights are arbitrary integers in the interval $[a,b]$ then transform any weight w to $2(w-a)$. This does not change the maximum complete matchings. Note that in the time bound N can be taken as the largest magnitude of a given edge weight or alternatively as the spread $b-a$.

Maximum weight matching is done by complete matching using the reduction of [G83b]. In this case negative edges can be deleted, so in the time bound N is the largest positive weight. ■

When N is very large or weights are real numbers, our algorithm can find an approximately optimum matching by running a limited number of scales [G83b]. As an example consider minimum complete matching. For this problem there is a slight difficulty: If some complete matching contains only very small edges the initial scales may yield no useful information. This difficulty is overcome by choosing an appropriate starting scale. In the approximate minimum complete matching algorithm below, the input is a graph G with nonnegative real edge weights and an integer $e \ge 0$; the output is a complete matching weighing at most $(1 + 1/n^e)$ times optimum.

Step 1. Let b be weight of a minimum bottleneck matching, that is, the minimum value such that there is a complete matching M on the edges of weight b or less. Delete all edges weighing more than $w(M)$ from G.

Step 2. Let $N = \max\{w_{ij} \mid ij$ is still in $G\}$. Define new edge weights $w'_{ij} = 2\lfloor (N - w_{ij})n^{1+e} / b \rfloor$. Return the matching found by *scale*(w'). ■

Corollary 2.1. Consider a graph with nonnegative real edge weights and a positive integer e. For minimum (maximum) complete matching, a complete matching weighing at most $(1 + 1/n^e)$ (at least $(1 - 1/n^e)$) times optimum can be found in $O(\epsilon n^{3/4} m \lg n)$ time and $O(m)$ space.

Proof. First analyze the above algorithm for minimum matching. To prove the accuracy bound let O be an optimum matching and A the matching returned by the algorithm. Set $F = 2n^{1+\epsilon} / b$, so that for any edge ij the weights w_{ij} and w'_{ij} are related by

$$F(N - w_{ij}) - 2 \le w'_{ij} \le F(N - w_{ij}) \ .$$

No edge of O is deleted in Step 1 since $w(O) \le w(M)$. Thus O is a matching on the graph of Step 2 and $w'(O) \le w'(A)$. The above relation, on the edges of O and A, implies $Fw(A) \le Fw(O) + n$. The bottleneck matching implies $w(O) \ge b$. So $w(A) \le w(O) + b / n^{\epsilon} \le (1 + 1/n^{\epsilon}) w(O)$, as desired.

For the time bound, the bottleneck matching M can be found in $O((n \lg n)^{1/2} m)$ time [T85]. For Step 2, since $N \le w(M) \le nb/2$ any weight w'_{ij} is at most $n^{2+\epsilon}$. Thus *scale* (w') runs in the bound of the Corollary.

Maximum matching is similar but there is no need to find the starting scale: The algorithm defines new edge weights $w'_{ij} = 2\lfloor w_{ij} n^{1+\epsilon} / N \rfloor$ (for N the maximum given weight) and returns the matching found by *scale* (w'). ▪

In practice the minimum matching algorithm can be sped up by noting that the last $\lg n$ levels of recursion in *scale* are unnecessary. This stems from the fact that in these levels all edges of the bottleneck matching have essentially the largest weight, so good duals are easily defined. Also the above algorithms, and any approximation algorithms that follow this scaling approach, have a "restart" feature: If the final structured matching is saved then a solution with greater accuracy can be computed by resuming the computation at the last scale.

The only other approximation algorithm with a known bound on the relative accuracy is the greedy algorithm [A]. For maximum weight matching with nonnegative real weights it finds a solution that is at least half the optimum weight in $O(m \lg n)$ time. Corollary 2.1 extends to maximum weight matching by the reduction of Theorem 2.1.

The approximation algorithm speeds up Christofides' algorithm for the travelling salesman problem [C]. That algorithm can use a matching that is at most $(1 + 1/n)$ times optimum; the slight inaccuracy in the matching is compensated for by the spanning tree of the algorithm.

Corollary 2.2. For n cities with real-valued distances satisfying the triangle inequality, a travelling salesman tour that is at most 3/2 times optimum can be found in time $O(n^{2.75} \lg n)$ time and $O(n^2)$ space. ▪

Returning to integral weights, a useful special case is when $N = O(1)$. In this case scaling is unnecessary, and the algorithm behaves like *path*(G) after all blossoms are dissolved. Using this approach a maximum weight matching can be found in $O(n^{1/2} m)$ time and $O(m)$ space [G83b].

When $N = 1$ the problem is *red-green matching:* Given a graph with each edge colored red or green, find a complete matching with the greatest possible number of red edges. Since on bipartite graphs our algorithm is identical to that of [G83b], that paper exhibits a red-green matching problem where our algorithm uses $\Theta(n^{3/4} m)$ time. Thus our timing analysis for one scale is tight.

We close this section by discussing update problems on matchings. Update algorithms have been presented in [BD, CM, and W] but they change the dual variables too much for our purposes. We give an update algorithm for graphs with arbitrary real weights, that refines [BD]. Like the three above algorithms its asymptotic running time is the time to find one weighted augmenting path in a structured matching. However it appears simpler in that it does not use artificial vertices or edges, nor does it modify Edmonds' search routine. The algorithm gives an asymptotic speedup for "small" update problems.

The *matching update problem* is to start with a complete structured matching and process (online) a sequence of update operations. Each update operation changes the graph arbitrarily at one vertex i – edges incident to i can be added, deleted or changed in weight. A new complete structured matching must be found after each update.

An update is accomplished as follows. First suppose i is not in a weighted blossom. Then unmatch the matched edge incident to i, set y_i to ensure dominance on the new edges incident to i, and search for an augmenting path from i. Next suppose i is in a weighted blossom; let the maximal such blossom be B with base vertex b. Unmatch the matched edge incident to b, make i the base of B (by rematching one alternating path [G76]) and distribute all units of all blossoms containing i. Observe that now i is not in a weighted blossom, i and exactly one other vertex are free, and the matching is structured. So proceed as in the first case.

Now consider the update problem for red-green matching, or more generally, for complete matching with $N = O(1)$. An update operation changes the weight of the optimum matching by $O(n)$. In the above update algorithm, the value of the dual objective $(y,z)V$ right before the search begins is its original value plus the net increase in y_i. Assume for the moment that this net increase is $O(n)$. Then the dual objective is within $O(n)$ of optimum. Using the data structures of Section 2.4 an augmenting path can be found in time $O(m\alpha(m,n))$, which is the time for the update.

Note that the value used for y_i in the update algorithm is $-y_j + O(1)$ for some vertex j. So the assumption is valid if all duals are $O(n)$ in magnitude. This is the case for the initial structured matching if it is found by our algorithm (see the proof of Theorem 2.1). It is maintained by searching an appropriate amount from each free vertex. (If this is not done correctly the magnitude of the duals can increase exponentially in the number of updates.)

Corollary 2.3. Consider the update problem for red-green matching, or more generally for complete matching with $N = O(1)$. Each update operation can be done in time $O(m\alpha(m,n))$, where n and m give the size of the current graph. The update operations allowed are adding and deleting arbitrary edges incident to a vertex, adding two new vertices (and incident edges), deleting two vertices, and changing the weights of the edges incident to a vertex (i.e., recoloring in red-green matching). Each operation of the last type can be done in time $O(m)$.

Proof. Further details of the search are given in the complete version of this paper. For weight changes, the *search* routine for a last shell (Section 2.4) is used. In fact even if N is arbitrary, this routine handles weight changes by $O(1)$ in the edges incident to a vertex in time $O(m)$. ▪

Other types of updates can be handled. For instance for maximum weight matching with arbitrary N, updates that add a vertex with incident edges weighing $O(m)$ can be done in time $O(m\alpha(m,n))$.

3. Degree-constrained subgraphs.

Consider a multigraph with integral edge weights, where each vertex i has associated integers l_i and u_i. A *degree-constrained subgraph* (*DCS*) is a subgraph where each vertex i has degree between l_i and u_i; in a *complete DCS* each degree is u_i. Let $U = \sum_{i \in V} u_i$. This section gives a DCS algorithm that runs in time $O(U^{3/4}m \lg N)$. Applications to shortest paths on undirected graphs, the Chinese postman problem and maximum cut of a planar graph are also given.

Space allows just an overview of the DCS algorithm; details are in the complete paper. The approach is to reduce complete DCS to complete matching. A DCS on a multigraph G corresponds to a matching on a graph G', where each vertex i of G is replaced by a *vertex substitute* that is a complete bipartite graph. The *scale* routine is modified to work efficiently on G'. Each scale initializes the matching to cover all "extra" vertices in the substitutes. This is done by working with the weighted blossom tree instead of the blossom tree, to keep the vertices of each substitute in the same shell. (To prove this works, a characterization is given for an alternating cycle of four equal weight edges in a structured matching.) The dual objective function at the start of a scale is at most U above optimum, since matched edges in substitutes are tight. So the partition into major paths is defined not with respect to n_B but u_B, the number of original edges that have both vertices in B and are matched in the previous scale.

For further efficiency the algorithm does not work on G' directly since it can have $\Omega(nm)$ edges, but rather on graphs with $O(m)$ edges. For shell searches G' is replaced by a *sparse substitute graph* G_k, constructed using a *sparse substitute* for each vertex. This method was introduced in [G83a]. Slightly larger substitutes are used here in order to ensure that structured matchings on G' and G_k correspond. This allows the algorithm to switch between the two graphs at will. The alternating cycle characterization is used to construct these sparse substitutes. A different sparse model is used for the cardinality matching in *path*, similar to [G83b].

Theorem 3.1. A maximum complete DCS on a multigraph with integral edge weights can be found in $O(U^{3/4}m \lg N)$ time and $O(m)$ space. The same bounds hold for maximum weight DCS. ▪

Now consider the update problem. DCS update operations like adding or deleting an edge, increasing or decreasing two degree constraints by one, and adding or deleting a degree zero vertex all translate into $O(1)$ matching updates of the types in Corollary 2.5 on G'. The matching updates can be done by searching the sparse substitute graph G_k. So the analog of Corollary 2.3 holds for DCS.

Certain path problems are solved most efficiently as DCS problems. Consider the *p-paths problem*: Given an undirected graph with nonnegative edge weights and a set of $2p$ vertices S, find p paths of minimum total weight whose endpoints are the vertices of S. To solve this, for any vertex i let δ_i be one if $i \in S$ and zero otherwise. Let l_i be the number of paths that i can be an intermediate vertex on, that is, $l_i = \min\{p-\delta_i, \left\lfloor \frac{d_i-\delta_i}{2} \right\rfloor\}$. Construct a multigraph $\hat{G}$ from G by adding l_i weight zero loops at each node i and using the degree constraint $u_i = 2l_i+\delta_i$. A minimum complete DCS on $\hat{G}$ solves the p-paths problem. (This follows because the p-paths problem has a solution where no edge is in more than one path.) Hence the p-paths problem can be solved in $O(\min\{np,m\}^{3/4}\, m \lg N)$ time and $O(m)$ space.

On planar graphs the p-paths problem can be solved in $O(n^{3/2} \lg n)$ time and $O(n)$ space. First observe that a DCS on a planar graph where all degree constraints u_i are $O(1)$ can be found in $O(n^{3/2} \lg n)$ time. (This generalizes a result of [MNS].) The idea is to use the planar separator theorem as in the matching algorithm of [LT], and use the sparse substitute graph G_k to search for augmenting paths. The p-paths problem on a graph G reduces to a DCS problem on a graph with bounded degrees: Replace any vertex i of G having degree d by a cycle of d vertices, distinguishing one vertex of the cycle if $i \in S$; attach one edge of G to each cycle vertex, and attach a loop at each nondistinguished vertex; edges of G have their original weight and all other edges weigh zero; the degree constraint of a vertex is one if it is distinguished and two if it is not. The new graph is planar if G is, so the bounds for the p-path problem follow.

Now consider the shortest path problem for two given vertices in an undirected graph with no negative cycles. The above reductions are still valid (they hold for the p-paths problem on graphs with negative edges as long as there are no negative cycles and the paths must be disjoint). Here the first reduction becomes the one given in [L]. The all-pairs version of this problem is solved by a two phase algorithm of [G83a]. The first phase finds a minimum complete DCS. The second phase amounts to adding a vertex with one edge, n times. So Corollary 2.3 applies.

Corollary 3.1. The single-source shortest path problem on an undirected graph with integral edge lengths of magnitude at most N and no negative cycles can be solved in $O(n^{3/4}m \lg N)$ time and $O(m)$ space; on planar graphs, $O(n^{3/2} \lg n)$ time and $O(n)$ space. The all-pairs problem, where edge lengths are plus or minus one (more generally, magnitude $O(1)$) can be solved in $O(nm\,\alpha(m,n))$ time and $O(m)$ space. ▪

Perhaps a different approach can remove the α factor from the bound for the seemingly simple all-pairs problem.

The Chinese postman problem gives another p-paths problem [EJ]. Finding a maximum cut of a planar graph with nonnegative edge weights reduces to a p-paths problem on the dual of the given graph [H]. The cardinality versions of these problems (i.e., find a tour with the fewest edges or a cut with the most edges) are DCS problems with $N = 1$.

Regarding updates, adding or deleting an edge in the Chinese postman problem translates into $O(1)$ DCS update operations of the type listed above. The same holds for maximum planar cut, assuming the reduction to DCS for planar graphs is used. However it is necessary to work with a fixed embedding of the planar graph: The update problem starts with a plane graph and any edge added joins two vertices currently on the same face.

Corollary 3.2. The Chinese postman problem can be solved in $O(m^{7/4} lg\ N)$ time and $O(m)$ space. A maximum cut of a planar graph with nonnegative edge weights can be found in $O(n^{3/2} lg\ n)$ time and $O(n)$ space. For the update problem for cardinality Chinese postman or maximum cardinality cut of a plane graph, an edge can be added or deleted in time $O(m\alpha(m,n))$, where n and m give the size of the current graph. ▪

An alternate approach to the Chinese postman problem and the maximum planar cut problem is based on matching [L,H]. It uses $O(n^3)$ time and $O(n^2)$ space. The p-paths approach is more efficient for maximum cut, and more efficient for the postman problem on sparse graphs. The matching approach to these problems does not seem to support efficient updates.

4. The list splitting algorithm.

The *splitting problem* is defined on a universe of elements partitioned into lists. Every element x is in a list $L(x)$ and has a *cost* $c(x)$, a real number or infinity. The cost $c(L)$ of a list L is defined as the smallest cost of an element of L. The object is to process (online) a sequence of two types of operations:

decreasecost(x,d) - replace $c(x)$ by $\min(c(x),d)$ and $c(L(x))$ by $\min(c(L(x)),d)$.
split(x) - replace list $L(x)$ by L_1, the sublist of all elements up to and including x, and L_2, the remainder; set $c(L_i)$ to its proper value.

The operations can be intermixed. It is convenient to allow a third operation that can also be intermixed: *initialize*(L) initializes a list L of elements with arbitrary costs. This section gives an algorithm that processes a universe of n elements, hence *initializes* of at most n elements and at most n *splits*, and $m \geq n$ *decreasecosts* in time $O(m\alpha(m,n))$. The algorithm can be used in the scaling algorithm for matching and in Edmonds' algorithm.

The splitting problem models the portion of Edmonds' algorithm that expands blossoms. At the start of a search the algorithm constructs a list of all vertices that are in blossoms, in their left-to-right order as leaves in the blossom tree. Any blossom, maximal or not, corresponds to a sublist. The algorithm maintains the cost of a nonouter vertex x to be the minimum slack of an edge xy, where y is an outer vertex. It uses *decreasecost* for this. It uses *split* to expand a blossom into its subblossoms. It uses the costs of lists (i.e., blossoms) to choose the next step of Edmonds' algorithm to execute.

Define Ackermann's function by the relations

$A(i,0) = 2$, for $i \geq 1$;
$A(1,j) = 2^j$, for $j \geq 1$;
$A(i,j) = A(i-1,A(i,j-1))$, for $i \geq 2, j \geq 1$.

Define inverse functions

$a(i,n) = \max\{j \mid 2A(i,j) \leq n\}$, for $n \geq 4$;
$\alpha(m,n) = \min\{i \mid A(i,\lfloor m/n \rfloor) \geq n\}$, for $m \geq n$.

These inverses differ by at most one from those of [T83] and are more convenient for our purposes.

The splitting algorithm is recursive. We describe it as an infinite family of algorithms $\mathbf{A}_i$, $i = 1,2, \cdots$, where $\mathbf{A}_i$ calls $\mathbf{A}_{i-1}$. Algorithm $\mathbf{A}_i$ partitions each list L into a *head* H and a *tail* T. Any element in H precedes any element in T; H or T can be empty. H and T are themselves partitioned into "superelements". For $j \geq 0$, a ***level j superelement*** for $\mathbf{A}_i$ consists of $2A(i,j)$ consecutive elements. A superelement e in H has the highest possible level $a(i,|H_e|)$, where H_e is the portion of H from the first element of H to the last element of e. Thus H is partitioned into superelements plus at most three leftover elements at the start of the list (since $A(i,0) = 2$). Similarly a superelement e in T has highest possible level $a(i,|T_e|)$, where T_e is the portion of T from the first element of e to the last element of T. T is partitioned into superelements plus at most three leftovers at the end of the list. A maximal sequence of two or more consecutive level j superelements in H or T is called a ***level j sublist*** of L; a level j superelement not in a sublist, or a leftover element at the start of H or the end of T, is called a *singleton*. Algorithm $\mathbf{A}_i$ works by running $\mathbf{A}_{i-1}$ on $\mathbf{A}_i$'s sublists.

These data structures are used: Each list L has a doubly-linked list of elements, and its cost $c(L)$. Also L has a list of its sublists (of all levels) and singletons, and each sublist or singleton has a pointer to L. Each element x stores its cost $c(x)$ and a pointer to its superelement $e(x)$. It is also convenient to store a table of values of Ackermann's function $A(i,j)$ that are n or less.

The *initialize* operation can be done by either of two routines, *initialize-head* or *initialize-tail*. The former scans the elements right-to-left, partitioning into superelements, sublists and singletons; it calls $\mathbf{A}_{i-1}$ to do *initialize-head* on each sublist. The total time for *initialize-head* on a list of k elements is $O(k)$, since $\mathbf{A}_{i-1}$ is called on lists collectively containing at most $k/4$ elements. The *initialize-tail* routine is similar, using a left-to-right scan.

To do *decreasecost*(x,d), assume that *decreasecost* also returns a pointer to the list containing x. Suppose that x is not a singleton. Algorithm $\mathbf{A}_i$ calls $\mathbf{A}_{i-1}$ to do *decreasecost*$(e(x),d)$. Then it updates $c(x)$, uses the returned pointer to $e(x)$'s sublist to find $L(x)$, updates

$c(L(x))$ and returns $L(x)$. If x is a singleton there is no recursive call (a singleton superelement updates $c(e(x))$ directly). The time for one *decreasecost* is $O(i)$.

To do *split*(x) suppose that x is not a singleton. Algorithm $\mathbf{A}_i$ calls $\mathbf{A}_{i-1}$ to do *split*$(e(x))$ and then another split to make $\{e(x)\}$ into a list. If x is not the last element of $e(x)$ it does *initialize−head* on the elements of $e(x)$ up to and including x, and *initialize−tail* on the remaining elements. Then it constructs the rest of the data structure for the new lists L_i, $i = 1,2$, by examining each of the sublists of L_i. If x is a singleton there are no recursive calls.

To check *split* is correct, suppose that x is in a superelement and $e(x)$ is in the head of L. The new superelements of L_1 form its tail. The new superelements of L_2 are part of its head. Obviously the superelements after $e(x)$ in the head of L are on the correct level for the head of L_2. A symmetric argument holds if $e(x)$ is in the tail.

We show that the time for all *splits* is $O(n a(i,n))$. First consider the time exclusive of recursive calls. For *initialize−head* and *initialize−tail*, whenever an element x is in an initialization the level of $e(x)$ decreases (or x becomes a leftover). Since $\mathbf{A}_i$ uses $a(i,n)$ levels of superelements this gives $O(na(i,n))$ time. Next consider constructing the data structures for L_i. Since a list has $O(a(i,n))$ sublists and singletons and there are at most n *splits*, the total time is $O(na(i,n))$.

If $i = 1$ every superelement is a singleton. Hence there are no recursive calls and the time bound for *split* follows. Suppose $i > 1$. By induction assume that for some constant c, $\mathbf{A}_{i-1}$ uses at most $cka(i-1,k)$ time for *splits* on a universe of k elements. In algorithm $\mathbf{A}_i$ for any j, a level j sublist has at most $2A(i,j+1) / 2A(i,j) \le A(i,j+1)$ superelements. Since $a(i,n)$ is nondecreasing in n, the time per level j superelement is at most c times $a(i-1,A(i,j+1)) = a(i-1,A(i-1,A(i,j))) < A(i,j)$. There are at most $n / 2A(i,j)$ level j superelements. So the total time on level j superelements is at most $cn/2$. The total time on all levels is at most $cna(i,n)/2$. Choose c so that the time exclusive of recursive calls is at most $cna(i,n)/2$. Then the total time for *splits* is at most $cna(i,n)$. This completes the induction and proves the time for *splits* in $\mathbf{A}_i$ is $O(na(i,n))$.

Theorem 4.1. A splitting problem on a universe of n elements with $m \ge n$ *decreasecosts*, where m is known in advance, can be solved in time $O(m\alpha(m,n))$ and space $O(n)$.

Proof. Run algorithm $\mathbf{A}_i$ for $i = \alpha(m,n)$. The above discussion shows the time is $O(im+na(i,n))$. Since $a(\alpha(m,n),n) < \lfloor m/n \rfloor$ the time is $O(m\alpha(m,n))$. For the space note that the data structures for $\mathbf{A}_i$, exclusive of the recursive calls, use $O(n)$ space. Since there are at most $n/4$ superelements the space bound follows. ▪

If m is not known in advance choosing $i = \alpha(n,n)$ gives total time $O(m\alpha(n,n))$. The above algorithm also solves the *split−find* problem. (Here *split* is defined as above and *find*(x) returns a pointer to $L(x)$.) Hopcroft and Ullman [HU] solve this problem in $O(m\ lg^*n)$ time. The *split−find* algorithm of [GT] uses $O(m+n)$ time. However it requires a random access machine, while our algorithm is easily implemented on a pointer machine [T79]; also it does not seem to generalize to the splitting problem.

Our approach leads to other algorithms with similar time complexity. For instance it gives a set merging algorithm that runs in the bounds of Theorem 4.1.

Acknowledgment.

The author thanks Dr. Robert Tarjan for many inspiring conversations.

References.

[A] D. Avis, "A survey of heuristics for the weighted matching problem", *Networks 13*, 4, 1983, pp. 475-493.

[B] C.A. Bateson, "Performance comparison of two algorithms for weighted bipartite matching", M.S. Thesis, Department of Computer Science, University of Colorado, Boulder, Co. 1985.

[BD] M.O. Ball and U. Derigs, "An analysis of alternative strategies for implementing matching algorithms", *Networks 13*, 4, 1983, pp. 517-549.

[BJ] R.G. Bland and D.Jensen, manuscript to appear, Cornell University, Ithaca, New York.

[C] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem", Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon Univ., Pittsburgh, Pa., 1976.

[CM] W.H. Cunningham and A.B. Marsh, III, "A primal algorithm for optimum matching", *Math. Programming Study 8*, 1978, pp. 50-72.

[D] G.B. Dantzig, *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, N.J., 1963.

[E] J. Edmonds, "Maximum matching and a polyhedron with 0,1-vertices", *J. Res. Nat. Bur. Standards 69B* (1965), 125-130.

[EJ] J. Edmonds and E.L. Johnson, "Matching, Euler tours and the Chinese postman", *Math. Programming 5*, 1973, pp. 88-124.

[EK] J. Edmonds and R.M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM 19*, 2, 1972, pp. 248-264.

[EvK] S. Even and O. Kariv, "An $O(n^{2.5})$ algorithm for maximum matching in general graphs", *Proc. 16th Annual Symp. on Found. of Comp. Sci.*, 1975, pp.100-112.

[FHK] G.N. Frederickson, M.S. Hecht and C.E. Kim, "Approximation algorithms for some routing problems", *SIAM J. Computing 7, 1978*, pp. 178-193.

[G76] H.N. Gabow, "An efficient implementation of Edmonds' algorithm for maximum matching on graphs," *J. ACM 23*, 2, 1976, pp. 221-234.

[G83a] H.N. Gabow, "An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems," *Proc. Fifteenth Annual ACM Symp. on Th. of Computing*, 1983, pp. 448-456.

[G83b] H.N. Gabow, "Scaling algorithms for network problems", *Proc. 24th Annual Symp. on Found. of Comp. Sci.*, 1983, pp. 248-257; also *J. CSS*, to appear.

[GBT] H.N. Gabow, J.L. Bentley and R.E. Tarjan, "Scaling and related techniques for geometry problems", *Proc. 16th Annual ACM Symp. on Th. of Computing*, 1984, pp. 135-143.

[GGS] H.N.Gabow, Z.Galil, T.H.Spencer, "Efficient implementation of graph algorithms using contraction", *Proc. 25th Annual Symp. on Found. of Comp. Sci.*, 1984, pp.347-357.

[GT] H.N. Gabow and R.E. Tarjan, "A linear-time algorithm for a special case of disjoint set union", *Proc. 15th Annual ACM Symp. on Th. of Comp.*, 1983, pp. 246-251; also *J. CSS*, to appear.

[GMG] Z. Galil, S. Micali, H. Gabow, "Priority queues with variable priority and an $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs," *Proc. 23rd Annual Symp. on Foundations of Comp. Sci.*, 1982, pp. 255-261; also *SIAM J.Comp.*, to appear.

[H] F. Hadlock, "Finding a maximum cut of a planar graph in polynomial time", *SIAM J. Comp. 4*, 3, 1975, pp. 221-225.

[HK] J. Hopcroft, and R. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comp.* 2, 4, 1973, pp. 225-231.

[HU] J.E. Hopcroft and J.D. Ullman, "Set merging algorithms", *SIAM J. Comp. 2*, 4, 1973, pp. 294-303.

[IA] H. Imai and T. Asano, "Efficient algorithms for geometric graph search problems", RMI83-05, Dept. Math. Eng. and Instrumentation Physics, Univ. of Tokyo, 1983; also *J.Algorithms*, to appear.

[L] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[LT] R.J. Lipton and R.E. Tarjan, "Applications of a planar separator theorem", *SIAM J. Computing 9*, 3, 1980, pp. 615-627.

[MNS] K. Matsumoto, T. Nishizeki, and N. Saito, "Planar multicommodity flows, maximum matchings and negative cycles," *SIAM J. Computing*, to appear.

[MV] S. Micali and V.V. Vazirani, "An $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding maximum matching in general graphs," *Proc. 21st Annual Symp. on Found. of Comp. Sci.*, 1980, pp. 17-27.

[PS] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[RT] E.M. Reingold and R.E. Tarjan, "On a greedy heuristic for complete matching", *SIAM J. Computing 10*, 4, 1981, pp. 676-681.

[T79] R.E. Tarjan, "Applications of path compression on balanced trees", *J. ACM 26*, 4, (1979), pp. 690-715.

[T79] R.E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets", *J. CSS 18*, (1979), pp. 110-127.

[T83] R.E.Tarjan, *Data Structures and Network Algorithms*, SIAM Monograph, Philadelphia, Pa., 1983.

[T85] R.E.Tarjan, Problem 85-2, *J. Algorithms 6*, 2, 1985, p. 284, and private communication.

[W] G.M. Weber, "Sensitivity analysis of optimal matchings", *Networks 11*, 1981, pp. 41-56.