

Scaling Algorithms for Network Problems

HAROLD N. GABOW*

*Department of Computer Science, University of Colorado at Boulder,
Boulder, Colorado 80309*

Received April 17, 1984; revised May 31, 1985

This paper gives algorithms for network problems that work by scaling the numeric parameters. Assume all parameters are integers. Let n , m , and N denote the number of vertices, number of edges, and largest parameter of the network, respectively. A scaling algorithm for maximum weight matching on a bipartite graph runs in $O(n^{3/4}m \log N)$ time. For appropriate N this improves the traditional Hungarian method, whose most efficient implementation is $O(n(m + n \log n))$. The speedup results from finding augmenting paths in batches. The matching algorithm gives similar improvements for the following problems: single-source shortest paths for arbitrary edge lengths (Bellman's algorithm); maximum weight degree-constrained subgraph; minimum cost flow on a 0-1 network. Scaling gives a simple maximum value flow algorithm that matches the best known bound (Sleator and Tarjan's algorithm) when $\log N = O(\log n)$. Scaling also gives a good algorithm for shortest paths on a directed graph with nonnegative edge lengths (Dijkstra's algorithm). © 1985 Academic Press, Inc.

1. INTRODUCTION

A *network* is a graph with numeric parameters such as edge lengths, capacities, costs, or weights. Throughout this paper, n , m , and N denote the number of vertices, number of edges, and largest numeric parameter of the network, respectively.

For optimization problems on networks, one important technique is efficient priority queues. Priority queues often allow a network problem to be solved by the approach used for the analogous nonnumeric graph problem, increasing the time by only a factor of $\log n$.¹ For example the shortest path problem is solved by Dijkstra's algorithm on networks and breadth-first search on graphs. But priority queues don't always do the trick. For example a maximum cardinality matching can be found in $O(n^{1/2}m)$ time [17, 23] whereas the best bound for weighted matching is $O(n(m + n \log n))$ for bipartite graphs [10] and slightly more for general graphs [16]. The complexity gap here and in other problems results from the fact that the nonnumeric algorithm finds objects in batches, but numeric parameters force objects to be found one by one.

* This research was supported in part by the National Science Foundation under Grant MCS-8302648.

¹ Throughout this paper, \log denotes logarithm to an arbitrary constant base and \lg denotes logarithm base two.

This paper explores the scaling approach to network problems. Scaling can sometimes be used as an alternative to efficient priority queues. More importantly it can achieve batching in numeric problems. Edmonds and Karp [8] introduced the scaling approach for the minimum cost network flow problem. Since then the method has been largely ignored.

The scaling approach can be stated recursively or iteratively, and both viewpoints are useful. The recursive version is as follows: Given a network problem, halve the numeric parameters (i.e., a number l becomes $\lfloor l/2 \rfloor$) and solve this scaled-down problem recursively. Double the solution to get a near-optimum solution on the original network. Then transform the near-optimum solution to optimum.

The iterative version of scaling is as follows. View each number as its k -bit binary expansion $b_1 \cdots b_k$, where $k = \lfloor \lg N \rfloor + 1$. First solve the problem on the network where all parameters are 0. Then for $s = 0, \dots, k - 1$, transform the solution for parameters $b_1 \cdots b_s$ to a solution for parameters $b_1 \cdots b_{s+1}$.

Scaling relies on two assumptions. *Integrality* ensures applicability of the method. It requires the given numbers to be integers. This permits an easy transition from near-optimum solution to optimum. If the given numbers are rational they must be scaled up to integers before the method is applicable. Incommensurable real-valued inputs cannot be handled directly by this method, although this is not a limitation in practice. In the absence of integrality scaling gives an approximation scheme: Assume all parameters are normalized to lie in the interval $[0, 1)$ and view them as binary numbers $0.b_1b_2\cdots$. Compute solutions to the network, first with all parameters 0, then with parameters $0.b_1$, then $0.b_1b_2$, etc. The accuracy of each solution can be bounded. So scaling can find a solution with any desired accuracy.²

The second assumption, *similarity*, concerns efficiency of the method. It requires that the numbers of the problem be polynomially bounded, that is, if N is the largest number and n is the number of vertices, then $N = n^{O(1)}$. Scaling time bounds typically involve a factor $\log N$ due to $\lg N$ scalings of the problem. Similarity implies $\log N = O(\log n)$. This allows us to compare scaling bounds to previous ones. The assumption is for comparative purposes only, and we explicitly state when it is used. In the absence of similarity scaling algorithms can still be superior. For example for weighted matching scaling is faster for $N = 2^{o(n^{1/4})}$.

We suspect that similarity holds quite often in practical cases. Actually if it does not hold, then traditional algorithms must use a word size of $\lg N$ bits rather than $\lg n$ for the time bound to be valid (see Sect. 2.1). This opens the door for new approaches like bit vector algorithms. For instance if $N \geq 2^n$ so that a machine word has n bits, the Hopcroft-Karp algorithm (used in the scaling approach) can be implemented in time $O(n^{3/2} \log n + m)$ on a bit vector machine, improving the usual bound of $O(n^{1/2}m)$ [12].

Table I lists several network problems, the time for the best-known algorithm and the time for the scaling algorithm. The scaling algorithms for minimum spanning tree and bottleneck shortest path are based on the fact that n integers can be

² This approach was suggested by Dr. Robert Tarjan.

TABLE I
Algorithms for Network Problems

Best known algorithm	Scaling algorithm
<i>Minimum spanning tree</i> $O(m \log \beta(m, n))$ [16]	$O(m(\log_n N + \alpha(m, n)))$
<i>Bottleneck shortest path</i> $O(m + n \log n)$ [10]	$O(m \log_n N)$
<i>Shortest path (nonnegative lengths)</i> $O(m + n \log n)$ [10]	$O(m \log_{2 + m/n} N)$
<i>Maximum value network flow</i> $O(\min(n m \log n, n^3))$ [25, 19]	$O(n m \log N)$
<i>Maximum weight matching (bipartite graph)</i> $O(n(m + n \log n))$ [10]	$O(n^{3/4} m \log N)$
<i>Shortest path (arbitrary lengths)</i> $O(nm)$ [3]	$O(n^{3/4} m \log N)$
<i>Degree-constrained subgraph (bipartite graph)</i> $O(U(m + n \log n))$ [10]	$O(U^{3/4} m \log N)$, multigraph $O(U^{1/2} n^{1/3} m \log N)$, graph
<i>Minimum cost network flow (unit capacity)</i> $O(m(m + n \log n))$ [10]	$O(m^{7/4} \log N)$ $O(n^{1/3} m^{3/2} \log N)$, type 1 $O(n^{3/4} m \log N)$, type 2

Note. n = number of vertices, m = number of edges, N = largest network parameter, U = sum of upper bounds.

radix sorted in $O(n \log_n N)$ time. (Note that radix sorting is a scaling method.) Under similarity scaling is faster in both problems.

The remaining entries in the table are covered in this paper. The next two entries are discussed in Section 2. They illustrate how scaling can emulate efficient priority queues and other data structures. For shortest paths on directed graphs with non-negative edge lengths, Dijkstra's algorithm relies on an efficient priority queue. Under similarity scaling achieves the same efficiency as Johnson's algorithm [18], although it is not quite as efficient as the recent algorithm of Fredman and Tarjan [10]. For maximum flow in a network, Sleator and Tarjan's algorithm [25] depends on the dynamic tree data structure. Under similarity scaling achieves the same efficiency.

The main results of this paper are algorithms where scaling takes advantage of batching in nonnumeric algorithms. Weighted bipartite matching is discussed in Section 3. Table I shows that scaling is superior to the traditional Hungarian method, under similarity. Its efficiency results from combining the Hungarian method with the cardinality matching algorithm of Hopcroft and Karp. Scaling

works for all variants of matching, such as maximum weight matching, complete matching, cardinality- k matching and maximum weight maximum cardinality matching. Section 3 also gives nonscaling algorithms for matching that are efficient when $N = O(1)$. For instance in this case maximum weight matching is $O(n^{1/2} m)$.

The algorithms in the rest of the paper work by reducing the problem to matching. This is done for conceptual simplicity; direct approaches can be given and are preferable in practice.

Section 4 presents an algorithm for single-source shortest paths on a directed graph with possibly negative edge lengths. Table I shows that similarity implies scaling is superior.

Section 5 presents an algorithm for degree-constrained subgraphs on bipartite multigraphs. The two time bounds of Table I both apply to graphs; only the first applies if there are multiple edges. The best traditional algorithm works by treating the problem as a minimum cost network flow. Similarity implies scaling is superior.

Section 6 uses the results of Section 5 to solve the minimum cost flow problem on 0-1 networks. These networks were studied by Even and Tarjan [9] for the maximum value flow problem. The scaling bounds for the minimum cost problem are analogous to their bounds for the maximum value problem. Scaling is superior to the traditional algorithm, under similarity. The scaling algorithm works even in the presence of negative cycles. Section 7 summarizes some experimental results and gives conclusions.

2. EMULATING EFFICIENT DATA STRUCTURES

This section describes instances where the efficiency of priority queues and dynamic trees can be achieved by simple applications of scaling. It also indicates how in some models, the assumption of similarity is unnecessary.

2.1. Shortest Paths with Nonnegative Lengths

Consider a directed graph G with source vertex s and nonnegative integral edge lengths l_{ij} . The *shortest path problem* is to find d_i , the minimum length of a path from s to i , for each vertex i . This section sketches a scaling algorithm that runs in $O(m \log_{2+m/n} N)$ time. Under similarity this bound equals Johnson's [18] but is not quite as good as Fredman and Tarjan's [10]. The scaling algorithm is used in the matching algorithm of Section 3. It uses Dijkstra's algorithm [6], and also ideas due to Edmonds and Karp [8], Dial [5], and Wagner [27]. For a discussion of all this material see [26].

Define a "near-optimum" solution to the shortest paths problem to be a function d_i such that (i) $d_s = 0$; (ii) d_i dominates the edge lengths, that is for any edge ij , $d_i + l_{ij} \geq d_j$; (iii) each vertex i has a path from s of length between d_i and $d_i + m$. The shortest paths algorithm is based on a procedure that converts a near-optimum solution to optimum. To do this first compute modified edge lengths $l'_{ij} = d_i + l_{ij} - d_j$. Then compute shortest paths for the modified lengths using

Dijkstra's algorithm. The modified edge lengths do not change the shortest paths. The priority queue of Dijkstra's algorithm can be implemented by an array $Q(k)$, $0 \leq k \leq m$, where $Q(k)$ contains all vertices whose tentative distance from s is k . (Condition (iii) implies that all final distances from s are at most m , so this priority queue works correctly.) It is easy to see that the total time for this conversion procedure is $O(m)$.

The given shortest path problem is solved by scaling, as follows. If the given lengths are all m/n or less, use the conversion procedure directly, starting with the near-optimum solution $d_i = 0$. Otherwise scale the graph G to \bar{G} , a graph with the same vertices and edges as G and edge lengths $\lfloor l_{ij}/2 \rfloor$. Calculate d_i , the distances from s in \bar{G} , recursively. The values $2d_i$ are near-optimum for G . So use the conversion procedure to find the correct shortest paths. The total time is $O(m \log N)$, since there are $\lg N$ scaled graphs \bar{G} .

The scaling algorithm can be refined to achieve a time bound of $O(m \log_{2+m/n} N)$, which is similar to Johnson's bound, $O(m \log_{2+m/n} n)$. To do this scale by $s = 2 + m/n$ rather than 2, so that \bar{G} has edge lengths $\bar{l}_{ij} = \lfloor l_{ij}/s \rfloor$. Otherwise the algorithm is unchanged. (Note that in scaling up the distances sd_i are still near-optimum.)

It is an interesting exercise to see why the scaling algorithm fails in graphs with negative edge lengths but no negative cycles!

We close this section by noting that in models that account for the cost of arithmetic (e.g., when N is so large that multiprecision arithmetic must be used) the time bounds for Johnson's algorithm and scaling are identical. This results from the fact that the scaling algorithm can be modified to work with integers no larger than $2m$. Hence on a machine with words of $\lg n$ bits, each arithmetic operation in the scaling algorithm is $O(1)$ and the time bound remains $O(m \log_{2+m/n} N)$. However in Johnson's algorithm distances can be up to nN . This makes an arithmetic operation $O(\log_n N)$, so the total time becomes $O(m \log_{2+m/n} N)$. Similar results hold for times calculated in the logarithmic cost model [1].

The scaling algorithm that uses small numbers works as follows: For convenience we sketch an algorithm that scales by two and uses size $2n$ numbers; the algorithm that scales by $2 + m/n$ and uses size $2m$ numbers is similar. Consider the iterative version of scaling, that is, there are $k = \lfloor \lg N \rfloor + 1$ scales corresponding to successive bits in the binary numbers $b_1 \cdots b_k$. Scale 1 views numbers as their leading bit b_1 and computes distances d_i^1 for each vertex i ; in general scale s computes distances d_i^s . To derive the scale $s + 1$ problem, first replace the current edge lengths l_{ij} by $l'_{ij} = d_i^s + l_{ij} - d_j^s$; delete edges with $l'_{ij} \geq n$; finally scale lengths up to $2l'_{ij} + b_{s+1}$, where b_{s+1} is the $s + 1$ st bit of the (given) length of edge ij . The scale k shortest path tree is a shortest path tree for the original graph. (To prove this, first observe that in scale s an edge ij with original length $b_1 \cdots b_k$ has length $b_1 \cdots b_s + \sum_{t=1}^{s-1} 2^{s-t}(d_i^t - d_j^t)$. This implies that any deleted edge is too long to be on a shortest path. Further, the quantities $\sum_{t=1}^s 2^{s-t}d_i^t$ are valid distances for edge lengths $b_1 \cdots b_s$, as desired.) Each distance d_i^s is less than n (by the edge length transformation). Hence the algorithm always works with numbers $2n$ or smaller. Similar

transformations can be made for the other scaling algorithms of this paper, but are omitted.

2.2. Maximum Value Network Flow

Consider a directed graph G with source vertex s , sink vertex t , and integral edge capacities c_{ij} . The *maximum value network flow problem* is to find a flow of maximum value from s to t . For a discussion of this problem see [26]. This section sketches an algorithm with run time $O(nm \log N)$.

Define a “near-optimum” solution to be a flow function f whose value is within m of the maximum value. To convert a near-optimum flow f to a maximum flow, construct the residual network G^* for f . This network is defined by assigning edge ij a capacity $c_{ij} - f_{ij} + f_{ji}$. A maximum flow on G^* , when added to f , gives a maximum flow on G [26]. So find a maximum flow on G^* using Dinic’s algorithm [7], and add it to f to get the desired flow.

The time for this conversion procedure is $O(nm)$. To see this recall that Dinic’s algorithm has n phases, and each phase uses time $O(m + an)$, where a is the number of augmenting paths found. Each augmenting path increases the flow by at least one, since the flow is integral. A maximum flow on G^* has value at most m , by near-optimality. Hence the a ’s sum to at most m . The time bound follows easily.

The given flow problem is solved by scaling, as follows. If all capacities are m/n or less, use the above conversion procedure directly, starting with flow $f = 0$. Otherwise scale G to \bar{G} , the same graph with capacities $\lfloor c_{ij}/2 \rfloor$. Find a maximum flow f on \bar{G} recursively. The flow $2f$ is near-optimum for G , by the max flow–min cut theorem. So the conversion procedure finds a maximum flow on G . The total time for the algorithm is $O(nm \log N)$. The relative simplicity of this scaling algorithm should make it perform well in practice.

3. WEIGHTED MATCHING

Consider an undirected graph with integral edge weights w_{ij} . A *matching* is a set of edges with at most one edge incident to any vertex. A *free* vertex is not incident to any matched edge. A *complete matching* has no free vertices. The *weight* $w(M)$ of a matching M is the sum of all weights of edges of M . The *maximum weight matching problem* is to find a matching with largest possible weight; the *maximum complete matching problem* is defined similarly with respect to complete matchings. This section presents algorithms for these problems on bipartite graphs, that run in $O(n^{3/4}m \log N)$ time. Modifications are given for the useful case $N = O(1)$, that are more efficient and also apply to general graphs.

First consider maximum complete matching. For convenience assume that the input graph has a complete matching. Also assume that the edge weights w_{ij} are nonnegative integers. (If the given weights lie in an interval $[a, b]$, decrease all weights by a . This puts the weights in $[0, b - a]$, does not change the maximum complete matchings and only improves the time bound.)

The algorithm is based on the Hungarian method for weighted matching [20, 21]. We begin by sketching this method, concentrating on aspects that are relevant to the scaling algorithm. Complete developments are in [22, 24].

The Hungarian algorithm is a primal-dual algorithm in the sense of linear programming [4]. Each vertex i has a real-valued dual variable y_i . The dual variables are *dominating* if for every edge ij ,

$$y_i + y_j \geq w_{ij}.$$

An edge ij is *tight* if equality holds. The dual variables are *tight* with respect to a matching M if every edge $ij \in M$ is tight. The quantity $\sum_{i \in V} y_i$ is the *dual objective function*; the reader need not be familiar with duality theory, but he or she will still observe the importance of this function in what follows.

A complete matching M^* has maximum weight if it has a set of dominating and tight dual variables. To see this observe that completeness and tightness imply that the matching's weight is the dual objective,

$$w(M^*) = \sum_{i \in V} y_i; \quad (1)$$

dominance implies that any complete matching weighs at most the dual objective.

The Hungarian algorithm has input consisting of a bipartite graph and a set of dominating dual variables. The output is a complete matching with dominating tight dual variables. Hence the output is a maximum complete matching.

The Hungarian algorithm starts with the given duals and the empty matching. It repeatedly does a "Hungarian search" followed by an "augment step" until the matching is complete. A *Hungarian search* does zero or more "dual variable adjustments" until it finds a "wap." Each *dual variable adjustment* starts by computing a quantity δ . The duals of all free vertices are decreased by δ ; for some matched edges ij , y_i is increased by δ while y_j is decreased by δ . The adjustment keeps the duals dominating and tight. Eventually the search finds a *weighted augmenting path (wap)* P . This is a path joining two free vertices, whose edges are tight and alternately unmatched and matched. The *augment step* enlarges the matching by one edge, by adding the currently unmatched edges of P and removing the currently matched ones.

The timing analysis of the scaling algorithm depends on an integrality property of the Hungarian method: If all edge weights and all given duals are integers, then at any point in the algorithm any dual is a half-integer $y/2$, $y \in \mathbb{Z}$ [24, p. 267, Ex. 2]. An alternate formulation, useful in an actual implementation of the scaling algorithm, is that if all edge weights are even and all given duals are integers of the same parity, then at any point any dual is integral. (Both properties follow easily from the observation that in a Hungarian search, the duals of all vertices in search trees have the same parity.)

Another issue in the analysis of the scaling algorithm is the magnitude of the dual

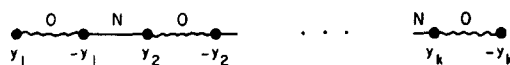


FIG. 1. Graph and dual values.

variables. The Hungarian method, and indeed any dual variable method, can produce dual values of size $\Omega(nN)$. To see this consider the graph of Fig. 1, a path of $n = 2k$ vertices with weights alternately 0 and N . The unique complete matching is shown. If y_i (as shown) is any corresponding set of dual variables, dominance and tightness imply $y_{i+1} \geq y_i + N$. Hence $y_k \geq y_1 + (k-1)N$, as desired.

The scaling algorithm relies on the fact that the Hungarian method does not make many dual variable adjustments if it starts with good initial values. More precisely let

$$D = \sum_{i \in V} y_i - w(M^*), \quad (2)$$

where y_i are the input dual variables and M^* is a maximum complete matching. (D is nonnegative by dominance.) Consider any given point in the Hungarian algorithm. Let the current matching have f free vertices; let Δ be the total of the quantities δ in all dual variable adjustments up to the given point.

LEMMA 3.1. $f\Delta \leq D$.

Proof. Consider the dual objective $\sum_{i \in V} y_i$ as dual variables are adjusted. A dual variable adjustment of δ decreases the objective by $g\delta$, where g is the number of free vertices when the adjustment is made. (To see this note that if vertex i is free, y_i decreases by δ ; if edge ij is matched, $y_i + y_j$ does not change.) Every augment decreases g by two, so every value of g is at least f . Thus the total decrease in the dual objective (up to the given point in the algorithm) is at least $f\Delta$. On the other hand the total decrease is at most D , since (1) holds when the Hungarian algorithm halts. This proves the lemma. ■

The scaling algorithm first solves the matching problem on a graph whose weights are half the original ones. This gives duals that have a small D value on the original graph, and makes the algorithm efficient.

The recursive procedure S below implements this strategy. S has input G , a bipartite graph with vertex sets V_1, V_2 . It returns with a maximum complete matching M and corresponding dual variables. It works by scaling G , with weights w_{ij} , to \bar{G} , with weights $\lfloor w_{ij}/2 \rfloor$. (G and \bar{G} have the same vertices and edges.) Step 3.2 below uses the cardinality matching algorithm of Hopcroft and Karp [17]. The version used accepts as input an arbitrary matching M , and transforms M to a maximum cardinality matching.

PROCEDURE $S(G)$. *Step 0.* If all weights w_{ij} are 0, return any complete matching M and all dual variables $y_i = 0$.

Step 1. Construct the graph \bar{G} . Call $S(\bar{G})$ recursively to find a maximum complete matching and corresponding dual variables y_i .

Step 2. Let M be the empty matching on G . For each vertex $i \in V_1$, set $y_i \leftarrow 2y_i + 1$; for each $j \in V_2$, set $y_j \leftarrow 2y_j$.

Step 3. Repeat the following steps until M is a complete matching on G :

Step 3.1. Do a Hungarian search to find a *wap*.

Step 3.2. Let T be the "tight" subgraph of G containing all edges that are tight for the current dual variables. Transform M to a maximum cardinality matching on T , by the Hopcroft-Karp cardinality matching algorithm.

Step 4. Return the complete matching M and the dual variables y_i .

LEMMA 3.2. Procedure S returns a maximum complete matching, with dominating tight dual variables.

Proof. Induct on the number of calls to S . The base case, when Step 0 is executed, is clearly correct. For the inductive step assume that the recursive call of Step 1 returns dominating duals on \bar{G} , that is for every edge ij , $\lfloor w_{ij}/2 \rfloor \leq y_i + y_j$. This implies that Step 2 computes dominating duals on G .

Observe that Step 3 maintains duals that are dominating and tight for M : Each Hungarian search (Step 3.1) maintains dominance and tightness, and Step 3.2 matches only tight edges.

Each iteration of Step 3 increases the number of matched edges by at least one, since a Hungarian search halts with a *wap*. So eventually a complete matching with corresponding duals is found. ■

To derive the time bound, focus on the execution of Step 3 in one recursive call to S . First observe that Lemma 3.1 holds for Step 3: Define D by (2), where y_i are the duals of Step 2; at any given point in Step 3 let the current matching have f free vertices; let Δ be the total of the quantities δ in all dual variable adjustments in all Hungarian searches (Step 3.1) up to this point. Lemma 3.1 holds for these quantities, by exactly the same proof.

The number of iterations of Step 3 satisfies two properties:

- (i) Step 3.1 is executed less than $2n^{1/2}$ times.
- (ii) Step 3.1 is executed at most $n^{1/4}$ times with $f \geq n^{3/4}$.

To show these properties, first note that

$$D \leq n/2.$$

For if M^* is a maximum complete matching, the matching of Step 1 shows that $w(M^*) \geq \sum_{i \in V} y_i - n/2$.

To prove property (i), count the number of executions of Step 3.1 with $f \geq n^{1/2}$

and the number with $f < n^{1/2}$. If $f \geq n^{1/2}$ then Lemma 3.1 shows that $\Delta \leq D/f \leq n^{1/2}/2$. Every execution of Step 3.1 adjusts the duals by some positive δ , since after any execution of Step 3.2 no augmenting path consists entirely of tight edges. Since δ is a half-integer, $\delta \geq \frac{1}{2}$. Thus the number of executions with $f \geq n^{1/2}$ is at most $n^{1/2}$. The number with $f < n^{1/2}$ is less than $n^{1/2}/2$, since each execution matches two or more free vertices.

Property (ii) is similar: If $f \geq n^{3/4}$ then $\Delta \leq D/f \leq n^{1/4}/2$, implying (ii).

Now compute the total time for Step 3 (in one recursive call). Step 3.1 can be implemented in $O(m)$ time; the data structures are essentially the same as for Dijkstra's algorithm, Section 2.1. The total time in Step 3.1 is $O(n^{1/2}m)$ by (i).

Step 3.2 uses the Hopcroft-Karp algorithm, so one execution uses $O(\min(n^{1/2}, a)m)$ time. Here a is the number of augmenting paths found (the $O(am)$ bound follows from inspecting the Hopcroft-Karp algorithm). By (ii), the time for all executions of Step 3.2 with $f \geq n^{3/4}$ is $O(n^{1/4} \cdot n^{1/2}m) = O(n^{3/4}m)$. The executions with $f < n^{3/4}$ find fewer than $n^{3/4}/2$ augmenting paths, and so use $O(n^{3/4}m)$ time.

Thus Step 3 is $O(n^{3/4}m)$. Step 2 is $O(n)$, and Step 0 is $O(n^{1/2}m)$. Since the number of recursive calls is $\lfloor \lg N \rfloor + 2$, the total time is $O(n^{3/4}m \log N)$. The following result summarizes the discussion.

THEOREM 3.1. *A maximum complete matching on a bipartite graph can be found in $O(n^{3/4}m \log N)$ time and $O(m)$ space.*

Regarding the tightness of this time bound, the discussion after Corollary 3.2 shows that one scale can use time $\Theta(n^{3/4}m)$.

The above derivation implicitly assumes that the dual values y_i do not grow too large, so that arithmetic operations are $O(1)$. Now we show this is true: the dual values have magnitude $O(nN)$.

One execution of Step 3 changes a dual value y_i by at most $\pm n/4$. (To see this note that y_i changes by at most $\pm \Delta$, by the Hungarian algorithm. Further, $\Delta \leq D/2 \leq n/4$, since any execution of Step 3.1 has $f \geq 2$.) So if a_i is defined as the largest magnitude of a dual value after the i th recursive invocation of procedure S , then $a_0 = 0$ and $a_{i+1} \leq 2a_i + n/4 + 1$, for $0 \leq i \leq \lfloor \lg N \rfloor$. It follows that a dual value is at most $(2^{\lfloor \lg N \rfloor + 1} - 1)(n/4 + 1) \leq N(n/2 + 2)$, as desired.

As noted previously, any dual variable method (including the Hungarian method) needs dual values of size $\Omega(nN)$ on some graphs. So a word size of $\lg N + \lg n + O(1)$ bits is necessary and sufficient. This is not unreasonable since the problem description needs $\max(\lg N, \lg n)$ bits. Thus at worst the algorithm uses double-word integers for the dual variables.

In an actual implementation some changes would be made to procedure S . Recursion would be changed to iteration. The programming nuisance of half-integers can be avoided in two ways. First, one can do the Hungarian search starting from the free vertices of only one of the vertex sets of the bipartite graph [22]. Alternatively a different scaling regime can be used: In scale s , instead of con-

sidering an edge weight with binary expansion $b_1 \cdots b_k$ to be $b_1 \cdots b_s$ it is considered to be $b_1 \cdots b_s 0$. Step 2 scales up all dual values by the assignment $y_k \leftarrow 2y_k + 1$. Hence all weights are even and all duals are odd. This ensures that throughout the algorithm all duals are integers.

The scaling algorithm extends to variants of the matching problem, including maximum weight matching, maximum weight maximum cardinality matching, and maximum weight cardinality- k matching. Here we discuss maximum weight matching (other variants are in Sect. 5). Even if a graph has a complete matching and all weights are positive, a maximum weight complete matching need not be a maximum weight matching. However maximum weight matching reduces to maximum complete matching as follows.

Let G be the graph for maximum weight matching. Construct a graph \hat{G} from two copies of G , say G_1, G_2 ; each vertex i in G , with copies i_1, i_2 in G_1, G_2 , has an edge $i_1 i_2$ in \hat{G} with weight 0. \hat{G} is bipartite if G is. A maximum complete matching on \hat{G} gives a maximum weight matching on $G_1 = G$.

COROLLARY 3.1. *A maximum weight matching on a bipartite graph can be found in $O(n^{3/4}m \log N)$ time and $O(m)$ space.*

For maximum weight matching all negative edges can be deleted from the graph. So in the above time bound N refers to the largest positive weight. (If the given weights lie in an interval $[a, b]$ then $N = b$; N cannot be reduced to $b - a$ as in complete matching.)

The time bound for maximum weight matching can be refined to $O(n_1^{3/4}m \log N)$, where $n_1 = |V_1|$. The algorithm must be modified as follows: Recall that graph \hat{G} is the input to procedure S . In Step 2, set $y_i \leftarrow 2y_i + 1$ for all vertices i in either copy of V_1 (that is, $i \in V_1(G_1) \cup V_1(G_2)$); set $y_i \leftarrow 2y_i$ for all i in either copy of V_2 . (Note that $V_1(G_1) \cup V_1(G_2), V_2(G_1) \cup V_2(G_2)$ is not a bipartition of \hat{G} .) In Step 3 (and Step 0), always maintain the same matching and duals on G_1 and G_2 .

The timing analysis depends on two observations. First, $D \leq 2n_1$, because of the modified Step 2. Second, the Hopcroft-Karp algorithm runs in time $O(n_1^{1/2}m)$. This follows from an analysis similar to [17]. The key point is that a set of augmenting paths contains mostly edges of the original graph G . More precisely, on the tight graph T of Step 3.2, a matching M that is identical on G_1 and G_2 has a set A of disjoint augmenting paths such that $M \oplus A$ is a maximum cardinality matching and further, each path in A has at most one edge $i_1 i_2$ joining G_1 to G_2 . The rest of the timing analysis follows the one given above and is left as an exercise.

Scaling is efficient on "small" edge weights like $N = n^{O(1)}$. If the weights are extremely small like $N = O(1)$, scaling can be dispensed with. This case is useful in practice, especially $N = 1$ or 2 . Now we sketch algorithms for this case, for maximum complete matching and maximum weight matching.

The algorithm for complete matching is: Start with M empty and all duals $y_i = N$; then execute Step 3 of procedure S . This gives the desired matching. An analysis similar to Theorem 3.1 shows the time is $O(N^{1/2}n^{3/4}m)$. This matches

COROLLARY 3.2. *If $N = O(1)$ a maximum complete matching on a bipartite graph can be found in $O(n^{3/4}m)$ time and a maximum weight matching can be found in $O(n^{1/2}m)$ time; both algorithms use $O(m)$ space.*

These algorithms extend to general graphs (see [13]).

The above bounds on the running times are tight. For maximum weight matching, see [9]. Here we consider maximum complete matching, and in particular the case where all edges weights are zero or one. We show that there are graphs of any density (i.e., any value of m between n and n^2) with run time $\Theta(n^{3/4}m)$.

Consider the family of graphs A_k illustrated in Fig. 2a. For each integer $i = k^2, \dots, 2k^2$, A_k contains a "module" consisting of a "base path" plus "side paths." The base path for i consists of $2i + 1$ edges, each of weight one. For each integer $j = 0, \dots, k$, every base path has two side paths of $2j + 1$ edges, with weights alternately zero and one; the last vertex of a side path for j is the j th vertex from one of the ends of the base path. Finally, to achieve arbitrary densities, A_k contains S , a bipartite graph on $2k^4$ vertices that has a complete matching plus any number of additional, arbitrary edges; each edge of S has weight one; in each module the first vertex of one side path for $j = k$ is joined by a weight zero edge to each vertex of S_1 , one of the two vertex sets of S . Observe that there are $\Theta(k^2)$ modules, each containing $\Theta(k^2)$ vertices and edges. Hence for A_k , $n = \Theta(k^4)$ and m can take on any value between n and n^2 (to within a constant factor) by adjusting the number of edges in S .

The algorithm starts with the duals shown in Fig. 2b, and the first execution of the Hopcroft–Karp algorithm gives the matching shown. (Assume duals are initialized in the best way, i.e., Step 2 of procedure S .) Then for $j = 0, \dots, k$, the j th Hungarian search makes a dual variable adjustment with $\delta = 1$; augmenting paths are found that use all side paths with $2j + 1$ edges; the Hopcroft–Karp algorithm finds these $\Theta(k^2)$ augmenting paths, one per phase. (Observe that each augmenting path has a different length, and so is found in a different phase of the Hopcroft–Karp algorithm.) The matching and duals after j iterations is illustrated in Fig. 2c.

To summarize, the number of iterations is $\Theta(k)$. Each execution of the Hopcroft–Karp algorithm finds $\Theta(k^2)$ augmenting paths, one per phase. Each phase uses $\Theta(m)$ time, since the number of tight edges is always $\Theta(m)$. Hence the total time is $\Theta(k^3m) = \Theta(n^{3/4}m)$.

The problem we have discussed might be dubbed *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. It would be of interest to improve the time bound for red–green matching.

4. SHORTEST PATHS WITH ARBITRARY LENGTHS

This section shows that in a directed graph where edge lengths are arbitrary positive or negative integers, the single-source shortest path problem (defined in Sect. 2.1) can be solved in $O(n^{3/4}m \log N)$ time.

The approach is based on Edmonds and Karp's transformation to get non-negative edge lengths [8; also see 26]. As in Section 2.1 this amounts to finding a function d_i that dominates the edge lengths, that is for any edge ij , $d_i + l_{ij} \geq d_j$. A dominating function is derived from the matching dual variables as follows.

Given a directed graph G , construct a corresponding bipartite graph G^* . A vertex i of G corresponds to two vertices i_1, i_2 in G^* . An edge ij of G corresponds to an edge $i_1 j_2$ of G^* with weight $-l_{ij}$. In addition G^* has an edge $i_1 i_2$ of weight 0 for each i .

There is a one-to-one correspondence between the complete matchings on G^* and the partitions of the vertices of G into cycles and single vertices. Specifically, a set of matched edges of the form $i_1 j_2, j_1 k_2, \dots, r_1 s_2, s_1 i_2$ corresponds to the cycle i, j, k, \dots, r, s, i ; a matched edge $i_1 i_2$ corresponds to the vertex i . So a maximum complete matching on G^* has positive weight when G has a negative cycle, and it has zero weight when G has no negative cycles.

Now without loss of generality assume that every vertex in G is reachable from the source s . The shortest path problem on G has a solution if and only if G has no negative cycles. So assume a maximum complete matching on G^* has weight zero. Observe that in a corresponding set of dual variables, $y_{i_1} = -y_{i_2}$ for every vertex i of G . This is clear if $i_1 i_2$ is matched. If $i_1 i_2$ is unmatched then there is an alternating cycle of the form $i_1 j_2, j_2 j_1, j_1 k_2, \dots, s_2 s_1, s_1 i_2, i_2 i_1$. The matched edges of the cycle have total weight zero, since the unmatched edges do. Hence dominance and tightness imply $y_{i_1} = -y_{i_2}$.

For each vertex i of G let $d_i = y_{i_1}$. Then for any edge ij of G , $-l_{ij} \leq y_{i_1} + y_{j_2} = d_i - d_j$. Thus the function d_i dominates.

This justifies the following shortest path algorithm: Construct G^* and find a maximum complete matching with corresponding duals. (If the matching has positive weight, stop; the problem is ill-defined.) Use the dominating function derived from the duals to transform edge lengths so they are nonnegative (thus the transformed length of ij is $l'_{ij} = d_i + l_{ij} - d_j$). Finally run Dijkstra's algorithm on the graph with transformed edge lengths.

THEOREM 4.1. *In a directed graph with arbitrary integral edge lengths, the single-source shortest path problem can be solved in $O(n^{3/4}m \log N)$ time and $O(m)$ space.*

This algorithm solves the shortest path problem for $O(n^{3/4})$ sources in the same time.

5. DEGREE-CONSTRAINED SUBGRAPHS

Consider an undirected multigraph where each vertex i has two associated integers l_i, u_i . In a *degree-constrained subgraph* (DCS) each vertex i has degree d_i , $l_i \leq d_i \leq u_i$. In a *complete* DCS each degree achieves its upper bound, $d_i = u_i$. This section presents algorithms for the maximum complete DCS and maximum weight DCS problems. (These problems generalize matching and are defined analogously.) The time is $O(U^{3/4}m \log N)$ on multigraphs; it is also $O(U^{1/2}n^{1/3}m \log N)$ on graphs. Here $U = \sum_{i \in V} u_i$, and m counts each edge according to its multiplicity. (The time bounds are oriented toward graphs of small multiplicity.)

We reduce the complete DCS problem to matching. Let G be the given multigraph. Define a graph G' as follows. A vertex i in G corresponds to a *vertex substitute* $K_{\Delta, d}$ in G' . Here d is the degree of i in G ; Δ is the desired "deficiency" of i , that is if i has upper degree bound $u \equiv u_i$ then $\Delta = d - u$; $K_{\Delta, d}$ is a complete bipartite graph consisting of Δ internal vertices in one vertex set and d external vertices in the other. An edge ij in G corresponds to an edge joining an external vertex in i 's substitute to one in j 's; each external vertex of each substitute in G' is on exactly one copy of an edge of G . In G' copies of edges of G have the same weight as in G . An edge in any substitute $K_{\Delta, d}$ has weight $N^* = 2^{\lceil \lg N \rceil + 1} - 1$. A maximum complete DCS on G corresponds to a maximum complete matching on G' . Hence DCS reduces to matching.

The graph G' can have $\Omega(nm)$ edges. For efficiency we do not work directly with G' , but instead simulate the scaling algorithm S on G' . To simulate Step 3.1 (the Hungarian search) we use a "sparse substitute" graph, employing the technique of [11]. For Step 3.2 (cardinality matching) we use another sparse graph. For the simulations to work correctly we maintain this additional invariant: the matching on G' always covers every internal vertex of every vertex substitute. The details of the two sparse graphs are as follows.

Consider a matching M on G' , and a vertex i of G whose substitute has $w \leq u$ external vertices matched on edges of G . The *sparse substitute* for a vertex i (with respect to M) is illustrated in Fig. 3. Of the $d - w$ external vertices that are not on matched edges of G , $u - w$ vertices (chosen arbitrarily) are free; each of the other Δ external vertices is matched to a vertex of the sparse substitute. The sparse sub-

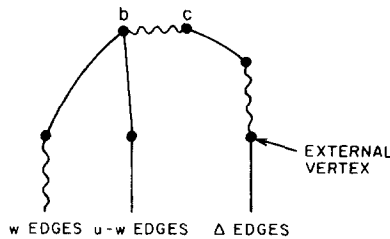


FIG. 3. Sparse substitute for i .

stitute contains two vertices b, c joined by a matched edge. The other edges of the sparse substitute are as illustrated (each edge of Fig. 3, except bc , stands for $w, u-w$, or Δ edges). Each edge in the sparse substitute has weight N^* . The *sparse substitute graph* G_k (with respect to a matching M) is the graph G' except that every vertex has a sparse substitute.

Graph G_k has these properties (assuming the invariant that the matching on G' covers every internal vertex of every vertex substitute):

- (i) G_k has $O(m)$ edges.
- (ii) There is a one-to-one correspondence between augmenting paths in G_k , and augmenting paths in G' that pass through each substitute at most once.
- (iii) Dominating tight dual variables on G' give dominating tight duals on G_k , and vice versa.

Property (i) holds because a sparse substitute has $u + 2\Delta + 1 \leq 2d + 1$ edges. Property (ii) follows from Fig. 3. (The term "augmenting path" is topological and ignores edge weights, as contrasted with *wap*. Observe that the edge bc allows an augmenting path in G_k to pass through a substitute only once. An augmenting path in G' need only pass through a substitute once.)

Property (iii) is proved as follows. The precise statement is that for a given substitute, the dual values of the external vertices can be taken the same in G' and G_k ; the remaining duals are then implied. Given duals on G' , there is a corresponding set of duals on G_k , since every edge in the sparse substitute has a corresponding edge in the vertex substitute. (Edge bc corresponds to any matched edge of the vertex substitute.) Conversely assume dual values on G_k . In a given sparse substitute let b and c have dual values y and $N^* - y$, respectively. Without loss of generality the Δ other matched edges in the substitute have dual values y for the internal vertex and $N^* - y$ for the external vertex. (Dominance implies that the internal dual is at least y , so the external dual is at most $N^* - y$; increasing the external dual to exactly $N^* - y$ preserves dominance on the external edge.) Hence in the corresponding vertex substitute of G' all external vertices can be assigned their dual values in G_k and all internal vertices can be assigned the dual value y .

Step 3.1, the Hungarian search on G' , is simulated as follows. Assign dual variables to G_k , using property (iii). Then do a Hungarian search on G_k . The search gives new dual variables on G' that have a tight *wap*, by (ii)–(iii), as desired.

The second sparse graph, T' , models the tight graph T of G' used in Step 3.2. T' is defined as follows. Consider a typical vertex substitute $K_{\Delta,d}$ of G' . Each of the Δ internal vertices has the same dual value y . (This follows from tightness and dominance, or alternatively from the proof of (iii).) So the tight edges of $K_{\Delta,d}$ form a complete bipartite graph $K_{\Delta,\delta}$, for some δ with $\Delta \leq \delta \leq d$. Construct the multigraph T' from T as follows. For each vertex substitute of T , contract the tight subgraph $K_{\Delta,\delta}$ and give the resulting vertex a degree constraint of $\delta - \Delta$; also contract the remaining $d - \delta$ external vertices (to a single vertex) and give the resulting vertex a degree constraint of $d - \delta$. It is easy to see that T' has these properties:

- (i) T' has $2n$ vertices, at most m edges, and the same U value as G .
- (ii) There is a one-to-one correspondence between DCS's on T' and matchings on T that cover all internal substitute vertices.

To simulate Step 3.2 on T , construct T' and find a maximum cardinality DCS on it. This gives the desired maximum cardinality matching on T , by (ii).

To summarize, the following scaling algorithm finds a maximum complete DCS on a bipartite multigraph G . Construct the graph G' (without explicitly constructing the vertex substitutes $K_{A,a}$). Then simulate $S(G')$: In Step 0 find the complete matching on G' by solving the complete DCS cardinality problem on G . In Step 2 match the internal vertices of each substitute of G' to A (arbitrary) external vertices. In Step 3.1 construct the sparse substitute graph G_k and do a Hungarian search on it. In Step 3.2 construct the multigraph T' and find a maximum cardinality DCS on it. Convert this to a matching on G' .

The timing analysis is similar to the matching algorithm. Lemma 3.1 shows that $fA \leq D$, where these quantities are calculated on G' . Observe that $D \leq U/2$: A complete matching on G' consists of $U/2$ edges of G plus edges in vertex substitutes. The substitute edges are tight after Step 2 by the definition of N^* . This shows $D \leq U/2$.

These inequalities imply that at most $U^{1/4}$ iterations of Step 3 have $f \geq U^{3/4}$. Clearly less than $U^{3/4}/2$ augments are done with $f < U^{3/4}$. Since Step 3.2 is $O(\min(U^{1/2}, a)m)$ [9], the total time is $O(U^{3/4}m \log N)$.

An alternative bound holds if G has no parallel edges. In this case Step 3.2 is $O(n^{2/3}m)$ [9]. A similar calculation shows the total time is $O(U^{1/2}n^{1/3}m \log N)$. This improves the first bound when $U = \Omega(n^{4/3})$.

THEOREM 5.1. *A maximum complete DCS on a bipartite multigraph can be found in $O(U^{3/4}m \log N)$ time. The time is also $O(U^{1/2}n^{1/3}m \log N)$ for graphs. The space is $O(m)$.*

Other time bounds can be derived for multigraphs (and are sometimes superior). For instance if Step 3.2 is implemented with Karzanov's algorithm [19] the time is $O(n(Unm)^{1/2} \log N)$.

Theorem 5.1 implies that a maximum weight cardinality- k matching can be found in $O(n^{3/4}m \log N)$ time, since the matching problem can be converted to a complete DCS problem by adding just two vertices. The same bound holds for maximum weight maximum cardinality matching.

Next consider the maximum weight DCS problem. Let G be a bipartite multigraph with lower bounds l_i and upper bounds u_i . Construct a bipartite multigraph \hat{G} by making two copies of G , say G_1 and G_2 ; for each vertex i , with copies i_1 and i_2 , add $u_i - l_i$ copies of edge $i_1 i_2$ with weight 0. A maximum complete DCS on \hat{G} gives a maximum weight DCS on G . So the complete DCS algorithm, executed on \hat{G} , gives an algorithm for maximum weight DCS that has the multigraph time bound of Theorem 5.1.

The graph time bound of Theorem 5.1 also holds. This relies on the fact that the bound for maximum cardinality DCS on a graph extends to multigraphs with

“limited parallelism”: Recall that the cardinality DCS algorithm works by finding shortest length augmenting paths (*saps*). The argument of [9] generalizes to prove the following.

PRINCIPLE 5.1. *Let the maximum cardinality DCS algorithm be executed on a bipartite multigraph. Suppose that for any k , when the algorithm finds saps of length k there are $\Omega(k)$ levels where no two saps contain parallel edges. Then the algorithm runs in $O(n^{2/3}m)$ time.*

We analyze Step 3.2 of the scaling algorithm, executed on the multigraph \hat{G} , by using the principle as follows. Assume as in Section 3 that the scaling algorithm maintains the same matching and duals on G_1 and G_2 , the two isomorphic halves of \hat{G} . So T' , the sparse multigraph of Step 3.2, is composed of T_1 and T_2 , two isomorphic graphs, joined by a number of multiedges i_1i_2 . A *sap* contains at most one “joining” edge i_1i_2 ; further, all *saps* contain their joining edge at the same level. Thus Principle 5.1 applies to show that Step 3.2 is $O(n^{2/3}m)$. This in turn implies the graph bound of Theorem 5.1.

COROLLARY 5.1. *A maximum weight DCS on a bipartite multigraph can be found in $O(U^{3/4}m \log N)$ time. The time is also $O(U^{1/2}n^{1/3}m \log N)$ for graphs. The space is $O(m)$.*

Corollary 5.1 remains valid if we take $U = \min(\sum_{i \in V_1} u_i, \sum_{i \in V_2} u_i)$. (This depends on the analogous fact for maximum weight matching shown in Sect. 3.) So for instance if all vertices in V_1 have degree constraint one, the time bound for maximum weight DCS is the same as matching.

For very small weights Corollary 3.2 has the following analog.

COROLLARY 5.2. *If $N = O(1)$ a maximum weight DCS on a bipartite multigraph can be found in $O(U^{1/2}m)$ time. The time is also $O(n^{2/3}m)$ on a graph. The space is $O(m)$.*

6. 0-1 NETWORK FLOW

Consider a directed graph G with source vertex s , sink t , nonnegative integral edge capacities c_{ij} , and integral costs a_{ij} . The *minimum cost flow problem* is to find a flow with smallest possible cost, subject to the constraint that it has a given flow value v .

A 0-1 *network* has all capacities one. Parallel edges are allowed (and need not have equal cost.) A *type 1* network has no parallel edges, except for edges directed from s or to t . A *type 1+* network has no parallel edges at all. The *mindegree* of a vertex is the minimum of its indegree and its outdegree. In a *type 2* network each vertex (other than s or t) has mindegree exactly one. (These important special cases were introduced by Even and Tarjan [9]. Their type 1 corresponds to our type 1+. Our type 1 is more general, but their results still apply by Principle 5.1. Our type 1

and type 1+ are both natural classes of networks: Type 1 corresponds to DCS problems. Type 1 networks have flow value at most m ; type 1+ networks have value at most n .)

The 0-1 minimum cost flow problem reduces to maximum complete DCS, as follows. Given a network G , construct a bipartite multigraph G^* (similar to Sect. 4): A vertex i of G corresponds to two vertices i_1, i_2 of G^* ; G^* has an edge $i_1 i_2$ of weight 0 and multiplicity $\text{mindegree}(i)$. An edge jk of G corresponds to an edge $j_1 k_2$ of G^* of weight $-a_{jk}$. Finally the degree constraints on G^* are $u_{i_1} = u_{i_2} = \text{mindegree}(i)$ for $i \neq s, t$; $u_{s_2} = \text{mindegree}(s)$, $u_{s_1} = u_{s_2} + v$, $u_{t_1} = \text{mindegree}(t)$, $u_{t_2} = u_{t_1} + v$.

A flow of value v on G corresponds to a complete DCS on G^* ; minimum cost corresponds to maximum weight. So the flow problem can be solved using the algorithm of Section 5 on G^* . Note that for a 0-1 network, G^* has $2n$ vertices, $O(m)$ edges, and $U = O(m)$; for type 2, $U = O(n)$. Although G^* is a multigraph, if G is type 1 then the graph bound of Theorem 5.1 applies, by Principle 5.1.

THEOREM 6.1. *A minimum cost flow on a 0-1 network can be found in $O(m^{7/4} \log N)$ time. The time is also $O(n^{1/3} m^{3/2} \log N)$ for type 1 networks and $O(n^{3/4} m \log N)$ for type 2. The space is $O(m)$.*

Theorem 6.1 is valid even on networks with negative cycles. The same bounds apply to finding a minimum cost flow of maximum value, since the desired value v can be found using Dinic's algorithm [9].

The Edmonds-Karp algorithm can be superior to the scaling algorithm on type 1+ networks with no negative cycles. It runs in $O(n(m + n \log n))$ time [10], and if $m = \Omega(n^{4/3})$ this is faster than Theorem 6.1. However using an approach similar to Dijkstra's algorithm (Sect. 2.1), the scaling bound improves to $O(nm \log_{2+m/n} N)$.

7. CONCLUSIONS

Scaling leads to efficient algorithms for network problems. Under the assumption of similarity these algorithms are asymptotically faster than previous ones. In practice scaling algorithms perform well because of their simplicity. This was confirmed in experiments on an implementation of the weighted matching algorithm in PASCAL on a VAX 11/780. About 1000 randomly generated networks were tested, with between 100 and 1500 vertices, various densities and N values. For networks with $N = n$ scaling usually beat the Hungarian method by the theoretically predicted factor of $n^{1/4}$ or more. In order for the Hungarian method to be faster than scaling (because of large N) the weights had to be so large that machine overflow usually occurred in computing the weight of the maximum matching [2].

Scaling is a general method. Applications to computational geometry are given in [15]. Network problems on general graphs are discussed in [13]; matroid intersection problems are in [14].

ACKNOWLEDGMENTS

The author thanks Dr. Robert Tarjan for insightful advice that resulted in new algorithms and better presentation, and Dr. Ann Bateson for many perceptive comments.

REFERENCES

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, Mass., 1974.
2. C. A. BATESON, "Performance Comparison of Two Algorithms for Weighted Bipartite Matchings," M. S. thesis, Department of Computer Science, University of Colorado, Boulder, Colo., 1985.
3. R. E. BELLMAN, On a routing problem, *Quart. Appl. Math.* **16** (1958), 87-90.
4. G. B. DANTZIG, "Linear Programming and Extensions," Princeton Univ. Press, Princeton, N. J., 1963.
5. R. B. DIAL, Algorithm 360: Shortest path forest with topological ordering, *Comm. ACM* **12** (1969), 632-3.
6. E. DIJKSTRA, A note on two problems in connexion with graphs, *Numer. Math.* **1**, (1959), 269-271.
7. E. A. DINIC, Algorithm for solution of a problem of maximum flow in a network with power estimation, *Soviet Math. Dokl.* **11** No. 5, (1970), 1277-1280.
8. J. EDMONDS AND R. M. KARP, Theoretical improvements in algorithmic efficiency for network flow problems, *J. Assoc. Comput. Mach.* **19**, No. 2 (1972), 248-264.
9. S. EVEN AND R. E. TARJAN, Network flow and testing graph connectivity, *SIAM J. Comput.* **4** No. 4 (1975), 507-518.
10. M. L. FREDMAN AND R. E. TARJAN, Fibonacci heaps and their uses in improved network optimization algorithms, in "Proc. 25th Annual Symp. on Found. of Comput. Sci., 1984," pp. 338-346.
11. H. N. GABOW, An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems, in "Proc. Fifteenth Annual ACM Sympos on Theory of Computing, 1983," pp. 448-456.
12. H. N. GABOW, Efficient graph algorithms using adjacency matrices, manuscript, 1985.
13. H. N. GABOW, A scaling algorithm for weighted matching on general graphs, in "Proc. 26th Annual Symp. on Found. of Comput. Sci., 1985," to appear.
14. H. N. GABOW, Scaling algorithms for matroid intersection problems, manuscript, 1985.
15. H. N. GABOW, J. L. BENTLEY, AND R. E. TARJAN, Scaling and related techniques for geometry problems", in "Proc. 16th Annual ACM Symp. on Theory of Computing, 1984," pp. 135-143.
16. H. N. GABOW, Z. GALIL, AND T. H. SPENCER, Efficient implementations of graph algorithms using contraction, in "Proc. 25th Annual Symp. on Found. of Comput. Sci., 1984," pp. 347-357.
17. J. HOPCROFT AND R. KARP, An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, *SIAM J. Comput.* **2**, No. 4, (1973), 225-231.
18. D. B. JOHNSON, Efficient algorithms for shortest paths in sparse networks, *J. Assoc. Comput. Mach.* **24**, No. 1 (1977), 1-13.
19. A. V. KARZANOV, Determining the maximal flow in a network by the method of preflows, *Soviet Math. Dokl.* **15** (1974), 434-437.
20. H. W. KUHN, The Hungarian method for the assignment problem, *Naval Res. Logist. Quart.* **2** (1955), 83-97.
21. H. W. KUHN, Variants of the Hungarian method for assignment problems, *Naval Res. Logist. Quart.* **3** (1956), 253-258.
22. E. L. LAWLER, "Combinatorial Optimization: Networks and Matroids," Holt, Rinehart, & Winston, New York, 1976.
23. S. MICALI AND V. V. VAZIRANI, An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs, in "Proc. 21st Annual Symp. on Found. of Comput. Sci., 1980," pp. 17-27.

24. C. H. PAPADIMITRIOU AND K. STEIGLITZ, "Combinatorial Optimization: Algorithms and Complexity," Prentice-Hall, Englewood Cliffs, N. J. 1982.
25. D. D. K. SLEATOR AND R. E. TARJAN, A data structure for dynamic trees, *J. Comput. System Sci.* **26**, (1983), 362–391.
26. R. E. TARJAN, "Data Structures and Network Algorithms," Soc. Indus. Appl. Math., Philadelphia, P., 1983.
27. R. A. WAGNER, A shortest path algorithm for edge-sparse graphs, *J. Assoc. Comput. Math.* **23**, No. 1 (1976), 50–57.