SCALING ALGORITHMS FOR NETWORK PROBLEMS

by

Harold N. Gabow*

CU-CS-266-84 April, 1984

* University of Colorado at Boulder, Department of Computer Science, Campus Box 430, Boulder, Colorado, USA.

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

Scaling Algorithms for Network Problems

Harold N. Gabow *

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

Abstract.

Efficient algorithms for network problems that work by scaling the numeric parameters are given. Scaling takes advantage of efficient nonnumeric algorithms such as the Hopcroft-Karp matching algorithm. Let n, m and N denote the number of vertices, number of edges, and largest numeric parameter of the network, respectively; assume all numeric parameters are integers. A scaling algorithm for maximum weight matching on a bipartite graph runs in $O(n^{\frac{3}{4}} m \log N)$ time. This can improve the traditional Hungarian method which runs in $O(n\ m \log n)$ time. This result 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 (Edmonds and Karp). Scaling also gives simple algorithms that match the best time bounds (when $\log N = O(\log n)$) for shortest paths on a directed graph with nonnegative lengths (Dijkstra's algorithm) and maximum value network flow (Sleator and Tarjan).

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

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

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. Often they allow the numeric network problem to be solved by the same approach, and to within a log n factor in time, as the non-numeric graph analog of the problem. An example is the shortest path problem, solved by Dijkstra's algorithm on networks and breadth-first search on graphs. Unfortunately this is not always so. For example a maximum cardinality matching can be found in $O(n^{\frac{1}{2}}m)$ time (Hopcroft and Karp, 1973; Micali and Vazirani, 1980) whereas the best bound for weighted matching is $O(n\ m \log n)$ (Papadimitriou and Steiglitz, 1982; Galil, et.al., 1982). The complexity gap here and elsewhere results from the fact that the nonnumeric algorithm finds objects simultaneously; conversely, numeric parameters force searches to be done sequentially.

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 simultaneity in numeric problems. Edmonds and Karp (1972) used scaling for the minimum cost network flow problem. However since then the method has been largely ignored.

Scaling works as follows. Given a network problem, divide all numbers by two (i.e., a number l becomes $\left\lfloor \frac{l}{2} \right\rfloor$) 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.

This method can be stated iteratively. View each number as its l bit binary expansion $b_1 \cdots b_l$, where $l = \lfloor lg\ N \rfloor + 1$. First solve the problem on the network where all parameters are 0. Then for $s = 0, \ldots, l-1$, transform the solution

for the network with parameters $b_1 \cdots b_s$ to a solution for parameters $b_1 \cdots b_{s+1}$.

Scaling relies on two assumptions. *Integrality* assures the applicability of the method. It requires that all given numbers 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 be in $[0,1)$, and view them as binary numbers, $0.b_1b_2 \cdots$. Scaling successively computes solutions to the network with parameters 0, $.b_1$, $.b_1b_2$, etc. The accuracy of each solution can be bounded. So scaling can find a solution with any desired accuracy. [1]

The second assumption, *similarity*, concerns the efficiency of the method. It requires that the numbers of the problem be polynomially bounded, i.e., if N is the largest number and n is the number of vertices, then $N = O(n^k)$ for some constant k. Scaling time bounds typically have 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 matching scaling is faster for $N = n^{O(n^{\frac{1}{4}})}$.

We suspect that similarity holds in most practical cases: N is not many orders of magnitude above n. Actually if similarity does not hold, then traditional algorithms must use a word size of $lg\ N$ bits rather than $lg\ n$ in order for the traditional time bound to be valid (see Section 2.1.) This opens the door for

[1] This approach is due to Dr. Robert E. Tarjan.

new approaches, e.g., if $N \geq 2^n$ then bit vector types of algorithms are possible (e.g., on a bit vector machine with words of n bits, the Hopcroft-Karp algorithm used in the scaling approach is $O(n^{\frac{3}{2}} \log n + m)$ rather than $O(n^{\frac{1}{2}} m)$ (Gabow, 1984a).)

Figure 1 lists a number of network problems, the time for the best known algorithm [2] and the time for the scaling algorithm. The first two entries for scaling result from the fact that n integers can be radix sorted in $O(n \log_n N)$ time. (Note that radix sorting itself works by scaling.) Under similarity, scaling is faster in both problems.

The remaining entries in the table are covered in this paper. The next two entries are presented in Section 2. They are instances where scaling takes the place of efficient priority queues and other data structures. For the shortest path problem on directed graphs with nonnegative edge lengths, Dijkstra's algorithm depends on the proper choice of priority queue for its efficiency. For maximum value flow in a network, Sleator and Tarjan's algorithm depends on the dynamic tree data structure. In both cases scaling achieves the same efficiency, under similarity.

The main results of this paper are algorithms where scaling takes advantage of simultaneity in efficient nonnumeric algorithms. Weighted bipartite matching is discussed in Section 3. 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,

---

[2] Fredman and Tarjan (1984) have recently developed the F-heap data structure, which imporves the logarithmic terms in most of these bounds.

when $N = 0(1)$. These algorithms run in $0(n^{\frac{3}{4}}m)$ time for maximum complete matching and $0(n^{\frac{1}{2}}m)$ time for maximum weight matching; they work for nonbipartite graphs.

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 would be preferable in practice.

Section 4 presents an algorithm for the next entry, single-source shortest paths on a directed graph with possibly negative edge lengths. Similarity implies scaling is superior.

Section 5 presents an algorithm for the degree-constrained subgraph problem on bipartite multigraphs. The two time bounds of Figure 1 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 (1975) for the maximum value flow problem. The scaling bounds for the cost problem are analogous to their bounds for the value problem. Scaling is superior to the traditional algorithm, under similarity. The scaling algorithm works even in the presence of negative cycles.

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 an algorithm that runs in $O(m \log_{2+\frac{m}{n}} N)$ time.

The algorithm is based on a procedure that converts a "near-optimum" solution to optimum. It uses ideas due to Edmonds and Karp (1972), and also Dial (1969) and Wagner (1976). A "near-optimum" solution is a function d_i such that (i) d_i *dominates* the edge lengths, i.e., for any edge ij, $d_i+l_{ij} \geq d_j$; (ii) each vertex i has a path from s of length between d_i and d_i+m. The algorithm computes new edge lengths $l'_{ij} = d_i+l_{ij}-d_j$. These edge lengths do not change the shortest paths. The algorithm computes shortest paths for these lengths using Dijkstra's algorithm. The priority queue of Dijkstra's algorithm is 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 (ii) implies that all final distances from s are at most m). It is easy to see the total time for the algorithm is $O(m)$.

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

The scaling algorithm can be refined to achieve a bound similar to Johnson's (1977), $O(m \log_{2+\frac{m}{n}} N)$. To do this construct $\overline{G}$ by dividing lengths by

$s = 2+\frac{m}{n}$, i.e., $\bar{l}_{ij} = \left\lfloor \frac{l_{ij}}{s} \right\rfloor$.

It is an interesting exercise to see why this algorithm fails in graphs with negative 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 so it always works on integers no larger than $2n$. 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 $O(m \log_{2+\frac{m}{n}} N)$. However in Johnson's algorithm distances can be up to nN, so an arithmetic operation is $O(log_n N)$. This makes the time bound identical to scaling. Similar results hold for the logarithmic cost model (Aho et.al., 1974).

The scaling algorithm that uses size n numbers works as follows. Consider the iterative version of scaling, i.e., there are $l = \lfloor lg\ N \rfloor + 1$ scales corresponding to successive bits in the binary numbers $b_1 \cdots b_l$. (Scaling by two is discussed for simplicity; scaling by $2+\frac{m}{n}$ is similar). In scale 1 view numbers as their leading bit b_1 and compute 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 l shortest path tree is a shortest path tree for the original graph. (This follows from the inductive assertion that in scale s, an edge ij with original length $b_1 \cdots b_l$ has length $b_1 \cdots b_s + \sum_{t=1}^{s-1} 2^{s-t}(d_i^t - d_j^t)$. This implies that the deleted edges are irrelevant. Further, the quantities $\sum_{t=1}^{s} 2^{s-t} d_i^t$ are valid dis-

tances for the edge lengths $b_1 \cdots b_s$, and the algorithm is correct.) Each distance d_i^s is less than n (by the edge length transformation). Here the algorithm always works with numbers $2n$ or smaller.

Similar transformations can be done 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. This section sketches an algorithm with run time $O(n\ m\ \log N)$.

Again the algorithm is based on a procedure to convert a near-optimum solution to optimum. A near-optimum solution is a flow function f whose value is within m of the maximum value. To find the maximum, construct the residual network G^* for f, i.e., edge ij has capacity $c_{ij}-f_{ij}+f_{ji}$ (Tarjan, 1983). A maximum flow on G^*, when added to f, gives a maximum flow on G; hence the value of a maximum flow on G^* is at most m. Find a maximum flow on G^* using Dinic's algorithm (1970). The time is $O(nm)$. (Dinic's algorithm has n phases. Each phase requires time $O(m+an)$, where a is the number of augmenting paths found. Since all a's sum to at most m the bound follows).

The given network flow problem is solved by scaling. If all capacities are bounded by $\frac{m}{n}$ use the above procedure directly (start with $f = 0$). Otherwise scale G to $\overline{G}$, the same graph with capacities $\left\lfloor \frac{c_{ij}}{2} \right\rfloor$. Find a maximum flow f on $\overline{G}$ recursively. The flow $2f$ is near-optimum for G, by the max flow-min cut theorem. So the above procedure finds the maximum flow on G. The total time for the algorithm is $O(n\ m\ \log N)$. The relative simplicity of this scaling algo-

rithm should make it perform well in practice.

3. Weighted matching.

Consider an undirected bipartite graph with integral edge weights w_{ij}. A *matching* is a set of edges that has 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 *maximum weight matching problem* is to find a matching whose edges have largest possible total weight; the *maximum complete matching problem* is defined similarly with respect to complete matchings. This section presents algorithms for these problems that run in $O(n^{\frac{3}{4}}m \log N)$ time. Modifications of the algorithm, that also apply to nonbipartite graphs, are given for the case of "small" edge weights (e.g., $N = O(1)$).

Consider first 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. For if the given weights are in the interval $[a,b]$, adding $-a$ to all weights puts them in $[0,b-a]$, and does not change the maximum complete matching. Our time bound for complete matching is oriented toward the case of nonnegative weights in $[0,N]$; for weights in $[a,b]$ replace N by $b-a$.

The algorithm is based on the Hungarian method for weighted matching (Kuhn, 1955, 1956). We begin by sketching a variant of this method that works with arbitrary given dual values. Our treatment concentrates on the aspects of the method that are relevant to the scaling algorithm. Complete developments are in (Lawler, 1976; Papadimitriou and Steiglitz, 1982).

The Hungarian algorithm is a primal-dual algorithm in the sense of linear programming (Dantzig, 1963). 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 it satisfies this inequality with equality. The dual variables are *tight* if every matched edge ij is tight.

If a complete matching M^* has a set of dual variables that are dominating and tight then M^* has maximum weight. For tightness and completeness imply

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

and dominance implies that any complete matching weighs at most this quantity.

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

The Hungarian algorithm starts with the given dual variables and the empty matching, and repeatedly adjusts the dual variables and augments the matching. More precisely the algorithm is organized as a sequence of *searches*. Each search (i) adjusts the dual variables (zero or more times) and (ii) eventually finds a *(weighted) augmenting path (wap)*. Each dual variable adjustment starts by computing a quantity δ. Then the dual variables of all free vertices are decreased by δ. In addition for some matched edges ij, one dual variable y_i is increased by δ while the other y_j is decreased by δ. These adjustments are done so the dual variables are always dominating and tight. Eventually the search finds a *wap*. The *wap* is used to augment the matching. Then the next search is done. After a number of searches the matching is complete and the algorithm halts.

The timing analysis of the scaling algorithm depends on an integrality property of the Hungarian method: If all edge weights and all given dual values are integers, then at any point in the algorithm any dual is a half-integer $\frac{y}{2}$, $y \in \mathbf{Z}$ (Papadimitriou and Steiglitz, 1982, 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 search, the dual values of all vertices in search trees have the same parity. Other versions of the Hungarian method (e.g., Lawler, 1976) have a stronger integrality property, but lead to more complicated scaling algorithms.)

Another point in the analysis of the scaling algorithm is the magnitude of the dual variables. The Hungarian method, and indeed any dual variable method, can produce dual values of size $\Omega(n\ N)$. To see this consider the graph of Figure 2, 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, the dominating and tightness conditions imply $y_{i+1} \geq y_i + N$. Hence $y_k \geq y_1 + (k-1)N$. This gives the desired bound.

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

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

where y_i are the input dual variables and M^* is a maximum complete matching. ($D \geq 0$ by dominance.) At 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 this point.

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

Proof. Consider the sum $\sum_{i \in V} y_i$ as dual variables are adjusted. A dual variable adjustment of δ decreases the sum by $g\delta$, where g is the number of free vertices when the adjustment is made. (Each free vertex decreases by δ, and the net change for two matched vertices is zero.) Every augment decreases g by two, so every value of g is at least f. Thus the total decrease (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. ∎

Scaling achieves a small value of D: It first solves the matching problem on a graph whose weights are half the original ones. This solution gives a set of dual variables that are near-optimum, i.e., D is small, on the original graph.

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 $\left\lfloor \frac{w_{ij}}{2} \right\rfloor$. ($G$ and $\bar{G}$ have the same vertices and edges.) Note that Step 3.2 (below) uses cardinality matching algorithm of Hopcroft and Karp (1973). This algorithm can be viewed as transforming an arbitrary input matching 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 $\bar{G}$. Recursively call $S(\bar{G})$ 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 complete:

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

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

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

Lemma 3.2. Procedure S returns a maximum complete matching and corresponding dual variables.

Proof. The recursive call of Step 1 returns dual variables that dominate in $\overline{G}$, i.e., for every edge ij, $\left\lfloor \frac{w_{ij}}{2} \right\rfloor \leq y_i + y_j$. Hence Step 2 computes dual variables that dominate in G.

Step 3 then simulates the Hungarian algorithm: Step 3.1 adjusts dual variables so an augment is possible. Step 3.2 achieves the effect of augmenting the matching while preserving the dominating and tightness conditions. Eventually a complete matching is found. ∎

To estimate the time, observe two properties of the number of iterations of Step 3 (in one recursive call). Recall the quantities f and D from Lemma 3.1.

(*i*) Step 3.1 is executed less than $2\,n^{\frac{1}{2}}$ times.

(*ii*) Step 3.1 is executed at most $n^{\frac{1}{4}}$ times with $f \geq n^{\frac{3}{4}}$.

To show these properties first note that Step 2 assigns dual variables y_i so that

$$D \leq \frac{n}{2} .$$

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

To show property (i), count the number of executions of Step 3.1 with $f \geq n^{\frac{1}{2}}$ and the number with $f < n^{\frac{1}{2}}$. If $f \geq n^{\frac{1}{2}}$ then Lemma 3.1 shows that $\Delta \leq \frac{D}{f} \leq \frac{n^{\frac{1}{2}}}{2}$. Every execution of Step 3.1 adjusts the dual variables 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 (by the integrality property of the Hungarian method), $\delta \geq \frac{1}{2}$. Thus the number of executions with $f \geq n^{\frac{1}{2}}$ is at most $n^{\frac{1}{2}}$. The number with $f < n^{\frac{1}{2}}$ is less than $\frac{n^{\frac{1}{2}}}{2}$ (since each execution matches two or more free vertices).

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

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 $O(n^{\frac{1}{2}}m)$ by (i).

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

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

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

The above derivation implicitly assumes that the dual values y_i do not grow inordinately large. Now we show this is true: the dual values are $O(nN)$ in magnitude.

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

The last part of the analysis of the scaling algorithm is to show that the bound of $O(n^{\frac{3}{4}}m)$ for one scale is tight. Specifically there are graphs of any density (i.e., any value of m between n and n^2) where in the last scale (i.e., the last time Step 3 is executed in S), there are $\Theta(n^{\frac{1}{4}})$ iterations; further, each

execution of the Hopcroft-Karp algorithm works on a graph of $\Theta(m)$ edges and finds $\Theta(n^{\frac{1}{2}})$ augmenting paths. The examples are reminiscent of (Even and Tarjan, 1975).

The family of graphs is A_k. A_k contains S, a bipartite graph on $2k^4$ vertices that has a complete matching and any desired number of additional edges. Further for each integer $i = 1, \ldots, k$, A_k contains k^2 copies of a path on $4i$ vertices, with edge weights alternately 0 and 1; the last vertex (in V_2) is joined by one edge to a V_1 vertex of S (see Figure 3(a)). For A_k, $n = 4k^4+2k^3$ and $3k^4+2k^3 \leq m \leq k^8+2k^4+2k^3$. Initially (after Step 2 and the first execution of Step 3.1) the matching and duals are as in Figure 3(b). For $i = 1, \ldots, k$, the i^{th} Hungarian search makes a dual variable adjustment with $\delta = 1$. An augmenting path is found on the paths with $4i$ vertices; the matching and duals on paths with more than $4i$ vertices are shown in Figure 3(c). The number of iterations is $k = \Theta(n^{\frac{1}{4}})$. Each execution of the Hopcroft-Karp algorithm finds $k^2 = \Theta(n^{\frac{1}{2}})$ augmenting paths. The number of tight edges is always at least $\frac{m}{2}$.

In a real implementation some changes should be made to procedure S. The recursion should be replaced by iteration. A different scaling regime is preferable: In scale s, instead of considering an edge weight with binary expansion $b_1 \cdots b_l$ 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 $y_i \leftarrow 2y_i+1$. Hence all edge weights are even and all duals are odd. This ensures that throughout the algorithm all duals are integers, by the integrality property of the Hungarian method.

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. Now we discuss maximum weight matching (other variants are in Section 5). Even if a graph has a com-

plete matching and all weights are positive, a maximum weight complete matching need not be a maximum weight matching. 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 $G_1 = G$. So the call $S(\hat{G})$ computes a maximum weight matching.

Corollary 3.1. A maximum weight matching on a bipartite graph can be found in $O(n^{\frac{3}{4}}m \log N)$ 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 are 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 $O(n_1^{\frac{3}{4}}m \log N)$, where $n_1 = |V_1| \leq |V_2|$. (This bound is used in Section 5.) To achieve this modify the algorithm as follows: In Step 2, the dual variables y_i for i in either copy of V_1 (i.e., $i \in V_1(G_1) \cup V_1(G_2)$) are increased by an extra unit, $y_i \leftarrow 2y_i + 1$; the duals for i in either copy of V_2 are doubled, $y_i \leftarrow 2y_i$. (Note that $V_1(G_1) \cup V_1(G_2)$ is *not* a bipartition set of $\hat{G}$). Step 3 (and Step 0) always maintain the same matching and dual variables on G_1 and G_2.

The timing analysis is similar to the one above. There are two key points. First, $D \leq 2n_1$. (This follows from the modified Step 2). Second, the Hopcroft-Karp algorithm is $O(n_1^{\frac{1}{2}}m)$. (This follows from an analysis similar to (Hopcroft

and Karp, 1973). The key observation is that a set of augmenting paths contains mostly edges of G. More precisely, 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 $i_1 i_2$ that joins G_1 to G_2.)

Scaling handles "small" edge weights efficiently. If the weights are *extremely* small, especially $N = 0(1)$, 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 dual variables y_i set to N; then execute Step 3 of procedure S. This gives the desired matching. An analysis similar to Theorem 3.1 shows the time is $0(N^{\frac{1}{2}} n^{\frac{3}{4}} m)$. This bound is inferior to Theorem 3.1, except when $N = 0(1)$; in this case the simplicity of this approach makes it preferable. More importantly, this algorithm works on nonbipartite graphs; further details are in (Gabow, 1984b).

The algorithm for maximum weight matching is: Start with M empty and all dual variables y_i set to N; then execute Step 3 of procedure S, terminating when either a complete matching has been found, or the dual values of all free vertices are 0. (Note that at any point of the algorithm all free vertices have the same dual value.)

The correctness of this method follows from the fact that a matching with dual variables that are tight, dominating, and *consistent* (i.e., all free vertices have dual value 0, and all matched vertices have nonnegative dual values) has maximum weight. This can be proved by the argument at the beginning of this section, or alternatively see (Lawler, 1976).

The time bound is $0(min(N^{\frac{1}{2}} n^{\frac{3}{4}}, Nn^{\frac{1}{2}})m)$. The first term of the bound is

derived as above. The second term results from the fact that Steps 3.1-3.2 are executed at most N times.

As before, the method works on nonbipartite graphs.

Corollary 3.2. If $N = O(1)$, a maximum complete matching can be found in $O(n^{\frac{3}{4}}m)$ time, and a maximum weight matching can be found in $O(n^{\frac{1}{2}}m)$ time; both algorithms use $O(m)$ space. Both bounds hold on nonbipartite graphs. ∎

4. Shortest paths with arbitrary lengths.

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

The approach is based on Edmonds and Karp's idea (1972) of transforming the lengths to make them nonnegative. (Also see Tarjan, 1983.) As in Section 2.1 this amounts to finding a function d_i that dominates the edge lengths, i.e., 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_1j_2 of G^* with weight $-l_{ij}$. In addition G^* has an edge i_1i_2 of weight 0 for each i.

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. The latter holds if and only if G^* has a maximum complete matching of weight 0. To see this first note that the complete matching $\{i_1i_2 | i \text{ a vertex of } G\}$ has weight 0. Next consider any complete matching on G^*. It parti-

tions the vertices of G into cycles. (Specifically a set of matched edges of the form $i_1j_2, j_1k_2, \ldots, r_1s_2, s_1i_2$ corresponds to the cycle $i,j,k, \ldots,r,s,i$. As a special case the matched edge $i_1 i_2$ corresponds to the cycle i.) Thus G^* has a positive weight complete matching if and only if G has a negative cycle.

So suppose a maximum complete matching has weight 0. In a corresponding set of dual variables, $y_{i_1} = -y_{i_2}$ for every vertex i of G. This is clear if i_1i_2 is matched. If i_1i_2 is unmatched there is an alternating cycle of the form $i_1j_2, j_2j_1, j_1k_2, \ldots, s_2s_1, s_1i_2, i_2i_1$. Since the cycle has weight 0, dominance and tightness imply $y_{i_1} = -y_{i_2}$.

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

This implies that the following algorithm finds shortest paths: Construct G^* and find a maximum complete matching. (If the weight is positive, stop - the problem is ill-defined.) Use the dual values to transform weights on G so they are nonnegative. Then run Dijkstra's algorithm on the transformed graph G.

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

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

5. Degree-constrained subgraphs.

Consider a multigraph where each vertex i has two associated integers l_i, u_i. A *degree-constrained subgraph (DCS)* is a subgraph in which 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 $O(U^{\frac{3}{4}} m \log N)$ on multigraphs; it is also $O(U^{\frac{1}{2}} n^{\frac{1}{3}} m \log N)$ 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, following (Gabow, 1983): 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, i.e., if i has (upper) degree bound $u \geq u_i$ 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 $N^* = 2^{\lfloor \lg N \rfloor + 1} - 1$. A maximum complete DCS on G corresponds to a maximum complete matching on G'. Hence DCS reduces to matching.

In general G' can have $\Omega(nm)$ edges. So for efficiency we do not work directly on G' to find the desired matching: We simulate the scaling algorithm S on G'. The simulation uses "sparse substitutes" to get a graph with $O(m)$ edges, as follows.

Given a matching on G', the *sparse substitute* for a vertex i of G is illustrated in Figure 4. This figure assumes that in G', $w \leq u$ external vertices in the substitute for i are matched on edges of G. Of the remaining $d - w$ external vertices, $u - w$ vertices (chosen arbitrarily) are unmatched, and each of the other Δ external vertices is matched to a vertex in the sparse substitute. The sparse

substitute contains two more vertices b, c joined by a matched edge. The other edges of the sparse substitute are as illustrated (each edge of Figure 4, except for bc, stands for w, $u-w$, or Δ edges of the sparse substitute). Each edge in the sparse substitute has weight N^*. G_k denotes the graph G' with sparse substitutes.

Assume (as will be the case) that the matching on G' covers every internal vertex of every vertex substitute. Then G_k has these properties:

(*i*) G_k has $O(m)$ edges.

(*ii*) Augmenting paths in G_k, and augmenting paths in G' that pass through each substitute at most once, correspond.

(*iii*) Dual variables that are dominating and tight on G' give similar variables 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 Figure 4. (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 at most once. However 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 dual values are then implied. Given dual values on G', there is a corresponding set of dual values 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 of the substitute have dual values y (for the internal vertex) and N^*-y (for the external vertex). (Dominance implies that the internal vertex is at least y, so

the external vertex is at most N^*-y; but increasing the external vertex 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.

Now we show how to simulate Steps 3.1-2 of procedure S on the graph G'. Step 3.1 does a Hungarian search on G'. To simulate it assign dual variables to G_k (by property (iii)) and search on G_k. Property (ii) shows that an augmenting path will be found if the current matching is not complete. The search gives new dual variables on G' that have a tight *wap* (by (ii)-(iii)). Thus the Hungarian search on G' is simulated using G_k.

Step 3.2 finds a maximum cardinality matching on T, the graph of tight edges of G'. (This matching must also cover all internal substitute vertices, so that properties (ii)-(iii) hold). T can contain $\Omega(nm)$ edges because of vertex substitutes. However consider a substitute $K_{\Delta,d}$. 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 subgraph of $K_{\Delta,d}$ that is included in T is a complete graph $K_{\Delta,\delta}$, where $\Delta \leq \delta \leq d$. Transform T to a graph T' as follows. In each vertex substitute of G' 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 and give the resulting vertex a degree constraint of $d-\delta$. Observe that T' has $2n$ vertices, at most m edges, and U value unchanged from G. To simulate Step 3.2 find a maximum cardinality DCS on T'.

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 $K_{\Delta,d}$). Then simulate $S(G')$: In Step 0 find the complete matching by solving the complete DCS cardinality problem on G; convert this to a matching on G'. In Step 3.1 construct the sparse substitute graph

G_k and do the Hungarian search on it. In Step 3.2 construct the graph T' (defined above) and find a maximum cardinality DCS on it. Convert this to a matching on G'. Then continue to iterate Step 3.

As in Section 3, the timing analysis of this algorithm is based on an estimate of the number of iterations of Step 3. Lemma 3.1 shows $f\Delta \leq D$, where these quantities are calculated on G'. Observe that $D \leq \frac{U}{2}$. (A complete matching on G' consists of $\frac{U}{2}$ edges of G and plus edges in vertex substitutes. The substitute edges are tight after Step 2 by the definition of N^*. So they contribute zero to D.) As in Section 3 at most $U^{\frac{1}{4}}$ iterations of Step 3 have $f \geq U^{\frac{3}{4}}$ and less than $\frac{U^{\frac{3}{4}}}{2}$ augments are done with $f < U^{\frac{3}{4}}$. Since Step 3.2 is $O(min(U^{\frac{1}{2}}, a)\ m)$ (Even and Tarjan, 1975), the total time is $O(U^{\frac{3}{4}} m \log N)$.

A possibly better bound holds if G has no parallel edges. In this case Step 3.2 is $O(n^{\frac{2}{3}} m)$ (Even and Tarjan, 1975). Since at most $\frac{U^{\frac{1}{2}}}{n^{\frac{1}{3}}}$ iterations have $f \geq U^{\frac{1}{2}} n^{\frac{1}{3}}$ and less than $\frac{U^{\frac{1}{2}} n^{\frac{1}{3}}}{2}$ augments are done with $f < U^{\frac{1}{2}} n^{\frac{1}{3}}$, the total time is $O(U^{\frac{1}{2}} n^{\frac{1}{3}} m \log N)$. This improves the first bound when $U = \Omega(n^{\frac{4}{3}})$.

Theorem 5.1. A maximum complete DCS on a bipartite multigraph can be found in $O(U^{\frac{3}{4}} m \log N)$ time. The time is also $O(U^{\frac{1}{2}} n^{\frac{1}{3}} m \log N)$ 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 (1974) the time is $O(n(U\ n\ m)^{\frac{1}{2}} \log N)$.

This result implies that a maximum weight cardinality-k matching can be found in $O(n^{\frac{3}{4}}m \log N)$ time. (The matching problem reduces to complete DCS). Similarity 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 $\hat{G}$ by making two copies of G, 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 executing the complete DCS algorithm 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 can also be derived. This depends on the fact that the bound of $O(n^{\frac{2}{3}}m)$ 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 path (*sap*s). The argument of (Even and Tarjan, 1975) generalizes as follows.

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

Now we show that Step 3.2 of the scaling algorithm finds a maximum cardinality DCS in $O(n^{\frac{2}{3}}m)$ time. This implies the graph bound of Theorem 5.1. Assume (without loss of generality) that the scaling algorithm maintains the same matching and dual variables on G_1 and G_2, the two isomorphic halves of $\hat{G}$. So T', the DCS multigraph of Step 3.2, is composed of T_1 and T_2, two isomorphic

graphs, joined by a number of multiedges $i_1 i_2$. An *sap* contains at most one "joining" edge $i_1 i_2$; further, all *sap*s contain their joining edge at the same level. So Principle 5.1 implies the desired bound.

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

The above analysis 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 Section 3.) So for instance if all vertices in V_1 have degree constraint one, the time bound for maximum weight DCS becomes that of 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 $O(U^{\frac{1}{2}} m)$ time. The time is also $O(n^{\frac{2}{3}} m)$ 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 minimum cost flow with a given value v.

A *0-1 network* has all capacities one. Parallel edges are allowed (and need not have equal cost.) Following (Even and Tarjan, 1975) we also consider some important special cases: 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. ("Type 1" of Even and Tarjan (1975) corresponds to type 1+ above. Type 1 above is more general, but the results of (Even and Tarjan, 1975) still apply by Principle 5.1. Type 1 and Type 1+ above 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 Section 4): A vertex i of G corresponds to two vertices i_1, i_2 of G^*; G^* has an edge i_1i_2 of weight 0 and multiplicity *mindegree(i)*. Further an edge jk of G corresponds to an edge j_1k_2 of G^* of weight $-a_{jk}$. Finally the degree constraints on G^* are $u_{i_1} = u_{i_2} =$ *mindegree(i)* for $i \neq s,t$; $u_{s_2} =$ *mindegree*(s), $u_{s_1} = u_{s_2}+v$, $u_{t_1} =$ *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 a 0-1 network G^* has $U = 0(m)$; for type 2, $U = 0(n)$. Also, even though G^* is a multigraph the graph bound of Theorem 5.1 applies if G is type 1 (by Principle 5.1).

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

The bounds of Theorem 6.1 apply to finding a minimum cost flow of maximum value, since the desired value v can be found using Dinic's algorithm (Even and Tarjan, 1975).

The Edmonds-Karp algorithm (1972) can be superior to the scaling algorithm on type 1+ networks with no negative cycles. It runs in $O(n\ m\ \log_{2+\frac{m}{n}} n)$ time on such networks (Tarjan, 1983), and if $m = \Omega(n^{\frac{4}{3}})$ this is faster than Theorem 6.1. However scaling can emulate this method: An approach similar to Dijkstra's algorithm (Section 2.1) achieves $O(n\ m\ \log_{2+\frac{m}{n}} N)$ time.

7. Conclusion

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 has been borne out by experiments on a PASCAL implementation of the matching algorithm).

Scaling is a general method. Applications to computational geometry are given in (Gabow et.al, 1984). Network problems on nonbipartite graphs are discussed in (Gabow, 1984b).

Acknowledgments.

The author thanks Dr. Robert E. Tarjan for insightful comments and advice that resulted in new algorithms and better presentation. Also thanks to Ann Bateson for programming the matching algorithm and spotting an error in a preliminary version.

References.

A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

R.E. Bellman, "On a routing problem," *Quart. Appl. Math* 16, 1958, pp. 87-90.

D. Cheriton, and R.E. Tarjan, "Finding Minimum Spanning Trees," *Siam J. on Computing*, 5, 1976, pp. 724-741.

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

R.B. Dial, "Algorithm 360: Shortest path forest with topological ordering," *C.ACM* 12, 1969, pp. 632-3.

E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik* 1, pp. 269-271, 1959.

E.A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Sov. Math. Dokl.* 11, 5, 1970, pp. 1277-1280.

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

S. Even and R.E. Tarjan, "Network flow and testing graph connectivity," *SIAM J. Comput.* 4, 4, 1975, pp. 507-518.

M.L. Fredman and R.E. Tarjan, "Fibonacci heaps and their uses", preprint.

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.

H.N. Gabow, "Efficient graph algorithms using adjacency matrices," in preparation, 1984a.

H.N. Gabow, "Efficient implementations of graph algorithms involving contraction", in preparation, 1984b.

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, to appear.

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.

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

D.B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. ACM* 24, 1, pp. 1-13.

A.V. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Soviet Math. Dokl. 15*, 1974, pp. 434-437.

H.W. Kuhn, "The Hungarian method for the assignment problem," *Naval Res. Logist. Quart.* 2, 1955, pp. 83-97.

H.W. Kuhn, "Variants of the Hungarian method for assignment problems," *Naval Res Logistics Quart.*, 3, 1956, pp. 253-258.

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

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.

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

D.D.K. Sleator, "An $O(n\ m \log n)$ algorithm for maximum network flow," Ph.D. Diss., Tech. Rept. STAN-CS-80-831, Stanford, Ca., 1980.

D.D.K. Sleator, and R.E. Tarjan, "A data structure for dynamic trees," *J. Comp. and System Sci. 26*, 1983, pp. 362-391.

R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA., 1983.

R.A. Wagner, "A shortest path algorithm for edge-sparse graphs," *J. ACM* 23, 1, 1976, pp. 50-57.

A.C. Yao, "An $O(|E| \log \log |V|$ algorithm for finding minimum spanning trees", *Information Processing Lett.*, 4 (1975), pp. 21-23.

Best Known Algorithm (see also, Tarjan, 1983)	Scaling Algorithm
minimum spanning tree	
$O(m \log \log_{2+\frac{m}{n}} n)$ (Yao, 1975; Cheriton and Tarjan, 1976)	$O(m(\log_n N + \alpha(m,n)))$
bottleneck shortest path	
$O(m \log_{2+\frac{m}{n}} n)$ (Edmonds and Karp, 1972)	$O(m \log_n N)$
shortest path (nonnegative lengths)	
$O(m \log_{2+\frac{m}{n}} n)$ (Dijkstra, 1959; Johnson, 1977)	$O(m \log_{2+\frac{m}{n}} N)$
maximum value network flow	
$O(min(n\ m \log n, n^3))$ (Sleator and Tarjan, 1983; Karzanov, 1974)	$O(n\ m \log N)$
maximum weight matching (bipartite graph)	
$O(n\ m \log_{2+\frac{m}{n}} n)$ (Kuhn, 1955, 1956)	$O(n^{\frac{3}{4}} m \log N)$
shortest path (arbitrary lengths)	
$O(nm)$ (Bellman, 1958)	$O(n^{\frac{3}{4}} m \log N)$

degree-constrained subgraph (bipartite graph)

$O(U\, m \log_{2+\frac{m}{n}} n)$	$O(U^{\frac{3}{4}} m \log N)$, multigraph
(Edmonds and Karp, 1972)	$O(U^{\frac{1}{2}} n^{\frac{1}{3}} m \log N)$, graph

minimum cost network flow (unit capacity)

$O(m^2 \log_{2+\frac{m}{n}} n)$	$O(m^{\frac{7}{4}} \log N)$
(Edmonds and Karp, 1972)	$O(n^{\frac{1}{3}} m^{\frac{3}{2}} \log N)$, type 1
	$O(n^{\frac{3}{4}} m \log N)$, type 2

Figure 1.

Algorithms for network problems.

n = number of vertices, m = number of edges; N = largest network parameter; U = sum of upper bounds ($U \leq m$).

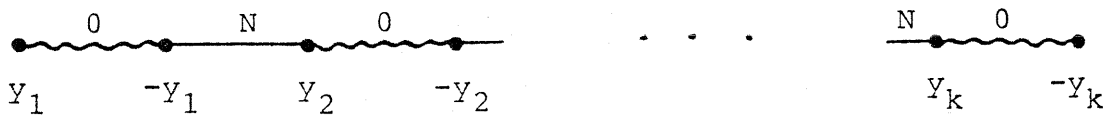
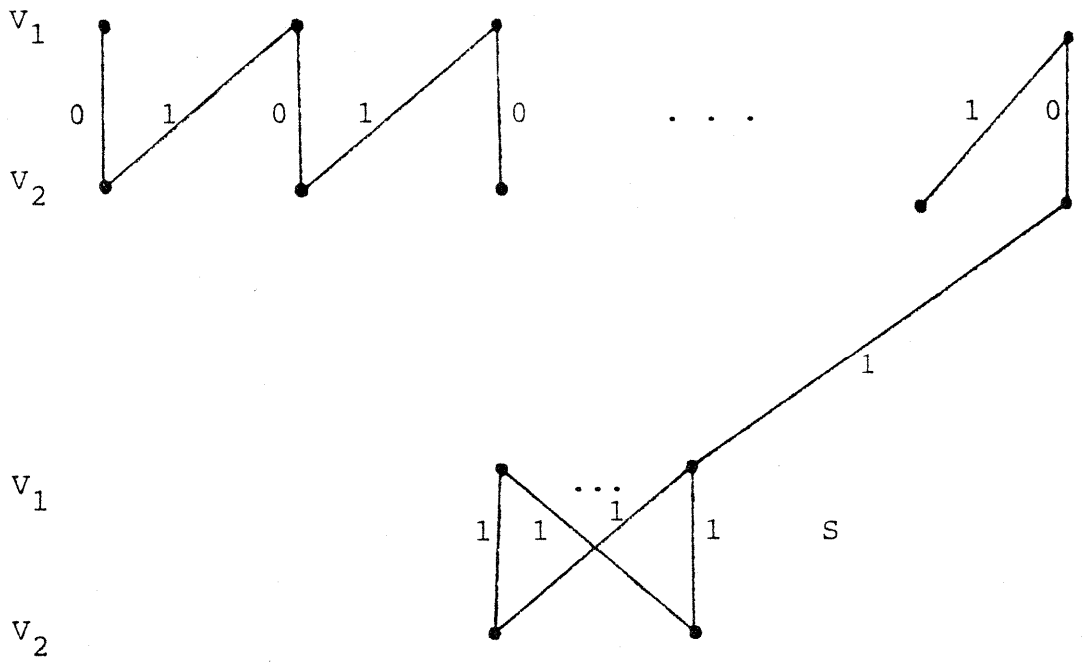
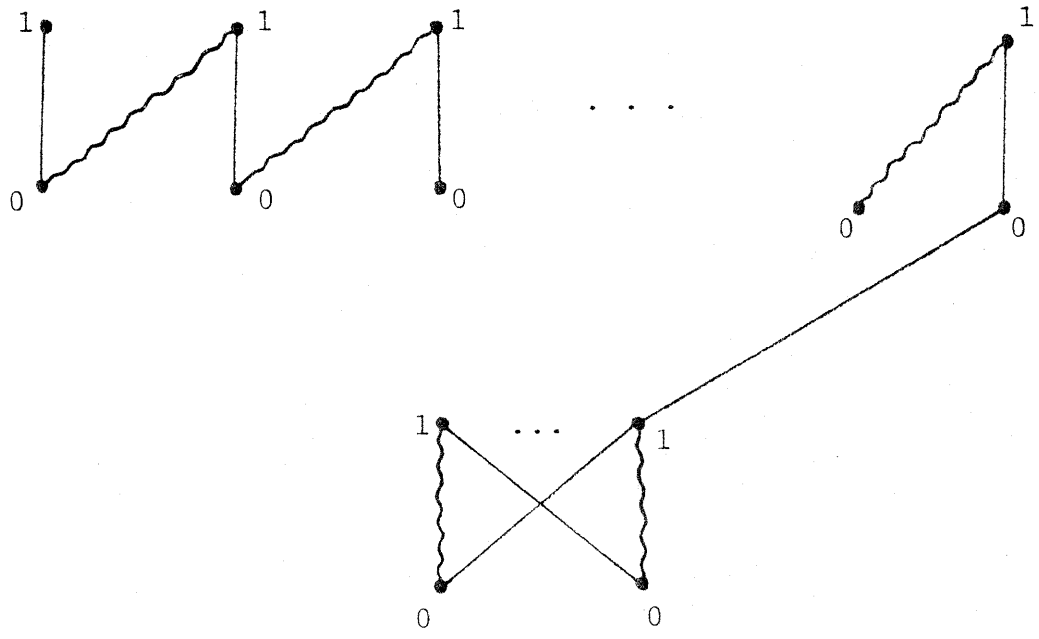


Figure 2.

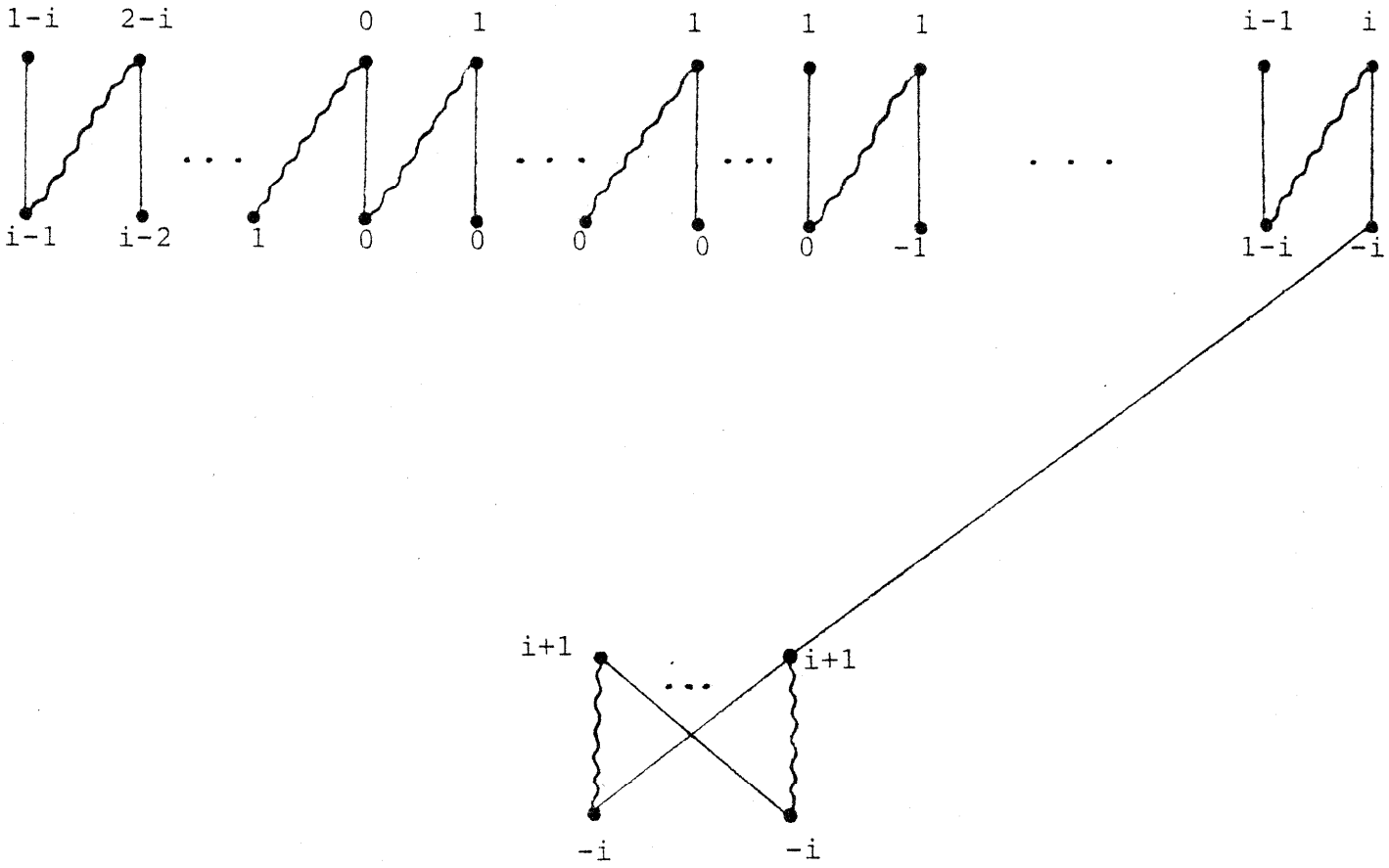
A graph and dual values.



(a)



(b)



(c)

Figure 3.

Worst-case graph for matching.

(a) Schematic representation of A_k .

(b) Initial matching and duals.

(c) Matching and duals after $i^{\underline{th}}$ search.

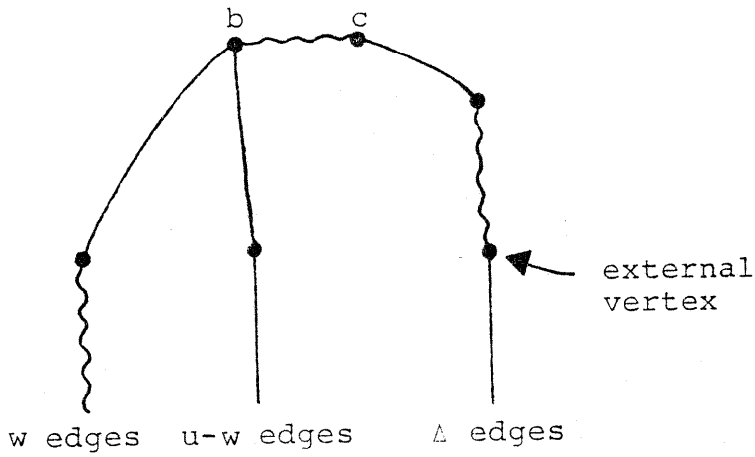


Figure 4.

Sparse substitute for i.