Karlsruhe Institute of Technology

# Coloring Complex Networks

Bachelor Thesis of


## Klaus Lukas Hübner

At the Department of Informatics
Institute for Theoretical Computer Science, Algorithmics II

Supervisors: Prof. Dr. Peter Sanders
Dr. Christian Schulz


Submission Date: 24.09.2014

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

. . . . . .  . . . . . .

**Ort, Datum**  (**Klaus Lukas Hübner**)

**Abstract**

In this thesis an overview over existing graph coloring algorithms and their applicability to complex graphs representing social networks is given. Based on the observation that for a small $k$, the $k$-core of these graphs consists of very few vertices, two new algorithms to color social network graphs are proposed. Both try to color a $k$-core of the graph using an expensive heuristic and then coloring the remaining graph using a fast heuristic. The running time and number of colors used by the two proposed algorithms on social networks is then tested and compared to various other heuristics and a parallel version of LDF. LDF and the parallel version Parallel LDF proved very well. They where the fastest algorithms tested and used as many colors as Dsatur for most of the graphs.

**Zusammenfassung**

In dieser Thesis wird ein Überblick über existierende Graphfärbungsalgorithmen und deren Anwendbarkeit auf Soziale Netzwerk Graphen gegeben. Basierend auf der Beobachtung, dass für ein relativ kleines $k$ der $k$-core eines solchen Graphen aus sehr wenigen Knoten besteht, werden zwei neue Algorithmen zur Färbung Sozialer Netzwerk Graphen vorgeschlagen. Beide färben zunächst den $k$-core eines Graphen mit einer laufzeitintensiven Heuristik und den den verbleibenden Graphen mit einer schnellen Heuristik. Die Laufzeit und Anzahl der benutzten Farben der beiden vorgeschlagenen Algorithmen auf Sozialen Netzwerk Graphen wird getestet und mit anderen Heuristiken sowie einer parallelen Version von LDF verglichen. LDF und die parallele Version Parallel LDF stellten sich als sehr geeignet heraus. Sie waren die zwei schnellsten getesteten Algorithmen und erzeugten auf den meisten Graphen eine Färbung welche genauso wenige Farben benötigte wie die von Dsatur berechnete.

**Acknowledgements**

I would like to thank Prof. Dr. Peter Sanders and Dr. Christian Schulz for the opportunity to write my bachelor thesis on the very interesting subject of developing and evaluating heuristics for $\mathcal{NP}$-complete problems.

I would also like to thank Kristin Bachmann and Matthias Cipold for proofreading my thesis, Ilja Kantorovitch for getting me interested in Computer Programming nearly seven years ago and of cause my parents, which enabled me to study Computer Science.

# Contents

# 1. Motivation

The optimization version of the graph vertex coloring problem can be informally described as: given a graph $G = (E, V)$, assign every vertex $v \in V$ a color such that no adjacent vertices are colored with the same color. Use as few colors as possible.

The graph vertex coloring problem was posed in 1852 by Francis Guthrie. He tried to color the countries on a map in a way, that no countries sharing a common border are assigned the same color. He also noted that four colors would suffice to solve any instance of this problem, which would later be formalized as coloring of planar graphs. In 1893 Alfred Kempe drew the attention to the general, non-planar case of the graph coloring problem [29, p. 2]. As an algorithmic problem, graph coloring has been studied since the early 1970s and is one of the famous 21 $\mathcal{NP}$-complete problems by Karp (1972) [27].

Today graph coloring is largely used to tackle scheduling problems. Two examples being assigning aircrafts to flights or allocating bandwidth for radio stations. In the simplest form, a number of tasks has to be completed as fast as possible. Every task takes one time unit to complete and there is no restriction on how much tasks can be worked on at the same time. In this example, tasks are modeled as vertices and conflicts between tasks, for example the usage of a shared resource, are modeled using edges. The minimum number of colors needed to color this graph is exactly the number of time units it takes to complete all tasks.

## 1.1 Contribution

Until now most of the research in graph coloring has been done on relatively small graphs ($< 4000$ vertices). None of the graphs used in the Second DIMACS

Implementation Challenge has more than 1000 edges [17]. Even today most papers measure their proposed algorithms performance based on test run on the DIMACS's and other small graphs. There are however exceptions as for example "Ordering heuristics for parallel graph coloring" ([30]). The graphs we target in this thesis are in the order of magnitude of the 10th DIMACS Implementation Challenge graphs. For example the graph *uk-2002*, which has $\sim 1.8 * 10^7$ nodes and $\sim 2.6 * 10^8$ edges. Because coloring general graphs this size would take too long, we restricted the graphs of interest to those which have a "social network like structure". These can be informally defined as graphs consisting of many disjoint subgraphs which are "dense" and having relatively "few" edges between these subgraphs. A more practical approach using $k$-cores is described in Chapter 4.1. A characterization based on the degree distribution of social networks is also given.

We have implemented a few classic graph coloring heuristics such as LargestDegreeFirst, Matula & Beck [10] (SmallestDegreeLast) and Dsatur [6]. In this work, we propose two new algorithms which exploit the structural property described by choosing a subgraph that is "hard" to color, processing it using Dsatur and then coloring the remaining graph with LDF. A parallel version of LDF has also been implemented and tested.

## 1.2 Overview

After the introduction and preliminaries in Chapter 2, an overview of the wide spectrum of existing algorithms for graph coloring is given in Chapter 3. This includes implementation details of the algorithms. We also implemented the two proposed algorithms, kCoreFirst and SelectedFirst. Both algorithms are described in Chapter 4.

In Chapter 5 we compared the implemented heuristics. The experimental setup, the graphs used for testing and how they where determined is described. We then proceed to report our experimental results and conclude in Chapter 6.

# 2. Preliminaries

In this chapter, we define all concepts used in this thesis, such as graphs, subgraphs, $k$-cores and the graph vertex coloring problem.

## 2.1 Definitions

A graph $G = (V, E)$ is defined as a tuple of two sets. The first set $V$ contains all vertices, sometimes called nodes, of the graph. As we are only dealing with undirected graphs in this thesis, the second tuple $E \subset V \times V$ contains edges $e = \{u, v\}$. These edges describe a "connection" between the two vertices $u$ and $v$. Note, that we are not dealing with multigraphs here, there cannot be two edges connecting the same two vertices. Two vertices $v$ and $u$ are called adjacent if $\{u, v\} \in E$. The neighbors of a vertex $v$ are defined as all vertices which are adjacent to $v$. A color is called adjacent to a vertex $v$ if one of $v$'s neighbors is colored with this color. A graph is called loop free, if $\forall \{u, v\} \in E : u \neq v$.

The degree $d(v)$ of a vertex $v$ is defined as $d(v) = |\{e \in E \mid v \in e\}|$. A graph $G' = (V', E')$ is called a vertex induced subgraph of $G$ if $V' \subset V$ and $E' = \{\{u, v\} \in E \mid v \in V', u \in V'\}$. $G - v$ is the vertex induced subgraph $(V', E')$ with $V' = V \setminus \{v\}$. Two vertex induced subgraphs $G$ and $G'$ are called disjoint if $\forall v \in G : v \notin G'$. A path between two vertices $v$ and $u$ is a sequence of edges which connect a sequence of vertices starting with $v$ and ending with $u$. An undirected graph $G$ is called connected if there exists a path between every two vertices in $G$. A connected subgraph $G' = (V', E')$ of $G$ is called maximal connected component of $G$ if there is no other connected subgraph $G'' = (V'', E'')$ of $G$ with $V' \subset V''$.

The graph vertex coloring problem, sometimes called only graph coloring problem, is formally defined as follows: given an undirected, loop free graph $G = (V, E)$, find a coloring $c : V \to C \subset \mathbb{N}$ such that $\forall u, v \in E : c(u) \neq c(v)$ while minimizing $|C|$. Graph coloring can be more vividly described as the problem of assigning colors to the vertices of a graph such that no two adjacent vertices have the same color assigned to them.

The $k$-core of a given graph $G = (V, E)$ is defined as the maximal subgraph $G_k = (V_k, E_k)$ of $G$ with $\forall v \in V_k : d_{G_k}(v) \geq k$ where $d_{G_i}(v)$ denotes the degree of $v$ in $G_i$. One can compute the $k$-core of a graph by iterative removal of all vertices which have a degree lower than $k$. In the remaining subgraph all vertices have a degree of at least $k$. See Figure 2.1 for an example regarding $k$-cores. All red vertices are in the 3-core of the graph, all blue vertices are not. Vertex $g$ is obviously not in the 3-core because it has a degree of $d(g) = 2 < 3$. Because $g$ is not in the 3-core of $G$, the vertices $e$ and $f$, despite having a degree of $d(e) = d(f) = 3 \geq 3$ in $G$, cannot be in the 3-core, because their degree in a subgraph of $G$ without $g$ can be at most 2. In the $V' = \{a, b, c, d\}$ induced subgraph of $G$, all vertices have a degree of at least 3. As all other vertices cannot be in the 3-core of $G$ as outlined above, this subgraph is also maximal. Therefore the 3-core of $G$ is $G_3 = \{\{a, b, c, d\}, E_3\}$.

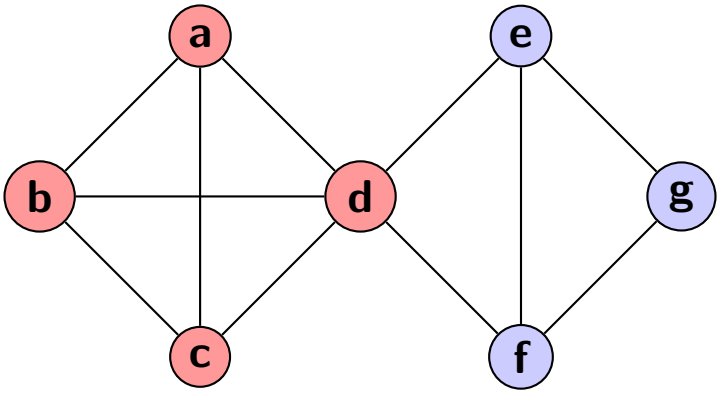


Figure 2.1: Red vertices are in the 3-core, blue vertices are not.

# 3. Related Work

Graph coloring has been studied for more than 40 years and a lot of algorithms have been developed. As graph coloring is $\mathcal{NP}$-hard [27] there are few exact algorithms and a lot more heuristics which only solve the problem approximately. Exact algorithms are too slow to handle the graphs we run our tests on in acceptable time. In the following section, we are giving an overview over the previous work done on graph coloring.

## 3.1 LargestDegreeFirst

The simplest heuristic to color the vertices of a graph is to color them greedy in whatever order they are present in the underlying data structure. This is called the Greedy approach. One step up would be to order the vertices somehow and then color them in this particular order. LargestDegreeFirst (LDF), arranges the vertices of the graph by degree, in descending order, and then colors them using the Greedy method.

We have implemented LDF largely as a reference for speed and quality. This means that we are expecting every other algorithm to be slower but yield a coloring using fewer colors than LDF.

We also implemented a simple parallel version of LargestDegreeFirst (PLDF), which is described in Chapter 4.

### Implementation

LargestDegreeFirst is trivially implemented by first sorting all nodes by degree in descending order using Bucket Sort and then iterating over them in this

sorted order, iteratively assigning them the lowest possible color.

This is done using a method `lowestValidColor`, which returns the lowest color that can be assigned to $v$ without hurting the graph coloring property, that is the lowest color none of the neighbors of $v$ has assigned to it. Formally speaking: $\texttt{lowestValidColor}(v) = \min\{x \in \mathbb{N} \mid \forall\{v, u\} \in E : x \neq c(u)\}$. This runs in $\mathcal{O}(d(v))$ time. Caching the adjacent colors of a vertex would result in no improvement, as the cache would have to be updated for every coloring of an adjacent node, which also happens $d(v)$ times, and `lowestValidColor` is called exactly once per vertex and coloring. The running time of this operation amortizes and all invocations together run in $\Theta(|E|)$ time. The total running time of LargestDegreeFirst is therefore $\Theta(|V| + |E|)$.

## 3.2 SmallestDegreeLast

The SmallestDegreeLast heuristic colors the vertices in the reverse order they where removed from the graph by the Matula & Beck algorithm (see below) used to build the $k$-cores of the graph. It is expected to be slower than LargestDegreeFirst but yield a coloring using fewer colors.

### 3.2.1 Matula & Beck

Matula & Beck finds the $k$-cores of a graph in linear time by repeatedly removing the vertex with the smallest degree. It works as follows:

1. Initialize $d_v = d(v)$ for every node $v \in V$. Later $d_v$ will be updated to only represent the degree of $v$ in the current graph (i.e. not counting the edges to nodes already removed). For nodes no longer in the current subgraph, $d_v$ will be invalid. $d_v$ is implemented using an array, this means writing and reading uses constant time.

2. Initialize a bucket priority queue $D$, such that $D[i] = \{v \mid d_v = i\}$. Later on, only nodes not already removed from the graph will be in this priority queue. Only the first element of each bucket $D[i]$ is accessed, we can therefore make $D$ an array to get access in constant time.

3. Initialize $k \leftarrow 0$, and $i \leftarrow 0$. Later, $i$ points to the first non-empty bucket.

4. Repeat $|V|$ times

   4.1. If the bucket $i$ points to is empty, search for the first non-empty bucket in $D$, start at $i$. This runs in time linear to the number of buckets which

has been set to the maximum degree of any node in $G$. Notice that $i$ can become smaller again when a vertex from the bucket $i$ points to is moved down and therefore a previously empty bucket is no longer empty.

4.2. Set $k \leftarrow \max\{k, i\}$, all skipped $k$-cores are set to "empty". This operation amortizes over all runs. In total, this step takes $\Theta(k)$ time, $k$ being the largest $k$ for which the $k$-core of the graph is not empty.

4.3. Remove the first node $v$ from the first non-empty bucket, add it to the current $k$-core and invalidate $d_v$. As $i$ points to the desired bucket, this step runs in $\mathcal{O}(1)$.

4.4. Update $d_u$ and move $u$ to $D[d_u]$ for all neighbors of $v$, not including the already removed ones. We used a secondary data structure, an array, to save the current index of $u$ in $D[d_u]$. This gets updated every time a node $u$ is moved between buckets. As $D[d_u]$ is a vector the size of $|V|$ and has therefore never to be resized, this enables us to move $u$ between buckets in constant time. As vertices are only moved down and only by one bucket at a time, we can maintain a valid value for $i$ by decreasing it by one every time a vertex from the lowest non-empty bucket is moved one bucket lower. The whole update process over all neighbors of $v$ runs in $\mathcal{O}(d(v))$ time.

The whole algorithm uses $\Theta(|V|)$ space and $\Theta(|V| + |E|)$ time.

## 3.3 DSATUR

DSATUR [6] is one of the slowest heuristics for coloring the vertices of a graph but also one of the best regarding number of colors used. We implemented DSATUR as a reference for speed and quality. We expected all other algorithms to be faster and produce a coloring using more colors than DSATUR.

The algorithm works by iteratively selecting one of the vertices which have a maximum saturation degree, that is the number of colors it is adjacent to, and then color the chosen vertex with the lowest possible color. A tie between multiple vertices with the same saturation degree is resolved by choosing one with a maximal degree in the uncolored subgraph. No specifics regarding the implementation is given in the original work [6].

There also exists a large array of different variations of DSATUR. Two examples being PASS [22] and SEWELL [28], which both propose a different tie breaking strategy for DSATUR. As they are both algorithms used to find an exact solution, they are expected to take far too long to produce a result on the graphs we are interested in.

**Implementation**

Our DSATUR implementation uses two levels of priority bucket queues to select the next node to color. The first level consists of one priority queue in which the saturation degree of the vertices is used as the key. In the secondary level, there is another priority queue for every bucket in the first levels queue. These secondary priority queues are maintained to contain the same elements as the corresponding bucket in the primary bucket queue, but with the vertices degree in the uncolored subgraph as the key.

DSATUR first initializes a vector `adjacentColors` to contain an empty set for every vertex. This data structure is used to update the saturation degree of the nodes in constant time. Next all nodes are inserted into the primary bucket priority queue, which has $\max\{d(v) \mid v \in V\}$ buckets, using the initial saturation degree (zero) as the key. Every vertex is also inserted into the secondary bucket queue corresponding to the first bucket of the primary bucket queue with the initial degree in the uncolored subgraph ($d(v)$) as the key. This runs in $\Theta(|V|)$ time.

Until the primary priority queue is empty, a vertex with a maximum saturation degree amongst all vertices is chosen using the primary priority queue. A tie is resolved by choosing a vertex with the largest degree in the uncolored subgraph using the secondary level of priority queues. Next, the chosen node is removed from the priority queues, colored using the lowest available color and the saturation degree of all adjacent nodes is updated. This is done by checking for every neighboring vertex if the corresponding entry in its $adjacentColors$ array is set and if not, setting the entry and moving the vertex to the corresponding bucket.

## 3.4 Genetic Algorithms

On smaller graphs genetic algorithms are amongst the most successful when it comes to coloring graphs. Especially hybrid genetic algorithms, which use an evolutionary approach combined with a local search algorithm in the "mutation" step, proved very well. Some examples include: AMACOL [20], a genetic algorithm using a proposed UIS as crossover in combination with Tabu Search described in [26], MACOL [31] and many more algorithms, some combining multiple metaheuristics include: [5], [21], [9], [25], [11], [8], [19] and [18].

Although genetic algorithms generally yield a coloring using less colors than most other heuristics, they are also known to be relatively slow compared to these other heuristics. The algorithms above are tested on graphs the size of the Second

DIMACS Implementation Challenge graphs ($125 - 1000$ nodes) only. The graphs we are interested in are much larger and genetic algorithms are known to perform very poorly on large graphs regarding running time.

## 3.5 Other Time-Expensive Metaheuristics

Other time-expensive meta heuristics that have been applied to graph coloring include IteratedLocalSearch [16], SimulatedAnnealing [15], TabuSearch as for example in Tabucol [1] and Partialcol [14], TS-Div and TS-Int [7] as well as VariableSpaceSearch [2].

We did not implement these because they are not expected to be efficient enough to handle graphs the size of those we are running our tests on.

## 3.6 Extracol

Extracol is presented by Q. Wu and J. K. Hao in "Coloring large graphs based on independent set extraction" [23]. As the title suggests, it uses a new preprocessing method proposed in the same paper to extract large independent sets from the graph. The algorithm however has only been tested on graphs the size of 1000-4000 nodes. Unfortunately, the performance declines rapidly when the graphs get larger. Because of this, we do not expect an acceptable running time on our test graphs and did not include Extracol in our experiments.

## 3.7 Other Algorithms

We also found some other algorithms who may be of interest but are not available to us ([4]) or are neither available in English nor German ([3]). Others, for example "A new graph coloring algorithm" [24] have no published test results and are described only very vague. In this particular case it is suggested that the heuristic has a polynomial running time, which would be too slow for our purposes.

# 4. Coloring Complex Networks

Social network graphs, including web graphs, have a very distinct degree distribution as can be seen in Figure 4.1. Notice that both scales in this figure are logarithmic. Social network graphs have very few nodes with a very high degree and a lot of nodes with a very small degree. Drawn with a double logarithmic scale their degree distribution exhibits a straight, descending line.

Graphs representing a social network, for example all citations from Citeseer, have an additional structural property. Inside one area of studies, for example Computer Science, there are a lot of papers who cite each other. Between different area of studies, for example between Biology and Computer Science, there are not as much citations to be expected.

In graph terms, this can be formulated as follows: A given graph has a "social-network like structure" if for a small $k \in \mathbb{N}$ the $k$-core of this graphs consists of multiple disjoint subgraphs which have no edges connecting them.

In this chapter, we propose two new algorithms which try to exploit these characteristics in order to provide a method to color large social networks using relatively few colors, while achieving a speed close to that of LDF. We also describe a parallelization of the LargestDegreeFirst algorithm.

## 4.1 kCoreFirst

The following kCoreFirst algorithm is based upon the observation regarding $k$-cores described above.

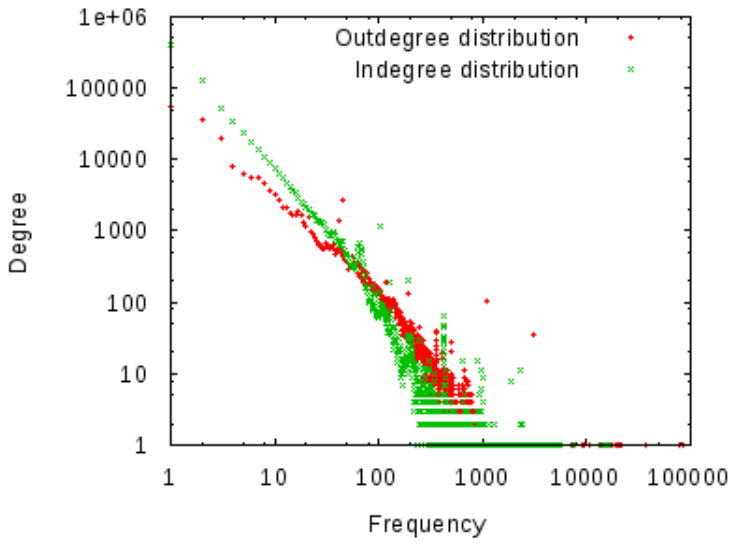


Figure 4.1: Degree Distribution of a web graph ([12])

1. Find a suitable $k$
2. Build the $k$-core
3. Color the connected components using an expensive heuristic
4. Color the nodes that have been removed when building the $k$-core in the reverse order they where removed

**Theorem:** The degeneracy of a graph is the largest number $k$ for which its $k$-core is not empty. A graph with degeneracy $k$ can be colored with at most $k+1$ colors.

**Proof:** Proof by induction over $n = |V|$: Every graph with $n \leq k+1$ is obviously $k+1$ colorable. Given a graph $G = (E, V)$ with $n > k+1$ vertices and degeneracy of at most $k$, there must exist at least one vertex $v \in V$ with $d(v) < k+1$, else the $k+1$ core of this graph would not be empty. The graph $G - v$ has a degeneracy of at most $k$ and is $k+1$ colorable by our induction hypothesis. Insert $v$ into the colored graph $G - v$ and color it with a non-adjacent color. This color is at most $k+1$ as $d(v) < k+1$. $\Rightarrow G$ is $k+1$ colorable.$\square$

The argument for the graph $G$ being a $k$ degenerate subgraph of a larger graph is analogous. There is always a vertex $v$ with $d(v) < k+1$ in the remaining graph when iteratively coloring and removing these vertices. This implies, that in Step 4 of the kCoreFirst algorithm, no new colors are used, except in the case that the already colored subgraph used less than $k+1$ colors. In this case at most $k+1$ colors are used.

A suitable $k$ has to be chosen in a way, that on one hand it is large enough that the amount of vertices in the $k$-core is crucially reduced and one the other hand is small enough that a coloring using $k+1$ colors in the best case is considered acceptable. Coloring vertices with a high degree is more difficult than coloring vertices with a small degree. The chosen subgraph should therefore encompass mostly vertices with a high degree. We use a modified version of Quickselect which selects the vertices with the highest degree in the graph until the sum of all adjacent edges of all selected vertices equals $\sim 10\%$ of all edges in the graph. Then this subgraph is colored using GreedyColor to get a very coarse estimation on how many colors are needed to color the whole graph. This number of is then chosen as $k$. Other heuristics to do this would be possible.

We have used Dsatur as the expensive heuristic in step three. Other viable options exist here, too. Step four is done using the GreedyColor heuristic described in Section 3.1 using the reversed output of the Matula & Beck run obtained when building the $k$-core. All already colored nodes are of cause removed first.

### 4.1.1 SelectXPercentEdges

The purpose of SelectXPercentEdges is to select the nodes with the highest degree until the sum of the degree of all selected nodes equals $\sim x\%$. Notice, that edges may be counted twice if both end nodes are selected.

The algorithm is coarsely based upon the canonical Quickselect. This means it first chooses a pivot element and then uses the 3-way partition also used in Quicksort to organize the input vector in three partitions. In the first partition, all nodes having a degree lower than that of the chosen pivot are located. In the second partition all nodes have the same degree as the chosen pivot. In the third partition are all nodes which have a degree larger than the pivot element. While doing this, we save the sum of all degrees in each of the three partitions.

We then decide which, if any, nodes to select and on which of the three partitions to recurse. If we would have selected less than $x\%$ of all edges if we choose all nodes in the third partition (higher degree than pivot), we do so. We then continue to choose all the nodes from the middle partition if this would still give us a total under $x\%$ of all edges. We then would recurse on the first (lower degree than pivot) partition. If either all edges included in the third or the middle partition would yield a selection encompassing more than $x\%$ of all edges, we recurse on that part. As the size of each partition has been computed beforehand, we can make this decision in constant time.

## 4.2 SelectedFirst

SelectedFirst has been implemented as a simplification of kCoreFirst. It directly uses the nodes selected in the first step as the "hard to color" subgraph and should therefore be a little bit faster and easier to implement.

1. Select nodes which are expected "hard" to color
2. Color the selected nodes using a expensive heuristic
3. Color the remaining nodes using a cheap heuristic

We used the above mentioned modified version of Quickselect to choose the nodes which are "hard" to color, then color them with the expensive heuristic Dsatur and use LDF to color the remaining nodes.

## 4.3 Parallel LDF

Parallel LDF is a parallelization of the LargestDegreeFirst heuristic. It works like LDF. First the vertices are sorted by degree in descending order, then they are colored in this order choosing the color greedily.

Parallel LDF starts with sorting the nodes by degree in parallel. The vertices are then colored in order by spawning a number of threads which in parallel each try to color one node at a time. The chunks of work are set to be the size of one node, which causes the threads to behave as follows: Thread $i$ first colors node number $i$ in the given ordering. If the thread has colored that node, it proceeds to color node number $i + \#\text{threads}$ and so on, where #threads is the number of threads spawned. The produced coloring may be incorrect as `lowestValidColor(`$v$`)` does not make a guaranty about its correctness if any of the nodes adjacent to $v$ changed color while the function is working. There is also a possibility that two threads are finished with their call to `lowestValidColor()` to two adjacent nodes, deciding on the same color and are then simultaneously both coloring their node. This would also result in an invalid coloring of these two nodes.

We must therefore check if any nodes are colored invalidly (i.e. at least one adjacent node has the same color) when all threads are finished. These nodes are then committed back to the queue of nodes still to be colored and the whole process is repeated. Checking for a invalid coloring of a node is also done in parallel. To avoid unnecessary locks, every thread has its own conflict bucket which are then, in a second step, collected sequentially. Parallelization of the collection step is not worthwhile as the number of conflicts per run is too small.

If a certain stop criterion is reached the remaining invalidly colored nodes are colored using only one thread. Currently the stop criterion $|C_i| < \alpha * |C_{i-1}| \vee |C_i| > \frac{\varepsilon * |V|}{\#\text{threads}}$ is used. $|C_i|$ denotes the number of conflicts in the $i$th run, $\alpha$ and $\varepsilon$ where both set to 0.5.

# 5. Experimental Results

In this chapter we present our experimental setup and results. Two separate experiments are described. First Matula & Beck is applied to a variety of graphs in order to select those with the desired structure described in Chapter 4. The second experiment compares the implemented algorithms, namely LargestDegreeFirst, Dsatur, kCoreFirst, SelectedFirst and Parallel LDF against each other based on running time and colors used when coloring the test graphs.

The expected result was that LDF would be by far the fastest algorithm but produce a coloration which would use the most colors. Dsatur on the other hand would take a lot longer to run than all other algorithms but provide a coloring using relatively few colors. It was expected that kCoreFirst would be slower than LDF but a lot faster than Dsatur and produce a coloring as good as or nearly as good as Dsatur. We also expected the SelectedFirst algorithm to be slightly faster than the kCoreFirst algorithm – but of course still slower than LDF – while producing a coloring which would use somewhere between the number of colors used by kCoreFirst and LDF.

## 5.1 Experimental Setup

All experiments except the ones testing only PLDF were run on two Intel Xeon 5355 CPU running at 2.66 GHz with 24 GB of memory. All tests were run sequentially. The PLDF scaling and variance tests where run on four Intel Xeon E5-4640 at 2.4 GHz and 528 GB of memory. GCC 4.6.3 [13] was used as the compiler with the `-funroll-loops` and `-O3` flags set.

## 5.2 Graphs used

### 5.2.1 First Experiment: Matula & Beck

Following graphs were tested for the desired structure described above. This was done by computing the number of nodes and disjoint connected components in each $k$-core of the graph. The Matula & Beck implementation described in Chapter 3 has been used for this.

- Some **Street Networks** used in the DIMACS 10 Challenge, namely *luxembourg* (very small), *netherlands* (small), *germany* (medium) and *europe* (huge)
- Some **Matrix** graphs also used in the DIMACS 10 Challenge, namely *af_shell9*, *ecology1*, *cage15*, *nlpkkt200* and *nlpkkt240*
- Various large **Social Network** graphs used in the DIMACS 10 Challenge: *coAuthorsDBLP*, *coAuthorCiteseer*, *coPapersDBLP* and *citationCiteseer*
- Some smaller **Social Network** graphs used in the DIMACS 10 Challenge: *jazz*, *email*, *PGPgiantcompo*, *cond-mat-200{3,5}*, *lesmis*, *dolphins*, *football*, *hep-th*, *karate*, *netscience*, *polblogs* and *polbooks*
- Various **Web** graphs used in the DIMACS 10 Challenge: *cnr-200*, *eu-2005*, *in-2004* and *uk-2002*
- **FEM** graphs used in the DIMACS 10 Challenge: *hugebubbles-000{00,10,20}*, *hugetrace-000{00,10,20}* and *hugetric-000{00,10,20}*
- A **Power grid** graph from the DIMACS 10 Challenge: *power*
- A graph describing adjacency of **Internet routers**: *caidaRouterLevel*
- Two **generated** graphs form the DIMACS 10 Challenge: *smallworld* and *G_n_pin_pout*
- A lot of instances used in the **DIMACS Graph Coloring Challenge** taken from `http://mat.gsia.cmu.edu/COLOR/instances/`
- All **Walshaw** graphs found here: `http://staffweb.cms.gre.ac.uk/~wc06/partition/`

### 5.2.2 Second Experiment: Comparing Graph Coloring Algorithms

For every graph listed above all non-empty $k$-cores were computed, plotting the number of nodes in each $k$-core and the number of disjoint connected components

over $k$. In the following, we discuss the results of the first experiment in order to conclude the selection of graphs used in the second experiment.

Those algorithms which showed a great decrease in number of nodes as $k$ got larger, preferably even for a very small ($< 10$) $k$ are interesting, as they show the structural property we are trying to exploit. The number of disjoint maximal connected components the graphs decomposed into is also of interest, as a high number would provide a leverage point for parallelization.

In the following a selection of graphs, each representing the behavior of a whole group of graphs, is shown. The blue lines indicated the number of nodes in the $k$-core of the graph. The green line indicates the number of disjoint maximal connected components the $k$-core of the graph has and the red line indicates the number of nodes in the biggest of these connected components.

### Street Networks

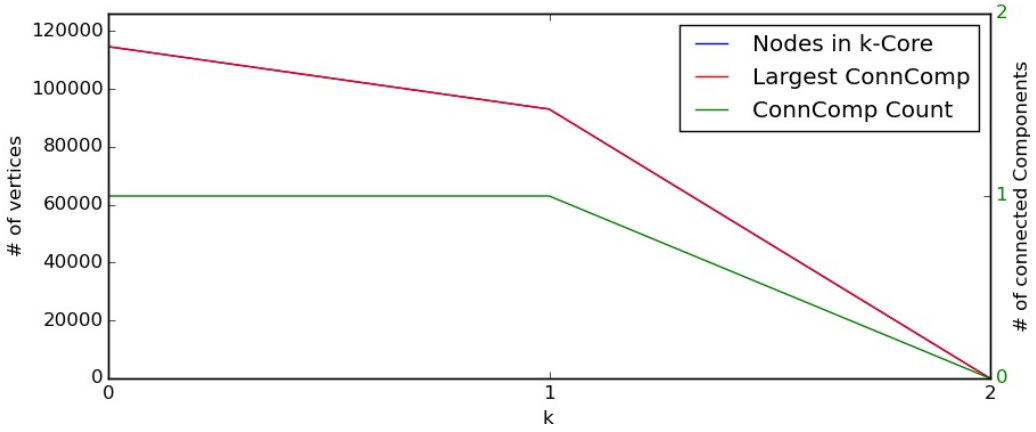


Figure 5.1: Luxembourg OSM

As seen in Figure 5.1, OSM street network graphs have a low chromatic number and can be easily colored greedily in the order given by Matula & Beck using very few colors and linear time. They are therefore not of interest to us.

### Matrix Networks

Matrix networks where also not expected to have the desired structure. As you can see in Figure 5.2, the number of nodes in the $k$-core does not decrease for $k \leq 23$ and then jumps to zero for $k = 24$. Except for *cage15*, the other Matrix Graphs behave similar. Even *cage15* is 26 colorable in linear time and therefore of no interest to us.

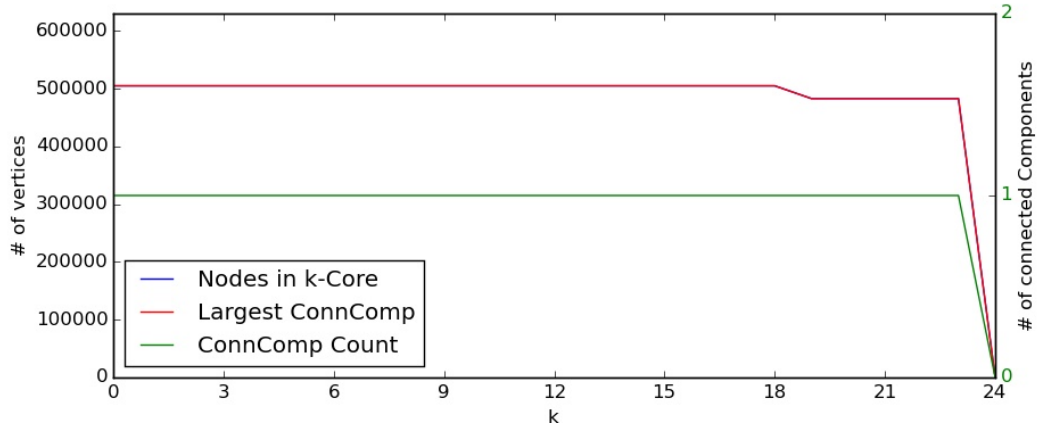


Figure 5.2: af_shell9

### Social Networks

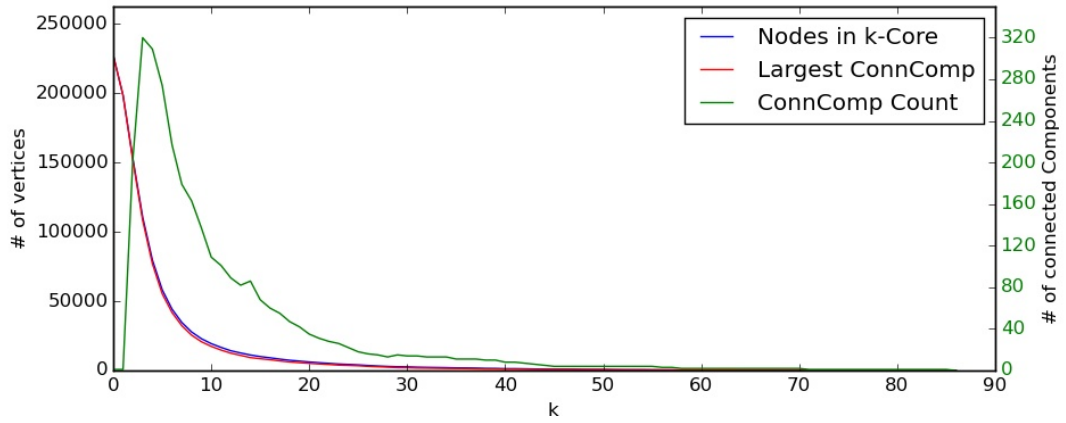


Figure 5.3: coAuthorsCiteseer

Social Networks show exactly the results we where hoping for. As it is clearly visible in the plots for *coAuthorsCiteseer* (Figure 5.3) and *coAuthorsDBLP* (Figure 5.4), the number of nodes in the $k$-core (blue) decreases rapidly for small $k$. Although the number of disjoint maximal connected components (green) would suggest to parallelize the coloring algorithm, the size of the biggest maximal connected component (red) indicates, that there is one very large connected component and many very small ones. Simple parallelization by assigning one thread per disjoint connected component would therefore be useless as many threads would be finished very quickly and one thread would take very long to color its subgraph.

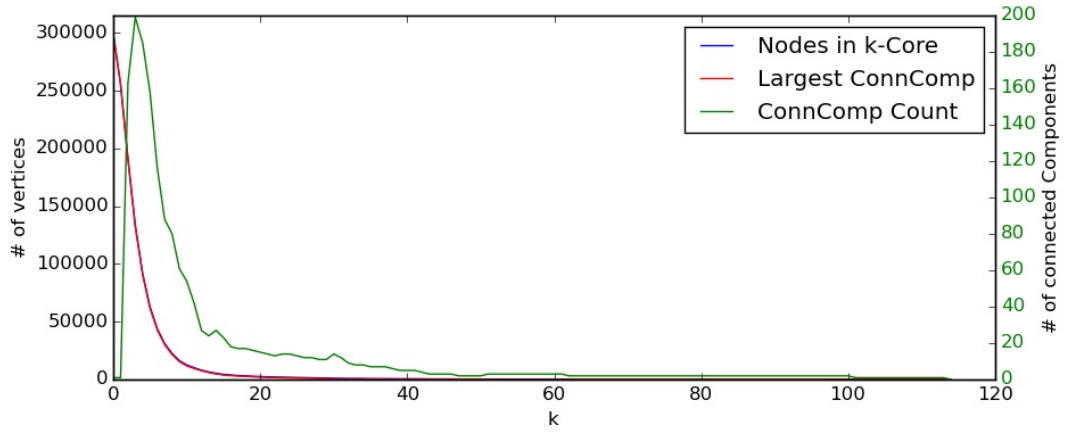


Figure 5.4: coAuthorsDBLP

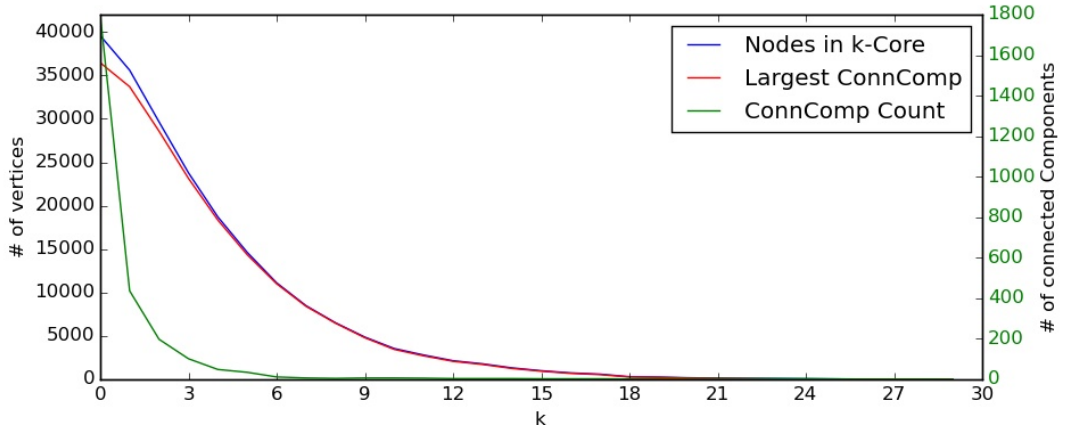


Figure 5.5: cond-mat-2005

As seen in Figure 5.5, *cond-mat-2005* behaves in the same way. The only difference here is, that the number of connected components starts high, in contrast to starting low in the other two shown plots, and then drops. This can be explained by *cond-mat-2005* being a graph representing collaborations between condensed matter scientists, *cond-mat-2005* could therefore be a typical subgraph of *coAuthorsCiteseer* or *coAuthrosDBLP*.

All other tested social network graphs behaved the same way as described above. Some of them, for example *jazz*, *email*, *dolphins* and *karate* where to small to be of interest and were therefore excluded from the tests.

We decided to include *coAuthorsCiteseer*, *coAuthorsDBLP*, *cond-mat-2003*, *cond-mat-2005*, *citationCiteseer*, *coPapersDBLP*, *hep-th*, *astro-ph* and *PGPgiantcompo*

into our tests.

### Net Graphs

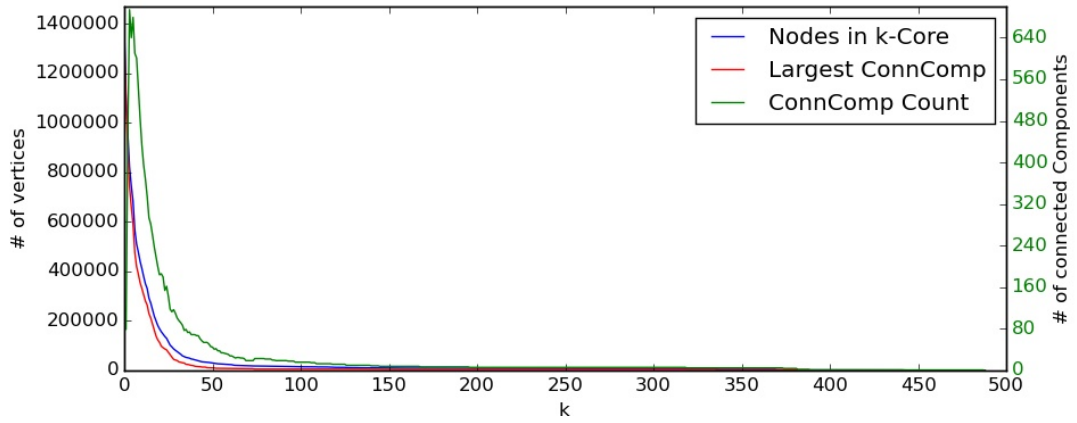


Figure 5.6: in-2004

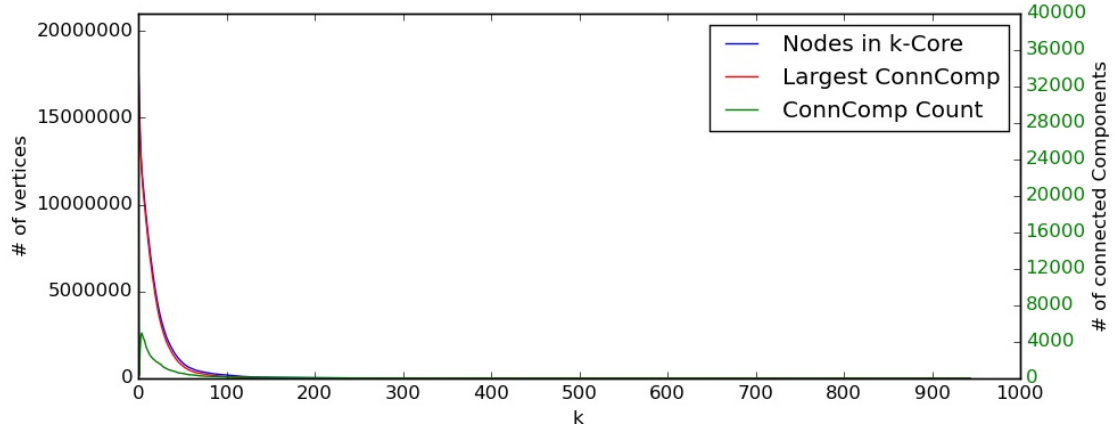


Figure 5.7: uk-2002

Net graphs show a behavior very similar to social networks. There is a significant decrease in nodes in a $k$-core even for small $k$ and for all tested graphs the number of disjoint connected components spikes for a small $k$ and then quickly approaches 1 again. As in social network graphs, naive parallelization is not possible, as there is one large connected component and many small ones. It has to be noted, that most net graphs we tested are significantly larger than the tested social network graphs.

We included *uk-2002*, *in-2004*, *cnr-2000* and *eu-2005* in our tests. *uk-2002* was only used when testing PLDF, as running DSATUR on it would take too long.

### FEM graphs

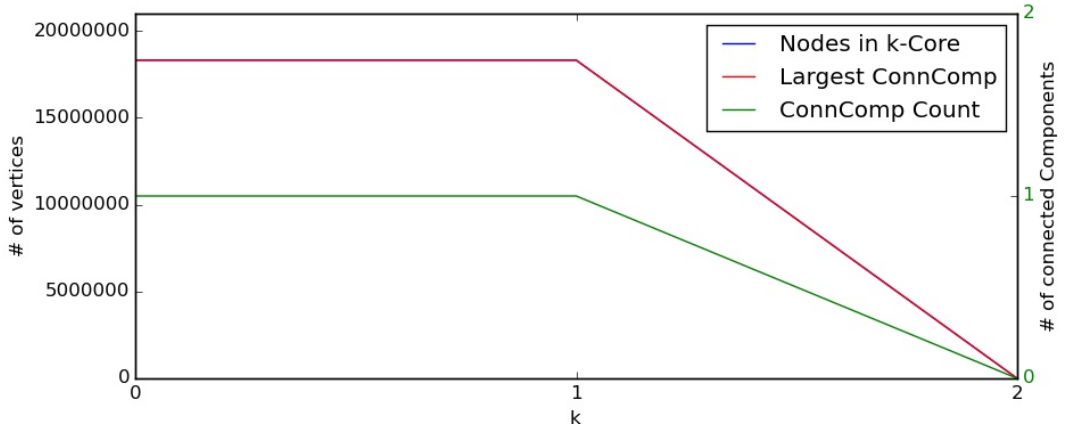


Figure 5.8: hugebubbles-00000.jpg

As expected FEM graphs are three colorable in linear time and therefore not of interest to us. An example can be seen in Figure 5.8.

### Power grid graphs

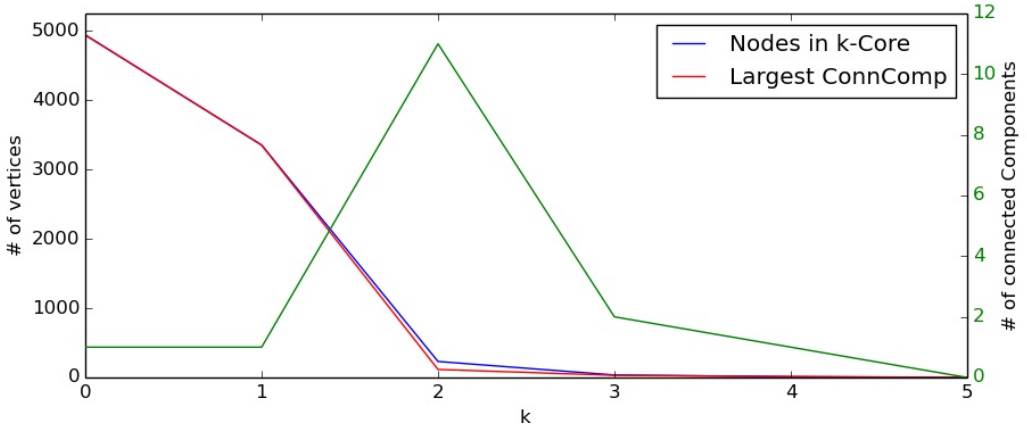


Figure 5.9: power

We only had access to one graph representing a power grid. Although it seems to be suitable in terms of a small number of nodes in a $k$-core for a small $k$, the graph only has 5000 nodes and is therefore to small to be of interest.

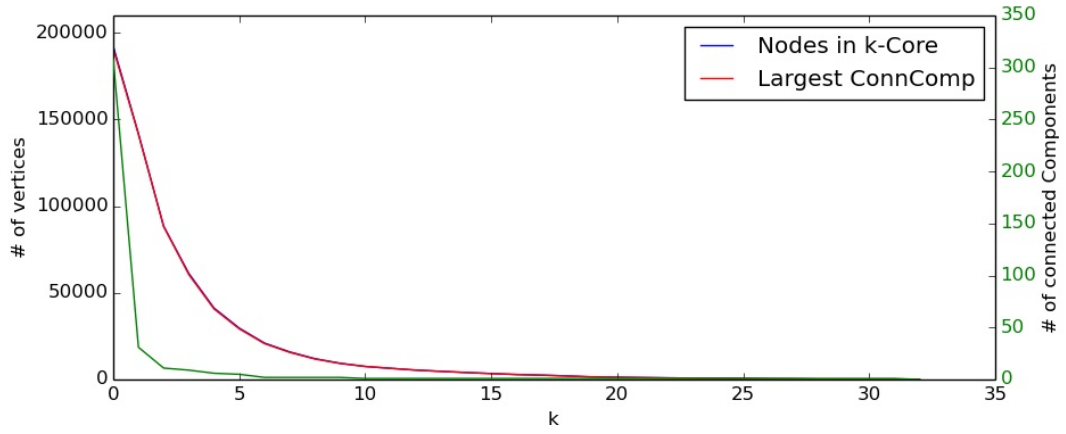


Figure 5.10: caidaRouterLevel

### Internet Router networks

We also had only one graph available which described a network of Internet routers. But opposed to *power* it qualifies for our tests. You can clearly see a decrease of nodes in the 10-core compared to the 0-core. Therefore we decided to include the graph *caidaRouterLevel* into our tests.

### Other graphs

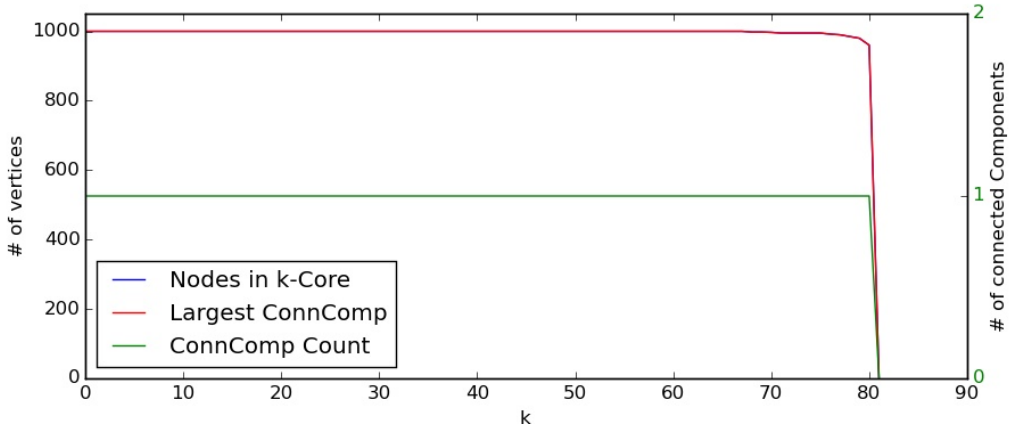


Figure 5.11: DSJC1000.1

All other tested graphs, especially the ones designed to be difficult to color, behaved like the shown DSJC1000.1 in Figure 5.11. You can clearly see that there is not much of a decrease in number of nodes in the $k$-core until a relatively high $k$. These graphs are therefore not of interest to us.

## 5.3 Results

### 5.3.1 Running time

As shown in Figure 5.12, the running time of the algorithms behaved largely as we had expected. Dsatur was by far the slowest algorithm with running times 3–140 times larger than that of the kCoreFirst algorithm. As expected: The larger the graph, the more of an advantage kCoreFirst had in running time over Dsatur.

The SelectedFirst algorithm performed better than kCoreFirst on some graphs and worse on others. It seems to perform better on webgraphs as seen for *in-2004*, *eu-2005* and *cnr-2000* but worse on large social network graphs as seen for *coAuthors** and *coPapersDBLP*; *citationCiteseer* being an exception. All these graphs have between 300000 and 600000 nodes. On smaller social network graphs like *cond-mat-**, *hep-th*, *astro-ph* and *PGPgiantcompo*, which have between 8000 and 40000 nodes, SelectedFirst performed comparable to kCoreFirst.

Dsatur was set to 1 in the plot and all other running times where normalized using $t_{\text{algo}}/t_{\text{Dsatur}}$. As expected kCoreFirst run faster than Dsatur and slower than LDF on nearly all graphs.

LDF was expected to have the fastest running time amongst the non parallel algorithms. It exhibits this behavior on all graphs except *coAuthors** and *coPaperDBLP*, that is all large social network graphs except *citationCiteseer*.

PLDF is the only algorithm tested which uses parallelism and therefore was expected to be the fastest running algorithm. It is on all but the two smallest graphs – *PGPgiantcompo* and *hep-th*, having 12000 and 8000 nodes respectively. On *astro-ph*, which has 15000 nodes, the performance of *PLDF* is only marginally better than LDF. PLDF was run on the same machine as the other algorithms for this test and therefor the number of threads was limited to 8.

### 5.3.2 Colors used

The numbers of colors used have been given relative to the number of colors Dsatur used. Against all expectations, only very slight differences in colors used could be observed. Amongst all sequential algorithms, Dsatur delivered the best colorings for every graph, as expected. But only for two graphs, *cnr-2000* and *caidaRouterLevel*, Dsatur did better than LDF.

As a result, the both algorithms combining Dsatur – SelectedFirst and kCoreFirst – did not produce a coloring having a significant advantage over the

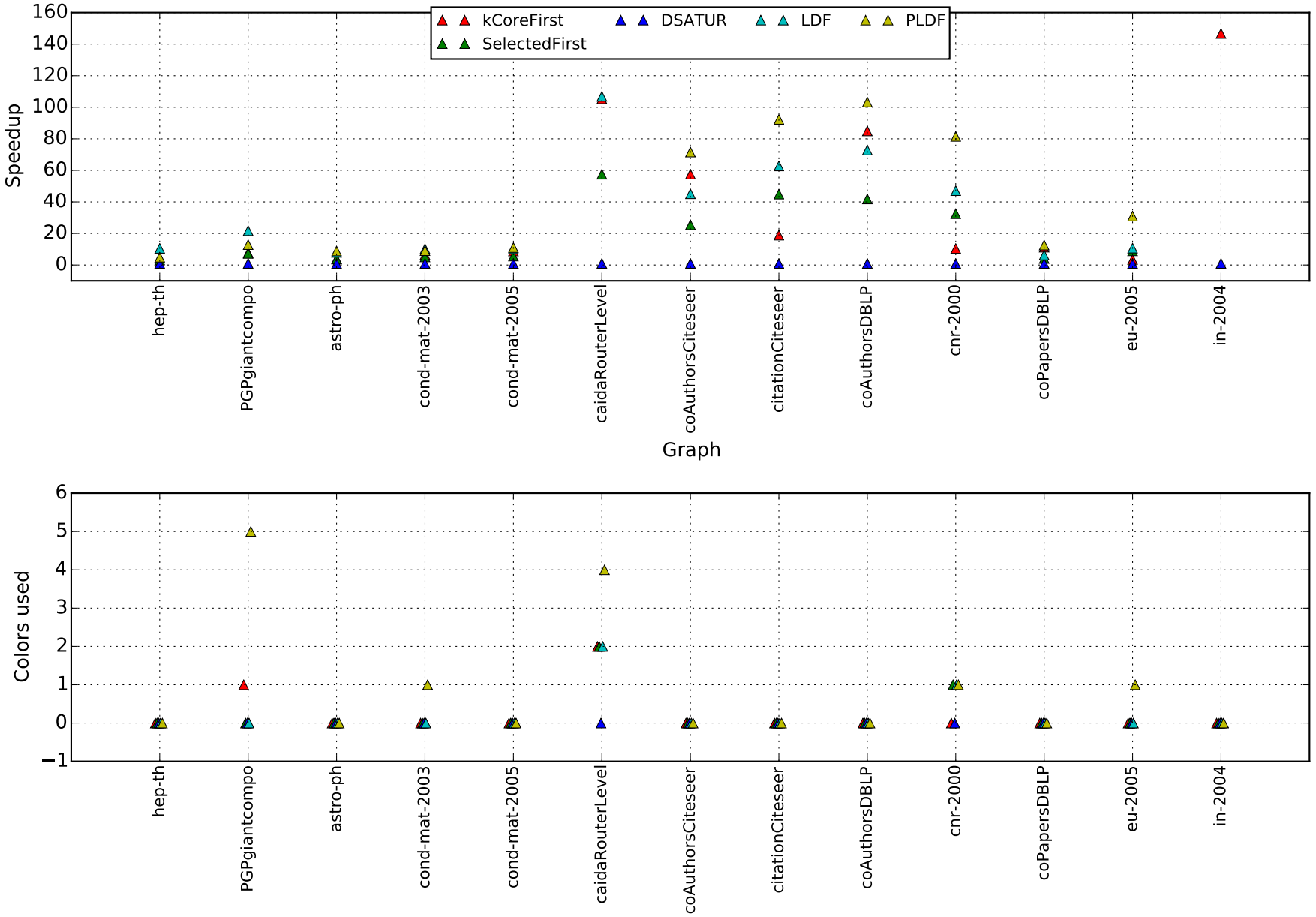


Figure 5.12: Comparison of algorithm run time and colors used. The graphs are sorted by degree in descending order. The not-shown values for the speedup on *in-2004* are: SelectedFirst: 210, LDF: 290, PLDF: 900

one produced by LDF. On most of the graphs, all sequential algorithms used up the same number of colors.

PLDF uses the same number of colors as LDF for 9 out of 13 graphs. On the remaining four graphs – *PGPgiantcompo*, *caidaRouterLevel*, *cond-mat-2003* and *eu-2005* – PLDF used one to four colors more than LDF. These four graphs do not seem to belong to any logical subgroup of the tested graphs.

## 5.4 PLDF Evaluation

PLDFs running time varies only slightly between runs. Six runs on each of the 13 graphs were done. Except for two instances, the variance of running time was below 8%. In these two runs the running time differed by 20% and 30% from the mean. On the largest tested graph, *uk-2002*, the total running time was $\sim$ 14.5–14.8 seconds. Every other graph ran in under one second.

The colors used by PLDF in different runs vary by one or zero except on *caidaRouterLevel*, where the run using the most colors used six colors more than the run using the least colors. For ten out of 14 graphs, the different test runs produced a coloring using exactly the same number of colors. On four more graphs, the colors used differed only by one.

As seen in Figure 5.13, PLDF scales not so well for most of the tested graphs. The speedup however does improve with the size of the graph. On *uk-2002* the speedup when using 8 threads is slightly above 4. The problem is mainly the large number of arising conflicts when two threads color neighboring graphs simultaneously.

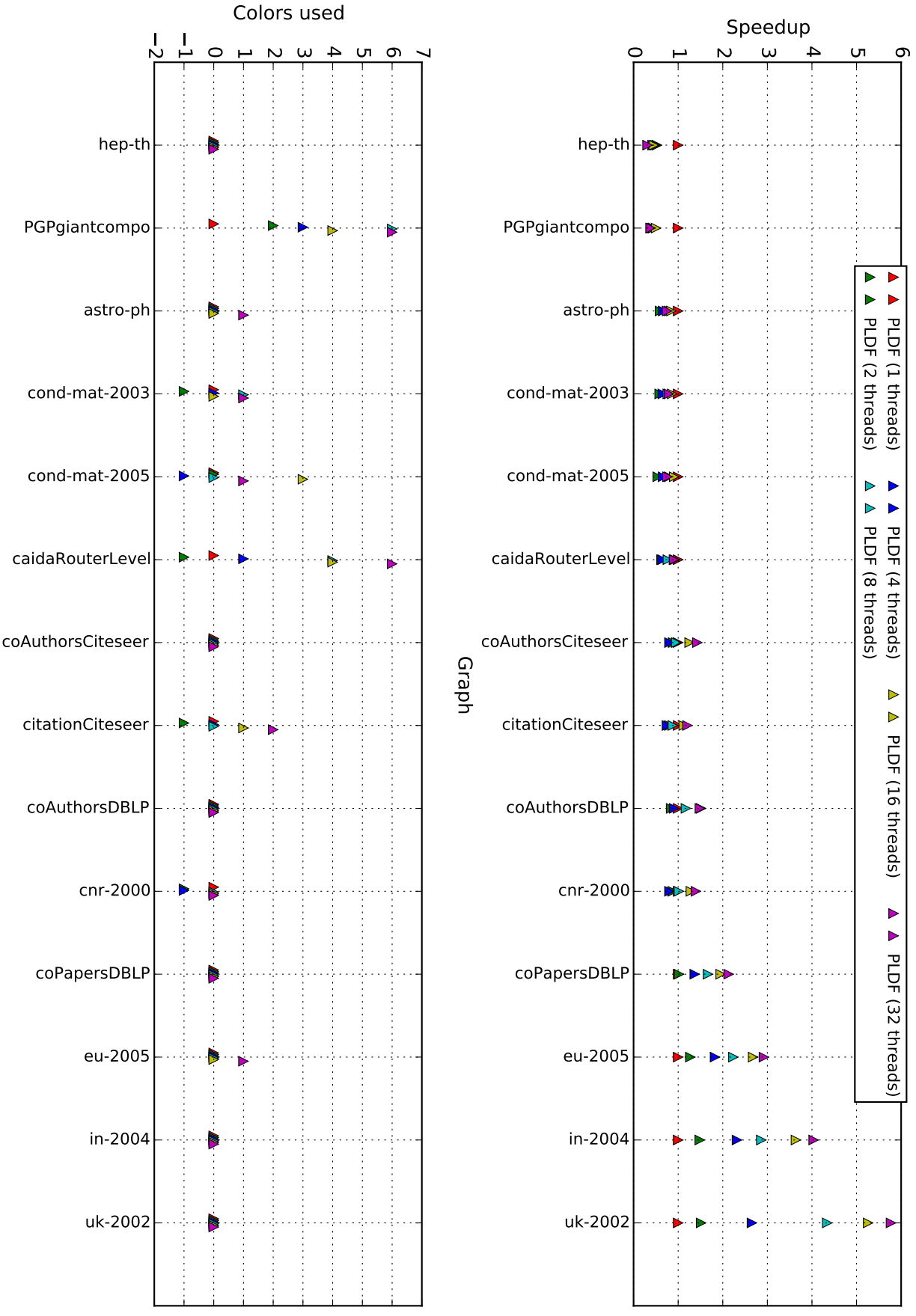


Figure 5.13: PLDF scalability, graphs sorted ascending by number of nodes

# 6. Conclusion

The results are rather astonishing. Nearly every prediction we made about the number of colors used by the different algorithms was wrong. LargestDegreeFirst scored best on all but two graphs, tying with the much more time expensive Dsatur. As a result, both proposed algorithms did also not do noticeably better than LargestDegreeFirst.

One possible explanation would be, that nodes present in the chosen $k$-core are likely to have a very high degree and therefore are colored first by LDF, too. This would however not explain why Dsatur does not do better than LDF when run on the complete graph. This could be explained by LDF and Dsatur choosing a very similar order in which to color the vertices. This could be caused by the special structure of the graphs we did our test on. It could also be the case that these graphs are generally very easy to color. Further tests would be necessary here.

One very interesting aspect of our results is, that LDF was actually slower than kCoreFirst on three graphs: *coAuthorsCiteseer*, *coAuthorsDBLP* and *coPapersDBLP*, the large social network graphs. Further investigation could yield some very interesting results here, too.

For someone looking to color social network or net graphs, the best recommendation seems to be LDF for smaller and PLDF for larger graphs. We determined the value 0.5 for both $\alpha$ and $\varepsilon$ to yield good results after rudimentary testing.

# Bibliography


[1] A. Hertz and D. de Werra. Using Tabu Search techniques for graph coloring. *Springer — Computing*, 39:345–351, 1987.

[2] A. Hertz, M. Plumettaz and N. Zufferey. Variable Space Search for graph coloring. *Discrete Applied Mathematics*, 156:2551–2560, 2008.

[3] A. Tehrani. Un algorithme de coloration. *Cahiers du Centre d'Études de Recherche Opérationnelle*, 17:395–398, 1975.

[4] C. Morgenstern. Distributed coloration neighborhood search. *Proceedings of the Second DIMACS Implementation Challenge*, 26:335–358, 1996.

[5] D. A. Fotakis, S. D. Likothanassis and S. K. Stefanakos. *An Evolutionary Annealing Approach to Graph Coloring*. Applications of Evolutionary Computing, 2001.

[6] D. Brélaz. New Methods to Color the Vertices of a Graph. *Communications of ACM*, 22:251–256, 1979.

[7] D. C. Porumbel, J. K. Hao and P. Kuntz. A search space 'cartography' for guiding graph coloring heuristics. *Computers & Operations Research*, 37:769–778, 2010.

[8] D. Porumbel, J. K. Hao and P. Kuntz. An Evolutionary Approach with Diversity Guarantee and Well-Informed Grouping Recombination for Graph Coloring. *Computers & Operations Research*, 37:1822–1832, 2010.

[9] D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. *Operations Research*, 39:378–406, 1991.

[10] D. W. Matula, L. L. Beck. Smallest-last Ordering and Clustering and Graph Coloring Algorithms. *Communications of ACM*, 30:417–427, 1983.

[11] E. Malaguti, M. Monaci and P. Toth. A Metaheuristic Approach for the Vertex Coloring Problem. *INFORMS Journal on Computing*, 20:302, 2008.

[12] Eötvös University. Erdös WebGraph (`http://web-graph.org/index.php/download`).

[13] GCC Team. GNU Compiler Collection (`http://gcc.gnu.org/`).

[14] I. Blöchliger and N. Zufferey. A Graph Coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & Operations Research*, 35:960–975, 2008.

[15] M. Chams, A. Hertz and D. de Werra. Some experiments with simulated annealing for coloring graphs. *European Journal of Operational Research*, 32:260–266, 1987.

[16] M. Chiarandini and T. Stützle. An application of Iterated Local Search to Graph Coloring Problem. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pages 112–125, 2002.

[17] M. Trick. Graph Coloring Instances `http://mat.gsia.cmu.edu/COLOR/instances.html`.

[18] O. Titiloye and A. Crispin. Graph Coloring with a Distributed Hybrid Quantum Annealing Algorithm. In *Agent and Multi-Agent Systems: Technologies and Applications*, volume 6682, pages 553–562. 2011.

[19] Olawale Titiloye and Alan Crispin. Quantum annealing of the graph coloring problem. *Discrete Optimization*, 8:376–384, 2011.

[20] P. Galinier, A. Hertz and N. Zufferey. An adaptive memory algorithm for the k-coloring problem. *Discrete Applied Mathematics*, 156:267–279, 2008.

[21] P. Galinier and J. K. Hao. Hybrid Evolutionary Algorithms for Graph Coloring. *Journal of Combinatorial Optimization*, 3:379–397, 1999.

[22] P. S. Segundo. A new DSATUR-based algorithm for exact vertex coloring. *Computers & Operations Research*, 39:1724–1733, 2012.

[23] Q. Wu and J. K. Hao. Coloring large graphs based on independent set extraction. *Computers & Operations Research*, 39:283–290, 2012.

[24] R. D. Dutton and R. C. Brigham. A New Graph Colouring Algorithm. *Computation Journal*, pages 85–86, 1981.

[25] R. Dorne and J. K. Hao. A new genetic local search algorithm for graph coloring. In *Parallel Problem Solving from Nature*, volume 1498, pages 745–754. 1998.

[26] R. Dorne, J. K. Hao, P. Scientifique and G. Besse. A New Genetic Local Search Algorithm for Graph Coloring. In *Parallel Problem Solving from Nature - PPSN V, 5th International Conference, volume 1498 of LNCS*, pages 745–754, 1998.

[27] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. 1972.

[28] Sewell E. An improved algorithm for exact graph coloring. *Proceedings of the Second DIMACS Implementation Challenge*, 26:359—73, 1996.

[29] T. R. Jensen, B. Toft. *Graph Coloring Problems, Wiley-Interscience, New York*. 1995.

[30] W. Hasenplaugh, T. Kaler, T. B. Schardl and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 166–177.

[31] Z.Lü and J. K. Hao. A memetic algorithm for graph coloring. *European Journal of Operational Research*, 203:241–250, 2010.