

CSC520

Analysis of Algorithms

Exact Algorithm for the Vertex Cover Problem
in Simple Undirected Random Graphs

Stefan Pantazi

School of Health Information Science

University of Victoria

July 31, 2002

Abstract

This report contains theoretical and practical information about a Java implementation of an exact algorithm for finding a vertex cover of a given size in simple undirected random graphs. The algorithm is based on a combination of divide-and-conquer and backtracking techniques which are often employed for solving combinatorial optimization problems exactly. Moreover, the vertex cover problem is approached by replacing it with its complement problem, i.e. the problem of the independent set.

In order to test the algorithm on different classes of simple undirected random graphs, three different types of random graphs generators are described and implemented as well. The first two types generate random graphs with a number of edges or the edge density specified by the user. The third type generates the special class of unit square random graphs whose vertices correspond to points in the unit square and whose edges connect only vertices which are situated under a specified, maximum Euclidean distance apart.

Executive summary

The first chapter of the paper deals with generating various classes of simple, undirected random graphs. Three different graphs generators are introduced and their algorithms described. Finally, a Java implementation of the generators is presented together with three examples.

The second chapter presents an exact algorithm for the vertex cover problem. The algorithm's pseudo-code is explained and tested on different classes of generated random graphs. In addition, the algorithm is also tested on random graphs whose vertex cover set sizes are known.

Chapter 1

Random Graph Generators

1.1 Introduction

A *graph* G consists of a set V of *vertices* and a set E of *edges* where each edge in E is associated with a pair of vertices in V . A *simple graph* is a graph with no loops or multiple edges. In an *undirected graph* pairs of vertices in V are unordered. All graphs considered in this report are simple and undirected.

There are many classes of random graphs which differ by certain parameters like edge density, edge distribution, planarity, etc. In this chapter only the following types of graph generators are discussed:

1. fixed number of edges random graph generator
2. predefined edge density random graph generator
3. unit square random graph generator

1.2 Random Graph Generation Algorithms

1.2.1 Fixed number of edges random graphs

The first algorithm accepts as an input a adjacency matrix G , the number of vertices n and the number of edges m . The principle of generating the graph is based on making m random selections from a matrix containing all possible edges in the n vertex graph. The total number of edges in an undirected graph is $n(n - 1)/2$. The first step is to create the matrix with all possible edges of the graph, followed by the m random selections of edges.

```

GR-GEN1(G, n, m)
1      n_edges ← 0
2      for i ← 0 to n - 1 do
3          for j ← i + 1 to n - 1 do
4              aux[n_edges] ← (i, j)
5              n_edges ← n_edges + 1
6      while m > 0 do
7          index ← random(0..n_edges-1)
8          G[aux[index]] ← 1
9          m ← m - 1
10         swapaux ← aux[index]
11         aux[index] ← aux[n_edges]
12         aux[n_edges] ← swapaux
13         n_edges ← n_edges - 1

```

1.2.2 Predefined edge density random graphs

The second algorithm accepts as inputs the number of vertices n and the edge density d . For each possible edge in the graph a random number r is generated. If r is less or equal to the desired edge density d , then the edge is added to the graph.

```

GR-GEN2(G, n, d)
1      for i ← 0 to n - 1 do
2          for j ← i + 1 to n - 1 do
3              r ← random(0..1)
4              if r ≤ d then
5                  G[i][j] ← 1
6                  G[j][i] ← 1
7              else
8                  G[i][j] ← 0
9                  G[j][i] ← 0

```

1.2.3 Unit square random graphs

The third algorithm is based on the unit square for generating random graphs which have a higher expected edge density in the neighborhood of a vertex than the edge density of the entire graph.

GR-GEN3(G, n, d)

```
1      for i ← 0 to n - 1 do
2          aux[i] ← (random(0..1), random(0..1))
3      for i ← 0 to n - 1 do
4          for j ← i + 1 to n - 1 do
5              dist ← EuclideanDistance(aux[i], aux[j])
6              if dist ≤ d then
7                  G[i][j] ← 1
8                  G[j][i] ← 1
9              else
10                 G[i][j] ← 0
11                 G[j][i] ← 0
```

1.3 Java implementation of random graph generator algorithms

The graph generation algorithms are implemented as a single Java class whose constructors are overloaded to accommodate all three graph generator algorithms.

```
public G(int vertice_number, int edge_number) throws Exception
public G(int int_param1, double double_param2, String type) throws
Exception
```

All input parameters are verified and the constructors throw exceptions when the parameters are invalid or not within the expected bounds. The class has public methods which make possible operations like printing the graph, printing the adjacency matrix and accessing the adjacency matrix which is stored as a private 2-dimensional integer array.

```
int[][] get()
```

The graph can be printed in a vertex by vertex fashion with each vertex followed by the vertices to which it is connected to.

```
public void print()
```

The adjacency matrix can be printed/displayed in the reduced form or in the complete form. The last possibility is useful for debugging purposes and for the detection of loops in the generated graphs.

```
public void printMatrix(boolean half)
```

1.4 Examples

1.4.1 Generating a random graph with a fixed number of edges

The constructor for this type of random graph generator accepts as inputs two integers: the number of vertices and the number of edges. For generating and displaying a 10 vertices random graph with 30 edges the following code can be used:

```
try
{
    G gs = new G(10,30);
    gs.print();
    gs.printMatrix(true);
}
catch (Exception E)
{
    System.out.println("error initializing: "+E.toString());
}
```

The output of the program is the following:

```
Generating graph with 10 vertices and 30 edges
0: 1 4 7 9
1: 0 3 6 7 9
2: 4 5 6 7 8 9
3: 1 4 5 6 7 9
4: 0 2 3 5 7 8 9
5: 2 3 4 7 9
6: 1 2 3 7 8 9
7: 0 1 2 3 4 5 6 8 9
8: 2 4 6 7
9: 0 1 2 3 4 5 6 7
10
100100101
01001101
01111111
1111101
10111
0101
111
11
0
```

1.4.2 Generating a random graph with predefined edge density

The constructor for this type of random graph generator accepts as inputs an integer, a double and a string: the number of vertices the edge density and the string “density”. For generating and displaying a 10 vertices random graph with 0.5 edge density, the following code can be used:

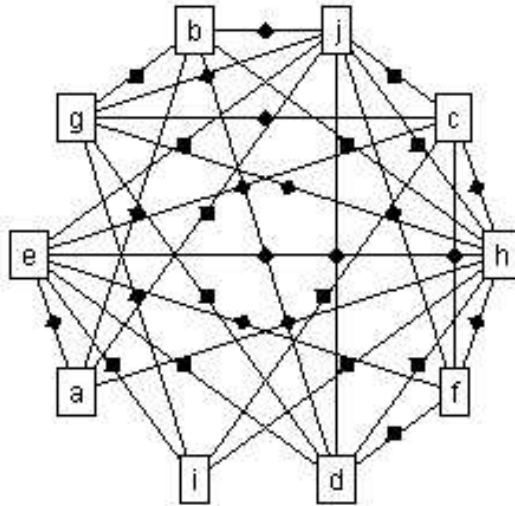


Figure 1.1: Random graph with 10 vertices and 30 edges

```
try
{
  G gs = new G(10,0.5,"density");
  gs.print();
  gs.printMatrix(true);
}
catch (Exception E)
{
  System.out.println("error initializing: "+E.toString());
}
```

The output of the program is the following:

```
Generating graph with 10 vertices and 0.5 edge density
0: 2 3 4 6 7 9
1: 2 3 4 5 7 8
2: 0 1 4 5 7 8
3: 0 1 4 7
4: 0 1 2 3 5 6 8
5: 1 2 4 8 9
6: 0 4
7: 0 1 3 9
8: 1 4 5 9
9: 0 5 7 8
10
011101101
111101110
0110000
100100
11010
0011
000
01
1
```

1.4.3 Generating a unit square random graph

The constructor for this type of random graph generator accepts as inputs an integer, a double and a string: the number of vertices, the maximum Euclidean distance between vertices and the string “distance”. For generating and displaying a 10 vertices random

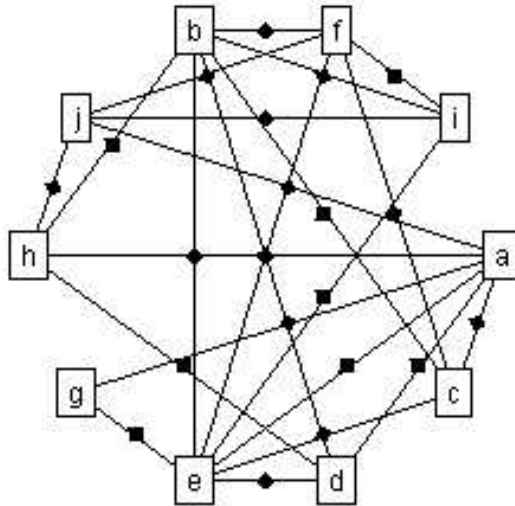


Figure 1.2: Random graph with 10 vertices and 0.5 edge density

graph with a maximum distance between connected vertices of 0.7, the following code can be used:

```

try
{
    G gs = new G(10,0.4,"distance");
    gs.print();
    gs.printMatrix(true);
}
catch (Exception E)
{
    System.out.println("error initializing: "+E.toString());
}

```

The output of the program is the following:

```

Generating graph with 10 vertices and 0.4 max distance between vertices
0: 1 6 8
1: 0 3 6 8
2: 4
3: 1 4 5 8 9
4: 2 3 5
5: 3 4 7
6: 0 1 8
7: 5
8: 0 1 3 6
9: 3
10
100001010
01001010
0100000
110011
10000
0100
010
00
0

```

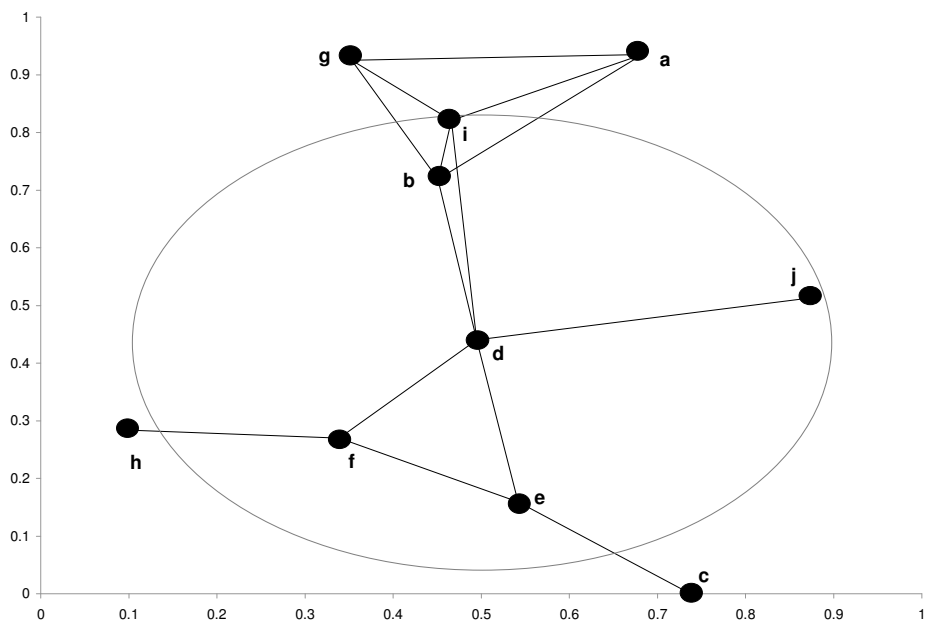


Figure 1.3: Unit square random graph with 10 vertices and 0.4 maximum Euclidean distance; the circle of radius 0.4 centered in vertex “d” shows which vertices get connected with vertex “d”

Chapter 2

Exact Vertex Cover Algorithm

2.1 Introduction

Given a graph $G = (V, E)$, a *vertex cover* of G is a set of vertices $V' \subseteq V$ such that each edge in E is adjacent to at least one vertex in V' . The set $V - V'$ is an empty subgraph of G and it is called an *independent set*. Therefore, finding whether a graph with n vertices has a vertex cover of size k can be answered by searching for an independent set of size $n - k$ in the same graph. These problems are known to belong to the class of NP complete problems.

The chosen data structure for graph representation is the adjacency matrix and therefore testing whether a particular subgraph forms of an independent set is a trivial task which can be achieved by counting the total number of edges of the subgraph. Therefore the algorithm approach is based on searching for independent sets rather than explicitly working with vertex cover sets.

As an interesting remark, we should note that an independent set of size k in a graph G is a clique of size k in the complement graph of G . A *clique* of a graph G is a complete subgraph of G .

An *induced subgraph* $G' = (V', E')$ is a subset of the vertices of a graph $G = (V, E)$ together with any edges whose endpoints are both in this subset: $V' \subseteq V, E' \subseteq E$. The *closed neighborhood* of a vertex v denoted by $N[v]$, is the set of vertices adjacent to v including itself.

2.2 The algorithm

NP complete problems, in order to be solved exactly, must make use of exponential time algorithms. A brute force approach to search for a solution iterates through all the possible states in the search space and therefore can be used only for very small problems, due to the exponential explosion of search possibilities. However, many problems are inherently constrained by conditions in their definitions, and in these cases the solution search space can be pruned based on the judicious use of these constraints. In addition, heuristics may be also added to the search algorithms, improving their search times.

Backtracking algorithms are able to search for solutions of a given problem methodically, by traversing a solution tree in a deep-first-search manner. Starting from the root, a backtracking algorithm will expand search paths and then backtrack to previous states when a particular search proves not to be leading to a solution. The backtracking algorithm expanding can be bounded by the problem constraints: a particular search path should not be expanded further when the bounding condition indicates that a solution is impossible to be found. In this case the algorithm backtracks and tries the next search path.

For certain problems is also possible to implement the backtracking as a divide-and-conquer strategy by splitting the initial problems into similar sub-problems of smaller size. This pertains to the use of a recursive algorithm which also assures the backtracking logic for the solution search strategy. The level of recursion can be controlled by a bounding condition contributing to the pruning of the problem space.

```

VC_BACKTRACK( $n$ , ind_set,  $G$ , result)
1   for  $i \leftarrow 0$  to  $n - 1$  do
2       n_vertices  $\leftarrow 0$ 
3       n_edges  $\leftarrow 0$ 
4       for  $j \leftarrow 0$  to  $n - 1$  do
5           if  $j \neq i$  and  $G[j][i] = 0$  then
6               m  $\leftarrow$  n_vertices + 1
7               for  $k \leftarrow j + 1$  to  $n - 1$  do
8                   if  $k \neq i$  and  $G[k][i] = 0$  then
9                        $G'[n\_vertices][m] \leftarrow G[j][k]$ 
10                       $G'[m][n\_vertices] \leftarrow G[k][j]$ 
11                      if  $G'[n\_vertices][m] = 1$  then
12                          n_edges  $\leftarrow$  n_edges + 1
13                          m  $\leftarrow$  m + 1
14                      n_vertices  $\leftarrow$  n_vertices + 1
15          if n_edges = 0 and n_vertices  $\geq$  ind_set - 1 then
16              result  $\leftarrow$  true
17              break
18          if n_vertices > ind_set - 1 then
19              VC_BACKTRACK(n_vertices, ind_set - 1,  $G'$ , sub_result)
20              if sub_result = true then
21                  result  $\leftarrow$  true
22                  break
23      result  $\leftarrow$  false

```

The algorithm for finding whether a graph has a vertex cover of size k consists of a main function which takes as parameters the an adjacency matrix of a graph G , the number n of vertices in the graph and the cardinality $n - k$ of the complementary independent set. The function returns a boolean result.

The algorithm contains a main loop which is executed n times, for each vertex v in the graph. The nested inner loops (lines 4-14) compute, for each vertex v , the subgraph G' induced on G by the subset $G - N[v]$, where $N[v]$ is the closed neighborhood of v . If G' has no edges and its number of vertices is greater or equal to $n - k - 1$, then the set of vertices

in G' and vertex v form an independent set of size $n - k$ in G , causing the search to return true (lines 15-17). If G' is not an empty subgraph and its number of vertices is strictly greater than $n - k - 1$, then the algorithm continues with the search for an independent set of cardinality $n - k - 1$ in G' (line 18). This is accomplished by the recursive call to VC_BACKTRACK in line 19 which expands the search tree and reduces the problem size by one. If a recursive call turns out to be successful, then the algorithm returns true (lines 20-21).

2.3 Testing the algorithm

The exact vertex cover algorithm can be tested on different classes of random graphs. However, it is most useful that the graphs chosen have minimum vertex cover cardinalities which are known in advance.

2.3.1 Graph with 8 vertices and vertex cover of size 2

The graph in figure 2.1 has a minimum vertex cover size of 2. In order to verify this, the vertex cover algorithm searches for an independent set of size 6 in the same graph. In the first step, G' is computed for the vertex “a”.

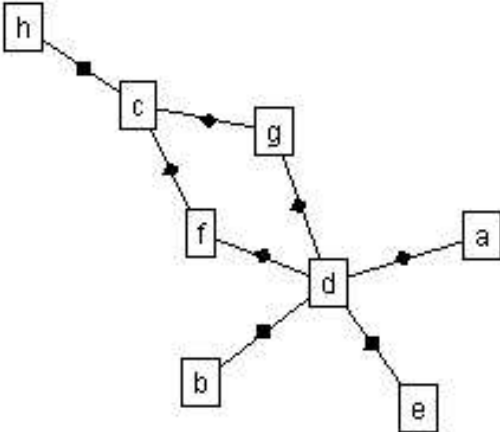


Figure 2.1: Graph with 8 vertices and minimum vertex cover of size 2

In figure 2.2 the sought independent set size is one unit smaller, therefore equal to 5. Since a 6 vertex graph may contain an independent set of size 5, the search is continued.

For the vertex “c”, the computed G' consists only of one vertex and therefore it cannot

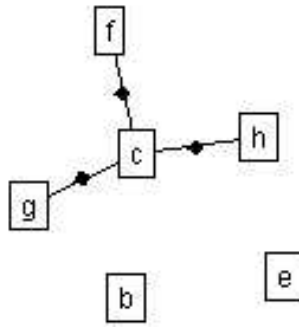


Figure 2.2: G' computed for vertex “a”

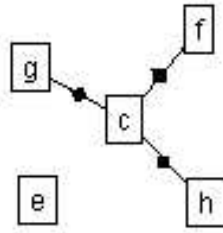


Figure 2.3: G' computed for vertex “b”

contain the sought independent set of size 4. The search is continued with the graph in figure 2.3 with vertex “e”, for an independent set of size 4 and produces the graph in figure 2.4.

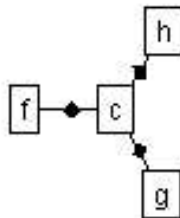


Figure 2.4: G' computed for vertex “e”

The next selected node is again “c” which yields a G' with no vertices. Therefore the algorithm continues with vertex “f” in the graph in figure 2.4. The resulting G' contains indeed an independent set of size 2 (vertices “g” and “h”) which causes the algorithm to stop and return true after 6 G' operations induced by examining 6 vertices in G (“a”, “b”, “c”, “e”, “c”, “f”). For this particular test, because of the small graph structure, the algorithm did not return unsuccessfully (i.e. false) from the recursive call and therefore did not perform any backtracking.

2.3.2 Peterson graph

Peterson graph 2.5 does not have a vertex cover of size 5 but has one of size 6. The vertex cover algorithm answers correctly with “NO” and “YES” after 70 and 4 G' operations, respectively.

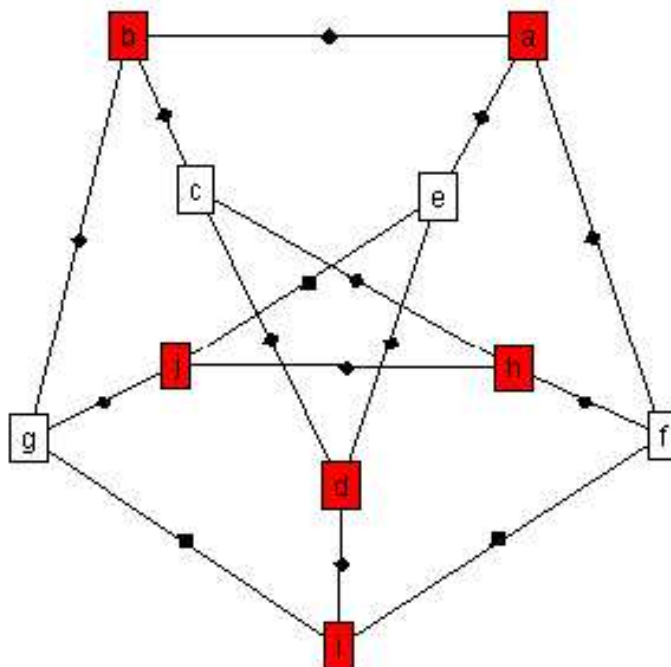


Figure 2.5: Peterson graph

2.3.3 Random graph with minimum vertex cover of size 6

The graph in figure 2.6 also does not have a vertex cover of size 5 but has one of size 6. It has been generated with a special purpose graph generator capable of producing a random graph which is assured to contain a vertex cover of a user specified size. The vertex cover algorithm answers correctly with “NO” and “YES” after performing 135,909 and 7 G' operations, respectively.

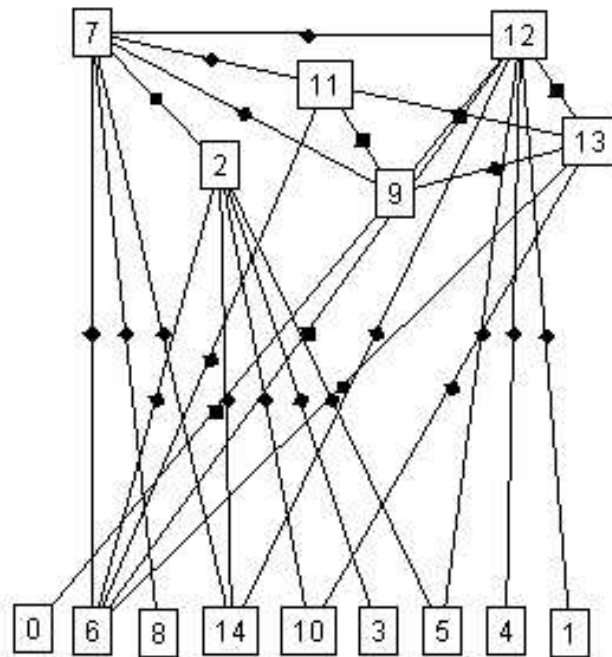


Figure 2.6: Random graph with a minimum vertex cover of size 6

2.3.4 Random graphs with known vertex cover sizes

For the further testing of the algorithm, a number of 20 vertices random graphs with variable vertex cover sizes and edge densities were generated. For each vertex cover size value from 1 to 19 and edge density value from 0.1 to 1.0 in 0.1 increments, 10 different random graphs were generated, accounting for a number of 1,900 different graphs. It must be noted that the edge density in the graphs generated using a generator which assures that the resulting graph has a vertex cover of size k is slightly different than an edge density in a complete random graph.

In order to avoid random variations of the results, the algorithm performance, given in the number of G' graph computations, was averaged over a number of 10 graphs. After inspecting the numerical results, the wide range of variations of the algorithm performance indicated that a logarithmic scale representation of the results is more suitable.

In figure 2.7 we can observe that the algorithm performs poorly on some particular 20 vertices graphs which are characterized by a low edge density (between 0.1 and 0.6) and a vertex cover size between 5 and 14. For these random graphs the number of G' operations is over 200 with a maximal value in the order of 10^6 for very sparse graphs with an edge density of 0.1 and a vertex cover of size equal to 7 (table 2.1).

Indeed, after testing the algorithm with other very sparse random graphs with predefined

Dens\VC	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0.1	0	1	1	1	2	5	5	5	3	3	2	1	1	1	1	1	1	1	1
0.2	0	0	1	1	1	3	3	3	3	4	3	3	1	1	1	1	1	1	1
0.3	0	0	1	1	1	1	4	3	3	2	3	2	1	1	1	1	1	1	1
0.4	0	1	1	1	1	1	1	2	3	2	2	2	2	2	1	1	1	1	1
0.5	0	0	1	1	1	1	1	1	2	2	3	2	2	1	1	1	1	0	0
0.6	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
0.7	0	0	1	1	1	1	1	1	1	1	1	1	2	1	1	1	0	0	0
0.8	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0.9	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
1.0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0

Table 2.1: The algorithm performance on random graphs with variable edge densities and vertex cover sizes, given in $\log(G'$ computations)

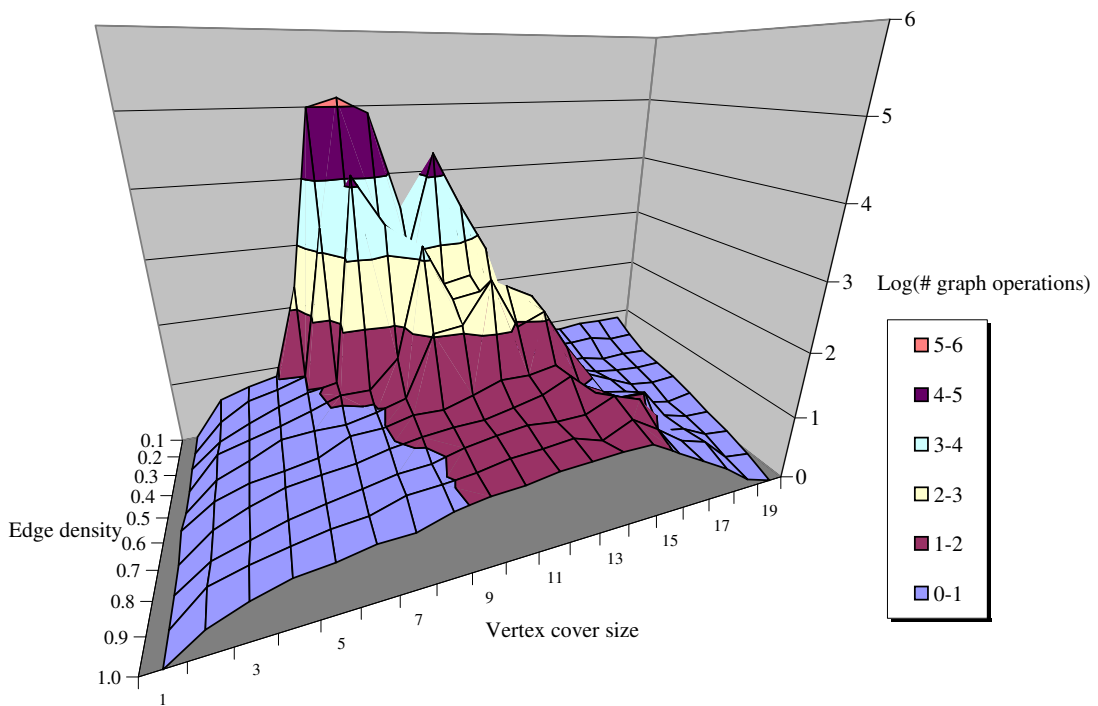


Figure 2.7: Algorithm performance on random graphs with variable edge densities and vertex cover sizes

vertex cover sizes equal to 7, in many instances the algorithm performed poorly, proving that these classes of graphs are characterized by a high probability of difficulty for this particular algorithm. As an example, for the graph in figure 2.8 the algorithm needed 211,245 G' operations in order to find a cover set of size 7.

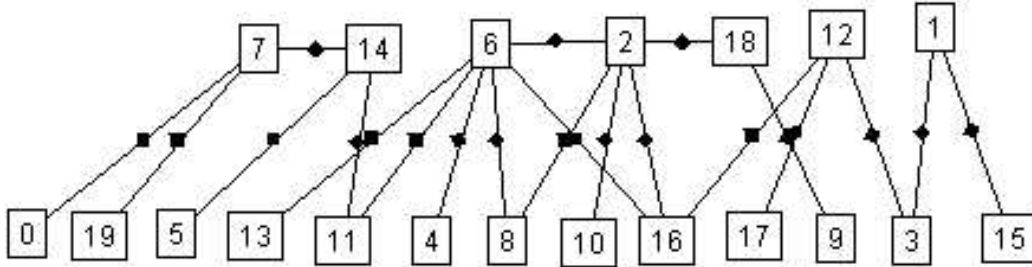


Figure 2.8: Graph with 0.1 edge density and a vertex cover of size 7

2.4 Improved vertex cover algorithm

The main strategy for improving the algorithm is the heuristic which states that vertices with smallest degrees are most likely to belong to an maximum independent set of a given graph. Therefore, by sorting the vertices on their degrees and beginning the search with vertices of smallest degree, the search could yield a solution faster, when looking for a vertex cover which exists in the given graph. The only problem that remains to be discussed is which algorithm to choose from the multitude of known sorting algorithms.

2.4.1 Bucket-sort algorithm

The bucket-sort algorithm, called sometimes counting-sort when dealing with integers, is a very fast sorting algorithm which takes advantages of some special properties of the data that need to be sorted and which mostly consists of integers with values in a small range. The vertices degrees in a simple graph are a very good example of integers with values between 0 and the total number of vertices in the graph.

Our bucket sort algorithm version maintains a list of indexes (the vertices labels) and a count of occurrence of each integer (the vertex degree) in the data to be sorted, operation which takes a single scan through the data array. Initially the vertices degrees are computed in $O(n^2)$ time (lines 1-5). For a simple graph with n vertices, the buckets' sizes must be equal to n in order to account for the extreme cases where all data items have the same value, and

the bucket-assigning operation takes $O(n)$ time (lines 6-8). In the original sorting algorithm the data array would have to be rearranged, operation which would also take $O(n)$ time for an array of size n .

BUCKET-SORT-BY-DEGREE(n, G)

```

1      for  $i \leftarrow 0$  to  $n - 1$  do
2          bucket-size[ $i$ ]  $\leftarrow 0$ 
3          degree[ $i$ ]  $\leftarrow 0$ 
4      for  $j \leftarrow 0$  to  $n - 1$  do
5          degree[ $i$ ]  $\leftarrow$  degree[ $i$ ] +  $G[i][j]$ 
6      for  $i \leftarrow 0$  to  $n - 1$  do
7          bucket[degree[ $i$ ]][bucket-size[degree[ $i$ ]]]  $\leftarrow i$ 
8          bucket-size[ $d$ ]  $\leftarrow$  bucket-size[ $d$ ] + 1

```

GET-VERTEX($idx, result$)

```

1       $j \leftarrow 0$ 
2      count  $\leftarrow 0$ 
3      do
4          count  $\leftarrow$  count + bucket-size[ $j$ ]
5           $j \leftarrow j + 1$ 
6      while count < ( $i + 1$ )
7      result  $\leftarrow$  bucket[ $j-1$ ][count -  $i - 1$ ]

```

However, because the vertex cover algorithm is a backtracking procedure, it is more useful to apply the first part of the bucket-sort algorithm followed by the second part (GET-VERTEX) only when needed. In this way we are able to avoid the rearranging of all data items when, potentially, only a subset of the data array is needed by the search which does not continue beyond a particular level.

The improved vertex cover algorithm will therefore call the first part of the bucket-sort procedure before its main loop and the second part of the bucket-sort inside the main loop. In this way, the exploration of the graph vertices will be performed in the ascending order of their degrees, improving the chances of finding a solution faster. Indeed, after comparing

the improved algorithm with the initial one, the number of G' operations was not influenced so drastically by low edge densities. However, the improved algorithm, although faster in these circumstances, proved to perform still poorly on some extreme vertex cover problem instances where the graphs consist of vertices which are disconnected and the sought vertex cover size is small. This behavior is probably due to the algorithm approach which tries to find a vertex cover of size k by searching for independent sets of size $n - k$ which are very abundant in extremely disconnected graphs.

```

VC_BACKTRACK(n, ind_set, G, result)
1      BUCKET-SORT-BY-DEGREE(n, G)
2      for idx  $\leftarrow$  0 to  $n - 1$  do
3          GET-VERTEX(idx, i)
4          n_vertices  $\leftarrow$  0
5          n_edges  $\leftarrow$  0
6          for j  $\leftarrow$  0 to  $n - 1$  do
7              if  $j \neq i$  and  $G[j][i] = 0$  then
8                  m  $\leftarrow$  n_vertices + 1
9                  for k  $\leftarrow$  j + 1 to  $n - 1$  do
10                     if  $k \neq i$  and  $G[k][i] = 0$  then
11                          $G'[n\_vertices][m] \leftarrow G[j][k]$ 
12                          $G'[m][n\_vertices] \leftarrow G[k][j]$ 
13                         if  $G'[n\_vertices][m] = 1$  then
14                             n_edges  $\leftarrow$  n_edges + 1
15                             m  $\leftarrow$  m + 1
16                             n_vertices  $\leftarrow$  n_vertices + 1
17                     if n_edges = 0 and n_vertices  $\geq$  ind_set - 1 then
18                         result  $\leftarrow$  true
19                         break
20                     if n_vertices > ind_set - 1 then
21                         VC_BACKTRACK(n_vertices, ind_set - 1, G', sub_result)
22                     if sub_result = true then
23                         result  $\leftarrow$  true
24                     break
25      result  $\leftarrow$  false

```

2.4.2 Testing the improved algorithm

For the graph with 8 vertices and vertex cover of size 2 in Figure 2.1, the improved algorithm performs only 2 G' operations compared to 6 of the initial algorithm. For the Peterson graph in figure 2.5 the new algorithm give correct answers for a vertex cover of size 5 and 6, after 70 and 2 G' operations. Obviously, the algorithm does not improve speed when looking for the non existent vertex cover of size 5. For the random graph with known minimum vertex cover of size 6 in figure 2.6, the improved algorithm answers after 135,909 and 9 G' operations, confirming that it does not improve speed when searching for non existent vertex covers, and showing that it is possible to even be outperformed by the initial algorithm. For random graphs with edge density of 0.1 and predefined vertex covers of sizes equal to 7, the improved algorithm proved to be more efficient by answering correctly after an average of 13 G' operations.

2.5 Other possible improvement directions

As it has been shown in section 2.3.1, the algorithm yields G' subgraphs containing disconnected vertices. These disconnected vertices are undoubtedly going to be included in the current independent set solution, but this will happen in subsequent algorithm steps. One possible improvement can be to remove any disconnected vertices from G' in one step and lower the current value of the sought independent set size accordingly. Therefore the algorithm will “jump” through a number of steps, breaking the problem into subproblems of sizes which are smaller than $n - k - 1$.

As shown in previous section, the improved algorithm still performs poorly on extremely disconnected graphs. For these cases, which are characterized by an extremely low edge density, a different algorithm should be used, decision which should be based on a previous assessment of the graph edge density.

Appendices

1 Graph generator

```
import java.lang.*;
public class G {
    private int NMAX=300;
    private static int gm[][];
    private int n;
    public int getn()
    {
        return n;
    }
    //cloning seems to be a weak point
    private static int[][] clone2D() throws CloneNotSupportedException {
        int[][] result = (int[][]) gm.clone();
        for (int i = 0; i < gm.length; ++i)
        {
            if (gm[i] != null) result[i] = (int[]) gm[i].clone();
        }
        return result;
    }
    //verification for input parameters
    private void verifyBounds(int n, double p, int e, double d) throws Exception
    {
        if ((n<2)|(n>NMAX)) throw new Exception("The number of vertices is minimum 2 and maximum 20");
        if ((p<0)|(p>1)) throw new Exception("The edge density is a real number between 0 and 1");
        if ((e<0)|(e>(n*n-n)/2)) throw new Exception("The number of vertices is minimum 0 and maximum (n*n-n)/2");
        if ((d<0)|(d>Math.sqrt(2))) throw new Exception("The distance is a real number between 0 and sqrt(2)");
    }
    //constructor for int, int type parameters
    public G(int n_vertices, int edge_number) throws Exception
    {
        //    System.out.println("Generating graph with "+n_vertices+" vertices and "+edge_number+" edges");
        verifyBounds(n_vertices,0,edge_number,0);
        this.n=n_vertices;
        gm=new int[n][n];
        int [][] aux=new int [(n*n-n)/2][2];
        int i, j;
        int max=0;
        for (i=0; i<n; i++)
        {
            for (j=i+1; j<n; j++)
            {
                aux[max][0]=i;
                aux[max][1]=j;
                max++;
            }
        }
        //    max=(int)(n*n-n)/2;
        int index;
        int swapaux;
        do
        {
            index=(int)(Math.random()*max);
            gm[aux[index][0]][aux[index][1]]=1;
            gm[aux[index][1]][aux[index][0]]=1;
            max--;
        }
        //swap positions
        swapaux=aux[index][0];
        aux[index][0]=aux[max][0];
        aux[max][0]=swapaux;
        swapaux=aux[index][1];
        aux[index][1]=aux[max][1];
        aux[max][1]=swapaux;
        edge_number--;
    } while (edge_number>0);
}
private void densityGen(int n_vertices, double edge_density)
{
    this.n=n_vertices;
    gm=new int[n][n];
    //    System.out.println("Generating graph with "+n_vertices+" vertices and "+edge_density+" edge density");
```

```

int i, j;
double r;
for (i=0; i<n; i++)
{
    for (j=i+1; j<n; j++)
    {
        r=Math.random();
        System.out.println(r);
        if (r<=edge_density)
        {
            gm[i][j]=1;
            gm[j][i]=1;
        }
        else
        {
            gm[i][j]=0;
            gm[j][i]=0;
        }
    }
}
}
private void distanceGen(int n_vertices, double distance)
{
//    System.out.println("Generating graph with "+n_vertices+" vertices and "+distance+" max distance between v
this.n=n_vertices;
gm=new int[n][n];
double [][] aux=new double [n][2];
int i, j;
for (i=0; i<n; i++)
{
    aux[i][0]=Math.random();
    aux[i][1]=Math.random();
//    System.out.println("vertex "+i+" location in the unit square: "+aux[i][0]+","+aux[i][1]);
}
for (i=0; i<n; i++)
{
    for (j=i+1; j<n; j++)
    {
        double dist=Math.sqrt(((aux[i][0]-aux[j][0])*(aux[i][0]-aux[j][0]))+
            ((aux[i][1]-aux[j][1])*(aux[i][1]-aux[j][1])));
        if (dist<=distance)
        {
            gm[i][j]=1;
            gm[j][i]=1;
        }
        else
        {
            gm[i][j]=0;
            gm[j][i]=0;
        }
    }
}
}
//constructor for vertex cover int,double,int type params
public G(int n_vertices, double edge_density, int cover_size) throws Exception
{
    if ((cover_size<1)|(cover_size>n_vertices))
        throw new Exception("The cover size is minimum 1 and maximum "+n_vertices);
    this.n=n_vertices;
    gm=new int[n][n];
    int[] cover=new int[cover_size];
    int[] covered=new int[n-cover_size];
//    System.out.println("Generating graph with "+n_vertices+" vertices, "+
        edge_density+" edge density and "+cover_size+" vertex cover size");
    int i, j;
    double r;
    int [] aux=new int [n];
//choosing randomly cover_size distinct vertices in the cover set
//first we create an auxiliary array with all vertex indices in it
    for (i=0; i<n; i++) aux[i]=i;
    int index;
    int swapaux;
    int max=n;
    int vertices_needed=cover_size;
    do
    {
        index=(int)(Math.random()*max);
        max--;
        cover[cover_size-vertices_needed]=aux[index];
//        System.out.println("vertex "+aux[index]+" is in the cover set");
//        System.out.print(aux[index]+" ");
    }
}

```

```

//swap positions
    swapaux=aux[index];
    aux[index]=aux[max];
    aux[max]=swapaux;
    vertices_needed--;
} while (vertices_needed>0);
//creating the covered vertices set
for (i=0; i<n-cover_size; i++)
{
    covered[i]=aux[i];
//    System.out.println("vertex "+aux[i]+" is in the coverED set");
//    System.out.print(aux[i]+" ");
}
//condition for cover set: every vertex must have at least an edge with a vertex in the cover except itself
if (cover_size>1)
{
    for (i=0; i<cover_size; i++)
    {
        for (j=i+1; j<cover_size; j++)
        {
            r=Math.random();
//            System.out.println(r);
            if ((r<=edge_density)&(cover[i]!=cover[j]))
            {
                gm[cover[i]][cover[j]]=1;
                gm[cover[j]][cover[i]]=1;
            }
            else
            {
//we do not want to remove other edges between vertices in the cover set - we're on the CONSTRUCTIVE side
//this impacts the final graph density but there's no easy solution
//                gm[cover[i]][cover[j]]=0;
//                gm[cover[j]][cover[i]]=0;
            }
        }
    }
}
//condition for covered set: every vertex must have at least an edge with a vertex in the cover set
for (i=0; i<n-cover_size; i++)
{
    index=(int)(Math.random()*cover_size);
    gm[covered[i]][cover[index]]=1;
    gm[cover[index]][covered[i]]=1;
}
//adding other random edges between vertices in the covered set and vertices in the cover set
for (i=0; i<cover_size; i++)
{
    for (j=i+1; j<n; j++)
    {
        r=Math.random();
//        System.out.println(r);
        if ((r<=edge_density)&(cover[i]!=j))
        {
            gm[cover[i]][j]=1;
            gm[j][cover[i]]=1;
        }
    }
}
//constructor for int,double type params - the graph type must be specified
public G(int int_param1, double double_param2, String type) throws Exception
{
    if (type.equalsIgnoreCase("density"))
    {
        verifyBounds(int_param1,double_param2,0,0);
        densityGen(int_param1,double_param2);
    }
    else if (type.equalsIgnoreCase("distance"))
    {
        verifyBounds(int_param1,0,0,double_param2);
        distanceGen(int_param1,double_param2);
    }
    else throw new Exception("Unknown graph type");
}
//constructor for int,double type params - the graph type must be specified
public G(String name) throws Exception
{
    int i,j;
    String [] adj_mat=new String[NMAX];
}

```

```

if (name.equalsIgnoreCase("peterson"))
{
    this.n=10;
//    String [] adj_mat=new String[n-1];
    adj_mat[0]="100110000";
    adj_mat[1]="10001000";
    adj_mat[2]="1000100";
    adj_mat[3]="100010";
    adj_mat[4]="00001";
    adj_mat[5]="0110";
    adj_mat[6]="011";
    adj_mat[7]="01";
    adj_mat[8]="0";
}
else if (name.equalsIgnoreCase("testgraph"))
{
    this.n=6;
//    String [] adj_mat=new String[n-1];
    adj_mat[0]="11100";
    adj_mat[1]="1010";
    adj_mat[2]="001";
    adj_mat[3]="00";
    adj_mat[4]="0";
}
else if (name.equalsIgnoreCase("vc2graph"))
{
    this.n=8;
    adj_mat[0]="0010000";
    adj_mat[1]="010000";
    adj_mat[2]="00111";
    adj_mat[3]="1110";
    adj_mat[4]="000";
    adj_mat[5]="00";
    adj_mat[6]="0";
}
else if (name.equalsIgnoreCase("vc6graph"))
{
    this.n=15;
    adj_mat[0]="00000000100000";
    adj_mat[1]="0000000000100";
    adj_mat[2]="101110010001";
    adj_mat[3]="00000000000";
    adj_mat[4]="0000000100";
    adj_mat[5]="000000100";
    adj_mat[6]="10001110";
    adj_mat[7]="1101101";
    adj_mat[8]="000000";
    adj_mat[9]="01110";
    adj_mat[10]="0010";
    adj_mat[11]="010";
    adj_mat[12]="11";
    adj_mat[13]="0";
}
else throw new Exception("Unknown graph type");
gm=new int[n][n];
for (i=0; i<n; i++)
{
    gm[i][i]=0;
    gm[i][i]=0;
    for (j=i+1; j<n; j++)
    {
        if (adj_mat[i].charAt(j-i-1)=='1')
        {
            gm[i][j]=1;
            gm[j][i]=1;
        }
        else
        {
            gm[i][j]=0;
            gm[j][i]=0;
        }
    }
}
}
//printing the graph by vertex index followed by vertices list which it is connected to
public void printJoin()
{
    int i, j;

```

```

for (i=0; i<n; i++)
{
    System.out.print(i+": ");
    for (j=0; j<n; j++)
    {
        if (gm[i][j]==1) System.out.print(j+" ");
    }
    System.out.println();
}
System.out.println();
}
public void printDisjoin()
{
    int i, j;
    for (i=0; i<n; i++)
    {
        System.out.print(i+": ");
        for (j=0; j<n; j++)
        {
            if (gm[i][j]==0) System.out.print(j+" ");
        }
        System.out.println();
    }
    System.out.println();
}
}
//printing the adjacency matrix, full or half of it
//full is good for debugging to see if there are loops
public void printMatrix(boolean half)
{
    int i, j;
    System.out.println(n);
    if (half)
    {
//printing only half of the matrix
        for (i=0; i<n; i++)
        {
            for (j=0; j<i; j++) System.out.print(" ");
            for (j=i+1; j<n; j++)
            {
                System.out.print(gm[i][j]);
            }
            System.out.println();
        }
    }
    else
    {
        for (i=0; i<n; i++)
        {
            for (j=0; j<n-1; j++)
            {
                System.out.print(gm[i][j]+" ");
            }
            System.out.println(gm[i][n-1]);
        }
//
    }
}
}
int[][] get()
{
    return gm;
}
public void complement()
{
    int i,j;
    for (i=0; i<n; i++)
    {
        for (j=i+1; j<n; j++)
        {
            if (gm[i][j]==0)
            {
                gm[i][j]=1;
                gm[j][i]=1;
            }
            else
            {
                gm[i][j]=0;
                gm[j][i]=0;
            }
        }
    }
}
}

```

```
}  
public int degree(int v)  
{  
    int j,d;  
    d=0;  
    for (j=0; j<n; j++)  
    {  
        if (gm[v][j]==1) d++;  
    }  
    return d;  
}  
}
```

2 Vertex cover exact algorithm

```
import java.lang.*;
/**
 * <p>Title: Vertex cover exact algorithm</p>
 * <p>Description: uses backtracking</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * @author Stefan Pantazi
 */
public class VCspantazi {
    private boolean verbose=false;
    private int[] [] g;
    private int n;
    public int trials=0;
    public VCspantazi()
    {
        // System.out.println("minimum vertex cover class initialized...");
    }
    //int[] []
    public void setup(int[] [] gr,int n_vertices)
    {
        this.g=(int[] [])gr.clone();
        this.n=n_vertices;
    }
    /*
    private boolean slow_backtrack(int n,int ind_set,int [] [] g_copy, boolean verbose)
    {
        int i,j,k,m;
        int [] [] g_prime=new int [n-1][n-1];
        if (verbose)
        {
            //displaying the graph copy
            System.out.println("Trying the following "+n+" vertices graph");
            for (j=0; j<n; j++)
            {
                for (k=0; k<n-1; k++) System.out.print(g_copy[j][k]+" ");
                System.out.println(g_copy[j][n-1]);
            }
            int n_vertices,n_edges;
            for (i=0; i<n; i++)
            {
                trials++;
            //deriving G' graph from G
            if (verbose) System.out.println("vertices: "+n+", wanted ind_set: "+ind_set+
                ", assume a max independent set contains vertex "+i);
            n_vertices=0;
            n_edges=0;
            for (j=0; j<n; j++)
            {
                if ((j!=i)&&(g_copy[j][i]==0))
                {
                    m=n_vertices+1;
                    for (k=j+1; k<n; k++)
                    {
                        if ((k!=i)&&(g_copy[k][i]==0))
                        {
                            g_prime[n_vertices][m]=g_copy[j][k];
                            g_prime[m][n_vertices]=g_copy[k][j];
                            if (g_prime[n_vertices][m]==1) n_edges++;
                            m++;
                        }
                    }
                    n_vertices++;
                }
            }
            if (verbose)
            {
            //print gprime
            for (j=0; j<n_vertices; j++)
            {
                for (k=0; k<n_vertices-1; k++) System.out.print(g_prime[j][k]+" ");
                System.out.println(g_prime[j][n_vertices-1]);
            }
            System.out.println("Density: "+n_edges+"/"+n_vertices*(n_vertices-1)/2);
            }
            //if indeed an independent set then n_edges should stay 0
            if ((n_edges==0)&&(n_vertices>=ind_set-1)) return true;
            if (n_vertices>ind_set-1)
            {
            //look deeper in the tree
```

```

        if (verbose) System.out.println("try subgraph with: "+n_vertices+
            " vertices, for a ind_set of: "+(ind_set-1));
        int [][] sub_g_prime=(int [][])g_prime.clone();
        if (slow_backtrack(n_vertices,ind_set-1,sub_g_prime,verbose)) return true;
        if (verbose) System.out.println("backtracking to graph with: "+n+
            " vertices, ind_set value: "+ind_set);
    }
    else if (verbose) System.out.println("no REASON to try to find a "+ind_set+
        " max ind set in a graph with "+n_vertices+" vertices");
    }
    return false;
}
public boolean slow_exactVC(G g, int kvc, boolean verbose)
{
    trials=0;
    if (kvc==g.getn()) return true;
    else return slow_backtrack(g.getn(),g.getn()-kvc,(int [][])g.get().clone(),verbose);
}
*/
////cliques algorithm
public boolean exactClique(G g, int kClique, boolean verbose)
{
    trials=0;
    g.complement();
    if (kClique==0) return true;
    else return fastBacktrack(g.getn(),kClique,(int [][])g.get().clone());
}
//////////end testing for cliques
///faster faster faster
private boolean fastBacktrack(int n,int ind_set,int [][] g_copy)
{
    int i,j,k,m;
    int [][] g_prime=new int [n-1][n-1];
    if (verbose)
    {
//displaying the graph copy
        System.out.println("Trying the following "+n+" vertices graph");
        for (j=0; j<n; j++)
        {
            for (k=0; k<n-1; k++) System.out.print(g_copy[j][k]+" ");
            System.out.println(g_copy[j][n-1]);
        }
    }
    int n_vertices,n_edges;
    int idx;
    BucketSort bs = new BucketSort(n,g_copy);
    for (idx=0; idx<n; idx++)
    {
        trials++;
        i=bs.get(idx);
//deriving G' graph from G
        if (verbose) System.out.println("vertices: "+n+" , wanted ind_set: "+ind_set+
            " , assume a max independent set contains vertex "+i);
        n_vertices=0;
        n_edges=0;
        for (j=0; j<n; j++)
        {
            if ((j!=i)&&(g_copy[j][i]==0))
            {
                m=n_vertices+1;
                for (k=j+1; k<n; k++)
                {
                    if ((k!=i)&&(g_copy[k][i]==0))
                    {
                        g_prime[n_vertices][m]=g_copy[j][k];
                        g_prime[m][n_vertices]=g_copy[k][j];
                        if (g_prime[n_vertices][m]==1) n_edges++;
                        m++;
                    }
                }
                n_vertices++;
            }
        }
        if (verbose)
        {
//print gprime
            for (j=0; j<n_vertices; j++)
            {
                for (k=0; k<n_vertices-1; k++) System.out.print(g_prime[j][k]+" ");
                System.out.println(g_prime[j][n_vertices-1]);
            }
        }
    }
}

```

```

    }
    System.out.println("Density: "+n_edges+"/"+n_vertices*(n_vertices-1)/2);
}
//if indeed an independent set then n_edges should stay 0
if ((n_edges==0)&&(n_vertices>=ind_set-1)) return true;
if (n_vertices>ind_set-1)
{
//look deeper in the tree
if (verbose) System.out.println("try subgraph with: "+n_vertices+
    " vertices, for a ind_set of: "+(ind_set-1));
int [][] sub_g_prime=(int [][])g_prime.clone();
if (fastBacktrack(n_vertices,ind_set-1,sub_g_prime)) return true;
if (verbose) System.out.println("backtracking to graph with: "+n+
    " vertices, ind_set value: "+ind_set);
}
else if (verbose) System.out.println("no REASON to try to find a "+ind_set+
    " max ind set in a graph with "+n_vertices+" vertices");
}
return false;
}
public boolean exact(int kvc)
{
    trials=0;
    if (kvc==n) return true;
    else return fastBacktrack(n,n-kvc,g);
}
//inner class bucket sort
class BucketSort {
    private int [][] bucket;
    private int [] bsize;
    private int n;
    public BucketSort(int n,int [][] gm)
    {
        bucket=new int[n][n];
        bsize=new int[n];
        this.n=n;
        int i,j,d;
        for (i=0; i<n; i++)
        {
            d=0;
            for (j=0; j<n; j++) if (gm[i][j]==1) d++;
            bucket[d][bsize[d]]=i;
            bsize[d]++;
        }
    }
    /* public void print()
    {
        int i,j;
        for (i=0; i<n; i++)
        {
            System.out.print(bsize[i]+" vertices of degree "+i+": ");
            for (j=0; j<bsize[i]-1; j++)
            {
                System.out.print(bucket[i][j]+",");
            }
            if (bsize[i]>0) System.out.println(bucket[i][bsize[i]-1]);
            else System.out.println();
        }
        for (i=0; i<n; i++)
        {
            System.out.println("get("+i+")="+get(i));
        }
    }
    */
    public int get(int i)
    {
        int j,count;
        j=0;
        count=0;
        do
        {
            count+=bsize[j];
            j++;
        } while (count<i+1);
        return bucket[j-1][(count-i-1)];
    }
}
}

```

3 Vertex cover algorithm testing program

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author unascribed
 * @version 1.0
 */
public class VCTestng {
    public static void researchTest1()
    {
        int i,j;
        int errors=0;
        System.out.println("20 vertices random graphs, v_cover, density x 10, average trials x 10");
        int v_cover;
        for (v_cover=1; v_cover<=20; v_cover++)
        {
            for (i=1; i<=10; i++)
            {
                int tot_trial_slow=0,tot_trial_fast=0;
//slow algorithm
                for (j=0; j<10; j++)
                {
                    try
                    {
                        G gs = new G(20,(double)(i/10),v_cover);
                        VCspantazi vc = new VCspantazi();
//                        if (!vc.slow_exactVC(gs,v_cover,false)) errors++;
//                        tot_trial_slow+=vc.trials;
                        vc.setup(gs.get(),gs.getn());
                        if (!vc.exact(v_cover)) errors++;
                        tot_trial_fast+=vc.trials;
                    }
                    catch (Exception E)
                    {
                        System.out.println("error initializing: "+E.toString());
                    }
                }
                System.out.print(v_cover+" "+i+" "+tot_trial_slow+" "+tot_trial_fast+"\n");
            }
        }
        System.out.println("errors: "+errors);
    }

    public static void main(String[] args)
    {
        try
        {
            System.out.println("creating the graph...");
            int i;
            int k=1;
//generating a fixed number of edges graph
//            G gs = new G(20,10);
//generating a edge density type graph
//            G gs = new G(50,1,"density");
//generating a vertex max distance type graph
//            G gs = new G(10,0.7,"distance");
//            G gs = new G(5,0.9,2);
            G gs = new G("peterson");
//            G gs = new G("testgraph");
//            G gs = new G("vc2graph");
//            G gs = new G("vc6graph");
//            gs.printJoin();
//            gs.printDisjoin();
//            gs.printHamming();
//            System.out.println("250 graphs with n=10, density=0.5, cover=4");
//            System.out.println("cover list=4 items, covered list=6 items, max, min, cover, covered");
//            G gs = new G(20,0.1,k);
//the complement is only for clique
//            gs.complement();
//            gs.printMatrix(true);
//            gs.printMatrix(false);
//            BucketSort bs = new BucketSort(gs.getn(),gs.get());
//            bs.print();
//            gs.printResearch();
            VCspantazi vc = new VCspantazi();
//            researchTest1();
        }
    }
}
```

```

        long startTime,endTime;
//fast fast fast
vc.setup(gs.get(),gs.getn());
startTime=System.currentTimeMillis();
if (vc.exact(k)) System.out.println("after "+vc.trials+
    " trials, it turns out (supposedly faster) that the graph HAS a vertex cover="+k);
else System.out.println("after "+vc.trials+
    " trials, it turns out (supposedly faster) that the graph does NOT HAVE a vertex cover="+k);
endTime=System.currentTimeMillis();
System.out.println("elapsed time "+(endTime-startTime)+" msec");
//slow slow slow
/*
    startTime=System.currentTimeMillis();
    if (vc.exactVC(gs,k,false)) System.out.println("after "+vc.trials+
        " trials, it turns out that the graph HAS a vertex cover="+k);
    else System.out.println("after "+vc.trials+
        " trials, it turns out that the graph does NOT HAVE a vertex cover="+k);
    endTime=System.currentTimeMillis();
    System.out.println("elapsed time "+(endTime-startTime)+" msec");*/
//
//    if (vc.exactClique(gs,k,false)) System.out.println("after "+vc.trials+
//        " trials, it turns out that the graph HAS a clique="+k);
//
//    else System.out.println("after "+vc.trials+
//        " trials, it turns out that the graph does NOT HAVE a clique="+k);
}
catch (Exception E)
{
    System.out.println("error initializing: "+E.toString());
}
}
}

```

4 Timing results

Chris'

- algorithm performs very well on unit square random graphs with existent vertex cover of size k
- algorithm performs poorly (time increases with number of vertices) on unit square random graphs with non-existent vertex cover of size k

Marc's

- algorithm performs well most of the times on unit square random graphs with existent vertex cover of size k
- algorithm performs increasingly slower (time increase with number of vertices) on unit square random graphs with non-existent vertex cover of size k, and, in these situations, spends more time on graphs with low number of edges

Tricia's

- algorithm performs well most of the times on unit square random graphs with existent vertex cover of size k
- algorithm performs best of all on unit square random graphs with non-existent vertex cover of size k, although in some of these situations spends increasingly more time (time increases with the value of k) until a peak is reached and then the time decreases as k increases

Stefan's

- the algorithm performs very well on unit square random graphs with existent vertex cover

of size k

- the algorithm performs poorly on unit square random graphs with non-existent vertex cover of size k which have a very low edge density; the time increases with the value of k until reaches a peak and then the time decreases as k increases

cover size	distance	Tricia t	Marc t	Chris t	Stefan t	Tricia r	Marc r	Chris r	Stefan r
1	0.3525	3	40	2	5	FALSE	FALSE	FALSE	FALSE
1	0.52875	2	1	3	50	FALSE	FALSE	FALSE	FALSE
1	0.705	4	1	3	0	FALSE	FALSE	FALSE	FALSE
1	0.88125	12	1	19	1	FALSE	FALSE	FALSE	FALSE
1	1.0575	44	7	0	5	FALSE	FALSE	FALSE	FALSE
2	0.3525	1	0	2	4	FALSE	FALSE	FALSE	FALSE
2	0.52875	1	0	2	1	FALSE	FALSE	FALSE	FALSE
2	0.705	0	0	7	1	FALSE	FALSE	FALSE	FALSE
2	0.88125	0	11	2	0	FALSE	FALSE	FALSE	FALSE
2	1.0575	0	5	2	0	FALSE	FALSE	FALSE	FALSE
3	0.3525	1	7	9	2	FALSE	FALSE	FALSE	FALSE
3	0.52875	1	7	10	1	FALSE	FALSE	FALSE	FALSE
3	0.705	1	5	11	0	FALSE	FALSE	FALSE	FALSE
3	0.88125	1	0	12	1	FALSE	FALSE	FALSE	FALSE
3	1.0575	0	1	13	0	FALSE	FALSE	FALSE	FALSE
4	0.3525	0	0	40	0	FALSE	FALSE	FALSE	FALSE
4	0.52875	1	0	49	0	FALSE	FALSE	FALSE	FALSE
4	0.705	0	0	56	0	FALSE	FALSE	FALSE	FALSE
4	0.88125	0	1	59	1	FALSE	FALSE	FALSE	FALSE
4	1.0575	0	0	61	0	FALSE	FALSE	FALSE	FALSE
5	0.3525	0	33	151	15	FALSE	FALSE	FALSE	FALSE
5	0.52875	0	0	170	0	FALSE	FALSE	FALSE	FALSE
5	0.705	1	0	195	0	FALSE	FALSE	FALSE	FALSE
5	0.88125	0	1	222	0	FALSE	FALSE	FALSE	FALSE
5	1.0575	1	0	218	0	FALSE	FALSE	FALSE	FALSE

Table 2.2: 20 vertices unit square random graphs, vertex cover from 1 to 19 with 8 trials per cover size value; time in milliseconds, distance is the unit square distance chosen

cover size	distance	Tricia t	Marc t	Chris t	Stefan t	Tricia r	Marc r	Chris r	Stefan r
6	0.3525	3	17	384	13	FALSE	FALSE	FALSE	FALSE
6	0.52875	0	0	532	0	FALSE	FALSE	FALSE	FALSE
6	0.705	0	1	651	0	FALSE	FALSE	FALSE	FALSE
6	0.88125	0	1	631	0	FALSE	FALSE	FALSE	FALSE
6	1.0575	0	0	639	0	FALSE	FALSE	FALSE	FALSE
7	0.3525	13	80	890	12	FALSE	FALSE	FALSE	FALSE
7	0.52875	1	1	1156	1	FALSE	FALSE	FALSE	FALSE
7	0.705	1	0	1408	0	FALSE	FALSE	FALSE	FALSE
7	0.88125	0	0	1430	0	FALSE	FALSE	FALSE	FALSE
7	1.0575	0	0	1514	0	FALSE	FALSE	FALSE	FALSE
8	0.3525	15	36	1907	16	FALSE	FALSE	FALSE	FALSE
8	0.52875	0	1	2357	1	FALSE	FALSE	FALSE	FALSE
8	0.705	0	0	2671	0	FALSE	FALSE	FALSE	FALSE
8	0.88125	0	0	2961	0	FALSE	FALSE	FALSE	FALSE
8	1.0575	0	0	2962	0	FALSE	FALSE	FALSE	FALSE
9	0.3525	19	53	2968	118	FALSE	FALSE	FALSE	FALSE
9	0.52875	1	0	3997	1	FALSE	FALSE	FALSE	FALSE
9	0.705	1	0	4623	0	FALSE	FALSE	FALSE	FALSE
9	0.88125	0	1	4877	0	FALSE	FALSE	FALSE	FALSE
9	1.0575	0	1	5058	0	FALSE	FALSE	FALSE	FALSE

Table 2.3: 20 vertices unit square random graphs, vertex cover from 1 to 19 with 8 trials per cover size value; time in milliseconds, distance is the unit square distance chosen

cover size	distance	Tricia t	Marc t	Chris t	Stefan t	Tricia r	Marc r	Chris r	Stefan r
10	0.3525	1	34	4853	118	FALSE	FALSE	FALSE	FALSE
10	0.52875	0	1	6675	0	FALSE	FALSE	FALSE	FALSE
10	0.705	0	0	6935	1	FALSE	FALSE	FALSE	FALSE
10	0.88125	0	1	7291	0	FALSE	FALSE	FALSE	FALSE
10	1.0575	0	0	7561	0	FALSE	FALSE	FALSE	FALSE
11	0.3525	24	78	6021	124	FALSE	FALSE	FALSE	FALSE
11	0.52875	0	4	7970	4	FALSE	FALSE	FALSE	FALSE
11	0.705	0	1	8593	0	FALSE	FALSE	FALSE	FALSE
11	0.88125	3	1	9376	0	FALSE	FALSE	FALSE	FALSE
11	1.0575	0	2	9475	0	FALSE	FALSE	FALSE	FALSE
12	0.3525	5	146	7315	240	FALSE	FALSE	FALSE	FALSE
12	0.52875	14	35	9305	2	FALSE	FALSE	FALSE	FALSE
12	0.705	1	19	10279	1	FALSE	FALSE	FALSE	FALSE
12	0.88125	0	4	10715	0	FALSE	FALSE	FALSE	FALSE
12	1.0575	0	3	10827	0	FALSE	FALSE	FALSE	FALSE
13	0.3525	24	332	7777	126	FALSE	FALSE	FALSE	FALSE
13	0.52875	15	169	10527	3	FALSE	FALSE	FALSE	FALSE
13	0.705	4	19	10522	1	FALSE	FALSE	FALSE	FALSE
13	0.88125	0	13	11593	0	FALSE	FALSE	FALSE	FALSE
13	1.0575	1	13	11747	0	FALSE	FALSE	FALSE	FALSE
14	0.52875	30	663	10221	8	FALSE	FALSE	FALSE	FALSE
14	0.705	1	41	11187	1	FALSE	FALSE	FALSE	FALSE
14	0.88125	0	50	12160	0	FALSE	FALSE	FALSE	FALSE
14	1.0575	0	50	12142	0	FALSE	FALSE	FALSE	FALSE
15	0.52875	190	2734	10965	2	FALSE	FALSE	FALSE	FALSE
15	0.705	24	355	11149	1	FALSE	FALSE	FALSE	FALSE
15	0.88125	0	197	12148	0	FALSE	FALSE	FALSE	FALSE
15	1.0575	1	199	12307	0	FALSE	FALSE	FALSE	FALSE

Table 2.4: 20 vertices unit square random graphs, vertex cover from 1 to 19 with 8 trials per cover size value; time in milliseconds, distance is the unit square distance chosen

cover size	distance	Tricia t	Marc t	Chris t	Stefan t	Tricia r	Marc r	Chris r	Stefan r
16	0.52875	449	2955	11123	1	FALSE	FALSE	FALSE	FALSE
16	0.705	71	743	11190	1	FALSE	FALSE	FALSE	FALSE
16	0.88125	0	790	12063	0	FALSE	FALSE	FALSE	FALSE
16	1.0575	1	802	12335	1	FALSE	FALSE	FALSE	FALSE
17	1.0575	1	3223	12231	0	FALSE	FALSE	FALSE	FALSE
14	0.3525	2	8	24	1	TRUE	TRUE	TRUE	TRUE
15	0.3525	5	45	6	0	TRUE	TRUE	TRUE	TRUE
16	0.3525	1	1	2	0	TRUE	TRUE	TRUE	TRUE
17	0.3525	1	3	0	0	TRUE	TRUE	TRUE	TRUE
17	0.52875	1	17	0	1	TRUE	TRUE	TRUE	TRUE
17	0.705	36	198	0	0	TRUE	TRUE	TRUE	TRUE
17	0.88125	3	1	3	1	TRUE	TRUE	TRUE	TRUE
18	0.3525	0	1	0	0	TRUE	TRUE	TRUE	TRUE
18	0.52875	0	1	0	1	TRUE	TRUE	TRUE	TRUE
18	0.705	6	1	0	0	TRUE	TRUE	TRUE	TRUE
18	0.88125	2	0	0	0	TRUE	TRUE	TRUE	TRUE
18	1.0575	0	6548	1	0	TRUE	TRUE	TRUE	TRUE
19	0.3525	1	0	0	0	TRUE	TRUE	TRUE	TRUE
19	0.52875	1	2	0	0	TRUE	TRUE	TRUE	TRUE
19	0.705	1	0	0	1	TRUE	TRUE	TRUE	TRUE
19	0.88125	1	0	0	0	TRUE	TRUE	TRUE	TRUE
19	1.0575	1	0	0	0	TRUE	TRUE	TRUE	TRUE

Table 2.5: 20 vertices unit square random graphs, vertex cover from 1 to 19 with 8 trials per cover size value; time in milliseconds, distance is the unit square distance chosen