Algorithmica (1994) 11: 291–319

Algorithmica
© 1994 Springer-Verlag New York Inc.

Network Flow and 2-Satisfiability

Tomás Feder[1]

Abstract. We present two algorithms for network flow on networks with infinite capacities and finite integer supplies and demands. The first algorithm runs in $O(m\sqrt{K})$ time on networks with m edges, where $K = O(m^2/\log^4 m)$ is the value of the optimal flow, and can also be applied to the capacitated case by letting K be the sum of the *finite* capacities alone. The second algorithm runs in $O(wm \log K)$ time for arbitrary K, where w is a new parameter, the *width* of the network. These algorithms as well as other uses of the notion of width lead to results for several questions on the 2-satisfiability problem: minimizing the weight of a solution, finding the transitive closure, recognizing partial solutions, enumerating all solutions. The results have applications to stable matching, where w corresponds to the number of people and m to the instance size (usually $m \approx w^2$).

Key Words. Network flow, 2-Satisfiability.

1. Introduction.

The maximum-flow problem has been extensively studied. The complexity of the fastest-known algorithms is essentially nm on networks with n vertices and m edges, with an additional logarithmic factor that can be usually attributed to the data structures used (e.g., see [31]). This paper examines two special cases in which faster algorithms can be obtained. The model adopted here is that of networks with integer supplies and demands at the vertices, where all the edges have infinite capacities.

The first case is that of flow problems for which the value K of the optimal flow is relatively small (not too close to m^2). In that case we obtain an $O(m\sqrt{K})$ time bound. A standard reduction yields results for the more common capacitated case (with a single source and a single sink), where K is now the sum of the *finite* edge capacities alone. This extends known results for the case of networks where all edge capacities are 1 (parallel edges allowed) and for the case of networks where all vertex capacities are 1. The second case is that of networks which are not too wide. We define the *width* of a network and prove an $O(wm \log K)$ time bound for networks of width w, which holds for all values of the parameters.

For general blocking flows on acyclic networks, the best-known result is the $O(m \log(n^2/m))$ bound of Goldberg and Tarjan [11]. We prove an $O(m \log(w^2/m + 2))$ bound for the case of infinite capacities.

These results can be applied to the 2-satisfiability problem. For general 2-satisfiability instances, the problem of finding a minimum-weight solution is known to be $\mathcal{NP}$-complete, and indeed as hard to approximate as the vertex-cover problem, for which the best-known results guarantee an asymptotic approxima-

[1] Bell Communications Research, 445 South Street, Morristown, NJ 07960, USA.

Received August 20, 1991; revised January 3, 1992. Communicated by Harold N. Gabow.

tion factor of 2. We show that the blocking-flow result can be used to obtain a factor of 2 approximation to the minimum-weight solution in $O(m \log(w^2/m + 2))$ time, where m is the number of clauses and the width w is bounded above by the number of variables. An algorithm obtained independently by Gusfield and Pitt [18] gives the same approximation guarantee in $O(nm)$ time, where n is the number of variables. For *bipartite* 2-satisfiability instances, on the other hand, the problem can be reduced to network flow, and therefore a minimum-weight solution can be obtained in $O(m\sqrt{K})$ time if K is small, or in $O(wm \log K)$ time in general, where K is the weight of an optimal solution. We also show that the transitive closure of a 2-satisfiability instance can be obtained in $O(wm)$ time; this can be used to recognize partial solutions for a given instance efficiently. Finally, the solutions of a 2-satisfiability problem can be enumerated, with $O(m)$ preprocessing time, in $O(d)$ time per solution, using $O(m)$ space. Here d is the *maximum degree* of the 2-satisfiability instance, with $d \leq w$. The results of Gusfield [13] for stable matching give implicitly an enumeration algorithm that has $O(wm \log w)$ preprocessing time and takes $O(m)$ time per solution, using $O(m)$ space.

Our main application of these results is in stable matching. An account of much of the literature on the structure and algorithms for this problem can be found in [15]. It is known that the set of solutions to a stable-matching problem (and to more general stability problems) is characterized by a 2-satisfiability instance (see [13], [7], and [8]). Both the number of variables and the number of clauses can be as large as the total size m of the preference lists in the stable-matching instance. However, the number of people w in the stable-matching instance gives a bound on the *width* of the 2-satisfiability instance, with $m \approx w^2$ in the case of complete preference lists. Thus algorithms whose complexity depends on the width, rather than just the number of variables or clauses, are particularly useful. The results mentioned above for the 2-satisfiability problem yield the following, for stable-matching instances with w people and preference lists of total length m: For the nonbipartite stable-matching problem, known as the *stable-roommates* problem, finding a minimum-weight solution is $\mathcal{NP}$-hard [7], [8], but a solution within a factor of 2 of the optimum can be obtained in $O(m \log(w^2/m + 2))$ time. (Note that the logarithmic factor goes away if $m \approx w^2$.) Here an algorithm of Gusfield and Pitt [18] has an $O(m^2)$ time bound. For the bipartite stable-matching problem, known as the *stable-marriage* problem, a minimum-weight solution can be found in $O(m\sqrt{K})$ time if K is small, or in $O(wm \log K)$ time in general, where K is the weight of an optimal solution. In particular, the egalitarian stable-marriage problem, which has $K \leq m$, can be solved in $O(m^{1.5})$ time (independently of w). An algorithm of Irving *et al.* [23] gives an $O(m^2 \log m)$ time bound for the weighted stable-marriage problem, and an $O(m^2)$ bound for the egalitarian stable-marriage problem. With $O(wm)$ preprocessing time (a transitive-closure computation), the stability of a set of k pairs can be determined in $O(k^2)$ time. In particular, the stability of a single pair can be determined in constant time. An algorithm of Gusfield and Irving for this problem [15] has $O(m^2)$ preprocessing time, and the same query time. Finally, all stable matchings can be enumerated after $O(m)$ preprocessing time in $O(w)$ time per solution, using $O(m)$ space. An algorithm of Gusfield [12] has the same complexity in the marriage case; in the roommates case [13] the algorithm has $O(wm \log w)$ preprocessing time and takes $O(m)$ time per solution, using $O(m)$ space.

The remainder of the paper is organized as follows. Section 2 gives some basic definitions and preliminary results. Sections 3 and 4 give the two maximum-flow algorithms. Section 5 describes the blocking-flow algorithm. Sections 6 and 7 give the applications to the 2-satisfiability problem, and Section 8 describes the enumeration algorithm. Section 9 gives the applications to stable matching. Section 10 concludes with some open questions.

2. Flow Problems and Width. A *capacitated flow problem* is defined by a directed graph $G = (V, E)$ with two distinguished vertices, a *source* s and a *sink* t, and a positive (possibly infinite) integer capacity $\mathrm{cap}(u, v)$ on every edge (u, v). For convenience we define $\mathrm{cap}(u, v) = 0$ if (u, v) is not an edge in G. A *flow* on G is an integer-valued function f on vertex pairs satisfying the following three properties:

(1) $f(v, u) = -f(u, v)$. If $f(u, v) > 0$, we say that there is a flow from u to v.
(2) $f(u, v) \le \mathrm{cap}(u, v)$. If (u, v) is an edge such that $f(u, v) = \mathrm{cap}(u, v)$, we say that the flow *saturates* (u, v).
(3) For every vertex u other than s and t, $\sum_v f(u, v) = 0$.

The *value* of a flow is the net flow out of the source, $\sum_v f(s, v)$. The *maximum-flow problem* is that of finding a flow of maximum value. Given a flow f, the residual network $R(f)$ is the graph with vertex set V, source s, sink t, and an edge (u, v) of capacity $\mathrm{res}(u, v) = \mathrm{cap}(u, v) - f(u, v)$ (the residual capacity) for every pair (u, v) such that $\mathrm{res}(u, v) > 0$. A flow g in the residual network $R(f)$ can be transformed into a flow $g + f$ in the original network (and vice versa, by subtracting f instead of adding f).

A cut $X, \bar{X}$ is a partition of the vertex set V into two parts X and $\bar{X} = V - X$ such that X contains s and $\bar{X}$ contains t. It will sometimes be convenient to view a cut as an assignment of boolean values to the vertices in V, with the vertices in X given the value 1 and the vertices in $\bar{X}$ given the value 0. The edges *across the cut* are the edges that start in X and end in $\bar{X}$. The *capacity* of a cut is the sum of the capacities of the edges across the cut. A cut of minimum capacity is a *minimum cut*. The *flow across the cut* is the sum of all $f(u, v)$ with $u \in X$ and $v \in \bar{X}$; it can be shown that the flow across any cut is equal to the flow value. By the capacity constraint, the flow across any cut cannot exceed the capacity of the cut. Therefore the value of a maximum flow is no greater than the capacity of a minimum cut. The *max-flow min-cut theorem* states that these two quantities coincide. Thus, for a minimum cut, the flow across the cut equals the capacity of the cut; this property holds for a given cut if and only if all the edges that start in X and end in $\bar{X}$ are saturated, and there is no flow along the edges that start in $\bar{X}$ and end in X. (See [34] for a more detailed exposition.)

Sometimes the flow problem has the following special structure. All directed edges have infinite capacities and link vertices other than the source and the sink, except for finite capacity edges (s, u) joining the source to a vertex other than the sink, and finite capacity edges (v, t) joining a vertex other than the source to the sink. In that case we discard the finite capacity edges, the source, and the sink, and say that vertex u is a *supply vertex* with supply cap(s, u), and that vertex v is a *demand vertex* with demand cap(v, t). We can assume without loss of generality that no vertex is both a supply and a demand vertex. We apply the terms residual supply (demand) and saturated supply (demand) to a vertex u (resp. v) as they would apply to the corresponding edge (s, u) (resp. (v, t)); the flow out of u (resp. into v) in the simplified network is just the flow along (s, u) (resp. along (v, t)).

We refer to this problem with only infinite capacities, supplies, and demands as the *uncapacitated flow problem.* An important special case is the *bipartite matching problem*: here the vertex set is the union of two disjoint sets $V = V_1 \cup V_2$, with edges of infinite capacity directed from V_1 to V_2, and where vertices in V_1 have supply 1 and vertices in V_2 have demand 1.

We let $n = |V|$ be the number of vertices and let $m = |E|$ be the number of edges in a directed graph G. We also associate with G a third parameter, the *width* of G. We define two closely related notions of width. The *implicit width* of G is the maximum cardinality of a set of vertices S with the property that, given two distinct vertices u and v in S, there is no directed path in G from u to v. Our algorithms actually require the notion of *explicit width,* defined as follows. A *path cover* for G is a set of vertex-disjoint directed paths in the transitive closure of G whose union contains all the vertices in G. By Dilworth's theorem [4], the minimum number of paths in a path cover for G equals the implicit width of G. We say that G has explicit width w' if a path cover for G consisting of w' paths is known. We always add the edges of the paths in the known path cover (at most $n - 1$ edges) to the graph G, in the uncapacitated flow problems; this does not affect the flow results, since these infinite capacity edges correspond to infinite capacity paths in the graph.

It is possible to find a path cover consisiting of w paths for a graph of implicit width w by running a costly min-flow algorithm. For our purposes, however, even a good path cover can sometimes be useful. The lemma below implies that, for the flow algorithm of Section 4, the distinction between implicit and explicit width will cost at most a logarithmic factor.

LEMMA 2.1. *A greedy algorithm finds a path cover with $w' \leq w \log n$ paths for a graph of implicit width w in $O(w'm)$ time.*

PROOF. First transform the graph into an acyclic graph G by merging strong components, and determine a consistent linear order, in $O(m)$ time [33]. The algorithm now marks the vertices of G in some order, starting from the graph with all vertices unmarked. At each stage, it finds in $O(m)$ time a path in G containing the largest possible number of unmarked vertices, then mark these vertices. Since a path cover with w paths exists, some path in that cover must contain at least a fraction $1/w$ of the unmarked vertices; hence each path found will mark at least this many. After $w \log n$ paths have been found, the number of unmarked vertices left is at most $n(1 - 1/w)^{w \log n} < 1$, so all vertices are marked and hence covered. □

The following result gives a simple illustration of the notion of width.

LEMMA 2.2. *All the edges in the transitive closure of a directed graph which involve at least one vertex from a given path P can be obtained in $O(m)$ time. Therefore the transitive closure can be obtained in $O(wm)$ time for a graph of explicit width w.*

PROOF. Let $u_1 \to u_2 \to \cdots \to u_r$ be the path P. For every vertex v, let $\varphi(v)$ be the greatest i such that the edge $u_i \to v$ is in the transitive closure of the graph. If no such i exists, we let $\varphi(v) = 0$. Note that an edge (u_j, v) is in the transitive closure if and only if $j \leq \varphi(v)$. To obtain $\varphi(v)$ for all vertices v, we consider $i = r, \ldots, 2, 1$ in turn. For each i, we find all vertices v that are reachable from u_i in the graph, set $\varphi(v) = i$ since $u_i \to v$ is in the transitive closure, and remove these vertices and incident edges from the graph. After the last value $i = 1$ has been considered, we set $\varphi(v) = 0$ for all vertices v that were not removed from the graph. The correctness of the algorithm follows from the fact that if $\varphi(v) = i$, then a path from u_i to v cannot go through a vertex v' with $\varphi(v') > i$. The time complexity is $O(m)$ since every vertex and every edge of the graph is considered and removed only once. The edges in the transitive closure that are of the form $v \to u_i$ can be obtained with the same algorithm, after reversing all directed edges in the graph. □

A capacitated flow problem (with both finite and infinite capacity edges) can be reduced to an uncapacitated problem (with infinite capacity edges only) by using the following standard reduction. For each capacitated edge (u, v) of positive finite capacity c, introduce two new vertices x and y with demand and supply c, respectively, and replace (u, v) by three directed edges (u, x), (y, x), and (y, v) of infinite capacity. An amount $f \leq c$ of flow along (u, v) in the original problem can then be represented with an amount f of flow along (u, x) and (y, v), and an amount $c - f$ of flow along (y, x). The resulting uncapacitated maximum-flow problem will then have an maximum flow whose value differs from that of the original capacitated flow problem by exactly the sum C of the finite capacities c. Note that if the optimal flow for the original capacitated problem is finite, then its value is at most C. This reduction may significantly increase the number of vertices of the graph, but it does not increase the number of edges by more than a factor of 3. Therefore any time bound for the uncapacitated problem which depends only on the number of edges and the value of the optimum flow (or even on the total supply and demand) can be translated into a time bound for the capacitated problem in terms of the number of edges and the sum of the finite capacities alone.

For uncapacitated flow problems, the measures of implicit and explicit width are those associated with the underlying graph. For capacitated flow problems, we consider an alternative notion of width; this notion is of interest mainly because it allows a reduction from uncapacitated networks of explicit width w to capacitated networks of width $2w$, and because we can give a fast flow algorithm for the later. In the capacitated case, which has a source, a sink, and edges of both finite and infinite capacity, we define the *cut width* as the maximum number of edges across a *finite capacity cut*. The reduction from the uncapacitated to the capaci-

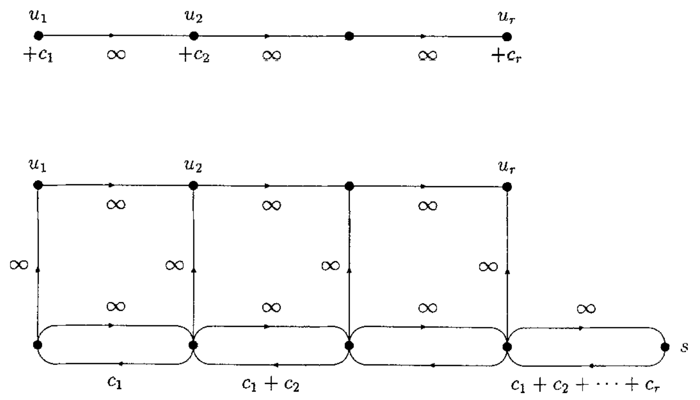

Fig. 1. Providing flow at supply vertices.

tated case that does not increase the width by more than a factor of 2 is as follows. Given an uncapacitated network G and a path cover of w paths for G, we construct a capacitated network G' by adding vertices and edges to G as follows (see Figure 1). For each path P in the path cover with vertices $u_1, u_2, \ldots, u_r$, and infinite capacity edges (u_i, u_{i+1}), we create a new path with new vertices $u'_1, u'_2, \ldots, u'_r$, $u'_{r+1} = s$ (where s is the source). We add infinite capacity edges (u'_i, u'_{i+1}) and (u'_i, u_i) for $1 \le i \le r$, as well as finite capcity edges (u'_{i+1}, u'_i) of capacity $C_i = \sum_{j \le i} c_j$, where c_j is the supply of vertex u_j in the uncapacitated problem. A similar construction is carried out for the demands: we create a new path with new vertices $t = u''_0$, u''_1, $u''_2, \ldots, u''_r$ (where t is the sink), add infinite capacity edges (u''_{i-1}, u''_i) and (u_i, u''_i) for $1 \le i \le r$, as well as finite capacity edges (u''_i, u''_{i-1}) of capacity $D_i = \sum_{j \ge i} d_j$, where d_j is the demand of vertex u_j in the uncapacitated problem. This network will have a cut width at most $2w$: all the finite capacity edges are on the paths of u'_i and u''_i, and the set edges across a finite capacity cut $X, \bar{X}$ cannot contain two finite capacity edges from the same path of u'_i (or of u''_i) because if it contains two edges (u'_{i+1}, u'_i) and (u'_{j+1}, u'_j) with $i < j$, then u'_{i+1} is in X, u'_j is in $\bar{X}$, and u'_j is reachable from u'_{i+1} along an infinite capacity path that must go across the cut, contradicting the fact that the capacity of the cut is finite.

LEMMA 2.3. *An uncapacitated flow problem on a graph of explicit width w can be reduced to a capacitated flow problem of cut width at most $2w$, without increasing the number of vertices and edges by more than a constant factor.*

PROOF. We show a correspondence between flows in an uncapacitated network G and in the capacitated network G' constructed from it, in both directions. Given a flow on G, for each path P as above, if f_i is the flow out of vertex u_i with supply c_i (and $0 \le f_i \le c_i$), we assign flow f_i to the new infinite capacity edges (u'_i, u_i), and assign flow $\sum_{j \le i} f_j$ to the finite capacity edges (u'_{i+1}, u'_i). Demand vertices are handled similarly, so that if f_i is the flow into vertex u_i with demand d_i (and $0 \le f_i \le d_i$), we assign flow f_i to the infinite capacity edges (u_i, u''_i) and assign flow

$\sum_{j \geq i} f_j$ to the finite capacity edges (u_i'', u_{i-1}''). It is easy to check that the capacity bounds and flow conservation laws hold, and that the value of the flow in G' is the same as in G. This completes one direction of the reduction.

In the other direction, given a flow on G', we first perform a simple transformation on the flow; this transformation ensures that the flow for the uncapacitated network satisfies the supply and demand constraints. Let F_i be the flow along the edge (u_{i+1}', u_i'), so that the flow along (u_i', u_i) is $F_i - F_{i-1}$ by flow conservation (with $F_0 = 0$ by convention). Let $F_r' = F_r$ and $F_{i-1}' = \min(F_i', C_{i-1})$ by induction, for $1 \leq i \leq r$. Now assign flow F_i' to the edge (u_{i+1}', u_i') and flow $F_i' - F_{i-1}'$ to the edge (u_i', u_i); increase the flow along (u_i, u_{i+1}) by $F_i' - F_i$. We can check by induction that $F_i \leq F_i'$, and directly that flow conservation and capacity bounds hold. Furthermore, the flow along (u_i', u_i) is $F_i' - F_{i-1}' = \max(0, F_i' - C_{i-1}) \leq C_i - C_{i-1} = c_i$. Thus if we remove all the u_i' vertices and let the flow out of supply vertex u_i be $F_i' - F_{i-1}'$, then this amount of flow will satisfy the supply constraint c_i as required. The analogous transformation for the demand vertices is as follows. Initially the flow along (u_i'', u_{i-1}'') is F_i and the flow along (u_i, u_i'') is $F_i - F_{i+1}$, where $F_{r+1} = 0$. We let $F_1'' = F_1$ and $F_{i+1}'' = \min(F_i'', D_{i+1})$ by induction, assign flow F_i'' to the edge (u_i'', u_{i-1}'') and flow $F_i'' - F_{i+1}''$ to the edge (u_i, u_i''); we increase the flow along (u_{i-1}, u_i) by $F_i'' - F_i$. The flow along (u_i, u_i'') is $F_i'' - F_{i+1}'' = \max(0, F_i'' - D_{i+1}) \leq D_i - D_{i+1} = d_i$, so we can remove all the u_i'' vertices and let the flow into demand vertex u_i be $F_i'' - F_{i+1}''$, satisfying the demand constraint d_i. This completes the other direction of the reduction. □

Some flow algorithms use as a subroutine the computation of a blocking flow in a capacitated acyclic network. A *blocking flow* is a flow such that every directed path from the source to the sink traverses a saturated edge. In the uncapacitated case, this means that if there is a directed path from a supply vertex to a demand vertex, then at least one of the two vertices must be saturated. The acyclicity assumption can be removed in the uncapacitated case, because the graph can be made acyclic by merging strong components and combining the corresponding supplies and demands, in $O(m)$ time [33].

The efficient implementation of flow algorithms often requires a data structure called dynamic trees, due to Sleator and Tarjan [31], [34]. This data structure is used to maintain a collection of vertex-disjoint rooted trees, with costs associated with the vertices, under the following operations. The maketree(v) operation creates a new tree containing the single vertex v, previously in no tree, with cost zero; findroot(v) returns the root of the tree containing vertex v; findcost(v) returns the pair (w, x) where x is the minimum cost of a vertex on the tree path from v to findroot(v) and w is the last vertex (closest to the root) on this path of cost x; addcost(v, x) adds x to the cost of every vertex on the tree path from v to findroot(v); link(v, w) combines the distinct trees containing vertices v and w by making v a child of w, where v must be a root; cut(v) divides the tree containing vertex v into two trees by deleting the edge joining v to its parent, where v must not be a root; and evert(v) makes v the root of the tree containing v. In the algorithm of Section 3 we also need to be able to perform the addcost operation separately for vertices at even and at odd depth in the tree. This modified version of the addcost operation can be incorporated into the dynamic-tree data structure. This data structure makes it possible to execute an arbitrary sequence of any of these operations in time $O(\log t)$ per operation, where t is the size of the largest tree obtained during the execution.

3. Maximum Flow when the Optimum Is Small. We are given a maximum flow instance on a directed graph $G = (V, E)$ with infinite capacities and integer supplies and demands, where $|V| = n$ and $|E| = m$. Our strategy for this problem is the following. We first view the problem as a flow problem in an auxiliary graph H, and give an algorithm for obtaining a maximum flow on H. We then show how the algorithm on H can be implemented efficiently without explicitly constructing the graph H. We assume that G is acyclic. This can be ensured via an $O(m)$ strong components computation [33], where strong components are replaced by single vertices and the corresponding supplies and demands are combined.

The auxiliary graph is a bipartite graph $H = (A \cup B, F)$. The set A consists of the supply vertices in G, the set B consists of the demand vertices in G (with supply and demand values inherited from G), and F contains a directed edge of infinite capacity (a, b) with $a \in A$ and $b \in B$ if and only if there is a directed path in G from a to b. The graph H is thus essentially the transitive closure of G, Clearly, a maximum flow in H corresponds to a maximum flow in the original graph G. For convenience, we assume that in the original graph G, the supply vertices have in-degree zero and the demand vertices have out-degree zero. This property can always be enforced for a supply vertex v by introducing a new vertex v' and an infinite capacity edge from v' to v, and moving the supply from v to v'; a similar transformation works for demands.

A maximum flow in H can be obtained by matching supplies in A to demands in B by a multiset M of edges in F: the number of occurrences of the edge (a, b) in M is the flow from a to b, and the number of edges in M involving a vertex $a \in A$ (resp. $b \in B$) must be at most the supply (resp. demand) of the vertex. A maximum matching (one that maximizes the size of M) can be found with the Hopcroft and Karp matching algorithm, or equivalently with Dinits's flow algorithm [5], [21], [34]. This is done as follows. Suppose that some flow f in H is known. This flow induces a residual network which consists of the same infinite capacity edges from A to B, but which also contains edges (b, a) of capacity c whenever f has a flow of value $c > 0$ along an edge (a, b) in H. Furthermore, supplies and demands have been updated in this residual network in accordance with the flow f. (That is, the supply at a vertex $a \in A$ has been reduced by the amount of flow leaving a in f, and the demand at a vertex $b \in B$ has been reduced by the amount of flow entering b in f.)

From this residual network, a layered network is defined as follows. The vertices in A with positive supply in the residual network are at level 0, and so are the vertices in B adjacent in H to vertices of A at level 0. For $i > 0$, the vertices in A at level i are the vertices of A that are not at smaller levels and are adjacent to vertices at level $i - 1$ in B through a capacitated residual edge; the vertices in B

at level i are the vertices of B that are not at smaller levels and are adjacent in H to vertices of A at level i through an uncapacitated edge. The last level l is the level that contains vertices in B with positive residual demand. The value l is the *length* of the layered network. The layered network contains precisely those vertices that are at some level $0 \leq i \leq l$ and those edges of the residual network that are either infinite capacity edges joining a vertex in A at level i to a vertex in B at level i for some i, or finite capacity residual edges joining a vertex in B at level $i - 1$ to a vertex in A at level i. Therefore the layered network is acyclic. The algorithm finds a blocking flow g in this layered network, that is, a flow that cannot be increased by adding flow from a supply vertex to a demand vertex along unsaturated edges of the layered network. This blocking flow can now be added to the previous flow f to obtain a new flow $f' = f + g$. If we now construct the residual network and the layered network corresponding to f', it is known that the new layered network must have length at least $l + 1$. This in turn implies that the number of *phases*, i.e., the number of times that the flow f and the corresponding residual network are updated before the final optimal flow is obtained, is at most $2\sqrt{K}$, where K is the value of the optimal flow (see below for a justification of this bound).

If we could implement each phase in $O(m)$ time, we would then have an $O(m\sqrt{K})$ algorithm, as desired. However, even if we put aside the complexity of obtaining H from G via a transitive-closure computation, we face the difficulty that both the number of edges from A to B in H, and from B to A in the residual networks, could be as large as m^2 (i.e., we could have G sparse but H dense), giving an $O(m^2)$ complexity for each phase. We handle this difficulty differently for the two types of edges: For the capacitated residual edges from B to A, we maintain a "forest" solution, thus keeping their number bounded by $O(m)$ (in fact by $n - 1$). For the uncapacitated edges from A to B, we avoid computing the transitive closure explicitly, and work instead with edges in the original graph G, whose number is bounded by m. This gives the desired bound.

We first show how the number of capacitated edges from B to A in the residual graph can be kept within an $O(m)$ bound. The main idea to achieve this is to maintain a "forest" solution (sometimes known as a "spanning-tree" solution). Suppose then that the edges from B to A in the resudual network at the beginning of a phase form a forest (when viewed as undirected edges). We ensure that, after the blocking flow is obtained, the residual network is updated so that this forest property is maintained. This means that the number of residual edges is in fact always at most $n - 1$. *A priori*, each edge (a, b) from A to B along which the blocking flow g sends a positive flow must be added as an edge (b, a) to the residual network. It may happen that the addition of a new residual edge (b, a) completes a cycle among the edges from B to A in the residual network (when viewed as undirected edges). If this happens, we perform the following transformation. Let δ be the minimum residual capacity of an edge (b', a') in this newly created cycle of capacitated residual edges. Note that the cycle has an even number of edges, since the residual network is bipartite. The edges (b'', a'') on the cycle may thus be alternatively labeled even and odd, with the edge (b', a') of capacity δ labeled even by convention. We then decrease the flow f from A to B along (a'', b'') if (b'', a'')

is an even edge by δ, and increase the flow by δ for an odd edge. This transformation preserves the validity of the flow f. Furthermore, the edge (b', a') is now no longer a residual edge, so the residual edges from B to A constitute once again a forest. Since this transformation creates no new residual edges, the length of the layered network will still increase in each phase from l to at least $l + 1$, so the number of phases is still bounded by $2\sqrt{K}$. The trees in the forest can be maintained as dynamic trees. This data structure enables us to find the tree path from b to a (use *evert* to make a the root), to find the minimum capacity edge on this path (with *findcost* at b), to update the capacities of edges along the path (with *addcost*, provided that even and odd depth costs are updated separately, as mentioned in Section 2), and to link and cut trees in time $O(\log n)$ per operation. Thus, if R is the total number of capacitated residual edges created during all phases of the algorithm, then the complexity of maintaining a forest solution is $O(R \log n)$. Later we see that $R = O(K \log n)$, giving an $O(K \log^2 n)$ time bound.

We now show how to compute a blocking flow in the layered network efficiently. (See Figure 2.) Recall that the layered network can contain a large number of infinite capacity edges from A to B, but only $O(m)$ capacitated edges from B to A (by the forest property). We say that a vertex or an edge in G is at level i if i is the least number such that the vertex or edge can be reached in G from a vertex in A at level i. Note that if (a, b) is an edge in H with both endpoints at level i, then each path from a to b in G must be contained in level i. For given such a path, it is clear by definition that all vertices and edges along the path are at level i or smaller; if some such vertex or edge were at a level $j < i$, then this vertex or edge, and hence b itself, would be reachable in G from a vertex in A at level j, so

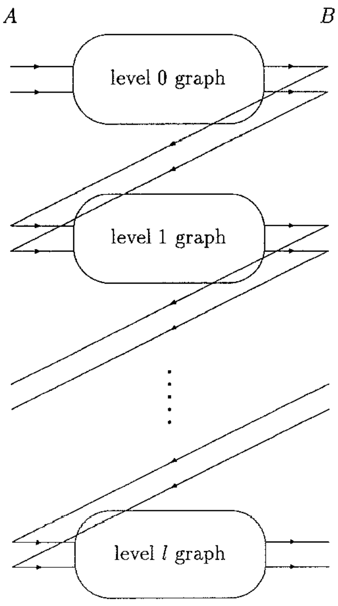

Fig. 2. Representation of the layered network.

b would be at level j or smaller. Levels of vertices can be determined by giving length 0 to all infinite capacity edges in G, length 1 to all finite capacity edges in the residual graph, and performing a shortest-path computation from the set of vertices in A at level 0, in $O(m)$ time.

We have this partitioned the vertices and edges of G into disjoint levels; the edges (a, b) in H with a and b at level i are represented by paths in G within level i. We can discard the edges in G that join vertices in two different levels, since they do not represent any edge in the layered network; after discarding such edges, we refer to the vertices and edges in level i as the *level i graph.* If we now combine level i graphs for all i with the capacitated edges joining two adjacent levels in the residual network, we have an implicit representation of size $O(m)$ for the layered network, with capacitated edges from B to A in the layered network given explicitly, and infinite capacity edges in the layered graph from A to B given implicitly by paths in G within a level. We refer to this graph as the *representation of the layered network*, illustrated in Figure 2. This representation is acyclic, because each level is acyclic and capacitated edges only go from level i to the next level $i + 1$. An $O(m \log n)$ blocking-flow algorithm [31], [34] can be run directly in this representation graph. We sketch here such an algorithm, but with some modifications that lead to a better time bound for our problem. The reader is referred to [11], [31], and [34] for details.

The algorithm executes the following two steps in alternation:

(1) Repeatedly remove all vertices of out-degree zero other than the unsaturated demand vertices in B at level l, together with their incoming edges, and remove the edges of capacity zero as well.
(2) Traverse a path from an unsaturated supply vertex to an unsaturated demand vertex, and send flow along this path until either the supply, the demand, or one of the capacitated edges along the path is saturated.

Note that the path started at an unsaturated supply vertex can always be extended until an unsaturated demand vertex at level l is reached, since all the other vertices have nonzero out-degree and the graph is acyclic. An additional requirement is that whenever a path reaches a vertex that had previously been visited, it proceeds from this vertex along a previously visited edge, if there is any such edge left in the graph. As a result, at any point in time, there is at most one outgoing edge for each vertex that has been visited but has not been removed, so the visited edges remaining in the graph form a collection of in-trees. These trees can be maintained using dynamic trees, linking trees when a new edge is visited and cutting trees when an edge is removed. The edges on the path in step (2) that belong to dynamic trees are not visited, because whenever a path reaches a dynamic tree, it can proceed directly from the root of the tree. As a result, each edge is visited only twice, once the first time it is visited during step (2) at which point it becomes part of the dynamic trees, and once when it is removed during step (1). Thus the number of dynamic-tree operations is $O(1)$ per edge, with an additional $O(1)$ per path created in step (2). The number of such paths is also $O(m)$, because each path saturates an edge or a vertex. This gives a total of $O(m)$ dynamic-tree operations for an $O(m \log n)$ time bound.

We introduce two modifications in this algorithm. First, we never link dynamic trees in different levels. Therefore the path in step (2) always visits the capacitated edges joining different levels and knows the vertices in A and B visited, hence the path in the (explicit) layered network used to increases the flow. Second, we enforce a fixed bound t on the dynamic-tree size as in [11]. These modifications give an additional cost due to the fact that certain edges are visited but not incorporated into the dynamic trees by the path in step (2). Each such edge costs a constant number of dynamic-tree operations; we give an upper bound on the number of such edges. The edges not incorporated into the dynamic trees because of the first modification are the l capacitated edges joining different levels along the path. The edges not incorporated because of the second modification are edges whose endpoints belong to two dynamic trees, where one of the two trees has at least $t/2$ vertices (oterwise the bound t on the joint tree size would be met and the edge would be incorporated into the dynamic trees). The number of trees with at least $t/2$ vertices is at most $\lfloor 2n/t \rfloor$, and each such tree is charged at most twice, once by the edge where the path enters the tree, and once by the edge where the path leaves the tree at the root. Therefore at most $2\lfloor 2n/t \rfloor$ edges are visited because of the second modification. This gives an additional $O(l + \lfloor 2n/t \rfloor)$ dynamic-tree operations per path. If the flow increase in a phase is h, then the number of paths in step (2) is at most h, giving a total of $O(m + (l + \lfloor 2n/t \rfloor)h)$ dynamic-tree operations per phase. Each operation takes $O(\log t)$ time, so the time bound for the blocking-flow computation at each phase is $O((m + (l + \lfloor 2n/t \rfloor)h) \log t)$.

To bound the total time complexity of the algorithm, we use the following fact, which is again known from standard matching algorithms: if the flow after a layered network of length l has been obtained is K', then the optimum flow $K \geq K'$ differs from K' by at most K'/l. We can cover the possible values of l $(0 \leq l < n)$ with $\log n$ intervals of the form $L \leq l < 2L$ plus the additional value $l = 0$. If h is the amount of flow when the length is l, then $\sum_{L \leq l < 2L} h \leq K/L$ and $\sum_{L \leq l < 2L} lh \leq 2K$. Therefore $\sum_{0 \leq l < n} lh \leq 2K \log n$. Note that the number of capacitated residual edges created in a phase is at most $(l + 1)h$. Therefore the number R of such edges created in all phases is indeed $O(K \log n)$ as claimed, giving the $O(K \log^2 n)$ bound for maintaining a forest solution.

We choose $t = \max(2, \min(2n + 1, K/l^2))$, so that the dynamic-tree size bound is reduced to a constant as the length l of the layered network approaches $\sqrt{K}$; intuitively, later phases push less flow, and so dynamic trees that can handle the interaction between the different paths where flow is sent become less necessary. By the bound in the preceding paragraph and the fact that $\log t \leq \log(2n + 1)$, the $O(lh \log t)$ term of the time spent in each phase gives an $O(K \log^2 n)$ bound overall. The remaining term is $O((m + \lfloor 2n/t \rfloor h) \log t)$. For the phases with $l \geq L = \sqrt{K}$, we have $t = 2$ and $\sum_{l \geq L} h \leq K/L = \sqrt{K}$. The number of such phases is, as a result, also at most $\sqrt{K}$. The term is then $O(m + nh)$ for each phase and $O(m\sqrt{K})$ for all phases $l \geq \sqrt{K}$.

For the phases with $1 \le l < \sqrt{K}$, we again consider intervals of the form $L \le l < 2L$, where L is of the form $L = \sqrt{K}/2^i$ with $i \ge 1$. Then

$$\sum_{L \le l < 2L} (m + \lfloor 2n/t \rfloor h) \log t \le (Lm + 2n((2L)^2/K)(K/L)) \log(K/L^2)$$
$$\le (m\sqrt{K}/2^i + 8n\sqrt{K}/2^i)(2i).$$

Adding this expression for all $i \ge 1$, we obtain an $O(m\sqrt{K})$ bound for all phases $1 \le l < \sqrt{K}$.

As a last observation, note that the tree-size bound $t = K/l^2$ depends on the unknown optimal flow value K. However, if the flow at the end of the first phase is K', then since $l \ge 1$ at that point, the flow in subsequent phases will increase by at most $K'/l \le K'$, so $K/2 \le K' \le K$. Thus the flow K' at the end of the first phase provides an adequate estimate for K. The first phase itself is run with t growing from 1 to at most K'^2, with $t = i^2$ for the ith path. The term for the first phase (which has $l = 0$ and $h = K'$) is thus

$$O((m + (\lfloor 2n/1^2 \rfloor + \lfloor 2n/2^2 \rfloor + \lfloor 2n/3^2 \rfloor + \cdots))(\log K'))$$
$$= O((m + n) \log K') = O(m \log K).$$

THEOREM 3.1. *A maximum flow of value K in an uncapacitated network with m edges can be obtained in $O(m\sqrt{K} + K \log^2 m)$ time. This gives an $O(m\sqrt{K})$ time bound for $K = O((m/\log^2 m)^2)$.*

As presented, the algorithm gives the solution in terms of direct edges from supplies to demands, each of which corresponds to an infinite capacity path in the graph. If we want to know what the actual flow along the edges of the original graph is, we need to map the direct edges back to the path to which they correspond. This path can be recovered (and the corresponding flow assigned to it) by using the dynamic-tree structure that represented the path when the direct edge was first discovered. This structure may no longer exist at the end of the algorithm, and a re-execution of the algorithm to recreate the structure and assign flow to the corresponding paths will then be necessary. For most applications, knowledge of the actual infinite capacity path followed from a supply to a demand is not needed.

Note that if $K = O(m)$, then it is not necessary to miaintain a forest solution to ensure an $O(m)$ bound on the number of residual edges, and furthermore the bound on the running time holds even if the $O(\log t)$ time used to implement dynamic trees is increased to $O(\sqrt{t}/\log^2 t)$.

In view of known bounds for network flow, an $O(m\sqrt{K})$ bound is only of interest if $K \le (n \log n)^2$. This leaves open a relatively narrow range of values for K where the $O(m\sqrt{K})$ bound cannot at present be achieved (namely, $\sqrt{K} \approx n \approx m$, up to logarithms).

4. Maximum Flow when the Width Is Small. This section describes a simple and efficient flow algorithm for networks of small width. We describe the algorithm in the framework of capacitated networks (with edges of finite or infinite capacity).

The algorithm uses capacity scaling. Starting from a network G, each finite capacity c is replaced by $\lfloor c/2 \rfloor$, and a maximum flow in this modified network G' is obtained. By doubling this flow, we obtain a maximum flow in a network G'' with capacities $2\lfloor c/2 \rfloor$. If we consider a corresponding minimum cut X, $\bar{X}$ in G'', then all the edges across the cut must be saturated, and must therefore have finite capacity. Therefore the number of such edges is at most the cut width w. We now restore the original capacities c of the network G. This increases all capacities by either 0 or 1, depending on whether c is even or odd. Therefore the capacity of the cut X, $\bar{X}$ increases by at most w. It follows that a maximum flow in G differs from the maximum flow in G'' by at most w units of flow. We may thus perform w augmentations, each running in $O(m)$ time, to obtain a maximum flow in G.

We have reduced a flow problem with optimum flow K to a flow problem with optimum flow at most $K/2$, by means of the reduction from G to G'. This reduction has time complexity $O(wm)$ and is referred to as a *scaling phase*. After $(\log K) + 1$ such scaling phases, we are left with a network with maximum flow value 0; when this network is reached, it is easily recognized in $O(m)$ time. An optimum flow for the original network can now be reconstructed in $O(wm)$ time per scaling phase, for a total $O(wm \log K)$ time. By Lemma 2.3, uncapacitated networks can be reduced to capacitated networks without affecting the size or maximum flow value of the given instance, provided that the notion of *explicit width* is used for the uncapacitated network.

THEOREM 4.1. *A maximum flow of value K in a capacitated (uncapacitated) network with m edges and of cut width (resp. explicit width) w can be found in $O(wm \log K)$ time.*

5. Uncapacitated Blocking Flow. We now turn our attention to the problem of finding blocking flows in an uncapacitated network. Our algorithm is based on the $O(m \log n)$ algorithm (for acyclic networks) of [31] and [34] that uses dynamic trees. We also use the bounded tree size idea from the $O(m \log(n^2/m))$ algorithm of [11], but do not require finger search trees due to the fact that all capacities are infinite. Infinite capacities also make it possible to replace the parameter n by the potentially smaller explicit width w, thus obtaining an $O(m \log(w^2/m + 2))$ algorithm. The given graph G has infinite capacities, supplies, and demands. Recall that we can assume without loss of generality that the graph is acyclic in the infinite capacity case. We are also given a set of w vertex-disjoint paths P_1, $P_2, \ldots, P_w$ in G that jointly cover all the vertices in G.

The algorithm proceeds by performing a series of depth-first searches on G, starting at different vertices of G. The depth-first searches are used to send flow from supplies to demands; we first concentrate on the rules that guide the execution of these searches, and only later indicate how these searches are used to send flow. A depth-first search retreats from a vertex v only if it succeeds in saturating the demand at v (this saturation is described later). Otherwise, the depth-first search stops (and the next depth-first search is started). As a result, if the depth-first search retreats from v, then the demand at v and at every vertex reachable from v is saturated, so no flow can be sent through v, and we can remove the vertex v together with its incoming edges from G. This also implies that a vertex v is removed from G only after all the vertices reachable from v have been removed from G. In particular, the vertices on a path P_i that have not been removed from G always constitute an initial segment of P_i. We denote the current initial segment by Q_i and the first and last vertices of Q_i by u_i and v_i, respecively. When the depth-first search advances from a vertex v, it always proceeds along an edge visited by an earlier depth-first search, if such an edge exists. As a result, for every vertex v remaining in G, there is always at most one edge out of v that has been traversed so far. This implies that the visited edges form a collection of in-trees.

In choosing an edge out of a vertex v, we always give priority to the edge joining v to the vertex following v on the path Q_j that v belongs to, if $v \neq v_j$. As a result, there is always at most one visited edge out of Q_j, and this edge is infact an edge (v, w) out of $v = v_j$. If w belongs to $Q_{j'}$, then we view this visited edge (v_j, w) as an edge joining Q_j to $Q_{j'}$. This implies that the visited edges joining two different Q_j form a collection of in-trees on the set of Q_j. At any point during the execution of a depth-first search, there is a single path R contained in the edges visited by the depth-first search joining the vertex where the search started to the current vertex; the remaining edges visited by the depth-first search have been removed from G because a retreat was performed through them. We always start a depth-first search at the vertex u_i of some Q_i; by the priority rule, the current path R will immediately and subsequently contain the initial segment Q_i. We can now specify how a vertex v is saturated before the depth-first search retreats. This is done simply by sending flow from the lowest unsaturated supply vertex u in Q_i to the vertex v, along R, so that either the supply at u or the demand at v is saturated. Therefore, when the depth-first search terminates because it cannot saturate the demand at v, all the supply vertices on P_i will have been either saturated or removed from G. The total number of depth-first searches performed is at most w, one for each P_i. By the time all these searches terminate, all supply vertices will have been either saturated or removed from G. The total number of depth-first searches performed is at most w, one for each P_i. By the time all these searches terminate, all supply vertices will have been either saturated or removed from G, and in either case no additional flow can be sent from them to an unsaturated demand, so we indeed have a blocking flow.

To implement this algorithm efficiently, we represent the in-trees joining the Q_j using dynamic trees. The nodes in the dynamic trees correspond to paths Q_j. Each path Q_j is represented separately by a doubly linked list, and we also remember the end-vertex v_j of Q_j. When a depth-first search advances from a vertex v in a Q_j, it can always go directly to the vertex v_j, so the edges contained in Q_j are never visited when the search advances. In fact, the depth-first search can proceed directly to the vertex $v_{j'}$ of the root $Q_{j'}$ of the tree containing Q_j. Therefore, when the search advances, it only visits edges joining two different Q_j that were not visited by an earlier depth-first search, and adds each such edge to the dynamic trees so that later searches will not need to examine it when they advance. When the depth-first search retreats from the vertex v_j of a root Q_j, it removes v_j and its incident edges from G. An edge joining some $Q_{j'}$ to this root Q_j is of the form $(v_{j'}, w)$ with w in Q_j, and is only removed from G if $w = v_j$; therefore only some of the edges joining the root Q_j to its children $Q_{j'}$ are removed. The initial segment Q_j is also shortened when the last vertex v_j is removed, and v_j is updated accordingly. If the tree whose root is Q_j was entered by the depth-first search through a vertex v in some $Q_{j''}$, then the depth-first search retreats from the old v_j to the vertex $v_{j'}$ of the root $Q_{j'}$ of the tree containing $Q_{j''}$ after the updates (possibly $j' = j$).

Only a constant number of dynamic-tree operations are performed for each edge in G, namely, those performed when the edge is seen for the first time by the advance of a depth-first search, and those performed when the edge is seen for the last time and removed from G. Therefore the number of dynamic-tree operations is $O(m)$, and each operation takes $O(\log(w + 1))$ time because the total number of nodes in the dynamic trees is at most w. This gives an $O(m \log(w + 1))$ time bound.

To improve on this time bound, we introduce a bound t on the maximum dynamic-tree size, and never link two Q_j if this linking would result in a dynamic tree with more than t nodes. Note that if a link is not performed, then one of the two trees involved must have at least $t/2$ nodes. We must now account for the traversal of an edge e by the advance of the depth-first search that does not result in a link. The edge e goes from the root Q_j of a tree to a node $Q_{j'}$ of a tree rooted at some $Q_{j''}$. If the depth-first search later retreats from the vertex $v_{j''}$ of $Q_{j''}$, then we charge the forward traversal of e to this retreat; otherwise we charge the traversal of e to one of the two trees involved, whichever is of size at least $t/2$. Note that in this second case the charged tree remains unchanged until the end of the depth-first search. There are at most $2w/t$ such trees, and each is charged by at most two edges, one where the depth-first search enters the tree and one where the depth-first search leaves the tree at its root. Therefore every such tree is charged a constant number of dynamic-tree operations by the depth-first search, and we have an additional $O(w/t)$ dynamic-tree operations per depth-first search for a total of $O(w^2/t)$ dynamic-tree operations over all w searches. Each dynamic-tree operation takes $O(\log(t + 1))$ time, giving a total $O((m + w^2/t) \log(t + 1))$ time bound. Letting $t = w^2/m + 1$ gives an $O(m \log(w^2/m + 2))$ time bound.

THEOREM 5.1. *A blocking flow in an uncapacitated network with m edges and explicit width w can be found in* $O(m \log(w^2/m + 2))$ *time.*

Note that if $m = \Omega(w^2)$, then we can let $t = 1$ and still obtain a linear-time algorithm without dynamic trees. The efficiency of this $O(m + w^2)$ algorithm comes solely from the fact that we avoid traversing paths P_j by always jumping to the end-vertex of the path. This simplifies the implementation considerably.

Since the edges inside the paths P_j are skipped, the algorithm (with or without dynamic trees) remembers the flow only along edges that join two different paths P_j and $P_{j'}$. If at the end of the algorithm we need to know the flow for the edges inside a path P_j, we can just traverse P_j from its start-vertex and use flow conservation at each vertex to infer the flow along the edges of P_j. This can be done in $O(m)$ time.

6. 2-Satisfiability. In this section we show how the results from the previous sections can be applied to solve several questions related to the 2-SAT problem. A 2-SAT instance is a set of boolean variables $\{x_1, x_2, \ldots, x_n\}$ and a set of m clauses $u \vee v$, where each of the two literals u and v is either a variable x_i or its negation $\overline{x_i}$. A solution to a 2-SAT instance is an assignment of boolean values to the boolean variables x_i such that all the clauses are satisfied. It is sometimes convenient to view the clauses as two implications $\bar{u} \to v$ and $\bar{v} \to u$. This can in turn be represented by an *implication graph* with vertices x_i, $\overline{x_i}$ and directed edges $(\bar{u}, v)$ and $(\bar{v}, u)$ corresponding to each clause. The *width* of the 2-SAT instance is then simply the width of this directed graph (as defined for uncapacitated flow graphs). We also use here the terms implicit and explicit width. The transitive closure of a 2-SAT instance I is the 2-SAT instance whose implication graph is the transitive closure of the implication graph of I. By Lemma 2.2, we have:

THEOREM 6.1. *All the clauses in the transitive closure of a* 2-SAT *instance which involve at least one literal from a given path P can be obtained in $O(m)$ time. Thus the transitive closure can be obtained in $O(wm)$ time for a* 2-SAT *instance of explicit width w.*

The *compatibility graph* of a 2-SAT instance is an undirected graph on the set of literals with an edge (u, v) for each clause $u \vee v$ of the 2-SAT instance. We assume that the trivial edges $(u, \bar{u})$ are always present. A partial solution to a 2-SAT instance is an assignment of values to a subset of the variables that can be extended to a complete solution by some assignment of values to the remaining variables. A partial assignment can be represented by a subset S of the vertices in the compatibility graph: if a literal u has been assigned the value 1, then the vertex u is included in S (thus $\bar{u}$ is included in S if u has been assigned the value 0); if u has not been assigned a value, then both u and $\bar{u}$ are included in S. A vertex cover in a graph is a set of vertices S with the property that at least one of the two endpoints of every edge is in S. Many results on the 2-SAT problem depend implicitly on the following observation:

LEMMA 6.1. *The partial solutions to a solvable* 2-SAT *instance are the vertex covers of the compatibility graph of its transitive closure.*

PROOF. Transitive closure in the implication graph corresponds to closure under the *resolution* rule in the compatibility graph. That is, if there is a clause $u \vee v$ and a clause $\bar{v} \vee w$, then there is also a clause $u \vee w$.

A partial solution to the 2-SAT instance corresponds to a vertex cover of the compatibility graph because at least one literal of each clause involving two variables with assigned values in the partial solution is satisfied. To prove the converse, suppose that we have a vertex cover. Note that all vertex covers correspond to partial assignments because they include at least one of the two complementary literals u, $\bar{u}$ (given the presence of the edge $(u, \bar{u})$). We show that if we reduce the vertex cover to a minimal vertex cover (by repeatedly removing vertices from the cover while preserving the vertex-cover property until no more vertices can be removed), then the corresponding extended partial assignment is indeed a complete assignment and hence a solution to the 2-SAT instance (since all the clauses are satisfied by the vertex-cover property). Thus the partial assignment was indeed a partial solution. Suppose then, toward a contradiction, that we have a minimal vertex cover in which, for some literal v, both v and $\bar{v}$ are in this cover. Consider first the case where there is no self-loop edge (v, v) or $(\bar{v}, \bar{v})$. By minimality, v cannot be removed from the cover, so an edge (u, v) with u not in the cover must exist. Similarly, $\bar{v}$ cannot be removed from the cover, so there is an edge $(\bar{v}, w)$ with w not in the cover. However, then, by resolution, there is an uncovered edge (u, w), a contradiction. If there is one self-loop, say the edge $(\bar{v}, \bar{v})$, but not the edge (v, v), then we can still conclude that there is an edge (u, v) with u not in the cover, and then resolution yields edges $(u, \bar{v})$ and (u, u), so u must be in the cover, a contradiction. If both self-loops (u, u) and $(\bar{u}, \bar{u})$ are present, then the 2-SAT instance has no solution. Therefore, if a solution exists, then every minimal vertex cover contains only one of v, $\bar{v}$ for each such pair, and hence defines a 2-SAT solution. □

Note that the preceding proof also shows that the complete solutions to a solvable 2-SAT instance are the *minimal* vertex covers of the compatibility graph of its transitive closure. The lemma implies that given a partial assignment to k variables of a transitively closed 2-SAT instance, it can be checked whether this partial assignment is a partial solution by just checking the clauses involving these k variables (at most $O(k^2)$ clauses). The reason is that the corresponding set S contains both u and $\bar{u}$ for every unassigned u, so all edges involving unassigned literals are automatically covered, and only edges involving the remaining $2k$ literals need to be checked. Using the transitive-closure result of Theorem 6.1, we obtain the following:

THEOREM 6.2. *Given a* 2-SAT *instance with m clauses and explicit width w, it can be determined, after $O(wm)$ preprocessing time, whether a query assignment to k of the variables is a partial solution, in $O(k^2)$ time.*

7. Optimization on 2-SAT Instances.

We now consider the *minimum-weight* 2-SAT problem. An instance of this problem is a 2-SAT instance with a non-negative weight associated with each variable. The weight of a solution is the sum of the weights of the true variables (variables of value 1 in the solution). In the minimum-weight 2-SAT problem, the aim is to find a solution of minimum weight for a 2-SAT instance. If we consider the compatibility graph of its transitive closure, with weights assigned to the vertices, and define the weight of a vertex cover to be the sum of the weights of vertices in the cover, then the minimum-weight 2-SAT problem can be viewed as a minimum-weight minimal vertex-cover problem.

Unfortunately, the minimum-weight vertex-cover problem is $\mathcal{NP}$-complete, even if all weights are 1. This also applies to the minimum-weight 2-SAT problem, because a graph on vertices x_i can be viewed as a 2-SAT instance on the variables x_i, with clauses $x_i \vee x_j$ corresponding to edges (x_i, x_j) and where the variables x_i inherit their weight from the graph (the 2-SAT instance is *monotone* and transitively closed). Therefore the minimum-weight 2-SAT problem is also $\mathcal{NP}$-complete.

On the other hand, algorithms exist for approximating the minimum-weight vertex cover within a factor of 2 of the minimum weight, and these algorithms thus give solutions within a factor of 2 for the minimum-weight 2-SAT problem [2], [3], [17], [20], [25]. We describe one algorithm for the minimum-weight 2-SAT problem in terms of blocking flows, and this enables us to use the efficient blocking-flow algorithm from the last section.

The algorithm is as follows. Given a weighted 2-SAT instance, we consider its implication graph. In this graph we assign a supply to $\bar{x}$ equal to the weight of x and a demand to x equal to the weight of x, for each variable x; the edges are given infinite capacities. We now find a *symmetric blocking flow* in this uncapacitated network; symmetric here means that the flow out of supply vertex $\bar{x}$ equals the flow into demand vertex x. The set S of saturated demands x together with all vertices $\bar{x}$ then gives a partial solution to the 2-SAT instance, which can be extended to a complete solution.

We first prove the correctness of the algorithm, and then discuss its implementation. If (x, y) is an edge of the compatibility graph of the transitive closure, then there is a path from $\bar{x}$ to y in the implication graph, with infinite capacity edges. Therefore a blocking flow must saturate the supply vertex $\bar{x}$ or the demand vertex y, and this means by the symmetry condition that either x or y must be saturated. Therefore the set S is a vertex cover of the compatibility graph of the transitive closure, and hence a partial solution by Lemma 6.1. Extending it to a complete solution means reducing S to a minimal vertex cover, and this can only reduce the weight of S,

The weight of S is at most the value of the blocking flow. The reason is that the weight of S is the sum of the saturated demands, i.e., the sum of the flow into saturated demand vertices, while the value of the flow is the sum of the flow into all demand vertices. On the other hand, let T be a solution to the 2-SAT instance represented as a vertex cover of the compatibility graph of the transitive closure. Charge the amount of flow from a supply $\bar{x}$ to a demand y along some path to one or the other of these two vertices, depending on whether x or y is covering the edge (x, y) in T. The total flow is thus charged to supplies $\bar{x}$ or demands x such that x is in T, and neither is ever charged more than its corresponding supply or demand, which equals the weight of x. This shows that the total flow is never more than twice the weight of T. We have therefore proved that the weight of S is at most twice the weight of T, and so the solution obtained from S has weight at most twice the weight of any other solution, as desired.

To implement this algorithm we use the blocking-flow algorithm from Section 5. To ensure the symmetry condition, whenever we send flow in the implication graph from supply $\bar{x}$ to demand y, we also send the same amount of flow from $\bar{y}$ to x (in the case $x \neq y$). This modification can be easily incorporated into the algorithm and does not affect the $O(m \log(w^2/m + 2))$ time bound. The saturated demands x define a partial solution. We can extend a partial solution to a complete solution using any $O(m)$-time algorithm for 2-SAT (e.g., see [6]). We therefore have:

THEOREM 7.1. *A solution to the minimum-weight* 2-SAT *problem with weight within a factor of* 2 *of the optimum can be obtained in* $O(m \log(w^2/m) + 2)$ *time for instances with m clauses and explicit width w.*

An $O(nm)$ algorithm was given by Gusfield and Pitt [18].

Note. If the given 2-SAT instance is transitively closed, then we only need to send flow from a supply $\bar{x}$ to a demand y along a single edge (rather than a path). We can then enforce the symmetry condition by working directly with the undirected edge (x, y) of the compatibility graph rather than the two directed edges $(\bar{x}, y)$ and $(\bar{y}, x)$ of the implication graph. Each edge is considered once and assigned a flow that saturates either x or y, giving an $O(m)$ algorithm. This simple linear algorithm can be applied to the minimum-weight vertex-cover problem, because the corresponding 2-SAT instance (see above) is always transitively closed. See also [2] and [18].

A 2-SAT instance is *bipartite* if the corresponding compatibility graph is a bipartite graph $G = (U \cup V, E)$. Note that for each pair of complementary literals u, $\bar{u}$, one of them must be in U and the other one in V, since the edge $(u, \bar{u})$ is in G. The fact that G is bipartite also tells us that every edge (u, v) in the implication graph must join either two vertices in U or two vertices in V, and that implications $u \rightarrow v$ in U correspond to impliations $\bar{v} \rightarrow \bar{u}$ in V. We therefore restrict the implication graph to the subgraph induced by the vertices in V, which always contains exactly one of each pair of complementary literals u, $\bar{u}$. If $u \in U$, then setting $u = 0$ must be interpreted as setting $\bar{u} = 1$ in V (and setting $u = 1$ as setting $\bar{u} = 0$ in V).

The minimum-weight solutions to a bipartite weighted 2-SAT instance can be obtained as follows. For each variable $x \in U$, assign a supply to $\bar{x}$ (in V) equal to the weight of x. For each variable $y \in V$, assign a demand to y equal to the weight of y. Then find a maximum flow in the implication graph of the 2-SAT instance with infinite capacities (in V). Now augment the 2-SAT instance by adding the following constraints: If $\bar{x}$ is an unsaturated supply vertex, set $x = 0$. If y is an unsaturated demand vertex, set $y = 0$. If there is positive flow along an edge $u \rightarrow v$ in the implication graph, then add the constraint $v \rightarrow u$ to the 2-SAT instance. The minimum-weight solutions to the original 2-SAT instance are then precisely the solutions to the modified 2-SAT instance. Thus a particular minimum-weight solution can be obtained by solving the modified 2-SAT instance. Note that the modified instance is simpler: some variables have been replaced by constants, and the literals u and v for which a constraint $v \rightarrow u$ has been added must now satisfy $u = v$ and can thus be replaced by a single literal.

To prove the correctness of this algorithm, recall the definition from Section 2 of uncapacitated flow problems in terms of capacitated problems with a source s, a sink t, and capacitated edges $(s, \bar{x})$ (resp. (y, t)) for supply vertices $\bar{x}$ (resp. demand vertices y). We can view the solution to the 2-SAT instance as cuts in this network by setting $s = 1$ and $t = 0$, while using for the remaining vertices in V their values assigned as literals in the solution. Furthermore, the weight of a solution is the capacity of the corresponding cut. To see this, note that an edge (u, v) in the network with $u = 1$ and $v = 0$ cannot be an infinite capacity edge, because it would then be an edge (u, v) in the implication graph and the constraint $u \rightarrow v$ in the 2-SAT instance would be violated. For capacitated edges, we have $u = 1$ and $v = 0$ iff (u, v) is $(s, \bar{x})$ and $x = 1$, or (u, v) is (y, t) and $y = 1$. Thus the true variables (variables set to 1) correspond to the edges across the cut, with the weight of the variable corresponding to the capacity of the edge, proving the claim.

Therefore the least possible weight for a solution is the weight of a minimum cut. By the max-flow min-cut theorem, given a maximum flow, the minimum cuts are the cuts such that if $(u, v) = (1, 0)$, then the edge (u, v) is saturated, and if $(u, v) = (0, 1)$, then the edge (u, v) has no flow [34]. For uncapacitated edges (u, v), this means that we cannot have $(u, v) = (1, 0)$, since these edges cannot be saturated; thus all the implications $u \rightarrow v$ from the implication graph must be satisfied (implying that all minimum cuts are solutions to the 2-SAT instance). Furthermore, if there is a positive flow along (u, v), then we cannot have $(u, v) = (0, 1)$, i.e., the additional constraint $v \rightarrow u$ must also be satisfied. For capacitated edges $(s, \bar{x})$ (which have $s = 1$), if this edge is not saturated (the supply $\bar{x}$ is not saturated), then we cannot have $\bar{x} = 0$, i.e., we must have $x = 0$. Similarly, for capacitated edges (y, t) (which have $t = 0$), if this edge is not saturated (the demand y is not saturated), then we cannot have $y = 1$, i.e., we must have $y = 0$. These conditions characterize the minimum cuts and therefore the minimum-weight solutions to the 2-SAT instance. From the flow algorithm of Sections 3 and 4 we obtain:

THEOREM 7.2. *A minimum-weight solution of weight K for a bipartite weighted* 2-SAT *instance with m clauses and explicit width w can be found on $O(m\sqrt{K})$ time for $K = O((m/\log^2 m)^2)$ and in $O(wm \log K)$ time for arbitrary K. In fact, a complete description of all minimum-weight solutions can be found within this time bound.*

The link between bipartite 2-SAT and max flow can also be implicitly found in [23]. By Lemma 6.1, it can also be viewed as the well-known link between vertex cover and matching for bipartite graphs (see [19]).

8. Enumeration of 2-SAT Solutions.

The last section looked at the problem of finding particular solutions to a 2-SAT instance. This section examines the problem of finding *all* solutions. Given a 2-SAT instance, we can run a strong components algorithm to transform it into an acyclic instance (an instance with an acyclic implication graph) in $O(m)$ time. Two literals in the same strong component of the implication graph have the same value in all solutions, and can therefore be treated as a single literal. We assume that the 2-SAT instance contains no implications of the form $u \rightarrow \bar{u}$. If it does, then we may set $u = 0$ and remove all occurrences of u from the 2-SAT instance. This type of implication may remain present in a hidden form, as a chain of implications such as $u \rightarrow v \rightarrow \bar{u}$; detecting this would require executing a transitive-closure algorithm. Fortunately, these hidden occurrences are of no consequence in the solution given below.

The *maximum degree* d of a 2-SAT instance in this *acyclic* form is the maximum over all literals u of the number of clauses of the form $u \rightarrow v$. If the 2-SAT instance has explicit width w, then we may assume that, for each path P in the corresponding path cover, and each literal u, there is at most one literal v in P such that the clause $u \rightarrow v$ is present in the 2-SAT instance. The reason is that if v' follows v on the path P, then the clause $u \rightarrow v'$ can be inferred from $u \rightarrow v \rightarrow v'$, and therefore does not need to be included in the 2-SAT instance. Hence $d \leq w$.

It is well known that a solution to a 2-SAT instance with m clauses on the variables $x_1, \dots, x_n$ can be found in $O(m)$ time. Given such a solution, we may rename all variables for convenience so that the given solution has $x_i = 0$ for all $1 \leq i \leq n$ (the all-zero solution). Once this is done, all clauses must be of the type $x_i \vee \overline{x_j}$ or of the type $\overline{x_i} \vee \overline{x_j}$. The clauses of the first type can be viewed as implications $\overline{x_i} \rightarrow \overline{x_j}$. These implications form an acyclic graph, so we can perform a topological sort (in $O(m)$ time) and rename the variables to ensure that $i < j$ for all such clauses. The clauses of the second type are symmetric in x_i and x_j and can therefore always be written as implications $x_i \rightarrow \overline{x_j}$ with $i < j$.

We therefore assume that the 2-SAT instance is given by implications of the form $\overline{x_i} \rightarrow \overline{x_j}$ or $x_i \rightarrow \overline{x_j}$ with $i < j$. We now observe the following property: given a solution $x = x_1 x_2 \cdots x_n$ with at least one $x_i = 1$, if we change the last such x_i (i.e., $x_i = 1$ and $x_j = 0$ for all $j > i$) to $x_i = 0$, then we obtain another solution. This property holds because if the new assignment violates some clause, it must be a clause involving x_i which forbids $x_i = 0$. This can only be a clause $\overline{x_i} \rightarrow \overline{x_j}$ with $x_j = 1$ and $j > i$, contradicting the assumption that x_i was the last variable equal to 1 in the given solution.

We refer to the solution obtained from a solution x by changing the last $x_i = 1$ to $x_i = 0$ as the *parent* of the given solution x. The solutions thus form a tree rooted at the known all-zero solution. We do not build this solution tree explicitly; however, the execution of the recursive enumeration algorithm corresponds to a depth-first search started at the root of the tree.

Given a current solution x, the *index* is the largest l such that $x_l = 1$. By convention, the index is 0 if x is the all-zero solution. Note that the children of x in the solution tree are the solutions that can be obtained from x by setting $x_j = 1$ for a single j greater than the index of x. We say that an implication (either $\overline{x_i} \rightarrow \overline{x_j}$ or $x_i \rightarrow \overline{x_j}$ with $i < j$) is *active* if the antecendent (either $\overline{x_i}$ or x_i) has value 1 in the current solution x. We say that a variable x_j is *active* if $\overline{x_j}$ is *not* the consequent of any active implication.

LEMMA 8.1. *The children of a solution x are precisely the solutions obtained from x by setting $x_j = 1$ for a single active x_j with j greater than the index of x.*

PROOF. The fact that x_j must be active is clear: if not, then $\overline{x_j}$ is the consequent of an active implication whose antecedent equals 1, and setting $x_j = 1$ violates this implication. On the other hand, it x_j is active, then all implications with consequent $\overline{x_j}$ are inactive (antecedent equal to 0), and setting $x_j = 1$ does not violate these implications. No implication of the form $x_j \rightarrow \overline{x_k}$ can be violated either, because such an implication has $k > j$ and therefore $x_k = 0$ by the definition of index. □

Therefore, in order to determine the children of x, it is sufficient to maintain a list of the active variables x_j, ordered by the value of j. When we set $x_i = 1$ for some i, the implications of the form $x_i \rightarrow \overline{x_j}$ become active and those of the form $\overline{x_i} \rightarrow \overline{x_j}$ become inactive (the opposite happens if we set $x_i = 0$). We can maintain a count of the number of active implications that have $\overline{x_j}$ as a consequent, for each x_j. When this count becomes zero, the variable x_j becomes active and is added to the active list. When the count becomes nonzero, the variable x_j becomes inactive and is removed from the active list.

The algorithm maintains globally the current solution x, the active list of variables L, and a count for each x_j as explained above. Initially, x is the all-zero solution, and L and the counts are set appropriately. The algorithm starts with a call to *enumerate*(0, 0). (Ignore for now the output calls and the depth argument.) The two assignments to x_i within *enumerate* are meaningless in the top-level call which has *index* = 0 by convention, and they can be ignored in that special case.

```
procedure enumerate(index, depth);
begin
  set $i$ = index and $x_i = 1$;
  if depth is even then output the current $x$;
  update the active implications with antecedent $x_i$ or $\overline{x_i}$;
  determine those $x_j$ that have just become active or have just become
    inactive;
  remove from the active list $L$ the $x_j$ that have just become inactive,
    but remember their position in $L$ (their predecessor in $L$) in an
    auxiliary local list $N$;
  let $M$ be the ordered list of $x_j$ that have become active;
  merge $M$ into the ordered list $L$ by traversing both lists in reverse
    order;
    as each variable $x_j$ is inserted in the merged list, call
      enumerate($j$, depth + 1);
    stop when $j = i$ has been reached;
  if depth is odd then output current $x$;
  set $x_i = 0$, and restore the active list by adding the $x_j$ from $N$ and
    removing those from $M$;
  update the active implications with antecedent $x_i$ or $\overline{x_i}$;
end;
```

The above procedure can be implemented so that it takes $O(d)$ time at the beginning and at the end of each call, plus an additional $O(1)$ in between recursive calls (ignore for now the output calls). To see this, note that the number of active implications to be updated in the beginning is at most $2d$; the corresponding count update then tells us which variables should become active or inactive (again at most $2d$). The variables that become inactive can be removed from L in $O(1)$ time per variable if L is maintained as a doubly linked list. Thus the operations before the merge take $O(d)$ time. As we merge the two lists, we do a recursive call for each element in the merged list, thus spending $O(1)$ time between recursive calls. Since the merge is done starting from the end of the lists, the merged L is always correct from x_j on, and this is the only portion of L that is used inside the recursive call. The operations after the merge again take $O(d)$ time.

The reason for outputing the even-depth solutions at entry time, and the odd-depth solutions at exit time, is that after a solution x is output, the next solution to be output will be at most the third solution visited after x. This can be easily verified: Suppose that an even-depth node e_1 has just been output. If e_1 has a child o_1, then either o_1 has no children and is immediately output, or o_1 has a child e_2 that is immediately output. If e_1 has no children, then we return to the parent o_2 of e_1, and again either o_2 has no children following e_1 and is immediately output, or its next child e_2 is immediately output. The other case is that of an odd-depth node o_1 which has just been output. Then the algorithm returns to the parent e_1 of o_1. If e_1 has a next child o_2, then either o_2 has no children and is immediately output, or it has a child e_2 that is immediately output. If e_1 has no next child, then the algorithm returns to the parent o_3 of e_1, which again has either no next child and is immediately output, or has a child e_3 which is immediately output. Thus, in all cases, the next node to be output is one of the next three nodes to be visited.

Since the algorithm spends $O(d)$ time at each node before moving to an adjacent node, the time between outputs is $O(d)$. Since adjacent solutions in the search tree differ in only one bit x_i, solutions that are consecutively output differ in at most three bits, and it is sufficient to output the values of the three bits that have changed (in constant time). The space used at each node is proportional to the number of clauses involving x_i for the current index i, and is therefore $O(m)$ overall. The information maintained globally also takes $O(m)$ space.

Theorem 8.1. *The solutions to a* 2-SAT *instance with m clauses and maximum degree d can be enumerated after $O(m)$ preprocessing time in $O(d)$ on-line time per solution, using $O(m)$ space.*

9. Applications to Stable Matching.

The stable-matching problem is the problem of pairing up people in a given set so that certain preference constraints are satisfied [9], [15], [24], [29]. Gusfield characterized the set of stable solutions of a stable-matching problem in terms of a partial order of dual rotations, and showed that this partial order can be found in $O(wm \log w)$ time for instances with w people and total preference list length m [13], and in $O(m)$ time for the bipartite version of the problem [12]. Subramanian showed that the stable-matching problem can be viewed as a special case of the boolean network stability problem on the class of *adjacency-preserving* networks. For stable-matching instances, the boolean variables in the corresponding network are variables x_{ij}, where i is a person and j is a position in the preference list of person i, with $0 \le j \le l(i)$, where $l(i)$ is the length of the preference list of person i. In a stable solution, we have $x_{ij} = 1$ if and only if person i is not matched to any of his first j choices [32].

Following this work, this author showed that the stable solutions for adjacency-preserving networks can be characterized by a 2-SAT instance on the corresponding boolean space. Thus the stable solutions to a stable-matching instance can be characterized by a 2-SAT instance on the variables x_{ij}, whose values are then interpreted as described above. Furthermore, this instance can always be found in $O(m)$ time [7], [8]. The characterization of Gusfield mentioned above can in fact be interpreted in this framework.

If we let $m = \sum_i l(i)$ be the size of the stable-matching instance, then clearly the number of x_{ij} variables is $O(m)$. It has been shown (both in the context of Gusfield [12] and of Feder [7]) that the number of clauses of the 2-SAT instance is $O(m)$ as well. On the other hand, if w is the number of people in the stable-matching instance, then the 2-SAT instance has explicit width $O(w)$. To see this, note that $x_{ij} \to x_{i(j-1)}$ is a valid clause, that is, if person i is not matched to any of his first j choices, then person i is not matched to any of his first $j - 1$ choices either. Thus the variables x_{ij} for a fixed i form a path, and the total number of paths to cover all literals is proportional to the number of people. In the case of complete preference lists, we have $m \approx w^2$.

The fact that person i is matched to his jth choice in a stable solution can be stated as $(x_{i(j-1)}, x_{ij}) = (1, 0)$. Thus a simultaneous pairing for a set of k people can be described by assigning values to at most $2k$ boolean variables. Theorem 6.2 then gives:

THEOREM 9.1. *In a stable-matching instance with w people and of size m, it can be determined, after $O(wm)$ preprocessing time, whether a query pairing for k people is stable, in $O(k^2)$ time.*

In fact, if the k people (paired-up with some other arbitrarily chosen k people) are chosen from a fixed set of k' people, then the preprocessing time can be reduced to $O(k'm)$. This is so because in that case the appropriate x_{ij} variables all belong to the k' paths corresponding to the fixed k' people i.

In the *weighted stable-matching* problem, there is a nonnegative weight w_{ij} associated with each entry j in the preference list of person i. These weights are nondecreasing in j for i fixed. The weight of a stable solution is the sum of the weights w_{ij} such that person i is matched to his jth choice in the stable solution. The goal is then to find a stable solution of minimum weight. The weight of a solution can be equivalently described as the sum of the weights of the variables x_{ij} such that $x_{ij} = 1$, where the weight of a variable x_{ij} is set to $w_{i(j+1)} - w_{ij}$, and $w_{i0} = 0$ by convention. The *egalitarian stable-matching* problem is the weighted stable-matching problem with $w_{ij} = j$. Thus each variable x_{ij} has weight 1.

The weighted stable matching can then be viewed as a weighted 2-SAT problem. Indeed, it can be shown that even the egalitarian stable matching is $\mathcal{NP}$-complete and as hard to approximate as vertex cover [7], [8]. On the other hand, Theorem 7.2 gives the following:

THEOREM 9.2. *A solution to the weighted stable-matching problem with weight within a factor of 2 of the optimum can be found in $O(m \log(w^2/m + 2))$, where w is the number of people and m is the size of the instance.*

Note that in the typical case $m \approx w^2$, the running time is linear in the size of the instance, and the implementation is considerably simplified because dynamic trees are not needed (see Section 5).

The bipartite version of the stable-matching problem is the stable-marriage problem, where people are either men or women and can only list people of the opposite sex. In that case it can be shown that the corresponding 2-SAT instance becomes bipartite [7], [12]. Theorem 7.2 then gives an efficient algorithm for the problem.

THEOREM 9.3. *A minimum-weight stable marriage of weight K for an instance with w people of size m can be found in $O(m\sqrt{K})$ time if $K = O((m/\log^2 m)^2)$, and in $O(wm \log K)$ time for K arbitrary. The egalitarian case has $K \leq m$ and can thus be solved in $O(m^{1.5})$ time. The algorithm gives in fact a description of all minimum-weight solutions.*

Note that in the egalitarian case, where $K \leq m$, the $O(m\sqrt{K})$ algorithm of Section 3 does not require maintaining a forest solution (the number of capacitated residual edges will never exceed m), thus simplifying the implementation. Pittel [28] has shown that a random egalitarian-marriage instance with complete lists (i.e., $m \approx w^2$) has $K \approx 2w^{1.5}$, so the algorithm will often run in less than the $O(m^{1.5})$ time bound. The algorithm of Irving *et al.* [23] is also based on maximum flows; the time bounds are then obtained by using the $O(nm \log n)$ Sleator–Tarjan algorithm for the weighted case and the $O(mK)$ Ford–Fulkerson algorithm in the egalitarian case [31], [34]. An $O(m^{1.5}\sqrt{\log m})$ time bound for the egalitarian-marriage case was obtained by Ng [26].

We can also enumerate stable solutions using Theorem 8.1.

THEOREM 9.4. *The stable solutions to a stable-matching instance with w people and size m can be enumerated after $O(m)$ preprocessing time in $O(w)$ on-line time per solution, using $O(m)$ space.*

In the case $k = 1$, for a stable-marriage problem, the time to find all stable pairs can be reduced from the $O(wm)$ time of Theorem 9.1 to $O(m)$ [12]. It has in fact been shown that even in the roommates case, the problem can be reduced in $O(m)$ time to the question of finding all the *trivial* variables of the 2-SAT instance, i.e., the variables that have the same value in all solutions. This can be seen from the fact that if x_{ij} and $x_{i(j-1)}$ are both trivial, then there is a solution with $(x_{ij}, x_{i(j-1)}) = (1, 0)$ iff any arbitrary solution has this property; if at least one of them is nontrivial, then a solution of this kind can only exist if the two variables are not in the same strong component of the implication graph, and in that case there is a solution with $x_{ij} \neq x_{i(j-1)}$ which can only be $(x_{ij}, x_{i(j-1)}) = (1, 0)$. This has lead to the conjecture (in the case of complete preference lists) that an $O(m)$ algorithm can be obtained for finding all stable pairs in a general stable-matching problem; this was also conjectured for the egalitarian stable-marriage problem [15]. It was earlier believed by this author that such an algorithm for the stable-pairs problem did exist, in view of the fact that other questions (minimizing the weight of solutions, enumerating solutions) manage to avoid the explicit computation of a transitive closure. We now know, however, that for these two problems, the dependency on transitive closure and bipartite matching, respectively, is inherent, as stated below. Let the *partial transitive-closure* problem be the problem of determining, given a directed graph G with n vertices and m edges as well as an arbitrary set S of n edges *not* in G, which edges of S are in the transitive closure of G. Let the *bipartite matching with multiplicities* problem be the problem of determing, given a bipartite graph G with n vertices, m edges, and multiplicities $m(v)$ associated with the vertices adding up to at most m, the maximum size of a multiset of edges T such that each vertex v is an endpoint of at most $m(v)$ edges in T. (Thus the standard bipartite-matching problem is the case $m(v) = 1$ for all v.) It can be shown that the stable roommate pairs problem and the egalitarian-marriage problem are at least as hard, for instances with n people and lits of length m, as the partial transitive closure and the bipartite matching with multiplicities problems, respectively. Linear-time algorithms are therefore unlikely. On the other hand, if an approximation factor of 2 can be tolerated, then Theorem 9.2 gives an $O(m)$ algorithm for the egalitarian-matching problem in the case of complete preference lists; in fact, running r phases of an algorithm similar to the one in Section 3 will guarantee a $1 + 1/r$ approximation factor in $O(rm \log m)$ time. By contrast, for random instances, the man/woman optimal solutions are a factor of $\sqrt{w}/2 \log w$ away from the optimal egalitarian solution [28].

Other versions of stable matching can also be solved by the approach presented here. The *lexicographic stable-marriage* problem, for which an $O(wm^2 \log^2 w)$ algorithm was given by Irving *et al.* [23], can be solved in $O(w^{1/2}m^{3/2})$ time; the basic idea takes advantage of the fact that flow algorithms give not only a single solution, but also a characterization of all solutions. The *balanced stable-marriage* problem is $\mathcal{NP}$-complete but can be approximated within a factor of 2 in $O(m \log(w^2/m + 2))$ time. See [8] for details.

10. Conclusion and Open Problems.

We have presented two algorithms for the uncapacitated maximum-flow problem. The $O(m\sqrt{K})$ algorithm owes its efficiency to an implicit transitive-closure representation via dynamic trees. The gradual reduction of the dynamic-tree size ensures that the associated logarithmic cost averages out to a constant over the entire execution of the algorithm. The time bound currently applies up to values of K that are slightly larger than $(m/\log^2 m)^2$. An open problem is to extend the bound to all $K \leq w^2$. This would yield, in combination with the second algorithm, a stronger $O(\min(\sqrt{K}, w)m \log(K/w^2 + 2))$ bound. The second algorithm runs in $O(wm \log K)$ time. This complexity is achieved by combining capacity scaling with the notion of width. An open question is whether the dependency on K can be significantly reduced (as in [1] for example) or removed (as in the capacity-scaling min-cost flow algorithm of Orlin [27]) without increasing the main wm factor.

For the 2-satisfiability problem, besides the optimization results that follow from the network-flow approach, we have studied the problems of recognizing partial solutions and of enumerating all solutions. The latter has an $O(m + dS)$ time complexity if the total number of solutions is S. The question of whether counting (a #P-complete problem even in the bipartite case [30]) is easier than enumerating remains open. The fact that consecutive solutions found by the algorithm differ in only a constant number of bits suggests that faster algorithms might be achievable.

The common element in the various results is the use of the width of a graph, showing that combinatorial problems are sometimes easier in skinny graphs than in more general graphs. These graphs arise naturally in the context of stable matching. The approach may well be applicable to other graph problems whose structure is the superposition of a simple collection of paths with a more complex structure.

Acknowledgment. I would like to thank Serge Plotkin for helpful suggestions during this work.

References

[1] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan, Improved Time Bounds for the Maximum Flow Problem, Technical Report CS-TR-118-87, Department of Computer Science, Princeton University, 1987. (*SIAM J. Comput.*, to appear.)

[2] R. Bar-Yehuda and S. Even, A linear time approximation algorithm for the weighted vertex cover problem, *J. Algorithms*, **2** (1981), 198–203.

[3] K. Clarkson, A modification of the greedy algorithm for vertex cover, *Inform. Process. Lett.*, **16** (1983), 23–25.

[4] R. P. Dilworth, A decomposition theorem for partially ordered sets, *Ann. of Math.*, **51** (1950), 161–166.

[5] E. A. Dinic, Algorithm for solution of a problem of maximum flow in a network with power estimation, *Soviet Math. Dokl.*, **11** (1970), 1277–1280.

[6] S. Even, A. Itai, and A. Shamir, On the complexity of timetable and multicommodity flow problems, *SIAM J. Comput.*, **5** (1976), 691–703.

[7] T. Feder, A new fixed point approach for stable networks and stable marriages, *Proc. 21st ACM Symp. on Theory of Computing* (1989), pp. 513–522. (Submitted to *J. Comput. System Sci.*)

[8] T. Feder, Stable Networks and Product Graphs, Ph.D. dissertation, Technical Report STAN-CS-91-1362, Stanford University (1991).

[9] D. Gale and L. S. Shapley, College admissions and the stability of marriage, *Amer. Math. Monthly*, **69** (1962), 9–15.

[10] A. V. Goldberg and R. E. Tarjan, A new approach to the maximum flow problem, *Proc. 18th ACM Symp. on Theory of Computing* (1986), pp. 136–146.

[11] A. V. Goldberg and R. E. Tarjan, Finding minimum-cost circulations by successive approximation, *Math. Oper. Res.*, **15**(3) (1990), 430–466.

[12] D. Gusfield, Three fast algorithms for four problems in stable marriage, *SIAM J. Comput.*, **16**(1) (1987), 111–128.

[13] D. Gusfield, The structure of the stable roommate problem: efficient representation and enumeration of all stable assignments, *SIAM J. Comput.*, **17**(4) (1988), 742–769.

[14] D. Gusfield and R. W. Irving, The parametric stable marriage problem, *Inform. Process. Lett.*, **30** (1989), 255–259.

[15] D. Gusfield and R. W. Irving, *The Stable Marriage Problem: Structure and Algorithms*, MIT Press Series in the Foundations of Computing, MIT Press, Cambridge, MA (1989).

[16] D. Gusfield, R. Irving, P. Leather, and M. Saks, Every finite distributive lattice is a set of stable matchings for a small stable marriage, *J. Combin. Theory Ser. A*, **44** (1987), 304–309.

[17] D. Gusfield and L. Pitt, Equivalent approximation algorithms for node cover, *Inform. Process. Lett.*, **22**(6) (1986), 291–294.

[18] D. Gusfield and L. Pitt, A Bounded Approximation for the Minimum Cost 2-SAT Problem, Technical Report CSE-89-4, University of California, Davis (1989).

[19] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA.

[20] D. S. Hochbaum, Approximation algorithms for the set covering and vertex cover problems, *SIAM J. Comput.*, **11**(3) (1982), 555–556.

[21] J. E. Hopcroft and R. M. Karp, An $n^{5/2}$ algorithm for maximum matching in bipartite graphs, *SIAM J. Comput.*, **2** (1973), 225–231.

[22] R. W. Irving and P. Leather, The complexity of counting stable marriages, *SIAM J. Comput.*, **15**(3) (1986), 655–667.

[23] R. W. Irving, P. Leather, and D. Gusfield, An efficient algorithm for the optimal stable marriage, *J. Assoc. Comput. Mach.*, **34**(3) (1987), 532–543.

[24] D. E. Knuth, *Mariages stables et leur relations avec d'autres problèmes combinatories*, Les Presses de l'Université de Montréal, Montréal, Québec (1976).

[25] G. L. Nemhauser and R. E. Trotter, Vertex packing structural properties and algorithms, *Math. Programming*, **8** (1975), 232–248.

[26] C. Ng, An $O(n^3\sqrt{\log n})$ Algorithm for the Optimal Stable Marriage Problem, Technical Report 90-22, University of California, Irvine (1990).

[27] J. B. Orlin, A faster strongly polynomial minimum cost flow algorithm, *Proc. 20th ACM Symp. on Theory of Computing* (1988), pp. 377–387.

[28] B. Pittel, On likely solutions of a stable marriage problem, Manuscript.

[29] G. Pólya, R. E. Tarjan, and D. R. Woods, *Notes on Introductory Combinatorics*, Birkhäuser, Basel (1983).

[30] J. S. Provan and M. O. Ball, The complexity of counting cuts and of computing the probability that a graph is connected, *SIAM J. Comput.*, **12**(4) (1983), 777–788.

[31] D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, *J. Comput. System Sci.*, **26** (1983), 652–686.

[32] A. Subramanian, A New Approach to Stable Matching Problems, Technical Report STAN-CS-89-1275, Stanford University (1989).

[33] R. E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.*, **1** (1972), 146–160.

[34] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA (1983).