An Algorithm for Counting Maximum Weighted Independent Sets and its Applications

Vilhelm Dahllöf* Peter Jonsson†

Abstract

We present an $O(1.3247^n)$ algorithm for counting the number of independent sets with maximum weight in graphs. We show how this algorithm can be used for solving a number of different counting problems: counting exact covers, exact hitting sets, weighted set packing and satisfying assignments in 1-in-k SAT.

1 Introduction

There has recently been a growing interest in the analysis of algorithms for NP-hard problems such as satisfiability [6] and graph colouring [3]. This interest in analysing super-polynomial algorithms may have a number of causes. It is recognised that many important applications can be modelled with these problems and that computers of today can solve moderately-sized problems efficiently. Improvements in exponential time bounds are critical in determining the size of problems that can be solved: even a slight improvement from $O(c^n)$ to $O((c-\varepsilon)^n)$ may significantly increase the size of computationally tractable problems.

In this paper, we will study algorithms for *counting* the number of solutions and not just for deciding the existence of a solution. Such algorithms have been considered earlier; for instance algorithms counting the number of models for SAT [7, 18, 4]; the complexity of this problem has been address by Littman et al. [14]. Greenhill [9] adresses the complexity of counting colourings and independent sets in sparse graphs. Counting independent sets in graphs with maximum degree four, or higher, is known to be $\#P$-complete [17].

The problem of finding one maximum independent set is much more studied than the corresponding counting problem. The first exact algorithm with an upper time bound better than $O(2^n)$ was given by Tarjan and Trojanowski [16] who in 1977 presented an $O(1.2599^n)$ algorithm. This was further improved in 1986 by Jian [12] and independently by Robson [15]. Jian's algorithm runs in $O(1.2345^n)$ time and polynomial space; Robson presented two algorithms: one in polynomial space running in $O(1.2243^n)$ time, and one consuming exponential space running in $O(1.2108^n)$ time. The so far best algorithm was presented by Beigel [2] in 1999; it runs in $O(1.2226^n)$ time and polynomial space.

The basis for our work is an $O(1.3247^n)$ algorithm for counting the number of independent sets with maximum weight in a graph. Our algorithm is the first (known to us) exact algorithm proposed to the end of *counting* maximum independent sets. By transformational techniques, and modifications of the original algorithm, we achieve non-trivial upper bounds for a number of interesting counting problems.

Section 2 introduces some basic terminology and notation for graphs needed for Section 3, where the main algorithm and its correctness are discussed. In Section 4 an upper bound for this algorithm and its help functions is established. Some of its various applications are shown in Section 5. The paper is concluded by a section containing a brief discussion of future work.

2 Preliminaries

A *graph* $G = (V, E)$ is an ordered pair consisting of a finite set V of *vertices* and a set of unordered pairs (u, v) of distinct vertices, called *edges*. Each vertex $u \in V$ has an associated weight $w(u) \in \mathbb{Z}^+$. Two vertices u, v are *adjacent* if $(u, v) \in E$. A set S of vertices is *independent* if $(u, v) \notin E$ for all $u, v \in S$. Let $Ind(G)$ denote the set of all independent sets in G. A *path* from v_1 to v_k is a sequence of vertices $v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$. A *cycle* consists of an edge (u, v) and a path of length > 1 from u to v.

A set of vertices S is *connected* if, for all $u, v \in S$, there is a path from u to v containing only vertices in S. The vertices of a graph can be partitioned into maximal connected subsets, called the *connected components* of G.

If $G = (V, E)$ is a graph and S is a set of vertices,

*Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, `vilda@ida.liu.se`. Partially supported by the *Swedish Foundation for Strategic Research* (SSF) as part of the project "Computational Complexity of Temporal and Spatial Constraint Reasoning".

†Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, `petej@ida.liu.se`. Partially supported by the *Swedish Research Council for Engineering Sciences* (TFR) under grant 2000-361.

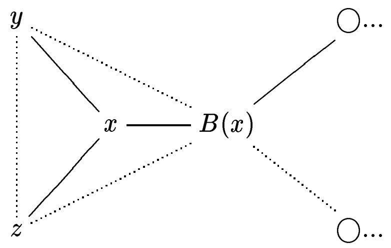

Figure 1: The case of $L_G(x) = 1$ in a graph of maximum degree 3; the filled lines indicate where there is an edge by necessity, whereas a pointed line indicates a possible edge

the graph $G|S = (S, E(S))$, where

$$E(S) = \{(u, v) \in E \mid u, v \in S\},$$

is called the *subgraph* of G *induced by* S. For $S \subseteq V$, we write $G - S$ to denote the graph $G|(V - S)$.

The *neighbourhood* of a vertex v is the set of all vertices adjacent to v (excluding v itself), and is denoted $N_G(v)$. The *degree* of a vertex v (denoted $\Delta(v)$) is the number of elements in E with v as a member. The degree of a graph ($\Delta(G)$) is $\max(\{\Delta(v) \mid v \in V\})$. Note that every connected graph with degree ≤ 2 is either an isolated vertex, a path or a cycle.

We will study a function $\#mis$ from the set of finite graphs to $\mathbb{N} \times \mathbb{N}$. It is defined as follows: $\#mis(G) = (c, s)$ where s is the weight of the heaviest independent set in G and c is the total number of independent sets in G of weight s.

3 Algorithm

We will now describe and prove the correctness of our algorithm for computing $\#mis$. The complexity analysis can be found in the next section.

The main algorithm (which is denoted A and can be found in Figure 1) relies on four auxiliary algorithms: A_2, A_3, A_E and *comb*. The algorithms A_2 and A_3 (Figure 2) compute $\#mis$ for graphs of maximal degrees 2 and 3, respectively. A_E is an algorithm that computes $\#mis$ by exhaustive search. We will only apply A_E to graphs of constant size (< 4 vertices) so we can safely assume that A_E runs in $O(1)$ time. The *comb* algorithm (Figure 1) is a simple, $O(1)$ time computable but very important help function that is used by A_2, A_3 and A.

There is a function L used by algorithm A_3. Informally, one can say that $L_G(x), x \in V$, tells how many of the neighbours of x that are in contact with vertices outside the neighbourhood (including x). Given a graph $G = (V, E)$ and a vertex $x \in V$, we define $L_G(x) = |U|$ where

$$U = \{u \mid (u, v) \in E \wedge u \in N_G(x) \wedge v \notin N_G(x) \cup \{x\}\}.$$

If $L_G(x) = 1$, then $B(x)$ denotes the unique vertex in $N_G(x)$ that is adjacent to one or more vertices outside $N_G(x) \cup \{x\}$. The function is illustrated in Figure 1.

The theorem of the correctness of A relies on four lemmata. As these lemmata are proved, further light is shed upon the structure of A.

LEMMA 3.1. *Let G be a graph consisting of k connected components $G_1, \ldots, G_k$ and assume that $(c_i, s_i) = \#mis(G_i)$. Then, $\#mis(G) = (\prod_{i=1}^{k} c_i, \sum_{i=1}^{k} s_i)$.*

Proof. Trivial.

Arbitrarily choose a graph $G = (V, E)$ and a vertex $x \in V$. We define $I_x^G \subseteq 2^V$ in the following way:

$$I = \{S \subseteq V \mid S \in Ind(G) \text{ and } x \in S\}.$$

Let k be the size of the largest set in I and define $I_x^G = \{I' \in I \mid |I'| = k\}$. We define $I_{\neg x}^G$ similarly but consider independent sets without x, i.e.

$$I = \{S \subseteq V \mid S \in Ind(G) \text{ and } x \notin S\},$$

let k equal the size of the largest set in I and $I_{\neg x}^G = \{I' \in I \mid |I'| = k\}$. Let $s(I_x^G), s(I_{\neg x}^G)$ denote the sizes of the sets in I_x^G, $I_{\neg x}^G$, respectively.

LEMMA 3.2. *Let G be a graph and x a vertex in G. Under the assumption that $(c_1, s_1) = \#mis(G - N_G(x) - \{x\})$ and $(c_2, s_2) = \#mis(G - \{x\})$, it holds that $\#mis(G) = comb(G, x, (c_1, s_1), (c_2, s_2))$.*

Proof. We note the following:

1. if $s(I_x^G) < s(I_{\neg x}^G)$, then $\#mis(G) = (|I_{\neg x}^G|, s(I_{\neg x}^G))$;
2. if $s(I_x^G) = s(I_{\neg x}^G)$, then $\#mis(G) = (|I_x^G| + |I_{\neg x}^G|, s(I_x^G))$ since I_x^G and $I_{\neg x}^G$ are disjoint sets;
3. if $s(I_x^G) > s(I_{\neg x}^G)$, then $\#mis(G) = (|I_x|, s(I_x))$;

By comparing these three cases with the definition of *comb*, we see that the lemma holds iff

1. $(|I_x^G|, s(I_x^G)) = (c_1, s_1 + w(x))$; and
2. $(|I_{\neg x}^G|, s(I_{\neg x}^G)) = (c_2, s_2)$.

It is obvious that (2) holds so we concentrate on (1). Let S be an arbitrary independent set in G containing x and note that $N_G(x) \cap S = \varnothing$. Let $G' = G - N_G(x)$ and note that $I_x^{G - N_G(x)} = I_x^G$. The graph G' contains at least two connected components, $G - N_G(x) - \{x\}$ and the graph only containing the vertex x. Thus, every independent set of maximum weight in G' contains the vertex x, i.e. $\#mis(G') = (|I_x^G|, s(I_x^G))$. By Lemma 3.1, $\#mis(G') = (c_1 \cdot 1, s_1 + w(x))$ which concludes the proof.

LEMMA 3.3. *Let G be a graph such that $\Delta(G) \leq 2$. Then, $A_2(G) = \#mis(G)$.*

Proof. By looking at the algorithm, we can see that Case 1 is correct by Lemma 3.1 and the inductive assumption that the algorithm is correct for smaller inputs. Case 2 is correct since we assume that A_E is correct. The correctness of Case 3 follows from Lemma 3.2.

LEMMA 3.4. *Let G be a graph such that $\Delta(G) \leq 3$. Then, $A_3(G) = \#mis(G)$.*

Proof. The proof is very similar to the proof of Lemma 3.3 – the only obstacle is to realize what is computed in lines 7–25. The if-statements in lines 10–20 will take care of the various possibilities of edges within the set of vertices $\{x, y, z, B(x)\}$, thus computing (c_1'', s_1''), the number and weight of the independent sets among those vertices. By combining Lemmata 3.2 and 3.1, one can see that (c_1, s_1) (computed in line 21) equals $\#mis(G - N_G(B(x)) - \{B(x)\})$ (under the inductive hypothesis that A_3 is correct for graphs that is smaller than G) while (c_2, s_2) (line 24) equals $\#mis(G - \{B(x)\})$. Thus, Lemma 3.2 implies that the value that is returned from line 25 is $\#mis(G)$.

Note that we do not need to consider the case $L_G(x) = 0$ in Algorithm A_3, since G is connected (by Case 3) and $|G| > 3$ (by Case 1).

THEOREM 3.1. *$A(G) = \#mis(G)$ for every graph G.*

Proof. The correctness of A either follows from Lemma 3.4 – for input graphs with maximum degree 3, or from Lemma Lemma 3.2 – for denser graphs.

4 Complexity Analysis

In this section upper bounds for the time complexity of A and its help functions will be proved. Going from functions taking simpler input, to the ones accepting more complicated, we start with a short proof for A_2, a slightly more advanced one for A_3 and ends with a short one for A. We will encounter several recurrences of the form

$$T(n) \leq \sum_{i=1}^{k} T(n - r_i) + \text{poly}(n).$$

algorithm $A(G)$
if $\Delta(G) \leq 3$ **then** return $A_3(G)$
else
  begin
    arbitrarily choose $x \in G$ with degree > 3
    $(c_1, s_1) = A(G - N_G(x) - \{x\})$
    $(c_2, s_2) = A(G - \{x\})$
    return $comb(G, x, (c_1, s_1), (c_2, s_2))$
  end

algorithm $comb(G, x, (c_1, s_1), (c_2, s_2))$
$s_1' = s_1 + w(x)$
if $s_1' < s_2$ **then** return (c_2, s_2)
if $s_1' = s_2$ **then** return $(c_1 + c_2, s_2)$
if $s_1' > s_2$ **then** return (c_1, s_1')

Figure 2: Main algorithm for counting independent sets and a helper function for combining two solutions

They satisfy $T(n) \in O(\lambda(r_1, \dots, r_k)^n)$ where $\lambda(r_1, \dots, r_k)$ is the largest, real-valued root of the function $f(x) = 1 - \sum_{i=1}^{k} x^{-r_i}$. Since this bound does not depend on the polynomial factor poly(n), we can safely assume that all polynomial-time calculations takes $O(1)$ time.

The algorithms A_2 and A_3 apply themselves to every connected component of their input. Note that this does not increase the running time since $\sum_{i=1}^{k} T(n_i) \leq T(n)$ when $n = \sum_{i=1}^{k} n_i$.

LEMMA 4.1. *Algorithm A_2 is a polynomial-time algorithm when applied to graphs of degree 2.*

Proof. Assume (without loss of generality) that the input graph G is connected. If G is a path with n vertices, then the time is bounded by the recurrence $T(n) \leq 4T(\lceil n/2 \rceil)$ and straightforward calculations show that $T(n) \in O(4^{\log_2 n}) = O(n^2)$.

If G is a cycle, then G is immediately transformed into a path in both recursive calls of the algorithm. Consequently, the algorithm is polynomial in this case, too.

LEMMA 4.2. *Algorithm A_3 runs in $O(1.3045^n)$ time when applied to graphs of degree 3.*

Proof. In this lemma, we will measure the size of a graph G by the number m of edges it contains. We will ultimately show that A_3 runs in $O(1.1939^m)$ time; by noting that a degree-3-graph with n vertices can contain at most $3n/2$ edges, we get that the algorithm runs in $O(1.1939^{3n/2}) = O(1.3045^n)$ time.

We assume that the input graph G is connected and contains m edges. Cases 1 and 2 are either trivial

1 **algorithm** $A_2(G)$
2 Case 1: G is not connected. Let $G_1, \ldots, G_k$ be the connected components of G, compute $(c_i, s_i) = A_2(G_i)$, $1 \le i \le k$, and return $(\prod_{i=1}^{k} c_i, \sum_{i=1}^{k} s_i)$.
3 Case 2: $|G| \le 3$. Return $A_E(G)$.
4 Case 3: If G is a cycle, then arbitrarily choose a vertex x. If x is a path, then choose x to be a vertex that splits G into two paths so that the length of one exceeds the other with no more than 1 (i.e. they should be so close to equal length as possible).
5 $(c_1, s_1) = A_2(G - N_G(x) - \{x\})$
6 $(c_2, s_2) = A_2(G - \{x\})$
7 return $comb(G, x, (c_1, s_1), (c_2, s_2))$.

1 **algorithm** $A_3(G)$
2 Case 1: $|G| \le 3$. Return $A_E(G)$.
3 Case 2: $\Delta(G) \le 2$. Return $A_2(G)$.
4 Case 3: G is not connected. Let $G_1, \ldots, G_k$ be the connected components of G, compute $(c_i, s_i) = A_3(G_i)$, $1 \le i \le k$, and return $(\prod_{i=1}^{k} c_i, \sum_{i=1}^{k} s_i)$.
5 Case 4: Arbitrarily choose a vertex x in G such that $\Delta(x) = 3$.
6 **if** $L_G(x) = 1$ **then**
7   **begin**
8     $\{y, z\} = N_G(x) - \{B(x)\}$
9     $(c_1', s_1') = A_3(G - N_G(B(x)) - N_G(x))$
10   **if** $(B(x), y) \in E$ **then**
11       $(c_1'', s_1'') = (1, w(z))$
12   **elsif** $(B(x), z) \in E$ **then**
13       $(c_1'', s_1'') = (1, w(y))$
14   **elsif** $(y, z) \in E$ **then**
15   **if** $w(y) \neq w(z)$ **then**
16         $(c_1'', s_1'') = (1, max(w(y), w(z)))$
17   **else**
18         $(c_1'', s_1'') = (2, w(y))$
19   **else**
20       $(c_1'', s_1'') = (1, w(y) + w(z))$
21     $(c_1, s_1) = (c_1' \cdot c_1'', s_1' + s_1'')$
22     $(c_2', s_2') = A_3(G - N_G(x) - \{x\})$
23     $(c_2'', s_2'') = A_E(G|\{x, y, z\})$
24     $(c_2, s_2) = (c_2' \cdot c_2'', s_2' + s_2'')$
25   return $comb(G, B(x), (c_1, s_1), (c_2, s_2))$
26   **end**
27 **else**(* $L_G(x) > 1$ *)
28   **begin**
29     $(c_1, s_1) = A_3(G - N_G(x) - \{x\})$
30     $(c_2, s_2) = A_3(G - \{x\})$
31   return $comb(G, x, (c_1, s_1), (c_2, s_2))$
32   **end**

Figure 3: The auxiliary algorithms A_2 and A_3

or follows from Lemma 4.1 so we only consider the final case where we have two subcases:

1. $L_G(x) = 1$. Both graphs $G - N_G(B(x)) - N_G(x)$ (line 9) and $G - N_G(x) - \{x\}$ (line 22) contain at most $m - 4$ edges; the graph $G|\{x, B(x), y, z\}$ contains at most 5 edges since $\Delta(x) = 3$ and there is at least one edge from $B(x)$ to some vertex not in $\{x, y, z\}$ since $L_G(x) = 1$. The time needed in this case is bounded by $T(m) \le 2T(m-4)$ and $T(m) \in O(1.1892^m)$.

2. $L_G(x) > 1$. The graph $G - N_G(x) - \{x\}$ contains at most $m - (3 + L_G(x)) = m - 5$ edges since $\Delta(x) = 3$ and $L_G(x)$ is at least 2. Similarly, the graph $G - \{x\}$ contains at most $m - 3$ edges. The time needed in this case is bounded by $T(m) \le T(m-3) + T(m-5)$ and $T(m) \in O(1.1939^m)$.

THEOREM 4.1. *Algorithm A runs in $O(1.3247^n)$ time.*

Proof. The result follows from Lemma 4.2 if $\Delta(G) \le 3$. Otherwise, assume the given graph contains n vertices. We see that the graphs constructed in lines 6 and 7 contain at most $n - 5$ and $n - 1$ vertices, respectively, since $\Delta(x) \ge 4$. This implies that the running time is bounded by the recurrence $T(n) \le T(n-1) + T(n-5)$, which solves to $O(1.3247^n)$.

5 Applications

Here we present some of the various applications of the algorithm A. First some definitions of the counting problems are given and then, for each of them, an upper bound is presented. In the corresponding proof an algorithm that transfers the problem instance to a weighted graph is shown. These problems and variants of them have been intensively studied, *cf.* [5, 8, 10].

#1-IN-k SAT
INSTANCE: A CNF F.
QUESTION: What is the number of satisfying assignments; a satisfying assignment having the property that exactly one literal be true in each clause?

#EXACT HITTING SET
INSTANCE: A finite set U and a collection C of subsets $c_1 \ldots c_n \in U$.
QUESTION: A solution is a minimum size subset $H \subseteq U$ hitting each c_i exactly once, i.e. $|c_i \cap H| = 1$. What is the number of solutions?

#EXACT COVER
INSTANCE: A finite set U and a collection C of subsets $c_1 \ldots c_n$ of U.

QUESTION: If C contains an exact cover for U – i.e. a subcollection $C' \subseteq C$ such that every element of U occurs in exactly one member of C' – what is the number of exact covers?

#WEIGHTED SET PACKING
INSTANCE: A finite set U and a collection C of subsets $c_1 \ldots c_n \in U$ and for each c_i there is an associated weight $w(c_i)$.
QUESTION: A solution is a collection $C' \subseteq C$ of disjoint sets of maximum weight. What is the number of such solutions?

PROPOSITION 5.1. *For arbitrary k, #1-IN-k SAT can be solved in $O(1.7548^n)$ time, where n is the number of variables.*

Proof. In order to use A for counting solutions we need to transfer the CNF F into an appropriate, weighted graph $W' = (V, E)$. In W' a solution to F corresponds to a maximum independent set with weight N, where N is the number of clauses.

For the sake of clarity, we will present an algorithm that builds a graph W in which a solution to F corresponds to a maximum clique of weight N. As is clear to the reader, the graph W' is simply the complement of W, and applying A to W' is in effect the same as counting maximum cliques in W. The graph W takes its vertices from the literals, and gives rise to a vertex set that has at most twice the cardinality of the set of variables – hence the $O(1.7548^n)$ bound.

The set V of vertices: Any literal α that appears in any clause will form a vertex A. The weight of A is the number of appearances of α. Let us call α the *regent* of A and the clauses where α appears the *regent clauses* of A. If among A's regent clauses there are appearances of both β and $-\beta$ (for some variable β), remove A from V.

The set E of edges: For two vertices A and B, add an edge iff:

1. They did not come from the same variable, just differing in the sign;
2. They do not overlap in F, i.e. A's regent does never appear in the same clause as B's regent;
3. In none of A's regent clauses there is an appearance of a variable β (or $-\beta$), which appears in any of B's regent clauses in the negated form $-\beta$ (β).

We now show that counting the number of cliques with weight N in the graph W is equal to finding the number of solutions to the 1-in-k SAT of the original F.

Every solution corresponds to an N-weighted clique: A solution means an assignment such that each of the N clauses each contains exactly one true literal. The clauses can be partitionated so that those having the same true literal belong to the same set. Assign each such set a number $1 \ldots j$. Let each of the true literals form a vertex. The sum of those vertices' weights is N. Pick an arbitrary vertex A. Since only one literal per clause is true, the clauses of A cannot contain both β and $-\beta$. So A is a vertex in the graph W. It remains to show that each pair of the vertices have an edge. Pick two vertices A and B. Since

1. both A and B come from true literals, they can not come from the same variable with opposite signs;
2. only one literal per clause is true, they can not overlap;
3. only one literal per clause is true, the variable u cannot be true in A's clauses but false in B's.

Thus, there is an edge between any pair of the vertices, providing that $j > 1$. If $j = 1$, there is one vertex with weight N which can also be regarded as a clique.

Every N-weighted clique corresponds to a solution found: This part of the proof is two-folded – first it is proved that it is possible to consistently assign truth values to the regents of the vertices, and then it is proved that such an assignment is a solution.

It is possible to consistently assign truth values: When a clique of weight N is found the regents of the clique vertices are assigned true. This can be done consistently since condition 1 regarding the edges guarantees that the same variable will not appear in both negative and positive form in the clique. The regents of the non-clique vertices are assigned false. This can be done consistently since:

1. The construction of V guarantees that the sets of clique and non-clique regents are disjunct – hence an assignment of a non-clique regent will not affect a clique regent;
2. To prove that no non-clique regent will be assigned true by some other non-clique being assigned false, one can note that:
    - Inspecting two arbitrarily chosen, connected vertices A and B, in no pair of their regent clauses is to be found the same variable but having opposite signs – condition 3 in the edge construction;
    - and furthermore, the clique is covering all clauses, by condition 2 in the edge construction;

Such an assignment is a solution: The regents of the clique vertices satisfy N clauses. This follows from the construction of vertices and conditions 1 and 2 in the edge construction.

It remains to show that at most one literal be true in each clause. Assume for the sake of contradiction that in some clause there is more than one true literal. We know that at least one of them is part of the clique, since the clique is covering all clauses. Two of the literals cannot be part the clique due to conditions 1 and 2 in the construction of edges. Neither can one literal be part of the clique and the others not – any such true non-clique regent is impossible, as was shown in the above discussion about consistent assignment.

Remark 1: Among the applications, this one we implemented to test A. After constructing W, one can note that picking a vertex A and its neighbourhood, if the sum of those vertices' weight is lower than N, A can not be part of a solution and can thus be safely removed from V. This procedure can be repeated until a fixed point is reached. For most CNFs tested this fixed point is 0. One CNF with fixed point 127 (i.e. 127 vertices) took about 100 secs to process and this CNF was testing the limits of the computer (an Ultra-SPARC at 440 MHz, 256 Mb RAM) – the bottleneck is the memory consumption. It should be noted that A was naively implemented. To obtain the code, please contact Vilhelm Dahllöf.

Remark 2: One can note that the problem instances are solved – the number of solutions is found – in $O(1.7548^n)$ regardless of the maximum clause length. This is interesting since Impagliazzo and Paturi [11] make the conjecture that *deciding the existence* of an ordinary satisfying assignment approaches $O(2^n)$ as k (the maximum clause length) grows larger.

PROPOSITION 5.2. #EXACT HITTING SET *can be solved in $O(1.3247^n)$ time, where n is $|U|$.*

Proof. This problem is very similar to 1-in-k SAT; the two differences being that there is no negation of the variables and there is a requirement that the solution be of minimum size. So among the maximum weighted solutions only the ones with the smallest number of vertices is kept. Using the same procedure as in the 1-in-k SAT to build W, we can again apply A to the complement W' provided we perform some minor changes to A. A has to return a *triple* (c, s, n) where c is the number of maximum weighted, minimum sized solutions, s is the weight of the found solutions (of course, it should equal $|U|$), and n is the number of vertices in the found solutions. So, for each occurence

algorithm *comb*$'(G, x, (c_1, s_1, n_1), (c_2, s_2, n_2))$
$s'_1 = s_1 + w(x)$
$n'_1 = n_1 + 1$
if $s'_1 < s_2$ **then** return (c_2, s_2, n_2)
if $s'_1 = s_2$ **then**
  if $n'_1 < n'_2$ **then** return (c_1, s'_1, n'_1)
  elsif $n'_1 > n'_2$ **then** return (c_2, s_2, n_2)
  else return $(c_1 + c_2, s_2, n_2)$
if $s'_1 > s_2$ **then** return (c_1, s'_1, n'_1)

algorithm $A'(G)$
if $\Delta(G) \leq 3$ **then** return $A'_3(G)$
else
  begin
    arbitrarily choose $x \in G$ with degree > 3
    $(c_1, s_1, n_1) = A'(G - N_G(x) - \{x\})$
    $(c_2, s_2, n_2) = A'(G - \{x\})$
    return *comb*$'(G, x, (c_1, s_1, n_1), (c_2, s_2, n_2))$
  end

Figure 4: Parts of an algorithm for computing #EXACT HITTING SET.

of a result tuple in the algorithm, there will be an ocurrence of such a triple. The main function and the *comb*$'$ function are presented in Figure 5; the other functions can easily be modified similarly.

PROPOSITION 5.3. #EXACT COVER *can be solved in $O(1.3247^n)$ time, where n is the number of sets.*

Proof. Let (U, C) be an arbitrary instance of the #EXACT COVER problem. Construct a weighted graph $W = (V, E)$ as follows: let each $c_i \in C$ give rise to a vertex $v_i \in V$ whose weight is $|c_i|$. Add an edge between v_i and v_j if and only if $c_i \cap c_j \neq \varnothing$. Clearly, no independent set in W can have a weight greater than $|U|$. Furthermore, the independent sets of weight $|U|$ in W corresponds to solutions of (U, C).

PROPOSITION 5.4. #WEIGHTED SET PACKING *can be solved in $O(1.3247^n)$ time, where n is the number of sets.*

Proof. Let (U, C) be an arbitrary instance of the #WEIGHTED SET PACKING problem. Construct a weighted graph $W = (V, E)$ as follows: introduce one vertex v_i for each $c_i \in C$ and assign it weight $w(c_i)$. Add an edge between v_i and v_j if and only if $c_i \cap c_j \neq \varnothing$. Obviously, a maximum weighted, independent set found in W constitutes a solution.

6 Conclusions and Future Work

We have presented an $O(1.3247^n)$ algorithm for counting the number of independent sets with maximum weight in graphs and shown how this algorithm can be used for solving a wide variety of counting problems. Our algorithm has been implemented and has thus descended from the ideal world where most of the algorithms for #P complete problems have dwelt up til now, and can be used and experimentally studied.

There are probably many ways to further improve the running time of the basic algorithm – for instance by introducing a new help function A_4 for handling graphs of maximal degree 4. Assuming that A_4 runs in $O(c^n)$ where

$$1.2852 \approx \lambda(1,6) < c < \lambda(1,5) \approx 1.3247,$$

the running time of the algorithm can be decreased to $O(c^n)$.

Another possibility for improving the algorithms is to utilise dynamic programming. As an example, the currently fastest algorithm for solving the maximum independent set problem, Robson's [15] algorithm, runs in $O(1.2108^n)$ time. This time bound is achieved by adding a highly memory-consuming dynamic programming scheme to an $O(1.2277^n)$ time, polynomial-space algorithm. Algorithm A uses only a polynomial amount of space and could thus benefit from such a modification.

Several authors (*cf.* [17] and [10]) have studied restricted versions of counting problems and shown that they are $\#P$-complete. For instance, Vadhan has shown that the problem of counting independent sets is $\#P$-complete even if restricted to planar, bipartite graphs of bounded degree. It seems probable that this problem can be solved by more efficient algorithms than the $O(1.3247^n)$ algorithm in this paper, however. One immediate idea is to use some decomposition technique for planar graphs, e.g. decomposition into k-outerplanar graphs [1] or using Lipton and Tarjan's planar separator theorem [13]. Such methods have been very successful in the construction of approximation algorithms but they have apparently not been used for the construction of counting algorithms.

References

[1] B. S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the ACM*, 41(1):153–180, 1994.

[2] R. Beigel. Finding Maximum Independent Sets in Sparse and General Graphs. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 856–857. ACM/SIAM, January 1999.

[3] R. Beigel and D. Eppstein. 3-coloring in time $O(1.3446^n)$: a no-MIS algorithm. In *Proc. 36th Symp. Foundations of Computer Science*, pages 444–453. IEEE, October 1995.

[4] E. Birnbaum and E. L. Lozinskii. The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research (JAIR)*, 10:457–477, 1999.

[5] B. Chandra and M. Halldórsson. Greedy local improvement and weighted set packing approximation. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-1999)*, pages 169–176, 1999.

[6] E. Dantsin, E. A. Hirsch, S. Ivanov, and M. Vsemirnov. Algorithms for sat and upper bounds on their complexity. *Electronic Colloquium on Computational Complexity (ECCC)*, 8(12), 2001.

[7] O. Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science (TCS)*, 81(1):49–64, 1991.

[8] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[9] C. S. Greenhill. The complexity of counting colourings and independent sets in sparse graphs and hypergraphs. *Computational Complexity*, 9(1):52–72, 2000.

[10] H. B. Hunt III, M. V. Marathe, V. Radhakrishnan, and R. E. Stearns. The complexity of planar counting problems. *SIAM Journal on Computing*, 27(4):1142–1167, 1998.

[11] R. Impagliazzo and R. Paturi. On the complexity of k-sat. *Journal of Computer and System Sciences (JCSS)*, 62(2):367–375, 2001.

[12] T. Jian. An $O(2^{0.304n})$ algorithm for solving maximum independent set problem. *IEEE Transactions on Computers,*, 35(9):847–851, 1986.

[13] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[14] M. L. Littman, T. Pitassi, and R. Impagliazzo. On the complexity of counting satisfying assignments. Appeared in the working notes of the LICS 2001 workshop on Satisfiability.

[15] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440, 1986.

[16] R. E. Tarjan and A. E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):266–283, 1977.

[17] S. P. Vadhan. The complexity of counting in sparse, regular, and planar graphs. *SIAM Journal on Computing*, 31(2):398–427, 2001.

[18] W. Zhang. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science (TCS)*, 155(1):277–288, 1996.