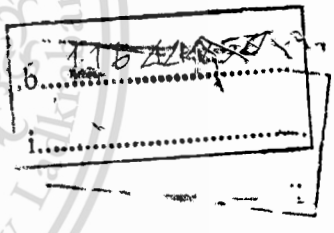
ALGORITHMS FOR SOLVING

THE MAXIMUM CLIQUE PROBLEM

RUNGAREE SANTAWAKUP



A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE IN APPLIED MATHEMATICS

SCHOOL OF GRADUATE STUDIES

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2005

ISBN 974-15-1939-7

หัวข้อวิทยานิพนธ์	ขั้นตอนวิธีเพื่อหาคำตอบของปัญหาคลิกที่ใหญ่ที่สุด
นักศึกษา	นางสาวรุ่งอารีย์ สันถวะคุปต์
รหัสประจำตัว	46063708
ปริญญา	วิทยาศาสตรมหาบัณฑิต
สาขาวิชา	คณิตศาสตร์ประยุกต์
พ.ศ.	2548
อาจารย์ผู้ควบคุมวิทยานิพนธ์	ผศ.ดร.ฉัฐไชย์ ลีนาวงศ์

บทคัดย่อ

เมื่อกำหนดกราฟมาให้ ในที่นี้จะทำการศึกษา ปัญหาคลิกที่ใหญ่ที่สุด (Maximum Clique Problem) ซึ่งเป็นปัญหาการหาค่าเหมาะสมเชิงการจัด (Combinatorial Optimization Problem) ที่รู้จักกันอย่างกว้างขวาง โดยคลิกของกราฟ คือ เซตของจุดยอด ที่จุดยอด 2 จุดยอดใด ๆ ประชิดกัน ส่วนปัญหาคลิกที่ใหญ่ที่สุด คือ ปัญหาที่สนใจหาคลิกที่มีจำนวนจุดยอดมากที่สุด

เนื่องจากปัญหาคลิกที่ใหญ่ที่สุดเป็นปัญหาเอ็นพีบริบูรณ์ (NP-Complete) ดังนั้นอาจคาดการณ์ได้ว่าไม่มีขั้นตอนวิธีที่มีความซับซ้อนเชิงเวลาเป็นพหุนามมาแก้ปัญหาที่มีขนาดใหญ่ได้ อย่างไรก็ดี ทางออกหนึ่งก็คือ พยายามพัฒนาขั้นตอนวิธีที่เร็ว ซึ่งอาจเป็นพหุนามหรือไม่ก็ได้ เพื่อนำมาแก้ตัวอย่างปัญหาที่มีลักษณะเฉพาะบางประการ

วิทยานิพนธ์นี้นำเสนอการปรับปรุงขั้นตอนวิธีเดิมที่ใช้วิธีขยาย และจำกัดเขต รวมทั้งกลยุทธ์การลดทอน ให้มีประสิทธิภาพดีขึ้น โดยการเรียงลำดับจุดยอดตามจำนวนเส้นเชื่อมที่มาตกกระทบแทน ขั้นตอนวิธีนี้และขั้นตอนวิธีเดิมจะนำมาพัฒนาขึ้นเป็นโปรแกรมคอมพิวเตอร์ และทดสอบกับกราฟที่มีลักษณะเฉพาะที่เรียกว่ากราฟสุ่ม จากนั้นจะนำผลที่ได้จากขั้นตอนวิธีนี้ไปเปรียบเทียบกับผลที่ได้จากขั้นตอนวิธีอื่น ๆ ที่มีอยู่ก่อนแล้ว

นอกจากนี้ ขั้นตอนวิธีเพื่อหาคำตอบของปัญหาสีของจุดยอดที่น้อยที่สุด จะนำมาใช้ในการประมาณคำตอบที่ต้องการได้อีกทางหนึ่ง อันเนื่องมาจากความจริงที่ว่า คำตอบของปัญหานี้เป็นขอบเขตบนของปัญหาคลิกที่ใหญ่ที่สุด

Thesis Title	Algorithms for Solving the Maximum Clique Problem
Student	Miss Rungaree Santawakup
Student ID	46063708
Degree	Master of Science
Programme	Applied Mathematics
Year	2005
Thesis Advisor	Asst.Prof.Dr.Chartchai Leenawong

ABSTRACT

Given a graph, in a well-known combinatorial optimization problem, the search for a maximum clique is investigated here. A clique of a graph is a set of vertices, any two of which are adjacent. The maximum clique problem asks for a clique having a largest vertex set.

Since the maximum clique problem is an NP-hard problem, no polynomial time algorithms are expected to be found for large-sized problems. However, one may try to develop a fast algorithm, not necessarily polynomial, that solves some certain types of instances.

This thesis modifies a specific previous algorithm that uses branch and bound as well as pruning strategy by reordering the vertices according to the degrees of vertices. Then the new algorithm and some previous algorithms are implemented on a computer to compare the results of these algorithms on a certain type of graphs, namely, random graphs.

In addition, for approximation proposes, an algorithm for solving the minimum vertex color problem is also implemented because of the fact that the solution to this problem is the upper bound to the solution of the maximum clique problem.

ACKNOWLEDGEMENTS

I am very grateful for the suggestion and encouragement received from Asst. Prof. Dr. Chartchai Leenawong, my thesis advisor. I am also indebted to all of lecturers for their previous valuable lectures while studying here. I would like to express my thanks to my thesis committee for many useful discussions, comments, and suggestions.

I also thank the School of Graduate Studies King Mongkut's Institute of Technology Landkrabang for providing me a scholarship.

I am very thankful for the suggestion about writing C programming received from Pasit Sasikarn, my friend.

Special thanks are extended to my friends for their assistance and encouragement during thesis preparation.

I would like to express my eternal gratitude to my parents for their endless support and understanding.

Finally, I would like to dedicate this thesis to all the readers.

Rungaree Santawakup

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF TABLES

LIST OF FIGURES

Figures Page

LIST OF FIGURES (CONTINUED)

Figures Page

CHAPTER 1

INTRODUCTION

1.1 Statement and Significance of the Problem

The maximum clique problem is a classical problem in combinatorial optimization. Its applications can be found in various areas such as production/ operations. This section is to state the problem by starting with basic graph theory concepts.

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*. An *independent set* of a graph is a subset S of V, such that every two nodes in S are not joined by an edge of E. The *maximum independent set problem* consists of finding the largest cardinality of an *independent set*. A *vertex cover* of a graph is a subset C of V, such that every edge of E has an endpoint in C. The *minimum vertex cover problem* consists of finding the smallest cardinality of a *vertex cover*.

The *maximum clique problem* asks for a *clique* having a largest node set. This problem is computationally equivalent to some other important graph problems, for example, the *maximum independent set problem* and the *minimum vertex cover problem*. Since these are *NP-hard* problems [1], no polynomial time algorithms are expected to be found. Therefore almost all types of algorithms have been used to try to solve it. 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 heuristic methods to find as large *cliques* as possible, without proving optimality. At the survey of Pardalos and Xue [4] provides an extensive bibliography on the *maximum clique problem*.

1.2 Objectives

The objectives of the research are following.

1.2.1 To develop heuristics algorithms for solving the maximum clique problem.

1.2.2 To improve the speed of algorithms in solving the maximum clique problem for a certain type of graphs.

1.2.3 To do a comparative study on the efficiency of the proposed algorithms and other algorithms.

1.3 Scope of the Study

1.3.1 This research uses heuristics methods to find as large cliques as possible in *random graphs*.

1.3.2 For the comparative study of the algorithms, computer experiments are conducted using C programming on a 1.8 GHz Pentium-M laptop computer with 768 MB RAM operated by Microsoft Windows XP.

1.3.3 Random graphs used in every algorithm are generated by the computer using C programming as well.

1.4 Benefits

1.4.1 More efficient algorithms for solving the maximum clique problem in a certain type of graphs.

1.4.2 Computer programs for generating random graphs and for solving the maximum clique problem.

1.5 Research Procedure and Schedule

Steps for doing this research are summarized here.

1.5.1 Study basic knowledge of graph theory and combinatorial optimization, clique, and the maximum clique problem.

1.5.2 Examine related literature on clique and the maximum clique problem.

1.5.3 Identify existing algorithms that can solve the problem.

1.5.4 Learn and practice C programming.

1.5.5 Find new ideas to improve the efficiency of existing algorithms.

1.5.6 Develop computer programs for both the existing and the proposed algorithms.

1.5.7 Run comparative experiments for all the algorithms.

1.5.8 Analyze and conclude the results.

1.5.9 Write the documents.

The tentative time schedule for each corresponding step of the research procedure is given here.

Research Step	Month											
	1	2	3	4	5	6	7	8	9	10	11	12
1.5.1	←	→										
1.5.2		←	→	→								
1.5.3			←	→	→	→						
1.5.4				←	→	→						
1.5.5						←	→	→				
1.5.6								←	→	→		
1.5.7									←	→	→	
1.5.8										←	→	→
1.5.9											←	→

CHAPTER 2

BASIC KNOWLEDGE AND LITERATURE REVIEWS

In this chapter, some useful backgrounds on graph theory and combinatorial optimization are presented. It is then followed by selective literature reviews on the maximum clique problem.

2.1 Basic Knowledge

A combinatorial optimization problem (COP) is a problem in which it is necessary to choose one of a large, but finite, number of alternatives so as to minimize or maximize an overall objective.

A COP consists of the following components:

1. Set of data.
2. Set of feasible solutions.
3. Set of building blocks.
4. An objective function value.
5. An overall objective of finding an optimal solution.

Algorithms for finding an optimal solution can be classified into 3 types:

1. A greedy algorithm that attempts to create an optimal solution by choosing the building blocks, one at a time.

2. A finite-improvement algorithm that uses a movement mechanism to create a sequence of feasible solution, each with a strictly better objective function value than the previous one.

3. A reduction that uses an existing algorithm for solving an old problem.

Problem transformation of a reduction algorithm is illustrated in Figure 2.1

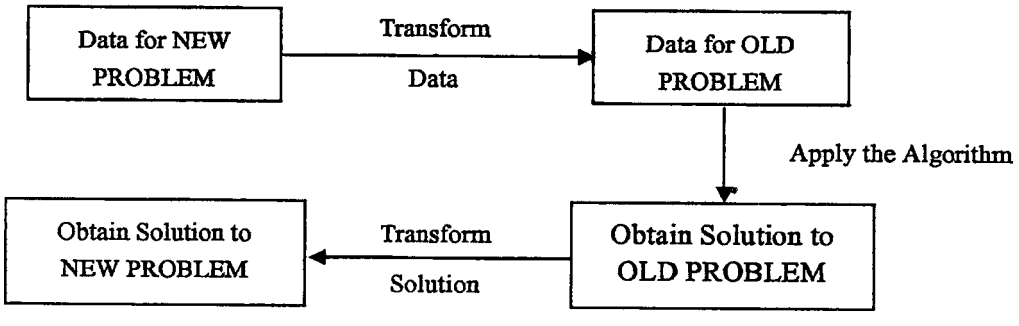


Figure 2.1 A Reduction from NEW PROBLEM to OLD PROBLEM

A COP can be converted into a collection of decision problems by including a real number R and asking the question "Does there exist a feasible solution whose objective function value is better than R?". By repeatedly trying different values of R, you can identify the optimal solution.

This thesis is interested in COPs in decision form. Relative Definitions and Theorems are provided [8, 9].

Definition 2.1 If two vertices are assigned to an edge x, we say that u and v are *adjacent* by x, or x is *incident* with u and v. The *degree* of a vertex v in a graph G is an integer

$\deg_G(v)$ = number of edges incident with v.

Definition 2.2 The *running time* (or *time complexity function*) of an algorithm A for solving a decision problem X is defined as

$T_A(n)$ = maximum amount of work needed by algorithm A to solve any instance of X of size n.

Definition 2.3 An efficient (polynomial bounded, polynomial time, or polynomial) algorithm is an algorithm A for which there is a polynomial p such that the time complexity function satisfies

$T_A(n) \leq p(n)$ for $n = 1, 2, \ldots$.

Definition 2.4 *P* is the class of all decision problems that can be solved in polynomial time.

Definition 2.5 *NP* is the class of all decision problems where a given solution can be verified in polynomial time.

Definition 2.6 A decision problem X is *NP-Complete* if

i) $X \in NP$ and

ii) For every $Y \in NP$, $Y \ \alpha_p \ X$ (α_p denotes polynomial reduction)

Definition 2.7 Let an undirected graph 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.

Definition 2.8 A *clique* of a graph is a set of vertices, any two of which are adjacent.

Definition 2.9 A clique is *maximal*, if it is not contained in any other clique.

Definition 2.10 A clique is *maximum*, if it is the biggest of all maximal cliques.

The following theorem starts a list of known NP-Complete problems using the definition of NP-Complete.

Theorem 2.1 (Cook's Theorem) CNF – Satisfiability is NP-Complete.

Theorem 2.2 If $X \in$ NP and there is a problem $Y \in$ NP-Complete such that $Y \alpha_p X$, then $X \in$ NP-Complete.

This theorem is useful for proving indirectly that a new problem is NP-Complete when a list of known NP-Complete problems is provided. Maximum clique problem is also proved according to this theorem and the known NP-Complete problem used is CNF-Sat [10].

The ways to solve this problem as follows:

1. Develop a non-polynomial algorithm for solving the problem optimally.
2. Develop a polynomial heuristic to obtain a relatively good solution.
3. Develop a polynomial algorithm for solving special cases of the problem optimally.

An implicit graph is one for which we have available a description of its nodes and edges. Relevant portions of the graph can thus be built as the search progresses. Therefore computing time is saved whenever the search succeeds before the entire graph has been constructed. The economy in memory space is even more dramatic when nodes that have already been searched can be discarded, making room for subsequent nodes to be explored.

Backtracking is a basic search technique on implicit graphs. One powerful application is in playing games of strategy by techniques known as minimax and alpha-beta pruning. Some optimization problem can be handled by the more sophisticated branch-and-bound technique. Discuss these notions.

1. Backtracking

Backtracking algorithms use a special technique to explore implicit directed graphs. These graphs are usually trees, or at least they contain no cycles. A backtracking algorithm carries out a systematic search, looking for solutions to some problem. At least one application of this technique dates back to antiquity: it allows one to find the way through a labyrinth without danger of going round and round in circles. To illustrate the general principle, we shall, however, use a different example. Consider classic problem of placing eight queens on a chess-board in such a way that none of them threatens any of the others.

2. Branch-and-Bound

Branch-and-bound algorithm developed here is an improvement on the backtracking algorithm. Its design strategy is very similar to backtracking in that a state space tree is used to solve a problem. The differences are that.

i) The branch-and-bound method does not limit us to any particular way of traversing the tree

ii) The branch-and-bound method is used only for optimization problem.

Branch-and-bound algorithm computes a number (bound) at a node to determine whether the node is promising. The number is a bound on the value of the solution that could be obtained by expanding beyond the node: If that bound is no better than the value of the best solution found so far, the node is nonpromising. Otherwise, it is promising. Because the optimal value is a minimum in some problems and a maximum in others, by "better" we mean smaller or larger depending on the problem. As is the case for backtracking algorithms, branch-and-bound algorithms are ordinarily exponential-time (or worse) in the worst case. However, they can be very efficient for many large instances.

2.2 Literature Reviews

In this section, related work about the maximum clique problem is surveyed.

2.2.1 Wood's Algorithm

Wood [6] presents his branch-and-bound algorithm named Algorithm MC for the maximum clique problem. Algorithm MC uses the Fractional Coloring Procedures (FCP) heuristic [11] to determine upper bounds. Like the algorithms of [3, 5], it activates exactly one new search tree node at each branching stage.

Given a graph G =(V,E) algorithm MC maintains the following conditions:

- If h is the depth of the search tree the set $\{ v_1, v_2, \ldots, v_{h-1} \}$ consists of pairwise adjacent vertices.
- M is the largest clique found by the algorithm; $h-1 \leq |M| \leq \omega(G)$, where $\omega(G)$ is maximum clique size.
- For $1 \leq i \leq h$, the vertex set $S_i \subseteq \bigcap_{j=1}^{i-1} N_G(v_j)$ consists of candidates for enlarging $\{ v_1, v_2, \ldots, v_{i-1} \}$ into a clique.
- For $1 \leq i \leq h$, $(C_1^i, C_2^i, \ldots, C_{ki}^i)$ is a vertex coloring of $G(S_i)$. Both k_i and k_i' (determined by FCP) are upper bounds for $\omega(G(S_i))$, with $k_i' \leq k_i$.
- An active node of the search tree corresponds to the subproblem of finding a maximum clique larger than M of the subgraph:

  $G_i = G(\{v_1, v_2, \ldots, v_{i-1}\} \cup S_i)$, for $1 \leq i \leq h$.

  Clearly, $\omega(G_i) \leq i - 1 + k_i' \leq i - 1 + k_i$.

FCP (at iteration i)

For each vertex v, include v in the first color class $C_j \in C$, if one exists, such that $C_j \cup \{ v \}$ is an independent set. Let U be the set of vertices not included in a color class. Find a vertex coloring $(C_1, C_2, \ldots, C_k)$ of G(U) (using COLOR or DSATUR), and set $C := C \cup \{C_1, C_2, \ldots, C_k\}$ and $t_i := |C| / i$. If $t_i < t_{i-1}$ then set $i := i + 1$ and repeat, otherwise return the upper bound $\lfloor t_{i-1} \rfloor$

Algorithm MC

Step 0 (Initialization):

Find a maximum clique M of an edge-maximal triangulated subgraph of G.

Set h := 1, S_h := V and go to Step 2.

Step 1 (Calculate Lower Bound):

Find a clique Q of $G(S_h)$. If h-1 +|Q| > |M| then set M := $\{v_1, v_2, .., v_{h-1}\} \cup Q$.

Go to step 2.

Step 2 (Calculate Upper Bound):

Find a vertex coloring ($C_1^h, C_2^h, \ldots, C_{kh}^h$) of $G(S_h)$.

If $h-1 + k_h \leq |M|$ then go to Step 4.

Apply FCP to $G(S_h)$ to obtain a further upper bound $k_h' \geq \omega(G(S_h))$.

If $h-1 + k_h' \leq |M|$ then go to Step 4.

Go to Step 3.

Step 3 (Branching):

Choose a vertex $v_h \in C_{kh}^h$ with maximum degree in G.

Set $S_{h+1} := S_h \cap N_G(v_h)$, $S_h := S_h \setminus \{v_h\}$, $C_{kh}^h := C_{kh}^h \setminus \{v_h\}$.

If $C_{kh}^h = \emptyset$ then decrement k_h and if $k_h < k_h'$

Then set $k_h' := k_h$.

Increment h and go to Step 1.

Step 4 (Backtracking):

If h = 1 then stop: M is a maximum clique of G.

Decrement h and if $h-1 + k_h' \leq |M|$ then go to Step 4.

Go to Step 3.

2.2.2 Östergård's Algorithm

Östergård [3] has proposed to use branch-and-bound algorithm to solve this problem. His algorithm tries to improve old algorithm of Carraghan and Pardalos [2], which can be seen as a basic form of most published algorithms.

Let $S_i = \{v_i , v_{i+1} ,\ldots, v_n\}$. The old algorithm searches by first considering cliques in S_1 that contain v_1, then cliques in S_2 that contain v_2, and so on. In new algorithm, this ordering is reversed: first consider cliques in S_n that contain v_n and so on.

Algorithm 1. Old algorithm.

```
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 ≠ 0 do
        if size + |U| ≤ 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
```

Algorithm 2. New algorithm.

```
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 ≠ 0 do
        if size + |U| ≤ max then
              return
        end if
        i :=min{ j | v_j ∈ U}
        if size + c[i] ≤ 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 ∩ N(v_i), 1)
        c[i]:=max
  end for
  return
```

2.2.3 Fahle's Algorithm

Fahle [7] consider a branch-and-bound algorithm for maximum clique problems. He introduces cost based filtering techniques for the so-called *candidate set* (i.e. a set of nodes that can possibly extend the clique in the current choice point).

Additionally, he presents taxonomy of upper bounds for maximum clique. Analytical results show that our cost based filtering is in a sense as tight as most of these well-known bounds for the maximum clique problem.

Experiments demonstrate that the combination of cost based filtering and vertex coloring bounds outperforms the old approach as well as approaches that only apply either of these techniques. Furthermore, the new algorithm is competitive with other recent algorithms for maximum clique

Fahle's Algorithm

function findClique2(set C, set P)

1: **if** $(|C| > |C^*|)$ **then**

2: $C^* \leftarrow C$

3: **if** $(|C| + |P| > |C^*|)$ **then**

4: **for** all $p \in P$ in predetermined order: **do**

5: $P \leftarrow P \setminus \{p\}$

6: $C' \leftarrow C \cup \{p\}$

7: $P' \leftarrow P \cap N(p)$

8: // domain filtering

9: // reduce possible set

10: **while** $(\exists v \in P' : \delta_{p'}(v) + |C'| < |C^*|)$ **do**

11: $P' \leftarrow P' \setminus \{v\}$ // lemma 1

12: // increase required set

13: **while** $(\exists v \in P' : \delta_{p'}(v) = |P'| - 1)$ **do**

14: $C' \leftarrow C' \cup \{v\}$ // lemma 2

15: $P' \leftarrow P' \setminus \{v\}$ // lemma 2

16: // here it holds: $(\forall v \in P' : (|C^*| - |C'|) \leq \delta_{p'}(v) < (|P'| - 1)$

17: findClique2(C', P')

2.2.4 Régin's Algorithm (ILOG)

Régin (ILOG) [8] aims to show that Constraint Programming can be an efficient technique to solve the maximum clique problem. He proposes two new upper bounds of $\mathcal{O}(G)$ and a new strategy to guide the search for an optimal solution. The interest of his approach is emphasized by the results we obtain for the DIMACS Benchmarks. Seven instances are solved for the first time and two better lower bounds for problems remaining open are found. Moreover, he show that the CP method we propose gives good results and quickly.

CP Algorithm

maximumClique(Current,Candidate,Not, **io** K)

while Candidate $\neq \emptyset$ **do**

select x in Candidate and remove it

save Candidate

save Not

add x to Current

remove from Candidate the nodes y s.t. $y \neq \Gamma(x)$

removeFromNot(x)

if FIterAndPropagate(Current,Candidate,Not,K) **then**

if Candidate = Ø then K ← Current // solution

else maximumClique(Current,Candidate,K)

restore Not

remove x from Current

add x to Not

restore Candidate

Filtering algorithm and propagation

filterAndPropagate(Current, Candidate, **io** K)

do

do

continue ← false

for each y in Candidate **do**

$N \leftarrow |\Gamma(y) \cap Candidate|$

if $N + |Current| < |K|$ **then** remove y from Candidate

else

if $N - \lceil \frac{N}{2} \rceil + |Current| < |K|$ **then**

Let H be the subgraph of G induced by $\Gamma(x) \cap Candidate$

compute $\mu(\bar{H}^d)$

if $N - \lceil \frac{\mu(\bar{H}^d)}{2} \rceil + |Current| < |K|$ **then** remove y from Candidate

if $y \notin Candidate$ then

Let H be the subgraph of G induced by Candidate

compute $\mu(\bar{H}^d)$

if $|Candidate| - \lceil \frac{\mu(\bar{H}^d)}{2} \rceil + |Current| < |K|$ **then** return false;

continue ← true

while continue

if continue then

continue ← FilteringFromNot(Not, Candidate)

while continue

return true

CHAPTER 3

PROPOSED ALGORITHMS FOR SOLVING THE MAXIMUM CLIQUE PROBLEM

In this chapter, the proposed algorithms for solving the maximum clique problem are presented. For comparison purposes, two previous algorithms are also shown and discussed in more details. General concepts behind each algorithm are first discussed, followed by the steps of the algorithm. However, a list of all the symbolic notations to be used throughout this thesis is first given here.

- $G(V,E)$ is an arbitrary simple graph where V is the set of vertices and E is the set of edges
- U, W are the set of vertices ($\subseteq V$)
- *max* is the maximum clique size
- *size* is a maximal clique size for each iteration
- i is the iteration counter
- n is the number of vertices in the graph
- *found* is true if the maximal clique is obtained
- $c[i]$ is an array for keeping track of the maximal clique size in each iteration i

3.1 Previous Algorithms

This section presents two maximum clique problem algorithms proposed by Carraghan and Pardalos [2] in 1990 and Östergård [3] in 2002. The ideas behind the algorithm are given now.

Algorithm 1 (Carraghan and Pardalos)

Given an arbitrary graph, the algorithm basically starts at a vertex to see how large a maximal clique size can be identified if that vertex is included in the clique. Repeat the process for other vertices until the last vertex is considered. In the mean time, after each iteration, update the maximum clique size found so far.

Details and steps of this algorithm are follows.

Steps of Algorithm 1

Step 0 : Set $max = 0$, $i = 1$, $size = 0$

Step 1 : Given a graph $G(V, E)$, randomly assign numbers 1 to n to the vertices.

Step 2 : If $(i > n)$ or the possible clique size of the remaining graph is not greater than the current maximum clique, stop, *max* is the maximum clique size. Otherwise, go to step 3.

Step 3 : Remove vertex i from V, set $size = size + 1$.

Step 4 : Set U equal to the intersection of neighboring vertices of the just-removed vertex in Graph V and the remaining vertices.

Step 5 : If the possible clique size of the remaining graph is not greater than the current maximum clique. Set $i = i + 1$, $size = 0$ and go back to step 2. Otherwise, go to step 6.

Step 6 : If $U = \emptyset$, *size* is a maximal clique size and set $max = size$ only if *max* is less than *size*. Also set $i = i + 1$, $size = 0$ and go back to step 2. Otherwise, remove a smallest vertex in U, set $size = size + 1$ and go back to step 4.

Algorithm 2 (Östergård)

This algorithm starts at a subgraph that has one vertex to see how large a clique can be. Then insert another vertex and consider again the maximal clique size in the induced subgraphs with those vertices. Repeat the process until the last vertex is inserted. Furthermore, to make it faster, the variable *found* is used to see if the maximum clique size is found. Details and steps of the algorithm are explained follows.

Steps of Algorithm 2

Step 0 : Set $max = 0$, $i = n$, $size = 1$, $found$ = false, $c[i] = 0$, $W = \emptyset$.

Step 1 : Given a graph $G(V,E)$, randomly assign numbers 1 to n to the vertices.

Step 2 : If ($i < 1$), stop, max is the maximum clique size.

Otherwise, go to step 3.

Step 3 : Insert vertex i (from V) into W.

Step 4 : Set U equal to the intersection of neighboring vertices of the just added vertex in Graph W and the vertices in Graph W.

Step 5 : If the possible clique size of the remaining graph (depending on U) is not greater than the current maximum clique, set $c[i] = max$, $i = i - 1$, $size = 1$ and go back to step 2.

Otherwise, go to step 6.

Step 6 : If $U = \emptyset$, $size$ is a maximal clique size and set $max = size$ only if max is less than $size$. Also set $c[i] = max$, $i = i - 1$, size = 0 and go back to step 2.

Otherwise, go to step 7.

Step 7 : If the possible clique size of the remaining graph (depending on $c[i]$) is not greater than the current maximum clique or $found$ = true then set $i = i-1$, $size = 1$ and go back to step 2

Otherwise, remove a smallest vertex in U , set $size = size + 1$ and go back to step 4.

3.2 Proposed Algorithms and Experimental Design

In this section, two proposed algorithms modified from the previous algorithms are presented. It is hoped that the proposed algorithms will outperform the previous two algorithms, especially on a certain type of graphs, namely, random graphs.

The definition of random graphs is given later in this section. A way of generating random graphs by the computer is then followed. Examples of random graphs generation are demonstrated.

3.2.1 The Proposed Algorithms

In this section, two proposed algorithms (and will be called Algorithm 3 and Algorithm 4) for solving the maximum clique problem are presented. Again, the general concepts along with the steps of the algorithms are shown.

Algorithm 3

For good performance of the algorithm, 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.

We will now consider an improved algorithm by first sorting the vertices using selection sort with respect to their degrees (the number of *incident* edges). Additional notation to be used is defined here. Let $S_i = \{v_1, v_2, \ldots, v_i\}$ where $\deg_G(v_1) \geq \deg_G(v_2) \geq \ldots \geq \deg_G(v_i) \geq \ldots \geq \deg_G(v_n)$. This improved algorithm searches for the maximum clique by first considering S_1 then S_2, and so on. Details and steps of the algorithm are explained follows.

Steps of Algorithm 3

Step 0 : Set $max = 0$, $i = 1$, $size = 1$, $found$ = false, $c[i] = 0$, $W = \emptyset$

Step 1 : Given a graph $G(V,E)$, sort the vertices in descending order of their degrees and assign numbers 1 to n to the sorted vertices.

Step 2 : If $(i > n)$ or the possible clique size of the remaining graph (depended on U , $c[i]$) is not greater than the current maximum clique, stop, max is the maximum clique size.

Otherwise, go to step 3.

Step 3 : Insert vertex i (from V) into W.

Step 4 : Set U equal to the intersection of neighboring vertices of the just added vertex in Graph W and the vertices in Graph W.

Step 5 : If the possible clique size of the remaining graph (depending on U) is not greater than the current maximum clique, set $c[i] = max$, $i = i - 1$, $size = 1$ and go back to step 2.

Otherwise, go to step 6.

Step 6 : If $U = \emptyset$, $size$ is a maximal clique size and set $max = size$ only if max less than $size$. Also set $c[i] = max$, $i = i - 1$, size = 0 and go back to step 2.

Otherwise, go to step 7.

Step 7 : If the possible clique size of the remaining graph (depending on $c[i]$) is not greater than the current maximum clique or $found$ = true then set $i = i-1$, $size = 1$ and go back to step 2

Otherwise, remove a smallest vertex in U , set $size = size + 1$ and go back to step 4.

Algorithm 4

This algorithm uses minimum vertex coloring problem to approximate the maximum clique size. Because the maximum clique size is less than or equal to the minimum number of vertex colors [12], or $\omega(G) \leq \chi(G)$.

A vertex coloring is an assignment of labels or colors to each vertex of a graph such that no edge connects two identically colored. The most common type of vertex coloring seeks to minimize the number of colors for a given graph and this number is an upper bound of the maximum clique size.

First, initialize parameters i and color c to be assigned to v_i. Then assign the minimum possible color c to v_i. Repeat the process until the last vertex is assigned.

Details and steps of the algorithm are as follows.

Steps of Algorithm 4

Step 0 : [This step initializes the parameter i and color c , used for naming the current vertex v_i and to be assigned to v_i.]

$i = 1,\ c = 1.$

Step 1 : Given a graph $G(V,E)$, assign numbers 1 to n to the vertices.

Step 2 : [The minimum possible color c is assigned to v_i.]

2.1 Sort the colors adjacent with v_i in nondecreasing order and call the resulting list L_i.

2.2 If c does not appear on L_i , then assign color c to v_i and go to Step 4

Otherwise; continue.

Step 3 : [The color c is incremented.]

$c = c + 1$ and return to Step 2.2.

Step 4 : [The parameter i is incremented.]

If $i < n$, then $i = i + 1$, and return to Step 1;

Otherwise, stop.

3.2.2 Experimental Design

In this section, the ideas of random graphs to be used in the computer experiment are first explained. It is then followed by the experimental design together with the desired forms of expected results.

3.2.2.1 Definition of Random Graphs

Random graphs are graphs randomly generated by the computer. Input parameters are the number of vertices and the edge density. The edge density is defined as the proportion of number of edges in the graph being considered and the number of edges in the complete graph on the same set of vertices. For example, for a graph of 10 vertices, if our graph has 27 edges, the edge density will be $27/\binom{10}{2} = 0.6$ or 60%.

3.2.2.2 Random Graphs Generation

For each instance, a random graph is constructed by the following steps.

Step 1 : Define an edge density.

Step 2 : Select a pair of vertices.

Step 3 : Random a number between 0 – 1 uniformly. If the random number is not greater than edge density then insert an edge.

Step 4 : Repeat Step 2 and Step 3 for every other pair of vertices

Eventually, after many instances have been undertaken, the average edge density of the graphs actually constructed will be very close to the predetermined edge density.

An example of how to computer-generate random graphs of 10 vertices with a predetermined edge density of 0.4 is given in Table 3.1. Note that the table shows only the case when the random numbers are within the given edge density.

Table 3.1 Examples of Random Graphs

No.	Instance #1		Instance #2		Instance #3		Instance #4		Instance #5	
	Random Number	Edge List	Random Number	Edge List	Random Number	Edge List	Random Number	Edge List	Random Number	Edge List
1	0.3460	1 2	0.3640	1 4	0.0080	1 2	0.3230	1 3	0.3650	1 2
2	0.1300	1 3	0.1510	1 6	0.2970	1 5	0.3780	1 5	0.3650	1 3
3	0.0900	1 5	0.1000	1 8	0.2750	1 8	0.3270	1 7	0.0790	1 4
4	0.1170	1 7	0.1080	2 3	0.0400	1 9	0.1640	1 8	0.1090	1 8
5	0.1260	2 3	0.3470	2 5	0.1370	2 4	0.0200	1 10	0.3130	1 9
6	0.0040	2 4	0.0420	2 6	0.3430	2 6	0.1130	2 3	0.1790	1 10
7	0.3600	2 9	0.1690	2 8	0.2670	2 7	0.2490	2 5	0.3670	2 3
8	0.0470	3 6	0.3830	2 10	0.3120	2 8	0.2430	2 6	0.3100	2 4
9	0.1190	3 7	0.3900	3 6	0.1110	2 9	0.3440	3 6	0.1460	2 5
10	0.1900	3 9	0.1900	3 10	0.3530	3 6	0.2310	3 7	0.0280	2 8
11	0.2140	4 5	0.2330	4 5	0.1260	3 7	0.2050	4 7	0.1770	2 9
12	0.2520	4 7	0.0530	4 6	0.0180	3 8	0.1920	4 9	0.0130	2 10
13	0.3100	6 7	0.2850	4 8	0.0860	3 9	0.1130	4 10	0.0940	3 7
14	0.0920	7 8	0.3330	4 10	0.0860	4 9	0.1720	5 7	0.0250	3 8
15	0.2880	7 10	0.2680	5 8	0.0150	5 9	0.1210	5 8	0.1110	4 8
16			0.1600	6 8	0.1430	5 10	0.3810	5 9	0.2810	4 10
17			0.1420	6 10	0.2460	6 7	0.3320	6 7	0.0990	5 6
18			0.1150	7 9	0.1800	6 9	0.1180	7 8	0.0510	5 7
19			0.1160	7 10	0.1680	6 10	0.3230	8 9	0.1030	5 8
20			0.1560	8 10	0.3650	8 9	0.1520	8 10	0.2940	5 10
21			0.2790	9 10	0.1350	9 10			0.1470	6 10
22									0.2870	7 8
	Total Edges	15	Total Edges	21	Total Edges	21	Total Edges	20	Total Edges	22

Table 3.1 (Continued)

No.	Instance #6		Instance #7		Instance #8		Instance #9		Instance #10	
	Random Number	Edge List	Random Number	Edge List	Random Number	Edge List	Random Number	Edge List	Random Number	Edge List
1	0.1930	1 2	0.0590	1 5	0.2140	1 3	0.1230	1 4	0.3120	1 2
2	0.1250	1 3	0.1030	1 9	0.3210	1 4	0.3030	1 6	0.3190	1 7
3	0.2070	1 6	0.3090	2 9	0.1370	2 3	0.1490	1 9	0.0020	1 8
4	0.1040	1 7	0.2700	2 10	0.2260	2 6	0.3350	2 10	0.3600	1 9
5	0.0160	1 8	0.1860	3 6	0.0280	2 7	0.3080	3 5	0.3240	2 3
6	0.1840	1 10	0.2430	3 7	0.3890	2 8	0.2100	3 6	0.3390	2 4
7	0.3260	2 3	0.1230	3 10	0.0360	2 9	0.2450	3 7	0.1520	2 7
8	0.0960	2 4	0.0700	4 7	0.0200	3 9	0.1330	3 9	0.2330	3 4
9	0.2690	2 7	0.1810	4 10	0.3360	3 10	0.0870	3 10	0.3310	3 6
10	0.0400	3 4	0.0460	5 7	0.1040	4 6	0.3240	4 5	0.2780	3 9
11	0.1000	3 6	0.2590	5 8	0.1010	4 8	0.2870	4 6	0.1870	3 10
12	0.2120	3 9	0.0060	5 9	0.3740	4 9	0.0870	4 7	0.1750	4 5
13	0.0570	3 10	0.1850	5 10	0.3920	6 8	0.0440	4 10	0.2520	4 6
14	0.2880	4 8	0.3710	6 8	0.2900	6 10	0.3450	5 7	0.0400	5 6
15	0.2110	4 9	0.2050	6 9	0.0040	7 8	0.1750	5 8	0.1770	5 7
16	0.3960	5 9	0.2210	6 10			0.3080	5 9	0.0190	5 9
17			0.3020	7 8			0.3910	6 7	0.1240	6 8
18			0.0130	8 9			0.1800	6 10	0.3220	6 9
19							0.1280	7 8	0.2210	6 10
20							0.2500	7 10	0.1390	7 10
21									0.1820	8 9
	Total Edges	16	Total Edges	18	Total Edges	15	Total Edges	20	Total Edges	21

The average number of edges for the above 10 instances = 189/10 = 18.9

$\therefore$ Actual Edge Density = $18.9 / \binom{10}{2} = 0.42$

A complete graph of 10 vertices and graphs associated with each instance in Table 3.1 are illustratively depicted here.

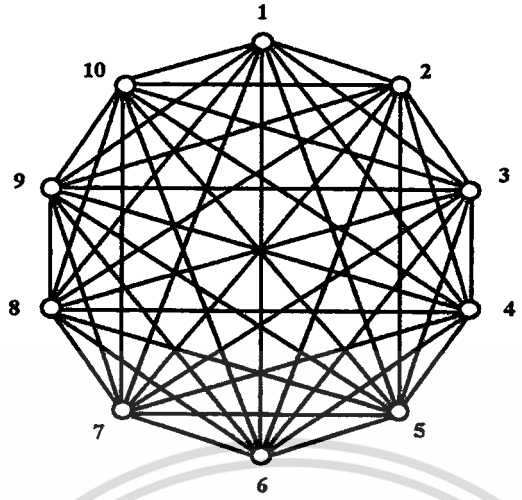
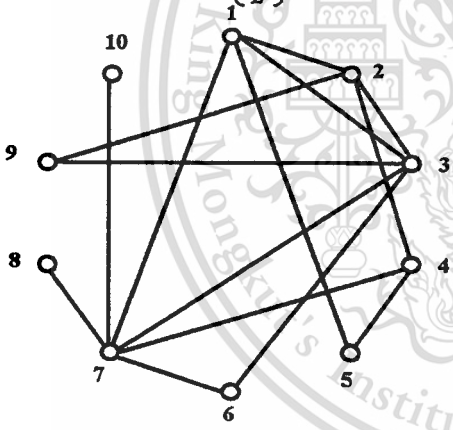


Figure 3.1 Complete Graph of 10 Vertices

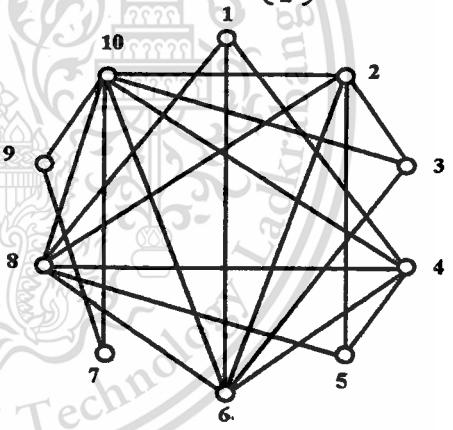
Instance #1 number of edge = 15,

edge density = $15/\binom{10}{2} = 0.33$



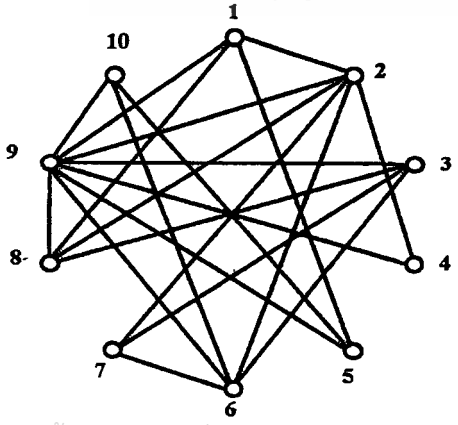
Instance #2 number of edge = 21,

edge density = $21/\binom{10}{2} = 0.47$



Instance #3 number of edge = 21,

edge density = $21/\binom{10}{2} = 0.47$,



Instance #4 number of edge = 20,

edge density = $20/\binom{10}{2} = 0.44$

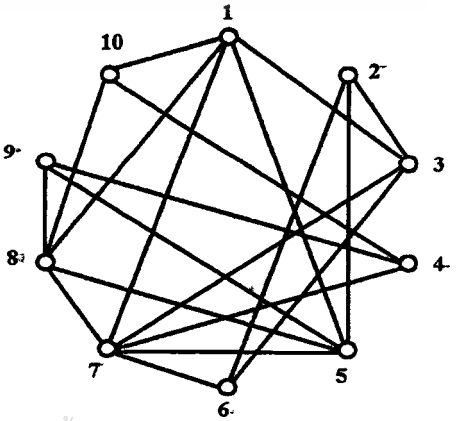
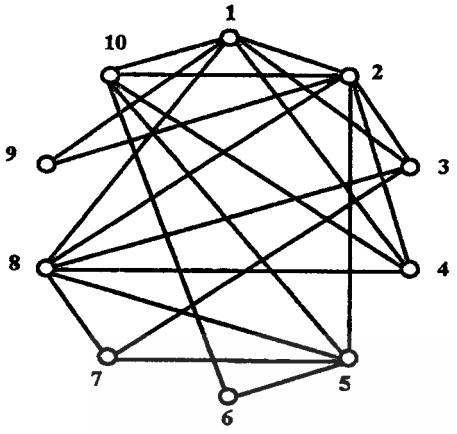


Figure 3.2 Graphs Associated with Each Instance in Table 3.1

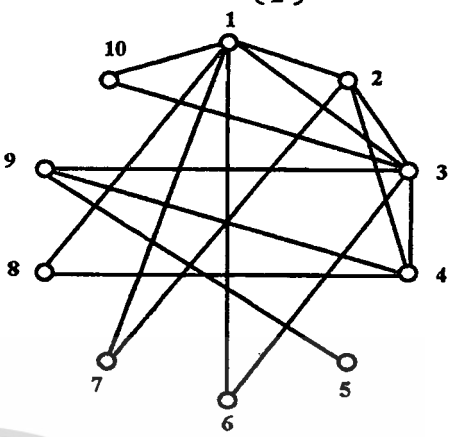
Instance #5 number of edge = 22,

edge density = $22/\binom{10}{2} = 0.49$



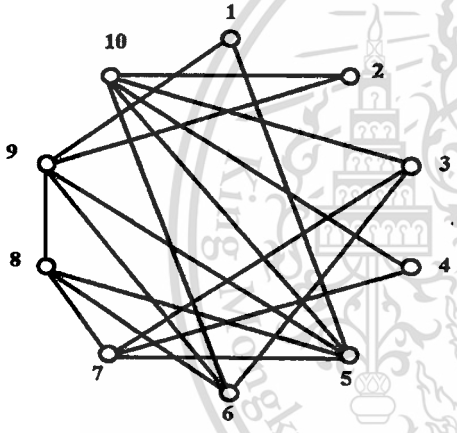
Instance #6 number of edge = 16,

edge density = $16/\binom{10}{2} = 0.36$



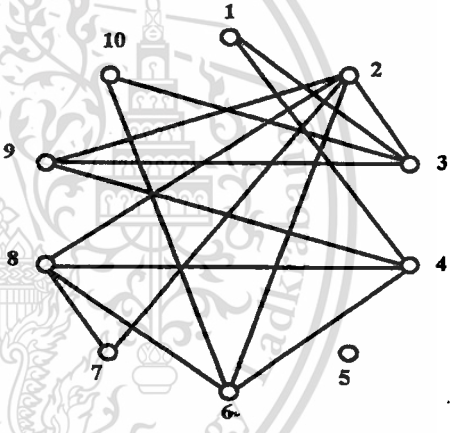
Instance #7 number of edge = 18,

edge density = $18/\binom{10}{2} = 0.4$



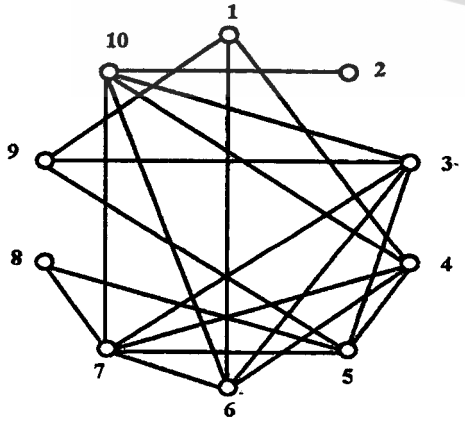
Instance #8 number of edge = 15,

edge density = $15/\binom{10}{2} = 0.33$



Instance #9 number of edge = 20,

edge density = $20/\binom{10}{2} = 0.44$



Instance #10 number of edge = 21,

edge density = $21/\binom{10}{2} = 0.47$

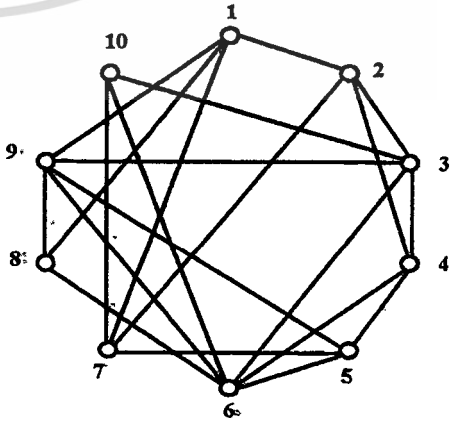
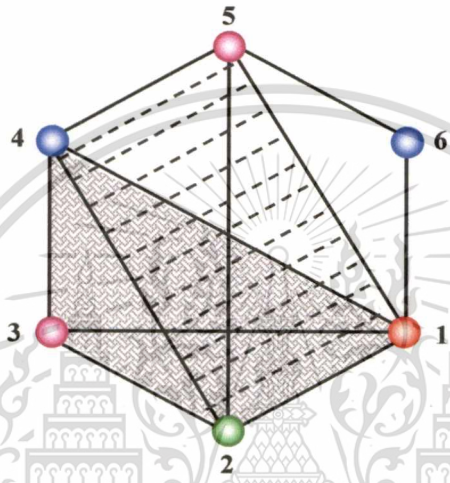


Figure 3.2 Graphs Associated with Each Instance in Table 3.1 (Continued)

3.2.2.3 Algorithm Demonstration

Algorithm 2 to 4 will be illustrated here on some examples of random graphs. Various examples shown in this section compare the maximum clique sizes obtained from algorithm 2 and 3 with the number of minimum colors obtained from algorithm 4 are exhibited here. All the instances are made different according to different values of edge density.

Example 1. A 5-vertex graph with 11 edges. So, edge density = $11/\binom{10}{2} = 0.73$.



$\deg_G(v_1) = 5$ $\deg_G(v_2) = 4$ $\deg_G(v_3) = 3$

$\deg_G(v_4) = 4$ $\deg_G(v_5) = 4$ $\deg_G(v_6) = 2$

Algorithm 4 : By vertex coloring, the minimum colors = 4.

Algorithm 2 : When the vertices are not sorted. (Random)

Assume $S_n = \{v_4, v_1, v_6, v_5, v_3, v_2\}$

So

$S_1 = \{4\}$ maximal clique size of $S_1 = 1$.

$S_2 = \{4, 1\}$ maximal clique size of $S_2 = 2$.

$S_3 = \{4, 1, 6\}$ maximal clique size of $S_3 = 2$.

$S_4 = \{4, 1, 6, 5\}$ maximal clique size of $S_4 = 3$.

$S_5 = \{4, 1, 6, 5, 3\}$ maximal clique size of $S_5 = 3$.

$S_6 = \{4, 1, 6, 5, 3, 2\}$ maximal clique size of $S_6 = 4$.

Hence the maximum clique size = 4.

Algorithm 3 : When the vertices are sorted.

We have $\deg_G(v_1) \geq \deg_G(v_2) \geq \deg_G(v_4) \geq \deg_G(v_5) \geq \deg_G(v_3) \geq \deg_G(v_6)$.

Then $S_n = \{v_1 , v_2 , v_4 , v_5 , v_3 , v_6\}$

So $S_1 = \{1\}$ maximal clique size of $S_1 = 1$.

$S_2 = \{1 , 2\}$ maximal clique size of $S_2 = 2$.

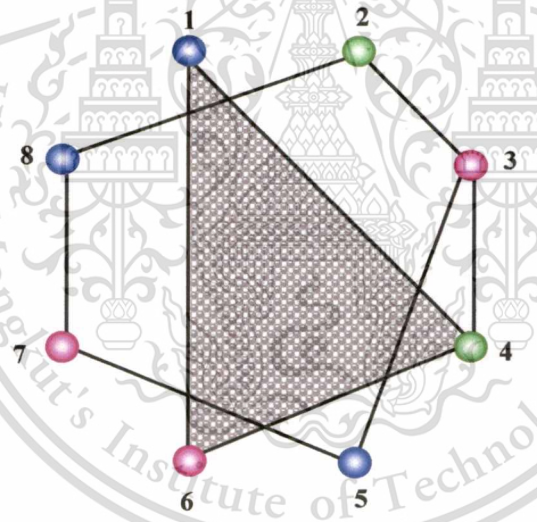
$S_3 = \{1 , 2 , 4\}$ maximal clique size of $S_3 = 3$.

$S_4 = \{1 , 2 , 4 , 5\}$ maximal clique size of $S_4 = 4$.

Hence the maximum clique size = 4.

We can see that with sorted vertices, the process is finished after 4 iterations. On the other hand, without sorting, it takes 6 iterations.

Example 2 . Graph 8 vertices. Edge density = $9/\binom{8}{2} = 0.32$.



$\deg_G(v_1) = 2$ $\deg_G(v_2) = 2$ $\deg_G(v_3) = 3$ $\deg_G(v_4) = 3$

$\deg_G(v_5) = 2$ $\deg_G(v_6) = 2$ $\deg_G(v_3) = 2$ $\deg_G(v_4) = 2$

Algorithm 4 : By vertex coloring , the minimum colors = 3.

Algorithm 2 : When the vertices are not sorted. (Random)

Assume $S_n = \{v_1 , v_2 , v_5 , v_6 , v_7 , v_8 , v_4 , v_3\}$

So $S_1 = \{1\}$ maximal clique size of $S_1 = 1$.

$S_2 = \{1, 2\}$ maximal clique size of $S_2 = 1$.

$S_3 = \{1, 2, 5\}$ maximal clique size of $S_3 = 1$.

$S_4 = \{1, 2, 5, 6\}$ maximal clique size of $S_4 = 2$.

$S_5 = \{1, 2, 5, 6, 7\}$ maximal clique size of $S_5 = 2$.

$S_6 = \{1, 2, 5, 6, 7, 8\}$ maximal clique size of $S_6 = 2$.

$S_7 = \{1, 2, 5, 6, 7, 8, 4\}$ maximal clique size of $S_7 = 3$.

Hence the maximum clique size = 3.

Algorithm 3 : When the vertices are sorted.

We have $\deg_G(v_4) \geq \deg_G(v_3) \geq \deg_G(v_1) \geq \deg_G(v_6) \geq \deg_G(v_5) \geq \deg_G(v_2) \geq \deg_G(v_7) \geq \deg_G(v_8)$.

Then $S_n = \{v_4, v_3, v_1, v_6, v_5, v_2, v_7, v_8\}$

So $S_1 = \{4\}$ maximal clique size of $S_1 = 1$.

$S_2 = \{4, 3\}$ maximal clique size of $S_2 = 2$.

$S_3 = \{4, 3, 1\}$ maximal clique size of $S_3 = 2$.

$S_4 = \{4, 3, 1, 6\}$ maximal clique size of $S_4 = 3$.

Hence the maximum clique size = 3.

It is similar to previous example. We can see that with sorted vertices, the maximum clique will be reached more quickly.

CHAPTER 4

COMPUTER EXPERIMENTAL RESULTS

In this chapter, computer experimental results of the maximum clique problem when solved by the four algorithms in Chapter 3 are presented. The two quantities of interest here are the average CPU time used to obtain an optimal solution and average maximum clique number of the optimal solution.

To see how good each algorithm in Chapter 3 is in solving the maximum clique problem, algorithm 1 to 4 are applied to random graphs with various numbers of vertices and different values of edge densities. Computer programs for all algorithms are coded in C^{++} language on a 1.8 GHz Pentium M laptop computer with 768 MB RAM.

Section 4.1 shows the results of the four algorithms on the average CPU time to an optimal solution and then Section 4.2 on the average maximum clique size. Finally, in Section 4.3, a summary and discussion of all the relevant results are provided.

4.1 Average CPU Time to a Maximum Clique

Computer experiments for solving the maximum clique problem by algorithms 1 to 4 have been conducted. Table 4.1 shows the results on the average CPU time to an optimal solution in random graphs for different numbers of vertices– 5, 10, 25, 50, 100, and 200— and different values of edge density—0.1, 0.2, 0.5, 0.8, and 0.9 (0.9 is skipped for the case of 200 vertices due to its complexity). To understand every aspect of the results better, various graphs are plotted in Figures 4.1 to 4.6

Table 4.1 Average CPU Time to a Maximum Clique When Solved by Algorithm 1 to 4 in Random Graphs with Different Numbers of Vertices and Different Values of Edge Density

Vertices	Edge Density	Average CPU Time (Seconds)			
		Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
5	0.1	0.000030	0.000009	0.000001	0.000001
	0.2	0.000048	0.000017	0.000007	0.000001
	0.5	0.000077	0.000049	0.000017	0.000004
	0.8	0.000082	0.000059	0.000025	0.000005
	0.9	0.000119	0.000073	0.000051	0.000009
10	0.1	0.000173	0.000100	0.000073	0.000013
	0.2	0.000203	0.000197	0.000159	0.000075
	0.5	0.000252	0.000202	0.000197	0.000102
	0.8	0.000300	0.000288	0.000210	0.000139
	0.9	0.000800	0.000499	0.000307	0.000201
25	0.1	0.000806	0.000531	0.000441	0.000294
	0.2	0.001000	0.000747	0.000500	0.000417
	0.5	0.004600	0.001906	0.000981	0.000500
	0.8	0.081500	0.009482	0.003220	0.000529
	0.9	0.707400	0.083146	0.010180	0.000779
50	0.1	0.003200	0.001900	0.001239	0.000802
	0.2	0.006700	0.002300	0.001700	0.000927
	0.5	0.084000	0.004530	0.002420	0.001040
	0.8	22.205072	0.036634	0.004370	0.002700
	0.9	558.863600	0.551210	0.013278	0.007500
100	0.1	0.030200	0.002400	0.002000	0.003540
	0.2	0.067800	0.059400	0.005152	0.020200
	0.5	3.210400	0.160240	0.011900	0.033700
	0.8	949.599100	6.027900	1.964358	0.040700
	0.9	3156.749600	417.359448	18.785400	0.052100
150	0.1	0.081600	0.004200	0.003600	0.007840
	0.2	0.591600	0.125080	0.024342	0.060400
	0.5	34.278208	0.156900	0.038596	0.097500
	0.8	4907.210400	58.003200	34.491617	0.157500
	0.9	11452.980700	8993.601200	1829.406020	0.178000
200	0.1	0.202800	0.009900	0.007828	0.016982
	0.2	4.009500	0.513200	0.072464	0.131300
	0.5	468.734900	1.349800	0.158844	0.141600
	0.8	51782.398200	1252.552400	827.210100	0.497100

Figures 4.1 through 4.4 report the average CPU time to a maximum clique as a function of edge density in random graphs with 5, 10, 25, and 50, respectively.

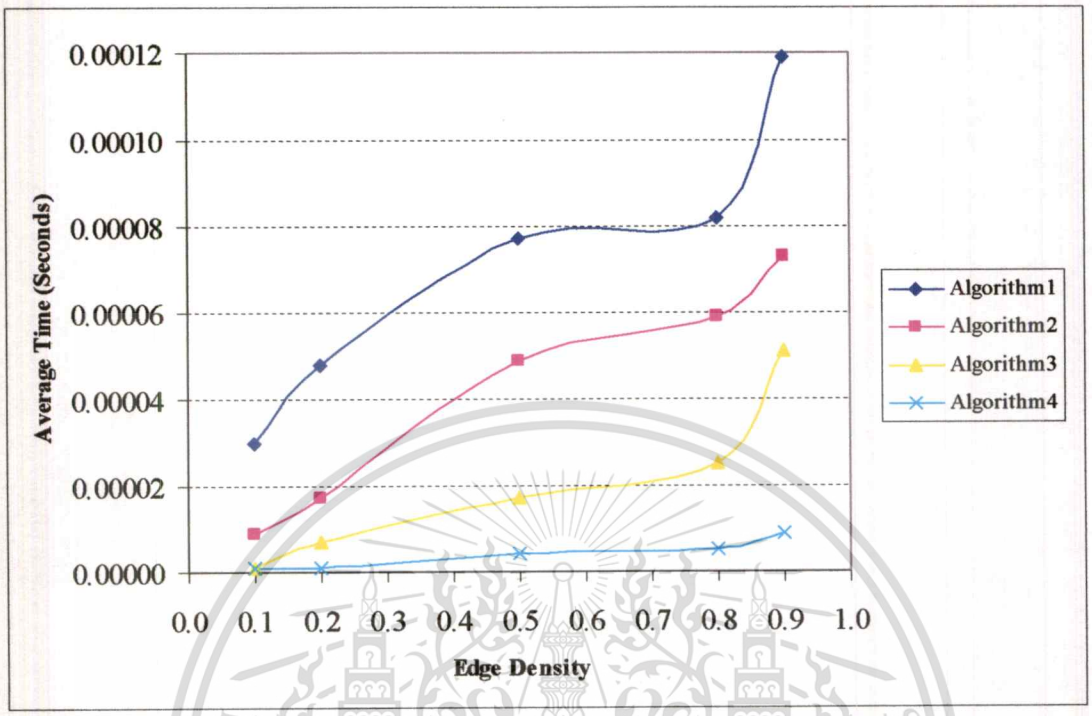


Figure 4.1: Average CPU Time to a Maximum Clique When Solved by Algorithm 1 to 4 as a Function of Edge Density in Random Graphs with 5 Vertices.

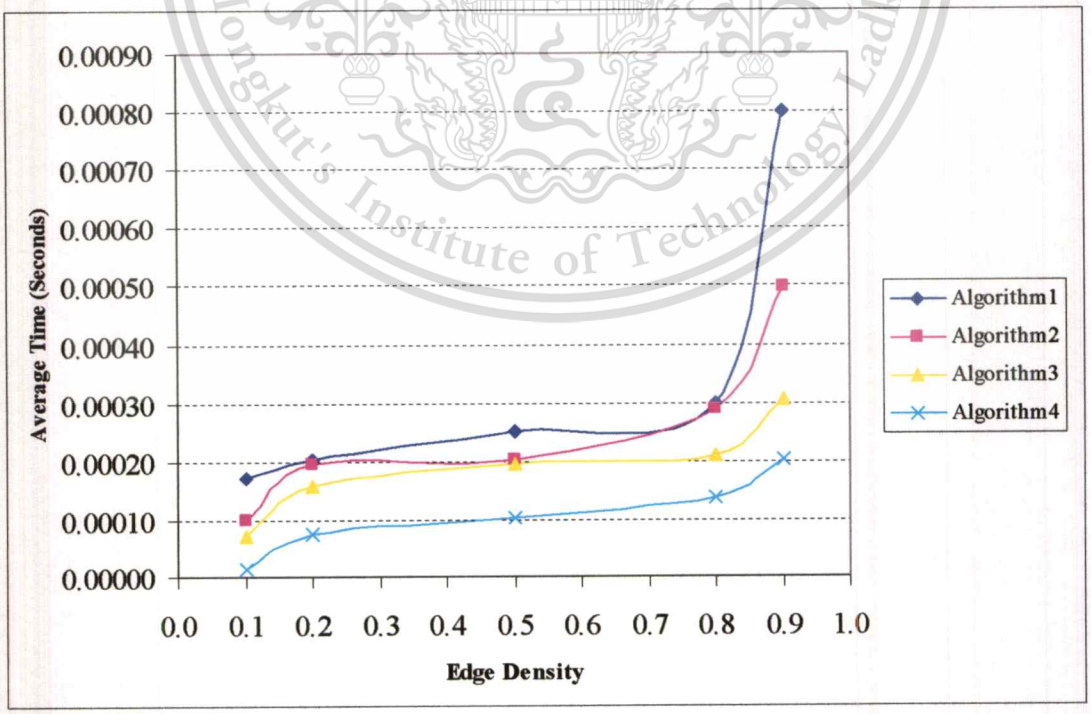


Figure 4.2: Average CPU Time to a Maximum Clique When Solved by Algorithm 1 to 4 as a Function of Edge Density in Random Graphs with 10 Vertices.

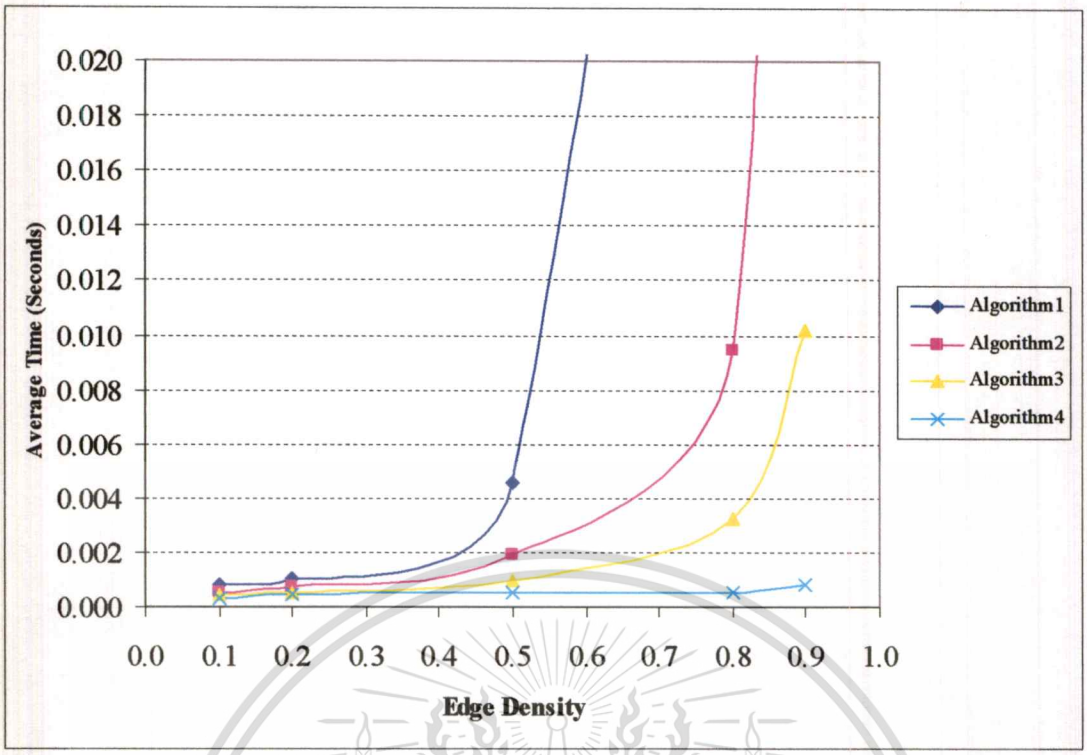


Figure 4.3: Average CPU Time to a Maximum Clique When Solved by Algorithm 1 to 4 as a Function of Edge Density in Random Graphs with 25 Vertices.

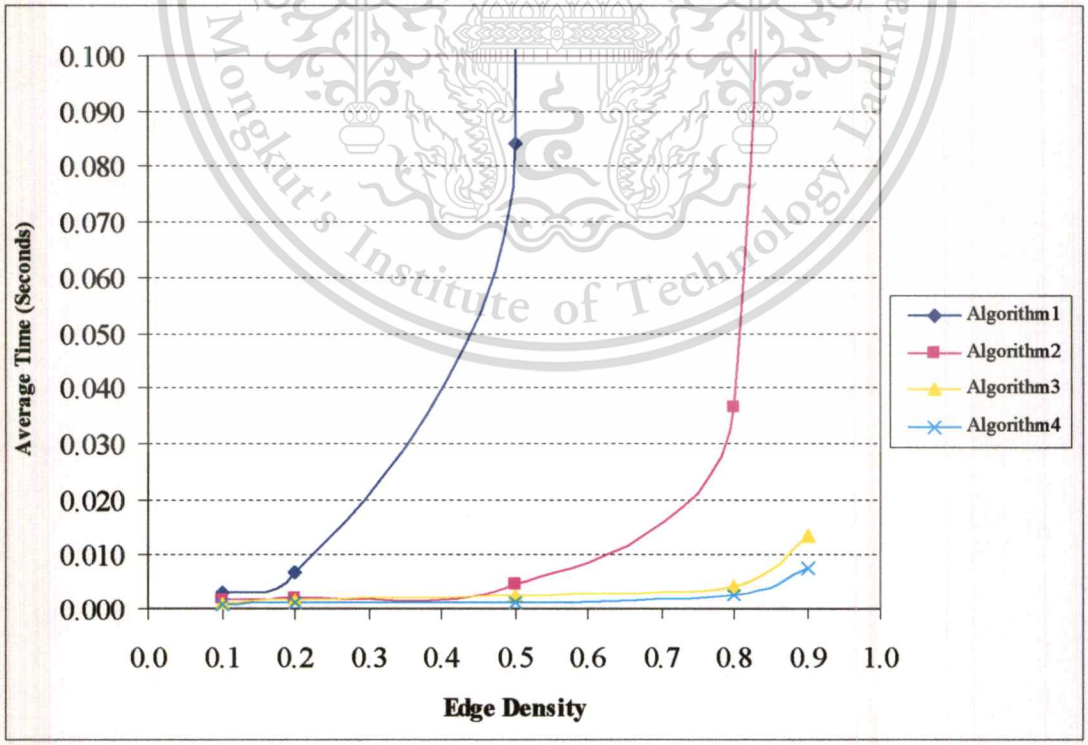


Figure 4.4: Average CPU Time to a Maximum Clique When Solved by Algorithm 1 to 4 as a Function of Edge Density in Random Graphs with 50 Vertices.

Figures 4.5 and 4.6 report the average CPU time to a maximum clique as a function of number of vertices in random graphs with edge density = 0.1 and 0.2 respectively.

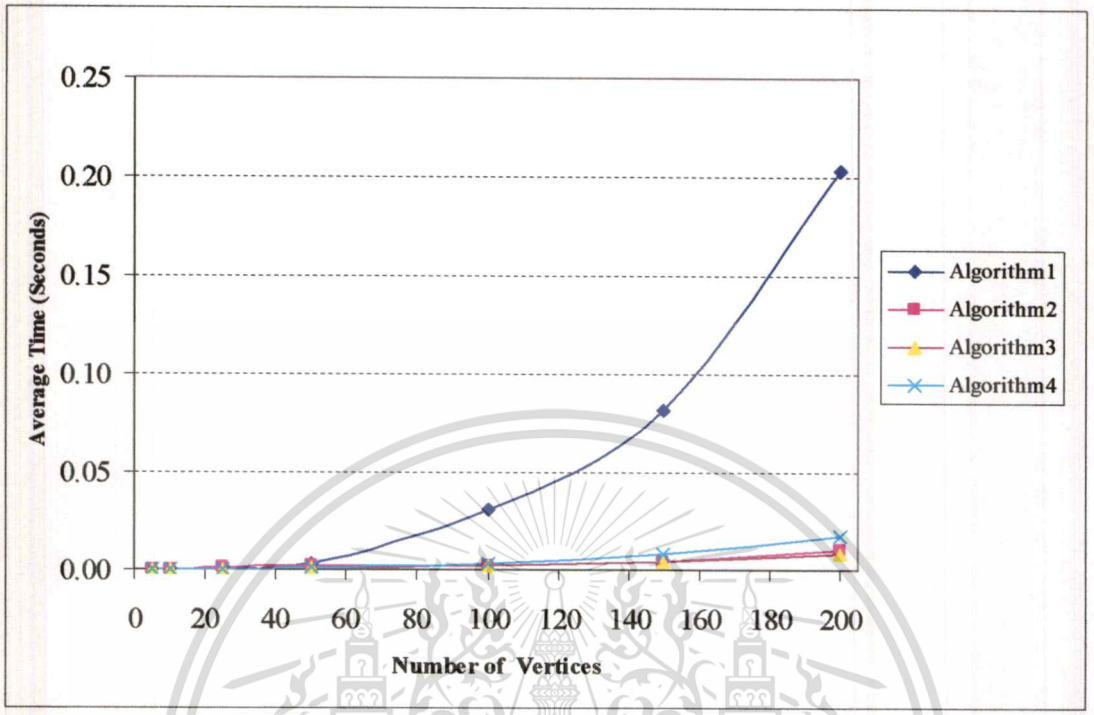


Figure 4.5: Average CPU Time to a Maximum Clique When Solved by Algorithm 1 to 4 as a Function of Number of Vertices in Random Graphs with Edge Density = 0.1.

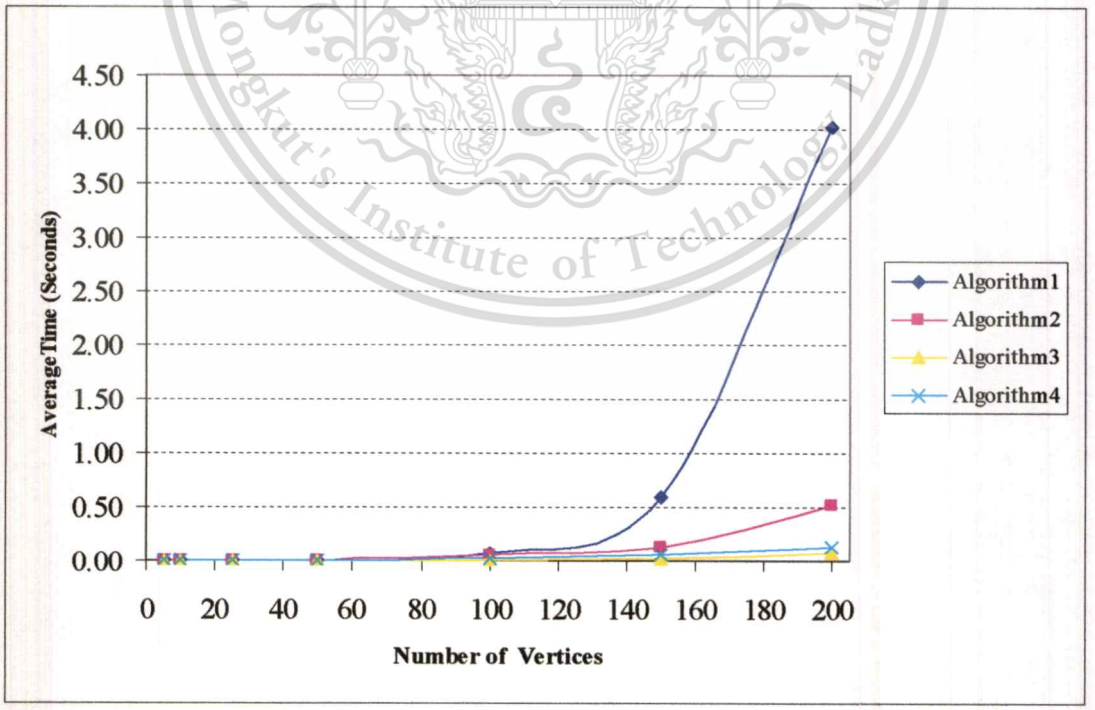


Figure 4.6: Average CPU Time to a Maximum Clique When Solved by Algorithm 1 to 4 as a Function of Number of Vertices in Random Graphs with Edge Density = 0.2.

The results in Table 4.1 and Figures 4.1 through 4.6 show that for small numbers of vertices and any values of edge density, algorithms 1 to 4 can solve the problem in reasonable and comparable time even though algorithms 3 and 4 seem to perform slightly better (see Figures 4.1 and 4.2). Notice that the values of the y-axis in all the figures are not the same. As for larger numbers of vertices, when the values of edge density remain small (such as 0.1 and 0.2 in Figures 4.3 and 4.4), the four algorithms still perform relatively good. However, when the values of edge density increase, the average time to a maximum clique enormously improve in algorithms 3 and 4. The growing rate of the average CPU time with respect to the increasing values of edge density is highest for algorithm 1 followed by algorithm 2. Meanwhile, the CPU time increases much slower for algorithms 3 and 4 even with larger numbers of vertices.

4.2 Average Maximum Clique Size of a Maximum Clique

The results of these algorithms on the average maximum clique size are presented. Table 4.2 shows the results on the average maximum clique size in random graphs for different numbers of vertices— 5, 10, 25, 50, 100 and 200 — and different values of edge density — 0.1, 0.2, 0.5, 0.8 and 0.9 (0.9 is skipped for the case of 200 vertices due to its complexity). To understand every aspect of the results better, various graphs are plotted in Figures 4.7 to 4.16

Table 4.2 Average Maximum Clique Size Obtained from Algorithm 1 to 4 in Random Graphs with Different Numbers of Vertices and Different Values of Edge Density

Vertices	Edge Density	Average Maximum Clique Size			
		Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
5	0.1	1.709	1.640	1.632	1.702
	0.2	1.970	1.950	1.974	1.970
	0.5	2.710	2.678	2.680	2.790
	0.8	3.690	3.674	3.726	3.820
	0.9	4.240	4.248	4.200	5.180
10	0.1	2.100	2.084	2.120	2.109
	0.2	2.530	2.572	2.538	2.550
	0.5	3.960	3.966	3.856	4.030
	0.8	6.080	6.052	6.066	6.100
	0.9	7.480	7.394	7.450	7.490
25	0.1	2.890	2.872	2.862	2.970
	0.2	3.510	3.470	3.484	3.620
	0.5	6.250	5.808	5.970	6.710
	0.8	10.630	10.546	10.894	11.670
	0.9	14.180	14.140	14.254	14.841
50	0.1	3.210	3.164	3.204	3.640
	0.2	4.180	4.552	4.196	6.310
	0.5	7.550	7.810	7.994	11.250
	0.8	15.860	15.140	14.960	20.000
	0.9	22.390	21.818	21.488	24.470
100	0.1	3.960	3.958	3.954	5.040
	0.2	5.030	5.034	5.042	9.730
	0.5	9.340	9.288	9.284	19.840
	0.8	20.170	20.180	20.174	34.780
	0.9	31.129	31.113	31.825	43.940
150	0.1	4.060	4.019	4.189	8.190
	0.2	5.950	5.351	5.632	12.680
	0.5	10.180	10.245	10.832	26.920
	0.8	23.460	23.779	23.006	48.190
	0.9	36.887	36.530	37.328	62.010
200	0.1	4.220	4.193	4.229	9.811
	0.2	6.728	6.523	6.420	17.080
	0.5	11.090	11.695	11.832	33.662
	0.8	29.420	29.920	28.100	61.209

Figures 4.7 through 4.11 report the average maximum clique size as a function of edge density in random graphs with 5, 10, 25, 50, and 100 vertices, respectively.

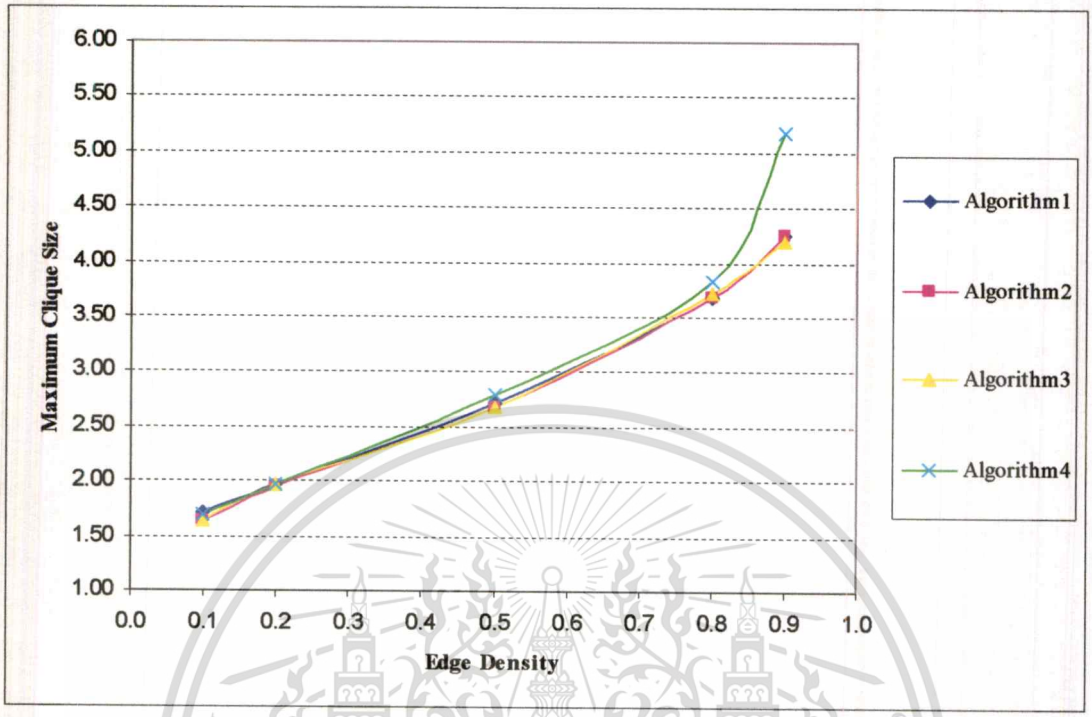


Figure 4.7: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a Function of Edge Density in Random Graphs with 5 Vertices.

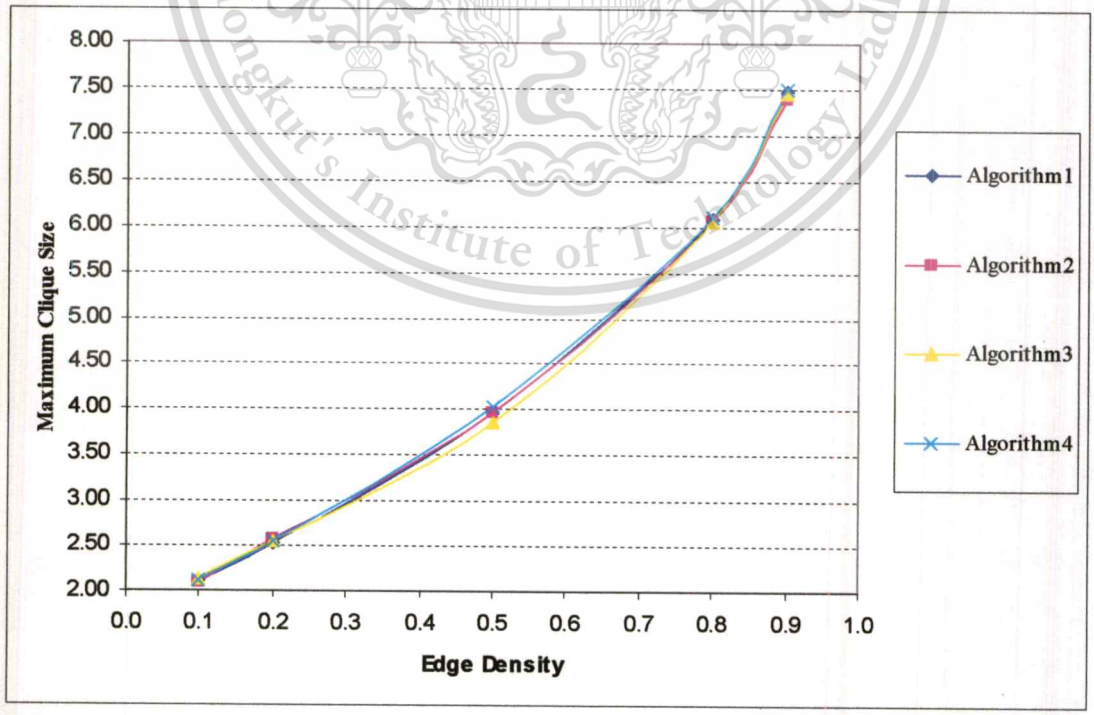


Figure 4.8: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a Function of Edge Density in Random Graphs with 10 Vertices.

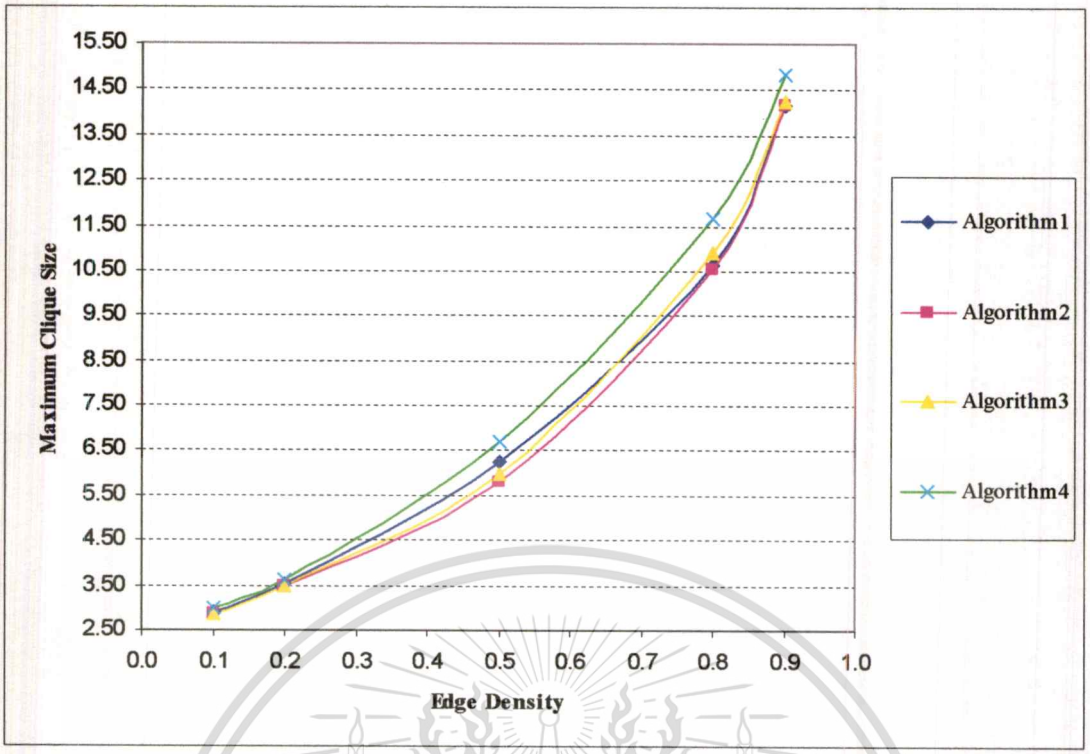


Figure 4.9: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a Function of Edge Density in Random Graphs with 25 Vertices.

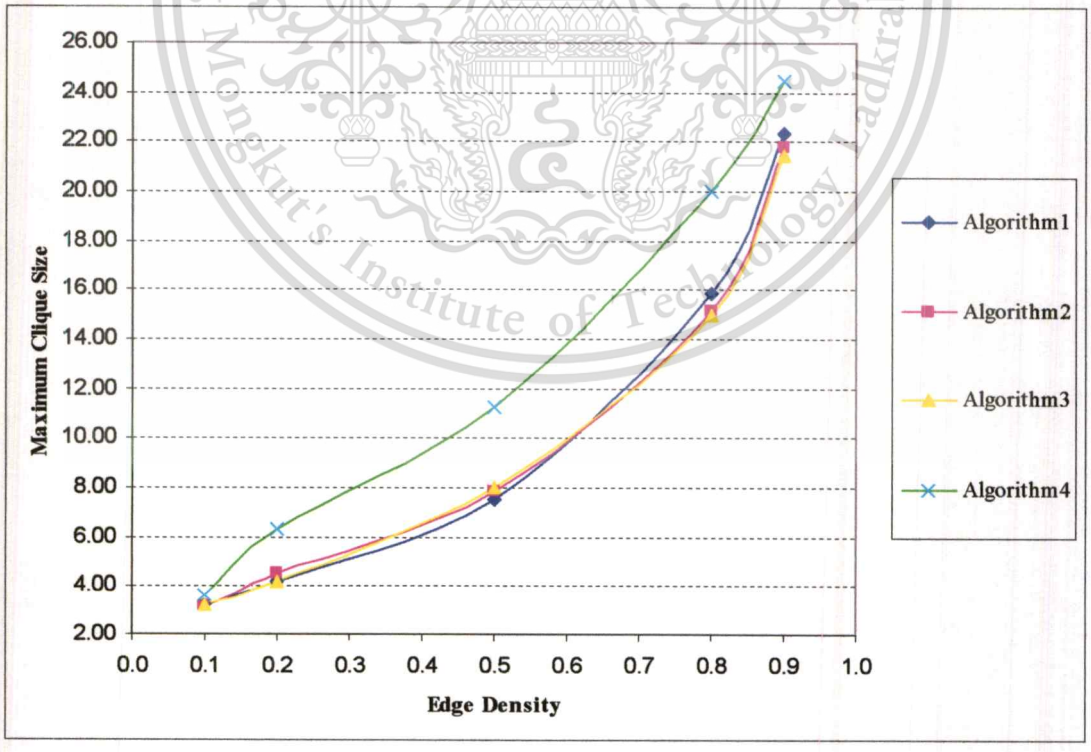


Figure 4.10: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a Function of Edge Density in Random Graphs with 50 Vertices.

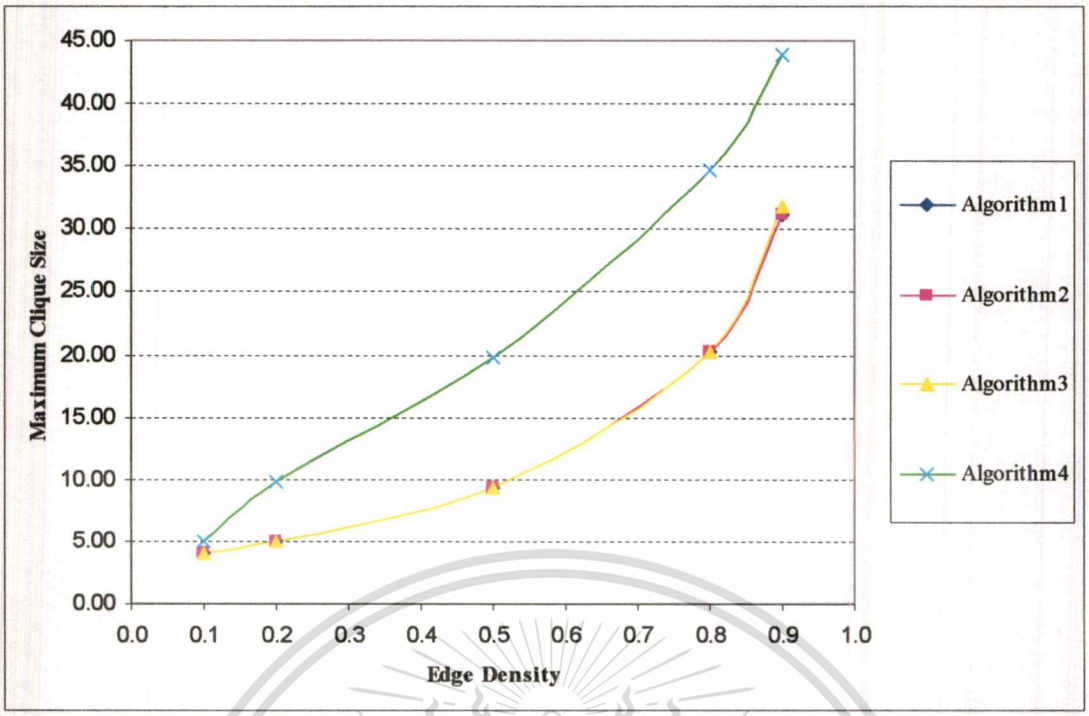


Figure 4.11: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a Function of Edge Density in Random Graphs with 100 Vertices.

Figures 4.12 through 4.16 report the average maximum clique size as a function of number of vertices in random graphs with edge density = 0.1, 0.2, 0.5, 0.8, and 0.9, respectively.

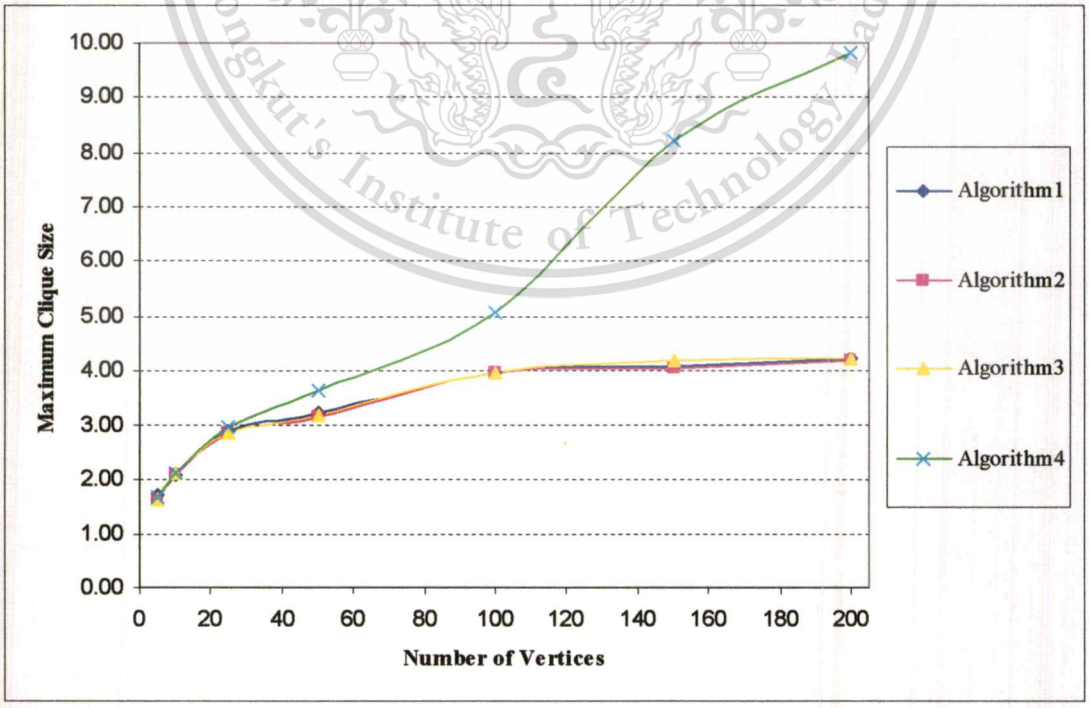


Figure 4.12: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a Function of Number of Vertices in Random Graphs with Edge Density = 0.1.

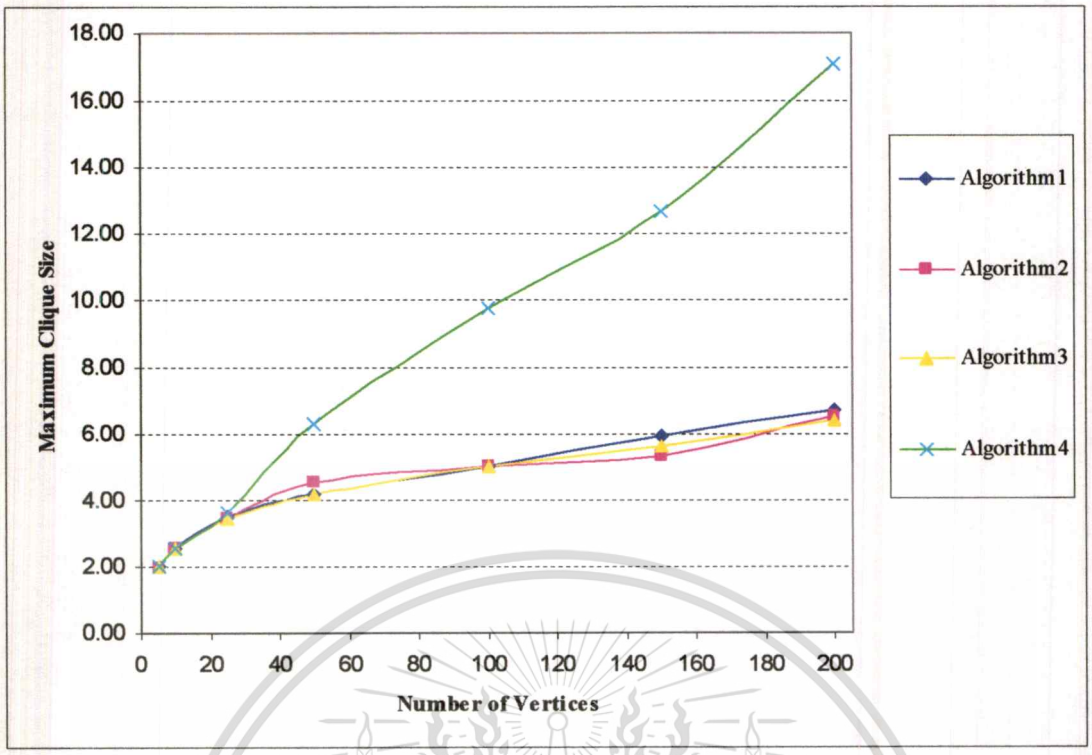


Figure 4.13: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a function of Number of Vertices in Random Graphs with Edge Density = 0.2.

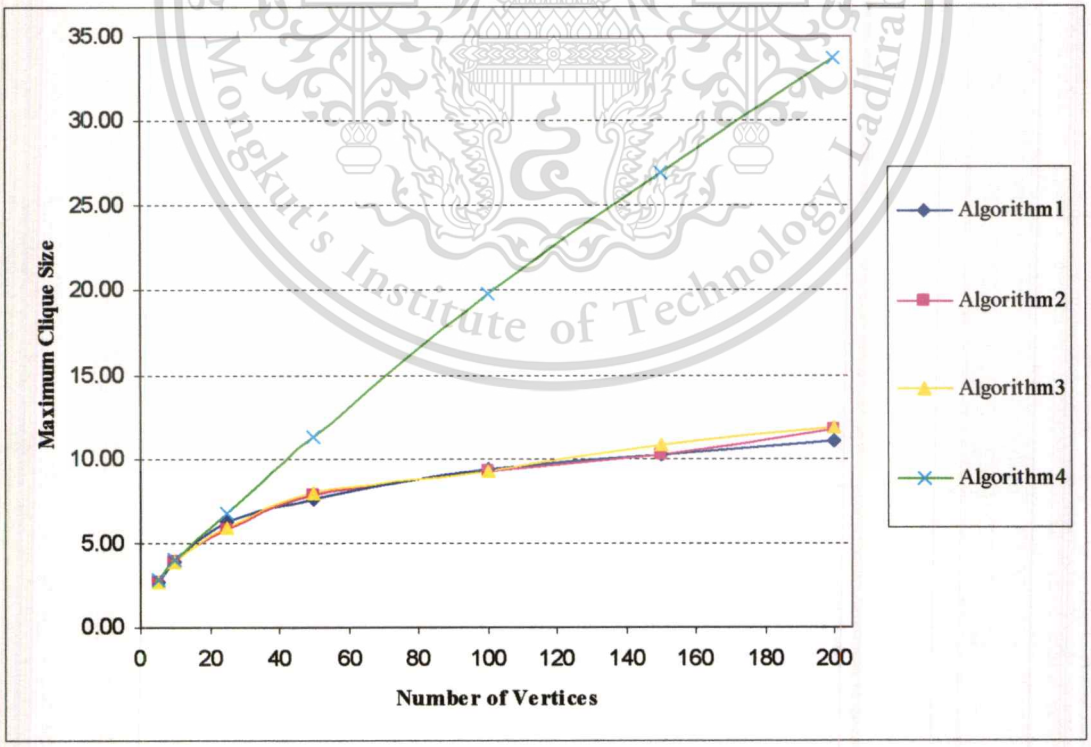


Figure 4.14: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a Function of Number of Vertices in Random Graphs with Edge Density = 0.5.

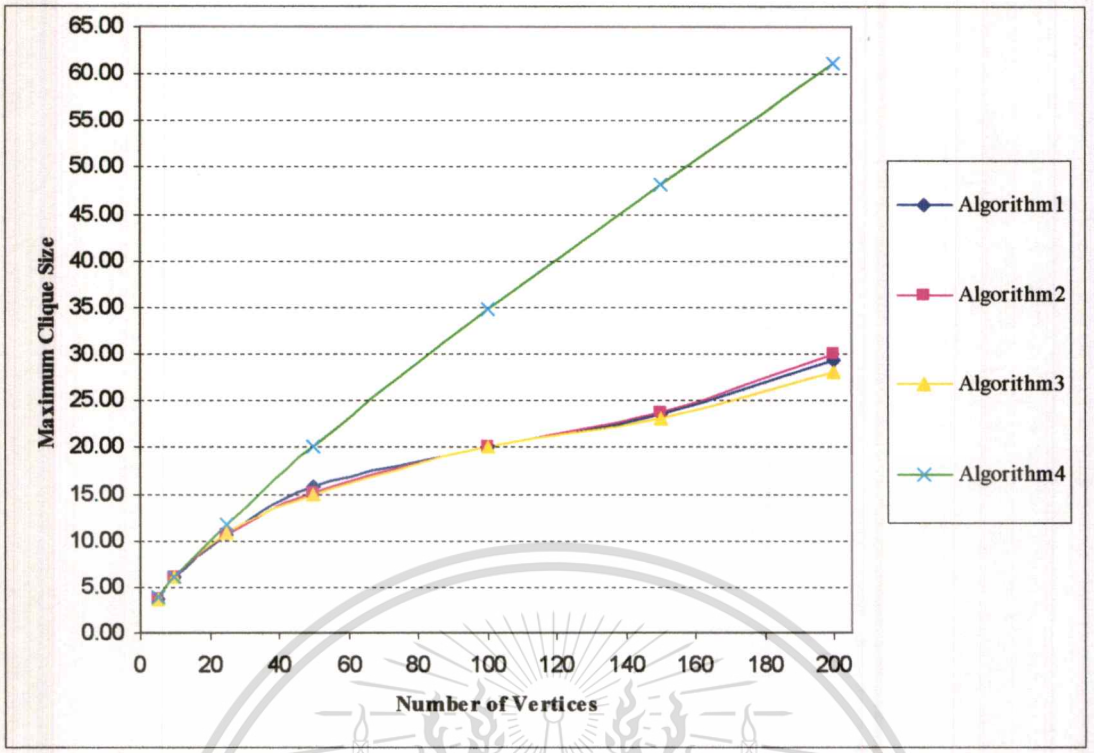


Figure 4.15: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a Function of Number of Vertices in Random Graphs with Edge Density = 0.8.

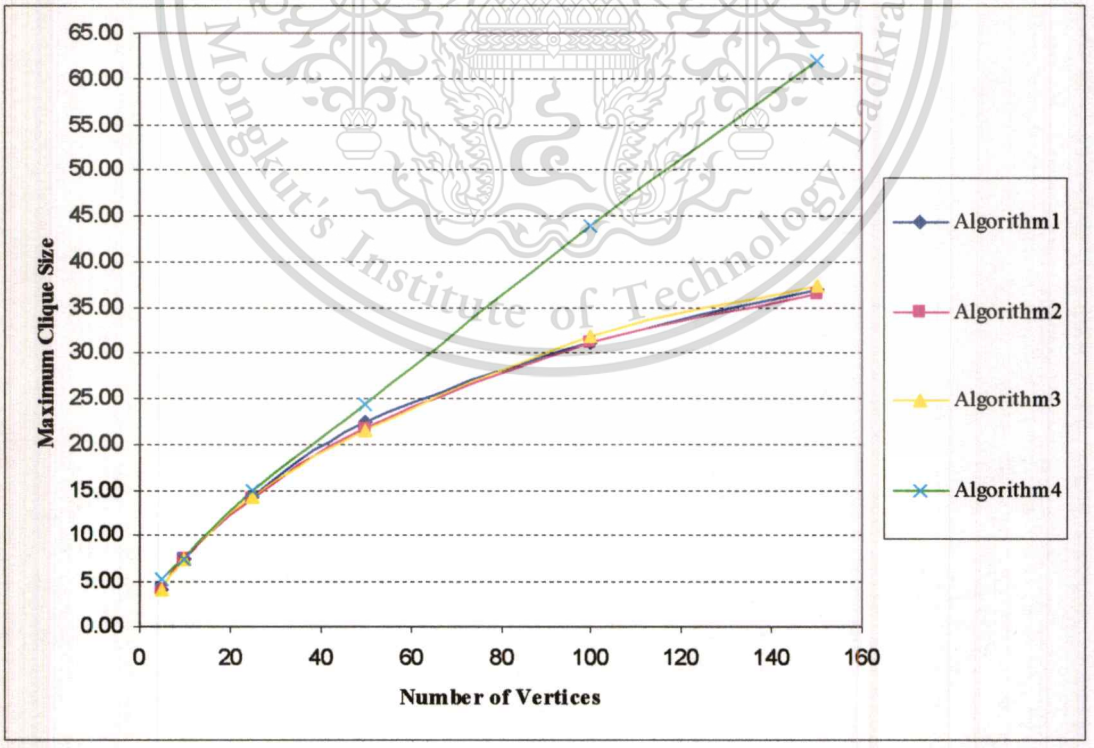


Figure 4.16: Average Maximum Clique Size Obtained from Algorithm 1 to 4 As a Function of Number of Vertices in Random Graphs with Edge Density = 0.9.

The results in Table 4.2 and Figures 4.7 through 4.16 show that for small numbers of vertices and any values of edge density the average maximum clique size obtained from the four algorithms are indifferent (see Figures 4.7 through 4.9). As for larger numbers of vertices, when the values of edge density = 0.1 (see Figures 4.10 and 4.11), the four algorithms still perform relatively good. However, when the values of edge density increase, algorithm 1, 2, and 3 still give comparable results while the curve of algorithm 4 increase faster than the curves of the other three algorithms (see Figures 4.10 and 4.11).

Similar conclusions can be drawn from Figures 4.12 through 4.16 when the average maximum clique size is plotted as a function of number of vertices for a fixed value of edge density. More precisely, algorithms 1 through 3 yield comparable results for all the cases while algorithm 4 yields similar results only when the numbers of vertices are small.

4.3 Summary and Discussion of the Results

All four algorithms tested on random graphs by using different numbers of vertices and edge density shows the results as following:

(1) Average CPU time to a maximum clique

To consider the average CPU time to solve the problem, we find that, when the numbers of vertices are fixed at 5, 10, 25, 50, 100, and 200, respectively, algorithms 1 to 4 are insignificantly different for low edge density. In the case of high edge density, the average CPU time of algorithm 1 increases dramatically, followed by algorithm 2. Algorithms 3 and 4 perform relatively well in terms of the average CPU time.

When the values of edge density are fixed at 0.1, 0.2, 0.5, 0.8, and 0.9, similar results are obtained as in the case when the numbers of vertices are fixed, that is, algorithms 3 and 4 need less time than algorithm 1 and 2.

(2) Average maximum clique size

To consider on the number of maximum clique from each algorithm in average, we find that, when the numbers of vertices are fixed at 5, 10 and 25 respectively, the algorithms 1 to 4 give quite similar results for every edge density. However, when the numbers of vertices increase, algorithm 4 give higher average maximum clique size compared to algorithm 1, 2 and 3 which are quite similar among three of them.

When the values of edge density are fixed at 0.1, 0.2, 0.5, 0.8, and 0.9, all four algorithms give similar results when vertices are less than 40. On the contrary, when the numbers of vertices increase, the average maximum clique sizes obtained from algorithms 1, 2 and 3 are approximately equal while those obtained from algorithm 4 keep increasing.

Referred to both cases as above mentioned, it is obvious that by considering the average CPU time to a maximum clique, algorithm 4 works the fastest, then algorithm 3, and algorithms 1 and 2, respectively. When considering the average maximum clique size, the algorithm 4 yields the result not so far from other three algorithms on the condition that vertices and edge density are small. If vertices and edge density become higher, algorithm 4 may not give as good result while the algorithm 3 still give good result compared to the remaining algorithms on the condition that vertices and edge density are either more or less.

In conclusion, compared to algorithms 1 and 2, algorithm 3 works faster and algorithm 4 works the fastest. As described in the Chapter 3, by looking at the minimum number of vertex colors obtained from algorithm 4 as the maximum clique size, the minimum number of vertex colors may be the least upper bound of maximum clique size, that is, equal to the maximum clique size only in some cases.

CHAPTER 5

CONCLUSIONS AND SUGGESTIONS

5.1 Conclusions

This research examines the maximum clique problem, a classical problem in combinatorial optimization. Its applications can be found in various areas. A clique of a graph G = (V,E) is a set of vertices, any two of which are adjacent. The maximum clique problem looks for a clique having a maximum number of nodes in the clique, that is, a largest clique.

Since this is NP-hard problems [1], no polynomial time algorithms are expected to be found. However, heuristics algorithms have been proposed for solving the problem. Two of these previous algorithms are examined in details. Then, a proposed algorithm modified from these previous algorithms are proposed and illustrated. It is hoped that the proposed algorithm will outperform the previous two algorithms, especially on a certain type of graphs, namely, random graphs. In addition, for comparison purposes, an algorithm for solving the minimum vertex coloring problem is used as an approximation solution for the maximum clique problem.

The ideas behind each algorithm are summarized again here.

Algorithm 1 (Carraghan and Pardalos)

Given an arbitrary graph, the algorithm basically starts at a vertex to see how large a maximal clique size can be identified if that vertex is included in the clique. Repeat the process for other vertices until the last vertex is considered. In the mean time, after each iteration, update the maximum clique size found so far.

Algorithm 2 (Östergård)

This algorithm starts at a subgraph that has one vertex to see how large a clique can be. Then insert another vertex and consider again the maximal clique size in the induced subgraphs with those vertices. Repeat the process until the last vertex is inserted.

Algorithm 3

This algorithm involves reordering the vertices with respect to their degrees (the number of *incident* edges). Additional notation to be used is defined here. Let $S_i = \{v_1, v_2, \ldots, v_i\}$ where $\deg_G(v_1) \geq \deg_G(v_2) \geq \ldots \geq \deg_G(v_i) \geq \ldots \geq \deg_G(v_n)$. This improved algorithm searches for the maximum clique by first considering S_1 then S_2, and so on.

Algorithm 4

This algorithm starts at initialize parameter i and color c to be assigned to a vertex v_i. Then assign the minimum possible color c to a vertex v_i. Repeat the process until the last vertex is assigned.

After these four algorithms have been coded,

(i) Algorithm 3 absolutely outperform algorithm 1, 2 for random graphs according to the average the CPU time to a local maximum clique. Note that the maximum clique size obtained from algorithms 1, 2 and 3 are insignificantly different.

(ii) In terms of the average CPU time to a local maximum clique, algorithm 4 is the best. However, the average maximum clique size is really far from that of the remaining three algorithms. This is because algorithm 4 is meant for the minimum vertex coloring problem, not the maximum clique problem.

5.2 Limitations and Suggestions

In this research, all the algorithms are applied to one certain type of graphs —random graphs. Therefore, the results obtained here are compared based on this fact. It is natural to try all the algorithms on other types of graphs and then compare and contrast the results.

In algorithm 3, there are other ways to reorder the vertices such as insertion sort, merge sort, or bubble sort. These different sorting methods may affect the efficiency and the results of the algorithms.

The number of minimum colors obtained from algorithm 4 can be used in another kind of algorithms for finding the maximum clique. More precisely, if such algorithms start at a highest-degree vertex and work downward. A vertex having the degree equal to the number of minimum colors can be used as a starting vertex instead. This definitely improves the efficiency of such algorithms.

REFERENCES

[1] Garey, M.R. and Johnson, D.S. **Computers and Intractability; A Guide to the Theory of NP-Completeness.** New York: W. H. Freeman. 1979.

[2] Carraghan, R. and Pardalos, P.M. "An Exact Algorithm for the Maximum Clique Problem." **Operation Research Letters,** vol. 9, 1990. pp. 375-382.

[3] Östergård, P.R.J. "A Fast Algorithm for the Maximum Clique Problem." **Discrete Applied Mathematics**, vol. 120, 2002. pp. 195-205.

[4] Pardalos, P.M. and Xue, J. "The Maximum Clique Problem." **Journal of Global Optimization**, vol. 4, 1994. pp. 301–328.

[5] Pardalos, P.M. and Rodgers, G.P. "A Branch and Bound Algorithm for the Maximum Clique Problem." **Computers and Operations Research**, vol. 19, 1992. pp. 363-375.

[6] Wood, D.R. "An Algorithm for finding a maximum clique in a graph." **Operation Research Letters**, vol. 21, 1997. pp. 211–217.

[7] Fahle, T. 2002. "Simple and fast: Improving a branch-and-bound algorithm for maximum clique." 485-498. in ESA 2002, **10th Annual European Symposium**. R. Möring and R. Raman, editors. Rome. Italy.

[8] Régin, J.C. 2003. "Solving the Maximum Clique Problem with Constraint Programming." 634-648. in CP'03, **Ninth International Conference on Principles and Practice of Constraint Programming**. Francesca Rossi, editors. Kinsale, Ireland. Springer-Verlag Heidelberg.

[9] Brassard, G. and Bratley, P. **Algorithmics Theory and Practice**. New Jersey : Prentice-Hall. 1987.

[10] Neapolitan, R. and Naimipour, K. **Foundations of Algorithms**. Toronto : D.C. Heath. 1996.

[11] Balas, E. and Xue, J. "Weighted and Unweighted Maximum Clique Algorithms with Upper Bounds from Fractional Coloring." **Algorithmica**; vol. 15, 1996. pp. 397-412.

[12] Gould, R. **Graph Theory**. California : The Benjamin/Cummings Publishing Company. 1988.

APPENDIX

Source Codes of Proposed Algorithms for Solving the Maximum Clique Problem

1. Source Codes of Algorithm 3

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

#define NMAX 1000     /* maximum number of vertices handles */
#define FMAX 500      /* maximum number of files handle for each edge density */
#define TIMEUSES(t) ((clock()-( t ) )/(double)CLOCKS_PER_SEC)
#define REORDER
int edge[NMAX][NMAX];          /* array 2 dimension to keep edge status */
int total_set_node;
int set_node[NMAX];
int N;                         /* number of vertices in graph */
int setlim;                    /* size at which exhaustive search begins */
int vertex[NMAX];              /* array in which vertices reside and are moved */
int degree[NMAX];              /* vertex degrees with resp. to uncolored vertices */
int set[NMAX];                 /* non-zero entries are members of current set */
int bestset[NMAX];             /* non-zero entries constitute currently best set */
int bestsize;                  /* size of current best set */
int loadfile();
```

```
int main (void)
{
        FILE *file ;
        int i,j,k,cand,newcand,dmin;
        double timeuse,totaltime,totalmc,avgmc,avgtime;
        clock_t begin;
        if ((file = fopen("output_ c.txt","w+"))== NULL)
        {
                fprintf(stderr, "Cannot open output file.\n");
                return 1;
        }
        totaltime = 0;totalmc=0;
        for(k=1;k<=FMAX;k++)
        {
                loadfile(k);
                setlim = 1;
                begin=clock();
                /* Sorted the vertices */
                #ifdef  REORDER
                for (i=1;i<=N;i++)
                {
                        degree[i] = 0;
                        for (j=1;j<=N;j++)
                                if (edge[i][j]==1)
                                        degree[i]++;
                }
                dmin = 1000;
                for (i=1;i<=N;i++)
                        if (degree[i] < dmin)
                        {
                                dmin = degree[i];
                                cand = i;
                        }
                vertex[N] = cand;
```

```
for (j=N-1;j>=1;j--)
{
        degree[cand] =1000;
        dmin = 1000;
        for (i=1;i<=N;i++)
        {
                if (edge[cand][i]==1)
                        degree[i]--;

                if (degree[i] < dmin)
                {
                        dmin = degree[i];
                        newcand = i;
                }
        }
        vertex[j] = cand = newcand;
}
#else
for (j=1;j<=N;j++)
        vertex[j] = set_node[j-1];
#endif
printf("\nAfter REORDER \n Total set node = %d\n",total_set_node);
for(i=1;i<=total_set_node;i++)
        printf(" %d ,",vertex[i]);
bestsize = 0;
bestsize = maxind(N,setlim,vertex,1);
timeuse=TIMEUSES(begin);
printf("\nThe maximum clique = %d",bestsize);
printf("\nVertices in maximum clique : ");
for (i=1; i<=bestsize; i++) {
   printf (" %d",bestset[i]);
}
printf ("\n");
printf("The time was : %0.10f second \n",timeuse);
```

```
                fprintf(file,"rd%d : max-clique = %d , timeuse = %0.10f seconds
                \n",k,bestsize,timeuse);
                totaltime = totaltime+timeuse;
                totalmc = totalmc + bestsize;
        }
        fprintf(file,"Total %d files\n",FMAX);
        fprintf(file,"----------\n");
        avgtime = totaltime/FMAX;
        avgmc = totalmc/FMAX;
        printf("\nThe average time was : %0.10f second \n",avgtime);
        printf("\nThe average mc was : %0.03f vertices \n",avgmc);
        fprintf(file,"Average maximum clique = %0.03f vertices \n
        Average time = %0.10f seconds \n",avgmc,avgtime);
        fclose(file);
        return 0;
}
int loadfile(int fileno)
{
        FILE *fp;
        struct list *newnode;
        const char  deli[]=" \n";
        const char filename[20] = "rd";
        const char *path = "\\rdg3\\rd100_8\\";
        char graphname[20],newname[20];
        char str[100];
        int  i,j,k,row,col,found;
        char *token;
        char array[256];
        char str_node[99] ;
        strcpy(newname,filename);
        itoa(fileno,str,10);
        strcat(newname,str);
        strcat(newname,".txt");
        strcpy(graphname,path);
```

```
strcat(graphname,newname);
printf("\n%s\n",graphname);
if ((fp = fopen(graphname,"rb"))== NULL)
{
        fprintf(stderr, "Cannot open test file.\n");
        return 1;
}
i=0;
/* initial edge list */
/* edeg[i][j] = 0 : mean no edge connect between node i and j */
/* edeg[i][j] = 1 : mean has an edge connect between node i and j */
for (i=0;i<NMAX;i++)
        for (j=0;j<NMAX;j++)
                edge[i][j] = 0;
while(fgets(array, 256, fp) != NULL)
{
        token = strtok(array, deli);
        if(token != NULL)
                row = atoi(token);
        while(token != NULL)
        {
                token = strtok(NULL,deli);
                if(token != NULL)
                {
                        col = atoi(token);
                        edge[row][col] = 1;
                        edge[col][row] = 1;
                        /* found = 0 : not found node x in set_node
                        found = 1 : found node x in set_node */
                        i = 0 ; found=0;
                        while ((i<total_set_node)&&(found == 0))
                        {
                                if (set_node[i]== row)
                                        found = 1;
```

```
                    i++;
                }
                if (found ==0)
                {
                    set_node[total_set_node] = row;
                    total_set_node++;
                }
                found = 0;   i=0;
                if (row != col)
                {
                    while ((i<total_set_node)&&(found == 0))
                    {
                        if (set_node[i]== col)
                            found = 1;
                        i++;
                    }
                    if (found ==0)
                    {
                        set_node[total_set_node] = col;
                        total_set_node++;
                    }
                }
            }
        }
    }
    fclose(fp);
    N = total_set_node;
    return 0;
}
```

```
int maxind(top,goal,array,depth)
register int top, goal,depth;
int *array;
{
        int newarray[NMAX];
        int i,v,u,w,z;
        int best, restbest, newgoal;
        int *pnew, *pold;
        int canthrow;


        if (top <= 1)
        {
                if (top == 0) depth--;
                if (depth > bestsize)
                {
                        bestsize = depth;
                        if (top == 1) set[bestsize] = array[top];
                        for (i=1;i<=bestsize;i++) bestset[i] = set[i];
                }
                return(top);
        }
        best = 1;
        newgoal = goal-1;
        if (newgoal <= 1) newgoal = 1;
        for (i = top; i >= goal; i--)
        {
                pnew = newarray;
                w = array[i];
                set[depth] = w;
                canthrow = i - goal;
                pold = array+1;
                while (pold<array+i)
                {
```

```
                z = *pold++;
                if (edge[z][w]==1)
                        *++pnew = z;
                else
                {
                   if (canthrow == 0) goto breakout;
                   canthrow--;
                }
        }
        restbest = maxind(pnew-newarray,newgoal,newarray,depth+1);
        if (restbest >= newgoal)
        {
                best = newgoal = restbest+1;
                goal = best+1;
        }
        breakout:;
    }
    return(best);
}
```

2. Source Codes of Algorithm 4

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#define NMAX 1000      /* maximum number of vertices handles */
#define FMAX 500       /* maximum number of files handle for each edge density */
#define TIMEUSES(t)  ((clock()-( t ))/(double)CLOCKS_PER_SEC)
#define REORDER

int edge[NMAX][NMAX];          /* array 2 dimension to keep edge status */
int newedge[NMAX][NMAX];
int total_set_node;
int set_node[NMAX];
int N;                         /* number of vertices in graph */
int setlim;                    /* size at which exhaustive search begins */
int vertex[NMAX];              /* array in which vertices reside and are moved */
int degree[NMAX];              /* vertex degrees */
int set[NMAX];                 /* non-zero entries are members of current set */
int bestset[NMAX];             /* non-zero entries constitute currently best set */
int bestsize;                  /* size of current best set */
int loadfile();
int int_comp(const void *n,const void *m);
int coling();
int delcolors(int *availablecolors,int assign,int N);
int empty(int *availablecolors,int N);
int numof(int *availablecolors,int N);
int numof2(int *availablecolors,int N);
```

```
int main (void)
{
        FILE *file ;
        int i,j,k,cand,newcand,dmin,color = 0;
        double timeuse,totaltime,totalmc,avgmc,avgtime;
        clock_t begin;
        if ((file = fopen("output_d.txt","w+"))== NULL)
        {
                fprintf(stderr, "Cannot open output file.\n");
                return 1;
        }
        totaltime = 0;totalmc = 0;
        for(k=1;k<=FMAX;k++)
        {
                loadfile(k);
                setlim = 1;
                #ifdef  REORDER
                for (i=1;i<=N;i++)
                {
                        degree[i] = 0;
                        for (j=1;j<=N;j++)
                                if ((edge[i][j]==1)&&(edge[j][i]==1))
                                        degree[i]++;
                }
                dmin = 1000;
                for (i=1;i<=N;i++)
                        if (degree[i] < dmin)
                        {
                                dmin = degree[i];
                                cand = i;
                        }
                vertex[N] = cand;
```

```
for (j=N-1;j>=1;j--)
{
        degree[cand] =1000;
        dmin = 1000;
        for (i=1;i<=N;i++)
        {
                if ((edge[cand][i]==0)&&(edge[i][cand]==0))
                        degree[i]--;
                if (degree[i] < dmin)
                {
                        dmin = degree[i];
                        newcand = i;
                }
        }
        vertex[j] = cand = newcand;
}
#else
for (j=1;j<=N;j++)
        vertex[j] = set_node[j-1];
#endif
/*Create new edge list of ordering vertices */
for (i=1;i<=N;i++)
{
        for (j=1;j<=N;j++)
        {
                newedge[i][j] = edge[vertex[i]][vertex[j]];
        }
}
begin=clock();
color = coling();
timeuse=TIMEUSES(begin);
printf("This graph use %d colors\n",color);
printf("The time was : %0.10f second \n",timeuse);
fprintf(file,"rd%d : max-clique = %d , timeuse = %0.10f seconds
```

```
                \n",k,color,timeuse);
                totaltime = totaltime+timeuse;
                totalmc = totalmc + color;
        }
        fprintf(file,"Total %d files\n",FMAX);
        fprintf(file,"----------\n");
        avgtime = totaltime/FMAX;
        avgmc = totalmc/FMAX;
        printf("\nThe average time was : %0.10f second \n",avgtime);
        printf("\nThe average mc was : %0.03f vertices \n",avgmc);
        fprintf(file,"Average maximum clique = %0.03f vertices \n
        Average time = %0.10f seconds \n",avgmc,avgtime);
        fclose(file);
        return 0;
}
int loadfile(int fileno)
{
        FILE *fp;
        struct list *newnode;
        const char  deli[]=" \n";
        const char filename[20] = "rd";
        const char *path = "\\marujung\\program\\rdg4\\rd10_9\\";
        char graphname[20],newname[20];
        char str[100];
        int  i,j,k,row,col,found;
        char *token;
        char array[256];
        char str_node[99] ;
        strcpy(newname,filename);
        itoa(fileno,str,10);
        strcat(newname,str);
        strcat(newname,".txt");
        strcpy(graphname,path);
        strcat(graphname,newname);
```

```
printf("\n%s\n",graphname);
if ((fp = fopen(graphname,"rb"))== NULL)
{
        fprintf(stderr, "Cannot open test file.\n");
        return 1;
}
i=0;
/* initial edge list */
/* edeg[i][j] = 0 : mean no edge connect between node i and j */
/* edeg[i][j] = 1 : mean has an edge connect between node i and j */
for (i=1;i<=NMAX;i++)
        for (j=1;j<=NMAX;j++)
                edge[i][j] = 0;
while(fgets(array, 256, fp) != NULL)
{
        token = strtok(array, deli);
        if(token != NULL)
                row = atoi(token);
        while(token != NULL)
        {
                token = strtok(NULL,deli);
                if(token != NULL)
                {
                        col = atoi(token);
                        edge[row][col] = 1;
                        edge[col][row] = 1;
                        /* found = 0 : not found node x in set_node
                        found = 1 : found node x in set_node.*/
                        i = 0 ; found=0;
                        while ((i<total_set_node)&&(found == 0))
                        {
                                if (set_node[i]== row)
                                        found = 1;
                                i++;
```

```
                    }
                    if (found ==0)
                    {
                            set_node[total_set_node] = row;
                            total_set_node++;
                    }
                    found = 0;   i=0;
                    if (row != col)
                    {
                            while ((i<total_set_node)&&(found == 0))
                            {
                                    if (set_node[i]== col)
                                            found = 1;
                                    i++;
                            }
                            if (found ==0)
                            {
                                    set_node[total_set_node] = col;
                                    total_set_node++;
                            }
                    }
            }
        }
    }
    fclose(fp);
    N = total_set_node;
    return 0;
}
```

```
int maxind(top,goal,array,depth)
register int top, goal,depth;
int *array;
{
        int newarray[NMAX];
        int i,v,u,w,z;
        int best, restbest, newgoal;
        int *pnew, *pold;
        int canthrow;

        if (top <= 1) {
                if (top == 0) depth--;
                if (depth > bestsize)
                {
                        bestsize = depth;
                        if (top == 1) set[bestsize] = array[top];
                        for (i=1;i<=bestsize;i++) bestset[i] = set[i];
                }
                return(top);
        }
        best = 1;
        newgoal = goal-1;
        if (newgoal <= 1) newgoal = 1;
        for (i = top; i >= goal; i--)
        {
                pnew = newarray;
                w = array[i];
                set[depth] = w;
                canthrow = i - goal;
                pold = array+1;
                while (pold<array+i)
                {
                        z = *pold++;
```

```
                if (edge[z][w]==1)

                        *++pnew = z;
                else
                {
                   if (canthrow == 0) goto breakout;
                   canthrow--;
                }
        }
        restbest = maxind(pnew-newarray,newgoal,newarray,depth+1);
        if (restbest >= newgoal)
        {
                best = newgoal = restbest+1;
                goal = best+1;
        }
        breakout:;
    }
    return(best);
}
int empty(int *availablecolors,int N)
{
    int i;
    for(i=1;i<=N;i++)
    {
        if (availablecolors[i]!=0)
        {
                return 1;
        }
    }
    return 0;
}
```

```
int numof(int *availablecolors,int N)
{
        int i;
        int count=0;
        for(i=1;i<=N;i++)
        {
                if (availablecolors[i]!=0)
                {
                        count++;
                }
        }
        return count;
}
int chkcolors(int availablecolors[],int N,int j)
{
        int i=0;
        for(i=j;i<N;i++)
        {
                if (availablecolors[j]!=0)
                {
                    return  availablecolors[j];
                }
        }
}
int delcolors(int *availablecolors,int assign,int len)
{
        int i=0;
        int j=1;
        int temp[200] ;

        for(i=1;i<=len;i++)
        {
```

```
            if(availablecolors[i] == assign)
                availablecolors[i]=0;
            if(availablecolors[i]>0)
                temp[j++]=availablecolors[i];
        }
        for(i=j;i<=len;i++)
            temp[i]=0;
        for(i=1;i<=len;i++)
            availablecolors[i]=temp[i];
        return 0;
}
int numof2(int assign[],int N)
{
        int i=0;
        int j=0;
        int k=0;
        int temp;
        int temp2[200];
        int count=0;
        qsort(assign,N,sizeof(int),int_comp);
        temp   =0 ;
        for(i=1;i<N;i++)
        {
            if(temp != assign[i])
            {
                count++;
                temp = assign[i];
            }
        }
        return count;
}
```

```
int int_comp(const void *n,const void *m)
{
        return((*(int *)n)-(*(int *)m));
}
/* Calculate minimum color */
int coling()
{
        int assign[201];
        int availablecolors[201][201];
        int crcount=0;
        int maxcount=0;
        int c=0;
        int i=0;
        int j=0;
        int k=0;
        int n=0;
        int m=0;

        for(n=1;n<=N;n++)
        {
                assign[n]=0;
                for(m=1;m<=N;m++)
                        availablecolors[n][m]=m;
        }
        for(i=1;i<=N;i++)
        {
                if(empty(availablecolors[i],N)==0)
                {
                        printf("in sufficient no of colors");
                        return 0;
                }
                j=1;
                assign[i]=availablecolors[i][j];
                maxcount=0;
```

```
            for(j=1;j<=numof(availablecolors[i],N);j++)
            {
                c = availablecolors[i][j];
                crcount=0;
                for(k =(i+1);k<=N;k++)
                {
                    if(newedge[i][k]&&(availablecolors[i][j]!=c))
                        crcount++;
                }
                if(crcount>maxcount)
                {
                    maxcount=crcount;
                    assign[i]=c;
                }
            }
            for(j=1;j<=N;j++)
            {
                if((newedge[i][j]==1))
                    delcolors(availablecolors[j],assign[i],N);
            }
        }
}
```

AUTHOR BIOGRAPHY

Author	Rungaree Santawakup
Date of Birth	November 6, 1977
Place of Birth	Samutprakarn, Thailand
Address	123/2 M.3, Taiban, Muang, Samutprakarn 10280 Thailand Tel.0-1822-5052
Bachelor Degree	B.Sc.(Computer Science)
Institution	Faculty of Science, Burapha University, Thailand
Year of Graduation	1998