A PROJECT REPORT ON

Algorithm for finding cliques

Submitted to:
Dr. Subhash C. Nandi
Applied Computers and
Microelectronics Unit
Indian Statistical Institute - Kolkata

by

TALE PRAFULLKUMAR P.
Inte. M.Sc. in Applied Mathematics
Indian Institute of Technology-Roorkee

ACKNOWLEDGEMENTS

I am heartily thankful to my guide, Dr. Subhash C. Nandy, whose encouragement, guidance and support enabled me to develop an understanding of the subject and made it possible for me to complete this project.

I offer my regards to the Dean ,Indian Statistical Institute - Kolkata and other concerned Professors and staff for allowing me to stay at campus and to use other facilities during project duration.

Tale Prafullkumar P.

Contents

1 INTRODUCTION

In an undirected graph G, a *clique* is a complete subgraph of G, i.e., a subgraph in which any two vertices are adjacent.The set of vertices of a maximal clique of the complementary graph of G is a maximal independent set of G. Generating maximal cliques or maximal independent sets of a given graph is one of the fundamental problems in the theory of graphs, and such a generation has many diverse applications.A number of algorithms have been presented and evaluated experimentally or theoretically for this problem.

In this project I worked on a depth-first search algorithm for generating all the maximal cliques of an undirected graph, in which pruning methods are employed by reducing the possible set of *candidate vertices.* All the maximal cliques generated as output.In the mathematical analysis of this algorithm we will find that its worst-case running time complexity is $O(3^{n/3})$ for a graph with n vertices.

In some practical application,we don't need all the cliques in the graph but only the maximum cliques.This can be done without actually finding all cliques.Elaborate *coloring* can significantly reduce the search space. However, coloring is time-consuming, and therefore, it becomes important to choose an appropriate trade-off between the time required for approximate coloring and the reduction in the search space thereby obtained. Ones the maximum cliques can be found,we get maximum independent set of the complement of graph. And this is a major challenge in the study of mobile ad-hoc networks. Mobile ad-hoc networks are the next generation in communication networks. Devices can enter or leave the network anytime, devices can be mobile, and no centralized control points or base stations are necessary. This flexibility makes mobile ad-hoc networks very interesting for the consumer and military market, but also makes them more complex than many existing wireless networks (such as GSM). The distinct properties of mobile ad-hoc networks have given rise to various new problems and challenges. These can be very practical and theoretical. In this report, I concentrated on the more theoretical aspects of mobile ad-hoc networks and a graph model for such networks.The important question in this type of study is of finding the *maximum independent set* of a graph and we will address to this question.A mobile ad-hoc network can be naturally modeled as a so called *(unit) disk graph.* Each node in such a graph has a disk around it containing all points reachable by that node. The intersections of these disks then determine the edges of the graph. Before going for further discussion we should first see the organisation of this project report.

2 ORGANISATION

This report is organised as follows. We will define the basic definitions which will create our mathematical backgroud for the discussion of the topic. This section is followed by the preliminaries in which we will define some terms and symbols which will be used for explaining algorithm and during coding. In the following section we will develop the algorithm and will improve it . Following it we will see the implementation of code in the python and testing of the code by generating random graphs.This project report ends with references for further reading and describing how to get additional modules in the python.

3 DEFINITIONS

3.1 Cliques

A clique in an undirected graph G = (V,E) is a subset of the vertex set C $\subseteq$ V, such that for every two vertices in C, there exists an edge connecting the two. This is equivalent to saying that the subgraph induced by C is complete (in some cases, the term clique may also refer to the subgraph).

A maximal clique is a clique that cannot be extended by including one more adjacent vertex, that is, a clique which does not exist exclusively within the vertex set of a larger clique.

3.2 Maximal Cliques

A maximum clique is a clique of the largest possible size in a given graph. The clique number ω(G) of a graph G is the number of vertices in the largest clique in G.

3.3 Neighbourhood

An adjacent vertex of a vertex v in a graph is a vertex that is connected to v by an edge. The neighbourhood of a vertex v in a graph G is the induced subgraph of G consisting of all vertices adjacent to v and all edges connecting two such vertices. Γ(p) represent the neighbourhood of p.

3.4 Coloring of Graph

Graph coloring is an assignment of labels traditionally called ‘colors’ to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a vertex coloring.Let graph is defined as G = (V,E) than we assign in advance for each $p \in I^+$a positive integer value $N_0[p]$ called the Number or Color of p with the following property:

1. If (p,r) $\in$ E than $N_0[p] \neq N_0[r]$ and
2. $N_0[p] = 1$ or $N_0[p] = k > 1$then there exist$p_1 \in \Gamma(p), p_2 \in \Gamma(p), \ldots, p_{k-1} \in \Gamma(p)$ such that $N_0[p_1] = 1, N_0[p_2] = 2, \ldots, N_0[p_{k-1}] = k - 1$

3.5 Unit Disk Graphs

Let S be a set of geometric objects. Then the graph G = (V, E), where each vertex corresponds to an object in S and two vertices are connected by an edge if and only if the two corresponding objects intersect, is called an intersection graph. The graph G is said to be realized by S.

A graph G is a disk graph if and only if there exists a set of disks D =D_i | i = 1. . . n, such that G is the intersection graph of D. The set of disks is called a disk representation of G.

A graph G is a unit disk graph if and only if G is a disk graph and the radii of a set of disks realizing G are equal.

3.6 Complement of Graph

The complement or inverse of a graph G is a graph H on the same vertices such that two vertices of H are adjacent if and only if they are not adjacent in G. In formal words,let G = (V, E) be a simple graph and let K consist of all 2-element subsets of V. Then H = (V, K\ E) is the complement of G.

3.7 Independent Set

Independent set or stable set is a set of vertices in a graph, no two of which are adjacent. That is, it is a set I of vertices such that for every two vertices in I, there is no edge connecting the two.

A maximal independent set is an independent set such that adding any other vertex to the set forces the set to contain an edge.

3.8 Maximum Independent Set

A maximum independent set is a largest independent set for a given graph G and its size is denoted α(G). The problem of finding such a set is called the maximum independent set problem and is an NP-hard optimization problem.As such, it is unlikely that an efficient algorithm for finding a maximum independent set of a graph exists.

A set is independent if and only if it is a clique in the graphs complement, so the two concepts are complementary. Hence finding the cliques of a graph is equivalent to finding the maximum independent set in the complement graph. And hence when we need the maximum independent set in unit disk graph, we take compliment of graph and find the largest maximal clique in it.

4 PRELIMINARIES

1. During the algorithm and coding we are concerned with a simple undirected graph G = (V,E) with a finite set V of vertices and a finite set E of unordered pairs (v,w) of distinct vertices, called edges. A pair of vertices v and w are said to be adjacent if (v,w)$\in$E. During the coding, the program will ask for .xls files in which adjacent matrix is being stored. This is convenient for handling large graphs.

2. For a vertex v $\in$ V , let Γ(v) be the set of all vertices that are adjacent to v in G=(V,E), i.e., Γ(v) = w $\in$ V |(v, w)$\in$ E. During the coding we will be using T_p for the Γ(p).

3. SUBG stands for sub graph of G and at initially it is equated to V. This SUBG is divided into two mutually exclusive part but collectively exhaustive subset names CAND for candidates and FINI for finished

5 ALGORITHM AND ITS DEVELOPMENT

5.1 Basic Algorithm

We consider a depth-first search algorithm for generating all the maximal cliques of a given graph G = (V , E) (V $\neq \phi$).Here, we introduce a global variable Q of a set of vertices that constitutes a complete subgraph found up to this time. The algorithm begins by letting Q be an empty set, and expands Q step by step by applying a recursive procedure EXPAND to V and its succeeding induced subgraphs to search for larger and larger complete subgraphs until they reach maximal ones.

Let Q = $\{p_1, p_2, \ldots, p_n\}$ be a complete subgraph found at some stage, and consider the set of vertices SUBG = V$\cap\Gamma(p_1) \cap \Gamma(p_2) \cap \ldots \cap \Gamma(p_n)$.

Only one element of this set can be added to the complete graph found so far. Hence we pick one element from SUBG and push rest of the element in the stack so that we can pop them again and operate on them. Initially Q is equated to null and SUBG = V.Apply the procedure EXPAND to SUBG to search for larger complete subgraphs. If SUBG = $\emptyset$ then Q is clearly a *maximal complete subgraph*, or a *maximal clique*. Otherwise, Q$\cup$ {q} is a larger complete subgraph for every q$\in$ SUBG. Now, consider the smaller subgraphs G($SUBG_q$) that are induced by new sets of vertices.i.e. $SUBG_q = SUBG \cap \Gamma(q)$. Suppose we are working with a stack 's'.The basic algorithm can be stated as follows:

```
1. //Initialisation of stack
   for all p in V :
   SUBG = $V \cap \Gamma(p)$
   Q = $\emptyset$
   s.push(p,Q,SUBG)

2. while Stack is not empty:
   s.pop() //to get a vertex p,Q and SUBG
   Q = EXPAND$\{p, Q, SUBG\}$ //Q is clique formed
   CL = CL+Q //CL is the global variable holding all the cliques

3. EXPAND$\{p, Q, SUBG\}$
   Q = $Q \cup p$
   while SUBG $\neq \emptyset$
   q = Select a point form SUBG
   s.pop( rest of the points with Q,SUBG)
   return Q
```

This process can be represented by the following search forest, or the collection of search trees: the set of roots of the search forest is exactly the same as V of graph G = (V,E). For each q $\in$ SUBG, all the vertices in $SUBG_q$ (defined above) are children of q. Thus, a set of vertices along a path from the root to any vertex of the search forest constitutes a complete subgraph or a clique.

5.2 Development of Algorithm

In the above section we have seen only a basic framework of the algorithm for generating all the maximal cliques. But this methods leads to duplication of cliques which causes the wastage of time and space.Now in this section we will see two methods to prune unnecessary parts of the search forest. We regard the previously described set SUBG (=$\emptyset$) as an ordered set of vertices,and we continue to generate maximal cliques from the vertices in SUBG stepwise in this order. First, let FINI be a subset of vertices of SUBG that have already been processed by the algorithm (FINI is short for finished). We then denote the set of remaining candidates for expansion by CAND: CAND = SUBG FINI. Hence,we have

$$\text{SUBG} = FINI \cup CAND \qquad (FINI \cap CAND = \emptyset)$$

FINI = $\emptyset$ at the beginning.Consider the subgraph G($SUBG_q$)with $SUBG_q$ as defined above, and let

$$\text{SUBGq} = FINI_q \cup CAND_q \qquad (FINI_q \cap CAND_q = \emptyset),$$

where

$$FINI_q = FINI \cap \Gamma(q) \qquad CAND_q = CAND \cap \Gamma(q).$$

Following this, only the vertices in $CAND_q$ can be candidates for expanding the complete subgraph $Q \cup \{q\}$ to find new larger cliques, since all the cliques containing $(Q \cup \{q\}) \cup \{r\}$ with $r \in FINI_q \subseteq FINI$ have already been generated for any r by application of the procedure EXPAND to FINI as stated above.

Secondly, given a certain vertex $u \in SUBG$, consider that all the maximal cliques containing $Q \cup \{u\}$ have been generated. Then, every new maximal clique containing Q, but not $Q \cup \{u\}$, must contain at least one vertex $q \in SUB - \Gamma(u)$. This is because if Q is expanded to a complete subgraph $R = (Q \cup S) \cap (SUBG - \{u\})$ with $S \subseteq SUBG \cap \Gamma(u)$, then $R \cup \{u\}$ is a larger complete subgraph, and hence R is not maximal. Thus, any new maximal clique can be found by expanding Q to$Q \cup \{q\}$ such that q$\in SUBG - \Gamma(u)$, and by subsequently generating all the cliques containing $Q \cup \{q\}$.

Taking the previously described pruning method also into consideration, the only search subtrees to be expanded are from the vertices in $(SUBG - SUBG \cap \Gamma(u) - FINI = CAND - \Gamma(u)$. Here, in order to minimize $|CAND - \Gamma(u)|$, we choose a vertex $u \in SUBG$ as the one that maximizes $|CAND \cup \Gamma(u)|$.In this manner, the problem of generating all maximal cliques of G(CAND) can be decomposed into k = $|CAND - \Gamma(u)|$ such subproblems.

In the next part, we will see the python coding of the algorithm described above.

6 CODING IN PYTHON

6.1 Building blocks of the program

In this section we will build up a program to find all the maximal cliques from a given graph. To handle large graphs we will take input argument as an Adjacency matrix in the form of xls file.

```
from numpy import *
import xlrd

r = raw_input('Pls enter the name of sheet in which adjcent matrix
is being stored \n ')
wb = xlrd.open_workbook(r)#To get workbook
sh = wb.sheet_by_index(0)#To get first sheet

Adj_mat = []#Initialisation
V = []

for rn in range(sh.nrows):# creating Adj_mat and V
    Adj_mat = Adj_mat + [sh.row_values(rn)]
    V.append(rn+1)
```

Please see references for installing and getting the required modules in python.

With this we can get adjacent matrix store in variable Adj_mat and ordered set of vertices in V. These two are used as a global variable and remains unaltered till the end of program.

We will be needing some functions like Intersection of two sets, Subtraction of two sets,coloring,etc. These all functions are defined separately and all are embedded in the module named my_module.

```
#! usr/bin/python

# This script file is used to create a module which will contain all the
required function
# Adj_mat i.e. Adjecency matrix is supposed to be defined in the main
program and here that is taken as input argument.
# The functions are :
#   1) T = Adj_ver(p) returns adjecent vertices of point p
```

```
    #   2) C = Interseaction(A,B) returns the intersection of two sets A and
B
    #   3) C = Set_sub(A,B) subtract set B from set A
    #   4) A = Coloring(R,Adj_mat) To color the given subset R of a graph
represented by Adj_mat

    def Adj_ver(p,Adj_mat):
        T = [-1] # this is used to remove inconsistency in runtime
        Am = Adj_mat[p-1]
        for i in range (0,len(Adj_mat)):
            if (Am[i]==1):
                T = T + [i+1]
        del T[0]
        return T
    ##              End of function Adj_ver                 ##
    def Intersection(A,B):
        C = []
        for i in range (0,len(A)):
            for j in range (0,len(B)):
                if(A[i]==B[j]):
                    i1 = A[i]
                    C = C + [i1]
        return C
    ##              End of function Intersection                ##

    def Set_sub(A,B):
        C = A[:]
        i = 0;
        for j in range(0,len(B)):
            i = 0
            while i<len(C):
                if B[j]==C[i]:
                    del C[i]
                    if len(C)==0:
                        return C
                    i = i-1
                #end of it
                i=i+1
        return C
```

```
##              End of function Intersection             ##


def Coloring(R,Adj_mat):
    # R is a subset of graph represented by Adj_mat
    class Vertex(p):
        def ver(self,p):
            return p
        def deg(self,p,Adj_mat):
            T_p = Adj_ver(p,Adj_mat)
            return len(T_p)
        def adj_v(self,p,Adj_mat):
            T_p = Adj_ver(p,Adj_mat)
            return T_p
```

As we have stated earlier, it will be *depth first search* algorithm and hence we will be needing some stack to handle the tree.The definition of stack and other functions related to it are defined in the module named stack.This module is as follows:

```
#! usr/bin/python
# This is a module which will we used to handle tree structure.
# This function as two main functions
#   1) push(p,Q) pushes the given elements into stacts
#   2) pop() pops put the last element and return it
# and some other complimentery functions like
#   3) isempty() will return 1 is stack is empty


class Cla_stack:
    def __init__(self):   #This is a constructor
        self.Storage = []
    ##         End of constructor                    ##

    def isempty(self):
        if len(self.Storage)==0:
            return 1
        else:
            return 0
    ##          End of function isempty                ##


    def push(self,p,Q,SUBG):      # Storing a point and Q at that
time
        self.Storage = self.Storage + [[[p],[Q],[SUBG]]]

    ##          End of function push                ##


    def pop(self):
        p_Q_SUBG = []
        if self.isempty()==0:
            p_Q_SUBG = self.Storage[len(self.Storage)-1]
        del self.Storage[len(self.Storage)-1]
        return p_Q_SUBG

     ##          End of function pop                ##
```

6.2 Main program

Ones these building blocks are formed we can now turned on to main program. The main program can be divided into 3 parts. The main part is an EXPAND function. This function takes four input argument viz the vertex p, the clique formed so far Q,set of all candidate vertices CAND and a stack s to push the elements in. In this program a point p is selected using select function from CAND. Now all the points which lies in the neighbourhood of p and p are removed from the CAND. Rest of the points are pushed into stack and this while loop proceeds till there is no point left in the set of candidate vertices.

```
def Expand(p,Q,CAND,s):
    Q = Q + [p]
    #p is a point,Q is a clique formed so for,CAND is canditate set(replacing
SUBG) at that time and s is stack used
    while len(CAND)!=0:
        p1 = Select(CAND,Adj_mat)
        T_p1 = mm.Adj_ver(p1,Adj_mat)
        CAND_t = CAND[:]
```

```
                CAND_t = mm.Set_sub(CAND_t,[p1])
                CAND_t = mm.Set_sub(CAND_t,T_p1)
                while len(CAND_t)!= 0:
                    p2 = Select(CAND_t,Adj_mat)
                    T_p2 = mm.Adj_ver(p2,Adj_mat)
                    CAND_n = mm.Intersection(CAND,T_p2)
                    s.push(p2,Q,CAND_n)
                    CAND_t = mm.Set_sub(CAND_t,[p2])
                    CAND_t = mm.Set_sub(CAND_t,T_p2)

                CAND = mm.Intersection(CAND,T_p1)
                Q = Q + [p1]
            return Q
##          End of function Exapand          ##
```

In the above code we used the function Select. This function selects a point p from the CAND in such a way that the node will have the least number of children.This function takes two argument one in CAND and another is Adj_mat.It finds and compares the intersection of CAND and neighbourhood of all the points in CAND and returns the one for which it is maximum.

```
def Select(CAND,Adj_mat):
    size_i1 = -1 #size of intersection
    if len(CAND)==1:
        p = CAND[0]
        return p
    p = -1
    for i in range (0,len(CAND)):
        T_p = mm.Adj_ver(CAND[i],Adj_mat)
        if len(T_p)> size_i1:
            Inte = mm.Intersection(T_p,CAND)
            if len(Inte)>size_i1:
                p = CAND[i]
                size_i1 = len(Inte)
    return p

##          End of function Select          ##
```

Now we are only left with the main program which uses above two programs to give desirable result. The main program consist of initialising the stack and it has a *while* loop which runs as long as stack is not empty. Basically we are removing elements from the stack in the main program and in

EXPAND we are adding elements to the stack. When the removing overcomes the addition of elements or the when there is no more elements to add than program gets terminated and it display all the cliques present in the graph without any repetition.

```
Cliques = []#The final ans will be stored in this
s = st.Cla_stack()#Defining the stack
V_temp = V[:]

while len(V_temp)!=0:
        p = Select(V_temp,Adj_mat)
        Q = []
        T_p = mm.Adj_ver(p,Adj_mat)
        s.push(p,Q,T_p)
        V_temp = mm.Set_sub(V_temp,[p])
        V_temp = mm.Set_sub(V_temp,T_p)


while s.isempty()!= 1:
        p_Q_CAND = s.pop()
        p = p_Q_CAND[0]
        p = p[0]#converting list into int
        Q = p_Q_CAND[1]
        Q = Q[0]#converting item into list
        CAND = p_Q_CAND[2]
        CAND = CAND[0]
        C = Expand(p,Q,CAND,s)
        Cliques = Cliques + [C]


for i in range (0,len(Cliques)):#Displaying output
     print Cliques[i]

print len(Cliques)

##          End of main program            ##
```

7 FINDING MAXIMUM CLIQUE

Most of the time the question is not about finding all the cliques but its only about a finding the maximum clique in the graph. This can be done by exhaustive search for all cliques but it is very time and space consuming Elaborate *coloring* can significantly reduce the search space. However, coloring is also time-consuming, and therefore, it becomes important to choose an appropriate trade-off between the time required for approximate coloring and the reduction in the search space thereby obtained. Many efforts have been made along this line.

7.1 Algorithm for Coloring of Graph

As we have already defined coloring of graph, in this section we will see how to assign different colors i.e. N_0 to vertices. The value $N_0[p]$ for every $p \in R$ can be easily assigned step by step by a so-called *greedy coloring algorithm* as follows: Assume the vertices in R = $\{p_1, p_2, \ldots, p_m\}$ are arranged in this order. First, let $N_0[p_1] = 1$.Next,let $N_0[p_2] = 2$ if $p_2 \in \Gamma(p1)$, else $N_0[p1] = 1, \ldots$, and so on. After Numbers are assigned to all vertices in R, we sort these vertices in ascending order with respect to their Numbers.

In the coding,it is convenient to represent vertex as a *class* rather than representing it as just a point. We define a *class* Vertex(). This has four attributes namely ver for vertex,adj$_v forthesetofalltheadjacentverticestotheconcernedvertex.degf$

```
#! usr/bin/python
# This program is used as a module to color the given graph.

import my_module as mm
from numpy import *

def Coloring(R,Adj_mat):
    # R is a subset of graph represented by Adj_mat
    class Vertex():#Defining the class
        def __init__(self,p,Adj_mat,c=1):
            self.p = p
            self.Adj_mat = Adj_mat
            self.c = c

        def ver(self):
            return self.p
        def adj_v(self):
            T_p = mm.Adj_ver(self.p,self.Adj_mat)
            return T_p
        def deg(self):
            return len(self.adj_v())
        def color(self):
            return self.c
        def set_color(self,c):
            self.c=c
            return c

    V_cl=[]
    for i in range(0,len(Adj_mat)):
        V = Vertex(i+1,Adj_mat)
        V_cl = V_cl + [V]

    V_sorted = sorted(V_cl,key=lambda Vertex:Vertex.deg(),reverse=True)
    V_only = []# To stored sorted vertices only
    for i in range(0,len(V_sorted)):
        V_only = V_only + [V_sorted[i].ver()]

    for i in range(1,len(V_sorted)):#will start from 1 not 0
        v1 = V_sorted[i] #index of v1
        In = mm.Intersection(v1.adj_v(),V_only[0:i])
        Co_used = []
        for i in range(0,len(In)):
              v2 = V_cl[In[i]-1]
              Co_used = Co_used + [v2.color()]

        c1 = 1
        while c1 in Co_used != True:
              c1 = c1+1
        v1.set_color(c1)

    V_colors = []

    for i in range (0,len(V_sorted)):
          v1 = V_sorted[i]
          V_colors = V_colors +[[v1.ver(),v1.color()]]

    return V_colors
```

## End of program ##

8 REFERENCES

1. 'The worst-case time complexity for generating all maximal cliques and computational experiments' a paper publicshed by Etsuji Tomitaa, Akira Tanakaa, Haruhisa Takahashia.

2. 'An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments' paper published by Etsuji Tomita,Toshikatsu Kameda.

3. Other supplementary papers from www.sciencedirect.com

4. 'Approximation Algorithms for Unit Disk Graphs' technical report by Erik Jan van Leeuwen institute of information and computing sciences, utrecht university.

5. Python modules :

   - http://www.lexicon.net/sjmachin/xlrd.htm
     This module is used to take data from xls sheets.
   - http://www.numpy.scipy.org/ This module is used to perform operations on lists which is a common form of handling the set.