Simple Primality Testing Algorithms for Undirected, Connected graphs

Li Kwing Hei

31st August, 2018

Diocesan Boys' School
Teacher: Mr Chan, Long Tin
Registration Number: 180059
Statement: This is a report submitted to the Scientific Committee of Hang Lung Mathematics Award, 2018.

Abstract

In graph theory, a module is a set of vertices where all members have the same set of neighbors among vertices not in the set. A graph is prime if all its modules are trivial. This report aims to propose an algorithm that effectively determines whether a graph is prime or not. The algorithm should be easy to understand and implement. We discuss the time complexity and nature of each algorithm proposed and analyze the probability of success for each. After combining two randomized algorithms, we find an effective, deterministic algorithm that runs in $O(n^2)$.

1 Where the story all begins...

The first time I was introduced to algorithms in graph theory was when I started participating in computer programming contests(Olympiad in Informatics). As I began solving graph problems, I gradually became more interested in graph algorithms as well. But slowly, I realized that purely participating in computer programming contests could not satisfy my curiosity in learning about graph algorithms. I was merely taught to solve problems with a known solution. That was why I was motivated to take the initiative to investigate on unsolved graph problems and "invent" certain algorithms on my own.

In this report, I focused a lot on randomized algorithms. I was fascinated by how effective these algorithms are despite being so unpredictable. The first time I learnt about randomized algorithms was when I attended lectures provided by professors in the University of Waterloo while participating in the Canadian Computing Olympiad. One of the lectures was about randomized algorithms (it was taught by a Hong Kong professor coincidentally). When he talked about Karger's algorithm[1] or other randomized algorithms, I was completely shocked by how so simple yet so effective the algorithms were. To me, it was remarkable how something so simple to understand could tackle a problem so complicated. Therefore, I decided to start tackling my problem with randomized algorithms that are also simple and easy to understand in nature...

2 Introduction

In this report, we aim to find an algorithm that could determine whether a connected undirected graph is prime or not. The algorithm must be simple to understand and easy to implement. Below are definitions of some terms that we will use very often.

Unless otherwise specified, G refers to a graph, $V(G)$ and $E(G)$ are the vertex set and the edge set of G.

Definition. $S \subset G$ is a *module* if

$$\forall u, v \in S, \{t \notin S | ut \in E(G)\} = \{t \notin S | vt \in E(G)\}$$

In other words, a module of a graph is a set of vertices that have same set of neighbors which are not contained in the set.

The set of all vertices, the empty set and all one-element subset of a graph are called trivial modules. For the rest of the paper, unless otherwise specified, the term modules does not include trivial modules.

A graph is *prime* if it contains no other modules other than its trivial modules. Similarly, a graph is non-prime if it contains modules that are not trivial.

For the rest of the report, all graphs are connected and undirected. In other words, all vertices are reachable from each other.

This is because if the graph is not connected, we can directly tell the graph is not prime as the set of vertices of a connected component is a module. (The vertices of a connected component cannot have neighbors outside of the component, otherwise there will be a path connecting this connected component to another.)

2.1 Modular Decomposition

When we talk about modules, it is natural to ask a question: Is it possible to 'decompose' a graph into union of modules? The answer is 'yes'.

The modular decomposition of a graph is the partitioning of vertices into sets, in which every set is a module. Currently, there are many algorithms that find that the modular decomposition of the graph. There is even one that runs in linear time $O(n+m)$[2], where n, m are the order and the size of a graph. However, due to the complicated nature of the data structures involved, it is rarely implemented in practice. Besides, the focus of this paper is to purely determine whether a graph is prime or not. We do not intend to find its modular decomposition.

3 Naive solution

We shall now propose the simplest solution to tackle our problem. Though ineffective, this algorithm is very straightforward and intuitive to most people. The algorithm works by naively brute-forcing all possible sets of vertices. We then check whether the set is a module by checking what vertices each member of the set is connected with. There are two possibilities:

1. If none of the sets are modules, then the graph is a prime graph.

2. If at least one of the sets is a module, then the graph is a non-prime graph.

There are 2^N possible sets. However only $2^N - N - 2$ sets should be considered as the empty set, sets containing one vertex and the set containing all vertices are all trivial modules. Checking the neighbors of one vertex takes $O(N)$. By considering the worst case scenario, the size of the largest set is $N-1$, therefore the total time complexity is $O(2^N \cdot N^2)$. Obviously, this is very slow. However, we have two ways to optimize this algorithm:

1. Terminate the algorithm when we find one module. In this case we can directly determine the graph to be non-prime. However, this will not affect the time complexity.

2. For the sets we exhaust, we only need to consider those that have vertices that have a maximum distance length of 2 from each other. If there exists a pair of vertices that has a distance length larger than two from each other, we can directly tell the set cannot be a module without comparing their neighbors.

Lemma. For $S \subset G$, if $\exists u, v \in S$ s.t. distance between u, v is larger than 2, then S is not a module.

Proof. Note that since we do not consider trivial modules, we may assume $S \neq G$. If $\exists u, v \in S$ s.t. distance between u, v is larger than 2, then u, v have no neighbour outside S. Hence S is disconnected from $G \setminus S$, which contradicts connectedness of G. □

However in the worst case scenario, we can tell that this does not improve the time complexity.

Even though this algorithm is very slow and ineffective, the general idea behind it is very crucial. By using a method to generate some sets of vertices, if we are able to prove that a set is a module, then we can tell that the graph is a non-prime graph immediately.

In this report, we first propose two randomized algorithms that could generate sets that could possibly be modules faster. For each algorithm we propose, we shall first see how the algorithm works, illustrated by an example. Next, we will prove why the algorithm works. We will also determine the efficiency of the algorithm by looking into the time complexity of the algorithm, and the probability of getting a correct answer by running the algorithm.

4 Algorithm 1

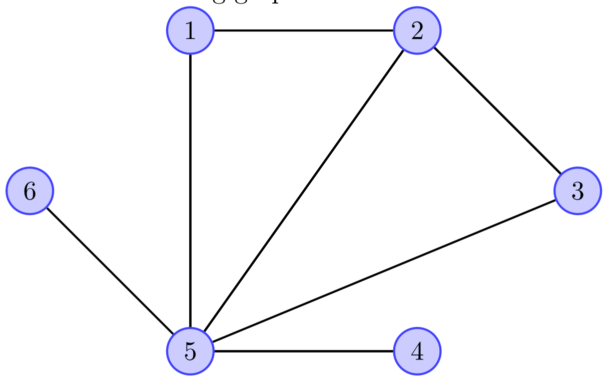
The idea of our naive algorithm is very important. If we have some very effective methods in exhausting different combination of sets of modules and checking whether they are modules or not, we can immediately determine that the graph is not prime if we can find one module.

4.1 Description of algorithm 1

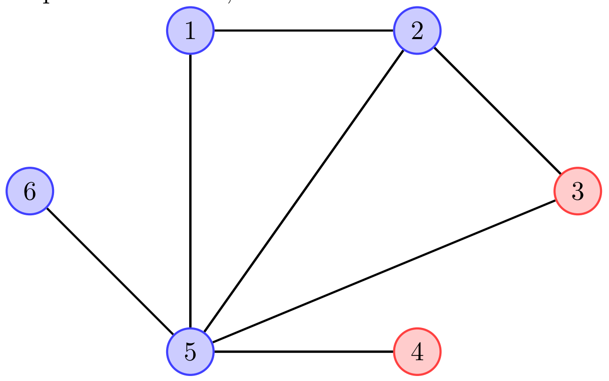
1. Randomly choose 2 vertices and add them to set S
2. If there exists a vertex u not in S that is a neighbor to at least one vertex in S and not a neighbor to at least one vertex in S , add u to S and repeat step 2
3. If the size of S is smaller than the size of G, then S is a module and G is not prime. Otherwise we accept that G is prime.

4.2 Example

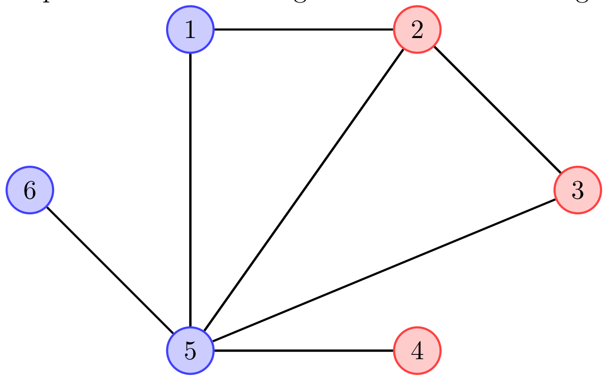
Consider the following graph:



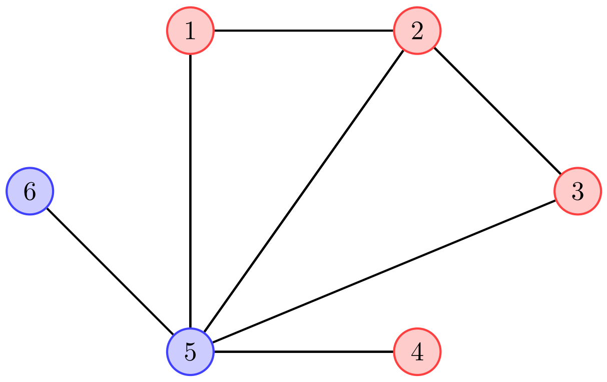
Step 1: At random, we choose vertices 3 and 4. We add them to set S



Step 2.1: Since 2 is a neighbor to 3 but not a neighbor to 4, we add 2 to S



Step 2.2: Since 1 is a neighbor to 2 but not a neighbor to 3 and 4, we add 1 to S



Step 3: Since the condition for step 2 cannot be satisfied, and size of S is smaller than n, the graph is not a prime graph. In fact S is a module.

4.3 Proof

Lemma. if there exists a module containing all vertices in S and a vertex u that satisfies the condition in step 2, then the module must also contain u

Proof. Assume there exists a module containing all vertices in S at any point during the execution of algorithm 1 and the module does not contain u. Then there is at least one vertex in the module connected to u and at least one other vertex that is not connected to u. This contradicts the definition of a module. $\square$

From the Lemma, we can see that when we cannot add any other vertex into S in Step 2, S is by definition a module. Hence the algorithm must work. Now what remains is whether this algorithm is indeed an efficient one.

4.4 Time Complexity

We can execute the algorithm by exhausting the neighbors of each member in S. For each vertex in S, we could find its neighbors not in S in $O(n)$. Since there are at most n members in S, the total time complexity is $O(n^2)$.

4.5 Analysis

This is a false-based Monte Carlo algorithm because if it determines the graph is not prime (false), it is always correct. If it determines the graph is prime, it may not be correct for there is a possibility that there is a module that exists in the graph that does not contain both the initial vertices chosen.

The algorithm always works if the graph is prime, since the algorithm will accept that the graph is prime if it is unable to find one module. Therefore, we only need to consider the probability of success of getting a correct answer with a non-prime graph.

Consider a graph with exactly one module, let the fraction of the size of the module to the size of the graph be x. The algorithm works if the two initial vertices chosen are contained in the module. Therefore, the probability of success is x^2.

Proposition. If the graph contains more than one module, the probability of success is larger or equal to x^2, where x is the fraction of the size of any module to the size of the graph

Proof. If there exist more modules, the number of possible pairs of vertices to choose from that could produce a correct answer cannot decrease. Therefore, the probability of success must not be smaller than x^2. However, the probability of success tends to zero in certain situations, especially if the size of the graph is large and there only exists one small module in the graph. For example, if there exists exactly one module of size 2, then the algorithm works if and only if we choose the two members in the module initially. $\square$

Remark. We can obtain a probability of success of 100% by repeating algorithm 1 $n \times (n-1)/2$ times. By brute-forcing every possible pair of vertices, we can ensure we can determine correctly whether the graph is prime or not. This is because if the graph is not prime, there must be a pair of vertices that is contained in at least one module. However this algorithm has time complexity of $O(n^4)$ which is very inefficient.

5 Algorithm 2

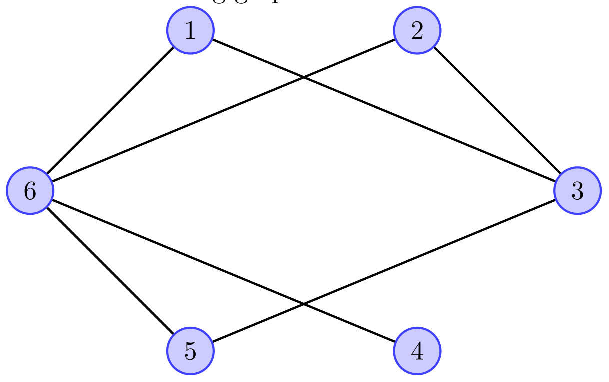
For algorithm 1, we first choose two vertices and assume there exists a module that contains the two vertices. We also find that the algorithm fails to work efficiently if the size of the modules is small. Therefore, we propose a second algorithm that works in the other way around (by assuming there exists a module not containing a certain vertex), and observe whether it will produce an different probability of success.

5.1 Description of algorithm 2

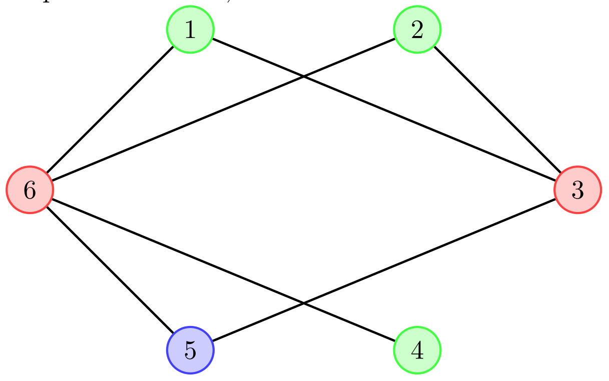
1. Randomly choose 1 vertex u
2. Split the other vertices into two groups, one containing neighbors of u and the other containing the rest. (From now on, we remove u from the graph)
3. If there exists a group that does not contain a vertex v, and the group contains at least one vertex that is a neighbor of v and at least one vertex that is not a neighbor of v, split the group into two groups, one containing neighbors of v and the other containing the rest
4. If there exists one group of size larger than 1, the group is a module and G is not prime. Otherwise, we accept that G is prime.

5.2 Example

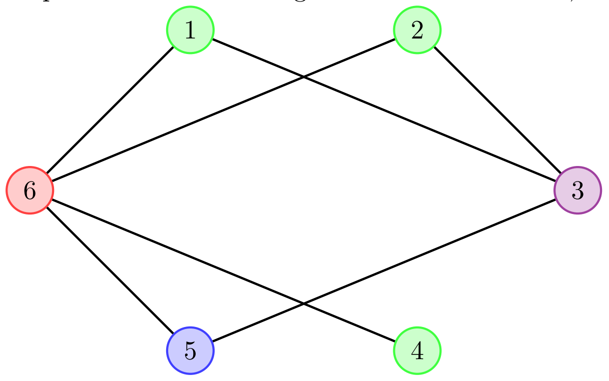
Consider the following graph:



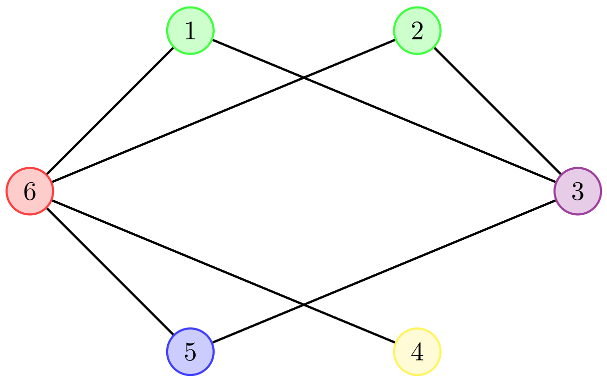
Step 1: At random, we choose vertex 5. Other nodes are split into 2 groups



Step 2.1: Since 4 is a neighbor to 6 and not to 3, the group $(3, 6)$ is split



Step 2.2: Since 3 is a neighbor to 1 and 2 and not to 4, the group $(1, 2, 4)$ is split



Step 3: Since the condition in step 2 cannot be satisfied, and there is one group with size larger than 1, the graph is not a prime graph. In fact group $(1, 2)$ is a module.

5.3 Proof

Lemma. There cannot be a module not containing u that contains vertices that belong to different groups at the end of algorithm 2.

Proof. Starting from the first splitting step (step 2), if there exists a module that does not contain u, we can obviously tell that the module cannot contain vertices from both groups. The module must exist within each group. Otherwise some of the vertices in the module will be a neighbor to u while some are not, which is a contradiction.

Then for the other splitting steps, the same rationale applies. A module cannot exist that contain vertices from the two split groups. Otherwise some of the vertices in the module will be a neighbor to v while some are not, which contradicts the definition of a module. □

From the Lemma, we can see that when we cannot split any group into smaller ones in Step 2, each group is a by definition a module itself. Hence the algorithm must work. Now what remains is whether this algorithm is indeed an efficient one.

5.4 Time complexity

We can implement this algorithm in a divide-and-conquer manner, since for each splitting step, we "divide" the groups of vertices into smaller groups. In fact, through this recursive manner, we only need to consider each pair of vertices twice only by checking whether an edge exists between them. Therefore the time complexity is $O(n^2)$.

5.5 Analysis

This is a false-based Monte Carlo algorithm because if it determines the graph is not prime (false), it is always correct. If it determines the graph is prime, then it may not be correct for there is a possibility that there are no modules that does not contain u but some modules that contain u.

The algorithm always works if the graph is prime, since the algorithm will accept that the graph is prime if it is unable to find one module. Therefore, we only need to consider the probability of success of getting a correct answer with a non-prime graph.

Consider a graph with exactly one module, let the fraction of the size of the module to the size of the graph be x. The algorithm works if the initial vertex chosen is not contained in the module. Therefore, the probability of success is $(1 - x)$.

Proposition. If the graph contains more than one module, the probability of success is larger or equal to $(1 - x)$, where x is the fraction of the size of any module to the size of the graph

Proof. If there exist more modules, the algorithm only fails if we choose a vertex contained in the intersection of all the modules. Note that the size of the intersection of all the modules is smaller or equal to the size of any modules. There the probability of success must be larger than $(1 - x)$.

However, the probability of success tends to zero in certain situations, especially if the size of the graph and the size of the intersection of all modules is large. For example, if there exists exactly one module of size $n - 1$, then the algorithm works if and only if we choose the vertex not containing in the module.

$\square$

Remark. We can obtain a probability of success of 100% by repeating algorithm n times. By brute-forcing every vertex, we can ensure we can determine correctly whether the graph is prime or not. This is because if the graph is not prime, there must a vertex that is not contained in at least one module. However this algorithm has time complexity of $O(n^3)$ which is very inefficient.

6 Algorithm 3

Note that both algorithms 1 and 2 work and fail in almost opposite conditions. Then, we must consider whether combining both algorithms together would produce a better result.

6.1 Description of algorithm 3

1. Execute algorithm 1 and 2
2. If any of the algorithms determines that the graph is not prime, then the graph is not prime. Otherwise, we assume the graph is prime

6.2 Time complexity

Since both algorithm 1 and 2 operate in $O(n^2)$, then the total time complexity is $O(n^2)$.

6.3 Analysis

The algorithm always works if the graph is prime, since the algorithm will accept that the graph is prime if it is unable to find one module. Therefore, we only need to consider the probability of success of getting a correct answer with a non-prime graph.

Let x be the fraction of the size of any module to the size of the graph. Algorithm 3 fails if both algorithm 1 and 2 fails. Because the results of algorithm 1 and 2 are independent, we can directly calculate the probability of success of algorithm 3 by using the probability of success of algorithms 1 and 2.

Probability of failure for algorithm 1 $\leq -x^2 + 1$ Probability of failure for algorithm 2 $\leq x$

Probability of failure for algorithm 3 $\leq -x^3 + x$

Probability of success for algorithm 3 $\geq x^3 - x + 1$

The probability of success for algorithm 3 is larger or equal to $x^3 - x + 1$. We then compute the lower bound by differentiation:

$$\begin{aligned}
\text{Let } y &= x^3 - x + 1 \\
\frac{dy}{dx} &= 3x^2 - 1 \\
\text{Sub } \frac{dy}{dx} &= 0 \\
x^2 &= \frac{1}{3} \\
x &= \frac{\sqrt{3}}{3} or - \frac{\sqrt{3}}{3}_{\ (rej)} \\
\frac{d^2y}{dx^2}\Big|_{x=\frac{\sqrt{3}}{3}} &= 6(\frac{\sqrt{3}}{3}) \\
&> 0 \\
\text{Lower bound for probability of success} &= (\frac{\sqrt{3}}{3})^3 + \frac{\sqrt{3}}{3} \\
&= 0.615(3s.f.)
\end{aligned}$$

6.4 Notes

Note that this is already a relatively good probability for success for a randomized algorithm. In fact, we can always repeat algorithm 3 several times to obtain a more accurate result. Repeating algorithm 3 for 3 times will produce a probability for success of at least 94.3%. We should also be aware that the final time complexity is still $O(n^2)$ if we repeat the algorithm by a constant factor.

7 Algorithm 4

For a long time, I thought algorithm 3 was the best result I could achieve in this report. Nevertheless, I still kept on thinking whether I could improve algorithm 3 to get a more accurate result.

After around a month, I suddenly realized if we choose to execute algorithm 1 and 2 on the same vertices, we could get an amazing probability of success of 100%. In other words, even though I started my research in the direction of randomized algorithms, I still found an accurate algorithm that perfectly determines whether a graph is prime or not at the end.

7.1 Description of algorithm 4

1. Choose any two vertices u and v
2. Execute algorithm 1 starting with the pair of vertices u and v
3. Execute algorithm 2 twice, starting with vertex u and v respectively
4. If any of the algorithms determines that the graph is not prime, then the graph is not prime. Otherwise, we assume the graph is prime

7.2 Time complexity

Since both algorithms operate in $O(n^2)$, then the total time complexity is $O(n^2)$.

7.3 Analysis

Proposition. Algorithm 4 can always accurately determine whether a graph is prime or not

Proof. There are only a few cases to consider.

If the graph is prime, then there are no modules in the graph. Since both algorithms 1 and 2 are unable to find a module within the graph, we will accept that the graph is prime, which is a correct result in this case.

If the graph is not prime, then this allows two possibilities.

If there is at least one module containing both u and v, then algorithm 1 will determine this graph is not prime. This is because algorithm 1 is able to detect a module that contains both the vertices chosen.

Otherwise, there is at least one module not containing at least one vertices u and v. Without loss of generality, suppose there exists a module not containing u. Then when we execute algorithm 2 with u chosen initially, it could correctly determine that this graph is not prime. This is because algorithm 2 is able to detect a module that does not contain the vertex chosen.

Therefore, in all cases, algorithm 4 always produces an accurate result.

□

8 Conclusion

By cleverly combining two rather inaccurate randomized algorithms, we are able to produce one deterministic algorithm that always determines whether a graph is prime or not accurately.

Not only is this algorithm easy to understand and implement, it has a time complexity of $O(n^2)$, which arguably is as good as the best algorithm for modular decomposition today. (It has a time complexity of $O(n+m)$. In the worst case scenario where we are dealing with a dense graph, m will tends to n^2 and its time complexity will be close to $O(n^2)$.

9 Ideas for further research

Even though we have successfully found an effective algorithm for determining whether a graph is prime or not, there are still a few ideas for possible extension.

1. Could we find a simple algorithm that determines whether a graph is prime or not in $O(n+m)$?

2. Could we find a simpler algorithm for modular decomposition than the current one?

10 Codes

The following codes for C++ source codes of algorithm 1 and 2. They illustrate a way to practically implement both algorithms.

10.1 Algorithm 1

```
#include<bits/stdc++.h>
using namespace std;
int N,M;//N number of vertices, M number of edges
int a[1001][1001];//maximum size of graph is 1000 for this program
int visited[1001];
int cnt[1001];
int main(){
        ios::sync_with_stdio(false);
        srand(time(NULL));
        cin>>N>>M;
        for(int i=1;i<=N;i++){
                for(int j=1;j<=N;j++){
                        a[i][j]=0;
                }
                visited[i]=0;
                cnt[i]=0;
                }
        if(N==1){
                cout<<"Prime"<<endl;
                return 0;
        }
        for(int i=0;i<M;i++){
                int q,w;
                cin>>q>>w;
                a[q][w]=1;
                a[w][q]=1;
        }
        int start1,start2;
        start1=rand()%N+1;
        start2=start1;
        while(start2==start1){
                start2=rand()%N+1;
        }
        cout<<start1<<" and "<<start2<<" are chosen"<<endl;
        vector<int> now;
        queue<int> q;
        q.push(start1);
        q.push(start2);
        visited[start1]=1;
        visited[start2]=1;
        while(q.size()!=0){
                int u=q.front();
                q.pop();
                for(int i=1;i<=N;i++){
                        if(visited[i]==0&&a[u][i]==1){
                                cnt[i]++;
```

```
                    }
                }
                now.push_back(u);
                if(q.size()==0){
                    for(int i=1;i<=N;i++){
                        if(visited[i]==0&&cnt[i]>0&&cnt[i]<now.size()){
                            visited[i]=1;
                            q.push(i);
                        }
                    }
                }
        }
        if(now.size()==N){
                cout<<"Prime"<<endl;
                return 0;
        }else{
                cout<<"Not prime"<<endl;
                sort(now.begin(),now.end());
                for(int i=0;i<now.size();i++){
                    cout<<now[i]<<" ";
                }cout<<endl;
                return 0;
        }
}
```

10.2 Algorithm 2

```
#include<bits/stdc++.h>
using namespace std;
int N,M;//N number of vertices, M number of edges
int a[1001][1001];//maximum size of graph is 1000 for this program
int visited[1001];
int cnt[1001];
int check=0;
vector<int> module;
void func(vector<vector<int> >l1,vector<vector<int> >l2){
        if(check==1){
                return;
        }if(l1.size()==0){
                l1.push_back(l2[0]);
                l2.erase(l2.begin()+0);
                func(l1,l2);
                return;
        }
        if(l2.size()==0&&l1[0].size()>1){
                check=1;
                module=l1[0];
                return ;
        }if(l2.size()==0&&l1[0].size()==1){
                return ;
        }
        for(int i=0;i<l1[0].size();i++){
                for(int j=0;j<l2.size();j++){
                        vector<int> one,two;
                        for(int k=0;k<l2[j].size();k++){
                                if(a[l1[0][i]][l2[j][k]]==0){
                                        one.push_back(l2[j][k]);
                                }else{
                                        two.push_back(l2[j][k]);
                                }
                        }if(one.size()!=0&&two.size()!=0){
                                l2[j]=one;
                                l2.push_back(two);
                        }
                }
        }
        for(int c=0;c<l2.size();c++){
                for(int i=0;i<l2[c].size();i++){
                        for(int j=0;j<l1.size();j++){
                                vector<int> one,two;
                                for(int k=0;k<l1[j].size();k++){
                                        if(a[l2[c][i]][l1[j][k]]==0){
                                                one.push_back(l1[j][k]);
                                        }else{
                                                two.push_back(l1[j][k]);
                                        }
                                }if(one.size()!=0&&two.size()!=0){
                                        l1[j]=one;
                                        l1.push_back(two);
                                }
                        }
                }
        }
        vector<vector<int> >temp1,temp2;
        temp1.push_back(l1[0]);
        temp2=l1;
        temp2.erase(temp2.begin()+0);
        func(temp1,temp2);
        temp1.erase(temp1.begin()+0);
        temp1.push_back(l2[0]);
        temp2=l2;
        temp2.erase(temp2.begin()+0);
```

```
        func(temp1,temp2);
        return;
}
int main(){
        ios::sync_with_stdio(false);
        srand(time(NULL));
        cin>>N>>M;
        for(int i=1;i<=N;i++){
                for(int j=1;j<=N;j++){
                        a[i][j]=0;
                }
                visited[i]=0;
                cnt[i]=0;
                }
        if(N==1){
                cout<<"Prime"<<endl;
                return 0;
        }
        for(int i=0;i<M;i++){
                int q,w;
                cin>>q>>w;
                a[q][w]=1;
                a[w][q]=1;
        }
        int start=rand()%N+1;
        cout<<start<<" is chosen"<<endl;
        vector<vector<int> > list1, list2;
        vector<int> group1, group2;
        for(int i=1;i<=N;i++){
                if(i!=start){
                        if(a[i][start]==0){
                                group1.push_back(i);
                        }else{
                                group2.push_back(i);
                        }
                }
        }
        list1.push_back(group1);
        list2.push_back(group2);
        func(list1,list2);
        if(check==0){
                cout<<"Prime"<<endl;
                return 0;
        }
        cout<<"Not prime"<<endl;;
        for(int i=0;i<module.size();i++){
                cout<<module[i]<<" ";
        }
        return 0;
}
```

References

[1] David Ron Karger *Global Min-cuts in RNC and Other Ramifications of a Simple Mincut Algorithm* 1993.

[2] Marc Tedder, Derek Corneil, Michel Habib, Christophe Paul *Simple, Linear-time Modular Decomposition* 2008

[3] Vadim Loin *Graph Theory Notes* University of Warwick