Test Case Generators and Computational Results for the Maximum Clique Problem

JONAS HASSELBERG
Royal Institute of Technology, Department of Computer Science, KTH, Stockholm, Sweden

PANOS M. PARDALOS and GEORGE VAIRAKTARAKIS
Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611, U.S.A.

(Accepted: 2 November 1992)

Abstract. In the last years many algorithms have been proposed for solving the maximum clique problem. Most of these algorithms have been tested on randomly generated graphs. In this paper we present different test problem generators that arise from a variety of practical applications, as well as graphs with known maximum cliques. In addition, we provide computational experience with two exact algorithms using the generated test problems.

Key words. Maximum clique problem, testing, global optimization.

1. Introduction

Let $G = (V, E)$ be an undirected graph where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices in G, and $E \subseteq V \times V$ is the set of edges in V. Throughout the paper we denote the size of a set V by $|V|$. The adjacency matrix of G is denoted by $A_G = (a_{ij})_{n \times n}$ where $a_{ij} = 1$ if $(i, j) \in E$, and $a_{ij} = 0$ if $(i, j) \notin E$. The complement graph of $G = (V, E)$ is denoted by $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(i, j) | i, j \in V,\ i \neq j$ and $(i, j) \notin E\}$. For a subset $S \subseteq V$ we call $G(S) = (S, E \cap (S \times S))$ the subgraph induced by S.

A graph $G = (V, E)$ is complete if and only if $\forall i, j \in V$, $(i, j) \in E$. A *clique* C is a subset of V such that the induced graph $G(C)$ is complete. The maximum clique problem is to find a clique C of maximum cardinality in a graph G. A *vertex cover* S is a subset of V such that every edge $(i, j) \in E$ is incident to at least one vertex in S. The minimum vertex cover problem is to find a vertex cover of minimum cardinality in the graph G. An *independent set* (*stable set*, *vertex packing*) is a subset of V, whose elements are pairwise non-adjacent. The maximum independent set problem is to find an independent set of maximum cardinality. It is easy to see that S is a clique in a graph $G = (V, E)$ if and only if $V - S$ is a vertex cover in the complement graph $\bar{G} = (V, \bar{E})$, and if and only if S is an independent set of $\bar{G}$. Thus, the maximum clique problem, the vertex cover problem and the maximum independent set problem are equivalent. In addition, they are all $\mathcal{NP}$-complete, which means that unless $\mathcal{P} = \mathcal{NP}$ there exists no

Journal of Global Optimization **3**: 463–482, 1993.

algorithm that can solve either problem in time polynomial to the size of the problem.

For more details about the maximum clique problem refer to [1, 2, 6, 13, 23] and the recent survey [15] which presents results concerning algorithms, complexity and applications. The survey also provides an up to date bibliography on the maximum clique problem.

In many applications, the underline problem can be formulated as a maximum clique problem while in others a subproblem of the solution procedure consists of finding a maximum clique. This necessitates the development of fast exact and approximate algorithms for the problem. Most of the proposed exact and heuristic algorithms have been tested on randomly generated graphs. In this paper we present different test problem generators that arise from a variety of practical applications, as well as graphs with known maximum clique.

The application areas considered in this paper are diverse. For example, we will present a class of graphs from which we can prove or disprove Keller's conjecture; a famous problem in geometry, a part of which is still open. Another example arises from coding theory where one wishes to find binary codes as large as possible that can correct a prespecified number of errors. The problem can be solved by solving the maximum clique problem in a corresponding graph. Also we discuss generation of random graphs with known maximum clique size. Furthermore, we present algorithms that generate all of the graphs discussed. As the graph generators are the main purpose of this paper they are thoroughly described in the text and the pseudocode of every generator is provided.

In the last section we briefly describe two different maximum clique algorithms, and present some computational results revealing some interesting differences between the "hardness" of the different test problems and the algorithms.

2. Coding Theory Problems

In this section we will describe how coding theory problems can be interpreted as maximum clique problems on graphs and we will present the graph generators that produce these graphs.

In Coding Theory, one wishes to find a binary code as large as possible that can correct a certain number of errors for a given size of the binary words (vectors), see [5, 20]. In order to correct errors, the code must consist of binary words among which any two differ in a certain number of positions so that a misspelled word can be detected and corrected. A misspelled word is corrected by replacing it with the word from the code that differs the least from the misspelled one.

The Hamming distance between the binary vectors $u = (u_1, u_2, \ldots, u_n)$ and $v = (v_1, v_2, \ldots, v_n)$ is the number of indices i such that $1 \leq i \leq n$ and $u_i \neq v_i$. We denote the Hamming distance by $dist(u, v)$.

It is well known that a binary code consisting of a set of binary vectors any two of which have Hamming distance greater or equal to d can correct $\lfloor \frac{d-1}{2} \rfloor$ errors

(see [11]). Thus, what a coding theorist would like to find is the maximum number of binary vectors of size n with Hamming distance d. We denote this number by $A(n, d)$.

Another problem arising from Coding Theory, closely related to the one mentioned above, is to find a weighted binary code, that is, to find the maximum number of binary vectors of size n that have precisely w 1's and the Hamming distance of any two of these vectors is d. This number is denoted by $A(n, w, d)$. A binary code consisting of vectors of size n, weight w and distance d, can correct $w - \frac{d}{2}$ errors (see [11]).

2.1. HAMMING GRAPHS

We define the Hamming graph $H(n, d)$, of size n and distance d, as the graph with vertex set the binary vectors of size n in which two vertices are adjacent if their Hamming distance is *at least* d. Then, $A(n, d)$ is the size of a maximum clique in $H(n, d)$.

The graph $H(n, d)$ has 2^n vertices, $2^{n-1} \sum_{i=d}^{n} \binom{n}{i}$ edges and the degree of each vertex is $\sum_{i=d}^{n} \binom{n}{i}$.

2.1.1. Generator of Hamming Graphs

To generate the Hamming graph corresponding to specified values of n, d, we would like to represent each binary vector by a decimal integer in such a way that every digit of the binary vector can be recovered easily from the decimal integer. The easiest way to represent a binary word by a decimal integer is by its decimal equivalence. Also, this representation allows for a quick recovery of any digit of the vector, as we see next. Let

$$\begin{aligned}(x)_{10} &= (a_{n-1} \cdots a_{i+1} a_i a_{i-1} \cdots a_0)_2 \\ &= (a_{n-1} 2^{n-1} + \cdots + a_{i+1} 2^{i+1} + a_i 2^i + a_{i-1} 2^{i-1} + \cdots + a_0)_{10}\end{aligned} \tag{1}$$

such that $x \in \mathbf{Z}^+$ and $a_i \in \{0, 1\}$, $0 \leq i \leq n-1$, where $(u)_b$ denotes an integer u in base b. It is easy to see that

$$a_i = \left\lfloor \frac{x}{2^i} \right\rfloor \bmod 2 .$$

In the computer code that generates Hamming graphs, the user enters the binary vector size, *vsize*, the Hamming distance, d, and the name of the file in which the user wants to save the graph, *outfile* (see the first three lines in Figure 1).

The graph generator uses two integer variables, *vert*1 and *vert*2, that represent the binary vectors. Since the graph is undirected the adjacency matrix is symmetric, and *vert*1, *vert*2 are assigned every possible value so that $0 \leq vert1 < vert2 \leq |V| - 1$. This is done in the two nested loops in lines six and seven in Figure 1. To

```
Program Hamming Graph
 1  vsize ← binary vector size
 2  d ← Hamming distance
 3  outfile ← output file name
 4  n ← 2^vsize
 5  write(outfile) n
 6  for vert1 = 0 to n − 2 do
 7      for vert2 = vert1 + 1 to n − 1
 8          dist ← 0
 9          for pos = 0 to n − 1 do
10              if (⌊vert1/2^pos⌋ mod 2 ≠ ⌊vert2/2^pos⌋ mod 2) then
11                  dist ← dist + 1
12              endif
13          endfor
14          if (dist ≥ d) then
15              write(outfile) TRUE
16          else
17              write(outfile) FALSE
18          endif
19      endfor
20  endfor
```

Fig. 1. Generator of Hamming graphs.

find whether *vert*1 and *vert*2 are adjacent or not, we have to check in how many positions the vectors *vert*1 and *vert*2 differ by checking whether the *pos*-th digit of the two vectors, $pos = 0, 1, \ldots, vsize - 1 pos$, is the same or not. This is done by testing whether

$$\left\lfloor \frac{vert1}{2^{pos}} \right\rfloor \bmod 2 = \left\lfloor \frac{vert2}{2^{pos}} \right\rfloor \bmod 2 .$$

If true, increment the counter *dist*. Once all components are tested, then if $dist \geq d$, *vert*1 and *vert*2 are connected in the graph and *true* is written to the output file (this is done in lines 14 through 18).

The program uses *vsize* to calculate the number of vertices in the graph, $|V| = 2^{vsize}$, and writes the result to the output file so that the size of the graph is easily found by the program in which the graph will be input. Then, the adjacency matrix of the graph will be saved by writing *true* when two vertices are connected and *false* when they are not. The entries in the output file correspond to the upper half of the adjacency matrix. The first $|V| - 1$ entries in the output file form the first row in the matrix, the next $|V| - 2$ entries form the second row, and so on. Hence, the output file will contain an integer denoting the graph size, followed by $\frac{|V|(|V|-1)}{2}$ *true* or *false*.

2.2. JOHNSON GRAPHS

We define the Johnson graph, $J(n, w, d)$, with parameters n, w and d, as the graph with vertex set the binary vectors of size n and weight w, where two vertices are adjacent if their Hamming distance is *at least* d. Then, similar to Hamming graph, the size of the weighted code, $A(n, w, d)$, equals the size of the maximum clique in $J(n, w, d)$.

The graph $J(n, w, d)$ has $\binom{n}{w}$ vertices, $\frac{1}{2}\binom{n}{w}\sum_{k=\lceil \frac{d}{2} \rceil}^{w}\binom{w}{k}\binom{n-w}{k}$ edges and the degree of each vertex is $\sum_{k=\lceil \frac{d}{2} \rceil}^{w}\binom{w}{k}\binom{n-w}{k}$.

2.2.1. *Generator of Johnson Graphs*

Again, when constructing the graph, we would like to represent the nodes, labeled by the binary vectors, by decimal integers. In this case, it is not as easy as with the Hamming graph since the vectors that are in the vertex set are not all the binary vectors of size n. Thus, we find appropriate to represent each vector by a list of indices in which the positions of the components that equal to 1 are stored. Since we have to examine every pair of vectors in the vertex set to see whether they are adjacent or not in $J(n, w, d)$, the index lists must be such that every binary vector that corresponds to a vertex is represented. We do this by assigning a numerical order to the lists, as follows.

Let $u^{(k)} = (u_n, u_{n-1}, \ldots, u_1)$ where $1 \leq k \leq \binom{n}{w}$, be a binary vector in the vector set of $J(n, w, d)$ represented by the index list $index = (i_2, i_2, \ldots, i_w)$ where i_j, $1 \leq j \leq w$, is the index of the j-th 1 from the *right* of $u^{(k)}$. To update *index* into representing $u^{(k+1)}$ instead of $u^{(k)}$, $u^{(k)} < u^{(k+1)}$, $1 \leq k \leq \binom{n}{w} - 1$, find the smallest j, $1 \leq j \leq w$, such that $i_{j+1} - i_j \geq 2$ and increment i_j by 1. Then, for all m, $1 \leq m < j$, set $i_m = m$. In other words, find the first $u_{i_j} = 1$ from the right in $u^{(k)}$ which has a 0 to its left and move that 1 one step to the left, then move all the 1s with index less than i_j in $u^{(k)}$ as far to the right as possible. We illustrate these ideas with the following example:

Let

$$u^{(k)} = (0, 1, 0, 1, 1, 1, 0, 0)$$

$$index = (3, 4, 5, 7).$$

Using our method, we find that the third 1 from the right is the first to have a 0 to its left. We move that 1 one step to the left and we shuffle the less significant 1's to the right. This results to:

$$u^{(k+1)} = (0, 1, 1, 0, 0, 0, 1, 1)$$

$$index = (1, 2, 6, 7).$$

which is the next binary word in numerical order that has w ones.

The Johnson graph generator is given in Figure 2. In the generator, the vector

```
Program Johnson Graph
 1  vsize ← binary vector size
 2  w ← vector weight
 3  d ← Hamming distance
 4  outfile ← output file name
 5  n ← (vsize choose w)
 6  write(outfile) n
 7  for one = 1 to w + 1 do
        index1[one] ← vsize
 8  index1(w + 1) = vsize + 2
    index2(w + 1) = vsize + 2
 9  endfor
10  for vert1 = 1 to n − 1 do
11      moved ← FALSE
12      for one = 1 to w do
13          if (¬moved) then
14              if (index1[one] < index1[one + 1] − 1) then
15                  index1[one] ← index1[one] + 1
16                  moved ← TRUE
17              else
18                  index1[one] ← one
19              endif
20          endif
21          index2[one] ← index1[one]
22      endfor
23      for vert2 = vert1 + 1 to n do
24          dist ← 0
25          moved ← FALSE
26          for one = 1 to w do
27              if (¬moved) then
28                  if (index2[one] < index2[one + 1] − 1) then
29                      index2[one] ← index2[one] + 1
30                      moved ← TRUE
31                  else
32                      index2[one] ← one
33                  endif
34              endif
35              equal ← FALSE
36              for m = 1 to w do
37                  if (index2[one] = index1[m]) then
38                      equal ← TRUE
39                  endif
40              endfor
41              if (¬equal) then
42                  dist ← dist + 2
43              endif
44          endfor
45          if (dist ≥ d) then
46              write(outfile) TRUE
47          else
48              write(outfile) FALSE
49          endif
50      endfor
51  endfor
```

Fig. 2. Generator of Johnson graphs.

size, *vsize* = n and weight w, as well as the Hamming distance, d, and output file, *outfile*, are given as input. The graph size, $\binom{vsize}{w}$, is calculated and written to the output file so that the size of the graph can easily be determined by any other program, using the adjacency matrix of a graph as input. Again, during execution, we will write *true* and false to the output file, in the same manner as for Hamming graphs.

Next, one of the two index lists, *index*1, is initialized by setting the $w + 1$ first positions to *vsize* (see lines 7–9). Note that here we use an index list of length $w + 1$ and not of length w as we did when describing the idea in the previous section. This is only because when the list is updated for the first time, after the initialization, the updating algorithm will find that no 1 in the corresponding binary vector has a 0 to its left and therefore will shift all the 1's to the right, just where we want them in our starting list. Note also that *index*2 does not have to be initialized since *index*1 is copied to *index*2 after the updating (see line 21 in Figure 2).

With *index*1 initialized, we can start examining every pair of vectors to see if they are adjacent or not. Just like for the Hamming graph, this is done in two nested for-loops where the decimal integers *vert*1 and *vert*2, representing the vertices, are assigned every possible value such that $1 \leq vert1 < vert2 \leq n$. The first thing that is done in each loop is the updating of the index lists. *Index*1, representing the vector corresponding to *vert*1, is updated in the outer loop while *index*2 is updated in the inner loop. They are both updated in the same way so we only describe the procedure of updating *index*1. To do this, we use a flag called *moved* which is initially set to *false*. Then we look through the index list, using yet another for-loop, and examine every index i_{one}, $one = 1, 2, \ldots, w$ to see whether the corresponding 1 has a 0 to its left or not. If it has not, we put the 1 back to its starting position by setting *index*1[*one*] = *one*. Otherwise, if the *one*-th 1 has a 0 to its left we increase *index*1[*one*] by one and set *moved* to *true* because now the list is updated and we do not want anything more to be done while finishing the loop. We then assign the same values to *index*2. Thus, when *index*2 is updated in the beginning of the inner loop, it is given the values corresponding to the vector following in numerical order the one represented by *index*1. In Figure 2, *index*1 is updated in lines 11 through 22 and *index*2 in lines 25 through 34.

When both lists are updated we can check the adjacency condition. This is done within the updating loop of *index*2 in the following way: for every index *one* we check if *index*2[*one*] = *index*1[m], $m = 1, 2, \ldots, w$ (lines 36–40 in the figure). If not, we increase the counter *dist* by 2. It is increased by 2 because if there is a 1 in one of the vectors in a position where the other vector has a 0, then the latter has to have a position which has a 1 while the former has a 0. Hence, the Hamming distance is increased by 2 for every unequal value in the index lists (lines 41–43 in the figure). Finally, we check whether $dist \geq d$, and if so, write *true* to the output file, otherwise *false* (lines 45–49).

3. Problems Arising From Keller's Conjecture

A family of hypercubes with disjoint interiors whose union is the Euclidean space $\mathbf{R}^n$ is a *tiling*. A lattice tiling is a tiling for which the centers of the cubes form a lattice.

In the beginning of the century, Minkowski conjectured that in a lattice tiling of $\mathbf{R}^n$ by translates of a unit hypercube, there exist two cubes that share $(n-1)$-dimensional face. About fifty years later, Hajós [8] proved Minkowski's conjecture.

At 1930, Keller suggested that Minkowski's conjecture holds even in the absence of the lattice assumptions. Ten years later Perron [16] proved the correctness of Keller's conjecture for $n \leq 6$. Since then, many papers have been devoted to prove or disprove this conjecture and recently, Lagarias and Shor [10] proved that Keller's conjecture fails for $n \geq 10$. Thus, it is left to prove whether the conjecture holds for $n = 7, 8, 9$.

3.1. THE KELLER GRAPHS Γ_n

We define the graph Γ_n as a graph with vertex set $V_n = \{(d_1, d_2, \ldots, d_n)$: $d_i \in \{0, 1, 2, 3\}$, $i = 1, 2, \ldots, n\}$ where two vertices $u = (d_1, d_2, \ldots, d_n)$ and $v = (d_1', d_2', \ldots, d_n')$ in V_n are adjacent if and only if

$$\exists i, 1 \leq i \leq n\colon\ d_i - d_i' \equiv 2 \bmod 4 \tag{2}$$

and

$$\exists j \neq i, 1 \leq j \leq n\colon\ d_j \neq d_j' \,. \tag{3}$$

In [7], Corrádi and Szabó presented a graph theoretic equivalent of Keller's conjecture. It is shown that, there is a counterexample to Keller's conjecture if and only if there exist a $n \in \mathbf{N}^+$ such that Γ_n has a clique of size 2^n.

Γ_n has 4^n vertices, $\frac{1}{2}4^n(4^n - 3^n - n)$ edges and the degree of each node is $4^n - 3^n - n$. Γ_n is very dense and has at least $8^n n!$ different maximum cliques. It can be shown (see [10]) that the maximum clique size of Γ_n is less than or equal to 2^n.

3.1.1. The Γ_n Generator

To construct the graph Γ_n, we use the same method for calculating the vector components as we did with the Hamming graph in Section 2.1.1 with the exception of interpreting the vertices as integers in base 4 instead of in base 2. Thus, if we have a vector $u = (u_{n-1}, u_{n-2}, \ldots, u_0)$ such that $u_i \in \{0, 1, 2, 3\}$, $0 \leq i \leq n-1$, and if we represent u by its corresponding decimal integer

$$x = u_{n-1}4^{n-1}d + u_{n-2}4^{n-2} + \cdots + u_i 4^i + \cdots + u_0$$

then we can calculate any coefficient u_i as we did for the Hamming graph by

$$u_i = \left\lfloor \frac{x}{4^i} \right\rfloor \bmod 4,\ 0 \leq i \leq n-1 .$$

An algorithm to generate Γ_n is given in Figure 3. As usual, the vector size, *vsize*, and the output file, *outfile*, is given as input by the user. Then, the graph size $n = 4^{vsize}$ is calculated and written to the output file (see lines 1–4 in the figure). In the two for-loops (starting at lines 5 and 6) we let *vert*1 and *vert*2 represent all pairs of nodes. By using two flags, *kongrt* and *diff*, we test the connectivity conditions (2) and (3) for each such pair of nodes. This is done by initializing *kongrt* and *diff* to *false*. Then, for every position *pos* in the vectors, we calculate the difference *sub*, between the *pos*-th component *comp*1 of node *vert*1 and the *pos*-th component *comp*2 of node *vert*2 (lines 10–12). Furthermore, if *sub* equals 2, i.e. if

$$comp1 - comp2 \equiv 2 \bmod 4 ,$$

and *kongrt* = *false*, then *pos* is the first position in the vectors such that the

```
Program Keller Graph
 1  vsize ← binary vector size
 2  outfile ← output file name
 3  n ← 4^n
 4  write(outfile) n
 5  for vert1 = 0 to n − 2 do
 6     for vert2 = vert1 + 1 to n − 1 do
 7        kongrt ← FALSE
 8        diff ← FALSE
 9        for pos = 1 to vsize − 1 do
10           comp1 ← ⌊vert1 / 4^pos⌋ mod 4
11           comp2 ← ⌊vert2 / 4^pos⌋ mod 4
12           sub ← |comp1 − comp2|
13           if (sub = 2 ∧ ¬kongrt) then
14              kongrt ← TRUE
15           elseif (sub ≠ 0) then
16              diff ← TRUE
17           endif
18        endfor
19        if (kongrt ∧ diff) then
20           write(outfile) TRUE
21        else
22           write(outfile) FALSE
23        endif
24     endfor
25  endfor
```

Fig. 3. Generator for Keller graphs.

congruence condition (2) is fulfilled and we set *kongrt* to *true* (lines 13–14). Then, if *diff* is still *false*, we test only the difference condition (3) and if found true, then set *diff* to *true* (lines 15–16). When both *diff* and *kongrt* are *true*, *vert*1 and *vert*2 are adjacent in Γ_n and we write *true* to the output file, otherwise, as usual, *false* (lines 19–23). Again, the output file consists of the graph size followed by the upper half of the adjacency matrix.

4. Problems Arising from Fault Diagnosis

A crucial problem in studying the reliability of large multiprocessor systems, is the problem known as system-level fault diagnosis. The task is to identify all faulty processors (units) in the system. The classical approach to fault diagnosis was originated over twenty years ago by Preparata *et al.* [17], leading to a fault diagnosis model known as the PMC model.

In the PMC model each unit can test some other units and it is assumed that fault-tree units always give the correct results while faulty ones are unpredictable and can output any results. Furthermore, it is assumed that the number of faulty units never exceed some upper bound t. Upon completion of all tests, the results are gathered by a monitoring unit which computes the status of all units based on the gathered results.

The assumption that a fault-free unit always detects faulty units may seem a little optimistic. Also, the upper bound assumption may restrict the model to unrealistic situations. Further, the PMC model is accurate only if the upper bound t does not exceed the number of neighbors of any unit. For large systems, however, the connectivity might be fairly low, making it quite probable that the number of faulty units exceed the number of neighbours for some units.

Yet another assumption in the PMC model, the existence of a central monitoring unit, makes it less reliable. In order to overcome this problem, distributed fault-tolerance was introduced. The goal of this approach is to find a ay to let every fault-free unit to be able to determine the status of every other unit.

The above observations have led to several different models one of which was introduced by Blough [4]. In his model, processors test each other and fault-free units always detect other fault-free processors correctly, while they detect faulty processors with a fixed probability less than 1. No assumption is made about how faulty units behave as testers.

4.1. C-FAT RINGS

In [3], Berman and Pelc study a realistic approach to fault diagnosis by simultaneously relaxing all the three assumptions from the PMC model described above. Their model is based on a probabilistic model presented by Blough performed in a distributed fashion. Consequently, a processor can never by sure that the information it receives is correct. Berman and Pelc define a system design

represented by a class of graphs, G_n. They show that the probability of correct diagnosis of fault processors for such systems, happens with probability at least $1 - n^{-1}$. The algorithm they propose is based on a model where a test by a fault-free unit on a faulty one does not detect a fault with probability q, while they assume that fault-free units never detect faults in each other.

For a given parameter c, a c-fat ring is the graph $G = (V, E)$ defined as follows. Let

$$k = \left\lfloor \frac{|V|}{c \log |V|} \right\rfloor$$

and let $W_0, \ldots, W_{k-1}$ be a partition of V such that

$$c \log |V| \leq |W_i| \leq 1 + \lceil c \log |V| \rceil \text{ for } i = 0, 1, \ldots, k-1 . \tag{4}$$

For $u \in W_i$ and $v \in W_j$ we have $(u, v) \in E$ iff $u \neq v$ and $|i - j| \in \{0, 1, k-1\}$.

A major step in the algorithm proposed in [3] is to find the maximum clique of a c-fat ring. Therefore we construct a c-fat ring generator and perform some computations to see how the maximum clique algorithms perform on such graphs.

4.1.1. C-Fat Ring Generator

An algorithm that generates c-fat rings is given in Figure 4. It works in a manner similar to the other algorithms given earlier in this paper. At first the input $|V|$ and c is given, that is, the graph size and the partition parameter. Then the number of partitions $k = \lfloor \frac{n}{c \log n} \rfloor$ is calculated (see line 5 in Figure 4). The two for-loops assign values to *vert*1 and *vert*2 so that every two nodes are tested to see

```
Program C-fat ring
 1  c ← c
 2  n ← number of vertices
 3  outfile ← output file name
 4  write(outfile) n
 5  k ← ⌊n / (c log n)⌋
 6  for vert1 = 0 to n − 2 do
 7     part1 ← vert1 mod k
 8     for vert2 = vert1 + 1 to n − 1 do
 9        part2 ← vert2 mod k
10        if (|part1 − part2| ≤ 1 ∨ |part1 − part2| = k − 1) then
11           write(outfile) TRUE
12        else
13           write(outfile) FALSE
14        endif
15     endfor
16  endfor
```

Fig. 4. The c-fat ring generator.

whether they are connected or not. Two nodes are connected if they are members of the same or adjacent partitions. We represent nodes of G by the integers 0, $1, \ldots, n-1$. For convenience we let the i-th, $0 \leq i \leq k-1$, part of the partition to contain the vertices labeled $i \cdot m$, $m = 1, 2, \ldots, |W_i|$, where $|W_i|$, the cardinality of partition i which is given in equation 4. By construction, every other k-th node is found in the same partition. This gives us an easy way to calculate which partition a node belongs to. Namely, the partition *comp*1 to which *vert*1 belongs to is calculated by

$$comp1 = vert1 \bmod k$$

(see lines 7 and 9 in the figure). When *comp*1 and *comp*2 are calculated we test whether the difference $|comp1 - comp2|$ is in $\{0, 1, k-1\}$. In this case, *true* is written to the output file, otherwise *false* (see lines 10–13). As in the previous sections, the output produced corresponds to the upper half of the adjacency matrix.

5. Graphs with Specified Clique Size and Density

In [18, 19] Sanchis proposes an algorithm for generating instances of the vertex covering problem. Regarding the difficulty of the problems generated, the reader is referred to [18]. The vertex covering is to find the smallest set of vertices V^* of a graph $G = (V, E)$ such that every edge in G is incident on at least one vertex in V^*. This problem is equivalent to solving the maximum clique problem for $\bar{G}$.

In this section we generate instances of the vertex covering problem according to Sanchis' algorithm and then convert them into instances of the maximum clique problem by using the complement graphs. Thus, if $G = (V, E)$ is a graph with minimum vertex cover of size c generated by Sanchis' algorithm, then the complement graph $\bar{G} = (V, \bar{E})$ has maximum clique size $cl(\bar{G}) = |V| - c$ (see also [15]).

To produce a graph $G = (V, E)$ with $|V| = n$, $|E| = m$ with minimum vertex cover of size c, Sanchis proposes the following. Let $k = n - c$. Choose a partition of the integer n into k parts $n_1, \ldots, n_k$, where $n_1 + n_2 + \cdots + n_k = n$ such that $m' = \sum_{i=1}^{k} \binom{n_i}{2} \leq m$. Form k cliques with sizes $n_1, \ldots, n_k$. For each i, $1 \leq i \leq k$, choose $n_i - 1$ vertices from the i-th clique to be in the vertex cover. Add $m - m'$ additional edges to the graph in such way that each added edge is incident on at least one of the selected cover vertices.

We can see the graph $G = (V, E)$ with $|V| = n$, $|E| = m$ and a minimum vertex cover of size c does not exist unless

$$0 \leq c \leq n - 1 \tag{5}$$

and

$$r\binom{b+1}{2} + (k-r)\binom{b}{2} \leq m \leq \binom{c}{2} + kc \tag{6}$$

where $k = n - c$ and $n = kb + r$. This follows from the fact that a graph with n vertices and minimum vertex cover of size c can have at most $\binom{c}{2}$ edges connecting cover vertices plus $c(n-c) = ck$ edges connecting cover to noncover vertices. Furthermore, a graph with n vertices and minimum vertex cover of size c must have at least $r\binom{b+1}{2} + (k-r)\binom{b}{2}$ vertices (see [19]). Therefore, the algorithm works only on input n, m, c that fulfills the conditions (5) and (6) above.

5.1. GENERATOR FOR GRAPHS WITH KNOWN CLIQUE

In the algorithm presented here, n is partitioned in k parts of nearly equal size and the additional edges are chosen randomly.

The generator is given in Figure 5. As usual the input is given by the user, and it must fulfill conditions (5) and (6). A seed to the **random**-procedure is also given. In this algorithm we write to the output file in a somewhat different way than before. To each *true*, *false* value that we write to the output file, we associate a pointer i. Effectively, we treat the values *true*, *false* as records. This way, the algorithm will be able to update the value of a record from *false* to *true* when the corresponding edge is added. The graph size n is, as seen in lines 6 and 9 in the figure, written in 6 positions to the output file, so the graph can not have more than 999999 vertices, unless the source code is modified.

To proceed on the algorithm, we represent every pair of nodes *vert*1, *vert*2 ($0 \leq vert1 < vert2 \leq n-1$) in two for-loops, and we include *vert*1 in the *part*1 part of the partition, where *part*1 = *vert*1 mod k (lines 11 and 13), just as we did with the c-fat rings in Section 4.1.1. Within the two for-loops, we connect every pair of nodes that are in the same partition by writing *true* to the output file if *part*1 = *part*2, otherwise *false*. For every *true* that we write to the output file we increment the counter *edges* by one (lines 15–20). Now we have created a graph with k cliques each having size $\lfloor \frac{n}{k} \rfloor$ or $\lfloor \frac{n}{k} \rfloor + 1$ and a cover of size c consisting of the nodes labeled $k, k+1, \ldots, n-1$ where the total number of edges equals *edges*. Thus, if *edges* is less than m we have to connect $m - edges$ additional edges to get a graph with the desired number of edges. We do this in a while-loop in which we randomly assign *vert*1 to a cover vertex and *vert*2 to any vertex (lines 25–28). The assignment is done in while-loop that ensures that *vert*1 $\neq$ *vert*2. We make use of a real-valued procedure **random** that takes as argument a *seed* and returns a random number $y \in [0, 1]$.

Suppose we have two nodes , *vert*1 and *vert*2, that we would like to connect. To do this we must know the position in the output file corresponding to edge, (*vert*1, *vert*2). Also, we need to know whether the value of this entry is already *true*. If *vert*1 is less than *vert*2 (lines 29–31) we can calculate the corresponding position in the file in the following way. There are 6 positions in which n is written, followed by a total of $pairs = \frac{n(n-1)}{2}$ entries representing edges, which results to a total of $i = 6 + pairs$ entries in the file. The $rowrest = \frac{(n-vert1)(n-vert1-1)}{2}$ last entries correspond to edges (u, v) such that $vert1 < u \leq v \leq n-1$. The $colrest = n - vert2$

```
Program Sanchis graph
 1  n ← number of vertices
 2  m ← number of edges
 3  c ← cover size
 4  seed ← random number generator seed
 5  outfile ← output file name
 6  k ← n − c
 7  i ← 6
 8  edges ← 0
 9  write(outfile, rec=i) n
10  for vert1 = 0 to n − 2 do
11      part1 ← vert1 mod k
12      for vert2 = vert1 + 1 to n − 1 do
13          part2 ← vert2 mod k
14          i ← i + 1
15          if (part1 = part2) then
16              write(outfile, rec=i) TRUE
17              edges ← edges + 1
18          else
19              write(outfile, rec=i) FALSE
20          endif
21      endfor
22  endfor
23  while (edges < m) do
24      vert1 ← vert2
25      while (vert1 = vert2) do
26          vert1 ← 1 + k + (n − k) · random(seed)
27          vert2 ← 1 + n · random(seed)
28      endwhile
29      if (vert1 < vert2) then
30          vert1 ↔ vert2
31      endif
32      pairs ← n(n−1)/2
33      rowrest ← (n−vert1)(n−vert1−1)/2
34      colrest ← n − vert2
35      i ← 6 + pairs − rowrest − colrest
36      read(outfile, rec=i) tmp
37      if (¬tmp) then
38          write(outfile, rec=i) TRUE
39          edges ← edges + 1
40      endif
41  endwhile
```

Fig. 5. Generator for graphs with known Clique.

entries preceding the last *rowrest* entries correspond to edges (*vert*1, v) such that *vert*2 $< v$. Therefore, $i = 6 + pairs - rowrest - colrest$ is the position in the file representing the edge (*vert*1, *vert*2) (see lines 32–35 in the figure). If this entry is *false* we change it to *true* and increment *edges* by one, otherwise we start again from line 23.

6. Computations

In this section we present some computational experiments using the generated graphs, with two exact algorithms. These computations will help us evaluate the difficulty of the generated instances and the efficiency of the algorithms. The two algorithms tested are the CP-algorithm presented in [6], and the PR-algorithm presented in [13].

In [13] the authors present computational results for graphs of up to 1000 vertices and 150000 edges while in [6] the authors present results for graphs of up to 3000 vertices and over one million edges. Those computations were made on an IBM 3090 computer while our computations were performed on a Sun4 computer. On this machine we were able to solve problems, in a reasonable period of time, where the size of the graphs did not exceed 256 vertices.

6.1. THE CP-ALGORITHM

The CP-algorithm presented in [6] is a simple and efficient algorithm that uses partial enumeration to find the maximum clique in an arbitrary graph. Computations on randomly generated graphs show that it compares favorably with all existing exact algorithms for the maximum clique problem. Next, we give a brief description on the CP-algorithm.

Consider the graph $G = (V, E)$. If the graph is dense, the algorithm orders and relabels the vertices so that $v_1 \in V$ is the vertex of smallest degree in G, $v_2 \in V$ is the vertex of smallest degree in $G - \{v_1\}$ and generally, $v_k \in V$ is the vertex of smallest degree in $G - \{v_1, \ldots, v_{k-1}\}$, $1 \leq k \leq n-2$, where $n = |V|$. Empirical tests show that computation time is greatly reduced if the vertex set is ordered in the above way. When the nodes are ordered the algorithm searches through the branch and bound tree in a depth-first manner.

The algorithm starts assigning to the root of the tree (at level 1), the vertex v_1. Then, v_1 is expanded by the vertices $v_2, v_3, \ldots, v_n$. Suppose we are at level d of the tree. Then there are $d-1$ vertices in the current partial clique. Assume v is the vertex assigned at level $d-1$. Then, we expand v. Suppose $V_d = \{v_{d_1}, \ldots, v_{d_i}, \ldots, v_{d_m}\}$ is the set of all vertices considered at depth d and that the vertices $\{v_{d_1}, \ldots, v_{d_{i-1}}\}$ have already been considered. Then v is expanded by v_{d_i}. Then, we proceed to level $d+1$ by expanding v_{d_i} by all vertices that are adjacent to v_{d_i} and are not included in the current partial clique. The algorithms continues in this fashion.

To eliminate branching, the algorithm uses the following branching rule: Let v_{d_i} be the vertex corresponding to level d. Then, vertices $\{v_{d_1}, \ldots, v_{d_{i-1}}\}$ have already been considered thus allowing for inclusion to the current clique, only vertices in $\{v_{d_{i+1}}, \ldots . v_{d_m}\}$. This means that the current clique cannot have size bigger than $d + (m - 1)$. If this size is less than or equal to the current incumbent size, then it is clear that expansion of v_{d_i} cannot result in a larger clique. In this case the algorithm backtracks to the previous level. If the algorithm is at depth 1 and the inequality holds it terminates and the current clique is maximum. For more details and for the Fortran code see [6].

6.2. THE PR-ALGORITHM

The PR-algorithm presented in [13] is a branch and bound algorithm that is based on a quadratic zero-one formulation of the maximum clique problem. This algorithm is proved to be efficient for dense graphs.

Solving the maximum clique problem for a graph $G = (V, E)$ with $n = |V|$ vertices, is equivalent to solving the following quadratic zero-one program:

$$\min f(\mathbf{x}) = -\sum_{i=1}^{n} x_i + 2 \sum_{\substack{(v_i, v_j) \notin E \\ i>j}} x_i x_j, \quad \mathbf{x} \in \{0, 1\}^n$$

or equivalently in symmetric form

$$\min f(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} A \mathbf{x}, \; A = A_{\bar{G}} - I, \quad \mathbf{x} \in \{0, 1\}^n$$

where $A_{\bar{G}}$ is the adjacency matrix of $\bar{G}$, and I is the $n \times n$ identity matrix. A solution $\mathbf{x}^*$ to the above program defines a maximum clique C for G as follows: if $x_i^* = 1$ then $v_i \in C$ and if $x_i^* = 0$ then $v_i \notin C$ with $|C| = -z = -f(\mathbf{x}^*)$.

To solve the problem stated above the algorithm uses a depth-first branch and bound technique by selecting a binary variable, x_i, and fixing it to zero for one subproblem and to one for the other. To reduce the size of the search tree, the algorithm two different pruning rules. An upper bound rule and a forcing rule. The upper bound rule prunes if a calculated upper bound for the subproblem is less than the current incumbent objective value. The forcing rule prunes by generating only one branch for a given variable if it can be shown that the alternate value can only yield suboptimal solutions.

The algorithm can use two different ways of determining the order with which vertices are considered for branching. The greedy approach and the non-greedy approach. The non-greedy approach branches on the vertex of smallest degree first, leading to a small search tree which takes less time to search through and hence, confirmation of optimality takes less time than the greedy approach.

On the other hand, the greedy approach branches on the vertex of largest degree first, leading to a large search tree where the maximum clique will be found early in the depth-first search. Of course confirming optimality can not be

done until the whole tree is searched which means that the greedy approach is expected to take more time. For these reasons, the greedy approach is only used to heuristically approximate the size of the maximum clique while the actual branching is done by using the non-greedy approach (for details see [13]).

6.3. COMPUTATIONAL RESULTS

The computations were performed on a Sun4 computer with the two exact maximum clique algorithms implemented in Fortran 77. The test graphs were generated by the algorithms described in the previous sections.

The sizes of the solved problems vary from 50 to 256 vertices. Our experience as well as the tables presented here indicate that the CP-algorithm is more effective for large problems while the PR-algorithm is faster for dense graphs. However, for graphs with size exceeding 256 vertices, no algorithm managed to produce answers in reasonable amount of time (a few hours). Also we observed that graphs with large maximum clique size are computationally harder.

In Table I we present computational results when the two algorithms are tested on c-fat rings of different characteristics. In the leftmost column, the partition parameter c is given. Note that even though the problems are of moderate size, the CP-algorithm has some difficulties with dense graphs.

In Table II we present the results from computations on Hamming graphs. In

Table I. Computational results on c-fat rings

c	Number of vertices	Number of edges	Graph density %	Size of max clique	Avg CPU-time (sec)	
					CP-alg	PR-alg
1	100	669	14	10	0.056	0.030
2	100	1450	29	20	0.092	0.092
3	100	2094	42	30	0.420	0.106
4	100	2950	60	40	8.102	3.096
5	100	3700	75	50	104.438	3.166

Table II. Computational results on Hamming graphs

n	dist	Number of vertices	Number of edges	Graph density	Size of max clique	Avg CPU-time (sec)	
						CP-alg	PR-alg
6	2	64	1824	90	32	13.170	6.872
6	3	64	1344	67	8	0.490	4.694
6	4	64	704	35	4	0.080	0.338
6	5	64	224	11	2	0.060	0.0116
7	3	128	6336	78	16	307.388	2576.500
7	4	128	4096	50	8	1.532	21.640
7	5	128	1856	23	2	0.176	1.078
8	2	256	31616	97	79[a]	$>10^5$	$>10^5$
8	4	256	20864	64	16	1029.970	$>10^5$

[a] This is a lower bound, calculated at termination.

the first column the size n of the binary vector is given and in the second the Hamming distance is given. Again we note that the PR-algorithm is more efficient on dense graphs of moderate size while the CP-algorithm can solve larger problems.

In Table III the values of binary vector size, vector weight and Hamming distance for the Johnson graphs are given in the first three columns. Notable here is that the CP-algorithm is the fastest even when the density is rather high.

The results on Keller graphs Γ_n are presented in Table IV. Here n denotes the vector size. Since the size of these graphs is 4^n for given n, we are not able to solve problems with $n > 4$.

In Table V we present the results we obtained by testing the two algorithms on the complement of graphs generated for the vertex cover problem. Both algorithms are efficient when the density is low. However, both algorithms have difficulties for dense graphs. Recall that the graphs are generated in such a way that the first node is always in the maximum clique. Thus, one might suspect that the labeling of the nodes of these graphs makes them easy to test. For this reason, for each test graph, we repeated the experiment with randomly relabeled nodes. It turns out that there is no notable difference in CPU time.

Table III. Computational results on Johnson graphs

n	w	dist	Number of vertices	Number of edges	Graph density	Size of max clique	AVG CPU-time (sec)	
							CP-alg	PR-alg
8	3	4	56	1120	73	8	0.808	7.190
8	3	6	56	280	18	2	0.050	0.022
8	4	4	70	1855	77	14	2.334	14.724
8	4	6	70	595	25	2	0.070	0.228
9	3	4	84	2730	78	12	26.694	229.760
9	3	6	84	840	24	3	0.090	0.366
10	3	6	120	2100	29	3	0.214	3.064
11	2	4	55	990	67	5	0.360	5.188
12	2	4	66	1485	69	6	1.068	18.067
13	2	4	78	2145	71	6	4.500	101.410
14	2	4	91	3003	73	7	15.538	356.480
15	2	4	105	4095	75	7	74.844	2145.660
16	2	4	120	5460	76	8	278.162	7880.820

Table IV. Computational results on Sanchis graphs

n	Number of vertices	Number of edges	Graph density	Size of max clique	Avg CPU-time (sec)	
					CP-alg	PR-alg
3	64	1088	54	5	0.200	1.952
4	256	21888	67	12	6302.600	$>10^5$

Table V. Computational results on Sanchis graphs

Number of vertices	Number of edges	Graph density	Size of max-clique	Avg CPU-time (sec)	
				CP-alg	PR-alg
50	245	20	5	0.048	0.870
50	613	50	5	0.096	0.590
50	980	80	5	1.396	26.952
90	597	15	5	0.088	0.320
90	2003	50	5	0.646	12.592
90	3204	80	5	30.160	841.360
90	3204	80	5	30.220[a]	852.300[a]
130	1677	20	10	0.200	0.988
130	4193	50	10	1.299	20.440
130	4193	50	10	1.725[a]	18.780[a]
130	6708	80	10	>3600	>3600

[a] Calculated on randomly relabeled graphs

7. Concluding Remarks

In this paper we presented a variety of test cases for the maximum clique problem. The generated graphs can be used to test and compare proposed algorithms for the maximum clique problem. Also, we discussed the generation of graphs with known maximum clique size (see also [12]). All Fortran codes of the test generators are available by *e-mail* from *pardalos@math.ufl.edu* or *vairakta@ise.ufl.edu*.

References

1. Balas, E. and J. Xue (1991), Minimum Weighted Coloring of Triangulated Graphs, with Application to Maximum Weight Vertex Packing and Clique Finding in Arbitrary Graphs, *SIAM J. Computing* **20**(2), 209–221.
2. Balas, E. and C. S. Yu (1986), Finding a Maximum Clique in an Arbitrary Graph, *SIAM J. Computing* **14**(4), 1054–1068.
3. Berman, P. and A. Pelc (1990), Distributed Fault Diagnosis for Multiprocessor Systems, *Proc. of the 20th Annual Intern. Symp. on Fault-Tolerant Computing* (Newcastle, UK), 340–346.
4. Blough, D. M. (1988), Fault Detection and Diagnosis in Multiprocessor Systems, Ph.D. Thesis, The John Hopkins University.
5. Brouwer, A. E., J. B. Shearer, N. J. A. Sloane, and W. D. Smith (1990), A New Table of Constant Weight Codes. *IEEE Transactions on Information Theory* **36**(6), 1334–1380.
6. Carraghan, R. and P. M. Pardalos (1990), An Exact Algorithm for the Maximum Clique Problem, *Operations Research Letters* **9**, 375–382.
7. Corrádi, K. and S. Szabó (1990), A Combinatorial Approach for Keller's Conjecture, *Periodica Math. Hung.* **21**(2), 95–100.
8. Hajós, G. (1950), Sur la factorisation des abeliens, *Casopis* **74**, 189–196.
9. Keller, O. H. (1930), Über die lückenlose Erfüllung des Raumes mit Würfeln, *J. Reine Angew. Math.* **163**, 231–248.
10. Lagarias, J. C. and P. W. Shor (1992), Keller's Cube-Tiling Conjecture is False in High Dimensions, *Bulletin AMS* **27**(2), 279–283.

11. MacWilliams, F. J. and N. J. A. Sloane (1979), *The Theory of Error-Correcting Codes*, North Holland, Amsterdam.
12. Pardalos, P. M. (1991), Construction of Test Problems in Quadratic Bivalent Programming, *ACM Transactions on Mathematical Software* **17**(1), 74–87.
13. Pardalos, P. M. and G. P. Rodgers (1992), A Branch and Bound Algorithm for the Maximum Clique Problem, *Computers and Operations Research* **19**(5), 363–375.
14. Pardalos, P. M. and G. Vairaktarakis (1992), Test Cases for the Maximum Clique Problem, *COAL Bulletin* **21**, 19–23.
15. Pardalos, P. M. and J. Xue (1992), The Maximum Clique Problem, Manuscript, University of Florida.
16. Perron, O. (1940), Über lückenlose Ausfullung des n-dimensionalen Raumes durch kongruente Würfel, *Math. Z.* **46**, 1–26, 161–180.
17. F. P. Preparata, G. Metze, and R. T. Chien (1967), On the Connection Assignment Problem of Diagnosable Systems, *IEEE Trans. Electr. Comput.* **16**, 848–854.
18. Sanchis, L. (1992), Test Case Construction for the Vertex Cover Problem (extended abstract), *DIMACS Workshop on Computational Support for Discrete Mathematics*, March.
19. Sanchis, L. (1989), Test Case Construction for NP-Hard Problems (extended abstract), *Proceedings of the 26th Annual Allerton Conference on Communication, Control, and Computing*, September.
20. Sloane, N. J. A. (1989), Unsolved Problems in Graph Theory Arising from the Study of Codes, *Graph Theory Notes of New York XVIII*, 11–20.
21. Stein, S. K. (1974), Algebraic Tiling, *Amer. Math. Monthly* **81**, 445–462.
22. Szabó, S. (1986), A Reduction of Keller's Conjecture, *Periodica Math. Hung.* **17**(4), 265–277.
23. Xue, J. (1991), Fast Algorithms for Vertex Packing and Related Problems, Ph.D. Thesis, GSIA, Carnegie Mellon University.