# An algorithm calculating upper bounds on the sizes of max-cliques using subgraphs

Svante Hellstadius and Ronnie Johansson
June 8, 1999

**Abstract**

In this text we provide an algorithm giving an upper bound on the size of the maximum clique in a graph. The algorithm uses induced subgraphs and rules designed to decrease a set of properties as much as possible. Time and memory requirements are $O(n^5)$ and $O(n^3)$ respectively. Evaluation on random graphs suggests that the upper bounds depend linearly on the graph orders, and the slope depends on the probability that certain edges exist in a random graph. This paper also provides an introduction to variants of the clique problem and discusses a memory efficient data structure representing undirected graphs.

## 1. Introduction

Many different algorithms can be constructed to solve one specific problem. Some might be more efficient than others. An algorithm's efficiency is determined by its time and memory consumption. Often these measures are analyzed by theoretical means, but when an algorithm's behavior is too complex to understand, simulations and statistics might come in handy.

### Computational complexity

When analyzing algorithms theoretically using mathematics we talk about *computational complexity*. An algorithm's time complexity is studied by investigating how the algorithm behaves for various inputs with different problem sizes. It is often useful to examine what kind of input constitutes the worst case input of the algorithm. Memory usage is another crucial property that has to be addressed. Throughout this text computational complexity will be an important matter of discussion.

### Problem classes

Also the problems themselves are analyzed theoretically. Sometimes the issue is whether a problem can be solved at all computationally, as there are many problems known to be undecidable or hard to solve due to the fact that no reasonable algorithm exists. Figure 1.1 shows three sets of problems.

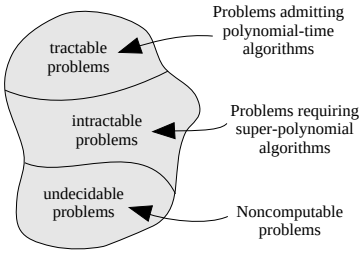



FIGURE 1.1. Algorithmic problems. Undecidable, intractable, and tractable problems can be regarded as three sets.

No algorithm requiring less than $cn^k$ computational steps for problem size $n$ and any constants $c$ and $k$ exists for intractable problems. Polynomial algorithms do exist for tractable problems.

### NP-completeness

Various complexity classes can be identified. In a complexity class there are many problems related to each other. Thousands of natural computational problems belong to the class of *NP-complete* (NPC) problems. The graph coloring problem, the traveling salesman problem (TSP), the Hamiltonian path problem, and the clique problem are some well known NPC problems.

These are important properties of NPC problems:

- A given solution to an NPC problem can be verified in polynomial time.
- Any NPC problem can be transformed into any other NPC problem by using polynomial-time transformations.

It is unknown whether NPC problems are tractable or not. Most computer scientists are convinced, however, that this is not the case. The fact that nobody has yet been able to present a polynomial-time algorithm solving an NPC problem, despite extensive research efforts, indicates that no such algorithm exists. However, the intractability of NPC problems has not been shown. A consequence of the properties above, is that all NPC problems are tractable if one of them is tractable.

No matter what status NPC problems have, instances of them do occur in applications and we have to cope with them. Finding a polynomial-time algorithm is hard, and using a super-polynomial algorithm is no good on large inputs, so trying to find a good approximate solution seems to be the best we can do. Such a solution might not be optimal, but we seek to make it reasonably good.

Cormen, Leiserson, and Rivest provide a good survey of NPC problems in [1].

## The clique problem

In this text, we will attack *the clique problem*. A clique in an undirected graph $G = \langle V, E \rangle$ is a complete subgraph of $G$. In other words, a clique is a subset $V' \subseteq V$ of vertices, each pair of which is adjacent. The size of a clique is $|V'|$. Sometimes we call a clique with size $k$ a $k$-clique. When speaking about the *order* (or *problem size*) of a graph $G = \langle V, E \rangle$ we mean $|V|$, sometimes also denoted by $n$. Also let $\omega(G)$ denote the actual size of a largest clique, often called *max-clique*, in a graph $G$. We use the notation $V[G]$ for the set of vertices of the graph $G$, and $E[G]$ for the set of edges of $G$. For convenience we sometimes use the "sloppy" notation $|G|$ meaning $|V[G]|$.

The clique problem, which is an NPC problem, is the problem of answering whether there is a $k$-clique in a given graph. The corresponding optimization problem, finding the largest clique of a graph, is called an *NP-hard* problem. Figure 1.2 illustrates a maximum clique.

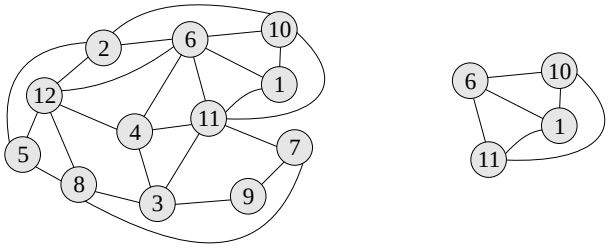



FIGURE 1.2. In this graph with 12 vertices, the maximum clique size is 4. To the right, the max-clique is shown extracted.

Finding an approximate solution to the problem is hard too. Håstad has shown that if NPC problems are intractable, no tractable algorithm can find a clique in a graph $G$ whose size is within a factor of $n^{1-\delta}$ of $\omega(G)$ for any constant $\delta > 0$ [2].

When we create a *random graph* $G$, $p$ is the probability that two vertices $v_i, v_j \in V[G]$ are adjacent, for any $i \neq j$, and we also call $p$ *the coverage* of $G$. In a random graph with coverage $\frac{1}{2}$, the expected max-clique size is $\sim 2 \log_2 n$.

Generally the expected max-clique size in a random graph with coverage $p$ is $\sim 2\log_{1/p} n$ [2].

The problem of finding complete subgraphs in the complement graph is called *the independent set problem*, which is essentially the same as the clique problem..

# 2. Data representation

An important problem that has to be solved efficiently is that of data representation. How should an undirected graph be represented in memory requiring as little resources as possible? With resources we mean time and memory. The discussion in this chapter is general and the results can be applied to any graph algorithm implementation. Note that we will only consider undirected graphs in this text.

## Adjacency lists

If the graphs to be stored are sparse (that is, the average valency of the vertices of the graph is low) adjacency lists are suitable. Each vertex has a link to a dynamically allocated and linked adjacency list. That list contains the names or some other kind of references to the other vertices that the vertex is connected to, as shown in Figure 2.1. This representation is easy to implement and easy to understand which makes it a natural choice in many applications. Our test implementation of our algorithm uses linked lists.

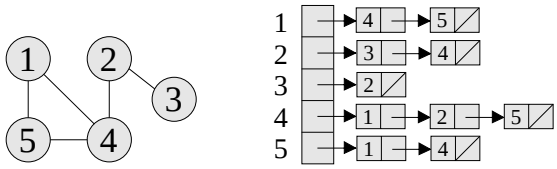



FIGURE 2.1. The advantages of the linked list representation is that it is very easy to understand and implement. The shortcomings are many though, which makes this representation unsuitable to us.

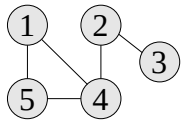
The problems with linked lists are many though. If the graph to be stored is not sparse then the linked list implementation is not efficient requiring lots of elements in the linked lists. Considering the fact that not only the name of a vertex is part of a list element, but also a pointer to another element, it is not hard to see why linked lists are not preferable when the graph is dense. Not only are lists memory consuming ($n^2$ list elements are required if the graph is complete [worst case] with $n$ vertices) but also time intense because you need to

search linearly though the entire adjacency list of a vertex to examine whether a certain connection exists.

## Adjacency matrices

To be able to run our algorithm on "large" problems we need another representation that is faster and less memory consuming.

In a graph with $n$ vertices there can at most be $\binom{n}{2}$ connections (the maximum case occurs when the graph is complete). The idea of adjacency matrices is to have a matrix element for each possible connection. The vertices are listed both vertically and horizontally giving rise to a square matrix where each matrix element corresponds to a combination of two vertices. If those two vertices are connected, an edge marker is put in that matrix element. If not connected, another kind of marker is put there, as shown in Figure 2.2.




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 1 | 1 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 |
| 4 | 1 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 0 | 1 | 0 |

FIGURE 2.2. The adjacency matrix is a neat way of representing a graph which is also very understandable.

### Optimized adjacency matrices

A quick implementation of this would likely implement the matrix as a two dimensional byte array. Some improvements can be done though. First note that we only need one bit to indicate whether an edge between two vertices exists or not. Using bits instead of bytes reduces the memory demand nicely (only $\frac{1}{8}$th is required) but not asymptotically. Also note that it is unnecessary to store the diagonal of the matrix since no vertices are connected to themselves. This observation saves $n$ bits of memory. Finally note that the transpose of an adjacency matrix $A$ should be identical to $A$ (that is, $A^T = A$) since the operation of connecting vertices is commutative (if vertex $v_1$ is connected to vertex $v_2$, then we can be sure that $v_2$ is connected to $v_1$ as well). Thus, half of the remaining matrix is redundant and should be omitted. We end up with an *optimized adjacency matrix*, which we, for the sake of brevity, will often refer to as "adjacency matrix" in the remainder of this text.

No further improvements regarding memory consumption can be done (unless the resulting bit pattern can be compressed with some algorithm). For a graph

with $n$ vertices, $\binom{n}{2} = \frac{n^2-n}{2} \in O(n^2)$ bits of information are required at most. Figure 2.3 illustrates this for $n = 5$. As seen the memory complexity is squared with respect to the number of vertices, so we might get memory problems when the number of vertices increases—especially if we need to store many graphs (later we will see that our algorithm will need at least $n+1$ graphs, which implies that we get a cubic memory complexity, which is uncomfortable).

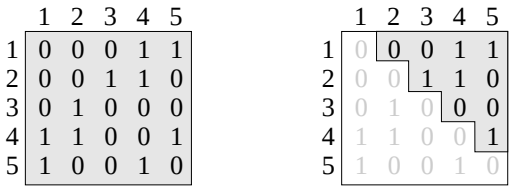



FIGURE 2.3. The optimized adjacency matrix representation reduces the memory consumption by half, but it adds more unwanted complexity.

An advantage of the adjacency matrix representation is that it is fast. If one wants to know whether two vertices are connected one only has to find the right index in the matrix and check the corresponding element. That is an operation which takes constant time.

### Finding the right index in an optimized adjacency matrix

We will store the bits in a dynamically allocated array of bytes. That means we need $\left\lceil \frac{n\cdot(n-1)/2}{8} \right\rceil$ bytes for a graph of order $n$. Seven bits might be wasted, but worse is that the reduction of matrix size makes it tricky to find right indices in the array. We need a function to map a pair of vertex numbers to an index. Once we have that we will have a minimal but efficiently working data structure.

Before continuing let us make a definition.

**Definition**. *The term index refers to a position in the array representing the adjacency matrix. The term adjacency matrix vertex, or* AMV*, refers to the matrix ordering number of a vertex.*

In other words an index refers to an element of the matrix, or equivalently a pair of coordinates. AMV specifies a coordinate. For instance, in Figure 2.4 we can check whether the two vertices with AMVs 3 and 5 are adjacent by examining index 12 of the adjacency matrix.

Suppose we want to find the index of two vertices numbered $a$ and $b$ (given that AMVs start at 1 and increases unitary, the first element of the array has index

0 and we have $n$ vertices in the graph). Also assume, without loss of generality, that $a > b$. For convenience in the rest of this text, let us call the indices of the vertices $\alpha$ and $\beta$ respectively, and define $a$ to be the greatest AMV and $b$ to be the smallest AMV. More formally we define a function

$$f\left(\begin{bmatrix} \alpha \\ \beta \end{bmatrix}\right) \underset{\alpha \neq \beta}{=} \begin{bmatrix} a = \max(\alpha, \beta) \\ b = \min(\alpha, \beta) \end{bmatrix}$$

mapping $(\alpha, \beta) \rightarrow (a, b)$. Figure 2.4 visualizes $a$ and $b$.

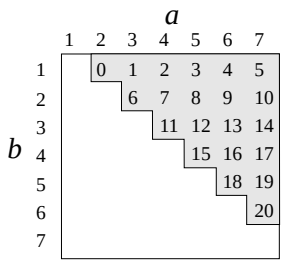



FIGURE 2.4. In this matrix, $n = 7$. We only need $(7^2-7)/2 = 21$ bits to represent this matrix, as can be seen in the figure.

We now want to find what index corresponds to $a = 4$ and $b = 3$, for instance. The answer is 11, as can be seen in Figure 2.4. A function ($\mathbf{N}^3 \rightarrow \mathbf{N}$) is needed to map $a$, $b$ and $n$ onto a single index.

To make reasoning simpler, consider Figure 2.5 showing how the matrix is stored in a linear array.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

FIGURE 2.5. The matrix of Figure 2.4 shown in a linear fashion. The vertical lines mark the beginning of a new row in the corresponding matrix.

Let $g$ indicate the index of the start of every row in the corresponding matrix. It is seen that $g$ depends on $b$, the row number, and $n$, the order of the graph. The function mapping $b$ and $n$ to $g$ is

$$g(b, n) = \begin{cases} \sum_{m=2}^{b}(n - m + 1) & b > 1 \wedge a > b \\ 0 & b = 1 \wedge a > 1 \\ undef. & a \leq b \end{cases}$$

Now, given $g(b,n)$ let us find the offset value $o$ in the row, enabling us to find the right index $i$ by adding $g$ and $o$. It is not hard to see that the function $o(a,b) = a-b-1$ will provide the correct solution. The iterative summing procedure of the $g$ function can be replaced with a simpler formula if we only consider the cases where $a > b$ and $b > 1$. The summation rule $\sum_{k=1}^{n} k = n(n+1)/2$ can be used. Rewrite $\sum_{i=2}^{b}(n-i+1)$ to $(b-1)(n+1) - \sum_{i=2}^{b} i$. Then note that the summation rule can be applied if $\sum_{i=2}^{b} i$ is rewritten to $-1 + \sum_{i=1}^{b} i$. Now we can express $g(b,n)$ as $bn + b - n - 1 - \frac{b(b+1)}{2} + 1$.

Checking whether two vertices are adjacent is an operation which is conducted a large number of times in a graph algorithm, and our algorithm is no exception. Therefore any simplifications of the above formula (resulting in fewer computations) will yield nice time savings.

$$
\begin{aligned}
g(b,n) = \quad & bn + b - n - 1 - \tfrac{b(b+1)}{2} + 1 = \\
& b(n+1) - \tfrac{b}{2}(b+1) - n = \\
& \tfrac{b}{2}\left(2(n+1) - (b+1)\right) - n = \\
& \tfrac{b}{2}(2n + 2 - b - 1) - n = \\
& \tfrac{b}{2}\left((2n+1) - b\right) - n
\end{aligned}
$$

So finally we get the function $i(a,b,n) = \frac{b}{2}\left((2n+1) - b\right) - n + a - b - 1$. As can be seen, the function is quite complex requiring many more arithmetic operations than a simple lookup in a two-dimensional array. This extra complexity is the price for gaining memory. However, the function can still be used in $O(1)$ time, so the loss in time efficiency is not asymptotically significant. Its details can also be hidden nicely by abstraction.

### Dynamic optimized adjacency matrices

We now have a memory efficient and acceptably fast data structure for representing an undirected graph. However, the structure is static and a problem we still have to deal with is how to modify the data structure when vertices are added or deleted from the graph.

The linked list representation easily allows vertex deletion and insertion—only some straightforward routines manipulating the lists are needed. Adjacency matrices are harder to modify.

One idea is to simply make an additional graph, copy whatever is needed from the original graph to the new graph and possibly extend it, and finally delete the original graph. This is straightforward, but requires two graphs in the memory

at the same time, which is unacceptable. We therefore need to modify the actual data structure, deleting or adding data.

### Inserting vertices

Consider the operation of adding $k$ vertices to the graph. The linear array of bits must then be modified in many places reflecting the insertion of $k$ matrix elements into each row and the appending of $k$ new rows to the matrix.

Recall that our array was allocated dynamically.[1] Since the graph is to be extended we need to append more allocated memory to the array.[2]

When the array has been enlarged the bits have to be moved since with a new $n$, where $n$ is the number of vertices, the previous locations in the matrix do not correspond any longer to the same vertices. When moving bits around we must not overwrite bits that has not already been moved to a safe place. Since bits are to be moved to higher indices, we have to work from the end of the array to the start. Whenever we find a 1, we observe its current index, calculate the new index (which can not be smaller), replace the 1 with a 0, and put a 1 at the calculated location. Figure 2.6 illustrates an example where $n = 5$ is increased to $n = 6$.

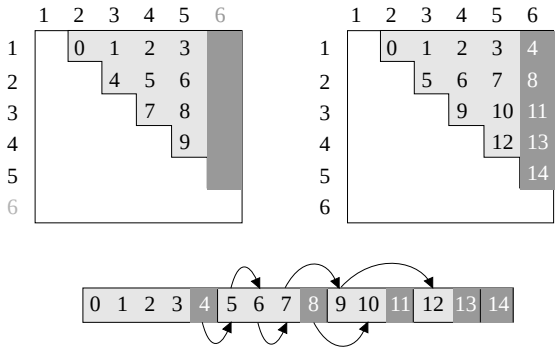



FIGURE 2.6. The elements of the array need to be copied around when the graph changes its size. Here a graph with 5 vertices is transformed into a graph with 6 vertices.

Given an old index $i_n$, the size $n$ of the original graph $G$, and the enlargement parameter $k$, how do we find the new index? Assume that we know in what row the index resides in $G$, let us call it $b$ as usual.

[1] A `malloc` call is used in C to allocate memory dynamically.

[2] In C the function `realloc` is used to append more memory to a previously allocated array of bytes. Here we assume that all bits are cleared when allocated.

We will see that we can easily compute the new index $i_{n+k}$ by using our function $i(a,b,n)$ previously defined. Simply replacing $n$ with $n+k$ yields the equation

$$i_{n+k} = \left(\frac{b}{2} \cdot (2\,(n+k)+1-b) - (n+k)\right) + (a-b-1)$$

This expression is quite complicated though, and we should be able to come up with a smaller formula. Let us instead find the difference $\Delta_+^k$ between the new and old index. Recall that we were given the current index $i_n$. The equation

$$i_n = \left(\frac{b}{2}\,(2n+1-b) - n\right) + (a-b-1)$$

still holds and by taking the difference $i_{n+k} - i_n$ we get $\Delta_+^k$.

$$\Delta_+^k = i_{n+k} - i_n = bk - k = k(b-1)$$

So now we can find the new index $i_{n+k}$ seemingly elegantly by using the function

$$i_{n+k} = i_n + \Delta_+^k = i_n + k(b-1)$$

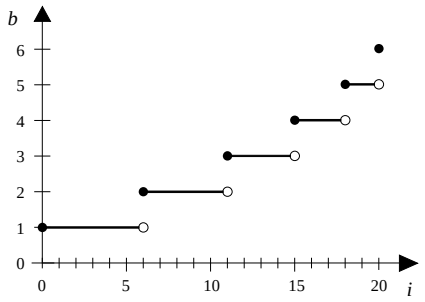



FIGURE 2.7. The mapping from $i_n$ and $n$ to $b$ is given by a discrete function, as seen above for $n = 7$.

However, we still have a tricky problem left. In the above calculations we assumed that we know $b$, the row number in the matrix corresponding to the index $i_n$. We have to find this value. Now recall that we do have a function $g(b,n)$ yielding the first index of a row given the row number and the order of the graph. Then the equation $i_n - g(b,n) = 0$ will hold. Solving this equation for $b$ yields two solutions. One is discarded, but the other $b = \left\lfloor \left(n+\frac{1}{2}\right) - \sqrt{\left(n+\frac{1}{2}\right)^2 - 2\,(i_n+n)} \right\rfloor$ is correct, and is displayed in Figure

2.7 for $n = 7$. Unfortunately this function is not elegant, and the final expression for the new index becomes

$$i_{n+k} = i_n + k\left(\left\lfloor\left(n+\frac{1}{2}\right) - \sqrt{\left(n+\frac{1}{2}\right)^2 - 2\left(i_n+n\right)}\right\rfloor - 1\right)$$

**Deleting vertices**

Deleting vertices is similar to inserting vertices, but there is an important difference. When adding vertices, only the number of vertices $k$ to be added needs to be specified. When deleting $k$ vertices we need more information than just the number $k$. We also need to know *which vertices* are to be deleted if we want a more useful operation than just random deletion.

Let us outline the algorithm we will use for vertex deletion. Let $L$ be the set of vertices to be deleted. Initialize the variable $d$ marking entries to be deleted to 0. Let $A$ be the array of bits representing the adjacency matrix. Go through all indices $i$ in $A$ from the beginning (starting with index 0) to the end. Call a function $h(i)$ deciding whether the entry at index $i$ should be preserved in the final matrix. If $h(i)$ returns false implying that the entry at index $i$ corresponds to a vertex which will be deleted, we simply add 1 to $d$ and move to the next index. If $h(i)$ returns true, which means that the entry at index $i$ is still needed, the entry at index $i$ is moved to index $i-d$. The example of Figure 2.8 provides intuition.

The function $h(i)$ has to be constructed. $h(i)$ decides whether index $i$ should be preserved by computing what row $b$ and what column $a$ the index belongs to in the adjacency matrix. We already have a function for computing $b$, but none for $a$ yet. Recall that we constructed a function $g(b,n)$ that finds the first index of row $b$ in a matrix representing a graph with $n$ vertices. We found that the equation $i = g(b,n) + (a-b-1)$ holds[3] and can now use it to find $a$ since we know $g(b,n)$ and $b$. When solving for $a$ we get

$$a = i + b + 1 - \left(\frac{b}{2}\left((2n+1) - b\right) - n\right)$$

$h(i)$ returns true if $a \in L \vee b \in L$ and false otherwise.

Let us analyze the time complexity of the deletion operation. There are $\frac{n^2-n}{2} \in O(n^2)$ indices to be processed with calls to $h(i)$ if there are $n$ vertices in the graph. If we hash all members of $L$ into a hash table with $n$ elements, using the identity function as hash function, $h(i)$ will have $O(1)$ time complexity since calculation of $a$ and $b$ takes constant time (assuming arithmetic operations are $O(1)$) and whether $a$ and $b$ are members of $L$ can be looked up in $O(1)$ time. In total we get $O(n^2)$ performance, which is good.

[3] $(a-b-1)$ is the offset value to be added to $g(b,n)$ to find index $i$.

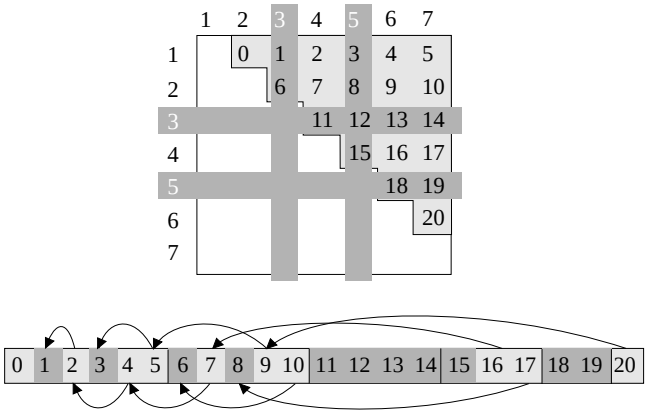



FIGURE 2.8. Deleting vertices in an adjacency matrix. Here we have $n = 7$ and wish to delete vertices 3 and 5. The figure displays which elements are removed and how the remaining elements are moved, resulting in a graph with $n = 5$.

### Identifying vertices

With the ability to dynamically insert and delete vertices, our data structure is flexible. In many applications, however, there is a need to identify vertices and we can not do that yet. For example consider the operation of deleting vertices. When some vertices are deleted, the remaining vertices often change AMVs and we lose track of them. In our algorithm the ability to identify vertices is essential. Thus we need to extend our data structure further.

Identification of vertices is possible if each vertex is assigned a unique name. A name could be any combination of characters, but for efficiency reasons we will only use integers as names. In algorithms we will often refer to vertices by name and let a function find the corresponding AMVs in the adjacency matrix. More formally let $x$ be the name of a vertex. The function $s(x)$ maps $x$ to its corresponding AMV $\alpha$. $s^{-1}(\alpha)$ works the other way around mapping the AMV $\alpha$ to the corresponding name $x$. If the name of a vertex $v$ is $q$, we may say that $v$ *is called* $q$.

When we refer to vertices in pairs we can identify three layers of abstraction. The names $x$ and $y$ of two vertices are mapped by $s$ to their AMVs $\alpha$ and $\beta$ respectively, and those AMVs are mapped by the previously defined function $f$ to find the right $a$ and $b$ to be used when retrieving adjacency information from the matrix. It is crucial that $s$ is bijective in order to avoid conflicts. A name must only refer to one AMV, and an AMV must only refer to a single name.

How should names be incorporated into the data structure? One way to solve the problem is by introducing a database where all the name-AMV pairs are

stored, thereby implementing both $s$ and $s^{-1}$. We will implement the database with an array. This solution requires $2n$ elements in the database if the order of the graph is $n$. However, if the elements are unordered, a lookup will require $\frac{n}{2}$ comparisons on the average, which is not efficient. Therefore we will keep the array sorted with respect to names and AMVs. That will enable us to use the binary search algorithm, which takes $O(\log n)$ time to find an element in a sorted array with $n$ elements.

Recall that no matter what operations we choose to perform, the AMVs of the adjacency matrix start at 1 and increase to $n$ with unitary steps, if the order of the graph is $n$. Therefore let us keep the pairs sorted with respect to AMVs. Now note that finding the right name given an AMV is an $O(1)$ operation. Also observe that storing the AMVs is not necessary, reducing the array's size by half. Figure 2.9 illustrates this fact.

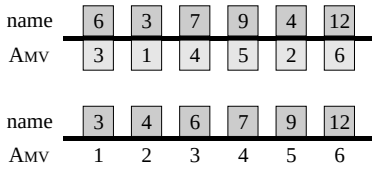



FIGURE 2.9. Different name-AMV database implementations using arrays. Retrieval is faster with a sorted array. Note that the AMVs need not be stored in such an array.

To be able to use binary search when searching for a name we must be sure that the array always is sorted with respect to the integer names. We will prove this after stating some rules.

- Let $m$ be the maximum integer name of the graph. When a vertex is inserted it is given the name $m + 1$. (This reflects the behavior of our insertion algorithm which always inserts vertices with greater AMVs.)
- Vertices can never change names, *i.e.* renaming is not allowed.

**Theorem 2.1**. *The name-*AMV *array is sorted with respect to* AMV*s and integer names.*

*Proof.* A graph can only be modified with respect to its vertices and edges. Modifying the edges does not affect the name-AMV array. Thus it suffices to show that a sorted array is left sorted after insertion or removal of vertices.

An added vertex always gets AMV $n + 1$ if the graph had $n$ vertices before the addition of the vertex. According to a rule, the name of the new vertex will be

$m+1$ if the greatest vertex name of the graph was $m$ prior to the insertion of the vertex. Thus insertion will add an element to the end of the array, which will remain sorted. Inductively this will hold no matter how many vertices we add.

Deletion of a vertex will remove one element from the array. The deletion procedure will compact the array in order to save memory. Removing one element from a sorted array results in a shorter but sorted array. (The AMVs are trivially always sorted.) Inductively this will hold no matter how many vertices we remove. This completes the proof. ■

The introduction of names forces us to do minor modifications of the insertion and deletion algorithms previously discussed. A variable keeping track of the graph's greatest integer name is needed, and the database has to be updated whenever vertices are inserted or deleted. These are straightforward extensions though and they do not affect the time complexity of the algorithms.

There is a problem with this scheme that we want to address, although it will not cause us any trouble in practice. A new vertex is given the name $m+1$ if the previously greatest name is $m$, even if there is an available name less than $m$, left by a deleted vertex. Thus it is possible that the integer names become large after many deletions and insertions, despite the order of the graph being low. One way to solve this problem is to, when inserting a vertex, search for a name left by a deleted vertex. That would, however, force us to rewrite our insertion algorithm to reflect the fact that a new vertex is no longer assigned the greatest AMV of the adjacency matrix.

To eliminate the need of using $O(\log n)$ binary search to map a name to an AMV, it is possible to introduce a second array mapping names to AMVs. The array would be used as a hash table with the identity function as hash function. Many entries in the table would be empty, but it would enable us to map names to AMVs in $O(1)$ time.

### Additional functionality

In a graph package some other basic functionality is welcome. The *complement graph* of $G = \langle V, E \rangle$ is defined as $\widehat{G} = \left\langle V, \widehat{E} \right\rangle$ where $e \in \widehat{E} \Leftrightarrow e \notin E$. Given a graph $G$ its complement can be obtained simply by inverting all bits in the array representation of the adjacency matrix, which takes time $O(n^2)$. Finding the *valency* of a vertex $v \in V$ is done by checking adjacencies with all other vertices and counting them, an $O(n)$ operation.

Another very useful feature is *random graphs*. In a random graph, the probability that two vertices are adjacent is $p$, where $0 \leq p \leq 1$. By letting a function (returning 0 or 1 with probabilities $p$ and $1-p$ respectively) decide for each element in the adjacency matrix whether there should be a 0 or 1, a random

graph is obtained. Later we will frequently use random graphs when evaluating our algorithm. The special case $p = 1$ gives a complete graph. This is a $O(n^2)$ operation. The order $n$ of a graph is stored in conjunction with the matrix and is therefore easy to retrieve in $O(1)$ time.

# 3. Algorithm

After having studied some properties of NPC problems and our way of representing graphs, it is now time to focus on our algorithm designed to provide an upper bound on the size of the max-clique in a graph $G$. Hopefully this upper-bound will be $\omega(G)$. Our ambition, when starting this project, was indeed to find a tractable algorithm for solving an NPC problem. As mentioned, there are thousands of such problems. Although they all belong to the same complexity class, we do believe some problems are easier to reason about and are more intuitive than others.

To get a better understanding for our viewpoint, we will look at some classic NPC problems and argue what we think makes these problems difficult to reason about.

### The graph coloring problem

First let us study the graph coloring problem. An undirected graph $G$ is given as input. Assign a color to each vertex so that no adjacent vertices share the same color. Let $\chi(G)$ denote the minimum number of colors needed to do this. Deciding whether $k \leq \chi(G)$ is an NPC problem.

Approximation algorithms often use greedy strategies, but usually the outputs are not optimal. Good decisions are made locally, but in a wider context the algorithm makes sub-optimal choices. When choosing colors, it seems like one has to consider almost every possible coloring.

### The traveling salesman problem

Another well known NPC problem is the traveling salesman problem (TSP). Given is a connected and directed graph with weights (or costs) associated with each edge. A *route* is a path in the graph visiting every vertex once, and its cost is the sum of all the weights the path passes. The problem is to find a route with minimum cost.

Many ways to find good TSP approximations are known. They suffer from the fact that greedy choices sometimes are sub-optimal. Algorithms try to overcome some of this by interchanging a constant number of vertices in a route to see if it gets cheaper, but alike the graph coloring problem, the whole graph has to be considered when making decisions. Intuitively, reducing the problem size seems hard to us.

## The clique problem

The introduction provided a formal description of the clique problem. Recall from Chapter 1 that finding a polynomial-time approximation algorithm which finds cliques whose size is within a factor $n^{1-\delta}$ of $\omega(G)$ for any constant $\delta > 0$, is impossible unless a tractable algorithm can solve an NPC problem. Thus, we can not expect to construct an approximation algorithm that guarantees good performance. The best known approximation algorithm, designed by Boppana and Halldórsson, has a performance guarantee of $O\left(\frac{n}{(\log n)^2}\right)$ [3].

Due to its NPC status, finding correct solutions to the clique problem is assumed to be hard. Tarjan and Trojanowski develop a recursive $O\left(2^{n/3}\right)$-algorithm that solves the clique problem correctly [4].

Our objective is to provide an approximation algorithm that works well on the average, or at least for some types of graphs. We will describe it in detail, and try to analyze and evaluate it by using theoretical and practical methods.

### Applications

Alike the graph coloring problem and TSP, variants of the clique problem arise in a variety of applications. In fact the clique problem is related to the graph coloring problem (optimal graph coloring can be done by finding max-cliques in the complement graph) and clique algorithms might therefore be useful when solving graph coloring problems.

Time tabling and scheduling involves coloring graphs indicating which events can not be scheduled for the same time slot. When assigning different frequencies to mobile radios, two radios sufficiently close can not be assigned the same frequencies, and consequently the graph coloring problem arises if one wants to use as few frequencies as possible [4]. The problem also arises when assigning variables to registers during compilation of a program.

Some cryptographic applications of the clique problem have been proposed by Juels and Peinado [2]. Hidden cliques in random graphs can be used in one-way functions and hierarchical key creation.

Another real world application of the clique problem is printed circuit board testing. When checking whether a circuit board is working correctly, probes are placed on it. Since probes have fixed sizes, only some components can be verified in one pass. If we let a vertex represent a component and an edge represent two components that can be checked simultaneously, a clique is a set of components that can be checked in one pass [4]. The clique problem has also arisen in analysis of archaeological data and pattern matching.

### Interesting questions

There are some interesting questions concerning the clique problem which can be taken into consideration when designing a new clique-finding algorithm:

- Can dividing the problem into many new smaller ones be a productive approach?
- Most algorithms created are deterministic. Can randomization provide any progress?
- Can parallelization be used successfully? Can one solve the problem more efficiently with multiple processors?
- Often we analyze graph algorithms by examining how it behaves when given random graphs as input. What do graphs arising in real world applications of the clique problem look like? Do they differ from the random graphs, and in that case, can we design algorithms that efficiently deal with different classes of "real world graphs?"
- Perhaps an algorithm does poorly for a class of instances of the clique problem, and does well for others. Can these classes of instances be identified?

As a side note, we can observe that one class of instances of the clique problem which we clearly can distinguish is *planar graphs*. A graph is planar if it can be drawn without any edge crossings. It is proven that a graph is planar if and only if it does not contain $K_5$ (the complete graph with 5 vertices) or $K_{3,3}$ (the bipartite graph with 6 vertices, where each vertex has valency 3) shown in Figure 3.1. Thus, if a graph is planar we know that we can not find a clique of greater size than 4. An example of this can be seen in Figure 1.2. In fact Hopcroft and Tarjan has shown that it is possible to determine whether a graph is planar in $O(n)$ time [5], so their test can be used at first in any clique-attacking algorithm. If the graph is planar then we can use a polynomial-time algorithm (checking all subsets of size less than 5) to find the max-clique.

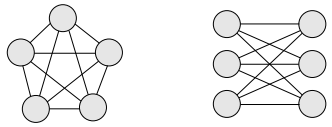


FIGURE 3.1. $K_5$ and $K_{3,3}$. If a graph does not contain $K_5$ and $K_{3,3}$ it is planar.

## Induced subgraphs

The rest of this text will be devoted to our proposed algorithm. Input is a graph $G = \langle V, E \rangle$ and we will calculate an upper bound on the size of the max-clique of $G$. Sometimes $G$ is called a *mother graph*. Let $n = |V|$ as usual. One of the most important concepts of our algorithm, which will be described later, is induced subgraphs. For brevity, we will use the shorter term "subgraph".

**Definition**. *The subgraph of a vertex $v \in V$ of the graph $G = \langle V, E \rangle$ is defined as $G' = \langle V', E' \rangle$ where $E' = \{e = (z, x) | e \in E \wedge z \in V' \wedge x \in V'\}$ and $V' = \{w | w = v \vee (w, v) \in E\}$. The notation $e = (z, x)$ means that the edge $e$ connects vertices $z$ and $x$.*

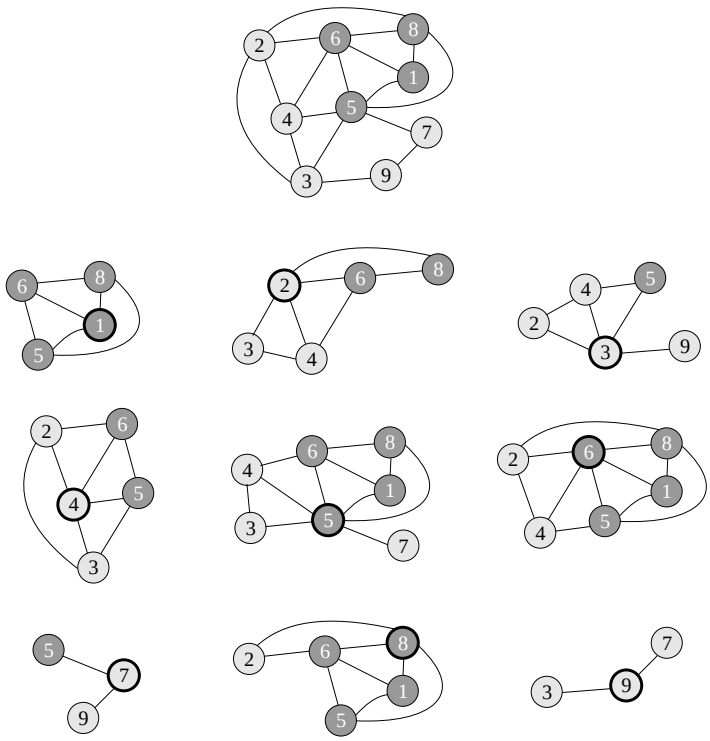


FIGURE 3.2. A graph $G$ with 9 vertices, and its corresponding subgraphs. The master vertices are marked with black borders. The max-clique is marked, and appears in $\omega(G) = 4$ subgraphs.

The vertex $v$, adjacent to all other vertices in its subgraph $G'$, is denoted the *master vertex* of $G'$. Intuitively, the subgraph of $v \in V$ consists of $v$ and all vertices adjacent to $v$, as well as all edges between these vertices that can be found

in $E$. Figure 3.2 shows a mother graph with 9 vertices, and its corresponding subgraphs.

Our strategy is to induce $n$ subgraphs, one for each vertex $v \in V$. In total we obtain $n+1$ graphs. The next theorem will hint at what makes this approach interesting.

**Theorem 3.1**. *(1) The max-clique of $G$ is contained in exactly $\omega(G)$ subgraphs of $G$. (2) Deciding whether a $k$-clique is present in $G$ can be done by applying clique-finding algorithms to $n-(k-1)$ subgraphs of $G$.*

*Proof.* (1) There are $\omega(G)$ vertices part of the max-clique $M$. Each of these vertices $w \in V[M]$ have subgraphs. Since $w$ is part of the max-clique, it is adjacent to all other vertices which are part of $M$, and according to the definition of a subgraph, these vertices will be included in the subgraph of $w$, along with the master vertex $w$ itself and all the edges of the max-clique. Figure 3.2 illustrates this. To see that the max-clique can not appear in more than $\omega(G)$ subgraphs, observe that if it appears in the subgraph of a vertex $u$ not in the max-clique, $u$ is adjacent to all vertices of the max-clique, thus the max-clique size is greater than $\omega(G)$—which is a contradiction. (2) According to (1) the max-clique can be found in exactly $\omega(G)$ subgraphs. Looking after a $k$-clique we need to examine $n-(k-1)$ subgraphs with a clique-finding algorithm to be sure to find a $k$-clique, if one exists in $G$. If we are "unlucky", we will miss $k-1$ subgraphs with $k$-cliques, but the remaining $k$th subgraph with a $k$-clique will be examined according to the pigeon hole principle, and we will find the $k$-clique anyway. ■

### A clique finding algorithm

Implicitly the above proof outlines a simple recursive algorithm, RECURSIVE-CLIQUE, for checking whether a $k$-clique is present in a graph. It induces all the subgraphs of the input graph, and then applies itself recursively to $n-(k-1)$ subgraphs (or $n$ subgraphs for simplicity [the constant $k$ does not affect the time complexity]). To avoid endless recursion, it removes the (redundant) master vertex of each subgraph.

```
RECURSIVE-CLIQUE(G)
1.  n ← order(G)
2.  if n = 1 return 1
3.  for i = 1...n do
4.      G_i ← subgraph of vertex with AMV i without master vertex
5.      c_i ← RECURSIVE-CLIQUE(G_i)
6.  return max(c_1, c_2 ... c_n) + 1
```

Let us study what happens when RECURSIVE-CLIQUE is used. In a graph with order $n$, the algorithm will make $n$ recursive calls on its $n$ subgraphs,

respectively, as seen on line 5. Note that the recursion stops on line 2 when the order is 1. The recursive nature of this algorithm produces a search tree, as can be seen in Figure 3.3. Note that the maximum depth of the tree is $\omega(G)$. The size of the search tree will be dependent on the coverage of the graph. The worst case is given by coverage 1. Then $\omega(G) = n$ and we get $n!$ leaves in the tree. According to Stirling's formula, $n! \sim n^{n+\frac{1}{2}}e^{-n}\sqrt{2\pi}$ and consequently RECURSIVE-CLIQUE is super-polynomial.

However, with coverage $\frac{1}{2}$ we get better performance. Then

$$1 + \sum_{j=0}^{\omega(G)-2} \prod_{i=0}^{j} \frac{n}{2^i}$$

will be the expected number of vertices in the search tree. That is $O(\omega(G) \cdot n^{\omega(G)})$. The expected size of the max-clique in a random-graph $G$ with coverage $\frac{1}{2}$ is $\sim 2\log_2(n)$, so clearly we will not have exponential performance in this particular case, but neither polynomial, because $\lim_{n\to\infty}(\log(n)) = \infty$ and there is no constant to bind $\log(n)$ from above.

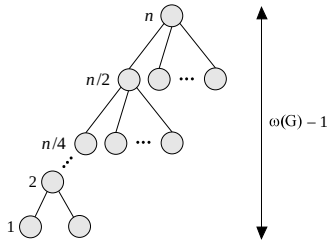



FIGURE 3.3. The search tree produced by RECURSIVE-CLIQUE. Its height is given by $\omega(G) - 1$. With coverage $p$ in a random graph the expected size of the max-clique is $2\log_{1/p} n$.

The problem with RECURSIVE-CLIQUE is that too many recursive calls are produced. When searching for a $k$-clique we saw that at most $n - k + 1$ recursive calls at one level are needed, but that $k$ can not improve the time complexity.

The algorithm can be turned into a probabilistic one. Assuming we have a $k$-clique in a graph with order $n$, at least $k$ subgraphs will contain a max-clique. $\left\lceil \frac{-n\cdot\ln(1-p)}{k} \right\rceil$ recursive calls will ensure us making a recursive call on a subgraph containing a max-clique with probability $p$ [6]. This only gives a constant improvement though.

Our algorithm soon to be presented is inspired by RECURSIVE-CLIQUE, but to admit reasonable running times we will reduce the need of recursion, or completely eliminate it.

## Properties and rules

Subgraphs is one key ingredient of our algorithm. We will now also introduce the concept of *properties* and *rules*. Assume that we will only allow a polynomial number of subgraphs to be created. Now we will assign a constant number $c$ of various properties $p_{v,1}, p_{v,2}, \ldots, p_{v,c}$ to each vertex of every graph. All of these will be initialized to $n$ and can not be less than 0. In our algorithm $c = 1$. Similarly graphs are assigned a constant number $l$ of properties $p_{g,1}, p_{g,2}, \ldots, p_{g,l}$. In our algorithm $l = 1$. Future extensions might increase $c$ or $l$.

Our algorithm will be equipped with a constant number $m$ of rules, forming a set of rules $R$. The rules are designed to decrease $p_{v,1}, p_{v,2}, \ldots, p_{v,c}$ and $p_{g,1}, p_{g,2}, \ldots, p_{g,l}$ as much as possible. The rules are evaluated sequentially, and when no rule can decrease a property the algorithm terminates. In total there is a polynomial number of properties and a constant number of rules. Consequently the running time of this scheme, called PROPERTY-RULE-CLIQUE, is polynomial. However, the quality of the solution is unknown, and so is whether there exists a finite set of rules $R_s$ such that application of PROPERTY-RULE-CLIQUE presented below solves the clique optimization problem.

### Interestingness

The concept of *interestingness* is important. Each vertex in every subgraph has an interestingness. Originally it is $n$. If a vertex in a subgraph is *$k$-interesting*, then we know that the vertex can be a part of a $k$-clique, but not a $(k+1)$-clique. Naturally, our algorithm will try to decrease these values as much as possible. This property is our $p_{v,1}$.

### Greatest possible clique

Every graph will have a number called its *greatest possible clique number*, or GPC. If a graph $G$ has GPC $k$ then it is possible that a $k$-clique exists in $G$, but not a $(k+1)$-clique. The GPC of the mother graph therefore gives an upper bound on the size of its max-clique, and is our primary result. Using the rules, our algorithm PROPERTY-RULE-CLIQUE will try to decrease the GPCs as much as possible. This property is our $p_{g,1}$.

PROPERTY-RULE-CLIQUE($G$)
1. $U = G \bigcup \{\text{The subgraphs of } G\}$
2. **while** a rule $r \in R$ decreases a property in a graph $g \in U$ **do**
3. apply $r$ to $g$
4. **return** GPC of $G$

**Property altering rules**

The rules in $R$ emanate from invariant properties of graphs and cliques, and each rule must fulfill the following requirements: *(1)* An algorithmic implementation corresponding to a rule must not require super-polynomial time. *(2)* Application of a rule must not increase any property. *(3)* A rule must not decrease the GPC of a graph $G$ below $\omega(G)$. *(4)* If a vertex $v$ is part of a $k$-clique in a graph $G$, then a rule must not decrease the interestingness of $v$ in $G$ below $k$.

Now we will present a number of rules, stated as theorems with formal proofs.

**Theorem 3.2**. *Let $G$ be a subgraph and let $m$ be the maximum valency of a vertex in $G$. Then* GPC *of* $G \leq m + 1$.

*Proof.* Assume there exists a $(m + 2)$-clique in $G$. In a clique with $m + 2$ vertices, each vertex must be adjacent to the other $m + 1$ vertices, thus having valency $m+1$, contradicting the assumption that $m$ is the maximum valency of $G$. Hence GPC $\leq m + 1$. ∎

**Theorem 3.3**. *Let $G$ be a subgraph and let $m$ be the number of vertices in $G$ which are $g$-interesting, where $g \geq k$. If $m < k$ then $G$ has* GPC $< k$.

*Proof.* According to the definition, a vertex is $g$-interesting if it is possible that it is part of a $g$-clique but not a $(g + 1)$-clique. To make a $k$-clique in $G$, at least $k$ $g$-interesting vertices are needed, where $g \geq k$. If we can not find those vertices, then clearly there is no $k$-clique in $G$ and GPC$< k$. ∎

**Theorem 3.4**. *Let $i$ be the number of subgraphs with* GPC $\geq k$. *If $i < k$ then the mother graph must have a* GPC $< k$.

*Proof.* Assume there is a $k$-clique in the mother graph. Then at least $k$ of its subgraphs will contain $k$-cliques, and their respective GPCs $\geq k$. If we can not find $k$ such subgraphs, then there is no $k$-clique in the mother graph and GPC $< k$. ∎

**Theorem 3.5**. *Let $m$ be the* GPC *of the mother graph. Then no subgraph can have* GPC $> m$.

*Proof.* Each subgraph is a subset of the mother graph $G$, and the maximum clique of a subgraph can also be found in $G$. Hence, if the maximum clique of the mother graph has size $m$, no subgraph can contain a $(m + 1)$-clique. Thus no subgraph has a GPC $> m$. ∎

**Theorem 3.6**. *If the subgraph of $v$ has* GPC *$k$, then the interestingness of $v$ in any subgraph $\leq k$.*

*Proof.* We know for sure that the clique of maximum size that $v$ is part of (not necessarily the max-clique) is in the subgraph of $v$, $G$. If GPC of $G < k$, then clearly $v$ can not be part of a $k$-clique in any subgraph. ■

**Theorem 3.7**. *Let $G$ be a subgraph with* GPC *$k$. No vertex in $G$ can be $g$-interesting, where $g > k$.*

*Proof.* If $G$ has GPC $k$, then no $(k+1)$-clique can exist in $G$. Then trivially no vertex in $G$ can be more than $k$-interesting. ■

**Theorem 3.8**. *For the subgraph of vertex $v$ to have* GPC *$k$, it is required that $v$ is at least $k$-interesting in at least $k$ subgraphs.*

*Proof.* If $v$ is part of a $k$-clique, then $v$ will occur in $k$ subgraphs, and it will be at least $k$-interesting in those graphs. If $v$ can not be found to be at least $k$-interesting in $k$ subgraphs, then clearly $v$ is not a part of a $k$-clique. ■

**Theorem 3.9**. *Let $v$ be a vertex in a graph $G$ and let $x$ be its valency. If $x < k$, then $v$ is less than $(k+1)$-interesting in $G$.*

*Proof.* If a vertex is part of a $k$-clique, it has at least valency $k-1$ because it is adjacent to all other vertices of the clique. Thus if a vertex has valency $< k$ then it can not be part of a $(k+1)$-clique. ■

**Theorem 3.10**. *Let $v$ be a vertex in a subgraph $G$. If $v$ is not connected to at least $k-1$ $g$-interesting vertices in $G$, where $g \geq k$, $v$ can not be $k$-interesting in $G$.*

*Proof.* Trivially, if $v$ is not adjacent to $k-1$ at least $k$-interesting vertices, then there is no way that $v$ could be part of a $k$-clique. ■

**Theorem 3.11**. *Let $v$ be a vertex in the mother graph and let $m$ be the number of subgraphs in which $v$ is $k$-interesting. If $m < k$ then the subgraph of $v$ can not have* GPC *$k$.*

*Proof.* Assume that $v$ is part of a $k$-clique. Then $v$ appears in the $k$ subgraphs corresponding to the vertices of the $k$-clique. Clearly $v$ will be at least $k$-interesting in all of those subgraphs. Consequently, if $v$ does not appear as $g$-interesting in $k$ subgraphs, where $g \geq k$, $v$ is not part of a $k$-clique, and there is no chance that a $k$-clique exists in the subgraph of $v$. Thus the subgraph of $v$ has GPC $< k$. ■

The union of theorems 3.2-3.11 is our rule set $R$, which we will use in the algorithm PROPERTY-RULE-CLIQUE. To be useful in an implementation, the theorems need to be converted into algorithmic functions which apply the rules respectively to the graphs in memory. They should all return Boolean values, indicating whether the application of the corresponding rule altered any properties. Appendix A discusses the conversion from theorems to algorithms in detail.

# 4. Evaluation

After having described our proposed algorithm, it is now time to evaluate it theoretically and in practice. First we will study the time and memory complexities of the algorithm.

## Time complexity

The algorithm runs until no property can be decreased. An *attempt* to decrease a property is done by applying, in turn, the functions corresponding to theorems 3.2–3.12. How many such attempts occur in the worst case? We have $O(n^2)$ vertices including the subgraphs, and every vertex has a constant number $t$ of properties, all of which are set to $n$ originally. If the application of the functions does not affect any property, the algorithm terminates. Thus, the worst case performance occurs when only one property is decreased by one in every conducted attempt. Since no property can be negative, the algorithm makes at most $tn \cdot O(n^2) = O(n^3)$ attempts.

Now we have to investigate the time complexity of an attempt. Note that we only have a constant number of rules in the set $R$ previously described (see Chapter 3). Assume $|R| = k$, and that the time complexities of the algorithmic implementations of the functions corresponding to the rules in $R$ is $O(c_1(n)), O(c_2(n)), \ldots, O(c_k(n))$ respectively. Then the time complexity of an attempt is $O(c_1(n) + c_2(n) + \ldots + c_k(n))$. Thus, the time complexity of an attempt will be determined solely by the single function $c_i(n)$ which grows fastest when $n \to \infty$. According to our discussion in Appendix A, the algorithmic implementation corresponding to theorem 3.10 has the worst time complexity, $O(n^3)$. Hence the running time of our algorithm is $O(n^3) \cdot O(n^3) = O(n^6)$.[4]

## Space complexity

As usual let $n$ denote the number of vertices in the input graph. With our data structure, $O(n^2)$ bits are required to store the graph. With $n$ subgraphs, we get a total requirement of $n \cdot O(n^2) = O(n^3)$ bits. Also, the properties of the vertices need to be stored somewhere. If each vertex has $c$ properties (in our case $c = 1$) then we get memory complexity $c \cdot O(n^3) = O(n^3)$. Additional structures attached to a graph, like the name-AMV arrays, do not affect this result.

## Experimental evaluation

What our algorithm does can probably be investigated by theoretical means. Theoretically, one can argue that the algorithm will not work well, since it

[4] Later in this text we will show an optimization reducing this to $O(n^5)$.

depends on patterns and correlations between subgraphs, and these patterns disappear as the problem sizes increase. This is indeed a realistic point of view, and we will soon see whether it is correct. Nevertheless, the algorithm might still be useful for some classes of input graphs, like sparse graphs.

Although it is tempting to adopt the pessimistic view and claim that the algorithm does not work well, we want to test the algorithm in practice to learn about its behavior in different situations. Because of this, we have implemented the algorithm and our test results will be presented later in this chapter.

### Naive solution

In many situations during evaluation, knowing the correct solution to a specific instance of the clique optimization problem is useful. That would enable us to verify how well (or badly) our own algorithm performs. Consequently we have implemented an algorithm which, given a graph, finds the optimal solution. This algorithm is super-polynomial and can only be used on relatively small problem sizes. Implementing a naive algorithm is not as easy as it seems. In Appendix B, some naive implementations are discussed.

It can be noted that in many situations finding an actual max-clique might not be essential. In fact, in large graphs one can with an amazing accuracy predict the size of the max-clique. For instance, in random graphs with coverage $\frac{1}{2}$ the max-clique size is expected to be $2\log_2 n$.

### Input sizes

When evaluating our algorithm, how large graphs should be used? To minimize the risk of correlations emerging due to the input graph being too small (and thereby causing the algorithm to perform treacherously well), we will use graphs as large as possible.

First, we should be quite happy with the $O(n^3)$ memory usage of our algorithm. Our equipment will enable us to use graphs with $n \approx 1500$, which we think is sufficient.

Theoretically an algorithm using $O(n^6)$ time is considered to be nice due to its polynomial behavior, but to some people, polynomial algorithms might seem anything but tractable. Figure 4.1 shows the rapidly ascending function $y = n^6$, and although we can hope for much better performance than the worst case, the figure does indicate that the running times will get much longer as the problem sizes increases.

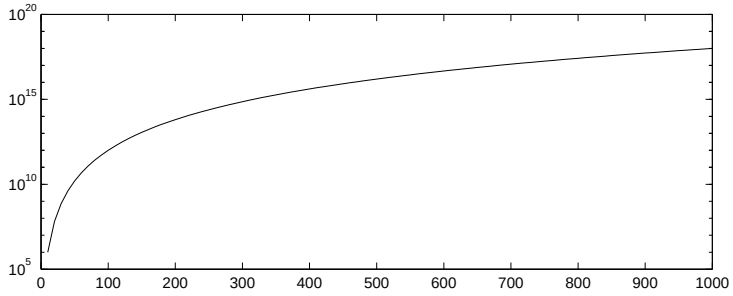



FIGURE 4.1. $y = n^6$ displayed for $0 \leq n \leq 1000$. Obviously the running times increase dramatically with $n$.

Great speed-ups are achieved if the most time consuming rules are abandoned, but on the other hand that might lead to poorer output. This is an issue that should be investigated in the future, if the algorithm proves to be useful.

## Results

Our algorithm has been tested with three different coverages; $p = 0.1$, $p = 0.5$, and $p = 0.9$. In Figure 4.2 the output is shown for $p = 0.5$ and various problem sizes $n$ where $n \leq 1000$.

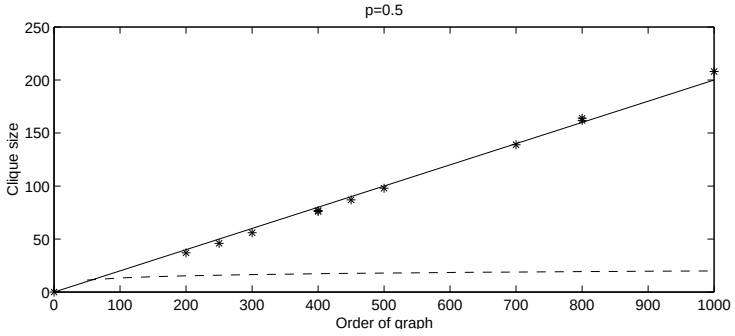



FIGURE 4.2. Results for $p = 0.5, n \leq 1000$. The dashed line is $\omega(G_{n,0.5}) = 2\log_2 n$, the stars are output data, and the straight line is adjusted to fit the stars.

The stars symbolize output data, the straight line is a line adjusted to fit as close as possible to the stars, and the dashed line is $\omega(G_{n,0.5}) = 2\log_2 n$—the expected size of the max-clique of a random graph $G_{n,0.5}$ with order $n$ and coverage $\frac{1}{2}$. We would have preferred the output to follow the logarithmic curve, but clearly

it does not. It is linear with slope $k \approx 0.2$. Indeed, this is a disappointing result, as the algorithm can not be worse than linear ($k = 1$ gives the worst possible algorithm which outputs $n$ as upper bound on the max-clique size for $G_{n,0.5}$). Only for small inputs does the algorithm provide output close to $\omega(G_{n,0.5})$ since $\frac{0.2 \cdot n}{2 \log_2 n} \to \infty$ as $n \to \infty$. Given random input, it is quite interesting that the output fits $k \approx 0.2$ so well.

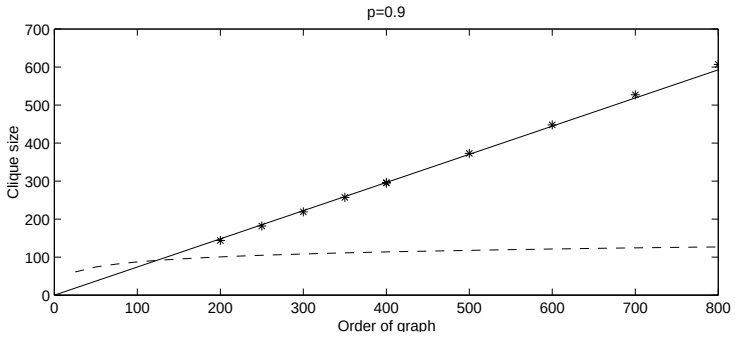



FIGURE 4.3. Results for $p = 0.9, n \leq 800$. The dashed line is $\omega(G_{n,0.9}) = 2 \log_{10/9} n$, the stars are output data, and the straight line is adjusted to fit the stars.

Figure 4.3 shows that the situation does not improve with $p = 0.9$. In this case, the slope is $k \approx 0.74$ and the line diverges even faster from the logarithmic function $\omega(G_{n,0.9}) = 2 \log_{10/9} n$. Again all outputs fit nicely to the line.

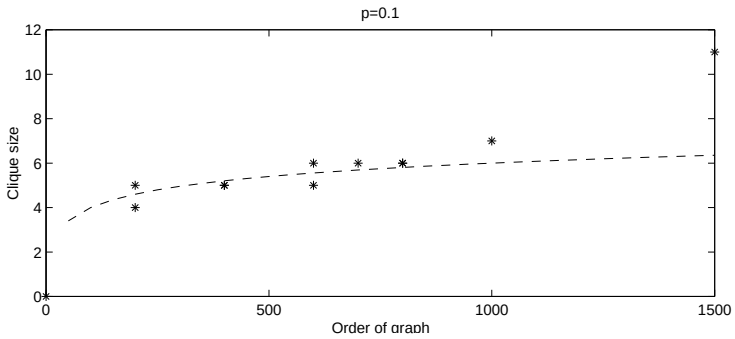



FIGURE 4.4. Results for $p = 0.1, n \leq 1500$. The dashed line is $\omega(G_{n,0.1}) = 2 \log_{10} n$ and the stars are output data.

The situation in Figure 4.4 looks interesting. It shows the output when the random graph is sparse, $p = 0.1$. The data do not suggest sub-linear performance. However, the algorithm performs quite well, often finding the optimal

upper bound $\omega(G_{n,0.1})$ for orders as high as 800. Using a naive algorithm, the max-clique size was most often confirmed to be what was expected.

Although the situation gets dramatically worse for higher orders due to the linear characteristics of the output, the results suggest that the algorithm might be useful for sparse graphs. It has been proposed that asymptotically, the slope of the lines of Figure 4.2–4.4 is approximately $p^2$ [6].

### Time requirements

Figure 4.5 shows the running time of the algorithm during our tests[5]. The data suggest that it gets slower as $p$ is increased. However, for a complete graph ($p = 1$) the algorithm is fast, since it would terminate after just one application of each rule.

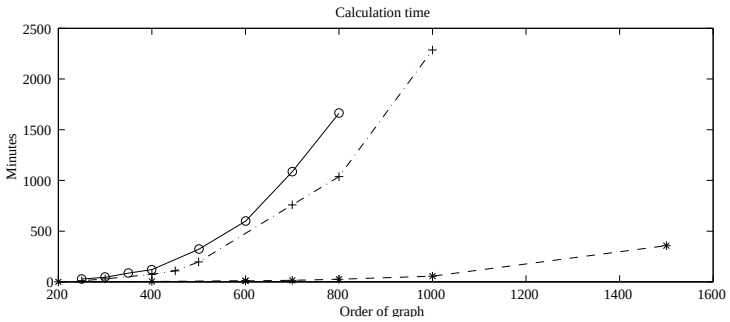



FIGURE 4.5. Execution times for various $n$ and $p = 0.1$ (dashed), $p = 0.5$ (dashed and dotted), and $p = 0.9$ (straight).

A sparse graph ($p = 0.1$) with 1500 vertices required approximately the same time as a dense graph ($p = 0.9$) with 500 vertices.

### Hidden cliques

When the algorithm terminated for random graphs, virtually all properties landed at the GPC, $g$, of the mother graph. We had hoped that there would be several subgraphs with GPCs less than $g$ in order to extend our algorithm with some rules using differences in GPCs. Unfortunately, when the algorithm terminates all vertices seem to have equal properties, making it unnecessary to implement our supplementary ideas.

Additional testing of the algorithm suggests that deliberately hidden cliques with size $s$ within the random graph are spotted if $s > kn$ (*i.e* $s$ is above the

[5]We used Sun Ultra-5 Sparcs with UltraSPARC-IIi 270 MHz processor and 128 MB RWM.

line). This is done by observing that the subgraphs of the vertices part of the hidden clique have a distinct higher GPC value (typically the size of the hidden clique) than the other subgraphs. For instance, with $p = 0.1$ we can often spot cliques larger than 8 for $n < 1000$.

## Optimization

It is possible to speed up the algorithm to $O(n^5)$, without affecting its space complexity. However, we did not implement this optimization[6]. A lot of time is spent in routines counting properties in neighborhood areas, but the need to do that can be eliminated. Equip each vertex in every subgraph with an additional array with $n$ elements, as shown in Figure 4.6. In the array of the vertex $v$ in the subgraph $G$ we store the number of vertices which are at least $k$-interesting and adjacent to $v$ in $G$ for $k = 1, 2, \ldots, n$. $O(n^3)$ memory is required for the arrays. This does not affect the space complexity of the algorithm, but the operation of finding out whether there are $k$ at least $k$-interesting vertices adjacent to the vertex $w$ in the subgraph of the vertex $v$ becomes an $O(1)$ operation—instead of $O(n)$.

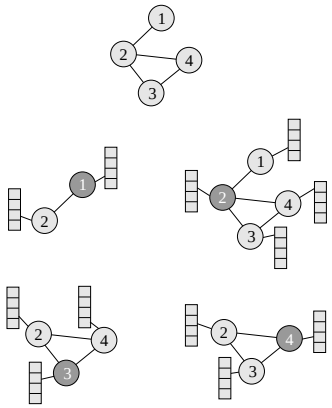



FIGURE 4.6. The running time of the algorithm can be decreased significantly by using an array attached to each vertex.

Assume that $d, 1 \leq d \leq n^3$ properties are decreased by every attempt. Then the time complexity of the algorithm is $O\left(\frac{n^3}{d}\right) \cdot \left(O(n^2) + d \cdot O(n)\right) = O\left(n^5\right)$. The first factor is the number of attempts. The second factor is the number of computational steps required to do an attempt. The first term represents the time it takes to use the rules. If we use the $O(1)$ name-AMV mapping of Chapter 2 it is $O(n^2)$. The second factor represents the updating. $d$ properties have changed (possibly affecting $dn$ arrays), and since the updating of one array takes $O(1)$ time, we get $dn \cdot O(1) = d \cdot O(n)$.

[6] Although the optimization does not require more space asymptotically, it does need a lot of extra memory—and due to lack of memory we would not have been able to conduct some of our tests using the optimization.

# Appendix A

## Algorithmic implementation of rules

Recall that our algorithm uses a set $R$ of rules. These rules were presented as theorems 3.2–3.11. To make use of them in an implementation, they need to be converted into algorithmic functions. In this appendix, we will take a look at how such a conversion can be done.[7]

Each algorithm returns a Boolean value, indicating whether the application of it affected any of the two properties GPC and interestingness. For convenience, Table A.1 summarizes the mapping between the theorems and the algorithms, the later of which will be examined and presented in pseudo code in this appendix.

| Theorem | Algorithm | Time complexity |
|---|---|---|
| 3.2 | ORDER-CHECK | $O(n)$ |
| 3.3 | INTERESTINGNESS-COUNT | $O(n^2)$ |
| 3.4 | GPC-COUNT | $O(n)$ |
| 3.5 | SUBGRAPH-CHECK | $O(n)$ |
| 3.6 | INTERESTINGNESS-SCAN | $O(n^2 \cdot \log n)$ |
| 3.7 | GPC-INFLICT | $O(n^2 \cdot \log n)$ |
| 3.8 | VERTEX-DISTRIBUTION | $O(n^2 \cdot \log n)$ |
| 3.9 | VALENCY-CHECK | $O(n^2 \cdot \log n)$ |
| 3.10 | INTERESTING-NEIGHBORHOOD | $O(n^3)$ |
| 3.11 | GLOBAL-COUNT | $O(n^2 \cdot \log n)$ |

TABLE A.1. Theorems 3.2–3.11 are associated with algorithms presented in this appendix.

```
ORDER-CHECK(G)
1.  r ← false
2.  n ← order of G
3.  for i = 1...n do
4.      H ← subgraph of vertex with AMV i in G
5.      o ← order of H
6.      if GPC of H > o
7.          GPC of H ← o
8.          r ← true
9.  return r
```

First let us study ORDER-CHECK implementing theorem 3.2. Each subgraph is visited at line 4, and the GPC is decreased at line 7 if the number of vertices in

[7] We will not use the optimisation of Chapter 4, nor will we use $O(1)$ name-AMV mapping. This appendix was especially intended to support our implementation efforts.

the graph is lower than the GPC. The for-loop of line 3 gives $n$ iterations, and since we do not use any names in the algorithm, lines 4–8 are $O(1)$. Thus the time complexity of ORDER-CHECK is $O(n)$.

```
INTERESTINGNESS-COUNT(G)
 1.  r ← false
 2.  n ← order of G
 3.  for i = 1 ... n do
 4.      H ← subgraph of vertex with AMV i in G
 5.      c ← 0
 6.      o ← order of H
 7.      g ← GPC of H
 8.      for j = 1 ... o do
 9.          if interestingness of vertex with AMV j in H ≥ g
10.              c ← c + 1
11.          if c < g
12.              GPC of H ← g − 1
13.              r ← true
14.  return r
```

INTERESTINGNESS-COUNT implements theorem 3.3. The nesting of for-loops gives $O(n^2)$ performance, since it is possible that $o = n$, and lines 5–7 and 9–13 are $O(1)$. Each subgraph is visited at line 4, and the number of vertices which are at least $g$-interesting are counted at line 10.

```
GPC-COUNT(G)
 1.  c ← 0
 2.  g ← GPC of G
 3.  n ← order of G
 4.  for i = 1 ... n do
 5.      H ← subgraph of vertex with AMV i in G
 6.      if GPC of H ≥ g
 7.          c ← c + 1
 8.  if c < g
 9.      GPC of G ← g − 1
10.      return true
11.  return false
```

What GPC-COUNT does is counting the number of subgraphs which have GPCs at least as great as the GPC of the mother graph. If those are not numerous enough, the GPC of the mother graph is decreased at line 9. Since the for-loop at line 4 gives $n$ iterations and lines 5–7 are $O(1)$ this is an $O(n)$ procedure. It corresponds to theorem 3.4.

```
SUBGRAPH-CHECK(G)
1.  r ← false
2.  g ←GPC of G
3.  n ← order of G
4.  for i = 1...n do
5.      H ← subgraph of vertex with AMV i in G
6.      if GPC of H > g
7.          GPC of H ← g
8.          r ← true
9.  return r
```

Corresponding to theorem 3.5, SUBGRAPH-CHECK makes sure that no subgraph has a higher GPC than the mother graph. This is a $O(n)$ procedure visiting each subgraph once.

```
INTERESTINGNESS-SCAN(G)
 1.  r ← false
 2.  n ← order of G
 3.  for i = 1...n do
 4.      H ← subgraph of vertex v with AMV i in G
 5.      h ← GPC of H
 6.      o ← order of H
 7.      for j = 1...o do
 8.          K ← subgraph of vertex with AMV j in H
 9.          g ← interestingness of vertex v in K
10.          if g > h
11.              interestingness of v in K ← h
12.              r ← true
13.  return r
```

INTERESTINGNESS-SCAN, implementing theorem 3.6, is a little bit more tricky than the other algorithms to implement. At line 3 all $n$ vertices of $G$ is processed. In particular, let us say we examine vertex $v$ as we do at line 4. Then at line 7, all vertices in the subgraph of $v$ are processed, because we know that $v$ will appear in those subgraphs and will not be allowed to be more than $h$-interesting, where $h$ is the GPC of the subgraph of $v$.

Let us analyze the time complexity of this scheme. At line 4 we find the name $v$ of a vertex with AMV $i$ in the graph $G$, to be used later. The operation is $O(1)$, as is the operation of line 8. However, finding the interestingness of a vertex by referencing to a name and not AMV takes $O(\log n)$ time, since we use binary search to map a name to AMV. Hence lines 9 and 11 are $O(\log n)$. The binary search and the two for-loops of lines 3 and 7 give INTERESTINGNESS-SCAN $O(n^2 \cdot \log n)$ time complexity.

```
GPC-INFLICT(G)
 1.  r ← false
 2.  n ← order of G
 3.  for i = 1...n do
 4.      H ← subgraph of vertex with AMV i in G
 5.      o ← order of H
 6.      p ← GPC of H
 7.      for j = 1...o do
 8.          g ← interestingness of vertex v with AMV j in H
 9.          if g > p
10.              interestingness of v in H ← p
11.              r ← true
12.  return r
```

GPC-INFLICT implements theorem 3.7. This procedure visits every subgraph and makes sure than no vertex can have a higher interestingness value than the GPC of the subgraph it is a part of. The two for-loops of lines 3 and 7 give $O(n \cdot o) = O(n^2)$ iterations, and since the implicit name-AMV conversion of line 10 uses binary search, GPC-INFLICT is $O(n^2 \cdot \log n)$.

```
VERTEX-DISTRIBUTION(G)
 1.  r ← false
 2.  n ← order of G
 3.  for i = 1...n do
 4.      c ← 0
 5.      v ← vertex with AMV i in G
 6.      H ← subgraph of v
 7.      k ← GPC of H
 8.      for each vertex u ∈ V[H] do
 9.          S ← subgraph of u
10.          g ← interestingness of v in S
11.          if g ≥ k
12.              c ← c + 1
13.      if c < k
14.          GPC of H ← k − 1
15.          r ← true
16.  return r
```

The GPCs of the subgraphs are attacked by the algorithm VERTEX-DISTRIBUTION implementing theorem 3.8. The idea is that the subgraph of $v$ can not have GPC $k$ if $v$ is not at least $k$-interesting in $k$ subgraphs. The for-loops of lines 3 and 8 give $O(n^2)$ iterations. The implicit name-AMV conversion of line 10 is $O(\log n)$ so the algorithm is $O(n^2 \cdot \log n)$.

```
VALENCY-CHECK(G)
 1.  r ← false
 2.  n ← order of G
 3.  for i = 1...n do
 4.      H ← subgraph of vertex with AMV i in G
 5.      o ← order of H
 6.      for j = 1...o do
 7.          g ← interestingness of vertex v with AMV j in H
 8.          w ← valency of v
 9.          if g > w + 1
10.              interestingness of v ← w + 1
11.              r ← true
12.  return r
```

VALENCY-CHECK implements theorem 3.9. As there are two nested for-loops visiting every vertex of every subgraph and an implicit name-AMV conversion at line 10, the time complexity of the algorithm is $O(n^2 \cdot \log n)$. VALENCY-CHECK makes sure that no vertex has a interestingness higher than its valency plus one, and since valencies are static it is easy to see that this algorithm is applied successfully at most once.

```
INTERESTING-NEIGHBORHOOD(G)
 1.  r ← false
 2.  n ← order of G
 3.  for i = 1...n do
 4.      H ← subgraph of vertex with AMV i in G
 5.      o ← order of H
 6.      for j = 1...o do
 7.          v ← vertex v with AMV j in H
 8.          k ← interestingness of v
 9.          c ← 0
10.          for each u ∈ the set of vertices adjacent to v in H do
11.              if interestingness of u ≥ k
12.                  c ← c + 1
13.          if c < k − 1
14.              interestingness of v ← k − 1
15.              r ← true
16.  return r
```

Theorem 3.10 is implemented by INTERESTING-NEIGHBORHOOD. This algorithm checks that every vertex in every subgraph can defend its current level of interestingness by verifying that a vertex which claims it is $k$-interesting in a subgraph $S$ is adjacent to $k-1$ other at least $k$-interesting vertices in $S$. The for-loops at lines 3, 6, and 10 give $O(n^3)$ iterations of line 11 since we have at most $n^2$ vertices in the subgraphs, and each of them might be adjacent to $n-1$ other vertices. At lines 8 and 14 we need to do name-AMV conversions, using

$O(\log n)$ binary search. In total we get $O\left(n \cdot (n \cdot (\log n + n))\right) = O(n^3)$ time complexity.

```
GLOBAL-COUNT(G)
 1.  r ← false
 2.  n ← order of G
 3.  for i = 1...n do
 4.      v ← vertex with AMV i in G
 5.      H ← subgraph of v
 6.      k ← GPC of H
 7.      c ← 0
 8.      for each vertex u ∈ V[H] do
 9.          S ← subgraph of u
10.          if interestingness of v in S ≥ k
11.              c ← c + 1
12.      if c < k
13.          GPC of H ← k − 1
14.          r ← true
15.  return r
```

Let us do a quick analysis of the time complexity of GLOBAL-COUNT. We have two for-loops at lines 3 and 8. The name-AMV conversion of line 11 is $O(\log n)$ since we use binary search. Consequently the algorithm is $O(n^2 \cdot \log n)$.

According to Theorem 3.11, if a vertex $v$ is not $g$-interesting in at least $k$ subgraphs, where $g \geq k$, then $v$ can not be $k$-interesting anywhere. In the corresponding implementation of the algorithm GLOBAL-COUNT, what value for $k$ should we use? We shall start at the maximum interestingness value that $v$ has in any subgraph. How do we get that value, without peeking into every subgraph? Theorem A.1 implies that $k$ should be the GPC of the subgraph of $v$.

**Theorem A.1**. *Let $k$ be the highest interestingness value of vertex $v$ in any subgraph. If* INTERESTINGNESS-SCAN *and* VERTEX-DISTRIBUTION *are applied, and no other algorithm thereafter, then $k = m$, where $m$ is the* GPC *of the subgraph of $v$.*

*Proof.* The proof is divided into two parts. (1) INTERESTINGNESS-SCAN makes sure that $v$ can not have a higher interestingness than $m$ in any subgraph. Hence $k \leq m$. (2) Let the GPC of the subgraph of $v$ be $m$. Then assume that $v$ is not $m$-interesting in any subgraph. Then following from part 1 of this proof, $v$ must be less than $m$-interesting in all subgraphs. This is impossible, since after applying VERTEX-DISTRIBUTION the GPC of $v$ could not be $m$ if there are no $g$-interesting vertices, where $g \geq k$. Thus $v$ is $m$-interesting in a subgraph. (1) and (2) implies that $k = m$. ∎

Now we have found the time complexity of each algorithm, and the results are summarized in Table A.1. It is useful to us in the time complexity analysis part of Chapter 4, where we need to know the worst time complexity of all the algorithms.

# Appendix B

## Naive solutions

As discussed in Chapter 4, when evaluating our own polynomial algorithm, it is sometimes useful to know the actual size of the max-clique. With that knowledge we can compare the outputs of a correct problem solver and our algorithm, and we are enabled to see exactly how well our algorithm performs.

Although the size of the max-clique can be estimated accurately in large random graphs, we felt it would be convenient at times, for evaluation and development purposes, to not only know the size of the max-clique, but also where it is located. Therefore we implemented a naive solution to the problem. In this appendix we will address some issues concerning naive solutions.

### Naive iteration

Since the optimization problem of finding the max-clique in a graph is NP-hard, any naive solution is super-polynomial, expressing the fact that we have to test all possible combinations of vertices in the graph, looking for cliques, to be able to tell where the max-clique is located.

A straightforward implementation is NAIVE-CLIQUE-ITERATION. It simply increases a variable $i$ from 0 to $2^n - 1$, where $n$ is the order of the input graph, and lets the binary representation of $i$ indicate which vertices are to be tested. That way we know for sure all combinations will be tested. Figure B.1 illustrates how the max-clique eventually is found.

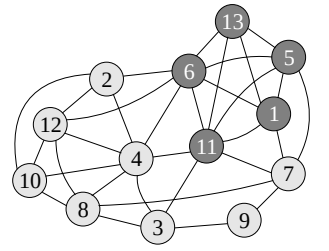
```
NAIVE-CLIQUE-ITERATION(G)
1.  B ← ∅
2.  n ← order of G
3.  for i = 0 ... 2^n − 1 do
4.      H ← INDUCE-BINARY(G, i, n)
5.      if H is complete ∧ |H| > |B|
6.          B ← H
7.  return B
```

The set $B$ always contains the largest clique found so far, and is initiated to $\emptyset$ at line 1. Line 3 increases $i$ from 0 to $2^n - 1$ and at line 4 we use the

algorithm INDUCE-BINARY to get $H$, the subgraph of $G$ corresponding to the binary representation of $i$. If $H$ is complete then lines 5 and 6 test whether $H$ is larger than $B$ and resets $B$ if necessary.

```
INDUCE-BINARY(G, i, n)
1.  V[M] ← E[M] ← ∅
2.  for j = n − 1 . . . 0 do
3.      if i ≥ 2^j
4.          V[M] = V[M] ∪ vertex v with AMV j + 1 in G
5.          E[M] = E[M] ∪ {e|e = (v, w) ∈ E[G] ∧ w ∈ V[M]}
6.          i ← i − 2^j
7.  return {V[M], E[M]}
```

Given the graph $G$, its order $n$ and the number $i$, INDUCE-BINARY induces a graph out of $G$ corresponding to the binary representation of $i$. At line 1 the sets $V[M]$ and $E[M]$ are initialized to $\emptyset$. We will work ourselves backwards in $i$. First we check whether the most significant bit of $i$ is set. If it is, that is $i \geq 2^{n-1}$, then we act accordingly at the lines 4–6. Then we check bits in $i$ sequentially until we have examined the least significant bit.




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

FIGURE B.1. NAIVE-CLIQUE-ITERATION tests all combinations of vertices. Eventually the max-clique is found.

If we execute line 4, we have found a set bit at position $j$. At that line, $V[M]$ is extended with the vertex corresponding to $j$. Line 5 adds all edges connected to the $j$ vertex in $G$ to $E[M]$, but ignores edges not incident of two vertices in $V[M]$. At line 6 the $j$th bit of $i$ is cleared.

The running time of NAIVE-CLIQUE-ITERATION is $\Omega(2^n)$ since the algorithm counts from $2^n - 1$ to 0. Hence it is virtually useless in practical applications.

### Depth First Search

Another naive, but much better way to solve the problem is by building up a search tree which can be searched recursively using the search technique known as *Depth First Search*. The algorithm DFS-Clique does this.[8] Only a minor part of the search tree will have to be stored in memory.

```
DFS-Clique(G)
1.  return DFS-Recursive-Clique(G, ∅, ∅)
```

```
DFS-Recursive-Clique(G, C, B)
 1.  a ← false
 2.  for each u ∈ V[G] − C do
 3.      if {u ∪ C, E[G] − {e|e = (a, b) ∧ (a ∉ u ∪ C ∨ b ∉ u ∪ C)}} is complete
 4.          N ← DFS-Recursive-Clique(G, u ∪ C, B)
 5.          a ← true
 6.          if |N| > |B|
 7.              B ← N
 8.  if a = false
 9.      return C
10.  return B
```

First DFS-Clique calls DFS-Recursive-Clique (the recursive main procedure) with the arguments $G$ (the input graph), $\emptyset$ (the clique currently under investigation), and $\emptyset$ (the greatest clique found so far by the algorithm).

Given these inputs, DFS-Recursive-Clique tries to enlarge the clique by testing whether adding a vertex yields an even bigger clique. $C$ is the set of vertices which are part of the clique currently under investigation.

$B$ is the set of vertices which forms the largest clique found so far. The variable $a$, initialized at line 1, is a flag telling us whether any larger clique was found. If it remains false at line 8, we are at the base case of the recursion and return $C$. Otherwise line 10 returns $B$, the set of the vertices part of the largest clique found so far.

The lines 3–7 check whether the vertices part of the set $U$ can be used to enlarge the current clique. If so, DFS-Recursive-Clique is executed recursively at line 4, and it is this part that produces the search tree.

What makes this algorithm better than the iterative one previously discussed is that it does not continue searching branches which have failed (*i.e.* do not represent cliques) and it requires less time if the input graph is sparse. Also, the recursion depth does not become greater than the size of the max-clique.

[8] We implemented DFS-Clique, but not the optimization described below. Hence our naive problem solver was very slow and could only be used successfully on sparse graphs.

This scheme can be optimized. Assume that all vertices can be sorted with respect to some relation $Q$ (vertex AMVs can be used for this purpose). For a specific clique consisting of vertices in the set $C$ there will be $|C|!$ paths from the root to leaves representing the clique in the search tree. Specifically $\omega(G)!$ paths will represent the max-clique. To avoid exploring all those paths leading to the same clique, only continue searching paths which are sorted with respect to $Q$.

## Acknowledgments

We thank some people who contributed to this project.

- Johan Håstad, professor at KTH, provided mathematical support and comments on the preliminary version of this paper.
- Frej Drejhammar gave us TEX support and contributed to the look of this document. Frej also helped us with UNIX and programming problems.
- For helping us with implementation problems we thank Roland Persson.
- Mikael Goldmann at KTH helped us with programming problems.
- Finally professor Stefan Arnborg should be credited for letting us do this project within the frameworks of a regular KTH course in computer science.

## References

**[1]** Cormen, Leiserson, and Rivest, Introduction to Algorithms, The MIT Press, 1990.

**[2]** Juels and Peinado, Hiding cliques for cryptographic security, *Proceedings of the Ninth Annual ACM-SIAM SODA*, 1998, pp. 678–684.

**[3]** Alon, Krivelevich, and Sudakov, Finding a Large Hidden Clique in a Random Graph, *Proceedings of the Ninth Annual ACM-SIAM SODA*, 1998, pp. 594–598.

**[4]** Unknown author, Clique and Coloring Problems—A Brief Introduction, with Project Ideas, `ftp://dimacs.rutgers.edu/pub/challenge`, 1992.

**[5]** Hopcroft and Tarjan, Efficient planarity testing, *Journal of the ACM, 21:549–569*, 1974.

**[6]** J. Håstad, personal communication.