A Project Report On

Implementation of Maximum Clique Problem

Submitted by

Divija Nagaraju - 14IT112

Mukta Kulkarni - 14IT220

Pooja Soundalgekar - 14IT230

Yogitha A N - 14IT251

IV Sem B.Tech (IT)

in partial fulfillment for the award of the degree

of

Bachelor of Technology

In

Information Technology

At

Department of Information Technology
National Institute of Technology Karnataka, Surathkal.
February 2016

ABSTRACT

The Maximum Clique problem statement states that given a graph, one desires to find the largest number of vertices, any two of which are adjacent.

A branch-and-bound algorithm for the maximum clique problem—is presented with a vertex order taken and with a new pruning strategy. The problem is computationally equivalent to the maximum independent (stable) set problem. The algorithm performs successfully for many instances when applied to random graphs.

An attempt has be made to optimise the algorithm presented in the paper by guaranteeing an upper bound on the maximum size of clique by implementing a greedy colouring strategy.

1. INTRODUCTION

A clique of a graph is a set of vertices, any two of which are adjacent. Two vertices are said to be adjacent if they are connected by an edge. Maximum cliques refer to those which are the largest among all cliques in a graph .In the maximum clique problem, one desires to find one maximum clique of an arbitrary undirected graph.

This problem is computationally equivalent to some other important graph problems, for example, the maximum independent set problem and the minimum vertex cover problem.

Considering the maximum independent set problem. An independent set or stable set is a set of vertices in a graph, no two of which are adjacent.A maximum independent set is an independent set of largest possible size for a given graph G. The problem of finding such a set is called the maximum independent set problem and is an NP-hard optimization problem.

Thus we may infer that the independent set problem and the clique problem are complementary: a clique in G is an independent set in the complement graph of G and vice versa. Since these are NP-hard problems , no polynomial time algorithms are expected to be found.

The Algorithm presented is a type of branch-and-bound algorithm. This consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

The Naive Approach

```
function : clique(U; size)
   if |U | = 0 then
      if size > max then
         max:=size
         New record; save it.
      end if
      return
   end if
while U # = ∅ do
   if size + |U | 6 max then
      return
   end if
   i:=min{j | v j ∈ U }
   U :=U \{v i }
   clique(U ∩ N (v i ); size + 1)
end while
return
function : old
   max:=0
   clique(V; 0)
return
```

The basic pruning strategy used in the algorithm is to backtrack when the set *U* becomes so small that even if all vertices left could be added to get a clique, the size of that clique would not exceed that of the largest clique encountered so far in the search. Moreover, if we explicitly search for a clique of a given size, we can modify the algorithm and use this information for pruning from the beginning of the search.This is being implemented because of the condition size+|U|⩽max, otherwise the algorithm would go through every single clique of the graph.

The Algorithm Presented

```
function : clique(U; size)
    if |U | = 0 then
        if size > max then
            max:=size
            New record; save it.
            found:=true
        end if
        return
    end if
while U # = ∅ do
    if size + |U | 6 max then
        return
    end if
    i:=min{j | v j ∈ U }
    if size + c[i] 6 max then
        return
    end if
    U :=U \{v i }
    clique(U ∩ N (v i ); size + 1)
    if found = true then
        return
    end if
    end while
    return
function : new
    max:=0
    for i:=n downto 1 do
        found:=false
```

clique(S i ∩ N (v i); 1)

c[i]:=max

end for

return

The previous algorithm searches for the maximum clique by sequentially considering the vertices in the respective sets. In the proposed algorithm, this ordering is reversed: we first consider cliques in S_n that contain v_n (the largest such clique is, of course, $\{v_n\}$), then cliques in S_{n-1} that contain v_{n-1} and so on. However, a different pruning approach has been used to make the algorithm faster :size+c[i]⩽max Namely, if we search for a clique of size greater than s, then we can prune the search if we consider v_i to become the (j+1)-th vertex and j+c(i)⩽s. This pruning strategy has a profound impact on the search.

Code

Naive Approach

```
#include <bits/stdc++.h>
using namespace std;
int maximum=0;

void clique(int u[], int n, int size,int e[10][10])
{
        int i,flag=0,mini=0,cnt=0;
        for(i=0;i<n;i++)
        {
                if(u[i]==1)
                {

                        cnt++;
                        flag=1;
                }
        }
   cout<<endl;
        if(flag==0)
        {
                if(size>maximum)
                {
                        maximum=size;
        cout<<maximum<<"ans";
                }
                return;
        }
        while(cnt!=0)
        {
                if((size+cnt)<=maximum)
                        return;
```

```
                for(i=0;i<n;i++)
                {
                        if(u[i]==1)
                        {
                                mini=i;
                                break;
                        }
                }
                u[mini]=0;
                cnt--;
                for(i=0;i<n;i++)
                {
                        if(u[i]==1&&e[mini][i]==1)
                                u[i]=1;
        else
           u[i]=0;
                }

                clique(u,n,size+1,e);
        }
        return;

}

void old(int v,int e[10][10])
{
        int i;
        int vertex[10]={0};
        for(i=0;i<v;i++)
                vertex[i]=1;
        clique(vertex,v,0,e);
   return;
}
```

```
int main()
{
   int n,e[10][10],src,dest,edge,i,j;
        cout<<"enter number of vertices in the graph"<<endl;
        cin>>n;
        cout<<"enter number of edges"<<endl;
        cin>>edge;
   for(i=0;i<n;i++)
   {
     for(j=0;j<n;j++)
     {
       e[i][j]==0;
     }
   }
        for(i=0;i<edge;i++)
        {
                cout<<"enter source vertex"<<endl;
                cin>>src;
                cout<<"enter destination vertex"<<endl;
                cin>>dest;
                e[src][dest]=1;
                e[dest][src]=1;
        }
        old(n,e);
        cout<<"Maximum clique is "<<maximum<<endl;
        return(0);
}
```

Presented New Code

```
#include <iostream>
using namespace std;
int maximum=0,c[1002]={0},n;
int graph[1002][1002]={0},d[1002]={0};
```

```
bool found;

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] >= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
```

```
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l+(r-l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
            merge(arr, l, m, r);
    }
}

void clique(int u[1002],int siz,int setu_index)
{
    int i,uu[1002]={0},setuu_index,j;

    if(setu_index==0)
    {
        if(siz>maximum)
        {
            maximum=siz;
        //    cout<<siz<<"\n";
            found=true;
        }
        return;

    }
    while(setu_index!=0)
```

```
{

   if(siz+setu_index<=maximum)
      return;

   for(j=0;j<n;j++)
      if(u[j]!=0)
      {
         i=j;

         break;
      }

   if(siz+c[u[i]]<=maximum)
      return;



   setuu_index=0;
   for(j=0;j<n;j++)
   {

      if(graph[u[j]][u[i]]==1 && u[j]!=0)
      {

            uu[setuu_index]=u[j];
            setuu_index++;
        }
   }
   u[i]=0;



   clique(uu,siz+1,setuu_index);
   if(found==true)
      return;
   setu_index--;
```

```
    }
    return;
}
int main() {
        int e,i,j,seta_index,a[1002],x,y;

        cout<<"Enter the number of vertices:";
        cin>>n;
        cout<<"Enter the number of edges:";
        cin>>e;
        cout<<"Enter the edges:";
        for(i=0;i<e;i++)
        {    cin>>x>>y;
           graph[x][y]=1;
           graph[y][x]=1;
           d[x]++;
           d[y]++;
        }
        mergeSort(d,0,n);
        for(i=n;i>=1;i--)
        {
           found= false;

           seta_index=0;
           for(j=0;j<n;j++)
           a[j]=0;
           for(j=i;j<=n;j++)
           {
              if(graph[i][j]==1)
              {
                 a[seta_index]=j;
                 seta_index++;
              }
           }
           c[i]=0;
```

```
            clique(a,1,seta_index);
            c[i]=maximum;

        }
       cout<<maximum;
   return 0;
}
```

Optimised Code :

```
#include <iostream>
using namespace std;
int maximum=0,c[1002]={0},n;
int graph[1002][1002]={0},d[1002]={0};
bool found;
void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 =  r - m;
   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1+ j];
   i = 0;
   j = 0;
   k = l;
   while (i < n1 && j < n2)
   {
      if (L[i] >= R[j])
      {
         arr[k] = L[i];
         i++;
```

```
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l+(r-l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
            merge(arr, l, m, r);
    }
}
void clique(int u[1002],int siz,int setu_index)
{
    int i,uu[1002]={0},setuu_index,j;
```

```
if(setu_index==0)
{
   if(siz>maximum)
   {
      maximum=siz;
   //    cout<<siz<<"\n";
      found=true;
   }
   return;

}
while(setu_index!=0)
{

   if(siz+setu_index<=maximum)
      return;

   for(j=0;j<n;j++)
      if(u[j]!=0)
      {
         i=j;

         break;
      }

   if(siz+c[u[i]]<=maximum)
      return;



   setuu_index=0;
   for(j=0;j<n;j++)
   {

      if(graph[u[j]][u[i]]==1 && u[j]!=0)
      {
```

```
                uu[setuu_index]=u[j];
                setuu_index++;
            }
        }
        u[i]=0;


        clique(uu,siz+1,setuu_index);
        if(found==true)
            return;
        setu_index--;
    }
   return;
}
int main() {
        int e,i,j,seta_index,a[1002],x,y;
        int k,u,cr;
        int result[1002];
        bool available[1002];
        cout<<"Enter the number of vertices:";
        cin>>n;
        cout<<"Enter the number of edges:";
        cin>>e;
        cout<<"Enter the edges:";
        for(i=0;i<e;i++)
        {    cin>>x>>y;
            graph[x][y]=1;
            graph[y][x]=1;
            d[x]++;
            d[y]++;
        }
        mergeSort(d,0,n);
        result[1]  = 1;
        for ( u = 2; u <=n; u++)
```

```
    result[u] = -1;

  for ( cr = 1; cr <=n; cr++)
    available[cr] = false;
  for ( u = 2; u <= n; u++)
  {

    for(i=1;i<=n;i++)
    {
      if(graph[i][u]==1)
      if (result[i] != -1)
        available[result[i]] = true;
    }
      for (cr = 1; cr <=n; cr++)
      if (available[cr] == false)
        break;

    result[u] = cr;

    for (i=1;i<=n;i++)
      if(graph[i][u]==1)
        if (result[i] != -1)
          available[result[i]] = false;
  }
  for(i=1;i<=n;i++)
  cout<<result[i]<<"a\n";
  mergeSort(result,1,n+1);
   cout<<"The upper bound to maximum clique- as in the upper bound to minimum coloring is
"<<result[1]<<"\n";

  for(i=n;i>=1;i--)
  {
    found= false;

    seta_index=0;
```

```
            for(j=0;j<n;j++)
            a[j]=0;
            for(j=i;j<=n;j++)
            {
               if(graph[i][j]==1)
               {
                  a[seta_index]=j;
                  seta_index++;
               }
            }
            c[i]=0;
            clique(a,1,seta_index);
            c[i]=maximum;

         }
         cout<<"Maximum clique is : "<<maximum;
      // your code goes here
      return 0;
}
```

Input Graph :

Vertices : 8

Edges : 15

1-3

1-5

1-6

1-8

2-7

2-6

2-5

3-8

3-7

3-6

4-8

4-7

4-6

5-7

5-8

Output : The Maximum clique is 3

For Optimised Code : The upper bound to maximum clique- as in the upper bound to minimum coloring is 3

Maximum Clique is 3

Description for Code Optimisation

During the analysis of the maximum clique problem we came to this realisation that minimum coloring of a graph is nothing but the maximum clique in terms of value.
For minimum colouring graph problem which itself is NP Complete problem we have used a greedy approach which is as follows :

1. Color first vertex with first color.
2. Do following for remaining V-1 vertices.
    a) Consider the currently picked vertex and color it with the
    lowest numbered color that has not been used on any previously
    colored vertices adjacent to it. If all previously used colors
    appear on vertices adjacent to v, assign a new color to it.

This is not a guaranteed answer but gives a guaranteed upper bound to the minimum colouring graph problem.The same upper bound is used by us to optimise the maximum clique problem.

Conclusion

The algorithm presented improves on old methods for several types of graphs, including sparse, random graphs and graphs with certain combinatorial properties. Since we are dealing with an NP-hard problem, it is expected that different algorithms perform well for different types of instances. One may then argue which instances are the most important ones.

Clique detection has traditionally been a very important tool in chemoinformatics, where one problem is to identify common substructures between a collection of molecules known to possess certain pharmacological properties. The molecules are viewed as (low treewidth) graphs and clique detection is used to find common structures.Their usage also includes modelling of social cliques graph-theoretically.The electrical field uses cliques to analyze communications networks and design efficient circuits for computing partially specified Boolean functions.

ELSEVIER

Discrete Applied Mathematics 120 (2002) 197–207

DISCRETE APPLIED MATHEMATICS

A fast algorithm for the maximum clique problem ☆

Patric R. J. Östergård *

Department of Computer Science and Engineering, Helsinki University of Technology, P.O. Box 5400, 02015 HUT, Finland

Received 12 October 1999; received in revised form 29 May 2000; accepted 19 June 2001

Abstract

Given a graph, in the maximum clique problem, one desires to find the largest number of vertices, any two of which are adjacent. A branch-and-bound algorithm for the maximum clique problem—which is computationally equivalent to the maximum independent (stable) set problem—is presented with the vertex order taken from a coloring of the vertices and with a new pruning strategy. The algorithm performs successfully for many instances when applied to random graphs and DIMACS benchmark graphs.

1. Introduction

We denote an undirected graph by $G=(V,E)$, where V is the set of vertices and E is the set of edges. Two vertices are said to be *adjacent* if they are connected by an edge. A *clique* of a graph is a set of vertices, any two of which are adjacent. Cliques with the following two properties have been studied over the last three decades: *maximal* cliques, whose vertices are not a subset of the vertices of a larger clique, and *maximum* cliques, which are the largest among all cliques in a graph (maximum cliques are clearly maximal). In this paper, the latter type of cliques are studied.

In the *maximum clique problem*, one desires to find one maximum clique of an arbitrary undirected graph. This problem is computationally equivalent to some other important graph problems, for example, the *maximum independent* (*or stable*) *set problem* and the *minimum vertex cover problem*. Since these are NP-hard problems [6], no polynomial time algorithms are expected to be found. Nevertheless, as these problems have several important practical applications, it is of great interest to try to develop fast, exact algorithms for small instances. Another direction of research, which has recently been fairly popular, is that of using stochastic methods to find

☆ The research was supported by the Academy of Finland.

* Tel.: +358-9-4512341; fax: +358-9-4512359.

E-mail address: patric.ostergard@hut.fi (P.R.J. Östergård).

0166-218X/02/$ - see front matter
PII: S0166-218X(01)00290-6

as large cliques as possible, without proving optimality; see the survey of Pardalos and Xue [9], which also contains an extensive bibliography on the maximum clique problem.

Since the early 1970s, many papers have been published with algorithms for the maximum clique problem. Unfortunately, it is often difficult to compare these algorithms, and no extensive comparison between the published algorithms has been carried out. Fortunately, a set of benchmark graphs are maintained by DIMACS. These graphs and random graphs can be used to get some indication of the quality of new algorithms.

Old algorithms and the new algorithm are discussed in Section 2. A comparison between these that is based on computational experiments with DIMACS benchmark graphs and with random graphs is carried out in Section 3. The paper is concluded in Section 4.

2. Maximum clique algorithms

Algorithm 1. Old algorithm.

```
function clique(U, size)
 1:     if |U| = 0 then
 2:         if size > max then
 3:             max:=size
 4:             New record; save it.
 5:         end if
 6:         return
 7:     end if
 8:     while U ≠ ∅ do
 9:         if size + |U| ⩽ max then
10:             return
11:         end if
12:         i:=min{j | v_j ∈ U}
13:         U:=U\{v_i}
14:         clique(U ∩ N(v_i), size + 1)
15:     end while
16:     return
function old
17:     max:=0
18:     clique(V, 0)
19:     return
```

Before presenting the new algorithm, we will briefly discuss old algorithms, starting with that of Carraghan and Pardalos [5] (and, independently, of Applegate and Johnson [1]), which can be seen as a basic form of most published algorithms.

2.1. Old algorithms

The algorithm in [5] is presented as Algorithm 1. The following notation is used. The set of vertices adjacent to a vertex v is denoted by $N(v)$ and the number of vertices in the graph is n. The variable *max*, which is global, gives the size of a maximum clique when the algorithm terminates.

The performance of the algorithm depends on the ordering the vertices, $v_1, v_2, \ldots, v_n$. We will return to heuristic for ordering later. Each vertex taken in line 12 should be saved to be able to extract the whole clique whenever line 4 is reached.

Without the pruning strategy in line 9 (in implementing the algorithm, the lines 8–11 can be combined into a **for** statement), this algorithm would go through every single clique of the graph. The pruning strategy is to backtrack when the set U becomes so small that even if all vertices left could be added to get a clique, the size of that clique would not exceed that of the largest clique encountered so far in the search. Moreover, if we explicitly search for a clique of a given size, we can modify the algorithm and use this information for pruning from the beginning of the search.

Some speed-up can be obtained if the test in line 1 is changed so that the recursion is stopped whenever very few vertices are left (often 0 or 1) and corresponding calculations are carried out on a case-by-case basis.

Although this algorithm is very simple, it is currently the best known algorithm for sparse graphs.

Most attempts to improve on this straightforward algorithm are based on methods for calculating upper bounds (other than from the size of the set U in Algorithm 1) during the search. Almost without exceptions, such bounds are obtained from vertex-colorings.

In a vertex-coloring, adjacent vertices must be assigned different colors. If a graph, or an induced subgraph, can be colored with, say, s colors, then the graph, or subgraph, cannot contain a clique of size $s+1$.

In implementing strategies based on calculating upper bounds, a trade-off has to be made: coloring can lead to a considerable reduction of the number of nodes in the search tree but is also very time-consuming. Recent algorithms of this kind have been published by Babel [2], Balas and Xue [3], Sewell [10], and Wood [11].

2.2. The new algorithm

We will now consider the new algorithm. Let $S_i = \{v_i, v_{i+1}, \ldots, v_n\}$. The old algorithm searches for the maximum clique by first considering cliques in S_1 that contain v_1, then cliques in S_2 that contain v_2, and so on. In the new algorithm, this ordering is reversed: we first consider cliques in S_n that contain v_n (the largest such clique is, of course, $\{v_n\}$), then cliques in S_{n-1} that contain v_{n-1}. If this procedure is carried out with the same pruning as in Algorithm 1, we get a slower algorithm. However, this approach makes it possible to introduce a new pruning strategy as shown in Algorithm 2.

The table $c[i]$ and the variables *found* (which is boolean) and *max* are all global.

Algorithm 2. New algorithm.

```
function clique(U, size)
 1:     if |U| = 0 then
 2:         if size > max then
 3:             max:=size
 4:             New record; save it.
 5:             found:=true
 6:         end if
 7:         return
 8:     end if
 9:     while U ≠ ∅ do
10:         if size + |U| ⩽ max then
11:             return
12:         end if
13:         i:=min{j | v_j ∈ U}
14:         if size + c[i] ⩽ max then
15:             return
16:         end if
17:         U:=U\{v_i}
18:         clique(U ∩ N(v_i), size + 1)
19:         if found = true then
20:             return
21:         end if
22:     end while
23:     return
function new
24:     max:=0
25:     for i:=n downto 1 do
26:         found:=false
27:         clique(S_i ∩ N(v_i), 1)
28:         c[i]:=max
29:     end for
30:     return
```

The function $c(i)$ gives the largest clique in S_i. Obviously, for any $1 \leqslant i \leqslant n-1$, we have that $c(i) = c(i+1)$ or $c(i) = c(i+1)+1$. Moreover, we have $c(i) = c(i+1)+1$ iff there is a clique in S_i of size $c(i+1)+1$ that includes the vertex v_i. So, starting from $c(n) = 1$, we search for such cliques. If a clique is found, $c(i) = c(i+1)+1$, otherwise $c(i) = c(i+1)$. The size of a maximum clique is given by $c(1)$.

Old values of the function $c(i)$ enables the new pruning strategy (in line 14). Namely, if we search for a clique of size greater than s, then we can prune the search if we consider v_i to become the $(j+1)$-th vertex and $j + c(i) \leqslant s$. The following example

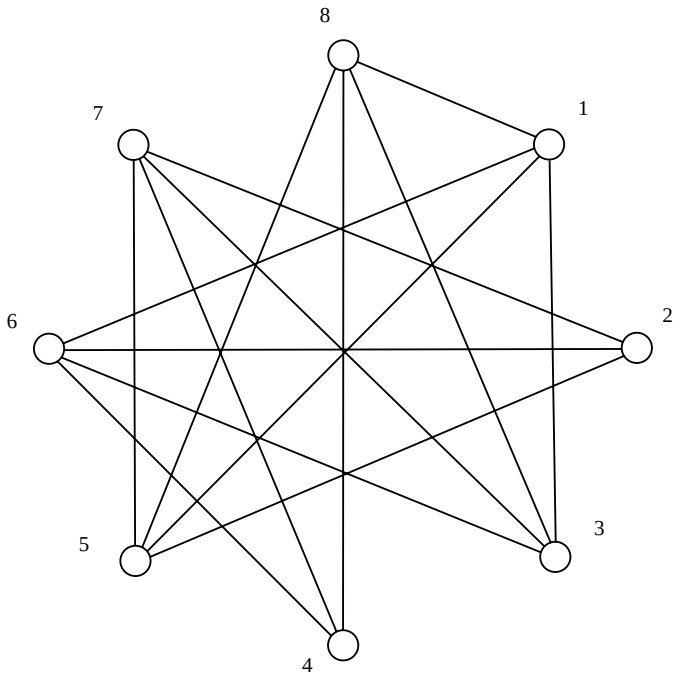

Fig. 1. Example graph.

illustrates that the new pruning strategy occasionally has a profound impact on the search.

Example. Consider the graph with vertex set $V = \{v_1, v_2, \ldots, v_{2n}\}$ and edges between v_i and v_j exactly when $|i - j| > 1$. Since v_i and v_{i+1} are not connected by an edge, combinatorial arguments show that a maximum clique has size n. With the new algorithm, we get that $c(i) = n + 1 - \lceil i/2 \rceil$. Moreover, these values are obtained in linear time. Namely, when i is odd, after fixing v_i, the smallest index among the vertices in the working set is $i + 2$, and as $c(i + 2) = c(i + 1) - 1$, we cannot achieve a clique of size $c(i + 1) + 1$ which we are aiming for (this part takes constant time). When i is even, we find a clique of size $c(i + 1) + 1$ without having to backtrack, that is, in linear time. The old algorithm finds a maximum clique in linear time, but continues to spend even exponential time in the search for a possible clique of size $n + 1$.

As another example, in Table 1, we carry out Algorithm 2 on the graph in Fig. 1. The word *prune* indicates that pruning based on the value of $c(i)$ is carried out. The numbers indicate the indices of the vertices in U when clique is called (on level 1 we just have the vertices in S_i). The vertex that is chosen in line 25 on level 1 and in line 13 on the other levels is underlined.

The presented algorithm can, with small modifications, be used to find *all* maximum cliques.

Table 1
Proceeding of algorithm for example graph

1: $\underline{8}$ ($c(8)=1$; largest clique is $\{8\}$)
1: $\underline{7}$ 8
 2: (empty; $c(7)=1$)
1: $\underline{6}$ 7 8
 2: (empty; $c(6)=1$)
1: $\underline{5}$ 6 7 8
 2: $\underline{7}$ 8 ($c(5)=2$; largest clique is $\{5,7\}$)
1: $\underline{4}$ 5 6 7 8
 2: $\underline{6}$ 7 8 (*prune*; $c(4)=2$)
1: $\underline{3}$ 4 5 6 7 8
 2: $\underline{6}$ 7 8 (*prune*; $c(3)=2$)
1: $\underline{2}$ 3 4 5 6 7 8
 2: $\underline{5}$ 6 7
  3: $\underline{7}$ ($c(2)=3$; largest clique is $\{2,5,7\}$)
1: $\underline{1}$ 2 3 4 5 6 7 8
 2: $\underline{3}$ 5 6 8
  3: $\underline{6}$ 8
   4: (empty)
 2: 3 $\underline{5}$ 6 8 (*prune*; $c(1)=3$)

2.3. Searching for a clique of given size

In [5], it is described how a lower bound on the size of a maximum clique can be used to speed up the search. A similar approach does not seem to be possible here; instead we will see how we can improve the algorithm when proving nonexistence results. That is, for proving that there is no clique of a prescribed size s, we can proceed as follows.

Before we start the main algorithm, we calculate the values of $d(i)$, the size of the largest clique with the vertices in $\{v_1, v_2, \ldots, v_i\}$. (If this is done up to $i=n$, we have solved the whole problem.) The values of $d(i)$ are calculated for $1 \leqslant i \leqslant n/2$. Then, in the main algorithm, we check the value of $c(i+1)+d(i)$ for all $i \leqslant n/2$. If for some i, $c(i+1)+d(i)<s$, we can stop the search as no clique of size s then exists in the graph.

2.4. Ordering the vertices

For good performance of the algorithm presented in Section 2.2, a proper heuristic for ordering the vertices has to be chosen. One can think of several ways of doing this, and these orderings may have different effects for different types of graphs.

The heuristic used in [5] for the algorithm in Section 2.1 is to sort the vertices with respect to their degrees (the number of incident edges) so that v_1 has smallest degree. To get $v_2, v_3, \ldots$, one repeatedly takes the vertex with smallest degree from the subgraph induced by the vertices that have not yet been taken.

Since previous algorithms that perform better than the old algorithm presented in Section 2.1 use vertex-colorings along the search, it seems plausible that a vertex-coloring

could be used in some way to get a good initial ordering. Using a coloring that can be found in reasonable time, the vertices are ordered so that those belonging to the same color class are grouped. To get such a coloring, which also is an upper bound on the maximum clique, we use the following greedy heuristic, which is due to Biggs [4]:

The color classes are determined one at a time. When determining a new color class, the graph induced by the uncolored vertices is first constructed. Then, as long as there exists a vertex that can be added to the color class, such a vertex with largest degree is added. The vertices are labeled $v_n, v_{n-1}, \ldots$ in the order they are added to a color class.

3. Experimental results

To properly evaluate the new algorithm, it is necessary to carry out practical experiments. We will here show how the new algorithm performs for random graphs and some of the DIMACS benchmark graphs.

We first look at random graphs. Six algorithms are compared in the performance tests: [5] as programmed in [1] (called Old), the new algorithm (called New), and four advanced algorithms using upper bounds from vertex-colorings.

The computational results with random graphs are presented in Table 2. The entries show the average run times in CPU seconds on a 500 MHz PC with Linux operating system. For each entry, ten random graphs were constructed and used as input; the same ten graphs were used for both Old and New. The graphs were constructed by considering each pair of vertices and inserting an edge with probability p. This probability is called the *edge density* (or *edge probability*).

The first four entries are, when applicable, taken from [2, Table 2], [3, Table 3], and [10, Table II] [11, Table 1]. The times from [10] have been adjusted based on the machine benchmarks in [10, Table VI]. Our user times for the DIMACS machine benchmarks `r100.5`–`r500.5` are 0.01, 0.23, 1.52, 10.05, and 39.41, indicating that the computer used in our experiments is 19 times faster than the one used in [10]. In [2,3,11], only the computer that was used is mentioned, so only a rough comparison is possible based on general benchmarks for computer speeds; these times are not adjusted in our table.

Looking at the results in Table 2, the new algorithm is clearly the fastest for sparse, random graphs. The break point where there are old algorithms that perform better is approximately 0.8–0.9 for 100 vertices, 0.6–0.7 for 200 and 300 vertices, and greater than 0.5 for 500 vertices. It is interesting that the new algorithm has the property, as for [3] but not for [5], that when the density gets very close to one, the speed of the algorithm increases. This is shown by the entries for 100 vertices and edge densities 0.9 and 0.95.

In many studies, the number of search tree nodes are presented for the various instances. This gives some information about the nature of an algorithm, but cannot be used to evaluate the quality of it. Some algorithms, like ours, traverse many nodes

Table 2
Benchmark results (random graphs)

Vertices	Edge density	[2][a]	[3][b]	[10]	[11][c]	Old	New
100	0.6	2.03	0.55	0.42	1.93	0.02	0.01
100	0.7	5.41	1.29	0.53	3.44	0.10	0.03
100	0.8	17.69	4.24	0.68	6.52	0.89	0.29
100	0.9	28.40	8.26	0.89	12.12	20.71	2.97
100	0.95		1.24			141.57	0.88
200	0.4	4.78	1.73		5.51	0.04	0.03
200	0.5	17.32	5.72	1.16	12.68	0.18	0.09
200	0.6	87.32	26.90	2.37	45.66	1.24	0.76
200	0.7	631.10	180.55	7.53	341.90	18.78	13.53
200	0.8	10,761.17	2331.26	39.68		1088.20	659.88
300	0.3	6.99				0.06	0.05
300	0.4	28.49	10.03			0.25	0.21
300	0.5	152.19	50.00	5.16		1.69	0.90
300	0.6	1242.19	378.75	22.74		21.74	15.66
300	0.7	16,221.38	6107.09			706.65	542.18
500	0.2	10.57				0.09	0.09
500	0.3	57.56				0.42	0.37
500	0.4	327.80	123.51			3.27	2.68
500	0.5	3082.30	1118.94	79.63		41.85	32.49
500	0.6		17,142.84			1177.01	821.08
1000	0.1					0.18	0.29
1000	0.2					0.95	0.93
1000	0.3					9.92	7.34
1000	0.4					161.45	103.90

[a]CDC Cyber 995.
[b]NeXTstation.
[c]Sun Sparcstation 10.

and spend little time in each node, whereas others traverse few nodes and spend more time in each. The CPU time alone should be the measure of the quality.

We also tested the new algorithm on DIMACS test graphs, which can be obtained electronically from <URL:ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/> and have been described in [7]. The results are displayed in Table 3. The algorithm from [5] follows the trend from Table 2 for these problems and performs worse or occasionally much worse than the new algorithm; hence it is not included in the table.

The DIMACS problems for which there are benchmark results in both [3] and [11] have been considered. Many of these graphs have high density, but the results are still encouraging. The new algorithm performs very well, especially for the problems of type `brock`, `keller`, and `p_hat`. The instances of type `brock` have a clique hidden that is much larger than what could be expected from the edge density; the instances of type `keller` are based on Keller's conjecture on tilings using hypercubes; and the instances of type `p_hat` are certain types of random graphs having wider node degree spread and larger cliques.

Table 3
Benchmark results (DIMACS graphs)

DIMACS	n	d	Size	[3][a]	[10]	[11][b]	New
brock200_1	200	0.75	21	172.19		805.20	54.29
brock200_2	200	0.50	12	1.90	1.58	3.83	0.05
brock200_3	200	0.61	15	7.96		38.50	0.44
brock200_4	200	0.66	17	26.00	4.16	92.95	0.98
c-fat200-1	200	0.08	12	0.03		0.02	0.01
c-fat200-2	200	0.16	24	0.02		0.02	0.01
c-fat200-5	200	0.43	58	0.13		0.13	7.81
c-fat500-1	500	0.04	14	0.08		0.02	0.07
c-fat500-2	500	0.07	26	0.11		0.08	0.08
c-fat500-5	500	0.19	64	0.24		0.16	10,442.64
c-fat500-10	500	0.37	126	0.64		0.03	0.07
hamming6-2	64	0.90	32	0.01		0.01	0.01
hamming6-4	64	0.35	4	0.01		0.07	0.01
hamming8-2	256	0.97	128	0.34		0.01	0.04
hamming8-4	256	0.64	16	1.74	4.58	79.15	0.86
hamming10-2	1024	0.99	512	30.89		0.07	2.52
johnson8-2-4	28	0.56	4	0.01		0.02	0.01
johnson8-4-4	70	0.77	14	0.01		0.18	0.01
johnson16-2-4	120	0.76	8	0.01		195.80	0.27
keller4	171	0.65	11	3.39	1.95	18.45	0.50
MANN_a9	45	0.93	16	0.02		0.10	0.01
MANN_a27	378	0.99	126	1362.82	12.58	704.30	$>10{,}000.00$
p_hat300-1	300	0.24	8	0.80	2.00	1.47	0.04
p_hat300-2	300	0.49	25	5.83	2.79	10.05	0.99
p_hat500-1	500	0.25	9	4.83		13.72	0.29
p_hat500-2	500	0.50	36	228.11		267.10	428.80
p_hat700-1	700	0.25	11	16.08	12.53	40.32	0.67
p_hat1000-1	1000	0.24	10	97.38		283.20	5.84
san200_0.7_1	200	0.70	30	1.69		0.92	0.56
san200_0.7_2	200	0.70	18	4.91		0.47	0.04
san200_0.9_1	200	0.90	70	0.26		11.48	0.27
san200_0.9_2	200	0.90	60	12.27		1052.00	4.28
san400_0.5_1	400	0.50	13	5.83		11.22	0.03
san400_0.7_1	400	0.70	40	190.59		198.70	$>10{,}000.00$
san400_0.7_2	400	0.70	30	347.19		6228.00	506.10
san1000	1000	0.50	15	365.26		653.90	0.51
sanr200_0.7	200	0.70	18	52.91		338.20	14.11
sanr400_0.5	400	0.50	13	105.43		350.90	6.62

[a]DEC alpha 300–400 AXP.
[b]Sun Sparcstation 10.

For some instances, such as those of type hamming and c-fat, the new algorithm is fast without reordering of the vertices. For those of type hamming this can be explained as follows: A graph from the Hamming space—where the vertices are in lexicographic order and vertices whose Hamming distance is at least d, for some fixed d, are adjacent—have the property that there is a strong correlation between the difference between the indexes of two vertices and the "probability" that these are adjacent (cf. earlier example).

As for the instance `c-fat500-5`, the speed-up is remarkable if the vertices are not reordered: the CPU time needed is then 0.02, an improvement on the value in Table 3 by a factor of more than 10^5. This, of course, shows that for some instances the algorithm is very sensitive. More importantly, this brings other questions into daylight, which concern all published algorithms. When benchmarks are published, are the algorithms tuned for each instance? In this work, no such polishing of the results was carried out but the same algorithm was used for all instances.

Since the vertex ordering is of importance, the ordering in the benchmark file may affect the results. To prevent this, we feel that one should shuffle the vertices of these benchmark graphs before they are used as input. This was done here.

4. Conclusions

We have presented a new algorithm for finding a maximum clique in an arbitrary graph. The algorithm improves on old methods for several types of graphs, including sparse, random graphs and graphs with certain combinatorial properties. Since we are dealing with an NP-hard problem, it is expected that different algorithms perform well for different types of instances. One may then argue which instances are the most important ones. The speed of the algorithm presented here was of great importance in the author's work on error-correcting codes [8], where tens of thousands of maximum clique problems had to be solved.

The algorithm is fairly easily programmed. In our experiments, we merely added and edited some 10 lines of the program [1] (so, due to copyright reasons, the new C code is not made available). A more general algorithm, for the maximum-weight clique problem, is available at <URL:http://www.tcs.hut.fi/~pat/wclique.html>. It is still to be explored whether particular orderings of the vertices may give further improvements on the new approach.

Acknowledgements

The author wishes to thank Harri Haanpää and the referees for helpful suggestions.

References

[1] D. Applegate, D.S. Johnson, dfmax.c [C program; online], available at <URL:ftp://dimacs.rutgers.edu/pub/challenge/graph/solvers/>.
[2] L. Babel, Finding maximum cliques in arbitrary and in special graphs, Computing 46 (1991) 321–341.
[3] E. Balas, J. Xue, Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring, Algorithmica 15 (1996) 397–412.
[4] N. Biggs, Some heuristics for graph coloring, in: R. Nelson, R.J. Wilson (Eds.), Graph Colourings, Longman, New York, 1990, pp. 87–96.
[5] R. Carraghan, P.M. Pardalos, An exact algorithm for the maximum clique problem, Oper. Res. Lett. 9 (1990) 375–382.

[6] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman, New York, 1979.
[7] J. Hasselberg, P.M. Pardalos, G. Vairaktarakis, Test case generators and computational results for the maximum clique problem, J. Global Optim. 3 (1993) 463–482.
[8] P.R.J. Östergå rd, T. Baicheva, E. Kolev, Optimal binary one-error-correcting codes of length 10 have 72 codewords, IEEE Trans. Inform. Theory 45 (1999) 1229–1231.
[9] P.M. Pardalos, J. Xue, The maximum clique problem, J. Glob. Optim. 4 (1994) 301–328.
[10] E.C. Sewell, A branch and bound algorithm for the stability number of a sparse graph, INFORMS J. Comput. 10 (1998) 438–447.
[11] D.R. Wood, An algorithm for finding a maximum clique in a graph, Oper. Res. Lett. 21 (1997) 211–217.