PROBABILISTIC GRAPH COLOURING ALGORITHMS USING ZYKOV TREES

by

PIKIE M. LEPOLESA
B.Sc., National University of Lesotho, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIRMENTS FOR THE DEGREE OF MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]
Dr. John Ellis

[redacted]
Dr. Frank Ruskey

[redacted]
Dr. Paul Smith

[redacted]
Dr. Donald Miller

University of Victoria

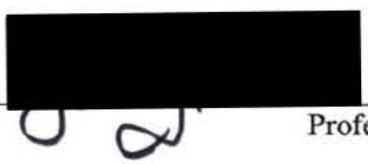
August 1986

Supervisor: Professor John A. Ellis

ABSTRACT

Let G=(V,E) be a simple, undirected graph. A colouring of G is a mapping c : $V \rightarrow \{1,2,...,k\}$. c is a *proper* colouring if $c(u) \neq c(v)$ for all $\{u,v\}$ in E. The chromatic number of G, denoted $\chi(G)$, is the smallest positive integer k for which a proper k-colouring exists. The graph colouring decision problem is to determine whether for a graph G and a positive integer k, there exists a proper colouring of G using at most k colours. A *Las Vegas* algorithm, i.e. a probabilistic algorithm that never lies, for the graph colouring decision problem that runs in polynomial time on almost all instances of the problem is given. Experimental results obtained with this algorithm suggest that it works well for sparse graphs. An approximation algorithm is suggested for graph colouring optimisation problem.

Examiners:



Professor John A. Ellis

[redacted]

Professsor Frank Ruskey

[redacted]

Professor Paul Smith

[redacted]

Professor Donald Miller

Table Of Contents

Abstract ii

Table Of Contents iv

List Of Figures vi

Acknowledgement viii

1. Introduction 1

2. Background 5

2.1 Probabilistic Algorithms 5

2.2 Examples of Probabilistic Algorithms 7

2.2.1 Primality Testing 7

2.2.2 Matrix Multiplication 8

2.2.3 Hamilton Circuit 9

2.3 Probabilistic Turing Machine and their Complexity classes 12

3. Zykov trees and colouring algorithms 17

3.1 Algorithms based on Zykov trees 22

Algorithm 1 22

Algorithm 2 24

Algorithm 3 27

4. Survey of probabilistic colouring algorithms 29

4.1 No-choice Algorithm 32

4.2 Brélaz's Algorithm 35

4.3 Local Improvement Algorithm 37

4.3.1 Extension of Local Improvement 40

5. A Las Vegas colouring algorithm 42

5.1 Preliminaries 42

5.2 Algorithm Description 43

5.3 Experimental Results 49

5.3.1 Random k-colourable graphs 50

5.3.2 Random NOT k-colourable graphs 54

5.3.3 Sparse k-colourable graphs 57

5.3.4 Exploring the leaves of the unpruned Zykov tree 61

5.4 Theoretical Results 67

5.4.1 k-colourable graphs 67

5.4.2 Not k-colourable graphs 72

6. Summary 73

Bibliography 75

List Of Figures

Figure 2.1 10

Figure 3.1 19

Figure 3.2 21

Figure 3.3 23

Figure 3.4 25

Figure 3.5 27

Figure 4.1 33

Figure 4.2 34

Figure 4.3 36

Figure 4.4 39

Figure 5.1 44

Figure 5.2 46

Figure 5.3 47

Figure 5.4 47

Figure 5.5 48

Figure 5.6 48

Figure 5.7 51

Figure 5.8 52

Figure 5.9 53

Figure 5.10 55

Figure 5.11 56

Figure 5.12 59

Figure 5.13 60

Figure 5.14 62

Figure 5.15 63

Figure 5.16 64

Figure 5.17 65

Figure 5.18 69

Acknowledgements

I wish to thank Professor John A. Ellis for suggesting the topics of this thesis to me and for his valuable guidance while the research for this thesis was in progress until its completion. I also thank *World University Service of Canada* for their continuous financial support whilst the research was in progress.

CHAPTER 1

INTRODUCTION

Let G=(V,E) be a simple, undirected graph. A colouring of G is a mapping c : $V \rightarrow \{1,2,...,k\}$. c is a *proper* colouring if $c(u) \neq c(v)$ for all $\{u,v\}$ in E. The chromatic number of G, denoted $\chi(G)$, is the smallest positive integer k for which a proper k-colouring exists.

Many practical problems can be modelled in terms of colouring a graph. An example is the storage of chemicals. Certain pairs of chemicals are kept in separate compartments of a warehouse since they may interact. Suppose we wish to use few compartments. We can construct a graph G with a vertex for each chemical and join a pair of vertices when the corresponding chemicals interact. A proper colouring of G using k colours then corresponds to a storage scheme using k compartments.

Another example is examination scheduling [50]. Here vertices correspond to exams and vertices are adjacent if some student takes both exams. We wish to schedule the exams in as few periods as possible. The list of other practical problems include timetable scheduling, stewardess scheduling by airlines, conference scheduling, allocation of radio frequencies and scheduling of traffic facilities. References for these applications can be found in [38].

The graph colouring problem can be expressed as **an optimisation problem** when we wish to determine the minimum k such that G is k-colourable; i.e. determine $\chi(G)$, or as **a decision problem** when we wish to determine whether for a graph G and a positive integer k, there exists a proper colouring of G using at most k colours, i.e. we ask whether $\chi(G) \leq k$.

A decision problem is in the class **P** if there exists a *deterministic* algorithm that runs in polynomial time on all instances of the problem and is in **NP** if there exists a *nondeterministic* algorithm that runs in polynomial time on all instances of the problem. A decision problem Π is said to be **NP-complete** if it is in **NP** and every problem in **NP** is polynomially reducible to Π. For detailed definitions see [15].

Karp [22] showed that the graph colouring decision problem is **NP-complete**. Stockmeyer [40] strengthened this by showing that it remains **NP-complete** even for k as small as 3 for planar graphs. The optimisation problem is **NP-hard**, i.e. not known to be in **NP** but proved to be *at least as hard* as the **NP-complete** problems [15]. It is widely conjectured that no deterministic polynomial time algorithm exists for any **NP-complete** problem.

All known deterministic colouring algorithms, have execution times proportional to a super polynomial function in the number of nodes on most instances of both the optimisation and decision problems [4-6, 8, 37]. Super polynomial algorithms are

often impractical for even moderately sized graphs and, consequently, considerate effort has been devoted towards developing polynomial algorithms that give an approximation to the optimisation problem [28, 46, 50]. These are called *approximation* algorithms. There is, however, a limit to the accuracy of such algorithms. Garey and Johnson [14] have shown that unless **P=NP** then for every polynomial time approximation algorithm there exists graphs G for which the colouring obtained by the algorithm will be at least twice $\chi(G)$. In other words, if A(G) is the number of colours which a polynomial time approximation algorithm A uses when applied to G, then there are graphs for which $\frac{A(G)}{\chi(G)} \geq 2$, unless **P=NP**.

A *probabilistic* approach has been used to determine the exact solution of decision problems faster than existing deterministic algorithms. Probabilistic algorithms make random choices during execution. They may produce a correct answer, a wrong answer or no answer at all. We will provide a more formal description later.

In Chapter 2 we review examples of probabilistic algorithms for various decision problems and review a formal model, the Probabilistic Turing machine. Relations between the complexity classes induced by this model and the common classes **P** and **NP** are discussed.

Chapter 3 reviews the concept of the Zykov tree. Formal definitions are given here together with some algorithms that use this concept. In Chapter 4, we survey existing *probabilistic algorithms* for colouring and their time complexity. A new *Las*

Vegas, algorithm that is probabilistic but never lies, is given in Chapter 5. The algorithm uses the Zykov tree. It will be shown that the algorithm runs in polynomial time *almost always* . Experimental results indicate that the algorithm works well on sparse graphs, for small k. Experimental results obtained and theoretical analysis for this algorithm are also given in this chapter. Finally, conclusions and open problems are given in Chapter 6.

CHAPTER 2

BACKGROUND

2.1 Probabilistic Algorithms

Let π be a set of all instances of a decision problem. For example the Hamiltonian circuit problem is to determine whether a graph G has a circuit that visits each and every vertex of G exactly once. The graph colouring problem is to determine for a graph G = (V,E) and an integer k whether the vertices of G can be coloured using k or less colours such that no two adjacent vertices have the same colour.

For each instance $x \in \pi$ we define a function *size(x)*, written $|x|$. For example in most problems involving graphs, the number of vertices serves as an appropriate measure of size. $|n|$, where n is an integer, is defined as the number of digits in the binary representation of n. Thus $|n| = \lceil \log_2 n \rceil$.

A probabilistic algorithm for π is one in which the algorithm makes one or more random decisions as to which next step to take. With the exceptions of these random choices, the algorithm proceeds deterministically. For some instances, the probabilistic algorithm may sometimes fail to produce a correct solution or it may terminate without any answer.

All probabilistic algorithms used to be called *Monte Carlo* algorithms. Lately however, the name *Monte Carlo* has been reserved for those probabilistic algorithms that run in polynomial time on all instance of the problem and have some small error probability. Other probabilistic algorithms solve all instances of the problem with zero error probability. Thus they never lie. However they may fail to produce an answer within an allocated time. These are called *Las Vegas* algorithms.

A subset of *Monte Carlo* algorithms are those which solve every instance of π in polynomial time together with the property that the probability of error can be made less than any $\varepsilon > 0$. Probabilistic algorithms of this nature are called *Rabin-type* algorithms.

The next section gives various probabilistic algorithms for decision problems. In 2.2.1 we give an example of a Rabin-type algorithm, the primality testing algorithm. Another example of a Rabin-type algorithm is given in 2.2.2. An example of a Monte Carlo type algorithm, for finding a Hamiltonian circuit is given in 2.2.3. Other Monte Carlo algorithms are discussed in [20, 21]. A Las Vegas algorithm for the graph isomorphism problem due to Babai can be found in [2].

2.2 Examples of Probabilistic Algorithms

2.2.1 Primality Testing Algorithm

The primality testing problem is to determine whether a given positive integer n is prime or composite. The algorithm to be described below is due to M.O. Rabin [35, 36] and uses the following two theorems :

Theorem 2.1. *Let n be an integer and* $1<i<n$. *If either of conditions (i) or (ii) holds the integer n is composite.*

(i) $i^{n-1} \neq 1 \bmod n$.

(ii) There exists an integer k such that n-1 is divisible by 2^k *giving an integer m such that* $1<\gcd(i^m-1, n)<n$.

Theorem 2.2. *If n is composite at least half of the integers i in the range* $1<i<n$ *satisfy (i) and (ii). If n is prime, neither is satisfied for any i.*

The integers that satisfy either condition for some n are called *witnesses* to the compositeness of n.

The algorithm randomly chooses an integer i, $1 \leq i \leq n-1$ and tests whether i is a witness to the compositeness of n. If so, it declares n as composite with certainty, otherwise it labels n as prime. If n is composite, the probability that the algorithm labeled it prime is equal to the fraction of integers between 1 and n-1 that were not witnesses, which is less than $\frac{1}{2}$, therefore the probability that the prime conclusion is

in error is less than $\frac{1}{2}$.

The test of conditions (i) and (ii) can be done in time $O(\log^3 n)$. By repeating the above algorithm k times with k independent random choices of i and declaring n as prime only if none is a witness, the probability of labeling a composite number as prime becomes less than $\frac{1}{2^k}$. Thus the probability of error can be made $<\varepsilon$ for any $\varepsilon > 0$ in polynomial time.

2.2.2 Matrix Multiplication

Here we consider the problem of deciding for three nxn matrices A,B,C whether A.B = C. Of course there exists a deterministic algorithm that has complexity $O(n^3)$. However we describe a probabilistic algorithm, due to R. Freivalds [13], that has time complexity $O(n^2)$ and proceeds as follows:

- Generate a random 1xn vector x with entries from {-1,1}
- Use the vector to test whether (A.(Bx))=C.x.
- If the test fails then A.B $\neq$ C with certainty. On the other hand conclude A.B = C. It can be shown that the probability that the equality conclusion is wrong is less than $\frac{1}{2}$.

Repeat the above test with k random 1xn vectors. If the matrices pass all k tests, then assume A.B=C. The probability of error in this case is $\leq \frac{1}{2^k}$. As with the

primality algorithm, the probability of error can be made $<\varepsilon$ for any $\varepsilon>0$.

2.2.3 Hamiltonian Circuit algorithm

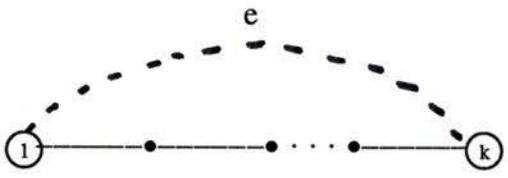
The Hamiltonian circuit problem is to decide if a given graph G=(V,E) has a cycle passing through each vertex of G exactly once. The problem is known to be **NP-complete** [14]. The algorithm described below has a polynomial time complexity and is due to Posa [23, 33]. It finds a Hamiltonian circuit almost always if one exists. We say a property holds *almost always* for graphs with n vertices if the probability that the property holds tends to 1 as n tends to infinity.

The algorithm randomly selects a vertex, say vertex 1, and proceeds to construct longer and longer paths starting at vertex 1. Let P be a simple path from vertex 1 to vertex k. k is called a *free end point* of P. Let e = {k,*l*} be an edge incident to k. Then any of three operations may be possible, each yielding either a Hamiltonian circuit or a new path.

(a) *closure* -- If l=1 and P contains all vertices, then P $\cup$ {e} is a Hamiltonian circuit;

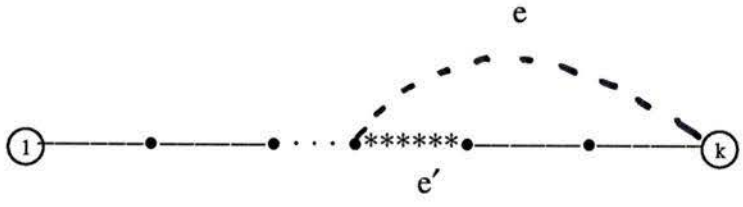
(b) *extension* -- If l does not occur in P, then e may be adjoined to P to form a longer path, with l as its end point;

(c) *rotation* -- If $l \neq 1$ and l occurs in P, then there is a unique edge e′ in P such that P $\cup$ {e} -{e′} is a path.



(a) closure

(b) extension



(c) rotation

Figure 2.1: Illustration of above operations.

Given a path of length i the algorithm repeatedly performs the rotation operation to obtain new paths of length i, continuing until extension to a path of length i+1 or (when i=n) closure of a Hamiltonian circuit becomes possible. The algorithm is constrained so that it will not form two paths of length i with the same free end

point. This constraint ensures that the algorithm will run in polynomial time, but enhances the likelihood that the algorithm will stall because permissible rotations become exhausted before extension or closure can occur. Nevertheless, Posa shows that the algorithm almost always finds a Hamiltonian circuit, if one exists.

Motivated by the success of the primality testing algorithm and other probabilistic algorithms [1,35,39], many theoreticians [16,25,26,47,51] set out to find the power of the these algorithms with respect to **NP-complete** decision problems.

Since the primality test problem is in **NP** $\cup$ **co-NP** but not known to be in **P** [31,34], it looks as though probabilistic algorithms might be more powerful than deterministic algorithms. Interesting questions arise such as; "Is there any polynomial time probabilistic algorithm with small error probability for some **NP-hard** problems?", "Is there any Rabin-type algorithm for some **NP-complete** problem?". Gill [16], using the probabilistic Turing machine model, shows probabilistic complexity classes and their relation to **P** and **NP**. He showed that every problem in **NP** can be solved by a polynomial time probabilistic algorithm with error probability less than or equal to $\frac{1}{2}$. However, it is not known whether such probabilistic algorithms can be improved to have error probabilities bounded above by a constant $\varepsilon < \frac{1}{2}$.

The following section, we review the work done by J.T. Gill [16], where a formal model for probabilistic algorithms, the Probabilistic Turing Machine, (PTM) is

given. A formal definition of PTM is given in Definition 2.1. Definition 2.2 gives the computational complexity classes induced by PTMs. Relations between these classes and the classes **P** and **NP** are given as Theorem 2.3.

2.3 Probabilistic Turing Machine and their Complexity classes

Definition 2.1. A *probabilistic Turing machine* (PTM) is a Turing machine with distinguished states called coin-tossing states. For each coin-tossing state, the finite control unit specifies two possible next states. The computation of a probabilistic Turing machine is deterministic except that in the coin-tossing states, the machine tosses an unbiased coin to decide between the two possible next states.

Definition 2.2. A probabilistic Turing machine is *polynomially bounded* if there exists a polynomial p(n) such that every possible computation of M on input of size n halts in at most p(n) steps.

Decision problems can be expressed as language recognition problems [15]. The correspondence is brought about by encoding schemes used in specifying instances of the problem. Let Π be a decision problem and e an encoding scheme then Π consists of a set D_Π of instances and a subset $Y_\Pi \subseteq D_\Pi$ of *yes-instances*. Each instance of Π can be encoded (described) by appropriate strings over a fixed alphabet $\sum$. The set of all string over the alphabet $\sum$ denoted by $\sum^*$, are partitioned into three classes of string: those that are not encodings of instances of Π, those that

encode instances of Π for which the answer is *'no'*, and those that encode instances of Π for which the answer is *'yes'*. This latter class is the language we associate with Π and *e*, setting

$$L = \left\{ x \in \Sigma^* : \begin{array}{c} \Sigma \textit{ is the alpabet used by } e, \text{ and } \textit{the} \\ \textit{encoding under } e \textit{ of an instance } I \in Y_\Pi \end{array} \right\}$$

For the remainder of this chapter, we shall use the terms "recognising languages" and "solving decision problems" interchangeably. We will also consider polynomially bounded probabilistic Turing machines unless otherwise stated. A computation path *rejects* if the path leads to a *'no'* answer and *accepts* if the path leads to a *'yes'* answer.

A probabilistic Turing machine M recognises a language L iff for all $x \in L$ the number of **yes** leaves, in the computation tree of x, is more than $\frac{1}{2}$ and the number of **no** leaves is more than $\frac{1}{2}$ for all $x \notin L$.

Definition 2.3. The complexity classes induced by probabilistic Turing machine are defined as follows:

PP is the class of languages recognised by polynomial time bounded PTMs.

BPP is the class of languages recognised by polynomial time bounded PTMs with bounded error probability $\varepsilon < \frac{1}{2}$.

R is the class of languages recognised by polynomial bounded PTMs which have zero error probability for inputs not in the language.

ZPP is the class of languages recognised by PTMs with polynomial bounded average time and zero error probability. In this class, the probabilistic Turing machine is not necessarily polynomially bounded.

Gill [16] discovered the following relations among the above mentioned classes and the classes **P, NP** and **PSPACE.**

Theorem 2.3.

$$\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{R} \subseteq \left\{\begin{matrix}\mathbf{BPP} \\ \mathbf{NP}\end{matrix}\right\} \subseteq \mathbf{PP} \subseteq \mathbf{PSPACE}$$

None of the above inclusions are known to be proper. Also it does not appear that either **NP** $\subseteq$ **BPP** or **BPP** $\subseteq$ **NP**.

For a problem in **R** or **BPP** we can achieve any small error probability by iterating the given machine a polynomial number of times. The primality testing algorithm can be adapted to show that the problem of deciding whether a number is composite is in **R**. However testing for primes has not yet shown to be in **R** since the primality test algorithm may mistakenly declare a composite number prime.

Monte Carlo algorithm with bounded error probability corresponds to **BPP**. A *Las Vegas* algorithm with average polynomial time corresponds to **ZPP**.

It seems that **BPP** and **NP** are incomparable, but we can ask what consequences arise if they are comparable.

Let A be a set of strings. An *oracle for A* is a mechanism that can answer the membership question for A in **one step**. Let P^A (NP^A) be the class of languages recognised by deterministic (nondeterministic) polynomial time oracle Turing machines aided by an oracle for A. If Ψ is a collection of sets, then $P^{\Psi} = \cup_{A\in\Psi} P^A$ and $NP^{\Psi} = \cup_{A\in\Psi} NP^A$.

The polynomial hierarchy [41] is defined recursively as:

$$\sum_0^p = \Pi_0^p = \Delta_0^p = P,$$

and for $k \geq 0$,

$$\sum_{k+1}^p = NP^{\sum_k^p}$$

$$\Pi_{k+1}^p = \left\{A : \overline{A} \in \sum_{k+1}^p\right\}$$

$$\Delta_{k+1}^p = P^{\sum_k^p}$$

Let $\mathbf{PH} = \cup_{k=0}^{\infty} \sum_k^p$ denote the polynomial hierarchy. It is known that **PH** $\subseteq$ **PSPACE** [41]. Karp and Lipton [24] showed that if **NP** $\subseteq$ **BPP** then **PH** $\subseteq \sum_2^p$. Two years later Ko [25] and Zachos [51] independently strengthened the above result and showed that if **NP** $\subseteq$ **BPP** then **PH** $\subseteq$ **BPP**.

All optimisation problems whose corresponding decision problems are **NP-complete** are known to be in Δ_2^p but not known to be in **NP**. The above result

claims that they are in **NP**, by assumption that **BPP** $\subseteq$ **NP**. Therefore it is very unlikely that **NP** $\subseteq$ **BPP**.

CHAPTER 3

ZYKOV TREES AND COLOURING ALGORITHMS

Let G=(V,E) be a simple graph, i.e no multiple edges or self loops. If G is complete then $\chi(G) = n$ where n is the number of nodes in G. Assume that G is not complete, then there exist at least two non-adjacent vertices u and v in G. Let G_1 denote the graph obtained from G by adding an edge between u and v, and let G_2 denote the graph obtained by identifying u and v, i.e. replacing u and v by a single vertex adjacent to each vertex which was adjacent to u or v.

In any proper colouring of G either u and v have the same colour or they have different colours. Thus we have

$$\chi(G) = \min\left\{ \chi(G_1), \chi(G_2) \right\} \tag{3.1}$$

Any graph G which is not complete can be reduced to both G_1 and G_2. If either G_1 or G_2 is reducible, the process can be continued until all graphs obtained are irreducible (complete).

Let $H_1, H_2, \ldots, H_r$ be the irreducible graphs obtained from G, then

$$\begin{aligned}\chi(G) &= \min\left\{\chi(H_1), \chi(H_2),..., \chi(H_r)\right\} \\ &= \min\left\{|H_1|, |H_2|,..., |H_r|\right\} ,\end{aligned} \tag{3.2}$$

since $\forall$ i $\chi(H_i) = |H_i|$.

Suppose we place a graph G at the root of a tree and branch repeatedly, using edge addition and vertex identification. We will then obtain a binary tree for G, called the Zykov tree [8, 18, 52]. Each node in the tree is a graph obtained from its parent node by either of the two operations. The leaf nodes are complete graphs with size ranging from n to $\chi(G)$.

By 3.2, $\chi(G)$ is the minimum value of $\chi(L)=|L|$ over all leaves L of any Zykov tree for G. Denote by Z(G) a Zykov tree derived from G.

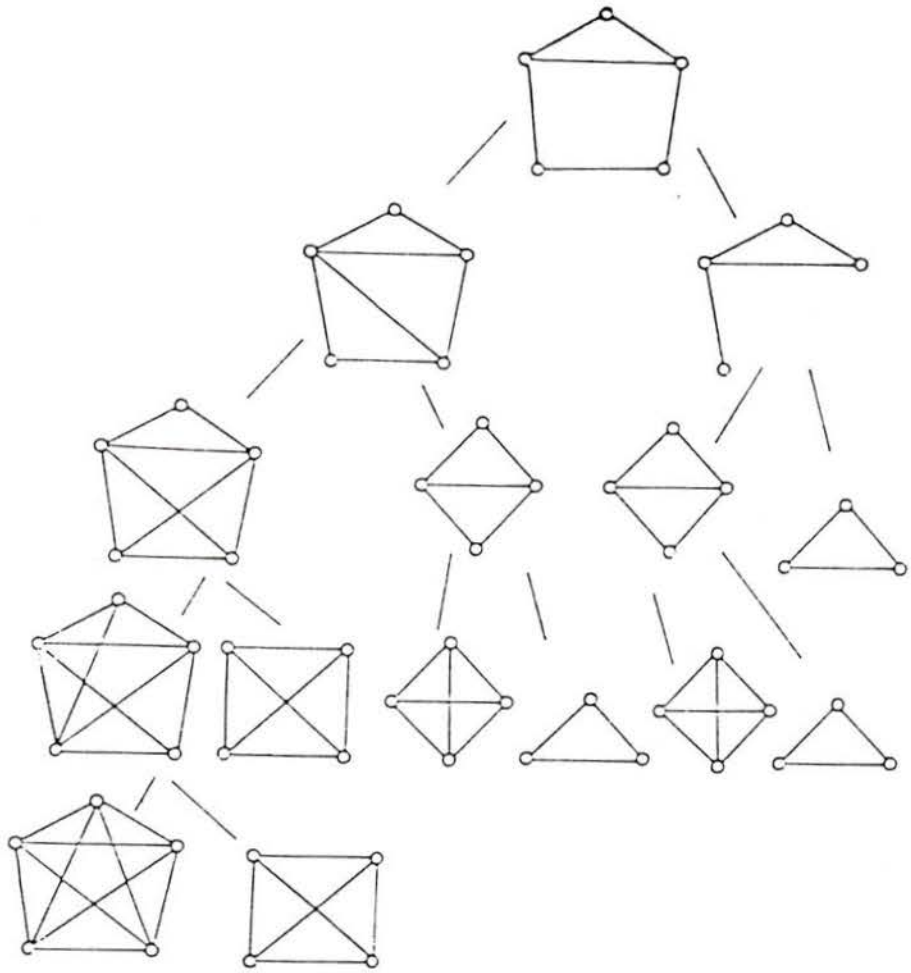


Figure 3.1: Example of an unpruned Zykov Tree .

The size of a Zykov tree is significant since any algorithm that uses these trees must explore at least some part of it. McDiarmid [29] showed that every Zykov tree for a given graph has the same size. Thus it does not matter which sequences of vertex pairs u and v is chosen.

Of course it may not be necessary to search the whole tree. Whenever we are exploring the tree and find a graph, say H, containing a clique of size k or more, it is pointless continuing with this branch of the tree since no graph appearing on the sub-tree rooted at the node that represents H will contain a clique of whose size is smaller than k. We can then terminate this branch. This is called pruning and is described in detail in the next subsection. Let $Z^*(G)$ denote the pruned Zykov tree for graph G, i.e. a tree none of whose nodes are graphs that have cliques of size k+1 or more. Figure 3.1 and 3.2 give examples of pruned and unpruned Zykov trees.

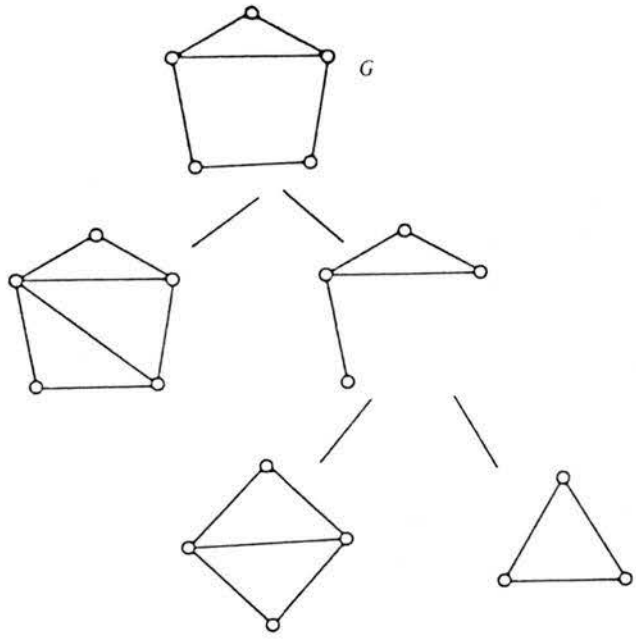


Figure 3.2: Example of a pruned Zykov Tree.

McDiarmid [29] gave the following results on the size of pruned and unpruned Zykov trees for graphs of size n.

For a graph G, let C(G) denote the number of proper partitions of G, i.e. the number of colourings with colour indifference.

Lemma 3.1.: *Every Zykov tree Z(G) for a given graph G has 2C(G)-1 nodes and the expected size of the unpruned tree is given by:*

$$|Z(G)| = O(e^{n\log n}). \tag{3.3}$$

Lemma 3.2.: *For almost all graphs G on n vertices the expected size of the pruned Zykov tree is given by:*

$$|Z^*(G)| \geq e^{.157n\sqrt{\log n}} \tag{3.4}$$

3.1 Algorithms based on Zykov trees

In this section we give the only three published algorithms that explicitly use the Zykov tree. The first two exactly determine the chromatic number of a graph. The last one is an approximation algorithm.

Algorithm 1

The algorithm given below constructs and searches the complete Zykov tree for a graph G, and is a direct consequence of Zykov's theorem [52]. By definition of Zykov tree, if follows that to determine the chromatic number, $\chi(G)$, we can construct the Zykov tree for G and record the size of the minimum complete graph. The algorithm terminates when all branches have been explored. The chromatic number is the size of the minimum complete graph seen.

```
function chromatic (G:graph):integer;
var approx_chromatic:integer;
procedure colour(G:graph);
var G1, G2 : graph;
begin
  if complete(G) then
    if |G|< approx_chromatic then number ← |G|;
  else
  begin
    find(u,v);
    G2 ← identify(u,v);
    G1 ← addedge(u,v);
    colour(G2 );
    colour(G1)
  end
end;
begin
  approx_chromatic ← |G|;
  colour(G);
  chromatic ← approx_chromatic
end;
```

Figure 3.3: Algorithm 1.

Functions *complete, addedge* and *identify* check if the graph in question is complete (irreducible), add an edge between vertices u and v and identify vertices u and v, respectively. Procedure *find* returns any two non-adjacent vertices. All these are straightforward and are not given here.

The performance of this algorithm depends on the size of the Zykov tree. Thus from equations (3.3) and (3.4), the execution time of this algorithm grows exponen-

tially with n and hence it is unlikely to be useful in practice.

Algorithm 2

Corneil and Graham [8] presented the following algorithm which improves on Algorithm 1 by bounding the search. Suppose we know that α is an upper bound of $\chi(G)$, then when a graph is encountered in the reduction of G which is known to contain an α–clique we need not reduce further. The pruning process searches for an α–clique in each graph H encountered in the Zykov tree for G.

Since the clique finding problem is NP-complete [22], the algorithm looks for an α–cluster in H. An α–cluster is a set of α vertices which has a high density of edges. The algorithm checks to see if these α vertices form an α-clique. H is reduced to H_1 by adding an edge to the any two non-adjacent vertices in the α–cluster. The process is repeated until the α–cluster becomes an α–clique, when this branch is terminated. Graphs formed by node identification are treated similarly.

If $\chi(G) = \alpha$, the algorithm will terminate by building α–cliques. On the other hand, if $\alpha > \chi(G)$, at some stage the graph introduced by identification will have $\alpha-1$ vertices implying that $\chi(G) \leq \alpha-1$. Whenever this occurs, the upper bound of $\chi(G)$ is reduced from α to $\alpha-1$ and the process continues. The algorithm terminates when every branch of the Zykov tree introduced has been forced to contain an α–clique, where $\alpha = \chi(G)$.

```
Program Main;
  Procedure Reduce(H:graph,N:integer);
  var
    β:integer;
  begin
    if α>N then α←N;
    β<–alpha;
    if H is not complete then
    Find Hα, a cluster of α–vertices;
    while ( not finish) do
    begin
      if Hα is an α–clique then finish ←true
      else
      begin
        choose non-adjacent vertices u and v in Hα;
        Form H1 and H2 by reduction of H;
        H←H2;
        Reduce(H2, N-1);
        if β≠α then
        begin
          Reduce (H,N);
          finish ← true
        end
      end
    end;
    begin
      Find Initial α, upper bound for χ(G). n=|G| will suffice;
      finish ←false;
      Reduce(G,n)
    end.
```

Figure 3.4: Overview of Algorithm 2.

The algorithm terminates when all of the pruned tree has been explored and hence its running time is bounded below by the size of $Z^*(G)$. By equa-

tion (3.4), the expected size of the pruned tree grows exponentially with the number of nodes. Therefore the running time of this algorithm grows exponentially with the number of vertices.

Corneil and Graham, using graphs with numbers of vertices between 10 and 60, compared this algorithm with that of Christofides [6] which depends on the concept of *Maximal Internally Stable Set* (or MISS), where a MISS of a graph is a set of vertices, no two of which are adjacent and which is maximal with respect to this property. A MISS in G is a clique in $\overline{G}$, the complement of G.

On all the graphs used, they found that the execution time of their algorithm was either significantly less than that of Christofides or at worst approximately equal to it [8]. Corneil and Graham concluded that the asymptotic behaviour of their algorithm is superior to that of Christofides.

McDiarmid [29] gives a result that contradicts the conclusion made by Corneil and Graham. He showed that an algorithms that uses *Maximal stable sets* is asymptotically faster than any algorithm that uses a Zykov tree. However, the empirical results obtained [8] show their algorithm to be one of the best practical algorithms for determining the chromatic number of a graph.

Algorithm 3

Because of the exponential size of both the pruned and unpruned Zykov trees for any particular graph, R.D. Dutton and R.C Brigham [9] suggested using the Zykov tree as a basis for an approximation algorithm.

Let Z(G) be the Zykov tree for a particular graph G. Then there exists a path from the root to a smallest leaf. The size of this leaf is $\chi(G)$. Dutton and Brigham [9] suggested a technique that is likely to find a path that leads to a smallest leaf in the tree.

A good algorithm must select pairs of nodes for identification so as to forestall the formation of a complete graphs as long as possible. Identifying the pair with the maximum number of common neighbours is probably a good choice. The algorithm proceeds as follows:

for all non-adjacent vertices v_i and v_j ***do*** compute c_{ij}, the number of common neighbours;
while there exists non-adjacent pairs ***do***
begin
  Select the non-adjacent pair v_i, v_j for which c_{ij} is maximum;
  Identify v_i and v_j;
  Adjust c_{rs} for all affected non-adjacent pairs;
  $n \leftarrow n - 1$
end

Figure 3.5: Overview of Algorithm 3.

At termination of the **while** loop, the current value of n is an estimate of the chromatic number of the graph. The time complexity is O (n^3). Dutton and Brigham tested several graphs found in [19] and the colourings produced were optimal. They also found graphs for which the estimated chromatic number was much larger than $\chi(G)$.

CHAPTER 4

A SURVEY OF PROBABILISTIC COLOURING ALGORITHMS

Colouring the vertices of a graph is often used to solve many practical problems, as described in the introduction. Thus the solution to the graph colouring problem has both the practical and theoretical importance. The *NP-completeness* of the problem does not imply that there are no polynomial time algorithms for some instances of the problem. In fact claims have been made [14, 27, 42] that most instance of some **NP-complete** problems can be solved in polynomial time and only a few instances require super polynomial time.

For the graph colouring problem, all known exact algorithms are super polynomial on most instances. Because of the *NP-completeness* of the problem, many researchers set out to find approximation algorithms [9, 19, 28, 32, 48] that do not use many extra colours. However, for all of these algorithms, the existence of a class of graphs for which they use more than twice the minimum number of colours has been shown [14].

Many authors [3, 7, 10, 11, 17, 23, 29, 30, 49] used a probabilistic analysis of

these approximation algorithms.

A simple *random graph* model was used in all the analyses. We say a graph G_n on n vertices is random if it is generated by the following process:
For each pair of vertices v_i and v_j create an edge with probability p.

The results obtained were very encouraging because they enable us to construct colouring algorithms that take advantage of the structure of random graphs. For example in [11] the probability of the existence of a clique of size r, $r>0$, is given and we can use this fact to decide whether a random graph is k-colourable.

Let P be a predicate defined on the set **R** of graphs on n nodes and let G_n be selected at random from **R**. We say that P holds for *almost all* graphs if

$$\lim_{n\to\infty} \text{Prob}(P(G_n)) = 1.$$

Grimmet and McDiarmid [17] used this approach to show that for almost all graphs on n vertices

$$\chi(G) \geq \frac{(1-\varepsilon)n}{(2\log_{1/(1-p)}n)}$$

where ε is any positive constant.

Even though this model gave promising results, Franco and Turner [12, 42] point out that some of these results may be very misleading. They suggest that the model is too simple because it can make even the most simple minded algorithm look good. This point is strengthened by Wilf [49] who gives an analysis of a back-

track search algorithm that shows that the expected size of the backtrack search tree explored by their algorithm is O(1), when graphs are selected using the simple random graph model.

Consider the following algorithm for 3-colourability and suppose we want to analyse its performance using the random graph model.

```
function naive(G:graph): boolean;
begin
   naive ← G is bipartite;
end;
```

Since most random graphs are not bipartite and also not 3-colourable, we conclude that for almost all graphs the above algorithm determines correctly whether G is 3-colourable and hence we have a very good algorithm. Obviously such a conclusion is misleading, because in the set of all 3-colourable graphs, most are not bipartite and yet the algorithm classifies them as not 3-colourable. It is thus preferable to use a more realistic model.

Franco [12] suggests that for the random graph model, we test our algorithms using $k = \frac{n}{\ln(n)}$ because the number of k-colourable graphs is then almost equal to the number of those that are not k-colourable. If the algorithm performs well on the set of k-colourable and on the set of not k-colourable graphs, we can claim that we

have a good algorithm.

Turner [42] suggests that we should try to select a model that gives a challenge to candidate algorithms. In [42] *randomly k-colourable* graphs are defined. We say a graph G=(V,E) is *randomly k-colourable* if G is constructed by the following experiment:

- Let V={1,2,...,n}.
- For each u∈ V let c(u) be a random integer in [1,k].
- For each pair u, v ∈ V such that c(u) ≠ c(v), include edge {u,v} in E with probability p.

The next part of this chapter will discuss existing probabilistic colouring algorithms and their performance.

4.1 No-choice algorithm

This algorithm is due to Turner [42] and heavily depends on the use of a probabilistic algorithm to find a k-clique.

Let G=(V,E) be a simple graph and define a *partial colouring* of G as a mapping $c : V \rightarrow [0,n]$. Given a partial colouring c, we define for each vertex x, a set avail(x) = { i | $1 \leq i \leq n \cap c(y) \neq i$ for any (x,y) ∈ E }. If x is currently uncoloured c(x) = 0, avail(x) is the set of colours that are available for properly colouring x. The clique finding is achieved by the probabilistic function given in Figure 4.2

```
function nochoice(G:graph;k:integer): boolean;
var
  S: Set of Vertices;
begin
  S ← clique(k);
  nochoice ← false;
  if | S | = k then
  begin
    Colour each vertex in S with a distinct colour in [1,k]
    repeat
      Select an uncoloured vertex x for which | avail(x) ∩ [1,k] | = 1 and
      let c(x) = min avail(x)
    until all vertices are coloured or selection failed.
    nochoice ← all vertices coloured;
  end
end
```

Figure 4.1: No-choice Algorithm.

```
function clique(G:graph, r:integer):Set of vertices;
var
  K,S : Set of vertices;
begin
  S ← [1,2...n]
  K ← ∅ ;
  while S ≠ ∅ do
  begin
    Randomly Select x ∈ S
    K ← K ∪ [x];
    S ← S ∩ Neighbours(x);
  end;
  clique ← K ;
end;
```

Figure 4.2: Clique Finding Function.

This algorithm will fail to produce a k-colouring if it is unable to find a k-clique or if at some point $|\ avail(x) \cap [1,k]\ | \neq 1$ for all uncoloured vertices x. Since the clique finding algorithm chooses x at random from S it is possible for it not to find an existing k-clique.

Turner [42-44] performed experiments on randomly k-colourable graphs which showed that the algorithm perfoms well for k small but its performance deteriorates as k gets larger. He also proved that for almost all k-colourable graphs, the No-choice algorithm successfully produces a k-colouring when k is small relative to n. Moreover the time complexity of the algorithm is O(n+mlogk).

4.2 The Brélaz algorithm

This algorithm is due to Brélaz [4] and consists of the repeated application of the following rule:

> *Select an uncoloured vertex x that minimises | avail(x) | and let c(x) = min avail(x). If there are several such vertices available, select one with maximum degree in the uncoloured subgraph.*

```
function brelaz(G:graph,k:integer);
var
   h : heap;
   x,y :integer;
   avail: array[1..n] of Set;
   deg: array[1..n] of integer;
begin
   V ← [1,2...n];
   for x ∈ V do
   begin
      c(x) ← 0;
      avail(x) ← {1,...,n}
      deg(x) ← | Neighbours(x) |;
   end;
   h ← makeheap({1,..., n});
   while h ≠ empty do
   begin
      x ← rootof(h);
      c(x) ← min avail(x);
      for y ∈ neighbours(x) do
      begin
         if c(y) = 0 then
         begin
            avail(y) ← avail(y) - c(x);
            deg(y) ← deg(y) - 1;
            siftup(root);
         end
      end
   end
end;
```

Figure 4.3: Implementation of the Brélaz Algorithm.

The heap contains the uncoloured vertices. For the purpose of heap operations, vertex x is smaller than vertex y if

$$|\,avail(x)\,| < |\,avail(y)\,| \quad \text{or} \quad |\,avail(x)\,| = |\,avail(y)\,| \text{ and } deg(x) > deg(y).$$

The heap operations *makeheap, rootof and siftup* can be found in standard texts.

It is clear that every graph that is found to be k-colourable by the No-choice algorithm will be found to be k-colourable by this algorithm. Let G be the graph found to be k-colourable by the No-choice algorithm. Then G contains a k-clique and it is possible to find vertices x, such that $|\,avail(x) \cap [1,k]\,| = 1$. Now when we run the Brélaz algorithm on G, the first k vertices coloured will form a k-clique. Once the first k vertices have been coloured, the algorithm repeatedly selects a vertex x for which $|\,avail(x)\,|$ is minimum. This vertex will be the same as the one selected by the No-choice algorithm. Thus Brélaz's algorithm mimics the No-choice algorithm. The converse is, however, not true since there is a chance that Brélaz's algorithm may succeed in finding a k-colouring where the No-choice algorithm fails. Experimental results [42] performed on both algorithms confirm this.

Both algorithms fail to produce a k-colouring on most k-colourable graphs on n nodes when $k > \log_2(n)$.

4.3 Local Improvement algorithm

This algorithm is due to M. B. Vitányi [45] and depends on an initial random colouring assigned to the vertices of G.

Let G=(V,E) be a k-colourable graph and let c be a random k-colouring, not necessarily proper, of G. Let *conflict* be a function from $[V \times k] \rightarrow [0,n-1]$, where

conflict(x,i) is the number of neighbours y of x for which c(y)=i.

The algorithm selects a random k-colouring c of G, not necessarily proper, and repeats the following step:

while a change in colour assignments is possible ***and*** c is ***not*** a proper colouring of G *do*

for each vertex x ***do*** c(x) ← i where i is chosen to minimise conflict(x,i)

The algorithm terminates when it finds a proper colouring of G or when the loop was executed and no change was made to the colouring.

```
function localimp(G:graph,k:integer):boolean;
var
   proper,nochange : boolean;
   x,y,cnt :integer;
   conflict: array[1..n] of integer;
begin
   cnt ← 0;
   V ← [1,2...n]
   for x ∈ V do
      c(x) ← Random number in [1,k];
   repeat
      proper ← true;
      nochange ← true;
      for x ∈ V do
      begin
         conflict(x) ← 0; {initialise for each node }
         for y ∈ neighbours(x) do conflict(c(y)) ←
         conflict(c(y))+1;
         i ← colour with minimum conflict;
         if i ≠ 0 then proper ← false;
         if i ≠ c(x) then nochange ← false;
         c(x) ← i;
      end;
   until proper or nochange;
  localimp ← proper;
end;
```

Figure 4.4: Implementation of the Local improvement Algorithm.

This algorithm was found [42] to produce k-colourings for k-colourable graphs where the previous two algorithms No-choice, Brélaz failed. The same randomly k-colourable graphs were used for experiments of this algorithm. It gave k-colourings on graphs with the number of vertices ranging from 100 to 1000 even when k was as

large as 18 before the performance deteriorated. The performsnce of previous two algorithms began to deteriorate for $k > 10$, on the same sized graphs.

The initial random colouring seems to produce several sets containing concentrations of vertices that may be given the same colour. The improvement step eventually separates the conflicting vertices into different colour sets. It seems hard to give a probabilistic analysis for this algorithm.

4.3.1 Extension of Local Improvement algorithm

It is clear that the Local Improvement algorithm runs in polynomial time and also that it heavily depends on the initial k-colouring randomly assigned. In rare cases, the initial colouring turns out to be a proper colouring.

Let Π be a collection of partitions of n vertices into k different classes, say $\pi_i = \{V_{i1}, V_{i2}, \ldots, V_{ik}\}$ where each V_{ij} is a set of vertices assigned the same colour. If G is k-colourable, there exist one or more partitions that define a proper k-colouring of G. The algorithm can be executed with the same input a polynomial number of times, say n^r, $r>0$, until a proper colouring is found or n^r executions have been made.

Experiments performed by the author using this extended algorithm, do indicate a significant increase to the percentage of random k-colourable graphs identified by the algorithm without the repetitions.

There is a striking similarity between the No-choice algorithm and the algorithm of Brélaz. Both of them may be viewed as following the rightmost path of the Zykov tree for graph G. The decision to assign the same colour to any two vertices corresponds to the *vertex identification* operation being performed in the Zykov tree. The only difference is the criterion used in choosing the two vertices to be given the same colour. They do not backtrack.

CHAPTER 5

A LAS VEGAS COLOURING ALGORITHM

In this chapter we give a new Las Vegas colouring algorithm i.e. one that is *probabilistic and never lies*, which takes polynomial time on almost all instances of the graph colouring decision problem, so long as k is not too large. The algorithm uses the Zykov tree described in Chapter 3. Of course an efficient algorithm for this decision problem implies an efficient algorithm for determining the chromatic number of a graph, the corresponding optimisation problem.

5.1 Preliminaries

Let $G_1=(V_1,E_1)$ be the graph obtained from G by adding an edge between non-adjacent vertices u and v. Let $G_2=(V_2,E_2)$ be the graphs obtained from G by identifying non-adjacent vertices u and v. Vertices and edges of both resultant graphs are given by:

(a) $V_1=V$; $E_1=E+\{u,v\}$ ——— *edge addition*

(b) $V_2=V-v$ ——— *node identification*;

$\{i,j\} \in E_2$ iff $\{i,j\} \in E$ for $i,j \neq u$.

$\{i,u\} \in E_2$ iff $\{i,u\} \in E$ or $\{i,v\} \in E$.

We note the following two important properties. The proof of (1) can be found in [18] and (2) is obvious:

(1) If either G_1 or G_2 is k–colourable then G is k–colourable.

(2) For any graph H, if H contains a k+1-clique, then H is not k-colourable.

5.2 Algorithm Description

Let G=(V,E) and k>0 be the input to the algorithm for determining whether G is k–colourable. The algorithm branches in two directions by reducing G to G_1 and G_2 using the operations *edge addition* and *vertex identification* as described above.

For any graph, H, thus formed, we **probabilistically** search for a k+1-clique. If it is found, this branch is terminated otherwise the reduction process continues until either:

(i) *All leaves of the resulting pruned Zykov tree for* G *are graphs that contain a k+1-clique, in which case the algorithm terminates with a NO answer;*

(ii) *we create a graph whose number of nodes is less than or equal to* k, *in which case the algorithm terminates with a YES answer.*

A recursive definition of our algorithm is given by Figure 5.1.

```
function colourable (G:graph,k:integer):boolean;
procedure colour(G:graph);
var G1, G2 : graph;
begin
  if |G| ≤ k then foundcolor ← true
  else
  begin
    find(u,v)
    G2 ← identify(u,v);
    G1 ← addedge(u,v);
    if not clique(G2, k+1) then colour(G2 );
    if not foundcolor and not clique(G1, k+1) then colour(G1)
  end
end;
begin
  initialise;
  foundcolor ← false;
  colour(G);
  colourable ← foundcolor;
end;
```

Figure 5.1: Overview of Algorithm.

Two significant features of the algorithm are:

1 Effective pruning of the Zykov tree.

2 Selection of u and v in such a way that identification is unlikely to create a k+1 clique. Since identification reduces the number of nodes in a graph by 1, we select u and v so as to leave the fewest edges. Intuitively, this is likely to be accomplished by choosing u and v such that they have the maximum number of common neighbours. The selection

also has the advantage that should the need arise to add an edge between u and v, when backtracking, it is added where there is a concentration of edges and so builds up larger cliques more quickly.

Assume the vertices of G are numbered $v_1, v_2, \ldots, v_n$. Procedure *initialise* is used to do some preliminary processing. It does the following:

(a) randomly selects a vertex, say x, and returns the set of vertices, together with x , that form an r-clique, $2 \leq r \leq n$, in a global variable **gclique**.

(b) computes an array B where b_i is the number of neighbours that vertex v_i has in the **gclique**. If v_i is in **gclique** then $b_i \leftarrow -1$.

(c) computes a two dimensional array W where w_{ij} is the number of neighbours common to both vertices v_i and v_j.

Part (a) is easily computed by modifying the clique finding function described in Figure 5.3 so that it returns a set of vertices that make the clique. For parts (b) and (c) a nested triple loop will do by going through the representation of the graph and incrementing the necessary array elements.

```
procedure initialise;
begin
  Set elements of array B to zero;
  Set elements of matrix W to zero;
  gclique ← [v1, . . . , vr];     {Obtained by modifying Figure 5.3 }
  for every vertex  vi do
    if (vi ∈ gclique) then B[i]← -1
    else B[i]← number of vertices in gclique adjacent to vi;
  for every pair of vertices vi, vj  j≠i  do
    for every vertex vk such that k ≠ i and k ≠ j do
      if (vi,vk) ∈ E and (vj,vk) ∈ E then
        W[i,j]← W[i,j]+1
end;
```

Figure 5.2: The procedure **initialise**.

Figure 5.3 gives the clique testing algorithm whose running time is linear in the number of nodes [42] and Figure 5.4 gives an algorithm that returns two non-adjacent vertices u and v that have the most common neighbours. This procedure, Figure 5.4, uses the global variables **gclique**, array B and matrix W, to select one vertex v_i not in the clique and another v_j in the clique. These are selected such that v_i is adjacent to most vertices in the clique relative to other vertices not in the clique and not adjacent to v_j. Moreover, v_i and v_j must have the most common neighbours. Neighbours(x) is the set of vertices adjacent to vertex x.

```
function clique(G:graph,r:integer):boolean;
var
   K,S : Set of vertices;
begin
   S ← [1,2...n]
   K ← ∅ ;
   while S ≠ ∅ do
   begin
      Randomly Select x ∈ S
      K ← K ∪ [x];
      S ← S ∩ Neighbours(x);
   end;
   clique ← | K | ≥ r
end;
```

Figure 5.3: Function for Finding a Clique

```
procedure find( var x,y);
begin
   Select v_i such that b_i is maximum;
   Select v_j such that
        (1) v_j ∈ gclique
        (2) v_j and v_i are non-adjacent
        (3) w_ij is maximum over all w_ik, with v_k ∈ gclique and (v_i,v_k) ∉ E
end;
```

Figure 5.4: Procedure for Finding two non-adjacent vertices.

During the reduction of G into either G_1 or G_2, the size of **gclique** may increase as bigger cliques are created. When such a case arises, the algorithm updates the elements of **gclique** and its size and continues with this branch as long as the size of

gclique is not greater than k. Moreover, as each operation is performed, the elements of array B and matrix W have to be updated.

```
function addedge(x,y:integer):graph;
begin
  Make edge (v_x,v_y);
  Update array B and matrix W;
  addedge ← G + {x,y}
end;
```

Figure 5.5: Function Addedge

```
function identify(x,y:integer):graph;
begin
  For every vertex v ∈ Neighbours(v_y) and v ∉ Neighbours(v_x) do
  begin
    Make edge (v_x,v);
    Delete edge (v_y,v);
    Update B[v]
  end;
  Update matrix W;
  identify ← modified graph
end;
```

Figure 5.6: Function Identify.

5.3 Experimental Results

The algorithm was executed using sample graphs from the following families of graphs; *randomly k-colourable, randomly NOT k-colourable* and *sparse graphs*. Formal descriptions of these families are given below.

The *randomly k-colourable* and *randomly NOT k-colourable* graphs were chosen so as to test the performance of the algorithm on the sets of instances where a solution exists (a k-colouring) and on instances where a solution does not exist. This was necessary because we wanted to show that the algorithm performs well on both of these sets, not just one.

Franco [12] observes that existing algorithms which perform well on instances of a decision problem where a solution exists, generally do not perform well on instances where a solution does not exist and vice versa.

The last class, *Sparse k-colourable*, is used to test the performance of the algorithm on graphs with few edges.

Representatives from each class, with the probability $\frac{1}{2}$ of an edge occurring and various k were generated, and the number of times the algorithm *backtracks* for various k were measured. We define backtracks as the number of times a k+1 clique is found. For each n, the number of vertices, twenty graphs were tested and the average number of backtracks was computed. The average number of backtracks was

rounded off to the nearest integer.

5.3.1 Random k-colourable graphs.

When k=3 and $p=\frac{1}{2}$, the algorithm did not have to backtrack for sample random 3-colourable graphs with the number of nodes ranging from 10 to 100 . Figure 5.7 illustrate the average number of backtracks taken by the algorithm, before determining whether G is k-colourable, for $4 \leq k \leq 6$ and increasing n, for sample graphs from this family. Figure 5.8 and 5.9 give results for k=8 and k=10 respectively.

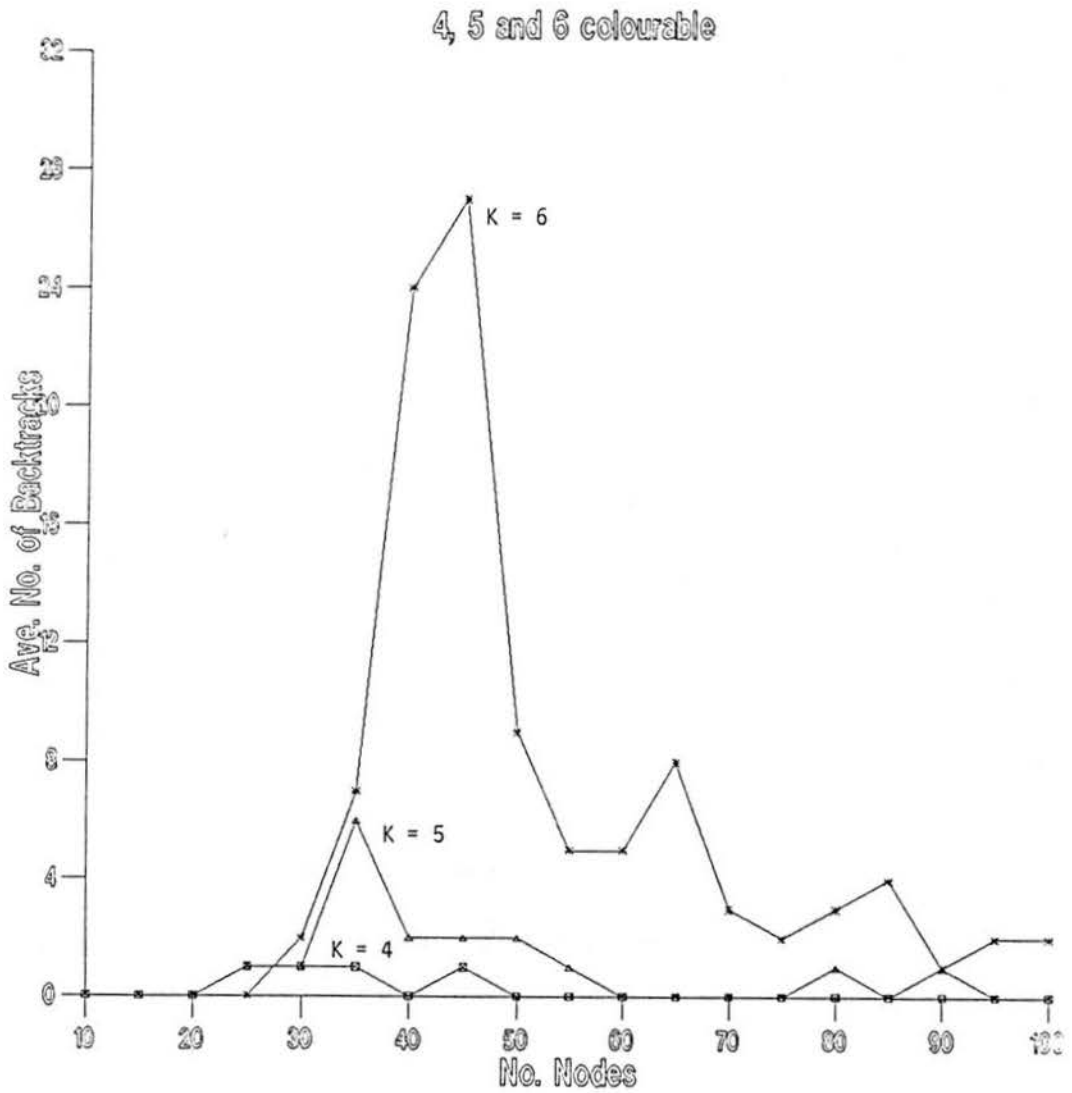


Figure 5.7 : Results obtained for 4, 5 and 6 colourable Graphs

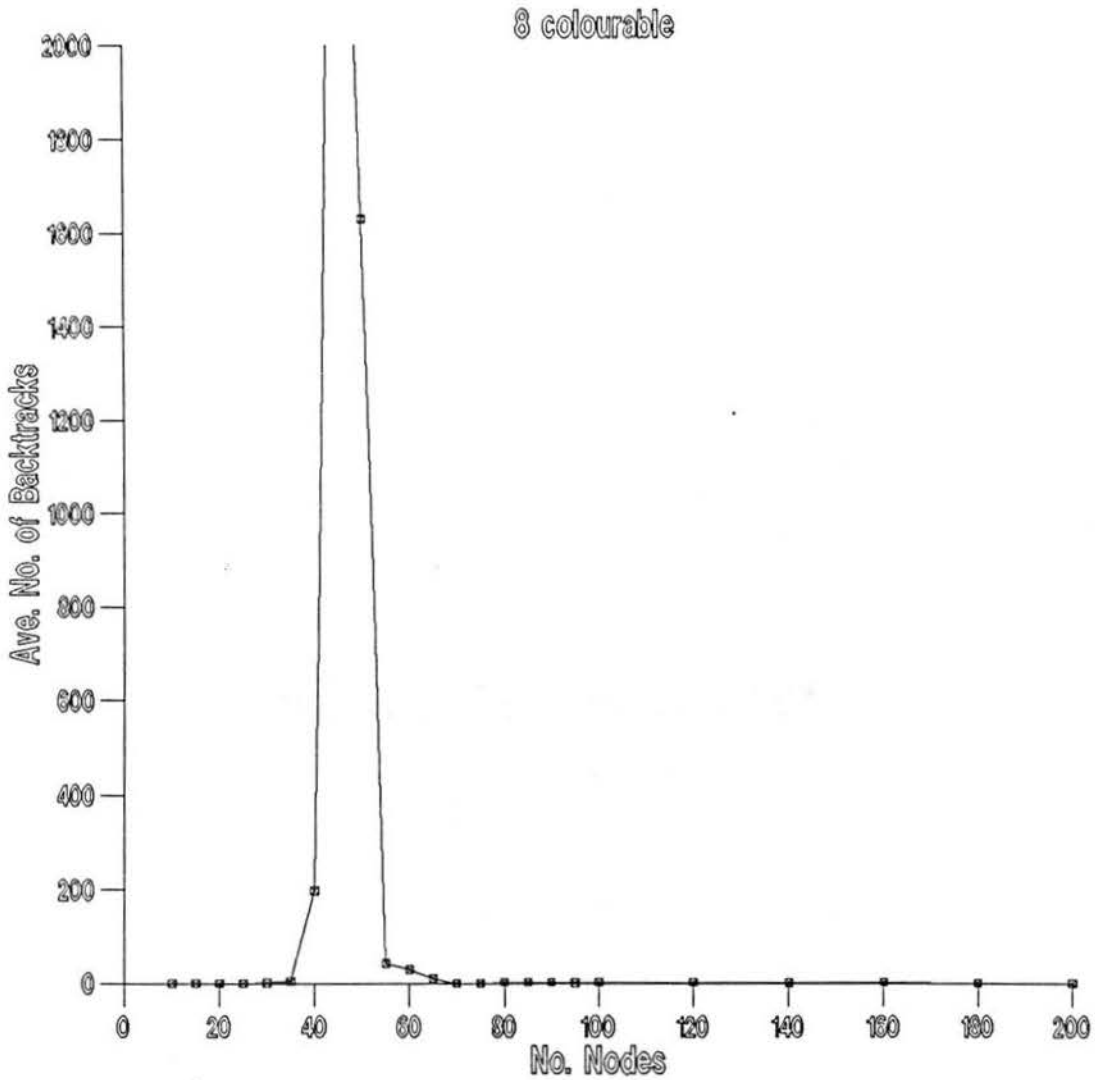


Figure 5.8 : Results obtained for 8-colourable Graphs

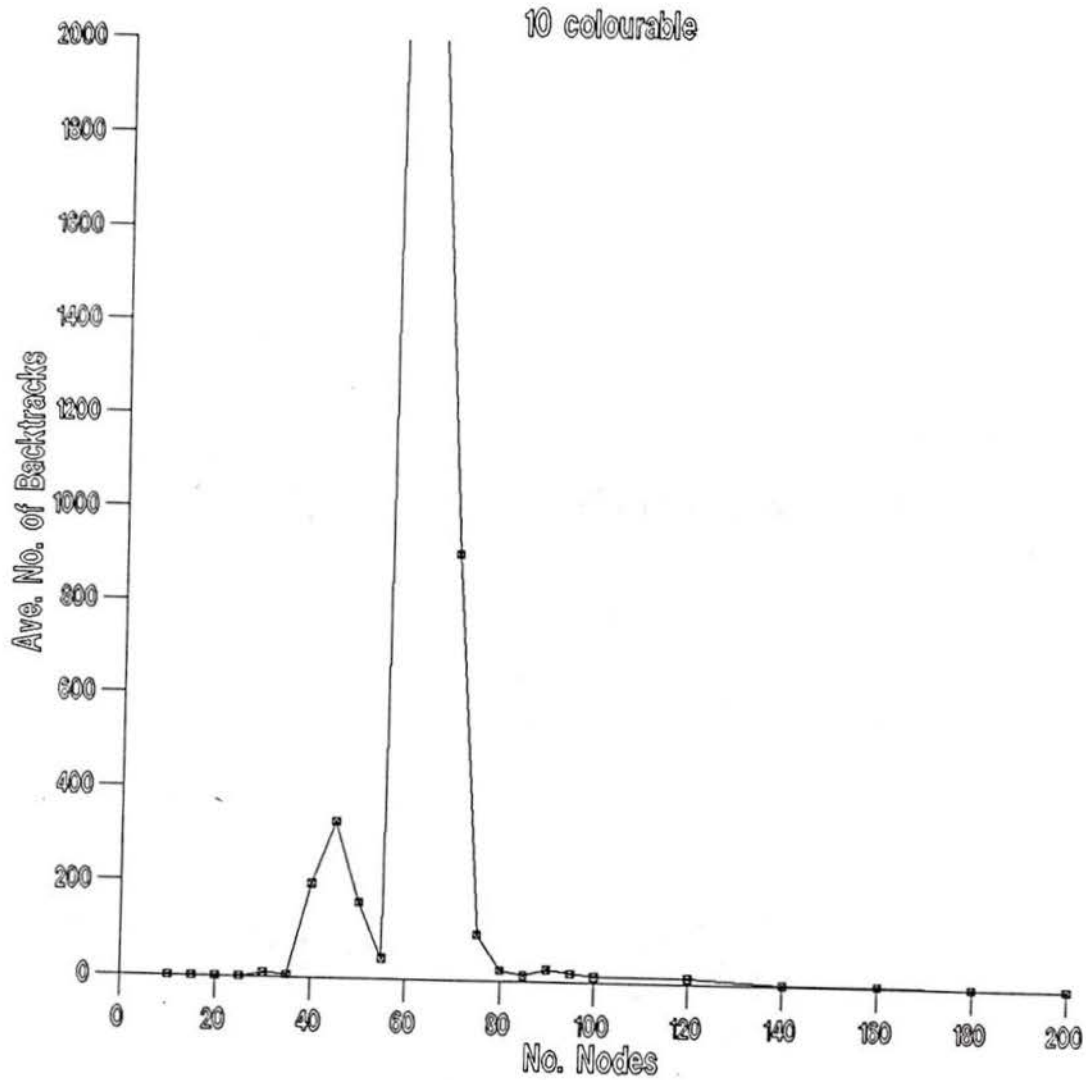


Figure 5.9 : Results obtained for 10-colourable Graphs

Figures 5.7, 5.8 and 5.9 show that the algorithm performs well for $k \leq 7$ where the algorithm terminates within reasonable time. However as k becomes larger than 7, the performance deteriorates in some cases with the algorithm failing to terminate after a reasonable time.

Figures 5.7, 5.8 and 5.9 also show that when $k \geq \log_2(n)$ the algorithm performs poorly for graphs with n vertices. This is consistent with the analysis given in section 5.4 where the limits tend to zero as long as $k \leq \log_2(n)$.

5.3.2 Random NOT k-colourable graphs.

These graphs are generated by the same method as above except that the number of colours used is greater than k. Some of these constructed graphs may be k-colourable. The results obtained for those that the algorithm identifies as k-colourable are discarded. For experimental purposes, those that the algorithm fails to identify as k-colourable or not k-colourable within a reasonable time are assumed to be not k-colourable and the number of backtracks set to infinity for their group.

Randomly k+1-colourable graphs were generated and tested for k-colourability with $3 \leq k \leq 7$. When k=3, the algorithm did not have to backtrack for sample 4-colourable graphs used. Figures 5.10 and 5.11 illustrate the average number of back-tracks taken by the algorithm, before determining that G is NOT k-colourable, for various k, $4 \leq k \leq 7$ and increasing n.

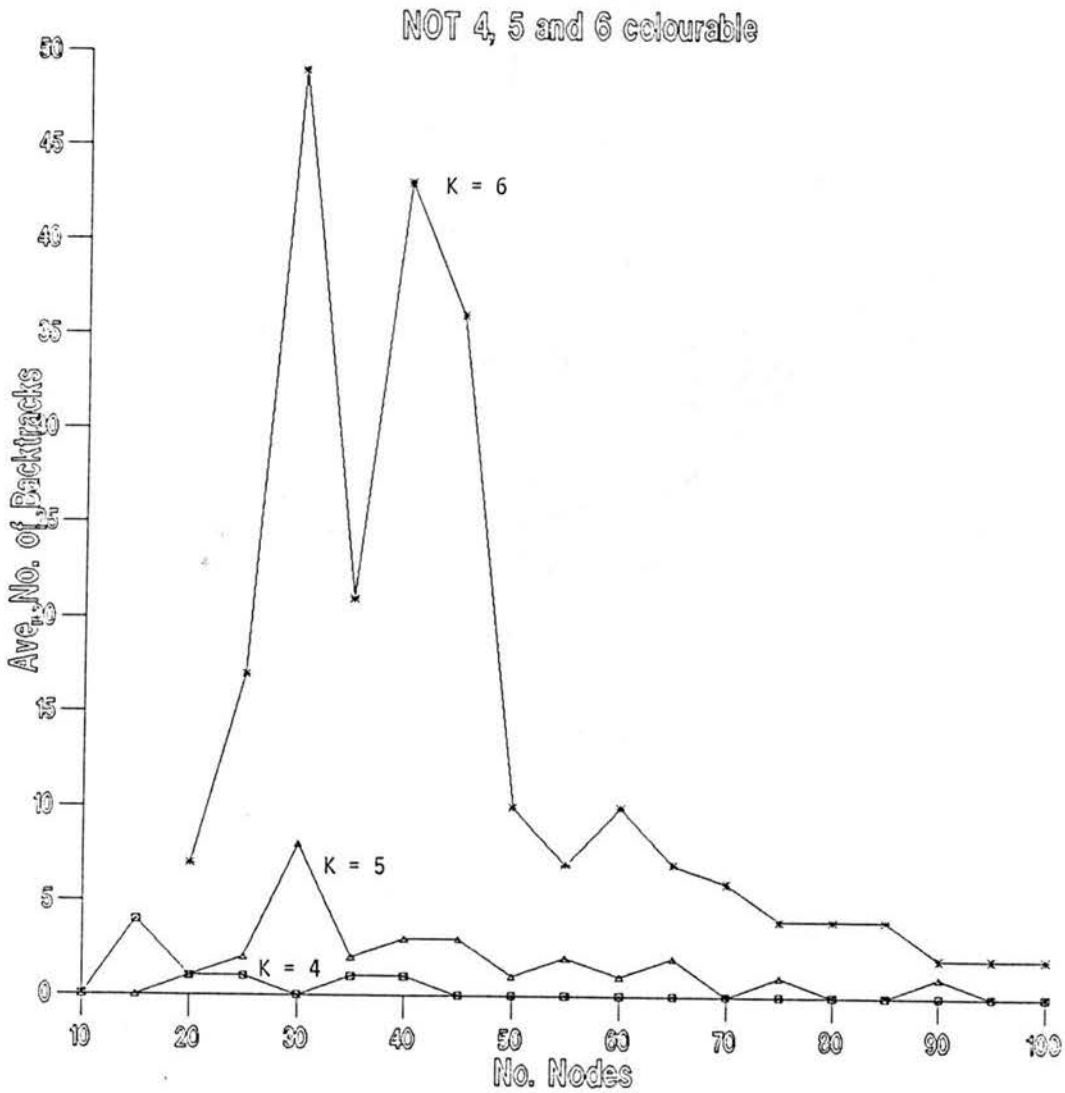


Figure 5.10 : Results obtained for NOT 4, 5 and 6 colourable Graphs

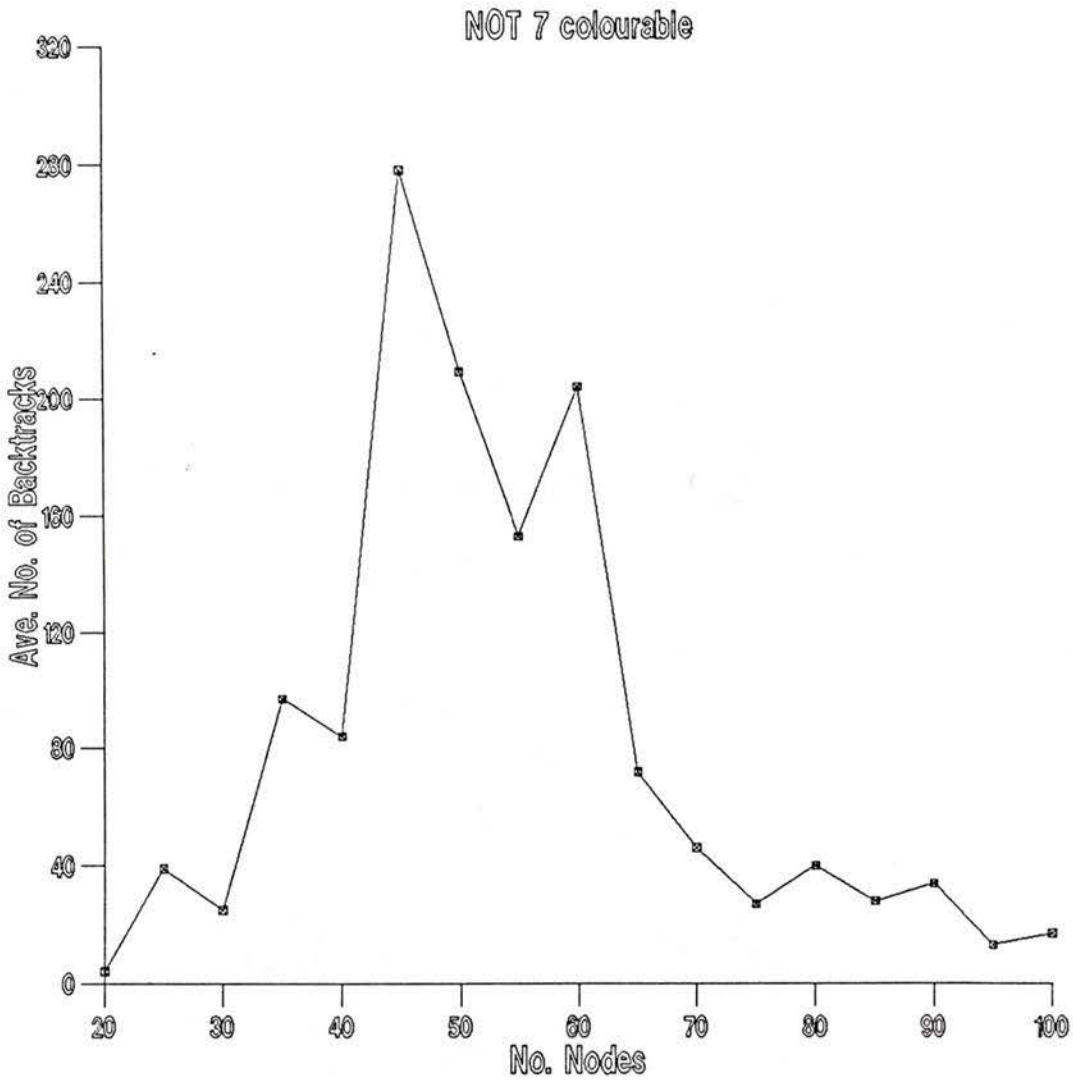


Figure 5.11 : Results obtained for NOT 7-colourable Graphs

For $n \geq 2^k$, the average number of backtracks the algorithm took is very small. The algorithm performs well for $k \leq 7$ even when $k \cong \log(n)$. As evident from the results, the number of backtracks is small in all cases. The results obtained here suggest that the pruning mechanism is very effective.

5.3.3 Sparse k-colourable graphs

These are generated by same the method for *k-colourable* graphs except that the number of edges is bounded above by some constant times the number of vertices. Randomly k-colourable graphs, have a number of edges proportional to n^2.

For this class of graphs, experimental results were obtained for $3 \leq k \leq 5$ only. The algorithm performed poorly for sparse graphs when tested on k larger than 5. In fact no results were obtained for $k \geq 6$ and $n \geq 30$ as the algorithm did not terminate within a reasonable time. For $k \geq 6$ and $n < 30$, the algorithm performed as well as testing for 5-colourability in the same range. We only speculate that the sample graphs generated were also 5-colourable.

The algorithm identified all sparse 3-colourable graphs without backtracking. Figures 5.12 and 5.13 summarise results obtained for k=4 and k=5 respectively.

No theoretical analysis is given about the performance of our algorithm on sparse graphs. However experimental results suggest that we can almost always quickly determine whether a graph is k-colourable for $k \leq 5$.

Because of the low density of edges, it is plausible that the the clique testing function frequently fails to find a k-clique. Also the edges are spread out thinly in the graph and so the algorithm takes longer to build a bigger clique. Thus the pruning mechanism is ineffective for this class of graph and hence the algorithm explores the better part of the Zykov tree.

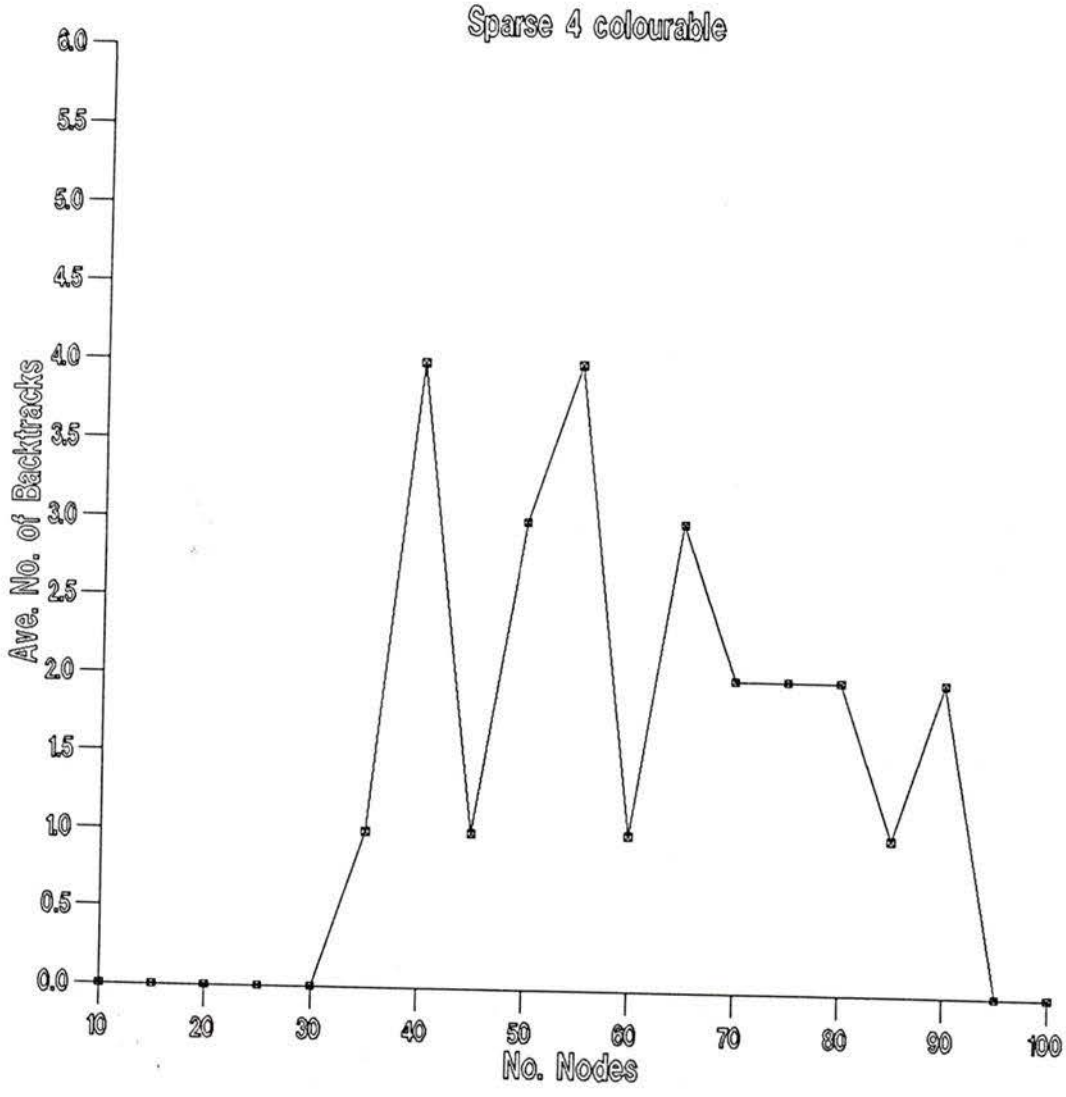


Figure 5.12 : Results obtained for Sparse 4-colourable Graphs

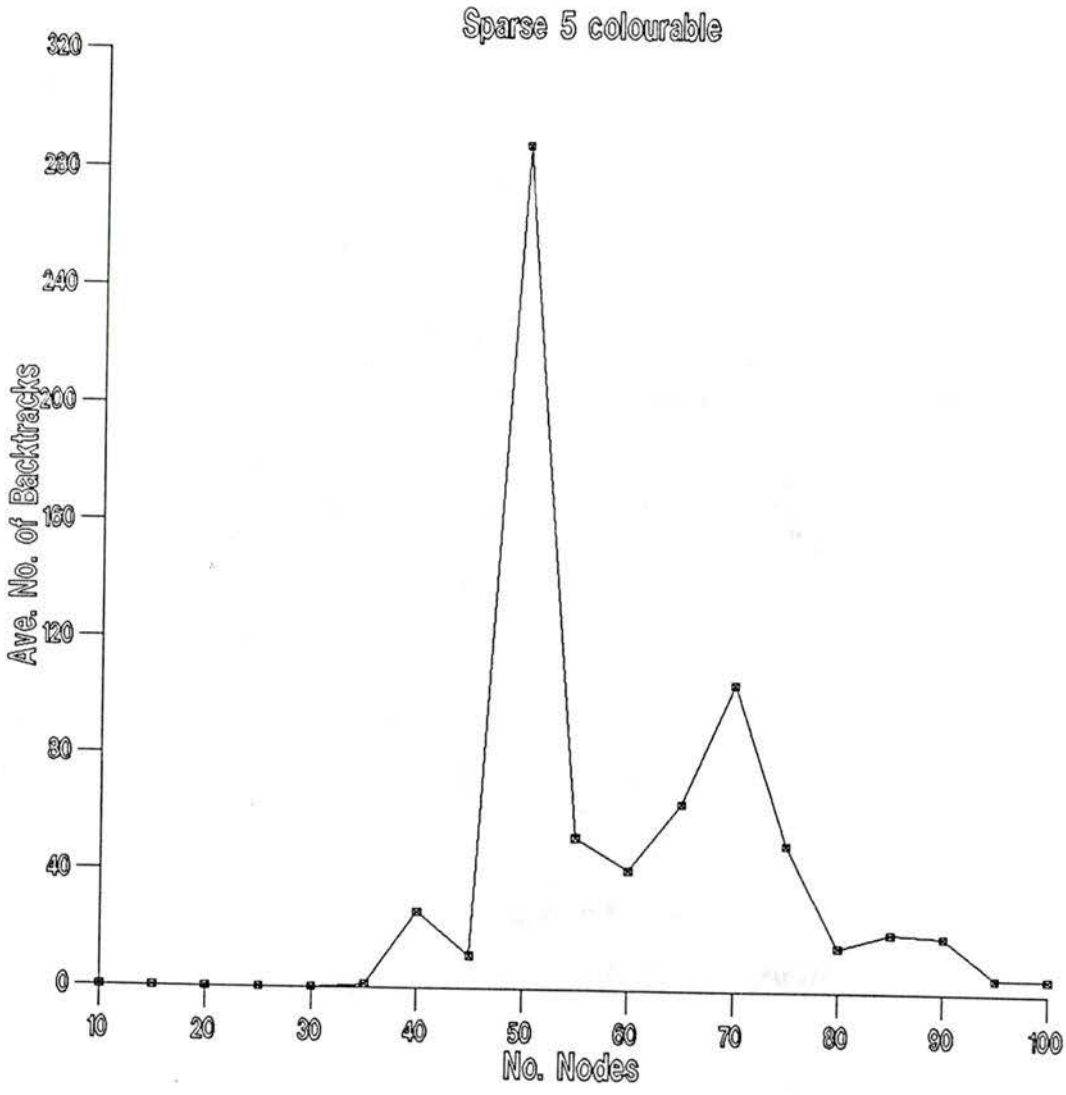


Figure 5.13 : Results obtained for Sparse 5-colourable Graphs

5.3.4 Exploring the leaves of the unpruned Zykov tree

Another experiment examined the proportion (concentration) of complete graphs of size k or less at the leaves of the unpruned Zykov tree for various $k \geq 3$. The algorithm traverses the Zykov tree in a right to left direction; i.e. the algorithm performs the *identification* operation until a clique is found, backtracks to perform *edge addition* operation and continues with the *identification* operation noting the size of cliques encountered. The number of leaves (cliques) scanned by the algorithm was limited to 2n.

Randomly k-colourable graphs of various sizes between 10 and 100 were generated, 10 from each size group. For each k, $3 \leq k \leq 8$ the number of cliques of size k or less were counted and the average percentage calculated. For all sample graphs used, two different vertex selection strategies were used. In the first experiment, the vertices were selected at random. In the second they were selected using the strategy described in Figure 5.4.

Results obtained for vertices selected at random are marked by ∇ and the others by $\boxtimes$ Figures 5.14, 5.15, 5.16 and 5.17 give the percentage of cliques of size k or less for of $k \in [3,4,6,8]$ and n increasing up to 100.

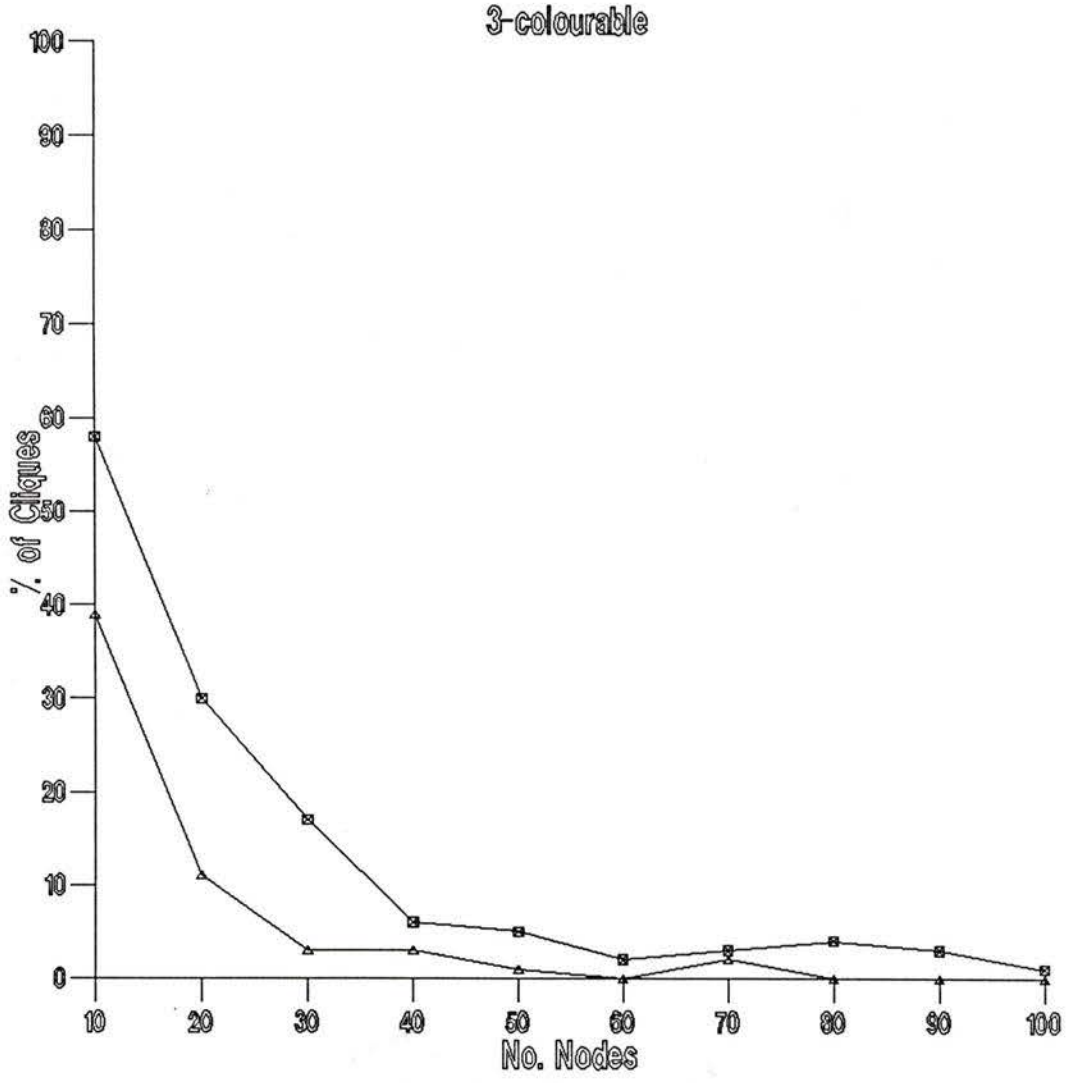


Figure 5.14 : Percentage of cliques of size 3 or less in 3-colourable Graphs

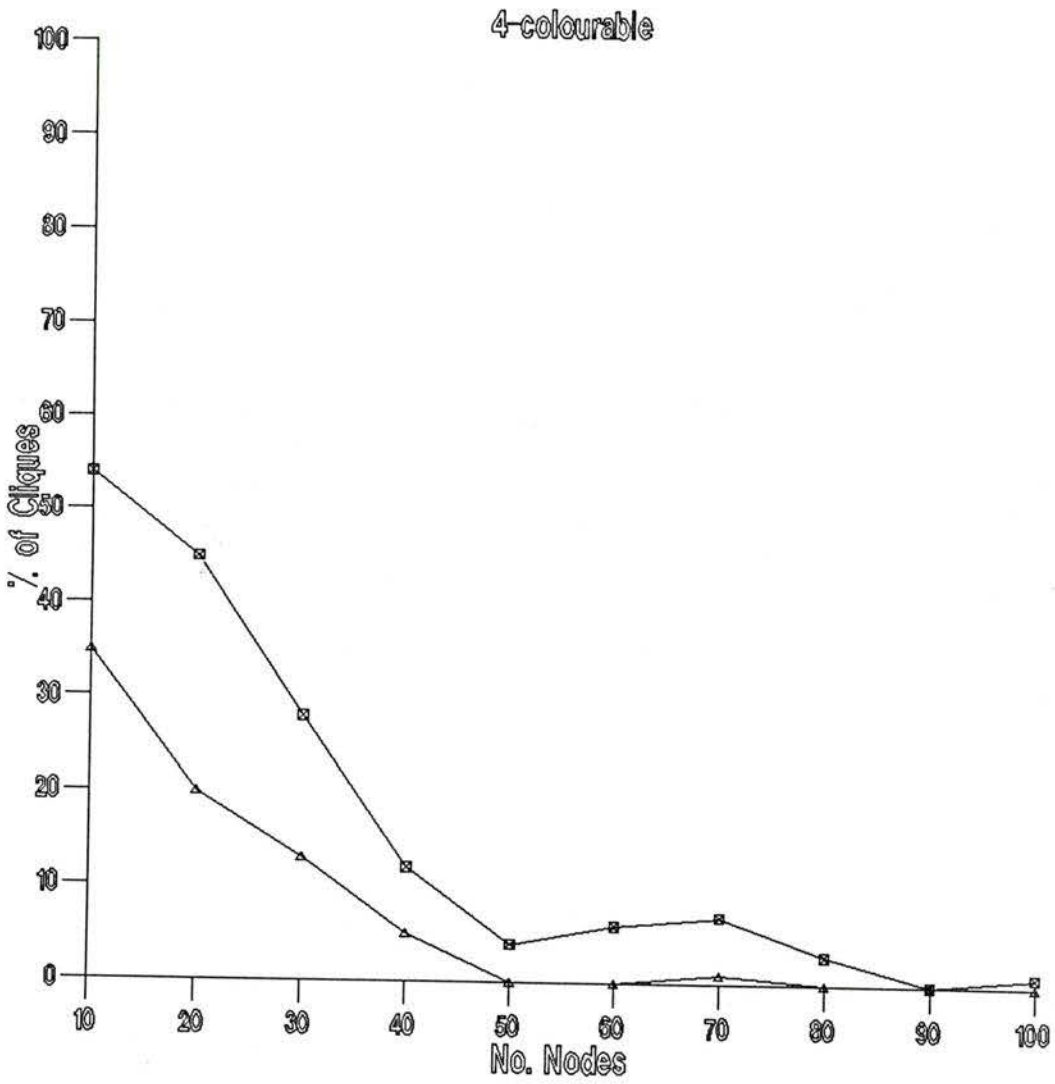


Figure 5.15 : Percentage of cliques of size 4 or less in 4-colourable Graphs

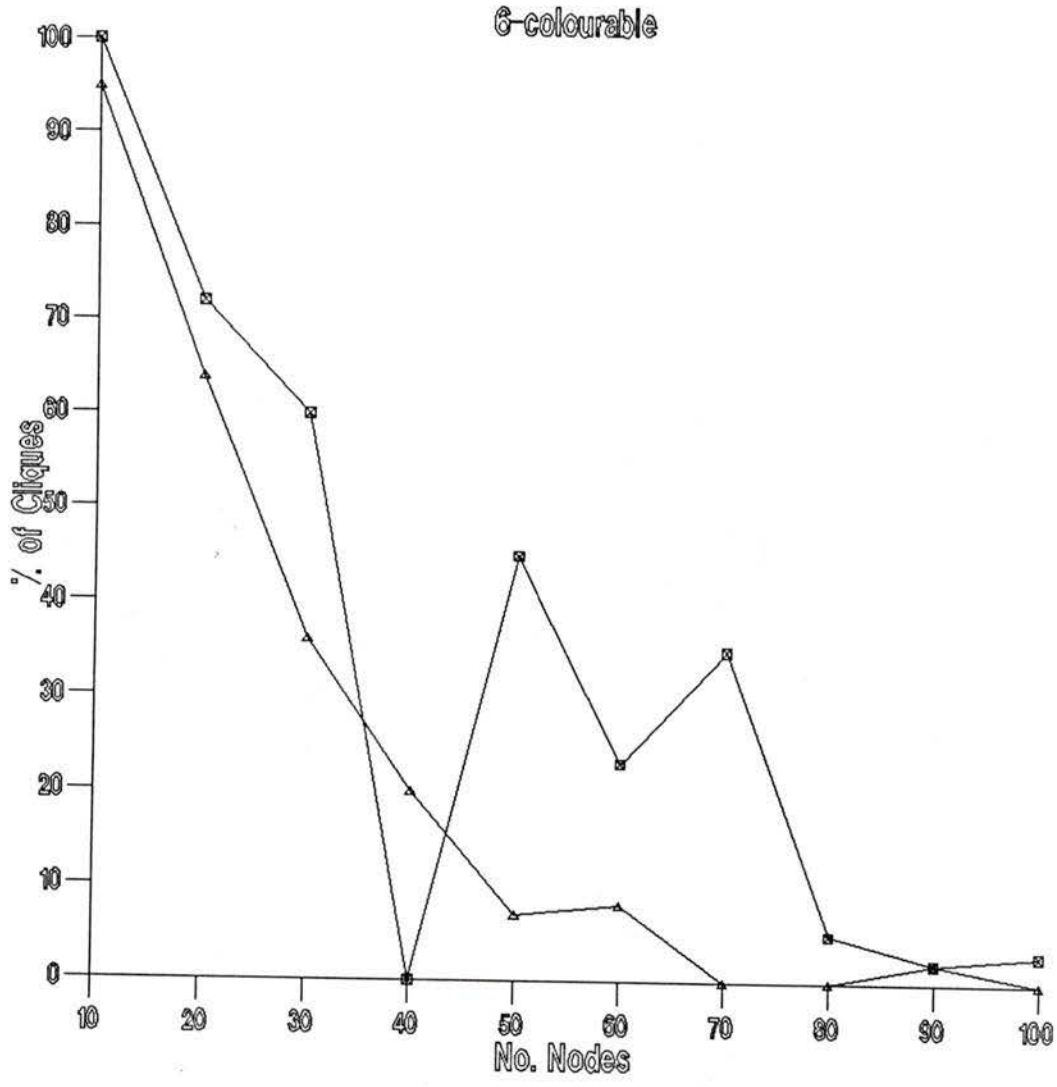


Figure 5.16 : Percentage of cliques of size 6 or less in 6-colourable Graphs

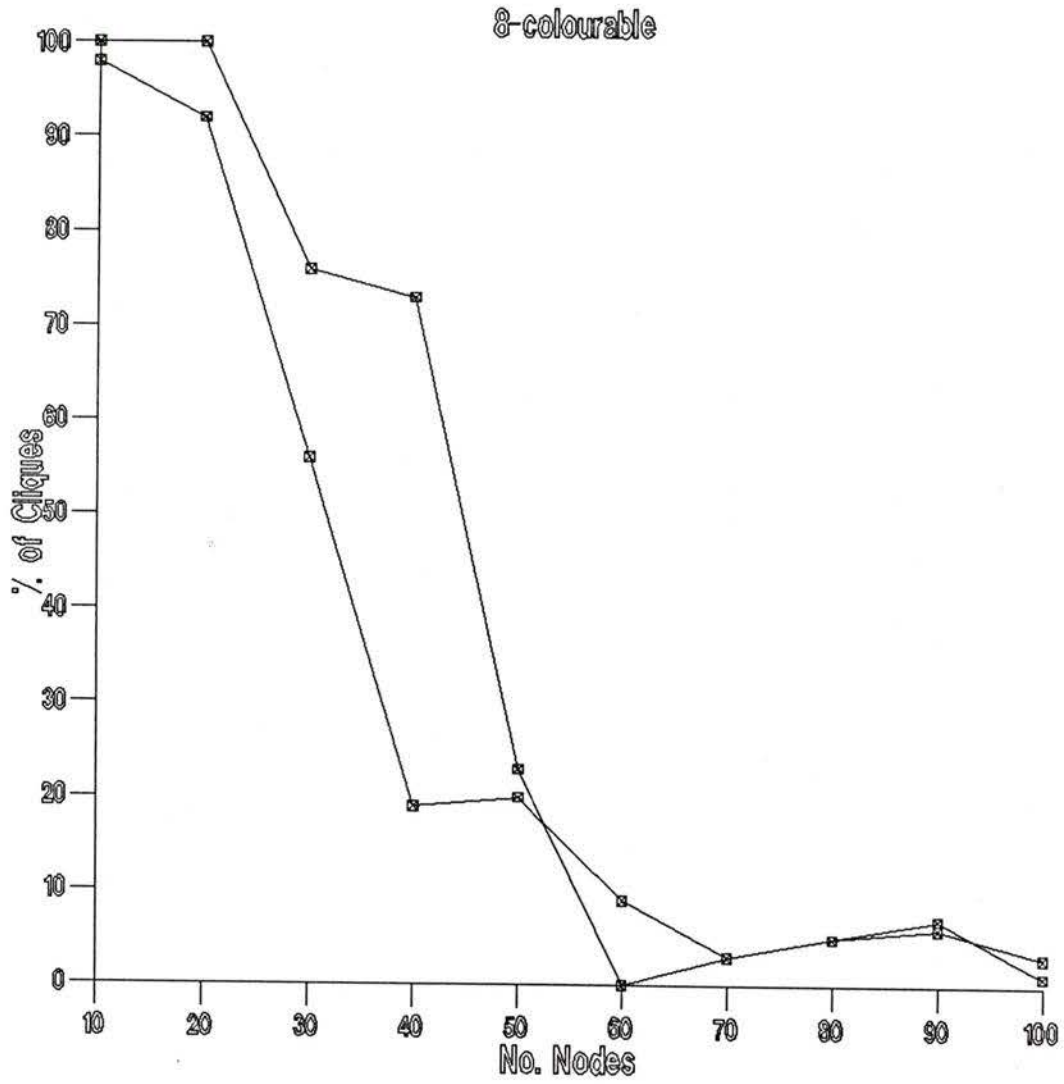


Figure 5.17 : Percentage of cliques of size 8 or less in 8-colourable Graphs

The experimental results obtained here suggest that, for $n \leq 100$ and $3 \leq k \leq 8$, scanning the first 2n leaves of the unpruned Zykov tree almost certainly guarantees finding a clique of size k or less. Places where the percentage of cliques of size k or less is close or equal to zero correspond to the break points obtained in the previous experiment on randomly k-colourable graph.

A probabilistic and practical algorithm might then be constructed along these lines. For cases where we are unable to find a clique of size k or less after scanning a polynomial number of cliques, the new algorithm will then terminate with a **NO** answer. The algorithm will always be correct when it terminates with a **YES**. However, there exists some probability greater than zero of wrongly deciding that a graph is not k-colourable.

The results also indicate that there is some value in selecting pairs of vertices using the strategy of Figure 5.4 rather than a random selection.

5.4 Theoretical Results

5.4.1 k-colourable graphs

In this section we prove that the algorithm given in Figure 5.1 determines if G is k-colourable for almost all randomly k-colourable graphs in polynomial time, when $k \leq \log_2(n)$.

The effectiveness of the algorithm depends on:

1. The finding of an initial k-clique and effective pruning of the tree. The pruning is achieved by testing for a k+1-clique. Once the k-clique has been found, the algorithm proceeds by identifying the remaining vertices with the vertices in the k-clique.
2. The effective selection of two non-adjacent vertices u and v. The strategy is to select u from the initial k-clique and v so that it has the most neighbours in the clique.

Turner [42] shows that for almost all randomly k-colourable graphs the clique finding function finds a k-clique and that there are vertices, not in the k-clique, adjacent to k-1 vertices in the clique.

We say that a k-colourable graph G satisfies the *clique property* if for every $r<k$, all cliques on r vertices can be extended to r+1 vertices.

Let G be a k-colourable graph containing at least one k-clique and let $\{x_1, x_2, \ldots, x_k\}$ be any k-clique in G. Consider the following 3k not necessarily disjoint subsets of the vertices of G $A_1, \ldots, A_k, B_1, \ldots, B_k, C_1, \ldots C_k$, where

$$A_i = \{y \in V \mid \forall j \neq i \ \ \{y, x_j\} \in E \}$$

$$B_i = \{y \in V \mid \forall j \neq i \ \ \exists z_j \in A_j \text{ such that } \{y, z_j\} \in E \}$$

$$C_i = \{y \in V \mid \forall j \neq i \ \ \exists z_j \in B_j \text{ such that } \{y, z_j\} \in E \}$$

Thus we define A_1 as the set of vertices adjacent to every vertex in the k-clique except x_1, A_2 as the set of vertices adjacent to every vertex in the k-clique except x_2 and so on. B_i is the set of vertices that have at least one neighbour in all of the A_j except A_i, $1 \leq i, j \leq k$. Likewise, C_i is defined as the set of vertices that have at least one neighbour in all of the B_j except B_i, $1 \leq i, j \leq k$. We note that $\forall$ i $x_i \in A_i \subseteq B_i \subseteq C_i$.

A k-colourable graph is said to be *easily colourable* if the clique property holds and for all cliques $\{x_1, x_2, \cdots, x_k\}$, $\bigcup_{i=1}^{k} C_i = V$.

Turner [43] showed that almost all random k-colourable graphs G, $G \in \chi(k,p)$, are *easily colourable*, i.e. they have the structure illustrated in Figure 5.18. He also presented a function that finds a k-clique almost always on random k-colourable graphs (see Figure 5.3), because almost all have the *clique property*.

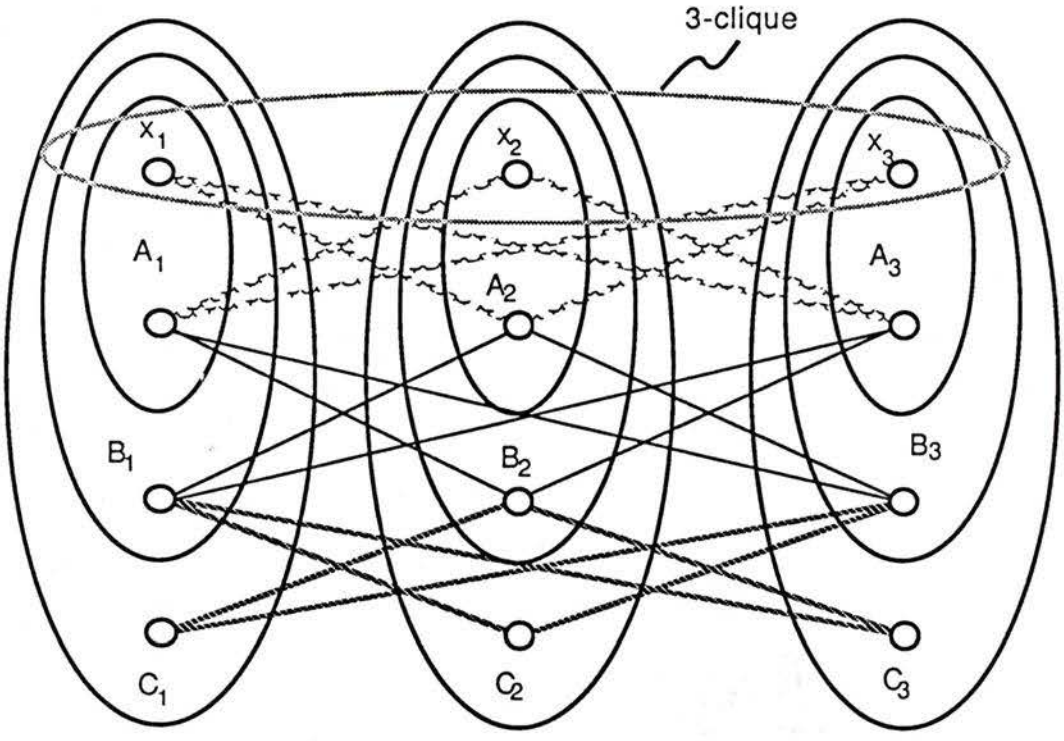


Figure 5.18 : Structure of *easily colourable* graphs.

We define the following disjoint subsets of V given that G is easily colourable.

$K = \{x_1, \ldots, x_k\}$

$A'_i = A_i - K$

$B'_i = B_i - A_i$

$C'_i = C_i - B_i$

Clearly if G is easily colourable then $K \cup A'_i \cup B'_i \cup C'_i = V, \quad 1 \leq i \leq k.$

We now consider the application of the algorithm in Figure 5.1 to random k-colourable graphs. Let $G \in \chi(k,p)$ and k be the input to the algorithm. On random k-colourable graphs, the algorithm can be expressed as follows:

- Find a k-clique $\{x_1, \ldots, x_k\}$.
- While there are non-adjacent vertices do
  - For each vertex y not in the k-clique, calculate $d^*(y)$ the number of neighbours of y in the k-clique
  - Select two non-adjacent vertices u and v such that $d^*(v)$ is maximum and v is in the k-clique
  - For all $y \in$ neighbours(v) - neighbours(u) add the edge {y,u} and delete the edge {y,v}.
  - remove vertex v

Turner [43] shows that the probability that a graph $G \in \chi(k,p)$ does not satisfy the clique property is $\leq kn^k e^{-np^k/2k}$ which tends to zero if $k \leq \log_2(n)$. Therefore the first step will almost always be successful on random k-colourable graphs. The algorithm then proceeds to find a vertex adjacent to the largest number of vertices in the clique. Because of the structure of G, the vertex that is selected has k-1 neighbours in the k-clique. Thus we select one of the vertices from any of the set A'_1, A'_2 up to A'_k.

The selection is always successful as long as $A'_i \neq \varnothing \ \forall$ i. We now show that when the loop terminates for graphs with the structure described above, the graph is reduced to a k-clique.

The algorithm selects vertices in the A′ sets. As they are identified, vertices in the B′ sets transfer to the A′ sets as soon as they are adjacent to k-1 vertices in the k-clique. Similarly vertices in the C′ sets move to the B′ sets as soon as their neighbours in each of the B′ sets moves to the A′. Thus all vertices eventually end up in the A′ sets and are then selected for identification. Hence the algorithm terminates with a k-clique.

Turner [44] showed that almost all k-colourable graphs are *easily colourable* as well. The above analysis is summarised by the following theorem whose proof follows immediately.

Theorem 1: *The algorithm presented in Figure 5.1 determines whether G is k-colourable in* **polynomial** *time for almost all k-colourable graph and* $k \leq \log_2(n)$.

The following corollary is an immediate effect of the selection criterion used in finding two non-adjacent vertices to be identified.

Corollary 5.1: *The algorithm described in Figure 5.1 yields a 2-colouring for all bipartite graphs in polynomial time.*

5.4.2 Not k-colourable

Let R be the set of random graphs. We wish to show that the *clique property* holds for almost all random graphs as long as $k \leq \log_2(n)$. Let K_r be any clique of size $r < k$. Let x be any vertex in the graph but not a member of K_r. The probability that there x is adjacent to each and every vertex in K_r is p^r. Therefore the probability that x is not adjacent to every member of K_r is $(1-p^r)$. The probability that there is no vertex adjacent to all vertices in K_r is $\leq (1-p^r)^{n-r}$. There are $\leq n^r$ ways to select K_r, so the probability that an r-clique cannot be extended is

$$\leq \sum_{r=1}^{k-1} n^r (1-p^r)^{n-r}$$

$$\leq \sum_{r=1}^{k-1} n^r e^{-p^r n-r}$$

$$\leq k n^k e^{-np^k} \quad \rightarrow 0$$

Since $k \leq \log_2(n)$. ■

It follows that the clique finding function almost always find a clique. For almost all not k-colourable graphs, the algorithm will almost always find a clique of size greater than k and easily determine that G is not k-colourable.

CHAPTER 6

SUMMARY

Summary

In Chapter 2 we discussed probabilistic algorithms for various decision problems and reviewed the probabilistic Turing machine. Relations between complexity classes induced by probabilistic Turing machine and the classes **P**, **NP** and **PSPACE** were discussed. Zachos, Ko, Karp and Lipton [24, 25, 51] showed that it is unlikely that fast, probabilistic algorithms, with bounded error probability on every instance, exist for **NP-complete** problems.

In Chapter 3, the Zykov tree and graph colouring algorithms that use the tree reviewed. McDiarmid [29] showed that the expected size of the Zykov tree for G grows exponentially with the number of node in G. Corneil and Graham [8] showed that their algorithm, which uses the Zykov tree, performs well in practice compared to that of Christofides.

In Chapter 4 existing probabilistic colouring algorithms were reviewed. All were found to be *Monte Carlo* and performed well for small k on k-colourable graphs.

A new *Las Vegas* colouring algorithm, that uses the Zykov tree, is presented in Chapter 5. It is shown that the algorithm runs in polynomial time for almost all

instances of the graph colouring problem when $k \leq \log_2(n)$. Experimental results indicate that the algorithm also works well for small k on sparse graphs. A new approximation algorithm for the corresponding optimisation problem is also suggested.

Further Research

We have shown that the new *Las Vegas* algorithm runs in polynomial time *almost always* for $k \leq \log_2(n)$. However we were unable to prove that the average time complexity is polynomial. A remaining open problem is to design a colouring algorithm that runs almost always in polynomial time for large k. The algorithm that works well for large k together with the one presented in this thesis can be used together to solve the graph colouring problem almost always in polynomial time.

Bibliography

[1] D. Angluin and L. G. Valiant, Fast probabilistic algorithms for Hamiltonian Circuits and Matchings, *Journal Of Computer and System Sciences 18*, (1979), 155-193.

[2] L. Babai and L. Kucera, Canonical labelling of graphs in Linear Average time, *Proceedings 20th Ann. Symp. on Foundations of Computer Science*, , 1979.

[3] E. A. Bender and H. S. Wilf, A Theoretical analysis of Backtracking in graph coloring Problem, *Journal of Algorithms 6*, (1985), 275-282.

[4] D. Brélaz, New Methods to color the vertices of a graph, *Communications of ACM 22 (4)*, (1979), 251-256.

[5] R. J. Brown, Chromatic Scheduling and The chromatic number Problem, *Management Science 19 (4)*, (December, 1972), 456-463.

[6] N. Christofides, An algorithm for the chromatic number of a graph, *Comput. J. 14*, (1971), 38-39.

[7] V. Chvátal, Probabilistic Methods in Graph Theory, *Annals of Operations Research 1*, (1984), 171-182.

[8] D. G. Corneil and B. Graham, An algorithm For determining the Chromatic Number of a Graph, *SIAM J. Comput. 2 (4)*, (December, 1973), 311-318.

[9] R. D. Dutton and R. C. Brigham, A new graph colouring algorithm, *The Computer J. 24*, (1981), 85-86.

[10] P. Erdös and J. Spencer, in *Probabilistic Methods in Combinatorics*, Academic Press, London, 1974.

[11] P. Erdös and B. Bollobás, Cliques in random graphs, *Math. Proc. Cambridge Phil. Soc. 80*, (1976), 419-427.

[12] J. Franco, Sensitivity of Probabilistic Results on Algorithms for NP-complete Problems to input distributions., *SIGACT News 17 (1)*, (Summer 1985), 40-59.

[13] R. Freivalds, Fast Probabilistic Algorithms, *Lecture Notes in Computer Science 74*, (1979), 57-69, Springer-Verlag.

[14] M. R. Garey and D. S. Johnson, The Complexity of Near-Optimal Graph Coloring, *Journal of ACM 23*, (1976), 43-49.

[15] M. R. Garey and D. S. Johnson, in *COMPUTERS AND INTRACTABILITY: A Guide to the Theory of NP-Comleteness*, W. H. Freeman And Company, San Francisco, 1979.

[16] J. Gill, Computational Complexity of Probabilistic Turing Machines, *SIAM J. Comput. 6*, (1977), 675-695.

[17] G. R. Grimmett and C. McDiarmid, On colouring random graphs, *Math. Proc. Cambridge Phil. Soc. 77*, (1975), 313-324.

[18] F. Harary, in *Graph Theory*, Addison-Wesley, Reading , Mass., 1969.

[19] D. S. Johnson, Worst Case Behaviour of Graph Colouring Algorithms, *Proc. Fifth Southeastern Conf. on Combinatorics, Graph Theory and Computing*, Winnipeg, 1974, 513-527.

[20] D. S. Johnson, NP-completeness column: An Ongoing Guide, *Journal Of Algorithms 5 (3)*, (September, 1984), 433-447.

[21] D. S. Johnson, NP-completeness column: An Ongoing Guide, *Journal of Algorithms 5 (2)*, (June, 1984), 278-299.

[22] R. M. Karp, Reducibility Among Combinatorial Problems, in *Complexity of Computer Computations*, J. W. Thatcher and R. E. Miller (ed.), Plenum Press, New York, 1972, 85-104.

[23] R. M. Karp, The Probabilistic Analysis Of some Combinatorial Search Algorithms, in *Algorithms and Complexity; New Directions and Recent Results*, J. F. Traub (ed.), Academic Press, Inc., New York, 1976, 1-19.

[24] R. M. Karp and R. J. Lipton, Some connections between nonuniform and uniform complexity classes, *Proceedings 12th Ann. ACM Symp. on Theory of Computing*, Los Angeles, 1980, 302-309.

[25] K. Ko, Some Observations on Probabilistic Algorithms and NP-Hard Pro blems, *Information Processing Letters 14*, (1982), 39-43.

[26] C. Lautemann, BPP and the Polynomial Hierarchy, *Information Processing Letters 17*, (1983), 215-217.

[27] L. A. Levin, Problems, complete in average instance, *Proceedings 16th Ann. ACM Symp. on Theory of Computing*, Los Angeles, 1984.

[28] D. W. Matula, G. Marble and J. D. Isaacson, Graph colouring algorithms, in *Graph Theory and Computing*, R. C. Read (ed.), Academic Press, New York, 1972, 39-50.

[29] C. McDiarmid, Determining the Chromatic Number of a Graph, *SIAM J. Comput. 8 (1)*, (1979), 1-14.

[30] C. McDiarmid, Colouring Random Graphs, *Annals of Operations Research 1*, (1984), 183-200.

[31] G. Miller, Reimann's Hypothesis and Test for Primality, *Proc. of the Seventh Anual ACM symp. on Theory of Computing*, , 1975, 234-239.

[32] J. Mitchem, On various algorithms for estimating the chromatic number of a graph, *The Computer Journal 19*, (1976), 182-183.

[33] L. Posa, Hamilton Circuits in Random Graphs, *Discrete Mathematics 14*, (1976), 359-364.

[34] V. R. Pratt, Every Prime has a Succint Certificate, *SIAM Journal on Comput. 4*, (1975), 214-220.

[35] M. O. Rabin, Probabilistic Algorithms, in *Algorithms and Complexity; New Directions and Recent Results*, J. F. Traub (ed.), Academic Press, Inc., New York, 1976, 21-39.

[36] M. O. Rabin, Probabilistic Algorithm for Testing Primality, *J. of Number Theory 12*, (1980), 128-138.

[37] S. I. Roschke and A. L. Furtado, An algorithm for obtaining the chromatic number and an optimal colouring of a graph, *Information Processing Letters 2*, (1973), 34-38.

[38] G. Schmidt and T. Strohleim, Timetable construction - an annotated bibliography, *Comput. J. 23*, (1980), 307-309.

[39] R. Solovay and V. Strassen, A Fast Monte Carlo test for primality, *SIAM J. comput. 6*, (1977), 84-85.

[40] L. J. Stockmeyer, Planar 3-Colorability is NP-complete, *SIGACT News*, , 1973, 19-25.

[41] L. J. Stockmeyer, The Polynomial Heirarchy, *Theoretical Computer Science 3*, (1977), 1-22.

[42] J. S. Turner, A linear Time Algorithm for Recognizing Random k-colorable Graphs, *Technical Report*, St. Louis, Missouri, May, 1984.

[43] J. S. Turner, On the Probable Performance of Graph Coloring Algorithms, *Technical Report WUCS-85-7*, St Louis, Missouri, June, 1985.

[44] J. S. Turner, Almost all k-colorable Graphs are Easy to Color, *Technical Report WUCS-86-2*, St Louis, Missouri, February, 1986.

[45] M. B. Vitányi, How well can a graph be n-coloured?, *Discrete Mathematics 34*, (1981), 69-80.

[46] D. J. Welsh and M. B. Powell, An upper bound to the chromatic number of a graph and its application to time tabling problems, *The Computer Journal 10*, (1967), 85-86.

[47] D. J. A. Welsh, Randomised Algorithms, *Discete Applied Mathematics 5*, (1983), 133-145.

[48] A. Wigderson, Improving the Performance Guarantee for Aproximate Graph Coloring, *Journal of the ACM 10*, (1983), 729-735.

[49] H. S. Wilf, Backtrack: an O (1) Expected Time Algorithm for the Graph coloring Problem, *Information Processing Letters 18*, (1984), 119-121.

[50] D. C. Wood, A Technique for colouring a graph applicable to large scale timetabling functions, *Computer Journal 12*, (1969), 317-319.

[51] S. Zachos, Robustness of Probabilistic Computational Complexity Classes under Definitional Pertubations, *Information and Control 54*, (1982), 143-154,

Academic Press.

[52] A. A. Zykov, On some properties of linear complexes, *Mat. Sb. 24*, (1949), 163-188. English transl. Amer. Math. Soc. Translation no. 79 (1952).

VITA

Surname: Lepolesa Given Names: Pikie Monaheng

Place of Birth: Mafeteng, Lesotho Date of Birth: May 5th, 1961

Educational Institutions Attended, with Dates of Entering and Leaving:

National University of Lesotho 1978 to 1982

University of Victoria, B.C. 1983 to 1986

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions:

B.Sc. (Mathematics) 1982 National University of Lesotho.

Honors and Awards:

W.U.S.C. Fellowship 1983/86

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make *single copies only* for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Probabilistic Graph Colouring Algorithms Using Zykov Trees

Author [redacted]

Pikie M. Lepolesa

20th August '86

Date