A NOVEL TYPE OF BRANCH AND BOUND FOR MAXIMUM INDEPENDENT SET

Marcel Wild

ABSTRACT: Several algorithms are presented. The standard algorithm generates all $N$ anticliques of a graph $G$ with $v$ vertices in time $O(Nv^2)$. It can e.g. be adapted to calculate the independence polynomial of $G$, to generate all maximum cardinality anticliques, or just one maximum anticlique. The latter was programmed using the Mathematica 6.0 code. For a random $(45,92)$-graph $G$ a maximum anticlique of size 21 was found in 1.344 sec, whereas the “hardwired” Mathematica command `MaximumIndependentSet[G]` clocked in at 155838 sec, which is five orders of magnitude slower.

1. Introduction

Throughout $G$ will be a finite simple graph with vertex set $V = \{1, 2, \cdots, v\}$. What we call an anticlique $X$ in this article is also known as independent or stable set of $G$, i.e. no two vertices in $X$ are adjacent. The maximum cardinality of an anticlique in $G$ is denoted by $\alpha(G)$, and the number of all anticliques of $G$ is its Fibonacci number $f(G)$. The name derives from the fact that for $n$-element paths $P_n$ one clearly has

$$f(P_n) = f(P_{n-1}) + f(P_{n-2})$$

More involved recurrence relations for $f(G_n)$ can be proven for other classes $\{G_n : n \in \mathbb{N}\}$ of graphs [PT].

We first present our standard algorithm which computes $f(G)$ for arbitrary graphs. The standard algorithm can be adapted to produce one maximum anticlique, or all maximum anticliques, or all inclusion-maximal anticliques. Although a clique of $G$ is the same as an anticlique in the complementary graph $G^c$, and thus all our results about anticliques carry over to cliques, the specific nature of our algorithms directly applies only to anticliques.

Here comes the section break up. In section 2 the standard algorithm is described in detail. Rather than generating the $N = f(G)$ anticliques one by one, it employs the principle of exclusion to generate “multivalued rows” $r$, each one of which may encode a vast number $N(r)$ of anticliques. The theoretic complexity is $O(Nv^2)$. Since $N(r)$ is readily computed and the number of rows $r$ is usually small compared to $N$, computing $N$ is considerably faster than generating all anticliques by expanding the rows. Refining for each row $r$ the arithmetic necessary to get $N(r)$, one obtains the numbers $s_k$ of $k$-element anticliques ($1 \leq k \leq v$) and hence the independence polynomial $I(G, x)$. In particular, $I(G, 1) = f(G)$. This is done in section 2.1.

Section 3 is devoted to modifications of the standard algorithm. For starters, in 3.1 we fix $k$ and adapt the standard algorithm so as to only produce all anticliques of cardinality $> k$ (possibly none if $k$ is too high.). If desired, one can stop as soon as the first anticlique of size $> k$ pops up. In particular, provided $\alpha(G)$ is known beforehand (as it is in many situations), choosing $k := \alpha(G) - 1$ yields all (or one) maximum cardinality anticliques.

What if $\alpha(G)$ is not known? In 3.2 a branch and bound technique yields $\alpha(G)$ along with an $\alpha(G)$-element anticlique. We compare it with the standard branch and bound method for
integer programming. In 3.2.1 we specialize to bipartite graphs because for them a maximum anticlique can be found by a well known and possibly competitive $O(w\sqrt{v})$ algorithm ($w =$ number of edges). Subsection 3.3 is devoted to numerics. For random graphs $G$ of various edge density our algorithm (the one tailored to one maximum anticlique) is pitted against Mathematica’s hardwired command \texttt{MaximumIndependentSet}[G], respectively against \texttt{BipartiteMatchingAndCover}[G] for bipartite graphs. We close in 3.4 with a few remarks on generating all inclusion-maximal anticliques.

2. Generating all anticliques

Let us introduce the standard algorithm, more precisely \textit{standard} $(a,B)$-\textit{algorithm} on this $(5,6)$-graph:

For each $y \in V$ let $B(y)$ be the set of neighbours of $y$, so e.g. $B(2) = \{1,4\}$. A subset $X$ of the vertex set $V = \{1,2,\cdots,v\}$ (here $v = 5$) is an anticlique if and only if it satisfies these $v$ conditions

\begin{equation}
\label{anticlique}
(1) \quad a \in X \Rightarrow B(a) \cap X = \phi \quad (a \in V)
\end{equation}

Beginning with the powerset $C = 2^V$ we shall continuously shrink $C$ by \textit{excluding} all $X$ which are not anticliques, until at the end exactly the family $C = 2^V$ of all anticliques remains. As with any application of the \textit{principle of exclusion} (which is discussed more formally in [W1]), its efficiency hinges upon the compact representability of the usually huge and fast changing set families $C$. In the “worst case” a subset $X \subseteq V$ must be represented by its characteristic $0,1$-vector of length $v$, (so $X = \{2,4\}$ is $(0,1,0,1,0)$) but often that can be avoided. For instance, we write $(2,2,2,2,2)$ for $2^V$ with the understanding that each label 2 stands for “either 1 or 0”. In view of (1), we impose upon $(2,2,2,2,2)$ the “anti-implication” $1 \rightarrow \overline{2},\overline{4},\overline{5}$, that is, we exclude the bad $X$’s and retain those $X \in (2,2,2,2,2)$ that satisfy $(1 \in X \Rightarrow 2,4,5 \notin X)$. Using the symbolism $a \rightarrow \overline{b}$ akin of (1) we define the \textit{many-valued row}

\[
\begin{align*}
\{a,\overline{b},2,\overline{b},\overline{b}\} & \quad \text{as the family of all these $X$’s. (Throughout the algorithm at most the five symbols $0,1,2,a,\overline{b}$, possibly with subscripts, will occur in any one row.) Call $r$ an \textit{unsplit son} of $(2,2,2,2,2)$. So far, so good. But how to sieve all $X \in r$ that satisfy the anti-implication $2 \rightarrow \overline{1},\overline{7}$? We first split $r$ into the disjoint union of two \textit{candidate sons}:
\end{align*}
\]

\[
\begin{align*}
r_1 & := \{X \in r | 2 \in X\} = (0,1,2,2,2) \\
r_2 & := \{X \in r | 2 \notin X\} = (a,0,2,\overline{b},\overline{b})
\end{align*}
\]

Notice that if we force $2 \in X$ then, because of $1 \rightarrow \overline{2},\overline{4},\overline{5}$, we also force $1 \notin X$. Since $1 \rightarrow \overline{2},\overline{4},\overline{5}$ holds for \textit{any} set $X$ with $1 \notin X$, the third, fourth and fifth component of $X$’s characteristic
vector are now free to be 1 or 0. This explains why \( r_1 = (0, 1, 2, 2, 2) \). On the other hand, if we force \( 2 \notin X \), then for such \( X \)'s the anti-implication \( 1 \rightarrow \overline{1}, \overline{2}, \overline{3} \) is equivalent to \( 1 \rightarrow \overline{4}, \overline{5} \). This explains why \( r_2 = (a, 0, 2, \overline{b}, \overline{b}) \). By splitting \( r \) we managed that the new anti-implication \( 2 \rightarrow \overline{1}, \overline{3} \) trivially holds for all \( X \in r_2 \), and is easily imposed upon \( r_1 \) by switching its fourth component from 2 to 0:

\[
\begin{array}{cccc|c}
  r'_1 &=& 0 & 1 & 2 & 0 & 2 & PA = 3 \\
  r_2 &=& a & 0 & 2 & \overline{b} & \overline{b} & PA = 3 \\
\end{array}
\]

One then proceeds as follows:

\[
\begin{array}{cccc|c}
  a & 0 & 2 & \overline{b} & \overline{b} & PA = 3 \\
  a & 0 & 1 & 0 & \overline{b} & PA = 4, 5 \\
  a & 0 & 0 & \overline{b} & \overline{b} & PA = 4 \\
  a & 0 & 0 & \overline{b} & \overline{b} & PA = 4 \\
  0 & 0 & 0 & 1 & 0 & PA = 5 \\
  a & 0 & 0 & 0 & \overline{b} & PA = 5 \\
  a & 0 & 0 & 0 & \overline{b} \\
  0 & 0 & 0 & 1 & 0 \\
  0 & 0 & 1 & 0 & 1 \\
  0 & 1 & 2 & 0 & 2 \\
  0 & 1 & 2 & 0 & 2 \\
\end{array}
\]

A few comments are in order. We could have continued by imposing the third anti-implication \( 3 \rightarrow \overline{4} \) on \( r \)'s proper sons \( r'_1 \) and \( r_2 \), then \( 4 \rightarrow \overline{1}, \overline{2}, \overline{3}, \overline{5} \) on all arising rows, and then \( 5 \rightarrow \overline{1}, \overline{4} \). But we did it the LIFO kind of way (last in, first out), i.e. we always only processed the top row of our working stack, and labelled the other rows with the index of their pending anti-implications (PA). Thus at first both \( r'_1 \) and \( r_2 \) had \( PA = 3 \). It just so happened that \( r'_1 \) satisfied \( 3 \rightarrow \overline{4} \) already. In fact it satisfied all remaining anti-implications. Hence it was removed and constituted the first row of a growing output stack (shown top right). The sole remaining row of the working stack being subjected to the condition \( 3 \rightarrow \overline{4} \) resulted in a splitting akin to before. The top row had \( PA = 4 \) (which held already) and then \( PA = 5 \). Having imposed \( 5 \rightarrow \overline{1}, \overline{4} \) by splitting, the arising two rows are put on the output stack. Continuing that way one arrives at the complete output stack on the bottom right which represents a total of

\[
f(G) = 3 + 1 + 1 + 2 + 4 = 11
\]
anticliques of $G$. For instance, the unique maximum anticlique $X = \{2, 3, 5\}$ is a member of the bottom row. Notice that all singleton anticliques, as well as the empty anticlique, occur in various rows. We sometimes refer to a collection of many-valued rows as a context.

**Theorem:** Let $G$ be a graph with vertex set $V = \{1, 2, \ldots, v\}$. Then the standard $(a, \overline{B})$-algorithm generates the $N$ anticliques $X \subseteq V$ in time $O(Nv^2)$.

**Proof.** We impose our anti-implications\(^1\) in the order $1 \to \overline{B(1)}, \ldots, v \to \overline{B(v)}$, where $1, 2, \ldots, v$ is any fixed ordering of $V$. Consider a “generic” top row $r$ of the working stack which has $t \to \overline{B}$ ($B = B(t)$) as pending anti-implication. In order to impose it upon $r$ we assume by induction that the symbol $\rho$ at position $t$ in row $r$ is either 0, 2 or $b$ (but neither $a$ nor 1). If we think of the vertices as being processed from “left to right”, the situation is as follows:

$$r = (\ldots, \rho, \ldots, \ldots, \ldots, \ldots)$$

By induction we may assume that to the right of $\rho$ there is no symbol $a$ or 1, but we also must not write these symbols to the right of $\rho$ in the upcoming process of changing row $r$.

**Case 1:** $\rho = 0$ or the whole of $B$ consists of 0’s. Then $t \to \overline{B}$ is satisfied by all $X \in r$, so $r$ survives unaltered.

**Case 2:** $\rho = 2$.

**Case 2.1:** There is a 1 within $B$. Then put $\rho = 0$ since the sets $X \in r$ that satisfy $t \to \overline{B}$ are precisely the ones with $t \not\in X$.

**Case 2.2:** $B$ consists only of 0’s and of 2’s (at least one of the latter by the Case 1). Then put $\rho = a$ and put $\overline{b}$ on all the 2’s within $B$.

**Case 2.3:** There is no 1 and at least one $a$ or $\overline{b}$ on a $B$-position. Consider the row

$$r = (\ldots, 1, \overline{\overline{b}}, a_4, a_3, a_1, a_2, \overline{\overline{b}}, 0, 2, \overline{\overline{b}}, \overline{\overline{b}}, 0, \ldots, \overline{\overline{b}})$$

which is typical in that it illustrates all four possibilities: from $a_1 \to \overline{\overline{b}}$ only $a_1$ is in $B$; from $a_2 \to \overline{\overline{b}} \overline{\overline{b}}$ the premise and a nonempty part (possibly all) of the anticonclusion is in $B$; from $a_3 \to \overline{\overline{b}} \overline{\overline{b}}$ only a proper part of the anticonclusion is in $B$; and from $a_4 \to \overline{\overline{b}}$ only the complete anticonclusion is in $B$. A moment’s thought shows that the sets $X \in r$ that satisfy $t \to \overline{B}$ are precisely the ones in the (disjoint) union of $r'$ and $r''$, where

$$r' = (\ldots, 1, \overline{\overline{b}}, a_4, a_3, a_1, a_2, 0, \overline{\overline{b}}, \overline{\overline{b}}, 0, 2, \overline{\overline{b}}, \overline{\overline{b}}, 0, \ldots, \overline{\overline{b}})$$

$$r'' = (\ldots, 1, \overline{\overline{b}}, 2, a_3, 0, 0, 0, 1, 0, 0, 0, 2, 2, 0, \ldots)$$

**Case 3:** $\rho = \overline{b}$.

\(^1\)The ensuing standard $(a, \overline{B})$-algorithm differs considerably from the almost-namesake $(a, B)$-algorithm in [W1] that handles implications $a \to B$.  

4
Case 3.1: There is a 1 within $B$. Then put $\rho = 0$ and leave the rest. That is, unless $B$ constitutes the \textit{whole} anticonclusion in its anti-implication $a \rightarrow \overline{b}$; in which case additionally change $a$ to 2.

Case 3.2: There is no 1 within $B$. Then split $r$ into $r'$ and $r''$. Here $r'$ has $\rho = 0$ and is defined exactly as in case 3.1. As to $r''$, it has $\rho = 1$ and 0's on all $B$-positions. This entails repercussions outside $B$ which are exactly analogous to Case 2.3. But how to change the anti-implication $a \rightarrow b \cdots b$ whose (say) first $b$ has been turned to $\rho = 1$? If $a$ has been in $B$, it has already been set to 0; otherwise put $a = 0$ now. As to the $b$'s in $a \rightarrow 1 b \cdots b$ which have not been put to 0 (being in $B$), switch them to 2.

The above shows that one is never forced to delete rows, and that imposing an anti-implication on a row of length $v$ costs $O(v)$. There are at most $N$ final rows (the ones on the output stack) because they are mutually disjoint and each encodes at least one anticlique. Each final row arises by the imposition of exactly $w$ anti-implications, and each imposing costs $O(v)$. Since no deletions occur, no other (superfluous) work has been done, and so the claimed $O(Nv^2)$ bound results.

Notice that the number of final rows is usually much smaller than $N$ since one final row can comprise a great many of anticliques. Also, it is easy to see that the working stack can at most contain $w$ rows at any given moment. If a row does not split (cases 1, 2.1, 2.2, 3.1), we say that it undergoes \textit{trivial changes}.

### 2.1. Calculating the independence polynomial

The \textit{independence polynomial} [LM1] of a graph $G$ is defined as

$$I(G, x) := \sum_{k=0}^{\alpha(G)} s_k x^k,$$

where $s_k$ is the number of $k$-element anticliques of $G$. Notice that $f(G) = I(G, 1)$. As to the computation of the coefficients $s_k$, each $s_k$ is the sum of all

$$s_k(r) := \{|X \in r : |X| = k\}|$$

where $r$ ranges over all finalized rows produced by the standard $(a, B)$-algorithm. Put $c := |\text{ones}(r)| + |\text{twos}(r)|$, and let $\beta_1, \ldots, \beta_t$ be the lengths of the anti-conclusions appearing in $r$. In order to get $s_k(r)$, first list all integer component solutions $x = (x_1, \cdots, x_t)$ of

\begin{equation}
(2) \quad x_1 + \cdots + x_t = k - c \quad (1 \leq x_i \leq \beta_i, \quad 1 \leq i \leq t)
\end{equation}

which is an easy matter. The \textit{weight} of a solution $x$ is defined as

$$w(x) := w(x_1)w(x_2)\cdots w(x_t), \quad w(x_i) := \begin{cases} (\beta_i) & \text{if } x_i \neq 1 \\ 1 + \beta_i & \text{if } x_i = 1 \end{cases}$$

It is easy to see that $s_k(r)$ is the sum of all $w(x)$ where $x$ ranges over all solutions of (2). For instance, consider the row $r$ below:
$r =$

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|
|   | 2 | 0 | 1 | 2 | $a_1$ | $b_1$ | $b_1$ | $b_1$ | $a_2$ | $b_2$ | $a_3$ | $b_3$ | $b_3$ |

Here $c = 3, \beta_1 = 3, \beta_2 = 1, \beta_3 = 2$. Putting say $k = 8$ the solutions of (2) are $x = (3, 1, 1)$ and $x' = (2, 1, 2)$, hence

$$s_8(r) = w(x) + w(x') = \left(\frac{3}{3}\right)(1 + 1)(1 + 2) + \left(\frac{3}{2}\right)(1 + 1)\left(\frac{2}{2}\right) = 12$$

It follows from the Theorem that computing the independence polynomial of a $v$-graph $G$ can be done in time $O(f(G)v^2)$.

3. Variations of the standard $(a, \overline{B})$-algorithm

In order to tweak our standard algorithm in various ways we need a bit more notation. Consider a generic 5-valued row $r$ and this $X \in r$:

$$r := (0, \cdots, 0, 1 \cdots, 1, 2, \cdots, 2, a_1, \overline{b_1} \cdots, \overline{b_1}, \cdots, a_s, \overline{b_s}, \cdots, \overline{b_s})$$

(3) $$X := (0, \cdots, 0, 1 \cdots, 1, 1 \cdots, 1, 0, 1 \cdots, 1, \cdots, 0, 1 \cdots, 1)$$

Like symbols are adjacent in (3) only for easier visualization and notation. It would be possible but distracting to give a completely precise definition of a 5-valued row $r = (\rho_1, \rho_2, \cdots, \rho_v)$. Rather, the following concepts will do. Let $\text{zeros}(r)$ be the set of positions of 0’s, so

$$\text{zeros}(r) := \{i \in V | \rho_i = 0\},$$

and define $\text{ones}(r)$ and $\text{twos}(r)$ similarly. Furthermore, let $\text{premset}(r)$ be the set of positions $k$ occupied by the symbols $a_i$ ($1 \leq i \leq s$). Thus $\text{premset}(r)$ collects all “premises” $a_i$. Finally, for each $k \in \text{premset}(r)$ let $\text{anticonc}(r, k)$ be the set of positions occupied by the symbols $\overline{b_i}$ (assuming that at position $k$ is $a_i$). Formally, the $X$ in (3) is $X = V - (\text{zeros}(r) \cup \text{premset}(r))$, and it is clear that $X$ is a maximum cardinality member of $r$. It is unique iff all $\text{anticonc}(r, k)$ ($k \in \text{premset}(r)$) have cardinality $\geq 2$. If the cardinality is 1, one can alternatively remove the unique (position of) $\overline{b_i}$ instead of $a_i$. For any row $r$ put

$$w_{\text{max}}(r) := \max\{|X| : X \in r\} = v - |\text{zeros}(r)| - |\text{premset}(r)|$$

3.1. Finding all (or one) anticliques of lower bounded cardinality

For any natural number $k$ we can generate the possibly empty family of all anticliques of cardinality $> k$ as follows. By induction, anchored at $r = (2, 2, \cdots, 2)$, we can assume that each row $r$ that reaches the top of the working stack when running the standard algorithm, has $w_{\text{max}}(r) > k$. If say $r$ has $PA = 12$, then the 12th anti-implication is imposed upon $r$ and gives rise to either an unsplit son $r'$ or two proper sons $r'_1, r'_2$ as in section 2. Say the latter happens (the case with $r'$ is similar), and say $w_{\text{max}}(r'_1) \leq k$ but $w_{\text{max}}(r'_2) > k$. Then $r'_1$ is thrown away (since all its proper sons will also have $w_{\text{max}} \leq k$) but $r'_2$ is put on top of the stack with its $PA$.
set to 13. Proceeding this way the surviving finalized rows will exactly comprise the anticliques of cardinality \( > k \). If we only need one such anticlique, we stop as soon as a finalized row \( r \) with \( w_{\text{max}}(r) > k \) comes up.

In particular, provided \( \alpha(G) \) is known beforehand, one gets all maximum cardinality anticliques by setting \( k := \alpha(G) - 1 \). For instance, there is a well known formula of Ryser to count all systems of distinct representatives. The \((a, \mathcal{B})\)-algorithm can be used to generate them because they can be viewed as maximum anticliques in some obviously defined graph; furthermore \( \alpha(G) \) is known beforehand. Another benefit of having all maximum anticliques of \( G \) concerns the core of \( G \) which is defined as the intersection of them all [LM2].

### 3.2 Computing one maximum cardinality anticlique

The approach of 3.1 needs to be refined if \( \alpha(G) \) is unknown. What we say below easily extends to the computation of all maximum (cardinality) anticliques and thus will not be spelled out (but see 3.3 for some numerics).

To find one maximum anticlique, we initialize the output stack by \((0, 0, \cdots, 0)\) and put \( \text{currentmax} := 0 \). Throughout the whole procedure the output stack contains only one member. We change the standard algorithm as follows. Besides its PA pointer each row \( r \) in the working stack is accompanied by its continuously updated \( w_{\text{max}}(r) \) value. In particular, the first row \( r = (2, 2, \cdots, 2) \) has \( w_{\text{max}}(r) = v \), and so \( w_{\text{max}}(r) > \text{currentmax} \). Generally, this is what happens when \( r \) reaches the top of the working stack. If \( w_{\text{max}}(r) \leq \text{currentmax} \), then \( r \) is deleted. Otherwise \( r \) is processed and either gives rise to an unsplit son \( r' \), or to two candidate sons \( r_1, r_2 \). Now \( r' \), respectively \( r_1, r_2 \) are processed as usual but additionally each is facing deletion if its \( w_{\text{max}} \)-value is \( \leq \text{currentmax} \). Suppose either \( r' \) or one (or both) proper son \( r'_2 \) is finalized, i.e. all anti-implications have been imposed on it. If its \( w_{\text{max}} \) is \( \leq \text{currentmax} \), then it is thrown away. On the other hand, if say \( w_{\text{max}}(r'_2) > \text{currentmax} \), then we update \( \text{currentmax} := w_{\text{max}}(r'_2) \) and substitute the previous member of the output stack by \( r'_2 \). The algorithm stops when the working stack is empty. At this moment the unique row \( r_{\text{max}} \) of the output stack is such that every (often unique) maximum cardinality member \( X \) of \( r_{\text{max}} \) is a maximum anticlique of \( G \). Thus \( \alpha(G) = |X| \) is found.

What we just described is a branch and bound algorithm, albeit of a novel kind with its 0, 1, 2, \( a, \overline{b} \) symbolism. Let us compare it with, say, the standard branch and bound procedure for integer programming (IP). As to branching, that is cheap for IP. Namely, some fractional component, say \( x_4^* = 6.8 \), of the relaxed linear programming (LP) solution \( (x_1^*, \cdots, x_n^*) \) yields two subproblems by adding the inequalities \( x_4 \leq 6 \) and \( x_4 \geq 7 \) respectively. As seen in the proof of the Theorem, branching (i.e. row splitting) is a bit more expensive in our algorithm but still benign. As to upper bounding, that is costly for (IP) where an LP-maximal solution needs to be computed, whereas evaluating (4) is almost gratuitous. As opposed to IP branch and bound, it does not make sense to pick the pending subproblems (= many-valued rows) merely according to their high \( w_{\text{max}}(r) \) value. Equally important is it that \( r \) has few anti-implications pending, i.e. \( PA(r) \) must be high. The present implementation always picks the top row in the working stack, but it is conceivable that selecting \( r \) according to another policy would be beneficial.

Notice that the \((a, \overline{B})\)-algorithm can be adapted to find a maximum weight anticlique with respect to a weighing \( w : V \rightarrow \mathbb{N} \). Furthermore, the \((a, \overline{B})\)-algorithm could easily be parallized \([XI]\): Each processor sends its updated \( \text{currentmax} \) values to the shared memory which in turn
forwards it to all processors.

### 3.2.1 The bipartite case

In each graph $G$ the complements $V - X$ of the anticliques $X$ are exactly the *vertex covers* $Y$, i.e. each edge of $G$ is incident with at least one vertex of $Y$. It is easily seen that running the $(a, B)$-algorithm (standard version or as in 3.2) it suffices to inflict all anti-implications $y \rightarrow B(y)$ where $y$ ranges over a vertex cover of $G$. In general it may be hard to find a small vertex cover but each bipartite graph has one of cardinality $\leq \frac{1}{2} |V|$, namely the smaller of the two color classes.

For any graph $G$ let $\nu(G)$ be the maximum size of a matching in $G$, and let $\tau(G)$ be the minimum size of a vertex cover of $G$. Then $\nu(G) \leq \tau(G)$ [S, p.260], and $\nu(G) = \tau(G)$ takes place whenever $G$ is bipartite. A way to compute $\alpha(G)$ in a $v$-element bipartite graph $G$ is thus

$$\alpha(G) = v - \tau(G) = v - \nu(G)$$

Here $\nu(G)$ can be computed in time $O(w \sqrt{v})$ where $w$ is the number of edges [S, Theorem 16.4]. Once a maximum matching is found one can construct from it a minimum vertex cover (and whence a maximum anticlique) in time $O(w)$ [S. Thm.16.6].

Our $(a, B)$-algorithm of 3.2 benefits twofold from $G$ being bipartite. Say the color classes have $v_1$ and $v_2$ vertices respectively, where $v_1 \leq v_2$ and $v_1 + v_2 = v$. First, the $(a, B)$-algorithm can be run on a $v_1$-element vertex cover (namely the smaller color class), and second we can start with $\text{currentmax} := v_2$ instead of $\text{currentmax} := 0$ as before. A comparison of these two methods to compute $\alpha(G)$ is part of 3.2.2.

### 3.3 Numerics

We compared the $(a, B)$-algorithm, implemented in the Mathematica 6.0 high level code, with the hardwired command `MaximumIndependentSet`. Mathematica 6.0 allows one to generate random $v$-graphs $G$ of any desired edge density $0 \leq d \leq 1$. For each choice of $v \in \{30, 40, 50, 60, 70, 80, 90\}$ and $d \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ we generated three random $(v, w)$-graphs $G_1, G_2, G_3$ of edge density $d$ and picked the $G_i$ for which the quotient of processing times was the least disastrous for `MaximumIndependentSet` - unless the running time of the latter was more than 20000 sec, in which case the other runs were omitted. Besides $\alpha(G_i)$, the running times and their proportions, we also report the number $\text{rsp}$ of row splittings occurring in the $(a, B)$-algorithm. The number of times that rows only underwent trivial changes (as defined in section 2) was usually 20% to 60% of $\text{rsp}$.

---

2It goes without saying that in thousands of test runs both algorithms agreed upon $\alpha(G)$. All times over 1 sec are rounded to full seconds, and dito the proportion of running times is rounded to an integer. I thank Günter Pilz, Helmut Prodinger, Stefan Wagner, and Yves Semegni for testifying many of these proportions.
What strikes the most, for \( v \) fixed the size of \( \alpha(G) \) effects \texttt{MaximumIndependentSet} vastly more than the \((a, B)-\text{algorithm}\). For instance, for \( v = 40 \) and \( \alpha(G) = 7 \) respectively \( \alpha(G) = 16 \), \texttt{MaximumIndependentSet} took 11 respectively 8058 seconds, whereas the times were 0.265 respectively 1.485 seconds for the \((a, B)-\text{algorithm}\). Put another way, for \( v \) fixed the performance gap gets the bigger the sparser \( G \) is.

Let’s conversely keep the density \( d \) fixed and see how the two algorithms compare when \( v \) is raised. For instance, a random graph \( G \) with \( v = 400 \) and \( d = 0.9 \) had \( \alpha(G) = 5 \) (rsp = 7050) and the times were 19 sec versus 30’843 sec. For \( v = 3000 \) and \( d = 0.95 \) the \((a, B)-\text{algorithm}\) still bravely delivered \( \alpha(G) = 6 \) (rsp = 309818) in 51161 sec; forget about \texttt{MaximumIndependentSet}.

For \( v = 3000 \) graphs with density smaller than 0.95 would outdo the \((a, B)-\text{algorithm}\) because \texttt{rsp} explodes. Interestingly however, \textit{one sporadic} huge antclidean in such a graph (here “artificially”

| \( d \) | \( (v, w) \) | \( \alpha(G) \) | \( \text{rsp} \) | \( (a, B)\)-alg. | \texttt{MaximumIndependentSet} | \texttt{proportion} |
|---|---|---|---|---|---|---|
| 0.9 | (30, 389) | 3 | 35 | 0.031 | 0.313 | 10 |
| | (40, 702) | 3 | 53 | 0.031 | 0.672 | 22 |
| | (50, 1092) | 3 | 87 | 0.062 | 2 | 24 |
| | (60, 1597) | 3 | 121 | 0.094 | 3 | 28 |
| | (70, 2169) | 3 | 166 | 0.156 | 5 | 30 |
| | (80, 2826) | 4 | 205 | 0.172 | 8 | 49 |
| | (90, 3608) | 4 | 241 | 0.235 | 12 | 53 |
| 0.7 | (30, 308) | 4 | 77 | 0.047 | 0.609 | 13 |
| | (40, 557) | 5 | 130 | 0.078 | 2 | 25 |
| | (50, 872) | 5 | 236 | 0.156 | 4 | 25 |
| | (60, 1249) | 5 | 438 | 0.297 | 9 | 32 |
| | (70, 1730) | 5 | 581 | 0.438 | 14 | 32 |
| | (80, 2207) | 6 | 948 | 0.734 | 37 | 51 |
| | (90, 2800) | 6 | 1394 | 1.203 | 53 | 44 |
| 0.5 | (30, 227) | 6 | 144 | 0.078 | 2 | 20 |
| | (40, 385) | 7 | 442 | 0.265 | 11 | 43 |
| | (50, 601) | 8 | 829 | 0.500 | 26 | 52 |
| | (60, 887) | 8 | 1599 | 1 | 75 | 72 |
| | (70, 1226) | 8 | 2644 | 2 | 125 | 66 |
| | (80, 1600) | 8 | 5360 | 4 | 337 | 85 |
| | (90, 2016) | 9 | 7995 | 7 | 751 | 106 |
| 0.3 | (30, 117) | 9 | 451 | 0.266 | 22 | 84 |
| | (40, 244) | 10 | 1516 | 0.922 | 117 | 126 |
| | (50, 381) | 11 | 3174 | 2 | 478 | 219 |
| | (60, 562) | 12 | 7195 | 4 | 1299 | 291 |
| | (70, 714) | 12 | 38023 | 26 | 5584 | 215 |
| | (80, 951) | 13 | 75533 | 56 | 24587 | 441 |
| | (90, 1155) | 14 | 200 753 | 151 | 119 610 | 792 |
| 0.1 | (30, 54) | 15 | 488 | 0.281 | 444 | 1581 |
| | (40, 96) | 16 | 2265 | 1 | 8058 | 5426 |
| | (45, 110) | 19 | 3609 | 3 | 82616 | 31473 |
| | (70, 270) | 24 | 380 260 | 279 | - | - |
| | (90, 405) | 29 | 16 430 117 | 12831 | - | - |
| 0.08 | (45, 92) | 21 | 1698 | 1 | 155 838 | 115 951 |
inserted, of cardinality 100 or 500) is detected very fast: For \( v = 3000 \) and \( d = 0.95 \) it took 1285 sec to find \( \alpha(G) = 100 \), and only 51 sec to find \( \alpha(G) = 500 \).

As to the \( k \)-threshold method of 3.1, for the \((90,1155)\)-graph \( G \) we set \( k := 13 \) and got a 14-element anticlique in less than 1 second! The catch is that we do not know whether this is the maximum size. Putting \( k := 14 \), the threshold-variant took 107 seconds to respond that all anticliques have size \( \leq k \). Thus \( \alpha(G) = 13 \) could be established in \( 107 + 1 = 108 \) seconds which is better than the 151 seconds of the currentmax-method of 3.2. Unfortunately, nobody tells us in advance that \( k = 13, 14 \) are the right values to pick.

With the \( k \)-threshold method finding all maximum anticliques is not much harder than finding one maximum anticlique. Specifically, the currentmax-method needed 0.156 sec to find \( \alpha(G) = 3 \) for the \((70,2169)\)-graph in the table. Applying the \( k \)-threshold method with \( k = 2 \) then delivered the total of 46 maximum anticliques in merely 0.234 sec. Similarly the time to get all 6 maximum anticliques (of size 21) for the \((45,92)\)-graph was \( 1.344 + 2.328 \) sec.

Let us compare the \((a,\overline{B})\)-algorithm with the MATHEMATICA-command \texttt{BipartiteMatchingAndCover} which for a bipartite graph \( G \) computes both a maximum matching and a minimum vertex cover, as outlined in 3.2.1. We fixed one color class to be of cardinality \( v_1 = 50 \), and the other had cardinality \( v_2 = 50, 150, 450, 1950 \) respectively. The edge densities were \( d = 0.1, 0.5, 0.9 \). Here are the times:

| \( v_2 \) | \( d \) | \((a,\overline{B})\)-algorithm | \texttt{BipartiteMatchingAndCover} |
|---|---|---|---|
| 50 | 0.1 | 6438 | 1 |
|    | 0.5 | 2   | 4   |
|    | 0.9 | 0.125 | 6 |
| 150 | 0.1 | 7   | 3   |
|    | 0.5 | 0.032 | 11 |
|    | 0.9 | 0.031 | 20 |
| 450 | 0.1 | 0.156 | 21 |
|    | 0.5 | 0.063 | 36 |
|    | 0.9 | 0.063 | 62 |
| 1950 | 0.1 | 0.125 | 127 |

For our random \( G \)'s we always got \( \alpha(G) = v_2 \) but this is irrelevant here. The key observation is that for fixed \( v_2 \) increasing the edge density \( d \) impedes \texttt{BipartiteMatchingAndCover} since this algorithm depends linearly on the number of edges \( w \). But it benefits the \((a,\overline{B})\)-algorithm because whenever an anti-implication \( i \rightarrow \overline{B}(i) \) causes a row \( r \) to split into \( r_1 \) (with 0 at position \( i \)) and \( r_2 \) (with 1 at position \( i \)), the row \( r_2 \) collects a large number of 0's (namely \( |B(i)| \) many) which in view of (4) likely makes \( w_{\max}(r_2) \) smaller than the big \( (\geq v_2) \) value of \texttt{currentmax}, and so \( r_2 \) is cancelled. Indeed, e.g. starting with \texttt{currentmax} := 0 instead of \texttt{currentmax} = \( v_2 \) increases the time from 0.063 to over 9 seconds for the bipartite graph with \( v_2 = 450 \) and \( d = 0.5 \). The above also entails, and the table confirms it, that the sheer increase of \( v_2 \) (edge density being fixed) already benefits the \((a,\overline{B})\)-algorithm. Of course, as testified by the 6438 versus 1 second, the flip side of the medal is that the \((a,\overline{B})\)-algorithm is severely beaten on sparse bipartite graphs with roughly equally sized color classes.
3.4. Finding all inclusion-maximal anticliques

How does one sieve out the $N_{\text{max}}$ inclusion-maximal (dually: inclusion-minimal) sets among $N$ subsets of $\{1, 2, \cdots, n\}$? Comparing each of the $N$ sets $X_i$ with the pile of currently maximal sets, and accordingly adjusting that pile, costs $O(N^2 n)$. If $N$ is much larger than $n$ (which is the case in 3.4 and many other instances of the principle of exclusion) there is a faster way - though the theoretic bound still is $O(N^2 n)$. Namely, rather than manipulating $N$ short sets $X_i$ (of cardinality $\approx n$) one manipulates $n$ long index sets $\text{contain}_a$ (of cardinality $\approx N$) defined by

$$\text{Contain}_a := \{1 \leq i \leq N : a \in X_i\}$$

(5)

It is fairly obvious how this is to be done; see [W2, p.113, 114] for more details.

Coming back to anticliques, consider again the row $r$ in 2.1. The inclusion-maximal sets contained in $r$ obviously are these $2^3$ sets:

$$X_i = \{1, 3, 4\} \cup U_1 \cup U_2 \cup U_3,$$

where

$$U_1 = \{5\} \text{ or } \{6, 7, 8\}$$

$$U_2 = \{9\} \text{ or } \{10\}$$

$$U_3 = \{11\} \text{ or } \{12, 13\}$$

Using the method that employs the lists in (5) one can build up the pile of maximal anticliques by weaving in or deleting the row-wise maximal sets. One can conceive examples where a set $X$ which is a maximal member of a row $r$ is contained in $X'$ which is maximal within another row $r'$. But this never occured in our tested random graphs. If we could simply pile up all row-wise maximal sets without worrying about (5), we would have an algorithm that produces the $N_{\text{max}}$ maximal anticliques in time $O(N_{\text{max}} v^2)$ (by the Theorem). This would be better than the best theoretical bound [TTT].

The inclusion-maximal anticliques $X_i$ of $G$ for instance feature in the chromatic number of $G$, since this is the minimum number $\chi(G)$ of $X_i$'s necessary to cover the vertex set $V$. Because it doesn’t harm to have a few superfluent non-maximal sets, it pays off to simply generate all row-wise maximal sets $X_i$. Computing $\chi(G)$ in this way compares very favorably to the hardwired Mathematical command $\text{ChromaticNumber}[G]$, despite the nasty bottleneck of finding a minimum cover by these $X_i$'s.

References

[LM1] V.E. Levit, E. Mandrescu, The independence polynomial of a graph-a survey, Proceedings of the 1st Internat. Conf. on Algebraic Informatics, 233-254, Thessaloniki 2005.

[LM2] V.E. Levit, E. Mandrescu, Combinatorial properties of the family of maximum stable sets of a graph, Discrete Appl. Math. 117 (2002), 149-161.

[PT] H. Prodinger, R. Tichy, Fibonacci numbers of graphs, Fibonacci Quarterly 20 (1982) 16-21.
[S] A. Schrijver, Combinatorial Optimization, Springer 2003.

[TTT] E. Tomita, A. Tanaka, H. Takahashi, The worst-case time complexity for generating all maximal cliques and computational experiments, Theoretical Computer Science 363 (2006) 28-42.

[W1] M. Wild, Generating all satisfying truth assignments for Horn-Sat, in particular all closed sets of a closure system. Submitted.

[W2] M. Wild, Computations with finite closure systems and implications, p.111-120, LNCS 959 (1995).

[XI] C. Xavier, S.S. Iyengar, Introduction to parallel algorithms, John Wiley 1998.