

A COMBINATORIAL ALGORITHM FOR WEIGHTED STABLE SETS IN BIPARTITE GRAPHS

GEREON FRAHLING* AND ULRICH FAIGLE†

Abstract. Computing a maximum weighted stable set in a bipartite graph is considered well-solved and usually approached with preflow-push, Ford-Fulkerson or network simplex algorithms. We present a combinatorial algorithm for the problem that is not based on flows. Numerical tests suggest that this algorithm performs quite well in practice and is competitive with flow based algorithms especially in the case of dense graphs.

Key words. Bipartite graph, stable set, weight, poset, antichain, tree algorithm

AMS subject classifications. 05C85, 90C27, 90C35

1. Introduction. The problem of finding an antichain of maximal weight in a (weighted) poset is (well-)known to be equivalent with the problem of computing a maximal weighted stable (or "independent") set of nodes in a bipartite graph. Using a network flow formulation, the latter can be solved in $O(n^{5/2})$ time in the unweighted case and $O(n^4)$ in the case of rational weights (where n denotes the number of nodes in the graph).

We propose here an algorithmic approach to the stable set problem that is *not* based on the formulation as a network flow problem. Our approach is motivated by the algorithm of Lerchs and Grossmann [2] ("LG-algorithm") for another problem: The computation of maximal ideals (a.k.a. "down sets") in weighted posets (see the analysis of Hochbaum [1]) or, equivalently, maximally weighted closures in directed graphs.

The LG-algorithm proceeds in a rather combinatorial way and, at first sight, does not seem to rely on flows in auxiliary graphs. Instead, it considers a spanning tree in the graph, solves the closure problem with respect to this spanning tree, adds and deletes edges to pass to a new spanning tree and finds a new optimal solution in that modified tree. Iterating this procedure the algorithm finally arrives at a tree solution that is feasible in the original graph (and hence overall optimal).

The algorithm we present here for the maximum weight stable set problem in bipartite graphs follows the same philosophy. We consider a spanning tree in the graph and determine a maximal stable set relative to this tree. If this solution is not feasible for the original problem (*i.e.*, not stable in the original bipartite graph), we find a restricting edge which was not yet considered during the computation, add it to the tree, delete another edge and compute a solution in the modified tree. As we will show, the algorithm will end with a solution that is feasible (and optimal) for our original problem.

Hochbaum [1] was able to exhibit the LG-algorithm as a flow algorithm. In spite of the similarity of our algorithmic approach, however, we have not been able to interpret our algorithm as a network flow method.

*Institute for Computer Science, University of Freiburg, Freiburg, Germany
(frahling@informatik.uni-freiburg.de)

†ZAIK, Center for Applied Computer Science, University of Cologne, Cologne, Germany

In the unweighted case we can guarantee a computation time of $O(n^4)$ and in the case of integral weights bounded by K a time of $O(K \cdot n^4)$. These theoretical worst case bounds are not as good as the ones known for flow algorithms. On the other hand, our numerical tests indicate that our algorithm performs very well in practice. We have compared implementations of various state-of-the-art algorithms for the stable set problem with our algorithm and report in §4 typical results. Especially in the case of dense graphs our algorithm appears to be considerably faster than network algorithms.

We review in §2 the basic definitions of bipartite graphs as well as the network flow model for the problem so that the difference between our model and the standard approach becomes more clear. Our algorithm is described in §3.

2. Maximum weight stable sets in bipartite graphs.

2.1. Definitions. A *bipartite graph* is a graph $G = (V, E)$ whose node set V can be partitioned into blocks \mathcal{A} and \mathcal{B} such that all edges have one endpoint in \mathcal{A} and the other in \mathcal{B} .

A *stable set* in $G = (V, E)$ is a subset $M \subseteq V$ such that no edge $e \in E$ has both endpoints in M . (For example, both blocks \mathcal{A} and \mathcal{B} of the node partition of a bipartite graph are stable sets.)

Given a weight function $c : V \rightarrow \mathbb{R}_+$, we seek to maximize the total weight

$$c(M) := \sum_{v \in M} c(v)$$

over the collection of stable sets M of G (see Fig. 2.1).

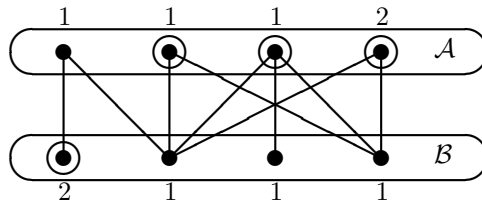


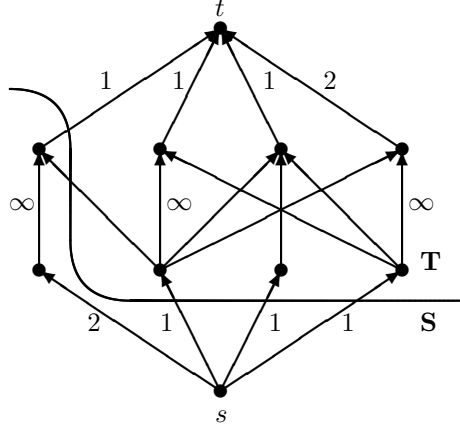
FIG. 2.1. *Maximum weight stable set in a bipartite graph G*

2.2. Solving the problem with network flows. In order to obtain a network flow formulation of the stable set problem, one constructs an oriented auxiliary graph $G_F = (V_F, E_F)$ in which the edges are oriented from \mathcal{B} to \mathcal{A} with additional nodes and edges in the following way:

$$\begin{aligned} V_F &= V \cup \{s, t\} \\ E_F &= E \cup \{(s, b) | b \in B\} \cup \{(a, t) | a \in A\}, \end{aligned}$$

where all original edges between \mathcal{A} and \mathcal{B} are considered to be directed from \mathcal{B} to \mathcal{A} . Next we define capacities $c(e)$ on all edges $e \in E_F$ as follows (see Fig. 2.2):

$$c(v, w) = \begin{cases} c(v) & \text{if } w = t \\ c(w) & \text{if } v = s \\ \infty & \text{otherwise.} \end{cases}$$


 FIG. 2.2. Cut of minimal capacity in G_F

An (s, t) -cut in G_F is a partition of the node set V_F into two sets S and T with $s \in S$ and $t \in T$. Its *capacity* is defined as

$$c(S, T) = \sum_{v \in S, w \in T} c(v, w).$$

Let (S, T) be an (s, t) -cut (S, T) in G_F with (finite) capacity $c(S, T) < \infty$. Then we obtain a stable set $M = M(S, T)$ in G of weight $c(M) = c(V) - c(S, T)$ via

$$M(S, T) := (S \cap \mathcal{B}) \cup (T \cap \mathcal{A}).$$

$M(S, T)$ is clearly stable: Any edge $e = (b, a)$ with $a \in \mathcal{A}$, $b \in \mathcal{B}$ and $a, b \in M(S, T)$ would have infinity capacity in G_F so that $c(S, T)$ were not finite.

Conversely, a stable set M in G yields an (s, t) -cut (S, T) of G_F with (finite) capacity $c(S, T) = c(V) - c(M)$ via

$$\begin{aligned} S &= S(M) = \{s\} \cup (\mathcal{B} \cap M) \cup (\mathcal{A} \setminus M) \\ T &= T(M) = \{t\} \cup (\mathcal{B} \setminus M) \cup (\mathcal{A} \cap M) \end{aligned}$$

Indeed, since M is a stable set in G , this cut cannot contain edges with infinite capacity and hence must be of finite capacity in G_F as stated.

This observation reduces the weighted stable set problem in G to the problem of computing a minimal cut in the auxiliary graph G_F . The min-cut problem can be solved in $O(n^3)$ time by standard flow algorithms such as Karzanov's [3] original preflow-push algorithm. In the case of unit weights the algorithm of Even and Tarjan [4] finds a solution in time $O(n^{5/2})$.

3. The algorithm. We now present our combinatorial algorithm for the maximum stable set problem in the weighted bipartite graph $G = (V, E)$. The subsequent analysis of the algorithm needs a *unique* optimal solution (with respect to some criterion). In order to guarantee uniqueness we introduce a partial order $<_{\mathcal{A}}$ on the collection of subsets of V as follows:

$$S <_{\mathcal{A}} \tilde{S} \Leftrightarrow \{c(S) < c(\tilde{S}) \text{ or } (c(S) = c(\tilde{S})) \wedge (|S \cap \mathcal{A}| < |\tilde{S} \cap \mathcal{A}|)\}$$

LEMMA 3.1. G admits only one maximum stable set with respect to $<_{\mathcal{A}}$.

Proof. Suppose there are two stable sets M and \tilde{M} that are both optimal in terms of $<_{\mathcal{A}}$. We assume w.l.o.g. $M \cap \tilde{M} = \emptyset$ and consider the sets $Z_1 = M \cap \mathcal{A}$, $Z_2 = \tilde{M} \cap \mathcal{A}$, $Z_3 = M \cap \mathcal{B}$ and $Z_4 = \tilde{M} \cap \mathcal{B}$ (see Fig. 3.1).

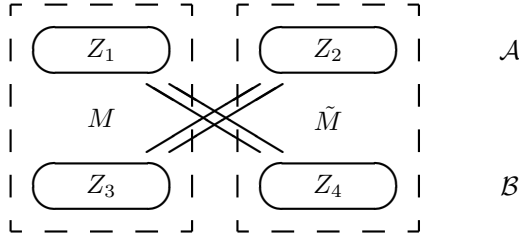


FIG. 3.1. Uniqueness of optimal solution

Edges can only exist between Z_1 and Z_4 and between Z_2 and Z_3 . We therefore infer that $Z_3 \cup Z_4$ is a stable set with weight $c(Z_3) + c(Z_4)$. Because $M = Z_1 \cup Z_3$ is optimal with weight $c(Z_1) + c(Z_3)$, we conclude

$$c(Z_1) \geq c(Z_4).$$

Consider now the stable set $Z_1 \cup Z_2$. In view of $c(Z_1) \geq c(Z_4)$, we must have

$$c(Z_1 \cup Z_2) \geq c(Z_2 \cup Z_4) = c(\tilde{M}).$$

Therefore \tilde{M} cannot be optimal with respect to $<_{\mathcal{A}}$ (as $Z_1 \cup Z_2$ contains a larger number of nodes of \mathcal{A}). \square

We will assume the bipartite graph G to be connected (as otherwise the problem decomposes naturally into subproblems on the respective connected components). Similarly we assume the given node weights $c(v)$ to be positive (as nodes with non-positive weights could be removed from G without affecting the problem).

The algorithm proceeds conceptually as follows:

1. Compute a spanning tree B in G .
2. Find the $<_{\mathcal{A}}$ -optimal solution M in B .
3. Find an edge $e = (a, b) \in E$ with $a \in M$ and $b \in M$. If there is no such edge, STOP: M is feasible in G and hence optimal.
4. Add the edge $e = (a, b)$ to B and create (exactly one) circuit C in $B \cup e$.
5. Delete another edge \tilde{e} from the circuit C and obtain again a spanning tree B .
6. Go to step 2.

3.1. Computing the tree solution. A solution in the spanning tree B can be found efficiently with a dynamic programming approach as we will now show. We fix an arbitrary node $r \in V$ as the *root* of the spanning tree B .

First, we want to compute, for each node $v \in V$ and the subtree B_v attached to v , the following values:

- $\omega_{v,m}$ (= the total weight $c(M_v)$ of a maximum weighted stable set M_v in B_v)
- $\omega_{v,j}$ (= the total weight $c(M_{v,j})$ of a maximum weighted stable set $M_{v,j}$ in B_v under the restriction $v \in M_{v,j}$)

- $\omega_{v,n}$ (= the total weight $c(M_{v,n})$ of a maximum weighted stable set $M_{v,n}$ in B_v under the restriction $v \notin M_{v,n}$)

A maximum weight stable set M_v in B_v either contains v or does not contain v . Therefore

$$(3.1) \quad \omega_{v,m} = \max \{ \omega_{v,n}, \omega_{v,j} \} .$$

A maximum weight stable set $M_{v,n}$ in B_v under the restriction $v \notin M_{v,n}$ decomposes into maximum weight stable sets in the subtrees of the sons of v . Let S_v be the set of the sons of v in the tree B . Then we have

$$(3.2) \quad \omega_{v,n} = \sum_{w \in S_v} \omega_{w,m} .$$

A maximum weight stable set $M_{v,j}$ in B_v under the restriction $v \in M_{v,j}$ decomposes into v itself and maximum weight stable sets in the subtrees of the sons of v under the restriction that the sons themselves are not part of the set. Hence

$$(3.3) \quad \omega_{v,j} = c_v + \sum_{w \in S_v} \omega_{w,n} .$$

Using equations (3.1), (3.2) and (3.3) we can recursively compute the values $\omega_{v,m}$, $\omega_{v,n}$ and $\omega_{v,j}$ for all nodes from the leaves to the root.

We now construct a $<_{\mathcal{A}}$ -maximal solution M in B top down (*i.e.*, from the root to the leaves) and begin by setting $M := \emptyset$. Should the root r be added to M ?

If $\omega_{r,n} > \omega_{r,j}$, there is no optimal solution containing the root r . So we do not add the root to M .

If $\omega_{r,n} < \omega_{r,j}$, each maximal stable set must include the root r . So we add r to M .

If $\omega_{r,n} = \omega_{r,j}$, there are some maximal weight solutions which contain the root r and some which do not. In order to decide whether the root r should be added to M , we establish the following

LEMMA 3.2. *Assume $\omega_{r,n} = \omega_{r,j}$. Then the $<_{\mathcal{A}}$ -maximal stable set M has the property*

$$r \in M \quad \iff \quad r \in \mathcal{A} .$$

Proof. We argue by induction on the number of nodes of the tree B . Let now $M_{r,j}$ be a $<_{\mathcal{A}}$ -maximal stable set with respect to the condition $r \in M_{r,j}$ and $M_{r,n}$ a $<_{\mathcal{A}}$ -maximal stable set relative to the requirement $r \notin M_{r,n}$ and consider an arbitrary node w .

If w belongs to both $M_{r,j}$ and $M_{r,n}$, the restrictions of the solutions $M_{r,j}$ and $M_{r,n}$ to the subtree B_w rooted in w must be both $<_{\mathcal{A}}$ -optimal (otherwise $M_{r,j}$ and $M_{r,n}$ could not be $<_{\mathcal{A}}$ -maximal under the stated conditions). Therefore, in view of the uniqueness property of $<_{\mathcal{A}}$, the solutions must coincide on B_w . Removing B_w from the tree, the claim of the lemma thus follows by induction.

If w belongs to neither $M_{r,j}$ nor $M_{r,n}$, the restrictions of $M_{r,j}$ and $M_{r,n}$ to the subtrees rooted in the sons of w are $<_{\mathcal{A}}$ -optimal and hence identical. The removal of those subtrees therefore establishes the claim by induction as before.

It remains to consider the case where the sets $M_{r,j}$ and $M_{r,n}$ partition the set of nodes.

If $r \in \mathcal{A}$, all children of r are in \mathcal{B} and do not belong to $M_{r,j}$. Recalling the partition property of $M_{r,j}$ and $M_{r,n}$, we conclude that all children belong to $M_{r,n}$. Applying the same argument repeatedly we see that all grandchildren, belonging to \mathcal{A} , must be part of $M_{r,j}$ *etc.*

Because of $\mathcal{A} \cup \mathcal{B} = M_{r,j} \cup M_{r,n}$, we conclude

$$\mathcal{A} = M_{r,j} \text{ and } \mathcal{B} = M_{r,n}$$

and see that $M_{r,j}$ has a larger number of \mathcal{A} -nodes than $M_{r,n}$ and therefore must be the $<_{\mathcal{A}}$ -maximal stable set in the tree. Similarly we find in the case $r \in \mathcal{B}$,

$$\mathcal{A} = M_{r,n} \text{ and } \mathcal{B} = M_{r,j}$$

and obtain $M_{r,n}$ as the $<_{\mathcal{A}}$ -maximal solution. \square

According to Lemma 3.2, we must add r to M if $r \in \mathcal{A}$.

In the situations where we have not added the root r to M , we may construct $<_{\mathcal{A}}$ -maximal stable sets in the subtrees under the sons of r independently from each other in order to obtain an optimal solution. So we apply the same construction to each son of r .

In the situation where we have added the root r to M , no son can belong to the stable set M to be constructed. So we continue the construction of the optimal stable set M with the grandchildren of r .

Applying this construction principle from the root to the leaves we obtain the (unique) $<_{\mathcal{A}}$ -maximal stable set M in B . During the course of the computation, the algorithm looks at each node v three times for the computation of $\omega_{v,\cdot}$, three times for the computation of $\omega_{w,\cdot}$ for the father w of v , and once to decide whether v should be added to M . Consequently, we find that an optimal tree solution can be found in time $O(n)$.

3.2. Avoiding cycling. During our algorithm we must make sure that no spanning tree is considered twice. This property guarantees that our algorithm does not cycle and terminates after a finite number of steps (since the number of spanning trees is finite). We will show that a selection rule for the leaving edge exists so that in each subsequent iteration of the algorithm the value of the tree solution is worse than the preceding tree solution (thus implying that no tree is repeated in the course of the iterations).

Let $e = (a, b)$ denote the entering edge in step 3 of the algorithm. So $a, b \in M$ holds and the edge e creates a circuit C in B (see Fig. 3.2).

CLAIM 1. *There exist adjacent nodes v and w in C none of which belongs to the computed stable set M .*

To verify the claim, note that no two adjacent nodes other than a and b can be in M since otherwise M would not be stable. Furthermore, it cannot happen that

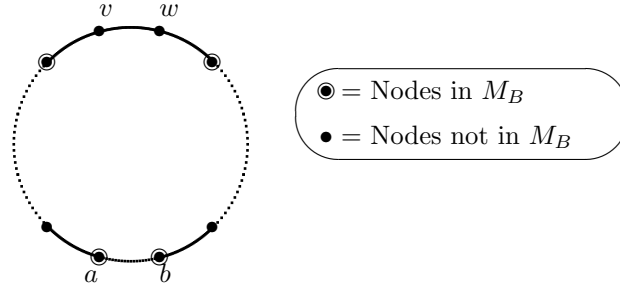


FIG. 3.2. Entering and leaving edge

every other node in C belongs to M because each circuit of a bipartite graph has an even number of nodes and the adjacent nodes a and b are both part of M . Hence we conclude that there must be adjacent nodes v and w in C as claimed.

We choose the edge $\tilde{e} = (v, w)$ as the leaving arc in step 5 of the algorithm and show that the subsequent tree solution \tilde{M} we compute will be worse (in the $<_{\mathcal{A}}$ -order) than M .

The removal of \tilde{e} from the old tree B yields two connected components L and R with, say, $a, v \in L$ and $b, w \in R$ (see Fig. 3.3).

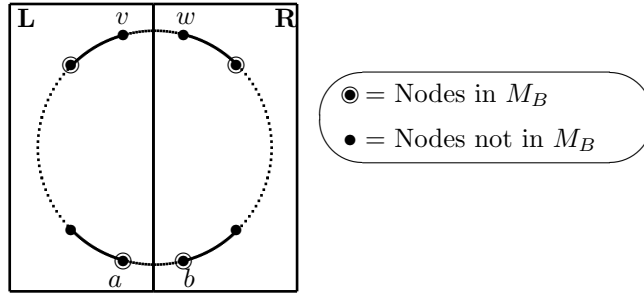


FIG. 3.3. Entering and leaving edge

Observe that the restriction of M to L is stable in L and $<_{\mathcal{A}}$ -optimal. Indeed, if there were another better stable set N in L , $(R \cap M) \cup N$ would yield a better solution than M (in B). Similarly we see that the restriction of M to R is optimal in R .

After adding the edge $e = (a, b)$ to B and deleting the edge $\tilde{e} = (v, w)$, the new solution \tilde{M} must be different from M . The solution changes in R or in L . Say it changes in R . Then the restriction of the new solution to R must be worse than the former solution M since the former optimal solution was unique. On L , the solution stays the same or becomes worse. In either case, the new solution will be worse than the former one. Thus we have proved:

THEOREM 3.3. *The algorithm terminates after a finite number of steps with the $<_{\mathcal{A}}$ -maximal stable set M .*

3.3. Complexity analysis of the algorithm. The running time analysis of our algorithm is based on the previous section. We implement the tree structure

using pointers from children to parents and linear lists of pointers from parents to their children.

- The computation of the spanning tree in step 1 can be implemented in $O(n^2)$ time (using Prim's algorithm, for example).
- As shown in §3.1, we can find a $<_{\mathcal{A}}$ -maximal solution in B in $O(n)$ time.
- Step 3 (the search for the entering edge e) can be implemented in $O(n^2)$ time by testing all edges. (This is a very weak bound since the algorithm rarely considers more than 5 edges to find an entering edge.)
- Step 4 (adding the edge) is carried out by changing the tree structure in step 5.
- Step 5 (determining the leaving edge and deleting it) takes $O(n)$ time. The update of the tree structure needs $O(|V|)$ operations if you do it in a way known from network simplex implementations. We implement the tree structure using son-to-father pointers and linear lists of father-to-son pointers (see [6] for details of this technique).
- Altogether each iteration of the loop in step 6 can be implemented to require $O(n^2)$ time.
- After each iteration we have a solution that is worse than the former one. If the weight of the solution does not decrease, we obtain a solution with a smaller number of nodes of the node set \mathcal{A} . In view of $|\mathcal{A}| \leq n$ we conclude that there are at most n subsequent iterations in which the weight does not decrease.
Hence: After $O(n^3)$ operations the weight of the tree solution must decrease.
- If the graph is unweighted (*i.e.*, all weights are 1), the weight of a candidate solution can decrease at most n times. Therefore, the algorithm then terminates after $O(n^4)$ operations.
- If the weights are integral and bounded by a constant K , the weight of a solution can decrease at most $K \cdot n$ times. The algorithm must therefore terminate after $O(K \cdot n^4)$ operations.

3.4. Further improvements. We now discuss further improvements of the algorithm by which the running time may be improved in practice.

3.4.1. Choice of the entering edge. The choice of the entering edge is crucial for the minimization of the number of iterations. We want to choose an edge that appreciably decreases the total weight of the optimal solution.

Given a possible entering edge $e = (a, b)$, take a look at the numbers $\omega_{a,n}$, $\omega_{a,j}$, $\omega_{b,n}$ and $\omega_{b,j}$. The old optimal solution M contains a and b . If the new solution \tilde{M} does not contain a , we can expect the total weight to drop by $\omega_{a,j} - \omega_{a,n}$ (This is not exactly true in theory but is a good estimate in practice).

We therefore choose an edge $e = (a, b)$ that maximizes

$$k_e = \min\{\omega_{a,j} - \omega_{a,n}, \omega_{b,j} - \omega_{b,n}\}$$

It is not efficient to scan all edges for the one with maximum k_e . In practice the following rule seems to work well:

- Find the p nodes v in partition \mathcal{A} with $v \in M$ and largest $\omega_{v,j} - \omega_{v,n}$.
- For these nodes v search for adjacent edges $e = (v, w)$ with $w \in M$ and maximum k_e . Consider at most q edges during the search.
- If no entering edge is found, search for any possible edge and take the first one found.

Finding the p nodes can be achieved efficiently with a max-heap. Good values for p and q in practice are $p = 10$ and $q = 50$.

3.4.2. Updating the $\omega_{v,\cdot}$. We do not need to compute all $\omega_{v,\cdot}$ in all iterations. Note that the values only change in the circuit C and on the path from C to the root r . All other nodes do not need to be computed.

3.4.3. Updating the stable set. For each node we store a boolean variable if this node is part of the stable set M . In addition, we store the cardinality and the total weight of M .

We do not always compute M from scratch, but update the current M from the root to the leaves. If we consider a node which does not belong to the circuit C or the path P from C to the root and if we do not change M at this node, we know that the whole computation of M in the subtree of this node must be identical with the computation in the former iteration. Hence we may stop updating M and save the computation time on the whole subtree.

4. Numerical comparison with other algorithms. In this section we report about a C++ implementation of our algorithm and compare its performance with implementations of flow algorithms.

4.1. Other algorithms. As pointed out in §2, the problem to determine maximum weight stable sets in bipartite graphs can be formulated as a min-cut problem and can be solved with known flow algorithms.

We compared our algorithm with the following flow-based implementations:

- *Ford-Fulkerson based algorithms.* We implemented a version of the Ford-Fulkerson flow algorithm adjusted and highly optimized for this special flow graph. We programmed two algorithms, which differ in the method of finding augmenting paths. The first uses depth-first-search (DFS), the second breadth-first-search (BFS).
- *Preflow-Push algorithms.* We used the Preflow Push implementation from the Graph Template Library (developed by M. Forster, A. Pick and M. Raitner from the University of Passau). Furthermore, we compared our algorithm with Goldberg's very efficient network optimization library [5].
- *Network-Simplex implementation of CPLEX 7.* A CPLEX file was generated for each test instance and solved using the primal and dual network simplex algorithm. The dual performed consistently much better than the primal.

Therefore, we list here just the running time of the dual solution.

All implementations were tested on the same machine, a Sun E450, with four 400MHz Ultra Sparc II CPUs, 3 GB RAM, not parallelized with the same priority at the same time of day. Our algorithm, the Ford-Fulkerson, Goldberg- and Preflow-Push algorithm have been compiled with GCC 2.95.2.

The time measured is the pure operation time for all algorithms, the time to create internal data structures (i.e. formulate the problem in a way the particular implementation understands) was not measured.

4.2. The test instances. All values shown in the following graphs and tables are averaged over 10 test instances. We used the same ten instances for all algorithms except for Goldberg's push-relabel algorithm. The test data for Goldberg instances, however, were generated according to the same setting of random parameters. For

clarity, we present the results in separate tables below. All node weights are chosen uniformly distributed over the integer set $\{1, 2, \dots, 100\}$.

4.2.1. Instances with a predefined expected edge density q . We construct graph instances by iterating over the edges and adding each edge independently with probability q to the graph.

4.2.2. Instances with predefined average node degree.

First, we computed the quota of all edges which results in the predefined node degree. Then we used the above generator to decide for each edge whether to add it to the graph or not.

4.3. Running time test results.

4.3.1. Dense graphs, 50% of all edges.

V	E	Average running time in seconds				
		A ¹	PP ²	FFB ³	FFD ⁴	NS ⁵
100	1247.7	0.0	0.7	0.0	0.0	0.0
200	4994.4	0.0	6.8	0.1	0.2	0.1
300	11227.8	0.1	26.8	0.8	0.1	0.4
700	61225.2	0.4		13.1	5.2	4.2
1000	125031	0.6			15.7	11.5
2000	500005	3.8				
2800	979940	9.2				

V	E	Average running time in seconds	
		Our Algorithm	Goldberg's Algorithm
1000	125077.0	0.3	0.5
2000	499713.6	2.0	3.4
3000	1125080.7	5.8	8.5
4000	1999611.9	12.6	15.3
5000	3125179.3	20.4	23.7
6000	4499709.7	34.4	34.8
7000	6124470.0	49.4	46.7
8000	8000806.9	70.6	62.6
9000	10124898.3	96.1	79.0
10000	12498361.4	140.8	100.7

¹Our algorithm

²Preflow-Push of Graph Template Library

³Ford-Fulkerson, Breadth First Search

⁴Ford-Fulkerson, Depth First Search

⁵Network-Simplex of CPLEX 7

4.3.2. Dense graphs, 10% of all edges.

V	E	Average running time in seconds				
		A ¹	PP ²	FFB ³	FFD ⁴	NS ⁵
100	255.1	0.0	0.3	0.0	0.0	0.0
200	997.6	0.0	1.6	0.0	0.0	0.0
300	2255.3	0.0	6.0	0.3	0.0	0.1
400	3983.1	0.1	14.6	0.9	0.1	0.1
500	6222.5	0.2		1.6	0.0	0.2
700	12230	0.1		4.1	0.4	0.4
1000	25003	0.4		13.2	2.4	1.3
1600	64020	0.9			13.1	4.6
2000	99917	1.6				10.3
4000	399912	10.0				

V	E	Average running time in seconds	
		Our Algorithm	Goldberg's Algorithm
1000	24973.1	0.0	0.1
2000	99948.0	0.9	0.5
3000	225167.3	2.0	1.6
4000	400186.3	5.1	3.0
5000	624784.4	10.5	4.8
6000	899605.3	17.9	6.9
7000	1224963.8	34.7	9.8
8000	1599952.6	46.8	12.0
9000	2025711.5	79.5	15.6
10000	2500101.5	91.7	20.3

4.3.3. Sparse graphs, average node degree 10.

V	E	Average running time in seconds				
		A ¹	PP ²	FFB ³	FFD ⁴	NS ⁵
40	197.9	0.0	0.1	0.0	0.0	0.0
120	596.3	0.0	0.6	0.0	0.0	0.0
200	988.9	0.0	1.7	0.2	0.0	0.0
500	2490.5	0.0	17.2	0.9	0.1	0.1
1000	4987.1	0.0		3.3	0.5	0.3
1800	9028.5	0.1		12.2	2.4	0.9
2000	10029	0.3			2.8	1.0
2800	14011	0.6			6.1	1.5
4000	20035	1.4			14.4	2.8
6000	30163	3.9				6.5
8000	40050	7.7				12.7

V	E	Average running time in seconds	
		Our Algorithm	Goldberg's Algorithm
2000	10016.3	0.1	0.1
4000	19985	1.2	0.2
6000	30159.3	3.9	0.2
8000	40013.6	6.8	0.4
10000	50258.	12.2	0.5
12000	60518.7	19.1	0.6
14000	70195.5	32.6	0.8
16000	80089	48.7	1.1
18000	91674.4	62.5	1.3
20000	100733.	92.0	1.6

4.3.4. Sparse graphs, average node degree 50.

V	E	Average running time in seconds				
		A ¹	PP ²	FFB ³	FFD ⁴	NS ⁵
200	4969.6	0.0	6.4	0.1	0.1	0.1
300	7485.0	0.1	17.0	0.7	0.1	0.2
500	12456.0	0.1		2.5	0.5	0.4
1000	24953.2	0.8		13.3	2.4	1.3
1400	34961.9	0.9			5.8	2.0
2000	50003.4	1.9			13.4	3.6
2800	69992.7	3.5				6.0
4000	100113	6.6				10.2
4400	110096	8.7				11.5

V	E	Average running time in seconds	
		Our Algorithm	Goldberg's Algorithm
2000	49941.2	0.9	0.2
4000	100037	3.4	0.6
6000	150143	8.5	1.1
8000	200150	16.5	1.6
10000	250466	28.7	2.2
12000	301461	39.3	2.7
14000	351298	61.7	3.3
16000	400483	87.8	3.8
18000	452232	110.3	4.4
20000	499909	160.7	5.1

5. Summary. We have presented a new, conceptually very simple and purely combinatorial algorithm for the stable set problem in bipartite graphs. While we cannot guarantee improved theoretical performance bounds with respect to traditional algorithmic approaches, numerical tests show that this straightforward procedure performs quite well in practice. It is particularly promising for dense graphs.

REFERENCES

- [1] D. S. HOCHBAUM, *A New-Old Algorithm for Minimum-cut and Maximum-flow in Closure Graphs*, NETWORKS 37(4), 2001, pp. 171–193.

- [2] H. LERCHS AND I. F. GROSSMANN, *Optimum design of open-pit mines*, Trans C.I.M., 68 (1965), pp. 17–24.
- [3] A. V. KARZANOV, *Determining the maximal flow in a network by the method of preflows*, Soviet Mathematics Doklady, 15 (1974), pp. 434–437.
- [4] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, SIAM Journal on Computing, 4 (1975), pp. 507–518.
- [5] A. V. GOLDBERG, *An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm*, J. Algorithms, 22 (1997), pp. 1–29.
- [6] R. K. AHUJA, T. L. MAGNATI AND J. B. ORLIN, *Network flows*, Prentice Hall, 1993, pp. 409–411.