TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Reversed Search Maximum Clique Algorithm Based on Recoloring

Magistritöö

Üliõpilane: Aleksandr Porošin
Üliõpilaskood: 121872 IAPM
Juhendaja: Deniss Kumlander

Tallinn
2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---------- ----------

(*kuupäev*) (*allkiri*)

Ülevaade

Tänapäeval on terve rida pobleeme, mille lahendamine pole sugugi lihtne ning neile lahenduse leidmine nõuab liiga palju aega. Enamik selliseid probleeme on pärit matemaatikast ja informaatikast; neid teatakse kui NP-täielikke probleeme ning need suunatakse edasi graafiteooria probleemideks. Graafiteooria kohaselt võib ära jätta kõik ebaolulised üksikasjad ning keskenduda probleemi juurele, kasutades selleks graafe – erilisi objektide vahelisi seoseid kujutavaid skeeme.

See töö keskendub suurima kliki leidmisele orienteerimata ja kaalumata graafidest. Suurima kliki probleem on üks enamlevinud NP täielikest probleemidest, kõige komplitseeritumatest NP liigi probleemidest. Paljud muud probleemid saab teisendada klikiprobleemideks, mistõttu nende lahendamine või vähemalt kiirema algoritmi leidmine kliki jaoks aitab automaatselt lahendada palju muid ülesandeid.

See tees algab graafiteooria põhikontseptsiooni kirjeldamisest, et anda lühisissejuhatus põhiteemale. Pärast seda kirjeldatakse mõningaid täpseid algoritme suurima kliki leidmiseks. On teada-tuntud fakt, et paljud harude ja tõkete algoritmid (mida kasutatakse suurima kliki leidmiseks) on muutunud paremaks neile kohaldatud erineva heuristika abil. Seetõttu on uuritud ka mõnesid heuristikaid graafide värvimiseks, sõltumatu hulga ja tippude katmise leidmiseks. Seejärel esitleti vägagi paljulubavaid moodsaid ja tõhusaid algoritme, mis tutuvustavad erinevaid ideid paremaks ja kiiremaks kliki leidmiseks.

Sellele teesile on põhiliselt kaasa aidanud uus täpne algoritm suurima kliki leidmiseks, mis toimib kiiremini kui ükski senine algoritm, ja seda väga laia valiku graafide puhul. Põhiidee on ühendada rida tõhusaid täiustusi erinevatest algoritmidest üheks uueks algoritmiks. Esmapilgul ei pruugi need täiustused koos toimida, kuid uus lähenemisviis, mis jätab ära tippude edasise avardumise harude kärpimise asemel võimaldab nende uuenduste kasutamist ühes algoritmis. Edaspidi tuleb samm-sammulisi näiteid koos selgitustega, mis demonstreerivad, kuidas kavandatud algoritmi kasutada.

Lõpuks kõiki algoritme omavahel võrreldatakse graafide juhusliku genereerimise teel ja DIMACS'i näited tõestavad, et uus algoritm leiab suurima kliki kiiremini kui ükski teine tihedustel alla 75%. On ka muid paljulubavaid ideid, mille kohaselt eelpoolkirjeldatu on hea

teema tulevaste uurimustööde jaoks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 99 leheküljel, 6 peatükki, 40 kujundit, 5 tabelit ja 1 lisa.

Abstract

A wide variety of problems nowadays cannot be solved easily and these problems require too much time to find a solution. Most of such problems come from mathematics and computer science and are known as NP-complete problems and they were abstracted into graph theory problems. Graph theory allows removing all insignificant details and focusing on the root of a problem using graphs, special representations of objects and relationships between them.

This work concentrates on finding maximum clique from undirected and unweighted graphs. Maximum clique problem is one of the most known NP-complete problems, the most complex problems of NP class. Many other problems can be transformed into clique problem, therefore solving or at least finding a faster algorithm for finding clique will automatically help to solve lots of other tasks.

This thesis starts from describing basic concepts of graph theory to introduce the main topic. After that, some basic exact algorithms for finding maximum clique are described. It is a well-known fact that many branch-and-bound algorithms (which are used for maximum clique finding) are improved by different heuristic applied to them. Due to this, some heuristic for graph coloring, independent set and vertex cover finding is studied as well. Thereafter most promising and efficient modern algorithms are presented, which introduce different ideas for improving and fastening clique finding.

The main contribution of this thesis is a new exact algorithm for finding maximum clique, which works faster than any currently existing algorithm on a wide variety of graphs. The main idea is to combine a number of efficient improvements from different algorithms into a new one. At first sight, these improvements cannot cooperate, but a new approach of skipping vertices from further expanding instead of pruning the whole branch allows to use all the upgrades at ones. There will be some step-by-step examples with explanations, which demonstrate how to use a proposed algorithm.

At last, all algorithms are compared to each other on randomly generated graphs and DIMACS instances therefore proving the new algorithm finding maximum clique faster than any other on densities lower than 75%. There are also some promising ideas stated that might

be a good themes for future research works.

The thesis is in English and contains 99 pages of text, 6 chapters, 40 figures, 5 tables and 1 appendix.

Abbreviations glossary

CBC	Current best clique. Abbreviation used in multiple algorithms to define an array for storing the largest clique vertices found by far. Sometimes it is used as |CBC| that means the number of vertices contained in a current best clique.
DIMACS	Center for Discrete Mathematics and Theoretical Computer Science. Presents a pack of benchmarks instances, which represent different graphs, constructed on the real life problem basis. These instances can be used for testing maximum clique algorithms performance.
ILS	Iterated local search algorithm. Heuristic algorithm for searching a better solution by applying different improvements to already existing heuristic solution. In this thesis, ILS abbreviation is applied to particular algorithm for finding maximum independent set [Andrade, Resende, Werneck 2012].
MCSI	MCS Improved algorithm. Exact maximum clique algorithm, successor of MCS, presented in 2014 by four authors [Batsyn, Goldengorin, Maslov, Pardalos 2014].
MCQ, MCR, MCS	Exact maximum clique algorithms published by Tomita and his colleagues [Tomita, Seki 2003] [Tomita, Kameda 2007] [Tomita, Sutani, Higashi, Takahashi, Wakatsuki 2010]. Each algorithm is a successor of the previous one and adds some improvements for fastening search.
MDG	Maximum degree greedy algorithm. Heuristic algorithm for finding maximum vertex cover published by Clarkson [Clarkson 1983].
MIS	Maximum independent set problem. The problem of finding the largest possible edgeless subgraph of a given graph.

NP complexity class	Nondeterministic polynomial time complexity class. Class of problems that can be solved with a polynomial amount of time by nondeterministic Turing machine.
P complexity class	Polynomial time complexity class. Class of problems that can be solved with a polynomial amount of time by deterministic Turing machine.
VColor-BT-u	Vertex color with backtracking for unweighted cases. Exact maximum clique finding algorithm published by D. Kumlander [Kumlander 2005] based on Östergård's algorithm. The main idea is to apply vertex coloring with backtracking for fastening maximum clique finding.
VColor-u	Vertex color for unweighted cases. Exact maximum clique finding algorithm published by D. Kumlander [Kumlander 2005] based on Carraghan and Pardalos algorithm [Carraghan, Pardalos 1990]. The main idea is to apply vertex coloring for fastening maximum clique finding.
VRecolor-BT-u	Vertex recolor with backtracking for unweighted cases. A new exact algorithm presented in the current thesis based on VColor-BT-u. The main idea is to apply additional in depth coloring (recoloring) to fasten maximum clique search.

List of tables

List of figures

Table of Contents

1. Introduction

1.1 Graph theory

Graph theory is a study of graphs which is the main topic of this work, it can be used as a tool that helps scientists to transform real life problem into special representations i.e. graphs. This process allows omitting unnecessary details, relaxing a problem, and concentrating on the source of the problem. Number of applications and algorithms for solving different types of problems within graph theory area is growing very fast, so a lot of tasks can be converted into already solved ones, which let people optimize and ease their daily life.

When working with graphs it is often convenient to imagine a graph as a diagram, which represents objects as a vertices or points and relationships between these objects are depicted as edges or lines joining the two relevant points. Let us assume that we need to organize a timetable in the airport. A number of aircrafts should be assigned to multiple flights in a set period of time. If two flights overlap, then it is not possible to assign one airplane to both flights. This problem can be transformed into a graph. We indicate each flight as a vertex and if two flights overlap then corresponding vertices will be connected to each other. When a real problem is modeled as a graph, we are going to solve it using already existing techniques, in our case it is a graph coloring. We need to assign a label i.e. color to each vertex in a manner that no two connected points share the same color. As a result, gained number of colors will show how many aircrafts we will need to organize all the flights.

To make things clear we are going to demonstrate a well-known example making use of graphs. In the 18^{th} century, there were seven bridges in the town of Königsberg. Residents were interested whether it is possible to cross all the bridges with one walk without recrossing the same road multiple times. Figures 1.1 and 1.2 show how the situation in this town is represented by a graph. Vertices are treated as land areas and the two vertices a connected by a number of edges equal to the number of bridges between corresponding lands. After this

transformation, the problem is narrowed to the question: is it possible to find a trail containing all the edges?

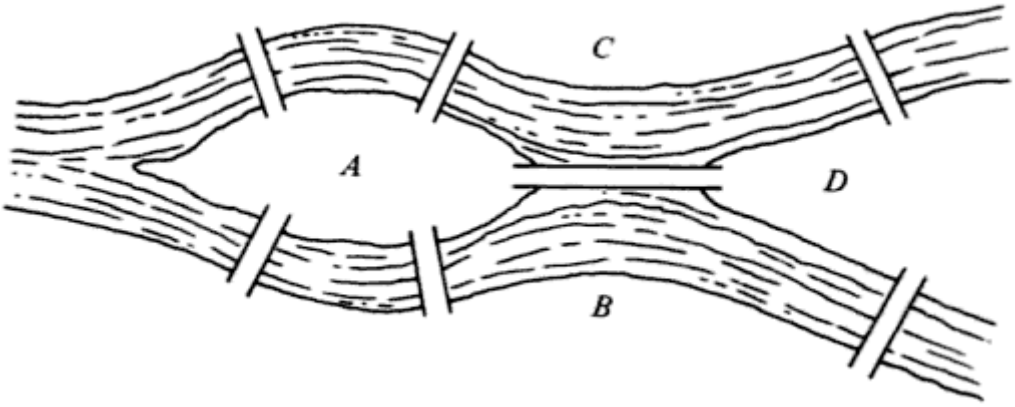

Figure 1.1 The Königsberg Bridge Problem. Map representation [Chartrand 1985]

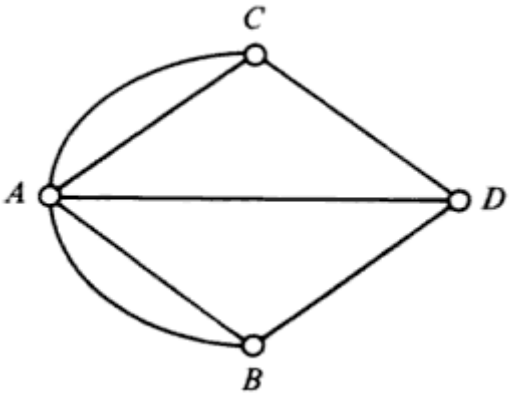

Figure 1.2 The Königsberg Bridge Problem. Graph representation [Chartrand 1985]

Swiss mathematician Leonhard Euler (1707-1783) solved the Königsberg Bridge Problem and it gave an answer to various different puzzles, mazes, and tasks that were similar to this problem. The same way nowadays, graph theory allows solving problems from multiple areas like computer science, sociology, medicine, biology and so on. Studying the root problems of graph theory is important not only for a particular problem by itself but for all connected areas.

1.2 Preliminaries

A graph G is a representation of objects, which is a set of vertices V, and a number of relationships between these objects, called edges i.e. a set of edges E. The order of G is a number of vertices in G and the number of edges is called the size of G. Therefore, order is $|V|$ and $|E|$ is equal to size of G. If two vertices u and v are connected to each other they are called adjacent $e = \{u, v\} \in E(G)$ and u and v are both incident to e. If $e \neq \{u, v\} \in E(G)$ then u and v are nonadjacent. It is essential on what position each vertex is located and by what lines (straight, curve) adjacent vertices are connected. The only crucial point is a fact that some vertices are connected. Figure 1.3 demonstrates exactly the same graph G, which might look different when vertices are relocated and curved lines used instead of straight ones. Both diagrams represent exactly the same set of vertices and set of edges, so they describe the same graph.

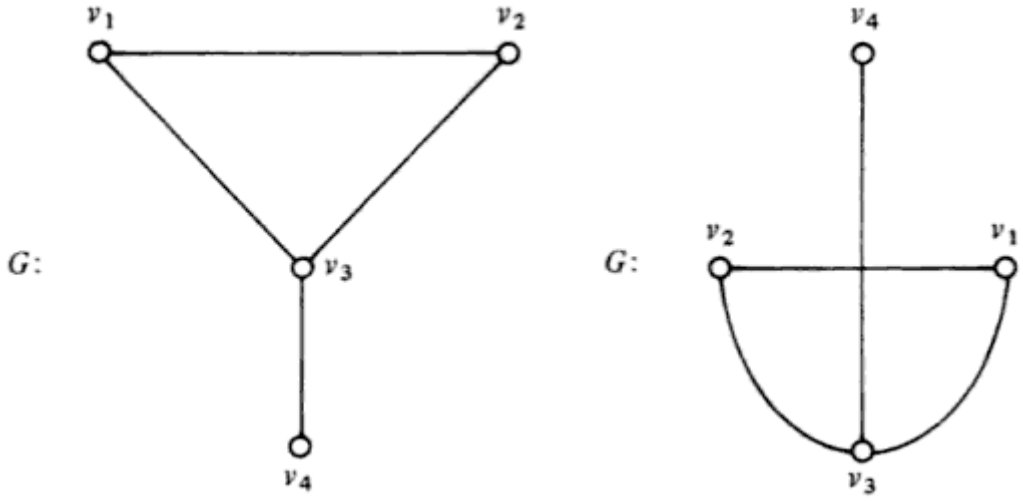

Figure 1.3 Different ways to draw the same graph G [Chartrand 1985]

The number of adjacent vertices or neighbors of a vertex is called vertex degree *deg(v).* Vertex can be called even or odd if its degree is even or odd. The maximum vertex degree of a graph G is denoted $\Delta(G)$. Vertex support is a sum of degrees of all neighbors of a given vertex. As we can see from figure 1.4 degree of v_3 is four and support of v_3 is equal to five.

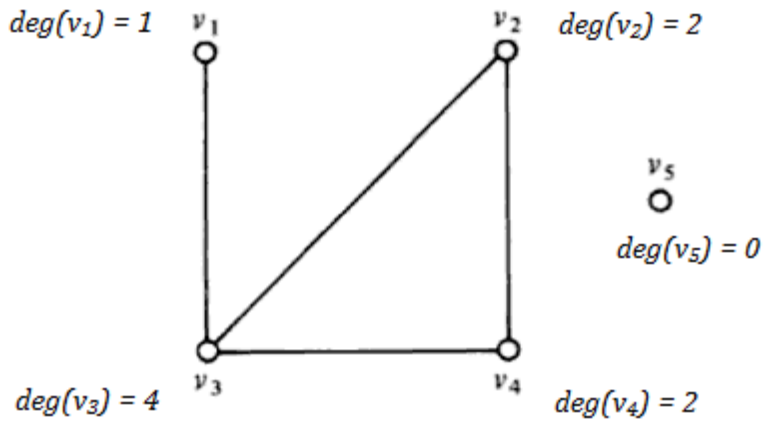

Figure 1.4 Degrees of vertices

Graphs can be divided into directed and undirected. A directed graph D i.e. digraph has non-symmetric arcs (directed edge is called arc), which means that vertex u can has relation to vertex v, but there might not be relation from v to u. From the other hand, directed graph always has symmetric relation between two vertices. Moreover, graphs are divided to weighted and unweighted. Weight is a number (generally non-negative integer) assigned to each edge or vertex that can represent additional property like length of a route, cost, required power, etc. depending on the problem context. On the opposite side, unweighted graph does not have weights or, in other words, all their weights are equal to one. Loop is an edge that connects a vertex to itself. Simple graph is an undirected graph that does not contain any loops and there is no more than one edge connecting two vertices. It should be noted that in this paper we are studying only unweighted simple graphs.

An undirected graph where all the vertices are adjacent to each other is called complete. Otherwise, a graph with no edges is called edgeless, in other words no two vertices are adjacent to each other. A clique is a complete subgraph of a graph G and an independent set is an edgeless subgraph of G. Complement graph G' of a simple graph G is a graph that has

the same vertex set, but the edge set consists only from vertices that are not present G. $G' = (V, K \backslash E)$, where K is the edge set consisting from all possible edges. Vertex cover of a graph G is a vertex set such that each edge of G is incident to at least one vertex from this set. Graph coloring is process of assigning labels i.e. colors to vertices with a special property that no two adjacent vertices can share the same color. A color class is a set of vertices containing vertices with the same color. It is clearly seen from coloring property that each color class is nothing more than an independent set. Graph is called k-colorable if it can be colored into k colors. The minimum number of colors required for coloring a graph G is called the chromatic number - $\chi(G)$ and in this case graph is called k-chromatic.

There are multiple problems stated from the definitions listed above. They are the following:

- Maximum clique problem – a problem of finding maximum possible complete subgraph of a graph G.
- Independent set problem – a problem of finding maximum possible edgeless subgraph of a graph G.
- Minimum vertex cover – a problem of finding the smallest possible vertex cover of a graph G.
- Graph coloring - a problem of coloring a graph with the least possible number of colors.

All the described problems are computationally equivalent and one problem can be transformed into another one. For instance, a clique of graph G is an independent set of a complement graph G' and a vertex cover of G' is a set containing all vertices of G', except those who belong to the found independent set. That means a clique problem can be transformed into an independent set problem and to a vertex cover problem.

All these problems are NP-Complete which means that there is no polynomial time algorithm can be found. On the other hand, there are heuristic algorithms that give a solution within polynomial time, but this solution is not guaranteed to be the best one (maximum or minimum depending on a problem). Heuristic algorithms are widely used to quickly gain additional information and perform a short analysis of a graph like defining independent sets or initializing upper and lower bounds.

1.3 Complexity

1.3.1 Complexity functions

Algorithm is a step-by-step procedure for solving different problems. We say that an algorithm solves the problem if it produces a guaranteed solution for any instance of the given problem. This means we cannot state that there is an algorithm, which completely solves maximum clique problem unless it will always give the maximum possible clique on any graph. As a result, we face some problems that cannot be solved easily. These problems are called NP-complete problems, in other words they are very hard to solve.

A function $f(n)$, where n is a size of its input, is said to have complexity $O(g(n))$ if there exists a constant c such that $|f(n)| \leq c * |g(n)|$ for each $n \geq 0$. An algorithm with time complexity function $O(p(n))$, where p is a polynomial time function with input length n, is called polynomial time algorithm. All other algorithms which complexity functions cannot be bounded this way are called exponential time algorithms. The definition of exponential algorithms also includes some non-polynomial time complexity functions, which are neither polynomial nor exponential, for example $n^{log(n)}$. Table 1.1 shows time consumption of different time complexity functions. It clearly seen that even a several times input length increment results in the explosive execution time growth for the exponential functions. Of course, polynomial time complexity functions are generally much more desirable than exponential ones. It should be noted that on some small inputs exponential complexity function takes less time than a polynomial one, for instance n^5 and 2^n for $n \leq 20$. Unfortunately, problems in real life are much larger than described in this table.

Table 1.1 Difference between polynomial and exponential time complexity functions. [Garey, Johnson 2003]

Time complexity function	**Size *n***					
	10	**20**	**30**	**40**	**50**	**60**
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second

$\mathbf{n^3}$	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
$\mathbf{n^5}$	.1 second	3.2 second	24.3 second	1.7 minutes	5.2 minutes	13.0 minutes
$\mathbf{2^n}$	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
$\mathbf{3^n}$	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

1.3.2 NP-complexity

The first serious result in algorithms complexity field were done by Alan Turing in 1940s. Turing showed that there are some "undecidable" problems. These problems are so hard that it is not possible to find an algorithm for solving them. Turing invented an abstract computer model called Turing machine. There are P class problems that can be solved with a polynomial time on a deterministic Turing machine. Problems that are solvable by non-deterministic Turing machine are NP class problems in polynomial time. It is not right to say that NP means non-polynomial and NP class problems cannot be solved on the deterministic Turing machine, because $P \subseteq NP$. All the problems of NP except P (NP-P) are not solvable by deterministic Turing machine.

The fundamentals of NP-completeness theory were published in "The Complexity of Theorem Proving Procedures" paper in 1971 [Cook 1971]. With his work, Cook presented the following important things:

- Importance of "polynomial time reducibility". That means if there is a polynomial time transformation from one problem into another, then it ensures that any polynomial time algorithm for the second problem can be converted into polynomial time algorithm for the first problem.
- Focused attention on the class NP of decision problems. A decision problem is a problem whose solution is either "yes" or "no".
- There is a "*satisfiability*" problem in NP class that has a special property. Every problem in NP can be reduced to the satisfiability problem. It means if the satisfiability problem will be solved with a polynomial time algorithm, then all the problems from NP are solvable in polynomial time. Otherwise, if it will be proved that it is not possible to solve some problem in NP with a polynomial time then

satisfiability problem does not have polynomial time solution too. As a result, satisfiability problem is the hardest problems in NP.

Now we can move to NP-complete problems. NP-complete is a class of problems that contains the "hardest" problems of NP. There is a polynomial time transformation from any problem of NP-class into NP-complete problem. It can be proven by the following algorithm that a decision problem H is NP-complete:

1. Show that the problem H is NP
2. Choose already existing NP-complete problem H' that is the most identical to H
3. Develop a transformation f from H' to H
4. Prove that f is a polynomial transformation

Nowadays scientists have found many NP-complete problems but some of them are more suitable for transforming other problems to them. These problems are used as the basic ones and all of them are decision problems. Here is a list of six basic NP-complete problems [Garey, Johnson 2003].

3-SATISFIABILITY (3SAT)

INSTANCE: Collection $C = \{c_1, c_2, \dots, c_m\}$ of clauses on a finite set U of variables such that $|c_i| = 3$ for $1 \leq i \leq m$.

QUESTION: Is there a truth assignment for U that satisfies all the clauses in C?

3-DIMENSIONAL MATCHING (3DM)

INSTANCE: A set $M \subseteq W \times X \times Y$, where W, X and Y are disjoint sets having the same number q of elements.

QUESTION: Does M contain a matching, that is, a subset $M' \subseteq M$ such that $|M'| = q$ and no two elements of M' agree in any coordinate?

VERTEX COVER (VC)

INSTANCE: A graph $G = (V, E)$ and a positive integer $K \leq |V|$.

QUESTION: Is there a *vertex cover* of size K or less for G, that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and, for each edge $\{u, v\} \in E$, at least one of u and v belongs to V'?

CLIQUE

INSTANCE: A graph $G = (V, E)$ and a positive integer $J \leq |V|$.

QUESTION: Does G contain a *clique* of size J or more, that is, a subset $V' \subseteq V$ such that $|V'| \geq J$ and every two vertices in V' are joined by edge in E?

HAMILTONIAN CIRCUIT (HC)

INSTANCE: A graph $G = (V, E)$.

QUESTION: Does G contain a Hamiltonian circuit, that is, an ordering $< v_1, v_2, \ldots, v_n >$ of the vertices of G, where $n = |V|$, such that $\{v_n, v_1\} \in E$ and $\{v_i, v_{i+1}\} \in E$ for all $i, 1 \leq i < n$?

PARTITION

INSTANCE: A finite set A and a "size" $s(a) \in Z^+$ for each $a \in A$.

QUESTION: Is there a subset $A' \subseteq A$ such that

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

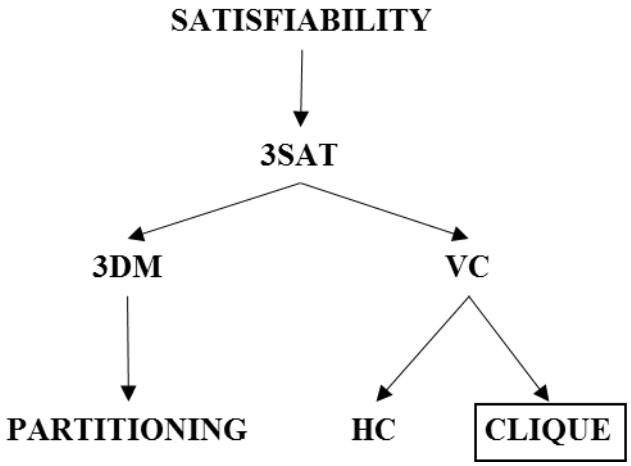

Figure 1.5 Diagram of the sequence of transformation of six basic NP-complete problems. [Garey, Johnson 2003]

As can be seen above "*Clique*" problem belongs to the basic NP-complete problems and maximum clique problem is polynomially equivalent to this. As a result solving maximum clique problem or upgrading algorithms for finding maximum clique will not only improve one specific, narrow problem but help to find better algorithms for all the problems reducible to maximum clique problem. Therefore current topic is very important.

1.4 Goals of the study

The topic of this thesis is quite extensive, so the following goals were determined to achieve a certain solution for some defined problems.

1. Implement and study modern algorithms for finding maximum clique
2. Define the most efficient and promising improvements for finding maximum clique
3. Study the influence of heuristic on exact algorithms
4. Develop a better algorithm for finding maximum clique
5. Implement testing environment to compare performance of the algorithms
6. Define if there exists some graph groups or special cases that are solved better by one or another algorithm.

1.5 Work overview

Chapter 1 of this thesis introduces the problem. Definitions and basic concepts of the studied area are presented. There is a short overview on the complexity of the problem giving explanations why current problem is valuable. After that, goals of study are identified.

Basic algorithms are described in the Chapter 2. This chapter contains exact maximum clique finding algorithms and other heuristic algorithms like coloring, maximum independent set or minimum vertex cover finding. These algorithms describe the fundamental ideas for solving maximum clique problem. Moreover, some important properties are outlined.

Chapter 3 presents different modern algorithm and shows the current state of the problem. There is a brief description of each algorithm and an overview of the results gained by the authors of those algorithms. The main focus of the chapter is to describe the implemented upgrades and analyze the impact of them on overall performance.

The main part of this thesis is demonstrated in the Chapter 4. A new algorithm for finding maximum clique is acquired. The idea of the algorithm is described giving step-by-step instructions of implementing it. After that, two examples are explained in details.

All the previously described algorithms are compared to the new algorithm in the Chapter 5. First of all, algorithms were tested on randomly generated graphs giving an overview of algorithms performance. Generated graphs are divided by their density and

presented as diagrams. After that, algorithms were tested on DIMACS benchmark instances and presented as tables analyzing time consumption and number of created branches.

Finally, Chapter 6 contains a summary of the study. Possible topics for future studies are also noted in here.

The new algorithm's code written on C# language is located in appendix. Code can be used to reproduce the algorithm exactly the way it was designed initially and avoid misunderstandings with implementation.

2. Algorithm basics

The first part of this chapter contains an overview of basic algorithms for solving maximum clique problem. These algorithms are branch and bound, but depict two different approaches of solving the clique problem. It will be clearly seen later that all modern algorithms are based on them.

There are not only clique finding algorithms but also graph coloring, maximum independent set and minimum vertex cover algorithms are included in this chapter. Other algorithms are needed to gather additional information about a graph for later use to skip unnecessary steps therefore fastening clique finding.

2.1 Basic maximum clique algorithms

2.1.1 Carraghan and Pardalos algorithm

Randy Carraghan and Panos M. Pardalos published "An exact algorithm for the maximum clique problem" article in 1990 [Carraghan, Pardalos 1990]. The main benefits of this algorithm are simplicity and efficiency. The algorithm gives basic concepts of how a clique can be found. Furthermore, even nowadays it shows relatively good results on lower density graphs.

One of the fundamental and crucial points for this is algorithm is notion of depth. Initially (depth 1) we take (expand in other words) one vertex v_1. Then, at depth 2, only vertices adjacent to v_1 are considered. We take v_2 from depth 2 and construct depth 3 from the vertices that are adjacent to v_1 and v_2 and so on. Every depth construction is creating a new branch in this branch and bound algorithm. Use of this approach leads to the fact that any vertex on the depth d is adjacent to all previously expanded vertices within current branch, giving us a clique of size d.

The second very important aspect is a good bound rule. Current best clique *(CBC)* or the biggest depth number found by far should be stored. Let d be the current depth, i - currently expanding vertex and m - number of vertices on current depth. In this case d is a current clique size and $m - i$ gives a number of vertices that not yet expanded and potentially can form clique. Obviously the biggest possible clique size in the current state is $d + (m - i)$. That means if $d + (m - i) \leq CBC$ we can prune this branch. It is not possible that current expanding vertex will give us a larger clique. Algorithm works on any depth until the pruning formula does not hold or there are some vertices to expand. If we are out of vertices or pruning formula holds on the first depth then algorithm stops.

Authors of the algorithm state that it can be improved by initially ordering vertices with response to their degrees. If we say that vertices in graph G ordered as $v_1, v_2, \ldots, v_n$ then v_1 is a vertex with the smallest degree in G, v_2 has the smallest degree in $G - \{v_1\}$ and so on. In general v_k is a vertex with smallest degree in G - $\{v_1, v_2, \ldots, v_k\}$ for $k < n - 2$. This ordering can be reapplied on depths higher than one. It lowers overall time consumed to find maximum clique, but only on dense graphs. It is advised not to use any ordering on lower densities.

This algorithm is based on a simple branch construction and efficient pruning formula. It shows great results on low densities. On the other hand, the main drawback is really poor performance on dense graphs. As long as all the vertices have a lot of connections, every branch consists of much more vertices than best clique size. As a result pruning formula is not working.

```
function Main
     CBC := 0 // the maximum clique's size
     clique (V, 0)
     return CBC
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 // prune
                return
            i := i + 1
            // form a new depth. N(vi) denotes a neighborhood of
vi.
            clique (N(vi) | ∀vj : j > i, j ≤ |V|, depth + 1)
        end while
        return
end function
```

Figure 2.1 Carraghan and Pardalos algorithm. Pseudo code [Kumlander 2005]

2.1.2 Östergård algorithm

Patrick R.J. Östergård published "A fast algorithm for the maximum clique problem" article in 2002 [Östergård 2002]. He introduced a new approach for finding maximum clique. Let $S_i = \{v_i, v_{i+1}, \dots, v_n\}$ be a subgraph processed on any depth. Previous Carraghan and Pardalos algorithm initially starts from the whole graph S_1 and considers all the vertices finding cliques in S_1that contain v_1 first. Then it searches for cliques in S_2 that contains v_2 and so on till S_n. In an Östergård's algorithm cliques are considered in reversed order starting from $S_n = \{v_n\}$. This subgraph contains only one vertex and initial clique size is one by default. Then S_{n-1} containing two vertices is being processed. Clique sizes for each subgraph S_i are stored in cache $c[i]$. Using this additional information is possible to implement a new pruning formula $d + c[i] \leq CBC$. d is a current depth, i is a vertex index currently being expanded and CBC (current best clique) is the biggest clique size found by far. The second crucial point to understand is that only one vertex is added to a new subgraph S_k compared to S_{k+1} which means that CBC potentially can be increased only by one and not more. It results in a new condition that if $d + c[i] > CBC$ we can stop further search within S_k and go to S_{k-1}.

There are some possibilities to improve algorithm performance with proper initial ordering. It is advised to use the approach as in Carraghan and Pardalos algorithm to sort vertices by their degree in increasing order, so that v_i is always a vertex with the smallest degree taken from the subgraph induced by the vertices that have not yet been ordered.

```
function Main
    max := 0
    for i := n downto 1 do
```

```
            found := false
            clique (Si & N(vi), 1)
            c[i] := max
      end for
      return max
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 ≠ Ø do
            // prune as Carraghan and Pardalos algorithm does
            if size + |U| ≤ max then
                  return
            i := min { j | vj ∈ U }
            // new pruning technique
            if size + c[i] ≤ max then
                  return
            U := U \ { vi }
            clique(U & N(vi), size + 1)
            if found = true then // stopping condition
                  return
      end while
      return
end function
```

Figure 2.2 Östergård algorithm. Pseudo code [Kumlander 2005]

2.2 Graph coloring heuristic algorithms

Graph *(G)* coloring is a graph vertices mapping to labels i.e. colors so that $V(G) \rightarrow S$, where V – set of vertices of G and S – set of colors.

A color class is a subset of V that was assigned to one color. The main coloring property is that no two adjacent vertices can obtain the same color. A graph is called *k*-

colorable if it can be colored into k colors. The least k such that a graph is k-colorable is a chromatic number $\chi(G)$. The best coloring of G is $\chi(G)$-coloring.

Graph coloring gives us some useful properties, which will be used later in algorithms for clique finding. These properties are:

- Each color class forms an independent set. This property comes from definition of graph coloring that vertices set to one color cannot be adjacent to each other.
- Colors number is an upper bound for maximum clique, i.e. k-colorable graph cannot contain clique of size larger than k. Clique of $k+1$ size in k-colored graph means that two adjacent vertices within a clique are set to one color, which is contrary to the definition of graph coloring and lead us to improper coloring.
- Each color class formed by coloring of complement graph H of G gives a clique within G.

2.2.1 Greedy coloring algorithm

Greedy is one of the simplest heuristic coloring algorithms. This algorithm has solid benefits such as easy implementation and high performance. On the other side number of color classes is not always close to chromatic number. In general, this algorithm is a great compromise between speed and result quality. Algorithm can be described in four steps:

1. Color the first vertex in color number 1.
2. Take not yet colored vertex v_i and try to color it to the lowest numbered color k, so that there is no any adjacent vertex to v_i with the k color number.
3. If it is not possible to color a vertex v_i into any of existing colors, a new color must be created and assigned to v_i.
4. Repeat steps two and three until all the vertices are colored.

```
// n – number of vertices, k – number of colors on each step
k = 1; Color v1 with C1 (Ck)
For i := 2 to n
     Try to color vi with color Cj, where j = min (1, ... , k)
     If none color was used to color vi then
          k := k+1 [Produce a new color]
          Color vi with Ck
     End if
```

```
Next
```

Figure 2.3 Greedy coloring algorithm. Pseudo code

Greedy algorithm results heavily depend on vertex coloring order. It is clearly seen on coloring bipartite graph. When vertices from bipartite graph are ordered in a way that we firstly color all the vertices from one partite set and after from the other partite set, then this approach results in a good coloring (left graph on figure 2.4). However, if vertices are taken from different partite sets one by one it leads to the huge amount of colors (right graph on figure 2.4).

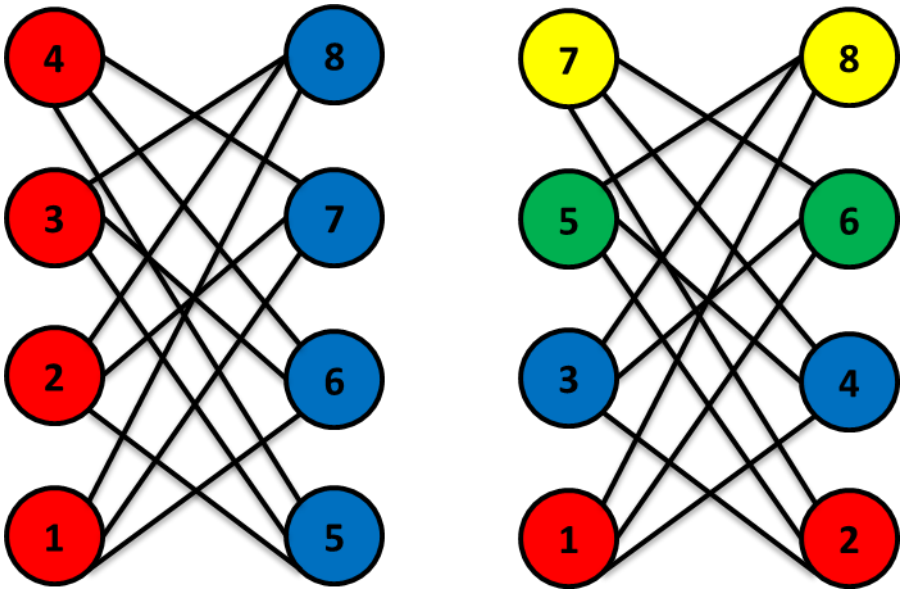

Figure 2.4 Bipartite graph coloring

There is one important point to keep in mind that with this coloring approach we take vertices in the exact same order as they were initially ordered. As a result, vertices within each color class keep initial ordering to each other. In other words if initial index $i < j$ and vertices v_i, v_j have the same color then index ki of v_i will definitely be less than index kj of v_j within k color class.

2.2.2 Greedy coloring algorithm with swaps

There is a slightly modified version of greedy coloring used in further algorithms [Kumlander 2005]. Instead of coloring, each vertex to the least possible color this algorithm tries to color all the vertices to the first color, then to the second one and so on. The main idea of this approach is to order vertices during the coloring process using vertex swaps to lower time consumption. Algorithm consists of the following steps:

1. Color the first vertex in color number k_1. Set the least not colored vertex index u (initially $u = 2$).
2. Take not yet colored vertex v_i starting from index u and try to color it to the lowest numbered color k, so that there is no any adjacent vertex to v_i with the k color number.
3. If a vertex v_i is colored swap v_i and v_u, where v_u is the least not colored vertex. Increase u by 1.
4. If all the vertices were processed and u is not bigger than total number of vertices, create a new color class and repeat steps 2 and 3.

```
// n – number of vertices, k – number of colors on each step
// u - the least not colored vertex index
k = 1; Color v1 with C1 (Ck)
u = 2
While True
     For i := u to n
          Try to color vi with color Ck
          If vi was colored then
               swap vu and vi
               u := u+1
               Color vi with Ck
```

```
            End if
      Next
      If u ≤ n Then
                  k := k+1 [Produce a new color]
      Else
            Exit While
      End If
End While
```

Figure 2.5 Greedy coloring algorithm with swaps. Pseudo code

As long as vertex swapping is used, it is not possible to maintain the same vertices ordering within color classes as in previous greedy algorithm. Current algorithm is suitable when solving maximum clique problem on low densities where initial vertex order is useless.

2.3 Maximum independent set and minimum vertex cover heuristic

Let $G = (V, E)$, where V – set of vertices and E – set of edges. Independent set problem tries to find a subset $S \subseteq V$ such that no two vertices in S are adjacent to each other i.e. S is an empty graph. Maximum independent set is the largest possible subset S in a graph.

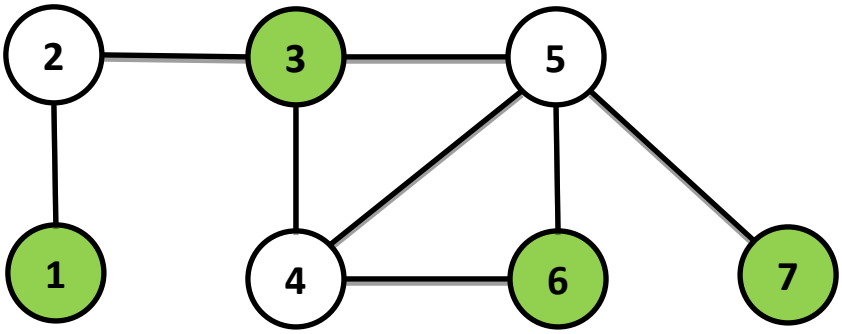

Figure 2.6 Independent set example.

Maximum independent set problem (MIS) is closely connected with maximum clique problem. Let K be a set of all possible edges between elements of V. Then the complement graph of G is $H = (V, K \setminus E)$. Each independent set in H is a clique in G from the first

property. Consequently, every independent set problem can be easily transformed into maximum clique problem and vice versa.

Heuristic independent set algorithm can be used to acquire initial clique size value for clique searching algorithms. The better heuristic is used, the closer initial clique size will be to the maximum clique size.

Vertex cover is a subset $C \subseteq V$ such that each edge in E is incident to at least one vertex in C i.e. vertices from C „*cover*“ the edges of a graph.

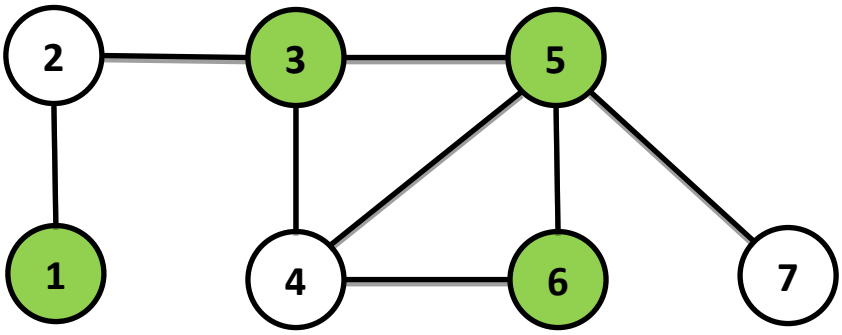

Figure 2.7 Vertex cover example

Here are some simple steps of how to obtain approximate vertex cover:

1. Take a random edge *{u, v}* from a graph
2. Add edge *{u, v}* to the current vertex cover set *C*.
3. Remove all edges incident to u or v from *E*.
4. Repeat steps from one to three until *E* is empty.

```
C = ∅ // vertex cover array
While E ≠ ∅
     take random edge {u, v} ∈ E
     C = C ∪ {u, v}
     remove all edges incident to u or v from E
End While
Return C
```

Figure 2.8 Approximate vertex cover algorithm. Pseudo code

Vertex cover algorithm has one good property that can be easily used to gain independent set quickly. When a vertex cover C is found using approximate algorithm there will be some set of vertices S left that are outside of vertex cover $S = V \setminus C$ and no edges in a graph. As a result, S is an independent set. Moreover, it means the "better" vertex cover is found the larger independent set is obtained.

2.3.1 Maximum Degree Greedy algorithm (MDG)

In the current work only heuristic vertex cover algorithms are studied. Therefore, result is influenced by random edge picks. There is a MDG algorithm [Clarkson 1983] to improve approximate vertex cover result. Random edge guessing is not a very good and reliable approach for vertex cover finding. MDG algorithm takes a vertex with the highest degree and removes all its edges. This step is repeated until there are no edges left.

```
// deg(v) - degree of vertex v
C = ∅ // vertex cover array
While E ≠ ∅
     find vertex v with maximum deg(v)
     C = C ∪ {v}
     remove all edges incident to v from E
End While
Return C
```

Figure 2.9 MDG algorithm. Pseudo code

There is an example of approximate vertex cover algorithm on figure 2.10. Red numbers mean in what order edges were chosen. In this case the result is quite bad *(C = {1, 2, 3, 5, 4, 6})*. Figure 2.11 demonstrates a result of MDG algorithm, blue numbers show in what order vertices were selected. There is a 50% reduction in a vertex cover size *(C = {5, 2, 4})*. It is clearly seen that random edge choice is not reliable technique and some more intelligent way must be implemented.

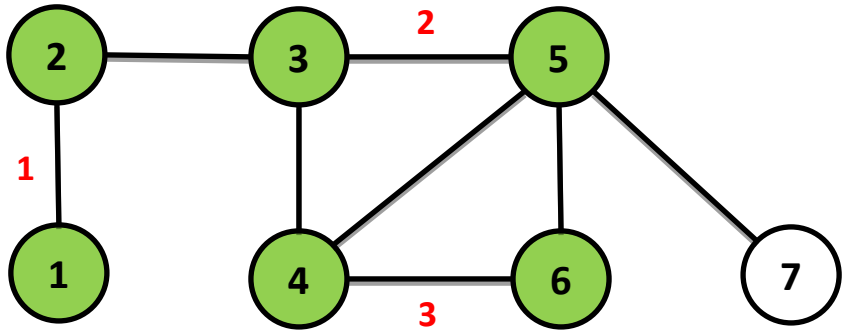

Figure 2.10 Approximate vertex cover algorithm result.

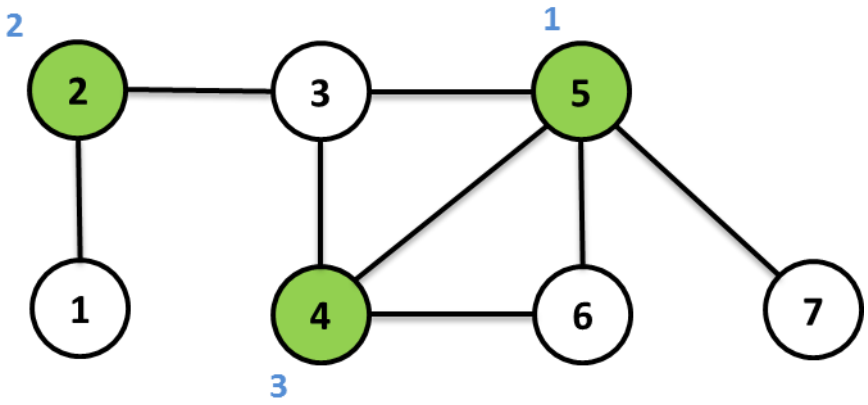

Figure 2.11 MDG algorithm result.

2.3.2 Iterated local search algorithm (ILS)

There are different heuristic algorithms to get independent set. The easiest way is to use greedy approach, which is not very good sometimes. MDG algorithm might give better solution. In general, we can try to improve already existing heuristic solution by using a notion of plateau search, which is based on vertex swaps. Swap is a replacement of one vertex by the other from its neighbors. Swap will not definitely improve the solution, but it possibly can lead to some non-solution vertices become free (without neighbors inside solution) and therefore they can be inserted into existing solution. “Fast local search for the maximum independent set problem” article [Andrade, Resende, Werneck 2012] gives a number of tools to perform swaps more efficiently and not in a random way.

Let *(j, k)* – swap will consist of removing *j* vertices from solution and inserting *k* vertices into solution. In particular, each made *(k-1, k)* – swap leads to increasing independent set by one. Let *(k-1, k)* – swap is called *k*-improvement. ILS algorithm core idea is to search for 2- and 3- improvements until no more can be found. ILS should have some initial solution to start improving it. For these purposes greedy or MDG algorithms can be used.

3. Modern algorithms

Multiple modern algorithms will be demonstrated in this chapter. All of them are based on the ideas presented in the previous topic. Old algorithms were focused on the information about adjacent vertices and almost absolutely ignored the opposite side – vertices that are nonadjacent. Modern algorithms are heavily depending on heuristic and, in particular, on vertex coloring. Two nonadjacent vertices cannot be added into one clique, therefore two vertices from different independent sets will not contain in a solution also. Graph coloring allows building several independent sets and use additional properties based on gathered information. Moreover, rationally used heuristic approaches will not increase time consumption dramatically. Unfortunately, it is not possible to use exact algorithms for finding color classes because coloring problem is NP-complete. In general, modern algorithms carry out preliminary work gathering and analyzing additional information before starting clique searching or, more precisely, branch processing.

3.1 VColor-u

Deniss Kumlander published "Some Practical Algorithms to Solve The Maximum Clique Problem" thesis in 2005 [Kumlander 2005] introducing VColor-u algorithm (Vertex Color unweighted). The core idea was to demonstrate efficiency of using independent sets within clique finding algorithms.

Let G_d be a subgraph of G on the depth d and V_d is a set of vertices of G_d. Each time a vertex from V_d is expanded and a branch created from it is analyzed it is then removed from a current depth d and the next vertex will be expanded. From this property, Carraghan and Pardalos created their pruning formula. D. Kumlander modified the formula to be $d-1+Degree(G_d) \leq CBC$, where $d-1$ is the number of vertices, which were expanded prior to *d-th* depth, and form current clique, CBC is currently biggest clique size found and $Degree(G_d)$ is a function that gives number of colors of G_d. The main point of this method

is that a graph cannot contain clique larger than the number of color classes obtained by coloring this graph.

It should also be noted that graph is being colored only once in the beginning of the algorithm and later on a degree of subgraph G_d is counted by specific approach. It would take too much time to recount number of colors each time a new branch is created. As long as and this order is not changed during the whole process, it is simpler to count number of color classes when a new depth is formed. Later on if a vertex on the same depth is expanded and it is from the same color class then degree is not changed. If color classes are different then the degree is decreased by one.

```
function Main
     Heuristic vertex coloring
     Order vertices that first color classes have the last
indexes
     CBC := 0 // the maximum clique's size
     clique (V, 1)
     return CBC
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 - 1 + degree(V) ≤ CBC then // prune
               return
          i := i + 1
          // form a new depth. N(v_i) denotes a neighborhood of
v_i.
          clique (N(v_i) | ∀v_j : j > i, j ≤ |V|, depth + 1)
     end while
     return
end function
```

Figure 3.1 VColor-u algorithm. Pseudo code

As noted by the author of the algorithm there is no any other optimizations than vertex coloring. It is done to evaluate influence of coloring on overall performance purely. Compared to two previous clique finding algorithms VColor-u demonstrates good results especially on high densities. Time consumption can be 50 times lower on 90% density graphs. Although on low densities (20%-50%) results are not so impressive, but anyway the new algorithm works about 15% faster.

An important note must be done that VColor-u algorithm works worse (~20% in time increase) than Carraghan and Pardalos algorithm on almost edgeless graphs (about 10% density). It can be explained by the fact that graph coloring and vertices ordering takes time and these steps are useless on a low density and pruning formula is not effective enough.

3.2 VColor-BT-u

There was a second algorithm introduced in the same article called VColor-BT-u [Kumlander 2005]. The idea was the same - to apply initial vertex coloring, but instead of the Carraghan and Pardalos approach in VColor-u, the new VColor-BT-u bases on the Östergård's algorithm.

As it was already noted, Östergård's algorithm starts with the only vertex and searches for a clique increasing graph size by one vertex. VColor-BT-u does the same, except it operates not with single vertices but with independent sets. Initially all the vertices are divided into several color classes $V = \{C_n, C_{n-1}, \dots, C_1\}$, where C_i contains vertices colored with color i. Note that color class indexes stand in reversed order because the algorithm starts from the rightmost vertex. First of all, algorithm tries to find the largest clique within C_1 on a first iteration (which, of course, equals 1, as there are no any adjacent vertices within independent set), then $C_1 \cup C_2$ (second iteration) and so forth until all the color classes are taken into account. In general at step i vertices of $C_1 \cup C_2 \cup \dots \cup C_i$ are considered.

VColor-BT-u uses two pruning formulas to skip even more unnecessary branches. The idea for the first bound rule comes from Östergård's algorithm. New algorithm holds clique sizes in special cache array b for each independent set added into consideration. Therefore, $b[i]$ contains a size of the largest clique inside $\{C_i, C_{i-1}, \dots, C_1\}$. Using this cache allow to use the following pruning formula $d - 1 + b[C(v_{di})] \leq CBC$, where d stands for depth level, v_{di}

is a vertex on depth d and index i, $C(v_{di})$ is a color of a vertex v_{di} and CBC is a current best (maximum) clique. Moreover, clique size for a current iteration can be equal or bigger on one compared to the previous iteration, because on each iteration we add a new color class and it is not possible that two vertices from an independent set will be added to a new clique, as these two vertices are not adjacent to each other by definition of independent set. Therefore, if on any step a larger clique is found we can continue with a new iteration.

In addition to the first pruning technique, it is possible to use the second one $d - 1 + Degree(G_d) \leq CBC$ taken from VColor-u algorithm in parallel. *Degree* function is copied from the previous algorithm as well.

VColor-BT-u is described using the following steps (Figure 3.2):

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 vi 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 $d - 1 + Degree(G_d) \leq |CBC|$ then go to the step 6.

Step 3.2. if $C(v_{d1}) > i$ then If $d - 1 + b[C(v_{d1})] \leq |CBC|$ 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.

Figure 3.2 VColor-BT-u algorithm. [Kumlander 2004]

A new algorithm results are much better compared to previously described ones. VColor-BT-u is approximately two times faster than VColor-u on almost all the densities. Compared to Östergård's algorithm the new algorithm is also faster 50%-100% on lower densities and 13-25 times on dense graphs, so a combination of two pruning techniques is really effective.

3.3 MCQ

MCQ algorithm was firstly introduced in 2003 by Tomita and Seki [Tomita, Seki 2003] and later Tomita and Kameda revised it with more computational experiments in 2007 [Tomita, Kameda 2007]. This algorithm bases on the Carraghan and Pardalos idea. Tomita and Seki noted that a number of vertices of a maximum clique $w(G)$ in a graph $G = (V,E)$ is always less or equal to the maximum degree $\Delta(G)$ plus 1 ($w(G) \leq \Delta(G) + 1$). Using this property, they reworked an existing pruning formula.

Tomita and Seki applied approximate coloring of vertices to prune unnecessary branches, giving a positive integer value called Number of Color (or color number) $No[p]$ for every vertex p. Number of Color has the special properties as described above:

1. Adjacent vertices cannot have the same color number i.e. if $(p, r) \in E$ then $No[p] \neq No[r]$

2. Number of Color is always set to lowest possible positive integer i.e. $No[p] = 1$, or if $No[p] = k > 1$, when there exist some vertices $p_1, p_2, \dots, p_{k-1}$ adjacent to p and $No[p_1] = 1, No[p_2] = 2, \dots, No[p_{k-1}] = k - 1$.

Consequently maximum color number inside a subset $R \subseteq V$ $Max\{No[p]|p \in R\}$ is always bigger or equal to the number of maximum clique in R i.e. $Max\{No[p]|p \in R\} \geq w(R)$. Therefore it is possible to prune that branch R if $|Q| + Max\{No[p]|p \in R\} \leq |Q_{max}|$ (Q stands for clique) holds. It should be noted here that each branch processing should start from a vertex having the biggest color number.

Color numbers can be easily assigned by greedy coloring algorithm applied to all vertices containing in a newly created branch R. It is important that vertices in R are ordered in a manner that vertices from the first color class C stand first, so that $R = C_1 \cup C_2 \cup \dots \cup C_{maxno}$. Authors of the algorithm say there might be more efficient way of coloring, but preliminary computation experiments show that more elaborate coloring requires more time. As soon as color numbers are assigned on each branch, more complicated coloring leads to overall negative impact on time consumption. Therefore, the key point of a "good" coloring algorithm is a balance between coloring quality and its performance.

For initial vertex numbering Tomita and Seki use special technique. Vertices with index i, where i from 1 to $\Delta(G)$ have color number equal to i. All others vertices are assigned to $\Delta(G) + 1)$ color number. Using such approach allows us to use $|Q| + Max\{No[p]|p \in R\} \leq |Q_{max}|$ pruning formula, as the largest clique size cannot exceed graph size or maximum degree plus 1.

Tomita and Seki demonstrate computational results to confirm their proposal on initial vertex ordering. Vertices should be sorted in the descending order in response to their degrees. This approach is approximately 50 times better than increasing ordering on dense graphs and gives about 15% time reduction on low densities.

```
// N(p) - set of neighbors of p
// No - set of color numbers

function MCQ
     Q := 0 // current clique
     Qmax:= 0// maximum clique
{SORT}
     Sort vertices of V in a descending order with respect to
their degrees;
```

```
{NUMBER}
     for i := 1 to Δ(G)
          No[V[i]] := i
     for i := Δ(G) + 1 to |V|
          No[V[i]] := (G) + 1
     EXPAND(V, No)
     return Qmax
end function

function EXPAND(R, No)
     while R ≠ ∅
          p := the vertex in R
               such that No[p] = Max{No[q] | q ∈ R};
               {i.e., the last (rightmost) vertex in R}
          if |Q| + No[p] > |Qmax| then
               Q := Q ∪ {p};
               Rp:= R ∩ N(p);
               if Rp≠ ∅ then
                    NUMBER-SORT(Rp, No'); // assign color
               numbers to Rp
                         {the initial value of No' has no
                    significance}
                    EXPAND(Rp, No')
          else if |Q| > |Qmax| then
               Qmax:= Q
          Q := Q - {p}
          else
               return
          R := R - {p}
end function
```

Figure 3.3 MCQ algorithm. Pseudo code

3.4 MCR

"An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments" article published by Tomita and Kameda in 2007 [Tomita, Kameda 2007] introduced a new MCR algorithm, a successor of MCQ algorithm. Compared to the older version, MCR mainly focused on initial sorting and color numbering. Branch processing i.e. EXPAND function was not changed, so we will spotlight only modified features and skip all the steps inherited from MCQ.

The main idea of improved initial sorting is that vertices in any subgraph R of $G = (V, E)$ should be ordered with response to their degrees in a decreasing order. If $V' = \{V[1], V[2], \ldots, V[i]\}$ is a vertex set of a subgraph R then $V[i]$ must always has the minimum degree for $1 \leq i \leq V$. To get this ordering we need to take a vertex with the smallest degree and set it to the last position of an array. This process is continued until all the rest unordered vertices have the same degree. Sometimes there might be several vertices with the same degree. For such cases, a new parameter vertex support S must be introduced. Support of a vertex v $S(v)$ is a sum of neighbor's degrees of v i.e. $S(v) = \sum_{r \in N(v)} \deg(r)$. If V[i-1] and V[i] have the same degree then $S(V[i-1]) \geq S(V[i])$.

At last, a subgraph R becomes induced by all the rest unordered vertices $V[1], V[2], \ldots, V[i]$, that have the same minimum degree. At this point R is regular graph. It is useless to continue vertex sorting on a regular graph. In this case, we start to assign color numbers (NUMBER-SORT function) to vertices in R as was described in MCQ. Vertices with index higher than i : $V[i+1], V[i+2], \ldots, V[n]$ must be numbered as $\text{Min}\{\text{maxno} + 1, \Delta G + 1\}, \text{Min}\{\text{maxno} + 2, \Delta G + 1\}$, ..., $\text{Min}\{\text{maxno} + (n - i), \Delta G + 1\}$ respectively, where maxno is a maximum color number acquired by NUMBER-SORT of R. Furthermore, if all vertices $V[1], V[2], \ldots, V[i]$ in R have the same degree *(i-1)*, then these vertices form a clique of size i and initial clique size can be set to i.

Initial sorting and color numbering is quite complicated and time consuming operations, but it has no significant influence on overall algorithm performance as it is done only one time at the begging of MCR algorithm.

```
// N(p) - set of neighbors of p
// No - set of color numbers
// deg(p) - degree of p
// s(p) - support of p

function MCR(G = (V, E))
     Q := 0 // current clique
     Q_max := 0// maximum clique
{SORT}
     i := |V|;
     R := V; V := ∅;
     R_min := set of vertices with the minimum degree in R;
     while |R_min| ≠ |R|
          if |R_min| ≥ 2 then
```

```
                p := a vertex in Rmin such that S(p) = Min{S(q) |
        q ∈ Rmin}
        else
                p := Rmin [1];
        V[i] := p; R := R − {p};
        i := i − 1;
        for j := 1 to |R|
                if R[j] is adjacent to p then
                deg(R[j]) := deg(R[j]) − 1
        Rmin:= set of vertices with the minimum degree in R
{Regular subgraph}
    NUMBER-SORT(Rmin, No);
    for i := 1 to |Rmin|
        V[i] := Rmin[i]
{NUMBER}
    m := Max{No[q] | q ∈ Rmin};
    mmax := |Rmin| + (G) − m;
    m := m + 1;
    i := |Rmin| + 1;
    while i ≤ mmax
        if i > |V| then
            goto Start
        No[V[i]] := m;
        m := m + 1;
        i := i + 1
    for i := mmax + 1 to |V| do
        No[V[i]] := Δ(G) + 1
Start:
    if degRmin(q) = |Rmin| − 1 for all q ∈ Rmin then
        Qmax:= Rmin
    EXPAND(V, No)
    return Qmax
end function
```

Figure 3.4 MCR algorithm. Pseudo code

3.5 MCS

Three years later after MCR was released a new improvement for the same algorithm appeared called MCS [Tomita, Sutani, Higashi, Takahashi, Wakatsuki 2010]. This time authors focused on approximate coloring enhancements. There is a crucial property derived from MCR bounding condition $|Q| + Max\{No[p]|p \in R\} \leq |Q_{max}|$. Greedy approximate

coloring assigns color numbers to vertices and order them in a manner that vertices with the biggest number stand last. On each depth, the rightmost vertex is expanded first i.e. the vertex with biggest color number $No[r] = Max\{No[p] | p \in R\}$). Therefore, if $No[r] \leq |Q_{max}| - |Q|$ we prune a branch. $|Q_{max}| - |Q|$ forms a kind of a threshold after which we skip all the vertices. Taking a new property into consideration it is much more important to reduce a number of vertices (it might look like reduction of number of color classes is the main goal, but it is not) from which searching is necessary in other words approximate coloring should produce more vertices with color numbers less than a threshold to skip them later.

A new approximate coloring algorithm was introduced to meet new requirements. It can be described with the following steps:

1. Calculate threshold No_{th}. Threshold is equal to maximum clique value minus current clique value $No_{th} = |Q_{max}| - |Q|$.
2. Try to find a vertex q within neighbors of p ($N[p]$) with a color number less than a threshold $(No[q] = k_1) \leq No_{th}$, such that $|C_{k1}| = 1$.
3. If q is found, the next step is an attempt to find color number k_2 such than there is no neighbor of q ($N[q]$) colored in k_2.
4. If k_2 is found, then q and p should change their color numbers so that $No[p] = k_1$ and $No[q] = k_2$. (It is crucial to understand that this operation changes initial vertex order, as after each coloring vertices are ordered with response to their color numbers.)
5. If no vertex q or color number k_2 is found, nothing happens.

A new approximate coloring algorithm triggers each time a new color number bigger or equal to a threshold is created. Then it tries to insert current vertex to any of the previous color classes less that the threshold. If operation succeeds, a new color class becomes empty and should be removed. The inserted vertex will not be expanded later because of the bounding condition that will prune a branch containing that vertex.

```
function Re-NUMBER(p, No[p], Noth, C1, C2, ..., Cmaxno)
     for k1 := 1 to Noth- 1
          if |Ck1∩ N(p)| = 1 then
               q := the element in (Ck1∩ N(p));
     for k2 := k1 + 1 to Noth
          if |Ck2∩ N(p)| = ∅ then
               {Exchange the Numbers of p and q.}
               CNo[p] := CNo[p] - {p};
               Ck1 := (Ck1 - {q}) ∪ {p};
```

```
C_k2 := C_k2 ∪ {q};
return
```

Figure 3.5 MCS algorithm. Renumbering function pseudo code.

As was already noted before a new approximate greedy coloring operation changes the initial vertex order. To inherit the same ordering through the whole algorithm a new array V_a (the same data structure as array of vertices V) must be created. Vertices are copied to V_a and then passed to coloring function. This means that vertices will be reordered inside V_a and initial order is still present in array V.

In comparison with older MCR version, a new MCS algorithm shows good results especially on dense graphs. It is clearly seen on DIMACS graphs such as r200.98 or r300.98 where MCS performs more than 100 times better than MCR.

3.6 MCS improved

"Improvements to MCS algorithm for the maximum clique problem" article was released in 2014 by Mikhail Batsyn, Boris Goldengorin, Evgeny Maslov and Panos M. Pardalos [Batsyn, Goldengorin, Maslov, Pardalos 2014]. Authors proposed the following improvements to fasten search of maximum clique:

- At the beginning of the algorithm ILS heuristic [Andrade, Resende, Werneck 2012] is applied to gain initially „good" (i.e. close to the maximum possible) solution. This value is then used to prune branches. This improvement gives the noticeable reduction of branches number especially on dense graph therefore decreasing time consumption.
- On each depth if a set of candidates contains some vertex which is connected to all other vertices in this set, this vertex is immediately added to current clique, consequently the vertex is not expanded later and upper bound is increasing faster. This improvement means that on each depth it is possible to increase current clique size more than on one. As a result, the faster upper bound grows the more branches we can prune.

- Authors of the article state that storing sets of candidate vertices and color numbers on stack is more efficient than in dynamic memory. This property was gained from experimental results.
- MCS Improved initially use simple vertex ordering with response to their degrees as proposed in Carraghan and Pardalos article [Carraghan, Pardalos 1990]. There is no any additional reordering applied as it was done in the previous MCR and MCS algorithms. Moreover, at the beginning of algorithm all the vertices are colored by greedy algorithm without swaps.

```
function MCSWithHeuristic( )
     Q* = HeuristicSolution( )
     InitialOrderingAndColouring(L^0)
     for i = |L^0|, 1 do
          v = L_i^0
          i if UpperBound(v) > |Q*| then
               ProcessBranch(v, L^0)
          end if
     end for
end function
```

Figure 3.6 MCS with incorporated ILS heuristic and other improvements. Pseudo code. [Batsyn, Goldengorin, Maslov, Pardalos 2014]

MCSI show very good results on dense graphs using high-quality solution gained by ILS heuristic algorithm. Authors compare their new algorithm to MCS on special DIMACS graphs. The most significant result is on gen400_p0.9_65 instance where number of branches was reduced more than 7000 times. Moreover, improved MCS algorithm solves p_hat1000-3 instance that was not possible to solve by MCS algorithm with a reasonable time. Authors also propose that there might be better heuristic algorithm for searching initial solution than ILS algorithm.

4. New algorithm

In this chapter, we are going to introduce a new algorithm solving maximum clique problem. It is called VRecolor-BT-u as this algorithm is a successor of VColor-BT-u algorithm and it implements recoloring on each depth. There were multiple algorithms described previously in this work. The idea of a new one is to gather and combine all the gained knowledge to fasten maximum clique finding even more.

It can be clearly seen from the modern algorithms that almost all of them are focused on Carraghan and Pardalos approach and only VColor-BT-u implements Östergård's idea. Moreover, initially Östergård's approach with reversed search showed much better results than Carraghan and Pardalos algorithm. Even after when D. Kumlander applied coloring to both these basic algorithms, performance of Östergård's algorithm successor VColor-BT-u was much faster than VColor-u, an improvement to Carraghan and Pardalos algorithm.

From the other hand, algorithms from Tomita and his colleagues proved that in-depth coloring is a very efficient technique and initial coloring is not enough as the "deeper" level we are constructing the more diffused initial coloring becomes. When depth is high, we definitely need to recolor vertices to update colors and gain the most accurate data about independent sets on this level.

4.1 Description

The main idea of a new algorithm is to combine reversed search by color classes (from VColor-BT-u) and in-depth coloring i.e. recoloring (from MCQ and successors). Before we can start there should be some useful properties from previous algorithms noted:

1. Reversed search by color classes means searching for a clique in a constantly increasing subgraph adding each color class one by one holding a cache *b[]* for each color class, where cache is a maximum clique found by given color class. First of all, we consider a subgraph S_1 consisting only from vertices of a first color class C_1. After

than subgraph S_2 is created with two color classes C_1 and C_2. In general $S_i = C_1 \cup C_2 \cup \ldots \cup C_i$.

2. Pruning formula for reversed search by color classes is $d - 1 + b[C(v_{di})] \leq |CBC|$ can be used only if vertices in each subgraph *Si* are ordered by initial color classes (using this color classes we are constructing a new subgraph on each iteration).
3. If vertices are ordered by their color numbers and are expanded starting from the largest color number then all the vertices with color number lower than a threshold ($th = |CBC| - (d - 1)$ can be ignored as they will not be expanded because of a pruning formula $d - 1 + Max\{No[p]|p \in R\} \leq |CBC|$.
4. Pruning formula $d - 1 + Max\{No[p]|p \in R\} \leq |CBC|$ can be used when we are reapplying coloring on each depth and vertices are reordered with response to these colors.

From this point, it is seen that properties 2 and 4 are conflicting with each other, as two pruning formulas require different vertex ordering. As a result, if both bounding rules are used we are going to miss some cliques when a promising branch will be pruned. To avoid such situations the formula $d - 1 + Max\{No[p]|p \in R\} \leq |CBC|$ was used not to prune a branch but to skip a current vertex as expanding it is not going to give us a better solution. This means that if vertices are recolored on each depth, but are not ordered with response to new colors, we can skip a vertex without expanding it, if and only if its color number is lower than a current threshold and there is no neighbors of this vertex with color number larger than threshold and who stand after the bound gained from the first pruning formula $d - 1 + b[C(v_{di})] \leq |CBC|$.

There is an example on figure 4.1 that shows how a conflict with two different colorings is solved. Green lines show adjacency of two vertices (not all the adjacent vertices are marked with green lines, but only two that are interesting for us in this specific example). Let us assume that current depth is two and we have the following prerequisites:

- $d = 2$ (depth is 2)
- $|CBC| = 3$ (current best clique is 3)
- $th = 3 - (2 - 1) = 2$ (threshold taken from skipping formula, we need to expand vertices having color number bigger than threshold)
- $b[1] = 1, b[2] = 2, b[3] = 3, b[4] = 3$ (cache values found from previous iterations)

- $bnd = 2$ (index of a rightmost vertex expanding which a pruning formula $d - 1 + b[C(v_{di})] \leq |CBC|$ will prune current branch)
- Ca – array storing initial color classes, Cb – array storing in-depth color classes

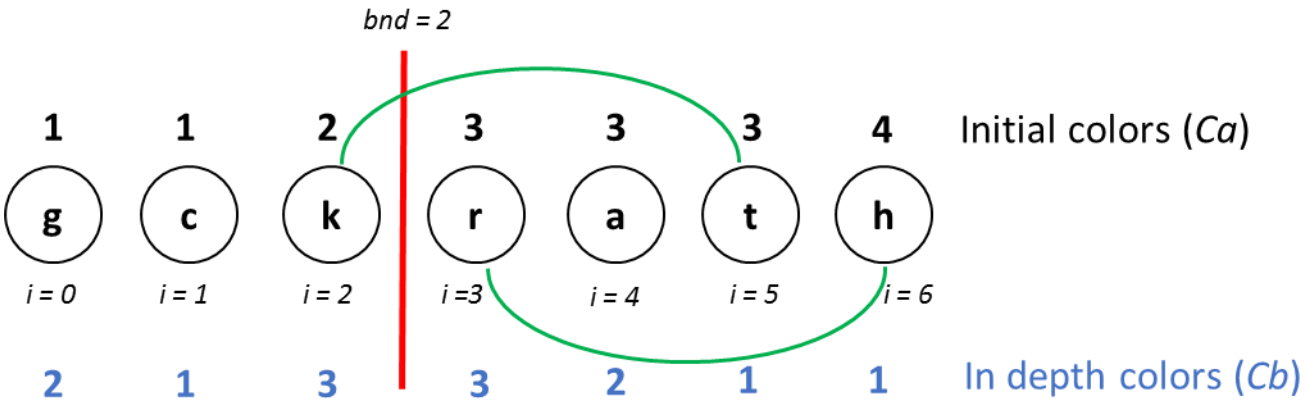

Figure 4.1 Different coloring conflict detailed example.

Let us analyze the current example (figure 4.1). We start with the rightmost vertex *h* with in-depth color number 1 (*No[h] = 1*). We skip this vertex as long as its color number is lower than a threshold (*th = 2*). As you can see vertex *h* might be contained in a larger clique as it is connected with a vertex *r* (*No[r] = 3*), but we skip it anyway because vertex *r* will be expanded later. Now we proceed with the next vertex *t*. Color number of *t* is 1 (*No[t] = 1*), the same as vertex *h* has, but in this case it is not possible to skip vertex *t*, because it is adjacent to vertex *k* (*No[k] = 3*). Vertex *k* stands after the pruning bound (*bnd = 2*), therefore it will not be expanded at all. If we skip vertex *t* right now we might possibly skip a larger clique, this means that vertex *t* should be expanded. The next vertex to analyze is vertex *a*, we skip it as its in-depth color number is equal to the threshold (*th = No[a] = 2*) and there are no adjacent vertex standing after bound. In addition, the last expanded vertex on current depth is *r* (*No[a] = 3*) as its color number is larger than the threshold. It should be noted that skipped vertices are not thrown away from further considerations (when building the next depth), they should be stored in a separate array and added to the next depth with preserved order.

There is another pruning formula used right after recoloring is done. As we already know, number of color classes obtained by coloring subgraph G_d is an upper bound for

maximum clique in a current subgraph. This property allows us to use the following pruning formula $d - 1 + cn \leq |CBC|$, where *cn* is a number of colors gained from recoloring.

4.2 Coloring choice based on density

There are two coloring algorithms used in VRecolor-BT-u. They are both greedy, but the first one is using swaps when coloring and the other one is not. Each time coloring is applied, we need to determine which algorithm to use. Moreover, there are two places where we need to use coloring: initial coloring performed one time at the beginning of the algorithm and in-depth coloring applied each time a new depth is constructed. Coloring algorithm choice is made according to graph density using special constants; they are 0.35 density for initial coloring and 0.55 density for in-depth coloring. Coloring choice can be described with the following diagram (figure 4.2).

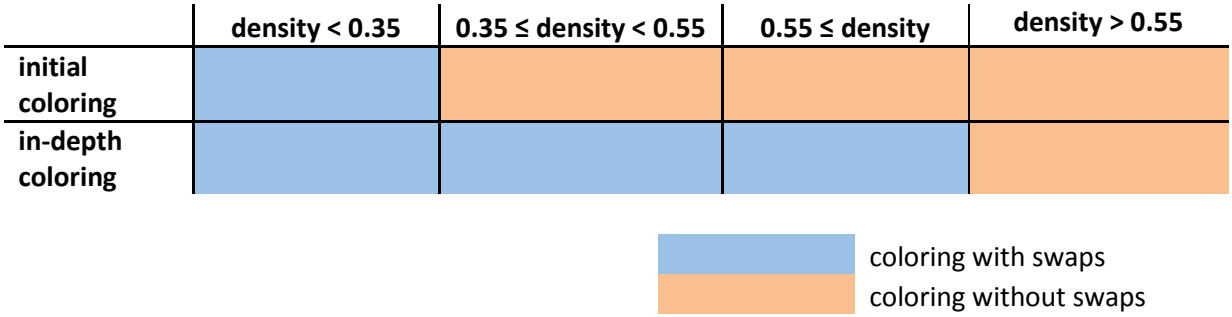

Figure 4.2 Coloring choice based on density

Constants 0.35 and 0.55 were found using experimental results and are a subject of future studies. Figures 4.3 and 4.4 demonstrate performance of VRecolor-BT-u with swaps and VRecolor-BT-u with swaps. As seen from these graphs algorithm with swapping works better on low densities (density < 0.35) and must be replaced by coloring without swaps already on density 0.4. The same approach works with in-depth coloring but coefficient is bigger in that case, it is 0.55. Figures 4.5 and 4.6 demonstrate a coefficient choice for recoloring algorithm. It should be noted that on figures from 4.3 to 4.6 y-axis demonstrates time (in milliseconds) consumed by tested algorithms for finding maximum clique.

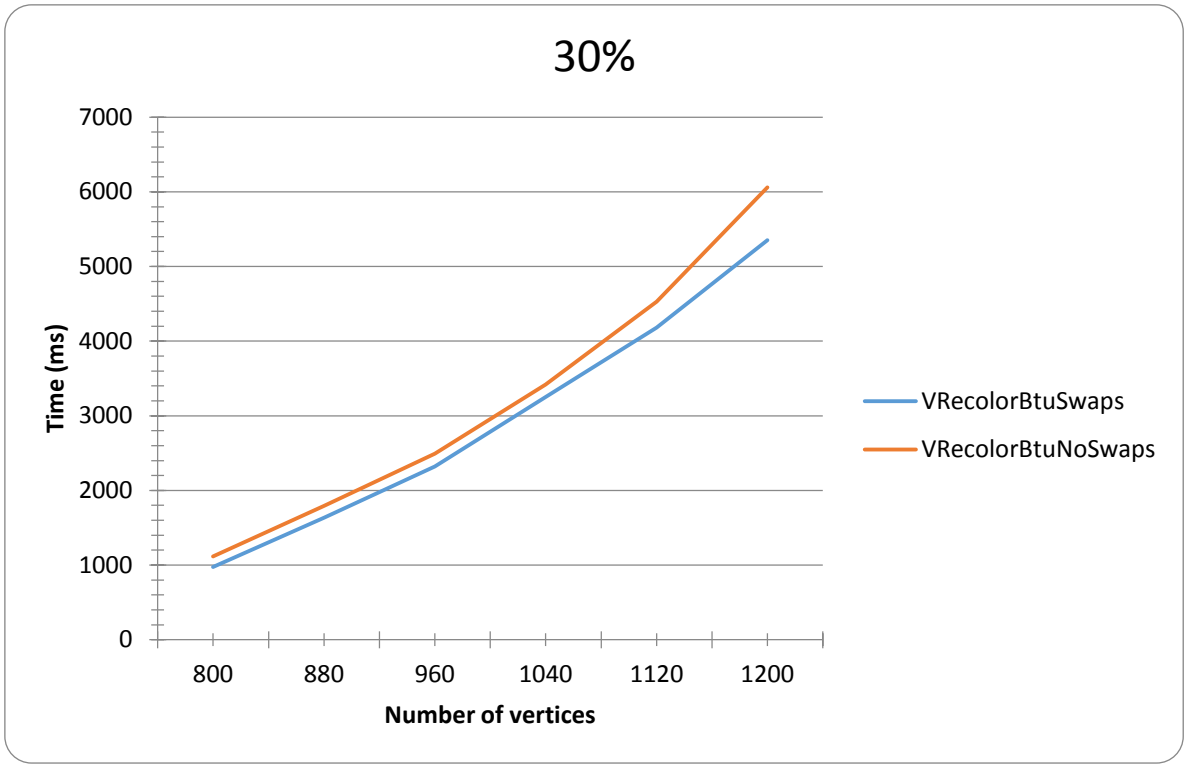

Figure 4.3 VRecolor-BT-u with and without swaps initial coloring comparison. Density 0.3.

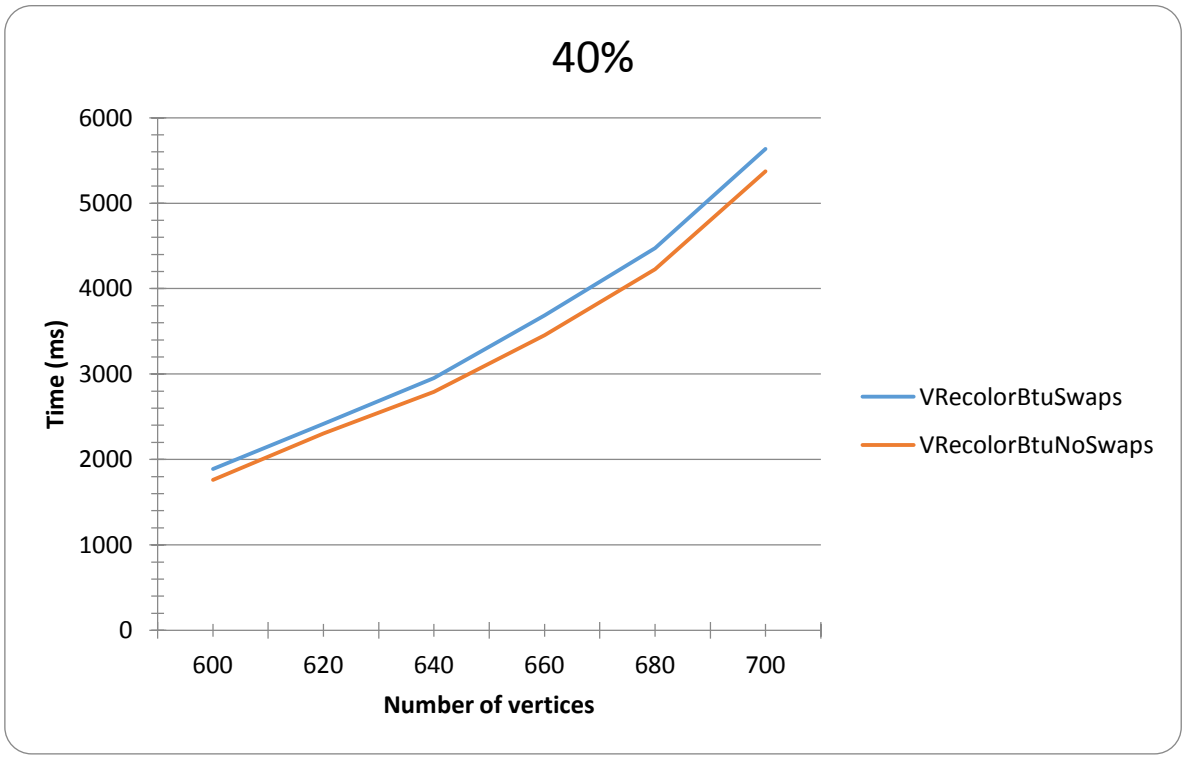

Figure 4.4 VRecolor-BT-u with and without swaps initial coloring comparison. Density 0.4.

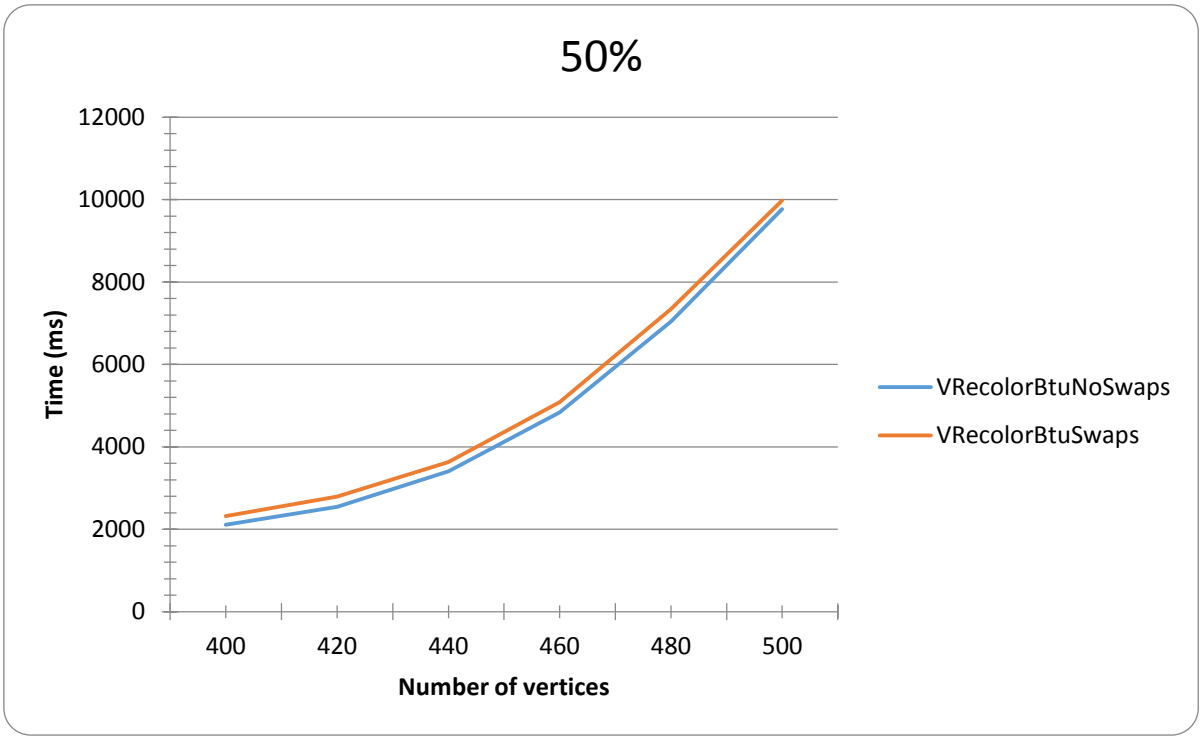

Figure 4.5 VRecolor-BT-u with and without swaps in-depth coloring comparison. Density 0.5.

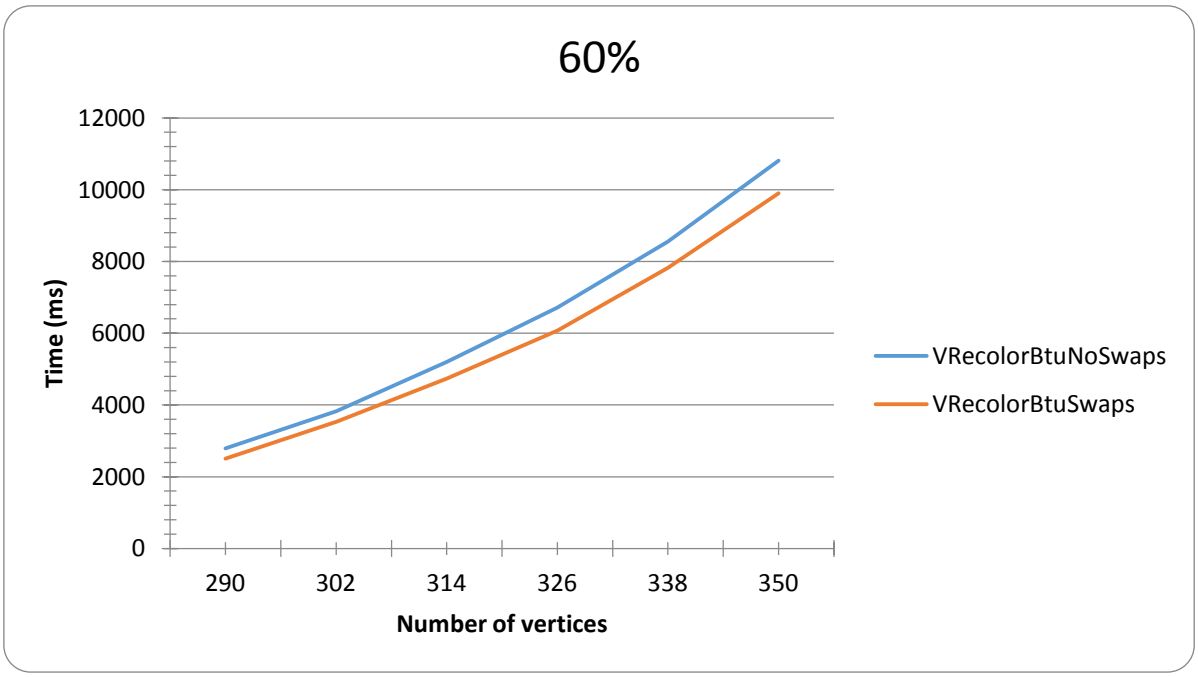

Figure 4.6 VRecolor-BT-u with and without swaps in-depth coloring comparison. Density 0.6.

4.3 Algorithm

This part demonstrates the VRecolor-BT-u algorithm. There are two example graphs are solved using the following algorithm.

4.3.1 VRecolor-BT-u

CBC – current best clique, largest clique found by so far.
d – depth.
c – index of the currently processed color class.
di – index of the currently processed vertex on depth d.
b – array to save maximum clique values for each color class.
Ca – initial color classes array.
Cb – color classes array recalculated on each depth.
G_d - subgraph of graph G induced by vertices on depth d.
cn – number of color classes recalculated on each depth.
$CanBeSkipped(v_{di}, c)$ - function that returns true if a vertex can skipped without expanding it.

1. **Graph density calculation.** If graph density is lower than 35% go to step 2a, else go to step 2b.
2. **Heuristic vertex greedy coloring.** There should be two arrays created to store initial color classes defined only once (Ca) and color classes recalculated on each depth (Cb). During this step, both arrays must be equal.
    a. Before coloring vertices are unordered and colored with swaps.
    b. Before coloring vertices are in decreasing order with response to their degree and colored without swaps.
3. **Searching.** For each color class starting from the first (current color class index c).
    3.1. **Subgraph (branch) building.** Build the first depth selecting all the vertices from color classes whose number c is equal or smaller than current. Vertices

from the first color class should stand first. Vertices at the end should belong to c color class.

3.2. **Process subgraph.**

3.2.1. **Initialize depth.** $d = 1$.

3.2.2. **Initialize current vertex.** Set current vertex index di to be expanded (initially the first expanded vertex is the rightmost one). $di = n_d$.

3.2.3. **Bounding rule check.** If current branch can possibly contain larger clique than found by so far. If $Ca(v_{di}) < c$ and $d - 1 + b[Ca(v_{di})] \leq |CBC|$ then prune. Go to step 3.2.7.

3.2.4. **Vertex skipping check.** If current vertex can possibly contain larger clique than found by so far. If $d - 1 + Cb(v_{di}) \leq |CBC|$ and $CanBeSkipped(v_{di}, c)$ skip this vertex. Decrease index $i = i - 1$. Go to step 3.2.3.

3.2.5. **Expand current vertex.** Form new depth by selecting all the adjacent vertices (neighbors) to current vertex v_{di} ($G_{d+1} = N(v_{di})$). Set the next expanding vertex on current depth $di = di - 1$.

3.2.6. **New depth analysis.** Check if new depth contains vertices.

a. If $G_{d+1} = \emptyset$ then check if current clique is the largest one it must be saved. Go to step 3.3.

b. If $G_{d+1} \neq \emptyset$ then check graph density. If graph density is lower than 55% apply greedy coloring with swaps to G_{d+1}, else use greedy coloring without swaps. Save number of color classes (cn) acquired by this coloring. If number of color classes cannot possibly give us a larger clique then prune. If $d - 1 + cn \leq |CBC|$ decrease index $i = i - 1$ and go to step 3.2.3, else increase depth $d = d + 1$. Go to step 3.2.2.

3.2.7. **Step back.** Decrease depth $d = d - 1$. Delete expanding vertex from the current depth. If d = 0 go to step 3.3, else go to step 3.2.3.

3.3. **Complete iteration.** Save current best clique value for this color. $b[c] = |CBC|$.

4. **Return maximum clique.** Return CBC.

4.3.2 CanBeSkipped function

th – threshold from which branch will be pruned

CBC – current best clique, largest clique found by so far.

d – depth.

c – index of the currently processed color class.

di – index of the currently processed vertex on depth d.

bnd – bound from which vertices cannot be skipped.

b – array to save maximum clique values for each color class.

Ca – initial color classes array.

Cb – color classes array recalculated on each depth.

1. **Define threshold.** $th = |CBC| - (d - 1)$.
2. **Find skipping bound.** For each vertex index dj from $di - 1$ to 0. If $Ca(v_{dj}) < c$ and b$[Ca(v_{dj})] \leq th$ then $bnd = j$.
3. **Decide whether vertex can be skipped.** For each adjacent (to currently expanded) vertex with index dj from bnd to zero. If $Cb(v_{dj}) > th$ then return false. If $Cb(v_{dj}) > th$ had never occurred return true.

4.4 Example 1

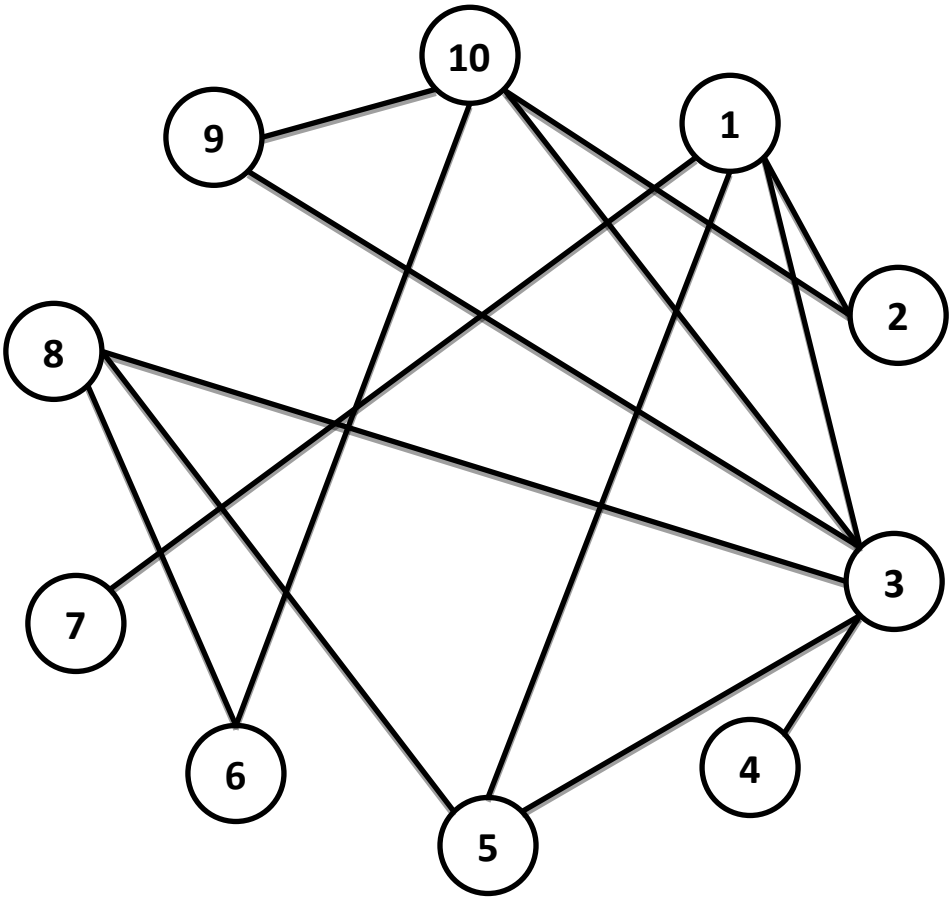

Figure 4.7 VRecolor-BT-u example 1. Processed graph.

First of all, we need to determine graph density. Number of vertices is 10 ($|V| = 10$), edge number is 14 ($|E| = 14$). Density is 0.31 ($D = \frac{2|E|}{|V|(|V|-1)} \Rightarrow D = \frac{2*14}{10*9} = 0.31$).

The next step is to apply greedy coloring and define color classes. As long as density is lower than 0.35 and 0.55, we use coloring with swaps without initial ordering in both cases in the algorithm. In result, we have the following initial color classes (Ca) (Please note that initially Ca values are copied to Cb. Later on Cb will be changed while Ca stays unmodified):

Class 1: {1, 4, 6, 9}
Class 2: {5, 7, 2}
Class 3: {8, 10}
Class 4: {3}

Now we will start depth construction and searching for a clique. Grayed out vertices are currently expanded.

Table 4.1 VRecolor-BT-u algorithm example 1

Depth	Subgraph G Color classes Cb	Current color class (c)	Description
d = 1	G = {1, 4, 6, 9} $Cb[1]$ = {1, 4, 6, 9}	1	Construct the first subgraph using first color class vertices only. $\lvert CBC \rvert$ = 0. Bounding rule check: $Ca(9) = 1 \Rightarrow 1 < 1$ - false. We continue search because 1 ≥ current color. Vertex skipping check: $1 - 1 + 1 \leq 0 \Rightarrow 1 \leq 0$ - false. We continue search because 1 > $\lvert CBC \rvert$. Search current vertex neighbors and construct new depth. $N(9) = \emptyset$.
d = 2	$G = \emptyset$	1	Save current clique if it is larger than |CBC|. Current clique = {9}. 1 > 0 – true. $\lvert CBC \rvert = \{9\}$. Complete iteration. $b[1]$ = 1.
d = 1	G = {1, 4, 6, 9, 5, 7, 2} $Cb[1]$ = {1, 4, 6, 9} $Cb[2]$ = {5, 7, 2}	2	Construct subgraph using first and second color class vertices. $\lvert CBC \rvert$ = 1. Bounding rule check: $Ca(2) = 2 \Rightarrow 2 < 2$ - false. We continue search because 2 ≥ current color. Vertex skipping check: $1 - 1 + 2 \leq 1 \Rightarrow 2 \leq 1$ - false. We continue search because 2 > $\lvert CBC \rvert$. $N(2) = \{1\}$.

d = 2	G = {1} *Cb[1]* = {1}	2	cn = 1. Check if number of color classes can possibly give a larger clique: $2 - 1 + 1 \leq 1 \Rightarrow 2 \leq 1$. We continue search because 2 > */CBC/.* Bounding rule check: $Ca(1) = 1 \Rightarrow 1 < 2$ - true. Check the second condition $b[1] = 1 \Rightarrow 2 - 1 + 1 \leq 1$ – false. We continue search because 2 ≤ */CBC/.* Vertex skipping check: $2 - 1 + 1 \leq 1 \Rightarrow 2 \leq 1$ - false. We continue search because 2 > */CBC/.* Search current vertex neighbors and construct new depth. $N(1) = \emptyset$.
d = 3	$G = \emptyset$	2	Save current clique if it is larger than */CBC/.* Current clique = {2, 1}. 2 > 1 – true. */CBC/* = {2, 1}. Complete iteration. *b[2]* = 2.
d = 1	G = {1, 4, 6, 9, 5, 7, 2, 8, 10} *Cb[1]* = {1, 4, 6, 9} *Cb[2]* = {5, 7, 2} *Cb[3]* = {8, 10}	3	Construct subgraph using vertices of color classes 1, 2, 3. */CBC/* = 2. Bounding rule check: $Ca(10) = 3 \Rightarrow 3 < 3$ - false. We continue search because 3 ≥ current color. Vertex skipping check: $1 - 1 + 3 \leq 2 \Rightarrow 3 \leq 2$ - false. We continue search because 3 > */CBC/.* Search current vertex neighbors and construct new depth. $N(10) = \{6, 9, 2\}$.
d = 2	G = {6, 9, 2} *Cb[1]* = {6, 9, 2}	3	cn = 1. Check if number of color classes can possibly give a larger clique: $2 - 1 + 1 \leq 2 \Rightarrow 2 \leq 2$. We prune this branch because 2 ≤ */CBC/.*

			Go to the previous depth.
d = 1	G = {1, 4, 6, 9, 5, 7, 2, 8, 10} *Cb[1]* = {1, 4, 6, 9} *Cb[2]* = {5, 7, 2} *Cb[3]* = {8, 10}	3	Take the next vertex to expand. Bounding rule check: $Ca(8) = 3 \Rightarrow 3 < 3$ - false. We continue search because 3 ≥ current color. Vertex skipping check: $1 - 1 + 3 \leq 2 \Rightarrow 3 \leq 2$ - false. We continue search because 3 > */CBC/.* $N(8) = \{6, 5\}$.
d = 2	G = {6, 5} *Cb[1]* = {6, 5}	3	cn = 1. Check if number of color classes can possibly give a larger clique: $2 - 1 + 1 \leq 2 \Rightarrow 2 \leq 2$. We prune this branch because 2 ≤ */CBC/.* Go to the previous depth.
d = 1	G = {1, 4, 6, 9, 5, 7, 2, 8, 10} *Cb[1]* = {1, 4, 6, 9} *Cb[2]* = {5, 7, 2} *Cb[3]* = {8, 10}	3	Take the next vertex to expand. Bounding rule check: $Ca(2) = 2 \Rightarrow 2 < 3$ - true. Check the second condition $b[2] = 2 \Rightarrow 1 - 1 + 2 \leq 2$ – true. We prune this branch because 2 ≤ */CBC/.* Complete iteration. *b[3]* = 2.
d = 1	G = {1, 4, 6, 9, 5, 7, 2, 8, 10, 3} *Cb[1]* = {1, 4, 6, 9} *Cb[2]* = {5, 7,	4	Construct subgraph using vertices of color classes 1, 2, 3, 4. */CBC/* = 2. Bounding rule check: $Ca(3) = 4 \Rightarrow 4 < 4$ - false. We continue search because 4 ≥ current color. Vertex skipping check: $1 - 1 + 4 \leq 2 \Rightarrow 4 \leq 2$ -

	2} *Cb[3]* = {8, 10} *Cb[4]* = {3}		false. We continue search because 4 > */CBC/.* Search current vertex neighbors and construct new depth. $N(3) = \{1, 4, 9, 5, 8, 10\}$.
d = 2	*G* = {1, 4, 9, 5, 8, 10} *Cb[1]* = {1, 4, 9, 8} *Cb[2]* = {5, 10}	4	*cn* = 2. Check if number of color classes can possibly give a larger clique: $2 - 1 + 2 \leq 2 \Rightarrow 3 \leq 2$. We continue search because 3 ≥ */CBC/.* Bounding rule check: $Ca(10) = 3 \Rightarrow 3 < 4$ - true. Check the second condition $b[3] = 2 \Rightarrow 2 - 1 + 2 \leq 2$ – false. We continue search because 3 > */CBC/.* Vertex skipping check: $2 - 1 + 2 \leq 2 \Rightarrow 3 \leq 2$ - false. We continue search because 3 > */CBC/.* Search current vertex neighbors and construct new depth. $N(10) = \{9\}$.
d = 3	*G* = {9} *Cb[1]* = {9}	4	*cn* = 1. Check if number of color classes can possibly give a larger clique: $3 - 1 + 1 \leq 2 \Rightarrow 3 \leq 2$. We continue search because 3 ≥ */CBC/.* Search current vertex neighbors and construct new depth. $N(9) = \emptyset$.
d = 4	$G = \emptyset$	4	Save current clique if it is larger than */CBC/.* Current clique = {3, 10, 9}. 3 > 2 – true. */CBC/* = {3, 10, 9}. Complete iteration. *b[4]* = 3.

Since there are no more color classes, we stop. The maximum clique is {3, 10, 9} and its size is three.

4.5 Example 2

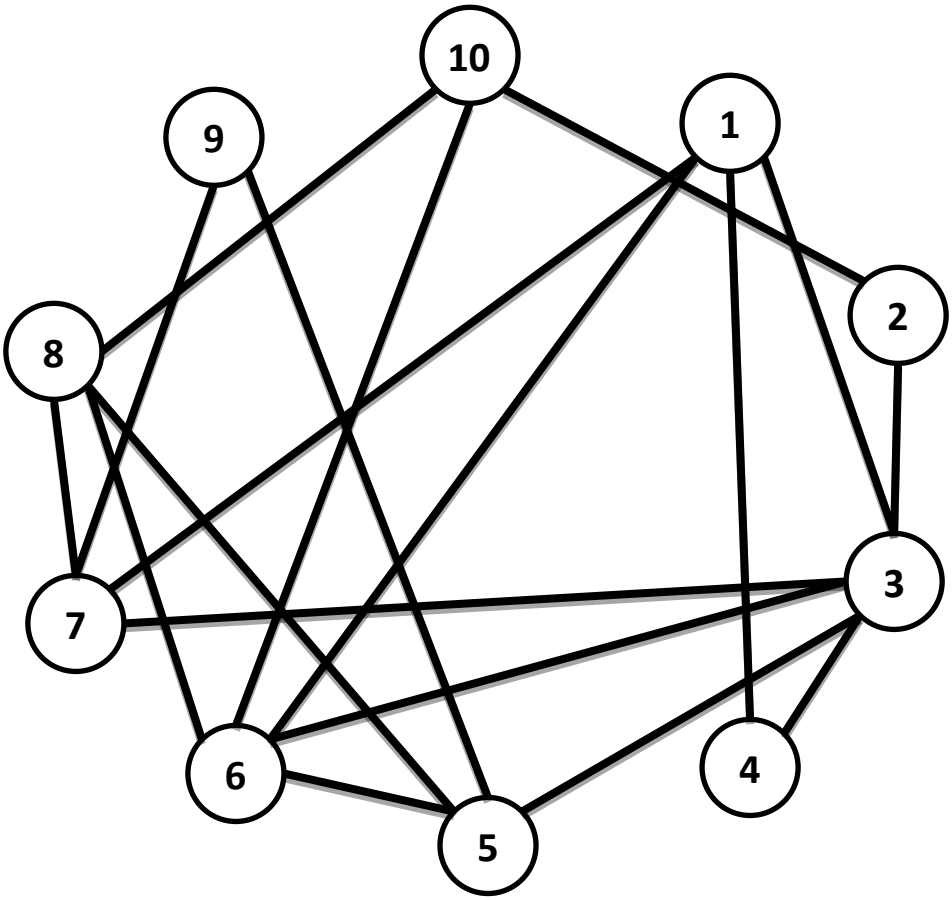

Figure 4.8 VRecolor-BT-u example 2. Processed graph.

Number of vertices is 10 ($|V| = 10$), edge number is 18 ($|E| = 18$). Density is 0.4 ($D = \frac{2|E|}{|V|(|V|-1)} \Rightarrow D = \frac{2*18}{10*9} = 0.4$).

Density is higher than 0.35 we set vertices in decreasing order with response to their degrees and obtain initial color classes (Ca) using greedy coloring without swaps. Since density is lower than 0.55, we use coloring with swaps without ordering by degree when recoloring on each depth. In result, we have the following initial color classes (Ca):

Class 1: {3, 8, 9}
Class 2: {6, 7, 2, 4}
Class 3: {1, 5, 10}

Table 4.2 VRecolor-BT-u algorithm example 2

Depth	Subgraph G Color classes Cb	Current color class (c)	Description
d = 1	G = {3, 8, 9} $Cb[1]$ = {3, 8, 9}	1	Construct the first subgraph using first color class vertices only. $\|CBC\|$ = 0. Bounding rule check: $Ca(9) = 1 \Rightarrow 1 < 1$ - false. We continue search because 1 ≥ current color. Vertex skipping check: $1 - 1 + 1 \leq 0 \Rightarrow 1 \leq 0$ - false. We continue search because 1 > $\|CBC\|$. Search current vertex neighbors and construct new depth. $N(9) = \emptyset$.
d = 2	$G = \emptyset$	1	Save current clique if it is larger than $\|CBC\|$. Current clique = {9}. 1 > 0 – true. $\|CBC\| = \{9\}$. Complete iteration. $b[1]$ = 1.
d = 1	G = {3, 8, 9, 6, 7, 2, 4} $Cb[1]$ = {3, 8, 9} $Cb[2]$ = {6, 7, 2, 4}	2	Construct subgraph using first and second color class vertices. $\|CBC\|$ = 1. Bounding rule check: $Ca(4) = 2 \Rightarrow 2 < 2$ - false. We continue search because 2 ≥ current color. Vertex skipping check: $1 - 1 + 2 \leq 1 \Rightarrow 2 \leq 1$ - false. We continue search because 2 > $\|CBC\|$. Search current vertex neighbors and construct new depth. $N(4) = \{3\}$.
d = 2	G = {3}	2	cn = 1. Check if number of color classes can possibly give a larger clique: $2 - 1 + 1 \leq 1 \Rightarrow 2 \leq 1$. We

	Cb[1] = {3}		continue search because 2 > */CBC/.* Bounding rule check: $Ca(3) = 1 \Rightarrow 1 < 2$ - true. Check the second condition $b[1] = 1 \Rightarrow 2 - 1 + 1 \leq 1$ – false. We continue search because 2 ≤ */CBC/.* Vertex skipping check: $2 - 1 + 1 \leq 1 \Rightarrow 2 \leq 1$ - false. We continue search because 2 > */CBC/.* Search current vertex neighbors and construct new depth. $N(3) = \emptyset$.
d = 3	*G* = ∅	2	Save current clique if it is larger than */CBC/.* Current clique = {4, 3}. 2 > 1 – true. */CBC/* = {2, 1}. Complete iteration. *b[2]* = 2.
d = 1	*G* = {3, 8, 9, 6, 7, 2, 4, 1, 5, 10} *Cb[1]* = {3, 8, 9} *Cb[2]* = {6, 7, 2, 4} *Cb[3]* = {1, 5, 10}	3	Construct subgraph using vertices of color classes 1, 2, 3. */CBC/* = 2. Bounding rule check: $Ca(10) = 3 \Rightarrow 3 < 3$ - false. We continue search because 3 ≥ current color. Vertex skipping check: $1 - 1 + 3 \leq 2 \Rightarrow 3 \leq 2$ - false. We continue search because 3 > */CBC/.* Search current vertex neighbors and construct new depth. $N(10) = \{8, 6, 2\}$.
d = 2	*G* = {8, 6, 2} *Cb[1]* = {8, 2} *Cb[2]* = {6}	3	*cn* = 2. Check if number of color classes can possibly give a larger clique: $2 - 1 + 2 \leq 2 \Rightarrow 3 \leq 2$. We continue search because 3 ≥ */CBC/.* Bounding rule check: $Ca(2) = 2 \Rightarrow 2 < 3$ - true. Check the second condition $b[2] = 2 \Rightarrow 2 - 1 + 2 \leq 2$

			– false. We continue search because 3 > */CBC/.* Vertex skipping check: $2 - 1 + 1 \leq 2 \Rightarrow 2 \leq 2$ - true. Call *CanBeSkipped(2)* function. $th = 2 - (2 - 1) = 1$. $bnd = 1$, since $Ca(8) = 1$ and $b[Ca(8)] \leq 1$. CanBeSkipped returns true because there are no adjacent vertices to currently expanding vertex 2 with index lower or equal to *bnd*. Skip this vertex it will not give bigger clique.
d = 2	$G = \{8, 6, 2\}$ *Cb[1]* = {8, 2} *Cb[2]* = {6}	3	Take the next vertex to expand. Bounding rule check: $Ca(6) = 2 \Rightarrow 2 < 3$ - true. Check the second condition $b[2] = 2 \Rightarrow 2 - 1 + 2 \leq 2$ – false. We continue search because 3 > */CBC/.* Vertex skipping check: $2 - 1 + 2 \leq 2 \Rightarrow 3 \leq 2$ - false. We continue search because 3 > */CBC/.* $N(6) = \{8\}$.
d = 3	$G = \{8\}$ *Cb[1]* = {8}	3	$cn = 1$. Check if number of color classes can possibly give a larger clique: $3 - 1 + 1 \leq 2 \Rightarrow 3 \leq 2$. We continue search because 3 > */CBC/.* Bounding rule check: $Ca(8) = 1 \Rightarrow 1 < 3$ - true. Check the second condition $b[1] = 1 \Rightarrow 3 - 1 + 1 \leq 2$ – false. We continue search because $3 >$ */CBC/.* Vertex skipping check: $3 - 1 + 1 \leq 1 \Rightarrow 3 \leq 2$ - false. We continue search because 3 > */CBC/.* Search current vertex neighbors and construct new depth. $N(8) = \emptyset$.

d = 4	$G = \emptyset$	3	Save current clique if it is larger than $/CBC/$. Current clique = {10, 6, 8}. 3 > 2 – true. $/CBC/ =$ {10, 6, 8}. Complete iteration. $b[3] = 3$.

Since there are no more color classes, we stop. The maximum clique is {10, 6, 8} and its size is three.

5. Results

In this chapter, we are going to compare the new algorithm to all the previously described ones. The following algorithms take part in testing: Carraghan and Pardalos, Östergård, VColor-u, VColor-BT-u, MCQ, MCR, MCS, MCS Improved and VRecolor-BT-u.

All algorithms were implemented on C# language using Visual Studio 2013 Professional (.NET Framework 4.5).

The first part of this chapter consists of randomly generated graphs. These random tests give a general overview of algorithms performance and therefore whether a new algorithm is worth to be used for clique finding. All test cases are divided by graphs density and for each density different algorithms are being tested. Note that algorithms that perform much worse compared to others are removed from test results figures to show behavior of the best algorithms.

The second part contains analysis of algorithm results of DIMACS instances. Each DIMACS graph has a special structure with response to some specific real problem. Four algorithms were tested with this benchmark: MCS, MCSI, VColor-BT-u and VRecolor-BT-u.

5.1 Generated test results

All algorithms were tested on randomly generated graphs. Randomness was generated using Random class from .NET Framework 4.5 which represents a pseudo-random number generator. Figure 5.1 demonstrates a function used for generation random graphs, where Graph is an object containing adjacency matrix inside Values array. Generation function takes number of vertices and density of a graph as parameters and returns a generated graph object.

```
public static Graph GenerateGraph(int nodes, double density)
{
      int numberOfEdges = Convert.ToInt32(Math.Round(nodes *
      (nodes - 1) *
           density / 2, 0));
```

```
        var graph = new Graph
        {
              Values = new bool[nodes, nodes],
              Edges = numberOfEdges
        };

        var random = new Random();
        Thread.Sleep(40);
        var random2 = new Random();

        int x, y;
        for (int i = 0; i < numberOfEdges; i++)
        {
              do
              {
                    x = random.Next(0, nodes);
                    y = random2.Next(0, nodes);
              } while (x == y || graph.Values[x, y]);
              graph.Values[x, y] = true;
              graph.Values[y, x] = true;
        }

        return graph;
}
```

Figure 5.1 Random graph generation code. (C# language)

Figures from 5.2 to 5.5 demonstrate that VRecolor-BT-u consumes the least amount of time than the fastest of the rest algorithms on sparse graphs where density is lower than 40%. On graphs where density is very low (about 10%) basic algorithms (Carraghan and Pardalos, Östergård) show really good results as they does not perform any additional operations like coloring, searching for initial solution, reordering and so on. Basic pruning formulas are really effective on such small density. Although VRecolor-BT-u outperforms them proving that skipping technique gives overall positive impact, even with a fact that algorithm needs to spend time for coloring and proving that a vertex can be skipped. On densities from 20% to 40%, the closest to VRecolor-BT-u are results of MCQ and MCR but the new algorithm performs about 20-25% faster. On all the figures from 5.2 to 5.10 y-axis shows time (in milliseconds) consumed by tested algorithms to find the maximum clique.

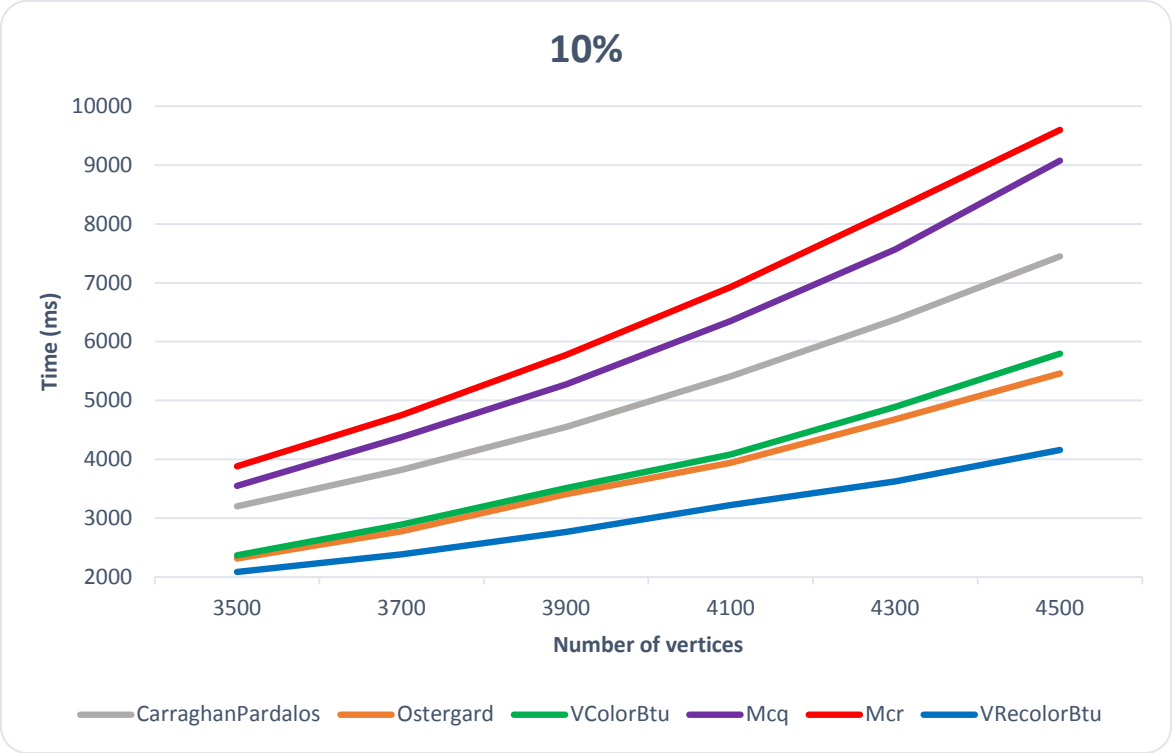

Figure 5.2 Randomly generated graphs test. Density 10%.

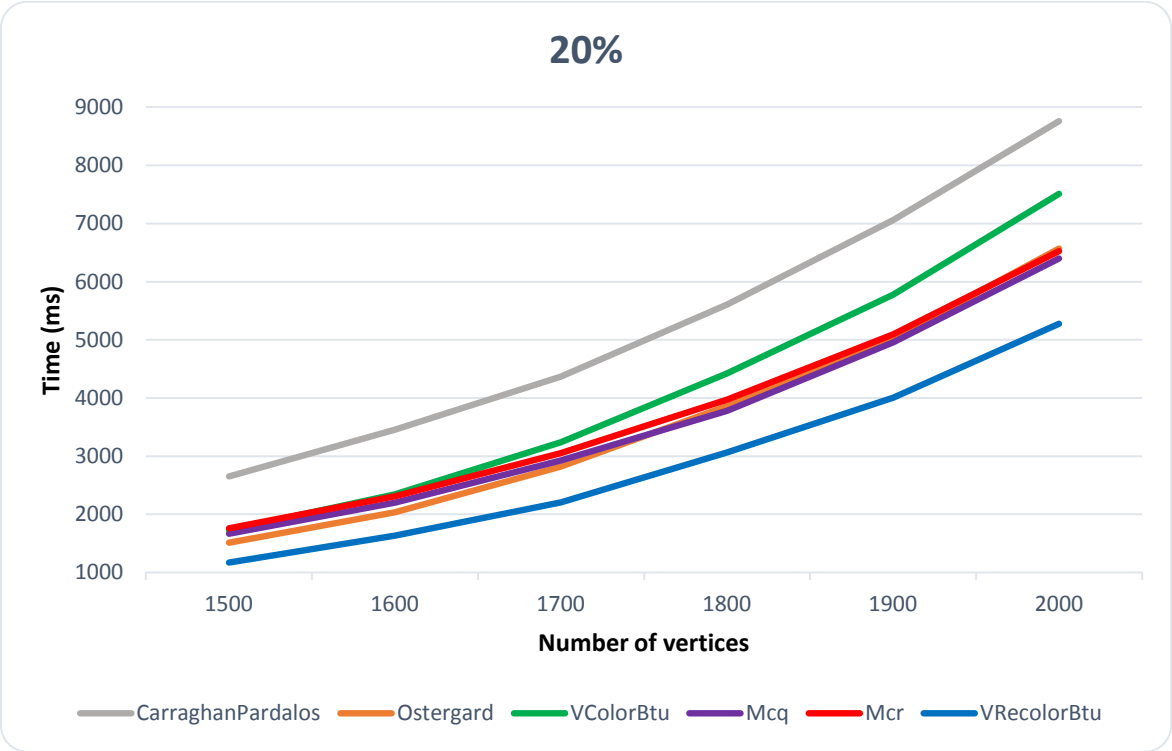

Figure 5.3 Randomly generated graphs test. Density 20%.

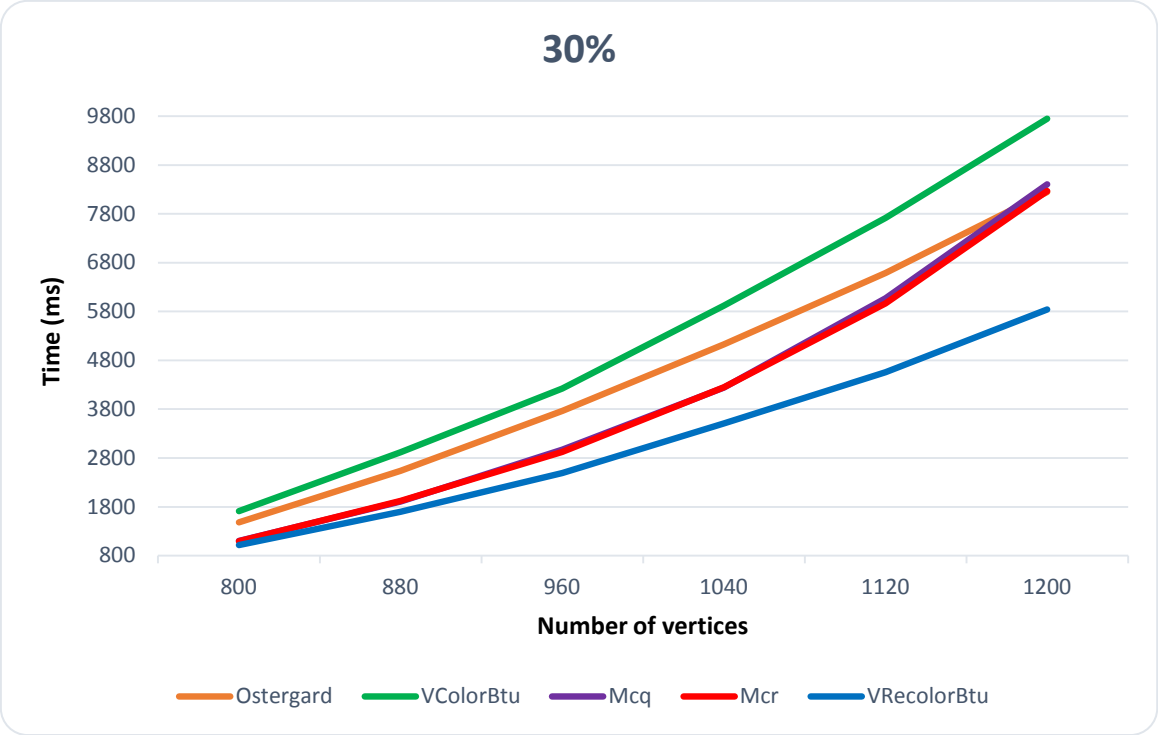

Figure 5.4 Randomly generated graphs test. Density 30%.

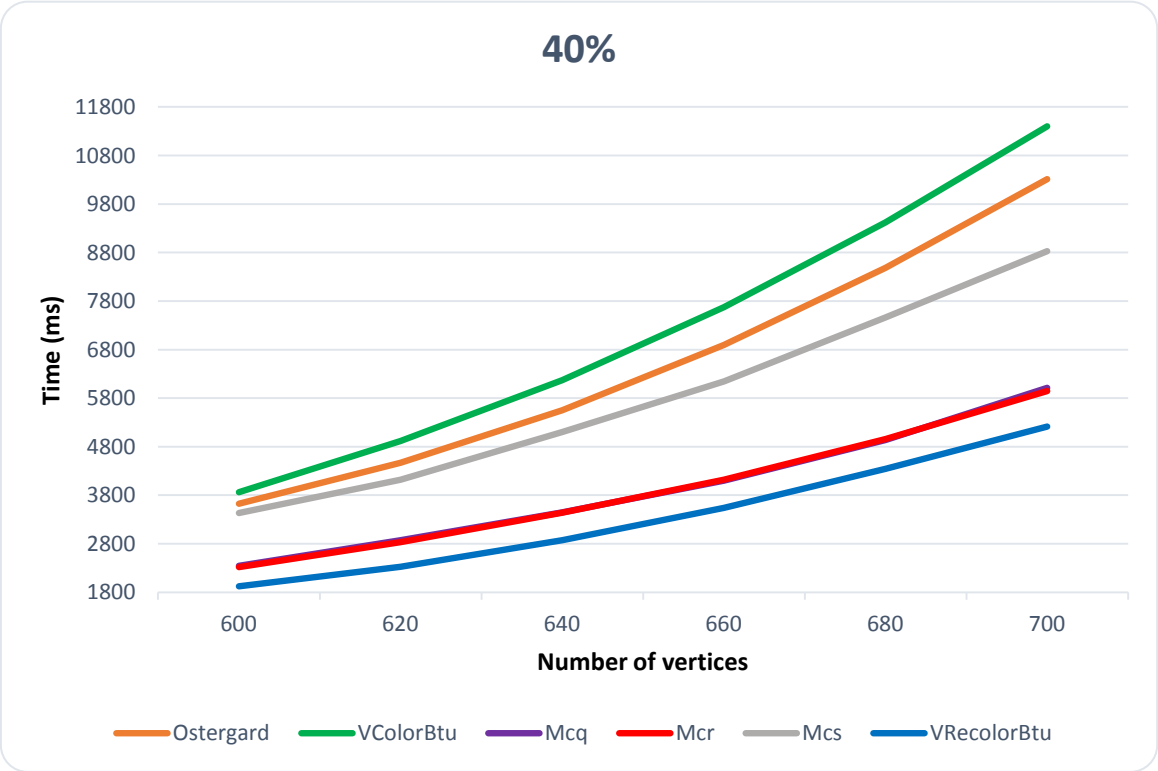

Figure 5.5 Randomly generated graphs test. Density 40%.

At first sight, there might be a strange behavior visible on a figure 5.6. VRecolor-BT-u time consumption is growing faster than MCQ and MCR have. Initially the new algorithm performs better (about 10%) when number of vertices is low (less than 440). It is clearly seen that already when number of vertices reaches 500 VRecolor-BT-u falls behind MCQ and MCR. This behavior can be explained by special constant, which determines what coloring algorithm is chosen for recoloring. When graphs density is 0.5 (which is our case), there is still a greedy coloring with swaps used for in-depth coloring, but after 0.55 density we switch to greedy coloring without swaps and this improvement gives significant impact on overall performance. If we move to figures 5.7 and 5.8 which demonstrate results on random graphs with 60% and 70% density you will see that VRecolor-BT-u shows stable best result from all the algorithms (about 5-10% faster).

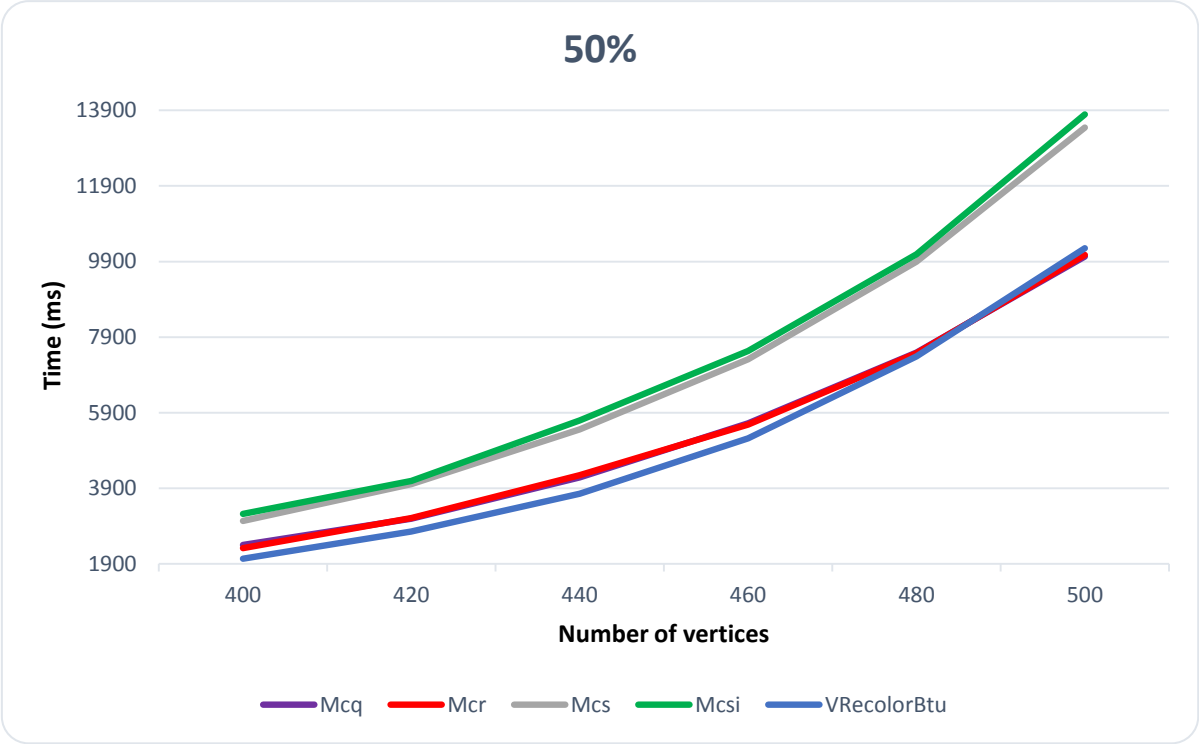

Figure 5.6 Randomly generated graphs test. Density 50%.

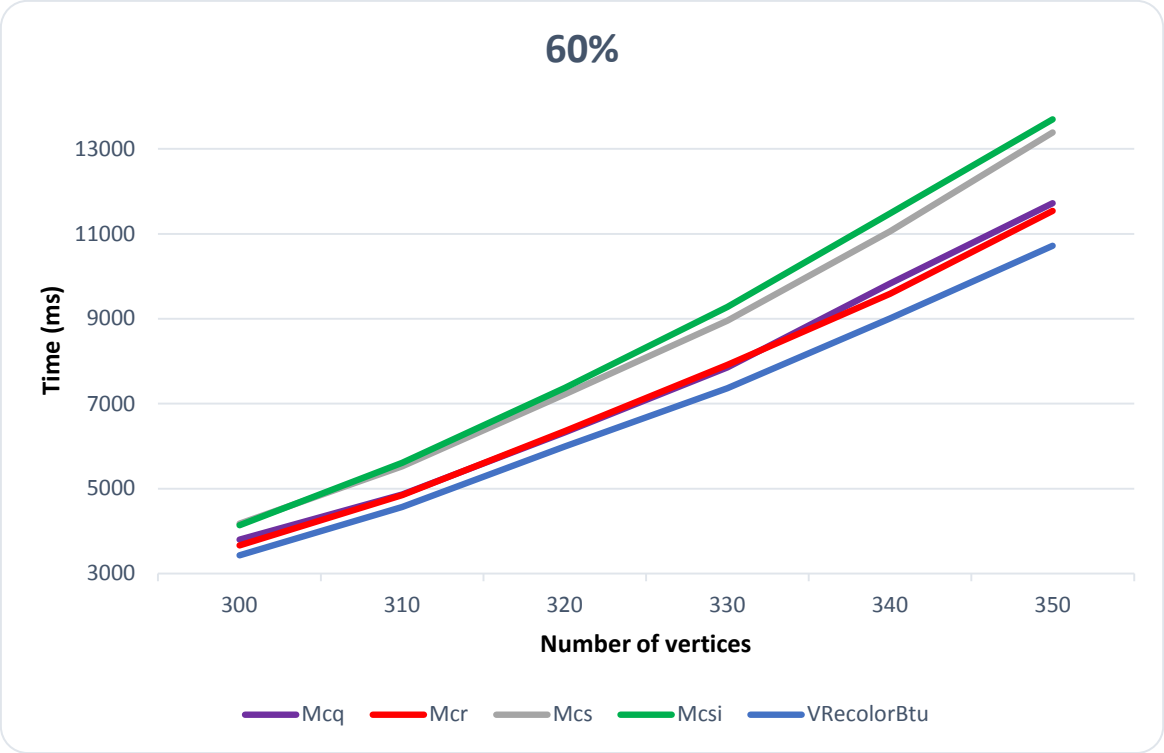

Figure 5.7 Randomly generated graphs test. Density 60%.

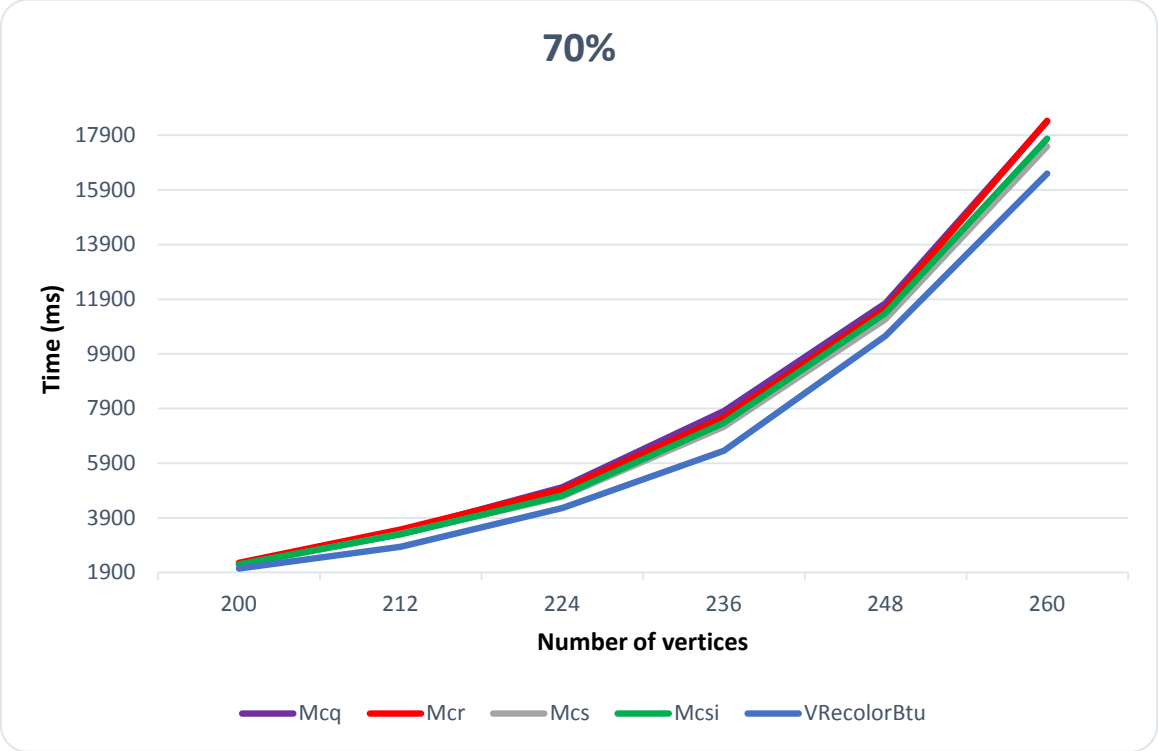

Figure 5.8 Randomly generated graphs test. Density 70%.

It is easy to see from figures 5.9 and 5.10 that VRecolor-BT-u algorithm's performance is not the best on dense graph. MCS and MCS Improved (MCSI) algorithms were specially designed for dense graphs and their techniques as in-depth vertex reordering or initial solution analysis result in lower time consumption. Although the new algorithm still demonstrates acceptable results and is able to find maximum clique on dense graphs where most other algorithms cannot.

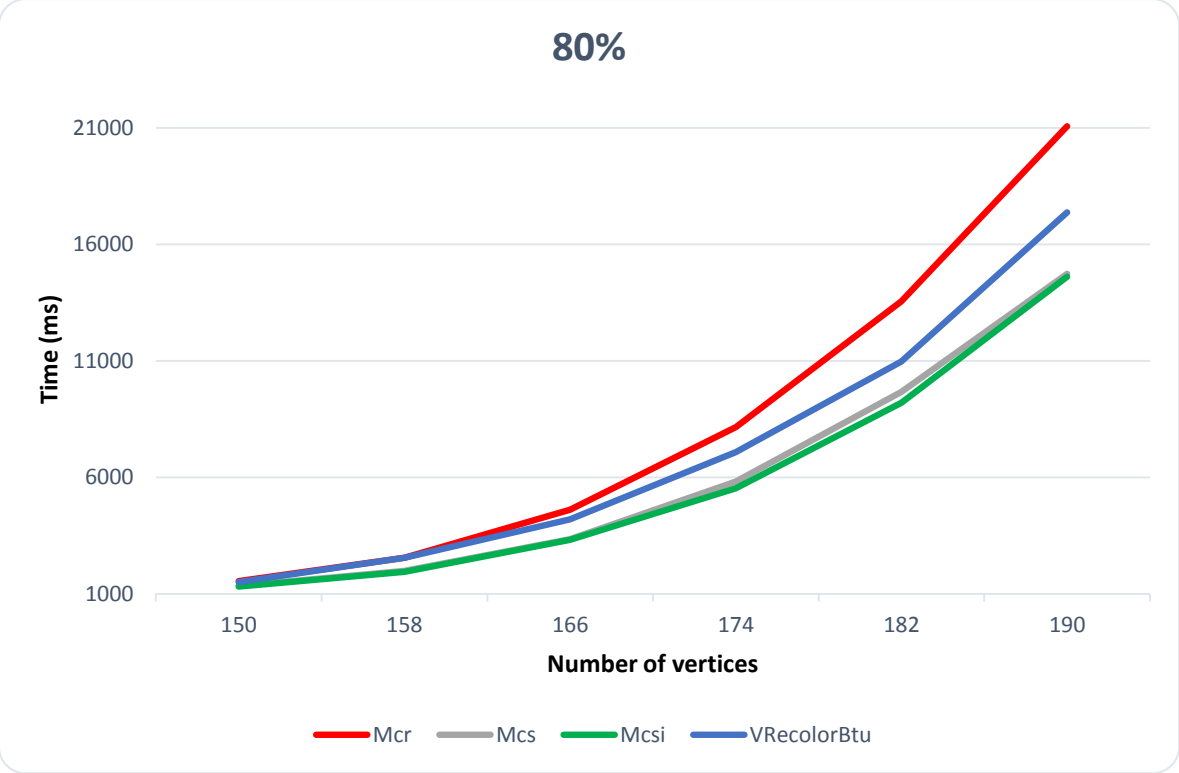

Figure 5.9 Randomly generated graphs test. Density 80%.

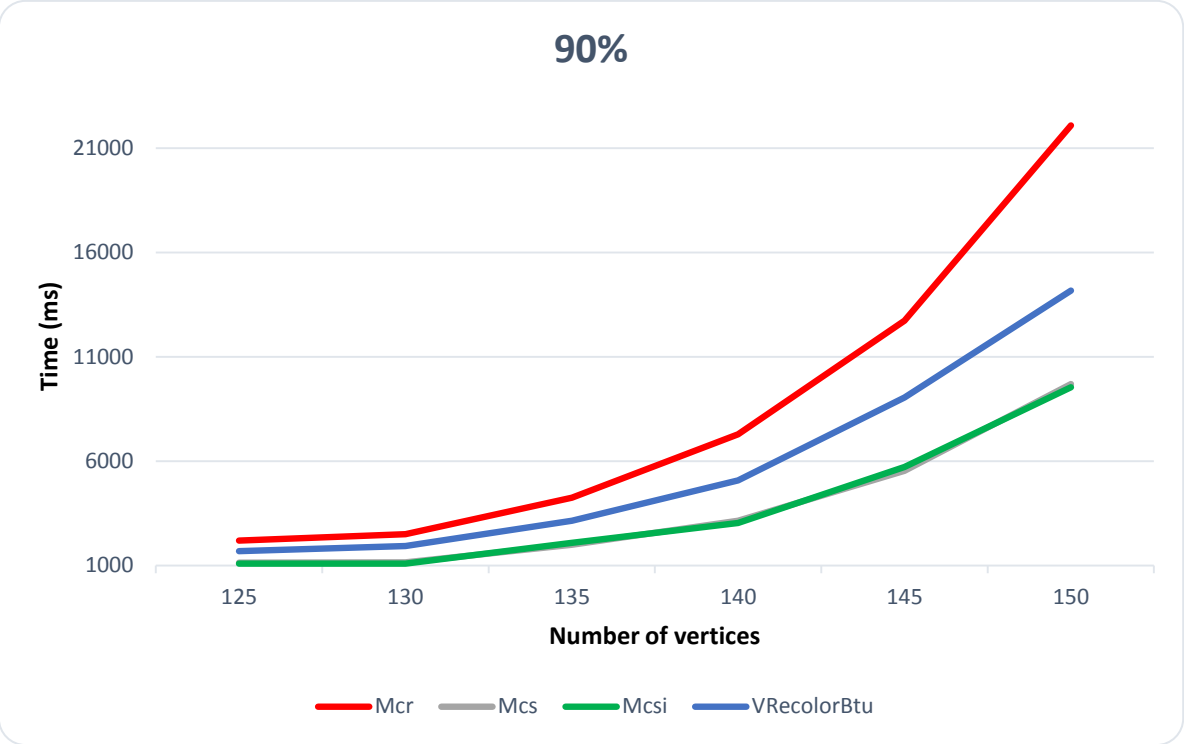

Figure 5.10 Randomly generated graphs test. Density 90%.

Based on randomly generated graph results we can conclude with the following statements:

- Graphs with densities lower than 50% are best solved using VRecolor-BT-u algorithm
- When graphs density is about 50%, there are three algorithms MCQ, MCR and VRecolor-BT-u that are the fastest but time consumption fluctuates a bit compared to each other
- If density of graph lies between 55% and 75%, then VRecolor-BT-u algorithm is a best choice
- For dense graphs with density more than 75%, MCS Improved is fastest algorithm.

5.2 DIMACS test results

In this subchapter, four algorithms are tested on DIMACS graph instances. These algorithms are MCS, MCSI, VColor-BT-u and VRecolor-BT-u. MCS and MCSI were chosen for testing because they demonstrated the best results on DIMACS instances of all modern

algorithms. VColor-BT-u is a predecessor of VRecolor-BT-u and is the best candidate to be compared with a new algorithm. With these tests, we are adding a new important characteristic as number of traversed branches. This parameter helps to understand how many branches each algorithm is analyzing and, of course, the better pruning formulas work the less branches are created. All considered algorithms are "branch and bound" which means the less branches analyzed the faster algorithm works. Number of branches demonstrates why one algorithm works faster than another, but time consumption and number of branches are not in a linear dependence. There are many factors that influence time consumption and branches number is only one of them.

Table 5.1 demonstrates that VRecolor-BT-u algorithm works faster compared to MCS and MCSI on almost all graphs where density is lower than 0.75. When VColor-BT-u and VRecolor-BT-u are compared, it is clearly seen that the new algorithm consumes less time for each test except "hamming" instances and "johnson16-2-4". This behavior can be explained simply if we move to table 6-2 and check number of branches created by these two algorithms. VRecolor-BT-u creates less branches on all the DIMACS instances taken into testing compared to VColor-BT-u, which means that on "hamming" instances improvements from additional skipping formula and recoloring are not giving positive effect on overall performance. On these specific graph types vertex skipping is almost useless as branch number reduction is insignificant.

It should be noted that on table 5.2 number of branches between MCS, MCSI and VColor-BT-u, VRecolor-BT-u pairs vary dramatically, this is a result of two diametrically different approaches. The first pair is based on Carraghan and Pardalos approach where initially all the vertices are taken into account and later analyzed graph size is decreasing, on the other hand Östergård's approach state that we start with the only vertex (the only color class in VColor-BT-u and VRecolor-BT-u) and later on graph size is growing. This means that branches by themselves differ a lot (and so number of branches as well) between these two different approaches.

One more detail about number of branches on table 5.2. There are quite a lot of instances where MCSI branch number is equal to zero. This algorithm is taking initial clique size from heuristic algorithm that is, of course, done at the very beginning. When heuristic solution is equal to the best possible clique size this results in a situation, when it is not needed to create any branches at all.

In general, DIMACS instances test proves results gained from randomly generated graphs testing. VRecolor-BT-u algorithm works better on densities lower than 75%.

Table 5.1 DIMACS graphs results. Time consumption (ms).

Graph	Size	Density	Time (ms)			
			MCS	MCSI	VColor-BT-u	VRecolor-BT-u
c-fat500-1.clq	500	0,04	44	229	6	2
c-fat500-10.clq	500	0,37	190	175	18	136
c-fat500-2.clq	500	0,07	27	112	4	1
c-fat500-5.clq	500	0,19	62	100	6	11
gen200_p0.9_44.clq	200	0,9	3867	2103	140082	21045
gen200_p0.9_55.clq	200	0,9	8988	98	3650	2276
hamming10-2.clq	1024	0,99	496	50636	**1290**	**61271**
hamming6-2.clq	64	0,9	0	1	**0**	**1**
hamming6-4.clq	64	0,35	0	1	0	0
hamming8-2.clq	256	0,97	10	38	**22**	**245**
hamming8-4.clq	256	0,64	633	654	**3**	**14**
johnson16-2-4.clq	120	0,76	702	800	**244**	**581**
johnson8-2-4.clq	28	0,56	0	0	1	0
johnson8-4-4.clq	70	0,77	1	3	0	0
keller4.clq	171	0,65	85	97	133	73
MANN_a27.clq	378	0,99	7201	291385	68105	10231
MANN_a9.clq	45	0,93	0	2	0	0
p_hat1000-1.clq	1000	0,24	2592	2788	3540	2046
p_hat300-1.clq	300	0,24	35	78	15	12
p_hat300-2.clq	300	0,49	108	109	464	238
p_hat300-3.clq	300	0,74	17796	9200	161323	16421
p_hat500-1.clq	500	0,25	110	198	139	91
p_hat500-2.clq	500	0,5	4816	2613	24391	8539
p_hat700-1.clq	700	0,25	386	453	239	236
san1000.clq	1000	0,5	7270	4989	410	945
san200_0.7_1.clq	200	0,7	18	39	889338	1819
san200_0.7_2.clq	200	0,7	16	45	3	5
san200_0.9_1.clq	200	0,9	2166	38	250	51
san200_0.9_2.clq	200	0,9	265	35	2828	1402
san400_0.5_1.clq	400	0,5	57	341	42	29

Table 5.2 DIMACS graphs results. Number of branches.

Graph	Size	Density	Branches			
			MCS	MCSI	VColor-BT-u	VRecolor-BT-u
c-fat500-1.clq	500	0,04	486	0	105	105
c-fat500-10.clq	500	0,37	374	0	8001	8001
c-fat500-2.clq	500	0,07	474	0	351	351
c-fat500-5.clq	500	0,19	436	0	2080	2080
gen200_p0.9_44.clq	200	0,9	38520	16605	72627446	932250
gen200_p0.9_55.clq	200	0,9	124975	544	1629229	97769
hamming10-2.clq	1024	0,99	512	136714	140033	131328
hamming6-2.clq	64	0,9	32	0	569	528
hamming6-4.clq	64	0,35	82	80	138	70
hamming8-2.clq	256	0,97	128	0	8849	8256
hamming8-4.clq	256	0,64	31794	31782	1524	788
johnson16-2-4.clq	120	0,76	237952	256098	489432	323070
johnson8-2-4.clq	28	0,56	26	22	74	44
johnson8-4-4.clq	70	0,77	126	114	692	252
keller4.clq	171	0,65	6978	7317	203053	11236
MANN_a27.clq	378	0,99	9091	1893248	7528324	55389
MANN_a9.clq	45	0,93	43	149	375	189
p_hat1000-1.clq	1000	0,24	120465	116675	5457636	357619
p_hat300-1.clq	300	0,24	1519	964	16737	2538
p_hat300-2.clq	300	0,49	2027	1368	272364	24826
p_hat300-3.clq	300	0,74	228931	121147	88917523	664515
p_hat500-1.clq	500	0,25	7953	7374	213594	19821
p_hat500-2.clq	500	0,5	63031	28547	11620047	584983
p_hat700-1.clq	700	0,25	22447	13656	304679	45157
san1000.clq	1000	0,5	83831	0	57284	13356
san200_0.7_1.clq	200	0,7	403	0	2241214630	547738
san200_0.7_2.clq	200	0,7	768	0	1490	398
san200_0.9_1.clq	200	0,9	31555	0	97985	3350
san200_0.9_2.clq	200	0,9	2063	0	1502178	87250
san400_0.5_1.clq	400	0,5	1562	0	16340	1241

6. Conclusion

The main topic of this study was to develop a new improved algorithm for maximum clique finding. Only undirected, unweighted graphs were researched. Importance of the problem was stated in subchapter 1.3. Clique finding problem belongs to the NP-complete class, which means that finding a better algorithm for this kind of problems allows improving all the problems transformed to it. That is why better understanding of the clique problem provides us with a better solution for almost any other NP problem. As currently described problem belongs to NP-compete, all the existing algorithms complexity (for this problem) can be described with exponential functions, which means that even a small increase in the size of the problem can result in additional days or weeks of work time. Therefore, development of a better algorithm even for some specific graph groups can save this working time and can seriously influence different areas of real life.

6.1 Summary

With this resume, we are going to summarize all the work done to reach the goals stated in the chapter 1.4. They all are successfully completed and described in the current work. Although, there is enough space for further improvements, that will be presented in the next subchapter.

The two basic algorithms for finding maximum clique are studied in chapter 2 giving introduction to branch and bound algorithms. There are two general approaches of traversing a graph. The first one is Carraghan and Pardalos algorithm [Carraghan, Pardalos 1990], which starts considering all the vertices of a graph. On the other hand, the second approach is demonstrated by Östergård's algorithm that uses reversed search, taking into account only one vertex initially and constantly adds vertices one by one. In addition, other basic coloring, independent set and vertex cover finding heuristic algorithms are reviewed.

Chapter 3 contains description of the most efficient modern algorithms nowadays. Studying these algorithms allows understanding what are the main properties and upgrades, which influence algorithms performance the most. VColor-u and VColor-BT-u algorithms published by D. Kumlander [Kumlander 2005] demonstrate a high positive impact of heuristic coloring on exact algorithms performance. What is more D. Kumlander applied coloring to both basic approaches of finding maximum clique. Such algorithms as MCQ [Tomita, Seki 2003], MCR [Tomita, Kameda 2007] and MCS [Tomita, Sutani, Higashi, Takahashi, Wakatsuki 2010] show that initial vertex ordering does matter and needs to be chosen properly. Moreover, in-depth heuristic coloring proved its efficiency and confirmed the fact that only initial coloring is not enough as the deeper search goes the more diffused initial color classes become. MCS algorithm introduced a notion of color number threshold and demonstrated how it can be successfully used to reduce the amount of expanded vertices, therefore lowering unnecessary branch creation. Finally, MCS Improved algorithm [Batsyn, Goldengorin, Maslov, Pardalos 2014] showed that initial clique value obtained using good heuristic combined with in-depth clique vertices analysis can sometimes reduce the number of produced branches dramatically. As can be clearly seen from all the modern algorithm heuristic has a great positive overall impact on the clique finding exact algorithms.

The new maximum clique algorithm called VRecolor-BT-u is demonstrated in chapter 4. This algorithm is a successor of VColor-BT-u and is constructed based on reversed search by color classes. The main idea of the new algorithm is quite simple: we need to apply coloring on each depth to preserve the most up-to-date color classes and combine updated vertex colors with the reversed search approach. At the first sight, the idea of in-depth recoloring might be unclear as reversed search is built around initial color classes, but introduction of a new skipping technique instead of pruning allows avoiding this conflict. Furthermore, there are two different greedy coloring algorithms (with swaps and without swaps) used for initial and in-depth coloring. Experimentally gained constants, which depend on graph density, determine which coloring is applied (subchapter 4.2). The algorithm is described as a step-by-step operation set in subchapter 4.3. The previous experience with different algorithms realization shows that it is very easy to miss or distort the meaning of some inaccurately described details. Each small mistake in implementation might lead to extreme performance drops or result in improper solutions. To prevent such cases there are two examples in subchapters 4.4 and 4.5 which demonstrate VRecolor-BT-u workflow in

details. Moreover, the implementation of the new algorithm, written in C# language, is attached in the appendix 1. These subchapters make it easy to understand and implement the new algorithm and the exact implementation on a real programming language allows excluding all the possible misunderstandings.

One of the most important things to do with a new algorithm is a proper testing. All the implemented algorithms were compared using two types of tests in chapter 5. The first one is randomly generated graphs tests. Generated tests allow obtaining comparative diagrams that graphically demonstrate time consumption of different algorithms. The new algorithm shows the best results on the graphs with low or average densities and loses only on dense graphs to MCS and MCSI algorithms specially designed for high densities. The second type of testing is DIMACS benchmark instances. Firstly, these instances already contain the best solution, so small DIMACS graph are very convenient to use as the smoke tests for a new algorithm. Secondly, these test instances allows testing the algorithm on close to real life problems as they are constructed based on real tasks. Moreover, in addition to time consumption comparison there are branch number results. Number of branches is not the primary characteristic but allows explaining why one algorithm works faster or slower than the other does. VRecolor-BT-u produces less branches that its predecessor for all the DIMACS instances. However, there are some cases where the new algorithm consumes more time. Decreasing branch number resulting in performance degradation might be misleading at a glance, but can be described with a simple fact that on some special cases additional in-depth recoloring consumes a lot of time while skipping technique is practically not working. As a result, we have a slightly lower branch number but increased time consumption.

Finally, it was noted that each graph should be solved by a different algorithm with response to graphs density. On low to mid densities, it is advised to use VRecolor-BT-u algorithm while the best option for dense graphs is MCS Improved algorithm.

6.2 Future studies

In this subchapter, we are going to introduce some ideas for further studies. First of all, there are multiple possible improvements for VRecolor-BT-u algorithm:

- Improved initial coloring. Only greedy coloring is currently applied, but the less color classes we have the less iterations will be performed. Moreover, initial coloring is applied only once before branching starts, which means that there might be more complex and time-consuming coloring applied. That time spend on initial coloring should be compensated due to reduction of iterations number.
- Improved in-depth coloring. As long as recoloring is used each time a new branch is created, it is not acceptable to apply any time consuming in-depth coloring algorithm. There should be a balanced solution found between the "*good*" coloring and time consumption. What is more, "*good*" recoloring does not mean the least number of color classes; the main goal is to assign as much vertices as possible to color numbers that are lower or equal to the threshold value. Well optimized for these specific needs coloring algorithm might be a key to imposing performance improvements.
- Overall subgraph analysis on each new iteration. With each iteration, we are adding a new color class into consideration. As we know this might increase current clique value by one at maximum. The easiest way of analysis is to check whether any vertex of a new color class can be added to the already existing clique. If yes, then the whole iteration can be skipped. There might be the more complex ways of analysis introduced such as obtaining maximum clique of a new subgraph by heuristic algorithm. If heuristically gained value is bigger than current clique, then iteration is skipped.

As rough computational results show, when using Carraghan and Pardalos searching approach (without reversed search), largest clique is found at 30% of total time consumed. The rest 70% algorithm is trying to prove that current clique is the largest one. The situation can be even worse on dense graph with a lot of parallel cliques with the same size. Incomplete solution and excessive expectations are two interesting topics to be studied that might improve this field. Incomplete solution studies how fast the maximum clique is found. Using this data, we might predict at what point we have already obtained solution. After that point there should be a way, other that analyzing all the rest vertices, to prove that current clique is the largest one. Excessive expectation is a proposal of searching a clique with initial clique value n larger than expected. If an excessive clique is not found this means that value n is an upper bound for the maximum clique. Combination of these two approaches might fasten already existing algorithms even more.

Kokkuvõtte

Selle uurimistöö põhiteemaks on välja töötada uus täiustatud algoritm suurima kliki leidmiseks. Uuritud on ainult orienteerimata ja kaalumata graafe. Probleemi olulisus on välja toodud alapeatükis 1.3. Klikileidmise problem kuulub NP-täielik-klassi, mis tähendab, et sedalaadi probleemide lahendamiseks parema algoritmi leidmine võimaldab lahendada kõiki probleeme, mis on selliseks ümber muudetud. Seetõttu klikiprobleemi parem mõistmine pakub meie jaoks paremat lahendust peaagu igale muule NP probleemile. Kuna praegu kirjeldatud probleem kuulub NP-täieliku hulka, siis kõigi olemasolevate algoritmide keerukust (selle probleemi jaoks) saab kirjeldada eksponentfunktsiooni abil, mis tähendab, et pisimgi probleemi suuruse tõus võib kaasa tuua lisatööpäevi või isegi nädalaid. See tähendab, et parema algoritmi väljatöötamine kas või mõne spetsiifilise graafirühma jaoks säästab tööaega ning avaldab olulist mõju erinevatele reaalelu valdkondadele.

Tehtud töö

Selle resümeega võtame kokku kogu töö, mis on tehtud selleks, et saavutada peatükis 1.4 kirjeldatud eesmärke. Kõik on edukalt lõpule viidud ja kirjeldatud käesolevas töös. Ehkki muidugi oleks ruumi veel edasisekski täiustamiseks, mida kirjeldatakse järgmises alapeatükis.

Kahte põhialgoritmi maksimaalse kliki leidmiseks käsitletakse 2. peatükis, kus on ka sissejuhatus harude-tõkete algoritmidele. On kaks peamist lähenemisviisi graafist üle liikumiseks. Kiireim on Carraghani and Pardalos algoritm [Carraghan, Pardalos 1990], mis hakkab arvestama graafi kõiki tippe. Teisest küljest, teistsugust lähenemisviisi tutvustab Östergårdi algoritm, mis kasutab pöördotsingut, võttes alguses arvesse ainult üht tippu, seejärel lisab ükshaaval tippe juurde. Lisaks vaadeldakse muid põhivärve, sõltumatut hulka ja tipu katet, mis aitavad leida heuristilisi algoritme.

3. peatükis on tänapäeva kõige tõhusamate ja moodsamate algoritmide kirjeldus. Nende algoritmide uurimine aitab aru saada nende põhilistest omadustest ja uuendustest, mis algoritmide suutlikkust kõige rohkem mõjutavad. D. Kumlanderi poolt avaldatud VColor-u ja

VColor-BT-u algoritmid [Kumlander 2005] demonstreerivad heuristilise värvimise kõrget positiivset mõju algoritmi täpsele sooritusKiirusele. Veelgi enam, D. Kumlander rakendas värvimist mõlema põhilähenemisviisi puhul suurima kliki leidmiseks. Sellised algoritmid nagu MCQ [Tomita, Seki 2003], MCR [Tomita, Kameda 2007] ja MCS [Tomita, Sutani, Higashi, Takahashi, Wakatsuki 2010] näitavad, et algne tippude järjestus on oluline ja seda tuleb hoolikalt valida. Veelgi enam, sügavuti heuristiline värvimine tõestas oma tõhusust ja kinnitas fakti, et esialgne värvimine ei ole piisav, ja mida põhjalikum on uurimine, seda laialivalguvamaks muutuvad algsed värviliigid. MCS algoritm võttis kasutusele värvinumbrite mõiste ning demonstreeris, kuidas on võimalik vähendada avardatud tippude arvu, mis omakorda vähendab tarbetut harude loomist. Lõpuks MCS täiustatud algoritm Batsyn, Goldengorin, Maslov, Pardalos 2013] näitas, et esialgne klikiväärtus, mis saavutati kasutades head heuristikat kombinatsioonis sügavuti teostatud klikitippude analüüsiga võib mõnikord dramaatiliselt vähendada toodetud harude arvu. On selge, et moodsate algoritmide puhul on heuristikal tohutu positiivne mõju täpsete algoritmide klikileidmisele.

Uut suurima kliki mehhanismi nimega VRecolor-BT-u'd kirjeldatakase 4. peatükis. See algoritm on VColor-BT-u järeltulija ning on konstrueeritud värviliikide poolt pöördotsingu põhjal. Uue algoritmi idee on üsna lihtne: kanname värvi peale igal sügavusel, et säilitada kõige uuemaid värviliike ja kombineerida uusimaid tipuvärve pöördotsingu abil. Esmapilgul võib idee sügavuti ülevärvimisest tunduda ebareaalsena, kuna pöördotsing põhineb algsetel värviliikidel, kuid uus vahelejätmistehnika kärpimise asemel võimaldab sellise konflikti vältimist.

Lisaks on kaks erinevat ahnet värvivat algoritmi (värvivahetusega ja ilma), mida kasutatakse nii esialgseks kui sügavuti värvimiseks. Katse tulemusena saadud konstandid, mis olenevad graafi tihedusest, määravad ära, millist värvi kasutada (alapeatükk 4.2). Algoritmi kui samm-sammult teostatavat tegevuste jada kirjeldatakse alapeatükis 4.3. Eelnevad kogemused erinevate algoritmide teostamisel näitavad, et on väga lihtne tähelepanuta jätta või moonutada mõningaid ebatäpselt kirjeldatud üksikasjade tähendusi. Iga pisemgi viga algoritmi rakendamisel võib tuua kaasa sooritusE halvenemise või ebaõige lahenduse. Selliste juhtumite ärahoidmiseks on alapeatükkides 4.4 ja 4.5 toodud kaks näidet, mis demonstreerivad üksikasjalikult VRecolor-BT-u tööprotsessi. Lisaks sellele on C# keeles kirjutatud uue algoritmi rakendusjuhend manusena Lisas 1. Need alapeatükid muudavad uue

algoritmi mõistmise ja rakendamise lihtsamaks. Täpne rakendamine päris programmeerimiskeeles aitab välistada kõikvõimalikke arusaamatusi.

Üks väga oluline asi uue algoritmi puhul on korralik testimine. Kõiki rakendatavaid algoritme võrreldi kaht liiki testide abil peatükis 5. Esimene neist on juhuslikult genereeritud graafitest. Genereeritud testid võimaldavad saada võrdlevaid diagramme, mis näitavad graafiliselt erinevate algoritmide tarbimist. Uus algoritm näitab parimaid tulemusi madala või keskmise tihedusega graafil, jäädes alla ainult tihedatele graafidele MCS ja MCSI algoritmide jaoks, mis ongi spetsiaalselt disainitud kõrgete tiheduste jaoks. Teine testimisliik on DIMACS võrdlused. Esiteks, need näited sisaldavad juba iseenesest parimat lahendust, nii et väikesed DIMACS graafid on väga sobilikud kasutamiseks ”suitsutestidena” uue algoritmi jaoks. Teiseks, sel viisil saab testida algoritme reaalse elu probleemide jaoks, kuna nende väljatöötamine põhineb reaalelu ülesannetel. Veelgi enam, ajakulu võrdluse kõrval näitavad nad ka harude arvu tulemusi. Harude arv pole küll kõige olulisem karakteristik, kuid annab selgust selles, miks üks algoritm toimib kiiremini kui teine. Kõik DIMACS juhtumid näitavad, et VRecolor-BT-u toodab vähem harusid kui tema eelkäija. Ometi on juhtumeid, kus uus algoritm vajab rohkem aega. Vähenev harude arv, mille tulemuseks on jõudluse nõrgenemine, võib olla esmapilgul eksitav, aga seda seletab lihtne fakt, et teatud juhtudel sügavuti ülevärvimine võtab palju aega, samal ajal kui vahelejätutehnika prakiliselt ei toimi. Tulemuseks on väiksem harude arv, aga suurem ajakulu. Lõpuks täheldati, et iga graaf tuleks lahendada erineva algoritmi abil, mis vastab graafi tihedusele. Madala ja keskmise tiheduse puhul on soovitav kasutada VRecolor-BT-u algoritmi, suurema tihedusega graafide puhul on parim variant MCS Improved (täiustatud) algoritm.

Tulevased uuringud

Selles alapeatükis tutvustame mõnigaid ideid edasisteks uurimusteks. Kõigepealt on mitmeid võimalikke täiustusi VRecolor-BT-u algoritmi jaoks:

- Täiustatud esialgne värvimine. Momendil on saada ainult ahnet värvimist, kuid mida vähem värviliike me kasutame, seda vähem iteratsioone toimub. Veelgi enam, esialgne värvimine tehakse ainult üks kord, enne kui harude moodustamine algab, mis tähendab, et võib ette tulla keerulisemat ja rohkem aeganõudvat värvimist. Aeg, mis kulutatakse esialgsele värvimisele, peaks saama korvatud iteratsioonide arvu vähendamisega.

- Täiustatud sügavuti värvimine. Nii kaua kui iga uue haru loomise puhul viiakse läbi ülevärvimine, ei ole vastuvõetav, et kohaldatakse mistahes aeganõudvat sügavuti värvimise algoritmi. Tuleks leida tasakaalustatud lahendus ”*hea”* värvimise ja vastava ajakulu vahel. Edasi, ”*hea”* ülevärvimine ei tähenda võimalikult väheseid värviliike, põhieesmärgiks on vastavusse seada nii palju tippe kui võimalik, nii et see oleks madalam kui piirväärtus või sellega võrdne. Kui nende spetsiifiliste vajadustega optimaalselt ümber käia, võib värvimise algoritm olla võti jõudluse tõstmise jaoks.
- Üldine alamgraafi analüüs iga uue iteratsiooni puhul. Iga iteratsiooni puhul võtame kaalumisele uue värviliigi. Nagu teame, see võib tõsta praegust klikiväärtust maksimaalselt 1 võrra. Lihtsaim viis analüüsi teha on kontrollida, kas ühtegi uut värviliiki saab lisada olemasolevale klikile. Kui saab, siis võib kogu iteratsiooni ära jätta. On olemas kindlasti ka keerulisemaid viise analüüside tutvustamiseks, nagu näiteks saavutada uue alamgraafi suurim klikk, kasutates heuristilist algoritmi. Kui heuristilisel teel saadud väärtus on suurem, kui käesolev klikk, siis iteratsioon jäetakse vahele. Olles nende spetsiifiliste vajaduste jaoks hästi optimeeritud, võib värviv algoritm olla võtmeks sooritusjõudluse parandamisel.

Nagu töötlemata arvutuslikud tulemused näitvad, kasutades Carraghani ja Pardalose otsivat lähenemisviisi (jättes välja pöördotsingu), siis suurim leitud klikk oli 30% kogu kulutatud ajast. Ülejäänud 70% algoritm üritab tõestada, et käesolev klikk on suurim. Olukord võib olla hullem tiheda graafi puhul, millel on palju ühesuguse suurusega paralleelklikke. Puudulik lahendus ja liigne ootus on kaks huvitavat teemat, mille uurimine võiks seda valdkonda paremaks muuta. Puudulik lahendus uurib, kui kiiresti on võimlik leida kiireimat klikki. Seda andmebaasi kasutades võime ennustada, mis hetkel lahenduseni jõutakse. Seejärel tuleb leida viis, mõni muu kui kõigi ülejäänud tippude analüüsimine, tõestamaks et just see klikk on kõige suurem. Liigne ootus on sellise kliki otsimine, mille esialgne väärtus n on oodatust kõrgem. Kui liigset klikki ei leita, tähendab see, et väärtus n on suurima kliki jaoks ülempiir. Nende kahe lähenemisviisi ühendamine võib juba olemasolevaid algoritme veelgi enam kiirendada.

References

Andrade DV, Resende MGC, Werneck RF (2012) Fast local search for the maximum independent set problem. J Heuristics 18(4), pp 525–547

Batsyn M., Goldengorin B., Maslov E., Panos M. Pardalos (2014) Improvements to MCS algorithm for the maximum clique problem. Journal of Combinatorial Optimization 27(2), pp 397-416

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

Chartrand G (1985) *The Königsberg Bridge Problem: An Introduction to Eulerian Graphs*, Introductory Graph Theory. New York: Dover 3(1), pp 51-60

Clarkson K. (1983) *A modification to the greedy algorithm for vertex cover problem*, Information Processing Letters 16(1), pp 23-25

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

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

Kumlander D. (2005) Some Practical Algorithms to Solve The Maximum Clique Problem

Tomita E, Kameda T (2007) An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. J Global Optim 37(1), pp 95–111

Tomita E, Seki T (2003) An efficient branch-and-bound algorithm for finding a maximum clique. In: Proceedings of the 4th international conference on discrete mathematics and theoretical computer science, DMTCS'03. Springer-Verlag, Berlin, Heidelberg, pp 278–289

Tomita E, Sutani Y, Higashi T, Takahashi S, Wakatsuki M (2010) A simple and faster branch-and-bound algorithm for finding a maximum clique. In: Proceedings of the 4th international conference on algorithms and computation, WALCOM'10. Springer-Verlag, Berlin, Heidelberg, pp 191–203

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

Clique Benchmark Instances ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/

The Microsoft Developer Network (MSDN) https://msdn.microsoft.com/en-us/library/system.random(v=vs.110).aspx

Appendix 1

Base class Algorithm

```
using System.Diagnostics;
using MaximumClique.Base;

namespace MaximumClique.NewAlgorithms
{
    public abstract class Algorithm
    {
        protected readonly Stopwatch _stopwatch = new
Stopwatch();
        private bool _solved;
        protected Graph Graph;

        protected Algorithm(Graph graph)
        {
            Graph = graph;
            SolutionFoundElapsed = 1;
        }

        #region Properties

        public bool IsSolved
        {
            get
            {
                return _solved;
            }
        }

        public double Elapsed
        {
            get { return _stopwatch.ElapsedMilliseconds; }
        }

        protected long branches;
        public long Branches
        {
            get { return branches; }
        }
```

```
        public double SolutionFoundElapsed { get; protected
set; }

        private int _nodesNumber = -1;
        protected int NodesNumber
        {
            get
            {
                return _nodesNumber == -1 ? (_nodesNumber =
Graph.Values.GetLength(0)) : _nodesNumber;
            }
        }

        public abstract int Result { get; }

        #endregion //Properties

        protected abstract void Solution();

        public void Start()
        {
            _stopwatch.Start();
            Solution();
            _stopwatch.Stop();
            _solved = true;
        }
    }
}
```

VRecolor-BT-u algorithm

```
using System.Linq;
using MaximumClique.Base;

namespace MaximumClique.NewAlgorithms
{
    public class VRecolorBtu : Algorithm
    {
        private int maxCliqueSize;
        private int[,] levelNodes;
        private int initialColorsNumber;
        private int[][] initialColorClasses;
        private int[] initialNodesNumInColorClass;
        private int[] initialColors;
        private int[,] inDepthColors;
        private int[] numberOfNodesArr;
        private int[] inDepthElementIndex;
```

```
        private int[,] skippedNodes;
        private int[] skippedNodesNumber;
        private int[] cache;

        public VRecolorBtu(Graph graph)
            : base(graph)
        {
            levelNodes = new int[NodesNumber, NodesNumber];
            initialColorClasses = new int[NodesNumber][];
            initialNodesNumInColorClass = new
int[NodesNumber];
            initialColors = new int[NodesNumber];
            inDepthColors = new int[NodesNumber, NodesNumber];
            numberOfNodesArr = new int[NodesNumber + 1];
            inDepthElementIndex = new int[NodesNumber + 1];

            skippedNodes = new int[NodesNumber, NodesNumber];
            skippedNodesNumber = new int[NodesNumber + 1];
            cache = new int[NodesNumber];
        }

        public override int Result
        {
            get { return maxCliqueSize; }
        }

        protected override void Solution()
        {
            if (Graph.Density < 0.35)
                InitialColoringWithSwaps();
            else
                InitialColoring();

            var inDepthDegree = new int[NodesNumber];
            for (int c = 0; c < initialColorsNumber; c++)
            {
                skippedNodesNumber[0] = 0;
                int depth = 0;
                numberOfNodesArr[depth] = 0;
                inDepthDegree[depth] = 0;
                for (int i = 0; i <= c; i++)
                {
                    for (int j = 0; j <
initialNodesNumInColorClass[i]; j++)
                    {
                        levelNodes[depth,
numberOfNodesArr[depth]] = initialColorClasses[i][j];
                        numberOfNodesArr[depth]++;
                    }
```

```
                }
                inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;
                while (depth >= 0)
                {
                    int inDepthIndex =
inDepthElementIndex[depth];
                    if (inDepthIndex == -1)
                    {
                        depth--;
                        continue;
                    }
                    int p = levelNodes[depth, inDepthIndex];
                    var color = initialColors[p - 1];
                    if (color < c + 1 && depth + cache[color -
1] <= maxCliqueSize)
                    {
                        depth--;
                        continue;
                    }
                    if ((depth + inDepthColors[depth, p - 1]
                        <= maxCliqueSize) &&
CanBeSkipped(inDepthIndex, depth, c + 1))
                    {
                        skippedNodes[depth,
skippedNodesNumber[depth]] = p;
                        skippedNodesNumber[depth]++;
                        inDepthElementIndex[depth]--;
                        continue;
                    }
                    branches++;
                    int prevDepth = depth;
                    depth++;
                    numberOfNodesArr[depth] = 0;
                    inDepthElementIndex[depth] = 0;

                    for (int i = 0; i < inDepthIndex; i++)
                    {
                        if (Graph.Values[levelNodes[prevDepth,
inDepthIndex] - 1, levelNodes[prevDepth, i] - 1])
                        {
                            levelNodes[depth,
numberOfNodesArr[depth]] = levelNodes[prevDepth, i];
                            numberOfNodesArr[depth]++;
                        }
                    }
                    for (int i = skippedNodesNumber[prevDepth]
- 1; i >= 0; i--)
                    {
```

```
                        if (Graph.Values[levelNodes[prevDepth,
inDepthIndex] - 1, skippedNodes[prevDepth, i] - 1])
                        {
                            levelNodes[depth,
numberOfNodesArr[depth]] = skippedNodes[prevDepth, i];
                            numberOfNodesArr[depth]++;
                        }
                    }

                    inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;

                    if (numberOfNodesArr[depth] > 0)
                    {
                        int colNum = Graph.Density < 0.55 ?
RecolorWithSwaps(depth) : Recolor(depth);

                        if (depth + colNum <= maxCliqueSize)
                            depth--;
                    }
                    else
                    {
                        if (depth > maxCliqueSize)
                        {
                            maxCliqueSize = depth;
                            break;
                        }
                        depth--;
                    }
                    inDepthElementIndex[prevDepth]--;

                }
                cache[c] = maxCliqueSize;
            }
        }

        private bool CanBeSkipped(int vertIndex, int depth,
int currentColor)
        {
            int threshold = maxCliqueSize - depth;
            int vert = levelNodes[depth, vertIndex];

            // on current depth on what vertex index we will
cut??
            // if (color < c + 1 && depth + cache[color - 1]
<= maxCliqueSize){ depth--; }
            int initialColorThresholdIndex = -1;
            for (int i = vertIndex - 1; i >= 0; i--)
            {
```

```
                int vert2 = levelNodes[depth, i];
                if (initialColorThresholdIndex == -1)
                {
                    int color = initialColors[vert2 - 1];
                    if (color < currentColor && cache[color -
1] <= threshold)
                    {
                        initialColorThresholdIndex = i;
                        break;
                    }
                }
            }

            // if we want to skip a vertex, we have to check
if current vertex
            // is adjacent to any vertex INSIDE vertices that
will be CUT!! (from initialColorThresholdIndex to zero) on
this depth
            // with color higher than threshold
            for (int i = initialColorThresholdIndex; i >= 0;
i--)
            {
                int vert2 = levelNodes[depth, i];
                if (Graph.Values[vert - 1, vert2 - 1])
                {
                    if (inDepthColors[depth, vert2 - 1] >
threshold)
                        return false;
                }
            }
            return true;
        }

        public int FindNumberOfColorClasses(int depth, int
numberOfNodes)
        {
            int nPrevColor = 0;

            int numberOfColorClasses = 0;
            for (int i = 0; i < numberOfNodes; i++)
            {
                int currentColor =
initialColors[levelNodes[depth, i] - 1];
                if (currentColor != nPrevColor)
                {
                    numberOfColorClasses++; nPrevColor =
currentColor;
                }
            }
```

```
            return numberOfColorClasses;
        }
        private void InitialColoring()
        {
            var verticesWithDegrees = new int[NodesNumber][];
            // count degrees of vertices

            for (int i = 0; i < NodesNumber; i++)
            {
                verticesWithDegrees[i] = new int[2];
                verticesWithDegrees[i][0] = i + 1;
            }

            for (int i = 0; i < NodesNumber; i++)
                for (int j = i + 1; j < NodesNumber; j++)
                    if (Graph.Values[i, j])
                    {
                        verticesWithDegrees[i][1]++;
                        verticesWithDegrees[j][1]++;
                    }

            // order vertices by degree
            var orderedVertices =
verticesWithDegrees.OrderByDescending(i => i[1]).ToArray();

            // color vertices, find color classes
            for (int i = 0; i < NodesNumber; i++)
            {
                int vert = orderedVertices[i][0];
                bool isAdded = false;
                for (int j = 0; j < initialColorsNumber; j++)
                {
                    bool connected = false;
                    for (int k = 0; k <
initialNodesNumInColorClass[j]; k++)
                    {
                        if (Graph.Values[vert - 1,
initialColorClasses[j][k] - 1])
                        {
                            connected = true;
                            break;
                        }
                    }
                    if (!connected)
                    {

initialColorClasses[j][initialNodesNumInColorClass[j]] = vert;
                        initialColors[vert - 1] = j + 1;
                        inDepthColors[0, vert - 1] = j + 1;
```

```
                        initialNodesNumInColorClass[j]++;
                        isAdded = true;
                        break;
                    }
                }
                if (!isAdded)
                {
                    initialColorClasses[initialColorsNumber] =
new int[NodesNumber];

initialColorClasses[initialColorsNumber][initialNodesNumInColo
rClass[initialColorsNumber]] = vert;

initialNodesNumInColorClass[initialColorsNumber]++;
                    initialColorsNumber++;
                    initialColors[vert - 1] =
initialColorsNumber;
                    inDepthColors[0, vert - 1] =
initialColorsNumber;
                }
            }
        }

        private void InitialColoringWithSwaps()
        {
            var array = new int[NodesNumber];
            for (int i = 0; i < NodesNumber; i++)
                array[i] = i + 1;

            int colored = 0;
            initialColorsNumber = 0;

            while (true)
            {
                initialColorClasses[initialColorsNumber] = new
int[NodesNumber];

initialColorClasses[initialColorsNumber][initialNodesNumInColo
rClass[initialColorsNumber]] = array[colored];

initialNodesNumInColorClass[initialColorsNumber]++;
                initialColorsNumber++;
                initialColors[array[colored] - 1] =
initialColorsNumber;
                inDepthColors[0, array[colored] - 1] =
initialColorsNumber;

                colored++;
                int lowerBound = colored - 1;
```

```
                for (int i = colored; i < NodesNumber; i++)
                {
                    bool canBeColored = true;
                    for (int j = lowerBound; j < colored; j++)
                        if (Graph.Values[array[i] - 1,
array[j] - 1])
                        {
                            canBeColored = false;
                            break;
                        }
                    if (canBeColored)
                    {
                        if (i != colored)
                        {
                            var node = array[i];
                            array[i] = array[colored];
                            array[colored] = node;
                        }
                        inDepthColors[0, array[colored] - 1] =
initialColorsNumber;
                        initialColors[array[colored] - 1] =
initialColorsNumber;

initialColorClasses[initialColorsNumber -
1][initialNodesNumInColorClass[initialColorsNumber - 1]] =
array[colored];

initialNodesNumInColorClass[initialColorsNumber - 1]++;
                        colored++;
                    }
                }
                if (colored == NodesNumber)
                    break;
            }
        }

        private int Recolor(int depth)
        {
            int colorsNumber = 0;
            var colorClasses = new int[NodesNumber][];
            var nodesNumInColorClass = new int[NodesNumber];
            skippedNodesNumber[depth] = 0;
            // color vertices, find color classes
            for (int i = 0; i < numberOfNodesArr[depth]; i++)
            {
                int vert = levelNodes[depth, i];
                bool isAdded = false;

                for (int j = 0; j < colorsNumber; j++)
```

```
                {
                    bool connected = false;
                    for (int k = 0; k <
nodesNumInColorClass[j]; k++)
                    {
                        if (Graph.Values[vert - 1,
colorClasses[j][k] - 1])
                        {
                            connected = true;
                            break;
                        }
                    }
                    if (!connected)
                    {

colorClasses[j][nodesNumInColorClass[j]] = vert;
                        inDepthColors[depth, vert - 1] = j +
1;
                        nodesNumInColorClass[j]++;
                        isAdded = true;
                        break;
                    }
                }
                if (!isAdded)
                {
                    colorClasses[colorsNumber] = new
int[NodesNumber];

colorClasses[colorsNumber][nodesNumInColorClass[colorsNumber]]
= vert;
                    nodesNumInColorClass[colorsNumber]++;
                    colorsNumber++;
                    inDepthColors[depth, vert - 1] =
colorsNumber;
                }
            }
            return colorsNumber;
        }

        private int RecolorWithSwaps(int depth)
        {
            int colorsNumber = 0;
            int length = numberOfNodesArr[depth];
            skippedNodesNumber[depth] = 0;
            int colored = 0;
            var array = new int[length];
            for (int i = 0; i < length; i++)
            {
                array[i] = levelNodes[depth, i];
```

```
            }
            while (true)
            {
                colorsNumber++;
                inDepthColors[depth, array[colored] - 1] =
colorsNumber;
                colored++;
                int lowerBound = colored - 1;
                for (int i = colored; i < length; i++)
                {
                    bool canBeColored = true;
                    for (int j = lowerBound; j < colored; j++)
                        if (Graph.Values[array[i] - 1,
array[j] - 1])
                        {
                            canBeColored = false;
                            break;
                        }
                    if (canBeColored)
                    {
                        if (i != colored)
                        {
                            var node = array[i];
                            array[i] = array[colored];
                            array[colored] = node;
                        }
                        inDepthColors[depth, array[colored] -
1] = colorsNumber;
                        colored++;
                    }
                }
                if (colored == length)
                    break;
            }
            return colorsNumber;
        }
    }
}
```