TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Anastassia Duditš 123685IAPB

MULTITHREADED WEB APPLICATION FOR THE MAXIMUM CLIQUE PROBLEM

Bachelor's thesis

Supervisor: Deniss Kumlander
Doctor's degree

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Anastassia Duditš

MITMELÕIMELINE VEEBIRAKENDUS SUURIMA KLIKI PROBLEEMILE

Bakalaureusetöö

Juhendaja: Deniss Kumlander
Doktorikraad

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Anastassia Duditš

08.01.2020

Abstract

The maximum clique problem is one of the known NP-complete problems. Although there is a number of solutions to this problem, which are both relatively cost efficient in terms of computing power as well as execution time, scientists continue to look for better ways to solve this problem since even small improvement gives huge benefits for NP-complete family tasks.

The IT company Denikum OÜ, which develops different optimization algorithms was interested in porting the efficient maximum clique finding algorithm to web. The main goal is to allow the user to run the algorithm smoothly in the browser on a modern PC, without the need to load the server with heavy and time-consuming operations. It was decided to port the VRecolor-BT-u algorithm because it is currently one of the best algorithms to solve the maximum clique problem.

The thesis starts with a brief description of graph theory as well as the main goals of this work. After that VRecolor-BT-u algorithm and its predecessors are presented. Next, the key points of development process are shown along with the description of the technology stack.

The main contribution of this thesis is the parallelization of VRecolor-BT-u algorithm using Web Workers, which greatly improved the performance of the algorithm on graph densities of 30% and more. The second big goal that was achieved in this work was to combine the parallelized VRecolor-BT-u-parallel algorithm with VRecolor-u algorithm, which improved the algorithm performance on lower density graphs, making it efficient on all graph densities.

At last the newly improved algorithms are compared with their predecessors on both randomly generated as well as DIMACS graphs. The results are presented in a series of charts, which display the relation of the number of vertices to the time taken for different algorithms to find the result. The results show that the combined algorithm performs better than all previously created algorithms on all densities on randomly

generated graphs. The paper is concluded by some of the interesting ideas for future research.

This thesis is written in English and is 53 pages long, including 4 chapters, 31 figures and 2 tables.

Annotatsioon

Mitmelõimeline veebirakendus suurima kliki probleemile

Maksimaalse kliki probleem on üks teadaolevatest NP-täielikkuse probleemidest. Ehkki sellele probleemile on mitmeid lahendusi, mis on nii arvutusvõimsuse kui ka täitmisaja poolest suhteliselt kulutõhusad, otsivad teadlased endiselt selle probleemi lahendamiseks paremaid viise, sest isegi väike täiustamine annab NP-täielikkuse perekondlike ülesannete puhul suured eelised.

Mitmesuguseid optimeerimisalgoritme arendav IT-ettevõte Denikum OÜ oli huvitatud tõhusa maksimaalse kliki leidmise algoritmi veebi porteerimisest. Peamine eesmärk on võimaldada kasutajal tänapäeva arvuti brauseris algoritmi sujuvalt kasutada, ilma vajaduseta serverit laadida keerukate ja aeganõudvate toimingutega. Otsustati porteerida VRecolor-BT-u algoritm, sest see on praegu üks parimatest algoritmidest maksimaalse kliki probleemi lahendamiseks.

Lõputöö algab graafiteooria lühikirjeldusega, samuti selle töö peamiste eesmärkidega. Seejärel tutvustatakse VRecolor-BT-u algoritmi ja selle eelkäijaid. Järgmisena näidatakse arendusprotsessi põhipunkte koos tehnoloogiapinu kirjeldusega.

Selle lõputöö peamine panus on VRecolor-BT-u algoritmi paralleelistamine veebitöötajate abil, mis täiustas oluliselt algoritmi tulemuslikkust graafi tihedusel 30% ja enam. Teine selles töös saavutatud suur eesmärk oli paralleelse VRecolor-BT-u-parallel algoritmi ühendamine VRecolor-u algoritmiga, mis täiustas algoritmi tulemuslikkust väiksema tihedusega graafidel, muutes selle tõhusaks kõigil graafi tihedustel.

Lõpuks võrreldakse hiljuti täiustatud algoritme nende eelkäijatega nii juhuslikult genereeritud kui ka DIMACS-graafide abil. Tulemused on esitatud diagrammisarjana, mis näitavad tippude arvu seost ajaga, mis kulub eri algoritmidele tulemuse leidmiseks. Tulemused näitavad, et kombineeritud algoritm genereeritud graafide puhul toimib

kõigil tihedustel paremini kui kõik varem loodud algoritmid. Töö lõpus on mõned huvitavad ideed edaspidiseks uurimistööks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 53 leheküljel, 4 peatükki, 31 joonist, 2 tabelit.

List of abbreviations and terms

AJAX	*Asynchronous JavaScript and XML* – a set of web development techniques used on the client side to create asynchronous web applications
Angular	Platform for building mobile and desktop web applications
API	*Application Programming Interface* – a set of definitions and protocols for building and integration application software
CPU	*Central Processing Unit* – the unit which performs most of the processing inside a computer
DIMACS	*Center for Discrete Mathematics and Theoretical Computer Science*
DOM	*Document Object Model* – object model for HTML
FIFO	*First-In, First-Out* – a method of processing and retrieving data
HTML5	*HyperText Markup Language, version 5* – markup language for the structure and presentation of World Wide Web contents
JavaScript	The programming language for the Web
NP-complete	The problem can be solved in *Polynomial* time using a *Non-deterministic* Turing machine
TypeScript	A typed superset of JavaScript that compiles to plain JavaScript
UI	*User Interface*
VColor-BT-u	*Vertex Color Backtrack unweighted* – algorithm name
VColor-u	*Vertex Color unweighted* – algorithm name
VRecolor-BT-u	*Vertex Recolor Backtrack unweighted* – algorithm name
VRecolor-BT-u-parallel	*Vertex Recolor backtrack unweighted parallel* – algorithm name
VRecolor-u	*Vertex Recolor unweighted* – algorithm name
Web Workers	A JavaScript that runs in the background, independently of other scripts

Table of contents

1 Introduction 13
1.1 Graph theory 13
1.2 Goals of the study 16
1.3 Work overview 16
2 Maximum clique algorithms 17
2.1 Basic algorithms 17
2.2 VColor-u 18
2.3 VColor-BT-u 18
2.4 VRecolor-BT-u 19
3 Implementation 22
3.1 Technology stack 22
3.2 Porting VRecolor-BT-u to web 23
3.3 Multithreading 23
3.4 Multithreaded VRecolor-BT-u 26
3.4.1 Results 30
3.5 Multithreaded VRecolor-BT-u and VRecolor-u in parallel 39
3.5.1 Results 43
4 Conclusion 50
4.1 Summary 50
4.2 Future studies 51
References 53

List of figures

Figure 1. Degrees of the vertices. 14
Figure 2. Directed weighted graph and undirected unweighted graph. 14
Figure 3. Complete graph. 15
Figure 4. VRecolor-BT-u algorithm [4]. 21
Figure 5. Data preparation in the main thread. 27
Figure 6. The process of building the first level and isolating jobs. 28
Figure 7. Initialization of the limited number of workers. 29
Figure 8. Workers initialization. 29
Figure 9. Random graph generation function. 31
Figure 10. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 10%. 32
Figure 11. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 20%. 32
Figure 12. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 30%. 33
Figure 13. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 40%. 34
Figure 14. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 50%. 34
Figure 15. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 60%. 35
Figure 16. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 70%. 35
Figure 17. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 80%. 36
Figure 18. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 90%. 36
Figure 19. Use of SharedArrayBuffer object for the shared data. 40
Figure 20. Stop condition check in VRecolor-BT-u-parallel worker. 41

Figure 21. Stop condition check in VRecolor-u worker. 42
Figure 22. Handler for “worker done” event. 43
Figure 23. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 10%. 44
Figure 24. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 20%. 44
Figure 25. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 30%. 45
Figure 26. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 40%. 45
Figure 27. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 50%. 46
Figure 28. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 60% 46
Figure 29. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 70%. 47
Figure 30. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 80%. 47
Figure 31. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 90%. 48

List of tables

Table 1. VRecolor-BT-u and VRecolor-BT-u-parallel DIMACS graph test results (ms). 38
Table 2. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel DIMACS graph tests (ms). 49

1 Introduction

The graph theory became known for the first time after the publication of the paper written in 1736 by L. Euler on the Seven Bridges of Königsberg problem. The graph theory solved an entertaining problem, which was presented there.

Graph theory is a section of discrete mathematics that studies the graphs, which are structures used to represent relationships between objects. Graphs got their name from the fact that they can be depicted graphically. Anything that looks like the interrelated components can be represented as a graph, where components are vertices and relations between them are edges. In practice, graphs are used to solve problems of different complexity from various fields of science and real life, as well as just for representing complex things in the form of graphs. The great advantage of graphs usage is the possibility to simplify problems by omitting the irrelevant details, thereby concentrating only on the core details. Graph theory is widely used in social networks, information networks, GPS navigation, planning the transportation routes, networks of neurons and so on. Graphs are also successfully used to resolve complete and NP-complete decision tasks. Some of the most known NP-complete problems are graph coloring and maximum clique finding [1].

1.1 Graph theory

A graph G is a collection of objects, i.e. vertices V, and edges E that represents relationships between these objects. A number of vertices in G is called the order of G and denoted by $|V|$, and the number of edges is the size of G, denoted by $|E|$ [2].

Two vertices v and w of G are called *adjacent* if there is an edge vw joining them, and both vertices are *incident* with edge vw. If two distinct edges have a vertex in common, they are also called adjacent. *Loop* is an edge connecting vertex v to itself [3].

The *degree* of a vertex v of G is the number of edges incident with v – $deg(v)$ [3].

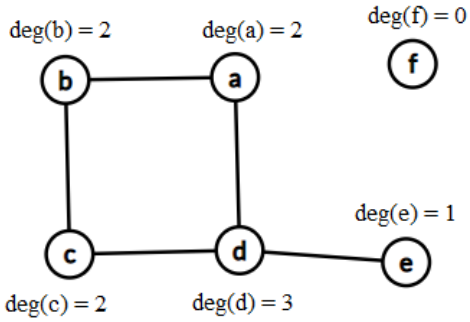

Figure 1. Degrees of the vertices.

A *directed graph* is a graph in which every edge is directed, and a graph in which every edge is undirected is called an *undirected graph*. Graphs can also be *weighted* and *unweighted*. Each edge in a weighted graph has a weight – a number (usually positive) assigned to it, respectively, edges of an unweighted graph do not have numerically represented weights.

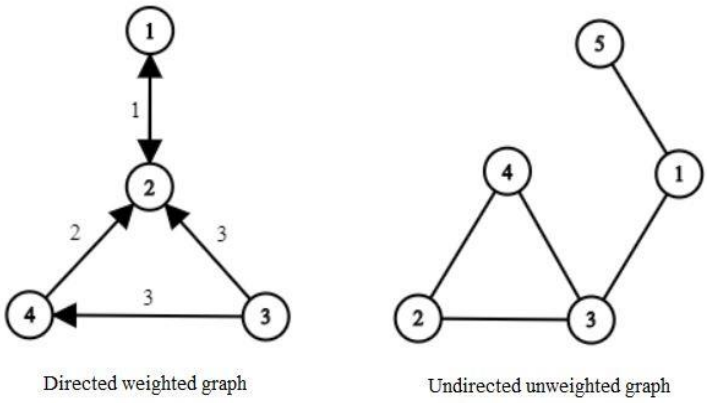

Figure 2. Directed weighted graph and undirected unweighted graph.

Graph is called *simple* when it is undirected, without loops and multiple edges between two vertices. A simple graph where each pair of vertices is adjacent is a *complete graph*. A graph with no adjacent pairs of vertices is called *edgeless* [3]. A *clique* is a complete graph of G, whereas an edgeless subgraph of *G* is an *independent set* [2]. This work will consider only undirected, unweighted and simple graphs.

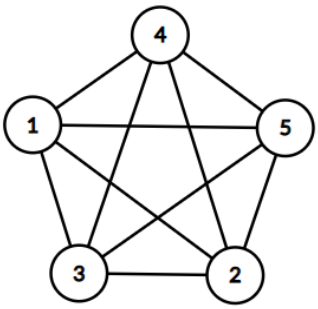

Figure 3. Complete graph.

Complement graph G' of a simple graph *G* is a graph with vertex set *V(G)* where two vertices are adjacent if they are not adjacent in *G*. Accordingly, edgeless graph is a complement to complete graph [3].

A vertex set *K* is called *vertex cover* of a graph *G*, when each edge of *G* is incident to at least one vertex from *K* [2].

Graph coloring is a process of assigning colors to each vertex so that adjacent vertices have different colors. A graph without loops is said to be *k-colorable*, if it can be properly colored using *k* colors. The *chromatic number* of *G*, write $\chi(G) = k$, is the minimum number of colors required to color a graph *G* [3]. A set of verices with the same color is a *color class*. Since the vertices with the same color are not connected to each other, the color class is nothing more than an independent set.

The following problems are stated from the definitions mentioned above [1]:

- 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 finding a chromatic number of a graph *G*.

All these problems are NP-complete, moreover they are computationally equivalent and one problem can be transformed into another one [4].

1.2 Goals of the study

1. Port VRecolor-BT-u algorithm to the web environment.

2. Improve VRecolor-BT-u by applying multithreading techniques.

3. Implement algorithm that executes VRecolor-BT-u-parallel algorithm and VRecolor-u from different ends of the graph in parallel threads.

4. Compare execution performance of VRecolor-BT-u, VRecolor-BT-u-parallel and combination of VRecolor-BT-u-parallel and VRecolor-u.

During the work, in order to achieve the stated goals, we will have to do also:

1. Study one of the best maximum clique finding algorithm – VRecolor-BT-u and related.

2. Study Web Workers that make it possible to implement a multithreaded application.

1.3 Work overview

Current work consists of four main chapters. In the beginning of the first chapter graph theory is introduced as well as some of the main problems, which can be solved with it. The goals of the thesis are presented afterwards. The second chapter starts with the presentation of the two main maximum clique finding algorithms, which are the base for many modern algorithms. Afterwards a few modern algorithms are presented, namely VColor-u, VColor-BT-u and VRecolor-BT-u. The third chapter begins with the description of the chosen technology stack and porting algorithm to web. The next topic is dedicated to multithreading and its usage in the web environment. The last two sections of the third chapter give an overview of two approaches to improve performance of the VRecolor-BT-u algorithm by applying multithreading, the description of the development process and the analysis of the test results, carried on the random and DIMACS graphs. The last chapter includes the conclusion of the thesis as well as the ideas for future research.

2 Maximum clique algorithms

The maximum clique problem is one of the known NP-complete problems. The goal of this problem is to find the maximum possible complete subgraph in graph *G*. This problem is considered as NP-complete due to the difficulty of finding the best result with usual methods. The first exact solutions with good computational performance were presented in the ’90s, however, the maximum clique problem still remains popular among scientists. Nowadays there is a lot of interesting and operative solutions that are already implemented. Nevertheless, scientists continue to look for better ways to solve this problem [1].

This chapter presents two basic branch and bound algorithms for finding the maximum clique by applying the different approaches of traversing the graph. Three modern algorithms, which are based upon the upper mentioned, are described after that.

2.1 Basic algorithms

The basic concepts of how a clique can be found were given in an algorithm invented by Randy Carraghan and Panos M. Pardalos. It was presented in the article “An exact algorithm for the maximum clique problem” published in 1990 [5]. The algorithm itself is simple but very efficient, moreover, even nowadays it gives great results on lower density graphs. It is based on a branch construction and a good pruning formula. The main point of the algorithm is the notion of depth. At first, the first vertex from all vertices of graph S – vertex v_1 (depth 1) is considered. The next depth will be formed from vertex $v1$ and all vertices adjacent to v_1. At depth 3 all vertices from depth 2, which are adjacent to the first vertex in current depth v_3 are considered, and so on. In this way, every expanded vertex forms a new branch (until there are no vertices left to expand) and the reached depth considers as a current maximum clique. The pruning formula is carried out on each branch, and in case it does not hold, the branch will be truncated. Consequently, the analyzed graph will be covered much faster. However, this formula is very efficient on low-density graphs, but on high densities it is almost useless

[5]. In short, the approach used in Carraghan and Pardalos algorithm considers all vertices at the start and traverses the graph by building the branches.

The second efficient approach was given by Östergård's algorithm, introduced in the article "A fast algorithm for the maximum clique problem" [6]. This algorithm is based on the previous one with important addition as a backtrack search and extra pruning formulas. The previous algorithm searches for the maximum clique by first considering cliques in S_1 that contain v_1, then cliques in S_2 that contain v_2, and so on. In this algorithm the ordering is reversed: at first cliques in S_n that contain v_n, are considered, then cliques in S_{n-1} that contain v_{n-1} [6]. The obtained clique size for each subgraph S_i are stored in a cache that is later used to apply the new pruning formula. The performance of this approach does not differ much from the previous one on low densities, but on high densities it is about 30%-50% faster. Furthermore, if the density is very close to one, the speed of the algorithm is increased even more [6].

2.2 VColor-u

In 2005 Deniss Kumlander introduced VColor-u algorithm in his thesis "Some practical algorithms to solve the maximum clique problem" [7]. This algorithm is based on Carraghan and Pardalos approach and the idea of using independent sets by performing initial vertex coloring. Before the maximum clique is being searched for, the graph is analyzed and results gained from the analysis are stored for later use. Compared to the two previous branch and bound algorithms VColor-u demonstrates better results: about 15% faster on low densities (20% – 50%) and up to 50% faster on density 90%. Nevertheless, VColor-u is slower than Carraghan and Pardalos algorithm on densities about 10% and lower, since graph coloring and vertices ordering are useless on low densities and extra pruning formula is not that efficient [4].

2.3 VColor-BT-u

In the same dissertation another algorithm VColor-BT-u was also introduced. This algorithm differs from the previous one by the traversing approach – VColor-BT-u is based on powerful backtracking idea from Östergård algorithm [6]. At the start of the program initial vertex coloring is applied and then the graph is backtracked on a higher level than Östergård algorithm: by independent sets instead of individual vertices [7].

The computational speed of VColor-BT-u is approximately two times higher than VColor-u on almost all densities. The combination of pruning formula from backtracking idea and pruning technique based on the usage of independent sets made this new algorithm also faster than Östergård's algorithm: 50%-100% on lower densities and 13-25% on dense graphs [4].

2.4 VRecolor-BT-u

In his master's thesis "Reversed search maximum clique algorithm based on recoloring" [4] Aleksandr Porošin introduced a new algorithm to solve maximum clique problem, which was also presented on the 6th World Congress on Global Optimization (WCGO 2019) this year. This algorithm is a successor of VColor-BT-u algorithm with additional recoloring on each depth – idea inherited from the MCQ algorithm (firstly introduced in 2003 by Tomita and Seki and its' successors [8]). Algorithms from Tomita proved that initial coloring is not enough as the deeper search goes the more diffused initial color classes become [4]. On high levels, the recoloring is needed to obtain precise information about independent sets on current depth. Considering that reversed search is built around initial color classes, efficient pruning formulas on recoloring cannot be applied. The solution to this conflict was to use a new skipping technique instead of pruning the branches immediately basing on recoloring.

VRecolor-BT-u is described using the following steps (Figure 4):

Algorithm for the maximum clique problem – "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 possible 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.

Figure 4. VRecolor-BT-u algorithm [4].

Compared to VColor-BT-u and “MCS Improved” (the last successor of MCQ algorithm [9]) VRecolor-BT-u algorithm shows the best results on densities until 75%. On highly dense graphs the “MCS Improved” is still the fastest one [4].

3 Implementation

According to the nature of the problem and the technology stack of the company, for whom the multithreaded algorithm is being developed, it was decided to create the algorithm using following technologies: Angular 8, TypeScript, Web Workers. Next, above-mentioned technologies will be examined in more detail and their use will be justified. Finally, the practical part of this work will be presented – developed solutions and overview of the results.

3.1 Technology stack

The company Denikum OÜ has a strong request to implement solution for the maximum clique problem web based, but without the backend part, since algorithms developed in this work is going to be applied within the cloud-based products and many of them do require optimization of the cost by minimizing the server-side load. Additionally, the company would like to add the algorithms to be developed into the demo package and again likes to avoid additional charges related to the server load.

Since the customer company uses Angular framework for creating web applications, it was requested to develop our solution with Angular (version 8.2.1) as well. Angular is a JavaScript framework developed by Google, and currently is one of the best solutions for web development. With great features like templating, two-way binding, hierarchical dependency injection, AJAX handling and so on, Angular makes it possible to build modern, interactive and dynamic web pages and applications. With Angular the applications can be built for any deployment target: for web, mobile web, native mobile and native desktop [10]. The proper use of all the features of this framework does not just significantly increases the speed of development, but also aids in creating quality and efficient software. Other advantage of Angular framework is, that it’s TypeScript-based. TypeScript is a superset of JavaScript, which ensures higher security by supporting types. Typed languages help us catch and fix errors early in the process of writing the code or performing maintenance tasks. Furthermore, the TypeScript code can be directly debugged in the browser, which also simplifies the development process.

Web Workers are used to execute tasks in separate threads. This mechanism solves the concurrency problem in JavaScript. With the help of Web Workers it is possible to execute scripts in background threads, separately from the main thread, which prevents it from being blocked. It will be discussed in more detail in the upcoming chapter.

3.2 Porting VRecolor-BT-u to web

The VRecolor-BT-u algorithm was first introduced in Aleksandr Poroshins' master's thesis, and was implemented with C# programming language. One of the goals of this work is to port VRecolor-BT-u to web. It's worth to note that the structure and work principle of the algorithm will remain the same, since the algorithms used in the thesis have one very important nature: those were written one by one inheriting each consequent implementation from the previous one as the algorithm evolved through several improvement rounds. Therefore, every possible improvement in base algorithm is applicable to the next algorithm and therefore, on the high level, even if there are small improvements possible, it will not change the compatibility of the algorithm, which is the most important point of this work. It is also worth mentioning that the algorithms were carefully checked through all rounds by all involved authors [Kumlander, Porosin] including results comparison and the probability of finding large scale improvements is low and therefore can be omitted due the fact we do compare algorithms.

3.3 Multithreading

As already noted, VRecolor-BT-u algorithm is going to be improved with the use of multithreading. Essentially, JavaScript by its nature is single threaded environment – it executes only on a single processor thread. This means that every executed line of code blocks further code execution i.e. only one command is processed at a time. This is not a problem for the majority of websites, because it is possible to create pretty fast web apps without the use of multithreading. Sometimes, however, we encounter more complicated computations or time-consuming procedures, which might require the client to wait for too long. Computational operations that can be executed independently from other logic potentially is executable in a separate thread, thus leaving the main

thread as well as UI unblocked. In order to solve this problem with blocking, the asynchronous features were introduced in JavaScript, but even using those the code is still executed in the same thread. Therefore, we still need a true multithreading features for our implementation.

As it was mentioned earlier, parallel processing in web development is achieved with the use of Web Workers, which were introduced in HTML5. It is possible to transfer execution of some independent scripts to the background threads with the aid of so-called workers. Worker is an object, which is created using the *Worker()* constructor, which runs a separate JavaScript file with the code, executed in an isolated browser thread. In order to make use of the worker thread, the scopes of worker thread and main thread need to be able to communicate. It is done by using a messaging system. A worker subscribes to a message. When the message is received, the worker processes it and optionally sends another message back to the main thread. The main thread also has to subscribe to the message event if it needs to react on it.

There are certain specifics and restrictions making web workers challenging to apply. The success of the work at whole depends on the proper choice of solutions to these limitations.

First of all, workers have some difficulties with data transfer. The data passed to the worker directly will be shallow cloned, leading to the loss of some data. In the interest of preserving the data, it should be serialized and cloned with the use of the structured cloning algorithm or transferable objects. Structured cloning means that all of the properties of an object must be iterated through and the values of those properties duplicated. In short, this process is called deep cloning. Transferable objects are objects that implement the *Transferable* interface [11] and can be moved to a different JavaScript context (i.e. another window or worker). Although the content of transferable objects is transferred from one context to another with a zero-copy operation, it is literally moved to the new context. For example, when passing an *ArrayBuffer* (which belongs to transferable objects) from the main thread to a worker, the original *ArrayBuffer* is cleared and will no longer be available in the original context [12].

It can be noticed that both approaches have downsides. The deep cloning and the transferring of the data is significant overhead to the message transmission, since the

larger amount of data is, the bigger the message gets, and the longer it takes to send it. The drawback of using the transferable object is the loss of the object from the original thread. In order to avoid these disadvantages, which are unprofitable for the performance, it is possible to use *SharedArrayBuffer* for sharing the data between threads - it is done without any loss of data in the parent thread and without time and memory consuming cloning. As it was written in one of the source materials: "The *SharedArrayBuffer* object is used to represent a generic, fixed-length raw binary data buffer, similar to the *ArrayBuffer* object, but in a way that they can be used to create views on shared memory. Unlike an *ArrayBuffer*, a *SharedArrayBuffer* cannot become detached" [13]. Shared memory can be created and updated simultaneously in multiple threads. Atomic operations need to be used to synchronize those modifications and to be sure that shared data in any of the threads are up to date. Those guarantee that read and write operations will always be completed before the next operation is executed. The *Atomics* object embedded into JavaScript represents atomic operations in form of static functions, which are used together with the *SharedArrayBuffer* object.

Another aspect that needs to be accounted for is the fact that productivity might suffer from the creation of workers and their excessive amount. The initialization of one worker takes about 40ms [14], which means that improper use of workers might decrease the overall productivity due to their initialization. Furthermore, it is worth considering the number of cores (the basic computation units of the CPU) the system where the algorithm will be run has. If there are more workers spawned than there are cores, it will slow down the workload, as the workers will be queued. Due to this fact, we need to use a reasonable number of workers in order to get the maximum benefit from them.

Multithreading is used to improve initial algorithms. In order to get the most from the use of workers it is crucial to understand the structures of the algorithms to find the proper place for workers to be added into. The first idea is to find out a recurring part in the algorithm, which could be processed as an independent task, and run those parts simultaneously in the separate threads. It is also important to set up the parallel task execution system and try to minimize the necessary performance costs to run workers.

The second idea is to run two maximum clique algorithms simultaneously, which will traverse the graph from different ends. To make this idea work, both algorithms have to

be based on the same approach of the initial vertex coloring and ordering. After execution, those algorithms will reach the same vertex and after analyzing that one the further traversing of the graph will be useless. Therefore, the execution of the algorithms should be interrupted, and the largest clique is picked from the maximum cliques found in both algorithms. The challenge in this approach is to properly set up communication between threads, by use of shared data. It is crucial since two algorithms need to share the information about their current positions to determine the moment of termination.

3.4 Multithreaded VRecolor-BT-u

The branch and bound algorithms' computational speed can be significantly increased by parallelizing the process of the traversing branches. When we are on the first level (depth 1), no matter if the algorithm is using the reversed search or not, each expanded vertex forms a separate branch. VRecolor-BT-u algorithm is based on backtrack search and operates with independent sets divided by the color classes instead of single vertices. It starts searching for a clique from the vertices of the first color class, increasing the analyzed graph by one independent set (step 3 from Figure 4) on backtracking iteration. At first glance, it may seem logical to parallelize the processes by processing in separate threads each subgraph formed for each backtracking iteration. Unfortunately, certain pruning formulas are based on backtracking, i.e. are using the information gained from previously traversed subgraphs. It means, that these iterations cannot be multithreaded, as they depend on each other. If we look deeper at the logic that is performed inside the iterations, it will be found, that the processing of the subgraphs is nothing other than the simple branch and bound algorithm, which is based on Carraghan and Pardalos approach. In the mentioned approach all the vertices are considered one after another. The analyzed graph is decreased on each iteration by removing the processed vertex from it. This particular place is excellent for parallelizing the process: being at the first level, where we have all the vertices from the current set of independent sets, it is possible to process branches formed from each first level vertex independently.

The code, which is going to be executed in parallel threads, is located in separate workers' file *v-recolor-bt-u-parallel.worker.ts*. All the data, related to the graph and

analysis, is prepared before the processing by the worker: graph, initial colors, current maximum clique size, cache and others needed in these workers. As some of the data can be quite massive, the copying and the passing operations may take too long. In order to decrease the time spent to minimum, we are creating a shared memory area for this data with the use of *SharedArrayBuffer*, and then send that memory to workers. In workers, as well as in the main thread, we create an array-like view on top of that shared memory (binary data buffer), to work with shared data as with a simple array.

As seen in Figure 5, the data, which will be later sent to workers, is prepared before the search for the maximum clique is started. Static functions of *Atomics* object are used to synchronize the read/write operations to the shared data. There are the following steps inside the backtrack search for-loop:

1. Wait until the independent sets of the current color classes is processed.

2. Save the current maximum clique size to cache.

```
this._sharedBufferMaxCliqueSize = new SharedArrayBuffer(2);
let sharedMaxCliqueSize = new Uint16Array(this._sharedBufferMaxCliqueSize);
Atomics.store(sharedMaxCliqueSize, 0, 0);

this._sharedBufferJsonGraphValues = SharedArrayBufferUtil.str2sharedArrayBuff
er(JSON.stringify(this._graph.values));

this._sharedBufferInitialColors = new SharedArrayBuffer(this._initialColors.l
ength * 2);
let sharedInitialColors = new Uint16Array(this._sharedBufferInitialColors);
sharedInitialColors.set(this._initialColors, 0);

this._sharedBufferMaxCliqueCache = new SharedArrayBuffer(initialColorsNumber
* 2);
let sharedMaxCliqueCache = new Uint16Array(this._sharedBufferMaxCliqueCache);

for (let initialColor = 0; initialColor < initialColorsNumber; initialColor++
) {
    await this.analyseColorSetAsync(initialColor);
    let currentMaxClique = Atomics.load(sharedMaxCliqueSize, 0);
    Atomics.store(sharedMaxCliqueCache, initialColor, currentMaxClique)
}
```

Figure 5. Data preparation in the main thread.

In the first step we are waiting for all the threads spawned for the subgraph processing to finish their work. After that, the gained maximum clique is saved to cache and the

next iteration is activated. The obtained result from the previous analysis will be used in subsequent calculations.

```
private async processColorSetAsync(initialColor: number): Promise<void> {
    return new Promise(async (res) => {
        let numberOfNodes = 0;
        let depthNodes = [];
        for (let i = 0; i <= initialColor; i++) {
            for (let j = 0; j < this._initialNodesNumInColorClass[i]; j++) {
                depthNodes[numberOfNodes] = this._initialColorClasses[i][j];
                numberOfNodes++;
            }
        }

        for (let i = numberOfNodes - 1; i >= 0; i--) {
            this._jobsQueue.push({
                startNodeIndex: i,
                firstLevelNodes: depthNodes,
                initialColor: initialColor,
            })
        }

        await this.startJobs(res);
    });
}
```

Figure 6. The process of building the first level and isolating jobs.

The process of building the first level and isolating jobs is shown on the Figure 6. We start from building the first level, afterwards a separate job, containing necessary information for the branch processing, is created for each vertex on this depth. The jobs list represents the queue which follows the FIFO approach, so the first job from the list will be processed first and the new job will be placed in the end of the list and executed last. After the jobs for each branch are created, we need to initialize the worker threads, where the jobs will be executed. As already mentioned before, each worker initialization takes about 40ms, besides the time for transferring data to the worker also adds up. Since we are interested in the maximum profit of using the workers, it would not be rational to create a new worker each time the job is started and remove it after the job is finished. It definitely will impact performance negatively, so it was decided to create a fixed number of workers, depending on system CPU cores number, and make them reusable. The number of CPU cores can be obtained from the *window.navigator.hardwareConcurrency* object. Next, when some job is finished, the worker is released and may be re-used by the next job. Figure 7 shows that the number

of required workers is set in accordance with the limit based on available cores and the number of jobs to be processed, as it doesn't make sense to spawn more threads than necessary at the moment. If there will be more jobs in future iterations, and the limit will not be reached, additional workers will be created.

```
const jobsToStart = Math.min(scope.workersLimit, scope.jobsQueue.length);
let missingWorkers = jobsToStart - scope.workers.length;
if (missingWorkers)
    await initWorkers(missingWorkers);
```

Figure 7. Initialization of the limited number of workers.

Figure 8 demonstrates the initialization of the workers.

```
async function initWorkers(amount: number): Promise<void> {
    return new Promise((res, rej) => {
        scope.workersToInit = amount;

        const idStartPoint = scope.workers.length;
        for (let i = 0; i < amount; i++) {
            const worker = new Worker('../v-recolor-bt-u-parallel/v-recolor-
bt-u-parallel.worker', { type: 'module' });

            worker.onmessage = (event: MessageEvent) => {
                scope.workersToInit--;
                if (!scope.workersToInit)
                    res();
            };

            worker.onerror = () => {
                console.log("ERR");
            };

            worker.postMessage({
                workerId: idStartPoint + i,
                graphOrder: scope.graphOrder,
                graphDensity: scope.graphDensity,
                sharedBufferMaxCliqueSize: scope.sharedBufferMaxCliqueSize,
                sharedBufferMaxCliqueCache: scope.sharedBufferMaxCliqueCache,
                sharedBufferInitialColors: scope.sharedBufferInitialColors,
                sharedBufferJsonGraphValues: scope.sharedBufferJsonGraphValue
s
            });

            scope.workers.push(worker);
        }
    });
}
```

Figure 8. Workers initialization.

The jobs are started then and only then, when the separate workers are created and all the required data for the logic of branch traversal is prepared. When all jobs are completed, the algorithm returns to the original backtrack search process, where the current found maximum clique is saved. Thereafter the algorithm proceeds to the next iteration. At the end of all backtrack iterations previously created threads are deleted.

3.4.1 Results

In this chapter the performance of VRecolor-BT-u algorithm and its multithreaded version – VRecolor-BT-u-parallel algorithm is compared. The algorithms are tested on randomly generated graphs, as well as, on DIMACS graphs. The tests give us an overview of how the use of multithreading affects the speed of complex calculations.

Both algorithms are implemented in TypeScript using Visual Studio Code. Multithreading is implemented with the use of Web Workers API. All tests are run in Chrome browser (version 76). System information:

- Processor: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 1992 Mhz, 4 Cores, 8 Logical Processors
- RAM: 16GB
- System type: 64-bit operating system
- Operating system: Microsoft Windows 10 Pro

All tests on random graphs are divided by graph densities. 10-20 tests were run on different random graphs to obtain more accurate results for each density and a certain number of vertices.

Figure 9 demonstrates how the random graph is generated. It is based on the algorithm described in Kumlanders' work [7] with minor adjustments. The function takes two numeric parameters: order and density of the expected graph. The return value of the function is the Graph object, which represents a graph.

```
public generateGraph(order: number, density: number): Graph {
    if (density < 0 || density > 1)
        return null;

    const graph = new Graph(`gen_${order}_d${density}_${new Date().getTime()}
`, order);
    graph.edges = Math.round(order * (order - 1) * density / 2);

    for (let i = 0; i < graph.edges; i++) {
        let x: number;
        let y: number;

        do {
            x = this.getRandomInt(order);
            y = this.getRandomInt(order);
        } while (x == y || graph.values[x][y]);

        graph.setValue(x, y, true);
        graph.setValue(y, x, true);
    }

    return graph;
}

private getRandomInt(max: number): number {
    return Math.floor(Math.random() * Math.floor(max));
}
```

Figure 9. Random graph generation function.

As can be seen from the Figure 10, at a very low graph density (10%) the multithreaded algorithm is significantly slower in speed to the single threaded VRecolor-BT-u algorithm. However, already at a density of 20%, VRecolor-BT-u-parallel catches up with VRecolor-BT-u when the number of graph vertices reaches 1900 (Figure 11). The explanation of the lag of VRecolor-BT-u-parallel to VRecolor-BT-u at very low densities is simple: the time spent on going through the graph is supplemented by the time spent preparing data for additional threads, creating the threads themselves and transferring data between them, thereby making the multithreaded algorithm inefficient on densities 10% and also on 20% for the small number of vertices graphs.

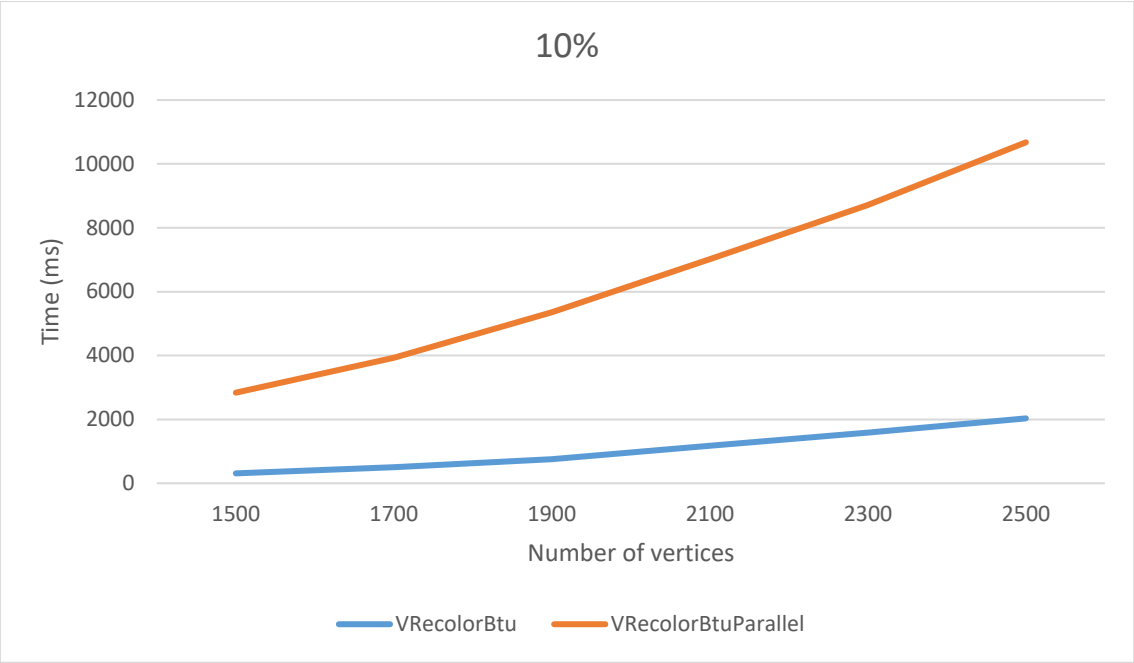

Figure 10. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 10%.

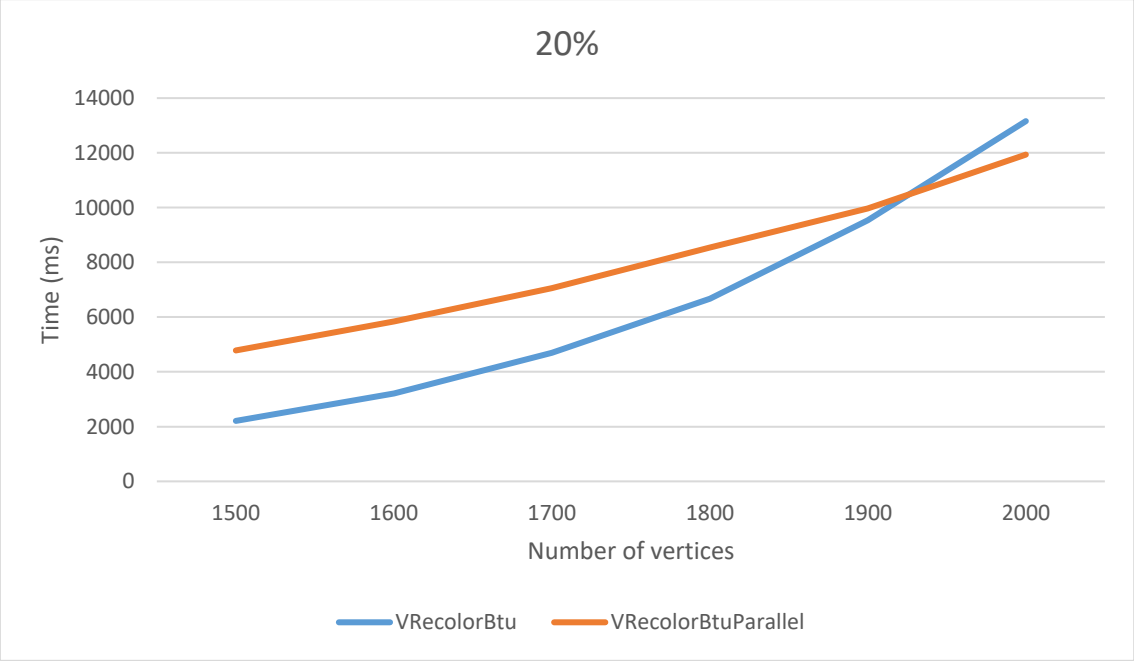

Figure 11. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 20%.

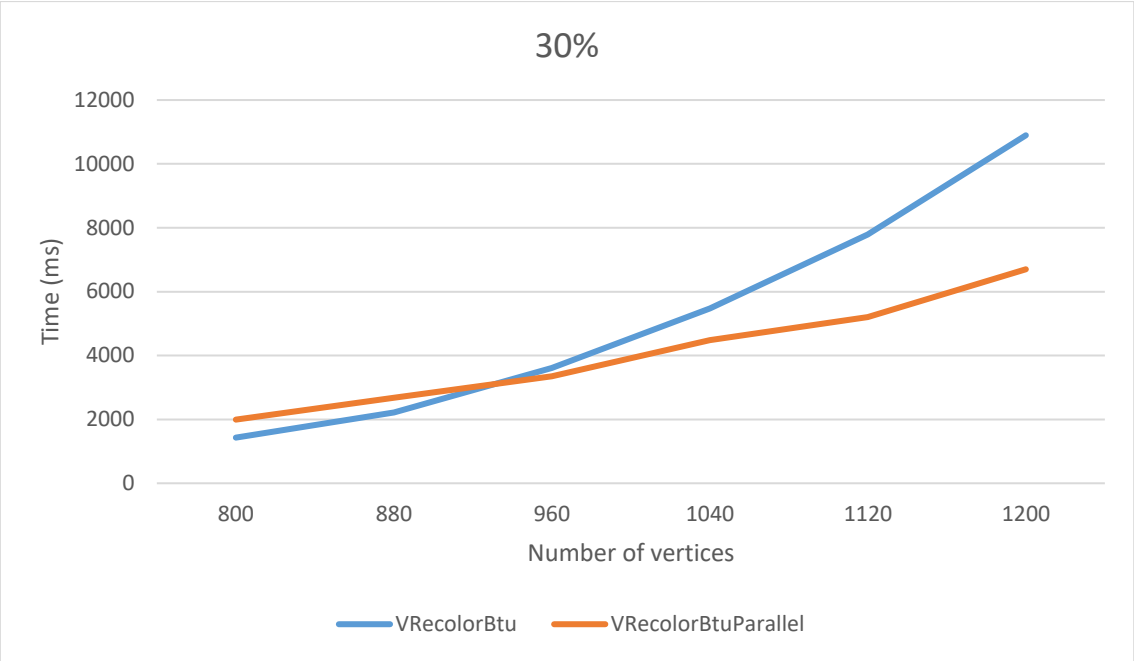

Figure 12. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 30%.

However, when the graph density is more than 30%, these additional time costs from workers execution become insignificant compared to the graph traversal process. As we see from Figure 12 to Figure 18, already at densities of 30%-90%, the multithreaded algorithm is more efficient than the single threaded one.

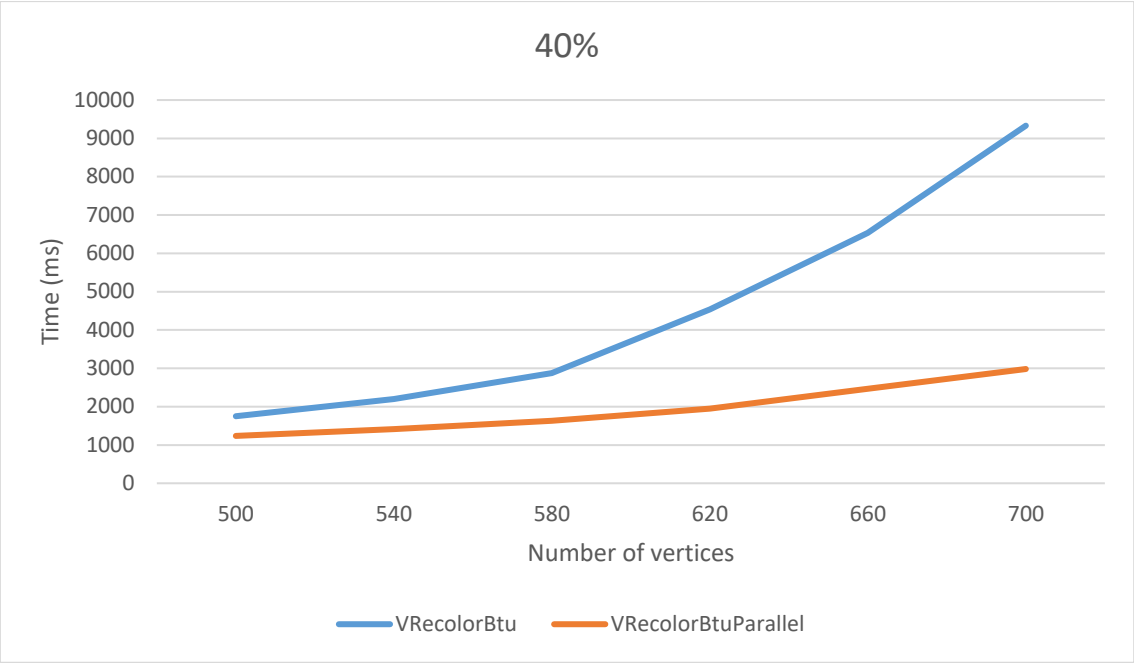

Figure 13. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 40%.

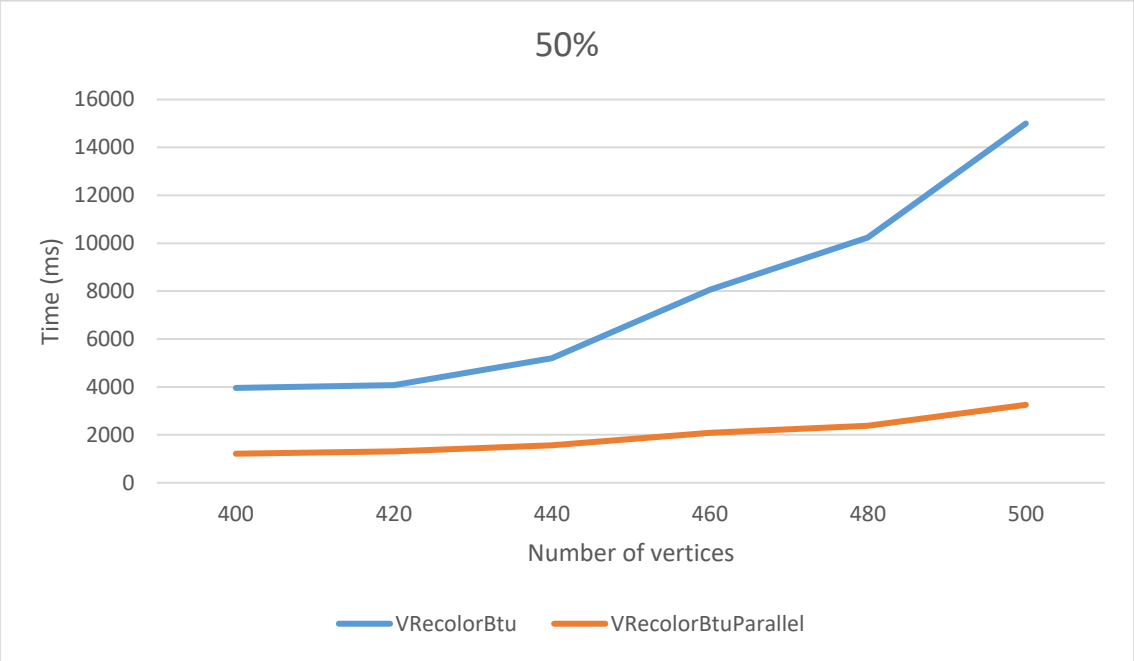

Figure 14. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 50%.

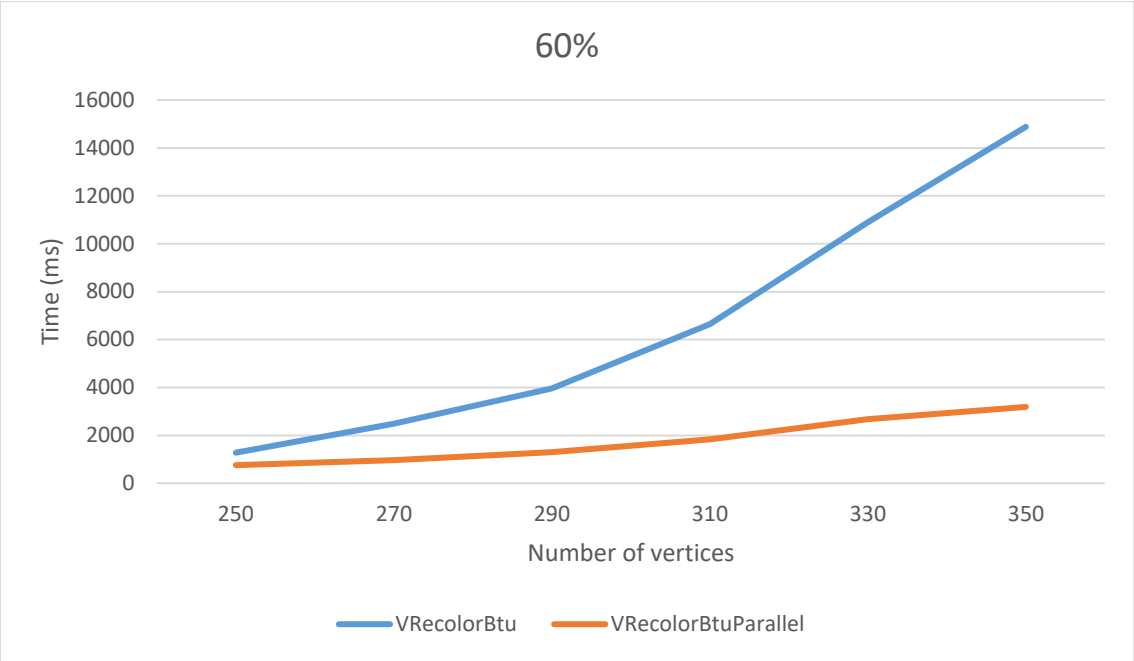

Figure 15. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 60%.

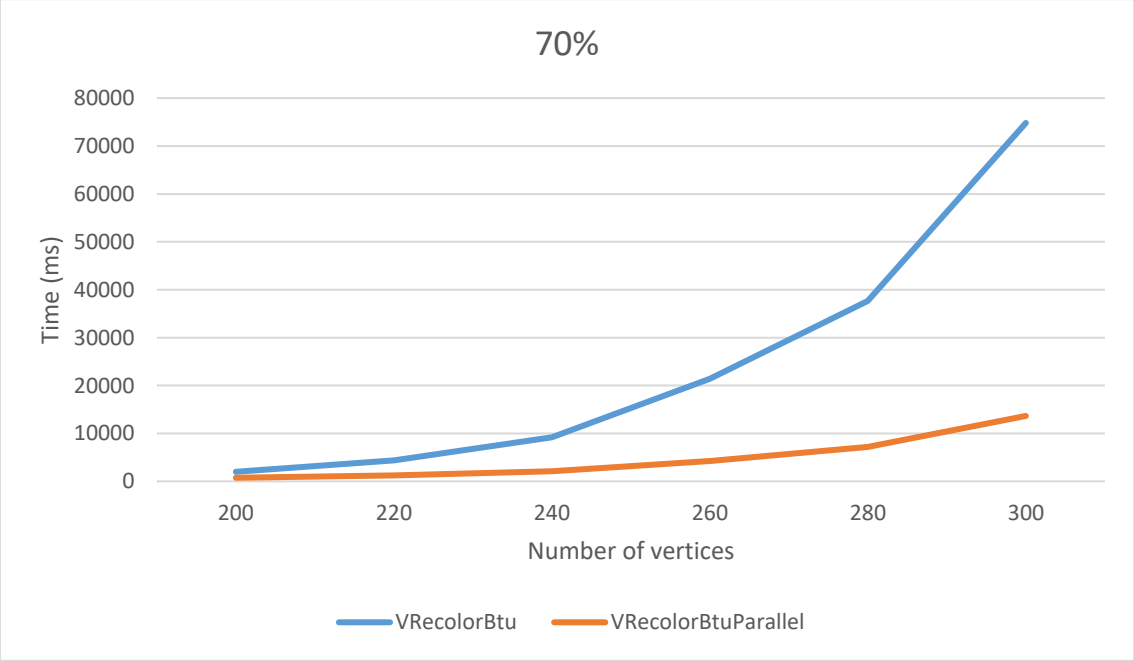

Figure 16. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 70%.

It is also worth noting that with an increase in the number of graph vertices, where the density varies from 30% to 90%, the VRecolor-BT-u-parallel algorithm becomes more and more efficient compared to VRecolor-BT-u.

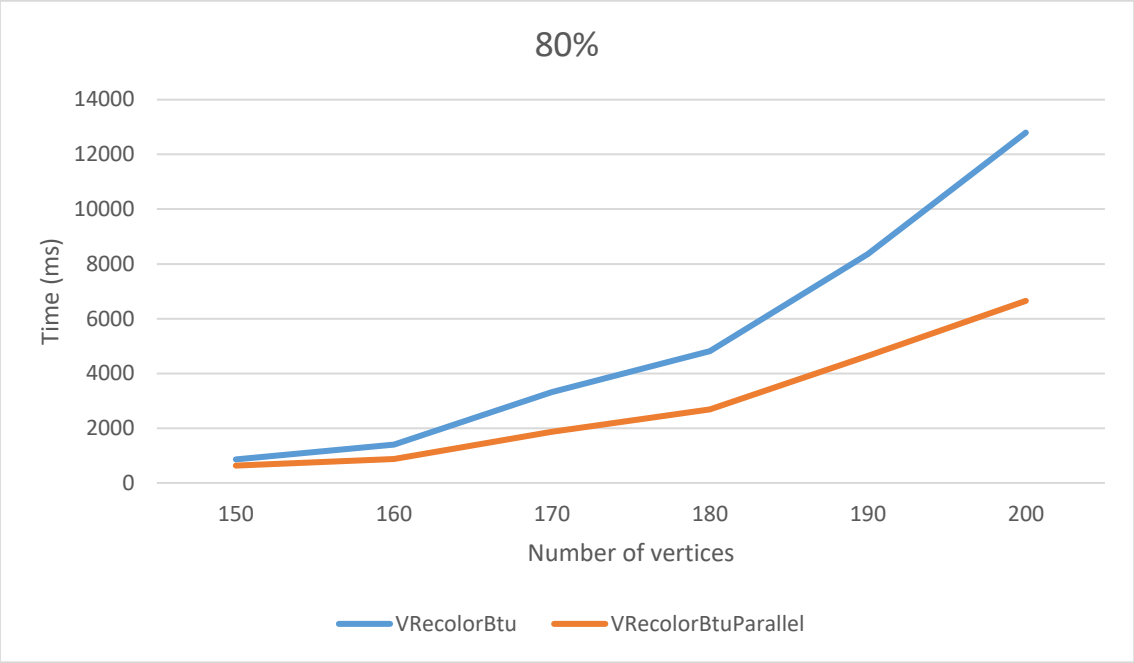

Figure 17. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 80%.

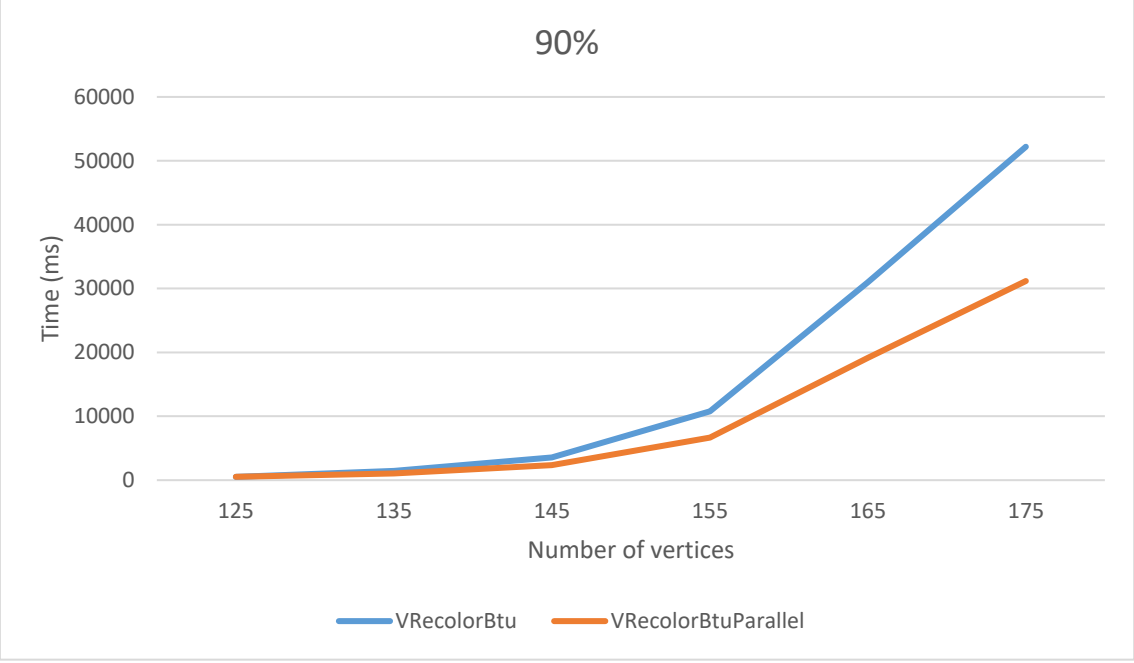

Figure 18. VRecolor-BT-u and VRecolor-BT-u-parallel randomly generated graph test results. Density 90%.

DIMACS graph is the graph presented in the standard format, which has been defined by DIMACS for problems in undirected graphs. DIMACS graphs have special structures representing the specific real problems. This standard was also chosen for several DIMACS Computational Challenges [15].

DIMACS graph test result (Table 1) demonstrate that the specificity of structures can negatively affect the speed of a multi-threaded algorithm. We see its superiority only on a small number of graphs.

Table 1. VRecolor-BT-u and VRecolor-BT-u-parallel DIMACS graph test results (ms).

Graph	Order	Density	VRecolor-BTu	VRecolor-BTuParallel
			Time(ms)	
c-fat500-1.clq	500	0,04	7	613
c-fat500-2.clq	500	0,07	6	372
c-fat500-5.clq	500	0,19	31	734
c-fat500-10.clq	500	0,37	107	1457
DSJC500_5.clq	500	0,5	8809	4345
DSJC1000_5.clq	1000	0,5	758014	306383
gen200_p0.9_44.clq	200	0,9	10456	5749
gen200_p0.9_55.clq	200	0,9	1104	1254
hamming6-2.clq	64	0,9	4	88
hamming6-4.clq	64	0,35	1	35
hamming8-2.clq	256	0,97	108	693
hamming8-4.clq	256	0,64	50	203
hamming10-2.clq	1024	0,99	24444	33051
johnson8-2-4.clq	28	0,56	2	336
johnson8-4-4.clq	70	0,77	6	58
johnson16-2-4.clq	120	0,76	641	401
keller4.clq	171	0,65	52	232
MANN_a9.clq	45	0,93	2	48
MANN_a27.clq	378	0,99	5077	5985
san200_0.7_1.clq	200	0,7	1472	370
san200_0.7_2.clq	200	0,7	3	109
san200_0.9_1.clq	200	0,9	28	377
san200_0.9_2.clq	200	0,9	769	751
san1000.clq	1000	0,5	466	893
p_hat300-1.clq	300	0,24	66	765
p_hat300-2.clq	300	0,49	191	483
p_hat300-3.clq	300	0,74	8307	4240
p_hat500-1.clq	500	0,25	160	719
p_hat500-2.clq	500	0,5	5405	4059
p_hat1000-1.clq	1000	0,24	1722	3135

3.5 Multithreaded VRecolor-BT-u and VRecolor-u in parallel

From the previous result, we see that the VRecolor-BT-u-parallel algorithm shows good results at densities above 30%, but is still slower than the single threaded graphs on very low density.

The second approach of how the performance can be improved even more by using multithreading was to execute two algorithms from different families in parallel: with backtrack search and without it. Thus, the search for maximum clique will be performed from both sides of the graph and stopped when these algorithms meet. The hypothesis regarding the performance improvement applying that approach was that the speed of execution the VRecolor-BT-u algorithm might increase around two times.

Since we already have a multithreaded version of the VRecolor-BT-u algorithm, which showed good results at almost all densities, it was chosen as the first algorithm. Being a backtracking algorithm, it starts the search for the maximum clique with a set of vertices from the first color class and continues the search by adding subsequent independent sets to the analysis.

VRecolor-BT-u is the successor of the VColor-BT-u algorithm, differing only in that it additionally applies recoloring on depths and uses additional pruning formulas. The VColor-BT-u algorithm is based on the same ideas as VColor-u (initial vertex coloring and the use of independent sets), except that VColor-u is based on the Carraghan and Pardalos algorithm and traverse graph by removing vertex by vertex from analysis, and VColor-BT-u on the Östergård algorithm, which analyze the graph by adding vertices one by one. Since those algorithms already by the approaches start traversing the graph in different directions and both are based on the same initial coloring and ordering of vertices, those families of algorithms seems to be the best choice for our approach.

In fact, if we remove the backtracking approach from VRecolor-BT-u, we get more advance variation of the VColor-u, same but with an additional vertex recoloring on every depth (similar to MCQ [8]), but with handling conflicts of initial and depth coloring proposed by Porošin [4]). Based on this, the modified VRecolor-BT-u will be called as VRecolor-u. Using additional techniques that accelerate the search for maximum clique, VRecolor-u will be more efficient than the VColor-u algorithm basing

on general investigations done by Porošin [4]. Therefore, VRecolor-u was selected as a companion for the VRecolor-BT-u-parallel in our improvement attempt.

In addition, from the results of comparing the performance of MCQ type algorithms (branch and bound with recoloring) and backtracking type algorithms [4, 7] we can see that the branch and bound with recoloring algorithms is faster on graphs with relatively low densities. This fact can positively affect the results of the combined algorithm at low graph densities, since VRecolor-u will likely quickly walk through the most of the graph, while VRecolor-BT-u-parallel will only begin the traversing. Thus, the time spent on creating workers and communication between threads will be compensated.

The combined algorithm begins with the initial graph coloring and ordering the vertices. Graph coloring gives us the number of color classes (i.e the number of independent sets). Next, two workers are created: one for the VRecolor-BT-u-parallel algorithm, the second for VRecolor-u. Since VRecolor-BT-u-parallel is a multithreaded algorithm, the workers required within it will be created inside the thread of the VRecolor-BT-u-parallel.

Figure 19 illustrates the creation of two important *SharedArrayBuffer* objects that will be passed to threads and used as triggers to determine when algorithms should be interrupted. The backtracking family algorithm VRecolor-BT-u starts from the vertices of the first color class, and the sharedBTColorsAnalysed object stores the number of colors that have already been taken into analysis. The branch and bound family algorithm VRecolor-u starts traversing the graph by excluding from analysis vertices and consequently color classes since vertices are ordered by them, so the sharedBnBColorsLeft object stores information about how many colors are left for research.

```
this._sharedBufferBnBColorsLeft = new SharedArrayBuffer(2);
const sharedBnBColorsLeft = new Uint16Array(this._sharedBufferBnBColorsLeft);
Atomics.store(sharedBnBColorsLeft, 0, initialColorsNumber);

this._sharedBufferBTColorsAnalysed = new SharedArrayBuffer(2);
const sharedBTColorsAnalysed = new Uint16Array(this._sharedBufferBTColorsAnal
ysed);
Atomics.store(sharedBTColorsAnalysed, 0, 0);
```

Figure 19. Use of SharedArrayBuffer object for the shared data.

In Figure 20 the part of code from the VRecolor-BT-u-parallel algorithms' worker is presented. At each new iteration, we increase the number of colors analyzed and store this value using atomic operations in shared memory. Atomic operations ensure that reading and writing data is synchronous and not interrupted. After the current subgraph has been analyzed, the number of left colors of the second algorithm is checked. If the second algorithm has the same (or less) number of colors left for analysis then further iterations are interrupted. The worker finishes his work and sends a message to the main stream with the maximum clique that it managed to find.

```
let colorsAnalysed = 0;
for (let c = 0; c < scope.initialColorsNumber; c++) {
    Atomics.store(scope.sharedBTColorsAnalysed, 0, ++colorsAnalysed);

    await processColorSet(c);

    let currentMaxClique = Atomics.load(sharedMaxCliqueSize, 0);
    Atomics.store(sharedMaxCliqueCache, c, currentMaxClique);

    if (colorsAnalysed >= Atomics.load(scope.sharedBnBColorsLeft, 0)) {
        scope.postMessage({ maxCliqueSize: currentMaxClique })
        destroyWorkers();
        break;
    }
}
```

Figure 20. Stop condition check in VRecolor-BT-u-parallel worker.

Figure 21 shows a code snippet from a VRecolor-u algorithm worker. Selecting a vertex for analyses on the first depth the algorithm picks the color class the vertex belongs to and checks if the second (i.e. backtracking) algorithm already analyzed that color. It is done by the following routine:

1. If the color of the vertex same as the color of the last analyzed vertex then we just continue.

2. If the algorithm moved to a new color class, it compares how many color classes are left to be analyzed to how many have already been analyzed in the backtracking algorithm. If these numbers match or the backtracking algorithm has analyzed more color classes than is left here, then the algorithm is interrupted and the maximum clique found to this moment is sent to the main thread.

```
const p = depthNodes[depth][inDepthIndex];
const color = scope.initialColors[p - 1];

if (depth == 0) {
    const prevColor = scope.initialColors[depthNodes[depth][inDepthIndex + 1]
] || c;

    if (prevColor != color) {
        if (colorsLeft <= Atomics.load(scope.sharedBTColorsAnalysed, 0))
            break;

        Atomics.store(scope.sharedBnBColorsLeft, 0, --colorsLeft);
    }
}
```

Figure 21. Stop condition check in VRecolor-u worker.

Whenever the algorithms finish their job and the response from each worker is received, the custom event „worker done“ is dispatched to the DOM. The Document object, which represents the DOM tree of our application, is subscribed to that custom event. After the „worker done“ events are dispatched from both algorithms, the main thread performs the last action: it compares the found maximum cliques and selects the larger one, which is returned as a result. This can be seen on Figure 22, where the behavior of „worker done“ event handler is demonstrated.

```
private workerDoneEventHandler(e: any): void {
    this._maxCliqueSizes.push(e.detail.maxCliqueSize);

    if (++this._workersDone != 2) {
        return;
    }

    e.detail.sw.stop();

    this._result = {
        maximumClique: Math.max(...this._maxCliqueSizes),
        timeElapsed: {
            total: e.detail.sw.timeElapsed
        }
    } as MaxCliqueAlgorithmResult;

    document.removeEventListener(this.WORKER_DONE_EVENT_NAME, this.workerDone
EventHandler, true);

    for (let i = 0; i < this._workers.length; i++)
        this._workers[i].terminate();
    this._workers = [];

    this._solutionRes();
}
```

Figure 22. Handler for “worker done” event.

3.5.1 Results

This chapter presents the test results of the VRecolor-u+VRecolor-BT-u-parallel algorithm, as well as their comparison with the results of the VRecolor-BT-u and VRecolor-BT-u-parallel algorithms. Those are tested on both random graphs and DIMACS graphs.

As can be seen from Figure 23 to Figure 24, the combined algorithm does better than the VRecolor-BT algorithm even on low-density graphs, thereby confirming the hypothesis that the VRecolor-u algorithm compensates the time spent on workers in VRecolor-BT-u-parallel algorithm.

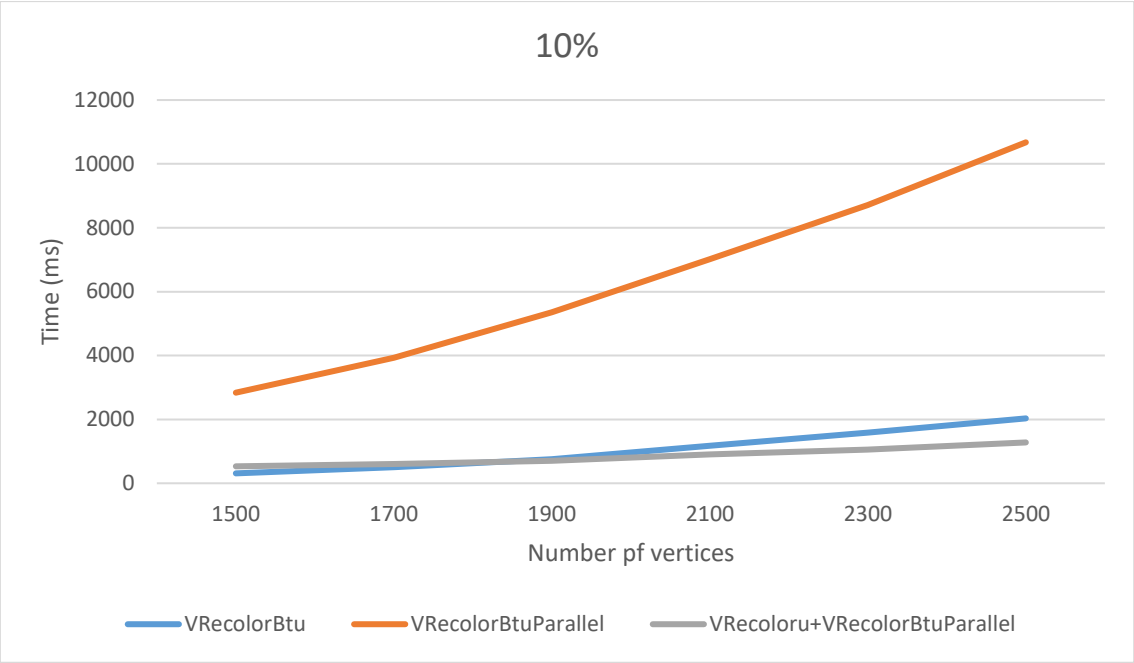

Figure 23. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 10%.

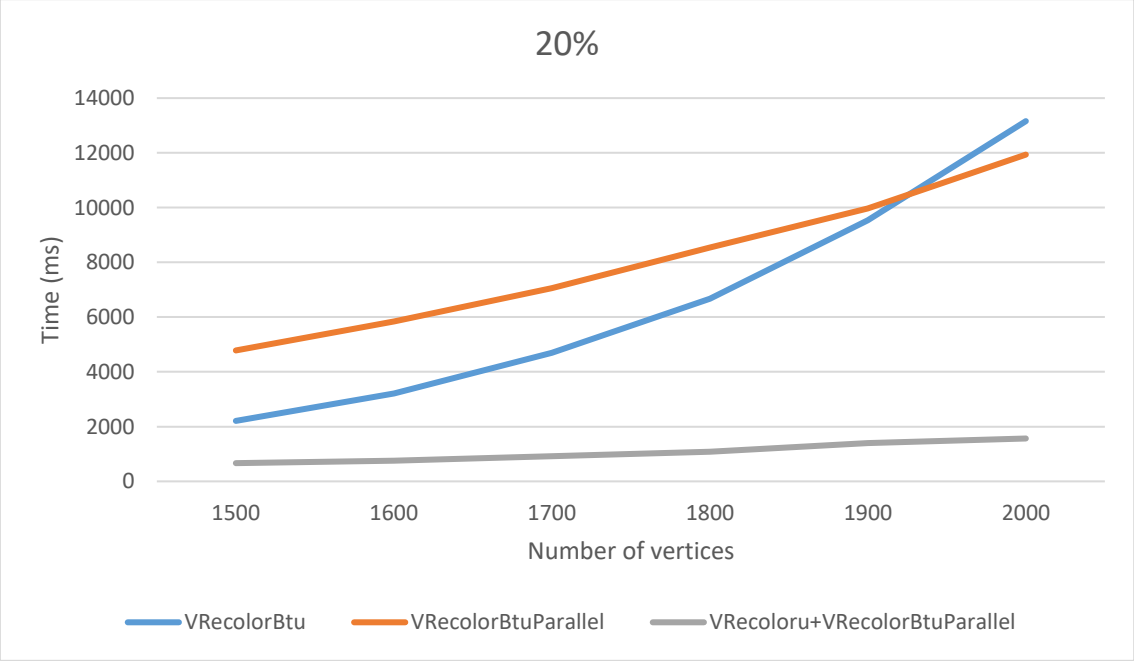

Figure 24. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 20%.

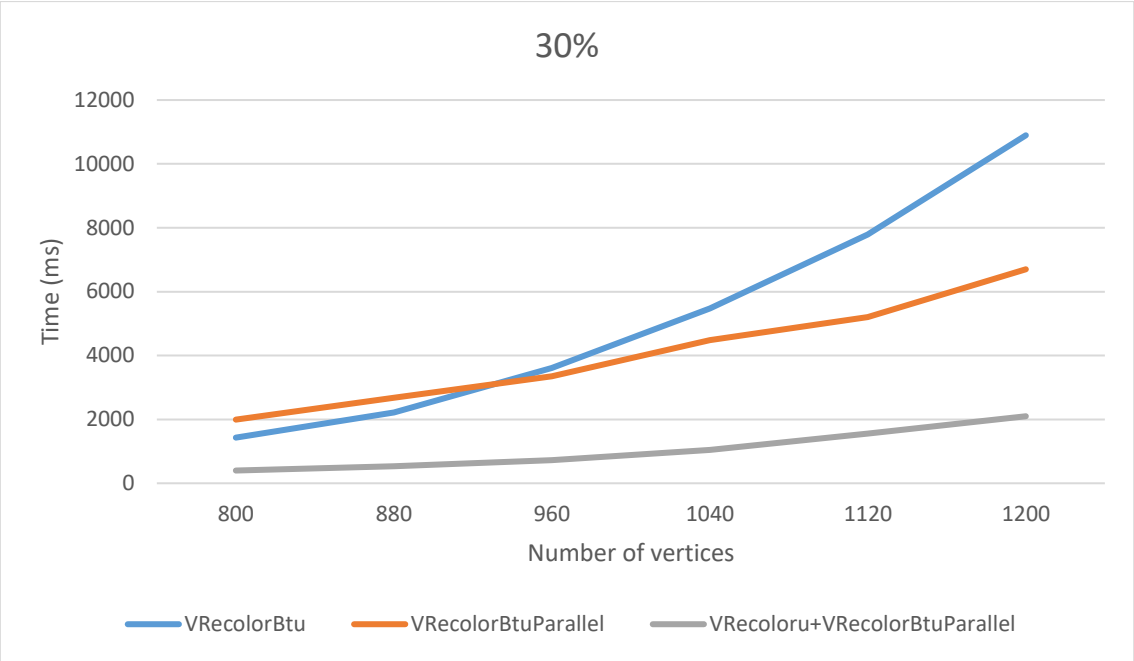

Figure 25. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 30%.

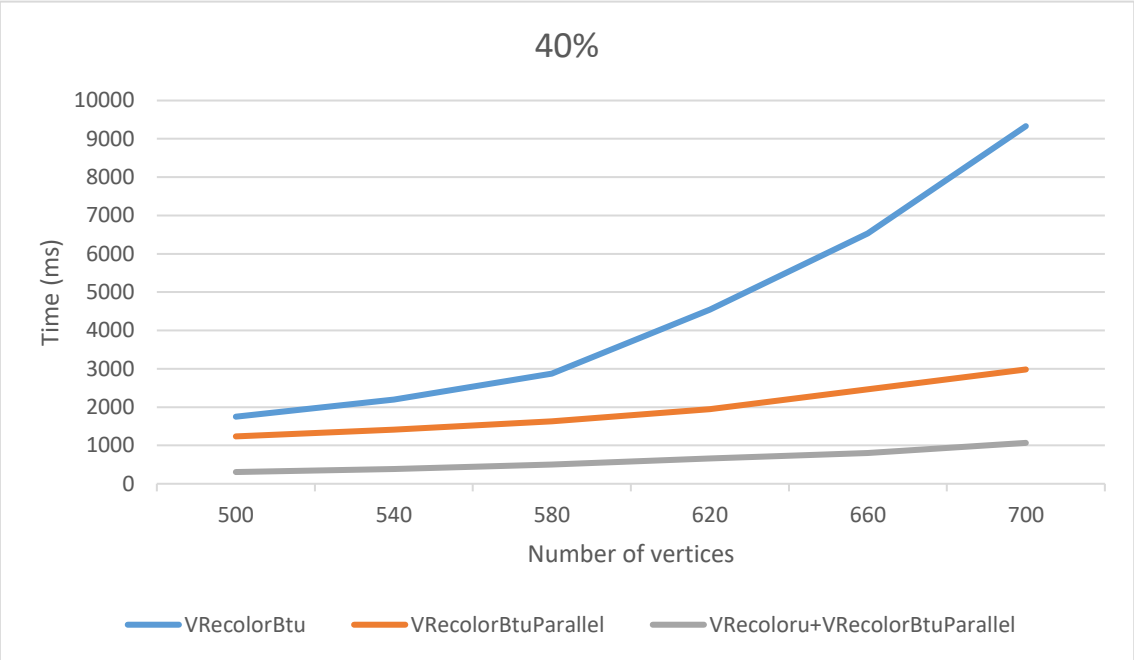

Figure 26. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 40%.

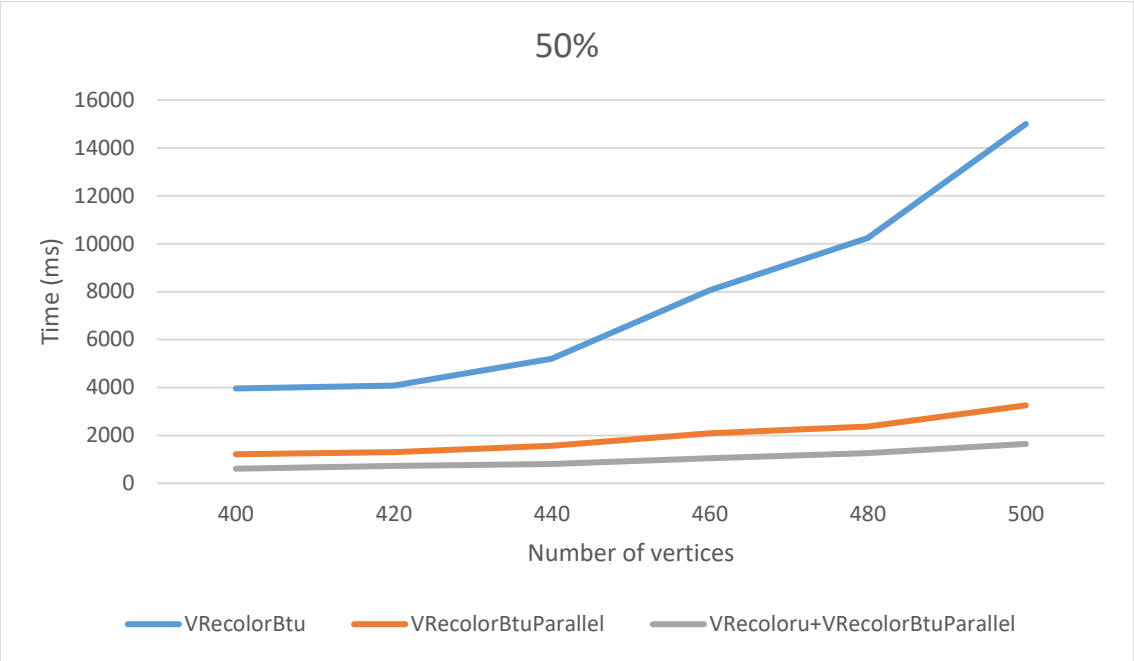

Figure 27. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 50%.

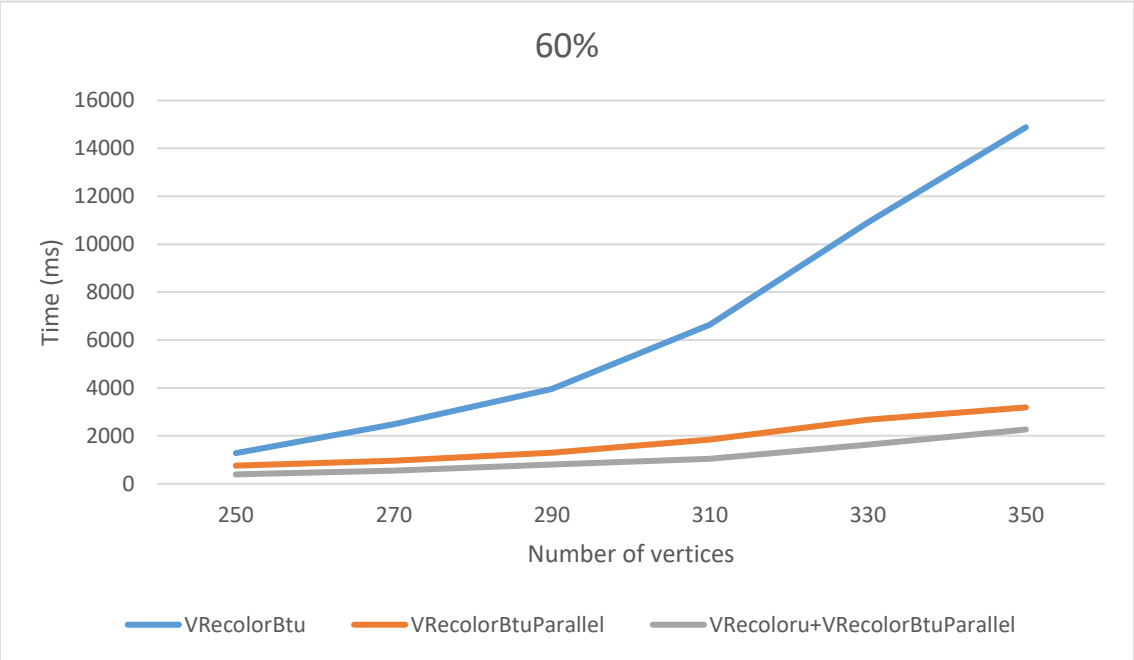

Figure 28. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 60%

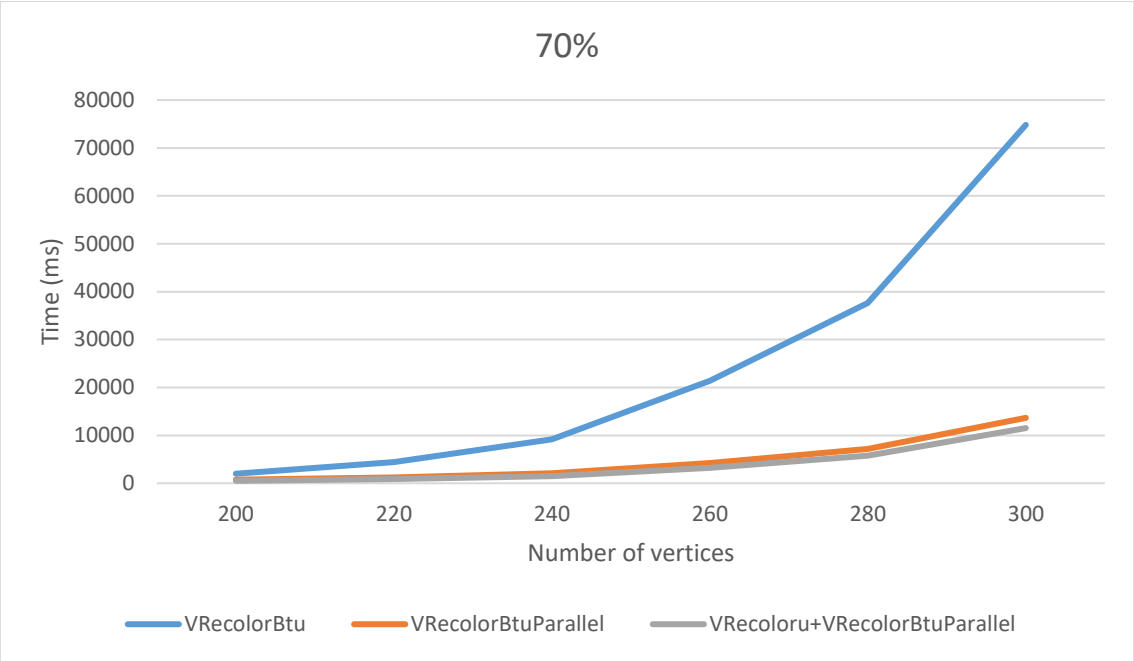

Figure 29. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 70%.

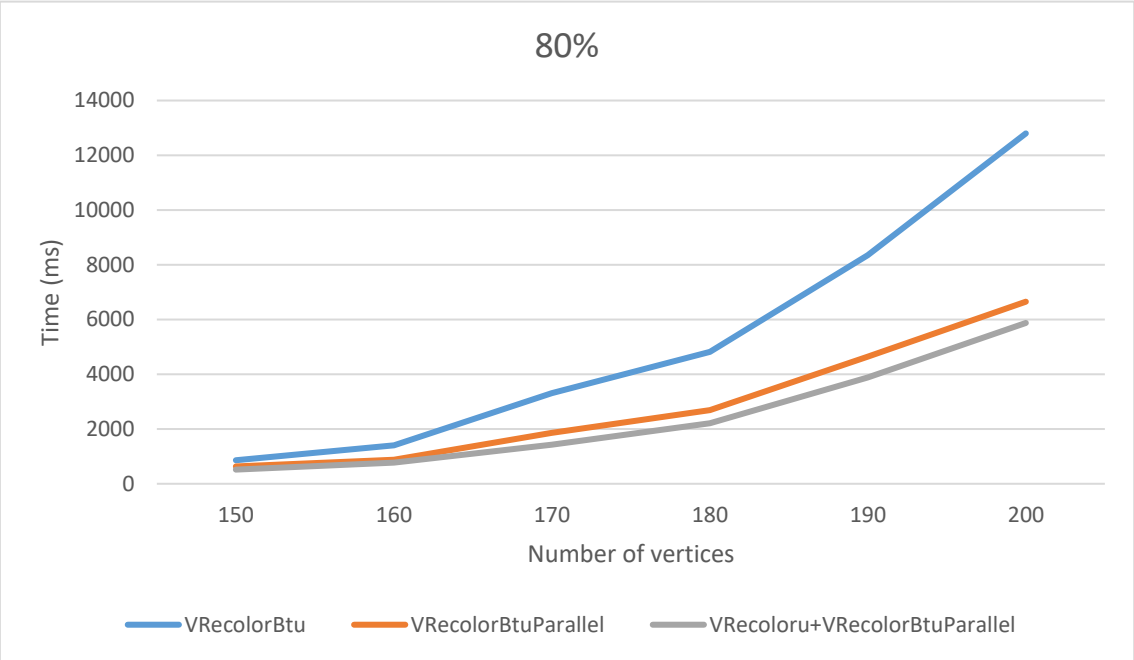

Figure 30. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 80%.

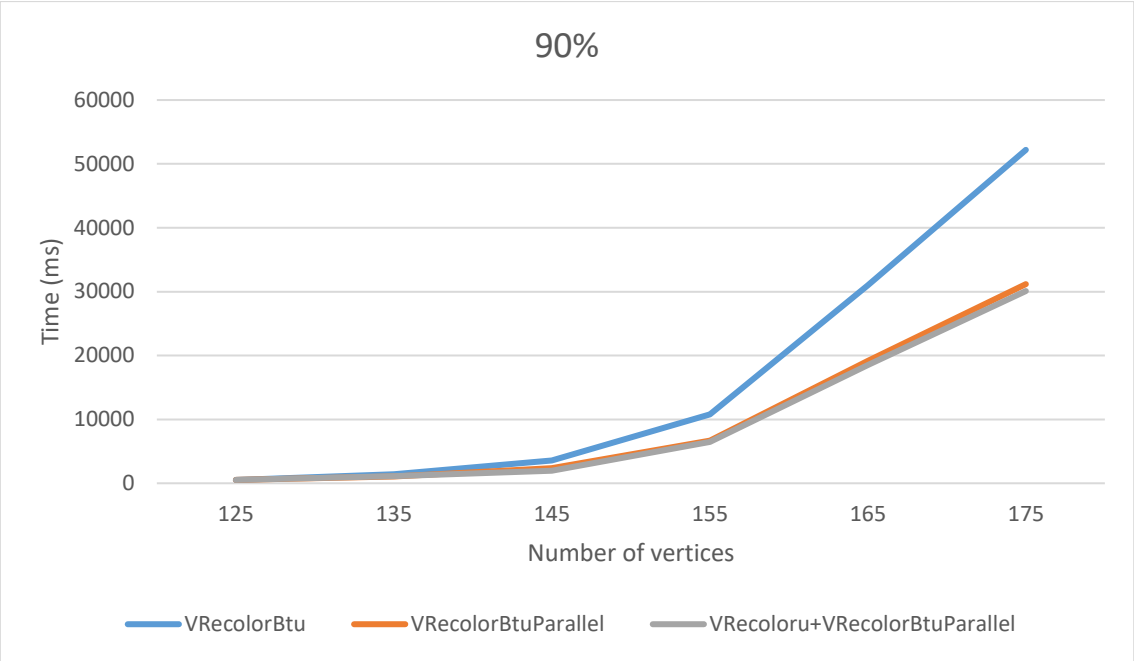

Figure 31. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel randomly generated graph test results. Density 90%.

All the results show that, the combined multithreaded algorithm oversteps all the algorithms under study. Its productiveness is especially evident in comparison with VRecolor-BT-u-parallel at densities of 10% - 50% (Figure 23 to Figure 27). However, the difference in execution speed becomes smaller with increasing density of graphs and is quite disappear by a density of 90%.

The DIMACS graphs result (Table 2) shows that the performance of algorithms very much depends on the structure of graphs and so the new one is better on some graphs than the previous one but still is not best for many cases and even slower for some individual graphs than the original parallel one. Nevertheless, we can see that the new algorithm outperforms other algorithms on quite many graphs, meaning that selection of algorithm to be used depends on the graph type and the algorithm proposed in this work will be selected as the best on quite some cases.

Table 2. VRecolor-BT-u,VRecolor-BT-u-parallel and VRecolor-u+VRecolor-BT-u-parallel DIMACS graph tests (ms).

Graph	Order	Density	VRecolor Btu	VRecolor BtuParallel	VRecoloru+ VRecolor BtuParallel
			Time (ms)		
c-fat500-1.clq	500	0,04	7	613	106
c-fat500-2.clq	500	0,07	6	372	195
c-fat500-5.clq	500	0,19	31	734	635
c-fat500-10.clq	500	0,37	107	1457	1358
DSJC500_5.clq	500	0,5	8809	4345	2154
DSJC1000_5.clq	1000	0,5	758014	306383	159338
gen200_p0.9_44.clq	200	0,9	10456	5749	7054
gen200_p0.9_55.clq	200	0,9	1104	1254	2542
hamming6-2.clq	64	0,9	4	88	77
hamming6-4.clq	64	0,35	1	35	31
hamming8-2.clq	256	0,97	108	693	579
hamming8-4.clq	256	0,64	50	203	238
hamming10-2.clq	1024	0,99	24444	33051	33736
johnson8-2-4.clq	28	0,56	2	336	40
johnson8-4-4.clq	70	0,77	6	58	32
johnson16-2-4.clq	120	0,76	641	401	231
keller4.clq	171	0,65	52	232	120
MANN_a9.clq	45	0,93	2	48	28
MANN_a27.clq	378	0,99	5077	5985	4893
san200_0.7_1.clq	200	0,7	1472	370	389
san200_0.7_2.clq	200	0,7	3	109	94
san200_0.9_1.clq	200	0,9	28	377	323
san200_0.9_2.clq	200	0,9	769	751	4580
san1000.clq	1000	0,5	466	893	246
p_hat300-1.clq	300	0,24	66	765	76
p_hat300-2.clq	300	0,49	191	483	200
p_hat300-3.clq	300	0,74	8307	4240	3554
p_hat500-1.clq	500	0,25	160	719	120
p_hat500-2.clq	500	0,5	5405	4059	2333
p_hat1000-1.clq	1000	0,24	1722	3135	905

4 Conclusion

4.1 Summary

In the course of the work, two basic branch and bound maximum clique search algorithms were studied: Carraghan and Pardalos algorithm [5] and Östergård's algorithm [6]. These algorithms are based on two different approaches of traversing a graph. The first one analyzes vertices one by one removing those from analysis and the second one uses a reversed search called backtrack search i.e. adds vertices into analysis one by one keeping the history of analysis in a cache to efficiently prone branches. Additionally, three modern algorithms based on the above-mentioned basic algorithms were also considered: VColor-u, VColor-BT-u and VRecolor-BT-u. The first two algorithms differ from the basic ones in that they work at a higher level, operating not with individual vertices, but with independent sets obtained from the initial graph coloring. Additional pruning formulas, based on the received color classes, increase the number of trimmed branches, thereby speeding up the execution of the algorithm. The third VRecolor-BT-u algorithm is a successor to the VColor-BT-u algorithm and is currently one of the most efficient maximum clique search algorithms. In addition to initial coloring, it also performs vertex coloring in depths, which makes it possible to use another additional formula to reduce the studied branches. Due to the fact that the VRecolor-BT-u algorithm was recognized as one of the fastest, it was chosen for the implementation of the goals of this thesis.

The third chapter is dedicated to the practical part of the work. At the beginning of the chapter, the technological stack of the study is introduced. By the request of the company we collaborated with writing this thesis, the VRecolor-BT-u algorithm was ported to the web using Angular framework. The corner stone of the technology used in the work is multithreading, which is implemented using Web Workers. Web Workers' capabilities are somewhat limited in some aspects, so experimental solutions have also been found that allows efficient usage of workers in our algorithms.

After the technology introduction, two hypotheses are presented on how the VRecolor-BT-u algorithm can be accelerated using multithreading. These ideas are accompanied by a detailed explanation of the main points of development, as well as the tests results, conducted on randomly generated graphs and DIMACS graphs.

The first idea of improving the VRecolor-BT-u algorithm is based on performing branch traversal in separate threads. The results are good on random graphs with densities more than 30%. The efficiency of the VRecolor-BT-u-parallel algorithm compared to VRecolor-BT-u increases exponentially with growth of the number of graph vertices. However, the multithreaded version is inferior to the single threaded version on graphs with a very low densities, because the time spent on workers is too long compared to the graph traversal process.

The second idea is to combine in one algorithm two sub-algorithms that are based on different graph traversal techniques: backtrack search and branch and bound, which are executed in separate threads. The idea is that the algorithms start traversing the graph from different ends, i.e. one is removing vertices from its' analysis and the other one is adding them. They stop the analysis when arrive to the same vertex and after analysis of that vertex the best so far found maximum clique is selected as a solution, since the first algorithm analyzed the vertices prior to the stop vertex, the second analyzed all vertices residing after that vertex and one of them analyzed the stop vertex. An important condition is – both algorithms should on the same vertex ordering (in fact we also use color classes and so those should have also same order). In terms of performance, the combined algorithm surpassed both single threaded VRecolor-BT-u and multithreaded VRecolor-BT-u-parallel algorithms at all densities on randomly generated graphs.

4.2 Future studies

This chapter will present two ideas for possible further research.

The first investigation target is usage of workers depending on the algorithm family and densities of the graph, since our tests in that area so far wasn't quite homogenous and need deep dive into details.

The second interesting topic would be applying the technique described in this thesis to the weighted case. Weighted maximum clique algorithms are both alike and different:

having similar structure those are different in some important details, therefore transferring the described approach will not be just a mechanical copy.

References

[1] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NPcompleteness, New York, 2003.

[2] A. Buldas, P. Laud ja J. Villemson, Graafid, Tartu, 2003.

[3] R. J. Wilson, Introduction to Graph Theory, California, 1996.

[4] A. Porošin, Reversed Search Maximum Clique Algorithm Based on Recoloring, Tallinn, 2015.

[5] R. Carraghan and P. M. Pardalos, "An exact algorithm for the maximum clique problem.," *Op. Research Letters 9,* pp. 375-382, 1990.

[6] P. R. Östergård, "A fast algorithm for the maximum clique problem," *Discrete Applied Mathematics 120,* pp. 197-207, 2002.

[7] D. Kumlander, Some Practical Algorithms to Solve The Maximum Clique Problem, Tallinn, 2005.

[8] T. Seki and E. Tomita, "An efficient branch-and-bound algorithm for finding a maximum clique," *DMTCS'03: Proceedings of the 4th international conference on Discrete mathematics and theoretical computer science,* p. 278–289, 2003.

[9] M. Batsyn, B. Goldengorin, E. Maslov and P. M. Pardalos, "Improvements to MCS algorithm for the maximum clique problem," *Springer Science+Business Media,* 2013.

[10] "Angular," [Online]. Available: https://angular.io/. [Accessed 1 01 2020].

[11] "Transferable," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Transferable. [Accessed 6 01 2020].

[12] "Transferable," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Transferable. [Accessed 3 01 2020].

[13] "SharedArrayBuffer," [Online]. Available: http://man.hubwiz.com/docset/JavaScript.docset/Contents/Resources/Documents/developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer.html. [Accessed 31 12 2019].

[14] "How fast are web workers?," [Online]. Available: https://hacks.mozilla.org/2015/07/how-fast-are-web-workers/. [Accessed 31 12 2019].

[15] "DIMACS format," [Online]. Available: http://lcs.ios.ac.cn/~caisw/Resource/about_DIMACS_graph_format.txt. [Accessed 31 12 2019].