THESIS ON INFORMATICS AND SYSTEM ENGINEERING C26

# Some Practical Algorithms to Solve The Maximum Clique Problem

DENISS KUMLANDER

Faculty of Information Technology

Department of Informatics

TALLINN UNIVERSITY OF TECHNOLOGY

Dissertation was accepted for the commencement of the degree of Doctor of Philosophy in Engineering on November 07, 2005.

Supervisor: Prof Rein Kuusik, Faculty of Information Technology

Opponents: Prof Dr Mati Tombak, University of Tartu, Estonia
Prof Dr Patric RJ Östergård, Helsinki University of Technology, Finland

Commencement: December 09, 2005

Declaration: Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any degree or examination.

*Deniss Kumlander /*

ISSN 1406-4731
ISBN 9985-59-581-5

# Abstract

It was found a long time ago that there are some problems that demand a lot of time to find a solution, although it seems to be very simple. Later tremendously quick evolution of computers has greatly motivated researches of algorithms and those problems. Mainly researches have concentrated on problems that are hard to solve algorithmically. A lot of such problems are abstracted into graph theory problems, since graph theory provides a possibility to represent an essence of a problem by distancing from its unimportant details.

The main topic of this thesis is the maximum clique finding from an arbitrary undirected graph. This task is well known to be NP-hard, so nobody has found an algorithm that can solve it in a polynomial time. Nowadays this problem has been recognised as a sub-problem in a plenty number of tasks in different people activities areas like medicine, transportation, design, geology, sociology and many others. Therefore any better algorithm for the maximum clique finding providing could be very important since would improve performance of different tasks in a lot of areas. Besides, such a high implication for real economical problems promote us to concentrate in our study also on practical aspects of the maximum clique finding. We tried to make thesis results to be easy to implement and therefore applicable for solving real problems.

In this thesis we first start from algorithms for finding the maximum clique from both weighted and unweighted graphs. We use the branch and bound type of algorithms to find maximum cliques and propose several algorithms for both graph cases. The main contribution of this thesis is a technique for finding the maximum clique using colour classes, which are produced by a heuristic vertex colouring. Unlike earlier algorithms those colour classes are found only once and then rapidly used to prune branches on all algorithm steps. A proposed algorithm of recalculating the number of existing colour classes is extremely important here. Unlike earlier attempts of using vertex colouring, this algorithm makes vertex-colouring pruning quick and efficient. Thereafter several step-by-step examples of using the proposed algorithms are demonstrated and analysed to identify why those are working effectively. Practical recommendations on algorithms programming are also provided to ensure their correct implementation. Comparative tests on random and DIMACS graphs are conducted and those tests demonstrate that the invented algorithms are quicker in most cases than the algorithms, which are known to be best at the moment. Especially sufficiently is the difference shown on dense graphs, where the new algorithms are up to 100 times quicker.

A test environment for maximum clique finding algorithms has been worked out during this study. A model of it is described in detail and consists of algorithms or modules, utilities, a meta-algorithm and a user interface. The meta-algorithm is a core of testing environment, which conducts tests and collects information. First of all each element of the model is described to highlight most important aspects. Thereafter we demonstrate how all those elements could be integrated including data flows and standards needed to make all parts of the testing environment work smoothly.

Finally we discuss algorithms intelligence in the maximum clique finding area and propose some ideas of a meta-algorithm and its implementation. This part of this thesis concentrates ideas derived from the previously described testing environment, algorithms for finding the maximum clique, expert systems and knowledge we obtained from algorithms' testing that was done in this study. We start from reviewing a real-time systems' case, i.e. the case when an algorithm can be suddenly stopped and the best result found so far returned. A new term – "incomplete solution" is introduced to identify a moment of finding a clique, which is already the maximum one although is not yet proved to be the maximum. Using this term we can analyse best known algorithms' performance on real-time systems. This is an important type of information for meta-algorithms. In the thesis we propose two types of meta-algorithms – a fixed rules meta-algorithm and an evolving one. We demonstrate how those meta-algorithms can be built and discuss different meta-algorithms' learning approaches.

**Keywords**: maximum clique, graph theory, branch and bound, incomplete solution, vertex-colouring, colour classes, algorithm, NP-hard, NP-complete, meta-algorithm, test environment.

# Kokkuvõte

## Mõned praktilised algoritmid suurima kliki probleemi lahendamiseks

Juba päris ammu leidsid inimesed, et on olemas probleeme, mille lahendamine nõuab palju aega, kuigi need algselt tunduvad olevat väga lihtsad. Arvutite kiire areng motiveeris oluliselt niisuguste probleemide ja algoritmide uurimist. Uurijad keskendusid peamiselt probleemidele, mis olid keerulised. Paljud niisugused probleemid on koondunud graafiteooriasse ja taandatavad graafidele, kuna see võimaldab esitada probleemi olemust vähetähtsatest detailidest vabana.

Käesoleva väitekirja peamiseks uurimisobjektiks on suurima kliki leidmine suunamata lõplikust graafist. Keerukusteooriast on teada, et see ülesanne on NP-keeruline ja senini pole leitud algoritmi, mis lahendaks selle ülesande polünomiaalse aja jooksul. Tänapäeval on see ülesanne identifitseeritud alamosana paljudes inimese tegevusvaldkondades nagu meditsiin, transport, disain, geoloogia, sotsioloogia jne. Seetõttu võib iga efektiivsem algoritm suurima kliki leidmiseks osutada väga tähtsaks, kuna võimaldab säästa palju aega erinevate valdkondade ülesannete lahendamisel. Sellepärast keskendusime suurima kliki leidmisele praktilisest aspektist lähtuvalt, proovides luua algoritme, mis ei oleks keerulised realiseerida ja oleks rakendatavad praktiliste ülesannete lahendamisel.

Käesolevas töös käsitletakse suurima kliki leidmise algoritme kaalutud ja mitte kaalutud graafide jaoks. Suurima kliki leidmise uute algoritmide loomisel kasutasime baasalgoritmidena „haru-ja-piir” tüüpi algoritme ja värviklasside kärpimise tehnikat, kus klasse genereeritakse heuristilise tippude värvimise abil. Teiste algoritmidega võrreldes seisneb peamine erinevus selles, et värviklassid leitakse ainult üks kord, töö alguses, samas kasutatakse neid pidevalt harude kärpimisel. Kirjeldatud lähenemine võimaldas tõsta algoritmide efektiivsust kuni kaks suurusjärku. Töös esitatakse ka näited uute algoritmide töösammude demonstreerimiseks ja analüüsitakse neid näiteid selgitamaks, miks uued algoritmid töötavad nii efektiivselt. Töös avaldatakse ka programmeerimise nõuanded, mis kindlustavad algoritmide õige ja efektiivse realiseerimise ning rakendamise. Loodud algoritme testiti DIMACS ning juhuslikel graafidel, mis näitas, et need on enamikul juhtudel efektiivsemad senituntuist. Eriti suur erinevus tekkib tihedatel graafidel, kus uued algoritmid töötavad kuni 100 korda kiiremini.

Töö käigus arendati välja suurima kliki leidmise algoritmide testimiskeskkond. Väitekirjas kirjeldatakse testkeskkonna mudelit, mis koosneb algoritmidest (mida testitakse), vahenditest, meta-algoritmist ja kasutajaliidesest. Meta-algoritm on testimiskeskkonna tuum – see sisaldab algoritmi, testib neid ning kogub tulemused ja statistika. Ka näidatakse, kuidas need elemendid integreeruvad koos andmevoogude ja sisemise standardite kirjeldamisega.

Väitekirja lõpuosas käsitletakse algoritmi intelligentsust suurima kliki leidmise alas ja esitatakse mõned ideed meta-algoritmi ehitamiseks. See väitekirja osa koondab kokku eelnevalt töös kirjeldatud testimiskeskkonna ideed, suurima kliki leidmise algoritmid, ekspertsüsteemid ja algoritmide testimise kogemused. Ka käsitletakse reaalaja süsteemide keskkonda – need on süsteemid, kus algoritm võib olla peatatud suvalisel ajahetkel, tagastades hetke parima tulemuse. Töös pakutatakse uue termini „lõpetamata lahendus” kasutusele võtmist. See võimaldab fikseerida hetke, millal tegelik suurim klikk on juba leitud, kuigi me ei ole jõudnud veel tõestada, et see on suurim. Nimetatud terminit kasutatakse töös tänapäeva parimate suurima kliki leidmise algoritmide analüüsimiseks reaalajasüsteemides, võimaldamaks määrata, millised nendest töötavad hästi ja milliseid nendest oleks parem mitte kasutada. See on oluline informatsioon meta-algoritmide jaoks. Töös esitatakse ka kaks meta-algoritmi liiki – fikseeritud reeglitega meta-algoritm ja keskkonnale adapteeruv meta-algoritm. Näidatakse, kuidas niisuguseid meta-algoritme ehitada ning diskuteeritakse erinevate õppimismetoodikate teemal.

**Võtmesõnad**: suurim klikk, graafi teooria, haru-ja-piir, lõpetamata lahendus, tippude värvimine, värvi klassid, algoritm, NP-keeruline, NP-raske, testimiskeskkond, meta-algoritm.

# Acknowledgements

*It's been a long way, but we're here*
Alan B. Shepard

Above all I would like to thank my supervisor Prof Rein Kuusik. He has always been able to find time for me although was constantly heavily occupied with work as a director of the department of informatics and with other students – teaching and supervising. He is a key person in my university study, starting from my second year in Tallinn University of Technology, when he was lecturing the “Monotonic systems” course up to now. Thank you, my Teacher.

The accomplishing of this thesis would have been impossible without the support I received from my wife Niina Kumlander. She has been so helpful and understanding during all this years. Working on my thesis I also had to do my everyday work. I have been missing you a lot spending many hours during weekends, vacations and evenings writing down my thesis. In addition to motivating and supporting me, she had to do a lot of things at our home instead of me and managed to survive without the man’s help. I greatly appreciate that and promise to compensate that.

I would also like to thank Prof Emeritus Leo Võhandu. I always felt his presence though he remained in a shadow in many cases. First of all I would like to thank him for his seminars that taught me both what to write (i.e. a subject) and how to present what has been written. He also motivated me a lot, especially during earlier years of my doctorate study. His scientific criticism is never devastating, but rather motivating, demanding, encouraging and influential.

I especially thank my colleague Veiko Laev and my university friends Roman Konovalov and Vladislav Loidap for their suggestions and corrections. You have always been patient with me and always offered your help when I needed it most. You have never complained and I appreciate that very much. I would also like to thank Mare-Anne Laane for her corrections in some of my papers. Her advices where both extremely valuable and helpful and I really regret that I wasn’t able to reach the level he wanted from me.

I am grateful to the Estonian Information Technology Foundation (EITSA) that supported my participation in practically all conferences I attended. This gave me a chance to present my work results and, what is much more important, to discuss them with a lot of researchers. Many parts of this thesis have been motivated by those conference conversations.

Last but not least – I thank my mother Jekaterina Barsukova. I have inherited her interest in math and logical thinking. She always motivated me to study hard and created the corresponding environment for me in my childhood. She accepts that I am not able to visit her for months and always loves me.

It would have been impossible to finish this thesis without a support from the company I am working for. Thank you, Simple Concepts and above all Lars Gunnar Svensson. You always provided me with day-offs, vacations when I needed those for conferences, you accepted all my university visits during some work hours and I am very thankful to you for doing that. I would also express my warmest thanks to all my colleagues who made all this possible. Besides I extend my thanks to Andy Cheetham and CODA Group Holdings Limited that acquired Simple Concepts. I already learned that you support my doctorate study and I appreciate that.

## Table of Contents

# 1 INTRODUCTION

## *1.1 Background of the Study*

The main area of this study is graph theory, which is a key technique of discrete optimisation, operations research, topology and a lot of others. A lot of scientists are using graph theory as a powerful tool to analyse problems within their own area of research. Although graph theory has already been used for many years it is still young and contains a lot of unsolved problems and there is still a lot of space for researches to introduce their own works and algorithms, for discussing and having fun. The number of applications is growing quite fast and surely it will grow further making people's life and activities much more "optimal" and easier.

A graph is a representation of relationships between some objects. Specifically a graph is built by two sets – a set of objects that is called **vertices** and a set of relations that is called **edges**. Consider, for example, a society that contains a set of persons. There could be relations between those persons of different type: friendship, parent-children and so forth. This society could be modelled as a graph where a vertex represents each person and if any two persons have any relation then it will be represented by an edge between vertices representing those persons. Another illustration of using graphs could be a problem of scheduling exams. Let's say we have a set of courses that will end up with an exam. Some students can take more than one course among those; therefore it is not possible to schedule some exams at the same time. We have to model somehow this situation before we will be able to apply any mathematical tool to solve it. The most natural way is to use a graph by letting the vertices to represent the courses and edges between any two courses if those two courses are "incompatible" to share the same time since have shared students. So, using graphs we can simplify a real world problem by abstracting it into a poor mathematical model, which will contain a main core/idea of it. Many difficult problems have been solved by converting them into graphs and using the graph theory. Today the graph theory is hosting a lot of applications in such fields like sociology, chemistry, geology, computer science and many others. With this consideration, it has become important to study core problems of the graph theory keeping in mind that it will help to advance the theory and the practice in a lot of others humans activities fields.

## *1.2 Definition of Basic Concepts*

Let $G=(V,E)$ be an undirected graph, where $V$ is the set of vertices and $E$ is the set of edges. Two vertices are called to be *adjacent* if they are connected by an edge. A *clique* is a complete subgraph of $G$, i.e. one whose vertices are pairwise adjacent. An *independent set* is a set of vertices that are pairwise nonadjacent. A *complement graph* is an undirected graph $\hat{G}=(V,\hat{E})$, where $\hat{E} = \{ (v_i , v_j) \mid v_i , v_j \in V, i \neq j, (v_i , v_j) \notin E \}$ – this is a slightly reformulated definition provided by Bomze et al 1999. A neighbourhood of a vertex $v_i$ is defined as a set of vertices, which are connected to this vertex, i.e. $N(v_i) = \{v_1, \ldots, v_k \mid \forall\, j\colon v_j \in V, i \neq j, (v_i , v_j) \in E\ \}$ A *maximal clique* is a clique that is not a proper subset of any other clique, in other words this clique doesn't belong to any other clique. The same can be stated about maximal independent set. All definitions listed so far are obtained from the following sources: Bomze et al 1999, Carraghan and Pardalos 1990a, Östergård 2002.

The *maximum clique problem* is a problem of finding maximum complete subgraph of $G$, i.e. maximum set of vertices from $G$ that are pairwise adjacent. In other words the maximum clique is the largest maximal clique. It is also said that the maximum clique is a maximal clique that has the maximal cardinality. The *maximum independent set problem* is a problem of finding the maximum set of vertices that are pairways nonadjacent. In other words, none of vertices belonging to this maximum set is connected to any other vertex of this set. A *graph-colouring problem* or a *colouring* of $G$ is defined to be an assignment of colours to the graph's vertices so that no pair of adjacent vertices shares identical colours. So, all vertices, which are coloured by the same colour, are nothing more than an independent set, although it is not always maximal. The *chromatic number* - $\chi(G)$ is the minimum number of colours needed for a colouring of $G$. The following references have been used to obtain previously listed definitions: Bomze et al 1999, Butenko et al. 2001, Carraghan and Pardalos 1990a, Klotz 2002, West 2001.

All those problems are computationally equivalent, in other words, each one of them can be transformed to any other. For example, any clique of a graph $G$ is an independent set for the graph's complement graph $\hat{G}$. So the problem of finding the maximum clique is equivalent to the problem of finding the maximum independent set for a complement graph.

For more information on other transformations see the "Complexity" subchapter below.

The same problems can be stated for weighted graphs, i.e. for an undirected graph $G=(V,E,W)$, where $V$ is the set of vertices and $E$ is the set of edges and $W$ is the set of weight of vertices from $V$ (each vertex have one weight). In this case, the maximum clique problem transforms into *the maximum weighted clique* (or *the maximum-weight clique*) problem asking to find a clique of the maximum weight [Bomze et al 1999], i.e. where sum of vertices' weights belonging to the clique is maximum for the graph.

It is also possible to formulate the same problems with weighted edges, but those problems are beyond of this work.

All those problems are NP-hard on general graphs [Garey and Johnson 2003] and no polynomial time algorithms are expected to be found. There are also so called "Heuristic algorithm" allowing finding a solution of a problem in a polynomial time without guarantee that the found solution is the maximum / best one. Those algorithms are widely used both as a possible way to solve problems at a reasonable time and as subtasks of exact algorithms, for example for finding boundaries.

## 1.3 Research Problem

The basic research problems addressed in this study are "Construct new effective algorithms for solving the problem of finding the maximum clique in an arbitrary undirected graph" and "Construct new effective algorithms for solving and the problem of finding the maximum weighted clique in an arbitrary undirected graph".

The following sub-problems have been formulated:

1. Identify properties of graphs that can be used to make quicker algorithms;
2. Develop better algorithms for the maximum clique and the maximum weighted clique finding;
3. Build a test environment allowing comparing algorithms;
4. Research those algorithms and identify why one or another is better and on what graphs;
5. Formulate a philosophy of building a meta-algorithm allowing increasing intelligence of applying maximum clique finding algorithms in different environments / for different graphs;

This study is based on the author's previous researches, which were included into his Master of Science work. That study reviewed some special graphs' cases like for example permutation graphs and contained algorithms based on finding triangles. Some ideas of those algorithms were converted into initial algorithms' that have shown quite promising results and evolved into algorithms presented in this study. The test environment idea was obtained in that study as a way for further development that will be very helpful, while a meta-algorithms idea came later – rather during formulating this study on later stages and wasn't included into earlier draft of tasks list. So, an initial task of developing better algorithms was reformulated by adding a test environment research and ended in a wide formulation on the high level of meta-algorithms, while maximum clique algorithms remained to be a core of the study.

## 1.4 Complexity

There are a lot of problems that are not so easy to solve as it looks like at first. Therefore it is important to know a complexity of a problem you are trying to solve before starting to work out an algorithm for solving this problem. More exactly, it is important to know if this problem is NP-complete (Nondeterministically Polynomial), i.e. very hard to solve.

We say that a function $f(n)$ has complexity $O(g(n))$, where $n$ is the size of the input parameter or its length, whenever there exists such constant $c$, that $|f(n)| \leq c\,|g(n)|$ for any $n \geq 0$. A *polynomial time algorithm* is an algorithm whose time complexity function is $O(p(n))$, where $p(n)$ is a polynomial function. Any algorithm whose time complexity function cannot be so bounded is an *exponential time algorithm*. An example of such time complexity function could be $a^n$ where $a$ is a constant. Sometimes such algorithms are called non-polynomial since this definition also includes algorithms having $n^{\log(n)}$ time complexity although such complexity is neither polynomial nor exponential. Of course, exponential algorithms can be even faster than polynomial ones in certain cases. For example consider algorithms having complexity $2^n$ and $n^5$, for $1 < n < 23$. Unfortunately real live problems are having much larger $n$ than in the previous example. That's why exponential algorithms are not regarded as being useful in practise although are known for many problems.

It is common to distinguish between two types of complexity:

1. Time complexity as we have seen so far, i.e. a problem is called hard to solve or intractable if there are only exponential time algorithms to discover a solution.
2. A problem's solution size complexity. This happens when the solution itself cannot be described with an expression having length bounded by a polynomial function of the input length [Garey and Johnson 2003]. A problem of finding all cliques from a graph can be an example since for a general case number of existing cliques is exponential.

The maximum clique problem's complexity is the time complexity.

It is not proved right now that the problem of finding the maximum clique cannot be solved in a polynomial time as well as vice versa. It means that nobody was able to construct even theoretically such algorithm for any graph and nobody was able to prove impossibility to have such algorithm. We are going to demonstrate the algorithm's complexity in terms of NP-completeness in the following subpart.

## NP-complexity

The first most important researches in the algorithms complexity area were done by Turing in the 1940s. Turing has demonstrated that some problems are "undecidable", i.e. those problems can be solved algorithmically. Moreover his works have greatly affected complexity theory for "decidable" problems since his abstract computer model, so called Turing machine, was used for researches and definitions in this area. NP-class problems are defined as problems that can be solved on a *none-deterministic* Turing machine in a *polynomial* time [Garey and Johnson 2003]. There is a P class as well, which contains problems that can be solved in a polynomial time on the deterministic Turing machine, which is also called just the Turing machine. Sometimes NP-class is defined as a class of problems that cannot be solved on the Turing machine, although it is not quite correct, since $P \subseteq NP$. So NP-P problems cannot be solved in a polynomial time on the deterministic Turing machine. Besides, it will be wrong to say that NP means none-polynomial, although it describes quite well an essence of those problems from nowadays programming point of view. NP-completeness theory foundations were laid in a Cook paper presented in 1971 [Cook 1971]. First of all, he highlighted importance of "polynomial time reducibility". It means that if we can show that there exists a polynomial time transformation from one problem into another, then any polynomial time algorithm for the second algorithm will provide us with a polynomial algorithm for the first problem. Besides, basing on those ideas, he has shown that any NP problem can be converted into the satisfiability problem in a polynomial time. He also has demonstrated that there are some other problems, which have the same complexity as the satisfiability problem. Those problems are "hardest" problems or are an essence of NP-class. Later a lot of problems have been shown to be as "hard" as the satisfiability problem [Karp 1972] and those problems were called NP-complete problems.

The formal definition for NP-complete is the following: a problem is *NP-complete* if the problem belongs to NP-class and *any* other problem of NP-class can be polynomially transformed into this problem.

NP-completeness proof for a problem *S* contains 4 steps:

1. Show that *S* is in NP;
2. Select a known NP-complete problem *S*' to which *S* is most similar (or select any);
3. Construct a function *f* that transforms *S*' into *S*;
4. Proof that the transformation function *f* is a polynomial one.

So, it is important to consider a connection between different types of tasks before starting to solve any of them since it could provide important and interesting information on how it can be done. Although there are a lot of NP-complete problems, just some of them are commonly use as "core" problems since those fit better for transforming. Those core problems are listed below.

**Satisfiability (SAT)**

*Condition:* Given a collection of clauses $C = \{c_1, c_2,\ldots,c_m\}$ over a finite set of variables $V$.

*Question:* Is it possible to find such set of values for those variables that each clause is satisfied.

**3-Satisfiability (3-SAT)**

*Condition:* Given a collection of clauses $C = \{c_1, c_2,\ldots,c_m\}$ where each clause contains exactly 3 literals ( $|c_i|=3,\ 1 \le i \le m$ ), over a finite set of variables $V$.

*Question:* Is it possible to find such set of values for those variables that each clause is satisfied.

**3-Dimensional Matching**

*Condition:* Given a set $M \subseteq W \times X \times Y$, where $W$, $X$, $Y$ are none intersecting (disjoint) sets containing each $q$ elements.

*Question:* Is it true, that $M$ contains 3-dimensional matching, in other words is there a subset $M' \subseteq M$ that have the following properties: $|\, M' \,| = q$ and there are no different elements inside $M'$-which have equal coordinates $(w, x, y)$?

**Vertex cover**

*Condition:* Given a graph $G = (V, E)$ and a number $K$: $0 \le K \le |\, V \,|$.

*Question:* Is it possible to find on this graph $G$ a vertex covering, which contains at most $K$ elements, or is their a subset $V' \subseteq V$, such that $|\, V' \,| \le K$ and $\{\forall(x, y) \in E;\ x \text{ or } y \in V'\}$?

**Hamilton cycle**

*Condition:* Given a graph $G = (V, E)$.

*Question:* Is it true, that $G$ contains a Hamilton cycle, or does there exist a vertices series $<v_1, v_2,\ldots, v_n>$, such that $n = |\, V \,|$, $\{v_i,v_{i+1}\} \in E : \forall i,\ 1 \le i \le n$ ?

**Clique**

*Condition:* Given a graph $G = (V, E)$ and a number $K$: $0 \le K \le |V|$

*Question:* Is it possible to find in the graph a clique containing $K$-vertices, in other words does this graph contain a subset $V' \subseteq V$, where $|V'| \ge K$ and $\{\forall\, x, y \in V';\ (x, y) \in E\}$?

**Partitioning**

*Condition:* Given a finite set $A$ and weights $s(a) \in Z^+$ for each $a \in A$.

*Question:* Is it possible to find such subset $A' \subseteq A$ that satisfies to the next condition

$$\sum_{a\in A'} s(a) = \sum_{a\in A\setminus A'} s(a)?$$

The next tree is used for NP-completeness proving:

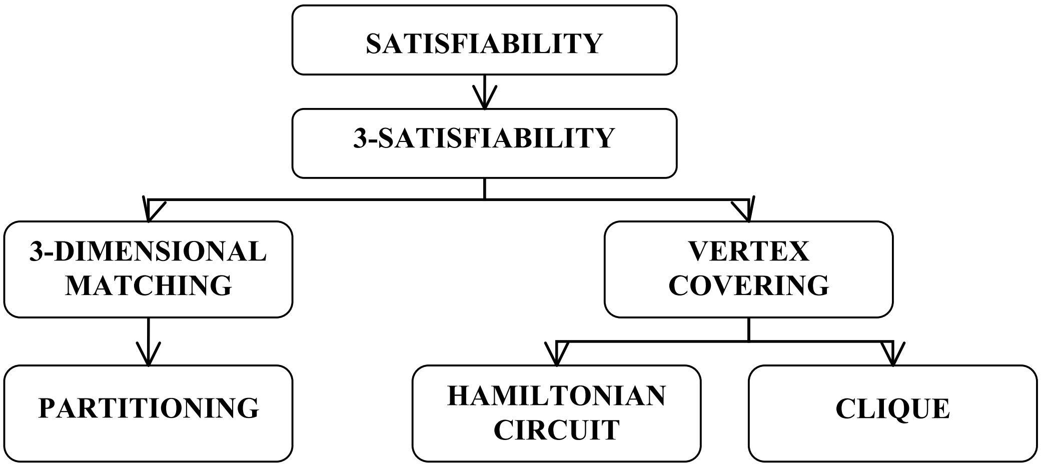

**Figure 1.** The sequence of transformation of the six basic problems' NP-completeness proving

The maximum clique problem is polynomially equivalent to the clique problem; therefore some authors refer this problem as NP-complete [Karp 1972, Bomze et al 1999]. The classical Garey and Johnson book names the maximum clique size problem to be NP-easy [Garey and Johnson 2003]. A search problem is defined to be *NP-easy* whenever there exists another problem belonging to NP to which the original problem is Turing reducible. So, the NP-easy problem is a problem, which is "no harder" than the NP complete problem for solving. A search problem is defined to be *NP-hard* if there exists some NP-complete problem that Turing reduces to this problem. So, all NP-complete problems are NP-hard. Unfortunately this terminology is a bit unstable [Garey and Johnson 2003], but the maximum clique problem is at least NP-complete, and is NP-hard by this definition since the "Clique", which is the NP-complete problem, is Turing reducible to it at vice versa, so it is neither easier nor harder than the clique problem. That's why we refer this problem in this study to be NP-hard as many other authors as well [Wood 1997, Östergård 2002].

All this mean that finding of any better solution / algorithm for the maximum finding algorithm will result in more than just in improving of a performance of solving this problem, but could also mean finding better algorithms for other NP problems. That indicates great importance of our problem for researching.

We conclude this subsection with some smaller issues regarding the maximum clique problem complexity. As we mentioned earlier the maximum clique problem is a time-complexity problem and therefore depends on a length of an input parameter; besides there are no polynomial algorithm to solve it. It is important to note here that any algorithm depend also on a programming language that implements the algorithm and on a computer on which the algorithm is run, but those two parameters are usually omitted. First of all those parameters can produce just some small polynomial differences from an algorithm point of view. Besides the computer science usually tries to depart from a particular computer and provides a function showing number of steps / processor tacts to solve the problem. The only important is to get this into account conducting comparative tests of different algorithms. We have to use the same computer, the same coding technique and the same language to avoid those parameters influence of on results. Moreover we will use algorithms' spent time ratios to be computer independent in results and make them reproducible on any platform.

Another important thing to note is a fact that most of classical or well-known algorithms work also fast on certain or a sufficient number of graph types. Those algorithms are having problems only because generally we say that we need an algorithm that has to solve all types of graphs and there are certain types of graphs, which are hard to solve by this algorithm. But those algorithms still provide us with a hope that we have captured some important graphs' properties and we can extract those for using in new algorithms that will be better than already existing.

*Note: See the chapter "Scope and Background of the Research" below for some examples of graph types that can be solved in a polynomial time.*

## 1.5 Scope and Background of the Research

During this work we will mainly concentrate on general graphs which are normally don't have a well-known structure basing on which we could simplify our search. Of course there are certain types of graphs where the maximum clique can be found in a polynomial time, see for example permutation graphs [Kim 1990], more general class of graphs – perfect graphs [Berge and Chv'atal 1984, Bomze et al. 1999] that includes interval graphs, bipartite graphs [Bomze et al. 1999], but our aim is to invite better algorithms for a general case. Besides we will try to invite a meta-algorithm that can work both with general case and with special graphs using corresponding algorithms, so we'll move to the higher, meta level in solving the maximum clique finding.

The best known algorithm for the maximum independent set finding is developed by Robson [Robson 1986] and has a time complexity upper bound $O$ ($2^{0.276n}$ ), but unfortunately no experimental results/tests of this is known [Pardalos et al. 1998]. Note again that the maximum clique finding is exactly the same complexity task as the maximum independent set finding. The most efficient algorithm for solving this problem was developed by Carraghan and Pardalos [Carraghan and Pardalos 1990a]. Moreover, it is an algorithm that is usually used for benchmarking new algorithms as it is proposed by DIMACS [DIMACS 1999, Johnson and Trick 1996]. Another set of algorithms that will be widely used in the study are algorithms proposed by Östergård and claimed to be the best at the moment – for unweighted [Östergård 2002] and weighted [Östergård 2001] graphs, so it should be enough to compare new algorithms with those to investigate new algorithms efficiency.

## 1.6 Applications

Here we investigate different applications of the maximum clique finding and show why inventing a better algorithm for the problem is so important.

The maximum clique problem has many theoretical and practical applications. The most important theoretical application lays in the connection between NP-complete problems, as it was shown in the previous chapter. Consider for example a connection between the maximum clique problem and a quadratic programming problem established by Motzkin and Straus [Motzkin and Straus 1965].

In fact, a lot of algorithms contain this problem as a subtask and this is another important applications area for the problem. We will list some such problems below.

The first area of applications is data analyses / finding a similar data. For example, the next construction is used: a graph is constructed with vertices corresponding to data elements and similar data elements (vertices) are connected by edges. Another construction starts from building a bipartite graph, where the first set of vertices are data elements and the second set of vertices are properties. Data elements having a particular property are connected to a vertex corresponding to this property by an edge. Now, it is possible to connect all data elements and all properties by edges and search for a clique having both properties vertices and data element vertices. In both cases the maximum clique is a cluster. Those constructions are widely used and below we are going list just some such areas: the identification and classification of new diseases based on symptom correlation [Bonner 1964], computer vision [Ballard and Brown 1982], and biochemistry [Miller 1992].

Another wide area of applying the maximum clique is the coding theory [Sloane 1989, Brouwer et al. 1990]. The simplest example of such situation occurrence is transmitting a set of bytes, during which one byte is lost, but neither sender nor receiver know which one – this corresponds to the "Single deletion correction code" [Sloane 2001]. The general task here is to find the largest binary code consisting of binary words that can correct a certain number of errors.

Another important application arises in the circuit design. A task is to build an optimal layout of elements on a circuit. In addition to that maximum cliques are widely used in the circuit testing for fault diagnosis [Brglez and Fujiwara, 1985]. For example, one such application is to place special elements on a circuit (or a board) part to detect if it is working correctly. Those elements can check only a restricted set of circuit components due the testing element size. The task here is to maximize a number of circuit components tested in one pass. Here components are vertices and edges connect components that can be checked in one pass. Clique detection can be used also for the distributed fault diagnosis in multiprocessor systems [Berman and Pelc 1990]. The task is to identify a faulty processor. It is assumed that a fault-free processor in the system detects the faulty processor with some probability, while no assumptions are made on the performance of faulty processors. A major step in the algorithm is to find the maximum clique in an appropriate graph (a c-fat ring) [Pardalos et al. 1998]. Those graphs are to be included into tests for algorithms we are going to invite.

The maximum-weight clique problem has also a lot of applications. For example in the coding theory [MacWilliams and Sloane 1979], geometric tiling [Corradi and Szabo 1990], fault diagnosis [Berman and Pelc 1990], pattern recognition [Horaud and Skordas 1989], molecular biology [Mitchell et al. 1989], and scheduling [Jansen et al. 1997]. Additional applications arise in more comprehensive problems that involve graph problems with side constraints. More this problem is surveyed in [Bomze et al. 1999].

There are much more application and we have listed only some of them. All it proves that the importance of the maximum clique finding for the today science is very high. The reason is highly abstracted model of the graph concept allowing applying it practically everywhere.

## 1.7 Outline of the Study

In the **Chapter 1** of this Study a background of it is reviewed, basic concepts are presented and tasks to solve formulated. The complexity of the problem is also analysed and scopes and limitations of the study are identified.

The review of earlier researches is done in the **Chapter 2**. This allows understanding of where the problem research is right now, what has been done, what methods gave negative results and what methods are best at the moment. Those best algorithms will be used either as a part of new algorithms or for comparing new algorithms to identify whether those are better and where.

Main parts of this study are opened by the **Chapter 3**, where new algorithms for finding the maximum clique are developed. Here both weighted and unweighted graphs' cases are researched. Each new algorithm is explained, published in a formal way and at least one example is played through. The second part of this chapter contains a description of tests and results including a description of graphs used for the testing. Results are presented as tables and graphs.

The second main part of the study is located in the **Chapter 4** and discusses a testing environment that was used to produce testing results of the previous chapter. The testing environment is an essential part of any researches and its architecture, philosophy of constructing are presented.

The **Chapter 5** introduces a new level in the maximum clique finding programs' implementation – an idea of a meta-algorithm containing the maximum clique finding algorithms. Here ideas from other data analyses areas are collected and applied to the maximum clique problem. Besides, the real-time systems' case is discussed. Some tests showing efficiency of algorithm in finding the maximum clique without proving that it is the maximum one are done and results are discussed.

Lastly **Chapter 6** concludes the study with a summary.

Programs' codes are provided in an appendix to show how algorithms were implemented. This ensures correctness of algorithms comparison tests by making them reproducible as well as unambiguous understanding of the implementation of new algorithms.

# 2 REVIEW OF THE STATE OF THE ART

## 2.1 *Maximum Unweighted Clique / Exact Approaches*

This chapter contains two algorithms designed to find the maximum clique from an unweighted graph. Those algorithms produce an exact solution, i.e. a clique produced by any algorithm is the maximum one for the graph – there is surely no clique that could contain more vertices than the produced one.

Two algorithms, which are presented below, are most important from the author's point of view at the moment, although there are some other algorithms as well. The first algorithm is a classical one introduced in the early years of applied combinatorics in computer science by Carraghan and Pardalos [Carraghan and Pardalos 1990a]. It is easy to implement and it works very effectively in practise as well although it's having certain problems on some graphs - for example on graphs with high density. The second algorithm, which is developed by Östergård [Östergård 2002], is claimed to be the best / fastest one at the moment. Another plus of this algorithm is that it is very similar to the first one although contains some important modifications making it much quicker. It happens because both algorithms use a branch and bound technique in finding the maximum clique. The same ideas will be used in our new algorithms. Therefore those two algorithms can be easily compared with new algorithms since we can use the same programming technique for all of them. This eliminates a risk of inadequate testing that occurs because of totally different algorithms that are implemented differently. Here it is easy to do mistakes in programming that leads to the inadequate testing. So, the only way to eliminate such risk is to use the same source code with some modifications for testing algorithms if it is possible. That could be perfectly done for the presented algorithms having very similar algorithmic structures.

### 2.1.1 Carraghan and Pardalos algorithm

This algorithm was introduced in 1990 by Carraghan and Pardalos [Carraghan and Pardalos 1990a]. The algorithm is very simple and efficient for finding the maximum clique on an arbitrary graph. It issued to be fastest for any types and densities of graphs until the end of the previous century. Currently it holds a title of the fastest algorithm for the low-density graphs.

The algorithm by its nature is a branch and bound algorithm. Crucial to the understanding of this algorithm is the notation of the *depth*. The algorithm forms depths by selecting an expanding vertex from the current depth and selecting into this new depth all vertices from the current, which are connected to the expanding vertex. Vertices at each step are expanded one by one while there are vertices that were not yet expanded on the depth and removed from the analysis on the depth.

The main advantage of this algorithm is its pruning formula for a depth (branch)

**if** $\boldsymbol{d + ( m - i ) \leq CBC}$ then prune (go to the higher/previous depth), where $d$ is a depth number (initial depth number equals to 1), $m$ is the total number of vertices on the current depth, $i$ is a sequential number of the expanding vertex on the current depth and *CBC* is a size of the current best (maximum) clique. If the pruning formula works/holds on the first/initial depth then the algorithm stops.

```
function Main
    CBC := 0          // the maximum clique's size
    clique (V, 0)
    return
end function

function clique(V, depth)
    if'|V| = 0 then
        if depth > CBC then
            New record - save it. CBC := depth
        end if
        return
    end if
    i := 0
    while i < |V| do
        if depth + |V| - i ≤ CBC then return        // prune
        i := i + 1
        // form a new depth. N(v_i) denotes a neighbourhood of v_i.
        clique (N(v_i) | ∀v_j : j > i, j ≤ |V|, depth + 1)
    end while
    return
end function
```

Authors have advised using the next order of vertices: starting from a vertex with the smallest degree up to a vertex with the highest degree. Such degree ordering can be done either only once before running the main part of the algorithm or can be reapplied on each depth except dense graph where it will not provide a lot improvement and is time consuming. The classical algorithm orders vertices only once.

This quite a simple algorithm has shown in practise its efficiency – it is the best-known algorithm for sparse graphs and very efficient for others as well. Mainly because it doesn't spend the valuable time on different checks, which are usually irrelevant, and starts to work immediately. The only area where the algorithm is quite slow is dense graphs. Those graphs are hard to solve by this algorithm since its pruning formula doesn't work on dense graphs as the number of remaining vertices on a subgraph usually is much bigger than *CBC* up to the last vertices.

### 2.1.2 Östergård algorithm

This algorithm is published by Östergård in 2002 and is based on the previous one with an important addition introduced: a backtrack search [Östergård 2002].

The algorithm considers subgraphs $G_i$ of $G$, where $i$ indicates the minimum sequential number of a vertex included into a subgraph: $V_i = \{v_i,\ldots,v_n\}$. Those subgraphs are searched for maximum cliques starting from the $i = n$-th upto the $i = 1$, i.e. in the backward order. The algorithm is also a branch and bound algorithm by its nature. It forms depths by selecting an expanding vertex form the current depth and putting to this new depth all vertices from the current, which are connected to the expanding vertex. Vertices at each step are expanded one by one while there are vertices that were not yet expanded on this depth. The pruning formula described for the previous algorithm is also used.

The maximum clique size for each subgraph is saved in a cache for the later use for the maximum clique search on a subgraph with a smaller index $i$ by the following pruning formula:

**if $d + c[i] \leq CBC$** then prune, where $d$ is a level starting form 0, $i$ is the minimal sequential number of a vertex among vertices existing on that level, $CBC$ is a size of the current best (maximum) clique and $c[i]$ is the maximum clique size of $G_i$.

This $c$ function or cache is the core idea of that algorithm. There is also defined a condition to stop searching the maximum clique basing on the $CBC$.

**if $d + c[i] > CBC$ then stop** since $\forall\ i$: $G_{i-1} \subseteq G_i$ and therefore $CBC$ is the maximum clique on $G_{i-1}$ as well as the maximum clique for $G_{i-1}$ either equal to the maximum clique of $G_i$ or is bigger on 1. So, as soon as a bigger clique is found we could stop since the graph cannot contain any bigger clique.

```
function Main
    max :=0
    for i := n downto 1 do
        found := false
        clique ($S_i$ & $N(v_i)$, 1)
         c [ i ] :=max
    end for
    return
end function

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 $U \neq \varnothing$ do
        // prune as Carraghan and Pardalos algorithm does
        if size + |U| ≤ max then  return
        i :=min { $j \mid v_j \in U$ }
        // new pruning technique
        if size + c [ i ] ≤ max then return
```

```
            $U$ :=$U$ \ { $v_i$}
            $clique(U \& N(v_i), size + 1)$

            if $found$ = $true$ then return  // stopping condition

        end while
        return
    end function
```

Author advices to reorder vertices using the following heuristic algorithm: first find the vertex-colouring and then group vertices by colour classes. This heuristic algorithm is a greedy one. He also marked that other heuristic algorithms can be used to produce a “good” initial ordering if a better algorithm will be found. Experimental results show that this algorithm is generally the quickest algorithm for finding the maximum clique and also has to be granted to be one of the easiest to understand and implement. There are certain DIMACS graphs instances, where the algorithm is slow, like c-fat500-5, MANN_a27 and some others, but mostly it is one of the fastest on those graph instances as well.

### 2.1.3 Some other heuristic vertex-colouring based algorithms

Here we review some historical attempts to use a heuristic vertex-colouring for maximum clique finding. The heuristic vertex-colouring idea is interesting for us since algorithms invented in this study are based on such colouring and therefore it could be very interesting to see where this idea was used so far. Algorithms that we are going to describe are not very fast and therefore we will not include those into our comparative tests and those will be interesting only from a historical point of view. Please note that we do not mean here algorithms that only use a vertex-colouring to derive once bounds for the maximum clique, but rather review algorithms containing vertex-colouring as an important part of an algorithm for the maximum clique finding, i.e. where those bounds are permanently found.

#### 2.1.3.1 Babel and Tinhofer’s algorithm

This is an algorithm proposed by Babel and Tinhofer in 1990 [Babel and Tinhofer 1990], i.e. in the same year when Carraghan and Pardalos has released their brilliant algorithm described before [Carraghan and Pardalos 1990a]. This algorithm is also the branch and bound one and was positioned by authors as an algorithm, which is especially efficient for graphs with great edge density. The branching is the first phase of the algorithm, which is done exactly in the same way as the Carraghan and Pardalos one does [Carraghan and Pardalos 1990a]. The second phase, which is bounding uses mainly a well known fact that the chromatic number of a graph is always bigger or equal to the size of this graph maximum clique. The problem of vertex-colouring is NP-complete therefore the algorithm uses DSATUR technique to find a heuristic vertex-colouring. This heuristic algorithm is to be described later in the “Vertex colouring / Heuristic approaches” subchapter. The following sentence describes an essence of Babel and Tinhofer algorithm: “In each $G_i$ we look for a clique and a colouring”, where $G_i$ is a subgraph that is produced during the branching algorithm. The algorithm can be presented using the following pseudo-code:

```
function Main
   CBC :=0            // the maximum clique’s size
   clique (V, 0)
   return
end function

function clique(V, depth)
   if’|V| = 0 then
      if depth > CBC then
         New record - save it. CBC := depth
      end if
      return
   end if

   while V ≠ Ø do
       if depth + |V| ≤ CBC then return       // prune
       //use DSATUR to find the max. clique and a colouring
       Q, C :=DSATUR(G(V)) // C = {C_1,..,C_k}, Q – max. clique
       if depth + |Q| > CBC then
          New maximum clique: Q + vertices of previous depths.
          CBC = depth + |Q|
       end if
       if depth + |C| ≤ CBC then return
       clique (N(v_1), depth + 1)
       V := V \ v_1
   end while
   return
end function
```

Please note that the DSATUR algorithm provides both a heuristic colouring and a heuristic maximum clique. The heuristic colouring is the main output and target of this algorithm, while the heuristic maximum clique is obtained first associations of colours to vertices, since saturation degree rule leads to colouring a clique first of all. The heuristic clique is formed by a set of vertices, which are coloured until any colour is reused – see the “DSATUR” subchapter below to find how this algorithm works. .

We should mention that different modifications of this base algorithm were proposed by Babel and Tinhofer in their work to decrease the time for computing the upper and lower bounds, but those modifications do not change the base algorithm dramatically and therefore can be omitted for this review. The algorithm was successfully used for some types of graphs, including chordal graphs [Ballas and

Tinhofer 1990], but the general performance of the algorithm was quite bad in comparison to the Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990a], which was released simultaneously, mostly due to the fact that the algorithm consumes too much time to find bounds in combinatorial cycles.

#### 2.1.3.2 Wood's algorithm

An algorithm invented by Wood in 1997 [Wood 1997] is another attempt to employ a heuristic vertex colouring for the maximum clique finding basing on previous works – mostly on Babel and Tinhofer algorithm [Babel and Tinhofer 1990] and Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990a] and some their later works. The algorithm can be described using the following pseudo-code with a certain simplification of unimportant details, which will not change the main idea:

```
function Main
   CBC :=0            // the maximum clique's size
   clique (V, 0)
   return
end function

function clique(V, depth)

   //Sub step 1
   Q :=greedy(V)   // find a clique using a greedy algorithm
   if depth + |Q| > CBC then
       New maximum clique: Q + vertices of previous depths.
       CBC = depth + |Q|
   end if

   // substep 2
   //Find a vertex colouring of G(V) by DSATUR
   C :=DSATUR(G(V)) // C = {$C_1$,..,$C_k$}
   if depth + |C| $\leq$ CBC then return

   while depth + |C| > CBC do   // pruning condition
       k := |C|
       v :=maxdegree(v $\in$ $C_k$)
       $C_k$ := $C_k$ \ v
       if $C_k$ = Ø then C := C \ $C_k$
       clique ( N($v_i$), depth + 1)
   end while
   return
end function
```

```
function greedy(V)
    S := V
    Q := Ø
    while S ≠ Ø do
        $v_i$ = maxdegree($v \in S$)
        $Q := Q \cup \{v_i\}$
        $S := S$ & $\{v_i\}$
    end while
end function
```

It is easy to see that the described algorithm is another branch and bound algorithm. The author mainly concentrates on three issues that he thought are very important for branch and bound algorithms:

- How to find a good lower bound, i.e. a clique of large size?
- How to find a good upper bound on the size of the maximum clique?
- How to branch, i.e. break a problem into subproblems? [Wood 1997]

The greedy algorithm for the heuristic maximum clique finding is used to find the lower bound, the heuristic DSATUR is used to find the upper bound and all this is done practically for each new branch, i.e. on each new depth. The original paper also employs a fractional colouring in addition to DSATUR to find a better upper bound, since the upper bound for pruning is set to be a minimum number of colours provide by those heuristic vertex-colouring algorithms – see this Wood paper for more details on fractional colouring and how it is used [Wood 1997]. The biggest differences from the previously described Babel and Tinhofer algorithm [Babel and Tinhofer 1990] are:

- The algorithm looks for a heuristic clique and colouring only for a new branch, i.e. it is not done in the internal cycle of a branch. Please note that Babel and Tinhofer tried in their efficiency improving modifications for the original algorithm to speed up the colouring of each branch by reordering vertices of a branch [Babel and Tinhofer 1990], while Wood decided not to colour in the internal cycle;
- More than one heuristic vertex colouring algorithm is used to provide a better bound.

Although the algorithm meets the defined issues, it seems to be rather impractical since it still spends too much time on finding bounds and this affects its performance characteristics. Therefore this algorithm has mostly been seen as another unsuccessful attempt to use vertex-colouring for the maximum clique finding, which rather proved that vertex-colouring cannot be used for that.

### 2.1.4 Other approaches

Here we are going to review some other approaches to the maximum clique finding. So far we mainly concentrated on the so-called "integer programming" enumerative algorithms although some other ideas exist. We are going to review some of them to provide a full picture on the study subject. Core ideas and pros and cons will be briefly described. We should mark in advance that those other algorithms are evolving and probably will be fastest in the future, but so far those are not a real alternative to the algorithms reviewed above.

It is possible to see the original maximum clique problem, which has been described in the “Basic concepts” subchapter, from different points of view or formulate it differently. Therefore before reviewing any other algorithms we are going to define, what is the “integer programming” formulation, to identify the technique we concentrate on in this work. Besides it will help us to demonstrate differences with other approaches and techniques. The simplest “integer programming” formulation is the following edge formulation:

$$\max \sum_{i=1}^{n} x_i ,$$

$$\text{subject to } x_i + x_j \leq 1,\ \forall\ (i,j) \in \hat{E},$$

$$x_i \in \{0, 1\},\ i = 1,..., n.\ \text{[Bomze et al. 1999]}$$

The *branch and bound* algorithms belong to the enumerative “integer programming” formulation where the branching is the enumerative part.

There is another historically quite popular technique for solving the problem - “integer programming” approach by relaxation and decomposition. The classical approach here is a Lagrangian relaxation [Guignard and Kim 1987]. Generally saying the relaxation means that the original problem is decomposed (relaxed) in one or another way into easier to solve problems, while the problem’s solution remains feasible. This is done repetitively until an optimal level of relaxing is reached and the problem can be solved directly. Quite often the relaxation technique is used basing on another, so called “continuous” formulation, of the maximum clique problem. This formulation is derived from the Motzkin and Straus [Motzkin and Straus 1965] work. We will not provide detail information on this problem formulation and would like to refer to the review of the maximum clique finding, which were published in 1999 [Bomze et al. 1999]. Unfortunately such “integer programming” relaxation generally needs a deep knowledge about a graph structure and therefore is used mostly as bounds for other algorithms or for special cases. That’s why we cannot recommend using this approach as a practical one for a general solution, although it is used very successfully to derive heuristic solutions of high quality [Pelillo 1995]. Besides, the relaxation is used much more for solving other NP-hard problems since it fits better there. Note that although *branch and bound* can be seen also as a decomposition some authors still name it enumerative “integer programming”.

Another popular technique comes from the genetic algorithms field. Genetic algorithms are inspired by the evolutional mechanism of nature and can be used for a parallel search. Long strings of bits are called chromosomes in the genetic algorithm terminology. There is a function that can compute a probability of survival of each “individual”, which is a potential maximum clique case. Algorithms work using three major operations> reproduction, crossover and mutation trying to increase probability of survival. The mutation randomly change each individual bit, while crossover means that basing on two cases a new case is produced by swapping two or more bits’ substrings. Please see [Goldberg 1989] for more information. Those attempts are also quite mathematical so far and those algorithms’ performance doesn’t provide currently hope on competing with other exact solutions. Therefore genetic algorithms are mostly used as heuristic [Hifi 1997, Marchiori 1998, Murthy et al. 1994] algorithms at present.

## 2.2 Maximum-Weighted Clique / Exact Approaches

The following algorithms are targeted to solve the same maximum clique problem, but on the weighted graphs. "Weighted" means that each vertex of a graph has a weight. There is also another type of weighted graphs – edge weighted graphs, but this type is not an issue of our study. Here we again describe only exact solutions, i.e. finding the clique that has maximum weight among cliques of the graph.

The maximum-weighted clique problem has less different algorithms than the unweighted case and presented algorithms are mostly just modifications of the unweighted case algorithms. It is so because the unweighted case is much easier to think about and to produce different ideas and algorithms, therefore the unweighted case is a case researches are concentrating on. It is important to mention that the importance of this problem is much bigger than for the maximum clique problem although the number of algorithms is smaller, since the unweighted case can be treated as a special case of the weighted one – all weights are equal and can be omitted.

This part contains the same algorithms as the previous one – "Maximum clique / Exact approaches", which were modified for graphs with vertices of different weights. Other algorithms described in the previous subchapter can also be converted into maximum-weighted clique algorithms more or less successfully but still do not reach efficiency of algorithms to be described in this subchapter.

### 2.2.1 Carraghan and Pardalos algorithm

The only modification we need to apply here for finding the maximum-weight clique is using weights / sums of weights instead of counts of vertices. The pruning formula is modified in the following way: **if $w(d) + w(m,i) \le CBC$** then prune (go to the previous depth), where $w(d)$ is an accumulated weight on previous (to $d$-th) levels, $m$ is the total number of vertices on the current depth, $i$ is a sequential number of the expanding vertex on the current depth, $w(m, i)$ is a function returning a weights sum of current depth vertices from $i$ to $m$, and $CBC$ is a weight of the current best (maximum) clique. If the pruning formula works/holds on the first/initial depth then the algorithm stops.

```
function Main
   CBC := 0                    // the maximum-weight clique's weight
   clique (V, 0)
   return
end function

function clique(V, w_depth)
   if'|V| = 0 then
      if w_depth > CBC then
         New record - save it. CBC = w_depth
      end if
      return
    end if

    $i$ := 0
    while $i$ < |$V$| do

        $wt$=($\sum weight$ [$v_j$] | $i < j <$ |$V$|)
        if $w\_depth$ + $wt \le CBC$ then return  // prune

        $i$ := $i$ + 1
        $clique$( (N($v_i$) | $\forall v_j : j > i, j \le$ |$V$| ), $w\_depth$ + $weight$ [ $i$ ] )

    end while
    return
end function
```

Described modifications produce a new algorithm for the weighted case, which is still easy to implement. Unfortunately it is not as dramatically fast as it used to be before modifications on the unweighted case, but it is still very good. [Carraghan and Pardalos 1990b]

### 2.2.2 Östergård algorithm

There are two modifications needed in addition to the general modification of the maximum clique definition, which is now weighted. First of all instead of the level *d* we have to keep in memory an accumulated weight for a branch constructed in reaching a current level. Besides the algorithm cannot use the stopping condition for finding the maximum-weight clique on a subgraph $G_i$ since the difference for stopping the search should be exactly equal to a weight of *i*-th vertex (instead of any as it was used before). The original paper and our study is going to use the algorithm that misses such stopping rule.

So, the backward's pruning formula now is formulated as **if *w(d)* + *c*[*i*] ≤ *CBC*** then prune, where *w*(*d*) is a weight accumulated on previous to *d* levels, *i* is the minimal sequential number of a vertex among vertices existing on that level, *CBC* is a weight of the current best (maximum)-weight clique and *c*[*i*] is the maximum-weight clique's weight of $G_i$.

```
function $Main$
    $max$ := 0
    for $i$ := $n$ downto 1 do
        $wclique$ ($S_i$ & $N(v_i)$, $weight$ [ $i$ ] )
        $C$[ $i$ ] := $max$
    end for
    return
end function
```

```
function wclique (U, w_depth)
   if | U | = 0 then
      if weight > max then
            max := weight
            New record;save it.
      end if
      return
   end if

   while $U \neq Ø$ do

      $wt=(\sum weight[v_j] \mid \forall j : v_j \in U)$
      if $w\_depth + wt(U) \leq max$ then return

      $i := \min\{j \mid v_j \in U\}$
      if $weight + C[i] \leq max$ then return

      $U := U \setminus \{v_i\}$
      Wclique ($U$ & $N(v_i)$, $w\_depth + weight[i]$ )

   end while
   return
end function
```

The initial ordering is also advised to be formed basing on a vertex colouring as for the unweighted case, i.e. using a heuristic greedy algorithm: first find the vertex-colouring and then group vertices by colour classes.

Practical results on randomly generated graphs show that this algorithm is the quickest one and the difference with others algorithms were sufficient. [Östergård 2001]

## *2.3 Vertex-Colouring / Heuristic Approaches*

### 2.3.1 A short review

A variety of algorithms have been produced to solve heuristically vertex colouring problem. The most elementary one is a greedy algorithm that is quick and provides sometimes reasonably good solutions. The greedy way to find an “optimum” solution is widely discussed: is it good or bad. See for example [Kucera 1991; Borodin et al. 2003] for this discussion. The next algorithm after the greedy one that is worth to name is DSatur, which was developed by Brelaz [Brelaz 1979]. This algorithm is widely adopted to be used as a benchmark for testing other algorithms. Later researches have been split on two major approaches in the finding the heuristic vertex colouring: local search algorithms and backtracking algorithms. The local search approach tries to search a better solution than already found in the neighbourhood of it using some set of constraint violations. During this search this set is minimised and a solution is found when this set becomes empty. This technique is not able to exploit fully a graph structure and performance quite bad if the global optimum lies behind a local minimum from the already found local optimum. One of the most known examples of this technique is a tabu search. The backtracking approach tries to construct solutions from partial consistent assignments of domain values to variables. They often use techniques such as constraint propagation, value and variable ordering heuristics, branch-and-bound and intelligent backtracking. [Prestwich 2001]. Both those approaches exist since none of them can outperform another one on all graphs. There are certain graph types or cases where one or another works much better: if we need to exploit a structure then backtrack is better, while local search could be used for large size problems. There are also different "add-in"s techniques that are able to optimise performance of core methods on 8-15%, for example see the article published by Walshaw in the year 2001 [Walshaw 2001]

## 2.3.2 Greedy algorithm

The greedy algorithm takes vertices one by one and tries to add them into one of the existing colour classes (i.e. colour it with a colour corresponding to this class). If none of classes can be used to assign this vertex then a new colour is organised. There exist different techniques for choosing an initial ordering of vertices to colour and for choosing colour classes in an attempt to colour a current vertex. The algorithm can be described in pseudo-code as following:

Let's say that we have *n* vertices and we have *k* colours at each step.

```
$k$ = 1; Colour $v_1$ with $C_1$ ($C_k$)
For $i$ := 2 to $n$
    Try to colour $v_i$ with colour $C_j$, where $j$ = min (1,…,$k$)
    If none colour was used to colour $v_i$ then
        $k$ := $k$+1 [Produce a new colour];
        Colour $v_i$ with $C_k$
    End if
Next
```

It was proved that there always exists such initial ordering that will allow generating an optimal vertex colouring by the greedy algorithm. So, the problem of this algorithm's poor quality can be also formulated as a problem of a bad initial vertices ordering. One of the earliest attempts to produce a "good" ordering was made by Welsh and Powell [Welsh and Powell 1967] who suggested using a decreasing degree to order vertices.

## 2.3.3 DSatur

This heuristic algorithm was introduced by Brelaz [Brelaz 1979] and it is named *degree of saturation largest first* or DSATUR. It is a sequential colouring algorithm where the saturation degree defined as a number of colours a vertex is adjusted to. So at each step of the algorithm we are identifying a vertex with the maximum saturation degree among uncoloured and colour it with the least possible colour for that vertex (in this coloured neighbourhood). If a saturation degree will be equal for several vertices then the number of uncoloured neighbours is advised to be the next measure to use for the choice.

The saturation degree core idea is to try minimizing probability of setting an incorrect colour (that will increase number of colours require to colour a graph) by setting colours to a vertex with a maximum number of identified restrictions (by colours of already coloured neighbours). The algorithm can be described in pseudo-code as the following:

Let's say that we have *n* vertices, *W* will be uncoloured vertices and *Colour*(*v*) function will provide a colour already assigned to the vertex *v*.

```
While $W \neq Ø$          ($n$ steps)
    Find a vertex $v \in W$ with a maximum saturation degree
    Find a minimum colour that is not used in neighbourhood of $v$:
        $k$ := min ($i$ | there is no $s$ : Colour($s$) = $i$ , ($s$,$v$) $\in E$)
    Colour $v$ with the $k$ colour
    $W = W \setminus v$
```

Practice shows that this method requires up to 30% less colours than for greedy type algorithms.

There are also exist "backtracking" modifications of DSatur algorithm. One idea, for example, is to reorder vertices during backtrack as it was done by Korman [Korman 1979]

### 2.3.4 Iterated greedy

The method that was invited by Culberson [Culberson 1992] follows the greedy way in finding a vertex colouring with one important modification. The greedy colouring is used several times, i.e. repeatedly. Moreover, the order of vertices is changed each time before running the algorithm basing on the previous colouring and in such a way that it is guaranteed that each call will produce a new colouring using no more colours than the previous colouring. DSatur algorithm described above can be used to generate an initial vertex colouring.

**Lemma**: Let $C$ be a $k$-colouring of a graph $G$, and $\pi$ a permutation of the vertices such that if $C(v_{\pi(i)}) = C(v_{\pi(m)}) = c$, then $C(v_{\pi(j)}) = c$, for $i \leq j \leq m$. Then, applying the greedy algorithm to the permutation $\pi$ will produce a colouring $C'$ using $k$ or fewer colours. [Culberson 1992]

The main idea behind this lemma is the fact that reordering of vertices inside colour classes will not produce a bigger colouring. For example, reordering vertices for the next iteration in order of increasing colours will produce exactly the same colouring.

The next reordering of colour classes found to be efficient:

1. Reverse order
2. Increasing colour classes size
3. Decreasing size
4. Mixed: Do some steps by reordering one, then 2 and finally by 3 and loop again

The method makes vertex reordering and re-colouring until the specified colours number is reached or a specified number occurs without colouring improvements.

### 2.3.5 Tabu search

Tabu search is a local improvement search. It is based on partitioning the vertices of a graph into colour classes that may not represent a legal colouring, then the search attempts to reduce the number of colouring violations, or *conflicts*, by moving vertices from one class to another. Each iteration of it consists of generating a sample of neighbours; they are partitions that can be obtained from the current one by moving one vertex to a different class. Then it selects the neighbour partition that has the fewest conflicts, even if the neighbour has more conflicts than the current partition. The set of neighbours is restricted by an algorithm’s list that prevents a vertex from moving back into a class that it was recently a member of in a previous iteration. This helps the algorithm struggle out of local minima [Hertz and de Werra 1987].

# 3 NEW ALGORITHMS

## 3.1 Introduction Into a New Method

We are going to present several new algorithms that are designed to solve the maximum clique problem in this chapter. There are two main classes of the maximum clique problem – weighted and unweighted case. Therefore two subchapters are introduced in this work, one for each problem's case, containing algorithms to solve this particular case.

New algorithms are based on the Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990a], which is very efficient, easy to implement and is nothing more than a simple branch and bound algorithm with a brilliant idea of pruning. In other words it is a good starting point to introduce any ideas that could further improve this branch and bound algorithm further.

The philosophy of researches says that the more we know about an object, the better we understand it and more efficiently we can resolve any problem about the investigated object. At the same time the practice shows that a lot of existing algorithms tend to do very complex researches that have a little implication on the later finding of the maximum clique. The most important property of the Carraghan and Pardalos algorithm is that this algorithm does not spend time on such "unusable" researches and concentrates on the primary task – the maximum clique finding. So, it looks like the theory and the practice are showing totally opposite results in case of the maximum clique and this seems to be illogical. Therefore we started our research and identified the main task of it to find such type of information about a graph that can be efficiently used. It means that a time needed to derive such information should be less that a time we will win during the maximum clique finding, i.e. we should benefit from discovering information.

We have analysed the Carraghan and Pardalos algorithm and found that it heavily employs information about adjacent vertices and uses much less information on nonadjacent vertices. Although it looks to be just an opposite formulation of a question about vertices connections, we have found that there is a possibility to derive from it much more. New algorithms described below are built around the fact that nonadjacent vertices cannot be included into the same clique by the clique definition – any clique is formed by pairwise adjacent vertices. This property could be expanded from two nonadjacent vertices to a set of such vertices, which is called an "independent set" in the graph theory, i.e. only one vertex from any independent set can participate in a forming maximum clique. We used a vertex colouring algorithm to find such independent sets – each colour is nothing more that an "independent set", and the vertex colouring task could provides us with the minimum number of such sets. Of course, we cannot use exact algorithms for finding a vertex colouring since this task is also NP-complete, but we can use a heuristic one that can provide us with a good enough colouring to start finding the maximum clique from.

So, generally saying, new algorithms analyse a graph to be solved before running a main part, store results of the analyse and use those later to make algorithms' work more efficient.

## 3.2 Unweighted Case

In this chapter we are going to introduce new algorithms, explain those and bring some examples for graphs, which vertices have equal weights. Those weights are usually ignored or set to be equal to one for simplicity. Therefore such graphs called unweighted.

### 3.2.1 "VColor-u" – An algorithm based on a vertex colouring

In this subchapter we introduce an algorithm purely based on the idea of using independent sets without any additional speeding techniques. This algorithm mainly is used to verify if the idea is really worth to use, although it is also the best algorithm on some graph types – see the "Tests and Results" subchapter below.

#### 3.2.1.1 Description

Before starting the algorithm we find a vertex-colouring by using any heuristic algorithm, for example in a greedy manner. We determine colour classes one by one as long as uncoloured vertices exist. The vertices are resorted in the order they are added into colour classes. This order affects the algorithm's performance in finding the maximum clique and therefore is very important.

Definition 1: A colour class is a set of vertices, which were coloured by the same colour during applying a vertex-colouring algorithm.
*Note: A similar definition has been proposed by West in 2001, who defined the colour class as the following: vertices receiving a particular label (colour) for a colour class.*

Definition 2: A colour class is called existing on a subgraph $G_p$ if any vertex from this colour class belongs to this subgraph $G_p$.

Definition 3: Degree of a subgraph $G_p$ equals to the number of colour classes existing on that subgraph.

Crucial to the understanding of the algorithm is a notation of the depth and pruning formula. Basely, at the depth 1 we have all vertices, i.e. $G_1 \equiv G$. We are going to expand all vertices of a subgraph so that vertex is deleted from the subgraph after it is expanded. Another way is to have a cursor pointing to the vertex under analyses, so vertices in the front of that are excluded from the analyses / a subgraph of the current depth. Suppose we expand vertex $v_1$. At the depth 2, we consider all vertices adjacent to $v_1$ from the previous depth vertices, i.e. belonging to $G_1$. Those vertices form a subgraph $G_2$. At the depth 3, we consider all vertices (that are at the depth 2) adjacent to the vertex expanded in depth 2 etc. Let $v_{d1}$ be the vertex we are currently expanding at the depth $d$. That is:

Let's say that $G_d$ is a subgraph of $G$ on a depth $d$ that contains the following vertices: $V_d=(v_{d1}, v_{d2}, \ldots, v_{dm})$. The $v_{d1}$ is the vertex to be expanded.
Then a subgraph on the depth $d+1$ is $G_{d+1} = (V_{d+1},E)$,
where $V_{d+1}=(v_{d+1\,1}, \ldots, v_{d+1\,k})$: $\forall i\ v_{d+1\,i} \in V_d$ and $(v_{d+1\,i}, v_{d1}) \in E$.

As soon as a vertex is expanded and a subgraph, which is formed by this expansion, is analysed, this vertex is deleted from the depth and the next vertex of the depth become active, i.e. will be expanded.

The pruning formula is the next: If $d-1 + Degree(G_d) \leq CBC$, where $CBC$ is a size of the current maximum clique then we prune, since the size of the largest possible clique (formed by expanding any vertex of $G_d$) would be less or equal to $CBC$. If we are at depth 1 and this inequality holds then we stop; we have found the maximum clique.

We can prove that this pruning formula can be applied by the following theorem.

Theorem 1: If a degree of a subgraph of $G$ formed by vertices existing on a $d$-th depth and induced by $E$ is smaller or equal to the size of the current maximum clique minus $(d-1)$ then this subgraph cannot form a clique, which is larger than the already found.

Prove: It is clear to see that $(d - 1)$ equals to the number of vertices formed the $d$-th depth subgraph, i.e which where expanded on previous depths. Those $d - 1$ vertices are connected pairways and to each vertex of the subgraph of the $d$-th depth by the logic of branch and bound algorithms. It will be possible to find a larger clique than the already found one if and only if this subgraph can contain a clique, which is larger than a size of the current maximum clique minus $(d-1)$. If such clique exists then the maximal clique of the graph $G$ will be the clique of the subgraph plus $d-1$ vertices selected on previous depths, which are connected to all vertices of the subgraphs by the branch and bound algorithms logic and this maximal clique will be larger than an already found, so it will be a new maximum one. So, the only statement we need to prove is: the *Degree* function's value of the subgraph is never smaller than the maximum clique size that can be found on the subgraph, because then we can use in the pruning formula the degree function to estimate the size of the clique instead of finding it. The degree function gives a number of colours (colour classes) by definitions above and each colour class is an independent set of vertices existing on the depth. No more than one vertex of each colour class can participate in the maximum clique by the independent set's definition. Therefore the number of colours classes existing on the subgraph always equals or is bigger than a size of the maximum clique that can exist on the subgraph. ■

Note, that this degree function can be bigger in case some colour classes are not presented in the maximum clique of the subgraph.

A resorting of vertices during the vertex colouring can be used in the *Degree* function calculation to speed-up the algorithm - instead of calculating the degree of a subgraph each time on a depth we will calculate it only once the depth is formed and later just adjust this value by the following rule: if the next vertex on the depth to be expanded is from the same colour class as the previous one then the degree remains the same otherwise the degree should be decreased by 1 (there are no more vertices from the previous vertex' colour class and it is eliminated).

### 3.2.1.2 Algorithm

Algorithm for the maximum clique problem – "VColor-u"

```
CBC - current best (maximum) clique
d – depth
G_d – subgraph of G formed by vertices existing on the d-th depth

Step 0. Heuristic vertex-colouring: Find a vertex colouring and reorder vertices so
that first vertices in the new order belong to the last found colour class, then vertices of
the previous to the last colour class and so forth – vertices at the end should belong to
the first colour class. Note: It is advisable to use a special array to solve order of
vertices to avoid changing the adjacency matrix during vertices reordering.
Step 1. Initialization: $d = 1$.
Step 2. Check: If the current depth can contain a larger clique than the already found:
    If $d - 1 + Degree(G_d) \leq |CBC|$ then go to the step 5.
Step 3. Expand vertex: Get the next vertex to expand.
    If all vertices have been expanded or there are no vertices then:
        Check if the current clique is the largest one. If yes then save it.
        Go to the step 5.
Step 4. The next depth: Form a new depth by selecting vertices that are connected to
the expanding vertex among remaining vertices on the current depth;
    $d = d + 1$;
    Go to the step 2.
Step 5. Step back:
    $d = d - 1$;
    Delete the expanded vertex from the analysis on this depth; either delete the vertex
    directly or move the cursor forward
    if $d = 0$, then go to the end, otherwise go to the step 2.
End: Return the maximum clique.
```

### 3.2.1.3 Examples

Here we are going to present some examples of the previously described algorithm's work step by step. It should make the algorithm and its logic easier to understand.

A Moon-Moser graph has to be defined here before we will use it for our examples. The original paper of Moon and Moser defines this graph as a graph, vertices of which are divided into groups by three vertices and any vertex is connected to any other vertex doesn't belonging to the same group. The number of clique in this graphs is $3^{n/3}$, where *n* is the number of vertices [Moon and Moser 1965]. The more general definition says that this is a graph, where vertices are divided into groups, where any vertex is connected to all vertices of other groups.

### *3.2.1.3.1 Example 1*

#### 3.2.1.3.1.1 Description of the example graph

Consider graph shown in **Figure 2**. It is easy to see that a core of that is the Moon-Moser type subgraph containing vertices 1, 2 and 5 for the first class and vertices 6, 4 and 7 for the second class. Vertices 3, 9 and 8 are added to make the graph's structure more complex and contain larger cliques that the Moon-Moser subgraph produces.

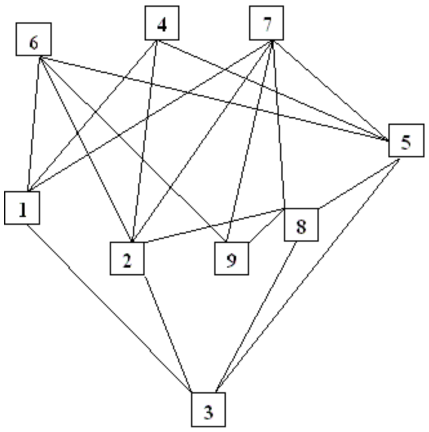

**Figure 2.** "VColor-u" – Graph of the example number 1

#### 3.2.1.3.1.2 Algorithm's steps

We determine colour classes one by one as long as uncoloured vertices exist in a greedy manner. This trivial algorithm for finding a vertex-colouring gives an acceptable result in average. The vertices are also resorted in an order they are added into colour classes. So, after the vertex colouring we have the next result:

Colour class 1 = {1, 2, 5, 9}
Colour class 2 = {3, 4, 6, 7}
Colour class 3 = {8};
The order of vertices is the following: {8, 7, 6, 4, 3, 9, 5, 2, 1}

Let's use the following notation in the example: *CBC* – the current best clique and |*CBC*| is its size. A grey vertex in the table below is a vertex under analysis and vertices in front of that are vertices that have been already analysed and cannot participate in the forming maximum clique any longer. So instead of deleting vertices we will just process them one by one in the example by moving a cursor, which always point to the grey vertex.

Steps of the main algorithm's part (finding the maximum clique) are described in the following table.

**Table 1.** "VColor-u" - Example 1 / Steps of finding the maximum clique

| **Depth** | **Subgraph** | **Step's description** |
|---|---|---|
| Depth 1: | 8,7,6,4,3,9,5,2,1 | \|*CBC*\| is 0; *Degree* = 3, since all colour classes are still under analysis. *d*-1+*Degree*=1-1+3=3. We continue our analyses since 3>\|*CBC*\| and go to the next depth: The grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | 7,3,9,5,2 | \|*CBC*\| is 0; *Degree* = 2, since colour classes 2 (vertices 7 and 3) and 1 (vertices 9, 5 and 2) exist. *d*-1+*Degree*=2-1+2=3. We continue our analyses since 3>\|*CBC*\| and go to the next depth: The grey vertex $v_{di}$ ($v_{21}$) to be expanded |
| Depth 3: | 9,5,2 | \|*CBC*\| is 0; *Degree* = 1, since only vertices of the first colour classes exist. *d*-1+*Degree*=3-1+1=3. We continue our analyses since 3>\|*CBC*\| and go to the next depth: The grey vertex $v_{di}$ ($v_{31}$) to be expanded. |
| Depth 4: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {8, 7, 9} and \|*CBC*\|=0, so *CBC* becomes {8, 7, 9}, and its size=3. Step back (up). |
| Depth 3: | 9,5,2 | *Degree*=1 since remaining vertices are 5 and 2. All of them belong to the colour class 1. So, the number of existing colour classes is 1. We prune since *d*-1+*Degree*=3-1+1 = 3 ≤ 3 (size of *CBC*). |
| Depth 2: | 7,3,9,5,2 | *Degree*=2 since remaining vertices (3, 9, 5, 2) belong to colour classes 1 and 2. We prune since *d*-1+*Degree*=2-1+2 = 3 ≤ 3 (size of *CBC*). |
| Depth 1: | 8,7,6,4,3,9,5,2,1 | *Degree*=2 since remaining vertices belong to colour classes 1 and 2. We prune since *d*-1+*Degree*= 1 - 1 + 2 = 2 < 3 (size of *CBC*). The current depth is 1 therefore we stop. |
| The maximum clique is {8, 7, 9}, size = 3. | | |

#### 3.2.1.3.1.3 Analysis of this example

The overall efficiency of the algorithm work on this example graph seems to be very high. It needed just 7 steps of the main algorithm to find the maximum clique and it was found directly during the first drill-down steps' sequence. The main reason of such efficiency is the Moon-Moser subgraph, which produce parallel structures. Those structures are the main successor part of the designed algorithm – instead of counting vertices of all parallel structures it gets into account just the largest one.

In this case we have found the maximum clique that covers all parallel maximum cliques – {8, 7, 9}. Besides, this maximum clique happens to be larger than Moon-Moser's subgraph, therefore we directly stepped back to the highest level and stopped the algorithm work.

## 3.2.1.3.2 *Example 2*

### 3.2.1.3.2.1 Description of the example graph

Consider the graph shown in **Figure 3**. It was shown in the previous example that the more parallel structures there are, the better it is for our algorithm. Besides, the closer a size of the maximum clique to the number of colours, the faster is the algorithm – we discuss this later in the "Preliminary analysis" subchapter of the "Test and Results" chapter. Therefore we tried to construct a graph that will be as "bad" as possible. Here we used the classical "Mycielski's construction" [West 2001] to construct the graph that needs 4 colours to be coloured, although it is triangle free.

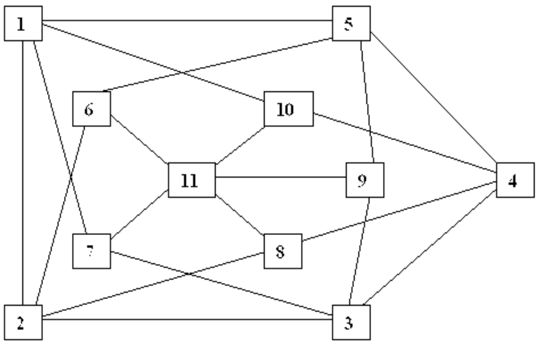

**Figure 3.** "VColor-u" – Graph of the example number 2

### 3.2.1.3.2.2 Algorithm's steps

Again we determine colour classes one by one as long as uncoloured vertices exist in a greedy manner. The vertices are also resorted in an order they are added into colour classes. So the vertex colouring gives as the next result:

Colour class 1 = {1, 3, 6, 8}
Colour class 2 = {2, 5, 7, 10}
Colour class 3 = {4, 9};
Colour class 4 = {11};
The order of vertices is the following: {11, 9, 4, 10, 7, 5, 2, 8, 6, 3, 1}

We use the same notation as in the previous example: *CBC* – the current best clique and |*CBC*| is the size of the current best clique. A grey vertex in the table below is a vertex under analysis and vertices in front of that are vertices that have been already analysed and cannot participate in the forming maximum clique any longer. So instead of deleting vertices we will just process them one by one by moving a cursor forward.

Steps of the main algorithm's part (finding the maximum clique) are described in the following table.

**Table 2.** "VColor-u" - Example 2 / Steps of finding the maximum clique

| **Depth** | **Subgraph** | **Step's description** |
|---|---|---|
| Depth 1: | 11,9,4,10,7,5,2,8,6,3,1 | \|*CBC*\| is 0; *Degree* = 4, since all colour classes are still in the analyses. *d*-1+*Degree*=1-1+4=4. Since 4>\|*CBC*\| we continue our analyses and go to the next depth: The grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | 9,10,7,8,6 | \|*CBC*\| is 0; *Degree* = 3, since colour classes 3 (vertex 9), 2 (vertices 10 and 7) and 1 (vertices 8 and 6) exist. *d*-1+*Degree*=2-1+3=4. Since 4>\|*CBC*\| we continue our analyses and go to the next depth: The grey vertex $v_{di}$ ($v_{21}$) to be expanded |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {11, 9} and \|*CBC*\| =0, so *CBC* becomes {11, 9}, and its size=2. Step up (to the previous depth) |
| Depth 2: | 9,10,7,8,6 | *Degree*=2 since colour classes 2 (vertices 10 and 7) and 1 (vertices 8 and 6) exist. So, there exist 2 colour classes. *d*-1+*Degree*=2-1+2=3. Since 3>\|*CBC*\| we continue our analyses and go to the next depth: The grey vertex $v_{di}$ ($v_{22}$) to be expanded |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {11, 10} and \|*CBC*\| =2, so *CBC* is not smaller. Step up. |
| Depth 2: | 9,10,7,8,6 | *Degree*=2 since colour classes 2 (vertex 7) and 1 (vertices 8 and 6) exist. *d*-1+*Degree*=2-1+2=3. Since 3>\|*CBC*\| we continue our analyses and go to the next depth: The grey vertex $v_{di}$ ($v_{23}$) to be expanded. |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {11, 7} and \|*CBC*\| =2, so *CBC* is not smaller. Step up. |
| Depth 2: | 9,10,7,8,6 | *Degree*=1 since remaining vertices (8 and 6) belong to the colour class 1. We prune since *d*-1+*Degree*=2-1+1 = 2 ≤ 2 (size of *CBC*). |
| Depth 1: | 11,9,4,10,7,5,2,8,6,3,1 | *Degree*=3 since remaining vertices belong to colour classes 1, 2 and 3. *d*-1+*Degree*=1-1+3=3. Since 3>\|*CBC*\| we continue our analyses and go to the next depth: The grey vertex $v_{di}$ ($v_{12}$) to be expanded. |

| | | |
|---|---|---|
| Depth 2: | 5, 3 | *Degree*=2 since colour classes 2 (vertex 5) and 1 (vertex 3) exist. $d$-1+*Degree*=2-1+2=3. Since 3>\|*CBC*\| we continue our analyses and go to the next depth: The grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {9, 5} and \|*CBC*\| =2, so *CBC* is not smaller. Step up. |
| Depth 2: | 5, 3 | *Degree*=1 since only the first colour class exists (vertex 3). We prune since $d$-1+*Degree*=2-1+1=2≤ 2 (size of *CBC*). |
| Depth 1: | 11,9,4,10,7,5,2,8,6,3,1 | *Degree*=3 since remaining vertices belong to colour classes 1, 2 and 3. $d$-1+*Degree*=1-1+3=3. Since 3>\|*CBC*\| we continue our analyses and go to the next depth: The grey vertex $v_{di}$ ($v_{13}$) to be expanded. |
| Depth 2: | 10, 5, 8, 3 | *Degree*=2 since colour classes 2 (vertices 10 and 5) and 1 (vertices 8 and 3) exist. $d$-1+*Degree*=2-1+2=3. Since 3>\|*CBC*\| we continue our analyses and go to the next depth: The grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {4, 10} and \|*CBC*\| =2, so *CBC* is not smaller. Step up. |
| Depth 2: | 10, 5, 8, 3 | *Degree*=2 since colour classes 2 (vertex 5) and 1 (vertices 8 and 3) exist. $d$-1+*Degree*=2-1+2=3. Since 3>\|*CBC*\| we continue our analyses and go to the next depth: The grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {4, 5} and \|*CBC*\| =2, so *CBC* is not smaller. Step up. |
| Depth 2: | 10, 5, 8, 3 | *Degree*=1 since only the first colour class exists (vertices 8 and 3). We prune since $d$-1+*Degree*=2-1+1=2≤ 2 (size of *CBC*). |
| Depth 1: | 11,9,4,10,7,5,2,8,6,3,1 | *Degree*=2 since remaining vertices belong to colour classes 1 or 2. We prune since $d$-1+*Degree*= 1 - 1 + 2 = 2 ≤ 2 (size of *CBC*). The current depth is 1 therefore we stop. |
| The maximum clique is {11, 7}, size = 2. | | |

#### 3.2.1.3.2.3 Analysis of this example

The general efficiency of the algorithm this time wasn't as high as it was for the previous example, although the main idea worked here also well on last steps. The algorithms had to analyse some parallel structures that seemed to be different, although it wasn't so. It is easy to see that the reason of that is the special graph construction we used – to be quite "bad" for resolving by our algorithm.

Note that the maximum clique was found again during the first sequence of steps as in the previous example. Besides, the pruning techniques worked here also and we needed just 19 steps to resolve a graph containing 11 vertices. There were still 8 vertices in the analyses when the algorithm was able to conclude that the maximum clique is already found and stop the process.

We can conclude, basing on all previously described, that even the "bad" construction of the graph wasn't able to eliminate the power of using the introduced independent sets technique completely and it still works and shows quite good results. Another interesting fact we have seen in this example – vertices that produce a higher chromatic number (in compare with the maximum clique size) were analysed during first steps. As soon as those vertices were eliminated the algorithm work became very efficient and it was able stop the analyses.

## 3.2.2 "VColor-BT-u" – An algorithm based on a vertex colouring

In this chapter we introduce another algorithm that is still based on the idea of using independent sets. This new algorithm is constructed basing on the Östergård algorithm [Östergård 2002]. This algorithm is another modification of the Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990a] and contains a very powerful backtracking idea that makes this algorithm to be the quickest one at the moment [Östergård 2002]. We are going to apply the idea of backtracking search on our previous algorithm, but we are going to backtrack on a higher level than Östergård algorithm does: by independent sets instead of by individual vertices.

### 3.2.2.1 Description

This algorithm is also based on the Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990a] as the previous one and we are going to describe this part of the algorithm, but instead will describe all modifications. First of all we introduce a "vertices" backtracking technique used by Östergård [Östergård 2002] and then an "independent sets" backtracking technique.

The original Carraghan and Pardalos algorithm considers first of all all cliques that contain the first vertex $v_1$ and could contain other graph vertices. Then it considers all cliques that contain $v_2$ and could contain all other vertices except $v_1$. Generally saying, it considers at the $i$-th step all cliques that contain $v_i$ and could contain vertices $\{v_{i+1}, v_{i+2}, \dots ,v_n\}$. This technique is nothing else than a standard branch and bound way of drilling a graph for finding the solution.

The backtracking technique does the graph research in the opposite order, although the list of vertices on the *i*-th step is the same. First of all it considers all cliques that could be built using only $v_n$. Then it considers all cliques that contain $v_{n-1}$ and could contain $v_n$, and so forth. The general rule – it considers at the *i*-th step all cliques that contain $v_i$ and could contain vertices $\{v_{i+1}, v_{i+2}, ... ,v_n\}$. So we move from the *n*-th step to the first step decreasing the step number. Initially it looks like a slower technique in comparison to the original Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990a], but makes it possible to introduce a new backtracking pruning technique speeding up the algorithm's work. First of all, note that the backtracking vertices selection is used only on the "general" level – as soon as vertices are selected for the *i*-th backtracking step, the same branch and bound technique is used. The branch and bound algorithm uses the same Carraghan and Pardalos pruning technique and the new backtracking pruning technique described below. The algorithm uses to remember the maximum clique found for each vertex at the highest level into a special array *b*. So $b[i]$ is the maximum clique for the *i*-th vertex while searching backward. These numbers are used later by the following rule: if we search for a clique of size greater than *s*, then we can prune the search, if we consider $v_i$ to become the $(j + 1)$-th vertex and $j+ b[i] \leq s$ [Östergård 2002]. Besides, we can stop the backtracking iteration and go to the next one if a new maximum clique is found since the maximum clique size of a subgraph formed by $\{v_{i+1}, v_{i+2}, ... ,v_n\}$ is either equal to the maximum clique size of a subgraph formed by $\{v_{i+2}, v_{i+3}, ... ,v_n\}$ (the previous step) or is larger on 1. Please refer to the original Östergård article [Östergård 2002] for proves that this technique always gives the exact solution.

Now we are going to introduce the "independent sets" (or colour classes) backtracking technique. We do the same as described above, except we operate on the "independent sets" level of considering a graph. Let's say that we have divided, as it is described previously in the "VColor-u" algorithm, all vertices by colour classes, i.e. $V = \{C_n, C_{n-1}, ..., C_1\}$, where $C_i$ is the *i*-th colour (or we call it the *i*-th colour class). First of all we consider all cliques that could be built only using vertices of the $C_1$, i.e. of the first colour class. Then we consider all cliques that could be built using vertices of $C_1$ and $C_2$, i.e. of the first and second colour classes, and so forth. The general rule – we consider at the *i*-th step all cliques that can contain vertices of $\{C_i, C_{i-1}, ..., C_1\}$. *Note that here we again move from the first step to the n-th since colour classes are in the backward order.*

Besides the algorithm uses to remember the maximum clique found for each step on the high level into a special array *b*. So $b[i]$ is the maximum clique for a subgraph formed by $\{C_i, C_{i-1}, ..., C_1\}$ vertices while searching backward. This numbers are used later by the following rule: if we search for a clique of size greater than *s*, then we can prune the search if we consider $v_i$ to become the $(j + 1)$-th vertex and it belongs to the *k*-th colour class and $j+ b[k] \leq s$. The stopping condition of the backtrack search iteration is also remains since the maximum clique size of a subgraph formed by $\{C_i, C_{i-1}, ..., C_1\}$ is either equal to the maximum clique size of a subgraph formed by $\{C_{i-1}, ..., C_1\}$ or is larger on 1. It is so because each time we just add a colour class, i.e. an independent set in addition to the analysed set of vertices. The new maximum clique cannot differ more than on 1 vertex from the maximum clique on the previous iteration since all added vertices are pairways nonadjacent and therefore there are no two or more vertices which are adjacent and can be used / added to a new maximum clique.

Note: It is important again to sort vertices as we have shown it at the start of the description: $V = \{C_n, C_{n-1}, ..., C_1\}$, i.e. first of all in the new sorted order vertices of the $n$-th colour class should appear, then vertices of the ($n$-1)-th colour class and so forth.

The colour classes pruning technique, which where introduced earlier for the "VColor-u" algorithm is also used in parallel with the backtracking pruning.

### 3.2.2.2 Algorithm

Algorithm for the maximum clique problem – "VColor-BT-u"

*CBC* - current best (maximum) clique
*d* – depth
*i* – index of the currently processed colour class in the backtracking
*b* – array of the backtrack search results
$C(v_i)$ – a function that return a colour class to which the vertex $v_i$ belongs
$G_d$ – subgraph of $G$ formed by vertices existing on the depth $d$

**Step 0. Heuristic vertex-colouring**: Find a vertex colouring and reorder vertices so that first vertices belong to the last found colour class then vertices of the previous to last colour class and so forth – vertices at the end should belong to the first colour class. *Note: It is advisable to use a special array to solve order of vertices to avoid changing the adjacency matrix during reordering vertices.*

```
Step 1. Backtracking: For each colour class starting from the first one up to the last,
        i.e. $i = i+1$:
    Step 1.1. Subgraph building. Form the first depth by selecting all vertices of the
    current colour class under the analysis and other colour classes, whose index is
    smaller than the index of the current colour class.
        $i$ = the index of the current colour class.
    Step 1.2. Run the subgraph research: Go to the step 2

        Step 2. Initialization: $d = 1$.
        Step 3. Check: If the current depth can contain a larger clique than already
        found:
            Step 3.1. If $d - 1 + Degree(G_d) \le |CBC|$ then go to the step 6.
            Step 3.2. if $C(v_{d\,1}) > i$ then If $d - 1 + b[C(v_{d\,1})] \le |CBC|$ then go to the step 6.
        Step 4. Expand vertex: Get the next vertex to expand.
            If all vertices have been expanded or there are no vertices then:
                Check if the current clique is the largest one. If yes then save it.
                Go to the step 1.3.
        Step 5. The next depth: Form a new depth by selecting all remaining vertices
        that are connected to the expanding vertex from the current depth;
            $d = d + 1$;
            Go to the step 3.
        Step 6. Step back:
            $d = d - 1$;
```

Delete the expanded vertex from the analysis on this depth;
if $d = 0$, then go to the step 1.3, otherwise go to the step 3.

**Step 1.3. Completing iteration:** $b[i]$ =*CBC*, go to the step 1.
**End:** Return the maximum clique.

Steps from 2 to 6 can be considered as a subprocedure that the backtracking runs iteratively in a cycle for each colour class.

## 3.2.2.3 Examples

In this chapter we are going to demonstrate some examples of the described algorithm work. The same graphs will be used as for the previous algorithm.

### *3.2.2.3.1 Example 1*

#### 3.2.2.3.1.1 Description of the example graph

Consider graph shown in **Figure 4**. Again it is a graph that is built using the Moon-Moser type subgraph containing vertices 1, 2 and 5 for the first class and vertices 6, 4 and 7 for the second class. Vertices 3, 9 and 8 are used to make the graph's structure more complex and contain larger cliques that the Moon-Moser subgraph produces.

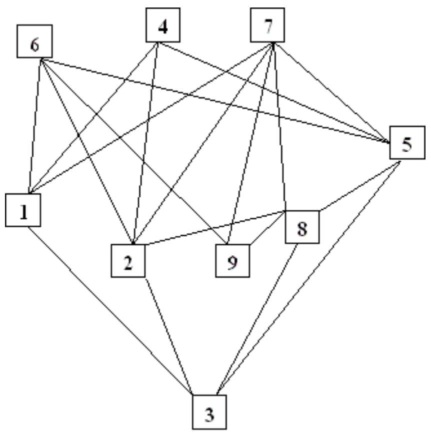

**Figure 4.** "VColor-BT-u" – Graph of the example number 1

#### 3.2.2.3.1.2 Algorithm's steps

We determine colour classes one by one as long as uncoloured vertices exist in a greedy manner. Vertices are also resorted in an order those are added into colour classes. So, vertex colouring gives the following result:

Colour class 1={1, 2, 5, 9}
Colour class 2={3, 4, 6,7}
Colour class 3={8};
The order of vertices is the following: {8, 7, 6, 4, 3, 9, 5, 2, 1}

We use the same notation as in the algorithm's description above. A grey vertex in the table below is a vertex under analysis and vertices in front of that are vertices that have been already analysed and cannot participate in the forming maximum clique any longer.

Steps of the main algorithm's part (finding the maximum clique) are described in the next table.

**Table 3.** "VColor-BT-u" - Example 1 / Steps of finding the maximum clique

| **Depth** | **Subgraph** | **Step's description** |
|---|---|---|
| Depth 0: | 8,7,6,4,3, *9,5,2,1* | To do: Start a backtrack search from the first class by selecting vertices of it into the depth 1 and run main steps.<br>$i = 1$ |
| Depth 1: | 9,5,2,1 | \|*CBC*\| is 0; *Degree* = 1, since only first colour class vertices exist.<br>*d*-1+*Degree*=1-1+1=1. Since 1>\|*CBC*\| we can continue.<br><br>$C(v_{11})$=1 since $v_{11}$ belongs to the colour class number 1. The backtracking pruning is skipped since $C(v_{11}) = i$.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {9} and \|*CBC*\| =0, so *CBC* becomes {9}, size=1.<br>*b*[1] =1. Go to the next iteration of the backtrack search. |
| Depth 0: | 8, *7,6,4,3,9,5,2,1* | To do: Start the next step of the backtrack search by selecting into the depth 1 vertices of colour classes 1 and 2, and run main steps. $i = 2$ |
| Depth 1: | 7,6,4,3,9,5,2,1 | \|*CBC*\| is 1; *Degree* = 2, since existing vertices belong to colour classes 1 and 2.<br>*d*-1+*Degree*=1-1+2=2. Since 2>\|*CBC*\| we can continue.<br><br>$C(v_{11})$=2 since $v_{11}$ belongs to the colour class number 2. The backtracking pruning is skipped since $C(v_{11}) = i$.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | 9,5,2,1 | \|*CBC*\| is 1; *Degree* = 1, since all vertices belong to the colour class number 1.<br>*d*-1+*Degree*=2-1+1=2. Since 2>\|*CBC*\| we can continue. |

| | | |
|---|---|---|
| | | $C(v_{21})$=1 since $v_{21}$ belongs to the colour class number 1 => We can check the backtrack pruning condition:<br>$d$ –1 + $b[C(v_{21})]$ = 2-1+1 = 2 >\|*CBC*\| we can continue.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {7, 9} and \|*CBC*\| =1, so *CBC* becomes {7, 9}, size=2.<br>$b$[2] =2. Go to the next iteration of the backtrack search. |
| Depth 0: | *8,7,6,4,3,9,5,2,1* | To do: Start the next step of the backtrack search by selecting into the depth 1 vertices of colour classes 1, 2 and 3, and run main steps. $i$ = 3 |
| Depth 1: | 8,7,6,4,3,9,5,2,1 | \|*CBC*\| is 2; *Degree* = 3, since vertices belong to colour classes 1, 2 and 3.<br>*d*-1+*Degree*=1-1+3=3. Since 3>\|*CBC*\| we can continue.<br><br>$C(v_{11})$=3 since $v_{11}$ belongs to the colour class number 3. The backtracking pruning is skipped since $C(v_{11}) = i$.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | 7,3,9,5,2 | \|*CBC*\| is 2; *Degree* = 2, since all vertices belong to colour classes 1 and 2.<br>*d*-1+*Degree*=2-1+2=3. Since 3>\|*CBC*\| we can continue.<br>$C(v_{21})$=2 since $v_{21}$ belongs to the colour class number 2 => We can check the backtrack pruning condition:<br>$d$ –1 + $b[C(v_{21})]$ = 2-1+2 = 3 >\|*CBC*\| we can continue.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | 9,5,2 | \|*CBC*\| is 2; *Degree* = 1, since all vertices belong to the colour class number 1.<br>*d*-1+*Degree*=3-1+1=3. Since 3>\|*CBC*\| we can continue. |

| | | |
|---|---|---|
| | | $C(v_{31})=1$ since $v_{31}$ belongs to the colour class number 1 => We can check the backtrack pruning condition:<br>$d-1+b[C(v_{31})] = 3-1+1 = 3 >\|CBC\|$ we can continue.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{31}$) to be expanded. |
| Depth 4: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {8, 7, 9} and \|*CBC*\| =2, so *CBC becomes* {8, 7, 9}, size=3.<br>*b*[3] =3. Go to the next iteration of the backtrack search. |
| Depth 0: | *8,7,6,4,3,9,5,2,1* | Since all colour classes are analysed the algorithm stops. |
| The maximum clique is {8, 7, 9}, size = 3. | | |

#### 3.2.2.3.1.3 Analysis of this example

The algorithm needed just 17 steps to find the maximum clique from the graph of 8 vertices. This result is very good, since the maximum clique finding problem is NP-hard and a lot of algorithms just do an exhaustive search or need a sufficient number of steps to find a solution. So, the improvement is huge from this point of view. We should mark that 17 steps is more than 7 steps of the "VColor-u" algorithm for the same graph. It happens as this graph is not so "good" for applying with this type of algorithm – as you have probably marked the backtracking pruning formula never worked in this example. At the same time it is possible to learn a lot from this example as well. It demonstrated to us a power of using:

1. The backtracking with independent sets – using of backtracking with independent sets has a set of advantages. First of all we do less iterations since select all vertices of a class. At the same moment the number of steps inside each iteration does not increase as colour class' vertices are "parallel", i.e. cannot be included into the same maximum clique and have equal *b*[*i*] value, since they are coloured into the same colour.

2. The stopping condition of the backtracking iteration – We have skipped a lot of steps using a rule that if we have found a new maximum clique then we can go directly into the next backtracking iteration, since the current iteration's subgraph cannot produce any larger clique. This stop condition is a very important technique in addition to the backtracking pruning rule.

### 3.2.2.3.2 *Example 2*

Although we have analysed this graph for the "VColor-u" algorithm previously, we are not going to do it for this algorithm. A reason of that is simple - our independent sets based pruning technique dominates over the backtracking pruning technique on it and an example on this graph will be practically identical to the previous example. Everyone, who is interested to see this, can easily apply step by step our algorithm to this graph and see that. Our aim is to demonstrate cases and explain using of each important part of our algorithm, therefore we will go directly to the next example.

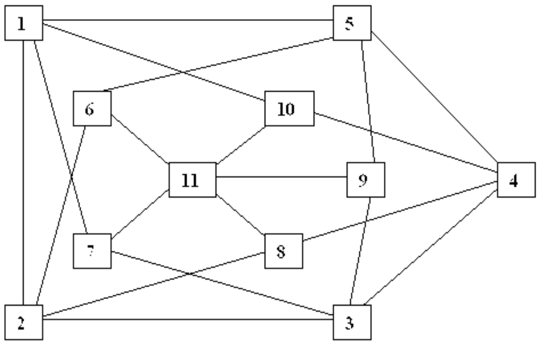

**Figure 5.** "VColor-BT-u" – Graph of the example number 2

### 3.2.2.3.3 *Example 3*

#### 3.2.2.3.3.1 Description of the example graph

Consider graph shown in **Figure 6**. Here we have constructed a graph to demonstrate the backtracking pruning technique work. As it is possible to see, on that graph we have started from the "Mycielski's construction" [West 2001] also to construct the graph that needs 3 colours to be coloured although is triangle free – vertices 1, 2, 3, 4 and 5. At the next construction step we have duplicated this graph's construction by adding vertices 6, 7 and 8 and using vertices 2 and 3. Then we added a vertex 9 to produce a triangle with vertices 3 and 7. Finally we added a vertex 10 that should produce one more colour if we will use the greedy colouring, but will not produce any larger cliques than already existing.

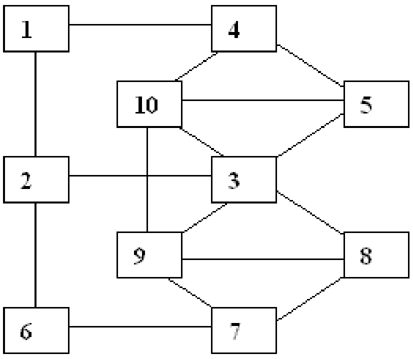

**Figure 6.** "VColor-BT-u" – Graph of the example number 3

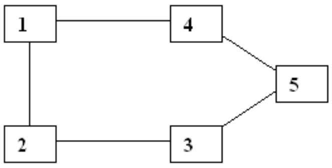

**Figure 7.** "VColor-BT-u" – Graph of the example number 3 / step 1

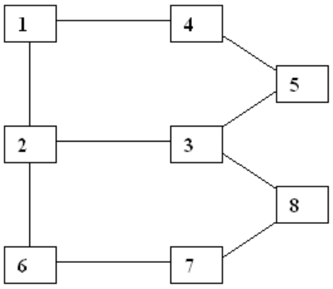

**Figure 8.** "VColor-BT-u" – Graph of the example number 3 / step 2

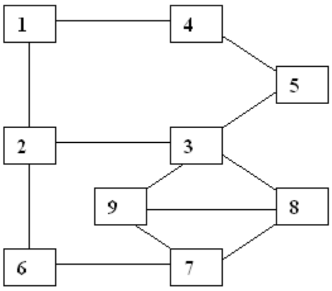

**Figure 9.** "VColor-BT-u" – Graph of the example number 3 / step 3

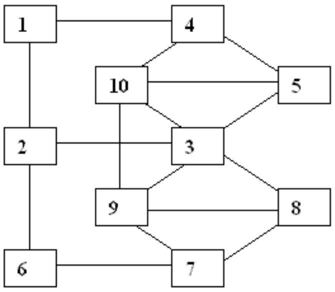

**Figure 10.** "VColor-BT-u" – Graph of the example number 3 / step 4

We made an assumption that only greedy colouring can be used to make the example smaller (less vertices), although we could construct a similar case using "Mycielski's construction" where any re-colouring will not reduce the number of colours produced by other colouring techniques, but the backtracking technique will still work and save us from doing a lot of unnecessary steps. We will discuss this under the "Analysis" section of that example.

### 3.2.2.3.3.2 Algorithm's steps

Again we determine colour classes one by one as long as uncoloured vertices exist in a greedy manner. Vertices are also resorted in an order they are added into colour classes. So, vertex colouring gives the following result:

Colour class 1={1, 3, 6}
Colour class 2={2, 4, 7}
Colour class 3={5, 8};
Colour class 4={9};
Colour class 5={10};
The order of vertices is the following {10, 9, 8, 5, 7, 4, 2, 6, 3, 1}

The same notation is used as in the algorithm's description above. A grey vertex in the table below is a vertex under analysis and vertices in front of that are vertices that have been already analysed and cannot participate in the forming maximum clique any longer.

The steps of the main algorithm's part (finding the maximum clique) are described in the next table.

**Table 4.** "VColor-BT-u" - Example 3 / Steps of finding the maximum clique

| **Depth** | **Subgraph** | **Step's description** |
|---|---|---|
| Depth 0: | 10,9,8,5,7,4,2,*6,3,1* | To do: Start a backtrack search from the first class by selecting vertices of it into the depth 1 and run main steps.<br>$i$ =1 |
| Depth 1: | 6,3,1 | \|*CBC*\| is 0; *Degree* = 1, since only the first colour class vertices exist.<br>*d*-1+*Degree*=1-1+1=1. Since 1>\|*CBC*\| we can continue.<br><br>$C(v_{11})$=1 since $v_{11}$ belongs to the colour class number 1. The backtracking pruning is skipped since $C(v_{11})$=***i*.**<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {6} and \|*CBC*\|=0, so *CBC* becomes {6}, size=1.<br>*b*[1] =1. Go to the next iteration of the backtrack search. |
| Depth 0: | 10,9,8,5,*7,4,2,6,3,1* | To do: Start the next step of the backtrack search by selecting into the depth 1 vertices of colour classes 1 and 2, and run main steps. $i$ = 2 |
| Depth 1: | 7,4,2,6,3,1 | \|*CBC*\| is 1; *Degree* = 2, since vertices belong to colour classes 1 and 2.<br>*d*-1+*Degree*=1-1+2=2. Since 2>\|*CBC*\| we can continue.<br><br>$C(v_{11})$=2 since $v_{11}$ belongs to the colour class number 2. The backtracking pruning is skipped since $C(v_{11})$ = ***i*.**<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | 6 | \|*CBC*\| is 1; *Degree* = 1, since all vertices belong to the colour class number 1.<br>*d*-1+*Degree*=2-1+1=2. Since 2>\|*CBC*\| we can continue. |

| | | |
|---|---|---|
| | | $C(v_{21})$=1 since $v_{21}$ belongs to the colour class number 1 => We can check the backtrack pruning condition:<br>$d$ –1 + $b[C(v_{21})]$ = 2-1+1 = 2 >\|*CBC*\| we can continue.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {7, 6} and \|*CBC*\|=1, so *CBC* becomes {7, 6}, size=2.<br>*b*[2] =2. Go to the next iteration of the backtrack search. |
| Depth 0: | 10,9,*8,5,7,4,2,6,3,1* | To do: Start the next step of the backtrack search by selecting into the depth 1 vertices of colour classes 1, 2 and 3, and run main steps. $i$ = 3 |
| Depth 1: | 8,5,7,4,2,6,3,1 | \|*CBC*\| is 2; *Degree* = 3, since vertices belong to colour classes 1, 2 and 3.<br>*d*-1+*Degree*=1-1+3=3. Since 3>\|*CBC*\| we can continue.<br><br>$C(v_{11})$=3 since $v_{11}$ belongs to the colour class number 3. The backtracking pruning is skipped since $C(v_{11}) = \boldsymbol{i}$.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | 7,3 | \|*CBC*\| is 2; *Degree* = 2, since vertices belong to colour classes 1 and 2.<br>*d*-1+*Degree*=2-1+2=3. Since 3>\|*CBC*\| we can continue.<br><br>$C(v_{21})$=2 since $v_{21}$ belongs to the colour class number 2 => We can check the backtrack pruning condition:<br>$d$ –1 + $b[C(v_{21})]$ = 2-1+2 = 3 >\|*CBC*\| we can continue.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {8, 7} and \|*CBC*\|=2, so *CBC* is not smaller.<br>Go up. |

| | | |
|---|---|---|
| Depth 2: | 7,3 | \|*CBC*\| is 2; *Degree* = 1, since remaining vertex 3 belongs to the colour class 1.<br>We prune since *d*-1+*Degree*=2-1+1=2 ≤ 2 (size of *CBC*).<br>Go up. |
| Depth 1: | 8,5,7,4,2,6,3,1 | \|*CBC*\| is 2; *Degree* = 3, since vertices belong to colour classes 1, 2 and 3.<br>*d*-1+*Degree*=1-1+3=3. Since 3>\|*CBC*\| we can continue.<br><br>$C(v_{12})$=3 since $v_{12}$ belongs to the colour class number 3. The backtracking pruning is skipped since $C(v_{12})$ = ***i*.**<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{12}$) to be expanded. |
| Depth 2: | 4,3 | \|*CBC*\| is 2; *Degree* = 2, since vertices belong to colour classes 1, and 2.<br>*d*-1+*Degree*=2-1+2=3. Since 3>\|*CBC*\| we can continue.<br><br>$C(v_{21})$=2 since $v_{21}$ belongs to the colour class number 2 => We can check the backtrack pruning condition:<br>$d - 1 + b[C(v_{21})]$ = 2-1+2 = 3 >\|*CBC*\| we can continue.<br>Go to the next depth: the grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {5, 4} and \|*CBC*\| =2, so *CBC* is not smaller. Step up. |
| Depth 2: | 4,3 | \|*CBC*\| is 2; *Degree* = 1, since remaining vertex 3 belongs to the colour class 1.<br>We prune since *d*-1+*Degree*=2-1+1=2 ≤ 2 (size of *CBC*).<br>Go up. |
| Depth 3: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {4, 5} and \|*CBC*\| =2, so *CBC* is not smaller. Step up. |
| Depth 1: | 8,5,7,4,2,6,3,1 | \|*CBC*\| is 2; *Degree* = 2, since vertices belong to colour classes 1 and 2.<br>*d*-1+*Degree*=1-1+2=2. Since 2≤ 2 (size of *CBC*).<br>Since d-1=0, then *b*[3]=2, go to the next iteration of the backtrack search. |

| | | |
|---|---|---|
| Depth 0: | 10, *9,8,5,7,4,2,6,3,1* | To do: Start the next step of the backtrack search by selecting into the depth 1 vertices of colour classes 1, 2, 3 and 4, and run main steps. $i = 4$ |
| Depth 1: | 9,8,5,7,4,2,6,3,1 | $\lvert CBC \rvert$ is 2; *Degree* = 4, since vertices belong to colour classes 1, 2, 3 and 4.<br>$d$-1+*Degree*=1-1+4=4. Since 4>$\lvert CBC \rvert$ we can continue.<br><br>$C(v_{11})$=4 since $v_{11}$ belongs to the colour class number 4. The backtracking pruning is skipped since $C(v_{11}) = \boldsymbol{i}$.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | 8,7,3 | $\lvert CBC \rvert$ is 2; *Degree* = 3, since vertices belong to colour classes 1, 2 and 3.<br>$d$-1+*Degree*=2-1+3=4. Since **4**>$\lvert CBC \rvert$ we can continue.<br><br>$C(v_{21})$=3 since $v_{21}$ belongs to the colour class number 3 => We can check the backtrack pruning condition:<br>$d - 1 + b[C(v_{21})]$ = 2-1+2 = **3** >$\lvert CBC \rvert$ we can continue.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | 7, 3 | $\lvert CBC \rvert$ is 2; *Degree* = 2, since vertices belong to colour classes 1, and 2.<br>$d$-1+*Degree*=2-1+2=3. Since 3>$\lvert CBC \rvert$ we can continue.<br><br>$C(v_{31})$=2 since $v_{31}$ belongs to the colour class number 2 => We can check the backtrack pruning condition:<br>$d - 1 + b[C(v_{31})]$ = 2-1+2 = 3 >$\lvert CBC \rvert$ we can continue.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{31}$) to be expanded. |
| Depth 4: | ∅ | The depth doesn't contain any vertices ⇒ Check if the formed clique is the largest one: The formed clique is {9, 8, 7} and $\lvert CBC \rvert$=2, so *CBC becomes* {9, 8, 7}, size=3.<br>$b[4]$=2. Go to the next iteration of the backtrack search. |

| | | |
|---|---|---|
| Depth 0: | *10,9,8,5,7,4,2,6,3,1* | To do: Start the next step of the backtrack search by selecting into the depth 1 vertices of colour classes 1, 2, 3, 4 and 5, and run main steps.<br>$i = 5$ |
| Depth 1: | 10,9,8,5,7,4,2,6,3,1 | \|*CBC*\| is 3; *Degree* = 5, since vertices belong to all colour classes.<br>$d$–1+*Degree*=1-1+5=5. Since 5>\|*CBC*\| we can continue.<br><br>$C(v_{11})$=5 since $v_{11}$ belongs to the colour class number 5. The backtracking pruning is skipped since $C(v_{11}) = \boldsymbol{i}$.<br><br>Go to the next depth: the grey vertex $v_{di}$ ($v_{11}$) to be expanded. |
| Depth 2: | 9,5,4,3 | \|*CBC*\| is 3; *Degree* = 4, since vertices belong to colour classes 1, 2, 3 and 4.<br>$d$–1+*Degree*=2-1+4=5. Since **5**>\|*CBC*\| we can continue.<br><br>$C(v_{21})$=4 since $v_{21}$ belongs to the colour class number 4 => We can check the backtrack pruning condition:<br>$d$–1 + $b[C(v_{21})]$ = 2-1+3 = **4** >\|*CBC*\| we can continue.<br>Go to the next depth: the grey vertex $v_{di}$ ($v_{21}$) to be expanded. |
| Depth 3: | 3 | \|*CBC*\| is 3; *Degree* = 1, since vertex belongs to the colour class 1.<br><br>$d$–1+*Degree*=3-1+1=3. Since 3≤\|*CBC*\| we prune. Go up. |
| Depth 2: | 9,5,4,3 | \|*CBC*\| is 3; *Degree* = 3, since vertices belong to colour classes 1, 2 and 3.<br>$d$–1+*Degree*=2-1+3=4. Since **4**>\|*CBC*\| we can continue.<br><br>$C(v_{22})$=3 since $v_{22}$ belongs to the colour class number 3 => We can check the backtrack pruning condition:<br>$d$–1 + $b[C(v_{22})]$ = 2-1+2 = **3** **≤**\|*CBC*\| we prune. Go up. |
| Depth 1: | 10,9,8,5,7,4,2,6,3,1 | \|*CBC*\| is 3; *Degree* = 4, since vertices belong to all colour classes except the 5-th colour class.<br>$d$–1+*Degree*=1-1+4=4. Since **4**>\|*CBC*\| we can continue. |

| | | |
|---|---|---|
| | | $C(v_{12})$=4 since $v_{12}$ belongs to the colour class number 4 => We can check the backtrack pruning condition:<br>$d$–1 + $b[C(v_{12})]$ = 1-1+3 = **3** ≤\|$CBC$\| we prune.<br>Since $d$-1=0: $b$[5] =3 (\|$CBC$\|). Go to the next iteration of the backtrack search. |
| Depth 0: | *10,9,8,5,7,4,2,6,3,1* | Since all colour classes are analysed the algorithm stops. |
| The maximum clique is {9, 8, 7}, size = 3. | | |

### 3.2.2.3.3.3 Analysis of this example

This example requires 30 main steps of the algorithm to find the maximum clique, which is also quite a good result with 10 vertices. The backtracking pruning technique worked at last steps. Potential sizes of subgraphs' maximum cliques, which are calculated by both pruning techniques, are highlighted at last steps by bold. You can see that the backtracking estimation is more accurate than the direct estimation by independent sets and it prevents the algorithm to continue searching on those subgraphs and allows stepping out.

So, the backtracking pruning technique is not an artificial technique that is always dominated by the independent sets pruning technique, but rather is another pruning way. Those techniques have to be combined in the algorithm and this produced the truly effective algorithm.

Let's now examine why the backtracking pruning estimation is more accurate than the independent sets one and what are those graph structures on which it occurs. As you probably remember we made an assumption in this example's graph construction that we have to use the greedy colouring to produce this situation. This assumption allowed us to receive the following distribution of vertices among colour classes:

**Table 5.** Values in the $b$ array - Maximum clique sizes for each colour class of a subgraph formed by vertices belonging to this and previous colour classes

| $b$ | $b[C_5]$ | $b[C_4]$ | $b[C_3]$ | $b[C_2]$ | $b[C_1]$ |
|---|---|---|---|---|---|
| V | $\{C_5, C_4, C_3, C_2, C_1\}$ | $\{C_4, C_3, C_2, C_1\}$ | $\{C_3, C_2, C_1\}$ | $\{C_2, C_1\}$ | $\{C_1\}$ |
| Value of $b[C_i]$ | 3 | 3 | 2 | 2 | 1 |

As you can see, starting from $b[C_3]$ a number of colour classes is bigger than a size of the maximum clique formed by those colour classes. Now, each time the algorithm has to analyse any subgraph formed by $\{C_3, C_2, C_1\}$ vertices on depths 1 or 0, having already found the maximum clique of size 3, it will stop and go back. It happens since the algorithm already has information that subgraphs formed by $\{C_3, C_2, C_1\}$ cannot produce a larger clique than a clique having only two vertices (in the $b$ array) - the current forming clique on those depths contains 1 or 0 vertices, so in the sum with 2 it is less that the already found clique size - 3.

So, in the previous case the size of the $b$ array value was less than a number of colour classes (independent sets) and the backtracking pruning worked (instead of the independent sets pruning) when those numbers difference was more than a size of the currently forming clique. The next steps construct one example of a graph that is hard to solve by using only independent sets in compare to using the backtracking as well:

1. Build a triangle free graph that needs a lot of colours (the more colours it needs the more complex it will be to solve by using independent sets and even more complex to solve without those);
2. Continue construction by introducing a triangle into this graph;
3. Continue by using the same principles as were used on the first step – introduce more vertices and colours, but keep the graph free from the maximum clique of size 4 (don't increase the maximum clique size).

An algorithm without the backtracking will have to do a lot of steps trying to prove that the maximum clique is found (having found the maximum clique) as long as vertices that were introduced during the first step of the construction process remain. The algorithm with backtracking will stop immediately basing on *b* values.

Note, that you cannot make this graph easily solvable by independent sets by re-colouring, since there is no colouring better than that received by the greedy algorithm.

### 3.2.3 Notes on the programming technique

An algorithm work time for NP complete tasks greatly depends on the programming techniques. All small mistakes or improper programming that might remain unmarked in a standard, usual programming, become very time consuming in the combinatorial tasks. All time leaks or unnecessary operations are executed again and again million times and can dramatically decrease algorithms performance. Therefore we introduce this subchapter containing some notes and recommendations on how to avoid improper programming of the presented algorithms.

#### 3.2.3.1 Calculating of a degree and recalculations

The new algorithms' pruning technique is based on calculating the number of remaining colour classes (independent sets) instead of just considering the number of remaining vertices. This process is done "combinatorically", i.e. for each analysed subgraph. So it is especially important to program this part correctly to avoid a sufficient decrease of algorithms speed. The number of colour classes is called a "degree" in presented algorithms.

- **Degree recalculation**: The resorting of vertices during vertex colouring can be used. We know that all vertices are grouped by colour classes and colour classes are ordered one by one. Therefore the number of colour classes should be calculated only once on each depth – the first time the algorithm enters into this depth, instead of calculating it for each subgraph. Later, as number of remaining vertices decrease, the algorithm should only adjust this number of remaining colour classes (the degree function value) by the following rule: if

the next vertex on this depth to be expanded belongs to the same colour class as the previous one then degree remains the same, otherwise it should be decreased on 1 (there are no more vertices from the previous vertex colour class, so this colour class should be eliminated from the number of colour classes).
- **Degree calculation**: The resorting of vertices can be used here again. The easiest way to calculate a degree (in an ordered set of vertices) is just to count the number of times two neighbour vertices belong to different colour classes. This method also does not require any sufficient memory use.

#### 3.2.3.2 Handling vertices and their sequence

##### *3.2.3.2.1 Vertex colouring vertices sequence*

It is advisable to use a special array to save/fix the order of vertices due re-colouring as well as for the vertices selection to each depth. The vertex re-colouring requires changing the sequence of vertices. Physical swapping of vertices' columns in an adjacency matrix has obvious minuses:
- It is very intensive process that might require a lot of time;
- Sometimes the adjacency matrix is passed by a reference and the meta-algorithm calling the algorithm finding the maximum clique could expect this matrix to remain unchanged.

Therefore the new sequence can be captured using an additional array where vertices numbers will be stored in a new sequence – in our case in the vertex re-colouring sequence as it is discussed above in algorithms' descriptions.

##### *3.2.3.2.2 Remaining vertices*

It is advisable to move through existing vertices on a depth instead of direct vertices elimination from the depth. In case of using an array for storing vertices' numbers remaining on the depth, the physical elimination will require shifting all those numbers. The number of required steps equals the number of remaining vertices. These steps again are done in the combinatorial search and therefore could mean a sufficient time loss. The moving through the array of existing vertices just needs to establish a cursor to point to a current vertex under analysis. The moving can be done just in one step, that is to re-point the cursor to the next vertex, and requires just one more memory cell on each depth to store the current position. Remaining vertices here will be all vertices starting from a vertex the cursor points to. Therefore this method of walking through the vertices array looks to be an acceptable technique to apply in described algorithms. This method is used in all examples of our work.

## *3.3 Weighted Case*

In this chapter we are going to introduce a new algorithm, explain it and bring an example for the graphs case, where vertices have different weights. Those weights can be of any value as long as we can compare these values and calculate the difference between them.

## 3.3.1 "VColor-BT-w" – An algorithm based on a vertex colouring

### 3.3.1.1 Description

The previously described algorithm called "VColor-u" is the base for the maximum-weight algorithm with the following changes. We cannot any longer determine values of the function *Degree* as a number of existing colour classes on a subgraph since vertices have different weights and a colour class' maximum weight can differ from 1. Therefore a degree of a subgraph will be calculated as a sum of maximum weights of each colour class existing on this subgraph: for each existing class we have to find a vertex of the maximum-weight and then sum up weights of those vertices.

The order of vertices here becomes even more important. Vertices should be resorted first of all by colour classes and then by weights inside each colour class. So, a vertex of the maximum weight in any colour class always will be the last inside this colour class. Therefore a degree of a subgraph equals the sum of the last vertex weights of each colour class existing on the subgraph independently of which vertices of a colour class exist on this subgraph. Moreover, instead of calculating a degree of a subgraph each time we will calculate it only first time on a depth and later only adjust by the following rule: if the next vertex on this depth to be expanded is from the same colour class as the previous one, then degree is decreased on a weight of the previous vertex and is increased on the weight of the current vertex, otherwise it should be decreased on a weight of the previous vertex (there is no more vertices from the previous vertex' colour class and the previous vertex weight was the largest in that colour class by resorting and therefore was used to calculate the degree).

Besides one more adjustment to the base algorithm will be done. We will use ideas of a backtrack search described by Östergård [Östergård 2001]. In the algorithm values of a function $c(i)$ is calculated ($i$ is a vertex number) which denotes the weight of the maximum-weight clique in the subgraph induced by the vertices $\{v_i, v_{i+1}, \ldots, v_n\}$. Obviously $c(n)$ = weight of $v_n$ and $c(1)$ is the weight of the maximum-weight clique. For each vertex starting from the last one and up to the first one a backtrack search is carried out to find $c(i)$. Those values are used to prune the search of the maximum-weight clique. As we search for a clique with a weight greater than $W$, if the total weight of the forming current clique vertices is $W'$ and we consider $v_i$ to be the next expanded vertex, then we can prune the search if $W' + c(i) \leq W$. Östergård has also advised using a vertex reordering by a vertex-colouring's colour classes [Östergård 2001, Östergård 2002], therefore the ordering for the first pruning strategy will not slow down this backtrack search.

Other steps of the algorithm remain unchanged.

*Note*: It is advisable to use a special array to solve the order of vertices to avoid changing adjacency matrix during reordering vertices.

### 3.3.1.2 Algorithm

Algorithm for the maximum – weight clique problem

*N* – number of vertices in the graph
*W* – weight of the current best (maximum-weight) clique
*d* – depth
$G_d$ – subgraph of *G* formed by vertices existing on depth *d*
*W(d)* – weight of all vertices in the forming clique
*w(i)* – weight of vertex *i*
*c – array of the backtrack search results*

**Step 0. Heuristic vertex-colouring**: Find a vertex colouring, reorder vertices and apply new vertices indexes (renumber vertices from 1 to *N* for using in the backtrack search).

```
Step 1. Backtrack search runner:
    For $n = N$ downto 1
        Go to the step 2
        $c(n) = W$
    Next
    Go to the End

Step 2. Initialization: Form the depth 1 by selecting all vertices with an index less
than $n$ and connected to the vertex $n$. $d$=1. $W(1)= w(n)$
Step 3. Check: If the current depth can contain a larger clique than already found:
    If $W(d) + Degree(G_d) \le W$ then go to the step 7.
Step 4. Expand vertex: Get the next vertex to expand.
    If all vertices have been expanded or there are no vertices then:
        Check if the current clique is the largest one. If yes then save it.
        Go to the step 7.
Step 5. Check: If the current level can contain a larger clique than already found:
    If $W(d) + c(expanding\ vertex\ index) \le W$ then go to step 7.
Step 6. The next level: Form a new depth by selecting all remaining vertices that are
connected to the expanding vertex from the current depth;
    $W(d+1)=W(d) + w(expanding\ vertex\ index)$
    $d = d + 1$;
    Go to the step 3.
Step 7. Step back:
    $d = d - 1$;
    if $d = 0$, then return to the step 1
    Delete the expanded vertex from the analysis on this depth;
    Go to the step 3.

End:    Return the maximum-weight clique.
```

## 3.3.1.3 Example

We will demonstrate here two examples. The first one will demonstrate a technique of pruning by colour classes in weighted graphs without backtracking. The second example will include both pruning strategies and therefore will follow the described algorithm.

### *3.3.1.3.1 Example 1: Pruning technique by colour classes*

#### 3.3.1.3.1.1 Description of the example graph

Consider the graph shown in **Figure 11**. and vertices' weights that are shown in the **Table 6**.

**Table 6.** „VColor-BT-w" – Example 1 / vertices' weights

| Vertex | Weight |
|---|---|
| 8, 7, 4 | 1 |
| 4, 3, 9, 5 | 3 |
| 2, 1 | 4 |
| 6 | 10 |

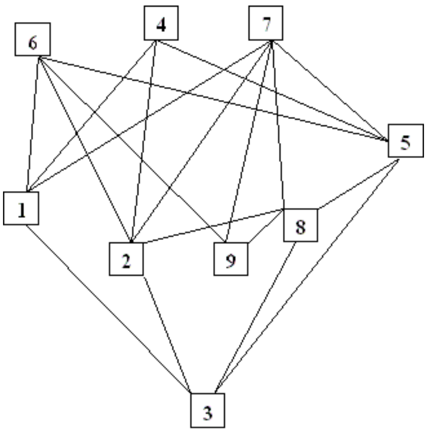

**Figure 11.** "VColor-BT-w" – Graph of the colour classes pruning example

Again it is a graph used for our examples before, which is built using the Moon-Moser type subgraph containing vertices 1, 2 and 5 for the first class and vertices 6, 4 and 7 for the second class plus vertices 3, 9 and 8, which are used to make the graph's structure a bit more complex and contain larger cliques that the Moon-Moser subgraph produces.

#### 3.3.1.3.1.2 Algorithm's steps

The first step is to determine colour classes one by one as long as uncoloured vertices exist in a greedy manner. Unlike the unweighted case, here vertices are sorted by weights inside each colour class in addition to sorting by colour classes. So, vertex colouring gives as the next result:

Colour 1 = {5, 9, 1, 2}
Colour 2 = {7, 4, 3, 6}
Colour 3 = {8};

We define a sequence by including the first colour class into the end, then the second colour class in front of it, and so forth. Besides, vertices in colour classes are listed from the biggest (by weight) up to the smallest (by weight).

Sequence of vertices: {8, 6, 3, 4, 7, 2, 1, 9, 5}

Let’s use the following notation in the example: $W$ – a weight of the current best weighted clique, $W_i$ is a weight accumulated on previous depths up to the $i$-th (incl.), $d$ is a depth number, $c$ is an array of the backtrack search results. A grey vertex in the table below is a vertex under analysis and vertices in front of that are vertices that have been already analysed and cannot participate in the forming maximum clique any longer.

Let’s say that all vertices exist on a depth one and we start the algorithm without any additional information.

**Table 7.** „VColor-BT-w“ - Example 1 / Steps of finding the maximum clique without backtracking

| **Depth** | **Subgraph** | **Step’s description** |
|---|---|---|
| 1: | 8,6,3,4,7,2,1,9,5 | *Degree*=1+10+4=15>$W$(0)<br>The grey vertex $v_{di}$ ($v_{11}$) to be expanded<br>$W_1$=1 (weight of the vertex 8) |
| 2: | 3,7,2,9,5 | *Degree*=3+4=7 +1($W_1$)> $W$(0)<br>The grey vertex $v_{di}$ ($v_{21}$) to be expanded<br>$W_2$=1($W_1$)+3(weight of the vertex 3) = 4(accumulated) |
| 3: | 2,5 | *Degree*=4+4($W_2$)> $W$(0)<br>The grey vertex $v_{di}$ ($v_{31}$) to be expanded<br>$W_3$=4+4=8 |
| 4: | Ø | *CBC*=$W_3$=8 ({8,3,2}) |
| 3: | 2, 5 | *Degree*=3 +4($W_2$)< $W$(8).<br>Go up on the previous depth |
| 2: | 3,7,2,9,5 | *Degree*=1+4=5 +1($W_1$)< $W$(8).<br>Go up on the previous depth |
| 1: | 8,6,3,4,7,2,1,9,5 | *Degree*=10+4=14 +0($W_0$)> $W$(8).<br>Go further. $W_1$=10 (weight of the vertex 6) |
| 2: | 2,1,9,5 | *Degree*=4 +10($W_1$)> $W$(8)<br>The grey vertex $v_{di}$ ($v_{21}$) to be expanded<br>$W_2$=10+4=14 |
| 3: | Ø | $W$=$W_2$=14 ({6,2}) |
| 2: | 2,1,9,5 | *Degree*=4 +10($W_1$)≤ $W$(14).<br>Go up on the previous depth. |
| 1: | 8,6,3,4,7,2,1,9,5 | *Degree*=3+4=7 +0($W_0$)< $W$(14).<br>Go up on the previous depth. Since $d$=1 (is the first depth) then goto *End* |
| The maximum weighted clique is {6,2}, and its weight is 14. | | |

### *3.3.1.3.2 Example 2: Full example*

This second example demonstrates how the previously described algorithm works. In other words, here both pruning techniques are included.

#### 3.3.1.3.2.1 Description of the example graph

Consider the graph shown in **Figure 12** and vertices' weights that are shown in the **Table 8**.

**Table 8.** „VColor-BT-w“ – Example 2 / Vertices' weights

| **Vertex** | **Weight (*w*)** |
|---|---|
| 2, 6 | 1 |
| 7 | 2 |
| 1 | 7 |
| 3, 5 | 10 |
| 4 | 11 |

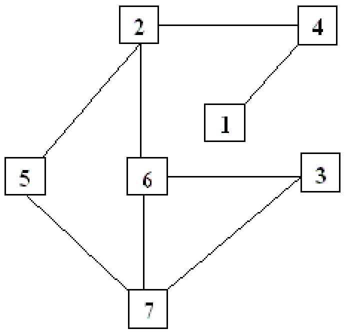

**Figure 12.** "VColor-BT-w" – Graph of the full example (example 2)

This is a graph constructed to have three colours, which is a minimum for the "VColor-BT-w" algorithm to demonstrate the backtracking technique.

#### 3.3.1.3.2.2 Algorithm's steps

The first step is to determine colour classes one by one as long as uncoloured vertices exist in a greedy manner. Unlike the unweighted case, here vertices are sorted by weights inside each colour class in addition to sorting by colour classes. So, vertex colouring gives as the next result:

Colour 1 = {2, 1, 3}
Colour 2 = {6, 5, 4}
Colour 3 = {7};

We define a sequence by including the first colour class into the end, then the second colour class in front of it, and so forth. Besides vertices in colour classes are listed from the biggest (by weight) up to the smallest (by weight).

Sequence of vertices: {7, 4, 5, 6, 3, 1, 2}

Let's use the following notation in the example: *W* – a weight of the current best weighted clique, $W_i$ is a weight accumulated on previous depths up to the *i*-th (incl), *d* is a depth number, *c* is an array of the backtrack search results. A grey vertex in the table below is a vertex under analysis and vertices in front of that are vertices that have been already analysed and cannot participate in the forming maximum clique any longer.

**Table 9.** „VColor-BT-w" - Example 2 / Steps of finding the maximum clique

| **Depth** | **Subgraph** | **Step's description** |
|---|---|---|
| 0: | 7, 4, 5, 6, 3, 1, 2 | To do: Start a backtrack search from the last vertex (which is grey) by selecting into the depth 1 all vertices that are adjacent to it among vertices after that. |
| 1: | Ø | $W_1$ = 1(weight of the vertex 2); $d$=1.<br>Check: $W_1$ + *Degree* = 1+0=1> $W$(0) go further<br>No vertices => Check if maximum: $W_1$ > $W$ => save new record: $W$=1, max weighted clique={2}.<br>$d$=$d$-1 => $d$=0 => $c$(vertex 2)=$W$=1;<br>Go to the next backtracking iteration. |
| 0: | 7, 4, 5, 6, 3, 1, 2 | To do: Start the next backtrack search from the previous vertex (which is grey) by selecting into the depth 1 all vertices that are adjacent to it among vertices after that. |
| 1: | Ø | $W_1$ = 7(weight of vertex 1); $d$=1.<br>Check: $W_1$ + *Degree* = 7+0=7> $W$(1) go further<br>No vertices => Check if maximum: $W_1$ > $W$ => save new record: $W$=7, max weighted clique={1}.<br>$d$=$d$-1 => $d$=0 => $c$(vertex 1)=$W$=7;<br>Go to the next backtracking iteration. |
| 0: | 7, 4, 5, 6, 3, 1, 2 | To do: Start the next backtrack search from the previous vertex (which is grey) by selecting into the depth 1 all vertices that are adjacent to it among vertices after that. |
| 1: | Ø | $W_1$ = 10(weight of vertex 3); $d$=1.<br>Check: $W_1$ + *Degree* = 10+0=10> $W$(7) go further<br>No vertices => Check if maximum: $W_1$ > $W$ => save new record: $W$=10, max weighted clique={3}.<br>$d$=$d$-1 => $d$=0 => $c$(vertex 3)=$W$=10;<br>Go to the next backtracking iteration. |
| 0: | 7, 4, 5, 6, 3, 1, 2 | To do: Start the next backtrack search from the previous vertex (which is grey) by selecting into the depth 1 all vertices that are adjacent to it among vertices after that. |
| 1: | 3, 2 | $W_1$ = 1(weight of vertex 6); $d$=1.<br>*Degree*: all vertices belong to the same colour class so let's use the most left vertex's weight<br>*Degree* = 10 (weight of vertex 3)<br>Check: $W_1$ + *Degree* = 1+10=11 > $W$(7) go further. |

| | | |
|---|---|---|
| | | The next vertex to expand is the vertex 3<br>Check: $W_1$+$c$(vertex 3)=1+10=11> $W$(7) go further<br>$W_2$= $W_1$+w(vertex 3)=1+10=11; $d$=$d$+1=2 |
| 2: | Ø | Check: $W_2$ + *Degree* = 11+0=11 > $W$(7) go further<br>No vertices => Check if maximum: $W_2$(11)>$W$(7) => save new record: $W$=11, max weighted clique={6,3}.<br>$d$=$d$-1 => $d$=1 => go up. |
| 1: | 3, 2 | *Degree*: all *remaining* vertices belong to the same colour class, so let's use the most left remaining vertex's *Degree* = 1 (weight of the vertex 2).<br>Check: $W_1$ + *Degree* = 1+1=2 ≤ $W$(11) go back<br>$d$=$d$-1 => $d$=0 => $c$(vertex 6)=$W$=11;<br>Go to the next backtracking iteration. |
| 0: | 7, 4, 5, 6, 3, 1, 2 | To do: Start the next backtrack search from the previous vertex (which is grey) by selecting into the depth 1 all vertices that are adjacent to it among vertices after that. |
| 1: | 2 | $W_1$ = 10(weight of vertex 5); $d$=1.<br>*Degree*: all vertices belong to the same colour class, so let's use the most left vertex's weight<br>*Degree* = 1 (weight of vertex 2)<br>Check: $W_1$ + *Degree* = 10+1=11 ≤ $W$(11) go back<br>$d$=$d$-1 => $d$=0 => $c$(vertex 5)=$W$=11;<br>Go to the next backtracking iteration. |
| 0: | 7, 4, 5, 6, 3, 1, 2 | To do: Start the next backtrack search from the previous vertex (which is grey) by selecting into the depth 1 all vertices that are adjacent to it among vertices after that. |
| 1: | 1, 2 | $W_1$ = 11(weight of vertex 4); $d$=1.<br>*Degree*: all vertices belong to the same colour class, so let's use the most left vertex's weight<br>*Degree* = 7 (weight of vertex 1)<br>Check: $W_1$ + *Degree* = 11+7=18 > $W$(11) go further.<br>The next vertex to expand is the vertex 1<br>Check: $W_1$+$c$(vertex 1)=11+7=18>$W$(11) go further.<br>$W_2$= $W_1$+w(vertex 1)=11+7=18; $d$=$d$+1=2. |
| 2: | Ø | Check: $W_2$ + *Degree* =18+0=18>$W$(11) go further. |

|  |  |  |
|---|---|---|
|  |  | No vertices => Check if maximum: $W_2(18)>W(11)$ => save new record: $W$=18, max weighted clique={4,1}. <br> $d=d$-1 => $d$=1 => go up. |
| 1: | 1,2 | *Degree*: all *remaining* vertices belong to the same colour class, so let's use the most left *remaining* vertex's *Degree* = 1 (weight of vertex 2). <br> Check: $W_1$ + *Degree* = 11+1=12 $\leq$ $W$(18) go back. <br> $d=d$-1 => $d$=0 => $c$(vertex 4)=$W$=18; <br> Go to the next backtracking iteration. |
| 0: | 7, 4, 5, 6, 3, 1, 2 | To do: Start the next backtrack search from the previous vertex (which is grey) by selecting into the depth 1 all vertices that are adjacent to it among vertices after that. |
| 1: | 5, 6, 3 | $W_1$ = 2 (weight of vertex 7); $d$=1. <br> *Degree*: vertices belong to colour classes 1 and 2, so let's use the most left vertices of those classes <br> *Degree* = 10 (weight of vertex 5) + 10 (weight of vertex 3) =20 <br> Check: $W_1$ + *Degree* = 2+20=**22 > *W*(18)** go further <br> The next vertex to expand is vertex 5 <br> Check: $W_1$+$c$(vertex 5)=2+11=**13 $\leq$ *W*(18)** go back <br> $d=d$-1 => $d$=0 => $c$(vertex 7)=$W$=18; <br> Go to the next backtracking iteration. |
| 0; | 7, 4, 5, 6, 3, 1, 2 | No more vertices to analyse, go to End. |
| The maximum weighted clique is {1, 4}, and its weight is 18. |  |  |

## 3.4 Tests and Results

### 3.4.1 Preliminary analysis

#### 3.4.1.1 General analysis

It is a natural way to use a chromatic graph number as an upper bound during the maximum clique search and it is widely used [Babel and Tinhofer 1990, Wood 1997]. The maximum clique by its definition requires a special colour for each vertex. Therefore the maximum clique size cannot be larger than a chromatic number.

There is a sandwich theorem [Knuth 1994] that is focused on a Lovasz number $\theta(\hat{G})$, which is said to be a sandwich between the minimum number of colours required (the chromatic number) and a size of the maximum clique:

$$w(G) \leq \theta(\hat{G}) \leq \chi(G),$$

where $\hat{G}$ is the complement to $G$ graph. The Lovasz number could be calculated in a polynomial time [Bomze et al. 1999]. First of all this theorem shows that chromatic number is always larger that the maximum clique size. Besides it demonstrates that there could be some distance between those numbers. A size of this distance is a core element defining efficiency of the proposed algorithms. The smaller this distance, the faster algorithms are. Fortunately, in comparison to other (best) well-known algorithms, independently of this distance, the pruning formulas based on the vertex-colouring are able to produce a faster solution than other algorithms, especially on dense graphs, where it is practically the only pruning technique that keeps working. The smaller density of a graph, the more depending on this distance become the algorithms in comparison to others. This dependency is explained by considering a question: if we already have found the maximum clique, then how fast could we prove somehow that it is the maximum one? This situation will be described in the "Incomplete solution" chapter. The main thing that makes our problem so hard is not the problem of finding the maximum clique but the problem of proving that it is the maximum. The closer a size of the maximum clique to a chromatic number, the more efficiently the algorithms prune and the faster it is possible to prove that the found clique is the maximum one.

#### 3.4.1.2 Graphs "easy" to solve

As it was shown before, the "best" graphs to be solved by the new algorithms are graphs where the chromatic number is close to the maximum clique size. There are a sufficient number of graphs' classes where it is true. The most interesting example of such graphs, if we consider the hardness to solve it by other algorithms, is a graph with a lot of semi-parallel structures like Moon-Moser graphs. The more such structures are there, the easier this graph is to solve by the introduced algorithms in comparison to other algorithms since the complexity of such structures are eliminated by using the vertex-colouring. It happens because the vertex-colouring degree function could produce a closer estimation for a potential clique size in such (sub)graphs than other techniques.

#### 3.4.1.3 Graphs “hard” to solve

Unfortunately there are much more graphs / graph classes where the chromatic number sufficiently differs from the maximum clique size [West 2001]. It is even possible to construct a graph with the chromatic number as big as you want while the maximum clique size remains the same. See for example the “Mycielski’s construction” [West 2001] that does it. Such graphs should be “hard” to solve since the new pruning technique that has been invited in this thesis will not help to avoid producing again the combinatorial branch and bound search although the situation should be still better than the pruning strategy invited by Carraghan and Pardalos [Carraghan and Pardalos 1990a]. Unfortunately this difference will not be enough for a sufficient change in a time needed to find the maximum clique. Another property of a graph “hard” to solve in addition to the previous one is to have as less parallel structure as possible. Otherwise vertices producing a large chromatic number would be eliminated during first steps and a graph will degenerate to the “easy” to solve case. Such example has been shown as the second example of the “VColor-u” algorithm.

### 3.4.2 Unweighted case

In this section results showing efficiency of the new algorithms will be presented.

As it has been mentioned earlier a very simple and effective algorithm for the maximum algorithm problem proposed by Carraghan and Pardalos [Carraghan and Pardalos 1990a] was used as a benchmark in the Second DIMACS Implementation Challenge [Johnson and Trick 1996]. Besides, using of this algorithm as a benchmark is advised in one of the DIMACS annual reports [DIMACS 1999]. That’s why it will be also used in the benchmarking below. Moreover, the proposed algorithms are nothing else than modifications of the Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990a] (we will call it a base algorithm). It gives us possibility to conclude that worse cases of new algorithms will not differ too much from worse cases of the base algorithm and comparing those algorithms on random graphs will be good enough to receive an overall picture. Later different graph classes will be checked as well to receive more precise picture by graph classes. We are going to use DIMACS graphs for that.

We have chosen one more algorithm proposed by Östergård [Östergård 2002] to participate in the comparison test since this algorithm is reported to be the quickest at the moment and this algorithm is also another modification of Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990a]. Moreover, our “VColor-BT-u” algorithm was based on his ideas.

Results are presented as ratios of algorithms spent times on finding the maximum clique – so the same results can be reproduced on any platforms. The compared algorithms were programmed using the same programming language and the same programming technique (it was possible since Östergård [Östergård 2002] algorithm and the new algorithms are just modifications of Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm). The greedy algorithm was used to find a vertex-colouring.

### 3.4.2.1 Random graphs

#### *3.4.2.1.1 Introduction*

First of all we look at randomly generated graphs. This will give us a general picture of the algorithms speed characteristic and it will be possible to conclude if those algorithms are worth to use.

The number of vertices for each density is chosen so, that any algorithm's work time is no less than 2-3 seconds and is no more than an hour. This enables us to eliminate a standard error of a function measuring time, which is approximately 0.01 seconds – less than 1%.

For each table's entry 100 graphs were generated and each generated graph is used as an input for each algorithm to be compared.

#### *3.4.2.1.2 Graphs generation model*

A meta-algorithm was used to test different algorithms. It includes a graph generation subtask and a subtask that runs algorithms to be researched and measures a spent time. The graph generation subtask uses a density and a number of vertices that a graph should have as an input. The next generation procedure, which is written in "Basic", was used:

```
ReDim arr(1 To Vertices, 1 To Vertices)
Randomize
For i = 1 To Vertices * (Vertices - 1) / 100 * Density / 2
    Do
        j = Int((Vertices * Rnd) + 1)
        k = Int((Vertices * Rnd) + 1)
    Loop Until arr(k, j) = False And j <> k
    arr(j, k) = True
    arr(k, j) = True
Next
```

where:
*Vertices* is the number of vertices to be generated,
*Density* is the density a graph should have (for example, if the density should be 70%, then this parameter should be 70),
*arr* is a Boolean array representing a generated graph's adjacency matrix, i.e. for any vertices *j*, *k* - *arr*(*j*, *k*) is true if and only if those vertices are connected.

As it is easy to see, main functions that provide random numbers are *Rnd* and *Randomize*. Both functions are native Microsoft random numbers' generators. Let's examine them more:

*Randomize* – a function that is used to initialise or reseed a sequence returned by *Rnd* using a value returned by the system timer as a new seed value.

*Rnd* – a function that returns random single-precision numbers between 0.000000 and 1.000000. It uses the linear-congruential method for random-number generation. The following pseudo code documents the algorithm used:

$$x_1 = ( x_0 * a + c ) \text{ MOD } (2^{24})$$

where:

$x_1$ is a new value,

$x_0$ is a previous value (for the first iteration it is called an initial value or seed),

$a$ equals 1140671485,

$c$ equals 12820163,

MOD is an operator that returns the integer remainder after an integer division. [http://support.microsoft.com/default.aspx?scid=kb;en-us;231847, 2005-08-29].

This random numbers generation algorithm produces numbers that are surprisingly good quality. It is used even for high security cryptography, although some newer modules can provide a higher security. The integer constant, $a$, is approximately factor 2 less than the square root of the maximum integer value (represented by $2^{24}$) [Bennett 1976]. It is not recommended to change the values of $a$ and $c$ without careful study and experimentation, since choosing the "wrong" values for $a$ and $c$ may compromise the pseudorandom characteristics. Often the choice of those values is "more of an art than a science", although this can be accomplished via specific algorithms [Schildt 1987].

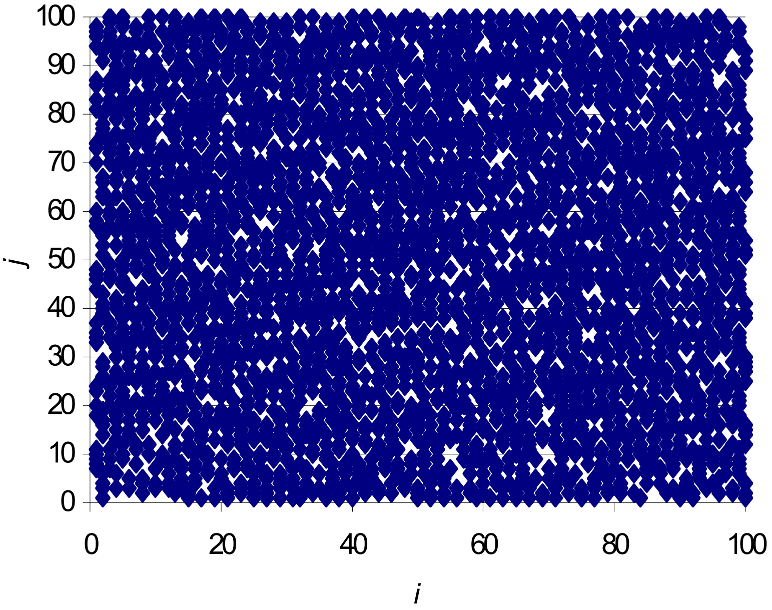

**Figure 13.** Distribution of randomly generated coordinates (adjacency matrix)

The above graph shows a randomly generated adjacency matrix to illustrate how the above algorithm generates graphs. A graph having 100 vertices and 60% density was produced. The *x*-axis here contains the first array dimension, the *y*-axis the second one and a dot is presented if there is an edge between vertices. As you see, numbers are distributed quite equally through the plot area and don't tend to be grouping in some areas producing special graphs. Two examples of graphs having 10% density and 100 vertices are shown below.

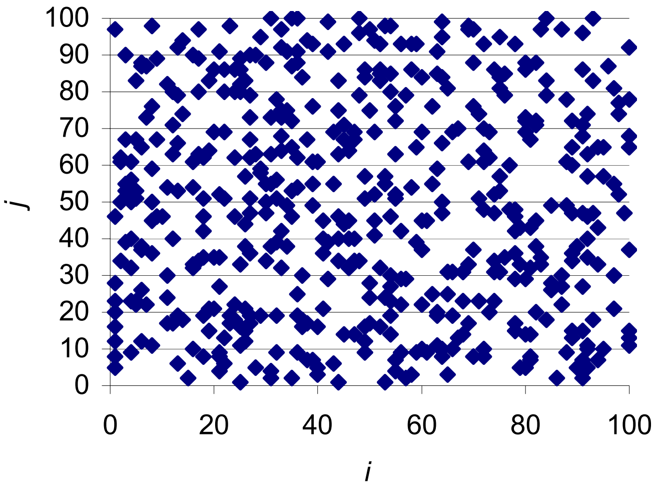

**Figure 14.** Example 1 of edges of a graph having 10% density and 100 vertices (adjacency matrix)

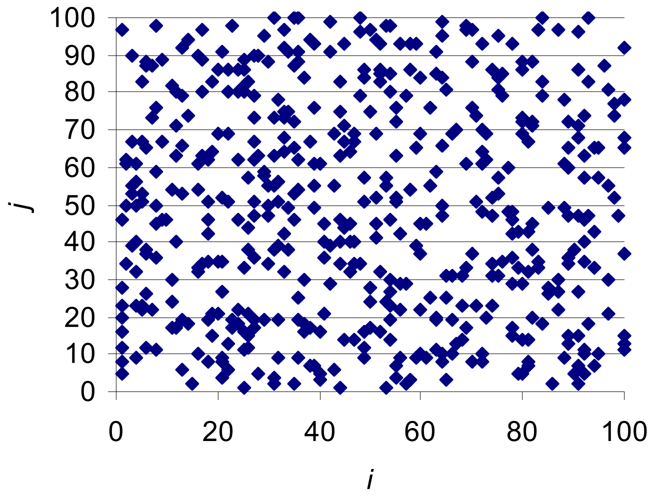

**Figure 15.** Example 2 of edges of a graph having 10% density and 100 vertices (adjacency matrix)

Those examples demonstrate that our graph generation function produce different graphs, therefore running the meta-algorithm enough times should test algorithm on sufficiently different graphs and give acceptable average to make a general conclusion on algorithms efficiency.

#### *3.4.2.1.3 Time spent on the maximum clique finding*

Here a time spent on finding the maximum clique is presented for all algorithms, which participate in tests.

*PO* – time needed to find the maximum clique by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm divided by time needed to find the maximum clique by Östergård [Östergård 2002] algorithm.
*VColor-u* – time needed to find the maximum clique by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm divided by time needed to find the maximum clique by the "VColor-u" algorithm.
*VColor-BT-u* – time needed to find the maximum clique by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm divided by time needed to find the maximum clique by the "VColor-BT-u" algorithm.

Note that the density parameter is shown first of all and only then the number of vertices since the second parameter depends on the first one as we stated earlier – the number of vertices is chosen so, that the time spent on finding the maximum clique for a corresponding density is no less than 2-3 seconds and also is not too big (in our experiments no more than 1 hour). That's why the lower density is, the more vertices are in use.

**Table 10.** Unweighted case / Benchmark results at random graphs – General view – average ratios of time spent on the maximum clique finding / the base algorithm's time divided by a corresponding algorithm's time

| Edge density | Vertices | *PO* | VColor-u | VColor-BT-u |
|---|---|---|---|---|
| 0.1 | 1800 | 0.8 | 0.8 | 1.0 |
| 0.2 | 1200 | 1.0 | 1.0 | 1.5 |
| 0.3 | 750 | 1.1 | 1.0 | 1.6 |
| 0.4 | 500 | 1.1 | 1.1 | 1.8 |
| 0.5 | 300 | 1.2 | 1.3 | 2.3 |
| 0.6 | 200 | 1.3 | 1.7 | 3.5 |
| 0.7 | 150 | 1.5 | 2.5 | 5.6 |
| 0.8 | 120 | 1.9 | 8.5 | 16.2 |
| 0.9 | 100 | 4.2 | 50.1 | 102.1 |

For example, 8.5 in the column marked *VColor-u* means that Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm requires 8.5 times more time to find the maximum clique than the "VColor-u" algorithm.

It is easy to see that all algorithms are faster than the base algorithm on densities more than 10%. The base algorithm is better on very small densities – 10% than others except "VColor-BT-u". The "VColor-u" algorithm is slightly slower than Östergård [Östergård 2002] algorithm on densities up to 40%, but starting from 50% it becomes better. The “VColor-BT-u” algorithm that inherits strong sides from both of those algorithms is the best one on all densities except 10% and smaller where it is approximately the same as the base one. The biggest speed difference is reached on dense graphs, where algorithms based on colour classes are 50-100 times faster than the base algorithm and 13-25 times faster than Östergård algorithm. The reason of that lies in a fact that on those densities the “colour classes” pruning technique still works while other algorithms’ pruning techniques don’t practically prune branches of the search tree.

Now it looks to be interesting to see more details of the algorithms’ spent times and see extreme cases – minimum and maximum speed differences, i.e. the difference in time spent on analysing randomly occurred “good”/“bad” cases.

**Table 11.** Unweighted case / Benchmark results at random graphs – Detailed view – ratios deviation of the time spent on the maximum clique finding / the base algorithm’s time divided by a corresponding algorithm’s time

| Edge density | Vertices | Type of measure | *PO* | VColor-u | VColor-BT-u |
|---|---|---|---|---|---|
| 0.1 | 1800 | minimum | 0.7 | 0.7 | 0.8 |
| | | average | 0.8 | 0.8 | 1.0 |
| | | maximum | 0.8 | 0.8 | 1.1 |
| 0.2 | 1200 | minimum | 0.9 | 1.0 | 1.0 |
| | | average | 1.0 | 1.0 | 1.5 |
| | | maximum | 1.1 | 1.0 | 2.0 |
| 0.3 | 750 | minimum | 0.9 | 1.0 | 1.1 |
| | | average | 1.1 | 1.0 | 1.6 |
| | | maximum | 1.3 | 1.2 | 2.6 |
| 0.4 | 500 | minimum | 0.9 | 1.0 | 1.2 |
| | | average | 1.1 | 1.1 | 1.8 |
| | | maximum | 1.4 | 1.2 | 2.9 |
| 0.5 | 300 | minimum | 0.9 | 1.2 | 1.2 |
| | | average | 1.2 | 1.3 | 2.3 |
| | | maximum | 1.5 | 1.5 | 3.5 |
| 0.6 | 200 | minimum | 0.9 | 1.4 | 1.8 |
| | | average | 1.3 | 1.7 | 3.5 |
| | | maximum | 2.0 | 2.1 | 5.0 |
| 0.7 | 150 | minimum | 1.0 | 1.7 | 3.0 |
| | | average | 1.5 | 2.5 | 5.6 |
| | | maximum | 2.6 | 4.2 | 11.0 |
| 0.8 | 120 | minimum | 1.0 | 2.5 | 4.4 |
| | | average | 1.9 | 8.5 | 16.2 |
| | | maximum | 4.5 | 12.2 | 29.0 |
| 0.9 | 100 | minimum | 1.4 | 14.9 | 25.2 |
| | | average | 4.2 | 50.1 | 102.1 |
| | | maximum | 8.9 | 88.9 | 287.0 |

The picture is practically the same as for averages. The “VColor-u” algorithm is never slower that Carraghan and Pardalos algorithm starting from 20% density and competes with Östergård [Östergård 2002] algorithm up to 40-50% starting from which it becomes also better. The “VColor-BT-u” algorithm surely wins all cases for all densities except densities smaller or equal to 10%, where is competes with the base algorithm. We are not presenting results of each our experiment here, but we should note that in all experiments this “VColor-BT-u” algorithm was never slower than either the “VColor-u” or Östergård algorithms.

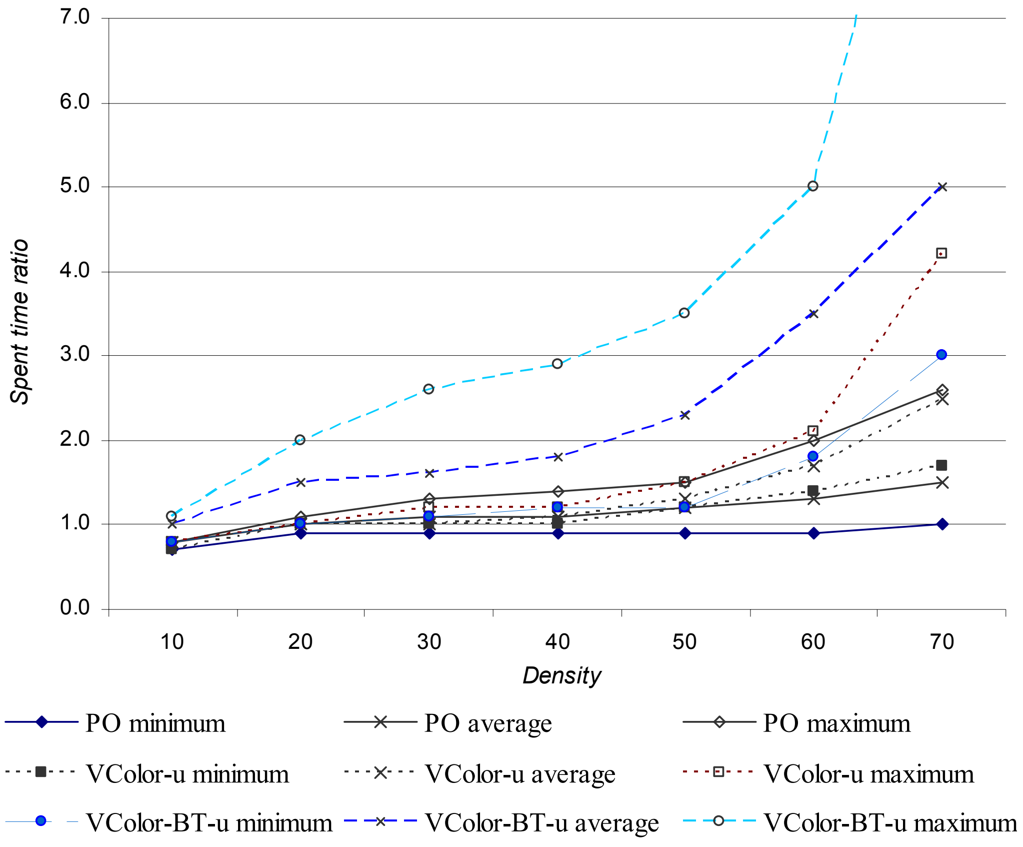

**Figure 16.** Unweighted case / Benchmark results at random graphs – Detailed view

This graph view containing results from the previous table by densities up to 70% provides most benefits for a graphical understanding of algorithms’ spent time ratios behaviour. Results on higher densities are easily extrapolateable from this graph, and only make presentation of results on low densities unclear – therefore those are omitted.

The only algorithm, which is better than the base algorithm by the average and the maximum ratio of time spent on the maximum clique finding is the “VColor-BT-u” one. It means that this is the only algorithm, which is quicker than the best one by those ratios on any density. The minimum ratio for that algorithm is more than 1 only starting from the 20% density, so the base algorithm is still sometimes better on low densities than the “VColor-BT-u” one. Another interesting part here are high densities, where even the minimum ratio for the “VColor-BT-u” algorithm more than the maximum ratio of Östergård [Östergård 2002] algorithm, so the first one is always quicker than the Östergård algorithm, and of course than the base Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990a]. The “VColor-u” algorithm is not so good as the “VColor-BT-u” and is slower than Östergård’s algorithm up to high densities where it starts to perform much better and overcomes Östergård’s algorithm but do not reach the performance of the “VColor-BT-u” one.

#### *3.4.2.1.4 Vertex colouring*

Accordingly to the first step of our algorithms we should mark that the problem of finding an efficient vertex colouring can be treated as a separate problem. The problem of colouring a graph by the minimum number of colours (i.e. pure vertex colouring problem) is an NP-hard task; therefore we had to use a heuristic algorithm to do this. The heuristic algorithm is an algorithm that:

- Doesn’t guarantee the best result, but finds a result that is close enough to the best one;
- Is quicker than an exact algorithm.

In our case we use a polynomial heuristic – a result is found in a polynomial time.

The vertex-colouring step affects the overall result in the following ways:

1. The closer number of colour classes to the size of the maximum clique, the quicker the maximum clique will be found because of more effective pruning;
2. The more time we spent on vertex colouring, the slower our algorithm works in general (since the vertex colouring is a subroutine, which is included into the main algorithm and its time should be taken into account).

Note that the presented algorithms can evolve without changing core steps by inventing a new and more effective heuristic algorithm for the vertex colouring.

**Table 12.** Number of colour classes by a greedy vertex colouring

| Edge density | Vertices | Average size of the maximum clique | Number of colour classes | Number of colour classes containing only 1 vertex |
|---|---|---|---|---|
| 0.1 | 100 | 3.88 | 7.16 | 0.40 |
| 0.2 | 100 | 5.08 | 10.36 | 0.48 |
| 0.3 | 100 | 6.52 | 13.88 | 0.64 |
| 0.4 | 100 | 8.24 | 17.20 | 0.92 |
| 0.5 | 100 | 10.44 | 20.76 | 1.12 |
| 0.6 | 100 | 13.60 | 24.80 | 1.56 |
| 0.7 | 100 | 18.00 | 30.00 | 1.76 |
| 0.8 | 100 | 24.04 | 37.24 | 3.16 |
| 0.9 | 100 | 34.36 | 46.08 | 4.80 |
| 0.99 | 100 | 69.56 | 71.20 | 42.48 |

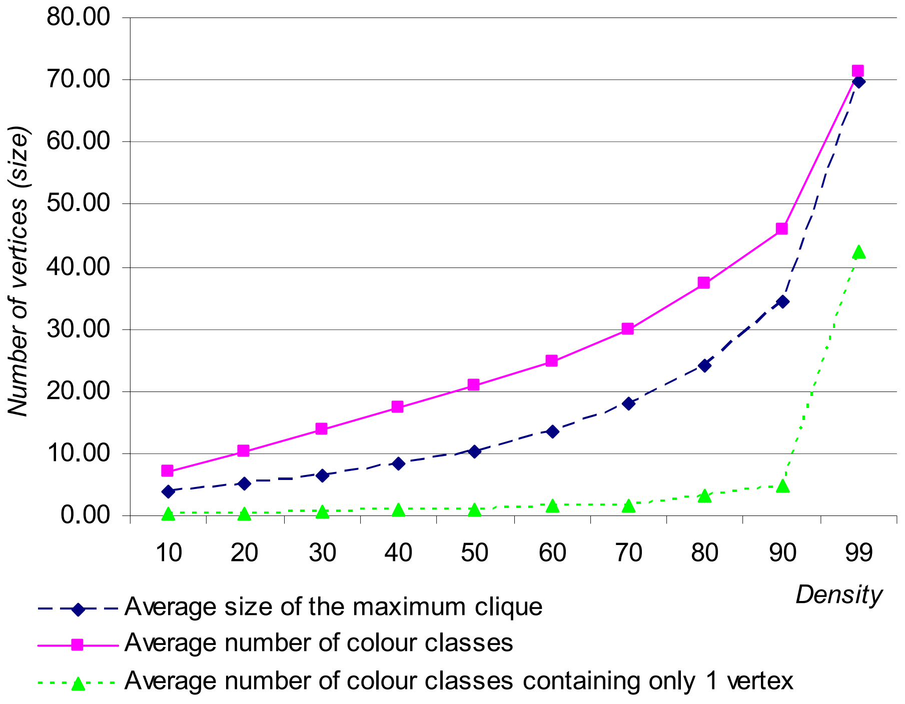

**Figure 17.** Number of colour classes by a greedy vertex colouring (for a graph with 100 vertices)

It is easy to see that quite effective results of the new algorithms were reached not on the best splitting vertices into colour classes – an average number of colour classes is larger approximately on 50% (25% on dense graphs) than an average size of the maximum clique. This difference can be explained easily by the fact that we used a very simple greedy heuristic to find a graph's vertices colouring. From another point of view there are a lot of graphs, where a number of colours will always be far from the maximum clique size [West 2001]. So, we cannot expect that the colours' number will be very close to the maximum clique size and therefore there is no point to use an expensive heuristic if it will not provide us with a sufficient decrease of a number of colour classes. Therefore, it looks like the greedy heuristic is a reasonable choice for the moment to use in our algorithms.

### *3.4.2.1.5 Number of analysed branches / subgraphs*

Another important characteristic that helps us to understand the invented algorithms' work efficiency is a number of analysed branches. This characteristic can also be seen as an efficiency factor of pruning techniques – how many branches one or another technique (set of techniques) was able to prune – the less the number of analysed branches is, the more branches were pruned. This factor could demonstrate why one or another algorithm is faster and when it becomes faster than others. All algorithms that we are comparing are nothing else than “branch and bounds” algorithms with different modifications, i.e. algorithms that are searching through all branches. That’s why the fewer branches we have to go through, the better an algorithm could be and therefore the factor is so important.

A ratio of analysed branches is used again since we test on randomly generated graphs and a particular number of branches is less important than information on how more/less branches were analysed by one or another algorithm.

*PO* – the number of branches analysed by Östergård [Östergård 2002] algorithm divided by the number of branches analysed by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm.
*VColor-u* – the number of branches analysed by the “VColor-u” algorithm divided by the number of branches analysed by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm.
*VColor-BT-u* – the number of branches analysed by the “VColor-BT-u” algorithm divided by the number of branches analysed by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm.

*Note that this branches’ ratio is calculated by dividing a target algorithm analysed branches number on the analysed branches number of Carraghan and Pardalos algorithm unlike previously used time spent ratios.*

**Table 13.** Unweighted case / Benchmark results at random graphs – Analysed branches ratios / a corresponding algorithm’s number of branches divided the base algorithm’s number of branches

| Edge density | Vertices | *PO* | *VColor-u* | *VColor-BT-u* |
|---|---|---|---|---|
| 0.1 | 1000 | 98% | 83% | 59% |
| 0.2 | 800 | 96% | 81% | 55% |
| 0.3 | 500 | 96% | 75% | 50% |
| 0.4 | 500 | 94% | 67% | 44% |
| 0.5 | 300 | 90% | 55% | 36% |
| 0.6 | 200 | 88% | 42% | 27% |
| 0.7 | 100 | 75% | 26% | 16% |
| 0.8 | 100 | 62% | 13% | 6% |
| 0.9 | 100 | 35% | 2% | >1% |

For example, 55% in the column marked *VColor-u* means that the “VColor-u“ algorithm has analysed 55% of branches analysed by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm.

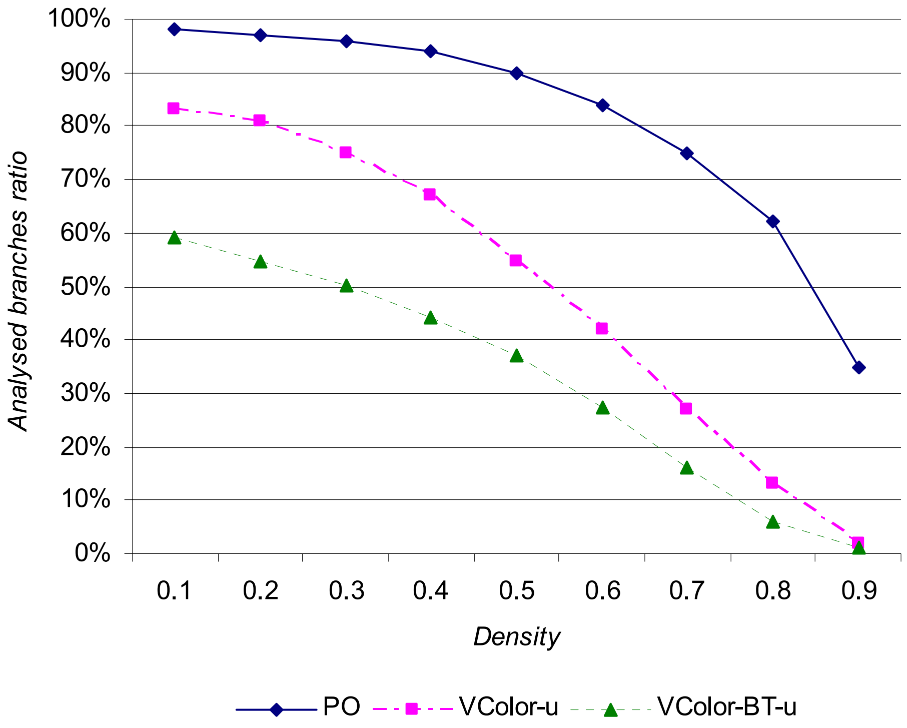

**Figure 18.** Unweighted case / Benchmark results at random graphs – Analysed branches ratios

All algorithms analyse fewer branches than the base algorithm [Carraghan and Pardalos 1990a]. The higher the density is, the fewer branches are analysed. The best algorithm from the fewer branches point of view is the "V-Color-BT-u" algorithm on all densities and the difference is sufficient! The number of analysed branches is almost twice smaller on low densities than for the base algorithm or the "PO" one and is almost 30–100 smaller on high densities. The "V-Color-u" algorithm is not very efficient on low or average densities, but is starting to be closer and closer on high densities to the "V-Color-BT-u" algorithm. The diagram shows that the combination of the "V-Color-u" and the "PO" pruning strategies inside the "V-Color-BT-u" algorithm leads to very good results. Those strategies do not compete, but rather support one another (in different situation one or another works), although additional tests have shown that the "V-Color-u" strategy works more often. The same can be seen on the diagram – the "V-Color-u" number of analysed branches is always smaller than for the "PO" algorithm – approximately 2–3 times. All this allows saying that a suggestion we had before this test, that the "PO" pruning strategy will never work as it will be covered by the "V-Color-u" pruning strategy, was completely wrong and those strategies can be used simultaneously.

### 3.4.2.1.6 Conclusion

Below we will concentrate into one graph view the ratios of time spent on the maximum clique finding and analysed branches numbers' ratios for algorithms, which are under analysis.

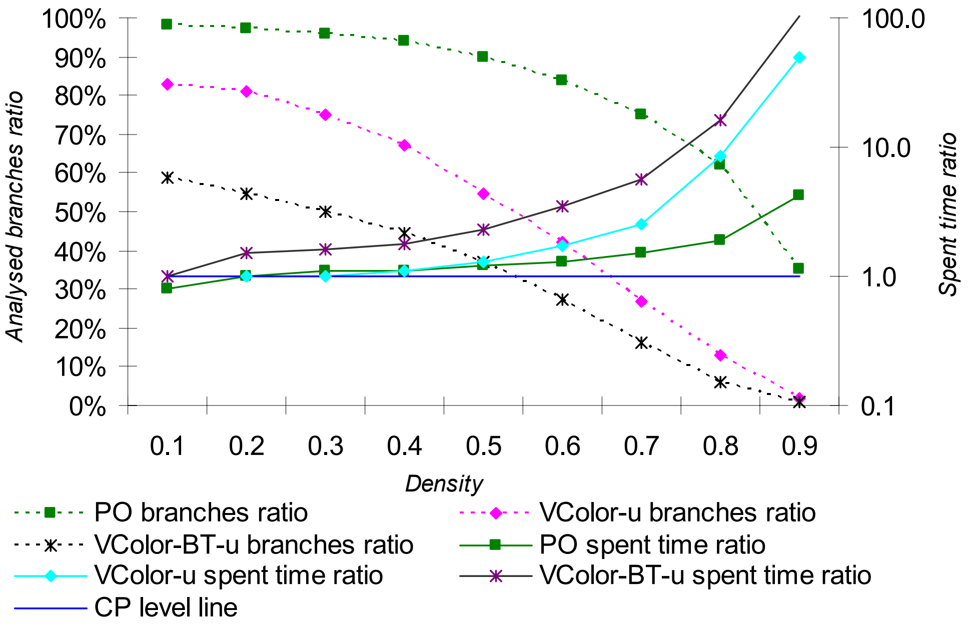

**Figure 19.** Concluding view on unweighted maximum clique finding algorithms' performance on random graphs

An additional line is added – a "CP" level line, i.e. "Carraghan and Pardalos" line. A level of this line is always 1, since the ratio of spent time by the "Carraghan and Pardalos" algorithm [Carraghan and Pardalos 1990a] divided by the same number is always 1. This line allows seeing if other algorithms are faster or slower than the base algorithm and how big is this difference. The right *y*-axis is logarithmic to fit all data on the graph.

It is easy to see that the fewer branches should be analyzed by an algorithm, the faster it is. An algorithm starts to be much faster (i.e. its time spent ratio line starts to grow very fast) as soon as fewer than approximately 40% of branches should be analysed. All algorithms are faster than the base one on average density and dense graphs, while the "V-Color-BT-u" algorithm is the best one. Algorithms are much faster than the base one on the dense graphs, where their pruning strategies are starting to be especially efficient. The "V-Color-BT-u" is faster than the base one in hundred times. The "V-Color-u" algorithm and the "PO" algorithm compete on average graphs, where those are approximately identical from the spent time point of view. The "V-Color-u" starts to be faster from the 70% density and move quickly to the "V-Color-BT-u" direction, while the "PO" algorithm's time spent ratio line grows slower. Another interesting part of this diagram, that should be noted, locates at very low densities. Here the base algorithm is very efficient. It is practically as good on densities

10%–20% as others and is the best on densities less than 10% (with the “V-Color-BT-u” algorithm). Sometimes it is even better than the “V-Color-BT-u” algorithm, which analyses 1.5–2 times less branches. So, it means that for very low densities the easiest base algorithm is the most efficient and additional steps that all remaining algorithms do to decrease number of branches to be analysed is rather unnecessary waste of time than valuable features.

The general conclusion for random graphs is:

- If the density is less than 10% then the best algorithm to use is the base one [Carraghan and Pardalos 1990a];
- If the density is higher, then the best choice is the “V-Color-BT-u” algorithm, which is especially efficient on average and high densities.

### 3.4.2.2 DIMACS graphs

Here the same algorithms are analysed on DIMACS graphs, which are a special package of graphs used in the Second DIMACS Implementation Challenge [DIMACS 1999; Johnson and Trick 1996] to test different algorithms and find out which of them is the best one on one or another type of graphs.

*PO* – time needed to find the maximum clique by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm divided by time needed to find the maximum clique by Östergård [Östergård 2002] algorithm.
*VColor-u* – time needed to find the maximum clique by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm divided by time needed to find the maximum clique by the "VColor-u" algorithm.
*VColor-BT-u* – time needed to find the maximum clique by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm divided by time needed to find the maximum clique by the "VColor-BT-u" algorithm

**Table 14.** Unweighted case / Benchmark results at DIMACS graphs – ratios of time spent on the maximum clique finding / the base algorithm's time divided by a corresponding algorithm's time

| Graph name | Edge density | Vertices | Maximum clique size | *PO* | *VColor-u* | *VColor-BT-u* |
|---|---|---|---|---|---|---|
| brock200_1 | 75% | 200 | 21 | 2.1 | 2.5 | 8.4 |
| brock200_2 | 50% | 200 | 12 | 2.3 | 1.0 | 4.0 |
| brock200_3 | 61% | 200 | 15 | 1.2 | 1.6 | 3.2 |
| brock200_4 | 66% | 200 | 17 | 2.0 | 1.6 | 6.0 |
| c-fat200-5 | 43% | 200 | 58 | 58.2 | 91.4 | 49.2 |
| c-fat500-1 | 4% | 500 | 14 | 0.7 | 1.0 | 0.8 |
| c-fat500-2 | 7% | 500 | 26 | 1.2 | 2.2 | 2.2 |
| c-fat500-5 | 19% | 500 | 64 | 72.1 | 185.0 | 85.4 |
| Hamming6-2 | 90% | 64 | 32 | 493.0 | 4 930.0 | 493.0 |
| Hamming8-4 | 64% | 256 | 16 | 247.8 | 7.9 | 7848.3 |
| johnson8-4-4 | 77% | 70 | 14 | 11.9 | 32.0 | 53.3 |
| johnson16-2-4 | 76% | 120 | 8 | 4.4 | 21.6 | 20.9 |
| keller4 | 65% | 171 | 11 | 2.8 | 4.2 | 11.8 |
| MANN_a9 | 93% | 45 | 16 | 12.5 | 42 400.0 | 42 400.0 |
| p_hat300-1 | 24% | 300 | 8 | 1.0 | 1.0 | 1.3 |
| p_hat300-2 | 49% | 300 | 25 | 2.0 | 0.8 | 6.6 |
| p_hat500_1 | 25% | 500 | 9 | 0.9 | 0.8 | 1.5 |
| p_hat700_1 | 25% | 700 | 11 | 1.1 | 0.8 | 1.9 |
| sanr400_0.7 | 70% | 400 | 21 | 1.7 | 2.5 | 5.6 |
| 2dc.256* | 47% | 256 | 7 | 4.6 | 12.5 | 14.5 |

* - An original task for those graphs is to find the maximum independent set, so the maximum clique is found from the complement graph.

*Note: An advantage of using spent time ratios – independency from a platform – was fully used in applying algorithms to DIMACS graphs. Graph instances are very different and some of them can be solved on the platform we used as a standard in less than 0.01 seconds. Therefore an older and slower computer was used sometimes. Anyway, the ratio remains the same and it is not important to mark where and which platform was used.*

Some graph instances were too hard for solving by Carraghan and Pardalos algorithm. Other algorithms were more than 10 000 times faster on those instances. Therefore, we have isolated them into another table presented below, where we calculated spent time ratios using Östergård algorithm as the base one.

*VColor-u* – time needed to find the maximum clique by Östergård [Östergård 2002] algorithm divided by time needed to find the maximum clique by the "VColor-u" algorithm.
*VColor-BT-u* – time needed to find the maximum clique by Östergård [Östergård 2002] algorithm divided by time needed to find the maximum clique by the "VColor-BT-u" algorithm

**Table 15.** Unweighted case / Benchmark results at DIMACS graphs – ratios of time spent on the maximum clique finding / the base algorithm's time divided by a corresponding algorithm's time / hard cases

| Graph name | Edge density | Vertices | Maximum clique size | *VColor-u* | *VColor-BT-u* |
|---|---|---|---|---|---|
| c-fat500-10 | 50% | 200 | 12 | 6.9 | 0.9 |
| Hamming10-2 | 99% | 1024 | 512 | 389.5 | 6.1 |
| san400_0.5_1 | 50% | 400 | 13 | 284.3 | 958.2 |
| | | | | | |
| 2dc.512* | 58% | 512 | 11 | 12.5 | 14.5 |

* - An original task for those graphs is to find the maximum independent set, so the maximum clique is found from the complement graph.

The following table provides a brief description of the used graphs:

**Table 16.** Unweighted case / Description of DIMACS graphs

| Graph type | Description |
|---|---|
| Bro | Instances from Mark Brockington and Joe Culberson's generator that attempts to "hide" cliques in a graph where the expected clique size is much smaller. For more instances, see their generator in graph/contributed/brockington. From Mark Brockington brock@cs.ualberta.ca. |
| CFat | Problems based on fault diagnosis problems [Berman and Pelc 1990]. For more instances, see the generator in graph/contributed/pardalos. From Panos Pardalos pardalos@math.uflorida.edu. |

| | |
|---|---|
| Joh* | Problems based on problem in coding theory. A Johnson graph with parameters *n*, *w*, *d* has a node for every binary vector of length *n* with exactly *w* 1s. Two vertices are adjacent if and only if their hamming distance is at least *d*. A clique then represents a feasible set of vectors for a code. For more instances, see the generator in graph/contributed/Pardalos. From Panos Pardalos pardalos@math.uflorida.edu. |
| Ham* | Another coding theory problem. A Hamming graph with parameters *n* and *d* has a node for each binary vector of length *n*. Two nodes are adjacent if and only if the corresponding bit vectors are hamming distance at least *d* apart. For more instances, see the generator in graph/contributed/pardalos. It has been noted by participants that *n*--2 graphs have a maximum clique of size $2^{n-1}$. For a proof of this, see the note in graph/contributed/bourjolly/hamming. From Panos Pardalos pardalos@math.uflorida.edu. |
| Kel* | Problems based on Keller's conjecture on tilings using hypercubes [Lagarias and Shor 1992]. For more instances (though they get very large very fast) see either the generator in graph/contributed/shor or the generator in graph/contributed/pardalos. From Peter Shor shor@research.att.com |
| MANN* (Stein) | Clique formulation of the set covering formulation of the Steiner Triple Problem. Created using Mannino's code to convert set covering problems to clique problems. From Carlo Mannino mannino@iasi.rm.cnr.it |
| PHat* | Random problems generated with the *p hat* generator which is a generalization of the classical uniform random graph generator. Uses 3 parameters: *n*, the number of nodes, and *a* and *b*, two density parameters verifying $0 \leq a \leq b \leq 1$. Generates problem instances having wider node degree spread and larger clique sizes [Gendreau et al. 1993]. From Patrick Soriano and Michel Gendreau patrick@crt.umontreal.ca. |
| San* | Instances based on Sanchis paper [Sanchis 1992] From Laura Sanchis laura@cs.colgate.edu |
| SanR* | These are random instances with sizes similar to those in *San*. From Laura Sanchis laura@cs.colgate.edu |
| 2dc* | Graphs From Two-Deletion-Correcting Codes. From N. J. A. Sloane njas@research.att.com |

The first result we see in previous result tables is that either the "VColor-u" or the "VColor-BT-u" algorithm is the quickest one in all graph instances. The only instances where the "VColor-u" is the best one are "CFat" type graphs and "hamming6-2", "hamming10-2", which are Hamming graphs having a high density. This occurs because the backtracking pruning technique is decreasing performance of the applying colour classes in these instances.

Another interesting result is an extremely good performance of new algorithms for MANN and hamming type graphs. Those graphs are again graphs of a high density, where the "VColor-BT-u" is the best for a bit lower density and the "VColor-u" for very high densities.

Note that ratios numbers (i.e. proportions of times spent on finding the maximum clique) remain the same for “2dc” type graphs – for both 256 and 512 vertices cases.

### 3.4.3 Weighted case

Usually two types of test cases are used: randomly generated graphs and fixed instances like the DIMACS test graphs. Unfortunately for the later type such instances are lacking for the maximum-weight clique problem. The DIMACS graphs are not weighted and cannot be therefore used for our testing. That’s why only random graphs are tested.

#### 3.4.3.1 Graphs generation model

The graph (vertices and edges) generation model remains the same as for the unweighted case. The only difference here is weights that we should also have. We could simplify the task by using only integer numbers from 1 to 100 since there is no big difference what numbers to use as long as we can compare them and find a difference.

A special array is used to store weights for each vertex. The following code was used to generate those weights in addition to the randomly generated graph.

```
ReDim w(1 To Nodes)
Randomize
For i = 1 To Nodes
  w(i) = Int((100 * Rnd) + 1)
Next
```

Refer to the “Graphs generation model” part of the “Unweighted case” - “Random graphs” subchapter for more details on *Rnd* and *Randomize* functions that are random number generators. *Int* is a function that returns only the integer part of a number.

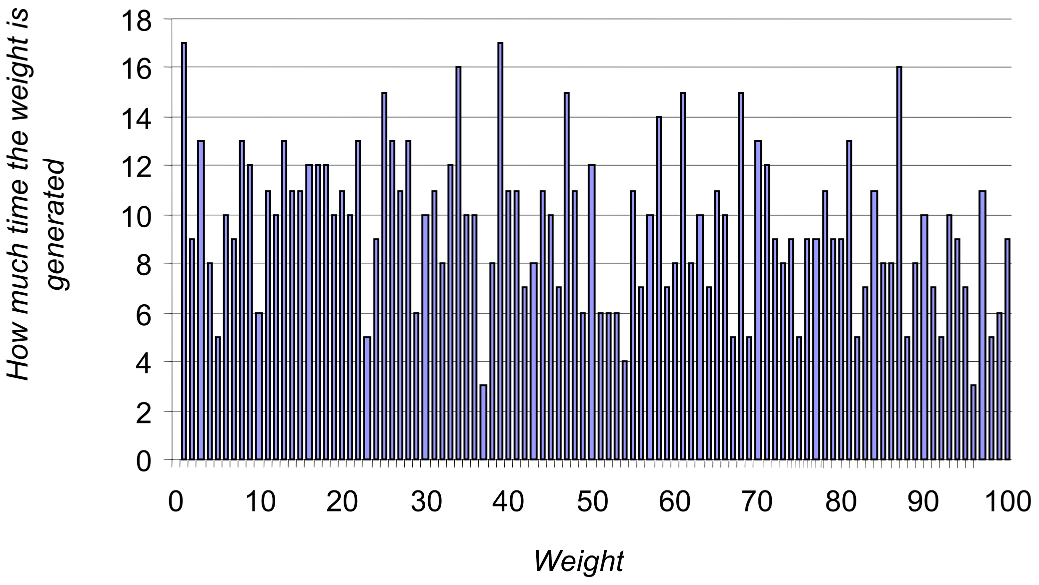

**Figure 20.** Distribution of randomly generated weights

Here you see a distribution of 1000 times generated weights. Different weights occur again quite random number of times and do not tend to be grouped somewhere.

### 3.4.3.2 Time spent on the weighted maximum clique finding

Several algorithms were published since the 1975s. The easiest and effective one was presented in an unpublished paper by Carraghan and Pardalos [Carraghan and Pardalos 1990b]. This algorithm is nothing else that their earlier algorithm [Carraghan and Pardalos 1990a] for the unweighted case applied to the weighted case. They have shown that their algorithm outperforms algorithms that they had compared it with. One more algorithm was published by Östergård [Östergård 2001] recently. He also has compared his algorithm with earlier published algorithms and had shown that his algorithm works better than other best known algorithms.

Results are presented as a ratio of algorithms' spent times on finding the maximum-weight clique – so the same results can be reproduced on any platforms. Compared algorithms were programmed using the same programming language and the same programming technique (since the new and Östergård algorithms are just modifications of Carraghan and Pardalos algorithm). The greedy algorithm was used to find a vertex-colouring.

For each vertices/density case 100 graphs were generated and average time was measured.

*PO* – time needed to find the maximum-weight clique by Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990b] divided by time needed to find the maximum-weight clique by Östergård algorithm [Östergård 2001].

*VColor-BT-w* – time needed to find the maximum-weight clique by Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990b] divided by time needed to find the maximum-weight clique by the "VColor-BT-w" algorithm.

**Table 17.** Weighted case / Benchmark results at random graphs - average ratios of time spent on the maximum-weight clique finding / the base algorithm's time divided by a corresponding algorithm's time

| Edge density | Vertices | *PO* | *VColor-BT-w* |
|---|---|---|---|
| 0.1 | 1000 | 1.12 | 1.28 |
| 0.2 | 800 | 1.23 | 1.93 |
| 0.3 | 500 | 1.42 | 2.78 |
| 0.4 | 300 | 1.63 | 2.81 |
| 0.5 | 200 | 1.73 | 2.81 |
| 0.6 | 200 | 1.90 | 4.90 |
| 0.7 | 150 | 2.12 | 5.54 |
| 0.8 | 100 | 2.27 | 6.83 |
| 0.9 | 100 | 11.25 | 69.85 |

For example, 6.83 in the column marked *VColor-BT-w* means that Carraghan and Pardalos [Carraghan and Pardalos 1990b] algorithm requires 6.83 times more time to find the maximum clique than the "VColor-BT-w" algorithm. Presented results show that the "VColor-BT-w" algorithm performs very well on any density. It is faster than both algorithms we compare it with. Especially great results are shown on the dense graphs, where the new algorithm is 69 times faster than Carraghan and Pardalos algorithm [Carraghan and Pardalos 1990b] and 6 times faster than Östergård algorithm [Östergård 2001].

## 3.5 Summary

Here we have proposed several new algorithms for finding the maximum clique both for the unweighted and the weighted graphs. Those algorithms were best on all densities for all graph types that we have researched using for the comparative testing best general type algorithms, i.e. those, which should solve any graph types. The highest difference is reached on dense graphs. The main new technique we used to make algorithms to be quick is the pruning formula based on the vertex colouring, i.e. independent sets using the fact that vertices of the same independent set cannot be included into the same clique. This new technique sufficiently decreases a number of branches that the algorithms have to analyse in finding the maximum clique and that is the main reason why those perform better than previous algorithms. Several new algorithms were produced from a composition of using the backtracking search and independent sets.

Other advantages of the invented algorithms are:

- They require less memory since they are oriented on the colour classes' number rather than on the vertices number, and this number is sufficiently smaller. Usually programs are allocating memory for all possible depth etc. at the start and it is where we are saving memory by allocating less;

- Algorithms inherit simplicity of programming and studying / explaining from the original branch and bound algorithm. This makes an algorithms' realisation simpler; a risk of making bugs decreases also; algorithms are simpler to test since number of source code's branches is smaller in comparison to many other algorithms' source codes. So the new algorithms are easy and efficient.

One component of the new algorithms is a heuristic vertex-colouring. We used the simple greedy heuristic because we wanted to concentrate mainly on the pruning technique using colours rather than on different vertex colouring heuristics. The heuristic's result was 2 times more colours than the size of the maximum clique in average. We believe that this result could be slightly improved on random graphs, although those numbers will never match on average and the difference is expected to be around 30% on average. Another important result here is a concatenation of those two main classical problems inside one algorithm: finding the maximum clique and the vertex-colouring. The vertex colouring here is not just a subtask providing a bound that loses quickly actuality, but is an essential and important part of the algorithms.

All those advantages make the proposed algorithms to be the best by the time spent on finding the maximum clique and from the applying point of view.

# 4 TESTING ENVIRONMENT

## 4.1 General Description

Here we describe the testing environment we used to test algorithms. In other words, it is the environment that is figured out during our work on the new algorithms. This discussion can be seen as the next step of experimental analysis of algorithms' discussion started by Johnson in the year 2002 [Johnson 2002].

The goal of the testing environment, which is discussed here, is to test different algorithms for finding the maximum clique and mainly measure a time needed to find a solution, although some other parameters can be measured if corresponding parts are implemented for each module (algorithm) to be tested.

The following requirements have figured out as essential needs for a testing environment of our type:

1. It should be able to test different types of graph classes, like:
    a. Random graphs – in other words the system should have a module that is able to generate random graphs. Note that a „true" randomisation is required, since each time a lot of graphs of the same type should be provided. There is no point to generate graphs that are very similar and moreover it should not happen that randomisation is restarted each time a graph is generated and it leads to generating exactly the same graphs;

    b. It should be possible to load into the environment external graphs. Note that there are different standards therefore the following types of graphs' definitions should be supported:
        i. DIMACS format graphs – both compressed and decompressed versions;
        ii. Adjacency matrix graphs, i.e. graphs that are defined by an adjacency matrix.

2. It should be able to solve both problems: finding the maximum clique and finding the maximum independent set using the same modules (algorithms), since those problems are equivalent and there are different graphs for both problems.

Taking into account those requirements the testing environment that contains the following main parts is proposed:

1. Algorithms or modules that implement one or another algorithm;
2. Utilities' module that generates graphs, saves results etc.;
3. A meta-algorithm that makes tests by rerunning algorithms with the same graphs;

4. A user interface
    a. Providing a feedback, i.e. info on events and the current processes status;
    b. Allowing selecting algorithms to be tested and graphs to be used for testing.

Let's now review each of those modules separately.

## 4.2 Modules

Modules are parts of the environment that are implementing algorithms. Each module should have two main properties:

- It should be standard from the input/output interface point of view;
- It should be written using the same programming language and techniques as other modules, as much as possible. This will ensure that neither algorithm is better due to the better programming. All tunings made for any algorithm should be transferred to others if it is possible.

So, each algorithm is implemented as a standard module and can be easily added into / removed from the testing environment. The input parameter is a graph to be solved and the output is the size of the maximum clique. It is necessary to control during tests if all algorithms are working correctly and the size of the maximum clique obtained by different algorithms is the same. Note that we are mainly concentrating on spent times and sizes of the maximum clique, here in tests, rather than on actual maximum clique vertices as an output.

It is also possible to measure some other parameters by programming into modules a standard block for that. The block is programmed once and then adopted inside each algorithm. The ideally programmed block should not require any adaptation since otherwise similarity of algorithms will decrease because of such unequal measuring. This is a way we have measured a number of analysed branches / iterations made by algorithms. An ideal method to activate such blocks is a global variable. Although it is not advised to have global variable, here it seems to be the best approach to go with as it allows controlling the algorithms work from one central place and makes algorithms easily moveable between the meta–algorithms slots that are activating algorithms' modules.

## 4.3 Utilities

This is a part of the environment that provides a general level functionality. First of all those are input/output functions:

1. Function allowing reading external graphs;
2. Function allowing generating a random graph;
3. Function allowing saving results in an output file.

As we already stated before two main formats have to be used: DIMACS and the adjacency matrix. The first one is the main format that is used in researches. Graphs of this format are often compressed and stored in so called binary format, although the decompression algorithm can be easily found from the Internet or in the same ftp folder of the DIMACS program, where graphs are stored. The second format is used in some university classes, since a graph definition using the adjacency matrix is more visual and therefore is easily understandable by students.

The question of generating a random graph is discussed under the "Graphs generation model" subchapter. The only note that we can do on that - we used to generate fully random graphs in this study, although sometimes it is necessary to generate random graphs of some type (with some properties). So, it is possible to use more than one graphs generation technique and choose one of them using an option somewhere on the main user interface.

Note that whatever way a graph is generated or whatever format of an external graph is used, internally the graph should be saved in one way, which is a "standard" for a particular test environment. This ensures that all graphs are treated in the same manner by modules. Besides, the graph's reference, i.e. the input parameter stream for modules will be the same for all cases.

## *4.4 Meta-algorithm*

A meta-algorithm is a main part of the testing environment that mainly glues parts together and manages those parts' work. The main function of the meta-algorithm is to run algorithms one by one using the same parameters and capture a time spent on finding the maximum clique and check correctness of algorithms work by comparing results – the size of the maximum clique produced by different algorithms. The testing process is done in iterations for all densities and numbers of vertices that are required to be tested as many times as it is required. An alternative testing process is providing algorithms with an externally defined graph and capturing the same output parameters, as it was defined above. Anyway, each time exactly the same graph should be provided for each algorithm to be tested. Note that each testing iteration should be able to use (or activate) different modules. There should be a set of options in the user interface that defines which algorithms to test.

Another important feature of the meta-algorithm is a possibility to store result and calculate statistic – minimal, maximal and average results. We have found that it is useful to output both individual numbers and the statistical information since the statistical information is the main research result while individual numbers allow understanding trends and make other calculations in case those were not planned in advance.

An ideal structure of activating modules can be the next:

- Modules should be built using the same base class, which will have a starting function having the same set of input/output parameters – those have been described earlier;
- The meta-algorithm should have a set of slots (an array or a collection), which can contain base classes, so any module can be placed into any slot. Modules are put into slots if and only if those should be tested and this is defined by options at the user interface;

- Modules from each slot should be run one by one either for each generated graph or for an external graph and modules' outputs captured. Note that ideally slots should be able to store those output parameters as well;
- The activation, which is described on the previous step, should be done as many times as it is defined in the user interface. For example, this thesis' comparative tests used to run each test 100 times to collect enough data to make a trustable statistic. The result of this process is an output using the utilities' module to an external file;
- If randomly generated graphs are tested then the previous step should be done for each vertices number / density. Densities should be defined as a range allowing testing more than one density at once (during one test process). Note that best practises make us to advise defining a vertices' number for each density rather than one vertices number for all densities, since a time spent on finding the maximum clique on different densities for the same time differs dramatically. Therefore it is useful to orient on the spent time you want to have rather than on a particular number of vertices.

The meta-algorithm should also produce events allowing seeing a status of the testing process.

So, the meta-algorithm is a core part of the testing environment that manipulates modules and storing results of the test process.

## 4.5 User Interface

This is the last element of the testing environment but not least. Of course, it looks like the testing, i.e. the meta-algorithm and modules are main important parts, but it isn't quite true. The visual feedback is very important as well as a possibility to define options in an easy and comprehensive manner. It makes the environment to be user friendly and allows testing more and does it quicker.

The user interface should allow defining the following:

- Should graphs be generated or provided externally;
- If the graph is provided externally then:
    - The source of the file containing the graph description;
    - The type of the graph's definition: DIMACS or the adjacency matrix;
    - The type of task: the "maximum clique finding" or the "maximum independent set finding";
- If graphs are to be generated then:
    - A range of densities and the step of moving inside the range. For example densities from 10% to 90% with a step equal to 10%;
    - A range of numbers of vertices and the step of moving inside the range or numbers of vertices for each density you are planning to have in the testing;
    - A number of times graphs should be produced and tested for each density / number of vertices;
- A destination's file for the output of the testing;
- Options defining which algorithms to test – one for each algorithm / module;
- Buttons allowing starting and cancelling of the testing process.

The user interface should also show the status of the testing process and a message on the end of it. It is practically impossible to calculate the time needed for all planned tests since different densities could require different time for finding the maximum clique, therefore the full testing time cannot be estimated. Under the status of the process we mean:

- How many test iterations of the total iterations number has been done;
- What algorithm is running now;
- What graph case is being tested now: density / number of vertices.

This helps to orient in the testing process workflow and detect as soon as possible if the test process has been poorly planned and if the graph case could require much more time to find the maximum clique than you have been expecting.

The size of the maximum clique and the spent time should be shown on a message when the process is completed in case of running an algorithm for an external graph. It will save the tester from opening the results file to see this information.

## *4.6 Integration*

This last subchapter gathers all modules together and shows how those are integrated / work together. Let us list parts of the environment once again:

- Algorithms or modules that implement one or another algorithm;
- Utilities module that generates graphs, etc.;
- A meta-algorithm that run tests by rerunning algorithms with the same graphs;
- A user interface providing a feedback and allowing defining options of the testing process.

The integration can be done if and only if all modules are using the same standards / interfaces, raising standard events and returning expected outputs. It was shown earlier in the “Modules” and the “Meta-algorithm” subchapters what the standardisation means for the modules’ structure and in the utilities’ part for graphs to be read or generated. It must be mentioned that the adjacency matrix was used as an internal structure to hold graphs. It is easier to manipulate with, although requires more memory than e.g. the DIMACS format. The integration of different parts can be illustrated using the following scheme:

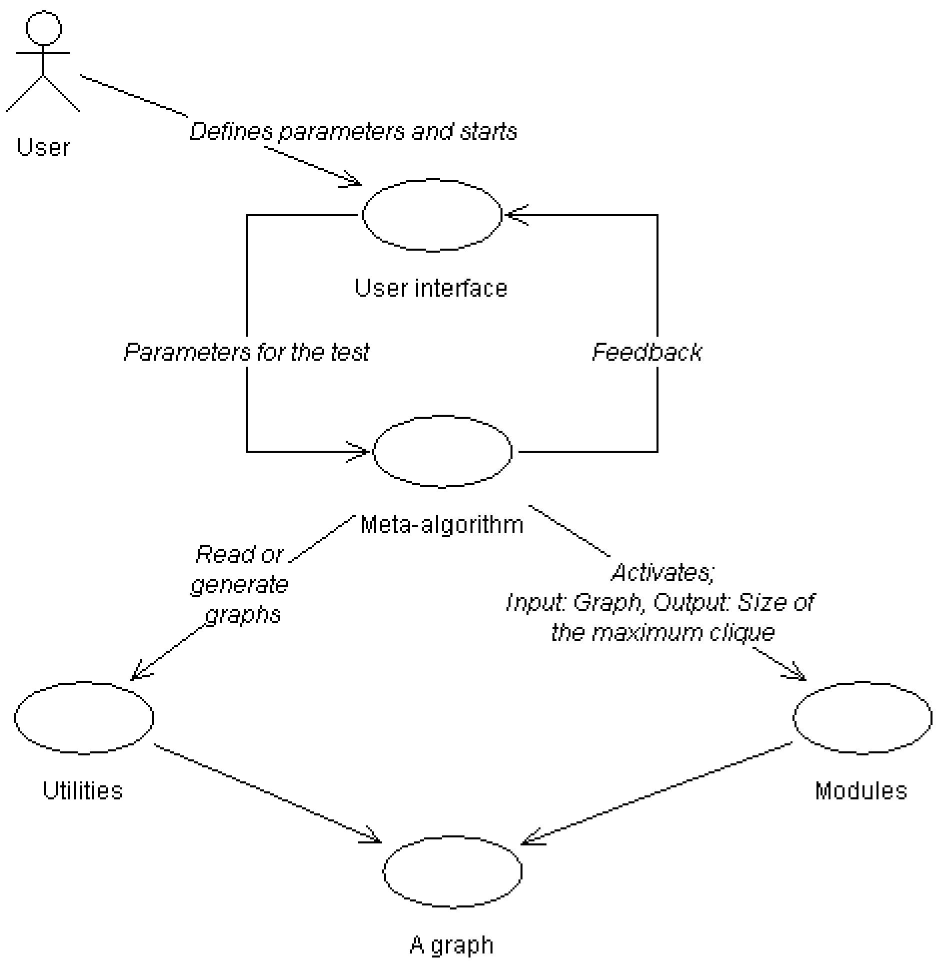

**Figure 21.** High-level structure of the test environment

The high-level model presented in the figure has three layers. The first layer is the user interface, which provides a user with possibilities to operate with the meta-algorithm. This first layer is an intermediate layer between users and the meta-algorithm that disables users to work directly with the meta-algorithm or modules (for example in a DOS like mode). This layer verifies correctness of input parameters. The second layer is the meta-algorithm's layer – the core of the system that receives parameters from the user interface and runs tests. It is a testing logic layer. The third layer contains both utilities and modules. Those parts are indirectly interacting using a graph object that is created by utilities and consumed by modules (implementing algorithms), which are finding the maximum clique in it.

This three layers' model, with all standards defined earlier, fully describes the proposed testing environment.

## 4.7 Summary

This chapter contains a description of the testing environment we used to test algorithms that were proposed and described in this work. This topic is usually undervalued in different researches, although it is an essential part of any research, even of a mathematical one, as it allows proving results and modelling real dependencies between different parameters, e.g. the graph type and the number of branches to be analysed. During our tests some guidelines where worked out in an enormous number of different experiments, mistakes and mis-modellings of the test environment. We hope that this model can be useful for other developers and researches at least as a starting point. Anyway it contains all our experience as a set of suggestions and as our vision of the testing environment.

One of the most interesting topics for the future researches in this area can be building international standards on programming algorithms (for example by inheriting algorithms from a standard base class), graphs presentation, storing and outputting data. Of course there are some standards, for example the DIMACS standard for graphs, but no more. Under international standards we mean such as the ones used in the XML for data exchange, activating services remotely, etc.

# 5 ALGORITHM'S INTELLIGENCE

Each complex problem has different aspects, varies in parameters and internal complexity. It is common for those problems that different problem cases can be solved using different algorithms or their variations. Besides problem "solving" can also mean different things in details, although a general problem remains the same.

This chapter's aim is to put together different ideas, possibilities and needs arising in the maximum clique finding and to synthesize an intelligent algorithm that could address all those issues. Here we look on the maximum clique finding problem again from the programming, i.e. applying point of view rather than from a poor mathematical point of view. Ideas about an intelligence of algorithms are widely discussed in data analyses, data mining and similar areas, and less in the NP-problems; although some ideas are used in heuristic algorithms – see for example a paper published by Jagota and Sanchis in 2001. An idea of a meta-algorithm that we are going to describe here sometimes is discussed in conference halls, but hasn't been formalised until now.

## 5.1 Incomplete Solution

### 5.1.1 Description

First of all we introduce an "incomplete solution" term that helps us to analyse algorithms. A term "most effective" algorithm classically means the quickest algorithm in finding a complete solution, see for example a review provided by Johnson and Trick in 1996, in other words a solution after all vertices of the graph have been analysed. But usually an algorithm finds the maximum clique somewhere in the middle of its work, and then tries to prove that it is the maximum one by looking through all

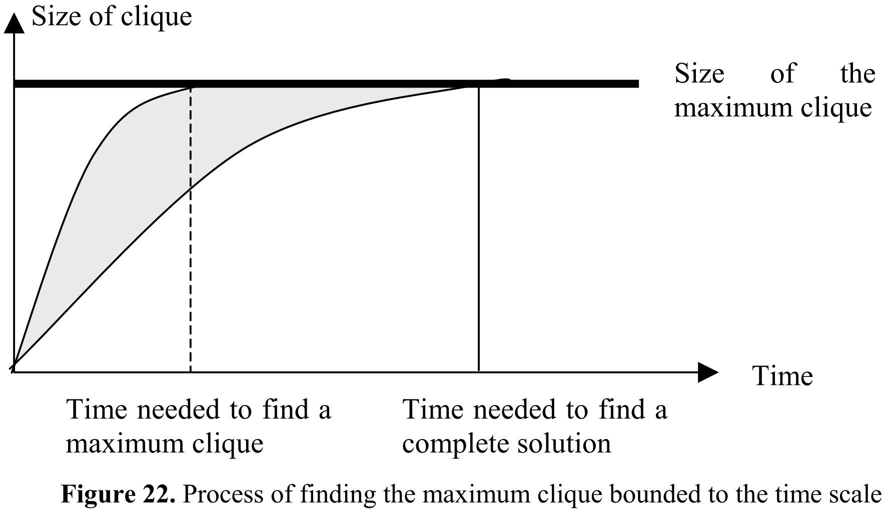

**Figure 22.** Process of finding the maximum clique bounded to the time scale

remaining vertices. Moreover, some algorithms are able to find the best solution even on the first or second iteration [Carraghan and Pardalos 1990a], although sometimes it depends on vertices sorting.

An "incomplete solution" is defined by us as a solution, which is already best/maximum for a particular graph, although it is not yet proved. As you see, it is rather a state of the maximum clique finding process than a type of solution, since at the end of ends the incomplete solution, will become a complete one. This state occurs as soon as the maximum clique is found and lasts up to the algorithm's work end. So it is a question of time – how soon this state will occur and how long it will lasts. This is a new algorithm characteristic that can be important for some types of applications [Musliner et al. 1995, Gat et al. 1990]. For example, there can be a real-time system, which runs a maximum clique finding algorithm. After a certain time the maximum clique algorithm could be interrupted as no more time can be spent on it and the current solution will be used as the best one.

It means that there can be a requirement to optimise the time needed to find the maximum clique in scope of the incomplete solution's definition. Besides, it is possible to define a task to optimise the time needed to find a solution in the predefined interval from the best solution as well as by speed of moving towards the best solution.

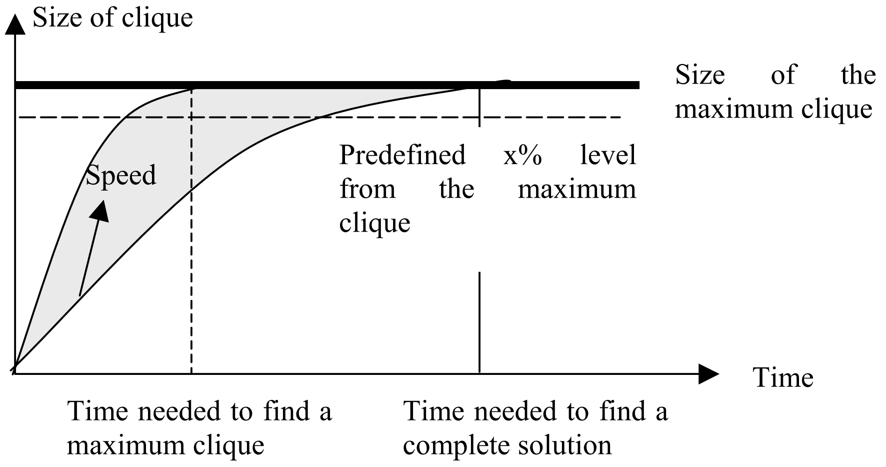
Size of clique
Size of the maximum clique
Speed
Predefined x% level from the maximum clique
Time
Time needed to find a maximum clique
Time needed to find a complete solution

**Figure 23.** Incomplete solution: optimisation points

So, we have two questions of optimisation:

1. Optimise the time needed to find a solution, which is at least *x*% from the best, where *x* is a number from 0 to 100;
   *Note that this is a general definition. The "incomplete solution" is a case when x equals 100.*
2. Optimise an overall speed of moving toward the maximum clique finding.

Those optimisation tasks are not exactly the same. For example, an algorithm could be efficient in finding a maximal clique that is 75% from the maximum one, but the maximum clique will be found just on last steps.

Note that we don't mean that using heuristic solutions is a bad idea. We just mean that in some cases exact algorithms can perform as a heuristic one in finding a solution and then continue trying to prove that this solution is optimal. Besides, it is always better to tune one algorithm to come up with a solution as soon as possible if there is a certain probability that the algorithm will work up to the end, than to run two algorithms: a heuristic and an exact. In our work we will research a question of how fast algorithms, analysed in this work, find the incomplete solution, i.e. 100%.

### 5.1.2 Tests

Here we are going to test algorithms that we used before to find how those perform in finding the "incomplete solution" to get a first overview of connections between existing algorithms and the "incomplete solution" concept. Two properties will be measured – time of finding the "incomplete solution" and the number of times an algorithm was the best. It shows an overall performance of algorithms and how close algorithms are to each other in finding the "incomplete solution".

#### 5.1.2.1 Preliminary analysis

The following preliminary results can be obtained from analysing examples that we played through in the "New algorithms" chapter before. The "VColor-u" algorithm should suit very well for finding the "incomplete solution" as it is efficient and is able to find a solution during first iterations. Especially it should be true for the dense graphs, since graphs with low densities can be efficiently solved by the Carraghan and Pardalos algorithm as well due to its speed of the direct solution's search. The backtracking model is not very good here as it is starting from the "end" of a graph adding vertices one by one into analysis. It should lead to finding a solution somewhere in the middle or at the end of its work – at least it will not be found until all vertices of the maximum clique are added into the backtrack analysis.

The question of vertices sorting could become especially important here. Different algorithms can be used. Generally saying the finding "incomplete solution" task is the sorting task much more than the maximum clique finding one, since here we don't have to prove that the found clique is the maximum one, but rather have to speed up finding it. Unfortunately the question of finding the best sorting has the same complexity as the problem of finding the "incomplete solution" since the number of different sortings is exponential. Therefore we have to choose some sorting to be used by all algorithms. Different sorting algorithms can be more or less suitable for particular graph cases, so the question of choosing a better sorting algorithm is rather a topic to be discussed under the "Adoptive Algorithm" and the "Algorithm Learning and Results Knowledge Base" subchapters. Now we are going to investigate the general algorithms' behaviour in terms of finding an "incomplete decision" and see what algorithms don't perform well from this point of view. Therefore we are going to apply for algorithms the same vertices sorting algorithm as we did for "VColor-u" or "VColor—BT-u" algorithms – sorting by colouring. The question of finding a sorting technique will be postponed for later studies.

### 5.1.2.2 Results

We use randomly generated graphs. See the "Graphs generation model" subsection for more details on the generation model. For each case 100 graphs of the case's density and number of vertices were produced, i.e. each case contained 100 iterations. The first step of each iteration was to find out the maximum clique size and then to run algorithms analysed. As soon as an algorithm found a clique of the maximum clique size, it was stopped.

The first property we are about to measure is a number of times one or another algorithm is the quickest one.

*CP* – Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm;
*PO* – Östergård [Östergård 2002] algorithm;
*VColor-u* – a new algorithm invited in this study – see the "VColor-u" algorithm.

Note that here only the "VColor-u" algorithm is used, as the "VColor" type pruning for the "incomplete solution" is to be measured, and therefore the "VColor-u" and the "VColor-BT-u" are not wanted to compete.

**Table 18.** Number of times algorithms are the quickest in finding an "incomplete solution"

| Edge density | Vertices | *CP* | *PO* | *VColor-u* |
|---|---|---|---|---|
| 0.1 | 1600 | 100 | 0 | 0 |
| 0.2 | 1200 | 68 | 0 | 32 |
| 0.3 | 750 | 48 | 0 | 52 |
| 0.4 | 500 | 44 | 0 | 56 |
| 0.5 | 300 | 41 | 0 | 59 |
| 0.6 | 200 | 33 | 0 | 67 |
| 0.7 | 150 | 24 | 0 | 76 |
| 0.8 | 120 | 13 | 0 | 87 |
| 0.9 | 100 | 5 | 0 | 95 |

For example 41 in the *CP* column means that the Carraghan and Pardalos algorithm was the quickest in finding the "incomplete solution" 41 times for graphs of this row density and number of vertices.

The first result seen in the table is zeroes in the "PO" column of Östergård algorithm that uses the backtracking search. It means that this algorithm, very efficient for finding the maximum clique, is not very good in finding the "incomplete solution" (as we were expecting). So results are distributed between the Carraghan and Pardalos algorithm – see the *CP* column, and the "VColor-u" algorithm – see the *VColor-u* column. The Carraghan and Pardalos algorithm was always the best on densities less than 10% and the best on the 20% density. The "VColor-u" has won more starting from 40% but was surely the best starting from the 60-70% densities. We should conclude that those algorithms were competing on densities starting from 30% and up to 50%.

The next property to be measured is the time spent on finding the "incomplete solution". Again we are using the ratios of time to make result easily reproducible on any platforms – see "Appendix" for algorithms' program codes.

*CP* – time needed to find the "incomplete solution" by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm divided by time needed to find the "incomplete solution" by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm.
*Note that we use this as a normalising line needed for the later presentation of results on the graph view, since this ratio is 1 always by definition.*
*PO* – time needed to find the "incomplete solution" by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm divided by time needed to find the "incomplete solution" by Östergård [Östergård 2002] algorithm.
*VColor-u* – time needed to find the maximum clique by Carraghan and Pardalos [Carraghan and Pardalos 1990a] algorithm divided by time needed to find the "incomplete solution" by the "VColor-u" algorithm.

**Table 19.** How fast algorithms are in finding an "incomplete solution" – spent time

| Edge density | Vertices | *CP* | *PO* | *VColor-u* |
|---|---|---|---|---|
| 0.1 | 1600 | 1.0 | 0.1 | 0.2 |
| 0.2 | 1200 | 1.0 | 0.3 | 0.8 |
| 0.3 | 750 | 1.0 | 0.3 | 3.4 |
| 0.4 | 500 | 1.0 | 0.2 | 5.4 |
| 0.5 | 300 | 1.0 | 0.3 | 6.9 |
| 0.6 | 200 | 1.0 | 0.3 | 8.5 |
| 0.7 | 150 | 1.0 | 0.86 | 10.9 |
| 0.8 | 120 | 1.0 | 0.75 | 17.5 |
| 0.9 | 100 | 1.0 | 1.8 | 140.48 |

For example 8.5 in the *VColor-u* column means that the Carraghan and Pardalos algorithm requires 8.5 times more time than the "VColor-u" algorithm to find the "incomplete solution".

First of all the PO column – the Östergård algorithm – is the slowest in finding the "incomplete solution" on all densities except 90%. We expected this result basing on zeroes in this column in the previous table showing number of times the algorithm was the quickest. It is surprising that this algorithm was quicker than that of Carraghan and Pardalos on average on the highest density, even when using the backtracking search. It should mean that the Carraghan and Pardalos algorithm's pruning technique degenerated fully on this density allowing the Östergård algorithm be quicker on average. The fact that the Carraghan and Pardalos algorithm still was 5 times the quickest on this density of 90% should mean that for those won cases the sorting was "right" for finding the solution during first iterations and those 5 times are rather a statistical deviation, since a right sorting cannot be easily found.

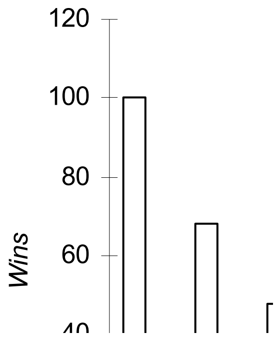

**Figure 24.** Incomplete solution finding: number of wins and time spent by algorithms

This graphical view contains results from both tables for two main algorithms that have been winning on different densities. It is easy to see that the number of wins and ratios of time for the "VColor-u" algorithm are growing proportionally with the density and therefore the same proportions are valid for the Carraghan and Pardalos algorithm but in the decreasing direction for the number of wins. The Carraghan and Pardalos algorithm is clearly the best by all parameters on low densities and the "VColor-u" algorithm is clearly the best starting from the middle-high densities. So, both algorithms are worse to use on different densities for general graphs, although for a particular graph cases those graphics could be different. This question will be discussed in the next subchapters both for the general problem of finding the maximum clique and for finding the "incomplete solution".

## 5.2 Adaptive Algorithm

Here we are going to present a philosophy of building an algorithm that concentrates inside itself all the best algorithms and is intelligent enough to apply the right one. This idea means that we need a meta-algorithm that will collect data and have some intelligence. Different types of intelligence could be used. The easiest way is to have an "expert systems" type meta-algorithm, which will have fixed type rules. The more complex one could be clever enough to learn like, for example, neural networks do.

### 5.2.1 "Expert" type intelligence

As we have already seen in the "Tests and results" chapter, there is no universal algorithm that solves all graphs cases faster than other algorithms. It is rather common to have a set of algorithms or modifications of those that have different strong sides and therefore are good in solving one or another particular graph case.

So, it is possible to build a meta-algorithm with fixed rules that will select the best algorithm basing on the preliminary information about a graph to be solved, or basing on an initial analysis of the graph. The easiest information that we usually have before running the main algorithm is the graph's density. The "Test and Results" chapter has shown that if the density is no bigger than 10% then there is no better algorithm than the trivial and powerful Carraghan and Pardalos one that will solve the problem of finding the maximum clique directly without spending valuable time to any unnecessary additional steps. Otherwise use the "VColor-BT-u" algorithm to solve all others densities in common case. The results of this work have shown that this algorithm is sufficiently faster than others.

Another kind of information that we can have is the type of a graph. This information is not always available, but if you have it or know how the graph is built then it is possible to save a lot of time by applying the right algorithm to find the maximum clique. For example, there are permutation and interval graphs that can be easily solved by corresponding algorithms in the polynomial time and there are no points to apply for them algorithms targeted to solve all possible types/structures of graphs. Certainly only some graphs can be solved in the polynomial time, but even for graphs that are hard to solve there could be algorithms that suit more. We have analysed some graph cases in the "DIMACS" subchapter of the "Tests and Results" chapter for the unweighted graphs and you can see, which algorithm is better for which graph's type. Moreover, we never tuned our algorithms to perform better on one or another graph type, but this could be done. Reasons why we never did it are:

- We tried to come up with a common solution/algorithm that will find the maximum clique on any graph;
- We have compared our algorithms with other algorithms that we are not going to tune and those were not tuned by their authors; therefore we should not tune our algorithms as well to be honest in comparison tests.

So there are a lot of possibilities to come up with tuning ideas for our main algorithm "VColor-BT-u" to make it better on certain graph types as well as for other algorithms. It is logical that all those modifications should be available for the meta-algorithm to choose which of them to run.

Another question we have been discussing before is the "incomplete solution" case, i.e. when it is necessary to provide with a maximal or even the maximum clique even if the algorithm will be interrupted somewhere in the middle of its work. An algorithm, that is chosen to be the best one, sometimes is not able to find the "incomplete solution" fast and could require running some other algorithms before to ensure returning of an acceptable solution in case it is interrupted.

The meta-algorithm should follow the next general rules:

1. If the type of a graph is known then it should run the best algorithm for that type;
   *Note: It means that the meta-algorithm should have some knowledge base in addition to available algorithms that will allow choosing the right algorithm to run.*
2. Choose an algorithm basing on the graph's density;
3. If there is a probability that the maximum clique finding algorithm will be stopped and the algorithm chosen on the previous step is not known to be good in finding the "incomplete solution", then run either a heuristic algorithm or an exact algorithm (and stop it after some time);
   *Note: The obtained result will not only ensure that the algorithm will return an acceptable result, but also should be used as a low bound for the main exact algorithm.*
4. Run the algorithm that was chosen on the second step.

We have figured out the following meta-algorithm rules basing on results published in this work for the unweighted case:

1. If the type of graph is known then it should run the best algorithm for that type;

2. If the density of graph is smaller or equal to 10% then run Carraghan and Pardalos algorithm;

3. If there is a probability that the maximum clique finding algorithm will be stopped then:
    a. If the density is up to 25%, run the Carraghan and Pardalos algorithm for some time and then go to the next step;
    b. If the density is between 25% and 60%, run the "VColor-u" algorithm for some time and then go to the next step;
    c. Otherwise go to the next step directly.

    *Note that different heuristics can be useful here if any is reported to be the best for a particular case to be solved.*

4. Run the "VColor-BT-u" algorithm.
   *Note that if the "VColor-u" algorithm was used on the previous step then the "VColor-u" algorithm's colouring can be directly reused by the "VColor-BT-u" algorithm.*

### 5.2.2 Algorithm learning and results knowledge base

In the previous chapter possibilities to use fixed type rules have been reviewed . We have built the meta-algorithm, which is an expert in the maximum clique finding. We used "Expert systems" ideas and provided our meta-algorithm with all knowledge we have at the moment. Unfortunately, the same assumption is made again that we have to invent an algorithm that will deal with very different graphs and that should solve any graphs. There is one motto that is widely used nowadays – "Think globally, operate locally". Any particular case could have its own aspects, properties etc. of graphs to be solved. We could not foresee all those aspects and, moreover, these can be opposite to what we were expecting, or those requirements could be opposite to an algorithm building point of view. Therefore the ideal case will be a self-learning algorithm. Of course, we do not talk about a meta-algorithm that will invent new algorithms to find the maximum clique. Probably it is too self-confidently to try invent such right know. What we mean is a meta-algorithm that will be able to collect some statistics about algorithms' performance on graphs that were solved and later, basing on this statistics, will be able to choose, which algorithm to run. This meta-algorithm will adapt to a particular environment and graphs existing in this environment, to the environment where it has to operate. This adaptation will mean that we move from the general "expert system" to a more evolving algorithm, which is able to "survive" in any particular environment in the best way.

Collecting information on which algorithm/modification is better generally means that the meta-algorithm will try to run all algorithms/modification with all graphs. Otherwise it will not be possible to answer a question: "Will any other algorithm perform better than the one we are going to use?" Another important aspect we should think about is providing more information than the meta-algorithm can collect by itself like a graph's density or number of vertices. Is there any additional information on the graph? Are all graphs the same or you know their types? Is it possible to distinguish a source of the graph? Any such information will be useful to keep statistics and better adapt for any particular graph cases.

Now, we know all additional information and we can pass these details into the meta-algorithm. The main question is how to start collecting data. This can be done in two ways:

**Run other algorithms while in the stand-by mode**

The meta-algorithm that finds the maximum clique is rarely asked to do it continuously. So it doesn't have to resolve immediately another problem after the previous one. In this case, after returning an answer, the meta-algorithm can use available free resources. It can try all other algorithms not used to give the answer, to find if there was a better/quicker way to perform the task. Basing on the collected information each algorithm could receive points (for example 1 point each time to the fastest one). Basing on those points an algorithm to be used next time should be chosen. If there is a high probability that a new task will arrive soon then the meta-algorithm should try algorithms in the already obtained points' sequence. This will allow trying first the most probable one to be the fastest, then the next probable one and so forth.

**Learn the meta-algorithm to use available algorithms**

Another idea is to train the meta-algorithm use its algorithms for finding the maximum clique prior to the real using. The idea allows collecting statistics before you start to use the meta-algorithm and it will not be necessary to spend resources on collecting statistics later while in operation.

The training could be done by asking to solve as many different types of graphs as possible in all required modes if any exists – like a requirement to stop after e.g. 10 seconds and provide the best found solution etc. For any type/mode as many graph examples as possible should be used. Instead of using artificial examples, it is always advisable to use such examples that will likely occur later during the real using of the meta-algorithm.

It is also important to monitor the performance and changes in the environment. If graphs to be solved are changing due some changes in the requirements or you suspect that the meta-algorithm is not providing its best, then it is time to re-train the meta-algorithm.

Both ways have another very important advantage in addition to the described – those ways allow collecting information that makes it (also for you) possible to learn how well graphs are solved and which algorithms are used to solve any particular graph case. This gives a possibility to analyse collected statistics and to invent even better modifications of existing algorithms.

## 5.3 Summary

This chapter has described some ideas of building an intelligent meta-algorithm that is able to adapt for graphs that have to be resolved in a particular environment and to apply the best variation of maximum clique finding algorithms for each individual graph case. We do not mean using special algorithms separately for any particular case, but rather using a universal meta-algorithm, which is able to adapt. These ideas came from expert systems and data analyses and can be successfully used in the maximum clique finding from our point of view. This discussion is more an applying part of the maximum clique finding, although this cannot be done without understanding of what the maximum clique finding problem is and which properties should be tracked by the meta-algorithm. Using such methods should increase the general performance of algorithms that are applied in this or another science area.

Another area discussed was finding the maximum clique in the real-time environment when an algorithm can be stopped at any moment. The term "incomplete solution" has been introduced and embedded into the overall logic of the meta-algorithm. The "incomplete solution" is nothing else than the maximum clique, which is not yet proved to be the maximum one. It means that although we know that the maximum clique is found, we have to scan all remaining branches to ensure that. Some tests have also been done to get the general overview of how existing best algorithms perform in the "incomplete solution" finding. These tests have shown that internal structures of algorithms are very different and some algorithms suit more for operating in the real-time environment than others. We also came up with some suggestions about meta-algorithm's rules, which can take those properties into account.

Using such meta-algorithm will allow automating the selection of the best maximum clique finding algorithm to run. This process could save a lot of time since we are dealing with NP-complete problems. Besides, it saves a lot of man-hours when a system is intelligent enough to make decisions.

# 6 SUMMARY

## 6.1 Researched Problem

The main topic of this study as it has been stated in the subchapter 1.3 is the maximum clique finding from an arbitrary undirected graph. Both weighted and unweighted graphs were researched. It has been shown in the thesis' introduction that the studied maximum clique finding problem is important first of all from two points of view: complexity and applications. The problem's complexity is reviewed in the subchapter 1.4, where it has been demonstrated that this problem belongs to the class of NP-hard problems. It has also been shown in the subchapter (1.4) that a better understanding of this problem lets us understand better all other NP problems as well as a better and quicker algorithm can provide us with better ways of solving practically any other NP problem including main NP complete problems. Applications of the maximum clique finding are the second important implication of this problem and those were reviewed in the subchapter 1.6. The treated problem has a lot of extremely important applications in a variety of areas like biology, circuits' design and testing, medicine, etc. This occurs because the theory of graphs provides all other science areas with a very good and sophisticated abstract model and the maximum clique problem is one of the central problems of the graph's theory. Here it should be noted again that the maximum clique finding problem belongs to the NP problems' class, i.e. is extremely time consuming for solving. That's why any better algorithm is able to save hours, days of maybe even months of work time. This shows importance of this task and why achieving the main objectives of this thesis, declared in the subchapter 1.3, could have a strong economical impact if applied in any business, economical, transportation etc. applications.

## 6.2 Thesis Summary

It has been defined in subchapters 1.3 and 1.5 that the goals of this thesis are: proposing new algorithms for the maximum clique finding in an arbitrary undirected graph (both weighted and unweighted cases) through identifying graphs properties that should accelerate the maximum clique search; developing a methodology for building a suitable test environment for new algorithms and then concentrating the experience obtained so far by working out a philosophy of building a meta-algorithm for solving the maximum clique finding problem. That's why the thesis has been divided into three major logical parts: the new algorithms, testing, and algorithms' intelligence.

During our researches we have detected a graphs' property that provided us with better algorithms – very simple and therefore having so big impact: any vertices of an independent set cannot be included into the same clique. This property is used for a pruning formula of the simple but efficient branch and bounds algorithm. We have suggested in the presented thesis finding those independent sets using a vertex-colouring algorithm. The last task is also NP-hard; therefore we used a heuristic algorithm to find the vertex-colouring of a graph. We demonstrated using of this new approach directly by describing three new algorithms – the "VColor-u" and the "VColor-BT-u" for the unweighted graph case and the "VColor-BT-w" for the weighted graph case. Generally our algorithms are a very successful evolution of the classical algorithm invited by Carraghan and Pardalos [Carraghan and Pardalos 1990a]. We have demonstrated in this work that a number of colour classes (independent sets produced by a vertex-colouring algorithm) much better estimates a possible maximum clique size of any (sub)graph than the number of vertices used so far. A special algorithm for recalculating the number of existing colour classes on a subgraph has been proposed here serving the pruning formula. This algorithm is another important contribution of this paper since it makes possible to use the vertex-colouring measure quickly and effectively. It includes vertices reordering accordingly to their colours and a sub-procedure for recalculating the number of existing colours by shifting a previously obtained result. In addition, it includes reordering of vertices by weights in each colour class for the weighted case. All this has been demonstrated inside an extended explanation of each algorithm's description, which makes the algorithms easy to implement and understand. A set of examples have been provided in addition to explanations and a formal formulation of the algorithm, which are simple to follow. We also have analysed each example to demonstrate what was important inside it and what knowledge could be derived from it. Our experience with other algorithm has shown that it is easy to mis-implement an algorithm by making a small mistake somewhere in algorithm's details. Such mistake can lead to dramatic decrease in the performance. In order to eliminate such possibilities we have worked out a set of guidelines on our algorithms implementation, which are provided in the 3.2.3 subchapter. We also have inherited a backtracking search idea invited by Östergård [Östergård 2002, Östergård 2001] into the algorithms having "BT" in their names. Here we have proposed using colour classes instead of individual vertices and have demonstrated that concatenation of the colour classes' approach and the backtracking idea makes the algorithms' work much more efficient.

In the subchapter 3.4 we conducted a comparative study of the new algorithms. We used an algorithm proposed to be used for benchmarking by DIMACS [Johnson and Trick 1996, DIMACS 1999], which is invited by Carraghan and Pardalos [Carraghan and Pardalos 1990a] and called in the thesis - "base" algorithm. In order to demonstrate that the proposed algorithms are the quickest at the moment we picked up also algorithms, which are reported to be the quickest right now – invited by Östergård [Östergård 2002, Östergård 2001] – both weighted and unweighted cases. It is enough to use Östergård's algorithms since he recently has done his comparative study with a lot of others algorithms and we fully trust into those results. We conducted our tests for the unweighted case both on randomly generated graphs – see the subchapter 3.4.2.1, and on special types of graphs mainly provided by DIMACS Challenges [DIMACS 1999] – see the subchapter 3.4.2.2. We used only randomly generated graphs for the weighted case – see the subchapter 3.4.3, since there is a lack of special graph cases right now. It was shown, that all our algorithms behave very good on all graph cases, and always outperform the compared algorithms. Especially good results are shown on dense graphs, where a lot of other algorithms degenerate to the exhaustive search. Usage of this new pruning technique gave us algorithms, which are better than the base algorithm in hundreds times on dense graphs. The colour classes' approach made the backtracking search perform much better as we have shown through our test. We moved the backtracking search on another, a higher level of the operation and performance. The algorithms “VColor-BT-u” (unweighted case) and “VColor-BT-w” (weighted case) were best in most cases although for some graph cases we have seen that “VColor- u” out perform others. We also investigated a reason of this increase in the performance and have demonstrated that a number of subgraphs analysed during the maximum clique finding using colour classes fall dramatically.

The algorithms proposed by us in this thesis also remain simple for understanding and implementation. This is an important property that is inherited from the branch and bound type of algorithms. Therefore the algorithms are not just the quickest (which is important by itself), but also are simple and this makes it possible to use those by universities for studying by students and effectively apply in real tasks. A lot of practical suggestions and examples are provided in this work to make implementation of those algorithms to be simple and quick.

We concentrated in our thesis a lot on practical aspects of the maximum clique finding instead of been limited by researching only mathematical or theoretical aspects of this problem because of the high implication on the real economical problems described in the subchapter 1.6. Our aim was to make thesis’ results easily applicable and motivate further interest in this research from an applications point of view. That’s why there is fewer maths than somebody could expect. The same is about algorithms that could be very good from a mathematical complexity point of view but are hard to apply. That’s why we used for our comparative tests in the subchapter 3.4.2 best algorithms that are not only best from the mathematical point of view but are also can be really used in different applications – algorithms invited by Carraghan and Pardalos [Carraghan and Pardalos 1990a and Carraghan and Pardalos 1990b] and by Östergård [Östergård 2002, Östergård 2001].

Different versions of the presented algorithms have been recently published and presented on conferences [Kumlander 2003, Kumlander 2004a, Kumlander 2004b, Kumlander 2005a].

We continued our thesis in the next thesis part by describing a test environment, i.e. a methodology of building it. This topic is usually undervalued although is an essential part of any research in the graph theory since any mathematical model should be usually proved by practical results. A philosophy of constructing the test environment is discussed in the chapter 4 basing on a huge number of experiments and tests that have been done to test the new invited algorithms. The proposed model has been divided into several independent parts, which are grouped into several layers. The following parts have been identified as parts of the model: modules implementing algorithms – see the subchapter 4.2; a utilities library that enables reading graphs, generating graphs and outputting results – is described in the subchapter 4.3; a meta-algorithm and a user interface are core elements of the model and are responsible for running tests – those are explained in subchapters 4.4 and 4.5. Those model parts are integrated using internal standards and a special architecture of the test environment. The model and data flows are presented in the subchapter 4.6. A lot of advices based on our experience are provided and the architecture is discussed in this chapter. This discussion is the next step of experimental analysis of algorithms discussion started by Johnson in the year 2002 [Johnson 2002].

Finally an algorithms' intelligence has been discussed in the subchapter 5 and several ideas of a meta-algorithm implementation are proposed. This chapter concentrates ideas derived from the meta-algorithm, which was described as a part of the test environment in the chapter 4 and the algorithms we started our thesis from. We have shown in this 5th chapter a philosophy of building the meta-algorithm and this study was initially influenced by different modifications of the new algorithms, which we wanted to have for some special graph cases. Here we proposed a model of the meta-algorithm, containing maximum clique finding algorithms, that can decide basing on some additional information, which of those algorithms to run. Although such meta-algorithms are widely used in other data analyses areas, there are not adopted for the maximum clique problem until now or at least are rather weak. In this part of our thesis we collected different ideas and applied them to the maximum clique finding. The meta-algorithm can increase the performance of the system mostly by applying "right" algorithms for a particular graph case. Again it occurs mainly because of the complexity of the maximum clique finding problem since applying the best algorithm can save a lot of time in solving NP-hard problems.

We started this topic with reviewing a case when an algorithm can be suddenly stopped during the maximum clique finding and a current best clique has to be returned as the best solution. A new term called "incomplete solution" has been introduced in the subchapter 5.1.1 meaning a clique, which is already the maximum one for a graph although it is not proved yet, i.e. a lot of branches should be analysed further until this clique will be returned as an answer. This helps us to fix a point of finding the maximum clique without proving its maximality and was used to test currently best algorithms, which have been used in the previous comparative testing, for such real-time environment. This is a type of information that the intelligent meta-algorithm should certainly get into account. Results of the test are presented in the subchapter 5.1.2.2.

In the end of ends we concentrated all meta-algorithm's issues in the subchapter 5.2 where different meta-algorithms are proposed and a philosophy of those is discussed. The first type of the meta-algorithm announced in the subchapter 5.2.1 is the meta-algorithm, which is implemented like expert systems containing some rules. The next proposed model is much more sophisticated and is able to train by collecting a statistic by main graph properties. Such self-learning meta-algorithm is presented in the subchapter 5.2.2. The strongest side of that meta-algorithm is its capability to adapt to any environment it should work in and resolve graphs in this environment in the best way. We have shown that this meta-algorithm acting locally rather than getting into account all possible cases that can exist in other environments. In addition to the intellectual way of working the described meta-algorithm collects statistic and provides a lot of interesting information for further development of the maximum clique finding algorithms used in it. The described approach to the algorithms intelligence in the maximum clique finding have been recently announced on the "Operational Research 2005" conference [Kumlander 2005b].

## 6.3 Future Research Work

The main direction of future researches would be in the maximum clique algorithms field. We should continue looking for more properties of graphs that can be efficiently used for building new algorithms for finding the maximum clique. We hope that information of none existing edges is undervalued in algorithms based on analysing existing edges. Another interesting research topic would be to analyse further performance of the new algorithms on different types of graphs. This hopefully would enable to detect a new type of graphs, where the algorithms work especially well or bad. We should also investigate more dense graphs in further attempts to increase performance of algorithms there. Even the proposed algorithms' performance there is not as good as we would like to have. Another type of graphs where algorithms are having problems is graphs where all so far developed estimations show that a larger maximum clique could be found than there really exists. Usually those graphs have quite a small maximum clique having average density, for example see error correction code graphs [Sloane 2002].

A necessary research topic to investigate could be whether there could be found a better heuristic vertex-colouring technique than the greedy colouring that we could use in our algorithms. From one hand we cannot expect much better results than the greedy algorithm can produce in general as we have shown in our analyses in the subchapters 3.4.1 and 3.4.2.1.4. From another hand there could be a vertex-colouring algorithm that fits more into the new technique of pruning by colour classes. Besides it is important to check different colouring algorithms for different types of graphs since this information could be very useful to include into algorithms intelligence as well as for the general vertex-colouring problem researches.

We should continue researching the meta-algorithm approach described in this dissertation. An interesting topic to investigate will be a number of iterations for the "incomplete solution" case for different algorithms. We also would like to find more properties basing on which graphs can be spread across different groups.

Another area of researches could be developing an online maximum clique solver exposing web-services for the maximum clique problem. Inside this project both test environments ideas and algorithms intelligence principles could be combined. In the scope of this project we also would try to develop some standards for different areas of maximum clique finding like for example there is for the finance sector – a standard XML format's data exchange – see XBRL. Those standards can be defined to make modules, which should be inherited from a base class or implement certain interfaces; standards on calling a service for finding the maximum clique via Internet and returned output etc. A free online service would promote using those standards and enable to gather opinions of the open society on further standards development. Another idea could be to release a component that can be used online on the customer side – this would decrease a load of the server. Anyway all those researches could promote further development the algorithms' intelligence ideas described so far.

# REFERENCES

Babel L, Tinhofer G (1990) A branch and bound algorithm for the maximum clique problem. Methods and Models of Operations Research, 34:207-217

Ballard DH, Brown M (1982) Computer Vision. Prentice-Hall, Englewood Cliffs, NJ

Bennett, William RJr. (1976) Scientific and Engineering Problem - Solving with the Computer. Englewood Cliffs: Prentice-Hall

Berge C, Chv'atal V, eds. (1984) Topics on Perfect Graphs. Ann. Discrete Math. vol 21, North-Holland, Amsterdam

Berman P, Pelc A (1990) Distributed fault diagnosis for multiprocessor systems. In Proc. of the 20th Annual Intern. Symp. on Fault-Tolerant Computing, pp 340–346, Newcastle, UK

Bomze M, Budinich M, Pardalos PM, Pelillo M (1999) The maximum clique problem. Handbook of Combinatorial Optimization, vol. 4, In D.-Z. Du and P. M. Pardalos, eds. Kluwer Academic Publishers, Boston, MA

Bonner RE (1964) On some clustering techniques. IBM J. of Research and Development, 8: 22-32

Borodin A, Nielsen MN, Rackoff C (2003) (Incremental) Priority Algorithms. Algorithmica 37(4): 295-326

Brelaz D (1979) New Methods to Color the Vertices of a Graph, Communications of the ACM, 22: 251-256.

Brglez F, Fujiwara H (1985) A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. Proceedings of the IEEE International Symposium on Circuits and Systems

Brouwer AE, Shearer JB, Sloane NJA, Smith WD (1990) A new table of constant weight codes. J.IEEE Trans. Information Theory, 36: 1334-1380

Butenko S, Festa P, Pardalos PM (2001) On the chromatic number of graphs. Journal of Optimization Theory and Applications, 109: 51-67

Carraghan R, Pardalos PM (1990a) An exact algorithm for the maximum clique problem. Op. Research Letters 9: 375-382

Carraghan R. and Pardalos PM (1990b) A parallel algorithm for the maximum weight clique problem. Technical report CS-90-40, Dept of Computer Science, Pennsylvania State University

Cook SA (1971) The complexity of theorem proving procedures, Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, pp 151-158

Corradi K, Szabo S (1990) A combinatorial approach for Keller's Conjecture. Periodica Mathematica Hungarica, 21: 95-100

Culberson JC (1992) Iterated Greedy Graph Colouring and the Difficulty Landscape, University of Alberta Computing Science Technical Report TR92-07

DIMACS, Center for Discrete Mathematics and Theoretical Computer Science, Annual Report (1999) Dec, http://dimacs.rutgers.edu/About/Reports/ann00.ps (2005-09-18)

Garey MR, Johnson DS (1979) Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman, New-York

Garey MR, Johnson DS (2003) Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman, New-York

Gat E, Slack MG, Miller DP, Firby RJ (1990) Path planning and execution monitoring for a planetary rover. In Proc. IEEE International Conference on Robotics and Automation, Cincinnati (USA), pp 20-25

Gendreau A, Salvail L, Soriano P (1993) Solving the maximum clique problem using a tabu search approach, Ann. Oper. Res., 41: 385-403

Goldberg DE (1989) Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, Massachusetts

Guignard M, Kim S (1987) Lagrangian decomposition: a model yielding stronger Lagrangian bounds. Mathematical Programming, 39: 215-228

Hertz A, de Werra D (1987) Using Tabu Search Techniques for Graph Colouring, In Computing, 39: 345-351

Hifi M (1997) A Genetic Algorithm-Based Heuristic for Solving the Weighted Independent Set and Some Equivalent Problems. Journal of the Operations Research Society, 48: 612-622

Horaud R, Skordas T (1989) Stereo correspondence through feature grouping and maximal cliques. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11: 1168-1180

Jagota A, Sanchis LA (2001) Adaptive, Restart, Randomized Greedy Heuristics for Maxi-mum Clique, Journal of Heuristics 7(6): 565-585

Jansen K, Scheffler P, Woeginger G (1997) The disjoint cliques problem. Operations Research, 31: 45-66

Johnson DS (2002), A Theoretician's Guide to the Experimental Analysis of Algorithms, in Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges, M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, eds, American Mathematical Society, Providence, pp 215-250

Johnson DS, Trick MA, eds (1996) Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, Vol. 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society.
Karp RM (1972) Reducibility among combinatorial problems. In complexity of Computer Computations, RE Miller and JW Thatcher eds. Plenum Press, New York, pp 85-103

Klotz W (2002) Graph coloring algorithms, Mathematics Report, Technical University Clausthal, May, pp 1-9

Knuth DE (1994) The sandwich theorem, Electr. J. Comb., 1(A1)

Korman SM (1979) The Graph-Colouring Problem. In Combinatorial Optimization, Christofides, Mingozzi, Toth and Sandi Editors, pp 211-235

Kim H (1990) Finding a Maximum Independent Set in a Permutation Graph. Inf. Process. Lett. 36(1): 19-23

Kucera L (1991), The greedy colouring is a bad probabilistic algorithm, J. Algorithms 12(4): 674-684.

Kumlander D (2003) Problems of optimisation: an exact algorithm for finding a maximum clique optimized for the dense graphs. ISMP2003: Program and Abstracts book, Copenhagen, pp 114

Kumlander D (2004a) An exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring. Modelling, Computation and Optimization in Information Systems and Management Sciences, Le Thi Hoai An and Pham Dinh Tao eds, Hermes Science Publishing, pp 202-208

Kumlander D (2004b) A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search. Proceedings of The Forth International Conference on Engineering Computational Technology, Civil-Comp Press, pp 137-138

Kumlander D (2005a) Problems of optimization: an exact algorithm for finding a maximum clique optimized for dense graphs. Proceedings of the Estonian Academy of Sciences, 54 (2): 79-86

Kumlander D (2005b) Algorithms Intelligence in the Maximum Clique Finding. OR2005: Program and Abstracts book, Bremen, 2005, pp 87

Lagarias JC, Shor PW (1992) Keller's Cube–Tiling Conjecture is False in High Dimensions. Bulletin of the AMS, 27: 279–283

MacWilliams J, Sloane NJA (1979) The theory of error correcting codes. North-Holland, Amsterdam

Marchiori E (1998) A Simple Heuristic Based Genetic Algorithm for the Maximum Clique Problem. Proc. ACM Symposium on Applied Computing (SAC98), pp. 366-373

Miller W (1992) Building multiple alignments from pairwise alignments. Computer Applications in the Biosciences

Mitchell EM, Artymiuk PJ, Rice DW, Willet P (1989) Use of techniques derived from graph theory to compare secondary structure motifs in proteins. Journal of Molecular Biology, 212: 151-166

Moon JW, Moser L (1965) On cliques in graphs. Israel Journal of Mathematics, 3:23-28

Motzkin TS, Straus EG (1965) Maxima for graphs and a new proof of a theorem of Tur'an. Canadian Journal of Maths, 17(4): 533-540

Musliner DJ, Hendler JA, Agrawala AK, Durfee EH (1995) The Challenges of Real-Time AI. IEEE Computer 28(1)

Murthy AS, Parthasarathy G, Sastry VUK (1994) Clique finding - A genetic approach. Proc. 1st IEEE Conf. Evolutionary Comput, pp 18-21

Pardalos PM, Resende MGC, Rappe J (1998) An exact parallel algorithm for the maximum clique problem, In R. De Leone, A. Murli, P.M. Pardalos, and G. Toraldo, eds, High performance algorithms and software in nonlinear optimization, pp 279-300. Kluwer Academic Publishers

Pelillo M (1995) Relaxation labeling networks for the maximum clique problem, In J. Artif. Neural Networks, 2: 313-328

Prestwich S (2001) Local search and backtracking vs non-systematic backtracking. In AAAI 2001 Fall. Symposium on Using Uncertainty within Computation

Robson JM (1986) Algorithms for maximum independent sets. J. of Algorithms, 7:425-440

Sanchis L (1992) Test Case Construction for Vertex Cover Problem. DIMACS Workshop on Computational Support for Discrete Mathematics

Schildt H (1987) Advanced Turbo Pascal. Berkeley: Osborne McGraw Hill

Sloane NJA (1989) Unsolved problems in graph theory arising from the study of codes. Graph Theory Notes of New York XVIII, pp 11–20 http://www.research.att.com/~njas/doc/graphs.html (2005-09-18)

Sloane NJA (2001) On Single-Deletion-Correcting Codes. D. Ray-Chaudhuri. Festschrift, available at. http://www.research.att.com/njaa/doc/dijen.pdf (2005-09-18)

Östergård PRJ (2002) A fast algorithm for the maximum clique problem, Discrete Applied Mathematics, 120: 197-207

Östergård PRJ (2001) A new algorithm for the maximum-weight clique problem. Nordic Journal of Computing, 8: 424-436

Walshaw C (2001) A Multilevel Approach to the Graph Colouring Problem. Tech. Rep. 01/IM/69, Comp. Math. Sci., Univ. Greenwich, London SE10 9LS, UK

Welsh D, Powell M (1967) An Upper bound for the Chromatic Number of a Graph and Its Application to Timetabling Problems. In Computer Journal, 10: 85-86

West DB (2001) Introduction to Graph Theory (2nd edition), Prentice Hall

Wood DR (1997) An algorithm for finding a maximum clique in a graph. Operations Research Letters, 21: 211-217

# APPENDIX A: PROGRAMS LISTINGS

## *Unweighted Case*

### VColor-u

```
Option Explicit

' Optimised Carraghan, R., Pardalos, P. M. algorithm
' by using a heuristic vertex colouring classes for pruning

Private moClasses() As Long, mnClassesCount As Long
Private nLevelDegree() As Long
Private level_nodes() As Long, nStart() As Long, NodesNum() As Long
Private t As Long, mnMaxClique As Long
'

Public Function Start() As Long
  Dim i As Long, t_minus_1 As Long

  ReDim level_nodes(1 To Nodes, 1 To Nodes)

  mnMaxClique = 0
  '''' each level has its own set of nodes
  For i = 1 To Nodes
    level_nodes(1, i) = i
  Next

  ''''
  DefineClasses

  ReDim NodesNum(1 To mnClassesCount)
  ReDim nStart(1 To mnClassesCount)
  ReDim nLevelDegree(1 To mnClassesCount)
  NodesNum(1) = Nodes

  t = 1
  nStart(t) = 0
  '''''''''''''''''''''''''''''''
  While t >= 1
    nStart(t) = nStart(t) + 1

    ''' Degree control
    If NodesNum(t) < nStart(t) Then
```

```
      t = t - 1
    Else
      ''' if it is not first node (for fist node degree can not be adjusted)
      ''' and prev. node class is not the same then decrease degree
      ''' (can be done since vertices are sorted
      ''' and if cur. vertex class is not the same as for previous then
      ''' the prev. class is not any longer existing)
      If nStart(t) > 1 Then
        If (moClasses(level_nodes(t, nStart(t))) <> _
             moClasses(level_nodes(t, nStart(t) - 1))) Then
          nLevelDegree(t) = nLevelDegree(t) - 1
        End If
      Else
        ''' calculate degree on new depth
        nLevelDegree(t) = LevelDegree()
      End If
      If (t - 1 + nLevelDegree(t)) > mnMaxClique Then
        t_minus_1 = t
        t = t + 1
        nStart(t) = 0
        NodesNum(t) = 0
        ''' define nodes for the next level
        For i = nStart(t_minus_1) + 1 To NodesNum(t_minus_1)
          If arr(level_nodes(t_minus_1, nStart(t_minus_1)), _
             level_nodes(t_minus_1, i)) Then
            NodesNum(t) = NodesNum(t) + 1
            level_nodes(t, NodesNum(t)) = level_nodes(t_minus_1, i)
          End If
        Next
        If NodesNum(t) = 0 Then
          t = t - 1
          If t > mnMaxClique Then
            mnMaxClique = t
          End If
        End If

      Else
        t = t - 1
      End If
    End If

  Wend

  ''' return size of maximum clique
  Start = mnMaxClique
End Function
```

```
Private Function LevelDegree() As Long
 Dim res As Long, i As Long, nClass As Long, aClass As Long

   For i = nStart(t) To NodesNum(t)
    ''' for node on level define class (moClasses) and mark it as existing
    nClass = moClasses(level_nodes(t, i))
    If nClass <> aClass Then
     aClass = nClass
     res = res + 1
    End If
   Next

 LevelDegree = res
End Function

Private Sub DefineClasses()
 Dim class_init() As Boolean '' show if node exist
 Dim i As Long, k As Long
 Dim mnRemainNodes As Long, bFirstNode As Boolean, nkNode As Long
 Dim nNodeNum As Long

 mnClassesCount = 0
 ReDim class_init(1 To Nodes)
 ''' get info about existing nodes
 ReDim moClasses(1 To Nodes)
 ''''
 mnRemainNodes = Nodes
 While True
   ''' build up new class
   mnClassesCount = mnClassesCount + 1
   bFirstNode = True
   ''' position of first node
   i = mnRemainNodes
   While i > 0
    ''' swap nodes
    nNodeNum = level_nodes(1, i)
    If i <> mnRemainNodes Then
     ''' swap rows
     level_nodes(1, i) = level_nodes(1, mnRemainNodes)
     level_nodes(1, mnRemainNodes) = nNodeNum
    End If
    '''
    moClasses(nNodeNum) = mnClassesCount

    mnRemainNodes = mnRemainNodes - 1
    If mnRemainNodes = 0 Then Exit Sub
```

```
      If bFirstNode Then
        For k = 1 To mnRemainNodes
          nkNode = level_nodes(1, k)
          class_init(nkNode) = arr(nNodeNum, nkNode)
        Next
      Else
        For k = 1 To mnRemainNodes
          nkNode = level_nodes(1, k)
          class_init(nkNode) = arr(nNodeNum, nkNode) Or class_init(nkNode)
        Next
      End If
      bFirstNode = False
      For i = mnRemainNodes To 1 Step -1
        If Not class_init(level_nodes(1, i)) Then Exit For
      Next
    Wend
  Wend
End Sub
```

## VColor-BT-u

```
Option Explicit

' Optimised Carraghan, R., Pardalos, P. M. algorithm
' by using a heuristic vertex colouring classes for pruning' and a backtrack search for
' colour classes

Private moClasses() As Long, mnClassesCount As Long
Private nLevelDegree() As Long
Private level_nodes() As Long, nStart() As Long, NodesNum() As Long
Private t As Long, mnMaxClique As Long
'
Public Function Start() As Long
  Dim i As Long, t_minus_1 As Long

  ReDim level_nodes(1 To Nodes, 1 To Nodes)

  mnMaxClique = 0

  '''' each level has its own set of nodes
  For i = 1 To Nodes
    level_nodes(1, i) = i
  Next

  DefineClasses

  ReDim NodesNum(1 To mnClassesCount)
  ReDim nStart(1 To mnClassesCount)
  ReDim nLevelDegree(1 To mnClassesCount)
  NodesNum(1) = Nodes

  t = 1
  nStart(t) = 0
  ''''''''''''''''''''''''''''''''''
  While t >= 1
    nStart(t) = nStart(t) + 1

    ''' Degree control
    If NodesNum(t) < nStart(t) Then
      t = t – 1
    Else
      ''' if it is not first node (for first node degree can not be adjusted)
      ''' and prev. node class is not the same then decrease degree
      ''' (can be done since vertices are sorted
      ''' and if the cur. vertex class is not the same as for previous then prev class is
      ''' not any longer existing)
      If nStart(t) > 1 Then

        If (moClasses(level_nodes(t, nStart(t))) <> _
          moClasses(level_nodes(t, nStart(t) - 1))) Then
          nLevelDegree(t) = nLevelDegree(t) – 1
        End If

      Else

        ''' calculate degree on new depth
        nLevelDegree(t) = LevelDegree()
      End If

      If (t - 1 + nLevelDegree(t)) > mnMaxClique Then

        t_minus_1 = t
        t = t + 1
        nStart(t) = 0
        NodesNum(t) = 0

        ''' define nodes for the next level
        For i = nStart(t_minus_1) + 1 To NodesNum(t_minus_1)

          If arr(level_nodes(t_minus_1, nStart(t_minus_1)), _
            level_nodes(t_minus_1, i)) Then
            NodesNum(t) = NodesNum(t) + 1
            level_nodes(t, NodesNum(t)) = level_nodes(t_minus_1, i)
          End If

        Next

        If NodesNum(t) = 0 Then

          t = t – 1
          If t > mnMaxClique Then
            mnMaxClique = t
          End If

        End If

      Else
        t = t – 1
      End If
    End If
  Wend

  ''' return size of maximum clique
  Start = mnMaxClique

End Function
```

```
Private Function LevelDegree() As Long
  Dim res As Long, i As Long, nClass As Long, aClass As Long

    For i = nStart(t) To NodesNum(t)
      ''' for node on level define class (moClasses) and mark it as existing
      nClass = moClasses(level_nodes(t, i))
      If nClass <> aClass Then
        aClass = nClass
        res = res + 1
      End If
    Next

  LevelDegree = res
End Function

Private Sub DefineClasses()
  Dim class_init() As Boolean '' show if node exists
  Dim i As Long, k As Long
  Dim mnRemainNodes As Long, bFirstNode As Boolean, nkNode As Long
  Dim nNodeNum As Long

  mnClassesCount = 0
  ReDim class_init(1 To Nodes)
  ''' get info about existing nodes
  ReDim moClasses(1 To Nodes)
  ''''
  mnRemainNodes = Nodes
  While True
    ''' build up new class
    mnClassesCount = mnClassesCount + 1
    bFirstNode = True
    ''' position of first node
    i = mnRemainNodes
    While i > 0
      ''' swap nodes
      nNodeNum = level_nodes(1, i)
      If i <> mnRemainNodes Then
        ''' swap rows
        level_nodes(1, i) = level_nodes(1, mnRemainNodes)
        level_nodes(1, mnRemainNodes) = nNodeNum
      End If
      '''
      moClasses(nNodeNum) = mnClassesCount

      mnRemainNodes = mnRemainNodes – 1
      If mnRemainNodes = 0 Then Exit Sub
      If bFirstNode Then
        For k = 1 To mnRemainNodes
          nkNode = level_nodes(1, k)
          class_init(nkNode) = arr(nNodeNum, nkNode)
        Next
      Else
        For k = 1 To mnRemainNodes
          nkNode = level_nodes(1, k)
          class_init(nkNode) = arr(nNodeNum, nkNode) Or class_init(nkNode)
        Next
      End If
      bFirstNode = False
      For i = mnRemainNodes To 1 Step -1
        If Not class_init(level_nodes(1, i)) Then Exit For
      Next
    Wend
  Wend
End Sub
```

## *Weighted Case*

## VColor-BT-w

```
Option Explicit

' Optimised Carraghan, R., Pardalos, P. M.
' by using a heuristic vertex colouring and a backtrack search

Private moClasses() As Long, mnClassesCount As Long
Private level_nodes() As Long, nStart() As Long, NodesNum() As Long ' number of
nodes on level
Private t As Long, mnMaxClique As Long
Private nLevelWAcc() As Long
Private nLevelDegree() As Long
Private nMaxCliques() As Long
'

Public Function Start() As Long
  Dim i As Long, t_minus_1 As Long, nn As Long, wt As Long

  ReDim level_nodes(1 To Nodes, 1 To Nodes, 0 To 1)
  ReDim nMaxCliques(1 To Nodes)
  """ each level has its own set of nodes
  For i = 1 To Nodes
    level_nodes(1, i, 0) = i
  Next
  ''''''''''''''''''''''''''''''''
  DefineClasses
  ResortByWeights
  For i = 1 To Nodes
    level_nodes(1, i, 1) = i
  Next
  ReDim NodesNum(1 To mnClassesCount)
  ReDim nStart(1 To mnClassesCount)
  ReDim nLevelDegree(1 To mnClassesCount)
  ReDim nLevelWAcc(1 To mnClassesCount)
  NodesNum(1) = Nodes

  For nn = Nodes To 1 Step -1
    t = 2
    NodesNum(t) = 0
    nLevelWAcc(t) = w(level_nodes(1, nn, 0))
    For i = nn + 1 To Nodes
      If arr(level_nodes(1, nn, 0), level_nodes(1, i, 0)) Then
    NodesNum(t) = NodesNum(t) + 1
    level_nodes(t, NodesNum(t), 0) = level_nodes(1, i, 0)
    level_nodes(t, NodesNum(t), 1) = level_nodes(1, i, 1)
  End If
Next
If NodesNum(t) = 0 Then
  t = t - 1
  If nLevelWAcc(t + 1) > mnMaxClique Then
    mnMaxClique = nLevelWAcc(t + 1)
  End If
Else
  nStart(t) = 0
End If

While t >= 2
  nStart(t) = nStart(t) + 1

  If NodesNum(t) < nStart(t) Then
    t = t - 1
  Else
    If (nLevelWAcc(t) + nMaxCliques(level_nodes(t, nStart(t), 1))) > _
       mnMaxClique Then

      If nStart(t) > 1 Then
        If (moClasses(level_nodes(t, nStart(t), 0)) <> _
                    moClasses(level_nodes(t, nStart(t) - 1, 0))) Then
            nLevelDegree(t) = nLevelDegree(t) - w(level_nodes(t, nStart(t) - 1, 0))
        Else
            nLevelDegree(t) = nLevelDegree(t) – (w(level_nodes(t, nStart(t) - 1, 0)) _
               + w(level_nodes(t, nStart(t), 0)) )
        End If
      Else
        ''' calculate degree on new depth
        nLevelDegree(t) = LevelDegree()
      End If

      If (nLevelWAcc(t) + nLevelDegree(t)) > mnMaxClique Then
        t_minus_1 = t
        t = t + 1
        nStart(t) = 0
        NodesNum(t) = 0
        nLevelWAcc(t) = nLevelWAcc(t_minus_1) + _
            w(level_nodes(t_minus_1, nStart(t_minus_1), 0))
        ''' define nodes for the next level
        For i = nStart(t_minus_1) + 1 To NodesNum(t_minus_1)
          If arr(level_nodes(t_minus_1, nStart(t_minus_1), 0), _
             level_nodes(t_minus_1, i, 0)) Then
             NodesNum(t) = NodesNum(t) + 1
             level_nodes(t, NodesNum(t), 0) = level_nodes(t_minus_1, i, 0)
             level_nodes(t, NodesNum(t), 1) = level_nodes(t_minus_1, i, 1)
           End If
         Next
         If NodesNum(t) = 0 Then
           t = t - 1
           If nLevelWAcc(t + 1) > mnMaxClique Then
             mnMaxClique = nLevelWAcc(t + 1)
           End If
         End If
       Else
         t = t - 1
       End If
     Else
       t = t - 1
     End If
   End If
  Wend
  nMaxCliques(nn) = mnMaxClique
 Next

 ''' return size of maximu clique
 Start = mnMaxClique
End Function

Private Sub DefineClasses()
 Dim class_init() As Boolean '' show if node exists
 Dim i As Long, k As Long
 Dim mnRemainNodes As Long, bFirstNode As Boolean, nkNode As Long
 Dim nNodeNum As Long

 mnClassesCount = 0
 ReDim class_init(1 To Nodes)
 ''' get info about existing nodes
 ReDim moClasses(1 To Nodes)
 ''''
 mnRemainNodes = Nodes
 While True
  ''' build up new class
  mnClassesCount = mnClassesCount + 1
  bFirstNode = True
  ''' position of first node
  i = mnRemainNodes
  While i > 0
   ''' swap nodes
   nNodeNum = level_nodes(1, i, 0)
```

```
    If i <> mnRemainNodes Then
      ''' swap rows
      level_nodes(1, i, 0) = level_nodes(1, mnRemainNodes, 0)
      level_nodes(1, mnRemainNodes, 0) = nNodeNum
    End If
    '''
    moClasses(nNodeNum) = mnClassesCount

    mnRemainNodes = mnRemainNodes - 1
    If mnRemainNodes = 0 Then Exit Sub
    If bFirstNode Then
      For k = 1 To mnRemainNodes
        nkNode = level_nodes(1, k, 0)
        class_init(nkNode) = arr(nNodeNum, nkNode)
      Next
    Else
      For k = 1 To mnRemainNodes
        nkNode = level_nodes(1, k, 0)
        class_init(nkNode) = arr(nNodeNum, nkNode) Or class_init(nkNode)
      Next
    End If
    bFirstNode = False
    For i = mnRemainNodes To 1 Step -1
      If Not class_init(level_nodes(1, i, 0)) Then Exit For
    Next
   Wend
  Wend
End Sub

Private Function LevelDegree() As Long
  Dim res As Long, i As Long, nClass As Long, aClass As Long

   For i = NodesNum(t) To nStart(t) Step -1
    ''' for node on level define class (moClasses) and mark it as existing
    nClass = moClasses(level_nodes(t, i, 0))
    If nClass <> aClass Then
      res = res + w(level_nodes(t, i, 0))
      aClass = nClass
    End If
   Next

  LevelDegree = res

End Function
```

```
Public Sub ResortByWeights()
  Dim i As Long, j As Long, maxi As Long, maxw As Long, aClass As Long
  Dim nNode As Long
  For i = Nodes To 2 Step -1
    maxi = i
    nNode = level_nodes(1, maxi, 0)
    maxw = w(nNode)
    aClass = moClasses(nNode)
    For j = i - 1 To 1 Step -1
      nNode = level_nodes(1, j, 0)
      If moClasses(nNode) <> aClass Then Exit For
      If maxw < w(nNode) Then
        maxi = j
        maxw = w(nNode)
      End If
    Next
    If i <> maxi Then
      nNode = level_nodes(1, i, 0)
      level_nodes(1, i, 0) = level_nodes(1, maxi, 0)
      level_nodes(1, maxi, 0) = nNode
    End If
  Next
End Sub
```

# APPENDIX B: CURRICULUM VITAE

1. Personal Data

| | |
|---|---|
| Name: | Deniss Kumlander |
| Date of birth and place: | 29.08.1976, Ivangorod, Russia |
| Citizenship: | Estonian |
| Marital status: | Married |
| Children: | - |

2. Contact Data

| | |
|---|---|
| Address: | pk 617, Tallinn 26, 12602 |
| Phone: | +372 5175942 |
| E-mail: | kumlander@gmail.com |

3. Education

| *Educational Institution* | *Graduation time* | *Speciality / grade* |
|---|---|---|
| Tallinn Technical University | 1998 | Informatics / Graduate Engineer (*cum laude*) |
| Tallinn Technical University | 1999 | Informatics / Master of Science in Engineering |

4. Language Skills (basic, intermediate or high level)

| *Language* | *Level* |
|---|---|
| Estonian | High Level |
| English | High Level |
| Russian | Mother tongue |
| Swedish | Basic Level |

5. Special Courses

| *Course and time* | *Educational institution or organisation* |
|---|---|
| Business Administration (organisation and management) 02-03.2003 | Tallinn Technical University |

6. Professional employment

| *Period* | *Institution* | *Position* |
|---|---|---|
| 08/1996 - 11/1997 | Nordica Insurance Co. | Software Engineer |
| 11/1997 - 06/1999 | Claims Handling Bureau Ltd | IT Manager |
| 06/1999 - | Simple Concepts Estonia | Project Manager |
| 10/2004 - 05/2005 | Tallinn University of Technology | Extraordinary Assistant |
| 09/2005 - | Tallinn University of Technology | Extraordinary Assistant |

8. Scientific Work

D. Kumlander, Algorithms Intelligence in the Maximum Clique Finding, OR2005: Program and Abstracts book, Bremen, 2005, p. 87

D. Kumlander, Problems of optimization: an exact algorithm for finding a maximum clique optimized for dense graphs, Proceedings of the Estonian Academy of Sciences, vol. 54 No. 2 2005, p. 79-86

D. Kumlander, Providing a Correct Software Design in an Environment with Some Set of Restrictions in a Communication between Product Managers and Designers, Proceedings of the Fourteenth International Conference on Information Systems Development: Pre-Conference, Karlstad University Studies, 2005, p. 1-11

D. Kumlander, A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search, Proceedings of The Forth International Conference on Engineering Computational Technology, Civil-Comp Press, 2004, p. 137-138

D. Kumlander, An exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring, Modelling, Computation and Optimization in Information Systems and Management Sciences (edited by Le Thi Hoai An & Pham Dinh Tao), Hermes Science Publishing, 2004, p. 202-208

D. Kumlander, Problems of optimization: an exact algorithm for finding a maximum clique optimized for the dense graphs, ISMP2003: Program and Abstracts book, Copenhagen, 2003, p. 114

9. Theses Accomplished and Defended

Graduate Engineer Thesis (1998): Maximum clique finding.

M. Sc. Thesis (1999): Maximum clique finding from arbitrary undirected graphs.

|  |  |
|---|---|
| 10. Research Interests | Maximum clique, Graph theory, NP-hard problems, Info systems. |
| 11. Research Projects | – |

Signature: Date:

1. Isikuandmed

|  |  |
|---|---|
| Ees- ja perekonnanimi: | Deniss Kumlander |
| Sünniaeg ja –koht: | 29.08.1976, Jaanilinn, Venemaa |
| Kodakondsus: | Eesti |
| Perekonnaseis: | Abielus |
| Lapsed: | pole |

2. Kontaktandmed

|  |  |
|---|---|
| Aadress: | pk 617, Tallinn 26, 12602 |
| Telefon: | +372 5175942 |
| E-posti aadress: | kumlander@gmail.com |

3. Hariduskäik

| *Õppeasutus (nimetus lõpetamise ajal)* | *Lõpetamise aeg* | *Haridus (eriala/kraad)* |
|---|---|---|
| Tallinna Tehnikaülikool | 1998 | Informaatika / Insener (*cum laude*) |
| Tallinna Tehnikaülikool | 1999 | Informaatika / Tehnikateaduste magister |

4. Keelteoskus (alg-, kesk- või kõrgtase)

| *Keel* | *Tase* |
|---|---|
| Eesti | Kõrgtase |
| Inglise | Kõrgtase |
| Vene | Emakeel |
| Rootsi | Algtase |

5. Täiendõpe

| *Kursus ja õppimise aeg* | *Õppeasutuse või muu organisatsiooni nimetus* |
|---|---|
| Ärrikorraldus (organisatsioon ja juhtimine) 02-03.2003 | Tallinna Tehnikaülikool |

6. Teenistuskäik

| *Töötamise aeg* | *Ülikooli, teadusasutuse või muu organisatsiooni nimetus* | *Ametikoht* |
|---|---|---|
| 08/1996 – 11/1997 | Nordika Kindlustus AS | Programmeerija |
| 11/1997 – 06/1999 | Kahjukäsitsus AS | IT juht |
| 06/1999 – | Simple Concepts Eesti OÜ | Projektijuht |
| 10/2004 – 05/2005 | Tallinna Tehnikaülikool | Erakorraline teadur |
| 09/2005 – | Tallinna Tehnikaülikool | Erakorraline teadur |

8. Teadustegevus

D. Kumlander, Algorithms Intelligence in the Maximum Clique Finding, OR2005: Program and Abstracts book, Bremen, 2005, p. 87

D. Kumlander, Problems of optimization: an exact algorithm for finding a maximum clique optimized for dense graphs, Proceedings of the Estonian Academy of Sciences, vol. 54 No. 2 2005, p. 79-86

D. Kumlander, Providing a Correct Software Design in an Environment with Some Set of Restrictions in a Communication between Product Managers and Designers, Proceedings of the Fourteenth International Conference on Information Systems Development: Pre-Conference, Karlstad University Studies, 2005, p. 1-11

D. Kumlander, A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search, Proceedings of The Forth International Conference on Engineering Computational Technology, Civil-Comp Press, 2004, p. 137-138

D. Kumlander, An exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring, Modelling, Computation and Optimization in Information Systems and Management Sciences (edited by Le Thi Hoai An & Pham Dinh Tao), Hermes Science Publishing, 2004, p. 202-208

D. Kumlander, Problems of optimization: an exact algorithm for finding a maximum clique optimized for the dense graphs, ISMP2003: Program and Abstracts book, Copenhagen, 2003, p. 114

9. Kaitstud lõputööd

Diplomitöö (1998): Suurima kliki leidmine.

Magistritöö (1999): Lõplikest lihtgraafidest suurima kliki leidmine.

|  |  |
|---|---|
| 12. Teadustöö põhisuunad | Suurima kliki ülesanne, graafi teooria, NP-keerulised ülesanned, infosüsteemid. |
| 10. Teised uurimisprojektid | - |

Allkiri: Kuupäev: