View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KITopen

Solving the Graph Coloring Problem with Cooperative Local Search

Bachelor Thesis of

Guangping Li

At the Department of Informatics
Institute of Theoretical informatics, Algorithmics II

Advisors: Dr. Tomáš Balyo
Prof. Dr. Peter Sanders

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 15th December 2016

Abstract

Tabucol is a local search algorithm to determine whether the vertices of an undirected graph can be colored with k colors, such that no two vertices connected by an edge have the same color. This thesis presents an algorithm that solves the graph coloring problem with parallel Tabucol searches. A hypothesis was experimentally evaluated that sharing information among the agents will improve the performance of the parallel search. In this paper, we introduce a new matrix data structure, which counts the repeated times of one change. This statistic matrix can help recognizing long-term cycling in the local search. The sharing of this matrix among the agents can bring further improvement to our algorithm.

Zusammenfassung

Diese Arbeit präsentiert einen Algorithmus, der eine gültige k-Knotenfärbung für ungerichtete Graph mit minimal k sucht. Der Algorithmus macht sich dabei einen lokalen Algorithmus Tabucol zu Nutzung. Die Leistung mit verschiedenen Strategien wurde anhand von mehreren Experimenten evaluiert und miteinander verglichen. Dabei zeigt der parallel laufende Algorithmus mit geeignetem Informationsaustausch unter den Agenten eine Verbesserung in Bezug auf Leistung aus. Wir stellen in diesem Papier eine neue Matrix Data Struktur vor, die die Wiederholung einer lokalen Änderung zählt. Diese Statistik Matrix kann helfen, lange Zyklen in der lokalen Suche zu erkennen. Das Teilen dieser Matrix unter den Agenten kann eine weitere Verbesserung unseres Algorithmus bringen.

Contents

1 Introduction

1.1 Problem/Motivation

The ***graph coloring problem*** (***GCP***) [1] is an NP-complete problem [2]. The problem is to assign colors to certain elements of a graph subject to certain constraints.

The vertex coloring problem is the most common GCP. The goal is to color the vertices of an undirected graph such that no two adjacent vertices share the same color. There are many different ways to color a graph. In most cases, one wants to color the graph with the smallest number of colors. This number is called the ***chromatic number*** of the graph.

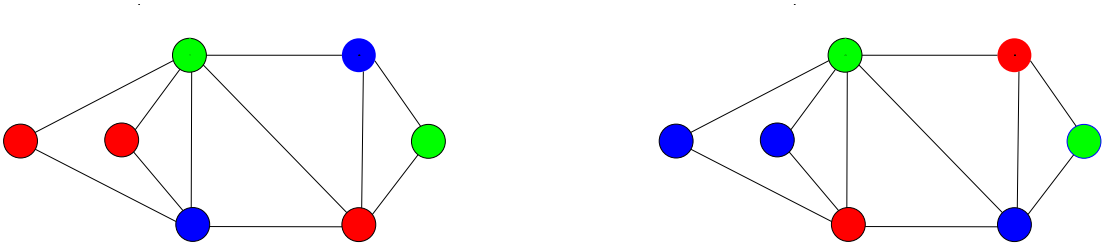


Figure 1: Two 3-colorings of one graph. The chromatic number of this graph is 3.

The vertex coloring problem has many applications like computer register allocation [3, 4] and printed circuit board testing [5]. Two examples in our daily life are as follows.

1. **Timetable and scheduling** [6, 7]

An Example of this application is to make an exam schedule. We suppose that several exams are going to be scheduled in a university. How to ensure the students do not have different exams in the same time slot? This problem can be seen as a vertex coloring problem, where the exams are represented by vertices in an undirected graph. Two exams involving the same student are connected with an edge. Here the minimum color number corresponds to a minimum number of time slots needed for all the exams.

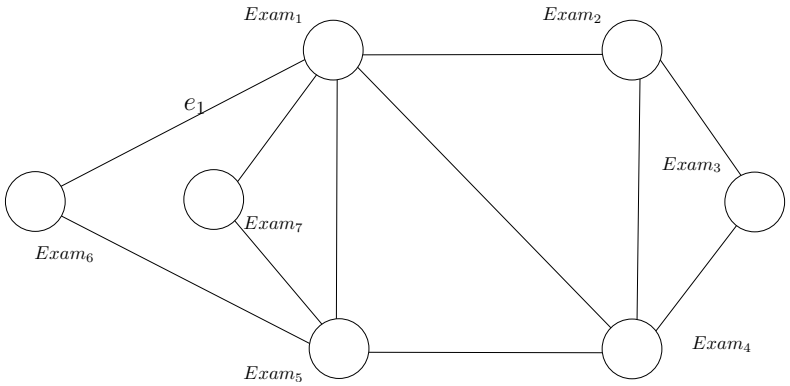


Figure 2: An example of the application in scheduling. There are 7 exams in this example. The edge e_1 means some students take the 1st exam as well as the 6th exam.

2. **Mobile Radio Frequency Assignment** [8]

Assigning frequencies to radio senders is also an example of the vertex coloring problem. The constraint is that senders received by the same location must use different frequencies. In a graph where the senders correspond to vertices, two senders have the same receiving location

are connected with one edge. The aim is to color the senders with a minimum number of colors, which represent different frequencies.

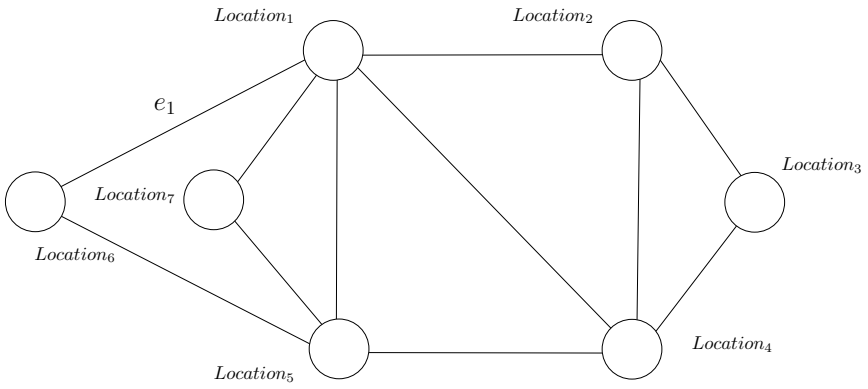


Figure 3: An example of the application in frequency assignment. There are 7 locations in this example. The edge e_1 means a sender can be received by the 1st location as well as the 6th location.

1.2 Content

The Graph coloring problem, as a well-known NP-complete problem [2], has received a great deal of attention and different search methods have been developed [9, 10]. This thesis concentrates on solving the vertex coloring problem with one of the first local search algorithms Tabucol. Tabucol was proposed in 1987 by Hertz and de Werra [11]. As a very popular and well performing local search algorithm, Tabucol is often used as a subroutine in hybrid algorithms for solving GCP. This paper presents a GCP-solver which runs Tabucol iteratively on the same graph instance. In the first part of this thesis, different techniques to improve this algorithm are discussed. In the second part, we want to search the potential benefit of cooperative search, in which different agents run the same algorithm in parallel and exchange information in the search process. In this paper, Different kinds of cooperative work among agents are evaluated and compared. Some techniques turned out to be more efficient than the original search.

2 Preliminaries

2.1 Definitions and Notations

A ***set*** is a container of unique elements. A set of 3 objects a, b, c is written as $\{a, b, c\}$. The size of a set is the number of elements in the set.
A ***graph*** $G = (V, E)$ [12] is a structure consisting of a finite set V of vertices and a finite edge set $E \subseteq V \times V$. Throughout the paper, we use $n = |V|, m = |E|$ if no other definitions are given. By v_i we denote the ith element in V and by e_i the ith element in E. Based on the forms of the edges, there are two types of graphs: undirected graph and directed graph. In a ***directed graph***, the edge set contains ordered pairs of vertices, called arrows or directed edges. An edge $e = (u, v) \in E$ represents a connection from node u to v. An ***undirected graph*** is a graph in which edges have no orientation. The edge $e = \{u, v\}$ represents a connection between the vertices u and v. It is identical to the edge $e' = \{v, u\}$. Two vertices are incident when they are

connected by an edge. When an edge connects a vertex with itself, this edge is called a loop in the graph.

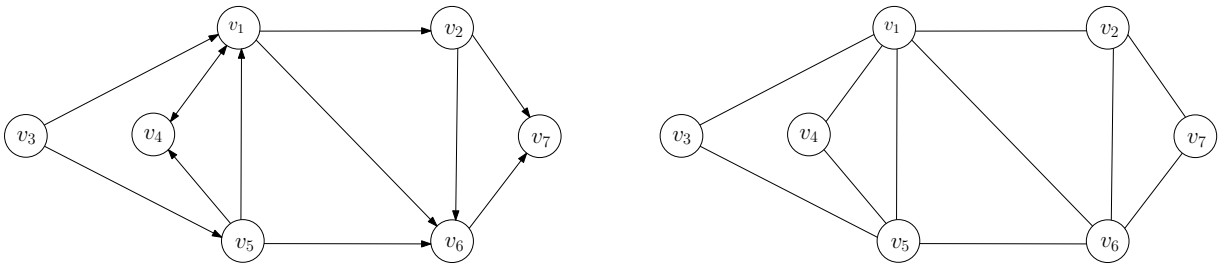


Figure 4: The figure above left shows a directed graph. There are two edges between v_1 and v_4. The edge (v_1, v_4) is represented by an arrow from v_1 to v_4. The edge (v_4, v_1) is represented by an arrow towards v_1. An undirected graph is shown in the figure above right. There is one edge $\{v_1, v_4\}$ between v_1 and v_4. The edge $\{v_1, v_4\}$ is represented by a line connecting v_1 and v_4. The edge $\{v_1, v_4\}$ is equal to the edge $\{v_4, v_1\}$.

The graph coloring problem (***GCP***) deals with the assignment of labels called "colors" to elements of a graph subject to certain constraints.
When used without any qualification, a coloring of a graph is almost always a proper vertex coloring, A vertex coloring of a graph is a function c: $V \to \{1...k\}$. The coloring with k colors is called a ***k-coloring***. The value $c(v_i)$ of a vertex v_i is called the color or the index of the color of the vertex v_i. A coloring is a ***legal coloring*** when no two adjacent vertices share the same color. Otherwise, when an edge connects two same-colored vertices in a coloring, the edge is called a ***conflicting edge***. A coloring with at least one conflicting edge is an ***illegal coloring***. The ***conflict number*** $f(c)$ of a coloring c is the number of conflicting edges in the coloring. In other words, $f(c) = \sum |E_i|$, $E_i = \{(v, w) \in E, c(v) = c(w) = i\}$. The ***k-GCP*** problem is to determine whether a legal k-coloring exists for the graph or not. The ***GCP*** problem is to determine the smallest k, such that the graph can be colored using k colors without conflicts. This lower bound k is called the ***chromatic number*** of G, denoted by $\chi(G)$.

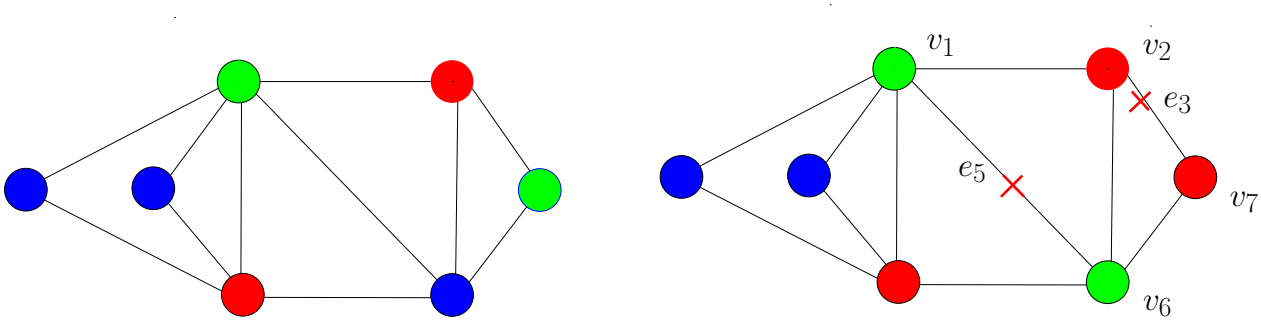


Figure 5: The figure left above shows a 3-coloring example. This is a legal coloring without conflicting edges. The chromatic number of the graph $\chi(G)$ is thus 3. The figure right above is an illegal 3-coloring with conflict number 2. A Conflicting edge e_5 connects two green-colored vertices v_1 and v_6. The conflicting edge e_3 connects two red-colored vertices v_2 and v_7.

Neighboring coloring
A neighboring coloring c' of a coloring c is a coloring differs from c in the color of one vertex v. $c(w) = c'(w)$ for $w \neq v$; $c'(v) \neq c(v)$.

Local search
A local search starts with an initial illegal coloring. To reach a neighboring coloring, the local search makes local changes to its current coloring iteratively, hence the name local search. Usually, each coloring has more than one neighbors. The decision which neighbor will be reached in next step depends on some criterion. Local search is widely used in hard problems such as the traveling salesman problem [13] and the boolean satisfiability problem [14].

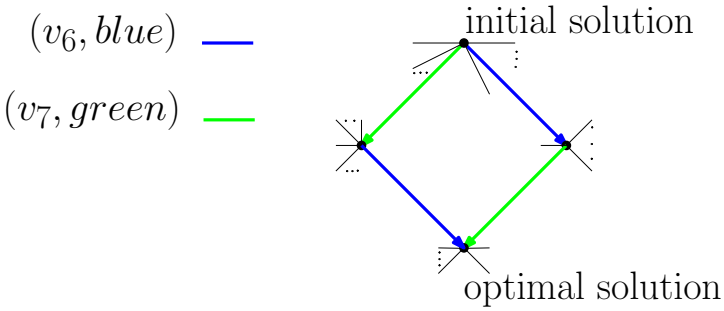
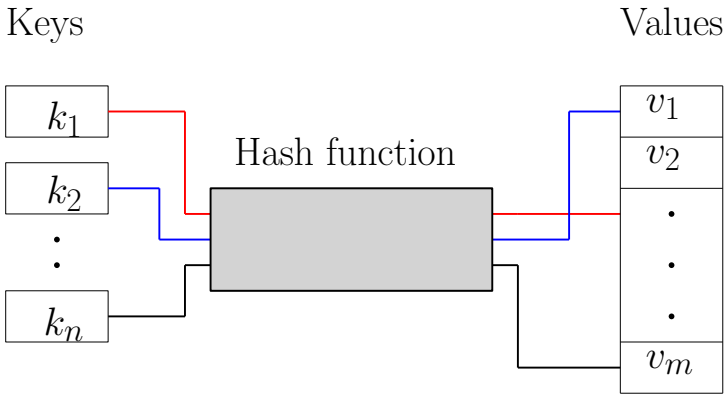


Figure 6: Figure 10 is an example of a local search in coloring problem. Colorings are represented by nodes in a net. The search changes one color iteratively to improve the coloring to have fewer conflicts. The initial coloring of this Example is the right coloring in Figure 5. The optimal coloring is the coloring shown left in Figure 5. Blue edges represent changing v_6 to blue (read from the upper node to the lower node). Green edges represent changing v_7 to green.

Tabu search
Local search methods move in a neighborhood and have a tendency to get stuck in suboptimal regions. Tabu search created by Fred W. Glover in 1986 [15] and formalized in 1989 [16][17]. The search trace is recorded in the process. The recently reached neighboring colorings are marked as tabu colorings. The tabu colorings will not be touched in the further search to discourage getting stuck in a region.

Hash table/map [18]
A hash table or a hash map is a directory, which maps keys to values. With the help of a hash function, the hash table is advantageous in respect of speed. Specifically, the overall complexity of common operations such as search, insertion or deletion is $O(1)$.



A hash function is similar to a directory, which maps keys to stored values.

Figure 7: Hash table

Matrix
A $p \times q$ matrix is a function $f_A : \{1...p\} \times \{1...q\} \to R$. A Matrix is normally represented by

an ordered scheme A with p rows and q columns. The value of the ith row and the jth column in a scheme A is denoted by A_{ij} and is equal to the function value $f_A(i, j)$.

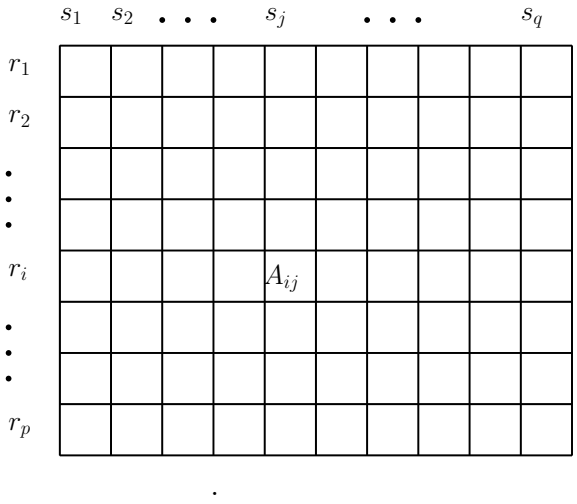


Figure 8: A $p \times q$ matrix

Queue
The queue is an abstract data structure with two operations: enqueue where an element is pushed in the collection and dequeue where the element added in the collection earliest and not removed yet is removed from the queue. The elements come in the queue in one end and come off from another end. This policy is called FIFO (first in-first out).

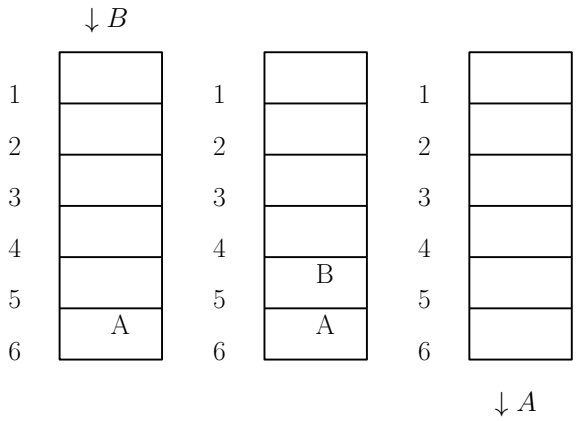


Figure 9: Enqueue and dequeue under FIFO policy

2.2 The algorithms for comparison

To evaluate the performance of our algorithm, three other algorithms for solving the GCP are used for a comparison with our algorithm (see Experiment 9 in 5.5.9). The three algorithms

for comparison are DSATUR, Trick's algorithm, and PASS.

DSATUR [19]
DSATUR is a sequential vertex coloring algorithm to color the vertices in turn until each vertex is colored. This greedy coloring algorithm maximizes saturation degree that is the number of colors assigned to its adjacent vertices. If multiple vertices with maximum saturation degree exist, the one with the maximal degree in the uncolored subgraph is chosen. The DSATUR algorithm assigns the smallest possible color to the chosen vertex in each step. There are some variations of DSATUR. Two examples are Trick's algorithm and PASS.

Trick's algorithm [20]
A clique of the graph is a set of mutually adjacent vertices. Naturally, to color, a clique of k vertices needs at least k different colors. The Trick algorithm makes use of this relationship of cliques and coloring. The algorithm finds and colors the large cliques in the graph and then applies the DSATUR algorithm on the uncolored subgraph.

PASS [21]
The tie breaking strategy of the DSATUR algorithm is to choose the vertex with the maximum degree if multiple vertices with maximum saturation degree exist. The PASS algorithm is a DSATUR-based algorithm with a different tiebreaking strategy. The tiebreaking strategy aims to color that candidate vertex first, which has the least number of admissible colors. This strategy is based on the observation, that the vertices with small admissible colors in the current step are likely to require a new color in the further search.

We cannot find any source code for Tabucol-based algorithms. We chose these tree DSATUR-based exact coloring algorithms for comparison because these algorithms are much employed with their simplicity and efficiency.

2.3 The Tabucol algorithm

Tabucol was introduced in 1987 by Hertz and de Werra [11]. Since then, some modifications are introduced to the original tabu search. The version I use in this paper is based on the paper by Galinier P [22].
The algorithm Tabucol is a tabu search to solve the k-GCP. Formally, for a Graph $G = (V, E)$, Tabucol wants to find a function $c : V \to \{1..k\}$ with constraint: $\forall \{u, v\} \in E, c(u) \neq c(v)$. An evaluation function f measures the number of conflicting edges in a coloring. In other words, the Tabucol is to determine, whether a k-coloring c exists with $f(c) = 0$. The local search will start from an initial k-coloring c. Changing the color of one vertex v to color i ($c(v) \neq i \wedge \exists$ $w \in V, c(w) = i$), is called ***one-step move*** $[v, i]$. The search space (the neighborhood) consists of the colorings which can be reached by a one-step move. A coloring c' resulting from $[v, i]$ is denoted by $c + [v, i]$:

$c(w) = c'(w)$ for $w \neq v$;
$c'(v) = i$;

$\Gamma(c, c') = f(c) - f(c')$ measures the improvement of c' to the current coloring c. [1] There are $(k - 1) \times n$ neighbors of a coloring. The neighbor coloring with the highest improvement will be chosen as the next move. The algorithm performs one-step moves iteratively and stops as

[1] The value of function Γ can be negative, which indicates an increase of the conflicting number.

soon as $f(c) = 0$, which means c is a legal coloring.
To avoid short-term cycling, recently performed moves are marked as forbidden moves for a given duration. The duration of the tabu status depends on the conflict number and two parameters L and α. More precisely, the duration is $f(c) \times \alpha + L$. Galinier suggests to choose L randomly in [0, 9] and use $\alpha = 0.6$. [23]

The pseudo code of ***Tabucol*** is shown below

Algorithm 1: Algorithm Tabucol

`input` : A Graph $G = \{V, E\}$, an integer $k > 0$
`parameter`: L, α, $Timeout$
`output` : Coloring c

1 Build a random k-coloring c';[a]
2 $c = c'$;
3 $i = 0$;
4 `while` ($f(c) \neq 0 \wedge$ *Timeout does not occur*) `do`
5 Evaluate all permitted neighbors of c with function Γ;
6 Choose neighbor c'' with maximum $\Gamma(c, c'')$;
7 Mark the corresponding one-step move [v,i] of c'' as a forbidden move with duration $L + \alpha \times f(c)$;
8 Change $c(v) = i$;

[a] This simplest option of building the initial coloring is suggested by Galinier. There is no tangible advantage to using a greedy heuristic as he said. [22]

3 Solving GCP by Tabucol

As mentioned in section 2 (See 2.1), the ***GCP*** is to find the smallest coloring of a graph. This problem can be solved by solving ***k-GCP*** iteratively, in which the minimum size k is found as shown in the following flow chart:

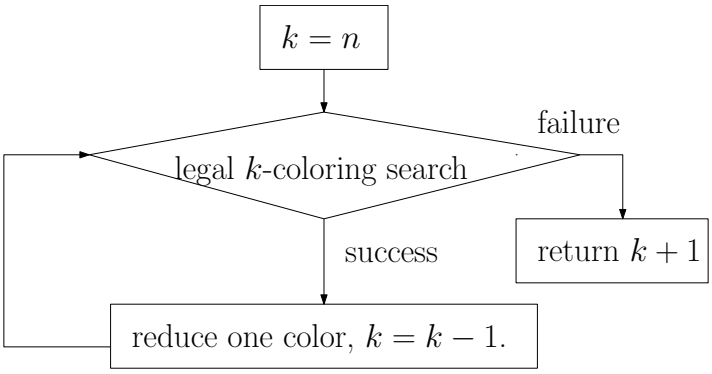


Figure 10: GCP-solver

Step 1: Build an initial coloring
A legal n-coloring is obviously available. In our GCP-solver, the initial coloring is c: $v_i \rightarrow i$. A possible alternative is to build the initial coloring randomly. In Experiment 1 in section 5 (see 5.5.1) we compare these two alternatives. Our suggestion shows some advantages in performance.

Step 2: Solve k-GCP using Tabucol

Step 3: Reduce one color
A $(k-1)$-coloring will be built by reducing the least used color in the previous k-coloring. If some color is chosen for reduction, all the nodes in this color will be colored with other remaining colors randomly. If only a few nodes are involved in the color reduction, the resulted $(k-1)$-coloring is more likely to be legal or illegal with few conflicting edges.

In next section, the data structure of our GCP-solver is introduced.

3.1 Data structures

Tabu Map and Tabu Queue
In the process of tabu search, all the legal candidates should be considered in the order of decreasing conflict numbers. To avoid using the forbidden candidates in the search process, a data structure should provide whether a candidate is a tabu or not. The forbidden moves are stored in a map, called *Tabu Map*. Here a one-step move [v, i] is represented by an ordered pair (v, i).
After its duration of tabu status, a forbidden move will be freed. In the meantime, a one-step move will be marked as a tabu move after one search step. Accordingly, a queue called *Tabu Queue* will be used to update the forbidden moves stored in the *Tabu Map*. The size of the *Tabu Queue* is $L + f(n) \times \alpha$.[2] When the color of a vertex v is changed, $(v, c(v))$ is recorded in the *Tabu Map* and also enqueued in the *Tabu Queue*. When the *Tabu Queue* is full, extra forbidden moves will be popped from the other end of the *Tabu Queue* and will be deleted in the *Tabu Map*, which means the moves are freed.

When the color of a vertex v is changed:
1. Record $(v, c(v))$ in the *Tabu Map*.
2. Insert $(v, c(v))$ in the back of the *Tabu Queue*.
3. If the queue is full (size of queue $> L + f(n) \times \alpha$), we remove the oldest pair (u, i).
4. Delete (u, i) in *Tabu Map*. go to step 3

An alternative data structure is using an $n \times k$ Matrix T [22]. T_{ij} stores the index of iteration, in which the one-step move $[v_i, j]$ is marked as a tabu move. To determine whether a one-step move $[v, i]$ is a tabu move or not, T_{ij} is checked in constant time. If $T_{ij} \geq$ *currentIter* - L - $\alpha \times f(n)$, this move is permitted [3]. Otherwise, the move is a tabu move.

Solution Matrix
Local search is a search where only small changes are made in each search step. In our situation, only one node is involved in getting to a neighbor of the current coloring. The most time in Tabucol is spent for finding the best one-step move in the complete neighborhood. To reuse the calculated results, Galinier uses a matrix to record the information about the neighborhood [22]. It is the most important data structure in our implementation. This matrix is denoted as *Solution Matrix M*. The matrix M evaluates the candidate moves of the current k-coloring c. If $c(v_j) = i$, M_{ij} is the number of conflicting edges incident to v_j in the current solution. $\frac{\sum_{j=1}^{n} M_{c(v_j)j}}{2}$ is the conflict number of solution c; If $c(v_j) \neq i$, M_{ij} is the number of conflicts involving v_j in a neighbor coloring c' of c:

$c'(v_j) = i$.
$c'(v_q) = c(v_q), q \neq i, q \in \{1..n\}$.

$\Gamma(j, i) = M_{c(v_j)j} - M_{ij}$ evaluates the improvement of the move $[v_j, i]$ in constant time. To find the best one-step move, all neighbors must be evaluated by the evaluation function Γ. To find the best one-step move, $O(k \times n)$ time will be spent scanning the *Solution Matrix*. This matrix is filled at the beginning based on the initial solution and constantly changed. If the chosen

[2] The size of the *Tabu Queue* is adapted to the conflict number.
[3] CurrentIter is the index of the current iteration.

one-step move is $[v_i, j]$, which means changing the color of v_i from current color $c(v_i)$ to j, some entities in matrix M must be updated. More precisely, for each incident vertex v_w of v_i, $M_{c(v_i)w}$ will be decreased by one and M_{jw} will be increased by one.

Data Structure	Relevant Operations	Functions
Tabu Map	insert an element erase an element search for an element	store and update tabu moves
Tabu Queue	enqueue an element dequeue en element	free tabu moves after a duration
Solution Matrix C	change values of few entities constantly	record information of neighbors

Table 1: Overview of used data structure

Here is a pseudo code of our GCP-solver with the data structures introduced above:

Algorithm 2: original GCP-solver with data structures

`input` : A Graph in DIMACS standard format
`parameter`: L, α
`output` : the minimum size k, a k-coloring c

```
1 Build the initial coloring c with c(v_i) = i and the corresponding Solution Matrix M;
2 k = n;
3 while (k > 1) do
4 |   c' = reduceOneColor(c);
5 |   if (Tabucol(c') succeed (See Algorithm 3)) then
6 |   |   c = c';
7 |   |   k = k − 1;
8 |   else
9 |   |   return k, c
```

3.2 Improvement through randomly generated solution

Generally, finding a $(k+1)$-coloring is easier than finding a k-coloring. In first iterations of our GCP-solver, a legal k-coloring can be quickly generated by reducing the least used color in the $(k+1)$-coloring to other remaining colors. The Tabucol searches take more time gradually and the most time spent in the process is in finding a legal c-coloring and trying to find a $(c-1)$-coloring, where c is the optimal solution. The idea here comes from the observation of time spent in Tabucol searches. It seems that the solution loses its potential in the process of reducing colors iteratively. So it should be helpful to use a new and perhaps more potential coloring. In our GCP-solver, we replace the current illegal solution occasionally by a new randomly generated coloring of the same size. This randomly generated coloring will be used for the further search.

Experiment 2 (see 5.5.2) describes the details of this suggestion and evaluates its performance. It shows Improvement in 53% of our benchmarks.

Algorithm 3: Tabucol with data structures

```
1  while (f(c') = Σ_{i=1}^{n} M_{c'(v_i),i} ≠ 0 ∧ Timeout does not occur) do
2      max = −n;
3      for (j < k, ) do
4          for (i < n ∧ j ≠ c'(i)) do
5              if ((i, j) not in Tabu Map ∧ M_{c'(i),i} − M_{ji} > max) then
6                  max = M_{c'(i),i} − M_{ji};
7                  (changedColor, changedVertex) = (j, i);
8      oldColor = c'(changedVertex);
9      c'(changedVertex) = changedColor;
10     for (i ∈ {1..n}) do
11         if ( v_i is incident to changedVertex) then
12             M_{oldcolor,v_i} = M_{oldcolor,v_i} − 1;
13             M_{changedColor,v_i} = M_{changedColor,v_i} + 1;
14     insert (oldColor, changedVertex) in Tabu Map;
15     push (oldColor, changedVertex) in Tabu Queue;
16     while (size(Tabu Queue) > L + α × f(c') ) do
17         Tabu Queue pops a move (u,v);
18         delete (u,v) in Tabu Map;
19 if (f(c') = 0) then
20     return "success"
21 else
22     return "failure"
```

3.3 Improvement through changing solution matrix traverse direction

When searching for the best move in our implementation, the maximum element in the conflict matrix must be found. In the pseudo code above (see Algorithm 3), the matrix is traversed row by row. If more than one candidate with maximum improvement exists, the first one is chosen as the next step. This matrix can also be traversed column by column. Through Experiment 3 (see 5.5.3), we found a GCP-solver with traversal by column usually gets a better result.

3.4 Improvement through statistic matrix

The Tabucol algorithm uses a tabu list to avoid short-term cycling. When the cycling is long (longer than the size of tabu queue), the loop of one-step moves will not be recognized in the search process. An Improvement in our implement is to use a matrix, called *statistic matrix S*. The S_{ij} represents how many times a one-step move $[v_i, j]$ was chosen as the next step. When the search is stuck in one long-term loop, the involved entities in the statistic matrix are increased constantly. If more than one candidate with the highest improvements exists, the candidate with the smallest statistic value will be chosen in the next step. We can also use the entity S_{ij} to determine the possibility of choosing the next move randomly (see Experiment 4 in 5.5.4).

4 Our Parallel Algorithm

With different kinds of cooperative search, this section presents 4 parallel local search algorithms. The algorithm is based on our GCP algorithm introduced in section 3 and the agents cooperate by sharing different kinds of information. The Performances were compared by experiments in section 5.

4.1 1st Approach: The pure portfolio approach

The local search in section 3 uses Tabucol as a subroutine to solve the graph coloring problem. The result of the algorithm is a legal coloring of the graph of minimum size. In the algorithms, some parameters like L, α, the search directions (see section 3.3) and whether a statistic matrix is introduced (see section 3.4) affect the one-step moves chosen by the search. The performance of the algorithm is different with different parameter combinations. In the pure portfolio version of our algorithm, the agents run the GCP solver with different parameter combinations. After collecting the solutions found by each agent, the search takes the coloring of the minimum size as the result. This approach improved the performance compared to the parallel GCP solver, in which all agents run with the same parameters (see experiment 5). For each graph, there is one parameter combination that is most suitable in the aspect of the size of tabu list and the search path of this combination is better than others. So trying different parameter combinations will improve the performance.

4.2 2nd Approach: Forced color reducing

This approach is based on the pure portfolio approach with different parameter combinations as shown above. This approach is called GCP-solver with forced color reducing (FCR). As its name suggests, the agents share the minimum size found by all agents. Because of different parameter combinations, some agents are "luckier" and decrease the size of coloring more quickly. Suppose that one agent has already found a k-coloring, where the other agents still make an effort to determine whether the graph is $(k+i)$-colorable (integer $i > 0$). In this approach, the lucky agent will broadcast this message. With this notification, all agents confirm that the graph is at least a k-colorable graph. Then the agents in the process of searching for a $(k+i)$ $(i \geq 0)$ coloring will abandon the current search and search for a legal $(k-1)$ coloring since a legal k-coloring is already found. This approach saves a lot of effort and makes the search space larger with different implementation in our experiment (see experiment 6 in section 5.5.6), the strategy we use is to reduce the least used $(i+1)$ colors in the current $(k+i)$ coloring. This generated $(k-1)$ coloring is used as the initial coloring in the Tabucol search for a legal $(k-1)$ coloring. In this paper, this strategy is called forced color reducing (FCR).

4.3 3rd Approach: Tabu sharing

One character of tabu search is using a tabu list to record the search path to avoid short-term cycling. In previous parallel approaches, each agent manages and uses one tabu list of its own. The idea behind this approach is to share the "traps" of local search loops. So sharing the tabu list should be able to bring an improvement in the cooperation of agents. This approach is called parallel GCP-solver with tabu sharing. We did an experiment to test the performance of this

approach (see experiment 7 in section 5.5.7). It shows, however, a worse performance compared to the original parallel GCP-solver. According to our observation in the implementation, we list 3 possible reasons:
1. Not all tabu moves are traps. Some critical moves which are necessary to get the optimal result are only explored by a part of threads while other threads threat them as tabu moves. The threads missing these critical steps would never contribute to the algorithm.
2. Best candidates are forbidden. In hard graphs, it is normal to have several best candidate moves in the current solution. Some candidates involve different parts in the graph. The solution will be optimized to the greatest extent by going over the candidate moves and the order of the moves have no effect on the optimization. Figure 11 is an example with two best candidates. The current coloring c has two best candidate moves m_1, m_2. The candidate m_1 is (v_1, white). The m_2 is (v_2, black). The optimal coloring of the example graph will be reached by making the move m_1 then m_2 or inversely. Imagine that we have two agents a_1 and a_2 work with the same coloring c. The agent a_1 inspects m_1 and pushes m_1 in shared tabu list. After that, the agent a_2 makes the move m_2 and marks m_2 as a tabu move. In this deadlock case, the agent a_1 cannot take m_2 even it brings the best improvement on the current solution. Similarly, agent a_2 recognize m_1 as a tabu move and does a compromise with other suboptimal moves.

Figure 11: An example with two moves m_1 and m_2. The candidate m_1 is (v_1, white). The m_2 is (v_2, black).

3. When a move is taken earlier by one agent, it will be ignored by other agents. The differences in parameters make the colorings of agents more different and thus the shared tabu moves may not be meaningful to other agents. In some cases, the tabus from other agents disturb the choice of next moves.

4.4 4th Approach: Statistic sharing

After the analysis of the failures in the 3rd Approach with tabu sharing, we came up with the approach with statistic matrix sharing. The intention of this sharing is to use the statistic matrix to recognize the real traps and warn other agents when one agent is stuck in one loop. The Tabucol search in agents follows the scheme below:
1. The agents use one common statistic matrix. The statistic matrix counts the times of moves chosen by one agent.
2. When two best candidates exist in one solution, the move with smaller statistic matrix value will be chosen.
3. When only one best candidate exists with a very large statistic value, the search will choose another candidate randomly.
This approach demonstrates a big improvement. (see Experiment 8)

5 Evaluation

The single-threaded experiments were run on computers that had four AMD(R) Opteron(R) processors 6168 (1.9 Ghz with 12 cores) and 256GB RAM. The computers ran the 64-bit version of Ubuntu 12.04. The multi-threaded experiments were run on fat nodes InstitutsClusterII (IC2) at Steinbuch Centre for Computing (SCC) of KIT. IC2 is a distributed memory parallel computer with 480 16-way so-called thin compute nodes and 5 32-way so-called fat compute nodes. The thin nodes are equipped with 16 cores, 64 GB main memory, whereas the fat nodes are equipped with 32 cores, 512 GB main memory [24].

5.1 DIMACS standard format

All the graphs used in experiments are in the DIMACS standard format [25]. This format is a widely used format to test and compare graph coloring algorithms. A DIMACS file contains the description of an instance using three types of lines[4]:

1. Comment line: Comment lines give information about the graph for human readers, like the author of the file or related works. A comment line starts with a lower-case character c and will be ignored by programs:

c # *this is an example of the comment line* #

2. Problem line: The problem line appears exactly once in each DIMACS format file. The problem line is signified by a lower-case character p. For a graph $G = (V, E)$, the problem line in its DIMACS file is:

p edge |V| |E|

3. Edge Descriptor: An edge $\{u, v\}$ in the graph is described in an edge Descriptor:

e u v

5.2 Benchmarks

The graphs used in our experiments are from the DIMACS benchmark collection [27, 28]. In the following experiments, 68 graphs are used. Some names of the graphs contain generation involved information:

dsjcX.Y and dsjrX.Y: Graphs generated by Johnson et.al [10]. X in the file name denotes the number of vertices. The probability that two nodes are incident is given by Y.
flatX_K: Graphs generated by J Culberson. The graphs are generated by partitioning its vertices in K nearly same sized sets and adding edges which connect vertices in different sets. So the chromatic number is theoretically smaller or equal to K.
le450_K: Graphs with 450 vertices and chromatic number K [6].
c*: Huge graphs with more than 4 million edges.

[4]Only unweighted undirected simple graphs are tested in our experiments. For other descriptors and details of the DIMACS format, see [26].

```
c This is a DIMACS file of the graph in Figure 6
p edge 7 11
e 1 3
e 1 2
e 2 7
e 2 6
e 1 6
e 1 4
e 1 5
e 4 5
e 3 5
e 5 6
e 6 7
```

Figure 12: A DIMACS file example of the graph in Figure 5

latin_square_10 and school*: A latin square graph (and class scheduling graphs respectivelly) generated by Gary Lewandowski for the second Dimacs challenge.
r*.X: Random graphs. The suffix "c" denotes the complement of a graph.
queenX_Y: Graphs translated from Stanford GraphBase with ID: gunion(board(X,Y,0,0,-1,0,0),board(X,Y,0,0,-2,0,0),0,0).
milesX: Graphs translated from Stanford GraphBase with ID:miles(128,0,0,0,X,127,0).
jean: Graphs translated from Stanford GraphBase with ID: book(jean,80,0,1,356,0,0,0).
fpsol2.i.*, inithx.i.*, mulsol.i.*, zeroin.i.*: Graph generated from a register problem based on real code.
brockX_*: Graphs with X nodes generated by Mark Brockington and Joe Culberson.

In our experiments, the graphs are divided into 4 classes according to their size and complexity for the graph coloring problem [5]. For huge graphs in the first class, the timeout is set to 20 minutes. For the second class, the timeout is 4 minutes. For the third class 2 minutes and for fourth class 1 minute.

class 1 (4 graphs): c2000.5, r1000.5, dsjc1000.9, c4000.5.

class 2 (3 graphs): dsjc1000.5, r1000.1c, latin_square_10.

class 3 (53 graphs):
miles250, jean, queen8_8, le450_5a, le450_5b, queen9_9, le450_5c, le450_5b, queen8_12, queen10_10, queen11_11, queen12_12, queen13_13, queen14_14, queen15_15, le450_15b, miles500, queen16_16, le450_15c, le450_25a, le450_25b, le450_15d, dsjc1000.1, school1, school1_nsh, zeroin.i.2, zeroin.i.3, fpsol2.i.3, fpsol2.i.2, inithx.i.2, inithx.i.3, miles750, mulsol.i.2, mulsol.i.3, mulsol.i.4, mulsol.i.5, miles1000, mulsol.i.1, zeroin.i.1, inithx.i.1, dsjc500.5, fpsol2.i.1, miles1500, brock400_1, brock400_2, brock400_3, dsjr500.1c, flat1000_60_0, flat1000_50_0, flat1000_76_0, brock800_1, brock800_2, brock800_4, dsjr500.5.col.

[5] In the experiments, the performance of the algorithms after a time interval was compared.

class 4 (7 graphs): dsjc500.1, le450_25c, le450_25d, dsjc250.5, flat300_28_0, r250.5, dsjc500.9.

5.3 Used plots and tables

Different plots and tables are used to illustrate the results of the following experiments.

Comparison Table
See Table 3 for example.
A comparison table shows the different results of algorithms. The first column contains the names of graphs. The fields of a comparison table in the following columns corresponds to the coloring sizes found with an algorithm.

Scatter Plot
See Figure 13 for an example.
A scatter plot compares the results of two algorithms. Similar to the comparison table, only the graphs with a difference in two algorithms are shown in the scatter plot. The x-axis shows the color sizes for an algorithm, denoted by x-axis algorithm. The vertical axis to the left shows the color sizes for y-axis algorithm. A graph with color size u in x-axis algorithm and color size v in y-axis algorithm corresponds to a mark (u, v). The line $x = y$ divides the plot into two parts. The marks in the upper part $(x < y)$ correspond to graphs with better results for the x-axis algorithm. The opposite corresponds to better results for the y-axis algorithm. In Figure 13, the part above the diagonal line contains more marks than the lower part, which means x-axis algorithm is better than the y-axis algorithm.

Cactus Plot
See Figure 14 for an example.
In a cactus plot, the problems are indexed in an ascending order of color size. The y-axis shows the result sizes of the graph. Each algorithm corresponds to a curve in different colors. The point (u, v) on a curve means a v-coloring is found in the corresponding algorithm for the uth graph.
Advantage Plot
See Figure 15 for an example.
An Advantage plot shows the advantage of algorithms to an comparison algorithm. The y-axis gives the ordered percentage difference. A relative difference upper the line $x = 0$ corresponds to an advantage of the algorithm in the corresponding graph. The opposite is true for an advantage of the comparison algorithm.

5.4 Automatic parameter optimization

The parameter combinations used in the experiments are generated with the help of the algorithm parameter optimization tool SMAC [29] (sequential model-based algorithm configuration). SMAC ran our algorithms on our instances (class 4 as training instances, class 1 as test instances.) using different parameter combinations and seeds. In this simulation process, the performance of different combinations was evaluated. With the help of SMAC, 32 optimal parameter combinations were found. The 32 parameter combinations used in our experiments are as follows:

Table 2: parameter combinations

Index	L	α	Initialization	Replace	Traverse	Statistic
1	9	0.38	Node-Index	true	Column	true
2	1	0.77	Node-Index	true	Column	true
3	11	0.90	Node-Index	true	Column	true
4	17	0.59	Random	true	Column	true
5	18	0.42	Node-Index	false	Column	false
6	4	0.92	Node-Index	true	Column	true
7	16	0.76	Node-Index	false	Row	false
8	17	0.47	Node-Index	false	Column	false
9	2	0.60	Node-Index	true	Column	false
10	2	0.54	Node-Index	false	Column	true
11	5	0.46	Random	true	Column	true
12	11	0.63	Random	true	Column	true
13	7	0.83	Node-Index	true	Column	true
14	8	0.98	Node-Index	false	Row	true
15	18	0.58	Node-Index	true	Column	false
16	13	0.90	Node-Index	false	Column	true
17	20	0.56	Node-Index	true	Column	false
18	10	0.95	Node-Index	true	Column	true
19	15	0.55	Node-Index	true	Row	true
20	17	0.39	Node-Index	true	Column	true
21	18	0.52	Node-Index	false	Column	true
22	11	0.32	Node-Index	true	Column	true
23	15	0.62	Node-Index	false	Column	true
24	6	0.94	Random	true	Column	true
25	9	0.94	Node-Index	false	Column	false
26	12	0.96	Node-Index	true	Column	true
27	16	0.58	Node-Index	false	Column	true
28	9	0.45	Node-Index	false	Column	true
29	19	0.95	Node-Index	true	Column	true
30	18	0.31	Node-Index	true	Column	false
31	6	0.50	Node-Index	false	Column	false
32	15	0.93	Node-Index	false	Column	false

For a single thread algorithm, we use the first parameter combination. In a multi-threaded algorithm, the ith thread uses the ith parameter combination. The parameter L and α determine the size of the tabu List with $L + f(n) \times \alpha$. For the initial solution generation, two choices exist: randomly generated initialization and node-index initialization (see section 3). The solution matrix is traversed row by row or column by column (see section 3.3). The combinations with "statistic = true" use the statistic matrix while searching and participating statistic sharing (see section 4.4).

5.5 Experiments

In experiment 1 to experiment 4, we compare strategies of the single-thread GCP-solver shown in section 3. The results are in table 3. The original solver is an implementation of algorithm 2 with the data structures shown in section 3.1. Experiment 1 compares this original Solver, which uses a randomly generated initial solution with a GCP-solver with the strategy "Node-Index initialization" (see section 3). Experiment 2 compares our original solver with the solver with the strategy "replacement" (see section 3.2). Experiment 3 compares the original solver, which traverses the solution matrix row by row with a solver with the column traversal (see section 3.3). Experiment 4 proves the advantage of adding the data structure statistic matrix (see section 3.4). The results of these experiments are summarized in table 3.
Experiments 5 to 8 are about the parallel GCP solver. The experiments perform with 1 core, 2 cores, 4 cores, 8 cores, 16 cores and 32 cores. The experiments use our single-thread GCP solver with Node-index initialization and the 3 advantageous strategies we found (see sections 3.2, 3.3, 3.4) for comparisons. Experiment 5 compares this single-thread GCP solver with multi-threaded solver with different parameter combinations, denoted as the pure portfolio GCP-solver. Experiments 6 to 8 compare our pure portfolio GCP-solver with the parallel solver with cooperation. Experiment 6 tests the performance of the parallel solver with minimum color size sharing among agents. Experiment 7 shows the solver with shared tabu list, which turns out to be a failed attempt. In experiment 8, the parallel solver with shared statistic matrix is compared with the pure portfolio solver.
In experiment 9, we compare our parallel GCP-solver with all found advantageous coopeartion with three DSATUR-based algorithms (see section 2.2).

5.5.1 Experiment 1: Random initialization vs Node-index initialization

Experiment 1 compares two different strategies of initialization in our GCP-solver. Our suggestion is to use c: $v_i \rightarrow i$ as the initial solution. Another alternative is to build a coloring randomly. In table 3, the column "Original solver" uses the random initialization. The column "NodeIndex" contains the results with the Node-index initialization. In 29 of the 68 graphs, there is a performance difference between two initializations. 24 graphs get a better result with our version. 5 graphs get a better result with a random initial solution.

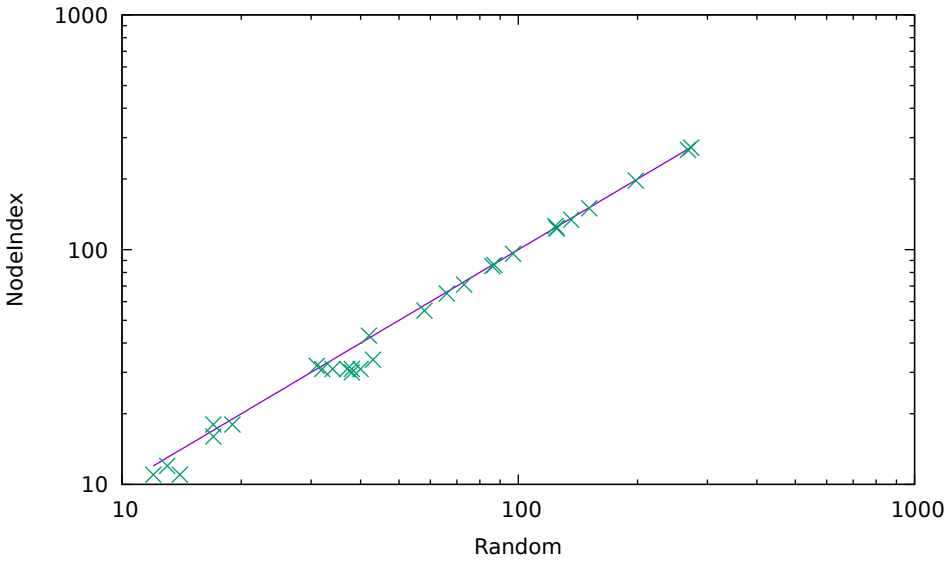


Figure 13: Two suggestions have very similar performance since all points are close to the diagonal. The fact that most points are over the diagonal shows that our suggestion has marginal advantages over random initialization. Every graph in Table 3 with different results is plotted as a green cross.

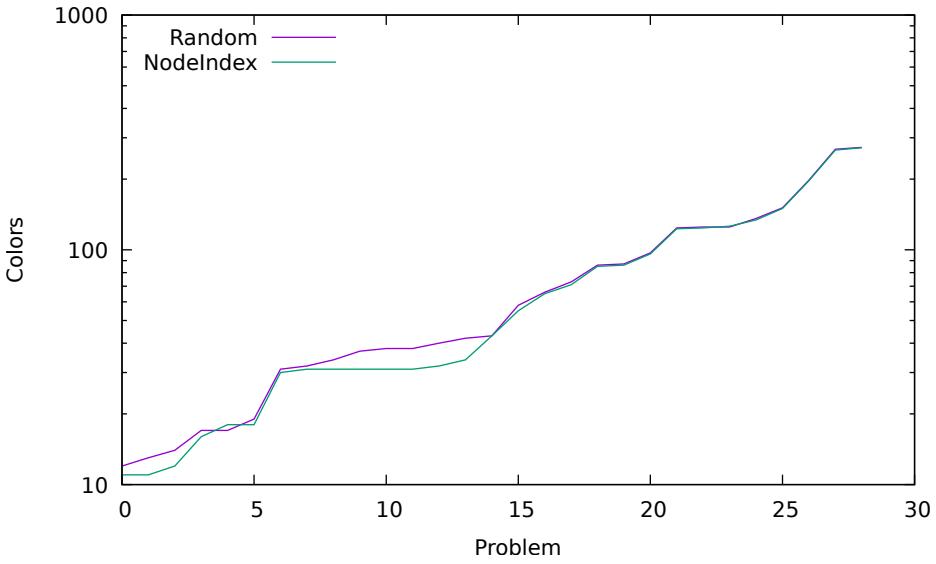


Figure 14: Our suggestion shows advantages especially for small graphs.

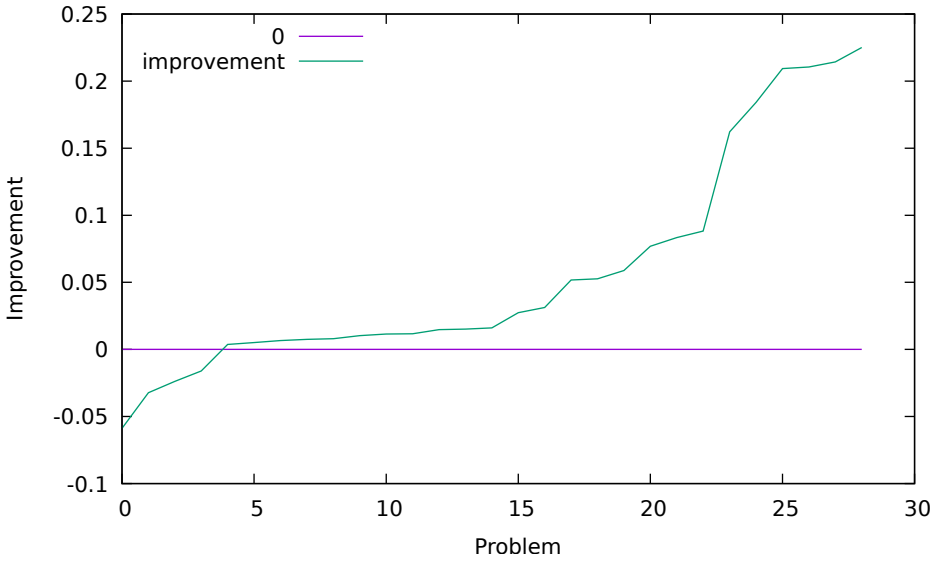


Figure 15: advantage plot NodeIndex: Random

5.5.2 Experiment 2: Original solver vs Solution replacement

The GCP-solver introduced in section 3 runs within a time interval t. In our suggestion, the current coloring will be replaced by a randomly generated coloring of the same size when a Tabucol search fails after $\lfloor t/2 \rfloor$.

Comparing the original GCP-solver and our suggestion which replaces the current illegal solution after half of the time interval, our suggestion shows improvement in 36 graphs. The results are shown in Table 3 (column “Replacement”).

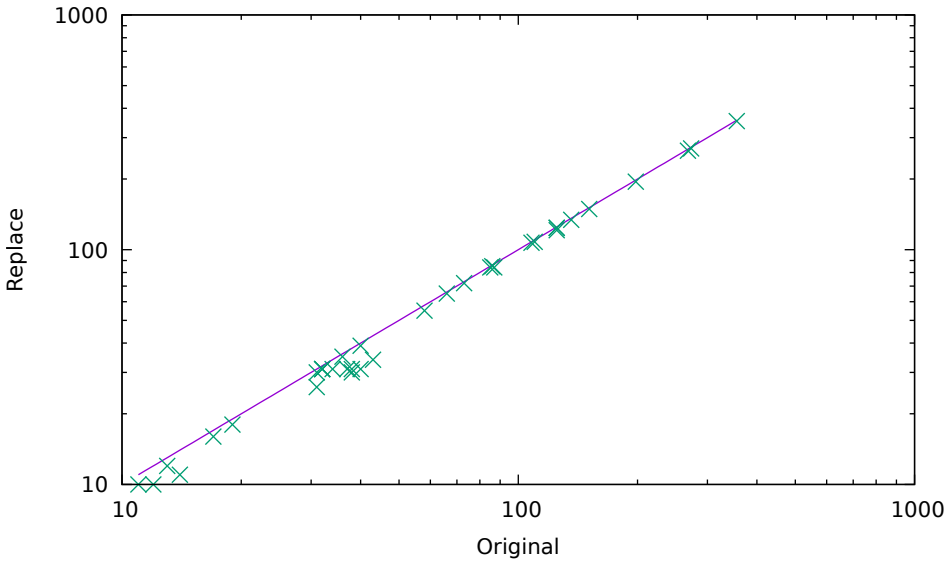


Figure 16: The points below the diagonal show an advantage of our suggestion.

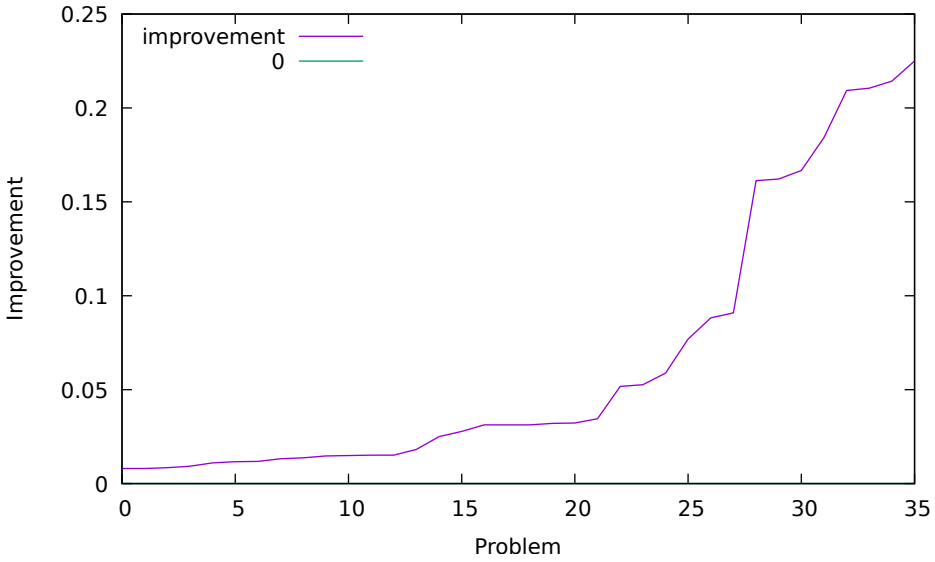


Figure 17: The improvement of our suggestion is very small (most relative differences are under 0.1%). However, this suggestion is adopted because of its universal improvement (more than 53% graphs are improved with our suggestion).

5.5.3 Experiment 3: RowTraverse vs ColumnTraverse

In our Implementation, we use the conflict matrix to evaluate the one step moves. The suggestion *RowTraverse* traverses the matrix row by row to find the best candidate. If more than one best candidates exist, it will always choose the first one for the next move. Our suggestion called *ColumnTraverse* traverses the conflict matrix column by column and uses the first best candidate for the next move. In experiment 3, the *ColumnTraverse* shows advantages in more than half of the graph instances (see table 3 column "Ctraverse").

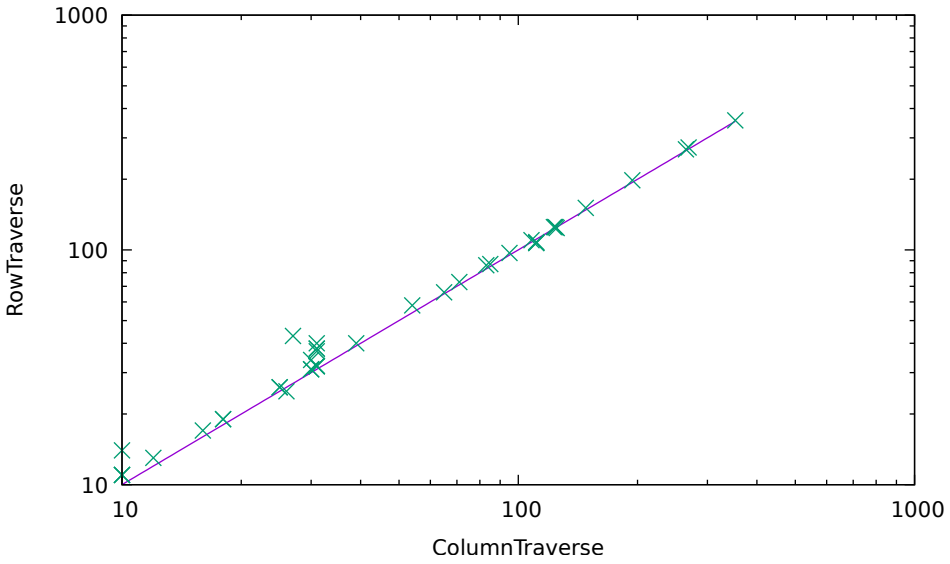


Figure 18: The scatter plot compares the *RowTraverse* and *ColumnTraverse*. The advantage of *ColumnTraverse* concentrates on small graphs. For large graphs, two traverse directions get similar results.

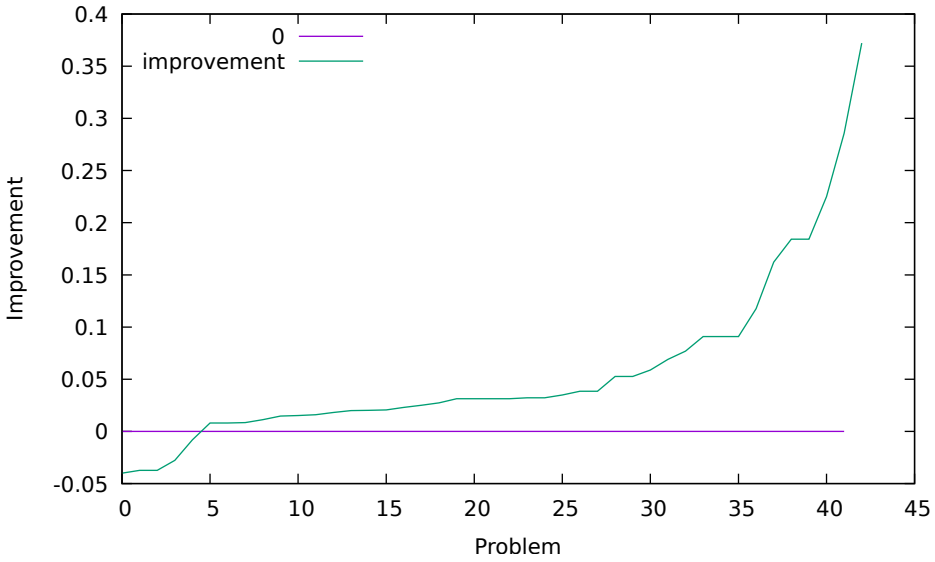


Figure 19: 43 graphs of our instances get small differences by *RowTraverse* and *ColumnTraverse*. 38 graphs get better results with *ColumnTraverse* and 5 graphs benefit from *RowTraverse*.

5.5.4 Experiment 4: Original solver vs Statistic solver

In our GCP-solver, we add an $n \times n$ statistic matrix to record the repeated steps in the past. The aim of adding this statistic matrix is to avoid long-term cycling of the local search in a neighborhood. When a cycling is longer than the length of the tabu list, the original GCP-solver will be stuck in the cycling, while the solver with statistic matrix can realize this and jump out of this cycling by choosing candidates which are not involved in this cycling. In Experiment 4, the original GCP-solver is compared with a GCP-solver with statistic matrix S. The S_{ij} corresponds to how many times [v_i, j] is chosen as the best move. When a move [v_i, j] is chosen for the next step in the search, S_{ij} is increased by one. If two best candidates exist, the corresponding entities in the statistic matrix will be compared. The candidate which is less used before will be chosen as the next move. When the value of one entity in statistic matrix reaches a predefined upper bound (In our experiment, the upper bound is n), the suboptimal candidate will be used in the next move. The results of this experiment are in table 3 (see column “Statistic”).

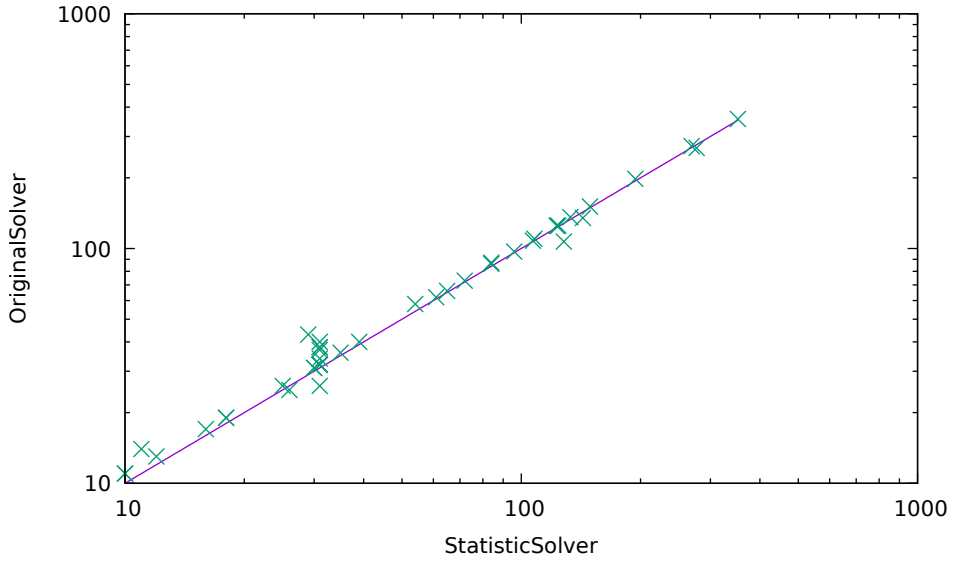


Figure 20: Scatter plot of 44 graphs in our benchmark with differences between the original GCP-solver (OriginalSolver) and the solver with statistic matrix (StatisticSolver)

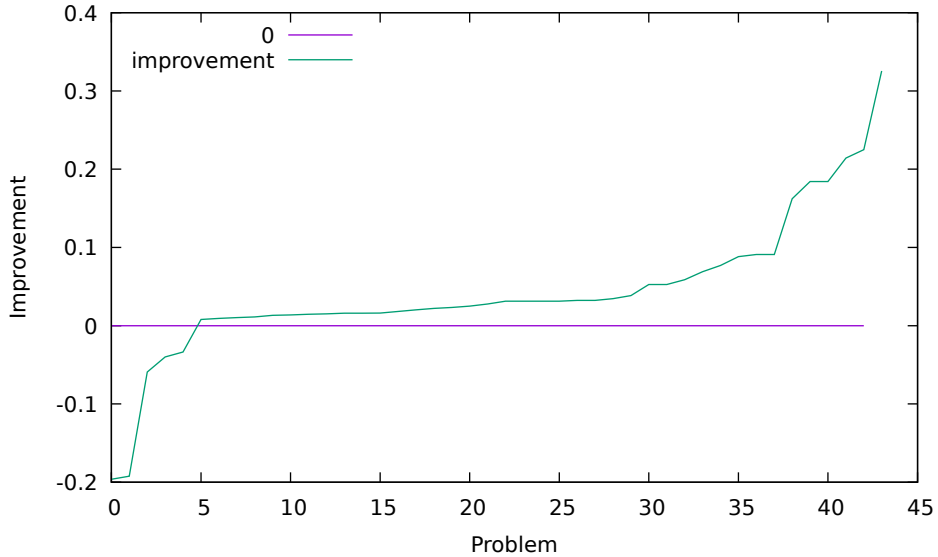


Figure 21: 39 graphs of 68 graph instances (57% graphs) get a better result with help of a statistic matrix. 5 graphs (7% graphs) get a better result without statistic matrix.

Graph	Original	Node-Index	Replace	Ctraverse	Statistic	Our solver
miles250	**8**	**8**	**8**	**8**	**8**	**8**
jean	**10**	**10**	**10**	**10**	**10**	**10**
queen8_8	11	11	**10**	**10**	**10**	**10**
le450_5a	11	11	11	**10**	**10**	**10**
le450_5b	11	11	11	**10**	11	11
queen9_9	12	11	**10**	12	12	11
le450_5c	13	**12**	**12**	**12**	**12**	**12**
le450_5d	14	11	11	**10**	11	12
queen8_12	**13**	**13**	**13**	**13**	**13**	**13**
queen10_10	**13**	**13**	**13**	**13**	**13**	**13**
queen11_11	**14**	**14**	**14**	**14**	**14**	**14**
queen12_12	**15**	**15**	**15**	**15**	**15**	**15**
queen13_13	**16**	**16**	**16**	**16**	**16**	**16**
dsjc500.1	17	**16**	**16**	**16**	**16**	**16**
queen14_14	**17**	18	**17**	**17**	**17**	**17**
queen15_15	19	**18**	**18**	**18**	**18**	**18**
le450_15b	19	19	19	**18**	**18**	**18**
miles500	**20**	**20**	**20**	**20**	**20**	**20**
queen16_16	**20**	**20**	**20**	**20**	**20**	**20**
le450_15c	26	26	26	**25**	**25**	**25**
le450_25a	26	26	26	**25**	26	**25**
le450_25b	**25**	**25**	**25**	26	26	**25**
le450_15d	31	32	**26**	31	31	**26**
dsjc1000.1	**26**	**26**	**26**	**26**	31	**26**
school1	43	34	34	**27**	29	33
school1_nsh	34	31	31	30	31	**29**
zeroin.i.2	31	31	**30**	**30**	**30**	**30**
zeroin.i.3	31	31	31	**30**	**30**	**30**
fpsol2.i.3	38	**30**	**30**	31	31	**30**
fpsol2.i.2	37	31	31	31	31	**30**
inithx.i.2	40	**31**	**31**	**31**	**31**	**31**
inithx.i.3	38	**31**	**31**	**31**	**31**	**31**
le450_25c	32	32	**31**	**31**	**31**	**31**
le450_25d	32	32	32	**31**	**31**	**31**

Graph	Original	Node-Index	Replace	Ctraverse	Statistic	Our solver
miles750	32	32	**31**	**31**	**31**	**31**
mulsol.i.2	**31**	**31**	**31**	**31**	**31**	**31**
mulsol.i.3	32	**31**	**31**	**31**	**31**	**31**
mulsol.i.4	**31**	**31**	**31**	**31**	**31**	**31**
mulsol.i.5	**31**	**31**	**31**	**31**	**31**	**31**
dsjc250.5	36	36	**35**	36	**35**	**35**
flat300_28_0	40	40	**39**	**39**	**39**	**39**
miles1000	**42**	43	**42**	**42**	**42**	**42**
mulsol.i.1	**49**	**49**	**49**	**49**	**49**	**49**
zeroin.i.1	**49**	**49**	**49**	**49**	**49**	**49**
inithx.i.1	58	55	55	**54**	**54**	**54**
dsjc500.5	62	62	62	62	**61**	**61**
fpsol2.i.1	66	**65**	**65**	**65**	**65**	**65**
r250.5	73	**71**	72	**71**	72	**71**
miles1500	**73**	**73**	**73**	**73**	**73**	**73**
brock400_1	86	85	85	**83**	84	84
brock400_2	87	86	**84**	85	**84**	**84**
brock400_3	85	85	**84**	85	85	**84**
dsjr500.1c	97	96	97	**95**	96	96
flat1000_60_0	108	108	107	111	107	**106**
flat1000_50_0	107	107	107	111	107	**106**
flat1000_76_0	**107**	**107**	**107**	111	128	**107**
dsjc1000.5	110	110	**108**	**108**	**108**	**108**
r1000.1c	125	123	**121**	123	123	123
brock800_1	125	124	124	124	**123**	**123**
brock800_2	125	125	**124**	**124**	**124**	**124**
brock800_4	**124**	126	**124**	125	**124**	**124**
latin_square_10	136	134	134	136	**133**	**133**
dsjr500.5	135	135	135	135	143	**134**
dsjc500.9	151	150	149	**148**	149	149
c2000.5	198	197	195	**194**	**194**	**194**
r1000.5	268	266	264	265	277	**263**
dsjc1000.9	273	272	270	**269**	**269**	**269**
c4000.5	356	356	353	353	**352**	**352**

Table 3: The original GCP-solver uses a randomly generated initial solution. It traverses the solution matrix row by row. The original solver is implemented with the data structure shown in 3.1, statistic matrix not included.

5.5.5 Experiment 5: Original solver vs Parallel solver with various parameter combinations

Our parallel implementation uses OpenMP to support shared memory multiprocessing. Our pure portfolio approach takes advantage of flexible parameter combinations. The slave threads execute the GCP-solver with different parameter combinations in parallel. Then the result of each agent is written in a shared memory. The master thread compares the results and chooses the result with minimum size as the final result.

Algorithm 4: parallel GCP-solver with different parameter combinations

`input` : A Graph in DIMACS standard format, number of agents t
`parameter`: parameter combinations $\{p_1, p_2, ..., p_t\}$
`output` : Solution s

1 start t agents;
2 Each agent runs GCP-solver with parameter combination $p_{index\_of\_t}$;
3 solutions $\{s_1, s_2, ..., s_t\}$ found by agents are collected and compared;
4 Output the coloring of the minimum size from $\{s_1, s_2, ..., s_t\}$;

In experiment 5, we compare our pure portfolio GCP-solver with a single-threaded GCP-solver. The parallel GCP-solver with t threads use the first t parameter combinations in table 2.

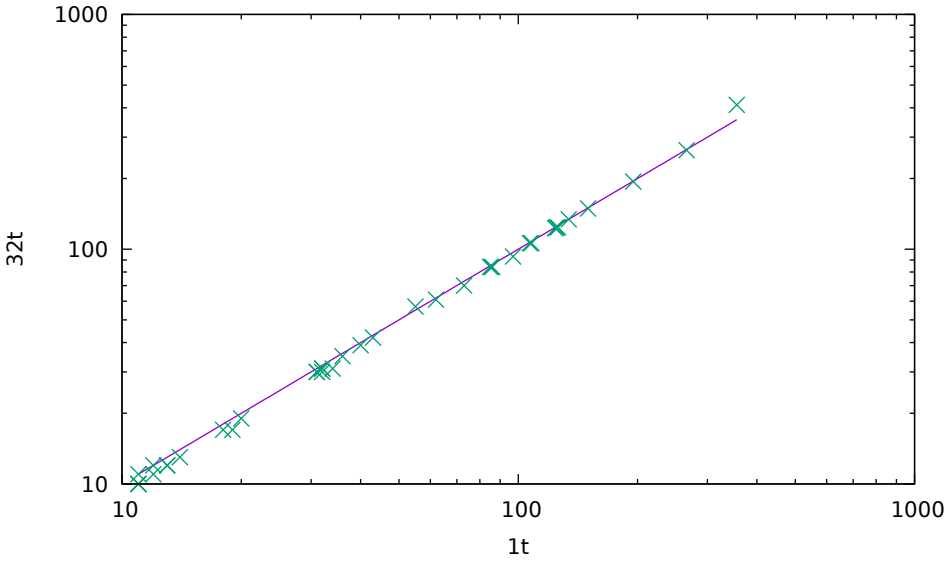


Figure 22: The scatter plot compares our single-threaded GCP-solver and the pure portfolio solver with 32 threads. Most points are under the diagonal. One outlier (the huge graph c4000.5) is on the right top, which seems to be caused by the time spent on swapping to hard disk when RAM is full.

	1t	2t	4t	8t	16t	32t
best	0	8	9	23	29	34
unique	0	0	2	0	1	9

Table 4: For a pure portfolio approach, the numbers of times of getting the minimum size among all the solvers are recorded in the row "best". The row "unique" records the times of getting the unique minimum size. This table is based on the 40 graphs shown in table 5.

Graph	1t	2t	4t	8t	16t	32t
queen8_8	11	11	11	**10**	**10**	**10**
le450_5a	11	11	11	**10**	**10**	**10**
le450_5b	11	11	11	**10**	**10**	**10**
queen9_9	11	11	**10**	11	11	11
le450_5c	12	**11**	**11**	**11**	**11**	12
le450_5d	12	**11**	**11**	**11**	**11**	**11**
queen8_12	13	13	13	13	**12**	**12**
queen10_10	13	13	13	13	**12**	**12**
queen11_11	14	14	14	14	14	**13**
queen14_14	18	**17**	**17**	**17**	**17**	**17**
le450_15b	19	19	19	18	**17**	**17**
queen16_16	20	20	20	20	**19**	**19**
le450_15c	26	26	26	26	26	**25**
school1	34	34	34	**31**	**31**	**31**
school1_nsh	31	31	31	31	31	**30**
zeroin.i.2	31	31	31	**30**	**30**	**30**
le450_25c	32	32	31	31	31	**30**
le450_25d	32	32	32	**31**	**31**	**31**
miles750	32	32	32	**31**	**31**	**31**
dsjc250.5	36	36	36	**35**	**35**	**35**
flat300_28_0	40	**39**	**39**	**39**	**39**	**39**
miles1000	43	**42**	**42**	**42**	**42**	**42**
inithx.i.1	55	**54**	55	**54**	**54**	57
dsjc500.5	62	**61**	**61**	**61**	**61**	**61**
r250.5	73	73	71	71	71	**70**
brock400_1	85	85	85	**84**	**84**	**84**
brock400_2	86	85	85	85	85	**84**
brock400_3	85	85	85	**84**	**84**	**84**
dsjr500.1c	97	97	96	**93**	**93**	**93**
flat1000_60_0	108	107	107	**106**	**106**	**106**
flat1000_50_0	107	107	107	107	**106**	**106**
r1000.1c	124	124	124	124	122	**123**
brock800_1	124	124	124	124	124	**123**
brock800_2	125	125	125	**124**	**124**	**124**
brock800_4	126	**124**	125	**124**	**124**	**124**
latin_square_10	134	134	134	**132**	**132**	134
dsjc500.9	150	150	150	150	150	**149**
c2000.5	195	195	195	195	195	**194**
r1000.5	266	266	**264**	**264**	**264**	**264**
c4000.5	356	357	**355**	363	412	412

Table 5: 40 graphs get different results when GCP-solver uses different numbers of threads. The column nt ($n \in \{1, 2, 4, 8, 16, 32\}$) is for the corresponding results of the pure portfolio GCP-solver with n threads. The GCP-solver with one thread is identical to the original GCP-solver introduced in section 3. The graphs in the table are ordered with color size found in the original GCP-solver.

5.5.6 Experiment 6: GCP-solver with FRC

In experiment 6 the approach forced color reducing (FRC) was tested. There is a global variable k_{min} shared by all agents. It represents the minimum size of the legal coloring found by the agents. The GCP-solver_FCR runs Tabucol iteratively on several agents with different parameter combinations. In the process, the minimum size is shared among the agents. The

pseudo code is as follows:

Algorithm 5: A parallel GCP-solver with forced color reducing

```
input     : A Graph G in DIMACS standard format, number of agents t
parameter: parameter combinations {p_1, p_2, ..., p_t}, Timeout
output    : Solution s
1  k_min = n;
2  start t agents;
3  //in t Agents
4  c = initialColoring(G, k);
5  k = n − 1;
6  while (Timeout does not occur) do
7      Tabucol(c, k); //search a k-coloring based on c and set the new coloring as c
8      if (k < k_min ∧ Tabucol(c, k) succeed) then
9          k_min = k;
10     k = k − 1;
11     c = reduceOneColor(c, k) //reduce the least used color in the current coloring
```

The results of our experiment are shown in table 7. 49 graphs get different results when GCP-solver_FCR uses different numbers of threads. Generally, the result is improved with the increase in the number of threads. For the graph c400.5, the results with 16 threads and 32 threads are unusual. A possible cause is that the swapping to hard disk in this huge graph takes a lot of time.

	1t	2t	4t	8t	16t	32t
best	2	8	18	24	32	45
unique	0	0	1	1	2	14

Table 6: As the number of threads increases, the numbers of times of getting the minimum size among all the solvers are increased. The table left is based on the 49 graphs shown in table 7.

Graph	1t	2t	4t	8t	16t	32t
queen8_8	11	**10**	**10**	**10**	**10**	**10**
le450_5b	11	**10**	11	**10**	**10**	**10**
queen9_9	**11**	12	**11**	**11**	**11**	**11**
le450_5c	12	12	12	11	11	**10**
le450_5d	12	12	12	11	11	**10**
queen8_12	13	13	13	13	**12**	**12**
queen10_10	13	13	13	13	**12**	**12**
queen11_11	14	14	14	14	14	**13**
queen12_12	15	15	15	15	15	**14**
dsjc500.1	16	16	16	16	**15**	**15**
queen14_14	18	18	**17**	**17**	**17**	**17**
queen16_16	20	20	20	**19**	**19**	**19**
le450_15c	26	26	26	**25**	**25**	**25**
le450_25a	26	**25**	**25**	**25**	**25**	**25**
le450_25b	**25**	26	**25**	**25**	**25**	**25**
le450_15d	26	26	26	**25**	**25**	**25**
dsjc1000.1	26	26	26	26	**25**	**25**
school1	34	36	33	34	33	**31**
school1_nsh	31	32	32	31	**29**	30
zeroin.i.2	31	31	**30**	**30**	**30**	**30**
zeroin.i.3	31	31	**30**	**30**	**30**	**30**
fpsol2.i.2	31	**30**	**30**	**30**	**30**	**30**
le450_25c	31	32	31	31	31	**30**
le450_25d	31	31	31	**30**	31	31
miles750	32	32	32	**31**	**31**	**31**
dsjc250.5	36	**35**	**35**	**35**	**35**	**35**
flat300_28_0	39	39	40	39	39	**38**
miles1000	43	43	**42**	**42**	**42**	**42**
dsjc500.5	62	62	62	**61**	**61**	**61**
r250.5	73	73	**71**	**71**	**71**	**71**
brock400_1	85	85	85	84	84	**83**
brock400_2	85	85	85	85	**84**	**84**
brock400_3	86	86	85	**84**	**84**	**84**
dsjr500.1c	95	**94**	**94**	96	95	**94**
flat1000_60_0	107	107	107	107	107	**106**
flat1000_50_0	107	**106**	**106**	**106**	**106**	**106**
flat1000_76_0	109	108	**107**	**107**	**107**	**107**
dsjc1000.5	109	110	109	108	109	**107**
r1000.1c	126	126	125	123	120	**119**
brock800_1	125	124	124	**123**	**123**	**123**
brock800_2	125	**124**	**124**	**124**	**124**	**124**
brock800_4	126	124	124	124	**123**	**123**
latin_square_10	136	134	**132**	133	**132**	**132**
dsjr500.5	136	136	135	135	135	**134**
dsjc500.9	152	150	**149**	**149**	**149**	**149**
c2000.5	195	196	195	195	195	**194**
r1000.5	268	267	266	264	**263**	264
dsjc1000.9	271	271	270	270	270	**268**
c4000.5	356	362	**354**	357	386	406

Table 7: GCP-solver_FCR

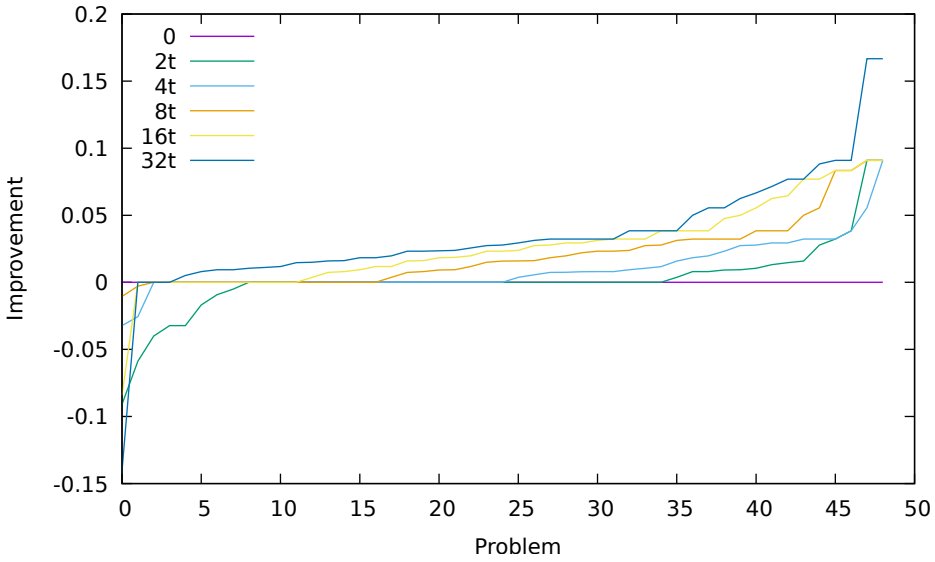


Figure 23: This advantage plot presents the improvement of GCP-solver through sharing the global minimum color size among all the agents. The result with 32 threads shows big and stable improvement.

5.5.7 Experiment 7: GCP-solver with tabu share

Experiment 7 tests the approach of tabu sharing (see 4.3). All the agents run the GCP-solver independently and the final result is the minimum size among the agents. The only difference of this approach from the pure portfolio GCP-solver is that all agents share one tabu list, which is built with the parameter combination of the original GCP-solver ($L = 9$ and $\alpha = 0.38$). We compare the results of tabu shared GCP-solver with different numbers of threads (see Table 9).

	1t	2t	4t	8t	16t	32t
best	52	32	18	11	7	5
unique	25	4	0	0	0	0

Table 8: GCP-solver with tabu sharing is a suggestion with bad performance. The more threads share the tabu list, the worse the performance of the search is. The statistical data here are based on table 9.

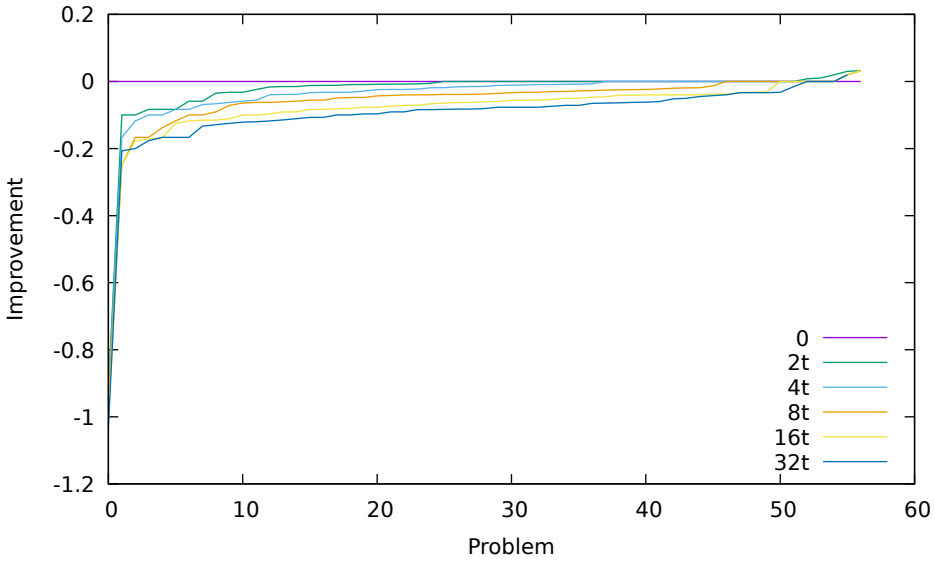


Figure 24: Advantage plot

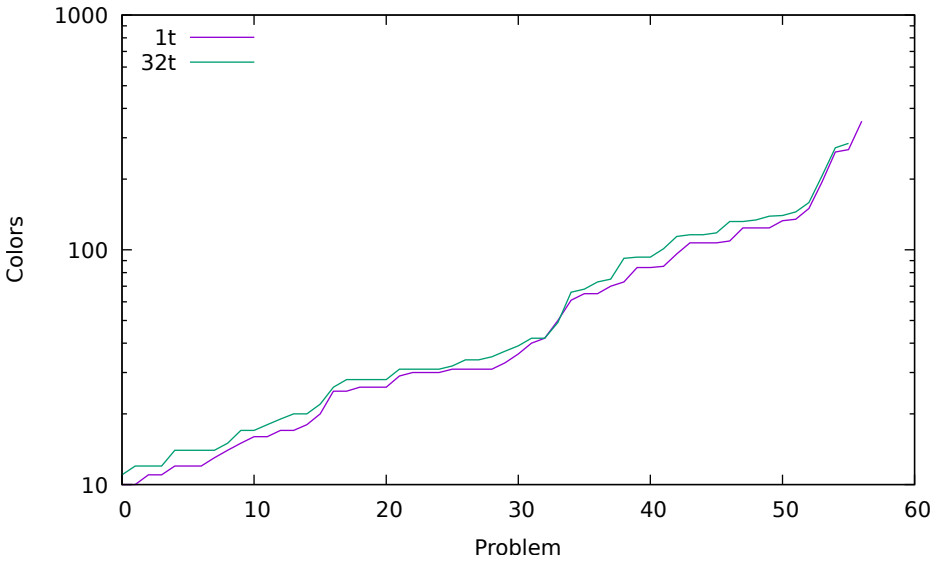


Figure 25: The performance of 32-thread tabu sharing GCP-solver is always worse than the performance of single-threaded GCP-solver

Graph	1t	2t	4t	8t	16t	32t
queen8_8	**10**	11	11	11	11	12
le450_5a	**11**	**11**	**11**	**11**	12	12
le450_5b	**10**	11	11	11	11	11
queen9_9	**11**	**11**	**11**	12	12	12
le450_5c	**12**	13	13	15	15	14
le450_5d	**12**	13	14	14	14	14
queen8_12	**12**	13	13	14	13	14
queen10_10	**13**	**13**	**13**	**13**	14	14
queen11_11	**14**	**14**	**14**	15	15	15
queen12_12	**15**	**15**	16	**15**	16	17
queen13_13	**16**	**16**	17	17	17	18
dsjc500.1	**16**	**16**	**16**	17	18	17
queen14_14	**17**	18	18	18	19	19
queen15_15	**18**	**18**	19	19	20	20
le450_15b	**17**	18	19	19	20	20
queen16_16	**20**	**20**	**20**	**20**	21	22
le450_15c	**26**	**26**	**26**	27	28	28
le450_25a	**25**	**25**	26	26	27	28
le450_25b	**25**	**25**	**25**	26	26	26
le450_15d	**26**	**26**	27	27	29	28
dsjc1000.1	**26**	**26**	27	27	29	28
school1	33	**32**	33	35	35	37
school1_nsh	**29**	30	31	33	34	35
zeroin.i.2	**30**	**30**	**30**	**30**	**30**	31
zeroin.i.3	**30**	**30**	**30**	**30**	31	31
fpsol2.i.3	31	**30**	**30**	**30**	**30**	31
fpsol2.i.2	**30**	**30**	31	31	**30**	31
le450_25c	**31**	32	32	32	33	34
le450_25d	**31**	32	32	33	34	34
miles750	**31**	**31**	32	**31**	32	32
dsjc250.5	**36**	**36**	**36**	38	38	39
flat300_28_0	**40**	**40**	**40**	41	42	42
miles1000	**42**	**42**	**42**	43	**42**	**42**
zeroin.i.1	50	**49**	**49**	**49**	**49**	**49**
dsjc500.5	**61**	62	**61**	64	66	68
fpsol2.i.1	**65**	**65**	**65**	**65**	**65**	66
r250.5	**70**	**70**	72	73	75	75
miles1500	**73**	**73**	**73**	**73**	**73**	**73**
brock400_1	**84**	85	86	88	91	93
brock400_2	**85**	86	86	88	89	92
brock400_3	**84**	86	86	88	89	93
dsjr500.1c	96	**95**	96	100	102	101
flat1000_60_0	**107**	**107**	109	111	113	116
flat1000_50_0	**107**	**107**	108	109	111	116
flat1000_76_0	**107**	108	108	110	113	114
dsjc1000.5	**109**	110	111	112	114	118
r1000.1c	124	**123**	127	129	129	140
brock800_1	**124**	125	126	127	129	132
brock800_2	**124**	125	125	128	129	134
brock800_4	**124**	125	125	127	130	132
latin_square_10	**135**	136	137	138	145	145
dsjr500.5	**133**	135	136	137	138	139
dsjc500.9	**150**	**150**	151	152	156	159
c2000.5	**195**	196	197	199	203	207
r1000.5	**261**	264	265	266	269	272
dsjc1000.9	**267**	271	270	275	278	284
c4000.5	**353**	356	356	362	372	545

Table 9: GCP-solver with tabu sharing. For c400.5, the performance with 32 thread is poor because of much memory swapping.

5.5.8 Experiment 8: GCP-solver with statistic Matrix sharing

The statistic matrix is an important data structure in our implementation of GCP-solver to avoid long-term cycling in the local search. In experiment 4 5.5.4, we found that with help of statistic matrix, the performance of our search in 55% graphs is improved (see table 11).

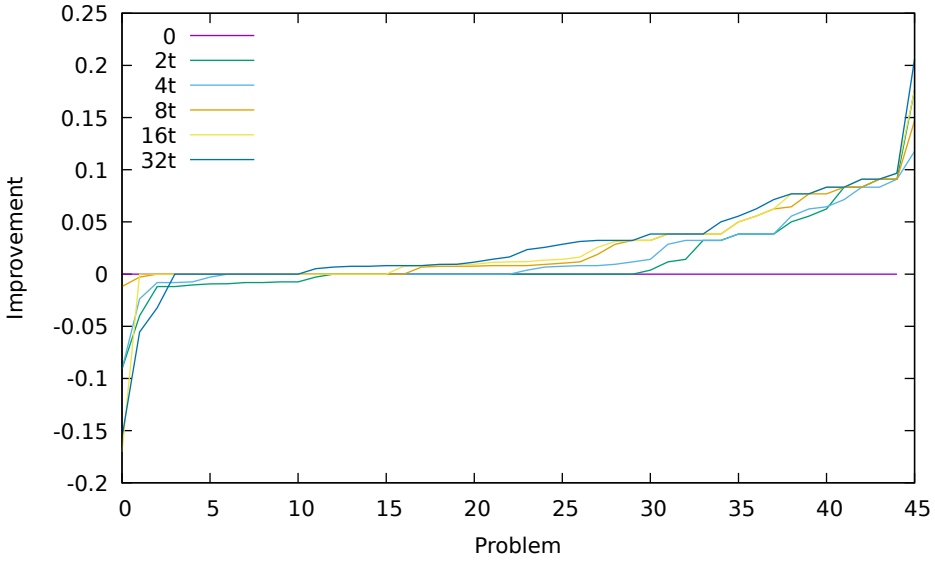


Figure 26: The improvement of GCP-solver through statistic matrix sharing is big. The improvement is however not stable.

	1t	2t	4t	8t	16t	32t
best	7	14	22	27	32	36
unique	1	0	0	2	4	6

Table 10: With more threads, the performance of our GCP-solver is improved through statistic matrix sharing. With 8 threads, more than half of the graphs have reached their minimum color size in local search.

5.5.9 Experiment 9: Comparison of our GCP-solver with other algorithms

In experiment 9, we compare our parallel GCP-solver with forced color reducing and statistic sharing with three other DSATUR-based algorithms: DSATUR algorithm, PASS algorithm, and TRICK algorithm (see table 12). The source code of DSATUR and PASS is supplied by Fabio Furini [30]. The source code of the algorithm TRICK is provided by Michael Trick [20] (see details in 2.2). The benchmark graphs are tested with the same timeout as in our GCP solver algorithm (see 5.2).

graph	1t	2t	4t	8t	16t	32t
queen8_8	11	**10**	**10**	**10**	**10**	**10**
le450_5b	11	**10**	11	**10**	**10**	**10**
queen9_9	**11**	12	12	**11**	**11**	**11**
le450_5c	12	**11**	**11**	**11**	**11**	**11**
le450_5d	12	**11**	**11**	**11**	**11**	**11**
queen8_12	13	13	13	**12**	**12**	**12**
queen10_10	13	13	13	**12**	**12**	**12**
queen11_11	14	14	**13**	14	14	**13**
dsjc500.1	16	**15**	**15**	**15**	**15**	**15**
queen14_14	18	**17**	**17**	**17**	**17**	**17**
queen16_16	20	**19**	20	**19**	**19**	**19**
le450_15c	**25**	26	**25**	**25**	**25**	**25**
le450_25a	26	**25**	**25**	**25**	**25**	**25**
le450_25b	26	**25**	**25**	**25**	**25**	**25**
le450_15d	26	**25**	26	**25**	**25**	**25**
dsjc1000.1	26	26	**25**	**25**	**25**	**25**
school1	34	28	30	29	28	**27**
school1_nsh	31	30	29	29	**28**	**28**
zeroin.i.2	31	**30**	**30**	**30**	**30**	**30**
fpsol2.i.3	31	31	**30**	**30**	**30**	**30**
le450_25c	31	31	**30**	31	31	**30**
miles750	**31**	**31**	**31**	**31**	**31**	32
dsjc250.5	35	35	**34**	**34**	35	**34**
flat300_28_0	39	39	39	39	**38**	**38**
inithx.i.1	**54**	**54**	**54**	**54**	**54**	57
dsjc500.5	61	61	61	61	61	**60**
r250.5	71	**70**	**70**	71	**70**	**70**
brock400_1	84	85	86	85	**83**	84
brock400_2	**84**	85	**84**	**84**	**84**	**84**
brock400_3	85	84	84	84	84	**83**
dsjr500.1c	96	97	96	95	**93**	**93**
flat1000_60_0	**106**	107	**106**	**106**	**106**	**106**
flat1000_50_0	106	106	106	**104**	105	106
flat1000_76_0	107	107	**106**	107	**106**	**106**
dsjc1000.5	108	109	108	**107**	**107**	**107**
r1000.1c	123	123	122	122	**121**	122
brock800_1	123	124	124	**122**	123	123
brock800_2	124	125	125	**123**	124	**123**
brock800_4	124	124	**123**	124	**123**	**123**
latin_square_10	133	134	**132**	**132**	133	133
dsjr500.5	134	135	135	134	134	**133**
dsjc500.9	149	149	148	148	**147**	148
c2000.5	194	194	194	194	194	**193**
r1000.5	263	263	263	261	261	**260**
dsjc1000.9	269	268	268	267	**266**	267
c4000.5	**352**	353	353	353	412	407

Table 11: 46 of our 68 benchmark graphs have a different performance with GCP-solver using statistic matrix sharing. The GCP-solver with one thread is identical to the GCP-solver using a statistic matrix. Through experiment 4, we know the statistic solver itself brought an improvement to original GCP-solver.

Graph	1t	2t	4t	8t	16t	32t	DSATUR	PASS	TRICK
miles250	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**
jean	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
queen8_8	11	10	10	10	10	10	**9**	**9**	**9**
le450_5a	10	10	10	10	10	10	**8**	9	9
le450_5b	11	10	11	10	10	10	**9**	**9**	**9**
queen9_9	11	12	11	11	11	11	**10**	**10**	**10**
le450_5c	12	11	11	11	11	10	9	**5**	**5**
le450_5d	12	11	11	11	11	10	10	9	**8**
queen8_12	13	13	13	**12**	**12**	**12**	**12**	**12**	**12**
queen10_10	13	13	13	**12**	**12**	**12**	**12**	**12**	**12**
queen11_11	14	14	**13**	14	14	**13**	**13**	**13**	**13**
queen12_12	15	15	15	15	15	**14**	15	15	15
queen13_13	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**
dsjc500.1	16	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**
queen14_14	18	**17**	**17**	**17**	**17**	**17**	18	**17**	**17**
queen15_15	**18**	**18**	**18**	**18**	**18**	**18**	19	20	**18**
le450_15b	18	18	18	18	18	18	**16**	**16**	**16**
miles500	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
queen16_16	20	**19**	20	**19**	**19**	**19**	20	**19**	**19**
le450_15c	25	26	25	25	25	25	**23**	**23**	**23**
le450_25a	26	**25**	**25**	**25**	**25**	**25**	**25**	**25**	**25**
le450_25b	**25**	**25**	**25**	**25**	**25**	**25**	**25**	**25**	**25**
le450_15d	26	25	26	25	25	25	**23**	**23**	**23**
dsjc1000.1	26	26	**25**	**25**	**25**	**25**	26	26	-
school1	34	28	30	29	28	27	**14**	**14**	**14**
school1_nsh	31	30	29	29	28	28	24	**14**	**14**
zeroin.i.2	31	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
zeroin.i.3	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
fpsol2.i.3	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
fpsol2.i.2	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
inithx.i.2	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**	-
inithx.i.3	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**	-
le450_25c	31	31	30	31	31	30	**27**	28	28
le450_25d	31	31	31	30	31	31	**27**	**27**	**27**
miles750	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**
mulsol.i.2	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**
mulsol.i.3	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**
mulsol.i.4	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**
mulsol.i.5	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**	**31**
dsjc250.5	35	35	**34**	**34**	35	**34**	-	-	35
flat300_28_0	39	39	39	39	**38**	**38**	41	40	39
miles1000	**42**	**42**	**42**	**42**	**42**	**42**	**42**	**42**	**42**
mulsol.i.1	**49**	**49**	**49**	**49**	**49**	**49**	**49**	**49**	**49**
zeroin.i.1	**49**	**49**	**49**	**49**	**49**	**49**	**49**	**49**	**49**
inithx.i.1	**54**	**54**	**54**	**54**	**54**	**54**	**54**	**54**	-
dsjc500.5	61	61	61	61	61	**60**	63	64	63
fpsol2.i.1	**65**	**65**	**65**	**65**	**65**	**65**	**65**	**65**	**65**
r250.5	71	70	70	71	70	70	**66**	67	-
miles1500	**73**	**73**	**73**	**73**	**73**	**73**	**73**	**73**	**73**
brock400_1	84	85	85	84	**83**	**83**	90	90	89
brock400_2	**84**	85	**84**	**84**	**84**	**84**	90	91	91
brock400_3	85	84	84	84	84	**83**	90	89	90
dsjr500.1c	95	94	94	95	93	93	-	-	**88**
flat1000_60_0	**106**	107	**106**	**106**	**106**	**106**	111	113	-
flat1000_50_0	106	106	106	**104**	105	106	114	112	-
flat1000_76_0	107	107	**106**	107	**106**	**106**	114	111	-
dsjc1000.5	108	109	108	**107**	**107**	**107**	116	114	-
r1000.1c	123	123	122	122	120	**119**	-	-	
brock800_1	123	124	124	**122**	123	123	133	132	-
brock800_2	124	124	124	123	124	**123**	132	131	-
brock800_4	124	124	**123**	124	**123**	**123**		131	-
latin_square_10	133	134	132	132	132	132	**130**	140	-
dsjr500.5	134	135	135	134	134	133	131	134	**130**
dsjc500.9	149	149	148	148	**147**	148	-	-	160
c2000.5	194	194	194	194	194	**193**	208	204	-
r1000.5	263	263	263	261	261	260	250	**245**	-
dsjc1000.9	269	268	268	267	**266**	267	297	300	-
c4000.5	**352**	353	353	353	386	**352**	377	376	-

Table 12: comparison with DSATUR, PASS, and TRICK. The column nt is for n-thread GCP-solver with forced color reducing and statistic sharing. "-" means no result at the timeout. 50 of 68 (73%) benchmark graphs get best results with our solver. 20 of 68 (29%) benchmark graphs get unique best results with our solver.

6 Conclusion

The scheme of finding a legal k-coloring of a graph and then reducing the size in search iteratively to find the minimum color size, is universally applicable to GCP algorithms. Our paper presents a parallel cooperative algorithm to solve GCP with this scheme. The local search used in our algorithm is a tabu search called Tabucol.

In section 3, we discuss the three steps of this scheme: solution initialization, k-GCP solution and color reduction. To each step, we give a suggestion in implementation to improve the performance of our algorithm. In 3.2, we compare the GCP-solver with a randomly generated initial solution and the version with an n-coloring which takes node indices as the initial color indices. The interesting fact is that the node-index initialization is advantageous in the result and also in execution time because of its simplicity. For the k-GCP solution, we discuss the way of choosing the next move if several critical moves exist with the maximal improvement. For the color reduction, we always reduce the least used color in the current solution. Apart from discussing the process of the GCP-solver, we introduce the data structures in our implementation. We add a matrix data structure called statistic matrix in our implementation, which supports the recognition of long-term cycling in tabu search and therefore improves the performance of our GCP-solver.

Most graphs get better results with the suggestions in our single-threaded GCP-solver. However, some graphs get better results with the original GCP-solver. With this observation, we make our GCP-solver parallel with different parameter combinations in agents. In this way, the agents run the search with some different settings and then take advantage of the reasonable combinations of the suggestions.

In section 4, we discussed the exchange of information among the agents in a parallel search. The first approach is minimum size exchange, in which the agents exchange the found color size. The approach, whose initial purpose is to save search time, was found experimentally to bring also an improvement on results.
Then we shared the information of the local search cycling among the agents. We tested the exchange of statistic matrix in the search process since tabu list sharing turned out to be a failed attempt. We found the statistic matrix sharing brings further improvement.

With experiments, the hypothesis was evaluated that certain information exchange among agents can improve the performance of the parallel search.

6.1 Further work

While this thesis has demonstrated the potential of cooperation of parallel searches for the GCP, many opportunities of other investigation directions remain. This section presents some of these directions.

Using different search strategies
In our algorithm, we use the k-GCP algorithm Tabucol as the subroutine. In further research, the agents can use a different search strategy. It is to investigate, whether our introduced

cooperation is only beneficial for our tabu local search or generally applicable.

Using different cooperation strategies
In this thesis, the cooperation is limited to information exchange. Some other directions like generic population-based metaheuristic are definitely worth further research.

Using different algorithms in agents
In our GCP-solver, all the agents run the same GCP algorithms. An interesting topic is the cooperation of agents, which run different GCP algorithms.

7 Bibliography

References

[1] M. Kubale, *Graph colorings*, vol. 352. American Mathematical Soc., 2004. (Page 1).

[2] M. R. Gary and D. S. Johnson, "Computers and intractability: A guide to the theory of np-completeness," 1979. (Pages 1, 2).

[3] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer languages*, vol. 6, no. 1, pp. 47–57, 1981. (Page 1).

[4] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 4, pp. 501–536, 1990. (Page 1).

[5] M. Garey, D. Johnson, and H. So, "An application of graph coloring to printed circuit testing," *IEEE Transactions on circuits and systems*, vol. 23, no. 10, pp. 591–599, 1976. (Page 1).

[6] F. T. Leighton, "A graph coloring algorithm for large scheduling problems," *Journal of research of the national bureau of standards*, vol. 84, no. 6, pp. 489–506, 1979. (Pages 1, 14).

[7] D. de Werra, "An introduction to timetabling," *European journal of operational research*, vol. 19, no. 2, pp. 151–162, 1985. (Page 1).

[8] A. Gamst, "Some lower bounds for a class of frequency assignment problems," *IEEE transactions on vehicular technology*, vol. 35, no. 1, pp. 8–14, 1986. (Page 1).

[9] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird, "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems," *Artificial Intelligence*, vol. 58, no. 1-3, pp. 161–205, 1992. (Page 2).

[10] 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*, vol. 39, no. 3, pp. 378–406, 1991. (Pages 2, 14).

[11] A. Hertz and D. de Werra, "Using tabu search techniques for graph coloring," *Computing*, vol. 39, no. 4, pp. 345–351, 1987. (Pages 2, 6).

[12] R. J. Trudeau, "Introduction to graph theory (corrected, enlarged republication. ed.)," 1993. (Page 2).

[13] T. Stutzle and H. Hoos, "Max-min ant system and local search for the traveling salesman problem," in *Evolutionary Computation, 1997., IEEE International Conference on*, pp. 309–314, IEEE, 1997. (Page 4).

[14] W. Zhang, "Configuration landscape analysis and backbone guided local search.: Part i: Satisfiability and maximum satisfiability," *Artificial Intelligence*, vol. 158, no. 1, pp. 1–26, 2004. (Page 4).

[15] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers & operations research*, vol. 13, no. 5, pp. 533–549, 1986. (Page 4).

[16] F. Glover, "Tabu search—part i," *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989. (Page 4).

[17] F. Glover, "Tabu search—part ii," *ORSA Journal on computing*, vol. 2, no. 1, pp. 4–32, 1990. (Page 4).

[18] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009. (Page 4).

[19] D. Brélaz, "New methods to color the vertices of a graph," *Communications of the ACM*, vol. 22, no. 4, pp. 251–256, 1979. (Page 6).

[20] "Trick's graph coloring algorithm." `http://mat.gsia.cmu.edu/COLOR/color.html`. Accessed: 2017-03-8. (Pages 6, 32).

[21] P. San Segundo, "A new dsatur-based algorithm for exact vertex coloring," *Computers & Operations Research*, vol. 39, no. 7, pp. 1724–1733, 2012. (Page 6).

[22] P. Galinier and A. Hertz, "A survey of local search methods for graph coloring," *Computers & Operations Research*, vol. 33, no. 9, pp. 2547–2562, 2006. (Pages 6, 7, 9).

[23] P. Galinier and J.-K. Hao, "Hybrid evolutionary algorithms for graph coloring," *Journal of combinatorial optimization*, vol. 3, no. 4, pp. 379–397, 1999. (Page 7).

[24] "Ic2 - hardware and architecture." `https://wiki.scc.kit.edu/hpc/index.php/IC2_-_Hardware_and_Architecture#Architecture_of_IC2`. Accessed: 2017-02-09. (Page 14).

[25] D. S. Johnson and M. A. Trick, *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, vol. 26. American Mathematical Soc., 1996. (Page 14).

[26] "Clique and coloring problems graph format." `http://www.or.uni-bonn.de/lectures/ss12/praktikum/ccformat.pdfl`, 1993. (Page 14).

[27] "Graph coloring instances." `http://mat.gsia.cmu.edu/COLOR/instances.html`. Accessed: 2017-01-14. (Page 14).

[28] "Dimacs graphs: Benchmark instances and best upper bounds." `http://www.info.univ-angers.fr/pub/porumbel/graphs/`. Accessed: 2017-01-14. (Page 14).

[29] "Smac: Sequential model-based algorithm configuration." `http://www.cs.ubc.ca/labs/beta/Projects/SMAC/`. Accessed: 2017-03-10. (Page 16).

[30] "Source code of dsatur and pass." `http://www.lamsade.dauphine.fr/coloring/doku.php`. Accessed: 2017-03-8. (Page 32).